Doc. No.: 01-0051/N1336-03
Date: 9 November 2001
Author: Performance WG

This not-yet-approved-by-Martin revision is dated 10 November 2001 00:51 BST-1

Technical Report on C++ Performance (DRAFT)

Executive summary:

The aim of this report is:

- to give the reader a model of time and space overheads implied by use of various C++ language and library features,
- to debunk widespread myths about performance problems,
- to present techniques for use of C++ in applications where performance matters, and
- to present techniques for implementing C++ language and standard library facilities to yield efficient code.

As far as run-time and space performance is concerned, if you can afford to use C for an application, you can afford to use C++ in a style that uses C++'s facilities appropriately for that application.

This report first discusses areas where performance issues matter, such as various forms of embedded systems programming and high-performance numerical computation. After that, the main body of the report considers the basic cost of using language and library facilities, techniques for writing efficient code, and the special needs of embedded systems programming.

Performance implications of object-oriented programming are presented. This discussion rests on measurements of key language facilities supporting OOP, such as classes, class member functions, class hierarchies, virtual functions, multiple inheritance, and run-time type identification (RTTI). It is demonstrated that, with the exception of RTTI, current C++ implementations can match hand-written low-level code for equivalent tasks. Similarly, the performance implications of generic programming using templates are discussed. Here, however, the emphasis is on techniques for effective use. Error handling using exceptions is discussed based on another set of measurements. Both time and space overheads are discussed. In addition, the predictability of performance of a given operation is considered.

The performance implications of *IOStreams* and locales are examined in some detail and many generally useful techniques for time and space optimisations are discussed here.

Finally, the special needs of embedded systems programming are presented, including ROMability and predictability. And appendices present general C and C++ interfaces to the basic hardware facilities of embedded systems.

Contents:

1 Introduction

"Performance" has many aspects - execution speed, code size, data size, and memory footprint at run-time, or time and space consumed by the edit/compile/link process. It could even refer to the time necessary to find and fix code defects. Most people are primarily concerned with execution speed, although program footprint and memory usage can be critical for small embedded systems where the program is stored in ROM, or where ROM and RAM are combined on a single chip.

Efficiency has been a major design goal for C++ from its earliest days, also the principle of "zero overhead" for any feature that is not used in a program. It has been a guiding principle from the earliest days of C++ that "you don't pay for what you don't use.”

Language features that are never used in a program should not have a cost in extra code size, memory size, or run-time. If there are places where C++ cannot guarantee zero overheads for unused features, this paper will attempt to document them. It will also discuss ways in which compiler writers, library vendors, and programmers can minimize or eliminate performance penalties, and will discuss the trade-offs among different methods of implementation.

Programming for resource-constrained environments is another focus of this paper. Typically, it is very small or very large programs that run into resource limits of some kind. Very large programs, such as database servers, may run into limits of disk space or virtual memory. At the other extreme, an embedded application may be constrained to run in the ROM and RAM space provided by a single chip, perhaps a total of 64K of memory, or even smaller.

Apart from the issues of resource limits, some programs must interface with system hardware on a very low level. Historically the interfaces to hardware have been implemented as proprietary extensions to the compiler (often as macros). This led to the situation that code has not been portable, even for programs written for a given environment, because each compiler for that environment has implemented different sets of extensions.

1.1 Brief Glossary

Virtual memory addressing -- a technique for enabling a program to address more memory space than is physically available. Typically, portions of the memory space not currently being addressed by the processor can be "swapped out" to disk space. A mapping function, sometimes implemented in specialized hardware, translates program addresses into physical hardware addresses. When the processor needs to access an address not currently in physical memory, some of the data in physical memory is written out to disk and some of the stored memory read from disk into hardware memory. Since reading and writing to disk is slower than accessing memory chips, minimizing swaps leads to faster performance.

Page -- a collection of memory addresses treated as a unit, for partitioning memory between applications or swapping out to disk.

Page fault -- an interrupt triggered by an attempt to access a virtual memory address not currently in physical memory, and thus the need to swap virtual memory from disk to hardware.

Overlays – another, older, technique for handling programs that are larger than available memory. Different parts of the program are arranged to share the same memory, with each overlay loaded on demand when another part of the program calls into it. The use of overlays has largely been succeeded by virtual memory addressing where it is available, but it may still be used in memory-limited embedded environments or where precise programmer or compiler control of memory usage improves performance.

Cache -- a buffer of high-speed memory used to improve access times to medium-speed main memory or to low-speed storage devices. If an item is found in cache memory (a "cache hit"), access is faster than going to the underlying device. If an item is not found (a "cache miss"), then it must be fetched from the lower-speed device.

Locality of reference -- the principle that most programs tend to make most accesses to locations near those accessed in the recent past. Keeping items accessed together in locations near each other increases cache hits and decreases page faults.

Working set -- the portion of a running program that is in physical memory, not swapped out, at any given time.

Data size -- the portion of a program's memory image devoted to static data.

Code size -- the portion of a program's memory image devoted to instructions. Sometimes immutable data is placed with the code.

Code bloat -- the generation of excessive amounts of code instructions, as from unnecessary template instantiations.

Real time -- refers to a system in which average performance and throughput must meet defined goals, but some variation in performance of individual operations can be tolerated (also "soft real time"). “Hard real time” means that ***every*** operation ***must*** meet specified timing constraints.

Embedded system -- a program which functions as part of a device. Often the software is burned into firmware instead of loaded from a storage device. It usually is a free-standing implementation rather than a hosted one with an operating system.

1.2 Typical Application Areas

Since no computer has infinite resources, all programs have some kind of limiting constraints. However, many programs never encounter these limits in practice. Very small and very large systems are those most likely to need effective management of limited resources.

1.2.1 Embedded Systems

Embedded Systems have many restrictions on memory-size and timing requirements that are more significant than are typical for non-Embedded Systems. Embedded Systems are used in various application areas as follows[1]:

- **Scale:**
  - **Small**

    These systems typically use single chips containing both ROM and RAM. Single-chip systems in this category typically hold approximately 32KBytes for RAM and 32, 48 or 64KBytes for ROM. (These numbers refer to the popular C8051 chip family.)

    Examples of applications in this category are:

    - Engine control for automobiles
    - Hard disk controllers
    - Consumer electronic appliances
    - Smart cards, also called Integrated Chip (IC) cards – about the size of a credit card, they usually contain a processor system with code and data embedded in a chip which is embedded (in the literal meaning of the word) in a plastic card. A typical size is 4KBytes of RAM, 96KBytes of ROM and 32KBytes EEPROM. An even more constrained smart card in use contains 12KBytes of ROM, 4KBytes of flash memory, and only 600 bytes of RAM data storage.
  - **Medium**

    These systems typically use separate ROM and RAM chips to execute a fixed application, where size is limited. There are different kinds of memory chip, and systems in this category are typically composed of

[1] Typical systems during the Year 2000

several kinds to achieve different objectives of cost and speed. Examples of applications in this category are:

- Hand-held digital VCR
- Printer
- Copy machine
- Digital still camera – one common model uses 32MBytes of flash memory to hold pictures, plus faster buffer memory for temporary image capture, and a processor for on-the-fly image compression.

- **Large**

  These systems typically use separate ROM and RAM chips, where the application is flexible and the size is relatively unlimited. Examples of applications in this category are:

  - Personal Digital Assistant (PDA) – equivalent to a personal computer without a screen, keyboard, or hard disk.
  - Digital television
  - Set-top box
  - Car navigation system
  - Central controllers for large production lines of manufacturing machines

- **Timing:**

  Of course, systems with real-time or hard real-time constraints are not necessarily embedded systems; they may run on hosted environments.

  - **Critical (real-time and hard real-time systems)**

    Examples of applications in this category are:

    - Motor control
    - Hand-held digital VCR
    - Mobile phone
    - CD or DVD player
    - Electronic musical instruments
    - Hard disk controllers
    - Digital television
    - Digital signal processing applications

  - **Non-critical**

    Examples of applications in this category are:

    - Digital still camera
    - Copy machine
    - Printer
    - Car navigation system

1.2.2 Servers:

For server applications, the performance-critical resources are typically speed (e.g. transactions per second), and working-set size (which also impacts throughput and speed). In such systems, memory and data storage are expressed in terms of megabytes or even gigabytes.

Often there are soft real-time constraints, bounded by the need to provide service to many clients in a timely fashion. Some examples of such applications include the central computer of a public lottery where transactions are heavy, or large scale high-performance numerical applications, such as weather forecasting, where the calculation must be completed within a certain time.

These systems are often described in terms of dozens or even hundreds of multiprocessors, and the prime limiting factor may be the Mean Time Between Failure (MTBF) of the hardware (increasing the amount of hardware results in a decrease of the MTBF – in such a case, high-efficiency code would result in greater robustness).

2 Overheads of C++ Features

Does the C++ language have inherent complexities and overheads, which make it unsuitable for performance-critical applications? For a program written in the C-conforming subset of C++, will penalties in code size or execution speed result from using a C++ compiler instead of a C compiler? Does C++ code necessarily result in "unexpected" functions being called at run-time, or are certain language features, like multiple inheritance or templates, just too expensive (in size or speed) to risk using? Do these features impose overheads even if they aren't explicitly used?

This paper examines the major features of the C++ language that are perceived to have an associated cost, whether real or not.

- Namespaces
- Type Conversion Operators
- Inheritance
- Run-Time Type Information (RTTI)
- Exception Handling (EH)
- Templates
- The Standard IOStreams Library

2.1 Namespaces

Namespaces do not add any space or time overheads to code. They do, however, add some complexity to the rules for name lookup. The principal advantage of namespaces is that they provide a mechanism for partitioning names in large projects in order to avoid name clashes.

Namespace qualifiers enable programmers to use shorter identifier names when compared with alternative mechanisms. In the absence of namespaces, the programmer has to explicitly alter the names to ensure that name clashes do not occur, and this usually takes the form of a canonical prefix being used:

```
static char* mylib_name      = "My Really Useful Library";
static char* mylib_copyright = "June 15, 2000";
```

Another approach is to place the names inside a class and use them in qualified form:

```
class ThisLibInfo {
    static char*  name;
    static char* copyright;
};

char* ThisLibInfo::name      = "Another Useful Library";
char* ThisLibInfo::copyright = "December 18, 2000";
```

With namespaces, the number of characters necessary is similar to the `class` alternative, but unlike the `class` alternative, qualification can be avoided with `using` declarations, which move the unqualified names into the current scope, thus allowing the names to be referenced by their shorter form. This saves the programmer the effort of typing those extra characters into the source program.

2.2 Type Conversion Operators

C and C++ permit explicit type conversion using *cast notation* (§IS-5.4). *[for example:*

```
int i = (int) 3.14159;
```

end example]

Standard C++ adds four additional *type conversion operator*s, using syntax that looks like *function-templates [for example:*

```
int i = static_cast<int> ( 3.14159 );
```

end example]

The four syntactic forms are:

```
const_cast<Type>(expression)        // §IS-5.2.11
static_cast<Type>(expression)       // §IS-5.2.9
reinterpret_cast<Type>(expression)  // §IS-5.2.10
dynamic_cast<Type>(expression)      // §IS-5.2.7
```

The semantics of *cast notation* (which is still recognized) are the same as the *type conversion operator*s, but distinguish between the different purposes for which the cast is being used. The *type conversion operator* syntax is easier to identify in source code, and thus contributes to writing programs that are more likely to be correct[2]. It should be noted that as in C, a cast may create a temporary object of the desired type, so casting can have run-time implications.

Lois: I'm having trouble getting the footnotes to number themselves correctly. Needs editorial attention at some point. Also, there seem to be a lot of bogus footnotes that I can't get rid of.

The first three forms of *type conversion operator* have no size or speed penalty versus the equivalent *cast notation*. Indeed, it is typical for a compiler to transform *cast notation* into one of the other *type conversion operator*s when generating object code. However, `dynamic_cast<T>` may incur some overhead at run-time if the required conversion involves using RTTI mechanisms such as cross-casting [see 2.3.8].

2.3 Classes and Inheritance

Programming in the object-oriented style often involves heavy use of class hierarchies. This section examines the time and space overheads imposed by the

[2] If the compiler does not provide the *type conversion operators* natively, it is possible to implement them using *function-templates*. Indeed, prototype implementations of the *type conversion operators* were often implemented this way.

primitive operations using classes and class hierarchies. Often, the alternative to using class hierarchies is to perform similar operations using lower-level facilities. For example, the obvious alternative to a virtual function call is an indirect function call. For this reason, the cost of primitive operations of classes and class hierarchies are compared to similar functionality implemented without classes.

Most comments about run-time costs are based on a set of simple measurements performed on three different machine architectures using six different compilers run with a variety of optimisation options. Each test was run multiple times to ensure that the results were repeatable. The code is presented in Appendix D. The aim of these measurements is neither to get a precise statement of optimal performance of C++ on a given machine nor to provide a comparison between compilers or machine architectures. Rather, the aim is to give developers a view of relative costs of common language constructs using current compilers, and also to show what is possible – what is achieved in one compiler is in principle possible for all. We know – from specialized compilers not in this study and reports from people using unreleased beta versions of popular compilers – that better results are possible.

In general, the statements about implementation techniques and performance are believed to be true for the vast majority of current implementations, but are not meant to cover experimental implementation techniques, which might produce better – or just different – results.

2.3.1 Representation Overheads

A `class` without a virtual function requires exactly as much space to represent as a `struct` with the same data members. That is, no space overhead is introduced from using a class compared to a C `struct`. A `class` object does not contain any data that the programmer didn't explicitly request. In particular, a non-virtual function does not take up any space in an object of its class. Similarly, a static member takes up no space in an object.

A `class` that has one or more virtual functions incurs a space overhead of one pointer per object plus a "virtual function table" (`vtbl`) of one to two words per virtual function plus a "type information" object with a size on the order of a couple of words + a name string + a couple of words per base-class. This latter "type information" (also called "run-time type information" or RTTI) is typically about 40 bytes per class. Whole program analysis (WPA) can be used to eliminate unused `vtbl`s and RTTI data. Such analysis is particularly suitable for relatively small programs that don't use dynamic linking and that have to operate in a resource-constrained environment, such as an embedded system.

Some current C++ implementations share data structures between RTTI support and exception handling support, thereby avoiding representation overhead specifically for RTTI.

Aggregating data items into a `class` or `struct` can impose a run-time overhead if a compiler doesn't effectively use registers or in other ways fails to take advantage of possible optimisations when class objects are used. The overheads incurred through the failure to optimise in such cases are referred to as "the abstraction penalty" and are usually measured by a benchmark produced by Alex Stepanov. For example, if accessing a value through a trivial smart pointer is significantly slower than accessing it through an ordinary pointer, the compiler is inefficiently handling the abstraction. In the past, most compilers had significant abstraction penalties and several current compilers still do. However, at least two compilers have been reported to have abstraction penalties below 1% and another a penalty of 3%, so eliminating this kind of overhead is well within the state of the art. These are production compilers, not just experimental ones.

2.3.2 Basic Class Operations

Calling a non-virtual, non-static, non-inline member function of a class costs as much as calling a freestanding function with one extra pointer argument indicating the data on which the function should operate. Consider a set of simple runs of the test program from Appendix D:

Table 2.3.1		#1	#2	#3	#4	#5
`non-virtual:`	`px->f(1)`	0.019	0.002	0.016	0.085	0
	`g(ps,1)`	0.020	0.002	0.016	0.067	0
`non-virtual:`	`x.g(1)`	0.019	0.002	0.016	0.085	0
	`g(&s,1)`	0.019	0	0.016	0.067	0.001
`static member:`	`X::h(1)`	0.014	0	0.013	0.069	0
	`h(1)`	0.014	0	0.013	0.071	0.001

The compiler/machine combinations #1 and #2 match "naïve expectations" exactly by having calls of a member function exactly match calls of a non-member function with an extra pointer argument. As expected, the two last calls (the `X::h(1)` call of a static member function and the `h(1)` call of a global function) are faster because they don't pass a pointer argument. Implementations #3 and #5 demonstrate that a clever optimiser can take advantage of implicit inlining and (probably) caching to produce results for repeated calls that are 10 times (or more) faster than what is achievable if a function call is generated. Implementation #4 shows a small (<15%) advantage to non-member function calls over member function calls, which (curiously) is reversed when no pointer argument is passed. #1, #2, and #3 were run on one system, while #4 and #5 were run on another.

The main lesson drawn from this table is that any differences that there may be between non-virtual function calls and non-member function calls are minor and far less important than differences between compilers/optimisers.

2.3.3 Virtual Functions

Calling a virtual function is roughly equivalent to calling a function through a pointer stored in an array:

Table 2.3.2		#1	#2	#3	#4	#5
`virtual:`	`px->f(1)`	0.025	0.012	0.019	0.078	0.059
`ptr-to-fct:`	`p[1](ps,1)`	0.020	0.002	0.016	0.055	0.052
`virtual:`	`x.f(1)`	0.020	0.002	0.016	0.071	0
`ptr-to-fct:`	`p[1](&s,1)`	0.017	0.013	0.018	0.055	0.048

Here it is the compiler/machine combination #3 that most closely matches the naïve model of what is going on. The cost of virtual and pointer to function calls is identical (when averaged over a few runs, the minor difference seen above averages out) except for `x.f(1)` where implementations #2 and #5 notice that the virtual function table need not be used because the exact type of the object is known and a non-virtual call can be used. This simple and useful optimisation was present in Cfront 1.1 (1986)[3] and it is a bit sad to see that it can't be taken for granted in 2001. Implementations #4 and #5 appear to have systematic overheads for virtual function calls (caused by treating single-inheritance and multiple-inheritance equivalently, and thus missing an optimisation). However, this overhead is on the order of 20% and 12% - far less than the variability between compilers.

Comparing table 1 and table 2, we see that implementations #1, #2, #3, and #5 confirm the obvious assumption that virtual calls (and indirect calls) are more expensive than non-virtual calls (and direct calls). Interestingly, the overheads are in the range 20% to 25% where one would it expect it to be, based on a simple count of operations performed. However, implementations #2 and #5 demonstrate how (implicit) inlining can yield much larger gains for non-virtual calls. Implementation #4 counter-intuitively shows virtual calls to be faster than non-virtual ones. If nothing else, this shows the danger of measurement artefacts. It may also show the effect of additional effort in hardware and optimisers to improve the performance of indirect function calls.

2.3.3.1 Virtual Functions of *class-template*s

Virtual functions of a ***class-template*** can incur an overhead that is sometimes unexpected: If a ***class-template*** has virtual member functions, then each time the ***class-template*** is specialised it will have to generate new specialisations of the member functions, and their associated support structures such as the virtual function table (`vtbl`). A naïve library implementation could produce hundreds of Kbytes in this case, much of which is pure replication at the instruction level of the program. The problem is a library modularity issue. Putting code into the `template` when it doesn't depend on template parameters, and could be separate code, may cause each instantiation to contain potentially large, redundant code sequences. One optimization available to the programmer is to use non-template helper functions, and describe the template implementation in terms of these helper functions. A similar technique

[3] The original implementation of a C++ to C translator from AT&T.

places functionality that doesn't need to be templatized in a non-template base class. This technique is used in several places in the standard library. Finally, it can be noted that templatization and use of virtual functions are often complementary techniques, so that a template class with many virtual function should be carefully considered to determine if it might be a design error.

2.3.4 Inlining

The discussion above considers the cost of a function call to be a simple fact of life (it doesn't consider it to be overhead). However, many function calls can be eliminated through inlining. C++ allows inlining to be explicitly requested for a function, and popular descriptions of the language seem to encourage this for small time-critical functions. Basically, C++'s `inline` is meant to be used as a replacement for C's function-style macros. To get an idea of the effectiveness of `inline`, we compare calls of an inline member of a class to a non-inline member and to a macro.

Table 2.3.3		#1	#2	#3	#4	#5
`non-inline:`	`px->g(1)`	0.019	0.002	0.016	0.085	0
`non-inline:`	`x.g(1)`	0.019	0.002	0.016	0.085	0
`inline:`	`ps->k(1)`	0.007	0.002	0.006	0.005	0
`macro:`	`K(ps,1)`	0.005	0.003	0.005	0.006	0
`inline:`	`x.k(1)`	0.005	0.002	0.005	0.006	0
`macro:`	`K(&s,1)`	0.005	0	0.005	0.005	0.001

The first observation here is that inlining provides a significant gain over a function call (the body of these functions are a simple expression, so this is the kind of function where one would expect the greatest advantage from inlining). The exceptions are implementations #2 and #5, which already have achieved significant optimisations through implicit inlining. However, implicit inlining cannot (yet) be relied upon for consistent high performance. For other implementations, the advantage of inlining is significant (factors of 2.7, 2.7, and 17).

2.3.5 Multiple-Inheritance

When implementing multiple-inheritance, there is a wider array of implementation techniques than for single-inheritance. The fundamental problem is that each call has to ensure that the "`this` pointer" passed to the called function points to the correct sub-object. This can cause time and/or space overhead. The `this` pointer adjustment is usually done in one of two ways:

- The caller retrieves a suitable offset from the `vtbl` and adds it to the pointer to the called object, or

- a "thunk" is used to perform this adjustment. A "thunk" is a simple piece of code that is called instead of the actual function, and which performs the actual constant adjustment to the object pointer before transferring control to the intended function.

Table 2.3.4		#1	#2	#3	#4	#5
`SI, non-virtual:`	`px->g(1)`	0.019	0.002	0.016	0.085	0
`Base1, non-virtual:`	`pc->g(i)`	0.007	0.003	0.016	0.007	0.004
`Base2, no-virtual:`	`pc->gg(i)`	0.007	0.004	0.017	0.007	0.028
`SI, virtual:`	`px->f(1)`	0.025	0.013	0.019	0.078	0.059
`Base1, virtual:`	`pa->f(i)`	0.026	0.012	0.019	0.082	0.059
`Base1, virtual:`	`pb->ff(i)`	0.025	0.012	0.024	0.085	0.082

Here, implementations #1 and #4 managed to inline the non-virtual calls in the multiple-inheritance case, where they had not bothered to do so in the single-inheritance case. This demonstrates the effectiveness of optimisation and also that we cannot simply assume that multiple-inheritance imposes overheads.

It appears that implementations #1 and #2 don't incur extra overheads from multiple-inheritance compared to single-inheritance. This could be caused by imposing multiple-inheritance overheads redundantly even in the single-inheritance case. However, the comparison between (single-inheritance) virtual function calls and indirect function calls in table 2.3.2 shows this not to be the case.

Implementations #3 and #5 show overhead when using the second branch of the inheritance tree, as one would expect to arise from a need to adjust a "`this` pointer". As expected, that overhead is minor (25% and 20%) except where implementation #5 misses the opportunity to inline the call to the non-virtual function on the second branch. Again, differences between optimisers dominate differences between different kinds of calls.

2.3.6 Virtual base classes

A virtual base class adds additional overhead compared to a non-virtual ("ordinary") base class. The "adjustment" for the branch in a multiply-inheriting class can be determined statically by the implementation, so it becomes a simple add of a constant when needed. With virtual bases, the position of the base object with respect to the complete object is dynamic and requires more evaluation, typically an indirection through a pointer, than for the non-virtual MI adjustment.

Table 5	#1	#2	#3	#4	#5
Single inheritance, non-virtual: px->g(1)	.019	.002	.016	.085	0
Virtual base non-virtual: pd->gg(i)	.01	.01	.021	.03	.027
Single inheritance, virtual: px->f(1)	.025	.013	.019	.078	.059
Virtual base, virtual: pa->f(i)	.028	.015	.025	.081	.074

For "non-virtual function calls", implementation #3 appears closest to the naïve expectation of a slight overhead. For implementations #2 and #5 that "slight overhead" becomes significant because the indirection implied by the virtual base causes them to miss the opportunity for optimisation. There doesn't appear to be a fundamental problem in inlining in this case, but it is most likely not common enough for the implementers to have bothered – so far. Implementations #1 and #4 again appear to be missing a significant optimisation opportunity for "ordinary virtual function calls". Using a virtual base counterintuitively produces faster code.

The overhead implied by using a virtual base in a virtual call appears small. Implementations #1 and #2 keep it under 15%, implementation #4 gets that overhead to 3% but (from looking at implementation #5) that is done by missing optimisation opportunities in the "normal" single inheritance virtual call case.

As always, simulating the effect of the language feature through other language features also carries a cost. If a programmer decides not to use a virtual base, yet requires a class that can be passed around as the interface to a variety of classes, an indirection is needed in the access to that interface and some mechanism for finding the proper class to be invoked by a call through that interface must be provided. This mechanism would at least be as complex as the implementation of a virtual base, much harder to use, and less likely to attract the attention of optimizers.

2.3.7 Type Identification

Given an object of a polymorphic class (a class with at least one virtual function), a **type_info** object can be obtained through the use of the **typeid** operator. In principle, this is a simple operation involving finding the **vtbl**, through that finding the most-derived-class object of which the object is part, and then extracting a pointer to the **type_info** object from that object's **vtbl** (or equivalent). To provide a scale, we have added the cost of a call of a global function taking one argument:

Table 2.3.6	#1	#2	#3	#4	#5
Global: h(1)	.014	0	.013	.071	.001
On base: typeid(pa)	.079	.047	.218	.365	.059
On derived: typeid(pc)	.079	.047	.105	.381	.055
On virtual base: typeid(pa)	.078	.046	.217	.379	.049

Vbase, on derived: typeid(pd)	.081	.046	.113	.382	.048

There is no reason for the speed of typeid to differ depending on whether a base is virtual or not, and the implementations reflect that. Conversely, one could imagine a difference between a typeid for a base class and a typeid on an object of the most derived class. Implementation #3 demonstrates that. In general, typeid seems very slow compared to a function call and the small amount of work required. We suspect this high cost is caused primarily by typeid being an infrequently used operation that so far hasn't attracted the attention of optimiser writers.

2.3.8 Dynamic cast

Given an object of a polymorphic class, we can cast to another sub-object of the same derived class object using a **dynamic_cast**. In principle, this operation involves finding the **vtbl**, through that finding the most-derived-class object of which the object is part, and then using type information associated with that object to determine if the conversion (cast) is allowed and perform any required adjustments of the **this** pointer. In principle, this checking involves traversing a data structure describing the base classes of the most derived class. Thus, the run-time of a **dynamic_cast** may depend on the relative positions of the two classes involved in the class hierarchy.

