Performance Tuning with the "Restrict" Keyword

David H Bartley *Systems and Applications R&D Center*

ABSTRACT

This report explains the ***restrict*** keyword and how to use it effectively with TI's C/C++ compilers to improve an application's performance. Few texts and articles explain this important feature of the C language well because the ANSI/ISO standard's formal definition is murky and compilers vary widely in how they implement it. In fact, many compilers ignore the keyword entirely or in all but a few circumstances.

Here we'll take some of the mystery out of what ***restrict*** is and how the language designers intended it to be used. We'll also show how to use it in your code to get the best performance out of TI's compilers for a variety of real-life scenarios.

The target audience is intermediate to advanced application developers. Familiarity with the C language and some experience developing code with TI compilers is assumed.

Contents

1 Introduction

This report explains the *restrict* keyword in ANSI/ISO C[1] and how to use it effectively with TI's C and C++ compilers to improve an application's performance. It also considers alternatives to *restrict* for communicating advice to the compiler.

The remainder of this document is structured as follows:

- Section 2 provides an overview of the *aliasing problem*, the *restrict* keyword and the `--no_bad_aliases` command line option. It also discusses strategic approaches to performance tuning and why using *restrict* is not a barrier to code portability.
- Section 3 explains how the ANSI/ISO standard for C [1] specifies the *restrict* keyword and its meaning in a program.
- Section 4 shows how to apply *restrict* effectively to real-life scenarios abstracted from actual applications.
- Section 5 describes alternatives to using *restrict*—assertions, pragmas, and *const* among them.
- Section 6 summarizes key points.
- Section 7 poses and answers frequently asked questions.
- Section 8 lists documents referenced in the report.

The capabilities described here apply to all of TI's C and C++ compiler versions released beginning in 2009. Of course, the compilers for the various processors differ in their ability to exploit user advice to optimize code.

2 Overview

C and C++ are powerful yet dangerous programming languages. At the heart of much of that power (and danger) is the fact that these languages allow freer use of pointers than do many languages designed specifically for application development, such as Fortran and Java.

All compilers are mandated to translate your C/C++ code strictly according to the dictates of their respective language standards. Optimizing compilers try to find faster (or smaller) instruction sequences for C/C++ operations that give equivalent results within the letter and spirit of the standards documents. Unfortunately, a strictly conforming compiler sees possible hazards and ambiguity that may not be obvious to a programmer. Many, perhaps most, of these potential problems are illusory, yet they force the compiler to generate less efficient code in order to ensure correctness.

In particular, the compiler may not be able to tell when two accesses to memory might read or write the same location. This potential *aliasing problem* can interfere with attempts to find the best instruction sequence in the vicinity of the accesses.

C99's *restrict* keyword and the TI compilers' `--no_bad_aliases` command line option[1] are powerful tools that allow you to tell the compiler when memory accesses through pointers cannot, or will not, address the same objects in memory. Learning to use these tools correctly and effectively is probably easier than you think, and one of the simplest ways to speed up your programs.

2.1 The aliasing problem

Consider function `f1()`:

```
int f1(short *p, short q[10], int *r)
{
  int i, acc=0;

  for (i=0; i<10; ++i){
    p[i] = q[i]+1;
    acc += r[i];
  }
  return acc;
}
```

Obviously `q[i]` must be fetched before `p[i]` may be stored. But may the compiler arrange to fetch `r[i]` before storing to `p[i]` or before fetching `q[i]`? If the compiler wanted to reorder instructions across loop iterations, could it fetch both `q[i]` and `q[i+1]` before storing to `p[i]`?

These questions are instances of the aliasing problem and are answered, as best they can be, by the compiler's *alias analysis*: Taken two at a time, might distinct scalar, array, structure or C++ class references access the same location in memory? More generally—especially for arrays, structures, unions, and classes—might they access the same *objects* in memory?

Aliases are distinct expressions that denote the same memory location. The term arises from thinking of them as different names for the same object. In the following code fragment,

```
char X[10];
char *p1 = X;
char *p2 = malloc(10*sizeof(char));
int Y;
int &Z = Y;  // C++: Z is a reference to Y
```

the object `X` may be accessed by its own name or as `*p1`. `Y` and `Z` are two C++ names for the same object.[2] And the unnamed object pointed to by `p2` can be accessed only via `p2` or copies of `p2`. Similarly, `X` may also be accessed via copies of `p1`. Alias analysis becomes complicated when copies are made, such as when passing a pointer on a call and when a pointer variable points to several objects over time through reassignment.

---

[1] The full discussion of `-no_bad_aliases` in Section 5.1 also describes other assertions it makes to the compiler about the program.

[2] See Section 4.11 for a discussion of reference variables in C++.

The compiler's alias analysis cannot remove or ignore true aliases—sometimes they're inherent in an algorithm or their presence must be tolerated as inherent to an interface. What alias analysis can do, however, is try to weed out false apparent aliases by analyzing the source code in light of hints and assertions supplied by the programmer.

A compiler has three general means of analyzing programs for aliasing problems. First, as we'll see in Section 3.1, the C and C++ standards [1,2] specify that a memory reference of a certain datatype may be aliased by other references only if they have the same or certain related types. Second, TI's compilers perform *points-to* analysis to track pointer values as they are copied and passed on calls. Third, C provides the *restrict* keyword and both C and C++ allow assertions and vendor-specific pragmas, which may provide user advice about aliasing. (TI's compilers also provide *restrict* for C++.)

We'll discuss what the C standard has to say about aliasing in Section 3. Then, in Section 4, we'll show how to use *restrict* effectively in scenarios typical of actual applications.

In the meantime, though, we might ask what effect adding `restrict` to function f1() would have:

```
int f1(short *restrict p, short q[10], int *r)
{
  int i, acc=0;

  for (i=0; i<10; ++i){
    p[i] = q[i]+1;
    acc += r[i];
  }
  return acc;
}
```

Results will generally vary from one target to another and across compiler versions. As I write, though, for the c64+ processor compiling at `-o2`, the loop speeds up from 6 cycles per iteration to two.

Interestingly, since the loop is so short, the compiler generates two loops for this test case when *restrict* is not used, one optimized as though *restrict* had been specified, the other optimized without the benefit of *restrict*. A runtime test of the pointer's values determines on each call to the function which loop is to be executed. The result? The optimized loop runs in two cycles, the unoptimized one in six.

2.2 Limits to compile-time analysis

No experienced programmer expects an "optimizing compiler" to produce truly optimal object code. The term dates to an era in the early history[3] of computer science when enthusiasm was high and languages and computers were primitive. Few could anticipate the plethora of programming languages and the diversity of programmable processors we have today.

[3] The term probably derives from the field of operations research, which was in its golden age when the first optimizing compilers were developed. "Optimization" in operations research signifies either "maximization" or "minimization" of an objective function. The O.R. community rightly qualifies the term as relative to a model of a problem and recognizes that models are often imperfect.

History also helps us appreciate the evolution of programming languages like C and its offspring C++. C's deficiencies as an application language are well known and are attested to by the widespread adoption of a multitude of other languages, many specialized to certain tasks or incorporating more modern views of program development.

C's relatively unfettered pointers reflect its origins as an alternative to assembly language. As with other features of its design, the idea was to give an expert programmer fine-level control of the processor: if the hardware had address registers and auto-incrementing addressing modes, then so would the language. With the machine's architecture modeled so closely by the language, the programmer was the optimizer and the compiler merely did exactly what it was told.

Today the relationship between programmer and compiler has reversed. Programmers want processor independence and smart compilers that embody detailed knowledge of how best to map source code to its target processor. High-performance DSPs in particular do not map well to C's model. To be effective, DSP compilers must raise the source code to a higher level of abstraction, losing insignificant details like the difference[4] between `x[i]` and `*px++`, in order to make analysis manageable and transformation to the target instruction set feasible.

With respect to aliasing, the compiler's analysis focuses on two tasks: using data types and local cues to determine whether two memory accesses may lawfully alias each other, and propagating points-to information across statements and functions.

Practical concerns limit points-to analysis in two ways. First, a pointer variable P may be assigned more than once and so point to several objects over its lifetime.[5] Second, function calls often cross source file boundaries. Tracking pointer parameter points-to sets generally requires at least interprocedural analysis across an entire file if not access to the entire program.

TI's compilers provide several levels of optimization characterized by the extent of the program over which they apply analyses and transformations. Level –o0, -o1, –o2, and –o3 optimizations examine individual statements, blocks of statements, loops and function bodies, and entire source files, respectively. Other command line options provide for compilation of many files as if they were one and even for whole program optimization.

2.3 Advising the compiler

Alias analysis is trivial for named objects (like `x[i]`) because they have fixed locations. It's much more difficult when object references are based on pointers (like `*px++`). If the compiler cannot determine the set of objects a pointer may address, its analysis may fail to prove that aliasing cannot occur.

By using *restrict*, you assert facts about your program that the compiler may not be able to prove on its own and therefore cannot always verify. It's your responsibility to guarantee that these assertions are correct.

[4] The second form uses pointer variable `px` to address each of the elements of array `x`. In the first form, `i` is incremented on each loop iteration; in the second, `px` is incremented. The array notation is generally easier for an optimizing compiler to analyze. This is especially true for multidimensional arrays.

[5] The "flip-flop" loop in Section 4.9 is a good example: two pointers to two objects in memory swap their values on each iteration. On any given iteration, their accesses do not alias, yet pairs of accesses spanning adjacent iterations do.

