Doc. No.:	03-0012/N1430
Date:	4 March 2003
Project Editor	Lois Goldthwaite

Technical Report on C++ Performance

Executive Summary:

The aim of this report is:

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

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

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

Performance implications of object-oriented programming are presented. This discussion rests on measurements of key language facilities supporting OOP, such as classes, class member functions, class hierarchies, virtual functions, multiple inheritance, and run-time type identification (RTTI). It is demonstrated that, with the exception of RTTI, current C++ implementations can match hand-written low-level

code for equivalent tasks. Similarly, the performance implications of generic programming using templates are discussed. Here, however, the emphasis is on techniques for effective use. Error handling using exceptions is discussed based on another set of measurements. Both time and space overheads are discussed. In addition, the predictability of performance of a given operation is considered.

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

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

Acknowledgments

The following people contributed work to this Technical Report:

Dave Abrahams

Mike Ball

Greg Colvin

Embedded C++ Technical Committee (Japan)

Hiroshi Fukutomi

Lois Goldthwaite

Yenjo Han

Seiji Hayashida

Howard Hinnant

Brendan Kehoe

Robert Klarer

Jan Kristofferson

Dietmar Kuehl

Jens Maurer

Fusako Mitsuhashi

Hiroshi Monden

Nathan Myers

Masaya Obata

Martin O'Riordan

Tom Plum

Dan Saks

Martin Sebor

Bill Seymour

Bjarne Stroustrup

Detlef Vollmann

Contents:

1 Introduction

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

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

Language features that are never used in a program should not have a cost in extra code size, memory size, or run-time. If there are places where C++ cannot guarantee zero overhead 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, programs that run into resource limits of some kind are either very large or very small. 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 at a very low level. Historically the interfaces to hardware have been implemented as proprietary extensions to the compiler (often as macros). This has 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 Glossary

ABC – commonly used shorthand for an **A**bstract **B**ase **C**lass – a base class (often a virtual base class) which contains pure virtual member functions and thus cannot be instantiated (§IS-10.4).

Access Method – refers to the way a memory cell or an I/O device is connected to the processor system and the way in which it is addressed.

Addressing Range – a processor has one or more addressing ranges. Program memory, data memory and I/O devices are all connected to a processor addressing range. A processor may have special ranges which can only be addressed with special processor instructions.

A processor's physical address and data bus may be shared among multiple addressing ranges.

Address Interleave – the gaps in the addressing range which may occur when a device is connected to a processor data bus which has a bit width larger than the device data bus.

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

Code Bloat – the generation of excessive amounts of code instructions, for instance, from unnecessary template instantiations.

Code Size – the portion of a program's memory image devoted to executable instructions. Sometimes immutable data also is placed with the code.

Cross-Cast – a cast of an object from one base class subobject to another. This requires RTTI and the use of the `dynamic_cast<...>` operator.

Data Size – the portion of a program's memory image devoted to data with static storage duration.

Device – this term 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 bit width has significance to the access method used for the I/O device.

Device Bus – the data bus of a device. The bit width of the device bus may be less than the width of the processor data bus, in which case it may influence the way the device is addressed.

Device Register – a single logical register in a device. A device may contain multiple registers located at different addresses.

Device Register Buffer – multiple contiguous registers in a device.

Device Register Endianness – the endianness for a logical register in a device. The device register endianness may be different from the endianness used by the compiler and processor.

Down-Cast – a cast of an object from a base class subobject, to a more derived class subobject. Depending on the complexity of the object's type, this may require RTTI and the use of the `dynamic_cast<...>` operator.

EEPROM – Electrically Erasable Programmable Read-Only Memory. EEPROM retains its contents even when the power is turned off, but can be erased by exposing it to an electrical charge. EEPROM is similar to flash memory (sometimes called flash EEPROM). The principal difference is that EEPROM requires data to be written or erased one byte at a time whereas flash memory allows data to be written or erased in blocks.

Endianness – if the width of a data value is larger than the width of data bus of the device where the value is stored the data value must be located at multiple processor addresses.

Big-endian and little-endian refer to whether the most significant byte or the least significant byte is located on the lowest (first) address.

Embedded System – a program which functions as part of a device. Often the software is burned into firmware instead of loaded from a storage device. It is usually a freestanding implementation rather than a hosted one with an operating system (§IS-1.4¶7).

Flash Memory – a non-volatile memory device type which can be read like ROM. Flash memory can be updated by the processor system. Erasing and writing often require special handling. Flash memory is considered to be ROM in this document.

Interleave – see address interleave.

I/O – **I**nput/**O**utput – in this paper, the term used for reading to and writing from device registers.

I/O Bus – special processor addressing range used for input and output operations on hardware registers in a device.

I/O Device – synonym for device.

I/O Mapped Device – device connected to a special processor addressing range used for input and output operation of hardware registers.

Locality of Reference – the heuristic that most programs tend to make most memory and disk accesses to locations near those accessed in the recent past. Keeping items accessed together in locations near each other increases cache hits and decreases page faults.

Logical Register – refers to a device register treated as a single entity. A logical register will consist of multiple physical device registers if the width of the device bus is less than the width of the logical register.

Memory Bus – a processor addressing range used when addressing data memory and/or program memory. Some processor architectures have separate data and program memory busses.

Memory Device – chip or function block intended for holding program code and/or data.

Memory Mapped I/O – I/O devices connected to the processor addressing range which are also used by data memory.

MTBF – **M**ean-**T**ime **B**etween **F**ailure – the statistically determined average time a device is expected to operate correctly without failing. The calculation takes into account the MTBF of all devices in a system. The more devices in a system, the lower the system MTBF.

Non-Volatile Memory – a memory device that retains the data it stores, even when electric power is removed.

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

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

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

POD – shorthand for "**P**lain **O**ld **D**ata" – term used in the Standard (§IS-1.8¶5) to describe a data type which is compatible with the equivalent data type in C in layout, initialization, and its ability to be copied with memcpy.

PROM – **P**rogrammable **R**ead **O**nly **M**emory. It is equivalent to ROM in the context of this document.

RAM – **R**andom **A**ccess **M**emory. Memory device type for holding data or code. The RAM content can be modified by the processor. Content in RAM can be accessed more quickly than that in ROM, but is not persistent through a power outage.

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

ROM – **R**ead **O**nly **M**emory. A memory device type, normally used for holding program code, but may contain data of static storage duration as well. Content in ROM can not be modified by the processor.

ROMable – refers to entities that are appropriate for placement in ROM in order to reduce usage of RAM or to enhance performance.

ROMability – refers to the process of placing entities into ROM so as to enhance the performance of programs written in C++.

RTTI – **R**un **T**ime **T**ype **I**nformation. Information generated by the compiler which makes it possible to determine at runtime if an object is of a specified type.

Swap –
Swapped Out –
Swapping – the process of moving part of a program's code or data from fast RAM to a slower form of storage such as a hard disk. See also Working Set and Virtual Memory Addressing.

System-on-Chip (SoC) – a term referring to an embedded system where most of the functionality of the system is implemented on a single chip, including the processor(s), RAM and ROM.

Text Size – a common alternative name for "Code Size".

UDC – commonly used shorthand for a **U**ser **D**efined **C**onversion, which refers to the use, implicit or explicit, of a class member conversion operator.

Up-Cast – a cast of an object to one of its base class subobjects. This does not require RTTI and can use the `static_cast<...>` operator.

VBC – commonly used shorthand for a **V**irtual **B**ase **C**lass (§IS-10.1). A single sub-object of the VBC is shared by every sub-object in an inheritance graph which declares it as a virtual base.

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

Working Set – the portion of a running program that at any given time is physically in memory and not swapped out to disk or other form of storage device.

WPA – **W**hole **P**rogram **A**nalysis. A term used to refer to the process of examining the fully linked and resolved program for optimization possibilities. Traditional analysis is performed on a single translation unit (source file) at a time.

1.2 Typical Application Areas

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

1.2.1 Embedded Systems

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

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

    These systems typically use single chips containing both ROM and RAM. Single-chip systems (System-on-Chip or SoC) in this category typically hold approximately 32 KBytes for RAM and 32, 48 or 64 KBytes for ROM[2].

    Examples of applications in this category are:

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

    These systems typically use separate ROM and RAM chips to execute a fixed application, where size is limited. There are different kinds of memory device, 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 32 MBytes 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 2003

[2] These numbers are derived from the popular C8051 chipset.

- **Large**

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

  - personal digital assistant (PDA) – equivalent to a personal computer without a screen, keyboard, or hard disk
  - digital television
  - set-top box
  - car navigation system
  - central controllers for large production lines of manufacturing machines

- **Timing:**

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

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

    Examples of applications in this category are:

    - motor control
    - nuclear power plant control
    - hand-held digital VCR
    - mobile phone
    - CD or DVD player
    - electronic musical instruments
    - hard disk controllers
    - digital television
    - digital signal processing (DSP) applications

  - **Non-critical**

    Examples of applications in this category are:

    - digital still camera
    - copy machine
    - printer
    - car navigation system

1.2.2 Servers

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

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

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

2 Language Features – Overheads & Strategies

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

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

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

2.1 Namespaces

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

Namespace qualifiers enable programmers to use shorter identifier names when compared with alternative mechanisms. In the absence of namespaces, the programmer has to explicitly alter the names to ensure that name clashes do not occur. One common approach to this is to use a canonical prefix on each name:

```
static char* mylib_name      = "My Really Useful Library";
static char* mylib_copyright = "June 15, 2003";

std::cout << "Name:       " << mylib_name      << std::endl
          << "Copyright:  " << mylib_copyright << std::endl;
```

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

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

char* ThisLibInfo::name      = "Another Useful Library";
char* ThisLibInfo::copyright = "August 17, 2003";

std::cout << "Name:       " << ThisLibInfo::name      << std::endl
          << "Copyright:  " << ThisLibInfo::copyright << std::endl;
```

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

```
namespace ThisLibInfo {
    char*  name        = "Yet Another Useful Library";
    char*  copyright   = "December 18, 2003";
};

using ThisLibInfo::name;
using ThisLibInfo::copyright;

std::cout << "Name:       " << name      << std::endl
          << "Copyright:  " << copyright << std::endl;
```

When referencing names from the same enclosing namespace, no using declaration or namespace qualification is necessary.

2.2 Type Conversion Operators

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

