Doc. No.: 01-0023/N1309
Date: 18 May 2001
Author: Performance WG

Technical Report on C++ Performance (DRAFT)

Editor's Note: The cover page text needs to be written.

The aim of this report is to give its readers 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 discussed areas where performance issues matters, such as various forms of embedded systems programming and high-performance numerical computation. After that, the main body of the report considered 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 information (RTTI). It is demonstrated that, with the exception of RTTI, current C++ implications 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

Definition of terminology and scope of the report:

- ?? Description of potential resource limitations
- ?? Problems often encountered in resource-limited environments
- ?? Criteria used in the selection of an appropriate programming language

"Performance" has many aspects - execution speed, code size, data size, and memory footprint at runtime, 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 runtime. 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 How do we Characterise Application Areas?

Embedded Systems:

Embedded Systems have many restrictions on memory-size and timing requirements that are more significant than are typical for non-Embedded systems. Some areas of concern the Embedded Systems are 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.

Note (Lois):	*The numbers relate to the C8051 chip family, which has a market share of approximately two-thirds of the embedded controllers in the world (according to Detlef quoting Chris Hills).*

  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 if 4KBytes of RAM, 96KBytes of ROM and 32KBytes EEPROM.

- **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 several kinds to achieve different objectives for 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.

[1] Typical systems during the Year 2000

- **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

Note (Lois):	*The last item is meant to refer to the central CPU that manages a collection of manufacturing machines in a production line. Each machine of course may have its own embedded brain.*

?? **Timing:**

Note (Lois):	*Of course, systems with real-time or hard real-time constraints are not necessarily embedded systems; they may run on hosted environments. Anton (who?) made the comment that timing-critical hard real-time systems are more applicable to industry.* *"Real-Time" refers to a system in which average performance and throughput must meet defined goals, but some variation in performance of individual components can be tolerated.* *"Hard Real-Time" means the **every** operation **must** meet specified timing constraints.*

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

  Examples of applications in this category are:

  - Motor control
  - Engine control – minimum cycle of engine (3ms; 10,000rpm; 4 cyclinders)
  - Hand-held digital VCR
  - Mobile phone
  - CD or DVD player
  - Electronic musical instruments
  - Hard disk controllers
  - Digital television

- **non-critical**

  Examples of applications in this category are:

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

Note (Lois)	*Eliminated table here. Its purpose appeared to be to show a cross-section of applications in terms of both size and timing constraints. I think Anton is putting together a revised version. Maybe it should come last so it can include the large-end applications as well?*

Servers:

For server applications, the performance-critical resources are typically speed (e.g. transactions per second?), and working-set[2] 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.

[Note (Lois):	*If it takes 26 hours to forecast the next 24 hours' weather, there's no point]*

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] the term "working set" refers to the amount of the application which is held in active (not swapped-out virtual) memory at any given time.

2 Overheads – Cost of Using 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 runtime, 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. Some of the language features are complex and are discussed in a section of their own, while other are discussed in the following brief:

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

2.1 Overheads from 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 so as to avoid name clashes.

Namespace qualifiers enable programmers to use shorter identifier name 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, or the names being placed inside a `class` and used in their qualified form. *[for example:*

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

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

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

end example]

With namespaces, the number of characters necessary is similar to the `class` alternative, but unlike the `class` alternative, qualification can be avoided by the use of `using` which moves the unqualified names into the current scope, thus allowing the names to be referenced by their shorter form. This has the effect of actually "reducing" the number of characters in the source program.

2.2 Overheads from 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*[3] *[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 correct[4].

It should be noted that as in C, a cast may create a temporary object of the desired type, so casting can have runtime implications.

[3] Indeed, prototype implementations of the *type conversion operators* were often implemented as *function-templates*.

[4] If the compiler does not provide the *type conversion operators* natively, it is possible to implement them using *function-templates*.

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 runtime if the required conversion involves using RTTI mechanisms *[for example, cross-casting:*

```
class Left  { ... };
class Right { ... };
...
void func ( Left* pL ) {
    Right* pR = dynamic_cast<Right*>( pL );
}
...
class Merged: Left, Right {};
Merged m;
func ( &m );
```

end example]

Note:	*Perhaps some pseudo code would be useful here to show how the compiler transforms the code? How much overhead is a* `dynamic_cast` *in a single-inheritance hierarchy? Or when virtual base classes are involved?*

2.3 Overheads from Inheritance

2.3.1 Overhead examples

?? runtime type identification (RTTI)
?? multiple inheritance
?? virtual template member functions
?? virtual inheritance
?? class hierarchies
?? unnecessary costs for empty base

Editor's Note:	*The cost of empty-bases is not elaborated.*

2.3.2 RTTI overheads

?? Typically, a pointer to a `type_info` object is stored in a class' "virtual table" or `vtbl`. RTTI can only be used with classes that have at least one virtual function[5]. This restriction is the result of a deliberate compromise that minimizes the cost per object[6] necessary for RTTI.

?? One typical implementation costs one static table per class with enough storage for the *class-name* (its `typeid`) plus 20 bytes, with a resulting cost of

[5] This is not an unnatural situation, as the need to determine the dynamic type of an entity is most typical of class hierarchies where it is probable that virtual functions are used; indeed, it is often recommended that polymorphic classes always provide a virtual destructor.

[6] Since a class with a virtual function already has the associated cost of a `vptr` (in a typical implementation), adding RTTI support need have no extra impact on the cost of an instance of that class. Conversely, adding RTTI for non-class data types, and classes with no virtual functions could incur a significant cost to the program and/or instances of that data type.

approximately 40 data bytes times the number of RTTI enabled classes in the application.

?? Often, RTTI is used with `dynamic_cast;` and if `dynamic_cast` is used, the RTTI mechanism is used to determine whether the cast is valid or not. One important thing to note, is that some typical implementations share common mechanisms between RTTI and EH, and the use of RTTI may implicitly require the overheads of Exception Handling to be present also.