A related feature particular to TI's compilers is the `-no_bad_aliases` command line option, which constrains pointer arguments of functions to not point to overlapping areas of memory.

Section 5 discusses other ways you can advise the compiler, including `-no_bad_aliases`, assertions, pragmas, and the `const` keyword.

Some definitions

In what follows we'll refer frequently to objects and variables. These terms are inter-related.

An *object* is a region of data storage, the contents of which are values of a specified type. At execution time, it may be a register, a location on the stack, at a fixed address in RAM or ROM, or an unnamed block of dynamically allocated storage.

An object may have *subobjects*. Typically these objects are arrays, classes, structures, and unions containing other arrays, classes, structures, unions and scalars. These are aggregates that are formed as compositions of types. Informally, we also speak of a conceptual subregion of an object as a subobject. Subarrays comprising non-overlapping subregions of an array are popular, such as interleaved subarrays of even- and odd-numbered rows or of disjoint buffers carved out of an array and treated logically as distinct objects.

Non-aggregate storage for the simple built-in types—integers, pointers, floats—are also objects. We'll take care to distinguish an `int` element (subobject) of an array, for example, from the array object.

A *variable* is a typed data object that has a name and an associated value. For our purposes, a variable will be assumed to be assigned a location in memory (or in the register file) at which its value is stored. Furthermore, we will use the term *variable* for all such syntactic names for memory accesses, such as `A.B` and `p->m`, as well as `X`.[6]

We'll use *variable* when talking about the object's syntactic name in the source code and *object* when talking about the variable's assigned location in a register or memory at execution time.

A *pointer* is an expression having a pointer type. We'll refer to variables having pointer type as *pointer variables* unless the context permits the shorter term *pointer* without confusion.

The term *memory access* refers to any use or modification of a value in an object. A memory access using a pointer variable in its address is said to be *based on* the pointer. When based on a pointer, *memory access* always refers to the object pointed to, not to the pointer object itself. That is, `*p` denotes two memory accesses, the first to the pointer object named `p`, the second to the object `p` points to.

Note that when aliasing occurs on pointer-based accesses, we say that the *accesses* are aliased. Don't be misled by imprecise statements elsewhere about "aliased pointers" and the like. The pointers aren't aliased, the objects they point to are.

[6] Programming language aficionados may prefer the arcane terms *lvalues* and *rvalues* for what we are calling variables.

The restrict *keyword*

ANSI/ISO C provides the *restrict* keyword as a qualifier of a pointer variable's type.[7] Like the *register* keyword, *restrict* is intended solely as a means of communicating facts about the program to the compiler as an aid to optimization. When used correctly, it does not affect the meaning (*i.e.*, observable behavior) of a program. For this reason, a compiler is free to ignore it.

Applying *restrict* to the type declaration of pointer **P** can enable better code to be generated. But in doing so you make this promise to the compiler:

> *Within the scope of the declaration of **P**, only **P** or expressions based on **P** will be used to access the object or subobject pointed to by **P**.*

This promise forms a contract between you and your compiler. Understanding this contract is your key to safe and effective use of *restrict* in your application. The remainder of this whitepaper serves only to elaborate on and explain this.

Let's break this promise down to its key elements:

1. *Within the scope of the declaration of **P**…*

   P is a pointer variable. Examples are `p1`, `s.p2`, `p3[i]`,and both `p4` and `p5` in `p4->p5[]`.[8] Unlike other approaches to aliasing advice, such as some pragmas, the *restrict* keyword appears at P's declaration, not at points in the program where memory accesses based on P occur. As we'll elaborate in Section 3.4, the program region over which the restriction applies is precisely the scope of P's declaration.

2. *only **P** or expressions based on **P**…*

   This refers to the pointer in such accesses as `*`P, P`[i]`, `*(`P`+10*i+j+3)`, and P`[i][j+3]`. It specifically does not include the access `*`Q, even when pointer Q is a copy of P. (But see Section 3.5.)

3. *will be used to access…*

   Only actual fetches and stores are accesses. P`[i]` is an access, but `&`P`[i]` and P`+i` are not (see Section 3.6).

4. *the object or subobject pointed to by **P***.

Since a memory access based on a restricted pointer may alias only other accesses based on the same pointer, the compiler's efforts to find effective instruction sequences are less constrained. We will explain further as we proceed.

Example:

[7] C99 [1], section 6.7.3. Previous versions of the standard did not provide *restrict*.
[8] This syntax is the same whether `p5` is a pointer to an array or the array itself. This text assumes the former case.

```
void move(char *restrict a, char *b)
{
  int i;
  for (i=0; i<100; i+=2)  // manually unrolled loop
  {
    a[i] = b[i];
    a[i+1] = b[i+1];
  }
}
```

The scope of pointers `a` and `b` is the entire body of function `move()`. Within that scope the accesses based on `a` are the assignments to `a[i]` and `a[i+1]`. By restricting `a`, you promise that the references `b[i]` and `b[i+1]` will not access the same object (or subobject) as either access based on `a`.[9] Given this promise, the compiler may, for example, choose to perform both accesses to `b` before either access to `a`—by a double width fetch, for instance. Without *restrict*, the compiler would be forced to assume that the value of `b[i+1]` might be modified by the assignment to `a[i]` and so would be unable to safely rearrange the accesses.

(If it surprises you that both `a` and `b` might access the same memory locations, you're probably more familiar with languages for which this cannot happen. Fortran, for example, can often produce higher performance than C/C++ simply because such aliasing is not permitted.)

We'll explain in Section 3 how the ANSI/ISO standard for C [1] specifies the *restrict* keyword and its meaning in a program. Section 4 will show how to use *restrict* effectively with TI's compilers.

The current C++ language standard [2] does not provide the *restrict* type qualifier. However, TI's C++ compilers fully support this C99 functionality for C++ programs.

The `--no_bad_aliases` *compiler option*

As we just saw, a function that takes pointer-valued parameters poses the possibility that accesses via one parameter might alias accesses via another. In the previous example, suppose the restrict keyword were removed from parameter `a`'s type declaration. The compiler could not prove in that case that `*a` and `*b` are not aliased without examining all of the function's callers and the ways in which the pointer values passed in for `a` and `b` were calculated.

The compiler's *`--no_bad_aliases`* (or *`-mt`*) option specifies that the values passed in to the pointer-valued parameters of a function in the source file never address the same object. With *`--no_bad_aliases`*, `*a` and `*b` in the example are asserted not to alias each other.

Note that this is not the same as restrict-qualifying the pointer parameters. That would further assert that memory accesses via other pointers declared in the function (or with file scope) would not alias accesses through the pointer parameters.

We'll revisit *`--no_bad_aliases`* in Section 5.1, where we'll discuss further assertions it makes to the compiler about the program.

[9] Note that there's no need to restrict-qualify both `a` and `b`. Here, in fact, we could qualify either or both with the same result.

2.4 How aliasing can hinder compiled code performance

An optimizing compiler improves performance by exploiting the strengths of the target processor. Achieving faster object code may entail reordering memory fetches and stores or performing them in parallel. When at least one of a pair of memory accesses is a store, the compiler must be sure the accesses are not aliased.

TI's compilers perform several code-rearranging optimizations that might be hampered by aliasing. These include redundant fetch elimination, instruction reordering, loop unrolling, software pipelining, and SIMD transforms and vectorization.

Redundant fetches

These are instances of redundant common subexpressions—expressions that compute the same value as a similar or identical expression executed previously. The compiler will modify the first instance to assign its value to a compiler *temp* (temporary variable) such as `C$1`, `U$2`, or `K$3`, and then replace the second instance with a use of that temp. These temps can usually be assigned to registers.

Identical fetch expressions might not return the same value twice if an intervening store aliases them. For instance, the compiler cannot safely eliminate the apparent redundant fetch `*p1` in the following example unless `p1` and `p2` can be proven to point to different objects:

```
a = *p1;
*p2 = b;
c = *p1+5;  // not redundant if *p1 and *p2 are aliases
```

If analysis can prove that `*p1` and `*p2` cannot alias, the compiler may safely rewrite the example like so:

```
a = C$1 = *p1;
*p2 = b;
c = C$1+5;  // redundant fetch of *p1 eliminated
```

Instruction reordering

Reduced Instruction Set (RISC) and Very Long Instruction Word (VLIW) pipelined processors such as TI's TMS470 (ARM) family and high-performance DSPs achieve their highest execution rates when consecutive instructions execute without intervening delays (NOPs). Unlike superscalar processors, these CPUs cannot issue instructions out of order. This leaves effective instruction scheduling up to the compiler.

Memory accesses, even to fast on-chip SRAM or a level 1 cache, have high latencies compared to register-register CPU operations. It's important to move fetches up in the instruction stream to allow other operations to execute in their latency shadows. False aliases with preceding stores can limit this movement.

Loop unrolling and software pipelining

The compiler will often unroll a small loop to permit instructions in a block of adjacent loop iterations to be scheduled together. This is especially important for high-performance DSPs that have relatively high fetch latencies. But this extra scheduling freedom can be over-constrained by false aliases between stores in one iteration and fetches or stores in another.

```
for (i = 0;  i < 50 ; ++i)
  x[i] = y[i];    // assume that x and y are pointers to arrays
```

Unrolling this loop produces half as many iterations with two stores and two fetches per iteration:

```
for (i = 0;  i < 50 ; i += 2)
{
  x[i] = y[i];
  x[i+1] = y[i+1];
}
```

Now the compiler is free to fetch both elements of `y` before storing to either element of `x`—that is, if it knows the store to `x[i]` does not alias the fetch of `y[i+1]`. Restricting array pointer `x` (or `y`) can make a difference here. Because fetches can take several cycles to complete the compiler tries to migrate fetches towards the top of a loop iteration and stores towards the bottom. Larger loop bodies allow larger separation.