Table 2.3.7	#1	#2	#3	#4	#5
Virtual call: px->f(1)	.025	.013	.019	.078	.059
Upcast to base1: cast(pa,pc)	.007	0	.003	.006	0
Upcast to base2: cast(pb,pc)	.008	0	.004	.007	.001
Downcast from base1: cast(pc,pa)	.116	.148	.066	.64	.063
Downcast from base2: cast(pc,pb)	.117	.209	.065	.632	.07
Crosscast: cast(pb,pa)	.305	.356	.768	1.332	.367
2-level upcast to base1: cast(pa,pcc)	.005	0	.005	.006	.001
2-level upcast to base2: cast(pb,pcc)	.007	0	.006	.006	.001
2-level downcast from base1: cast(pcc,pa)	.116	.148	.066	.641	.063
2-level downcast from base2: cast(pcc,pb)	.117	.203	.065	.634	.077
2-level crosscast: cast(pa,pb)	.3	.363	.768	1.341	.377
2-level crosscast: cast(pb,pa)	.308	.306	.775	1.343	.288

As with **typeid**, we see the immaturity of the optimizer technology. However, **dynamic_cast** is a more promising target for effort than is **typeid**. While **dynamic_cast** is not an operation we expect to see in a performance critical loop of a well written program, it does have the potential to be used frequently enough to warrant optimisation:

- An upcast (cast from derived class to base) can be compiled into a simple **this** pointer adjustment, as done by implementations #2 and #5.

- A downcast (from base class to derived) can be quite complicated (and therefore quite expensive in terms of run-time), but many cases are simple. Implementation #5 shows that a downcast can be optimised to the equivalent of a virtual function call, which examines a data structure to determine the necessary adjustment of the **this** pointer (if any). The other implementations use simpler strategies involving several function calls (about 4, 10, 3, and 10 calls, respectively).
- Crosscasts (casts from one branch of an multiple-inheritance hierarchy to another) are inherently more complicated than downcasts. However, a crosscast could in principle be implemented as a downcast followed by an upcast, so one should expect the cost of a crosscast to converge on the cost of a downcast as optimiser technology matures. Clearly the implementations have a long way to go.

2.4 Exception Handling

Exception Handling provides a systematic and robust approach to errors that cannot be handled locally at the point where they are detected.

The traditional alternatives to exception handling (in C, C++, and other languages) include:

Returning "error codes"

Setting error state indicators (e.g. `errno`)

Calling error handling functions

Escaping from a context into error handling code using `longjmp`

Passing along a pointer to a state object with each call

When considering exception handling, it must be contrasted to alternative ways of dealing with errors. Plausible areas of comparison include:

Programming style

Robustness and completeness of error handling code

Run-time system (memory size) overheads

Overheads from handling an individual error.

Consider a trivial example:

```
double f1(int a) { return 1/a; }
double f2(int a) { return 2/a; }
double f3(int a) { return 3/a; }
double g(int x, int y, int z)
{
      return f1(x)+f2(y)+f3(z);
}
```

This code contains no error handling code. Using a variety of pre-exception error handling techniques we could detect and report errors:

```
void error(const char* e)
```

```
{
      // handle error
}

double f1(int a)
{
      if (a<=0)error("bad input value for f1()");
      return 1/a;
}

int error_state = 0;

double f2(int a)
{
      if (a<=0) error_state = 7;
      return 2/a;
}

double f3(int a, int* err)
{
      if (a<=0) *err = 7;
      return 3/a;
}

int g(int x, int y, int z)
{
      double xx = f1(x);
      double yy = f2(y);
      if (error_state) {
            // handle error
      }
      int state = 0;
      double zz = f3(z,&state);
      if (state) {
            // handle error
      }
      return xx+yy+zz;
}
```

Ideally a real program would use a consistent error-handling style, but such consistency is often hard to achieve in a large program. Note that the "`error_state`" technique is not thread safe. Note also that it is hard to use the "`error()` function" technique effectively in programs where `error()` may not terminate the program. However, the key point here is that any way of dealing with errors that cannot be handled locally implies space and time overheads. It also complicates the structure of a program.

Using exceptions the example could be written like this:

```
struct Error {
      int error_number;
      Error(int n) :error_number(n) { }
};

double f1(int a)
{
      if (a<=0)throw Error(1);
```

```
        return 1/a;
}
double f2(int a)
{
        if (a<=0) throw Error(2);
        return 2/a;
}

double f3(int a)
{
        if (a<=0) throw Error(3);
        return 3/a;
}

int g(int x, int y, int z)
try
{
        return f1(x)+f2(y)+f3(z);
}
catch(Error& err) {
        // handle error
}
```

When considering the overheads of exception handling, we must remember to take into account the cost of alternative error handling techniques. Here are run times of the three variants of the trivial program on a few compilers:

????				

<<table, also measure with exceptions-disabled switch>>

The use of exceptions isolates the error-handling code from the normal flow of program execution, and unlike the error code approach, it cannot be ignored or forgotten. Also, automatic destruction of stack objects when an exception is thrown renders a program less likely to leak memory or other resources. With exceptions, once a problem is identified, it cannot be ignored – failure to catch and handle an exception results in program termination (for that reason, many programs catch all exceptions in `main()` to ensure graceful exit from totally unexpected errors). For a discussion of techniques for using exceptions, see Appendix E of TC++PL3.

Early implementations of Exception Handling resulted in sizeable increases in code size and/or some run-time overhead. This led some programmers to avoid it and compiler vendors to provide switches to suppress the use of exceptions. In some embedded and resource-constrained environments, use of exceptions was deliberately excluded either because of fear of overheads or because available exception implementations didn't meet a project's requirements for predictability.

We can distinguish three sources of overhead:

- **Try-block costs**: data and code associated with each *try-block* or *catch-clause*.

- **Regular function costs**: data and code associated with the normal execution of functions that would not be needed had exceptions not existed, such as missed opportunities for optimisation.
- **Throw costs**: data and code associated with throwing an exception.

Each source of overhead has a corresponding overhead of handling an error using traditional error-handling techniques.

2.4.1 Exception Handling implementation issues and techniques

The implementation of exception handling must address several issues:

- ***try-block*** Establishes the context for associated *catch-clauses.*
- ***catch-clause*** The EH implementation must provide some run-time type-identification mechanism for finding *catch-clause*s when an exception is thrown.

  There is some overlapping, but not identical, information needed by both RTTI and EH features. But the EH type-information mechanism must be able to match derived classes to base classes even for types without virtual functions, and to identify built-in types such as `int`. On the other hand, the EH type-information does not need support for *down-casting* or *cross-casting*.

  Because of this overlap, some implementations require that RTTI be enabled when EH is enabled.

- ***Cleanup of handled exceptions*** Exceptions which are not re-thrown must be destroyed upon exit of the *catch-clause*. The memory for the exception object must be managed by the EH implementation.
- ***Automatic and temporary objects with non-trivial destructors*** Destructors must be called if an exception occurs after construction of an object and before destruction, even if no try/catch is present. The EH implementation is required to keep track of all such objects.
- ***Construction of objects with non-trivial destructors*** If an exception occurs during construction, all completely constructed base classes and sub-objects must be destroyed. This means that the EH implementation must track the current state of construction of an object.
- ***throw-expression*** A copy of the exception object being thrown must be allocated in memory provided by the EH implementation. The closest matching *catch-clause* must then be found using the EH type-information. Finally, the destructors for automatic, temporary, and partially constructed objects must be executed before control is transferred to the *catch-clause*.
- ***Enforcing exception specifications*** Conformance of the thrown types to the list of types permitted in the *exception-specification* must be checked. If a mismatch is detected, the *unexpected-handler* must be called.

  A similar mechanism to the one implementing try/catch can be used, but if a mismatch does occur, the *unexpected-handler* is called.
- ***`operator new`*** After calling the destructors for the partially constructed object, the corresponding `operator delete` must be called if an exception is thrown during construction.

  Again, a similar mechanism to the one implementing try/catch can be used.

Implementations vary in how costs are allocated across these elements.

The two main strategies are

- the "code" approach, where code is associated with each try block, and
- the "table" approach that uses compiler-generated static tables.

There are also various hybrid approaches. This paper discusses only the two principal implementation approaches.

2.4.1.1 The "code" Approach.

Implementations using this approach have to dynamically maintain auxiliary data-structures to manage the capture and transfer of the execution contexts, plus other dynamic data-structures involved in tracking the objects that need to be unwound in the event of an exception. Early implementations of this approach used `setjmp/longjmp` to return to a previous context. However, better performance can be obtained using special-purpose code. It is also possible to implement this model through the systematic use of (compiler generated) return codes.

- ***try-block*** Save the execution environment and reference to catch code on EH stack at *try-block* entry.
- ***Automatic and temporary objects with non-trivial destructors*** Register each constructed object together with its destructor, preparing for later destruction. Typical implementations use a linked list structure on the stack. If an exception is thrown this list is used to determine which objects to destroy.
- ***Construction of objects with non-trivial destructors*** One well-known implementation increments a counter for each base-class and sub-object as they are constructed. If an exception is thrown during construction, the counter is used to determine which parts need to be destroyed.
- ***throw-expression*** After the *catch-clause* has been found, invoke destructors for all constructed objects in the part of the stack between the *throw-expression* and the *catch-clause*. Restore execution environment associated with the *catch-clause* .

2.4.1.1.1 Space Overheads of the Code Model

- No exception-handling cost is associated with an individual object, so object size is unaffected
- Exception handling implies a form of RTTI, which requires some increase in data size
- Exception-handling code is inserted into the object code for each try/catch
- Code registering the need for destruction is generated for each stack object of a type with a destructor.
- A cost is associated with checking the *throw-specification*s of the functions that are called

2.4.1.1.2 Time Overhead of the Code Model

- On entry to each *try-block*
  - commit changes to variables enclosing the *try-block*
  - stack the execution context
  - stack the associated *catch-clause*s
- On exit from each *try-block*
  - remove the associated *catch-clause*s
  - remove the stacked execution context
- On entry to each *catch-clause*
  - remove the associated *catch-clause*s
- On exit from each *catch-clause*
  - retire the current exception object (destroy if necessary)
- When calling regular functions
  - if the function has an *exception-specification*, register it for checking
- As each local and temporary object is created
  - register with the current exception context as they are created

- On throw
    - locate the corresponding *catch-clause* (if any) - this involves some RTTI-like check
      if found:
        - destroy the registered local objects
        - check the *exception-specification*s of the functions called in-between
        - use the associated execution context of the *catch-clause*

      if not found:
        - call the *unexpected-handler*

This "code" model distributes the code and associated data structures throughout the code. This means that no separate run-time support system is needed. Such an implementation can be portable and compatible with implementations that translate C++ to C or another language.

The primary disadvantage of the "code" model is that the stack space and run-time costs for *try-block* entry, and for the bookkeeping of automatic, temporary and partially constructed objects as the exception-handling stack is modified, must be done even when no exceptions are thrown. That is, code unrelated to error handling is slowed down by the mere possibility of exceptions being used. This is similar to error-handling strategies consistently checking error state or return values.

The cost of this (in this model, unavoidable) bookkeeping varies dramatically from implementation to implementation. However, one vendor reports speed impact of about 6% for a C++ to ANSI C translator. This is generally considered a very good result.

2.4.1.2 The "table" Approach

Typical implementations using the static approach will generate read-only tables for determining the current execution context, locating *catch-clause*s, and tracking objects needing destruction.

- ***try-block*** This method incurs no run-time cost. All bookkeeping is pre-computed as a mapping between program counter and code to be executed in event of an exception. Tables increase program image size but may be moved away from working set to improve locality of reference. Tables can be placed in ROM, and on hosted systems with Virtual Memory, can remain swapped out until an exception is actually thrown.

- ***Objects with non-trivial destructors, including automatic and temporary objects*** No run-time costs associated with normal execution. Only in the event of an exception is it necessary to intrude on normal execution.

- ***throw-expression*** The statically generated tables are used to locate matching handlers and intervening objects needing destruction. Again, no run-time costs are associated with normal execution.

2.4.1.2.1 Space Overhead of the Table Model

- No exception-handling cost is associated with an object, so object size is unaffected
- Exception handling implies a form of RTTI, which requires some increase in data size
- This model uses statically allocated tables and some common library run-time support
- A cost is associated with checking the *throw-specification*s of the functions that are called

2.4.1.2.2 Time Overhead of the Table Model

- On entry to each *try-block*
    - some implementations commit changes to variables in the scopes enclosing the *try-block* - other implementations use a more sophisticated state table[4]
- On exit from each *try-block*
    - no overhead
- On entry to each *catch-clause*
    - no overhead
- On exit from each *catch clause*
    - no overhead
- When calling regular functions
    - no overhead
- As each local and temporary object is created
    - no overhead
- On throw
    - using the tables, determine if the current frame has an appropriate *catch-clause*

      If it does, then:
        - destroy all local, temporary and partially constructed objects that occur between the *throw-expression* and the *catch-clause*
        - transfer control to the *catch-clause*

      Otherwise, check that the exception honours the *exception-specification* of the current function, and call the *unexpected-handler* if it does not.

      Otherwise, if there is a previous frame, repeat the above steps, otherwise call the *unexpected-handler*

The primary advantage of this method is that no stack or run-time costs are associated with managing the try/catch or object bookkeeping. Unless an exception is thrown, no run-time overhead is incurred.

---

[4] In such implementations, this effectively makes the variables partially `volatile` and may prejudice other optimisations as a result.

Disadvantages are that the implementation is more complicated, and does not lend itself well to implementations that translate to an intermediate language. The static tables can be quite large. This may not be a burden on systems with virtual memory, but the cost can be significant for some embedded systems. All run-time costs associated occur when an exception is thrown. However, because the size of the state table (or tables) depends on the size of the program, the time it takes to respond to an exception may be large, variable, and dependent on program size. This needs to be factored into the probable frequency of exceptions. The extreme case is a system optimised for infrequent exceptions where the first throw of an exception may cause disk accesses.

One vendor reports a code and data space impact of about 15% for the generated tables. It is possible to do better because this vendor had no need to optimise for space.

2.4.2 Predictability of Exception Handling Overhead

2.4.2.1 Prediction of throw/catch Performance

In some programs, it is a problem that it is hard to predict the time needed to pass control from a throw to an appropriate catch clause. This uncertainty comes from the need to destroy automatic objects and – in the "table" model – from the need to consult the table. In some systems, especially those with "Real Time" requirements, it is important to be able to accurately predict how long operations will take.

For this reason current exception-handling implementations may be unsuitable for some applications. However, if the call tree can be statically determined, and the table method of EH implementation is used, it is possible to statically analyse the sequence of events necessary to transfer control from a given throw-expression to the corresponding catch-clause. Each of the events could then be statically analysed to determine their contribution to the cost, and the whole sequence of events aggregated into a single cost domain (worst-case & best-case, unbounded, indeterminate). Such analysis does not differ in principle from current time estimating methods used for non-exception code.

It should be possible to accurately determine the costs of the EH mechanism itself, and the cost of any destructors invoked would need to be determined in the same way as the cost of any other functions is determined.

Given such analyses, the term "unpredictable" is inappropriate. The cost may be quite predictable, with a well-determined upper and lower bound. In some cases (recursive contexts, or conditional call trees), the cost may not be determined statically. For Real Time applications, it is generally most important to have a determinate time domain, with a small deviation between the upper and lower bound. The actual speed of execution is often less important.

2.4.2.2 Exception Specifications

In general, an *exception-specification* must be checked at run-time. For example:

```
void f(int x) throw(A,B)
{
       // …
}
```

will in a straightforward implementation generate code roughly equivalent to

```
void g(int x)
{
       try {
              // …
       }
       catch (A) {
              throw;
       }
       catch (B) {
              throw;
       }
       catch(…) {
              unexpected();
       }
}
```

In principle, static analysis (especially whole program analysis) can be used to eliminate such tests. This may be especially relevant for applications that do not support dynamic linking, are not so large or complex as to defeat analysis, and do not change so frequently as to make analysis expensive. Dependent on the implementation, empty *exception-specification*s can be especially helpful for optimisation.

The use of an empty *exception-specification* should reduce overheads. The caller of a function with an empty *exception-specification* can perform optimisations based on the knowledge that a called function will never throw any exception. In particular, objects with destructors in a block where no exception can be thrown need not be protected against exceptions. That is, in the "code" model no registration is needed, and in the "table" model no table entry needs to be made for that object. For example:

```
int f(int a) throw();

char g(const string& s)
{
       string s2 = s;
       int max = s.size();
       int x = f(max);
       if (x<0||max<=x) x = 0;
       return s2[x];
}
```

Here the compiler need not protect against the possibility of an exception being thrown after the construction of s2.

There is of course no guarantee that a compiler actually performs this optimization. However, a compiler intended for high-performance use could hardly fail to perform it.

Additional text goes here, discussing how empty throw specs on the lowest level functions can be particularly helpful.

2.5 Templates

2.5.1 Template Overheads

A class-template or *function-template* will generate a new instantiation of code each time it is specialised with different template parameters. This can lead to an unexpectedly large amount of code and data[5]. A typical way to illustrate this problem is to create a large number of Standard Library containers to hold pointers of various types. Each type can result in an extra set of code and data being generated.

In one experiment, a program instantiating 100 instances of a single specialisation of `std::list<T*>` for some type `T`, was compared with a second program instantiating a single instance of `std::list<T*>` for 100 different types `T`. These programs were compiled with a number of different compilers and a variety of different compiler options. The results varied widely, with one compiler producing code for the second program that was over 19 times as large as the first program; and another compiler producing code for the first program that was nearly 3 times as large as the second.

The optimisation here is for the compiler to recognise that while there may be many specialisations with different types, at the level of machine code-generation the specialisations may actually be identical (the type system is not relevant to machine code).

While it is possible for the compiler or linker to perform this optimisation automatically, the optimisation can also be performed by the Standard Library implementation or by the application programmer.

If the compiler supports *partial specialization* and *member-function-template*s, the library implementor can provide partial specialisations of containers of pointers to a single underlying implementation that uses `void*`. This technique is described in C++ PL 3rd edition.

In the absence of compiler or library support, the same optimization technique can be employed by the programmer, by writing a class-template called, perhaps, `plist<T>`, that is implemented using `std::list<void*>` to which all operations of `plist<T>` are delegated.

Source code must then refer to `plist<T>` rather than `std::list<T*>`, so the technique is not transparent, but it is a workable solution in the absence of tool or library support. Variations of this technique can be used with other templates too.

2.5.2 Templates vs. Inheritance

Any non-trivial program needs to deal with data structures and algorithms. Because data structures and algorithms are so fundamental, it is important that their use be as simple and error-free as possible.

[5] Virtual function tables, EH state tables, etc.

The template containers in the Standard C++ Library are based on principles of generic programming, rather than the inheritance approach used in other languages such as Smalltalk. An early set of foundation classes for C++, called the National Institutes of Health Class Library (NIHCL), was based on a class hierarchy after the Smalltalk tradition.

Of course, this was before templates had been added to the C++ language; but it is useful in illustrating how inheritance compares to templates in the implementation of programming idioms such as containers.

In the NIH library, all classes in the tree inherited from a root class `Object`, which defined interfaces for identifying the real class of an object, comparing objects, and printing objects[6]. Most of the functions were declared virtual, and had to be overridden by derived classes[7]. The hierarchy also included a class `Class` that provided a library implementation of RTTI (which was also not yet part of C++). The `Collection` classes, themselves derived from `Object`, could hold only other objects derived from `Object` which implemented the necessary virtual functions.

[6] The `Object` class itself inherited from `class NIHCL`, which encapsulated some static data members used by all classes.

[7] Presumably, had the NIHCL been written today, these would have been pure virtual functions.

But the NIHCL had several disadvantages stemming from its use of inheritance for implementing container classes. The following is a portion of the NIHCL hierarchy (taken from the README file for the library):

```
NIHCL - Library Static Member Variables and Functions
    Object - Root of the NIH Class Library Inheritance Tree
        Bitset - Set of Small Integers (like Pascal's type SET)
        Class - Class Descriptor
        Collection - Abstract Class for Collections
            Arraychar - Byte Array
            ArrayOb - Array of Object Pointers
            Bag - Unordered Collection of Objects
            SeqCltn - Abstract Class for Ordered, Indexed
                      Collections
                Heap - Min-Max Heap of Object Pointers
                LinkedList - Singly-Linked List
                OrderedCltn - Ordered Collection of Object Pointers
                    SortedCltn - Sorted Collection
                        KeySortCltn - Keyed Sorted Collection
                Stack - Stack of Object Pointers
            Set - Unordered Collection of Non-Duplicate Objects
                Dictionary - Set of Associations
                    IdentDict - Dictionary Keyed by Object Address
                IdentSet - Set Keyed by Object Address
        Float - Floating Point Number
        Fraction - Rational Arithmetic
        Integer - Integer Number Object
        Iterator - Collection Iterator
        Link - Abstract Class for LinkedList Links
            LinkOb - Link Containing Object Pointer
        LookupKey - Abstract Class for Dictionary Associations
            Assoc - Association of Object Pointers
            AssocInt - Association of Object Pointer with Integer
        Nil - The Nil Object
        Vector - Abstract Class for Vectors
            BitVec - Bit Vector
            ByteVec - Byte Vector
            ShortVec - Short Integer Vector
            IntVec - Integer Vector
            LongVec - Long Integer Vector
            FloatVec - Floating Point Vector
            DoubleVec - Double-Precision Floating Point Vector
```

Thus the `class KeySortCltn` (roughly equivalent to `std::map`), is seven layers deep in the hierarchy:

```
NIHCL
    Object
       Collection
           SeqCltn
               OrderedCltn
                    SortedCltn
                        KeySortCltn
```

Because a linker cannot know which virtual functions will be called at run-time, it typically includes the functions from all the preceding levels of the hierarchy for each class in the executable program. This can lead to code bloat without templates.

There are other performance disadvantages to inheritance based collection classes:

- Primitive types cannot be inserted into the collections. Instead, these must be replaced with classes in the `Object` hierarchy, which are programmed to have similar behaviour to primitive arithmetic types, such as `Integer` and `Float`. This circumvents processor optimisations for arithmetic operations on primitive types. It is also difficult to exactly duplicate the behaviour of arithmetic data types through class member functions and operators.
- Because C++ has compile-time type checking, providing type-safe containers for different contained data types requires code to be duplicated for each type. Type safety is the same reason that template containers are instantiated multiple times. To avoid this duplication of code, the NIHCL collections hold pointers to a generic type – the base `Object` class. However, this is not type safe, and requires run-time checks to ensure objects are type compatible with the contents of the collections. It also leads to many more dynamic memory allocations, which can hinder performance. Furthermore, type checking is always dynamic, adding further cost to the program using the collections.
- Because classes used with the NIHCL must inherit from `Object` and are required to implement a number of virtual functions, this solution is intrusive on the design of classes from the problem domain. The C++ Standard Library containers do not impose such requirements on their contents[8]. For this reason alone, the obligation to inherit from `class Object` often means that the use of Multiple Inheritance also becomes necessary, since domain specific classes may have their own hierarchical organization.
- The C++ Standard Library lays out a set of principles for combining data structures and algorithms from different sources. Inheritance-based libraries from different vendors, where the algorithms are implemented as member functions of the containers, can be difficult to integrate and difficult to extend.

2.6 *The Standard* IOStreams *Library – Overview*

The Standard *IOStreams* library (§IS-27) has a well-earned reputation of being inefficient! Most of this reputation is, however, due to misinformation and naïve implementation of this library component. Rather than tackling the whole library, this report addresses efficiency considerations related to a particular aspect used throughout the *IOStreams* library, namely those aspects relating to use of *Locales* (§IS-22). An implementation approach for removing most, if not all, efficiency problems related to locales is discussed in 3.2.

The efficiency problems come in several forms:

---

[8] A class used in a Standard container must be Assignable and CopyConstructible (23.1p3); often it additionally needs to be EqualityComparable and LessThanComparable (implement `operator ==` and `operator <, see 20.1)`. A default constructor for the class is required only if certain container member functions are called. These are the only requirements placed upon such a class by the C++ Standard.

2.6.1 Executable Size

Typically, using anything from the *IOStreams* library drags in a huge amount of library code, most of which is not actually used. The principle reason for this is the use of `std::locale` in all base classes of the *IOStreams* library (e.g. `std::ios_base` and `std::basic_streambuf`). In the worst case, the code for all required facets from the *Locales* library (§IS-22.1.1.1.1¶4) is included in the executable. A milder form of this problem merely includes code of unused functions from a facet, from which one or more functions are used. This is discussed in 3.2.2.

2.6.2 Execution Speed

Since certain aspects of *IOStream* processing are distributed over multiple facets, it appears that the standard mandates an inefficient implementation. This is not the case, and by using some form of pre-processing much of the work can be avoided. In addition, with a slightly smarter linker than is typically used, it is possible to remove additional inefficiencies. This is discussed in 3.2.3 and 3.2.5.

2.6.3 Object Size

The standard seems to mandate an `std::locale` object being embedded in each `std::ios_base` and `std::basic_streambuf` object, in addition to several options used for formatting and error reporting. This makes for fairly large `stream` objects. Using a more advanced organization for `stream` objects can shift the costs to those applications actually using the corresponding features. Depending on the exact approach taken, the costs are shifted to one or more of:

- compilation time
- higher memory usage when actually using the corresponding features
- execution speed

This is discussed in 3.2.6.

2.6.4 Compilation Time

A widespread approach to cope with the lack of support for the separation model is to include the template implementation in the headers. This can result in very long compile and link times if, for example, the *IOStreams* headers are included, and especially if optimisations are enabled. With an improved approach using pre-instantiation and consequent decoupling techniques, the compilation time can be reduced significantly. This is discussed in 3.2.4.

3 Performance – Techniques & Strategies

3.1 Programmer Directed Optimisations

There are many factors that influence the performance of a computer program. At one end of the scale is the high-level design and architecture of the overall system, at the other is the raw speed of the hardware and operating system software on which the program runs. Assuming that the applications programmer has no control over these factors in the system, what can be done at the level of writing code to achieve better performance?

Compilers typically use a heuristic process in optimising code that may be different for small and large programs. Therefore, it is difficult to recommend any techniques that are guaranteed to improve performance in all environments. It is vitally important to measure a performance-critical application in the target environment and concentrate on improving performance where bottlenecks are discovered. Because so many factors are involved, measuring actual performance can be difficult but remains an essential part of the performance tuning process.

The best way to optimise a program is to use space- and time-efficient data structures and algorithms. For example, changing a sequential search routine to a binary search will reduce the average number of comparisons required to search a sorted **`N`**-element table from about **`N`**`/2` to just `log`$_2$**`N`**; for **`N`**`=1000`, this is a reduction from 500 comparisons to 10. For **`N`**`=1,000,000`, the average number of comparisons is 20.

Another example is that `std::vector` is a more compact data structure than `std::list`. A typical `vector<int>` implementation will use about three words + one word per element, whereas a typical `list<int>` implementation will use about two words plus three words per element. That is, assuming `sizeof(int)==4`, a standard `vector` of 1,000 `int`s will occupy approximately 4,000 bytes, whereas a `list` of 1,000 `int`s will occupy approximately 12,000 bytes. Thanks to cache and pipeline effects, traversing such a `vector` will be much faster than traversing the equivalent `list`. Typically, the compactness of the `vector` will also assure that moderate amounts of insertion or erasure will be faster than for the equivalent `list`. There are good reasons for `vector` being recommended as the default standard library container.