?? Whole-Program Analysis (WPA) can help; there is no need to generate RTTI tables for types not tested, and WPA techniques can determine this, with potentially significant reduction in costs and overheads. Tools providing WPA are not yet commonly available, but implementers are encouraged to develop such WPA capable tools.

?? A class without any virtual functions is equivalent to a simple C `struct`.

The size of an object of the class is the sum of the sizes of its data members, (plus any padding required for correct alignment by the implementation).

?? ***Some observations from tests on commonly available implementations***

- Downcasts cost between three and four function calls. This is independent of:
  - ?? whether the class uses single or multiple inheritance
  - ?? which branch of MI
  - ?? the depth of inheritance (MI and SI)
- Cross-casts are more expensive. A cross-cast costs between 6 and 50 times a single function call, depending on the implementation. They vary with how deep you start and finish in the hierarchy. Each level adds about 60% to overhead.

2.3.3 General Overheads from Inheritance

In a typical implementation, data members of a base class occupy space at the beginning of an object of a derived class. This need not cost any more data space than the alternate design of creating a data member of the base class type. In the simplest case, inheritance may save in code size and execution speed, since delegating functionality to a member object requires pass-through functions in the containing class. Calls to non-virtual functions are resolved at compile-time, so there is no runtime penalty from single inheritance.

Indeed, in some cases, an implementation may be able to place new data members of the deriving class into "holes" in the base-class[7]; thus costing less than an equivalent C-style `struct`.

[7] Holes may be present dues to alignment restrictions of the implementation. However, these holes may be of appropriate size and alignment for new data members introduced by the deriving class.

?? **Do virtual functions add overhead?**

Calls to virtual member functions are resolved at runtime, depending on the dynamic type of the object. In a typical implementation, each object in the hierarchy acquires an extra data member, a `vptr`, pointing to a table (the `vtbl`) that lists the appropriate versions of the virtual functions for objects of that class type. So, the cost of virtual functions is an extra data pointer per object, plus a `vtbl` per class[8].

At runtime, there is a cost associated with calling the virtual function by indirection through the `vptr`, indexing into the `vtbl`, and calling the function through a pointer. This cost, in a typical implementation, adds between 3 and 10 instructions per call, versus a direct call to a class-specific function, resolved at compile-time. Alternate mechanisms of determining the appropriate function to call, such as an *if-statement* or a *switch-statement* also have overheads however, and these alternative mechanisms have a comparable cost, while lacking the natural extensibility of a virtual function. If a virtual function is called repeatedly inside a tight loop, a possible "Programmer Directed Optimisation" (PDO) is to determine the runtime type of the object outside the time-critical section, and use class-specific direct calls inside the loop.

Compiler implementations, and especially WPA enabled compilers can sometimes determine the static type of the object, and automatically perform this optimisation. However, PDOs can make use of knowledge about the program that a compiler will never be able to determine.

?? **"The principal disadvantage of virtual functions is that they prevent the compiler from inlining code, since the type of the object won't be known until runtime."**

This argument is often levelled at virtual functions. Typically, virtual functions are not also declared inline, due to the fact that a virtual function is normally called indirectly, requiring that the function be instantiated.

However, whether the programmer directs the implementation by qualifying the name of a virtual function, or the implementation determines the static type of the function to be called by other means, it becomes possible for the virtual function to be called statically, and hence inlined.

Contrary to popular belief, virtual and inline need not be considered mutually exclusive.

?? ***Some observations from tests on commonly available implementations***

- Static function calls with no arguments are slightly faster than ordinary member functions (less than 25%) with no arguments *(the member function has an implicit object pointer)*.

[8] The cost of the `vtbl` is typically a static data cost. On some older implementations, there may be an instance of the `vtbl` for each translation unit in which an instance of the associated data type is created or destroyed. Modern implementations typically have only one instance of the `vtbl` per program.

- Function calls are cheaper than they used to be (compared to inline).
- These figures have improved a lot from what they were a few years ago.
- People should make "less" use of explicit inlining these days, as modern compilers have got a lot better at determining when inlining is appropriate.
- Locality – forcing code out of cache. A virtual function call (through a pointer) had overhead of 20% compared to a plain function call[9]. Maybe even 30% if you do a lot of other work in the loop and in the function call and then factor it out. But still no overhead specific to Multiple-Inheritance.

2.3.4 Overheads from Multiple-Inheritance

?? Properly implemented, multiple-inheritance should have very little extra cost over single inheritance. Such small costs can also be restricted to only a part of the MI graph; typically, the left most branch having the same cost as SI (no adjustment), with other branches incurring a simple "offset adjustment" to the object pointer.

?? There is an "offset adjustment" in virtual calls to ensure that the `this` pointer passed to the called function is correct. Typical implementations use a "thunk" 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.

?? ***Some observations from tests on commonly available implementations***

- No significant differences in runtime speed between ordinary member function calls, virtual function calls, and virtual function calls among different branches of multiply inherited (MI) classes.
- The difference diminishes with the number of arguments being passed to the function; the associated cost of the call becomes proportionally smaller as the number of arguments increase.

2.3.5 Overheads from Virtual-Inheritance

Virtual base classes add additional overhead. The "adjustment" for the branch in a MI 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 than for the MI adjustment.

[9] This is sometimes the result of "block level linking" that attempt to place called code physically closer in memory to the code that makes the call, a technique that is defeated by indirect calls. A cache "miss" can result in costly reloads or even operating system intervention.

2.3.6 Overheads from Virtual Functions of *class-template*s

Virtual functions of a *class-template* can incur an overhead:

?? 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 PDO suggestion is to use non-template helper functions, and describe the template implementation in terms of these helper functions.