Unrolling can enable better processor utilization and thus faster loop execution, but it poses a time/space tradeoff. Since only the operations of a single iteration of an unrolled block may be scheduled together, more parallelism requires larger unrolling factors which result in larger loop bodies.

Software pipelining is a scheduling technique that restructures loops so that operations from adjacent iterations are interleaved in time. Unlike unrolling, software pipelining produces a sliding rather than blocked instruction window for scheduling. TI's C6x family compilers employ a pipelining technique called modulo scheduling, for which each operation in the original loop body appears exactly once in the restructured loop, yet may execute for a different original iteration than its neighbors. Although a software-pipelined loop usually requires additional "ramp up" and "ramp down" code for the first and last few iterations, it is usually more compact than one that is unrolled more than once or twice.

A simplified software pipelined schedule for our original example might look like this if reverted back to C code:

```
t1 = y[0];
for (i = 0;  i < 50 ; ++i) {
  t2 = y[i+1];
  x[i] = t1;
  t1 = t2;
}
```

A simplified software pipelined schedule for the unrolled example might look like this if reverted back to C code:

```
t1 = y[0];
t3 = y[i];
for (i = 0;  i < 50 ; ++i) {
  t2 = y[i+1];
  t4 = y[i+2]
  x[i] = t1;
  x[i+1] = t3;
  t1 = t2;
  t3 = t4;
}
```

Section 4 of *Hand-Tuning Loops and Control Code on the TMS320C6000* [4] presents useful tips for improving software-pipelinable loops.

SIMD transforms and loop vectorization

Single Instruction, Multiple Data (SIMD) is a term for a form of instruction-level data parallelism. ADD2 is an example of a C6x SIMD instruction. ADD2 separately and simultaneously adds the top 16 bits of its source registers to form the top 16 bits of its destination register, while doing the same for the three lower register halves.

```
dst.lo = src1.lo + src2.lo
dst.hi = src1.hi + src2.hi
```

Any processor supporting memory loads and stores of different widths allows SIMD loads and stores for smaller datatypes. For instance, a 32-bit load can fetch two 16-bit or four 8-bit values in parallel.

Loop vectorization is an optimization that unrolls a loop, usually by a factor that is a small power of two, to allow scalar operations in adjacent loop iterations to be performed in parallel.

Example:

```
short a[20];  // 16-bit elements
short b[20];
for ( i = 0 ; i < 20 ; ++i )
  a[i] += b[i];
```

After vectorization:

```
for ( i = 0 ; i < 20 ; i+=2 )
  STW(&a[i], ADD2(LDW(&a[i]), LDW(&b[i])));
```

Here `STW(&a[i],...)` means store a 32-bit register's contents to consecutive locations `&a[i]` and `&a[i+1]`. LDW is similar.

Summary

Rejecting apparent but false aliases can make a great difference to a program's performance by permitting important optimizations.

Yet, as with the medical profession, the first duty of a compiler is to do no harm. With respect to optimization, this obliges the compiler to assume an apparent problem that could invalidate a code optimization is real unless it can prove otherwise. If it's overly optimistic it could introduce an error; yet if it's too pessimistic, it could miss an opportunity to speed up the program..

But remember: the compiler cannot detect a misuse of restrict unless it's redundant. Getting it right is up to you.

2.5 Strategic performance tuning

A strategic approach to tuning an application's performance can greatly lessen the effort required to achieve the desired results.

As we will see, restrict-qualification can be especially effective for function parameters and variables with file-level scope. But *restrict* should be applied to a variable's type declaration only when it is safe to do so. Ideally, then, determining which parameters and global variables may and should be restrict-qualified would be part and parcel of designing the application's interfaces.

It can be useful as well, though, to apply these techniques to existing application codes, but preferably only after the application has been completely debugged.

Section 3 of *Hand-Tuning Loops and Control Code on the TMS320C6000* [4] is a valuable aid in devising a general strategy for tuning an application's performance.

2.6 Compatibility with other compilers

Fully compliant C99 compilers support `restrict`, as do many others. Some GNU compilers expect `__restrict__` instead, while some older Microsoft compilers expect `__restrict`.

Although the current C++ standard does not include it, commonly available implementations often do.

A useful compatibility trick is to use `restrict` but redefine it as needed by various compilers.

For example, if a compiler expects `__restrict__`, add this macro definition:

```
#define restrict __restrict__
```

If the compiler doesn't support restrict at all, add this:

```
#define restrict
```

3 Language Issues

In this section we'll discuss the implications of the ANSI/ISO C standard without regard to aspects of a compiler's implementation. While this presentation may appear legalistic or pedantic, it lays out the precise rules by which TI's compilers may operate. Knowing these rules may help you interpret the compiler's sometimes baffling behavior.

You may choose to scan most of this section lightly on first reading to get its gist, then move on to Section 4 to see *restrict* in action. If you do, you'll want to return to fortify your intuition with the "legal" details. Remember: you're using *restrict* to tell the compiler things it might not be able to determine on its own, so it usually can't verify that information and warn if you misuse it.

3.1 Aliasing in C

The C standard does not specify directly when aliasing is and is not possible. However, it states that an object of one type shall have its stored value accessed only by a reference that has one of a handful of related types.[10] Implicitly, then, pairs of accesses with types inconsistent with that list cannot alias. Restating these rules in terms of aliasing, only the following type differences allow aliasing:

- *Compatible* types. Types are trivially compatible when they are the same. C99 [1] also defines type specifiers (6.7.2), type qualifiers (6.7.3), and declarators (6.7.5) to be compatible in some circumstances.
- Types that are compatible when qualifiers (`const`, `volatile`, and `restrict`) are ignored.
- Types that differ only in their signedness, such as `short` and `unsigned short`.
- Types that differ in both their qualifiers and signedness.
- Array and layout-compatible structure and union types that include any of the type differences in this list among their members.
- A character type.

These rules generally match our intuition that signed and unsigned variants of a type may meaningfully refer to the same value but that quantities of different widths or with entirely different representations (such as integers and floats) cannot.

The exception for character types reflects C's heritage as a low-level systems language: it permits the contents of objects of arbitrary type to be examined a byte at a time via a pointer to a character type.[11]

[10] C99 [1], section 6.5, paragraph 7.
[11] This exception can be waived by a command line option. See Section 5.1.

Since members of a union share a common memory location, many programmers assume their compiler will treat them as explicitly aliased. In fact, the standard states to the contrary that "[t]he value of at most one of the members can be stored in a union object at any time."[12] (An exception is when a union contains structures that share a common initial sequence.[13]) However, TI's compilers take a conservative interpretation consistent with common practice and so consider members of the same union to be aliased.

The GNU Compiler Collection (GCC) compilers further constrain the set of accesses that may be aliased by adopting an optional "strict aliasing rule" via command line options [5]. These assert for example that pointers to array, structure, and union types with differing tags, and pointers declared with typedefs which differ only in name, do not permit aliasing. TI's compilers do not support these nonstandard interpretations.

3.2 The formal definition of *restrict*

The C standard's formal definition[14] of *restrict* is a model of succinctness and precision, if not clarity. Confusion often begins with the first statement: "Let **D** be a declaration of an ordinary identifier that provides a means of designating an object **P** as a restrict-qualified pointer to type **T**." Elsewhere[15] the standard defines *ordinary identifiers* as a category that excludes label names, the members of structures and unions, and tags of structures, unions, and enumerations. At first glance this appears to rule out restrict-qualifying members of structures and unions (and, by extension, C++ classes). But such is not the case, as we will see.

We'll resist the urge to dissect the standard's careful phrasing in any detail. But a few words here might help the curious reader demystify the formal definition. Others may choose to skip ahead.

In C99 [1], section 6.7.3.1, **D** is generally a declaration of a simple pointer variable or function parameter, or a *struct* or *union*. The "ordinary identifier" is the name of the variable, parameter, *struct*, or *union*. If we have a pointer-valued variable or parameter, **P** is that variable or parameter. If we have a *struct* or *union*, **P** may be any pointer-valued member thereof. In any event, **T** is the type of the object pointed to by **P**.

The standard goes on to define a block **B** in which a declaration **D** might occur. These are the scoping blocks we'll discuss below. It further defines an lvalue[16] **L** that has its address based on **P**. These **L**'s are the memory accesses that will be said to be unaliased with other lvalues that are not based on **P**. **X** gives a name to the object **P** points to; saying that memory accesses not based on **P** can't modify **X** allows other pointers to **X** in the scope of **P** but requires that any such pointers not be used in that scope to access **X**. It also says that **X** cannot be accessed directly by name.

[12] C99 [1], section 6.7.2.1, paragraph 14.
[13] C99 [1], section 6.5.2.3, paragraph 5.
[14] C99 [1], section 6.7.3.1.
[15] C99 [1], section 6.2.3.
[16] An *lvalue* is an expression that, when evaluated, designates an object. This conforms closely to our definition of a *variable* as a generalized syntactic name for an object (see Section 2.3). I use *lvalue* here because that is the term used in the standard's formal definition of *restrict*.

3.3 Type qualifiers and pointers

As mentioned in the Overview, we apply *restrict* to a pointer to advise the compiler about accesses using that pointer: pairs of memory references in the pointer's scope are not aliased when one is based on the restricted pointer unless both are.