```
int i_pi = (int)3.14159;
```

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

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

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 operators, but the latter distinguish between the different purposes for which the cast is being used. The type conversion operator syntax is easier to identify in source code, and thus contributes to writing programs that are more likely to be correct3. It should be noted that as in C, a cast may create a temporary object of the desired type, so casting can have run-time implications.

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

2.3 Classes and Inheritance

Programming in the object-oriented style often involves heavy use of class hierarchies. This section examines the time and space overheads imposed by the primitive operations using classes and class hierarchies. Often, the alternative to using class hierarchies is to perform similar operations using lower-level facilities. For example, the obvious alternative to a virtual function call is an indirect function call. For this reason, the costs of primitive operations of classes and class hierarchies are compared to those of similar functionality implemented without classes.

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

In general, the statements about implementation techniques and performance are believed to be true for the vast majority of current implementations, but are not meant to cover experimental implementation techniques, which might produce better – or just different – results. See “Inside the C++ Object Model” [BIBREF-17] for further information.

2.3.1 Representation Overheads

A `class` without a virtual function requires exactly as much space to represent as a `struct` with the same data members. That is, no space overhead is introduced from using a `class` compared to a C `struct`. A `class` object does not contain any data that the programmer does not explicitly request (apart from possible padding to achieve appropriate alignment, which may also be present in C structs). In particular, a non-virtual function does not take up any space in an object of its class, and neither does a static data or function member of the class.

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

A polymorphic `class` (a `class` that has one or more virtual functions) incurs a per-object space overhead of one pointer, plus a per-`class` space overhead of a “virtual function table” consisting of one or two words per virtual function. In addition, a per-`class` space overhead of a “type information object” (also called “run-time type information” or RTTI) is typically about 40 bytes per class, consisting of a name string, a couple of words of other information and another couple of words for each base classWhole program analysis (WPA) can be used to eliminate unused virtual function tables and RTTI data. Such analysis is particularly suitable for relatively small programs that do not use dynamic linking, and which have to operate in a resource-constrained environment such as an embedded system.

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

Aggregating data items into a small `class` or `struct` can impose a run-time overhead if the compiler does not use registers effectively, or in other ways fails to take advantage of possible optimizations when `class` objects are used. The overheads incurred through the failure to optimize in such cases are referred to as “the abstraction penalty” and are usually measured by a benchmark produced by Alex Stepanov(D.3). For example, if accessing a value through a trivial smart pointer is significantly slower than accessing it through an ordinary pointer, the compiler is inefficiently handling the abstraction. In the past, most compilers had significant abstraction penalties and several current compilers still do. However, at least two compilers[4] have been reported to have abstraction penalties below 1% and another a penalty of 3%, so eliminating this kind of overhead is well within the state of the art.

2.3.2 Basic Class Operations

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

Table 2.3-1		#1	#2	#3	#4	#5
`Non-virtual:`	`px->f(1)`	0.019	0.002	0.016	0.085	0
	`g(ps,1)`	0.020	0.002	0.016	0.067	0
`Non-virtual:`	`x.g(1)`	0.019	0.002	0.016	0.085	0
	`g(&s,1)`	0.019	0	0.016	0.067	0.001
`Static member:`	`X::h(1)`	0.014	0	0.013	0.069	0
	`h(1)`	0.014	0	0.013	0.071	0.001

[4] These are production compilers, not just experimental ones.

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

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

2.3.3 Virtual Functions

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

Table 2.3-2		#1	#2	#3	#4	#5
Virtual:	`px->f(1)`	0.025	0.012	0.019	0.078	0.059
Ptr-to-fct:	`p[1](ps,1)`	0.020	0.002	0.016	0.055	0.052
Virtual:	`x.f(1)`	0.020	0.002	0.016	0.071	0
Ptr-to-fct:	`p[1](&s,1)`	0.017	0.013	0.018	0.055	0.048

When averaged over a few runs, the minor differences seen above smooth out, illustrating that the cost of virtual function and pointer-to-function calls is identical. Here it is the compiler/machine combination #3 that most closely matches the naïve model of what is going on. For `x.f(1)` implementations #2 and #5 recognize that the virtual function table need not be used because the exact type of the object is known and a non-virtual call can be used. Implementations #4 and #5 appear to have systematic overheads for virtual function calls (caused by treating single-inheritannce and multiple inheritance equivalently, and thus missing an optimization). However, this overhead is on the order of 20% and 12% – far less than the variability between compilers.

Comparing Table 2.3-1 and Table 2.3-2, we see that implementations #1, #2, #3, and #5 confirm the obvious assumption that virtual calls (and indirect calls) are more expensive than non-virtual calls (and direct calls). Interestingly, the overhead is in the range 20% to 25% where one would expect it to be, based on a simple count of operations performed. However, implementations #2 and #5 demonstrate how (implicit) inlining can yield much larger gains for non-virtual calls. Implementation #4 counter-intuitively shows virtual calls to be faster than non-virtual ones. If nothing else, this shows the danger of measurement artifacts. It may also show the effect of additional effort in hardware and optimizers to improve the performance of indirect function calls.

2.3.3.1 Virtual functions of class templates

Virtual functions of a class template can incur overhead. If a class template has virtual member functions, then each time the class template is specialized it will have to generate new specializations of the member functions and their associated support structures such as the virtual function table.

A straight-forward 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 does not depend on *template-parameter*s and could be separate code, may cause each instantiation to contain potentially large and redundant code sequences. One optimization available to the programmer is to use non-template helper functions, and to describe the template implementation in terms of these helper functions. For example, many implementations of the `std::map` class store data in a red-black tree structure. Because the red-black tree is not a class template, its code need not be duplicated with each instantiation of `std::map`.

A similar technique places non-parametric functionality that doesn't need to be in a template into a non-template base class. This technique is used in several places in the standard library. For example, the `std::ios_base` class (§IS-27.4.2) contains static data members which are shared by all instantiations of input and output streams. Finally, it should be noted that the use of templates and the use of virtual functions are often complementary techniques. A class template with many virtual functions could be indicative of a design error, and should be carefully re-examined.

2.3.4 Inlining

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

Table 2.3-3		#1	#2	#3	#4	#5
Non-inline:	`px->g(1)`	0.019	0.002	0.016	0.085	0
Non-inline:	`x.g(1)`	0.019	0.002	0.016	0.085	0
Inline:	`ps->k(1)`	0.007	0.002	0.006	0.005	0
Macro:	`K(ps,1)`	0.005	0.003	0.005	0.006	0
Inline:	`x.k(1)`	0.005	0.002	0.005	0.006	0
Macro:	`K(&s,1)`	0.005	0	0.005	0.005	0.001

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

2.3.5 Multiple Inheritance

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

- The caller retrieves a suitable offset from the virtual function table and adds it to the pointer to the called object, or
- a “thunk” is used to perform this adjustment. A thunk is a simple fragment of code that is called instead of the actual function, and which performs a constant adjustment to the object pointer before transferring control to the intended function.

Table 2.3-4		#1	#2	#3	#4	#5
SI, non-virtual:	`px->g(1)`	0.019	0.002	0.016	0.085	0
Base1, non-virtual:	`pc->g(i)`	0.007	0.003	0.016	0.007	0.004
Base2, non-virtual:	`pc->gg(i)`	0.007	0.004	0.017	0.007	0.028
SI, virtual:	`px->f(1)`	0.025	0.013	0.019	0.078	0.059
Base1, virtual:	`pa->f(i)`	0.026	0.012	0.019	0.082	0.059
Base2, virtual:	`pb->ff(i)`	0.025	0.012	0.024	0.085	0.082

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

It appears that implementations #1 and #2 do not incur extra overheads from multiple inheritance compared to single inheritance. This could be caused by imposing multiple inheritance overheads redundantly even in the single inheritance case. However, the comparison between (single inheritance) virtual function calls and indirect function calls in Table 2.3-2 shows this not to be the case.

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

2.3.6 Virtual Base Classes

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

Table 2.3-5		#1	#2	#3	#4	#5
SI, non-virtual:	`px->g(1)`	0.019	0.002	0.016	0.085	0
VBC, non-virtual:	`pd->gg(i)`	0.010	0.010	0.021	0.030	0.027
SI, virtual:	`px->f(1)`	0.025	0.013	0.019	0.078	0.059
VBC, virtual:	`pa->f(i)`	0.028	0.015	0.025	0.081	0.074

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

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

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

2.3.7 Type Information

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

Table 2.3-6		#1	#2	#3	#4	#5
Global:	`h(1)`	0.014	0	0.013	0.071	0.001
On base:	`typeid(pa)`	0.079	0.047	0.218	0.365	0.059
On derived:	`typeid(pc)`	0.079	0.047	0.105	0.381	0.055
On VBC:	`typeid(pa)`	0.078	0.046	0.217	0.379	0.049
VBC on derived:	`typeid(pd)`	0.081	0.046	0.113	0.382	0.048

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

2.3.8 Dynamic Cast

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

Table 2.3-7		#1	#2	#3	#4	#5
Virtual call:	`px->f(1)`	0.025	0.013	0.019	0.078	0.059
Up-cast to base1:	`cast(pa,pc)`	0.007	0	0.003	0.006	0
Up-cast to base2:	`cast(pb,pc)`	0.008	0	0.004	0.007	0.001
Down-cast from base1:	`cast(pc,pa)`	0.116	0.148	0.066	0.640	0.063
Down-cast from base2:	`cast(pc,pb)`	0.117	0.209	0.065	0.632	0.070
Cross-cast:	`cast(pb,pa)`	0.305	0.356	0.768	1.332	0.367
2-level up-cast to base1:	`cast(pa,pcc)`	0.005	0	0.005	0.006	0.001
2-level up-cast to base2:	`cast(pb,pcc)`	0.007	0	0.006	0.006	0.001
2-level down-cast from base1:	`cast(pcc,pa)`	0.116	0.148	0.066	0.641	0.063
2-level down-cast from base2:	`cast(pcc,pb)`	0.117	0.203	0.065	0.634	0.077
2-level cross-cast:	`cast(pa,pb)`	0.300	0.363	0.768	1.341	0.377
2-level cross-cast:	`cast(pb,pa)`	0.308	0.306	0.775	1.343	0.288

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

- An up-cast (cast from derived class to base class) can be compiled into a simple `this` pointer adjustment, as done by implementations #2 and #5.
- A down-cast (from base class to derived class) can be quite complicated (and therefore quite expensive in terms of run-time overhead), but many cases are simple. Implementation #5 shows that a down-cast can be optimized to the equivalent of a virtual function call, which examines a data structure to determine the necessary adjustment of the `this` pointer (if any). The other implementations use simpler strategies involving several function calls (about 4, 10, 3, and 10 calls, respectively).
- Cross-casts (casts from one branch of a multiple inheritance hierarchy to another) are inherently more complicated than down-casts. However, a cross-cast could in principle be implemented as a down-cast followed by an up-cast, so one should expect the cost of a cross-cast to converge on the cost of a down-cast as optimizer technology matures. Clearly these implementations have a long way to go.

2.4 Exception Handling

Exception handling provides a systematic and robust approach to coping with errors that cannot be recovered from locally at the point where they are detected.

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

- Returning error codes
- Setting error state indicators (e.g. `errno`)
- Calling error handling functions
- Escaping from a context into error handling code using `longjmp`
- Passing along a pointer to a state object with each call

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

- Programming style
- Robustness and completeness of error handling code
- Run-time system (memory size) overheads
- Overheads from handling an individual error

Consider a trivial example:

```
double f1(int a) { return 1.0 / a; }
double f2(int a) { return 2.0 / a; }
double f3(int a) { return 3.0 / a; }

double g(int x, int y, int z)
{
    return f1(x) + f2(y) + f3(z);
}
```

This code contains no error handling code. There are several techniques to detect and report errors which predate C++ exception handling:

```
void error(const char* e)
{
    // handle error
}

double f1(int a)
{
    if (a <= 0) {
        error("bad input value for f1()");
        return 0;
    }
    else
        return 1.0 / a;
}
```

```
int error_state = 0;

double f2(int a)
{
    if (a <= 0) {
        error_state = 7;
        return 0;
    }
    else
        return 2.0 / a;
}

double f3(int a, int* err)
{
    if (a <= 0) {
        *err = 7;
        return 0;
    }
    else
        return 3.0 / a;
}

int g(int x, int y, int z)
{
    double xx = f1(x);
    double yy = f2(y);

    if (error_state) {
        // handle error
    }

    int state = 0;
    double zz = f3(z,&state);

    if (state) {
        // handle error
    }
    return xx + yy + zz;
}
```

Ideally a real program would use a consistent error handling style, but such consistency is often hard to achieve in a large program. Note that the `error_state` technique is not thread safe unless the implementation provides support for thread unique static data, and branching with `if(error_state)` may interfere with pipeline optimizations in the processor. Note also that it is hard to use the `error()` function technique effectively in programs where `error()` may not terminate the program. However, the key point here is that any way of dealing with errors that cannot be handled locally implies space and time overheads. It also complicates the structure of the program.

Using exceptions the example could be written like this:

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

```
double f1(int a)
{
    if (a <= 0)
        throw Error(1);
    return 1.0 / a;
}

double f2(int a)
{
    if (a <= 0)
        throw Error(2);
    return 2.0 / a;
}

double f3(int a)
{
    if (a <= 0)
        throw Error(3);
    return 3.0 / a;
}

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

When considering the overheads of exception handling, we must remember to take into account the cost of alternative error handling techniques.

The use of exceptions isolates the error handling code from the normal flow of program execution, and unlike the error code approach, it cannot be ignored or forgotten. Also, automatic destruction of stack objects when an exception is thrown renders a program less likely to leak memory or other resources. With exceptions, once a problem is identified, it cannot be ignored – failure to catch and handle an exception results in program termination[5]. For a discussion of techniques for using exceptions, see Appendix E of "The C++ Programming Language" [BIBREF-30].

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

We can distinguish three sources of overhead:

- ***try-block*s** Data and code associated with each *try-block* or catch clause.
- ***regular functions*** Data and code associated with the normal execution of functions that would not be needed had exceptions not existed, such as missed optimization opportunities.

[5] Many programs catch all exceptions in `main()` to ensure graceful exit from totally unexpected errors. However, this does not catch unhandled exceptions that may occur during the construction or destruction of static objects (§IS-15.3p13)

- ***throw-expressions*** Data and code associated with throwing an exception.

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

2.4.1 Exception Handling Implementation Issues and Techniques

The implementation of exception handling must address several issues:

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

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

  Because of this overlap, some implementations require that RTTI be enabled when EH is enabled.
- ***Cleanup of handled exceptions*** Exceptions which are not re-thrown must be destroyed upon exit of the catch clause. The memory for the exception object must be managed by the EH implementation.
- ***Automatic and temporary objects with non-trivial destructors*** Destructors must be called if an exception occurs after construction of an object and before its lifetime ends (§IS-3.8), even if no try/catch is present. The EH implementation is required to keep track of all such objects.
- ***Construction of objects with non-trivial destructors*** If an exception occurs during construction, all completely constructed base classes and sub-objects must be destroyed. This means that the EH implementation must track the current state of construction of an object.
- ***throw-expression*** A copy of the exception object being thrown must be allocated in memory provided by the EH implementation. The closest matching catch clause must then be found using the EH type-information. Finally, the destructors for automatic, temporary, and partially constructed objects must be executed before control is transferred to the catch clause.
- ***Enforcing exception specifications*** Conformance of the thrown types to the list of types permitted in the *exception-specification* must be checked. If a mismatch is detected, the *unexpected-handler* must be called.
- ***`operator new`*** After calling the destructors for the partially constructed object, the corresponding `operator delete` must be called if an exception is thrown during construction.

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

Implementations vary in how costs are allocated across these elements.

The two main strategies are:

- The “code” approach, where code is associated with each *try-block*, and
- The “table” approach, that uses compiler-generated static tables.

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

2.4.1.1 The "Code" Approach

Implementations using this approach have to dynamically maintain auxiliary data-structures to manage the capture and transfer of the execution contexts, plus other dynamic data-structures involved in tracking the objects that need to be unwound in the event of an exception. Early implementations of this approach used `setjmp/longjmp` to return to a previous context. However, better performance can be obtained using special-purpose code. It is also possible to implement this model through the systematic use of (compiler generated) return codes. Typical ways in which the code approach deals with the issues identified in 2.4.1 are as follows:

- ***try-block*** Save the execution environment and push a reference to catch code on EH stack at *try-block* entry.

- ***Automatic and temporary objects with non-trivial destructors*** Register each constructed object together with its destructor in preparation for later destruction. Typical implementations use a linked list structure on the stack. If an exception is thrown, this list is used to determine which objects need to be destroyed.

- ***Construction of objects with non-trivial destructors*** One well-known implementation increments a counter for each base class and subobject as they are constructed. If an exception is thrown during construction, the counter is used to determine which parts need to be destroyed.