2.4 Overheads from Exception Handling

2.4.1 Myths and Realities of Exception Handling Overheads

2.4.1.1 Preliminary Remarks

Exception Handling provides a systematic and robust approach to error handling.

Editor's Note:	*Error Handling and Exception Handling are not the same thing. Errors are a normal occurrence in a program exceptions are not.*
Note (Lois):	*The sort of errors that can normally be expected to occur – file not found, erroneous input, etc.—may more aptly be described as 'status' rather than 'errors'. Exceptions are highly unusual, and often unrecoverable-from problems that arise in exceptional conditions – out of memory, network failure, etc. Often the only reasonable response is to exit gracefully, or at least roll back to an earlier state*

The traditional C style of indicating runtime problems is to return an error code. This error code must be checked each time the function is invoked, and this check is quite often ignored or forgotten. EH isolates the rare problem-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 EH, once a problem is identified, it can't be ignored - failure to catch and handle an exception results in program termination.

Early implementations of Exception Handling resulted in sizable increases in code size. This led some programmers to avoid it and compiler vendors to provide switches to suppress the feature. In some embedded and resource-constrained environments, EH was deliberately excluded.

It is difficult to discuss EH overheads without a rough idea about possible implementations.

Presuming that exceptions are not the norm, we need to distinguish:

?? **Try overhead**: data and code associated with setting up each *try-block* or *catch-clause* (i.e. getting ready for catching exceptions that may never occur) - this is true overhead.

?? **Regular function overhead**: data and code associated with the normal execution of functions that do not specify any exception related feature (i.e. recompiling pre-EH code, thus breaking the "pay as you go" principle) – this is true overhead.

?? **Throw cost**: data and code associated with actually throwing an exception. This can hardly be regarded as an overhead! But different implementations will have different costs, the relative value or impact of which depends on the problem domain.

2.4.1.2 Compile-Time Overhead

?? Compilation is more difficult, depending on the complexity of the implementation.

?? Some compile-time optimisations may become trickier (or even impossible?):

- *we need examples*

Editor's Note: This section is never developed, should we remove it?

2.4.2 Exception Handling Issues Common to all Implementations

?? ***try-block*** Establishes the context for associated *catch-clauses*

?? ***catch-clause*** The EH implementation must provide some runtime type-information 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*. Since there is no declaration for the exception object, some "Magic 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 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 the "Magic 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.

2.4.3 Implementation Strategies

Implementations vary in how costs are allocated across these elements.

The two main strategies are the “dynamic” approach often implemented using the `setjmp` family of functions and “static” approach that uses compiler generated static tables.

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

2.4.3.1 The "dynamic" Approach.

Implementations using this approach have to dynamically maintain auxiliary data-structures to manage the capture and transfer of the execution contexts, and the dynamic maintenance of data-structures involved in tracking the objects that need to be unwound in the event of an exception.

?? ***try-block*** Save the execution environment and reference to catch code on EH stack at *try-block* entry (by calling `setjmp` or equivalent).

?? ***Automatic and temporary objects with non-trivial destructors*** Push each constructed object, with the address of its destructor onto a stack for later destruction. Pop them upon destruction. Typical implementations use a linked list structure for the stack.

?? ***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 destructed.

?? ***throw-expression*** After the *catch-clause* has been found, pop objects from the stack, invoking their destructors, until all objects between the *throw-expression* and the associated *catch-clause* are removed from the stack.

Restore execution environment of associated *catch-clause* (by calling `longjmp` or equivalent).

2.4.3.1.1 Space Overhead

?? No EH cost is associated with an object, so object size is unaffected
?? EH implies a form of RTTI, implying some increase in code and data size
?? The `setjmp` model implies code generation for try/catch
?? The `setjmp` model implies *dynamic* data structures to store the `jmp_buf` environments and their mapping to *catch-clause*s
?? The `setjmp` model implies the registration of local objects to be destroyed
?? A cost is associated with checking the *throw-specification*s of the functions that are called

2.4.3.1.2 Time Overhead

?? On entry to each *try-block*
- commit changes to variables enclosing the *try-block*
- stack the `jmp_buf` 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 (destruct 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:
    - destruct the registered local objects
    - check the *exception-specification*s of the functions called in-between
    - use the associated `jmp_buf` to `longjmp` to the execution context of the *catch-clause*

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

Advantages of this method are that it is relatively simple, portable, and compatible with implementations that translate C++ to C or another language.

Disadvantages are that the stack space and runtime costs for *try-block* entry, and for the bookkeeping of automatic, temporary and partially constructed objects as the EH stack is modified tends to be quite high.

One vendor reports speed impact of about 6% for a C++ to ANSI C translator. Another vendor reports that speed and stack space impacts can be greatly reduced by fine-tuning the code for saving the execution environment and doing object bookkeeping

Editor's Note: How should we include information such as the comment above?

2.4.3.2 The "static" 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 runtime 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. Tables can be placed in ROM, and on hosted systems with Virtual Memory, can remain swapped out until an exception is actually thrown.
- ***Automatic and temporary objects with non-trivial destructors*** No runtime costs associated with normal execution. Only in the event of an exception is it necessary to intrude on normal execution.
- ***Construction of objects with non-trivial destructors*** No runtime costs – see previous bullet.
- ***throw-expression*** The statically generated tables are used to locate matching handlers and intervening objects needing destruction. Again, no runtime costs are associated with normal execution.