Restrict is a type qualifier, like *const* and *volatile*, and often appears in typedefs and class, struct, and union type definitions as well as explicitly as part of a variable declaration. However, *restrict* qualifies the pointer's type, not the object's.

The `const` and `volatile` keywords qualify the type of an object:

```
short const *p1; // p1 points to an immutable object
short *const p2; // p2 is an immutable pointer
short const a[2] = {18, 14}; // a is initialized but immutable
short volatile b;            // Named object b is volatile
```

The `restrict` keyword qualifies the type of the pointer, not of the object pointed to.[17] It constrains *access* to the object, not the object itself.

```
short x[restrict 100];   // invalid: this doesn't declare any
                         // pointer for restrict-qualification
short *restrict y[100];  // valid: array of restricted pointers
short *z[restrict 100];  // invalid: "restricted" array
```

The first and third declarations above are invalid—they attempt to restrict-qualify arrays `x` and `*z` of type `short`. The second is valid—it qualifies a pointer to array `y` of short integers.

Arrays are passed by reference in C, so the following function declaration implicitly defines a restrict-qualified (unnamed) pointer to `a`:

```
void fun(short a[restrict 100]);
```

Two more examples:

```
typedef int *restrict RINT;

struct person {
    struct person *restrict spouse;
    struct person *mother;
    };
```

The typedef is valid, but ill-advised, as the restrict-qualification is not visible at variable declarations typed `RINT`. This conceals an important aspect of a pointer variable's behavior from the context in which it is defined.

Restrict-qualifying `spouse` would appear to be both desirable and correct (in most cultures). Since a person may have siblings, it likely would be incorrect to restrict-qualify `mother`.

[17] A bit of trivia for history buffs: The ANSI C committee at one time considered a `noalias` keyword that qualified the object, not the pointer. This paralleled `const` more closely than `restrict` does but was found to be unworkable.

3.4 Scopes and lifetimes

C is a lexically scoped language. *Scope* refers to the region of program text within which a variable's declaration is visible and the variable may be used. Scoping in C is said to be *lexical* because that region is lexically (or textually) contained within an area associated with the variable's declaration.[18]

A variable's scope is determined by the placement of its declaration. C specifies four kinds of scopes: function, file, block, and function prototype.[19]

Function scope applies solely to labels associated with the goto statements of a function definition; they can have no variable declarations and so are not relevant to our topic. However, we sometimes say that variable declarations having the scope of the outermost block of a function have function scope.

When declared outside all function definitions and prototypes, a variable has *file scope*. This scope terminates at the end of the translation unit (a single source file along with any files it may `#include`). Consequently each file has its own file scope and restricting a file-level pointer variable says nothing about aliasing between accesses based on that pointer and accesses in functions defined in other files.[20]

Parameter variables for function declarations (prototypes) have *function prototype* scope, which terminates at the end of the prototype.

In all other cases, a declaration has *block scope*. Blocks are sequences of statements enclosed in braces and containing one or more declarations. Parameter variables for function definitions have the scope of the outer block of the associated function body.

When the same identifier is used in two declarations, their scopes may overlap. If so, one scope (the outer scope) will be a strict superset of the other (the inner scope). The outer scope's declaration is overridden within the inner scope in this case and thus is not "visible" there.

A declaration with a storage class specifier `extern`, `static`, `auto`, or `register` within a block has the block's scope.

The scope of a member of a structure or union is that of the structure or union.[21] As we saw in Section 3.2, it is the declaration for the structure or union variable that "provides a means of designating" a pointer member within it.

Typedefs and type declarations allow the `restrict` keyword to appear somewhere other than immediately in the variable declaration. Regardless, it is the scope of the variable declaration, not that of the typedef or type declaration, over which the restrict-qualification holds.

[18] Textually-defined areas are regions of source code after preprocessing. The body of a macro invocation or an included file is textually contained within the area of its occurrence. The body of a called function, on the other hand, even when inlined, is not textually contained within the area of the call.
[19] C99 [1], section 6.2.1.
[20] This matters when a programmer merges files together. Doing so may invalidate a file-level *restrict* qualifier.
[21] C99 [1], section 6.2.1, paragraph 7. The same holds for C++ classes with TI's compilers.

For our purposes, the most interesting scopes are the file level (global pointers), function level (parameters and pointers visible across an entire function body), and block-level scopes associated with loops. Typically these pointers are simple variables or members of structures (or classes). Section 4 will elaborate on using *restrict* in various contexts and scopes to improve performance with TI's compilers.

In addition to the concept of the scope of a variable, C specifies its duration or lifetime. Scope refers to the textual region in which references may occur. A variable's *lifetime*[22] denotes the (continuous) interval of program execution time during which storage is guaranteed to be reserved for it. For example, storage for a variable will be retained across a call that leaves the variable's scope and across a nested block with an overriding declaration involving its identifier. On the other hand, the compiler is free to release an `auto` or `register` variable's storage early[23] if it can determine that it will not be accessed again before execution leaves its scope. For example, the compiler may assign the same register to two or more variables when their lifetimes do not overlap, even when their scopes do.

Each time a block scope is entered we get new instances of its `auto` and `register` variables. These new instances are not guaranteed to retain the contents or even the locations assigned in previous instantiations. Pairs of accesses in different scope instantiations are not constrained by restrict-qualification. See Section 4.6 for an example showing distinct scope instantiations for each iteration of a loop.

3.5 Accesses "based on" a pointer

A pointer expression is said to be *based on* a pointer **P** if modifying **P** would change the value of the expression. If **P** is restrict-qualified, memory accesses based on **P** are the sole means of (valid) access to the object or subobject **P** points to.

Examples:

- The pointer expression `&A[I]` is based on the address of `A[0]`. If `A` is a pointer, the expression `&A[I]` is equivalent to `A+i`.
- `p+3*i+5` is based on `p` and is the same as `&p[3*i+5]`.

A pointer expression based on a copy of **P** is not based on **P**, since the copy is not affected by a change to **P**. Accesses based on copies of a restrict-qualified **P** will be considered—correctly or not—to be unaliased with accesses based on **P**.

Example:

```
void f35(short *restrict p1)
{
  short *p2 = p1;
  ... *p1 ... *p2 ...
}
```

Function `f35()` has undefined behavior—`*p1` and `*p2` clearly alias and yet the latter is not based on the restricted pointer `p1`.

[22] C99 [1], section 6.2.4.
[23] By "early," we mean before execution of the program has exited the variable's scope.

Things get murky if **P** itself is aliased. Suppose we set $\mathbf{P}_2$ to the address of **P**. Do $*\mathbf{P}$ and $**\mathbf{P}_2$ alias? The standard says no, but the reality may be that they do. It's best to avoid these situations regardless of whether *restrict* is involved.

3.6 Using *restrict* correctly

We're now ready to summarize the conclusion of the C standard's formal definition of *restrict*.[24]

The standard's precisely calculated phrasing specifies whether certain usages of *restrict* have behavior that is defined or undefined. Defined behavior is sometimes termed "valid." Undefined behavior corresponds to incorrect use of the language. For example, when a pointer is restrict-qualified but the program improperly accesses the pointer's object without basing the access on that pointer, we have undefined behavior. The compiler is not required to generate "correct" code in that event.

Given two memory accesses,

(1) If one is based on a restrict-qualified pointer **P** to an object **X** and the other is also based on a pointer variable, then (1a) either the pointer variables are the same, (1b) the second access appears outside the scope of the declaration of **P**, (1c) neither access modifies **X**, or (1d) the behavior is undefined.

(2) If one is based on a restrict-qualified pointer **P** to an object **X** and the other accesses **X** by name,

> (2a) If the type of **X** is *const*-qualified and either access modifies **X**, the behavior is undefined. That is, the type of **X** (and of $*\mathbf{P}$) should not be *const*-qualified in this case.
>
> Otherwise, either (2b) the direct access of **X** is outside the scope of the declaration of **P**, (2c) neither access modifies **X**, or (2d) the behavior is undefined.

A restrict-qualified pointer variable may point to one or many objects during its lifetime. The qualification holds at each point in the program at which an access based on the pointer occurs and applies to the object pointed to at that point. Although, say, `p1` and `p2` might both reference object **X**, they must not do so at the same time when accesses via `p1` or `p2` occur.[25]

Aliasing does no harm when the two memory accesses are both fetches since neither access modifies the objects addressed.[26] In the following function, it is valid for `b[i]` and `c[i]` to alias since neither is modified. However, `a[i]` and `c[i]` must not, because the store to the former modifies its object.

[24] C99 [1], section 6.7.3.1, paragraph 4.
[25] We assume at least one of `p1` and `p2` is restrict-qualified, the accesses are within the scope of both pointers and at least one of the accesses modifies **x**.
[26] Fetches from volatile objects might modify those objects. TI's compilers ignore any restrict-qualification for pointers to such objects.

```
void add_vectors(int a[10], int b[10], int c[restrict 10])
{
  int i;
  for (i = 0; i < n; ++i)
    a[i] = b[i] + c[i];
}
```

A quick note about subobjects: Aliasing is defined only for an object's elements or members that are actually accessed and not for other components of those objects. The standard allows two pointers into the same object, at least one of them restricted, to be dereferenced as long as the sets of locations accessed by the two are disjoint. For instance, example 2 of the standard's formal definition calls valid a case where the lower and upper halves of a 100 element array are passed as a pair of restrict-qualified parameters of a function:

```
void copy(int n, int *restrict p, int *restrict q)
{
  while (n-- > 0) *p++ = *q++;
}

void test(void)
{
  extern int d[100];
  copy(50, d+50, d);  // valid
  copy(50, d+1,  d);  // undefined behavior
}
```

4 Using *restrict* effectively with TI's compiler tools

The goal of this section is to show you how to apply restrict qualification effectively to real-life scenarios we've abstracted from actual applications. These case studies illustrate the breadth of TI's compiler support for the construct and suggest ways to improve performance for common data access patterns.