The C++ Standard Library provides several different kinds of containers, and guarantees how they compare at performing common tasks. For example, inserting an element at the end of an `std::vector` takes constant time[9], but inserting one at the beginning or in the middle takes linear time increasing with the number of elements that have to be moved to make space for the new element. With an `std::list` on the

[9] It takes constant time unless the addition forces the vector to reallocate storage, but reallocation can be prevented by pre-allocating space with `reserve()`.

other hand, insertion of an element takes constant time at any point in the collection, but that constant time is somewhat slower than adding one to the end of a vector. Finding the N^{th} element in an `std::vector` involves a simple constant-time arithmetic operation on a random-access iterator accessing contiguous storage, whereas an `std::list` would have to be traversed one element at a time, so access time grows linearly with the number of elements. A typical implementation of `std::map` maintains the elements in sorted order in a "red-black" tree structure, so access to any element takes logarithmic time. Though not a part of the C++ Standard Library, `hash_maps` are capable of faster lookups than an `std::map`, but are dependent on a well-chosen hash function and bucket size. Poor choices can degrade performance significantly.

Always measure before attempting to optimise. It is very common for even experienced programmers to guess wrong about performance implications of choosing one kind of container over another. Often performance depends critically on the machine architecture and the quality of optimiser used.

The C++ Standard Library also provides a large number of algorithms with documented complexity guarantees. These are functions that apply operations to a sequence of elements. Achieving good performance, as well as correctness, is a major design factor in these algorithms. These can be used with the standard containers, or with native arrays, or with newly written containers, provided they conform to the standard interfaces.

If profiling reveals a bottleneck, small local code optimisations may be effective. But it is very important always to measure first. Transforming code to reduce run time or space consumption can often decrease program readability, maintainability, modularity, portability, and robustness as well. Such optimisations often sacrifice important abstractions in favour of improving performance, but while the performance cost may be reduced, the cost to program structure and maintainability needs to be factored into the decision to rewrite code to achieve other optimisation goals.

An old rule of thumb is that there is a trade-off between program size and execution speed – that techniques such as declaring code `inline` can make the program larger but faster. However, modern processors make extensive use of on-board cache and instruction pipelines, so the smallest code is often the fastest as well. Compilers are free to ignore inline directives and to make their own decisions about which functions to inline, but adding the hint is often useful as a portable performance enhancement. With small one- or two-line functions, where the implementation code generates less code than a function preamble, the resulting code may well be both smaller and faster.

Programmers are sometimes surprised when their programs call functions they haven't specified, maybe even haven't written. Just as a single innocuous-looking line of C code may be a macro that expands to dozens of lines of code, possibly involving system calls which trap to the kernel with resulting performance implications, a single line of C++ code may also result in a sequence of function calls which is not obvious without knowledge of the full program. For example:

```
X v1;         // looks innocent
X v2 = 7;     // obviusly initialized
```

However, the declaration of `v1` implicitly invokes the `class X`'s default constructor to initialise the object `v1`. Depending on the class design, proper initialisation may involve memory allocations or system calls to acquire resources[10]. It is important to remember, however, that in C the object would still need to be initialised, and that code would have to be explicitly called by the programmer. Resources would also have to be explicitly released at the appropriate time. The initialisation and release code is more visible to the C programmer, but possibly less robust because the language does not support it automatically.

Understanding what a C++ program is doing is important for optimisation. If you know what functions C++ silently writes and calls, careful programming can keep the unexpected code to a minimum. Some of the works cited in the bibliography (Appendix A:) provide more extensive guidance, but the following provides some suggestions for writing more efficient code:

- Shift expensive computations from the most time-critical parts of a program to the least time-critical parts (often, but not always, program start-up). Other techniques include lazy evaluation and caching of pre-computed values. Of course, these strategies apply to programming in any language, not just C++.
- In constructors, prefer initialisation of data members to assignment. If a member has a default constructor, that constructor will be called to initialise the member before any assignment takes place. Therefore, an assignment to a member within the constructor body can mean that member is initialised as well as assigned to, typically doubling the amount of work done.
- As a general principle, don't define a variable before you are ready to initialise it. Defining it early results in a constructor call (initialisation) followed by an assignment of the value needed, as opposed to simply constructing it with the value needed.
- Understand how and when the compiler generates temporary objects. Often small changes in coding style can prevent the creation of temporaries, with consequent benefits for run-time speed and memory footprint. Temporary objects may be generated when initialising objects, passing parameters to functions, or returning values from functions.

[10] This is a common idiom in C++, because the release of the resources can be triggered automatically when the object's lifetime ends (§IS-3.7).

- Passing arguments to a function by value [e.g. `void f(T x)`] is cheap for built-in types, but potentially expensive for class types since they may have a non-trivial copy constructor. Passing by address [e.g. `void f(T const* x)`] is light-weight, but changes the way the function is called. Passing by reference-to-const [e.g. `void f(T const& x)`] combines the safety of passing by value with the efficiency of passing by address.

- Calling a function with a type that differs from the function's declared argument type implies a conversion. Note that such a conversion can require work to be done at run-time For example:

```
void f1(double);
f1(7.0);     // no conversion (pass by value implies copy)
f1(7);       // conversion: f1(double(7);

void f2(const double&);
f2(7.0);     // no conversion
f2(7);       // means ``const double tmp = 7;  f(tmp);''

void f3(std::string);
string s = "MES";
f3(s);       // no conversion (pass by value implies copy)
f3("NES");   // conversion: f3(string("NES"))

void f4(const std::string&);
f4(s);       // no conversion (pass by reference, no copy)
f4("AS");    // means: ``const string tmp = "AS"; f4(tmp);''
```

  If a function is called several times with the same value, it can be worthwhile to put the value in a variable of the appropriate type (such as `s` in the example above) and pass that. That way, the conversion will be done once only.

- Unless you need automatic type conversions, declare all one-argument constructors[11] `explicit`. This will prevent them from being called accidentally. Conversions can still be done when necessary by explicitly stating them in the code, thus avoiding the penalty of hidden and unexpected conversions.

- Rewriting expressions can reduce or eliminate the need for temporary objects. For example, if `a`, `b`, and `c` are objects of `class Matrix`:

```
Matrix a;     // inefficient: don't create an object before
                    // it is really needed, default initialization
                    // can be expensive
a = b + c;    // inefficient: (b + c) creates a temporary

Matrix a = b; // better: no default initialization
a += c;       // better: no temporary objects created
```

  Better yet, use a library that eliminates need for the rewrite using +=. Such libraries, which are common in the numeric C++ community, usually use

[11] This refers to any constructor that may be called with a single argument. Multiple parameter constructors with default arguments can be called as one-argument constructors.

function objects and expression templates to yield uncompromisingly fast code from conventional-looking source.

- Use the return value optimisation to give the compiler a hint that temporary objects can be eliminated. The trick is to return constructor arguments instead of objects, like this:

```
const Rational operator * ( Rational const & lhs,
                            Rational const & rhs )
{
    return Rational( lhs.numerator() * rhs.numerator(),
                 lhs.denominator() * rhs.denominator() );
}
```

  Less carefully written code might create a local `Rational` variable to hold the result of the calculation, use the assignment operator to copy it to a temporary variable holding the return value, then copy that into a variable in the calling function:

```
// not this way:
const Rational operator * ( Rational const & lhs,
                            Rational const & rhs )
{
    Rational tmp; // calls default constructor, if there is one
    tmp.my_numerator = lhs.numerator() * rhs.numerator();
    tmp.my_denominator = lhs.denominator() * rhs.denominator();

    return tmp;
}
```

  But with the suggested hints, the compiler is able to construct the return value directly into the variable that is specified to receive it.

- Prefer pre-increment and -decrement to postfix operators.

  Postfix operators like `i++` copy the existing value to a temporary object, increment the internal value, and then return the temporary. Prefix operators like `++i` increment the actual value first and return a reference to it. With objects such as `iterator`s, creating temporary copies is expensive compared to built-in `int`s.

```
for ( list<X>::iterator it = mylist.begin();
      it != mylist.end();
      ++it )     // NOTE: rather than   it++
{
    // ...
}
```

- Dynamic memory allocation and de-allocation can be a bottleneck. Consider writing class-specific `operator new()` and `operator delete()` functions, optimised for objects of a specific size or type. It may be possible to recycle blocks of memory instead of releasing them back to the heap whenever an object is deleted.

- Sometimes it is helpful to "widen" a class's interface with functions that take different data types to prevent automatic conversions (such as adding an overload on `char *` to a function which takes an `std::string` parameter).

The numerous overloads for operators `+`, `==`, `!=`, and `<` in the `<string>` header are an example of such a "fat" interface[12]. If the only supported parameters were `std::string`s, then characters and pointers to character arrays would have to be converted to full `std::string` objects before the operator was applied.

- The Standard `string` class is not a lightweight component. Because it has a lot of functionality, it comes with a certain amount of overhead (and because Standard Library container classes throw C++ `std::string`s, and not C-style string literals, this overhead may be included in a program inadvertently).

  In many applications, strings are created, stored, and referenced, but never changed. As an extension, or as an optimisation, it might be useful to create a lighter-weight unchangeable-string class.

- Reference counting is a widely used optimisation technique. In a single-threaded application, it can prevent making unnecessary copies of objects. However, in multi-threaded applications, the overhead of locking the shared data representation may add unnecessary overheads, negating the performance advantage of reference counting[13].

- Pre-compute values that won't change. To avoid repeated function calls inside a loop, rather than writing:

  ```
  while( myListIterator != myList.end() ) ...

  for( size_t n = 0; n < myVector.size(), ++n ) ...
  ```

  instead call `myList.end()` or `myVector.size()` once before the loop, storing the result in a variable which can then be used in the comparison., for example:

  ```
  std::list<myT> myEnd = myList.end();
  while( myListIterator != myend ) ...
  ```

  On the other hand, note that if a function such as `myList.end()` is so simple that it can be inlined, the rewrite may not yield any performance advantage over what a good compiler produces for the original code.

- Object-oriented programming often leads to a number of small functions per class, often with trivial implementation. For example:

  ```
  class X
  {
  private:
      int     value_;
      double* array_;  // pointer to array of [size_] doubles
      size_t  size_;
  public:
  ```

[12] It is also worth noting, that even if a conversion is needed, it is sometimes better to have the conversion performed in one place, where an overloaded "wrapper" function calls the one that really performs the work. This can help to reduce program size, where each caller would otherwise perform the conversion.

[13] Of course, if space is the resource being optimised, reference counting may still be the best choice.

```
    int value()   { return value_; }
    size_t size() { return size_;  }
    // ...
};
```

Small forwarding functions can usually be inlined to advantage, especially if they occupy less code space than preparing the stack frame for a function call. As a rule of thumb, functions consisting of only one or two lines are generally good candidates for inlining.

- When processors read-ahead to maintain a pipeline of instructions, too many function calls can slow down performance because of branching or cache misses. Optimisers work best when they have stretches of sequential code to analyse, because it gives them more opportunity for register allocation optimization, code-movement, and common sub-expression elimination. This is why inline functions can help performance, as inlining exposes more sequential code to the optimiser. Techniques such as avoiding conditional code and unrolling short loops also help the optimiser do a better job.

- The use of dynamic binding and virtual functions has some overhead in both memory footprint and run-time performance. This overhead is minor, especially when compared against alternative ways of achieving run-time polymorphism (see 2.3). A bigger factor is that virtual functions may interfere with compiler optimisations and inlining.

  Note that virtual functions should be used only when run-time polymorphic behavior is desired. Not every function needs to be virtual and not every class should be designed to be a base class.

- Many programs written in some conventional (also called "old fashioned") object-oriented styles are very slow to compile, because the compiler must examine hundreds of header files and tens of thousands of lines of code. In principle, this can *[Bjarne, was this intended to say 'can not'?]* be avoided. However, code can be structured to minimize re-compilation after changes. This typically produces better, more maintainable, designs because they exhibit better separation of concerns.

  Consider a classical example of an object-oriented program:

```
class Shape {
public:             // interface to users of Shapes
       virtual void draw() const;
       virtual void rotate(int degrees);
       // ...
protected:   // common data (for implementers of Shapes)
       Point center;
       Color col;
       // ...
};

class Circle : public Shape {
public:
       void draw() const;
       void rotate(int) { }
       // ...
```

```
protected:
        int radius;
        // ...
};

class Triangle : public Shape {
public:
        void draw() const;
        void rotate(int);
        // ...
protected:
        Point a, b, c;
        // ...
};
```

The idea is that users manipulate shapes through Shape's public interface, and that implementers of derived classes (such as Circle and Triangle) share aspects of the implementation represented by the protected members.

It is not easy to define shared aspects of the implementation that are helpful to all derived classes. For that reason, the set of protected members is likely to need changes far more often than the public interface. For example, even though "center" is arguably a valid concept for all Shapes, it is a nuisance to have to maintain a point "center" for a Triangle - for triangles, it makes more sense to calculate the center if and only if someone expresses interest in it.

The protected members are likely to depend on "implementation" details that the users of Shapes would rather not have to depend on. For example, much code using a Shape will be logically independent of the definition of "Color", yet the presence of Color in the definition of Shape will probably require compilation of header files defining the operating system's notion of color.

When something in the protected part changes, users of Shape have to recompile – even though only implementers of derived classes have access to the protected members. Thus, the presence of "information helpful to implementers" in the base class that also acts as the interface to users is the source of several problems:

- instability in the implementation,
- spurious recompilation of user code (when implementation information changes), and
- excess inclusion of header files into user code (because the "information helpful to implementers" needs those headers).

This is sometimes known as the "brittle base class problem."

The obvious solution is to omit the "information helpful to implementers" for classes that are used as interfaces to users. In other words, interface classes should represent “pure” interfaces and therefore take the form of abstract classes:

```
class Shape {
```

```
public:         // interface to users of Shapes
        virtual void draw() const = 0;
        virtual void rotate(int degrees) = 0;
        virtual Point center() const = 0;
        // ...

        // no data
};

class Circle : public Shape {
public:
        void draw() const;
        void rotate(int) { }
        Point center() const { return center; }
        // ...
protected:
        Point cent;
        Color col;
        int radius;
        // ...
};

class Triangle : public Shape {
public:
        void draw() const;
        void rotate(int);
        Point center() const;
        // ...
protected:
        Color col;
        Point a, b, c;
        // ...
};
```

The users are now insulated from changes to implementations of derived classes. This technique has been known to decrease build times by orders of magnitude.

But what if there really is some information that is common to all derived classes (or even to several derived classes)? Simply make that information a class and derive the implementation classes from that also:

```
class Shape {
public:         // interface to users of Shapes
        virtual void draw() const = 0;
        virtual void rotate(int degrees) = 0;
        virtual Point center() const = 0;
        // ...

        // no data
};

struct Common {
        Color col;
        // ...
};
```

```
class Circle : public Shape, protected Common {
public:
        void draw() const;
        void rotate(int) { }
        Point center() const { return center; }
        // ...
protected:
        Point cent;
        int radius;
};

class Triangle : public Shape, protected Common {
public:
        void draw() const;
        void rotate(int);
        Point center() const;
        // ...
protected:
        Point a, b, c;
};
```

- Another technique for ensuring better separation between parts of a program involves an interface object holding a single pointer to an implementation object. This is often called "the PIMPL" (Pointer to IMPLementation) idiom. For example:

```
asdsadsad
```

more text on the way here

- Use function objects with the Standard Library algorithms rather than function pointers. Function pointers defeat the data flow-analysers of many optimisers, but function objects are passed by value and optimisers can easily handle inline functions on objects.

- Calling a function with a default argument requires the constructor to create a temporary object for the default. If the construction of that temporary is expensive and if the function is called several times, it can be worth while to construct the default argument value somewhere and use that value in each call. For example:

```
class C {
public:
   C(int i) { /* … */}
   int mf() const;
   // …
};

int f( const C & x = C(0)) // construct C(0) in each call f()
{
       return x.mf();
}

int g()
{
   static const C x(0); // construct x in first call
   return x.mf();
}
```

```
const C c0(0);   // construct c0 for use in calls of h()
int h(const C& x = c0)
{
    return x.mf();
}
```

- When programming "close to the metal”, such as when dealing with low-level hardware devices, some use of assembly code may be unavoidable. C++'s `asm` keyword enables the use of assembler to be kept to the necessary minimum. The syntax of an *asm declaration* (§IS-7.4) is:

```
asm-definition:
        asm ( string-literal ) ;
```

Note (lois):	*There seems to be so much difference of opinion over the syntax of the specific example that I have taken it out.*

The assembler statement is typically enclosed in quotation marks. The meaning of an *asm declaration* is implementation defined. Typically, it is used to insert assembly language directly into the translator output. Needless to say, it is highly non-portable.

The advantage of using short assembler functions can be lost if they have to be placed in separate source files where the efficiency gained is over-shadowed by the overhead of calling and returning a function, plus attendant effects on the instruction pipeline and register management. The `asm` keyword can insert just a few assembler statements inline where they provide the most benefit.

Typically, a compiler is not aware of the semantics of inlined assembly instructions. Thus, use of an `asm` can defeat other important optimisations such as common sub-expression elimination and global en-registering. Consequently, assembler inserts should typically be used only for operations that cannot be expressed in C++.

- Whenever possible, compute values and catch errors at translation time rather than run-time. With sophisticated use of templates, a complicated block of code can be compiled to a single constant in the executable, therefore having zero run-time overheads. This might be described as a code implosion (the opposite of a code explosion). Here is a simple example:

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

```
class Factorial<1> {
public:
    enum { value = 1 };
};
```

Using this class[14], the value **N!** is accessible at compile-time as `Factorial<N>::value`.

For another example, the following template class can be used to generate a compile-time constant:

// Compile-time Square Root Computation: `ceil(sqrt(N))`:

```
// <root.h>:
template <int Size, int Low = 1, int High = Size>
    struct Root;

template <int Size, int Mid>
    struct Root<Size,Mid,Mid> {
        static const int root = Mid;
    };

template <int Size, int Low, int High>
    struct Root {
        static const int mean  = (Low + High)/2;
        static const bool down = (mean * mean >= Size);
        static const int root  = Root<Size,
                   (down ? Low  : mean + 1),
                   (down ? mean : High)>::root;
    };

// User code:
//   compute sqrt(N), use it for static table size
int table[Root<N>::root];
```

Template meta-programming and expression templates are not techniques for novice programmers, but an advanced practitioner can use them to good effect.

14 Within limitations, remember that if an `enum` is 32-bits, the maximum `N` can be is just 12.

- Templates provide compile-time polymorphism, wherein type selection does not incur any run-time penalty. If appropriate to the design, consider using templates as interfaces instead of abstract base classes[15]. Templates have several useful properties: they impose no space or code overhead on the class used as a template argument, and they can be attached to the class for limited times and purposes. If the class does not provide the needed functionality, it can be defined externally through template specialization. If certain functions in the template interface are never used for a given class, they need not be defined because they will not be instantiated. In the example below, the `talk_in_German()` function in the "interface" is only defined for `class CuckooClock`, because that is the only object for which it is needed. Invoking `talk_in_German()` on an object of a different type results in a compiler diagnostic:

```
#include <iostream>
using std::cout;
using std::endl;

// some domain objects
class Dog {
public:
    void talk() { cout << "woof woof" << endl; }
};

class CuckooClock {
public:
    void talk()           { cout << "cuckoo cuckoo" << endl; }
    void talk_in_German() { cout << "wachet auf!"   << endl; }
};

class BigBenClock {
public:
    void talk()      { cout << "take a tea-break"    << endl; }
    void playBongs() { cout << "bing bong bing bong" << endl; }
};

class SilentClock {
    // doesn't talk
};

// generic template to provide non-inheritance-based
// polymorphism
template <class T>
class Talkative {
    T& t;
public:
    Talkative( T& obj ) : t( obj ) {  }
    void talk()                    { t.talk(); }
    void talk_in_German()          { t.talk_in_German(); }
};
```

[15] On the other hand, if run-time as opposed to compile-time polymorphism is needed, then using virtual functions is more appropriate.

```
// specialization to adapt functionality
template <>
class Talkative<BigBenClock> {
    BigBenClock& t;
public:
    Talkative( BigBenClock& obj )
    : t( obj )  {  }
    void talk() { t.playBongs(); }
};

// specialization to add missing functionality
template <>
class Talkative<SilentClock> {
    SilentClock& t;
public:
    Talkative( SilentClock& obj )
    : t( obj )  {  }
    void talk() { cout << "tick tock" << endl; }
};

// adapter function to simplify syntax in usage
template <class T>
Talkative<T>  makeTalkative( T& obj ) {
    return Talkative<T>( obj );
}

// function to use an object which implements the
// Talkative template-interface
template <class T>
void makeItTalk( Talkative<T> t )
{
    t.talk();
}

int main()
{
    Dog         aDog;
    CuckooClock aCuckooClock;
    BigBenClock aBigBenClock;
    SilentClock aSilentClock;

    Talkative<Dog> td( aDog );
    td.talk();                                      // woof woof

    Talkative<CuckooClock> tcc( aCuckooClock );
    tcc.talk();                                 // cuckoo cuckoo

    makeTalkative( aDog ).talk();                   // woof woof
    makeTalkative( aCuckooClock ).talk_in_German();  // wachet
                                                     //   auf!

    makeItTalk( makeTalkative( aBigBenClock ) );  // bing bong
                                                  // bing bong
    makeItTalk( makeTalkative( aSilentClock ) );  // tick tock

    return 0;
}
```

- Controlling the instantiation of *class-template*s and *function-template*s can help to reduce the footprint of a program. Some compilers instantiate a template only once into a separate "repository"; others instantiate every template into every translation unit. In the latter case, the linker typically eliminates duplicates. If it does not, the executable can suffer significant memory overheads.

- Explicit instantiation of a *class-template* specialisation causes instantiation of all of its members into the translation unit containing the explicit instantiation directive. In addition to a whole *class-template*, explicit instantiation can also be used for a member function, member class, or static data member of a class template, or a function template or member template specialisation.

  For example (see IS-14.7.2¶2):

```
template<class T> class Array { void mf(); };
template class Array<char>;
template void Array<int>::mf();

template<class T> void sort(Array<T>& v) { /* ... */ }
template void sort(Array<char>&);  //  argument is deduced here

namespace N {
template<class T> void f(T&) { }
}
template void N::f<int>(int&);
```

  Instantiating template code into a library once explicitly can save space in every translation unit which links to it. For example, some library vendors provide instantiations of `basic_string<char>` and `basic_string<wchar_t>` in their run-time libraries. Some compilers also have command-line options to force complete template instantiation or to suppress it as needed.

- In addition to these portable coding techniques, programming tools offer additional platform-specific help for optimising programs. Some of the techniques available include the following:

  - Compiler options are usually extra arguments or switches, which pass instructions to the compiler. Some of these instructions are related to performance, and control how to:

    - generate executable code optimised for a particular hardware architecture.
    - optimise the translated code for size or speed. Often there are sub-options to exercise finer control of optimisation techniques or how aggressively they should be applied.
    - suppress the generation of debugging information, which adds to code and data size.
    - instrument the output code for run time profiling, as an aid to measuring performance and to refine the optimisation strategies used in subsequent builds.
    - disable exception handling overhead, in code which does not use exceptions at all.
    - control the instantiation of templates.

- `#pragma` directives allow compilers to add features specific to machines and operating systems, within the framework of Standard C++. Some of the optimisation-related uses of `#pragma` directives are to:
    - specify function calling conventions (a C++ linkage-specification can also be used for this purpose).
    - influence the inline expansion of code.
    - specify optimisation strategies on a function-by-function basis.
    - control the placement of code or data into memory areas (to achieve better locality of reference at run time).
    - affect the layout of class members (through alignment or packing constraints, or by suppressing compiler-generated data members).

  Note that `#pragma`s are not standardized and not portable.
- Linking to static libraries or shared libraries, as appropriate. Linker options can also be used to control the amount of extra information included in a program (e.g., symbol tables, debugging formats).
- Utilities for efficiently allocating small blocks of memory. These may take the form of system calls, pragmas, compiler options, or libraries.
- Additional programs:
    - Many systems have a utility program[16] to remove the symbol table and line number information from an object file, once debugging is complete (This can also be done at link time by the linker). The purpose is to reduce file storage and in some cases, memory overhead.
    - Some systems have utilities[17] and tools to interpret profiling data and identify run time bottlenecks.

- Sometimes, minimising compile-time is important. When code is being created and debugged, suppressing optimisation may enable the compiler to run faster.

  The most effective technique for reducing compile time relies on reducing the amount of code to be compiled. The key is to reduce coupling between different parts of a program so that header files are small and so that few header files are needed in most translation units. Techniques for accomplishing that are the use of abstract classes as interfaces and the PIMPL idiom.

---

[16] For instance the ‘strip’ utility is part of the Software Development Utilities option in the IEEE Posix/Open Group Unix specifications.

[17] For instance the ‘prof’ utility is not part of the Posix/Unix standard, but is available on many systems nonetheless.

As discussed above, suppressing automatic template instantiation in a given translation unit may reduce compile-time.

- Reading and parsing header code takes time. Years ago, the common practice was to #include as few headers as possible, so that only necessary symbols were declared. But with technology to pre-compile headers, build time may be reduced by using a single header in each translation unit which #includes everything needed for the program. Furthermore, most compilers now implement the following "idempotent guard" optimization. Well-designed headers will usually protect their contents against multiple inclusion by following this pattern:

```
#if !defined THIS_HEADER_H
#define THIS_HEADER_H
  // here are the contents of the header
#endif /* THIS_HEADER_H */
```

  If the compiler provides the "idempotent guard" optimization, it will record in an internal table the fact that this header has an idempotent guard. If this header is subsequently #included again, and the macro THIS_HEADER_H still remains defined, then the compiler can avoid accessing the header contents.

  For more details about the idempotent guard optimization, a sample test program, and ongoing test results, see Bob Archer's discussion at http://www.hottub.demon.co.uk/software/include/index.htm.

  If the compiler does not perform this optimization, the check can be implemented by the programmer:

```
#if !defined MY_HEADER_H
#include "my_header.h"
#endif
```

  This has the disadvantage of coupling the header's guard macro to the source files which #include that header.

  As always, local measurements in specific circumstances should govern the decision.

3.2 Optimising Libraries: Reference Example: "An Efficient Implementation of Locales *and* IOStreams*"*

The definition of *Locales* in the C++ Standard (§IS-22) seems to imply a pretty inefficient implementation. This is however not true. It is possible to create efficient implementations of the *Locales* library, both in terms of run-time efficiency and executable size. This does take some thought and this reports discusses some of the possibilities that can be used to improve the efficiency of `std::locale` implementations with a special focus on the functionality as used by the *IOStreams* library.