2.4.3.2.1 Space Overhead

- ?? No EH cost is associated with an object, so object size is unaffected
- ?? EH implies a form of RTTI, implying some increase in code and data size
- ?? The static model implies *static* table generation and some common library runtime support
- ?? A cost is associated with checking the *throw-specification*s of the functions that are called

2.4.3.2.2 Time Overhead

- ?? On entry to each *try-block*
  - o some implementations commit changes to variables in the scopes enclosing the *try-block* - other implementations use a more sophisticated state table[10]
- ?? On exit from each *try-block*
  - o no overhead
- ?? On entry to each *catch-clause*
  - o no overhead
- ?? On exit from each *catch clause*
  - o no overhead
- ?? When calling regular functions
  - o no overhead
- ?? As each local and temporary object is created
  - o no overhead
- ?? On throw
  - o using the tables, determine if the current frame has an appropriate *catch-clause*

    If it does, then:
    - ✍ destruct each 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*

Advantages of this method are that no stack or runtime costs are associated with managing the try/catch or object bookkeeping.

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, and while this may not be a burden on hosted systems with a VM, the cost may be a significant factor for embedded systems. All runtime costs associated occur when an exception is thrown, but because of the need to examine

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

potentially complex state tables, the time it takes to respond to an exception may be large – this needs to be factored in to the probable frequency of exceptions.

One vendor reports a code and data space impact of about 15% for the generated tables. This is an upper limit, since in the vendor's environment there was no need to reduce the image size of programs provided the working set wasn't increased

Editor's Note: How should we include information such as the comment above?

2.4.4 Predictability of Exception Handling Overhead

2.4.4.1 Prediction of throw/catch Performance

One of the reservations expressed about EH is the unpredictable time that may elapse after a throw and before control passes to the catch clause, while automatic objects are being destroyed. It is important in some systems, especially those with "Real Time" requirements, to be able to predict accurately how long operations will take.

These concerns are well founded. 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).

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 RT 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.

Another reservation concerns the memory footprint of the necessary data structures. This has more to do with non-Real-Time embedded applications, where the system constraints may place a premium on the amount of space that the programs and/or data may take. Both approaches incur a space cost. The dynamic approach is likely to have a smaller "data-size" cost and a larger "code-size" cost, than the corresponding static approach.

Editor's Note: We do not appear to have comparative "size" costs for the dynamic versus the static approaches.

2.4.4.2 Empty *exception-specification* Considerations

Can empty *exception-specification*s help a compiler produce more optimal code?

The use of an empty *exception-specification* should reduce overheads. If the called function ensures (statically or dynamically) that it will never throw an exception that invalidates its *exception-specification*, then the caller can build on that guarantee, performing optimisations based on the knowledge that a called function will never throw any exception.

However, a less capable implementation might produce “worse” code if it produces an extra try-catch for functions that don't need it.

For example:

```
int g() throw();

void f() {
    int n = g();
}

// May be implicitly (and poorly) rewritten as --->
void f () {
    int n;
    try {
       n = g();
    } catch (...) {
        unexpected();
    }
}
```

2.4.4.3 Exception Specifications

The need to enforce *exception-specification*s at runtime has costs as described above. However, they can allow optimisation of other code by making *catch-clause*s unreachable and violations of other *exception-specification*s impossible. Empty *exception-specification*s can be especially helpful for optimisation.

2.4.4.4 The "you don’t pay for what you don’t use" Principle

Exception-Handling in general imposes costs even if it is not used. For example, if a function that constructs automatic objects then calls another function, and it cannot be proven by the compiler that the called function does not throw an exception then the calling function will incur object bookkeeping costs, even though the function may never participate in context where an exception is thrown. With the static approach, a possible optimisation is to strip the associated tables and runtime support code from the program if it is known that exceptions will never be thrown.

2.4.4.5 Other Error Handling Strategies

All approaches to error handling involve some runtime and static costs. Return codes, global error values; process termination and ignoring errors all have associated costs in runtime, space, program correctness, maintenance, and readability. In evaluating the costs of exception handling, the costs of the alternatives should not be ignored. If EH is not used, some other mechanisms are invariably required[11].

[11] And ignoring error conditions does not make for robust code

2.4.4.6 Missing stuff

There were some items discussed in the working group, which we were unable to flesh out. These include:

?? Advice to implementers, specifically references to literature on EH (e.g. 'C' Language Translation)
?? Potential implementation pitfalls
?? A comparison of the costs of other strategies

Editor's Note: Do we really need this section? It doesn't get elaborated anywhere.

2.5 Overheads from Templates

2.5.1 Template Overheads

*class-template*s or *function-template*s 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[12]. 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 programs 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.

The same technique can be employed as a PDO, where it is possible to write a class-template called, perhaps, `plist<T>`, that is implemented using `std::list<void*>` to which all operations of `plist<T>` are delegated.

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

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.

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 C++ had added templates to the 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. *[The `Object` class itself inherited from `class NIHCL`, which encapsulated some static data members used by all classes.]* Most of the functions were declared virtual, and had to be overridden by deriving classes[13]. The hierarchy also included a class `Class` that provided a library implementation of RTTI (which was also not yet part of the C++ language). The `Collection` classes, themselves derived from `Object`, could hold only other objects derived from `Object` which implemented the necessary virtual functions.

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

But the NIHCL had several disadvantages due to its use of inheritance versus templates for the implementation of container classes.