Look for these common themes:

1. Only pointers may be restricted. But function parameters that look like array declarations are actually pointers and so may be restrict-qualified.
2. Restricting pointers that appear at file level, in loop bodies, and as components of arrays, structures, and classes may also be worthwhile. If you copy one of the latter into a local pointer variable, you'll often be able to restrict-qualify that variable.
3. Don't restrict P's type unless you *know* that all references in the scope of P that access the object pointed to by P are based on P. Carefully document the interfaces to functions with restricted pointer parameters to help ensure they continue to be called correctly as your application evolves.
4. Pointers to character types may be used to access objects of any type, so restrict-qualifying them can be especially helpful to the compiler.
5. Use *restrict* sparingly. Generally it is sufficient to restrict-qualify only those pointers that are assigned through.
6. Use *restrict* explicitly in pointer variable declarations. Don't hide it in a macro or typedef. It should appear where its occurrence can be noted and its scope understood.

Of course, adding restrict to a declaration is no guarantee of higher performance. Other factors may inhibit optimization besides aliasing. But judicious use is probably the easiest way to improve your code once you've honed your algorithms.

4.1 Global pointer variables

A declaration that appears outside of any block or list of parameters has file scope. Taken together, these file scope declarations

```
int *restrict p1;
int *restrict p2;
extern int A[];
```

promise that if an object is accessed using any one of `p1`, `p2`, or `A[]` it will not be accessed using either of the others. Furthermore, since the file scope lexically encompasses all other scopes, no accesses through block scoped pointers can access the object pointed to by `p1` or `p2`:

```
int *p3;
void f(int *p4, int& a5) { int *p6, *restrict p7; … }
```

No accesses of `a5` or via pointers `p3`, `p4`, `p6`, or `p7` may alias accesses via either `p1` or `p2` within their respective scopes.

```
struct person {
    struct person *restrict spouse;
    struct person *mother;
    };

struct person *gracie;

void f41(struct person *george)
{
  ...
}
```

Here Gracie's spouse is unique for the entire extent of the program. George's spouse is unique only within `f41()`. Although the declaration with the restrict-qualification has file scope, the scope of each instance is defined by the appearance of a variable declared with that type. Gracie has file-level scope; George has block-level scope.

4.2 Function parameters

The parameters in a function declaration have function prototype scope, which terminates at the end of the declaration:

```
void vector_swap(short *restrict v1, short *v2, int n);
```

In this function's definition, the parameters have the same block scope as `i`:[27]

```
void vector_swap(short *restrict v1, short *v2, int n)
{
  int i;
  ...
}
```

Restricting `v1` asserts that no memory access in the body of `vector_swap()` that is not explicitly based on `v1` can alias a memory access in the body that is. The benefit of this assertion to the compiler is that the programmer has taken responsibility for assuring that *every* call to `vector_swap()` in the application passes the addresses of distinct vectors for `v1` and `v2`. This assurance may well lead to increased performance. The drawback, of course, is that this assurance becomes part of the interface to `vector_swap()` and can be enforced only by the programmer.

[27] C [1], section 6.9.1, paragraph 9.

You may have noticed that we've restrict-qualified only one of the pointer parameters in this example, despite the fact that vector swap is presumably a symmetric operation that fetches and stores through both pointers. We do so here to emphasize a point that is often misstated in the popular press and on the web: If *either* `p1` or `p2` is declared with restrict, `*p1` and `*p2` cannot alias. Some writers, perhaps reading too much into the handful of examples in the C standard,[28] imply or state outright that both must be restricted. (And some vendors' implementations may follow suit.)

A useful rule of thumb is to restrict just those pointers that are assigned through, since aliasing is only an issue when values are modified. We'll generally follow that practice in our examples.

A parameter is also restrict-qualified if its declared type is restricted:

```
typedef short *restrict S16RP;
void vector_swap(S16RP v1, S16RP v2, int n);
```

Though legal, this practice can make it harder to recognize that *restrict* is part of the function's interface.

Class, structure and union types may have restricted data members. These restrictions carry over to parameters of those types.

```
struct List { T *first;  struct List *restrict rest; }
```

This declaration states that, while there may be several pointers to a List, only the List itself can address its sublists. This restriction typically carries over into iterators over such a list:

```
void f42a(List *restrict pl)
{
  while (pl != 0)  // iterator pl is restricted
  {
    ...;  pl = pl->rest;
  }
}
```

If every object of type `T` is a member of at most one List, it would be appropriate to restrict-qualify member `first` as well:

```
struct List { T *restrict first;  struct List *restrict rest; }

void f42b(List *restrict pl)
{
  while (pl != 0)
  {
    T *restrict obj = pl->first;
    ...;  pl = pl->rest;
  }
}
```

[28] C [1], section 6.7.3.1.

If in addition every object of type `T` appears at most once in any List, it is better to declare the local copy `obj` outside the loop so it applies across iterations:[29]

```
struct List { T *restrict first;  struct List *restrict rest; }

void f42c(List *restrict pl)
{
  T *restrict obj;      // this restrict-qualification applies
  while (pl != 0)       // across all iterations of the loop
  {
    obj = pl->first;
    ...;  pl = pl->rest;
  }
}
```

4.3 Array parameters

A function parameter declared as an array is actually a pointer to the array. Section 6.7.5.3 of the C99 standard puts it this way:

> A declaration of a parameter as "array of *type*" shall be adjusted to "qualified pointer to *type*," where the type qualifiers, if any, are those specified between the `[` and `]` of the array type derivation.

Thus, to restrict-qualify an array parameter, the `restrict` keyword should appear as follows:

```
void f43a(short X[restrict]) { ... }
```

or

```
void f43b(int Y[restrict 10][10]) { ... }
```

We'll have more to say later about multidimensional arrays and arrays of pointers and structures.

4.4 Inlined functions

Inlining is a deceptively simple concept. Conceptually, a function call is inlined by substituting a copy of the called function for the call and adding a few assignments to bind the parameters to the arguments and return the result. The devil is in the details.

First, let's be clear on how inlining a function differs from expanding a macro invocation:

```
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
x = MAX(x, ++y);
```

[29] We'll show in Section 4.6 how loop-level scopes such as that in function `f42b()` can interfere with optimization across loop iterations.

This example is well-known, so most programmers know that the side effect `++y` presents a serious problem—the assignment expands into `(((x)>(++y))?(x):(++y))`, the caller's variable `y` is incremented twice, and the value assigned to `x` may be a singly or doubly incremented `y` or `x` itself. Had we defined `MAX` as a function, the compiler would have evaluated each of `x` and `++y` just once, assigned their values to local variables `a` and `b`, and arranged to return the greater of `a` and `b` as the result. Had we declared that function inline, we would expect the same to happen, albeit without the overhead of a call.

The driving principle is this: aside from considerations of code size and performance, a function call must behave exactly the same when it is inlined as when it is not.

Suppose we have this definition for a maximum value function:

```
inline int max(int a, int b) { return (a>b) ? a : b; }
```

Then the compiler might expand `x = max(x, ++y)` into something like this:

```
{                               // "outer scope"
  int compiler$temp1 = x;
  int compiler$temp2 = ++y;
  int compiler$temp3;
  {                             // "inner scope"
    int a = compiler$temp1;
    int b = compiler$temp2;
    compiler$temp3 = (a>b) ? a : b;
  }
  x = compiler$temp3;
}
```

Why does the compiler add two new scoping blocks?[30] Because it must carefully mix the scopes of the caller and the callee when setting up the parameters and returning the result.

The outer scope permits the introduction of local compiler temps with visibility to the caller's scopes where `x` and `y` are declared. We can't use the caller's scope because multiple calls to `max()` would otherwise put multiple declarations of those temps into the caller's scope.

The inner scope provides for the parameters and local variables of `max()` and allows local variables to override variables with the same names in the caller's scope. (Consider the chaos that would otherwise ensue if the parameter names were `y` and `x` instead of `a` and `b`!)

But even this model of inlining is a bit too simple. To see why, let's focus on pointer parameters and *restrict*.

[30] Actually it doesn't do so literally, because it doesn't inline the function textually as shown here. But it must maintain the scoping relationships quite carefully regardless of how it represents the program internally.

```
int g;  // file-level (global) variable

void callee(int *restrict p)
{
  short a, *b;
  ... g ...;
}

void caller(int a)
{
  char g;
  callee(&g);
  ... g ...;
}
```

A simple textual expansion like that in the previous example would produce this:

```
void caller(int a)
{
  char g;
  {                                   // "outer scope"
    int *compiler$temp1 = &g;
    {                                 // "inner scope"
      int *restrict p = compiler$temp1;
      short a, *b;
      ... g ...;       // this is not the caller's g
    }
  }
  ... g ...;           // but this is the caller's g
}
```

As expected, parameter variable `p` has become a block-scoped variable and kept its *restrict* qualifier. But now `g` seems to refer to the caller's character variable, not the `g` with file level scope! Fortunately the compiler retains the proper scoping for `g` and other references in inlined function bodies.