- ***throw-expression*** After the catch clause has been found, invoke the destructors for all constructed objects in the region of the stack between the *throw-expression* and the associated catch clause. Restore the execution environment associated with the catch clause.

2.4.1.1.1 Space Overhead of the “Code” Approach

- No exception handling cost is associated with an individual object, so object size is unaffected.
- Exception handling implies a form of RTTI, which may require some increase to code size, data size or both.
- Exception handling code is inserted into the object code for each try/catch.
- Code registering the need for destruction is inserted into the object code for each stack object of a type with a non-trivial destructor.

- A cost is associated with checking the *throw-specification*s of the functions that are called.

2.4.1.1.2 Time Overhead of the "Code" Approach

- On entry to each *try-block*
    - Commit changes to variables enclosing the *try-block*
    - Stack the execution context
    - Stack the associated catch clauses
- On exit from each *try-block*
    - Remove the associated catch clauses
    - Remove the stacked execution context
- 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 or re-throw
    - Locate the corresponding catch clause (if any) – this involves some run-time check (possibly resembling RTTI checks)

      If found, then:
        - destroy the registered local objects
        - check the *exception-specification*s of the functions called in-between
        - use the associated execution context of the catch clause

      Otherwise:
        - call the *unexpected-handler*
- On entry to each catch clause
    - Remove the associated catch clauses
- On exit from each catch clause
    - Retire the current exception object (destroy if necessary)

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

There are two primary disadvantages of the "code" model:

- The associated stack and run-time costs can be high for try-block entry.
- Even when no exceptions are thrown, the bookkeeping of the exception handling stack must be performed for automatic, temporary and partially constructed objects.

That is, code unrelated to error handling is slowed down by the mere possibility of exceptions being used. This is similar to error-handling strategies that consistently check error state or return values.

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

2.4.1.2 The "Table" Approach

Typical implementations using this approach will generate read-only tables for determining the current execution context, locating catch clauses and tracking objects needing destruction. Typical ways in which the table approach deals with the issues identified in 2.4.1 are as follows:

- ***try-block*** This method incurs no run-time cost. All bookkeeping is pre-computed as a mapping between program counter and code to be executed in the event of an exception. Tables increase program image size but may be moved away from working set to improve locality of reference. Tables can be placed in ROM or, 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 run-time costs are associated with normal execution. Only in the event of an exception is it necessary to intrude on normal execution.

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

2.4.1.2.1 Space Overhead of the “Table” Approach

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

2.4.1.2.2 Time Overhead of the “Table” Approach

- On entry to each *try-block*
  - Some implementations commit changes in execution state to variables in the scopes enclosing the *try-block* – other implementations use a more sophisticated state table[6]
- On exit from each *try-block*
  - No overhead

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

- When calling regular functions
  - No overhead
- As each local and temporary object is created
  - No overhead
- On throw or re-throw
  - Using the tables, determine if there is an appropriate catch clause
    If there is, then:
    - destroy all local, temporary and partially constructed objects that occur between the *throw-expression* and the catch clause
    - check that the exception honors the *exception-specification*s of functions between the `throw` and the *handler*
    - transfer control to the catch clause

    Otherwise:
    - call the *unexpected-handler*
- On entry to each catch clause
  - No overhead
- On exit from each catch clause
  - No overhead

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

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

One vendor reported a code and data space impact of about 15% for the generated tables. It is possible to do better, as this vendor had no need to optimize for space.

2.4.2 Predictability of Exception Handling Overhead

2.4.2.1 Prediction of throw/catch Performance

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

For this reason current exception handling implementations may be unsuitable for some applications. However, if the call tree can be statically determined, and the table method of EH implementation is used, it is possible to statically analyze 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 analyzed to determine their contribution to the cost, and the whole sequence of events aggregated into a single cost domain (worst-case and best-case, unbounded, indeterminate). Such analysis does not differ in principle from current time estimating methods used for non-exception code.

One of the reservations expressed about EH is the unpredictable time that may elapse after a *throw-expression* and before control passes to the catch clause while automatic objects are being destroyed. It should be possible to determine accurately 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 function is determined.

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

2.4.2.2 Exception Specifications

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

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

will in a straightforward implementation generate code roughly equivalent to:

```
void f(int x)
{
    try {
        // whatever
    } catch (A&) {
        throw;
    } catch (B&) {
        throw;
    } catch (...) {
        unexpected();
    }
}
```

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

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

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

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

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

There is of course no requirement that a compiler performs this optimization. However, a compiler intended for high-performance use is likely to perform it.

2.5 Templates

2.5.1 Template Overheads

A class template or function template will generate a new instantiation of code each time it is specialized with different *template-parameter*s. This can lead to an unexpectedly large amount of code and data[7]. 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 specialization of `std::list<T*>`, for some type `T`, was compared with a second program instantiating a single instance of `std::list<T*>` for 100 different types `T`. These programs were compiled with a number of different compilers and a variety of different compiler options. The results varied widely, with one compiler producing code for the second program that was over 19 times as large as the first program; and another compiler producing code for the first program that was nearly 3 times as large as the second.

The optimization here is for the compiler to recognize that while there may be many specializations with different types, at the level of machine code-generation, the specializations 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 optimization automatically, the optimization can also be performed by the Standard Library implementation or by the application programmer.

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

If the compiler supports *partial specialization* and member-function templates, the library implementer can provide *partial specialization*s of containers of pointers to a single underlying implementation that uses `void*`. This technique is described in The C++ Programming Language 3rd edition [BIBREF-30].

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

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

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 following the Smalltalk tradition.

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

In the NIH Class 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[8]. Most of the functions were declared virtual, and so had to be overridden by derived classes[9]. 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.

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

[9] 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 run-time, it typically includes the functions from all the preceding levels of the hierarchy for each class in the executable program. This can lead to code bloat without templates.

There are other performance disadvantages to inheritance-based collection classes:

- Primitive types cannot be inserted into the collections. Instead, these must be replaced with classes in the `Object` hierarchy, which are programmed to have similar behavior to primitive arithmetic types, such as `Integer` and `Float`. This circumvents processor optimizations for arithmetic operations on primitive types. In addition, it is difficult to duplicate exactly the behavior of primitive types through class member functions and operators.

- Because C++ has compile-time type checking, providing type-safe containers for different contained data types requires code to be duplicated for each type. Type safety is the same reason that template containers are instantiated multiple times. To avoid this duplication of code, the NIHCL collections hold pointers to a generic type – the base `Object` class. However, this is not type-safe, and requires run-time checks to ensure objects are type-compatible with the contents of the collections. It also leads to many more dynamic memory allocations, which can hinder performance. 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. 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 containers do not impose such requirements on their contents10.
- The C++ Standard Library establishes a set of principles for combining data structures and algorithms from different sources. Inheritance-based libraries from different vendors – where the algorithms are implemented as member functions of the containers – can be difficult to integrate and difficult to extend.

2.6 Programmer Directed Optimizations

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

Compilers typically use a heuristic process in optimizing 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 optimize a program is to use space- and time-efficient data structures and algorithms. For example, changing a sequential search routine to a binary search

[10] A class used in a Standard container must be `Assignable` and `CopyConstructible`; often it additionally needs to have a default constructor and implement `operator ==` and `operator <`.

will reduce the average number of comparisons required to search a sorted N-element table from about N/2 to just $\log_2 N$; for N=1000, this is a reduction from 500 comparisons to 10. For N=1,000,000, the average number of comparisons is 20.

Another example is that `std::vector` is a more compact data structure than `std::list`. A typical `std::vector<int>` implementation will use about three words plus one word per element, whereas a typical `std::list<int>` implementation will use about two words plus three words per element. That is, assuming `sizeof(int)==4`, a standard `vector` of 1,000 `int`s will occupy approximately 4,000 bytes, whereas a `list` of 1,000 `int`s will occupy approximately 12,000 bytes. Thanks to cache and pipeline effects, traversing such a vector will be much faster than traversing the equivalent list. Typically, the compactness of the vector will also assure that moderate amounts of insertion or erasure will be faster than for the equivalent list. There are good reasons for std::vector being recommended as the default standard library container[11].

The C++ Standard Library provides several different kinds of containers, and guarantees how they compare at performing common tasks. For example, inserting an element at the end of an `std::vector` takes constant time (unless the insertion forces a memory reallocation), but inserting one at the beginning or in the middle takes linear time increasing with the number of elements that have to be moved to make space for the new element. With an `std::list` on the other hand, insertion of an element takes constant time at any point in the collection, but that constant time is somewhat slower than adding one to the end of a vector. Finding the N^{th} element in an `std::vector` involves a simple constant-time arithmetic operation on a random-access iterator accessing contiguous storage, whereas an `std::list` would have to be traversed one element at a time, so access time grows linearly with the number of elements. A typical implementation of `std::map` maintains the elements in sorted order in a red-black tree structure, so access to any element takes logarithmic time. Though not a part of the C++ Standard Library (at the time this is written), a `hash_map` is capable of faster lookups than an `std::map`, but is dependent on a well-chosen hash function and bucket size. Poor choices can degrade performance significantly.

Always measure before attempting to optimize – it is very common for even experienced programmers to guess incorrectly about performance implications of choosing one kind of container over another. Often performance depends critically on the machine architecture and the quality of optimizer used.

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

[11] The recommendation comes from Bjarne Stroustrup in [BIBREF-30] and from Alex Stepanov in private correspondence.

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

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

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

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

can result in hidden code being executed. In this case, the declaration of `v1` implicitly invokes the `class X`'s default constructor to initialize the object `v1`. The purpose of constructors and destructors is to make it impossible to forget mandatory processing at the beginning and end of an object's lifetime. Depending on the class design, proper initialization may involve memory allocations or system calls to acquire resources

Although declaring a user-defined variable in C does not implicitly invoke a constructor, it is important to remember that the object must still be initialized and that code would have to be explicitly called by the programmer. Resources would also have to be explicitly released at the appropriate time. The initialization and release code is more visible to the C programmer, but possibly less robust because the language does not support it automatically.

Understanding what a C++ program is doing is important for optimization. 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 E:) provide more extensive guidance (e.g. [BIBREF-17]), but the following provides some suggestions for writing more efficient code:

- Shift expensive computations from the most time-critical parts of a program to the least time-critical parts (often, but not always, program start-up). Techniques include lazy evaluation and caching of pre-computed values. Of course, these strategies apply to programming in any language, not just C++.

- In constructors, prefer initialization of data members to assignment. If a member has a default constructor, that constructor will be called to initialize the member before any assignment takes place. Therefore, an assignment to a member within the constructor body can mean that the member is initialized as well as assigned to, effectively doubling the amount of work done.

- As a general principle, do not define a variable before you are ready to initialize it. Defining it early results in a constructor call (initialization) followed by an assignment of the value needed, as opposed to simply constructing it with the value needed. Apart from performance issues, there is then no chance that the variable can be used before it has received its proper initial value.

- Understand how and when the compiler generates temporary objects. Often small changes in coding style can prevent the creation of temporaries, with consequent benefits for run-time speed and memory footprint. Temporary objects may be generated when initializing objects, passing parameters to functions, or returning values from functions.

- 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[12].

[12] Of course if the argument type and the expression type differ, a temporary variable may be created by the compiler.

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

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

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

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

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

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

- Unless you need automatic type conversions, declare all one-argument constructors[13] `explicit`. This will prevent them from being called accidentally. Conversions can still be done when necessary by explicitly stating them in the code, thus avoiding the penalty of hidden and unexpected conversions.

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

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

  Better yet, use a library that eliminates need for the rewrite using `+=`. Such libraries, which are common in the numeric C++ community, usually use function objects and expression templates to yield uncompromisingly fast code from conventional-looking source.

[13] 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.