The approaches discussed in this report are primarily applicable to statically bound executables as are typically found in for example, embedded systems. If shared, or dynamically loaded libraries are used, different optimisation goals have precedence, and some of the approaches described here could be counterproductive. Clever organization of the shared libraries might deal with some efficiency problems too - however, this is not discussed in this report.

Nothing described in this report involves magic or really new techniques. It just discusses how well known techniques may be employed to the benefit of the library user. It does however involve additional work compared to a trivial implementation, for the library implementer as well as for the library tester, and in some cases for the compiler implementer. Some of the techniques focus on just one efficiency aspect and thus not all techniques will be applicable in all situations (e.g. certain performance improvements can result in "additional" code). Depending on the requirements, the library writer, or possibly even the library user, has to choose which optimisations are the most appropriate.

3.2.1 *Locale* Implementation Basics

Before going into the details of the various optimisations, it is worth introducing the implementation of locales, describing features implicit to the Standard definition. Although some of the material presented in this section is not strictly required and there are other implementation alternatives, this section should provide the necessary details to understand where the optimisations should be directed.

An `std::locale` object is an immutable collection of immutable objects – or more precisely, of immutable facets. This immutability trait is important in multi-threaded environments, because it removes the need to synchronize most accesses to locales and their facets. The only operations needing multi-threading synchronization are copying, assigning, and destroying `std::locale` objects and the creation of modified locales.

Instead of modifying a locale object to augment the object with a new facet or to replace an existing one, `std::locale` constructors or member functions are used, creating new locale objects with the modifications applied. As a consequence, multiple locale objects can share their internal representation and multiple internal representations can (in fact, have to) share their facets. When a modified locale object is created, the existing facets are copied from the original and then the modification is applied, possibly replacing some facets. For correct maintenance of the facets, the Standard mandates the necessary interfaces, allowing reference counting or some more or less equivalent techniques for sharing facets. The corresponding functionality is implemented in the class `std::locale::facet`, the base class for all facets.

Copying, assigning, and destroying `std::locale` objects reduces to simple pointer and reference count operations. When copying a locale object, the reference count is incremented and the pointer to the internal representation is assigned. When destroying a locale object, the reference count is decremented and when it drops to 0, the internal representation is released. Assignment is an appropriate combination of these two. What remains is the default construction of an `std::locale` which is just the same as a copy of the current global locale object. Thus, the basic lifetime operations of `std::locale` objects are reasonably fast.

Individual facets are identified using an ID, more precisely an object of type `std::locale::id` which is available as a static data member in all base classes defining a facet. A facet is a class derived from `std::locale::facet` which has a publicly accessible static member called `id` of type `std::locale::id` (§IS-22.1.1.1.2¶1). Although explicit use of a locale's facets seems to use a type **F** as an index, the *Locales* library internally uses **F**`::id`. The `std::locale::id` simply stores an index into an array identifying the location of a pointer to the corresponding facet or 0 if a locale object does not store the corresponding facet.

Taken together, a locale object is basically a reference counted pointer to an internal representation consisting of an array of pointers to reference counted facets. In a multi threaded environment, the internal representation, and the facets might store a mutex (or some similar synchronization facility) thus protecting the reference count. A corresponding excerpt of the declarations might look something like this (with `namespace std` and other qualifications or elaborations of names omitted):

```
class locale {
public:
    class facet {
    // ...
    private:
        size_t  refs;
        mutex   lock;     // optional
    };

    class id {
    // ...
    private:
        size_t  index;
    };

    // ...
private:
    struct internal {
    // ...
        size_t refs;
        mutex  lock;     // optional
        facet* members;
    };
    internal*  rep;
};
```

These declarations are not really required and there are some interesting variations:

- Rather than using a double indirection with an internal `struct`, a pointer to an array of unions can be used. The union would contain members suitable as reference count and possible mutex lock, as well as pointers to facets. The index 0 could, for example, be used as "reference count" and index 1 as "mutex", with the remaining array members being pointer to facets.
- Instead of protecting each facet object with its own mutex lock, it possible to share the locks between multiple objects. For example, there may be just one

global mutex lock, because the need to lock facets is actually relatively rare (only when a modified locale object is necessary is there a need for the mutex) and it is unlikely that this global lock remains held. If this is too coarse grained, it is possible to place a mutex lock into the static `id` object, such that an individual mutex lock exists for each facet type.

- If atomic increment and decrement/check are available, the reference count is sufficient, because the only operations needing multi-threading protection are incrementing and decrementing the reference count.
- The locale objects only need a representation if there are modified locale objects. If such an object is never created, it is possible to use an empty `std::locale` object. Whether or not this is the case can be determined using some form of "whole program optimisation" (§3.2.5).
- Whether an array or some other data structure is used internally does not really matter. What is important is that there is a data structure indexed by `std::locale::id`.
- A trivial implementation could use a null pointer to indicate that a facet is absent in a given locale object. If a pointer to a dummy facet is used instead, `std::use_facet()` can simply use a `dynamic_cast<>()` to produce the corresponding `std::bad_cast` exception.

In any case, it is reasonable to envision a locale object as being a reference counted pointer to some internal representation containing an array of reference counted facets. Whether this is actually implemented so as to reduce run-time by avoiding a double indirection and whether there are mutex locks and where these are does not really matter to the remainder of this discussion. It is, however, assumed that the implementer chooses an efficient implementation of the `std::locale`.

It is worth noting that the Standard definition of `std::use_facet()` and `std::has_facet()` differ from the CD2 (Committee Draft 2 – pre-IS) version quite significantly. If a facet is not found in a locale object, it is not available for this locale. In CD2, the global locale object was searched for a facet not present a given locale object. The Standard version can be more efficient - to determine whether a facet is available for a given locale object, a simple array lookup is sufficient. Basically, the functions `std::use_facet()` and `std::has_facet()` could look something like:

```
extern std::locale::facet dummy;
template <typename F>
bool has_facet(std::locale const& loc) {
    return loc.rep->facets[F::id::index] == &dummy;
}
template <typename F>
F const& use_facet(std::locale const& loc) {
    return dynamic_cast<F const&>(*loc.rep->facets[Facet::id::index]);
}
```

Editor's Note:	*Should the reference to the CD2 definition be removed, or relegated to a footnote?*

This version of the functions is tuned for speed. A simple array lookup, together with the necessary `dynamic_cast<>()` is used to obtain a facet. Since this implies that there is a slot for each facet possibly used by the program in the array, it may be somewhat wasteful with respect to memory. Other techniques might check the size of the array first or store id/facet pairs. In extreme cases, it is possible to locate the correct facet using `dynamic_cast<>()` and storing only those facets that are actually available in the given locale.

3.2.2 Reducing Executable Size

Linking unused code into an executable can have a significant impact on the executable size. Thus, it is best to avoid having unused code in the executable program. One source of unused code results from trivial implementations. The default facet `std::locale::classic()` includes a certain set of facets as described in IS-22.1.1.1.1¶2. It is tempting to implement the creation of the corresponding locale with a straightforward approach, namely explicitly registering the listed facets:

```
std::locale const& std::locale::classic() {
    static std::locale object;
    static bool uninitialized = true;

    if (uninitialized) {
        object.intern_register(new collate<char>);
        object.intern_register(new collate<wchar_t>);
        // ...
    }
    return object;
}
```

This approach however can result in a very large executable, as it drags in all facets listed in the table. The advantage of this approach is that a relatively simple implementation of the various locale operations is possible. An alternative is to include only those facets that are really used. A simple approach for doing this is to provide specialized versions of `use_facet()` and `has_facet()` which might be appropriate for `has_facet()` anyway, for example:

```
template <typename F> struct facet_aux {
    static F const& use_facet(locale const& l) {
        return dynamic_cast<F const&>(*l.rep
                                          ->facets[Facet::id::index]);
    }
    static bool has_facet(locale const& l) {
        return l.rep->facets[F::id::index] == &dummy;
    }
};
template <> struct facet_aux<ctype<char> > {
    static ctype<char> const& use_facet(locale const& l) {
        try {
            return dynamic_cast<F const&>(*l.rep
                                          ->facets[Facet::id::index]);
        } catch (bad_cast const&) {
            locale::facet* f = l.intern_register(new ctype<char>);
            return dynamic_cast<ctype<char>&>(*f);
        }
    }
    static bool has_facet(locale const&) { return true; }
};
// similarly for the other facets

template <typename F>
F const& use_facet(locale const& l) {
    return facet_aux<F>::use_facet(l);
}
template <typename F>
bool has_facet(locale const& l) {
    return facet_aux<F>::has_facet(l);
}
```

Again, this is just one example of many possible implementations for what is basically a recurring theme. A facet is created only if it is really referenced from the program. This particular approach is suitable in implementations where exceptions cause a run-time overhead only if they are indeed thrown because like the normal execution path, if the lookup of the facet is successful, it is not burdened by the extra code used to initialise the facet. Although the above code seems to imply that `struct facet_aux` has to be specialized for all required facets individually, this need not be the case. By using an additional template argument, it is possible to use partial specialization together with some tagging mechanism, to determine whether the facet should be created on the fly if it is not yet present.

Different implementations of the lazy facet initialisation include the use of static initialisers to register used facets. In this case, the specialised versions of the function `use_facet()` would be placed into individual object files together with an object whose static initialisation registers the corresponding facet. This approach implies however, that the function `use_facet()` is implemented out-of-line, possibly causing unnecessary overhead both in terms of run-time and executable size.

The next source of unused code is the combination of several related aspects in just one facet due to the use of virtual functions. Normally, instantiation of a class containing virtual functions requires that the code for all virtual functions be present, even if they are unused. This can be relatively expensive as for example, in the case of the facet dealing with numeric formatting. Even if only the integer formatting functions are used, the typically bigger code for the floating point formatting gets dragged in just to resolve the symbols referenced from the "virtual function table”.

A better approach to avoid linking of unused virtual functions involves changing the compiler such that it generates appropriate symbols, allowing the linker to determine whether a virtual function is really called. If it is, the reference from the virtual function table is resolved; otherwise, there is no need to resolve it because it will never be called anyway.

Author’s Note:	*Details for this are described elsewhere (currently, I don't have a reference I can point to but I know that Nathan Myers has dealt with this for gcc).*

For the Standard facets however, there is a “Poor Man's” alternative that comes close to having the same effect. The idea is to provide a stub implementation for the virtual functions, which is placed in the library such that it is searched fairly late. The real implementation is placed before the stub implementation in the same object file along with the implementation of the forwarding function. Since a use of the virtual function has to go through the forwarding function, this symbol is also un-referenced, and resolving it brings in the correct implementation of the virtual function.

Unfortunately, it is not totally true that the virtual function can only be called through the forwarding function. A class deriving from the facet can directly call the virtual function because these are `protected` rather than `private`. Thus, it is still necessary to drag in the whole implementation if there is a derived facet. To avoid this, another implementation can be placed in the same object file as the constructors of the facet, which can be called using a hidden constructor for the automatic instantiation. Although it is possible to get these things to work with typical linkers, a modified compiler and linker provide a much-preferred solution, which if often outside of the scope of the library implementers.

Basically, most of the normally visible code bloat can be removed using these two techniques, i.e. by including only used facets and avoiding the inclusion of unused virtual functions. Some of the approaches described in the other sections can also result in a reduction of executable size, but the focus of those optimisations is on a different aspect of the problem. Also, the reduction in code size for the other approaches is not as significant.

3.2.3 Pre-Processing for Facets

Once the executable size is reduced, the next observation is that the operations tend to be slow. Take numeric formatting as an example: to produce the formatted output of a number, three different facets are involved:

- `num_put` which does the actual formatting; i.e. determining which digits and symbols are there; doing padding when necessary; etc.
- `numpunct` which provides details about local conventions, such as the need to put in thousands separators; which character to use as a decimal point; etc.
- `ctype` which transforms the characters produced internally by `num_put`, into the appropriate "wide" characters.

Each of the `ctype` or `numpunct` functions called is basically a virtual function. A virtual function call can be an expensive way to determine whether a certain character is a decimal point; or to transform a character between a narrow and wide representation. Thus, it is necessary to avoid these calls wherever possible for maximum efficiency.

At first examination there does not appear to be much room for improvement. However, on closer inspection, it turns out that the Standard does not mandate calls to `numpunct` or `ctype` for each piece of information. If the `num_put` facet has widened a character already, or knows which decimal point to use, it is not required to call the corresponding functions. This can be taken a step further. When creating a locale object, certain data can be cached using for example, an auxiliary hidden facet. Rather than going through virtual functions over and over again, the required data is simply stored in an appropriate data structure.

For example, the cache for the numeric formatting might consist of a character translation table resulting from widening all digit and symbol characters during the initial locale set-up. This translation table might also contain the decimal point and thousands separator - combining data obtained from two different facets into just one table. Taking it another step further, the cache might be set up to use two different functions depending on whether thousands separators are used according to the `numpunct` facet or not. Some pre-processing might also improve the performance of parsing strings like the Boolean values if the `std::ios_base::boolalpha` flag is set.

Although there are many details to be handled like for example, distinguishing between normal and cache facets when creating a new locale object, the effect of using a cache can be fairly significant. It is important that the cache facets are not generally shared between locale representations. To share the cache, it has to be verified that all facets contributing to the cached data are identical in each of the corresponding locales. Also, certain things like, the use of two different functions for formatting with or without thousands separators only work if the default facet is used.

3.2.4 Compile-Time Decoupling

It may appear strange to talk about improving compile-times when discussing the efficiency of locales but there are good reasons for this. First of all, compile-time is just another concern for performance efficiency, and it should be minimized where possible. More important to this paper however, is that some of the techniques presented below, rely on certain aspects that are related to the compilation process.

The first thing that improves compile-time is the liberal use of declarations, avoiding definitions wherever possible. A Standard header may be required to include other headers that provide a needed definition (§IS-17.4.4.1¶1), however, this does not apply to declarations. As a consequence, a header need not be included just because it defines a type which is used only a as return or argument type where a declaration is sufficient. Likewise, a declaration is sufficient if only a pointer or a class is used as a member.

Looking at the members `imbue()` and `getloc()` of the class `std::ios_base`, it would seem that an object of this type is required to include `<locale>` simply for the definition of `std::locale`, because apparently, an `std::ios_base` object stores an object of this type in a member variable. This is, not required! Instead, `std::ios_base` could store the pointer to the locale's internal representation and construct an `std::locale` object on the fly. Thus, there is no need for the header `<ios>` to include the header `<locale>`. The header `<locale>` will be used elsewhere with the implementation of the `std::ios_base` class but that is a completely different issue.

Why does it matter? Current compilers lacking support for the `export` keyword require the implementation of the template members of the other stream classes in the headers anyway and the implementation of these classes will need the definitions from `<locale>` - won't they? It is true that some definitions of the template members will indeed require definitions from the header `<locale>`. However, this does not imply that the implementation of the template members is required to reside in the header files - a simple alternative is to explicitly instantiate the corresponding templates in suitable object files.

Explicit instantiation obviously works for the template arguments mentioned in the Standard, for example, explicit specialisation of `std::basic_ios<char>` and `std::basic_ios<wchar_t>` works for the *class-template* `std::basic_ios`. But what happens when the user tries some other type as the character representation, or a different type for the character traits? Since the implementation is not inline but requires explicit instantiation, it cannot always be present in the Standard library shipped with the compiler. The usual approach to this problem is to use the `export` keyword but in the absence of this, an entirely different approach is necessary. One such approach is to instruct the user on how to instantiate the corresponding classes using for example, some environment specific implementation file, and suitable compiler switches. For instance, instantiating the *IOStream* classes for the character type `mychar` and the traits type `mytraits` might look something like:

```
c++ -o io-inst-mychar-mytraits.o io-inst.cpp \
      -DcharT=mychar -Dtraits=mytraits -Dinclude="mychar.hpp"
```

Using such an approach causes some trouble to the user and more work for the implementor, which seems to be a fairly high price to pay for a reduction in dependencies, and a speed up of compile-time. But note that the improvement in compile-time is typically significant when compiling with optimisations enabled. The reason for this is simple: with all those inline functions, the compiler causes huge chunks of codes to be passed on to the optimiser which then has to work extra hard to improve them. Bigger chunks provide better optimisation possibilities, but they also cause significantly longer compile-times due to the non-linear increase in the complexity of the optimisation step as the size of the chunks increases. Furthermore, the object files written and later processed by the linker are much bigger when all used instantiations are present in each object file. This can also impact the executable size, because certain code may be present multiple times embedded in different inline functions which are different but which have some code from just one other function in common.

Another reason for having the *IOStream* and *Locale* functions in a separate place, is that it is possible to tell from the undefined symbols, which features are used in a program, and which are not. This information can then be used by a smart-linker to determine which particular implementation of a function is most suitable for a given application.

3.2.5 Smart Linking

The discussion above already addresses how to avoid unused code using a slightly non-trivial implementation of locales and virtual functions. It does not address how to avoid unnecessary code. The term "unnecessary code" refers to code that is actually executed, but which does not really have any effect. For example, the code for padding formatted results does not have an effect if the `width()` is never set to a non-zero value. Similarly, there is no need to go through the virtual functions of the various facets, if only the default locale ever used. As for all other aspects of C++, it is reasonable to avoid the costs in code size and performance when the corresponding feature is not used.

The basic idea for coping with this is to provide multiple implementations of the same function that avoids unnecessary overheads where possible. Since writing multiple implementations of the same function can easily become a maintenance nightmare, it makes sense to write one implementation, which is configured at compile-time to handle different situations. For example, a function for numeric formatting that optionally avoids the code for padding might look like this:

```
template <typename cT, typename OutIt>
num_put<cT, OutIt>::do_put(OutIt it, ios_base& fmt,
                           cT fill, long v) const
{
    char buffer[some_suitable_size];
    char* end = get_formatted(fmt, v);
    if (need_padding && fmt.width() > 0)
        return put_padded(it, fmt, fill, buffer);
    else
        return put(it, fmt, buffer);
}
```

The value `need_padding` is a constant Boolean which is set to `false` if the compilation is configured to avoid padding code. With a clever compiler (normally requiring optimisation to be enabled) any reference to `put_padded()` is avoided, as is the check for whether the `width()` is greater than zero. The library would just supply two versions of this function and the smart-linker would need to choose the right one.

To choose the right one, the linker has to be told under what circumstances it should use the one avoiding the padding, i.e. the one where `need_padding` is set to `false`. A simple analysis shows that the only possibility for `width()` being non-zero is the use of the `std::ios_base::width()` function with a parameter. The library does not set a non-zero variable, and hence the simpler version can be used if `std::ios_base::width()` is never referenced from user code.

The example of padding is pretty simple. Other cases are more complex but still manageable. Another issue worth considering is whether the *Locales* library has to be used or if it is possible to provide the functionality directly, possibly using functions that are shared internally between the *Locales* and the *IOStreams* library. That is, if only the default locale is used, the *IOStream* functions can call the formatting functions directly, bypassing the retrieval of the corresponding facet and associated virtual function call - moreover, bypassing any code related to locales - avoiding the need to drag in the corresponding locale maintenance code.

The analysis necessary to check if only the default locale is used is more complex however. The simple test is to check for the locale's constructors. If only the default and copy constructors are used, then only the default locale is used because one of the other constructors is required to even create a different locale object. Even then, if another locale object is constructed, it is not necessarily used with the *IOStreams*. Only if the global locale is ever changed, or if `std::ios_base::imbue()`, `std::basic_ios<...>::imbue()`, or `std::basic_streambuf<...>::imbue()` are ever called, can the streams be affected by the non-default locale object. Although it this is somewhat more complex to determine, it is still manageable. There are other things which might be exploited too, for example, whether the streams have to deal with exceptions in the input or output functions (this depends on the stream buffer and locales possibly used); whether calling of `callback` functions is needed (only if `callback`s are ever registered, is this necessary); etc.

The approach taken by the linker to decide which functionality is used by the application requires using a set of "rules" provided by the library implementor to exclude functions. It is important to base these rules only on the application code to avoid unnecessary restrictions imposed by unused library code. This however results in more rules and rules that are more complex. To determine which functionality is used by the application code, the unresolved symbols referenced by the application code are examined. This requires that any function used as a "rule" is indeed unresolved and results in the corresponding functions being non-inline.

There are basically three problems with this approach:

- The maintenance of the implementation becomes more complex because extra work is necessary. This can be reduced to a more acceptable level by relying on clever compilers eliminating code for branches that the compiler can determine, are never used.

- The analysis of the conditions under which code can be avoided is sometimes non-trivial. Also, the conditions have to be made available to the linker, which introduces another potential cause of error.
- Even simple functions used to exclude a simple implementation of the function `std::ios_base::width()` cannot be inline. This might result in less efficient and sometimes even bigger code (for simple functions the cost of calling the function can be bigger than the actual function). See 3.2.7 for an approach for avoiding this problem.

The same approach can be beneficial to other libraries, and to other areas of the Standard C++ library than just the *IOStreams* and *Locales* library. In general, it can simplify the library interface by removing similar functions applicable in different situations, while still retaining the same efficiency. It is however, not always applicable in such situations and should be used carefully where appropriate.

3.2.6 Object Organization

A typical approach to organise a class is to have member variables for all attributes to be maintained. This may seem to be a natural approach, but it can result in a bigger footprint than necessary. For example, in an application where the `width()` is never changed, there is no need to actually store the `width()`. When looking at the *IOStreams* library, it turns out that each `std::basic_ios` object might store a relatively large amount of data to provide functionality that many C++ programmers using *IOStreams* are not even aware of, for example:

- A set of formatting flags is stored in an `std::ios_base::fmtflags` object.
- Formatting parameters like the `width()` and the `precision()` are stored in `std::streamsize` objects.
- An `std::locale` object (or some suitable reference to its internal representation) is stored.
- The `pword()` and `iword()` lists are stored.
- A list of `callback`s is stored.
- The error flags and exception flags are stored in objects of type `std::ios_base::iostate`. Since these basically consist of just three bits, they may be folded into just one word.
- The fill character used for padding is stored.
- A pointer to the used stream buffer is stored.
- A pointer to the `tie()`ed `std::basic_ostream` is stored.

This results in at least eight extra 32-bit words, even when folding multiple data into just one 32-bit word where possible (the formatting flags, the state and exception flags, and the fill character can fit into 32-bits for the character type `char`). These are 32 bytes for every stream object even if there is just one stream, for example, `std::cout` which never uses a different precision, width (and thus no fill character), or locale; probably does not set up special formatting flags using the `pword()` or `iword()` facilities; almost certainly does not use `callback`s, and is not `tie()`ed to anything. It might get away with being an object needing no members at all, and in such a case - which is not very unlikely in an embedded application - by just sending string literals somewhere!

A different organization could be the use of an array of unions and using the `pword()/iword()` mechanism to store the data. Each of the pieces of data listed above is given an index of its position in an array of unions (possibly, several pieces can share just one union like they shared just one word in the conventional setting). Only the `pword()/iword()` pieces would not be stored in this array because they are required to access the array. A feature never accessed does not get an index and thus does not require any space in the array. The only complication is how to deal with the `std::locale`, because it is the only non-POD data. This can be handled using for example, a pointer to the locale's internal representation.

Depending on the exact organization, the approach will show different run-time characteristics. For example, the easiest approach for assigning indices is to do it on the fly when the corresponding data is initialised or first accessed. This may however, result in arrays which are smaller than the maximum index and thus the access to the array has to be bounds-checked (in case of an out-of-bound access, the array might have to be increased; it is only an error to access the corresponding element if the index is bigger than the biggest index provided so far by `std::ios_base::xalloc()`).

An alternative is to determine the maximum number of slots used by the Standard library at link time or at start-up time before the first stream object is initialised. In this case, there would be no need to check for out-of-bound access to the *IOStream* features. However, this initialisation is more complex.

A similar approach can be applied to the `std::locale` objects.

3.2.7 Library Recompilation

So far, the techniques described assume that the application is linked to a pre-packaged library implementation. Although the library might contain different variations on some functions, it is still pre-packaged (the templates possibly instantiated by the user can also be considered to be pre-packaged). This is however, often not a necessary assumption! If the library code is available, the Standard library can also be recompiled.

This leads to the “two phase” building of an application; where in a first phase, the application is compiled against a "normal", fully-fledged implementation. The resulting object files are automatically analysed for features actually used, by looking at the unresolved references. The result of this analysis is some configuration information (possible a file), which uses conditional compilation to remove all unused features from the Standard library; in particular, removing unused member variables and unnecessary code. In the second phase, this configuration information is then used to recompile the Standard library and the application code for the final program.

This approach does not suffer from drawbacks due to dynamic determination of what are effectively static features. For example, if it is known at compile-time which *IOStream* features are used, the stream objects can be organised to include members for exactly those features. Thus, it is not necessary to use a lookup in a dynamically allocated array, possibly using a dynamically assigned index. Also, in the final compilation phase, it is possible to inline functions that were not previously inlined (in order to produce the unresolved symbol references).

3.3 ROMability

For the purposes of this paper, the terms "ROMable" and "ROMability" refer to entities that are appropriate for placement in "Read-Only-Memory" and to the process of placing entities into Read-Only-Memory so as to enhance the performance of programs written in C++.

There are two principal domains that benefit from this process:

- Embedded programs which have constraints on available memory, where code and data must be stored in physical ROM whenever possible.
- Modern operating systems which support the sharing of code and data among many instances of a program, or among several programs sharing invariant code and data.

The subject of ROMability therefore has performance application to all programs, where immutable aspects of the program can be placed in a shared and "Read-Only" space. On hosted systems, Read-Only is enforced by the memory manager, while in embedded systems, it is enforced by the physical nature of ROM devices.

For embedded programs where memory requirements are scarce, it is critical that compilers identify strictly ROMable objects and allocate only ROM area for them. For hosted systems, the requirement to share ROMable information is not as critical, but there are inevitable performance advantages to hosted programs as memory footprint and the time it takes to load a program can be greatly reduced. All the techniques described in this section will benefit such programs.

3.3.1 ROMable Objects

Most constant information is ROMable. Obvious candidates for ROMability are objects of static extent that are declared `const`, and which have constant initialisers; but there are several other significant candidates too.

Objects which are not declared `const` can be modified, and are consequently not ROMable. But these objects may have constant initialisers, and those initialisers may be ROMable. This paper considers those entities in a program that are obviously ROMable such as global `const` objects; entities that are generated by the compilation system to support functionality such a *switch-statement*s; and also places where ROMability can be applied to intermediate entities which are not so obvious.