In sum, to achieve the requirement that there be no discernable difference between inlining and not inlining the call, the scoping blocks of the inlined function body must remain outside the scope of the call site. Even though they are now logically nested within that scope's operations, they are lexically disjoint.

4.5 Block-local pointer variables

Programmers often declare a block-local pointer as a convenient name with constrained scope for an object with wider scope. Common examples are to give simple names to complicated address expressions and to address expressions with side effects; to members of structs, classes, and arrays; and to pointer values returned from function calls.

Doing so also presents an opportunity to refine the nature of the accesses to be made via the pointer within the smaller scope by adding *const* or *restrict* qualification to the pointer's type:

```
if ( client[i].has_children() )
{
  short *restrict p1 = &clients[i].children[0];
  short *restrict p2 = malloc(100*sizeof(short));
  ...
}
```

Within the block comprising the *then* clause, only `p1` may access the elements of array `children[]` for each client. Applying *restrict* to `p2` reflects the fact that `malloc()` is guaranteed to return a unique pointer. (The same applies for operator `new` in C++.)

4.6 Character pointers

As was noted in Section 3.1, the C language generally allows two accesses to the same object only when their types are compatible. The principal exception is when one has character type. This is unfortunate since character types as employed in application programming rarely are used to examine bytes within objects of other types.

Restrict-qualifying pointers to characters can be especially helpful to the compiler's alias analysis.

An alternative is to assert for an entire compilation that character types never alias other types. This is discussed in Section 5.1.

4.7 Loop body scopes

Each time a scope is entered we get new instances of its variables. These new instances are not guaranteed to retain the contents or even the locations assigned in previous instantiations. Pairs of accesses in distinct scope instantiations are not constrained by restrict-qualification.

In particular, a declaration in a loop body's scope applies to just a single iteration of the loop.

Example:

```
for (i=0; i<N; ++i)
{
  int *restrict p1 = mother[i];
  int *restrict p2 = uncle[i];
  ... *p1 ... *p2 ...
}
```

Within any single iteration of this loop, accesses via `p1` and `p2` cannot alias each other. However, `*p1` in one iteration may well alias `*p2` in another iteration.[31] This fact can constrain the compiler's ability to unroll or software pipeline the loop in order to schedule operations across iteration boundaries.

`const` qualifiers in loop bodies have similar behavior:

[31] C99 [1], section 6.8.5, paragraph 5: "An iteration statement is a block whose scope is a strict subset of the scope of its enclosing block. The loop body is also a block whose scope is a strict subset of the scope of the iteration statement." Similarly, for C++ [2], section 6.5(2): "The substatement in an *iteration-statement* implicitly defines a local scope (3.3) which is entered and exited each time through the loop."

```
for (i=0; i<N; ++i)
{
  const x = i; …
}
```

Each iteration of this loop has an immutable instance of `x`, yet each instance has a different value from the others.

In the first example above, since we're modeling relationships in which no mother can be some person's uncle, we can avoid constraining the restricted scope to a single iteration by moving the declarations of `p1` and `p2` out of the loop:

```
int *restrict p1, *restrict p2;

for (i=0; i<N; ++i)
{
  p1 = mother[i];
  p2 = uncle[i];
  ... *p1 ... *p2 ...
}
```

Of course, we must make sure the restrictions remain valid in the larger outer scope.

C++ allows declarations in the control structure of a `for` loop. These have the same scope as the `for` itself; that is, of the block containing the `for`. In the next example, the scope of `p` extends to the end of the loop:[32]

```
// p's scope begins here
for (char *restrict p = list ; p != 0 ; p = p->rest)
{
  p->first = ...;
} // p's scope ends here
```

If we ignore the complexities of `break` and `continue` handling, we can represent the scopes for this loop by the two blocks like this:

```
{                           // New scope for the 'for'
  char *restrict p = list;
  while (p != 0)
  {
    p->first = ...;
  }
}
```

Moving `p`'s declaration to the `for`'s control structure clarifies that the scope of `p` is the entire set of iterations of the loop.

[32] C++ [2], section 6.5.3.

4.8 Disjoint subobjects and subarrays

Aliasing does not occur when memory references access distinct areas of an object. Here are some examples.

In `f48a()`, although it makes no sense in human terms, `*father` and `*son` could be the same `person` object because the `spouse` and `mother` components of a `person` object do not overlap.

```
void f48a(struct person *restrict father, struct person *son)
{
  father->spouse = son->mother;
}
```

Similarly, `a` and `b` in `f48b()` may properly be the same array object because subscripts `3` and `5` refer to distinct array elements.

```
void f48b(int a[restrict 10], int b[10])
{
  a[3] = b[5]+1;
}
```

More generally, arrays may have subarrays that are interleaved or non-overlapping:

```
void f48c(int *restrict w, int *x, int n,
          int *restrict y, int *z, int m)
{
  int i, j;

  for (i = 0; i < n; ++i)     // valid:
    w[2*i] += x[2*i+1];       // interleaved subarrays

  for (j = m; j < 2*m; ++j)   // valid:
    y[j] += z[j-m];           // non-overlapping subarrays
}
```

4.9 Flip-flop pointers and FIFOs

Programmers sometimes use a group of pointers to access subobjects of an object. These may be restrict-qualified even when the mapping of a set of distinct pointer values to pointer variables is permuted.

The simplest instance of this has two pointers, called “flip-flop” variables, that periodically swap their values. Here we have an example of swapped in/out buffers using a shared array:

```
void flipper(float buffers[2*BUFSIZ])
{
  float *flip=&buffers[0], *flop=&buffers[BUFSIZ];

  while (something)
  {
    int i;
    float *temp;
    float *restrict iPtr = flip;
    float *restrict oPtr = flop;

    for (i=0; i< BUFSIZ; ++i) oPtr[i] = process(iPtr[i]);

    temp = flip;
    flip = flop;
    flop = temp;
  }
}
```

Restricting `oPtr` or `iPtr` is valid here because all the accesses based on `oPtr` and `iPtr` occur while `oPtr` and `iPtr` point to distinct subarrays of `buffers`. It would not be correct, however, to declare them outside the `while` loop, since they do point to the same subarray on adjacent iterations of that loop. And although I declared both `restrict` for aesthetic reasons, restricting only one is sufficient.

A first-in, first-out buffer (a FIFO or queue) appears to have similar behavior:

```
int Q[100];
int *restrict headQ = &Q[0];  // incorrect use of restrict!
int *restrict tailQ = &Q[0];  // incorrect use of restrict!
   ...
// enqueue(Q,x):
*tailQ++ = x;
if (tailQ == 100) tailQ = 0;
   ...
// y=dequeue(Q):
y = *headQ++;
if (headQ == 100) headQ = 0;
```

Restricting `tailQ` or `headQ` seems at first to be valid because they have the property that they point to the same location if and only if the queue is empty. But that property can be broken by the compiler if it thinks it can reorder the fetches and stores. Adding restrict-qualification is incorrect here!

4.10 Multidimensional arrays and arrays of pointers

C uses the same notation for array object and array pointer accesses and declarations. Although this can be convenient, it makes identifying whether and where *restrict* can be applied more complicated.

```
void f410a(int a[4])
{
  int b[5][6];
  int *c = (int*)malloc(20*sizeof(int));
  b[1][2] = a[3]+c[4];
}
```

The array accesses `a[3]` and `c[4]` are pointer-based, while `b` is a named object. Thus, we may consider restricting `a` and `c` but not `b`:

```
void f410a(int a[restrict 4])
{
  int b[5][6];
  int *restrict c = (int*)malloc(20*sizeof(int));
  b[1][2] = a[3]+c[4];
}
```

Similarly, an array access `x[i][j]` has two interpretations, depending on `x`'s declaration:

```
void f410b(float **x1,  // Vector of pointers to float vectors
           float x2[20][20]); // 2-D array of 400 floats
```

The first involves two pointers and requires two memory accesses for each array access—one to fetch the i^{th} vector of floats, the second to access the j^{th} element. The second makes just one pointer-based access by linearizing the subscripts. That is, `x2[i][j]` is equivalent to `*(x2+20*i+j)`.

If appropriate, we may restrict any or all of these three pointers:

```
void f410b(float *restrict *restrict x1,
           float x2[restrict 20][20]);
```

This first `restrict` qualifies fetches of pointer-valued elements of the pointer array `x1`. The second qualifies fetches based on pointer-valued elements to yield a floating point value. The third qualifies the pointer `x2`. Note that it is meaningless and illegal to declare `x2[restrict 20][restrict 20]` as there is only one pointer involved.

4.11 C++ extensions for *restrict*

We mentioned earlier that TI's C++ compilers support the full functionality of C's *restrict* keyword. Although the current C++ standard [2] omits mention of *restrict*, we are in agreement with the apparent industry consensus on how a future standard will deal with it. Most compiler vendors that support both C and C++ support *restrict* for both languages. And many support proposed C++ extensions that are likely to be added to the next standard.

TI's C++ compilers allow *restrict* wherever it is allowed in C. They also allow it as a type qualifier for nonstatic member functions (qualifying the `this` pointer) and for reference and pointer-to-member types, as described below and in an X3J16 standards working group proposal [6].

Note: As this is written, TI's C++ compilers accept these extended forms of restrict-qualification but do not fully exploit them for code optimization. Future compilers are expected to do so. Consult your compiler's documentation.