- Use the return value optimization to give the compiler a hint that temporary objects can be eliminated. This technique enables the compiler to use the memory for a function's return value to hold a local object of the same type that would otherwise have to be copied into the return value location, thus saving the cost of the copy. This is usually signalled by inserting constructor arguments into the return statement:

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

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

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

    return tmp;     // copies tmp to the return value, which is
                    //  then copied into the receiving variable
}
```

  However, with recent improvements in compiler technology, modern compilers may optimize this code in a similar manner.

- Prefer the prefix versus the postfix forms for increment and decrement 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, which may be structures containing pointers to nodes, creating temporary copies may be expensive when compared to built-in `int`s.

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

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

- Sometimes it is helpful to "widen" the interface for a class with functions that take different data types to prevent automatic conversions (such as adding an overload on `char *` to a function which takes an `std::string` parameter). The numerous overloads for operators `+`, `==`, `!=`, and `<` in the `<string>` header are an example of such a "fat" interface[14]. 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 class `std::string` is not a lightweight component. Because it has a lot of functionality, it comes with a certain amount of overhead. And because the constructors of the Standard Library exception classes described in Clause 19 of IS 14882 (although not their base class `std::exception`) require an argument of type `std::string`, this overhead may be included in a program inadvertently.In many applications, strings are created, stored, and referenced, but never changed. As an extension, or as an optimization, it might be useful to create a lighter-weight, unchangeable string class.

- Reference counting is a widely used optimization 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[15].

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

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

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

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

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

- On the other hand, if a function such as `myList.end()` is so simple that it can be inlined, the rewrite may not yield any performance advantage over what a good compiler would produce for the original code.Some implementations of std::list<T>::size() have linear complexity rather than constant complexity. This latitude is allowed by the Standard container requirements specified in §IS-23.1.Calling such a function inside a loop would result in quadratic

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

[15] Of course, if optimization for space is more important than optimization for time, reference counting may still be the best choice.

behavior. For the same reason it is better to use `if(myList.empty())` rather than `if(MyList.size()==0)`.

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

  ```
  class X
  {
  private:
      int     value_;
      double* array_;  // pointer to array of [size] doubles
      size_t  size_;
  public:
      int     value()  { return value_; }
      size_t  size()   { return size_;  }
      // ...
  };
  ```

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

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

- The use of dynamic binding and virtual functions has some overhead in both memory footprint and run-time performance. This overhead is minor, especially when compared with alternative ways of achieving run-time polymorphism (§2.3.3). A bigger factor is that virtual functions may interfere with compiler optimizations and inlining.

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

- Many programs written in some conventional object-oriented styles are very slow to compile, because the compiler must examine hundreds of header files and tens of thousands of lines of code. However, code can be structured to minimize re-compilation after changes. This typically produces better and more maintainable designs, because they exhibit better separation of concerns.

  Consider a classical example of an object-oriented program:

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

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

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

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

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

  The protected members are likely to depend on implementation details that the clients of `Shape` would rather not have to depend on. For example, much code using a `Shape` will be logically independent of the definition of `Color`, yet the presence of `Color` in the definition of `Shape` makes all of that code dependent on the header files defining the operating system's notion of color, often requiring that the client code is recompiled whenever such header files are changed.

When something in the protected part changes, client code using `Shape` has to be recompiled, even though only implementers of derived classes have access to the protected members. Thus, the presence of "information helpful to implementers" in the base class – which also acts as the interface to users – is the source of several problems:

- Instability in the implementation,
- Spurious recompilation of client code (when implementation information changes), and
- Excess inclusion of header files into client code (because the "information helpful to implementers" needs those headers).

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

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

```
class Shape {
public:  // interface to users of Shapes
    virtual void draw() const = 0;
    virtual void rotate(int degrees) = 0;
    virtual Point center() const = 0;
    // ...
    // no data
};

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

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

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

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

```
class Shape {
public:   // interface to users of Shapes
    virtual void draw() const = 0;
    virtual void rotate(int degrees) = 0;
    virtual Point center() const = 0;
    // ...
    // no data
};

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

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

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

- Another technique for ensuring better separation between parts of a program involves an interface object holding a single pointer to an implementation object. This is often called “the PIMPL” (Pointer to IMPLementation[16]) idiom. For example:

```
// Interface header:
class Visible {
    class Hidden;

    Hidden* pImpl;
public:
    void fcn1();
    ...
};

// Implementation source:
class Visible::Hidden {
    ...
public:
    void fcn1_impl();
    ...
};

void Visible::fcn1() { pImpl->fcn1_impl(); }
```

- Use function objects[17] with the Standard Library algorithms rather than function pointers. Function pointers defeat the data flow analyzers of many optimizers, but function objects are passed by value and optimizers can easily handle inline functions used on objects.
- A function with one or more default arguments can be called without specifying its full argument list, relying on the compiler to insert the default values. This necessarily requires the constructor to create a temporary object for each default value. If the construction of that temporary is expensive and if the function is called several times, it can be worth while to construct the default argument value somewhere and use that value in each call. For example:

```
class C
{
public:
    C(int i) { ... }       // possibly expensive
    int mf() const;
    // ...
};

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

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

[16] Also known as the "Cheshire Cat” idiom.

[17] Objects of a class type that has been designed to behave like a function, because it defines operator () as a member function. Often all the member functions of such a type are defined inline for efficiency.

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

- When programming "close to the metal", such as for accessing low-level hardware devices, some use of assembly code may be unavoidable. The C++ *asm declaration* (§IS-7.4) enables the use of assembly code to be minimized.

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

  However, a compiler is typically unaware of the semantics of inlined assembly instructions. Thus, use of inlined assembly instructions can defeat other important optimizations such as common sub-expression elimination and register allocation. Consequently, inline assembly code should be used only for operations that are not otherwise accessible using C++.

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

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

class Factorial<1> {
public:
    static const int value = 1;
};
```

  Using this class template[18], the value **`N!`** is accessible at compile-time as `Factorial<N>::value`.

  As another example, the following class and function templates can be used to generate inline code to calculate the dot product of two arrays of numbers:

[18] Within limitations, remember that if an `int` is 32-bits, the maximum `N` can be is just 12.

```
// Given a forward declaration:
template <int Dim, class T>
  struct dot_class;

// a specialized base case for recursion:
template <class T>
  struct dot_class<1,T> {
    static inline T dot(const T* a, const T* b)
      { return *a * *b; }
  };

// the recursive template:
template <int Dim, class T>
  struct dot_class {
    static inline T dot(const T* a, const T* b)
      { return dot_class<Dim-1,T>::dot(a+1,b+1) +
                 *a * *b; }
  };

// ... and some syntactic sugar:
template <int Dim, class T>
  inline T dot(const T* a, const T* b)
    { return dot_class<Dim,T>::dot(a, b); }

// Then
int product = dot<3>(a, b);

// results in the same (near-)optimal code as
int product = a[0]*b[0] + a[1]*b[1] + a[2]*b[2];
```

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

- Templates provide compile-time polymorphism, wherein type selection does not incur any run-time penalty. If appropriate to the design, consider using class templates as interfaces instead of abstract base classes. For some designs it may be appropriate to use templates which can provide compile-time polymorphism, while virtual functions which provide run-time polymorphism may be more appropriate for others.

  Templates have several useful properties: they impose no space or code overhead on the class used as a template argument, and they can be attached to the class for limited times and purposes. If the class does not provide the needed functionality, it can be defined externally through template specialization. If certain functions in the template interface are never used for a given class, they need not be defined because they will not be instantiated.

  In the example below, the `talk_in_German()` function in the "interface" is only defined for `class CuckooClock`, because that is the only object for which it is needed. Invoking `talk_in_German()` on an object of a different type results in a compiler diagnostic:

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

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

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

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

class SilentClock {
    // doesn't talk
};

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

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

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

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

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

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

    // use objects in contexts which do not require talking
    // ...
    Talkative<Dog> td(aDog);
    td.talk();                                    // woof woof

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

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

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

    return 0;
}
```

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

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

  For example (from §IS-14.7.2¶2):

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

template<class T> void sort(Array<T>& v) { /* ... */ }
template void sort(Array<char>&);
```

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

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

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

- Compiler options are usually extra arguments or switches, which pass instructions to the compiler. Some of these instructions are related to performance, and control how to:
    - Generate executable code optimized for a particular hardware architecture.
    - Optimize the translated code for size or speed. Often there are sub-options to exercise finer control of optimization techniques and how aggressively they should be applied.
    - Suppress the generation of debugging information, which can add to code and data size.
    - Instrument the output code for run-time profiling, as an aid to measuring performance and to refine the optimization strategies used in subsequent builds.
    - Disable exception handling overhead in code which does not use exceptions at all.
    - Control the instantiation of templates.
- `#pragma` directives allow compilers to add features specific to machines and operating systems, within the framework of Standard C++. Some of the optimization-related uses of `#pragma` directives are to:
    - Specify function calling conventions (a C++ linkage-specification can also be used for this purpose).
    - Influence the inline expansion of code.
    - Specify optimization strategies on a function-by-function basis.

  - Control the placement of code or data into memory areas (to achieve better locality of reference at run-time).
  - Affect the layout of class members (through alignment or packing constraints, or by suppressing compiler-generated data members).

  Note that `#pragma`s are not standardized and are not portable.

- Linking to static libraries or shared libraries, as appropriate. Linker options can also be used to control the amount of extra information included in a program (e.g., symbol tables, debugging formats).
- Utilities for efficiently allocating small blocks of memory. These may take the form of system calls, `#pragma`s, compiler options, or libraries.
- Additional programs:
  - Many systems have a utility program[19] to remove the symbol table and line number information from an object file, once debugging is complete, or this can often be done at link-time using a linker specific option. The purpose is to reduce file storage and, in some cases, memory overhead.
  - Some systems have utilities[20] and tools to interpret profiling data and identify run-time bottlenecks.
- Sometimes, minimizing compile-time is important. When code is being created and debugged, suppressing optimization may enable the compiler to run faster.

  The most effective technique for reducing compile-time relies on reducing the amount of code to be compiled. The key is to reduce coupling between different parts of a program so as to minimize the size and number of header files needed in most translation units. Some techniques for accomplishing this include the use of abstract base classes as interfaces and the PIMPL idiom, as discussed above.

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

[19] For instance the ‘strip’ utility which is part of the Software Development Utilities option in the IEEE Posix/Open Group Unix /ISO/IEC 9945:2002 specifications.

[20] For instance the ‘prof’ utility which is not part of the Posix/Unix Standard, but is available on many systems.

- Reading and parsing header code takes time. Years ago, the common practice was to #include as few headers as possible, so that only necessary symbols were declared. But with technology to pre-compile headers, build time may be reduced by using a single header in each translation unit which #includes everything needed for the program.

  Well-designed headers will usually protect their contents against multiple inclusion by following this pattern:

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

  The header is said to be "idempotent" because, regardless of how many times it is #included, it has the effect of being #included only once. If the compiler provides the "idempotent guard" optimization, it will record in an internal table the fact that this header is guarded by a macro. If this header is subsequently #included again, and the macro THIS_HEADER_H still remains defined, then the compiler can avoid accessing the header contents.

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

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

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

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

- Input/output can be a performance bottleneck in C++ programs. By default, the standard iostreams (cin, cout, cerr, clog, and their wide-character counterparts) are synchronized with the C stdio streams (stdin, stdout, stderr), so that reads from cin and stdin, or writes to cout and stdout, can be freely intermixed. However, this coupling has a performance cost, because of the buffering in both kinds of streams. In the pre-standard "classic" iostreams library, unsynchronized mode was the default.

  If there is no need for a program to make calls to both standard C streams and C++ iostreams, synchronization can be turned off with this line of code:

  ```
  std::ios_base::sync_with_stdio(false);
  ```

  If any input or output operation has occurred using the standard streams prior to the call, the effect is implementation-defined. Otherwise, called with a false argument, it allows the standard streams to operate independently of the standard C streams (§IS-27.4.2.4).

  Another standard default is to flush all output to cout before reading from cin, for the purpose of displaying interactive prompts to the application user.

If this synchronized flushing is not needed, some additional performance can be gained by disabling it:

```
std::cin.tie(0);
```

- An empty body in a class constructor, or an unwritten default constructor, can invoke an amount of code which may be surprising at first glance. This is because all member subobjects and base subobjects with constructors must be initialized as part of the class construction. Compiler-generated default constructors are inline member functions, as are function definitions written within the body of the class definition. Therefore an innocent-looking `{}` can not be assumed to produce trivial machine code:

```
class X
{
   A a;
   B b;
   virtual void f();
};

class Y : public X
{
   C c;
   D d;
};

class Z : public Y
{
   E e;
   F f;
public:
   Z() { }
};
Z z;
```

The constructor for z, itself only empty brackets, causes the compiler to generate code to initialize all of the base classes and all data members, thus invoking defined or compiler-generated constructors for classes X, A, B, Y, C, D, E, and F. If all of these are inline and non-trivial, a substantial block of machine code can be inserted at this point in the program. It will also initialize the virtual table. Therefore it is important to know what functions will be called when an object is initialized and to make active decisions on whether that code should be placed inline. Empty-bracket functions are often used for destructors as well, but a similar analysis of the costs should be performed before making them inline.

3 Creating Efficient Libraries

This section discusses techniques which can be used in creating any library. These techniques are discussed in the context of an implementation of part of the C++ Standard Library.

3.1 The Standard IOStreams *Library – Overview*

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

The efficiency problems come in several forms:

3.1.1 Executable Size

Typically, using anything from the *IOStreams* library drags in a huge amount of library code, much of which is not actually used. The principal 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 any facet from which one or more functions are used. This is discussed in 3.2.2.

3.1.2 Execution Speed

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

3.1.3 Object Size

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

- Compilation time
- Higher memory usage when actually using the corresponding features
- Execution speed

This is discussed in 3.2.6.

3.1.4 Compilation Time

A widespread approach for coping with the ubiquitous lack of support for exported templates is to include the template implementations in the headers. This can result in very long compile and link times if, for example, the *IOStreams* headers are included, and especially if optimizations 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.

3.2 Optimizing Libraries – Reference Example: ”An Efficient Implementation of Locales *and* IOStreams”

The definition of *Locales* in the C++ Standard (§IS-22) seems to imply a pretty inefficient implementation. However, this is not true. It is possible to create efficient implementations of the *Locales* library, both in terms of run-time efficiency and executable size. This does take some thought and this report 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 optimization 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 space). Depending on the requirements, the library writer, or possibly even the library user, has to choose which optimizations are most appropriate.

3.2.1 Implementation Basics for *Locales*

Before going into the details of the various optimizations, 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 optimizations should be directed.

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

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

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

Individual facets are identified using an ID, more precisely an object of type `std::locale::id,` which is available as a static data member in all base classes defining a facet. A facet is a class derived from `std::locale::facet` which has a publicly accessible static member called `id` of type `std::locale::id` (§IS-22.1.1.1.2¶1). Although explicit use of a locale's facets seems to use a type as an index (referred to here as **F**), 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.

In summary, 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 pointers to facets.
- Instead of protecting each facet object with its own mutex lock, it is 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 relatively rare (only when a modified locale object is necessary is there a need for locking) and it is unlikely that this global lock remains held for extended periods of time. 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 are available, the reference count alone is sufficient, because the only operations needing multi-threading protection are incrementing and decrementing of 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 optimization" (§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, as stated earlier, it is reasonable to envision a locale object as being a reference counted pointer to some internal representation containing an array of reference counted facets. Whether this is actually implemented so as to reduce run-time costs 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 earlier Committee Draft (CD) versions quite significantly. If a facet is not found in a locale object, it is not available for this locale. In earlier CDs, if a facet was not found in a given locale, then the global locale object was searched. The definition chosen for the Standard was made so that the Standard could be more efficiently implemented – to determine whether a facet is available for a given locale object, a simple array lookup is sufficient. Therefore, the functions `std::use_facet()` and `std::has_facet()` could be implemented something like this:

```
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]);
}
```

These versions of the functions are 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 in the array for each facet possibly used by the program, 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 store 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;
}
```

However, this approach 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 one, producing smaller code, is to include only those facets that are really used, perhaps by providing specialized versions of `use_facet()` and `has_facet()`. 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<ctype<char> 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);
}
```

This is just one example of many possible implementations for a recurring theme. A facet is created only if it is indeed referenced from the program. This particular approach is suitable in implementations where exceptions cause a run-time overhead only if they are 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 initialize 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 initialization include the use of static initializers to register used facets. In this case, the specialized versions of the function `use_facet()` would be placed into individual object files together with an object whose static initialization registers the corresponding facet. This approach implies, however, that the function `use_facet()` is implemented out-of-line, possibly causing unnecessary overhead both in terms of run-time and executable size.

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

A better approach to avoid linking in unused virtual functions would be to change the compiler so that it generates appropriate symbols which enable 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.

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 non-virtual 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 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.

Editor's Note:	*It isn't completely clear to me how this works. Is it possible to illustrate it with a small amount of code, or a footnote with the compiler and linker options for some common implementation?*

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 approaches to work with typical linkers, a modified compiler and linker provide a much-preferred solution, unfortunately one which is often outside the scope of library implementers.

In many cases, most of the normally visible code bloat can be removed using the two techniques discussed above, 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 optimizations 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 essentially 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 are simply cached 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 setup. 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, such as 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 approaches, 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-time 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 technical report however, is that some of the techniques presented below rely on certain aspects that are related to the compilation process.

The first technique that improves compile-time is the liberal use of forward 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 as a return or argument type in a function declaration. Likewise, a forward declaration is sufficient if only a pointer to a class type is used as a class member (see the discussion of the PIMPL idiom in Section 2.6).

Looking at the members `imbue()` and `getloc()` of the class `std::ios_base`, it would seem that the `<ios>` header is required to include `<locale>` simply for the definition of `std::locale`, because apparently an `std::ios_base` object stores a locale object in a member variable. This is not required! Instead, `std::ios_base` could store a pointer to the locale's internal representation and construct an `std::locale` object on the fly. Thus, there is no necessity 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 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 specialization 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 preferred 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 *IOStreams* 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 implementer, 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 optimizations enabled. The reason for this is simple: with many inline functions, the compiler passes huge chunks of code to the optimizer, which then has to work extra hard to improve them. Bigger chunks provide better optimization possibilities, but they also cause significantly longer compile-times due to the non-linear increase in the complexity of the optimization 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 have some code from just one other function in common.

Another reason for having the *IOStreams* and *Locales* 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 omit unused code by means of 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 have any real effect. For example, the code for padding formatted results has no 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 is ever used. As in all other aspects of C++, it is reasonable to avoid paying a cost in code size or performance for any feature which is not used.

The basic idea for coping with this is to avoid unnecessary overheads where possible by providing multiple implementations of some functions. 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 optimization 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 version, the linker has to be instructed 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 value, 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 must 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 *IOStreams* functions can call the formatting functions directly, bypassing the retrieval of the corresponding facet and associated virtual function call – indeed, bypassing all code related to locales – thus avoiding any 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 simplest test is to check for use of 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 create a different locale object. Even then, if another locale object is constructed, it may not necessarily be used with the *IOStreams*. Only if the global locale is changed, or one of `std::ios_base::imbue()`, `std::basic_ios<...>::imbue()`, or `std::basic_streambuf<...>::imbue()` is ever called, can the streams be affected by the non-default locale object. Although this is somewhat more complex to determine, it is still feasible. There are other approaches 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 invoking `callback` functions is needed (only if `callback`s are ever registered, is this necessary); etc.

In order for the linker to decide which functionality is used by the application, it mustfollow a set of "rules" provided by the library implementer to exclude functions. It is important to base these rules only on the application code, to avoid unnecessary restrictions imposed by unused Standard Library code. However, this results in more, and more complex, rules. To determine which functionality is used by the application, the unresolved symbols referenced by the application code are examined. This requires that any function mentioned in a "rule" is indeed unresolved and results in the corresponding functions being non-inline.

There are 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 a clever compiler to eliminate code for branches that it 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 cannot be inline when they are used to exclude a simple implementation of the function `std::ios_base::width()`. This might result in less efficient and sometimes even larger code (for simple functions the cost of calling the function can be bigger than the actual function). See 3.2.7 for an approach to avoiding this problem.

The same approach can be beneficial to other libraries, and to areas of the Standard C++ library other than *IOStreams* and *Locales*. In general, the library interface can be simplified by choosing among similar functions applicable in different situations, while still retaining the same efficiency. However, this techniqueis not applicable to all situations and should be used carefully where appropriate.

3.2.6 Object Organization

A typical approach to designing 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 *IOStreams*, 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` subobject.
- Formatting parameters like the `width()` and the `precision()` are stored in `std::streamsize` objects.

- An std::locale subobject (or some suitable reference to its internal representation).
- The pword() and iword() lists.
- A list of callbacks.
- The error flags and exception flags are stored in objects of type std::ios_base::iostate. Since each of these has values representable in just three bits, they may be folded into one word.
- The fill character used for padding.
- A pointer to the used stream buffer.
- A pointer to the tie()ed std::basic_ostream.

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 in a given program 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 callbacks, and is not tie()ed to anything. In such a case – which is not unlikely in an embedded application – it might even need no members at all, and operate by simply sending string literals to its output.

A different organization could be to use an array of unions and the pword()/iword() mechanism to store the data. Each of the pieces of data listed above is given an index in an array of unions (possibly several pieces can share a single 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 a pointer to the locale's internal representation.

Depending on the exact organization, the approach will show different run-time characteristics. For example, the easiest approach for assigning indices is to do it on the fly when the corresponding data are initialized 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 initialized. In this case, there would be no need to check for out-of-bound access to the *IOStreams* features. However, this initialization 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). However, this assumption is not necessarily correct. If the source code is available, the Standard library can also be recompiled.

This leads to the “two phase” building of an application: a first phase, the application is compiled against a "normal", fully-fledged implementation. The resulting object files are automatically analyzed for features actually used by looking at the unresolved references. The result of this analysis is some configuration information (possibly 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 *IOStreams* features are used, the stream objects can be organized to include members for exactly those features. Thus, it is not necessary to use a lookup in a dynamically allocated array of facets, possibly using a dynamically assigned index, if the full flexibility of the *IOStreams* and *Locales* architecture is not used by the current application. 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).

4 Using C++ in Embedded Systems

4.1 ROMability

For the purposes of this technical report, 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 that have constraints on available memory, where code and data must be stored in physical ROM whenever possible.
- Modern operating systems that 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 portions of the program can be placed in a shared, 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 in whose environment memory is scarce, it is critical that compilers identify strictly ROMable objects and allocate ROM, not RAM, area for them. For hosted systems, the requirement to share ROMable information is not as critical, but there are performance advantages to hosted programs as well, if 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.

4.1.1 ROMable Objects

Most constant information is ROMable. Obvious candidates for ROMability are objects of static storage duration that are declared `const` and have constant initializers, but there are several other significant candidates too.

Objects which are not declared `const` can be modified;, consequently they are not ROMable. But these objects may have constant initializers, and those initializers 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 as `switch` statements, and also places where ROMability can be applied to intermediate entities which are not so obvious.

4.1.1.1 User-defined Objects

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

- An aggregate (§IS-18.5.1) object with static storage duration (§IS-3.7.1) whose initializers 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 tablesize = 48
  double table[tablesize];  // table has space for 48 doubles
  ```

  However, if such an object is used for initialization or assignment of pointer or reference variables (by explicitly or implicitly having its address taken), it requires storage space and is ROMable. For example:

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

- String literals:

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

  ```
  const char * const s1 = "abc";  // both s1 and abc are ROMable
  char s2[]             = "def";  // s2 is not ROMable
  ```

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

  ```
  const char* s1 = "abc";  // only one copy of abc needs
  const char* s2 = "abc";  // to exist, and it is ROMable
  ```

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

  ```
  const char* s1 = "Hello World";
  const char* s2 = "World";

  // Could be considered to be implicitly equivalent to:
  const char* s1 = "Hello World";
  const char* s2 = s1 + 6;
  ```

4.1.1.2 Compiler-generated Objects

- Jump tables for `switch` statements:

  If a jump table is generated to implement a `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[21] ROMable.

- Type identification tables:

  When a table is generated to identify RTTI types, the table is usually[22] ROMable.

- Exception tables:

  When exception handling is implemented using static tables, the tables are usually[23] ROMable.

- Reference to constants:

  If a constant expression is specified as the initializer 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 & a = tmp;
  ```

  If `a` is declared elsewhere as an extern variable, or if its address is taken, then space must be allocated for it. If this happens, `a` is also ROMable. Otherwise, the compiler may substitute a direct reference to `tmp` (more accurately, the address of `tmp`) anywhere `a` is used.

[21] For some systems, virtual function tables may not be ROMable if they are dynamically linked from a shared library

[22] For some systems, RTTI tables may not be ROMable if they are dynamically linked from a shared library

[23] For some systems, exception tables may not be ROMable if they are dynamically linked from a shared library

- Initializers for aggregate objects with automatic storage duration:

  If all the initializers for an aggregate object that has automatic storage duration are constant expressions, a temporary object that has the value of the constant expressions and code that copies the value of the temporary object to the aggregate object may be generated. This temporary object is 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 a = tmp;
  }
  ```

  Thus, the instruction code for initializing 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 = read_some_value();
      a += 1.0;
  }

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

4.1.2 Constructors and ROMable Objects

In general, `const` objects of classes with constructors must be dynamically initialized. However, in some cases compile-time initialization could be performed if static analysis of the constructors resulted in constant values being used. In this case, the object could be ROMable. Similar analysis would need to be performed on the destructor.

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

Even if an object is not declared `const`, its initialization "pattern" may be ROMable, and can be bitwise copied to the actual object when it is initialized. For example:

```
class A {
    int a;
    char* p;
public:
    A() : a(7) { p = "Hi"; }
};
A not_const;
```

In this case, all objects are initialized to a constant value (i.e. the pair `{7, "Hi"}`). This constant initial value is ROMable, and the constructor could perform a bitwise copy of that constant value.

4.2 Hard Real-Time Considerations

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

While it may not be possible to perform this analysis in the general case, it is possible for a detailed analysis to be worked out when the details of the specific architecture are well understood.

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

4.2.1 C++ Features for which Accurate Timing Analysis is Straightforward

4.2.1.1 Templates

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

4.2.1.2 Inheritance

4.2.1.2.1 Single Inheritance

Converting a pointer to a derived class to a pointer to base class[24] will not introduce any run-time overhead in most implementations (§2.3). If there is an overhead (in very few implementations), it is a fixed number of machine instructions (typically one) and its speed can easily be determined with a test program. This is a fixed overhead; it does not depend on the depth of the derivation.

4.2.1.2.2 Multiple Inheritance

Converting a pointer to a derived class to a pointer to base class might introduce run-time overhead (§2.3.5). This overhead is a fixed number of machine instructions (typically one).

4.2.1.2.3 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 (§2.3.6). This overhead is typically a fixed number of machine instructions.

4.2.1.3 Virtual functions

If the static type of an object can be determined at compile time, Ccalling a virtual function may be no more expensive than calling a non-virtual member function. If the type must be dynamically determined at runtime, the overhead will typically be a fixed number of machine instructions (§2.3.3).

4.2.2 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. The run-time overhead of these features is often quite small, and even in the real-time parts of the program, there may be a number of CPU cycles available to spend. If the real-time task is complex, a clean structure that allows for an easier overall timing analysis is often better than hand-optimized but complicated code – as long as the former is fast enough. The hand-optimized code might run faster but is in most cases more difficult to analyze correctly, and the features mentioned below often allow for clearer designs.

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

4.2.2.1 Dynamic Casts

In most implementations, `dynamic_cast<…>` from a pointer (or reference) to base class to a pointer (or reference) to derived class (i.e. a down-cast), 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 (§2.3.8).

As an alternate option to using `dynamic_cast`s, consider using the `typeid` operator. This is a cheaper way to check for the target's type.

4.2.2.2 Dynamic Memory Allocation

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

The most obvious way to avoid dynamic memory allocation is to preallocate the memory – either statically at compile- (or more correctly link-) time or during the general setup phase of the system. For deferred initialization, preallocate raw memory and initialize it later using *new-placement* syntax (§IS-5.3.4¶11).

If the real-time code really needs dynamic memory allocation, use an implementation for which all the implementation details are known. The best way to know all the implementation details is to write a custom memory allocation mechanism. This is easily done in C++ by providing class-specific `operator new` and `delete` functions or by providing an Allocator template argument to the Standard Library containers.

But in all cases, if dynamic memory allocation is used, it is important to ensure that memory exhaustion is properly anticipated and handled.

4.2.2.3 Exceptions

Enabling exceptions for compilation may introduce overhead on each function call (§2.4). In general, it is not so difficult to analyze the overhead of exception handling as long as no exceptions are thrown. Enable exception handling for real-time critical programs only if exceptions are actually used. 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. there is no need to handle any more input from a device when a connection has broken down).