3.3.1.1 User-defined objects

Objects declared `const` that are initialised with constant expressions. Examples:

- An aggregate (§IS-18.5.1) object with static storage duration (§IS-3.7.1) whose initialisers are all constants:

```
static const int tab[] = {1,2,3};
```

- Objects of scalar type with external linkage:

  A const-qualified object of scalar type has internal (§IS-7.1.5.1) or no (§IS-3.2¶5) linkage and thus can usually be treated as a compile-time constant, i.e. object data areas are not allocated, even in ROM. For example:

```
const int tabelsize = 48
double table[tablesize];  // table has space for 48 doubles
```

  However, if an object of scalar type is used for initialisation or assignment of pointer or reference variables, it has internal linkage and is ROMable. For example:

```
extern const int a = 1;   // extern linkage
const int b = 1;          // internal linkage
const int *c = &b;        // variable b should be allocated
const int tbsize = 256;   // it is expected that tbsize is not
                          // allocated at run-time
char ctb[tbsize];
```

- String literals:

  An ordinary string literal has the type "array of *n* `const char`" (§IS-2.13.4), and so they are ROMable. A string literal used as the initialiser of a character array if ROMable, but if the variable to be initialised is not a const-qualified array of `char`, then the variable being initialised is not ROMable:

```
const char *str1 = "abc"; // both str1 and "abc" are ROMable
char str2[] = "def";      // str2 is not ROMable
```

  A compiler may achieve further space savings by sharing the representation of string literals in ROM. For example:

```
const char* str1 = "abc"; // only one copy of "abc" needs
const char* str2 = "abc"; // to exist, and it is ROMable
```

  Yet further possibilities for space saving exists if a string literal is identical to the trailing portion of a larger string literal, as only the larger string literal is necessary, as the smaller one can reference the common sub-string of the larger. For example:

```
const char* str1 = "Hello World";
const char* str2 = "World";

// Could be considered to be implicitly:
const char* str1 = "Hello World";
const char* str2 = str1 + 6;
```

3.3.1.2 Compiler-generated objects

- Jump tables for switch statements:

  If a jump table is generated to implement switch statement, the table is ROMable, since it consists of a fixed number of constants known at compile-time.

- Virtual function tables:

  Virtual function tables of a class are usually ROMable.

Note:	*For some implementations, the virtual function tables may not be ROMable where dynamic linking is involved, and the virtual function tables are in a shared library.*
Note also:	*It may be appropriate to discuss flash cards here, and how they can introduce code into a system.*

- Type identification tables:

  When a table is generated to identify RTTI types, the table is usually ROMable.

Note:	*For some implementations, the type identification tables may not be ROMable where dynamic linking is involved, and the type identification tables are in a shared library.*

- Exception tables:

  When exception handling is implemented using a static table, the table is usually ROMable.

Note:	*For some implementations, the exception tables may not be ROMable where dynamic linking is involved, and the exception tables are in a shared library.*

- Reference to constants:

  If a constant expression is specified as the initialiser for a const-qualified reference, a temporary object is generated (§IS-8.5.3).

  This temporary object is ROMable, for example:

```
// The declaration:
const double & a = 2.0;

// May be represented as:
static const double tmp = 2.0; // 'tmp' is ROMable
const double & b = tmp;
```

- Initialisers for aggregate objects with automatic storage duration:

  If all the initialisers for an aggregate object that has automatic storage duration are constant expressions, a temporary object that has the value of the constant expressions and a code that copies the value of the temporary object to the aggregate object may be generated. This temporary object ROMable, for example:

```
struct A {
    int a;
    int b;
    int c;
};
void test() {
    A a = {1,2,3};
}

// May be interpreted as:
void test () {
    static const A tmp = {1,2,3};  // 'tmp' is ROMable
    A b = tmp;
}
```

Thus, the instruction code for initialising the aggregate object can be replaced by a simple bitwise copy, saving both code space and execution time.

- Constants created during code generation:

  Some literals such as integer literals, floating point literals and addresses can be implemented as either instruction code or data. If they are represented as data, then these objects are ROMable. For example:

```
void test() {
    double a = 1.0;
    a += 1.0;
}

// May be interpreted as:
void test () {
    static const double tmp  = 1.0;  // 'tmp1' is ROMable
    double a = 1.0;
    a += tmp;
}
```

3.3.2 Constructors and ROMable Objects

In general, objects of classes with constructors must be dynamically initialised. However, in some cases the initialisation could be performed if static analyses of the constructors resulted in constant values being used. In this case, the object could be ROMable. Similar analyses would need to be performed on the destructor.

```
class A {
    int a;
public:
    A(int v) : a(v) { }
};
const A tab[2] = {1,2};
```

Editor's Note:	*If sufficient analyses reveals that the object eventually gets a particular value, and the program cannot detect whether it acquired*

that value by constant or dynamic means, then it is quite legitimate for it to be ROMable[18].

Furthermore, even if it is not a `const` *object, the initialisation "pattern" may be ROMable, and bitwise copied to the object when it is initialised. For example:*

```
class X {
    int a;
    char* p;
public:
    A ()
    : a ( 7 )
    { std::cout << "Hello World" << std::endl";
      p = "Hi"; }
};
A not_const;
```

In this case, all objects are initialised to a constant value (the pair {7,&"Hi"}). This constant initial value is ROMable, and the constructor could perform a bitwise copy of that constant value and the calls to the IOStream library.

3.4 Hard Real-Time Considerations

For most embedded systems, only a very small part of the software is really real-time critical. But for that part of the system, it is important to exactly determine the time a specific piece of software needs to run. Unfortunately, this is not an easy analysis to do for modern computer architectures using multiple pipelines and different types of caches. Nevertheless, for a lot of code sequences it is still quite straightforward to do a worst-case analysis.

Note (Detlef): *Bjarne's Phrase goes here.*

Editor's Note: *What is "Bjarne's Phrase"?*

This statement also holds for C++. Here is a short description of several C++ features and their time predictability.

3.4.1 C++ Features for which an Accurate Timing Analysis is Easy

3.4.1.1 Templates

As pointed out in detail in 2.5, there is no real-time relevant overhead for calling template functions or member functions of class templates. On the contrary, templates often allow for better inlining and therefore reduce the overhead of the function call.

If the function is a virtual function, the normal rules for virtual functions apply.

[18] This is an optimisation, and is subject to the so-called "as if rule" (§IS-1.9¶1)

3.4.1.2 Inheritance

Converting a pointer to a derived class to a pointer to base-class[19] will not introduce any run-time overhead in most implementations (see **Error! Reference source not found.**). If there is an overhead (very few implementations), it is a fixed number of machine instructions (typically one) and can be easily tested with a test program. Being a fixed overhead, this overhead does not depend on the deepness of the derivation.

3.4.1.2.1 Multiple-Inheritance

Converting a pointer to a derived class to a pointer to base class might introduce run-time overhead (see **Error! Reference source not found.**). This overhead is a fixed number of machine instructions (typically one).

3.4.1.2.2 Virtual-Inheritance

Converting a pointer to a derived class to a pointer to a virtual base class will introduce run-time overhead in most implementations (see **Error! Reference source not found.**). This overhead is typically a fixed number of machine instructions.

3.4.1.3 Virtual Functions

Calling a virtual function often does not produce any run-time overhead (see **Error! Reference source not found.**). If it does, it will typically be a fixed number of machine instructions.

3.4.2 C++ Features, for which Real-Time Analysis is More Complex

The following features are often considered to be prohibitively slow for hard real-time code sequences. But this is not always true. For one, the run-time overhead of these features is often quite small, and on the other-hand even in the real-time parts of your program, you might have quite a number of CPU cycles to spend. And if you have a complex job to do in your real-time code, a clean structure that allows for an easier overall timing analysis is often better than a hand-optimised but complicated code – as long as the former is fast enough. The hand-optimised code might run faster but is in most cases more difficult to analyse correctly. And the features mentioned below often allow for clearer designs.

3.4.2.1 Dynamic Casts

In most implementations, dynamic casts from a pointer (or reference) to base-class to pointer (or reference) to derived-class (i.e. a downcast) will produce an overhead that is not fixed but depends on the details of the implementation and there is no general rule to test the worst case.

The same is true for cross-casts (see 2.2).

[19] Such a conversion is also necessary if a function is called for a derived-class object that is implemented in a base-class.

As an alternate option to using dynamic-casts, you should consider the `typeid` operator. If you know your target's dynamic type exactly, this is a much cheaper way to check for it.

3.4.2.2 Dynamic Memory Allocation

Dynamic memory allocation has in typical implementations a run-time overhead that is not easy to analyse. In most cases, for the purpose of real-time analysis it is appropriate to assume dynamic memory allocation (and also memory de-allocation) to be non-deterministic.

The most obvious way to avoid dynamic memory allocation is to pre-allocate the memory – either statically at compile- (or more correctly link-) time or during the general set-up-phase of your system. If you want to defer the initialisation, you can pre-allocate raw memory and initialise it later using placement new.

If you really need to do dynamic memory allocation in your real-time code, you need to use an implementation for which you know all the implementation details. The best way to know all the implementation details is to write your own memory allocation mechanism. This is easily done in C++ by overriding `operator new` in your own class (or globally) or by providing an allocator argument in standard library containers.

But in all cases, if you use dynamic memory allocation you need to consider the case when no more memory is available.

3.4.2.3 Exceptions

Enabling exceptions for compilation may introduce overhead on each function call in your code (see **Error! Reference source not found.**). In general, it is not so difficult to analyse the overhead of exception handling as long as you don't throw exception. But you should only enable exception handling for real-time critical programs if you really use exceptions, and therefore a complete analysis must always include the throwing of an exception, and this analysis will always be implementation dependent. On the other hand, the requirement to act within a deterministic time might loosen in the case of an exception (e.g. you don't need to handle any more input from a device when a connection broke down).

An overview of alternatives for exception handling is given in *[Note (Detlef): Insert Bjarne's new section]*. But as shown there, all options have their run-time costs, and throwing exceptions might still be the best way to deal with exceptional cases. And as long as you don't throw a long way (i.e. if you only leave very few functions in your throw), it might be even cheap in run-time.

Note (Detlef): Is this list complete?

3.4.3 Testing Timing

For those features that compile to a fixed number of machine instructions, the number and nature of these instructions (and therefore an exact worst-case timing) can be tested with a simple program that includes just this specific feature and then looking at the created code. In general, for those simple cases, optimisation should not make a difference. But e.g. if a virtual function call can be resolved to a static function call at compile-time, the overhead of the virtual function call will not show up in the code. So, you need to make sure that you really test what you want to test.

For the more complex cases, testing the timing is not so easy. Compiler optimisation can make a big difference, and a simple test case might produce completely different code than the real production code. To test those cases, you must really know the details for your specific implementation. Given this information, you can normally produce test programs that produce code from which you can correctly derive the timing information you need.

4 Embedded Systems – Special Needs

As the C language has matured over the years, various extensions for accessing basic I/O-Hardware registers have been added to address deficiencies in the language. Today almost all C compilers for free-standing environments and embedded systems support some method of direct access to I/O-Hardware registers from the C source level. However, these extensions have not been consistent across dialects. As a growing number of C++ compiler vendors are now entering the same market, the same I/O driver portability problems become apparent for C++.

As a simple portability goal, the driver source code for some given I/O-Hardware should be portable to all processor architectures where the hardware itself can be connected. Ideally, it should be possible to compile source code that operates directly on I/O-Hardware registers with different compiler implementations for different platforms and get the same logical behaviour at run-time.

Obviously standard interface definitions written in the common subset of C and C++ would have the widest potential audience, since they would be readable by compilers for both languages. But the additional abstraction mechanisms of C++, such as classes and templates, are useful in writing code at the hardware access layer. They allow the encapsulation of features into classes, providing type safety along with maximum efficiency through the use of templates.

Nevertheless, it is an important goal to provide an interface that allows device driver implementers to write code that compiles equally under C and C++ compilers. Therefore, this report specifies two interfaces: one using the common subset and a second using modern C++ constructs. Implementers of the common subset-style interface might use functions or inline functions, or might decide that function-like macros or intrinsic functions better serve their objectives.

A proposed interface for addressing I/O-Hardware in the C language is described in:

> **Technical Report ISO/IEC WDTR 18037**
>
> *"Extensions for the programming language C to support embedded processors "*

This interface is referred to as *iohw* in this report. It is included in this report for the convenience of the reader. If the description of *iohw* in this report differs from the description in the ISO/IEC WDTR 18037, the description there takes precedence. *iohw* is also used to refer to both the C and C++ interface where they share common characteristics. In parallel with that document, the interfaces using the common subset of C and C++ are contained in a header named <iohw.h>.

Although *iohw* is based on C macros, the C++ language provides features which make it possible to create efficient and flexible implementations of this interface, while maintaining I/O driver source code portability. The C++ interfaces are contained in a header named <hardware>, and their symbols are placed in namespace std::hardware.

The name is deliberately different, as it is the intention that <hardware> provides similar functionality to <iohw.h>, but through a different implementation, just as <iostream> provides parallel functionality with <stdio.h> through different interfaces and implementation. There is no header <ciohw> specified, as that name would imply (by analogy with other standard library headers) that the C++ interfaces were identical to those in <iohw.h> but placed inside a namespace.

This report provides

- a general introduction and overview to the *iohw* interface (4.1),
- the copy of the C interface (4.2.1),
- the description of the C++ interface (4.2.2),
- usage guidelines for the C++ interface (4.3),
- general implementation guidelines for both interfaces (4.3.1)
- implementation guidelines for the C++ interface (**Error! Reference source not found.**)
- implemention guidelines and example for the C interface on top of the C++ interface (**Error! Reference source not found.**)
- an example of a real world device driver using the C++ interface (4.5)

4.1 Introduction to I/O-Hardware Addressing

Detlef: This is essentially the old 4.1.3 and 4.2.

The purpose of the *iohw* access functions defined in the *iohw* header file is to promote portability of *iohw* driver source code across different execution environments.

4.1.1 Basic Standardisation Objectives

A standardisation method for basic hardware register addressing must be able to fulfil three requirements at the same time:

- A standardised interface must not prevent compilers from producing machine code that has no additional overhead compared to code produced by existing proprietary solutions. This requirement is essential in order to get widespread acceptance from the embedded programming community.
- The I/O driver source code modules should be completely portable to any processor system without any modifications to the driver source code being required *[i.e. the syntax should promote I/O driver source code portability across different execution environments]*.
- A standardised interface should provide an "encapsulation" of the underlying access mechanisms to allow different access methods, different processor architectures, and different bus systems to be used with the same I/O driver source code *[i.e. the standardisation method should separate the characteristics of the hardware register itself from the characteristics of the underlying*

execution environment (processor architecture, bus system, addresses, alignment, endianness, etc.)].

4.1.2 Overview and Principles

The *iohw* access functions create a simple and platform independent interface between I/O driver source code and the underlying access methods used when addressing the hardware registers on a given platform.

The primary purpose of the interface is to separate characteristics which are portable and specific for a given hardware register, for instance the register bit width, from characteristics which are related to a specific execution environment, such as the hardware register address; processor bus type and endianness; device[20] bus size and endianness, address interleave; compiler access method; etc. Use of this separation principle enables I/O driver source code itself to be portable to all platforms where the hardware registers can be connected.

In the driver source code, a hardware register must always be referred to using a symbolic name. The symbolic name must refer to a complete definition of the access method used with the given register. A standardised *iohw* syntax approach creates a conceptually simple model for hardware registers:

symbolic name for hardware register ⇔ complete definition of the access method

When porting the driver source code to a new platform, only the definition of the access method (definition of the symbolic name) needs to be updated.

4.1.3 The Abstract Model

The standardisation of basic *iohw* addressing is based on a three layer abstract model:

The user's portable source code
The user's I/O register definitions
The vendor's *iohw* implementation

The top layer contains the driver source code written by the compiler user. The source code in this layer is fully portable to any platform where the hardware device can be connected. This code may only access hardware registers via the standardized functions described in this section. Each hardware register must be identified using a symbolic name.

The bottom layer is the compiler vendor's implementation of *iohw*. It provides prototypes for the functions defined in this section and specifies the various access methods supported by the given processor and platform architecture ("access

[20] In this document, the term *device* is used to mean either a discrete *I/O chip* or an *I/O function block* in a single chip processor system. The data bus width has significance to the access method used for the I/O device.

methods" refers to the various ways of connecting and addressing hardware registers or hardware devices in the given processor architecture).

4.3 contains some general considerations that should be addressed when a compiler vendor implements the *iohw* functionality.

The middle layer contains the user's specification of the symbolic hardware register names used by the source code in the top layer. This layer associates a symbolic name with an *access-specification* for a specific hardware register on the given platform. The syntax notation and *access-specification* parameters used in this layer are specific to the platform architecture and are defined by the compiler vendor in the *iohw* header. The user must update these hardware register *access-specifications* when the hardware driver source code is ported to a different platform.

4.2.2 proposes a generic C++ syntax for hardware register *access-specifications*. Using a general syntax in this layer may extend portability to include user's hardware register specification, so it can be used with different compiler implementations for the same platform.

4.1.3.1 The Module Set

A typical device driver operates with a minimum of three modules, one for each of the abstraction layers. For example, it is convenient to locate all hardware register name definitions in a separate header file (called `"hw_ta.h"` in this example):

1. Device driver module
    - The I/O driver source code
    - Portable across compilers and platforms
    - Includes the interface header and `"hw_ta.h"`
2. Interface header
    - Defines I/O functions and access methods
    - Typically specific for a given compiler
    - Implemented by the compiler vendor
3. `"hw_ta.h"`
    - Defines symbolic hardware register names and their corresponding access methods
    - Specific to the execution environment
    - Implemented and maintained by the programmer

And might be used as follows:

In common subset of C and C++:

```
#include <iohw.h>
#include "hw_ta.h"   // my HW register definitions for target

unsigned char mybuf[10];
//...
iowr(MYPORT1, 0x8);                    // write single register
for (int i = 0; i < 10; i++)
```

```
    mybuf[i] = iordbuf(MYPORT2, i);  // read register array
```

In C++:

```
#include <hardware>
struct UCharBuf { unsigned char buffer[10]; };
#include "hw_ta.h"   // my HW register definitions for target
// contains definitions of MyPort1T, MyPort2T, MyPort3T
// the value type for MyPort1T and MyPort2T is unsigned char
// the value type for MyPort3T is UCharBuf

unsigned char mybuf[10];
using namespace std::hardware;
//...
MyPort1T myPort1;            // define HW register object
myPort1 = 0x8;               // write single register

MyPort2T myPort2;
for (int i = 0; i < 10; i++)
    mybuf[i] = myPort2[i];  // read register array bytewise

MyPort3T myPort3;
UCharBuf mybufBlock;
myBufBlock = myPort3;        // reads complete register array at once
```

The programmer only sees the characteristics of the hardware register itself. The underlying platform, bus architecture, and compiler implementation do not matter during driver programming. The underlying system hardware may later be changed without modifications to the driver source code being necessary.

4.1.4 Hardware Register Characteristics

The principle behind *iohw* is that all hardware register characteristics should be visible to the driver source code, while all platform specific characteristics are encapsulated by the header files and the underlying *iohw* implementation.

Hardware registers often behave differently from the traditional memory model. They may be "read-only", "write-only" or "read-modify-write", often read and write operations are only allowed once for each event, etc.

All such hardware register specific characteristics should be visible at the driver source code level and should not be hidden by the *iohw* implementation.

4.1.5 The Most Basic Hardware Access Operations

The most common operations on hardware registers are "***read***" and "***write***".

Bit-set, bit-clear and bit-invert of individual bits in an *iohw* register are also commonly used operations. Many processors have special machine instructions for doing these.

For the convenience of the programmer, and in order to promote good compiler optimisation of bit operations, the basic logical operations "***or***", "***and***" and "***xor***" are defined by *iohw* in addition to "***read***" and "***write***".

All other arithmetic and logical operations used by the driver source code can be built on top of these few basic operations.

4.1.6 The *access-specification*

The *access-specifications* defined in *iohw* are used only as parameters in the functions for defining hardware register access.

The access specification parameter represents or references a complete description of how the hardware register should be addressed in the given hardware platform. It is an abstract data type with a well-defined behaviour[21].

The definition method and the implementation of *access-specifications* are processor and platform specific.

In general, an access specification will specify at least the following characteristics:

- Register size (mapping to a data type)
- Access limitations (read-only, write-only)
- Bus address for register

Other access characteristics typically specified via the access specification:

- Processor bus (if more than one)
- Access method (if more than one)
- Hardware register endianness (if register width is larger than the device bus width)
- Interleave factor for hardware register buffers (if bus width for the device is smaller)
- User supplied access driver functions

The definition of a hardware register object may or may not require a memory instantiation, depending on how a compiler vendor has chosen to implement *access-specifications*. In general, *iohw* implementers should provide optimized implementations where the definition of an object of an access specification type does not produce any overhead. The concrete implementation techniques will differ and be dependent on specific architecture, compiler and programming language.

See also 4.3.1 for further details and implementation considerations.

4.2 Interface definitions

This section defines both the C-style and C++-style interfaces.

The *iohw* C-style interface is a copy of the *iohw* C interface from Technical Report ISO/IEC WDTR 18037 "Extensions for the programming language C to support embedded processors," included here as a convenience to the reader. If ever this copy differs from the original in WDTR 18037, that original takes precedence.

[21] This use of an abstract data type is similar to the philosophy behind the well-known `FILE` type in C. Some general properties for `FILE` and streams are defined in the Standard, but the Standard deliberately avoids describing how the underlying file system should be implemented.

Detlef: This is a verbatim copy from the C TR and not intended for any editing.

4.2.1 The <iohw.h> interface

The header <iohw.h> declares several function-like macros which together create a data type- independent interface for basic I/O hardware addressing.

4.2.1.1 Function-like macros for single register access

Synopsis

```
#include <iohw.h>

iord(  access_spec )
iowr(  access_spec, value )
ioor(  access_spec, value )
ioand( access_spec, value )
ioxor( access_spec, value )
```

Description

These names map an *iohw* register operation to an underlying (platform specific) implementation which provides access to the I/O register identified by *access_spec*, and performs the basic operation `READ, WRITE, OR, AND` or `XOR` as identified by the function name on this register.

The data type (the I/O register size) for *value* parameters and the value returned by `iord` is defined by the *access_spec* definition for the given register. The macro-like functions `iowr`, `ioor`, `ioand` and `ioxor` do not return a value.

LG: Should "macro-like functions" be "function-like macros" here?

4.2.1.2 Function like macros for register buffer access

Synopsis

```
#include <iohw.h>

iordbuf(  access_spec, index )
iowrbuf(  access_spec, index, value )
ioorbuf(  access_spec, index, value )
ioandbuf( access_spec, index, value )
ioxorbuf( access_spec, index, value )
```

Description

These names map an *iohw* register buffer operation to an underlying (platform specific) implementation which provides access to the I/O register buffer identified by *access_spec*, and performs the basic operation READ, WRITE, OR, AND or XOR as identified by the function name on this register.

The data type (the I/O register size) for *value* parameters and the value returned by `iordbuf` is defined by the *access_spec* definition for the given register. The functions `iowrbuf`, `ioorbuf`, `ioandbuf` and `ioxorbuf` do not return a value.

The *index* parameter is offset in the register buffer (or register array) starting from the I/O location specified by *access_spec*, where element 0 is the first element located at the address defined by *access_spec*, and element n+1 is located at a higher address than element n.

It should be noted that the *index* parameter is the offset in the I/O hardware buffer, not the processor address offset. Conversion from a logical index to a physical address requires that interleave calculations are performed by the underlying implementation. This is discussed further in B.2.4

4.2.1.3 Function like macros for access_spec initialisation

Synopsis

```
#include <iohw.h>

io_at_init( access_spec )
io_at_release( access_spec )
```

Description

The `io_at_init` function maps to an underlying (platform specific) implementation which provides any *access_specification* initialisation before performing any other operation on the I/O register (or set of I/O registers) identified by *access_spec*. This macro should be placed in the driver source code so it is invoked at least once before any other operations on the related registers are performed. This function does not return a value.

The `io_at_release` function maps to an underlying (platform specific) implementation which releases any resources obtained by a previous call to `io_at_init` for the same *access_specification*. This call should be placed in the driver source code so it is invoked once after all operations on the related registers have been completed. This function does not return a value.

Example:

> In an implementation for a hosted environment, the call to `io_at_init` is used to identify the point in an execution sequence where the underlying access method should obtain, or have obtained, a handle from the operating system. This handle obtained is used in all following access operations on the I/O register. The call to `io_at_exit` identifies the point in an execution sequence where the handle can return to the operating system.

If a set of memory mapped I/O registers is specified to use based addressing (defined in C.3), the underlying implementation would dynamically obtain the base address for the I/O range from the operating system when `io_at_init` is invoked (i.e. the base pointer is initialised). During all the following I/O access operations the I/O register address is calculated as (*base-address* + I/O register *offset*). The underlying implementation later releases the memory range when `io_at_exit` is invoked.

If no *access_specification* initialisation is required by a given <iohw.h> header implementation, the `io_at_init` and `io_at_release` definitions may be empty.

4.2.1.4 Function for access_spec copying

Synopsis

```
#include <iohw.h>
io_at_cpy( access_spec dest, access_spec src)
```

Description

This function maps to an underlying (platform specific) implementation which copies the dynamic part of the source *access_spec* to the destination *access_spec*. The two parameters must have the same *access_specification* type. The macro does not return a value.

If *access_specification* copying is not supported by a given <iohw.h> header implementation, or a given access specification does not contain any dynamic elements, the `io_at_cpy` function may be empty.

A typical use for `io_at_cpy` is when a set of driver functions for a given I/O device type is used with multiple hardware instances of the same device. It often provides a faster alternative than passing the *access_spec* as a function parameter.

Example

```
#include <iohw.h>
#include <iohw_ta.h>  // MYCHIP_CFG and MYCHIP_DATA are defined
                      // relative to a dynamic MYCHIP_BASE

// Portable driver function
uint8_t my_chip_driver(void)
{
    iowr(MYCHIP_CFG, 0x33);
    return iord(MYCHIP_DATA);
}

// User's driver application
uint8_t  d1, d2;
// Read from our 2 I/O chips
```

```
io_at_cpy(MYCHIP_BASE, CHIP1); // Select chip 1
d1 = my_chip_driver();
io_at_cpy(MYCHIP_BASE, CHIP2); // Select chip 2
d2 = my_chip_driver();
```

4.2.2 The C++ Interface <hardware>

Detlef: Incomplete. This is essentially a description of my header file <hardware>.

This is not completely up-to-date. The descriptions in Using Guidelines are newer, as well as the accompanying C++ files hardware.h and hardware.cpp.

The header `<hardware>` declares several types, which together provide a data-type-independent interface for basic I/O-Hardware addressing.