The following is a portion of the NIHCL hierarchy (taken from the README file):

```
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 runtime, 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 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 runtime 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[14] *[A class used in a Standard container must be assignable and copy-constructible; often it additionally needs to have a default constructor and implement* `operator ==` *and* `operator <` *].* 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.

Templates can provide powerful facilities for evaluation at compile-time. Doing more of the work at compile-time means less work at runtime.

Hints can be exchanged between the compiler and the library to select a more efficient specialisation, or to select linkage with a reduced-footprint version of the library. In C, it's possible to optimise `printf` this way – `printf` with floating point support vs. `printf` without floating point support.

When the linker sees `printf`, if the symbol `__crt_float` (or equivalent) is defined, then invoke `printf_float`, else invoke `printf_int`. Defining a `float f;` has the side effect of defining `__crt_float`.

[14] A class used in a Standard container must be assignable and copy-constructible; often it additionally needs to have a default constructor and implement `operator ==` and `operator <`.

2.6 Overheads from The Standard IOStreams Library

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 the *IOStream*'s use of the *Locales* library (§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.

2.6.1 Overview - 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 Overview - Execution Speed

Since certain aspects of *IOStream* processing are distributed over multiple facets, it seems that the standard mandates an inefficient implementation. This is not the case and using some form of pre-processing, lots 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 Overview - 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 Overview – Compile-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 results 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 compile-time can be reduced significantly. This is discussed in 3.2.4.

3 Performance – Techniques & Strategies

Description of current approaches:

- ?? Code generation control, including memory placement, initialisation characteristics, et al.
- ?? `#pragma`, and other language modifications
- ?? Application of measurement results in making choices
- ?? Transforming virtual calls into non-virtual calls
- ?? Alternatives to exception handling
- ?? Effects of restrictions upon character types
- ?? Characterization of performance guarantees
- ?? Coding style can affect performance

Editor's Note:	*There is no further mention of these, other than a bit about transforming virtual calls into non-virtual calls, and snippets showing how coding style can affect performance.*

3.1 Programmer Directed Optimisations

Programmers are sometimes surprised when their programs call functions they haven't specified, maybe even haven't written. While a line of C code typically translates to a few machine instructions, a single innocuous-looking line of C++ code may translate to a fairly large number of machine instructions. Simply declaring a variable such as:

```
C c;
```

has the potential to be quite expensive[15] in C++ if, for instance, the `class C` has a default constructor which requires a large amount of code or data to initialise the object `c`.

[15] It is important to remember however, that in C the object would still need to be initialised, but that code would have to be explicitly called and is hence visible to the programmer.

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 C:) provide more extensive guidance, but the following provides some suggestions for writing more efficient code:

?? In constructors, prefer initialisation of data members to assignment. Members of const and reference types must be initialised in the member initialisation list, but it is advisable to list other members as well. The sequence of steps taken to construct a variable of class type is as follows:

- the base classes for the class are initialised using their default constructors unless an explicit initialiser has been provided in the *mem-initializer-list*;
- the data members for the class are initialised using their default constructors[16] unless an explicit initialiser has been provided in the *mem-initializer-list*;
- finally, the body of the constructor is executed.

Therefore, an assignment to a data member within the constructor body means that member is effectively initialised twice[17].

?? As a general principle, don't define a variable before you are ready to initialise it. This prevents effectively initialising the variable twice.

?? Use the direct initialisation syntax `T a(b);` rather than the copy initialisation syntax `T a = b;`. Copy-initialisation is permitted to create an intermediate temporary object, while direct initialisation is not.

?? 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).

?? Whenever possible, compute values and catch errors at translation time rather than runtime. With sophisticated use of templates, a complicated block of code can be compiled to a single constant in the executable.

?? Know what functions the C++ compiler silently generates and calls. Simply defining a variable of some class type may invoke a potentially expensive constructor function.

?? 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. But be careful not

[16] Built-in data types do not have a default constructor, so unless they are explicitly initialised, they will have an unspecified or undefined value (according to their type).

[17] Actually, an object can only be initialised once – this is really an initialisation followed by an assignment, a distinction that is not so clear in C, but can very different in C++.

to create unnecessary temporary objects, by using an argument that must be converted to the type of the function parameter.

?? Unless you need automatic type conversions, make all one-argument constructors[18] explicit. This will prevent calling them accidentally. Conversions can still be done when necessary by stating them explicitly in the code, avoiding the penalty of hidden and unexpected conversions.

?? 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 runtime speed and memory footprint. Temporary objects may be generated when passing parameters to functions, returning values from functions, or initialising objects.

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

```
T a;              // inefficient: don't create an object before
                  // its initialization is really needed
a = b + c;        // inefficient: (b + c) creates a temporary
                  // object and then assigns it to a
T a( b ); a += c; // better:      no temporary objects created
```

?? 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. 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.

?? 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

[18] 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.

blocks of memory instead of releasing them back to the heap whenever an object is deleted.

?? Sometimes it is helpful to “widen” a class' interface with functions that take different data types to prevent automatic conversions (such as adding an overload on `char *` to a function which takes a `std::string` parameter). The numerous overloads for operators `+`, `==`, `!=`, and `<` in the `<string>` header are an example of such a "fat" interface[19]. 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 a PDO, 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[20].

?? Pre-compute values that won't change. To avoid repeated function calls, 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.

?? 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.

?? The use of dynamic binding and virtual functions has some overhead that can affect performance. In a typical implementation, every object (which has virtual functions) in the hierarchy needs an extra member for the `vptr`, and dynamic selection of functions at runtime involves de-referencing this `vptr`. But the biggest overhead factor, is that compilers are often unable to inline virtual functions (§2.3.3).

?? Use function objects with the Standard Library algorithms rather than function pointers. The data flow-analysers of many optimisers are defeated by function

[19] It is also worth noting, that even if a conversion is needed, it is so metimes 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.

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

pointers, but function objects are passed by value and optimisers can easily handle inline functions on objects.

?? Templates provide compile-time polymorphism, wherein type selection does not incur any runtime penalty. If appropriate to the design, consider using templates as interfaces instead of abstract base classes. 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.

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. But now that processors make extensive use of on-board cache and instruction pipelines, the smallest code is often the fastest as well. Furthermore, tests indicate that the cost of function calls has greatly reduced, and that modern optimisers are very good at deciding when and where to inline functions automatically (§2.3.3).

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 efficient algorithms. An algorithm with quadratic performance may be acceptable for small data sets, but gives abysmal performance on large inputs. Small local optimisations may be effective if profiling reveals a bottleneck.

3.2 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 runtime 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 (actually - 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.

The 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 runtime 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 runtime 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 runtime 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 runtime 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 runtime
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;
    a += 1.0;
}

// May be interpreted as:
void test () {
    static const double tmp1  = 1.0;  // ‘tmp1’ is ROMable
    const double *tmp2        = &tmp1;
    double a;
    a += *tmp2;
}
```