An overview of alternatives for exception handling is given in 2.4. 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. As long as no exceptions are thrown a long way (i.e. there are only a few nested function calls between the *throw-expression* and the *handler*), it might even reduce run-time costs.

4.2.3 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 by writing a simple program that includes just this specific feature and then looking at the created code. In general, for those simple cases, optimization should not make a difference. But, for example, 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. Therefore it is important to ensure that the program really tests what it needs to test.

For the more complex cases, testing the timing is not so easy. Compiler optimization can make a big difference, and a simple test case might produce completely different machine code than the real production code. It is important to thoroughly know the details of the specific implementation in order to test those cases. Given this information, it is normally possible to write test programs which produce code from which the correct timing information may be derived.

5 Hardware Addressing Interface

As C language implementations have matured over the years, various vendor-specific extensions for accessing basic I/O hardware 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 I/Ohardware registers from the C source level. However, these extensions have not been consistent across dialects. As a growing number of C++ compiler vendors are now entering the same market, the same I/O driver portability problems become apparent for C++.

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

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

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

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

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

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

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

The name is deliberately different, as it is the intention that `<hardware>` provides similar functionality to `<iohw.h>`, but through a different interface and implementation, just as `<iostream>` provides parallel functionality with `<stdio.h>` through different interfaces and implementation. There is no header `<ciohw>` specified, as that name would imply (by analogy with other standard library headers) that the C++ interfaces were identical to those in `<iohw.h>` but placed inside a namespace. Since macros do not respect namespace scope, the implication would be false and misleading.

This report provides:

- A general introduction and overview to the *iohw* interface (§5.1)
- A presentation of the common-subset interface (§5.2)
- A description of the C++ interface (§5.2.5)
- Usage guidelines for the C++ interface (§Appendix A:)
- General implementation guidelines for both interfaces (§A.1)
- Implementation guidelines for the C++ interface (§B.1.7)
- Implementation guidelines and example code for implementing the common-subset interface on top of the C++ interface (§Appendix C:)

5.1 Introduction to I/O Hardware Addressing

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

5.1.1 Basic Standardization Objectives

A standardization method for basic *iohw* addressing must be able to fulfill three requirements at the same time:

- A standardized 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 standardized 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 standardization 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, endianness, etc.)]*.

5.1.2 Overview and Principles

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

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

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

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

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

5.1.3 The Abstract Model

The standardization of basic *iohw* addressing is based on an abstract model with three layers:

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

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

The bottom layer is the compiler vendor's implementation of *iohw.* It provides prototypes for the functions defined in this section and specifies the various access methods supported by the given processor and platform architecture ("access methods" refers to the various ways of connecting and addressing hardware registers or hardware devices in the given processor architecture).

B.1.2 contains some general considerations that should be addressed when a compiler vendor implements the *iohw* functionality.

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

B.1.3 proposes a generic C++ syntax for hardware register *access-specification*s. Using a general syntax in this layer may extend portability to include user's hardware register specifications, so it can be used with different compiler implementations for the same platform.

5.1.3.1 The Module Set

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

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

These might be used as follows (in the common subset of C and C++):

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

unsigned char mybuf[10];
//...
iowr(MYPORT1, 0x8);                 // write single register
for (int i = 0; i < 10; i++)
    mybuf[i] = iordbuf(MYPORT2, i);  // read register array
```

In C++:

```
#include <hardware>
#include "iohw_ta.h" // My HW register definitions for target.
                     // Contains:
                     //    Definitions of MyPort1T, MyPort2T, MyPort3T
                     //    the value_type for MyPort3T is UCharBuf
                     //    the value_type for MyPort1T and MyPort2T is
                     //        unsigned char
using namespace std::hardware;

struct UCharBuf {
    unsigned char buffer[10];
};

unsigned char mybuf[10];
//...
MyPort1T myPort1;           // define HW register object
myPort1 = 0x08;             // write single register

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

MyPort3T myPort3;
UCharBuf mybufBlock;
mybufBlock = myPort3;       // reads the whole register array at once
```

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

5.1.4 Hardware Register Characteristics

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

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

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

5.1.5 The Most Basic Hardware Access Operations

The most common operations on hardware registers are `READ` and `WRITE`.

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

For the convenience of the programmer, and in order to promote good compiler optimization for bit operations, the basic logical operations `OR`, `AND`, and `XOR` are defined by *iohw* in addition to `READ` and `WRITE`.

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

5.1.6 The *access-specification*

The *access-specification*s defined in `"iohw_ta.h"` are used only as parameters in the functions for defining hardware register access.

Editor's Note: Is this the correct header name? It's better to be specific than to use the amorphous iohw everywhere.

The *access-specification* parameter represents or references a complete description of how the hardware register should be addressed in the given hardware platform. It is an abstract data type with a well-defined behavior[25].

The specification method and the implementation of *access-specification*s are processor and platform specific.

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

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

Other access characteristics typically specified via the *access-specification*:

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

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

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

[25] 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.

5.1.7 The *access-base-specification*

Often hardware registers are only portable between platforms as a single physical entity[26]. In such cases it is often convenient to make all the hardware register *access-specification* definitions relative to a single *access-base-specification* common to all registers in the physical entity.

When defining one or more registers as based in this way, the *access-specification* for the individual registers must at least identify the *access-base-specification* plus a logical offset relative to it. The properties of the logical offset are given in the context of the *access-base-specification.*

The use of based register definitions should be encapsulated in the two lower layers of the abstract model for hardware register access, and should therefore not be visible to the user driver source code.

However, if the access base initialization is completed at run-time, it must be possible to define in the user driver code when such initialization should or may take place. The *iohw* interface defines three functions for initialization, assignment, and eventual release of access bases. The *access-base-specification*s defined in the header `<iohw.h>` are used only as parameters in these functions.

5.1.7.1 Combined *access-specification* and *access-base-specification* Characteristics

When based register definitions are used, the hardware register access characteristics are given by the combined characteristics of *access-specification* and *access-base-specification.* The total access characteristics are divided in such a way that characteristics given by the hardware register are defined by *access-specification* and characteristics related to the processor and platform are defined by the *access-base-specification.*

With based register definitions, an *access-specification* definition will generally specify at least the following hardware register and hardware device characteristics:

- Logical register size (mapping to a data type)
- Logical offset relative to *access-base-specification*
- Access limitations (read-only, write-only)
- Hardware register endianness (if register width is larger than the device bus width)
- Interleave factor for hardware register buffers (if device bus width is smaller than the bus width defined by *access-base-specification*)

[26] For instance hardware registers in a chip, an FPGA cell or a plug-in board

The *access-base-specification* will, in general, define the following platform related characteristics:

- Bus address for *access-base-specification*
- Processor bus (if more than one)
- Access method (if more than one)
- Platform specific access driver functions (if any)

5.1.7.2 Virtual Addressing

A property of access bases is that they create their own virtual addressing range, and that all hardware register access must take place in a context given by the access base.

This concept gives a high degree of freedom and flexibility when implementing the two lower layers of the abstract model for hardware register access.

The access base can be a simple pointer, in which case the access base context is inherited from the underlying platform, or the access base can be implemented by use of access functions, in which case any virtual access base context can be created.

An implementation can elaborate this further, for instance, by enabling use of nested access functions. One perspective of such a feature is that the *iohw* interface itself can be used by the device driver programmer to create access functions, which are then used as the access base for *access-specification*s in other parts of the user source code.

5.2 The C Interface `<iohw.h>`

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

The *iohw* interface is described here in terms of function-like macros. An implementation is allowed to implement the interface by use of inline, template, or intrinsic functions and still be conforming, as long as the interface seen from the user source remains the same.

5.2.1 Function-Like Macros for Single Register Access

Synopsis

```
#include <iohw.h>

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

Description

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

The data type (the hardware register size) for `value` parameters and the value returned by `iord` is defined by the `access_spec` definition for the given register. The function-like macros `iowr`, `ioor`, `ioand` and `ioxor` do not return a value.

It is a requirement that a given hardware register is addressed exactly once during a `READ` or `WRITE` operation and exactly twice[27] during the read-modify-write operations `OR`, `AND` or `XOR`.

5.2.2 Function-Like Macros for Register Buffer Access

Synopsis

```
#include <iohw.h>

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

Description

These names map an *iohw* register buffer operation to an underlying (platform specific) implementation which provides access to the hardware 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 hardware register size) for `value` parameters and the value returned by `iordbuf` is defined by the `access_spec` definition for the given hardware register. `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 hardware register 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 hardware register buffer, not the processor address offset. Conversion from a logical index to a physical address requires that interleave calculations are performed by the underlying implementation. This is discussed further in B.1.2.2.

It is a requirement that a given hardware register is addressed exactly once during a `READ` or `WRITE` operation and exactly twice during the read-modify-write operations `OR`, `AND` or `XOR`.

[27] As seen from the device register, this requirement is independent of whether the read-modify-write operation is made by a single read-modify-write processor instruction or by separate read and write processor instructions.

5.2.3 Function-Like Macros for *access-base-specification* Initialization

Synopsis

```
#include <iohw.h>

io_abs_init(access_base_spec)
io_abs_release(access_base_spec)
```

Description

`io_abs_init` maps to an underlying (platform specific) implementation, which provides any *access-base-specification* initialization before performing any other operation on the hardware register (or set of hardware registers) identified by `access_base_spec`. A call to `io_abs_init` should be placed in the driver source code so that it is invoked exactly once before any other operations on the related registers are performed. `io_abs_init` does not return a value.

`io_abs_release` maps to an underlying (platform specific) implementation which releases any resources obtained by a previous call to `io_abs_init` for the same *access-base-specification*. A call to `io_abs_release` should be placed in the driver source code so it is invoked exactly once after all operations on the related registers have been completed. `io_abs_release` does not return a value.

Example

In an implementation for a hosted environment, the call to `io_abs_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 hardware registers based on this *access-base-specification*. The call to `io_abs_release` identifies the point in an execution sequence where the handle can be returned to the operating system.

5.2.4 Function-Like Macros for *access-base-specification* Re-Mapping

Synopsis

```
#include <iohw.h>

io_abs_remap(access_base_spec dest, access_base_spec src)
```

Description

`io_abs_remap` maps to an underlying (platform specific) implementation, which initializes the access information of the destination `access_base_spec` with access information taken from the source `access_base_spec`. The two parameters must have compatible *access-base-specification* types. The parameter `dest` must be an *lvalue*. The parameter `src` must be an *rvalue*. `io_abs_remap` does not return a value.

`io_abs_remap` can only be used with systems and implementations where the address can be initialized at run-time. If the `src` and `dest` *access-base-specification*s are incompatible, or the src *access-base-specification* cannot be initialized at run-time, a compile-time diagnostic is required.

Example

This example illustrates some simple cases of the underlying semantics for io_abs_remap:

```
// Some access bases
#define AddrA    ((uint8_t *)0x23456)

uint8_t*  get_os_base(void);

uint8_t*  base_a;
uint8_t*  base_b;

// Some implementation specific or user specific access base function
void set_my_base(uint8_t* base);

// Examples of some underlying functionality of io_abs_remap(...)

// The following statements could each be the resulting code after
// expansion of io_abs_remap(...)
base_a = AddrA;             // Initialize with a constant access base
base_b = get_os_base();     // Initialize via an access base function
base_a = base_b;            // Initialize from a variable base

set_my_base(AddrA);         // Initialize with a constant access base
set_my_base(get_os_base()); // Initialize via an access base function
set_my_base(base_a);        // Initialize from a variable access base

// Illegal access base definitions result in errors at compile-time.
AddrA         = base_a;     // Error, left operand must be an lvalue
get_os_base() = base_b;     // Error, left operand must be an lvalue
```

Example

A typical use for io_abs_remap and access_base_spec is when a set of driver functions for a given hardware device type are used with multiple hardware instances of the same device.

Editor's Note: As uint8_t isn't currently part of Standard C++, should we add a note referring to C's stdint.h or a compatible definition? We already have something like this in 5.3.2 for the <hardware> interfaces.