Header `<hardware>` synopsis:

```
// proposed definition of <hardware>
// this is definition only

namespace std
{
namespace hardware
{
#include <stdint.h>

struct hw_base;

// required access types
template <typename ValueType,
          hw_base::device_bus devWidth,
          hw_base::byte_order endian,
          hw_base::processor_bus nativeWidth,
          hw_base::address_type address>
class direct_address;
[others still missing]
template <typename ac_type>
class register_access;
} // namespace hardware

} // namespace std
```

4.2.2.1 Header <stdint.h>

The header stdint.h is specified by C99 (IS 9899-1999). Including it in <hardware> introduces the fixed size integer types into namespace std::hardware.

No names are introduced into global namespace.

4.2.2.2 struct hw_base

```
struct hw_base
namespace std {
namespace hardware {
struct hw_base
{
    enum device_bus {device8, device16, device32, device64};
```

```
    enum byte_order {msb_low, msb_high};
    enum processor_bus {bus8, bus16, bus32, bus64};
    // only identifiers should be present that are supported by
    // the underlying implementation! (Diagnostic required.)

    typedef implementation defined address_type;
};
} // namespace hardware
} // namespace std
```

1 Struct hw_base provides the names for the supported hardware characteristics. Only those names that are supported by the hardware shall be present.

```
enum device_bus {device8, device16, device32, device64};
```

2 This defines the names for the width of the hardware register device bus as seen from the processor.

```
enum byte_order {msb_low, msb_high};
```

3 This defines the names for the endianness of the device register.

```
enum processor_bus {bus8, bus16, bus32, bus64};
```

4 This defines the names for the width of the processor bus.

5 `address_type` is an integral type specified by the application to hold a hardware address.

6 An implementation may define additional names and types in hw_base.

4.2.2.3 Common specifications for access specification types

```
typename ValueType
```

1 All access specification template types have at least a `ValueType` parameter. The argument for this parameter shall be an Assignable and CopyConstructible type.

2 The arguments for `ValueType` are not restricted to integral values. [Note: e.g. it makes perfect sense for `ValueType` to be `double` or `long double` when accessing an external floating-point co-processor. It might even be useful sometimes to have a user-defined struct as `ValueType`.]

3 The memory location of an object of the `ValueType` argument shall be readable and writeable freely by the implementation of this interface. [Note: This requirement essentially disallows other hardware registers or types. Also, their value might be changed through the implementation by direct memory access instead of any (possibly overloaded) assignment operators.]

4 Most of the access specification types have a common set of template parameters, which are specified as follows:

```
hw_base::device_bus devWidth
```

5 `devWidth` defines the width of the device to be accessed as seen by the processor.

6 `sizeof(ValueType)` must be a whole multiple (including 1) of `devWidth`.

```
hw_base::byte_order endian
```

7 `endian` defines whether the device to be accessed is low-endian or high-endian is attached to the bus.

Lois: The preceding sentence doesn't make sense to me, but I don't know how it should be changed. Are big- and little-endian more commonly used than low- and high-endian?

```
hw_base::processor_bus nativeWidth
```

8 `nativeWidth` defines the width of the processor bus.

9 All access specification types may have additional template parameters specified by the implementation. The implementation may also define default arguments for some of the template parameters. [Note: e.g. on segmented architectures there might be an additional segment parameter.]

4.2.2.4 template struct direct_address

```
// required access types
template <typename ValueType,
          hw_base::address_type address,
          hw_base::device_bus devWidth,
          hw_base::byte_order endian,
          hw_base::processor_bus nativeWidth>
struct direct_address
{
    typedef ValueType value_type;

    template <hw_base::address_type other_address> struct rebind
    {
        typedef direct_address<ValueType,
                               other_address,
                               endian,
                               nativeWidth,
                               devWidth> other;
    };
};
```

*Lois: Looking at the two references to direct_address here, plus the one in 4.2.2, we seem to be giving the template parameters in three different orders. Also note that it is **class** direct_address in 4.2.2 and **struct** direct_address here. Detlef, could you look this over again please?*

1 `direct_address` defines the access specification type for hardware registers for which the hardware address is known at compile time and the register addresses are directly mapped to memory addresses.

```
hw_base::address_type address
```

2 The argument to `address` shall be the actual address of the hardware register to be accessed by this access specification type.

```
typedef ValueType value_type;
```

3 `value_type` holds the `ValueType` template parameter.

```
template <hw_base::address_type other_address> struct rebind
{
    typedef direct_address<ValueType,
                           devWidth,
                           endian,
                           nativeWidth,
                           other_address> other;
```

Lois: this matches the order of template parameters in 4.2.2.

4 `rebind::other` is a type with the same hardware characteristics but a different hardware register address.

[other access specification types follow]

4.2.2.5 Template class register_access

Template class register_access

```
template <typename ac_type>
class register_access
{
public:
    typedef typename ac_type::value_type value_type;

    operator value_type() const;
    void operator=(value_type val) const;
    void operator|=(value_type val) const;
    void operator&=(value_type val) const;
    void operator^=(value_type val) const;

    struct _ref
    {
        implementation defined constructor(s)
        operator value_type() const;
        void operator=(value_type val) const;
        void operator|=(value_type val) const;
        void operator&=(value_type val) const;
        void operator^=(value_type val) const;
    };

    _ref operator[](size_t index) const;
};
```

1 The class `register_access` provides direct access to hardware registers.

2 The argument to the template parameter `ac_type` must be an instantiation of an *access specification* template type (or a plain class) provided by the implementation.

```
typedef typename ac_type::value_type value_type;
```

3 `value_type` names the value_type of the *access specification*.

```
operator value_type() const;
```

4 The conversion function to `value_type` provides read access to the hardware register.

```
void operator=(value_type val) const;
```

5 The assignment operator writes the value_type argument to the hardware register.

```
void operator|=(value_type val) const;
```

6 operator |= bitwise ors the hardware register with the value_type argument.

```
void operator&=(value_type val) const;
```

7 operator &= bitwise ands the hardware register with the value_type argument.

```
void operator^=(value_type val) const;
```

8 operator ^= bitwise xors the hardware register with the value_type argument.

```
struct _ref;
```

9 The index operator returns the equivalent to a reference to the location specified by index inside of the hardware register. The return value can be used like a register_access object, i.e. it can be written, read, and the bitwise or, and and xor can be applied to it.

10 The struct _ref provides the same overloaded operators as register_access to allow the same operations.

11 [Note: The name _ref is here given for illustration purposes only. The actual implementation may use a different name. This name shall not be used directly by the user.]

Detlef: I have omitted the function interface for now. If anybody thinks it is really important, please feel free to add it.

Lois: I think it is important, but don't want to delay the document at this point. To be added later.

4.3 Guidelines for Using the iohw Interfaces

Detlef: This will include some general examples and usage descriptions for using the interfaces (mainly C++, but with references to the C interface and examples shown in C and C++).

4.3.1 Usage Introduction

The design of the *iohw* C++ interface follows two lines of separation: one is between the definition of *access-specifications* and design driver code, and the other is between what is known at compile-time and what is only known at run-time. Unfortunately, these two lines of separation are neither orthogonal nor identical: e.g. the base address for base/offset addressing is only known at run-time, but belongs to the *access-specification*. As C++ is a typed language, the differences for the interface are in type, and therefore the main separation line for the interface definition itself is between what is statically known at compile-time (this goes as template arguments into types) and what is only known at run-time (this goes as function arguments or operator operands into the interface of `register_access`).

4.3.2 access-specifications

access-specifications specify how a given device register can be accessed. As such, they are mainly implementation defined entities, as these access details vary widely over different platforms. But there are some aspects that *access-specifications* have in common:

- templates with at least `ValueType` as template parameter
- exposition of this `ValueType` argument as `value_type` typedef
- a "templated typedef" `rebind` to provide a simple way to define *access-specifications* that differ from an existing one only in a specific aspect (typically the hardware address)

Also, on platforms where they are available, the names of some *access-specification* templates are pre-defined:

- `direct_address` for memory-mapped addresses that are known at compile-time

- `dynamic_address` for memory-mapped addresses that are only known at run-time

`direct_address` has at least three template arguments: `ValueType`, `mode` (read, write, etc.) and `address`. So on some platforms the user should be able to define a specific hardware port like this:

```
typedef direct_address<uint16_t, hw_base::read, 0x1234> InPort1;
```

This assumes that the hardware register is 16 bits, and all other characteristics are pre-defined by the platform's implementation. But on most implementations, the user has to specify additional template parameters. But as already said, these template parameters are platform dependent and can vary widely for more exotic platforms. Even the address parameter might vary: for a segmented addressing architecture there might be two parameters for a segment and an offset address instead of a single address parameter.

If there already exists a quite elaborate type definition `ComplexPortA` for a specific device register with lots of template arguments and now another one is required with the same characteristics that differs only in its hardware address, this can be done with the `rebind` template:

```
typedef ComplexPortA::rebind<0x9876>::other ComplexPortB;
```

4.3.2.1 *access-specifications* with dynamic information

Some *access-specification*s may require additional information that is not available at compile-time. For those *access-specification*s, the *access-specification* template defines an additional parameter for the type of the dynamic data. The properties of this type are defined by the implementation, but the type itself is provided by the user to allow user-control of the initialization.

E.g. an implementation might provide a `dynamic_address` for which the dynamic data type must provide a public data member `address` of type `unsigned long`. Then the user can provide a respective class:

```
struct DynAddressPortDA
{
    DynAddressPortDA() : address(globalBase+0x120) {}

    unsigned long address;
};
typedef dynamic_address<uint32_t,
                        hw_base::random,
                        DynAddressPortDA> PortDA_t;
```

Here, the initialization of the dynamic data is provided by some global variable.

As another example, on a different platform an implementation might require for the dynamic data two public data members `segment` and `offset`, both of type `unsigned short`. A user class may look like this:

```
struct DynAddressPortDB
{
    DynAddressPortDB(unsigned short off)
        : segment(0xF0), offset(off)
    {}

    unsigned short segment;
```

```
    unsigned short offset;
};
typedef dynamic_address<uint32_t,
                        hw_base::random,
                        DynAddressPortDB> PortDB_t;
```

Here, the constructor requires an argument. Therefore some initialization code must provide that argument. But the mechanics of the initialization are always left to the user to choose the best fitting method.

4.3.3 Hardware Access

All hardware access is provided through the template class `register_access`. For *access-specification*s that require no dynamic information the respective `register_access` objects contain no data and therefore are optimized completely out of existence by most compilers. So, a typical usage would be:

```
// defined access-specifications with ValueType = uint8_t:
//   InPort, OutPort and ControlPort
register_access<InPort> ip;
register_access<OutPort> op;
register_access<ControlPort> ctl_p;

uint8_t tmp;

tmp = ip;            // read from InPort, uses
                     // register_access::operator value_type();
op = 0x12;           // write to OutPort, uses
                     // register_access::operator=(value_type);
ctl_p |= 0x34;       // set bits 5, 4 and 2 in ControlPort
```

As the `register_access` object is empty, there is no real need to define these objects, but it is also possible to use temporary objects created on the fly. The example above would then become:

```
// defined access-specifications with ValueType = uint8_t:
//   InPort, OutPort and ControlPort
typedef register_access<InPort> InP;
typedef register_access<OutPort> OutP;
typedef register_access<ControlPort> CtlP;

uint8_t tmp;

tmp = InP();         // read from InPort, uses
                     // register_access::operator value_type();
OutP() = 0x12;       // write to OutPort, uses
                     // register_access::operator=(value_type);
CtlP() |= 0x34;      // set bits 5, 4 and 2 in ControlPort
```

But this is a rather unnatural syntax and is generally not necessary as compilers are smart enough to optimize away the objects from the first example.

4.3.3.1 Indexed Access

`register_access` allows not only for access to single registers, but also for register blocks. The `ValueType` parameter of the *access-specification* denotes in this case the type of a single register and the address is the base address (index 0). The registers in the block can then be addressed through the index (subscript) operator:

```
// assume register block PortBuffer with random access
```

```
register_access<PortBuffer> portBuf;
uint8_t buf[sz];

portBuf[0] &= 0x03;
portBuf[1] = sz - 2;
for (int i=2; i != sz; ++i) buf[i] = portBuf[i];
```

If a full register block is always to be accessed, a respective `ValueType` can be defined:

```
struct Buffer32 { uint8_t data[32]; };
typedef direct_address<Buffer32,
                       hw_base::random,
                       0x35800,
                       ...> XYBlock;
register_access<XYBlock> blockBuf;
Buffer32 tmpBlock;

tmpBlock = blockBuf;   // read whole block at once
```

The binary layout of the `ValueType` must match the register block, which is normally only guaranteed for PODs. But if the register block has a complex layout (e.g. mix of different data types), the `ValueType` may be a correspondingly complex struct.

4.3.3.2 Initialization of register_access

For static *access-specification*s that are fully specified at compile-time `register_access` provides only a default constructor (in these cases there is nothing to construct). But if the *access-specification* contains dynamic data, this must be initialized at run-time. For those cases, `register_access` provides only a constructor that takes the dynamic data type of the *access-specification* as parameter. How this dynamic type is initialized is under control of the user, as explained above. So, regarding the examples from above, the initialization can either be:

```
register_access<PortDA_t> portDA=DynAddressPortDA();
```

or

```
register_access<PortDB_t> portDB=DynAddressPortDB(portDBOffset);
```

4.4 Implementing the iohw Interfaces

4.4.1 General Implementation Considerations

Detlef: This is mainly the old Appendix A.

4.4.1.1 Purpose

iohw defines a standardised function syntax for basic hardware addressing. The interface can either be provided by a library vendor or by the compiler vendor. If it is provided by the compiler vendor, it can contain special "compiler magic" that may be necessary to access special hardware with special addressing needs (or it might just provide better performance).

While a standardised function syntax for basic hardware addressing provides a simple, easy-to-use method for a programmer to write portable and hardware-platform-independent driver code, the *iohw* header itself may require careful consideration to achieve an efficient implementation.

This section gives some guidelines for implementers on how to implement *iohw* in a relatively straightforward manner given a specific processor and bus architecture.

4.4.1.1.1 Recommended Steps

Briefly, the recommended steps for implementing the iohw headers are:

- Get an overview of all the possible and relevant ways the hardware register is typically connected with the given bus hardware architectures, and get an overview of the basic software methods typically used to address such hardware registers.
- Define a number of functions, macros and *access-specifications* which support the relevant hardware access methods for the intended compiler market.
- Provide a way to select the right access function at compile-time and generate the right machine code based on the *access-specification* type or the *access-specification* object.

4.4.1.1.2 Compiler Considerations

In practice, an implementation will often require that very different machine code is generated for different hardware access cases. Furthermore, with some processor architectures, hardware register access will require the generation of special machine instructions not typically used when generating code for the traditional C or C++ memory model.

Selection between different code generation alternatives must be determined solely from the *access-specification* declaration for each hardware register. Whenever possible, this access method selection should be implemented such that it may be determined entirely at compile-time, in order to avoid any run-time or machine code overhead.

For a compiler vendor, selection between code generation alternatives can always be implemented by supporting different intrinsic *access-specification* types and keywords designed specially for the given processor architecture, in addition to the Standard types and keywords defined by the language.

However, with a conforming C++ compiler, an efficient, all-round implementation of both, the C++ and the C interface headers can usually be made using template functionality. A template-based solution allows the number of compiler specific intrinsic hardware access types or intrinsic hardware access functions to be minimized or even removed completely, depending on the processor architecture.

For compilers not supporting templates (such as C compilers) other implementation methods must be used. In any case, at least the most basic *iohw* functionality can be implemented efficiently using a mixture of macros, `inline` functions and intrinsic types or functions. Nevertheless, fully featured *iohw* implementations for several architectures will usually require direct compiler support.

But for many architectures, fully featured, zero-overhead implementations of *iohw* can be done using templates. An approach to doing this is discussed in 4.4.2.

4.4.1.2 Overview of Hardware Device Connection Options

The various ways of connecting an external device's register to processor hardware are determined primarily by combinations of the following three hardware characteristics:

- The bit width of the logical device register
- The bit width of the data-bus of the device
- The bit width of the processor-bus

4.4.1.2.1 Multi-Addressing and Device Register Endianness

If the width of the logical device register is bigger than the width of the device data bus, a hardware access operation will require multiple consecutive addressing operations.

The device register endianness information describes whether the most significant byte (MSB) or the least significant byte (LSB) of the *logical device register* is located at the *lowest* processor bus address.

(Note that the device register endianness is not coupled to the endianness of the underlying processor hardware architecture).

Table: Logical device register width / device bus width addressing overview[22]

Logical register widths	device bus widths							
	8-bit device bus		*16-bit device bus*		*32-bit device bus*		*64-bit device bus*	
	LSB-MSB	MSB-LSB	LSB-MSB	MSB-LSB	LSB-MSB	MSB-LSB	LSB-MSB	MSB-LSB
8-bit register	Direct		n/a		n/a		n/a	
16-bit register	r8{0-1}	r8{1-0}	Direct		n/a		n/a	
32-bit register	r8{0-3}	r8{3-0}	r16{0-1}	r16{1-0}	Direct		n/a	
64-bit register	r8{0-7}	r8{7-0}	r16{0-3}	r16{3-0}	r32{0-1}	r32{1-0}	Direct	

(For byte-aligned address ranges)

[22] Note, that this table describes some common bus and register widths for devices. A given hardware platform may use other register and bus widths.

4.4.1.2.2 Address Interleave

If the size of the device data bus is less than the size of the processor data bus, buffer register addressing will require the use of *address interleave*.

For example:

If the processor architecture has a byte-aligned addressing range with a 32-bit processor data bus, and an 8-bit device is connected to the 32-bit data bus, then three adjacent registers in the device will have the processor addresses:

```
<addr + 0>, <addr + 4>, <addr + 8>
```

This can also be written as

```
<addr + interleave*0>, <addr + interleave*1>, <addr + interleave*2>
```

where *interleave* = 4.

Table: Interleave overview: (bus to bus interleave relationship)

device bus widths	**Processor bus width**			
	8-bit bus	*16-bit bus*	*32-bit bus*	*64-bit bus*
8-bit device bus	interleave 1	interleave 2	interleave 4	interleave 8
16-bit device bus	n/a	interleave 2	interleave 4	interleave 8
32-bit device bus	n/a	n/a	interleave 4	interleave 8
64-bit device bus	n/a	n/a	n/a	interleave 8

(For byte-aligned address ranges)

4.4.1.2.3 Device Connection Overview:

The two tables above, when combined, show all relevant cases for how device registers can be connected to a given processor hardware bus, thus:

Table: Interleave between adjacent device registers in buffer

Register width	*Device bus*			*Processor data bus width*			
	Width	*LSB MSB*	*No. Oper- ations.*	*Width=8*	*Width=16*	*Width=32*	*Width=64*
				size 1	*size 2*	*size 4*	*size 8*
8-bit	*8-bit*	*n/a*	**1**	**1**	**2**	**4**	**8**
16-bit	*8-bit*	*LSB*	**2**	**2**	**4**	**8**	**16**
		MSB	**2**	**2**	**4**	**8**	**16**
	16-bit	*n/a*	**1**	n/a	**2**	**4**	**8**
32-bit	*8-bit*	*LSB*	**4**	**4**	**8**	**16**	**32**
		MSB	**4**	**4**	**8**	**16**	**32**
	16-bit	*LSB*	**2**	n/a	**4**	**8**	**16**
		MSB	**2**	n/a	**4**	**8**	**16**
	32-bit	*n/a*	**1**	n/a	n/a	**4**	**8**
64-bit	*8-bit*	*MSB*	**8**	**8**	**16**	**32**	**64**
		LSB	**8**	**8**	**16**	**32**	**64**
	16-bit	*LSB*	**4**	n/a	**8**	**16**	**32**
		MSB	**4**	n/a	**8**	**16**	**32**
	32-bit	*LSB*	**2**	n/a	n/a	**8**	**16**
		MSB	**2**	n/a	n/a	**8**	**16**
	64-bit	*n/a*	**1**	n/a	n/a	n/a	**8**

(For byte-aligned address ranges)

4.4.1.2.4 Generic Buffer Index

The interleave distance between two logically adjacent registers in a device register array can be calculated from[23]:

- The size of the logical register in bytes
- The processor data bus width in bytes
- The device data bus width in bytes

[23] For systems with byte-aligned addressing.

Conversion from register index to address offset can be calculated using the following general formula:

```
Address_offset = index *
                 sizeof( logical_IO_register ) *
                  sizeof( processor_data_bus ) /
                   sizeof( device_data_bus )
```

Assumptions:

- bytes are 8-bits wide
- address range is byte-aligned
- data bus widths are a whole number of bytes
- width of the `logical_IO_register` is greater than or equal to the width of the `device_data_bus`
- the width of the `device_data_bus` is less than or equal to the `processor_data_bus`

4.4.1.3 *access-specification*s for Different Device Addressing Methods

An implementer should consider the following typical addressing methods:

- ***Address is defined at compile-time:***

  The address is a constant. This is the simplest case and also the most common case with smaller architectures.

- ***Base address initialised at run-time:***

  Variable *base-address* + *constant-offset* i.e. the *access-specification* must contain an address pair (address of base register + offset of address).

  The user-defined *base-address* is normally initialised at run-time (by some platform-dependent part of the program). This also enables a set of driver functions to be used with multiple instances of the same device type.

- ***Indexed bus addressing:***

  Also called *orthogonal* or *pseudo-bus* addressing. This is a common method to connect a large number of device registers to a bus, while still occupying only a few addresses in the processor address space.

  This is how it works: first the *index-address* (or *pseudo-address*) of the device register is written to an address bus register located at a given processor address. Then the data read/write operation on the *pseudo-bus* is done via the following processor address, i.e. the *access-specification* must contain an address pair (the processor-address of the indexed bus, and the *pseudo-bus* address (or index) of the device register itself).

  This access method also makes it particularly easy for a user to connect common devices that have a multiplexed address/data bus, to a processor platform with non-multiplexed busses, using a minimum amount of glue logic. The driver source code for such a device is then automatically made portable to both types of bus architecture.

- ***Access via user-defined access driver functions:***

  These are typically used with larger platforms and with small single-chip processors (e.g. to emulate an external bus). In this case, the *access-specification* must contain pointers or references to access functions.

The access driver solution makes it possible to connect a given device driver source library to any kind of platform hardware and platform software using the appropriate platform-specific interface functions.

In general, an implementation shall always support the addressing methods *constant-address* and *base-address*. Apart from this, an implementer is encouraged to add additional methods required in a given domain.

Because of the different number of parameters required and the parameter ranges used in an *access-specification*, the C++ interface requires to define different type templates for the different addressing methods. For the C interface, it is often convenient to do the same and therefore implementing the C interface on top of the C++ interface.

Lois: The preceding paragraph needs some more work. It leaves the impression that a C-style programmer also wants to define several type templates, a syntactic impossibility. I suggest something more general, like "The C++-style interface using combinations of type templates provides greater type safety than the function-style macros. However, programmers using the common subset of C and C++ can still take advantage of this benefit by implementing <iohw.h> on top of the C++ code. See section 4.4.3 for more detailed discussion."

4.4.1.4 Atomic Operation

It is a requirement of the *iohw* implementation, that in each *iohw* function, a given (partial[24]) device register is addressed exactly once during a ***read*** or a ***write*** operation and at most[25] twice during a ***read-modify-write*** operation.

It is recommended that each *iohw* function in an *iohw* implementation be implemented such that the device access operation becomes *atomic* whenever possible. However, atomic operation is not guaranteed to be portable across platforms for the logical *read-modify-write* operations (i.e. the or, and and xor operations) or for multi-addressing cases. The reason for this is simply that many processor architectures do not have the instruction set features required for assuring atomic operation in these cases.

[24] A 32-bit logical register in a device with an 8-bit data bus contains 4 *partial* device registers.

[25] Depending on the processor's instruction set, modifying operations like '&=' can either be realized as one instruction and therefore require only one (write) access to the device register, or can require a full read-modify-write cycle with one read and one write access.

4.4.1.5 Read-Modify-Write Operations and Multi-Addressing

Detlef: Jan, how do you specify multi-address in your interface?

On processor architectures where the modifying operations (`or, and, xor`) can not be realized as single instruction operations, an implementation shall provide an *access-specification* that guarantees a *complete read – modify – complete write* realization for the modifying operations.

The rationale for this requirement is to allow *iohw* users to use the lowest common denominator of multi-addressing hardware implementations in order to support the widest possible range of *iohw* register implementations.

For instance, more advanced multi-addressing device register implementations often take a snap-shot of the whole logical device register when the first *partial* register is being read, so that data will be stable and consistent during the whole read operation. Similarly, write registers are often "double-buffered", so that a consistent data set is presented to the internal logic at the time when the access operation is completed by the last *partial* write.

Such hardware implementations often require that each access operation be completed before the next access operation is initiated.

4.4.1.6 Initialisation

With respect to the standardisation process, it is important to make a clear distinction between hardware (device) related initialisation, and platform related initialisation. Typically, three types of initialisation are related to device register operation:

- hardware (device) initialisation
- *access-specification* initialisation
- device selector initialisation[26]

Here only *access-specification* initialisation and device selector initialisation are relevant for the specification of *iohw*:

hardware initialisation: This is a natural part of a hardware driver, and should always be considered part of the device driver application itself. This initialisation is done using the standard functions for basic *iohw* addressing. Hardware initialisation is therefore not a topic for the standardisation process.

***access-specification* initialisation**: This concerns the initialisation and definition of `access_spec` objects themselves.

For many *access-specifications*, there is no run-time initalization necessary. However, for some address methods, some run-time initialization is required.

Using the *iohw* C-style interface, the function

```
io_at_init(access_spec)
```

[26] If for instance the access method is implemented as (base_address + constant_offset) then "device selector initialisation" refers to assignment of the base_address value.

can be used as a portable way to specify in the source code where and when such initialisation should take place.

Detlef: Sorry, but I can't see how you can provide a base address using io_at_init.

The *iohw* C++ interface provides the constructor

```
template <typename initType>
register_access::register_access(initType);
```

with an implementation defined `initType` for the same purpose.

device selector initialisation: This initialization can be used when, for instance, the same device driver code needs to service multiple devices of the same type.

One common possible solution is to define multiple *access-specification* objects, one for each of the devices, and then have the *access-specification* passed to the driver functions from the calling function.

The *iohw* C-style interface provides another solution: the use of *access-specification* copying, and *access-specifications* with dynamic access information. The function:

```
io_at_cpy(access_spec dest, access_spec src)
```

Detlef: I'd like to see a real example how this works.

In C++, this is most easily accomplished by providing a template function with the *access-specification* as template argument. For *access-specifications* with no run-time information this requires no data transfer (i.e. no function parameters). For *access-specifications* with dynamic information, this dynamic information must be passed as function parameters. The `rebind` struct in the *access-specification* provides a portable way to get an *access-specification* that differs from a formerly defined *access-specification* in only one parameter.