Editor’s Note: *Why does it need the intermediate ‘tmp2’? I think that this would be better interpreted as (quite apart from the undefined nature of adding `1.0` to an un-initialised `double`):*

```
void test () {
    static const double tmp = 1.0;  // ‘tmp’ is ROMable
    double a;
    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*[21].

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"?*

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

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.

3.4.1.2 Inheritance

Converting a pointer to a derived class to a pointer to base-class[22] will not introduce any run-time overhead in most implementations (see 2.3.3). 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 2.3.4). 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 2.3.5). 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 2.3.3). 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.

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

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).

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 2.4). 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

4.1 BASIC I/O-HARDWARE ADDRESSING

4.1.1 Scope

As the C language has matured over the years, various extensions for accessing basic I/O-Hardware (*iohw*) registers have been added to address deficiencies in the language. Today almost all C compilers for freestanding environments and embedded systems support some method of direct access to *iohw* 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++.

This report provides an approach to codifying common practice and providing a single uniform syntax for basic *iohw* register addressing.

4.1.2 Rationale

Ideally, it should be possible to compile C or C++ source code that operates directly on *iohw* registers with different compiler implementations for different platforms and get the same logical behaviour at runtime. 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.

The problem areas are the same for C and C++, and the standardization method proposed is applicable for both languages. A proposed *iohw* addressing interface for the C language is described in:

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

Although this interface is based on C macros, the C++ language provides features which make it possible to create more efficient and flexible implementations of this interface, while maintaining I/O driver source code portability.

4.1.3 Basic Standardisation Objectives

A standardisation method for basic I/O-Hardware 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 I/O register itself from the characteristics of the underlying execution environment (processor architecture, bus system, addresses, alignment, endian, etc.)]*.

4.2 Basic I/O-Hardware Addressing Header — <ciohw>

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.

4.2.1 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 I/O registers in a given platform.

The primary purpose of the interface is to separate characteristics which are portable and specific for a given I/O register, for instance the register bit width; from characteristics which are related to a specific execution environment, such as the I/O register address; processor bus type and endian; device[23] bus size and endian, 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 I/O registers can be connected.

In the driver source code, an I/O 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 I/O syntax approach creates a conceptually simple model for I/O registers:

symbolic name for I/O register ? complete definition of the access method

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

4.2.2 The Abstract Model

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

? The users portable source code

? The users I/O register definitions

? The vendors *iohw* implementation

[23] 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.

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

The bottom layer is the compiler vendors implementation of the `<ciohw>` header. 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 methods refers to the various ways of connecting and addressing I/O registers or I/O devices in the given processor architecture).

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

The middle layer contains the users specification of the symbolic I/O register names used by the source code in the top layer. This layer associates the symbolic names with *access-specifications* for the I/O 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 `<ciohw>` header. The user must update these I/O register *access-specifications* when the I/O driver source code is ported to a different platform.

Appendix B: proposes a generic C++ syntax for I/O register *access-specifications*. Using a general syntax on this layer may extend portability to include users I/O register specification, so it can be used with different compiler implementations for the same platform.

4.2.2.1 The Module Set

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

1. I/O Driver Module
   - The I/O driver source code
   - Portable across compilers and platforms
   - Includes `<ciohw>` and `"iohw_ta.h"`
2. `<ciohw>`
   - Defines I/O functions and access methods
   - Typically specific for a given compiler
   - Implemented by the compiler vendor
3. `"iohw_ta.h"`
   - Defines symbolic I/O register names and their corresponding access methods
   - Specific to the execution environment
   - Implemented and maintained by the programmer

And might be used as follows:

```
#include <ciohw>
#include "iohw_ta.h"   // my I/O 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
```

The programmer only sees the characteristics of the I/O 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 I/O driver source code being necessary.

4.2.3 I/O Register Characteristics

The principle behind the `<ciohw>` interface is that all I/O register characteristics should be visible to the driver source code, while all platform specific characteristics are encapsulated by the header files and the underlying `<ciohw>` implementation.

I/O 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 I/O register specific characteristics should be visible at the I/O driver code level and should not be hidden by the `<ciohw>` interface implementation.

4.2.4 The Most Basic I/O Operations

The most common operations on I/O-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 the `<ciohw>` interface 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 I/O operations.

4.2.5 The *access-specification*

The *access-specifications* defined in the header `<ciohw>` are used only as parameters in the functions for defining I/O register access.

The `access_spec` parameter represents or references a complete description of how the *iohw* register should be addressed in the given hardware platform. It is an abstract data type with a well-defined behaviour[24].

[24] 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.

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

In general, an `access_spec` definition 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_spec`:

- Processor bus (if more than one)
- Access method (if more than one)
- I/O register endian (if register width is larger than the device bus width)
- Interleave factor for I/O register buffers (if bus width for the device is smaller)
- User supplied access driver functions

The definition of an I/O register object may or may not require a memory instantiation, depending on how a compiler vendor has chosen to implement *access-specifications*. For maximum performance, this could be a simple definition based on compiler specific address range and type qualifiers, in which case no instantiation of an `access_spec` object would be needed in data memory.

See also Appendix A: for further details and implementation considerations.

4.3 The <ciohw> Interface

The header `<ciohw>` declares several functions, which together create a data-type-independent interface for basic *iohw* addressing. The provider of `<ciohw>` may choose to use inline functions, macros or *function-templates* to implement these functions. I/O driver modules using the functions defined by this header can potentially be compiled with both C and C++ compilers.

Editor's Note:	*Why is it a requirement that the C++ Hardware I/O has to be C compatible?*

4.3.1 Functions for Single Register Access

Synopsis:

```
#include <ciohw>
//...
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 perform the basic operations `READ`, `WRITE`, `OR`, `AND` or `XOR` as identified by the function named on this register.