```
#include <iohw.h>
#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;
uint8_t  d2;
```

```
// Read from both devices
io_abs_remap(MYDEV_BASE, DEV1); // Select device 1
d1 = my_device_driver();
io_abs_remap(MYDEV_BASE, DEV2); // Select device 2
d2 = my_device_driver();
```

Use of `io_abs_remap` and *access-base-specification*s often provide a faster alternative than passing an `access_base_spec` as a function parameter.

Another advantage of using `io_abs_remap` is that the driver function itself (for a device) can be written without any prior knowledge about whether the driver will be used with only a single device (address defined at compile-time) or with multiple devices (addresses defined at run-time). This can be selected later at a higher level. In both cases the same source code can generate machine code which has maximum performance.

5.2.5 Information Required by the Interface User

In order to enable a driver library user to define the *access-specification* and *access-base-specification*s for a particular platform, a portable driver library based on the *iohw* interface should (in addition to the library source code) provide at least the following information:

- All symbolic names for the device registers used by the library
- Device and register type information for all symbolic names:
  - ➢ Logical bit width of the device register
  - ➢ The register type – single register or a register buffer
  - ➢ Bit width of the device data bus
  - ➢ Relative address offset of registers in the device (if the device contains more than one register)
  - ➢ Endianness of the device (if the register has a width larger than the device bus)
- If run-time initialization of dynamic addresses is required, i.e. `io_abs_remap` is used by the library

5.3 The C++ Interface `<hardware>`

The programming model behind these definitions is described in 5.1.3. The header `<hardware>` defines an interface for two layers of that model, the top layer for the portable source code and the middle layer for the device register definitions. This is notably different to the C interface described in 5.2.

The header `<hardware>` declares several types, which together provide a data-type-independent interface for basic *iohw* addressing.

Header `<hardware>` synopsis:

```
namespace std {
    namespace hardware {
        #include "stdint.h"    // see §5.3.2

        struct hw_base { ... };

        // required access types
        template <typename                        ValueType,
                  typename hw_base::access_mode   mode,
                  typename hw_base::address_type  address,
                  typename hw_base::device_bus    devWidth,
                  typename hw_base::byte_order    endian,
                  typename hw_base::processor_bus nativeWidth>
        class mm_direct_address;

        // [others still missing]
        template <typename ac_type>
        class register_access;
    } // namespace hardware
} // namespace std
```

Editor's Note: Does the "others still missing" comment still apply?

5.3.1 The Class Template `register_access`

Editor's Note: We could use some text here explaining the class, and where it fits in the abstraction model. And why we are providing redundant access methods. :-)

Synopsis

```
template <typename ac_type>
class register_access
{
private:
    struct ref_
    {
        // implementation-defined constructor(s) go here
        operator value_type() const;
        void operator =  (value_type val);
        void operator |= (value_type val);
        void operator &= (value_type val);
        void operator ^= (value_type val);

        // Function-style interface
        value_type read() const;
        void write(value_type val);
        void or_with(value_type val);
        void and_with(value_type val);
        void xor_with(value_type val);
    };
    public:
        typedef typename ac_type::value_type value_type;

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

        ref_ operator [] (size_t index);
        ref_ operator [] (ptrdiff_t index);

        // Function-style interface
        value_type read() const;
        void write(value_type val);
        void or_with(value_type val);
        void and_with(value_type val);
        void xor_with(value_type val);

        ref_ get_buffer_element(size_t index);
};
```

Description

`struct ref_;`

- Provides the same overloaded operators as `register_access` to allow the same operations[28].

`class register_access<...>`

- Provides direct access to hardware registers. This defines the interface for the top layer as described in 5.1.3.

`typename ac_type`

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

`ac_type::value_type value_type`

- Names the `value_type` of the *access-specification*.

`operator value_type() const`
`value_type read() const`

- Provides read access to the hardware register.

`void operator = (value_type val)`
`void write(value_type val)`

- Writes the `value_type` argument `val` to the hardware register.

`void operator |= (value_type val)`
`void or_with(value_type val)`

- Bitwise `OR`s the hardware register with the `value_type` argument `val`.

`void operator &= (value_type val)`
`void and_with(value_type val)`

- Bitwise `AND`s the hardware register with the `value_type` argument `val`.

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

```
void operator ^= (value_type val)
void xor_with(value_type val)
```

- Bitwise `XOR`s the hardware register with the `value_type` argument `val`.

Note: The return types for all assignment operators is `void` to prevent assignment chaining that could inadvertently cause considerable harm with device registers.

```
ref_ operator [] (size_t index)
ref_ operator [] (ptrdiff_t index)
ref_ get_buffer_element(size_t index)
```

- Returns the equivalent of a reference to the location specified by `index` inside of the device register. The return value can be used like a `register_access` object, i.e. it can be written, read, and the bitwise `OR`, `AND` and `XOR` can be applied to it. The subscript operator is explicitly provided for `signed` and `unsigned` indices[29].

5.3.2 Header `"stdint.h"`

The header `<stdint.h>` is specified by C99 (IS 9899-1999), and is not part of the C++ Standard. Instead, the implementation defined header `"stdint.h"` included by `<hardware>` introduces the fixed size integer types described by `<stdint.h>` into namespace `std::hardware`.

No names are introduced into global namespace.

5.3.3 The `struct hw_base`

Editor's Note: We could use some text here explaining the class, and where it fits in the abstraction model.

Synopsis

```
namespace std {
    namespace hardware {
        struct hw_base
        {
           enum access_mode   { random, read_write, write, read };
           enum device_bus    { device8,  device16,
                                device32, device64 };
           enum byte_order    { msb_low, msb_high };
           enum processor_bus { bus8, bus16, bus32, bus64 };
           // only identifiers should be present that are supported
           // by the underlying implementation -- diagnostic required

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

Description

Editor's Note: Should we recommend a typical set of enum values to select from?

[29] If `value_type` is any kind of pointer, overload resolution can result in an unexpected call to the conversion operator, followed by the selection of the built-in subscript operator rather than the member subscript operator provided. This is not an issue for functions, so the equivalent function `get_buffer_element` is not overloaded.

`struct hw_base`

- Provides the names for the supported hardware characteristics. Only those names that are supported by the hardware shall be present.

`enum access_mode`

- Defines the possible modes for accessing a device register.

`enum device_bus`

- Defines the names for the width of the hardware register device bus as seen from the processor.

`enum byte_order`

- Defines the names for the endianness of the device register.

`enum processor_bus`

- Defines the names for the width of the processor bus.

`address_type`

- Is an integral type specified by the application to hold a hardware address.

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

5.3.4 Common Specifications for *access-specification* Types

`typename ValueType`

- All *access-specification* template types have at least a `ValueType` parameter. The argument for this parameter shall be an `Assignable` and `CopyConstructible` type.
- The arguments for `ValueType` are not restricted to integral values. *[e.g. it makes perfect sense for `ValueType` to be `double` or `long double` when accessing an external floating-point co-processor. It might even be useful sometimes to have a user-defined `struct` as `ValueType`.]*
- The memory location of an object of the `ValueType` argument shall be freely readable and writeable (as required by the access operations) by the implementation of this interface. *[Note: this requirement essentially disallows other hardware registers or types. Also, their value might be changed through the implementation by direct memory access instead of any (possibly overloaded) assignment operators]*
- Most of the *access-specification* types have a common set of *template-parameter*s, which are specified as follows:

`hw_base::access_mode mode`

- Defines the access mode of the device register.

`hw_base::device_bus devWidth`

- Defines the width of the device to be accessed as seen by the processor.
- However, `sizeof(ValueType)` must be a natural multiple of `devWidth`.

`hw_base::byte_order endian`

- Defines whether the device attached to the bus is to be accessed as little-endian or big-endian.

`hw_base::processor_bus nativeWidth`

- Defines the width of the processor bus.
- All *access-specification* types may have additional *template-parameter*s specified by the implementation. The implementation may also define default arguments for some of the *template-parameter*s. *[e.g. on segmented architectures there might be an additional segment parameter]*

5.3.5 Access Methods

5.3.5.1 The Class Template `struct mm_direct_address`

Editor's Note: We could use some text here explaining the class, and where it fits in the abstraction model.

Synopsis

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

    template <hw_base::address_type other_address> struct rebind
    {
        typedef mm_direct_address<value_type,
                                  mode,
                                  other_address,
                                  devWidth,
                                  endian,
                                  nativeWidth> other;
    };
};
```

Description

`mm_direct_address`

- Defines the *access-specification* type for device registers for which the address is known at compile-time and the registers are directly mapped to the memory bus.

`hw_base::address_type address`

- The argument `address` shall be the actual address of the device register to be accessed by this *access-specification* type.

`typedef ValueType value_type`

- Holds the `ValueType` *template-parameter.*

`struct rebind<...>::other`

- This is a type with the same hardware characteristics but a different device register address.

Appendix A: Guidelines for Using the *iohw* Interfaces

A.1 Usage Introduction

The design of the C++ *iohw* interface follows two lines of separation between:

- The definition of *access-specification*s and the device driver code
- What is known at compile-time and what is only known at run-time

Unfortunately, these two lines of separation are neither orthogonal nor identical; for example, the base address for base/offset addressing is only known at run-time, but belongs to the *access-specification*.

As C++ is a typed language, the differences for the interface are in the type system, and therefore the main separation line for the interface definition itself is between what is statically known at compile-time (which becomes template arguments to types) and what is only known at run-time (which becomes function arguments or operator operands to the interface of `register_access`).

A.2 Using access-specifications

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

- Templates with at least `ValueType` as *template-parameter*
- Exposing this `ValueType` argument as `typedef value_type`
- A class template `rebind` to provide a simple way to define *access-specification*s that differ from an existing one only in a specific aspect (typically the hardware address)

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

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

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

```
typedef mm_direct_address<uint8_t,
                          hw_base::read,
                          0x12345678,
                          hw_base::device8,
                          hw_base::msb_high,
                          hw_base::bus32
                          ...>  InPort1;
```

- A similar *access-specification* template `io_direct_address` exists for addresses on the I/O bus.
- `general_address` is not actually defined in the C++ interface, but provided in the sample implementation. It provides a very general *access-specification* for all kinds of addressing methods, including different processor busses, multi-part memory addresses, dynamic base addresses and user-supplied functions.

As already said, these *template-parameter*s are platform dependent and can vary widely for more exotic platforms. Even the `address` parameter might vary; for example, on a segmented addressing architecture there might be two parameters for a segment and an offset address instead of a single address parameter.

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

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

A.2.1 Using *access-specification*s with Dynamic Information

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

For example. an implementation might provide a `general_address` for which the dynamic data type must provide a public member function `value()` with the return type `unsigned long`. Then the user can provide a corresponding class:

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

    unsigned long value() const {  // some complicated calculation
                                   // based on the current mode of
                                   // the processor
                                }
    unsigned long val;
};
typedef general_address<uint32_t,
                        hw_base::random,
                        DynAddressPortDA>  PortDA_t;
```

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

In a different case, the constructor might require an argument. Therefore some initialization code must provide that argument. But the mechanics of the initialization are always left to the user to choose the best fitting method.

A.3 Hardware Access

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

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

unint8_t tmp = ip;     // read from InPort, uses
                       // register_access::operator value_type();
op     = 0x12;         // write to OutPort, uses
                       // register_access::operator=(value_type);
ctl_p |= 0x34;         // set bits 5, 4 and 2 in ControlPort
```

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

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

uint8_t tmp = ip();    // read from InPort, uses
                       // register_access::operator value_type();
op()    =  0x12;       // write to OutPort, uses
                       // register_access::operator=(value_type);
ctl_p() |= 0x34;       // set bits 5, 4 and 2 in ControlPort
```

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

A.3.1 Indexed Access

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

```
// assume register block PortBuffer with random access
register_access<PortBuffer> portBuf;
uint8_t buf[sz];

portBuf[0] &= 0x03;
portBuf[1] =  sz – 2;

for (int i=2; i != sz; ++i)
    buf[i] = portBuf[i];
```

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

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

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

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

A.3.2 Initialization of register_access

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

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

or

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

Appendix B: Implementing the *iohw* Interfaces

B.1 General Implementation Considerations

B.1.1 Purpose

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

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

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

B.1.1.1 Recommended Steps

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

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

B.1.1.2 Compiler Considerations

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

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

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

However, with a conforming C++ compiler, an efficient, all-round implementation of both the C and C++ interface headers can usually be achieved using the C++ template functionality (see also §5.2 and §B.1.3). A template-based solution allows the number of compiler specific intrinsic hardware access types or intrinsic hardware access functions to be minimized or even removed completely, depending on the processor architecture.

For compilers not supporting templates (such as C compilers) other implementation methods must be used. In any case, at least the most basic *iohw* functionality can be implemented efficiently using a mixture of macros, inline functions and intrinsic types or functions.

For many architectures, fully featured, zero-overhead implementations of *iohw* can be achieved using templates. An approach is discussed in §5.2.5. Nevertheless, fully featured *iohw* implementations for a number of architectures will usually require direct compiler support.

B.1.2 Overview of Hardware Device Connection Options

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

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

B.1.2.1 Multi-addressing and Device Register Endianness

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

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

[Note: while this section illustrates architectures that use 8-bit bytes and word widths that are factorable by 8, it is not intended to imply that these are the only possible architectures.]

[The device register endianness is not coupled to the endianness of the underlying processor hardware architecture.]

Table B-1: Logical I/O register / I/O device addressing overview[30]

Logical register width	Device bus width							
	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)

B.1.2.2 Address Interleave

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

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

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

This can also be written as

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

where *interleave* = 4.

Table B-2: Interleave overview: (bus to bus interleave relationship)

Device bus width	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)

[30] This table describes some common bus and register widths for I/O devices. A given hardware platform may use other register and bus widths.

B.1.2.3 Device Connection Overview

A combination of the two tables above shows all relevant cases for how device registers can be connected to a given processor hardware bus:

Table B-3: Interleave between adjacent I/O registers in buffer

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

(For byte-aligned address ranges)

B.1.2.4 Generic Buffer `index`

The interleave distance between two logically adjacent registers in a device register array can be calculated from[31]:

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

[31] For systems with byte-aligned addressing.

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

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

Assumptions:

- Bytes are 8-bits wide
- Address range is byte-aligned
- Data bus widths are a whole number of bytes
- The 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 width of the `processor_data_bus`

B.1.3 Implementing *access-specification*s for Different Device Addressing Methods

A processor may have more than one addressing range[32]. For each processor addressing range 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 initialized at run-time:***

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

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

- ***Indexed bus addressing:***

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

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

[32] Processors with a single addressing range use only memory mapped I/O.

operations should be applied to indexed bus addressing in order to prevent an interrupt occurring between setting up the address and the data operation

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

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

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

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

In general, an implementation 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 numbers of parameters required and the parameter ranges used in an *access-specification*, the C++ interface requires the definition of different *access-specification* templates for each of the different addressing methods.

For the C-style interface, it is often convenient for the implementer of the *iohw* middle layer to provide definitions for each of the different addressing methods using *access-specification* templates also, therefore implementing the C interface (§Appendix C:) on top of the C++ interface. This allows the implementer to share a common implementation between the C and C++ interfaces, while also providing greater type safety than the macro-based pure C implementation can provide.