With most freestanding environments and embedded systems, the platform hardware is well defined, so all *access-specifications* for device registers used by the program can be completely defined at compile-time. For such platforms, standardised *access-specification* initialisation is not an issue.

With larger processor systems, device hardware is often allocated dynamically at run-time. Here the *access-specification* information can only be partly defined at compile-time. Some platform dependent part of the software must be initialised at run-time.

When designing the `access_spec` objects, a compiler implementer must therefore make a clear distinction between static information and dynamic information; i.e. what can be defined and initialised at compile-time, and what must be initialised at run-time.

Depending on the implementation method, and depending on whether the `access_spec` objects need to contain dynamic information, the `access_spec` objects may or may not require instantiation in data memory. Better execution performance can usually be achieved if more of the information is static.

4.4.2 Implementation Guidelines for the C++ Interface

Detlef: This will contain guidelines to get zero-overhead.

There are two main design alternatives in implementing register_access for the different *access-specification*s:

- Using the *access-specification*s as full-fledged traits classes that contain the information for *register_access* to behave accordingly (this is the approach chosen in the sample implementation).
- Using the *access-specification*s as mere labels and specializing *register_access* for each of these *access-specification*s (this is a useful approach if there are very few commonalities between the different *access-specification*s).

In any case, carefully implemented specializations of helper classes used in *register_access* can provide resulting code that only contains the necessary hardware access statements and produces absolutely no overhead.

The ultimate hardware access statements will nearly always be realized either as inline assembler or as compiler intrinsics. But this is hidden in the implementation; the user does not see them.

Detlef: The rest of this section will be a mere walk-through of the attached code hardware.cpp.

4.4.3 Implementing the C interface in Terms of the C++ Interface

Detlef: This will contain a full implementation example plus the old Appendix B.

4.4.3.1 Generic *access-specification* for *iohw* Addressing

4.4.3.1.1 Generic access-specification *Descriptor*

This chapter proposes a consistent and complete specification syntax for defining device registers and the access method parameters for the *iohw* C-style interface (i.e. though the syntax is C++ style for *access-specification*s, it is intended to be /~~used for the C interface at~~ /callable from code written in the common subset interfacing with/ the device register address level).

Lois: Jan and Detlef, is this change OK?

Prior art has used a number of (compiler intrinsic) memory type qualifiers or special keywords, which have varied from compiler to compiler and from platform to platform. The syntax described below represents an alternative approach and a superset solution, intended to replace prior art.

For optimal performance, the compiler should pick the right access method implementation at compile-time based on the *access-specification* type. This can be achieved in C++ by using `typedef`s and template specialisations.

4.4.3.1.2 Syntax Specification

access_spec specification:

> *typedef ACCESS_METHOD_CLASS_NAME < parameter list >*
> *SYMBOLIC_PORT_NAME;*

parameter list:

> *access method independent parameter list , access method specific parameter list*

access method independent parameter list:

> *type for device register value (size of device register) ,*
> *access limitation type ,*
> *device bus type (size and endian of device bus)*

type for device register value (size of device register):

> ***uint8_t***
> ***uint16_t***
> ***uint32_t***
> ***uint64_t***
> ***bool***
> *(+ optionally any basic type native to the implementation)*

access limitation type: // for compile-time diagnostic

rmw_e	*// read_modify_write*
rw_e	*// read_write*
wo_e	*// write_only*
ro_e	*// read_only*

device register device bus type:

device8	*// register width = device bus width = 8 bit*

per Jan's comment: These definitions device8l and device8h concern use of bus widths less than 8 bit and are be removed as the entire chapter only refer to register and bus widths of 8-16-32-64 bits.

device16	*// register width = device bus width = 16 bit*
device16l	*// register width > device bus width, MSB on low address*
device16h	*// register width > device bus width, MSB on high address*
device32	*// register width = device bus width = 32 bit*
device32l	*// register width > device bus width, MSB on low address*
device32h	*// register width > device bus width, MSB on high address*
device64	*// register width = device bus width = 64 bit*

> *(+ optionally any bus width native to the implementation)*

access method specific parameter list:

> *// Depends on the given access method. Examples are given later.*
> *// Three typical parameters are:*
> *primary address constant ,*
> *processor bus width type,*
> *address mask constant*

processor bus width type:

bw8 *// 8 bit bus*
bw16 *// 16 bit bus*
bw32 *// 32 bit bus*
bw64 *// 64 bit bus*
(i.e. any bus widths native to the implementation)

4.4.3.1.2.1 Bus Connection Parameters

The possible device register to bus connections can be completely specified using only two parameters:

- A bus parameter, which specifies the access relationships between the device data bus and the processor data bus
- A multi-addressing and endian parameter, which specifies the access relationships between the logical device register and the device data bus

For example, a possible definition of general device register connection types might be:

```
enum bus_t    { bw8 = 1, bw16 = 2, bw32 = 4, bw64 = 8 };
enum device_t { device8, device16,  device16l,
                device16h, device32, device32l, device32h, device64 };
```

For another example, an implementation for a given processor architecture may only support a subset of the device register connection types. Possible device register connections with the processor H8/300H (supporting only an 8-bit and a 16-bit processor data bus):

```
enum bus_t    { bw8 = 1, bw16 = 2 };
enum device_t { device8, device16, device16l, device16h };
```

4.4.3.1.2.2 Detection of Read / Write Violations in Device Registers

The *access-specifications* can specify a *limitation* parameter, which makes it possible to detect illegal use of a device register at compile-time.

The minimal parameter set for a read / write limitation specification would be:

- Defined as Read-Modify-Write register (behaves like a RAM cell)
- Defined as Read and Write register (read value may be different from write value)
- Defined as Write-Only register
- Defined as Read-Only register

Table: Allowed operations on different device register types:

	iowr	**iord**	**Ioor**	**ioand**	**ioxor**
Read-Modify-Write *rmw_e*	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*
Read-and-Write *rw_e*	*Yes*	*Yes*	*No*	*No*	*No*
Write-Only *wo_e*	*Yes*	*No*	*No*	*No*	*No*
Read-Only *ro_e*	*No*	*Yes*	*No*	*No*	*No*

The "not-allowed" cases should generate some kind of error message at compile-time. With a template implementation of `<iohw.h>`, the compiler will typically diagnose that no matching *function-template* can be found for the "not-allowed" cases.

For example:

```
// --- part of the <iohw.h> header
//
// Define a type to validate device register access
enum rw_t          // Access mode type
{
    rmw_e,         // Read-Modify-Write access
    rw_e,          // Read-and-Write access
    wo_e,          // Write-Only access
    ro_e           // Read-Only access
};

// Include 'exact-width' integer types (defined in the header
// 'stdint.h' in C)
#include <stdint.h>  // Or possibly <cstdint>27

// Define access_spec template for direct addressing
template <class T, rw_t access, device_t devicetype,
          address_t address, bus_t buswidth>
    class IO_MM { };

// --- part of the "iohw_ta.h" header
//
// User declaration of I/O registers in platform
typedef IO_MM <uint8_t, wo_e,  device8, 10200, bw8>  WR_PORT;
typedef IO_MM <uint8_t, ro_e,  device8, 20400, bw8>  RD_PORT;
typedef IO_MM <uint8_t, rmw_e, device8, 20800, bw8>  RDWR_PORT;

// --- portable user code
uint8_t myval;
myval = iord(RD_PORT);        // ok
myval += iord(RDWR_PORT);     // ok
iowr(WR_PORT,myval);          // ok
iowr(RDWR_PORT,0x45);         // ok

myval = iord(WR_PORT);        // Illegal, generate compile-time error
iowr(RD_PORT,0x55);           // Illegal, generate compile-time error
```

[27] ISO C++ was ratified in 1997. At that time, the header file `<stdint.h>` was not present in ISO C, and was added to ISO C in 1999. The naming convention used for C headers by ISO C++ would result in this being known as `<cstdint>`. The addition of this header is currently informally proposed to the C++ Stnadards committee as a library extension.

4.4.3.1.2.3 *access-specifications* for Different Processor Busses

An implementation shall define at least one access method for each processor addressing range. If the processor architecture has multiple different addressing ranges (i.e. it requires different sets of machine instructions for the different busses), each addressing range shall have its own set of *access-specification*s.

For example, on the 80x86 family, an implementation must define at least two sets of access methods; one for the memory-mapped range, and another for the I/O mapped range:

```
typedef uint32_t address_t;   // Memory-mapped address range
typedef uint16_t io_addr_t;   // IO-mapped address range

template <class T, rw_t access, device_t devicetype,
          address_t address, bus_t buswidth>
    class IO_MM { };
template <class T, rw_t access, device_t devicetype,
          io_addr_t address, bus_t buswidth>
    class IO_IOM { };
```

4.4.3.1.2.4 *access-specifications* for Different Access Methods

If several different access methods are supported for a given address range, then an *access-specification* must exist for each access method.

For example:

```
// Define types used in access_spec declarations
typedef uint32_t address_t;      // Memory mapped address range
typedef uint8_t  sub_address_t;  // Sub address on indexed bus
typedef uint16_t io_addr_t;      // User device driver address
typedef uint8_t  bit_pos_t;      // Bit position in register

// Define access_spec template for direct addressing
template <class T, rw_t access, device_t devicetype,
          address_t address, bus_t buswidth>
    class IO_MM { };

// Define access_spec template for addressing via base register
template <class T, rw_t access, device_t devicetype,
          address_t* base, address_t offset, bus_t buswidth>
    class IO_MM_BASE { };

// Define access_spec template for indexed bus addressing
template <class T, rw_t access, device_t devicetype,
          address_t address, sub_address_t idx, bus_t buswidth>
    class IO_MM_IDX { };

// Define access_spec for user-supplied access driver functions
template<class T, rw_t access, io_addr_t address,
         T iord( io_addr_t address),
                  void iowr( io_addr_t address, T val)>
    class IO_MM_DRV { };

// Define access_spec for direct addressing of bit in register
template<class T, rw_t access, device_t devicetype,
         address_t address, bit_pos_t bitpos, bus_t buswidth>
    class IO_MM_BIT { };
```

4.4.3.1.2.5 Optimisation Possibilities for Typical Implementations

4.4.3.1.2.5.1 Pre-Calculation of Constant Expressions

A high performance compiler would resolve all constant expressions at compile-time. Using `inline` functions, both interleave factors and constant buffer indices would be folded into the address value(s) used in the machine code.

Therefore, the following two write statements would result in exactly the same machine code:

```
iowr(PORT1,0x33);
iowrbuf(PORT1, 0, 0x33);
```

An implementation can take advantage of this, because the number of hardware register functions that have to be implemented can be reduced with no efficiency penalty using simple delegation, using inline-functions or *function-templates* such as:

```
template <class access_spec>
inline void iowr(typename access_spec::value_type val)
{ iowrbuf<access_spec>(0,(val)); }
```

4.4.3.1.2.5.2 Multi-Addressing and Endianess

Typical candidates for platform dependent optimisations are hardware *iohw* functions for the multi-addressing cases (*logical device register* width > *device bus* width) where the width of the *device data bus* matches the width of the *processor data bus*; e.g. the combinations of:

- (*device16h* or *device16l*) and *bw16*
- (*device32h* or *device32l*) and *bw32*

In these cases, multi-byte access can often use data types that are directly supported by the processor for either the LSB or MSB endianess functions. The other endianess functions can often be implemented efficiently using one load or store operation, plus one or more register swap operations.

4.5 Example of a Full Device Driver Using the C++ Interface

Detlef: This will be just a piece of code with some comments.

Appendix A: Bibliography

These references may serve as a starting point for finding more information about programming for performance.

Alexander, Rene, and Graham Bensley
C++ Footprint and Performance Optimization
Sams Publishing, 2000

> More general than the Bulka-Mayhew book, and omits any mention of the containers and algorithms in the C++ Standard Library.

Bentley, Jon Louis
Writing Efficient Programs
Prentice-Hall, Inc., 1982

> Unfortunately out of print, but a classic catalogue of techniques that can be used to optimise the space and time consumed by an application (often by trading one resource to minimise use of the other). Because this book predates the public release of C++, code examples are given in Pascal.
>
> *"The rules that we will study increase efficiency by making changes to a program that often decrease program clarity, modularity, and robustness. When this coding style is applied indiscriminately throughout a large system (as it often has been), it usually increases efficiency slightly but leads to late software that is full of bugs and impossible to maintain. For these reasons, techniques at this level have earned the name of "hacks".... But writing efficient code need not remain the domain of hackers. The purpose of this book is to present work at this level as a set of engineering techniques."*

Bentley, John
Programming Pearls, 2nd ed.
Addison-Wesley, 2000

Bulka, Dov, and David Mayhew
Efficient C++: Performance Programming Techniques
Addison-Wesley, 2000

> Contains many specific low-level techniques for improving time performance, with measurements to illustrate their effectiveness.
>
> *"If used properly, C++ can yield software systems exhibiting not just acceptable performance, but superior software performance."*

Cusumano, Michael A., and David B. Yoffie
What Netscape Learned from Cross-Platform Software Development
Communications of the ACM, October 1999.

Faster run-time performance brings commercial advantage, sometimes enough to outweigh other considerations such as portability and maintainability (an argument also advanced in the Bulka-Mayhew book).

Embedded C++ Technical Committee
Embedded C++ Language Specification, Rationale, & Programming Guidelines

`http://www.caravan.net/ec2plus`

EC++ is a subset of Standard C++ that excludes some significant features of the C++ programming language, such as:

- exception handling (EH)
- run-time type identification (RTTI)
- templates
- multiple-inheritance (MI)
- namespaces

Glass, Robert L
Software Runaways: Lessons Learned from Massive Software Project Failures
Prentice Hall PTR, 1998.

Written from a management perspective rather than a technical one, this book makes the point that a major reason why some software projects have been classified as massive failures is for failing to meet their requirements for performance.

"Of all the technology problems noted earlier, the most dominant one in our own findings in this book is that performance is a frequent cause of failure. A fairly large number of our runaway projects were real-time in nature, and it was not uncommon to find that the project could not achieve the response times and/or functional performance times demanded by the original requirements."

Gorlen, Keith, et al.
Data Abstraction and Object Oriented Programming in C++
NIH 1990

Based on the Smalltalk model of object orientation, the "NIH Class Library" also known as the "OOPS Library" was one of the earliest Object Oriented libraries for C++. As there were no "standard" classes in the early days of C++, and because the NIHCL was freely usable having been funded by the US Government, it had a lot of influence on design styles in C++ in subsequent years.

Henrikson, Mats, and Erik Nyquist.
Industrial Strength C++: Rules and Recommendations
Prentice Hall PTR, 1997.

Coding standards for C++, with some discussion on performance aspects that influenced them

Knuth, Donald E.
The Art of Computer Programming
Addison-Wesley

Vol. 1, Fundamental Algorithms	[1997]
Vol. 2, Seminumerical Algorithms	[1998]
Vol. 3, Sorting and Searching	[1998]

The definitive work on issues of algorithmic efficiency.

Koenig, A., and B. Stroustrup
Exception Handling for C++ (revised)
Proceedings of the 1990 Usenix C++ Conference, pp149-176, San Francisco, April 1990.

This paper discusses the two approaches to low-overhead exception handling.

Koenig, Andrew, and Barbara E. Moo
Performance: Myths, Measurements, and Morals
The Journal of Object-Oriented Programming

Part 1:	Myths	[Oct '99]
Part 2:	Even Easy Measurements Are Hard	[Nov/Dec '99]
Part 3:	Quadratic Behavior Will Get You If You Don't Watch Out	[Jan '00]
Part 4:	How Might We Speed Up a Simple Program	[Feb '00]
Part 5:	How Not to Measure Execution Time	[Mar/Apr '00]
Part 6:	Useful Measurements—Finally	[May '00]
Part 7:	Detailed Measurements of a Small Program	[Jun '00]
Part 8:	Experiments in Optimization	[Jul/Aug '00]
Part 9:	Optimizations and Anomalies	[Sep '00]
Part 10:	Morals	[Oct '00]

Measuring the run-time performance of a program can be surprisingly difficult, because of the interaction of many factors.

"The most important way to obtain good performance is to use good algorithms."

Lajoie, Joseé
"Exception Handling: Behind the Scenes."
(Included in **C++ Gems**, edited by Stanley B. Lippman)
SIGS Reference Library, 1996

A brief overview of the C++ language features, which support exception handling, and of the underlying mechanisms necessary to support these features.

Lakos, John
Large-Scale C++ Software Design
Addison-Wesley, 1996

Scalability is the main focus of this book, but scaling up to large systems inevitably requires performance issues to be addressed. This book predates the extensive use of templates in the Standard Library.

Levine, John R.
Linkers & Loaders
Morgan Kaufmann Publishers, 2000

Lippman, Stan
Inside the C++ Object Model

Explains typical implementations and overheads of various C++ language features, such as multiple inheritance and virtual functions. A good in-depth look at the internals of typical implementations.

Meyers, Scott
Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library.
Addison-Wesley 2001

In keeping with the philosophy of the Standard Library, this book carefully documents the performance implications of different choices in design and coding, such as whether to use `map::operator[]` or `map::insert`.

"The fact that function pointer parameters inhibit inlining explains an observation that long-time C programmers often find hard to believe: C++'s `sort` *virtually always embarrasses C's* `qsort` *when it comes to speed. Sure, C++ has function and class templates to instantiate and funny-looking* `operator()` *functions to invoke while C makes a simple function call, but all that C++ "overhead" is absorbed during compilation... It's easy to verify that when comparing function objects and real functions as algorithm parameters, there's an abstraction* ***bonus****."*

Mitchell, Mark
Type-Based Alias Analysis
Dr. Dobbs' Journal, October 2000.

> Some techniques for writing source code that is easier for a compiler to optimise.
>
> *"Although C++ is often criticized as being too slow for high-performance applications, ... C++ can actually enable compilers to create code that is even faster than the C equivalent."*

Musser, David R., Gillmer J. Derge, and Atul Saini
STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library
Addison-Wesley, 2001.

> Among the tutorial material and example code is a chapter describing a class framework for timing generic algorithms.

Prechelt, Lutz
Technical opinion: comparing Java vs. C/C++ efficiency differences to interpersonal differences
Communications of the ACM, October 1999.

> This article compares the memory footprint and run-time performance of 40 implementations of the same program, written in C++, C, and Java. The difference between individual programmers was more significant than the difference between languages.
>
> *"The importance of an efficient technical infrastructure (such as language/compiler, operating system, or even hardware) is often vastly overestimated compared to the importance of a good program design and an economical programming style."*

Saks, Dan
C++ Theory and Practice
C/C++ Users Journal

Standard C++ as a High-Level Language? [Nov '99]
Replacing Character Arrays with Strings, Part 1 [Jan '00]
Replacing Character Arrays with Strings, Part 2 [Feb '0]

These articles are part of a series on migrating a C program to use the greater abstraction and encapsulation available in C++. The run-time and executable size are measured as more C++ features are added, such as Standard strings, *IOStreams*, and containers.

"A seemingly small change in a string algorithm [such as reserving space for string data, or erasing the data as an additional preliminary step,] might produce a surprisingly large change in program execution time."

The conclusion is that you should "program at the highest level of abstraction that you can afford".

Schilling, Jonathan
Optimizing Away C++ Exception Handling
ACM SIGPLAN Notices, August 1998, also at

`http://www.ocston.org/~jls/ehopt.html`

This article discusses ways to measure the overhead, if any, of the exception handling mechanisms. A common implementation of EH incurs no run-time penalty unless an exception is actually thrown, but at a cost of greater static data space and some interference with compiler optimisations. By identifying sections of code in which exceptions cannot possibly be thrown, these costs can be reduced.

This optimization produces modest but useful gains on some existing C++ code, but produces very significant size and speed gains on code that uses empty exception specifications, avoiding otherwise serious performance losses.

Stepanov, Alex
Abstraction Penalty Benchmark
http://www.kai.com/C_plus_plus/benchmarks/_index.html

A brief discussion and downloadable code for this benchmark, mentioned in 2.3.1

Stroustrup, Bjarne
The C++ Programming Language, 3rd Edition
Addison-Wesley, 1998

This definitive work from the language's creator has been extensively revised to present Standard C++.

Stroustrup, Bjarne
The Design and Evolution of C++
Addison-Wesley, 1994

The creator of C++ discusses the design objectives that shaped the development of the language, especially the need for efficiency.

“The immediate cause for the inclusion of inline functions ... was a project that couldn't afford function call overhead for some classes involved in real-time processing. For classes to be useful in that application, crossing the protection barrier had to be free. [...]

Over the years, considerations along these lines grew into the C++ rule that it was not sufficient to provide a feature, it had to be provided in an affordable form. Most definitely, affordable *was seen as meaning "affordable on hardware common among developers" as opposed to "affordable to researchers with high-end equipment" or "affordable in a couple of years when hardware will be cheaper.”*

Stroustrup, Bjarne
Learning Standard C++ as a New Language
C/C++ Users Journal, May 1999

`http://www.research.att.com/~bs/papers.html`

`http://www.research.att.com/~bs/cuj_code.html`

This paper compares a few examples of simple C++ programs written in a modern style using the standard library to traditional C-style solutions. It argues briefly that lessons from these simple examples are relevant to large programs. More generally, it argues for a use of C++ as a higher-level language that relies on abstraction to provide elegance without loss of efficiency compared to lower-level styles.

"I was appalled to find examples where my test programs ran twice as fast in the C++ style compared to the C style on one system and only half as fast on another. ... Better-optimized libraries may be the easiest way to improve both the perceived and actual performance of Standard C++. Compiler implementers work hard to eliminate minor performance penalties compared with other compilers. I conjecture that the scope for improvements is larger in the standard library implementations."

Sutter, Herb
Exceptional C++
Addison-Wesley, 2000.

This book includes a long discussion on minimizing compile-time dependencies using compiler firewalls (the pimpl idiom), and how to compensate for the space and run-time consequences.

Veldhuizen, Todd
Five compilation models for C++ templates
Proceedings of the 2000 Workshop on C++ Template Programming

`http://www.oonumerics.org/tmpw00`

This paper describes a work in progress on a new C++ compiler. Type analysis is removed from the compiler and replaced with a type system library, which is treated as source code by the compiler.

"By making simple changes to the behavior of the partial evaluator, a wide range of compilation models is achieved, each with a distinct trade-off of compile-time, code size, and execution speed. ... This approach may solve several serious problems in compiling C++: it achieves separate compilation of templates, allows template code to be distributed in binary form by deferring template instantiation until run-time, and reduces the code bloat associated with templates."

Williams, Stephen
Embedded Programming with C++
Originally published in the Proceedings of the Third USENIX Conference on Object-Oriented Technologies and Systems, 1997

`http://www.usenix.org/publications/library/proceedings\`
`/coots97/williams.html`

Describes experience in programming board-level components in C++, including a library of minimal run-time support functions portable to any board.

We to this day face people telling us that C++ generates inefficient code that cannot possibly be practical for embedded systems where speed matters. The criticism that C++ leads to bad executable code is ridiculous, but at the same time accurate. Poor style or habits can in fact lead to awful results. On the other hand, a skilled C++ programmer can write programs that match or exceed the quality of equivalent C programs written by equally skilled C programmers.

The development cycle of embedded software does not easily lend itself to the trial-and-error style of programming and debugging, so a stubborn C++ compiler that catches as many errors as possible at compile-time significantly reduces the dependence on run-time debugging, executable run-time support and compile/download/test cycles.

This saves untold hours at the test bench, not to mention strain on PROM sockets.

more possible biblio refs, still being checked out:

http://extreme.indiana.edu/~tveldhui/papers/

http://www.compilerconnection.com/topics/topics.htm

http://liinwww.ira.uka.de/searchbib/index
c++ and (efficiency or performance)

Vendors of development tools often provide guidance on programming for maximum performance. Here are some of the documents available:

Hewlett-Packard Corp.
CXperf User's Guide

`http://docs.hp.com/hpux/onlinedocs/B6323-96001/B6323-96001.html`

This guide describes the CXperf Performance Analyzer, an interactive run-time performance analysis tool for programs compiled with HP ANSI C (c89), ANSI C++ (aCC), Fortran 90 (f90), and HP Parallel 32-bit Fortran 77 (f77) compilers. This guide helps you prepare your programs for profiling, run the programs, and analyze the resulting performance data.

IBM
AIX Versions 3.2 and 4 Performance Tuning Guide, 5th Edition (April 1996)

```
http://www.rs6000.ibm.com/doc_link/en_US\
                                /a_doc_lib/aixbman/prftungd/toc.htm
```

An extensive discussion of performance issues in many areas, such as CPU use, disk I/O, and memory management, and even the performance effects of shared libraries. It discusses AIX tools available to measure performance, and the compiler options, which can be used to optimise an application for space or time. The chapter "Design and Implementation of Efficient Programs"

```
http://www.rs6000.ibm.com/doc_link/en_US\
                          /a_doc_lib/aixbman/prftungd/desnimpl.htm
```

includes low-level recommendations such as these:

"Whenever possible, use int instead of char or short. In most cases, char and short data items take more instructions to manipulate. The extra instructions cost time, and, except in large arrays, any space that is saved by using the smaller data types is more than offset by the increased size of the executable. If you have to use a char, make it unsigned, if possible. A signed char takes

another two instructions more than an unsigned char each time the variable is loaded into a register."

Wind River Systems
Advanced Compiler Optimization Techniques

`http://wrs.com/products/html/optimization_wp.html`

This technical white paper discusses techniques for compiler optimizations in general, and more specifically those provided by the Wind River Systems "Diab" C++ compiler for embedded program development.