Restricting the `this` *pointer*

Nonstatic member functions have an implicit `this` parameter that points to the object on which the function was invoked. These functions may be declared `const`, `volatile` or both and the qualifier is applied to the `this` pointer.[33] For example, in

```
class C {
  int data;
  void mf() const;
};
```

the `this` pointer for `mf()` has the implicit declaration "`C const *this`". That is, the function cannot modify the object of type `C` pointed to by `this` and the class variable `data` in particular. The `volatile` qualifier is treated similarly.

By analogy, the *restrict* qualifier may also be applied to a nonstatic member function's `this` pointer using this syntax. Substituting `restrict` for `const` in the example,

```
class C {
    int data;
    void mf() restrict;
};
```

results in the implicit declaration "`C *restrict this`". As we saw in Section 3.3, unlike the case with `const`, the `restrict` keyword qualifies the pointer, not the object pointed to.

C++ operators may also have restrict-qualified `this` pointers. An assignment operator for class C might be defined as

```
C &C::operator=(C &rhs) restrict { . . .; return *this; }
```

Restricting reference variables

A *reference type*[34] in C++ is defined by adding an `&` after a type specifier:

```
T var;          // A variable (object) of type T
T &ref = var;   // Another name for var
```

Reference variables like `ref` are explicit aliases of the shared object they reference. Any operation using or assigning the value of `ref` has the effect of using or assigning the value of `var` and vice versa.

[33] C++ [2], sections 9.3.1 and 9.3.2.
[34] C++ [2], section 8.3.2.

Reference variables provide many of the capabilities of pointer variables but employ a different syntax. It is useful to think of a reference declaration for variable `X` as implicitly defining a hidden pointer variable p^X that always points to `X`. Likewise we might think of a use of `X` as an indirect reference based on its hidden pointer. Indeed, this is precisely how the compiler implements such variables.

Example:

```
short &Ai = x.y.z.array[i];
Ai = 3;
```

implicitly defines a compiler temp p^{Ai} and replaces occurrences of `Ai` with $*p^{Ai}$:

```
short *pAi = &x.y.z.array[i];
*pAi = 3;
```

The implicit pointer may be restrict-qualified by declaring the reference variable thus:

```
short &restrict Ai = x.y.z.array[i];
```

Perhaps the most common use of references is as function parameters, where the `&` annotation specifies that the corresponding argument on a call will be passed in by reference, not by value as is the default in C and C++.

Example:

```
void fill2(short *px, short &restrict y)
{
  px[0] = y;  // The stores do not
  px[1] = y;  // alias the fetches
}
```

Restricting pointer-to-member types

Both C and C++ support pointers to members of specific objects. In C++, the pointer-to-member type allows you to point to various members of a class, all of the same type, without specifying until later which object will have its member accessed. Think of it as designating which member will be accessed when the pointer-to-member is assigned but postponing identifying which object will have its member accessed.

A pointer-to-member is somewhat like an ordinary pointer in that it must be indirected to access its value. If `ptm` is a pointer to a member of some class and `X` is an instance of that class, that member of `X` is accessed as `X.*ptm`. Given a pointer `p` to an object of the class, `p->*ptm` accesses the member. It's perhaps useful to consider an analogy with array references involving variable and constant subscripts: `A[i]` is to `A[3]` as `X.*ptm` is to `X.member`.

The syntax to declare pointers-to-members is similar to that for ordinary pointers, but adds the class name followed by `::`.

C++ allows suitably typed pointers-to-members for both data and function members. In class `Z` below, `d1` and `d2` are two data members of the same type that we will access via a pointer-to-member `ptdm`. Likewise, `f1` and `f2` are two member functions of the same type that we will invoke with `ptfm`.

```
class Z {
      int d1, d2;
      char f1(short);
      char f2(short);
};

// Declare and initialize the pointer-to-member variables
int   Z::*ptdm         = &Z::d1;
char (Z::*ptfm)(short) = &Z::f1;

// Reassign the two variables
ptdm = &Z::d2;
ptfm = &Z::f2;

// Access data members via their pointer-to-member Z object;
object.*ptdm = 1492;
Z *pObject = &object;
pObject->*ptdm = 1066;

// Invoke the member functions via their pointer-to-member
char a = (object.*ptfm)(42);
char b = (pObject->*ptfm)(42);
```

A pointer-to-member may be const- or volatile-qualified in standard C++, but since these qualifiers apply to the member pointed to, the qualifiers must match those of the members to which it may point. In extended C++, a pointer-to-member may be restrict-qualified as well. This is a qualification of the pointer-to-member, not the member, so `restrict` does not appear in the types of the members themselves.

Adding *restrict* to a pointer-to-member uses the same syntax as for *const* and *volatile* and is similar to that for ordinary pointers:

```
// Restrict-qualify the pointer-to-member variables
int   Z::*restrict ptdm         = &Z::d1;
char (Z::*restrict ptfm)(short) = &Z::f1;
```

Note that both `pObject` and `ptdm` (or `ptfm`) in our example may be independently restrict-qualified. Restricting `pObject` promises that only `pObject` will be used to access the object it points to. Restricting `ptdm` promises that the member it addresses will not be accessed either directly (e.g. `object.d1`) or indirectly via another pointer-to-member within its scope.

4.12 Putting it all together

We've reviewed just the most common cases of potential aliasing problems in typical application code and how restrict can be helpful in resolving them. In summary, here are some lessons to take away from this discussion.

Do's

- Understand the promise you make to the compiler (Section 2.3) when you add *restrict* to a pointer **P**'s declaration:

  *Within the scope of the declaration of **P**, only **P** or expressions based on **P** will be used to access the object or subobject pointed to by **P**.*

- Understand that misapplying *restrict* can have undefined and possibly incorrect consequences (Section 3.6).
- Understand the lexical scopes of declarations and thus the extent of a restricted pointer's applicability (Section 3.4).
- Make pointer parameter restrictions part of the interface specification for your functions. In general, restricting one function's parameters implies that pointers in its caller should also be clearly marked as restricted. This becomes an important global property that deserves proper documentation.
- Learn how to examine the compiler's `.asm` and `.nfo` output files to find out if a source change like a restrict qualification was beneficial.
- Learn the most common and fruitful uses of *restrict* – function parameters and pointers that are dereferenced in loops. A C++ member function's *this* pointer can also often be restricted.
- Use restrict when an object is accessed as though it contained disjoint subobjects. Typical examples are interleaved rows of an array and buffers carved out of a single array.
- Assign a pointer in an outer scope to a restrict-qualified local variable in an inner scope. This can give a short name to a long pointer expression and make a promise about aliasing that may not hold for the original pointer over its entire scope. Example:

```
if (current->person[i]->age() >= 21)
{
  char *restrict name = current->person[i]->name;
  ...
}
```

Don'ts

- Don't hide *restrict* in a typedef. It's best that the keyword be visible at the point of declaration of a pointer (Section 3.3).
- Don't remove *restrict* from a pointer declaration if leaving it there clarifies your code, even if it appears to have no effect. Later compiler releases may do better at exploiting it than the one you're using.
- Don't restrict-qualify pointers declared in inner loops; try to move the declaration outside the loop. This avoids complicating such important optimizations as loop unrolling and software pipelining.

- Don't try to second-guess what the compiler can and cannot do with a questionable restrict-qualification. A compiler may give you just the performance and safety you're looking for today, but a later compiler – or the same compiler when the source code changes just a bit – may generate incorrect code.
- Specifying the `--no_bad_aliases` command line option to the compiler can be helpful in finding whether aliasing is a problem that might be correctable. But don't rely on it without verifying that it is safe for all functions in the file and documenting its use.

5 Alternatives to *restrict*

5.1 Assertions

The `assert()` macro provides for diagnostic testing at runtime. If its argument expression evaluates to zero (false), it writes diagnostic information to the standard error stream and calls `abort()`. Assertions are a good way to verify expected properties of the program state before executing code that relies on those properties. But it also provides useful information to the compiler: the asserted properties are true if execution continues past the point of the assertion. Some useful assertions:

```
assert(p1!=0);           // Pointer p1 is not null
assert(((int)p2)%8==0);  // *p2 is aligned on an 8 byte boundary
assert(p1!=p2);          // *p1 and *p2 do not alias
```

Asserting `p1!=p2` has limited usefulness in alias analysis. It says nothing about `p1[i]` and `p2[j]`. And, `*p1` and `*p2` do alias if, say, `p1==p2+1` and `*p2` is wider than `*p1`.

Asserting `N<16` just before `for(i=0;i<N;++i){}` indirectly states that `x[i]` and `x[i+16]` do not alias in the loop.

TI's compilers support a variant of `assert()` called `_nassert()`. Where `assert()` makes a claim to be tested, `_nassert()` makes an untested promise that its argument is non-zero. The declaration

```
#define promise _nassert
```

makes this more explicit. When porting your code to other compilers, just do the following:

```
#define promise
```

5.2 The *const* keyword

The *const* keyword designates an object that is not modifiable; that is, it cannot be assigned a value other than with an initializer. This holds regardless of how the object is accessed.

Example:

```
const float w = 0;     // though const, w may be initialized
const float *p1 = &w;  // const-qualified pointer to w
float       *p2 = &w;  // non-const pointer to w
```

Within the scope of these declarations, `w` may not be modified directly or indirectly via either pointer. Assigning to `*p1` is disallowed simply because `*p1` is declared `const`, regardless of the fact that it references `w`. Assigning to `*p2` is disallowed because attempting to modify a const-qualified object through the use of a reference that is not const-qualified has undefined behavior.[35]

[35] C99 [1] section 6.7.3, para 5.