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

4.3.2 Functions for Register Buffer Access

Synopsis:

```
#include <ciohw>
//...
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 perform the basic operations `READ`, `WRITE`, `OR`, `AND` or `XOR` as identified by the function named on this register.

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

The `index` parameter is an 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 *iohw* 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 A.2.2.

4.3.3 Functions for `access_spec` Initialisation

Synopsis:

```
#include <ciohw>
//...
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* specific initialisation before performing any other operation on the I/O register (or set of I/O registers) identified by `access_spec`. This call should be placed in the driver source code so that 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.

For 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 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 be returned to the operating system.
>
> If a set of memory mapped I/O registers is specified to use based addressing, then 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. when 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* specific initialisation is required by a given `<ciohw>` implementation, the `io_at_init` and `io_at_release` definitions may be empty.

In C++, the implementation may use a class whose constructor and destructor implement this functionality.

4.3.4 Functions for `access_spec` Copying

Synopsis:

```
#include <ciohw>
//...
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. This function does not return a value.

If *access-specification* copying is not supported by a given `<ciohw>` 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 are used with multiple instances of the same hardware device. It often provides a faster alternative than passing the `access_spec` as a function parameter.

For example:

```
#include <ciohw>
#include “iohw_ta.h”  // MYDEV_CFG and MYDEV_DATA are defined
                      // relative to a dynamic MYDEV_BASE

// Portable driver function
uint8_t my_device_driver(void)
{
    iowr(MYDEV_CFG, 0x33);
    return iord(MYDEV_DATA);
}

// Users driver application
uint8_t  d1,d2;
// Read from our 2 I/O devices
io_art_cpy(MYDEV_BASE, DEVICE1); // Select device 1
d1 = my_device_driver();
io_art_cpy(MYDEV_BASE, DEVICE2); // Select device 2
d2 = my_device_driver();
```

Appendix A: Implementing <ciohw>

(A guide for implementers)

A.1 Purpose

The `<ciohw>` header defines a standardised function syntax for basic I/O-Hardware (*iohw*) addressing. This header would normally be provided by the compiler vendor.

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

This section gives some guidelines for implementers on how to implement the `<ciohw>` header in a relatively straightforward manner given a specific processor and bus architecture.

A.1.1 Recommended Steps

Briefly, the recommended steps for implementing the `<ciohw>` header are:

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

A.1.2 Compiler Considerations

In practice, an implementation will often require that very different machine code is generated for different I/O access cases. Furthermore, with some processor architectures, *iohw* 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 I/O 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 runtime 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 the `<ciohw>` header can usually be made using template functionality. A template-based solution allows the number of compiler specific intrinsic I/O types or intrinsic I/O 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. Fully featured *iohw* implementations will usually require direct compiler support (or using extensions to the language).

Fully featured, zero-overhead implementations of `<ciohw>` can be done using templates. An approach to doing this is discussed in Appendix B:.

A.2 Overview of I/O Hardware Connection Options

The various ways of connecting an I/O register to processor hardware are determined primarily by combinations of the following three hardware characteristics:

- ?? The bit width of the logical I/O register
- ?? The bit width of the data-bus of the I/O device
- ?? The bit width of the processor-bus

A.2.1 Multi-Addressing and I/O Register Endian

If the width of the logical I/O register is greater than the width of the I/O device data bus, an I/O access operation will require multiple consecutive addressing operations.

The I/O register endian information describes whether the MSB or the LSB byte of the *logical I/O register* is located at the *lowest* processor bus address.

(Note that the I/O register endian has nothing to do with the endian of the underlying processor hardware architecture).

Table: Logical I/O register / I/O device addressing overview[25]

Logical I/O register widths	**I/O 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)

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

A.2.2 Address Interleave

If the size of the I/O 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 I/O device is connected to the 32-bit data bus, then three adjacent registers in the I/O 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)

I/O 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)

A.2.3 I/O Connection Overview:

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

Table: Interleave between adjacent I/O registers in buffer

I/O 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)

A.2.4 Generic Buffer `index`

The interleave distance between two logically adjacent registers in an I/O register array can be calculated from[26]:

- ?? The size of the logical I/O register in bytes
- ?? The processor data bus width in bytes
- ?? The device data bus width in bytes

[26] For systems with byte-aligned addressing.

Conversion from I/O 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`

A.3 access-specifications *for Different I/O 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 runtime:***

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 runtime (by some platform-dependent part of the program). This also enables a set of I/O driver functions to be used with multiple instances of the same *iohw*.

?? ***Indexed bus addressing:***

Also called *orthogonal* or *pseudo-bus* addressing. This is a common way to connect a large number of I/O 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 I/O 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 I/O register itself).

This access method also makes it particularly easy for a user to connect common I/O 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 an I/O 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 I/O driver source library to any kind of platform hardware and platform software using the appropriate platform-specific interface functions.

In general, an implementation should always support the simplest addressing case, whether it is the *constant-address* or *base-address* method that is used will depend on the processor architecture. Apart from this, an implementer is free to add any additional cases required to satisfy a given domain.

Because of the different number of parameters required and parameter ranges used in an *access-specification*, it is often convenient to define a number of different *access-specification* formats for the different access methods.

A.4 Atomic Operation

It is a requirement of the `<ciohw>` implementation, that in each I/O function, a given (partial[27]) I/O register is addressed exactly once during a ***read*** or a ***write*** operation and exactly twice during a ***read-modify-write*** operation.

It is recommended that each I/O function in an `<ciohw>` implementation, be implemented such that the I/O access operation becomes *atomic* whenever possible.

However, atomic operation is not guaranteed to be portable across platforms for *read-modify-write* operations (`ioor`, `ioand`, `ioxor`) 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.

A.5 Read-Modify-Write Operations and Multi-Addressing

In general, *read-modify-write* operations should perform a complete "read" of the I/O register, followed by the modify operation, and concluded by a complete "write" to the I/O register.

It is therefore recommended that an implementation of multi-addressing cases should not use *read-modify-write* machine instructions during *partial* register addressing operations.

The rationale for this restriction is 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 I/O register implementations often take a snap-shot of the whole logical I/O register when the first *partial* register is being read, so that data will be stable and consistent during the whole read operation.

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

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.

A.6 I/O Initialisation

With respect to the standardisation process, it is important to make a clear distinction between I/O-Hardware (device) related initialisation, and platform related initialisation. Typically, three types of initialisation are related to I/O handling:

- ? I/O-Hardware (device) initialisation
- ? I/O access initialisation
- ? I/O device selector initialisation

Here only I/O access initialisation and I/O device selector initialisation are relevant for basic *iohw* addressing:

Editor's Note: What it "I/O device selector"? The term is not defined.

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

I/O access initialisation: concerns the initialisation and definition of `access_spec` objects themselves.

This process is implementation defined. It depends on both the platform and the processor architectures, and also on which underlying access methods are supported by the `<ciohw>` implementation.

The function:

```
io_at_init(access_spec)
```

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

I/O device selector initialisation: is used when, for instance, the same I/O driver code needs to service multiple *iohw* devices of the same type.

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

Another solution is to use *access-specification* copying, and *access-specifications* with dynamic access information. The function:

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

provides a portable way to do this.

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

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

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 runtime.

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 perfo rmance can usually be achieved if more of the information is static.

Appendix B: Generic *access-specification* for *iohw* Addressing

B.1 *Generic* access-specification *Descriptor*

This chapter proposes consistent and complete specification syntax for defining I/O registers and the access method parameters.

Prior art has used a number of (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 super-set 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.

B.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 I/O register value (size of I/O register) ,
access limitation type ,
I/O register device bus type (size and endian of I/O device bus)

type for I/O register value (size of I/O 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*

I/O register device bus type:

device8	*// register width = device bus width = 8 bit*
device8l	*// register width > device bus width, MSB on low address*
device8h	*// register width > device bus width, MSB on high address*
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)

B.2.1 Bus Connection Parameters

The possible I/O register to bus connections can be completely specified using only two parameters:

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

For example, a possible definition of general I/O register connection types might be:

```
enum bus_t    { bw8 = 1, bw16 = 2, bw32 = 4, bw64 = 8 };
enum device_t { device8,   device8l, device8h,  device16,  device16l,
                device16h, device32, device32l, device32h, device64 };
```

For another example, an implementation for a given processor architecture may only support a subset of the I/O register connection types. Possible I/O 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, device8l, device8h, device16, device16l,
                device16h };
```

B.2.2 Detection of Read / Write Violations in I/O Registers

The *access-specifications* can specify a *limitation* parameter, which makes it possible to detect illegal use of an I/O 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 I/O 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 `<ciohw>`, the compiler will typically diagnose that no matching *function-template* can be found for the "not-allowed" cases.

For example:

```
// --- part of the <ciohw> header
//
// Define a type to validate I/O 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>[28]

// 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 { };
```

[28] 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>`.

```
// --- 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
```

B.2.3 *access-specifications* for Different Processor Busses

An implementation must 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 should have its own set of *access-specifications*.

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 { };
```

B.2.4 *access-specifications* for Different I/O Addressing 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 I/O 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 { };
```

B.2.5 Optimisation Possibilities for Typical Implementations

B.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 I/O 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 I/O functions that have to be implemented can be reduced with no efficiency penalty using simple delegation, possibly using macro definitions such as:

```
#define iowr(access_spec,val)  iowrbuf(access_spec,0,(val))
```

or equivalent inline-functions or *function-templates*.

B.2.5.2 Multi-Addressing and Endian

Typical candidates for platform dependent optimisations are I/O functions for the multi-addressing cases (*logical I/O register* width > *I/O device bus* width) where the width of the *device data bus* matches the width of the *processor data bus*; e.g. the combinations of:

- (*device8h* or *device8l*) and *bw8*
- (*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 endian functions. The other endian functions can often be implemented efficiently using one load or store operation, plus one or more register swap operations.

Appendix C: Bibliography

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."

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 runtime 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, including:

- exception handling (EH)
- runtime 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.

Hatton, Les
Does OO Sync with How We Think?
IEEE Software, May/June 1998.

During the life cycle of a software system, time spent on post-release maintenance is far larger than the time spent in its creation. Therefore, reliability and ease of modification are important quality factors. This paper describes two sizable software projects, one in C and one in C++, using objected-oriented design. The use of OO and inheritance appears to be

associated with more defects, and these defects required more effort to fix, compared to the C project.

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.

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 runtime 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."

Knuth, Donald E.
The Art of Computer Programming, Volume 1, Reissued 3rd Edition
Addison-Wesley

Fundamental Algorithms	[1997]
Seminumerical Algorithms	[1998]
Sorting and Searching	[1998]

Note (Lois):	*Unfortunately, I don't have these classic volumes in my library (yet -- I'm waiting until I'm smart enough to understand them). Can someone else add a brief remark?*

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 runtime 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.

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.

Lippman, Stanley B. and Lajoie, Josée
C++ Primer, 3rd Edition
Addison-Wesley, 1998

This thorough introduction to C++ includes discussions of how various language constructs produce different executable code, plus measurements of runtime performance. For example, using `reserve()` to pre-allocate space for a vector resulted in slower execution times when the vector held strings or doubles, but faster times if the value type was a large, complex class.

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."

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 runtime 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 runtime 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 runtime 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.

Stroustrup, Bjarne
The C++ Programming Language, 3rd Edition
Addison-Wesley, 1998

This definitive work from the language's author 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 runtime 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 runtime, 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.

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.