Appendix B: Timing Code

```
/*

	Simple/naive measurements to give a rough idea of the relative
	cost of facilities related to OOP.

	I think this could be fooled/foiled by clever optimizers and by
	cache effects.

	Run at least three times to ensure that results are repeatable.

	Tests:

		virtual function
		global function called indirectly
		nonvirtual member function
		global function
		inline member function
		macro
		1st branch of MI
		2nd branch of MI
		call through virtual base
		call of virtual base function

		dynamic cast
		two-level dynamic cast
		typeid()

		call through pointer to member

		call-by-reference
		call-by-value

		pass as pointer to function
		pass as function object

	not yet:

		co-variant return


	The cost of the loop is not measurable at this precision:
	see inline tests

	By default do 1000000 iterations to cout

	1st optional argument: number of iterations
	2nd optional argument: target file name
*/

//int body(int i) { return i*(i+1)*(i+2); }

	class X {
		int x;
```

```
		static int st;
	public:
		virtual void f(int a);
		void g(int a);
		static void h(int a);
		void k(int i) { x+=i; } // inline
	};

	struct S {
		int x;
	};

	int glob = 0;

	void f(S* p, int a);
	void g(S* p, int a);
	void h(int a);
	typedef void (*PF)(S* p, int a);
	PF p[10] = { g , f };
//	inline void k(S* p, i) { p->x+=i; }
	#define K(p,i)  ((p)->x+=(i))

struct T {
	const char* s;
	double t;

	T(const char* ss, double tt) : s(ss), t(tt) { }
	T() : s(0), t(0) { }
};

struct A {
	int x;
	virtual void f(int) = 0;
	void g(int);
};

struct B {
	int xx;
	virtual void ff(int) = 0;
	void gg(int);
};

struct C : A, B {
	void f(int);
	void ff(int);
};


struct CC : A, B {
	void f(int);
	void ff(int);

};
void A::g(int i) { x+=i; }
void B::gg(int i) { xx+=i; }
void C::f(int i) { x+=i; }
void C::ff(int i) { xx+=i; }
```

```
void CC::f(int i) { x+=i; }
void CC::ff(int i) { xx+=i; }

template<class T, class T2> inline T* cast(T* p, T2* q)
{
	glob++; return dynamic_cast<T*>(q);
}

struct C2 : virtual A { // note: virtual base
};

struct C3 : virtual A {
};

struct D : C2, C3 {	// note: virtual base
	void f(int);
};

void D::f(int i) { x+=i; }

struct P { int x,y; };

void by_ref(P& a) { a.x++; a.y++; }
void by_val(P a) { a.x++; a.y++; }

template<class F, class V> inline void oper(F f, V val) { f(val); }

struct FO {
	void operator()(int i) { glob+=i; }
};



#include<stdlib.h>
#include<iostream>
#include<fstream>
#include<time.h>
#include<vector>
#include<typeinfo>
using namespace std;

template<class T> inline T* ti(T* p)
{
	if(typeid(p)==typeid(int*)) p++; return p;
}

int main(int argc, char* argv[])
{
	int i;	// loop variable here for the benefit of non-
conforming compilers

	int n = (1<argc) ? atoi(argv[1]) : 10000000;	// number of
iterations

	ofstream target;
	ostream* op = &cout;
	if (2<argc) {		// place output in file
```

```
		target.open(argv[2]);
		op = &target;
	}
	ostream& out = *op;

	// output command for documentation:
	for (i = 0; i<argc; ++i) out << argv[i] << " ";
	out << endl;

	X* px = new X;
	X x;
	S* ps = new S;
	S s;

	vector<T> v;

	clock_t t = clock();
	if (t == clock_t(-1)) {
		cerr << "sorry, no clock\n";
		exit(1);
	}


	for (i = 0; i<n; i++) px->f(1);
	v.push_back(T("virtual px->f(1)            ",clock()-t));

	t = clock();
	for (i = 0; i<n; i++) p[1](ps,1);
	v.push_back(T("ptr-to-fct p[1](ps,1)       ",clock()-t));

	t = clock();
	for (i = 0; i<n; i++) x.f(1);
	v.push_back(T("virtual x.f(1)              ",clock()-t));

	t = clock();
	for (i = 0; i<n; i++) p[1](&s,1);
	v.push_back(T("ptr-to-fct p[1](&s,1)       ",clock()-t));

	t = clock();
	for (i = 0; i<n; i++) px->g(1);
	v.push_back(T("member px->g(1)             ",clock()-t));

	t = clock();
	for (i = 0; i<n; i++) g(ps,1);
	v.push_back(T("global g(ps,1)              ",clock()-t));

	t = clock();
	for (i = 0; i<n; i++) x.g(1);
	v.push_back(T("member x.g(1)               ",clock()-t));

	t = clock();
	for (i = 0; i<n; i++) g(&s,1);
	v.push_back(T("global g(&s,1)              ",clock()-t));

	t = clock();
	for (i = 0; i<n; i++) X::h(1);
	v.push_back(T("static X::h(1)              ",clock()-t));
```

```
t = clock();
for (i = 0; i<n; i++) h(1);
v.push_back(T("global h(1)                        ",clock()-t));

t = clock();
for (i = 0; i<n; i++) px->k(1);
v.push_back(T("inline px->k(1)                    ",clock()-t));

t = clock();
for (i = 0; i<n; i++) K(ps,1);
v.push_back(T("macro K(ps,1)                      ",clock()-t));

t = clock();
for (i = 0; i<n; i++) x.k(1);
v.push_back(T("inline x.k(1)                      ",clock()-t));

t = clock();
for (i = 0; i<n; i++) K(&s,1);
v.push_back(T("macro K(&s,1)                      ",clock()-t));

C* pc = new C;
A* pa = pc;
B* pb = pc;

t = clock();
for (i = 0; i<n; i++) pc->g(i);
v.push_back(T("base1 member pc->g(i)              ",clock()-t));

t = clock();
for (i = 0; i<n; i++) pc->gg(i);
v.push_back(T("base2 member pc->gg(i)             ",clock()-t));

t = clock();
for (i = 0; i<n; i++) pa->f(i);
v.push_back(T("base1 virtual pa->f(i)             ",clock()-t));

t = clock();
for (i = 0; i<n; i++) pb->ff(i);
v.push_back(T("base2 virtual pb->ff(i)            ",clock()-t));

t = clock();
for (i = 0; i<n; i++) cast(pa,pc);
v.push_back(T("base1 downcast cast(pa,pc)         ",clock()-t));

t = clock();
for (i = 0; i<n; i++) cast(pb,pc);
v.push_back(T("base2 downcast cast(pb,pc)         ",clock()-t));

t = clock();
for (i = 0; i<n; i++) cast(pc,pa);
v.push_back(T("base1 upcast cast(pc,pa)           ",clock()-t));

t = clock();
for (i = 0; i<n; i++) cast(pc,pb);
v.push_back(T("base2 upcast cast(pc,pb)           ",clock()-t));
```

```
t = clock();
for (i = 0; i<n; i++) cast(pb,pa);
v.push_back(T("base2 crosscast cast(pb,pa) ",clock()-t));

CC* pcc = new CC;
pa = pcc;
pb = pcc;

t = clock();
for (i = 0; i<n; i++) cast(pa,pcc);
v.push_back(T("base1 downcast2 cast(pa,pcc)",clock()-t));

t = clock();
for (i = 0; i<n; i++) cast(pb,pcc);
v.push_back(T("base2 downcast  cast(pb,pcc)",clock()-t));

t = clock();
for (i = 0; i<n; i++) cast(pcc,pa);
v.push_back(T("base1 upcast cast(pcc,pa)   ",clock()-t));

t = clock();
for (i = 0; i<n; i++) cast(pcc,pb);
v.push_back(T("base2 upcast2 cast(pcc,pb)  ",clock()-t));

t = clock();
for (i = 0; i<n; i++) cast(pb,pa);
v.push_back(T("base2 crosscast2 cast(pa,pb)",clock()-t));

t = clock();
for (i = 0; i<n; i++) cast(pa,pb);
v.push_back(T("base1 crosscast2 cast(pb,pa)",clock()-t));


D* pd = new D;
pa = pd;

t = clock();
for (i = 0; i<n; i++) pd->g(i);
v.push_back(T("vbase member pd->gg(i)      ",clock()-t));

t = clock();
for (i = 0; i<n; i++) pa->f(i);
v.push_back(T("vbase virtual pa->f(i)      ",clock()-t));

t = clock();
for (i = 0; i<n; i++) cast(pa,pd);
v.push_back(T("vbase downcast cast(pa,pd)  ",clock()-t));

t = clock();
for (i = 0; i<n; i++) cast(pd,pa);
v.push_back(T("vbase upcast cast(pd,pa)    ",clock()-t));

t = clock();
for (i = 0; i<n; i++) ti(pa);
v.push_back(T("vbase typeid(pa)            ",clock()-t));

t = clock();
```

```
	for (i = 0; i<n; i++) ti(pd);
	v.push_back(T("vbase typeid(pd)            ",clock()-t));

	void (A::* pmf)(int) = &A::f; // virtual

	t = clock();
	for (i = 0; i<n; i++) (pa->*pmf)(i);
	v.push_back(T("pmf virtual (pa->*pmf)(i)   ",clock()-t));

	pmf = &A::g;      // non virtual

	t = clock();
	for (i = 0; i<n; i++) (pa->*pmf)(i);
	v.push_back(T("pmf (pa->*pmf)(i)           ",clock()-t));


	P pp;

	t = clock();
	for (i = 0; i<n; i++) by_ref(pp);
	v.push_back(T("call by_ref(pp)             ",clock()-t));

	t = clock();
	for (i = 0; i<n; i++) by_val(pp);
	v.push_back(T("call by_val(pp)             ",clock()-t));

	FO fct;

	t = clock();
	for (i = 0; i<n; i++) oper(h,glob);
	v.push_back(T("call ptr-to-fct oper(h,glob)",clock()-t));

	t = clock();
	for (i = 0; i<n; i++) oper(fct,glob);
	v.push_back(T("call fct-obj oper(fct,glob) ",clock()-t));


	if (clock() == clock_t(-1)) {
		cerr << "sorry, clock overflow\n";
		exit(2);
	}

	out << "\n";
	for (i = 0; i<v.size(); i++) {
		out << v[i].s << " :\t" <<
v[i].t*(double(1000000)/n)/CLOCKS_PER_SEC << " ms\n";
	}

	if (argc<2) {     // if output is going to cout
		cout << "press any character to finish\n";
		char c;
		cin >> c;   // to placate windows console mode
	}

	return 0;   // shut up noncompliant compilers
}
```

```
int X::st = 0;
void X::f(int a) { x+=a; }
void X::g(int a) { x+=a; }
void X::h(int a) { st+=a; }

void f(S* p, int a) { p->x += a; }
void g(S* p, int a) { p->x += a; }
void h(int a) { glob += a; }
```

Appendix C: Sample C++ Implementation of Device Driver Code

This is *preliminary* example code for the C++-style portable interface to hardware:

```
// proposed definition of <hardware>
// this is definition only

namespace std
{
namespace hardware
{

#include <stdint.h>:

// the definitions of access_types' parameter types
struct hw_base
{
    enum access_mode {random, read_write, write, read};
    enum device_bus {device8, device16, device32, device64};
    enum byte_order {msb_low, msb_high};
    enum processor_bus {bus8, bus16, bus32, bus64};
    // only identifiers shall be present that are supported by
    // the underlying implementation! (Diagnostic required.)

typedef _ul address_type;
};

// access specification types
// only those access specifications shall be present
// in an implementation that are supported by
// the underlying hardware! (Diagnostic required.)

// direct address for memory mapped registers and address know at
compile time
template <typename ValueType,
          hw_base::access_mode mode,
          hw_base::address_type address,
          hw_base::device_bus devWidth,
          hw_base::byte_order endian,
          hw_base::processor_bus nativeWidth>
class direct_address
{
public:
    typedef ValueType value_type;
    typedef _empty_type dynamic_data;
    // probably, we should also provide the mode info, but how?
    enum { access_mode=mode }; // (would a type be a better solution?)

    template <hw_base::address_type other_address> struct rebind
    {
        typedef direct_address<ValueType, mode, other_address,
                               devWidth, endian, nativeWidth> other;
    };
};
```

```
// dynamic address for memory mapped registers and address only know
at run-time
template <typename ValueType,
          hw_base::access_mode mode,
          typename AddressType,  // UDT with implementation defined
semantics
          hw_base::device_bus devWidth,
          hw_base::byte_order endian,
          hw_base::processor_bus nativeWidth>
class dynamic_address
{
public:
    typedef ValueType value_type;
    typedef AddressType dynamic_data;
    enum { access_mode=mode }; // (would a type be a better solution?)
};

// the actual access interface
template <class AccessType, class DynamicData = typename
AccessType::dynamic_data>
class register_access
{
public:
    typedef typename AccessType::value_type value_type;

    register_access();
    // for empty dynamic_data, the following ctor might not be defined
    register_access(typename AccessType::dynamic_data const &);

    operator value_type() const;
    void operator=(value_type val) const;
    void operator|=(value_type val) const;
    void operator&=(value_type val) const;
    void operator^=(value_type val) const;

    struct _ref
    {
        explicit _ref(size_t) : idx(i) {}
        operator value_type() const;
        void operator=(value_type val) const;
        void operator|=(value_type val) const;
        void operator&=(value_type val) const;
        void operator^=(value_type val) const;
    };

    _ref operator[](size_t index) const
    {
        return _ref(index);
    }
};

} // namespace hardware

} // namespace std
```

This is *preliminary* example code for a sample C++ implementation for a simple platform (watch out for line wrapping):

```
// proposed implementation of <hardware>

// The idea of this whole interface is to provide device driver
implementers
// with a means to write a portable device driver for a specific
hardware device (chip).
// If the device is attached to a different CPU, the way to access the
hadware addresses
// might change, but the core of the device driver code should still
be the same.
// To achieve this, the device driver implementer has to provide a
different
// access type, but everything else remains stable.

namespace std
{
namespace hardware
{
// leading underscores in this implementation are there on purpose,
// though they are ugly.
// They make sure here that the use of these names are reserved to
// the library implementer, and this is exactly the role we have here.

// from #include <stdint.h>:
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;
// there is no long long in C++ :-(

typedef unsigned long int _ul;
typedef unsigned char _uc;

// helper class for saving integral values as types
template <_ul val>
struct _Int2Type
{
    enum { value=val };
};

// and to create an integral type for a given sizeof
template <_ul size> struct _uint_type;
template <> struct _uint_type<1> { typedef uint8_t ui_type; };
template <> struct _uint_type<2> { typedef uint16_t ui_type; };
template <> struct _uint_type<4> { typedef uint32_t ui_type; };
#ifdef UINT64_MAX
template <> struct _uint_type<8> { typedef uint64_t ui_type; };
#endif

// and an empty helper class for default DynamicData
struct _EmptyType {};

// the definitions of access_types' parameter types
struct hw_base
{
    // The I/O library <iostream> defines constants as statics.
    // This allows for easier implementation, but has some space
    // and possibly run-time overhead.
    // For performance reasons, here the enum approach is chosen.
    enum access_mode {random, read_write, write, read};
    enum device_bus {device8=1, device16=2, device32=4, device64=8};
```

```
    enum byte_order {msb_low, msb_high};
    enum processor_bus {bus8=1, bus16=2, bus32=4, bus64=8};
    // only identifiers should be present that are supported by
    // the underlying implementation! (Diagnostic required.)

typedef _ul address_type;
};

typedef _Int2Type<hw_base::msb_high> _native_endian;

// helper template to hold the info necessary to calculate the address
offset
template <_ul _valueSize,
          _ul _deviceWidth,
          _ul _procBusWidth,
          class _AddressHolder>
struct _AddressInfo
{
    enum { _registerSize=_valueSize,
           _devWidth=_deviceWidth,
           _nativeWidth=_procBusWidth };
    typedef _AddressHolder _AddressHolder;
};

// access specification types

// direct address for memory mapped registers and address know at
compile time
template <typename _ValueType,
          hw_base::access_mode mode,
          hw_base::address_type address,
          hw_base::device_bus devWidth,
          hw_base::byte_order endian,
          hw_base::processor_bus nativeWidth>
class direct_address
{
public:
    typedef _ValueType value_type;
    typedef _EmptyType dynamic_data;
    enum { access_mode=mode }; // (would a type be a better solution?)

    template <hw_base::address_type other_address> struct rebind
    {
        typedef direct_address<_ValueType, mode, other_address,
devWidth, endian, nativeWidth> other;
    };

    typedef _EmptyType _dynDataHolder;
    // we don't want to spend any space, so all arguments are saved as
types
    typedef _Int2Type<address> _BaseAddressHolder;
    typedef _Int2Type<endian> device_endian;
    typedef _AddressInfo<sizeof(_ValueType), devWidth, nativeWidth,
_BaseAddressHolder> _AddressT;
};

// dynamic address for memory mapped registers and address only know
at run-time
template <typename _ValueType,
          hw_base::access_mode mode,
          typename AddressType,  // UDT with implementation defined
semantics
          hw_base::device_bus devWidth = hw_base::device32,
```

```
           hw_base::byte_order endian = hw_base::msb_high,
           hw_base::processor_bus nativeWidth = hw_base::bus32>
class dynamic_address
{
public:
    typedef _ValueType value_type;
    typedef AddressType dynamic_data;
    enum { access_mode=mode }; // (would a type be a better solution?)

    struct _dynDataHolder
    {
        _dynDataHolder(AddressType const &addr) : value(addr.address)
{}
        _ul value;
    };
    typedef _dynDataHolder _BaseAddressHolder;

    typedef _Int2Type<endian> device_endian;
    typedef _AddressInfo<sizeof(_ValueType), devWidth, nativeWidth,
_BaseAddressHolder> _AddressT;
};

// dynamic address for memory mapped registers and address only know
at run-time
// alternate version using two-element AddressType
template <typename _ValueType,
           hw_base::access_mode mode,
           typename AddressType,  // UDT with implementation defined
semantics
           hw_base::device_bus devWidth = hw_base::device32,
           hw_base::byte_order endian = hw_base::msb_high,
           hw_base::processor_bus nativeWidth = hw_base::bus32>
class dynamic_address_alt
{
public:
    typedef _ValueType value_type;
    typedef AddressType dynamic_data;
    enum { access_mode=mode }; // (would a type be a better solution?)

    struct _dynDataHolder
    {
        _dynDataHolder(AddressType const &addr) : value(addr.segment
<< 16 + addr.offset) {}
        _ul value;
    };
    typedef _dynDataHolder _BaseAddressHolder;

    typedef _Int2Type<endian> device_endian;
    typedef _AddressInfo<sizeof(_ValueType), devWidth, nativeWidth,
_BaseAddressHolder> _AddressT;
};

// a helper class for all provided binary operations
enum _binops { _write_op, _or_op, _and_op, _xor_op };
template <typename int_type, _binops> struct _hwOp;
template <typename int_type> struct _hwOp<int_type, _write_op>
{
    static void f(int_type volatile &lhs, int_type rhs) { lhs = rhs; }
};
template <typename int_type> struct _hwOp<int_type, _or_op>
{
    static void f(int_type volatile &lhs, int_type rhs) { lhs |= rhs;
}
```

```
};
template <typename int_type> struct _hwOp<int_type, _and_op>
{
    static void f(int_type volatile &lhs, int_type rhs) { lhs &= rhs;
}
};
template <typename int_type> struct _hwOp<int_type, _xor_op>
{
    static void f(int_type volatile &lhs, int_type rhs) { lhs ^= rhs;
}
};

// a helper function for the actual address calculation
template <class _AddressInfo>
inline
_ul _addrCalc(_ul idx, typename _AddressInfo::_AddressHolder const
&addr)
{
    return addr.value +
idx*_AddressInfo::_registerSize*_AddressInfo::_nativeWidth/_AddressInf
o::_devWidth;
}

// a helper class to provide all useful (partial) specializations for
register_access
template <typename _ValueType,
          _ul devEndian,
          class _AddressInfo>
struct _AccessHelper
{
    static _ValueType _read(_ul baseIdx, typename
_AddressInfo::_AddressHolder const &);
    template <_binops function> static void _op(_ValueType val, _ul
baseIdx, typename _AddressInfo::_AddressHolder const &);
};
// no definition of the function for the general case:
// all valid cases must be provided as (partial) specializations

// value_size must always be a multiple of dev_width (everything else
doesn't make sense)

// here a specialization where deviceWidth matches nativeWidth and
ValueType
template <typename _ValueType,
          class _AddressHolder>
struct _AccessHelper<_ValueType,
                     _native_endian::value,
                     _AddressInfo<sizeof(_ValueType),
                                  sizeof(_ValueType),
                                  sizeof(_ValueType),
                                  _AddressHolder>
                    >
{
    typedef _AddressInfo<sizeof(_ValueType), sizeof(_ValueType),
sizeof(_ValueType), _AddressHolder> _AddressT;
    static _ValueType _read(_ul baseIdx, _AddressHolder const &addr)
    {
        return *const_cast<_ValueType volatile *>(
                 reinterpret_cast<_ValueType
*>(_addrCalc<_AddressT>(baseIdx, addr)));
    }
    template <_binops function>
```

```
    static void _op(_ValueType val, _ul baseIdx, _AddressHolder const
&addr)
    {
        _hwOp<_ValueType, function>
            ::f(*const_cast<_ValueType volatile *>(
                  reinterpret_cast<_ValueType
*>(_addrCalc<_AddressT>(baseIdx, addr))),
                val);
    }
};

// this is the general implementation for any sizeof(_ValueType)
// and _native_endian
template <typename _ValueType,
          class _AddressInfo>
struct _AccessHelper<_ValueType, _native_endian::value, _AddressInfo>
{
    typedef typename _AddressInfo::_AddressHolder _AddressHolder;
    enum { _wordCount =
_AddressInfo::_registerSize/_AddressInfo::_devWidth,
           _step = _AddressInfo::_nativeWidth };
    typedef typename _uint_type<_AddressInfo::_devWidth>::ui_type
reg_t;
    struct buf_t { reg_t value[_wordCount]; };

    static _ValueType _read(_ul baseIdx, _AddressHolder const &addr)
    {
        buf_t buffer;
        for (_ul idx=0; idx != _wordCount; idx++)
        { // use new style casts
            buffer.value[idx]
                = *const_cast<reg_t volatile *>(
                    reinterpret_cast<reg_t
*>(_addrCalc<_AddressInfo>(baseIdx, addr)
                                           + idx*_step));
        }
        return *((_ValueType *)buffer.value);
    }
    template <_binops function>
    static void _op(_ValueType val, _ul baseIdx, _AddressHolder const
&addr)
    {
        for (_ul idx=0; idx != _wordCount; idx++)
        {
            _hwOp<reg_t, function>
                ::f(*const_cast<reg_t volatile *>(
                      reinterpret_cast<reg_t
*>(_addrCalc<_AddressInfo>(baseIdx, addr)
                                             + idx*_step)),
                    reinterpret_cast<reg_t *>(val)[idx]);
        }
    }
};

template <typename T>
struct _TypeHolder
{
    _TypeHolder(T const &v) : value(v) {}

    T value;
};
// the actual access interface
```

```
template <class _RAType>
inline void operator|=(_RAType const &lhs, typename
_RAType::value_type val)
{
    lhs._op<_or_op>(val);
}

template <class _AcType, class _AddressHolder, typename _IndexType>
class _RAImpl
{
public:
    typedef typename _AcType::value_type value_type;

    static value_type _read(_AddressHolder const &_addr, _IndexType
const &_idx)
    {
        return _AccessHelper<value_type,
                              _AcType::device_endian::value,
                              typename _AcType::_AddressT
                             >::_read(_idx.value, _addr);
    }

    template <_binops function>
    static void _op(_AddressHolder const &_addr, _IndexType const
&_idx, value_type _val)
    {
        _AccessHelper<value_type,
                      _AcType::device_endian::value,
                      typename _AcType::_AddressT
                     >::_op<function>(_val, _idx.value, _addr);
    }
};

template <class _AcType, class _AddressHolder, typename _IndexType>
class _RAInterface
{
public:
    typedef typename _AcType::value_type value_type;
    typedef _RAImpl<_AcType, _AddressHolder, _IndexType> _Impl;

    _RAInterface() : _addr(), _idx() {}
    _RAInterface(typename _AcType::dynamic_data const &_d) :
_addr(_d), _idx() {}
    _RAInterface(_AddressHolder const &_a, _IndexType const &_i) :
_addr(_a), _idx(_i) {}

    operator value_type() const
    {
        return _Impl::_read(_addr, _idx);
    }
    void operator=(value_type val) const
    {
        _Impl::_op<_write_op>(_addr, _idx, val);
    }
    void operator|=(value_type val) const
    {
        _Impl::_op<_or_op>(_addr, _idx, val);
    }
    void operator&=(value_type val) const
    {
        _Impl::_op<_and_op>(_addr, _idx, val);
    }
    void operator^=(value_type val) const
```

```
    {
        _Impl::_op<_xor_op>(_addr, _idx, val);
    }

protected:
    const _AddressHolder _addr;
    const _IndexType _idx;
};

template <class _AcType, class _DynData = typename
_AcType::dynamic_data>
class register_access : public _RAInterface<_AcType, typename
_AcType::_BaseAddressHolder, _Int2Type<0> >
{
    typedef typename _AcType::_BaseAddressHolder _AddressHolder;
    typedef _RAInterface<_AcType, _AddressHolder, _TypeHolder<_ul> >
_RefT;
public:
    register_access(typename _AcType::dynamic_data const &d)
        : _RAInterface<_AcType, typename _AcType::_BaseAddressHolder,
_Int2Type<0> >(d) {}

    _RefT operator[](_ul index) const
    {
        return _RefT(_addr, index);
    }
};

// specialization for no dynamic data
template <class _AcType>
class register_access<_AcType, _EmptyType> : public
_RAInterface<_AcType, typename _AcType::_BaseAddressHolder,
_Int2Type<0> >
{
    typedef typename _AcType::_BaseAddressHolder _AddressHolder;
    typedef _RAInterface<_AcType, _AddressHolder, _TypeHolder<_ul> >
_RefT;
public:
    typedef typename _AcType::value_type value_type;

    register_access() {}

    _RefT operator[](_ul index) const
    {
        return _RefT(_AddressHolder(), index);
    }
};

} // namespace hardware

} // namespace std

namespace
{
// middle layer (access specifications)
using namespace std::hardware;

typedef direct_address<uint32_t, hw_base::random, 0x35800,
hw_base::device16, hw_base::msb_high, hw_base::bus32> PortXY_t;

unsigned long globalBase = 0;
// implementation defined requirements for dynamic_address:
// a public member 'unsigned long address'
```

```
struct DynAddressPortDA
{
    DynAddressPortDA() : address(globalBase+0x120) {}

    unsigned long address;
};
typedef dynamic_address<uint32_t, hw_base::random, DynAddressPortDA>
PortDA_t;

// implementation defined requirements for dynamic_address_alt:
// two public members: 'unsigned short segment' and 'unsigned short
offset'

unsigned short globalDBBase = 0;
struct DynAddressPortDB
{
    DynAddressPortDB() : segment(0xF0), offset(globalDBBase) {}

    unsigned short segment;
    unsigned short offset;
};
typedef dynamic_address_alt<uint32_t, hw_base::random,
DynAddressPortDB> PortDB_t;

} // anonymous namespace

// test
int main()
{
    register_access<PortXY_t> portXY;
    register_access<PortXY_t::rebind<0x36800>::other> portYZ;  // same
device characteristics, other base address

    uint32_t v;

    v = portXY[0];
    portXY[1] = 5;
    v |= portXY[2];

    v = portYZ;
    portYZ ^= v;

    register_access<PortDA_t> portDA = DynAddressPortDA();

    v = portDA;
    portDA[5] = v;

    register_access<PortDB_t> portDB = DynAddressPortDB();

    v = portDB;
    portDB[5] = v;

    return 0;
}
```