Const-qualification has little benefit to the compiler for alias disambiguation. *Const* merely states that the qualified object cannot be assigned to by the programmer. In practice, though, the compiler must generate code to write to stack and heap-allocated *const* objects at runtime in order to initialize them. Since it must ignore the *const* in those cases, it generally does in most cases.[36] Fortunately, *restrict* is a much more capable and precise tool than *const*.

5.3 The `--no_bad_aliases` command line option

As explained in Section 2.3, this compiler option asserts that, for each function in the source file that takes two or more pointer parameters, accesses via those parameters do not alias.

Example:

```
void f53a(int *a, int b[10])
{
  *a = b[5] + gvar;  // gvar is a file level int variable
}
```

With the option, the user promises that `*a` and `b[5]` will not alias. Unlike restrict-qualifying `a` or `b`, there is no promise that either will not alias `gvar`. (Note that arrays are passed by reference, so `b` is a pointer.)

The `--no_bad_aliases` (`-mt`) option actually asserts four departures from standard C which may help the compiler improve performance:

- Indirect references on two pointers **P** and **Q** do not alias if **P** and **Q** are distinct parameters of the same function activated by the same call at run time.
- Accesses via pointers to character types do not alias accesses of non-character type.
- Passing a copy of the address of an object as a function argument does not cause that object to be aliased via that copy after the call returns. That is, the callee may read or write memory with the copied pointer but cannot return it or save it away where it might be indirected after the function returns.
- Non-literal but invariant increments of loop index variables are non-zero.

The third assertion overrides the `--aliased_variables` command line option (if used), which asserts just the opposite.

The fourth has a subtle relationship to aliasing analysis. Consider this loop:

```
void f53b(short x[], short y[], int m, int n)
{
  int i;
  for (i=0; i<n; i+=m) x[i] = y[i];
}
```

[36] This decision reflects years of experience compiling programs with misapplied *const* qualifiers. It became inescapable with C++, which is far more likely than C to have initialization code for objects.

If increment `m` might be zero, then the loop might iterate forever, copying `y[0]` to `x[0]`. That's not likely the programmer's intent, but even with a simple loop the compiler might not be able to figure that out. So, in principle, a store to `x[i]` on any iteration could alias the same store on any another iteration.

A better alternative would be to add `_nassert(m!=0)` or, better, `_nassert(m>0)` at the top of the function body.

5.4 C++ valarrays

The C++ standard library provides an interesting capability for the processing of numeric arrays using standard arithmetic operations. Class *valarray* is of some interest here because that standard guarantees that vallarray references are alias free.

We mention this only for completeness --- many scientific users consider valarrays too inefficient. See for example, the discussion in 12.2 of Jusuttis' book [3].

6 Summary

Judiciously placing *restrict* qualifiers in variable and type declarations can help the compiler produce more effective code. This is best done by including *restrict* as part of the API as you design your application. Adding *restrict* to existing code can also be advantageous as part of a strategic approach to achieving full performance for debugged code.

The `-no_bad_aliases` compiler option is a powerful alternative when changes to the source are not permitted or as a quick way to discover whether restrict-qualifying function parameters might be beneficial. Because it applies to all functions in the source file, it requires careful analysis to avoid indicating that memory references cannot alias when in fact they could.

Full support for these features started to become available in compiler releases beginning in 2009. Note that compilers released earlier accept *restrict* and the `-no_bad_aliases` (or `-mt`) option but may exploit them less effectively.

7 Frequently asked questions

This section comprises answers to questions frequently asked about *restrict*.

Does *restrict* help with volatile pointers or pointers to volatile objects?

Few code optimizations can be applied safely to volatile objects. For maximal safety given the variety of uses of `volatile` in embedded programming, TI's compilers assume a write to any volatile object might modify any other, regardless of type. So, there's no advantage to restrict-qualifying a pointer to a volatile object. Note that accessing a volatile-qualified object through the use of a reference that is not volatile-qualified has undefined behavior.[37] The compiler therefore assumes that volatile and non-volatile accesses cannot alias.

On the other hand, a volatile pointer might usefully be restrict-qualified, as it's the access to the object it points to that is being restricted, not access to the pointer.

If I have a pointer to a pointer, which one gets restrict-qualified?

A useful rule of thumb is to mentally draw a vertical line through the ***** in a pointer declaration. Notations added to the right of the line describe the pointer; those to the left pertain to the object pointed to. So we have `int *restrict p` for a restricted pointer to `int`, `int **restrict p` for a restricted pointer to a pointer to `int`, and `int *restrict *p` for a pointer to a restricted pointer to `int`.

How do I know when it's helpful to add restrict to a declaration?

The examples given in this whitepaper have identified many of the most typical situations in which adding *restrict* may be helpful. It's harder to generalize about when it *will* be useful, as opportunities differ for various target processors. Other factors may also hinder optimization.

[37] C99 [1] section 6.7.3, para 5.

Simulation can help determine whether performance improves overall, but it may be impractical to run a simulation for each restrict-qualification you might consider. The next resort is to look at the compiler's output. The `.asm` file may provide useful hints, such as the software pipelining summary generated by the c6x compiler. Upon request (command line options `-on2 -o3`), the compiler will write optimization information to a `.nfo` file. This information includes a pseudo-source listing that shows the effects of redundancy elimination, loop unrolling and many other high-level code transformations. (It does not show the effects of instruction selection and scheduling and register allocation.) Look in particular for a line like

```
There are 14 memory accesses with 3 dependences.
```

The expected effect of improved alias analysis is that the number of dependences decreases.

How do I know when it's safe to add restrict to a declaration?

It's safest to design *restrict* in when designing and documenting your application's control flow and data structures. Restricted function parameters should follow constraints on the use of pointers that reflect the functionality. If a function logically should not pass in references to the same object as distinct pointer arguments, make that fact formal with *restrict* and apparent with suitable comments. If a pointer in a list or other data structure is unique, say so. This is usually true of *next* links in linked lists and *child* links in trees.

Adding restrict later requires careful analysis of the flow of data pointers through your code. It may be possible to retroactively redesign and document that flow as just described. If not, apply restrict-qualification sparingly and carefully. Remember that the compiler can't help you get it right.

I added restrict and it degraded performance! Why?

Constructing a compiler that always finds the best instruction sequence for a compiled source file is impossible. Even if it were possible such a compiler might run for days or years. So compiler developers must rely on a mixture of exact and inexact algorithms for code generation. Inexact methods are generally heuristics that strive for performance (or code size reduction) without precise knowledge of the intent of the programmer or of the runtime behavior of the code. Of course, they must always use conservative techniques that preserve correctness even if potential performance must be sacrificed.

For example, the "best" generated code for a loop may be quite different if it is executed 100 times rather than twice. One class of heuristics would be for a compiler to optimize for the asymptotic behavior of a loop. Yet if it has only a few iterations, the overhead of setting up the loop's iterations may overwhelm the time spent actually executing them.

Another example is code scheduling, which is constrained by dependences between instructions. It's provably impractical to find a perfect schedule for sequences of more than a dozen or two instructions in an acceptable amount of time, so the compiler must use heuristics. Generally we'll find that loosening constraints on the scheduler, such as by removing false aliases, allows for a better schedule to emerge. However, the inexact nature of a practical scheduler may occasionally do worse with the constraint removed.

You may respond to such a degradation by removing the `restrict`. Better, you might look for other ways to improve performance, such as requesting a specific unrolling factor or applying restrict elsewhere. *Hand-Tuning Loops and Control Code on the TMS320C6000* [4] is one good source for ideas.

I added restrict and the code got smaller! Why?

Sometimes smaller code is also faster code. But you may be seeing the removal of a "cloned" loop.

The compiler will sometimes generate two forms of a loop when space is of secondary concern. At runtime, generated code will test whether pointers have values sufficiently far apart that aliasing will not be a problem. If so, an optimized form of the loop will execute. If not, a more conservative form will execute.

Adding *restrict* might make the test and the cloned loop unnecessary. This saves space and slightly speeds up the code at entry to the loop.

When can I use *const* instead of *restrict*? Does it help to use both?

Const-qualification is an unreliable guide to alias analysis because it asserts only that the program does not modify the variable once it is initialized. But especially with C++ it is difficult for the compiler to ascertain whether a runtime assignment is an initialization or a later modification. And each time the scope of a *const* declaration is entered a new instance of the declared variable is created. With code-replicating transformations like function inlining and loop unrolling, the result is that a single auto-initialization may appear to be multiple assignments to the same variable.

Using `const` with a restrict-qualified pointer adds nothing useful to alias analysis.

8 References

1. *ISO/IEC 9899:1999 International Standard Programming Language C*
2. *ISO/IEC 14882:2003 International Standard Programming Language C++*
3. Josuttis, Nicolai M. [1999] *The C++ Standard Library: A Tutorial and Reference*, Addison-Wesley.
4. *Hand-Tuning Loops and Control Code on the TMS320C6000* (SPRA666)
5. http://www.cellperformance.com/mike_acton/2006/05/demystifying_the_restrict_keyw.html
6. Holly, Michael. [1992] "New Keyword for C++: Restrict." X3J16-92-0057 / WG21-N0134

9 Acknowledgements

My thanks go out to the optimizer team, Paul Fuqua, Gary Richey and Yi Qian, and to members of the SDO codegen team and Systems and Applications R&D Center, Alan Davis, Reid Tatge, Jon Humphreys and Branislav Kisacanin, who reviewed earlier drafts. SDO intern Van Bui was a great help early on as "real time" reviewer and tester. And the document would never have been completed without George Mock's encouragement. My sincere thanks to all.