B.1.3.1 Bus Connection Parameters

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

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

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

```
struct hw_base
{
    // ...
    enum device_bus    { device8 = 1,  device16 = 2,
                         device32 = 4, device64 = 8 };
    enum byte_order    { msb_low, msb_high };
    enum processor_bus { bus8 = 1, bus16 = 2, bus32 = 4, bus64 = 8 };
    // Only identifiers should be present that are supported by the
    // underlying implementation!
};
```

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

```
struct hw_base
{
    // ...
    enum device_bus    { device8 = 1, device16 = 2 };
    enum byte_order    { msb_low, msb_high };
    enum processor_bus { bus8 = 1, bus16 = 2 };
};
```

B.1.3.2 Detection of Read / Write Violations in Device Registers

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

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

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

Table B-4: Allowed operations on different device register types:

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

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

For example:

```
struct hw_base
{
    // ...
    enum access_mode { random, read_write, write, read };
};

// Access specification template for directly addressedregisters on
// the I/O bus:
template <typename _ValueType,
          hw_base::access_mode   mode,
          hw_base::address_type  address,
          hw_base::device_bus    devWidth,
          hw_base::byte_order    endian,
          hw_base::processor_bus nativeWidth>
class io_direct_address { ... };

// access-specifications from the middle layer:
typedef io_direct_address<uint32_t,
                          hw_base::write,
                          0x358,
                          hw_base::device16,
                          hw_base::msb_high,
                          hw_base::bus32> PortO_t;
typedef io_direct_address<uint32_t,
                          hw_base::read,
                          0x372,
                          hw_base::device16,
                          hw_base::msb_high,
                          hw_base::bus32> PortI_t;
typedef io_direct_address<uint32_t,
                          hw_base::random,
                          0x38c,
                          hw_base::device16,
                          hw_base::msb_high,
                          hw_base::bus32> PortIO_t;

// Object definitions for C++:
PortO_t   outPort;
PortI_t   inPort;
PortIO_t  inOutPort;

// Top layer C++ code:
uint32_t myval;

myval     =  inPort;       // ok
myval     += inOutPort;    // ok
outPort   =  myval;        // ok
inOutPort =  myval;        // ok

myval     = outPort;       // error, compile-time diagnostic
inPort    = myval;         // error, compile-time diagnostic

// NULL-pointer definitions for C interface:
PortO_t*  OutPort   = 0;
PortI_t*  InPort    = 0;
PortIO_t* InOutPort = 0;
```

```
// Top layer C code
uint32_t myval;

myval =  iord(InPort);     // ok
myval += iord(InOutPort);  // ok

iowr(OutPort,myval);       // ok
iowr(InOutPort,0x45);      // ok

myval =  iord(OutPort);    // error, compile-time diagnostic
iowr(InPort,0x55);         // error, compile-time diagnostic
```

B.1.3.3 Implementation for Different Processor Busses

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

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

```
// Direct address for memory mapped registers
template <typename _ValueType,
          hw_base::access_mode   mode,
          hw_base::address_type  address,
          hw_base::device_bus    devWidth,
          hw_base::byte_order    endian,
          hw_base::processor_bus nativeWidth>
class mm_direct_address { ... };

// Direct address for registers on I/O bus
template <typename _ValueType,
          hw_base::access_mode   mode,
          hw_base::address_type  address,
          hw_base::device_bus    devWidth,
          hw_base::byte_order    endian,
          hw_base::processor_bus nativeWidth>
class io_direct_address { ... };
```

B.1.3.4 Implementation for Different Access Methods

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

For example:

```
struct hw_base
{
    // ...
    // Different address types
    typedef uint32_t address_t;      // Memory mapped address range
    typedef uint8_t  sub_address_t;  // Sub address on indexed bus
    typedef uint16_t io_address_t;   // User device driver address
    typedef uint8_t  bit_pos_t;      // Bit position in register
};
```

```
// Direct address for memory mapped registers
template <typename _ValueType,
          hw_base::access_mode   mode,
          hw_base::address_type  address,
          hw_base::device_bus    devWidth,
          hw_base::byte_order    endian,
          hw_base::processor_bus nativeWidth>
class mm_direct_address { ... };

// A lot of methods can be done with a general template:
template <typename _ValueType,
          hw_base::access_mode   mode,
          typename _AddressType,    // UDT with implementation defined
                                    // semantics
          typename               _BusToggle  = hw_base::data_bus,
          hw_base::device_bus    devWidth    = hw_base::device32,
          hw_base::byte_order    endian      = hw_base::msb_high,
          hw_base::processor_bus nativeWidth = hw_base::bus32>
class general_address { ... };

// For the different cases respective AddressTypes are defined:
// Base addressing:
struct base_address_holder
{
    base_address_holder(hw_base::address_t* base,
                        hw_base::address_t  offset);
};

// Indexed addressing:
struct indexed_address_holder
{
    indexed_address_holder(hw_base::address_t address,
                           hw_base::sub_t     index);
};

// Single bit addressing:
struct bit_address_holder
{
    bit_address_holder(hw_base::address_t address,
                       hw_base::bit_pos_t index);
};

// for a user-supplied function an own class can be specified
```

B.1.3.5 Optimization Possibilities for Typical Implementations

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 hardware register access functions that have to be implemented can be reduced with no efficiency penalty using simple delegation, using inline functions or macros such as:

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

or function templates such as:

```
template <class access_spec>
    inline void iowr(typename access_spec::value_type val)
    {
        iowrbuf<access_spec>(0, val);
    }
```

Multi-Addressing and Endianness

Typical candidates for platform dependent optimizations are *iohw* functions for the multi-addressing cases (*logical device register* width > *device bus* width) where the width of the *device data bus* matches the width of the *processor data bus*; for example, the combinations of:

- (*device16h* or *device16l*) and *bw16*
- (*device32h* or *device32l*) and *bw32*

In these cases, multi-byte access can often use data types that are directly supported by the processor for either the LSB or MSB endianness functions. The other endianness functions can often be implemented efficiently using one load or store operation, plus one or more register swap operations.

B.1.4 Atomic Operation

It is a requirement of the *iohw* implementation that in each *iohw* function a given (partial[33]) device 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 *iohw* function in an *iohw* implementation, be implemented such that the device access operation becomes *atomic* whenever possible. However, atomic operation is not guaranteed to be portable across platforms for the logical-write operations (i.e. the `OR`, `AND`, and `XOR` operations) or for multi-addressing cases. The reason for this is simply that many processor architectures do not have the instruction set features required for assuring atomic operation.

B.1.5 Read-Modify-Write Operations and Multi-Addressing

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

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

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

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

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

B.1.6 Initialization

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

- hardware (device) initialization
- *access-specification* initialization
- device selector initialization[34]

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

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

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

  For many *access-specification*s, there is no run-time initialization necessary. However, for some access methods, some run-time initialization is required.

  using the *iohw* C-style interface, the function:

  ```
  io_abs_init(access_base_spec)
  ```

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

  The *iohw* C++ interface provides the constructor

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

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

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

- **device selector initialization**: This is used when, for instance, the same device driver code needs to service multiple devices of the same type.

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

  The *iohw* C-style interface provides another solution – the use *access-specification* copying, and *access-specification*s with dynamic access information:

  ```
  io_abs_remap(access_base_spec dest, access_base_spec src)
  ```

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

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

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

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

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

B.1.7 Intrinsic Features for I/O Hardware Access

The implementation of *iohw* access require for many platforms use of special machine instructions not otherwise used with the normal C/C++ memory model. It is recommended that the compiler vendor provide the necessary intrinsics for operating on any special addressing range supported by the processor.

In C++ special machine instructions can be inserted inline using the *asm declaration*. However when using `asm` in connection with hardware register access, intrinsic functionality is often still required in order to enable easy load of symbolic named variables to processor registers and to handle return values from `asm` operations.

An *iohw* implementation should completely encapsulate any intrinsic functionality.

B.2 Implementation Guidelines for the C++ Interface

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

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

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

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

B.2.1 Annotated Sample Implementation

As the hardware header belongs in some way to the implementation of a (non-standard) part of the C++ library and a user of that may place any macros before this header, the header itself should only use symbols reserved to the implementation, i.e. names beginning with an underscore. Right now, this is not completely the case but will be cleaned out in the next revision.

B.2.1.1 Common Definitions — struct hw_base

In this sample implementation the *access-specification* holds all necessary address information and provides them to the register_access implementation. To produce as few overheads as possible in cases where the address information is known at run-time, no object data is produced. To achieve this, this implementation generally uses typedefs where the real address information is kept in an enum value. For this, a small helper struct _Int2Type is used (the typedef _ul is purely to save some typing):

```
typedef unsigned long _ul;

// helper class for saving integral values as types
struct _Int2Type
{
    enum constants    // Technique uses an enum to group constants
    {
        value_ = val
    };
    _ul value() const { return value_; }
};
```

As the implementation has to deal with value types of unknown size, this implementation uses internally unsigned integer of an appropriate size. For that purpose, another helper template is defined that provides that type:

```
// and to create an integral type for a given sizeof
template <_ul size> struct _uint_type;
template <> struct _uint_type<1> { typedef uint8_t  ui_type; };
template <> struct _uint_type<2> { typedef uint16_t ui_type; };
template <> struct _uint_type<4> { typedef uint32_t ui_type; };

#ifdef UINT64_MAX
template <> struct _uint_type<8> { typedef uint64_t ui_type; };
#endif
```

And _EmptyType is a simple placeholder that can be used anywhere where a type *template-parameter* is needed that is not useful for this particular instantiation:

```
// and an empty helper class for default DynamicData
struct _EmptyType {};
```

hw_base defines all the constants that are necessary in the *access-specification*s. Of course, this is highly dependant on the specific hardware, and only those that are used in this implementation are shown here. In general, there are two different ways to define constants: the standard *IOStreams* library defines constants as static. This allows for easier implementation, but has some space and possibly run-time overheads. For performance reasons, the enum approach is chosen here, where all constant values are defined as enumerates. This has the additional advantage that they can be used as type-safe template value parameters in the *access-specification* templates.

According to the interface specification, an implementation can define additional members in hw_base. This implementation defines two tagging types data_bus and io_bus for use *access-specification*s:

```
// the definitions of access_types' parameter types
struct hw_base
{
    enum access_mode   { random, read_write, write, read };
    enum device_bus    { device8  = 1, device16 = 2,
                         device32 = 4, device64 = 8 };
    enum byte_order    { msb_low, msb_high };
    enum processor_bus { bus8 = 1, bus16 = 2, bus32 = 4, bus64 = 8 };

    // only identifiers should be present that are supported by
    // the underlying implementation! (Diagnostic required.)

    typedef _ul address_type;

    // specialization types for different implementations for
    // different bus types
    enum data_bus { ... };
    enum io_bus   { ... };
};
```

_native_endian is a helper to optimize behavior for the byte ordering of the underlying processor:

```
typedef _Int2Type<hw_base::msb_high> _native_endian;
```

B.2.1.2 Implementation for *access-specification*s

For this implementation a fairly simple addressing scheme is assumed, but on any implementation, all address information should be a small bounded set that fits into a respective class. Here, a helper template to hold the information necessary to calculate the address offset is defined. In this implementation all *access-specification*s contain the same address information, but they require different operations for different busses. Therefore, the _AddressInfo class contains a marker _BusTag that differentiates the different busses:

```
// helper template to hold the info necessary to calculate the
// address offset
template <_ul _valueSize,
          hw_base::device_bus    _deviceWidth,
          hw_base::processor_bus _procBusWidth,
          class                  _AddressHolderT,
          class                  _Toggle>
struct _AddressInfo
{
    enum constants    // Technique uses an enum to group constants
    {
        _registerSize = _valueSize,
        _devWidth     = _deviceWidth,
        _nativeWidth  = _procBusWidth
    };
    typedef _AddressHolderT _AddressHolder;
    typedef _Toggle         _BusTag;
};
```

In general, a lot of different *access-specification* types are possible. But for any given implementation only a small set makes sense, and only that small set should be provided. This implementation only provides two *access-specification*s for direct address (`mm_direct_address` for memory mapped registers as specified in the interface description, `io_direct_address` for registers on a separate I/O bus) and one quite general *access-specification* `general_address` to provide a user function to calculate the address. This `general_address` is used for simple dynamic addressing and segmented addressing by providing `fix_address_holder` and `segmented_address_holder`. Both *access-specification*s are templates with *template-parameter*s for the value type and all relevant hardware parameters required for the correct accessing of device registers on a simple platform.

As both direct address types are essentially the same and differ only in the associated bus, a common base class `_direct_address` is provided. The actual *access-specification*s are derived from this base class and just specify the respective `_BusToggle`. The direct *access-specification*s have all necessary information at compile-time, so it doesn't contain any run-time data but provides everything as types or enumerates (a typical traits class). Some additional types (`dynamic_data` and `_BaseAddressHolder`) are provided as empty types to provide a consistent interface for `mm_direct_address` and `general_address`:

```
// common direct address for address known at compile-time
template <typename _ValueType,
          hw_base::access_mode   mode,
          hw_base::address_type  address,
          hw_base::device_bus    devWidth,
          hw_base::byte_order    endian,
          hw_base::processor_bus nativeWidth,
          typename               _BusToggle>
class _direct_address
{
public:
    typedef _ValueType value_type;
    typedef _EmptyType dynamic_data;
    enum constants
    {
        access_mode = mode
    };

    template <hw_base::address_type other_address>
    struct rebind
    {
        typedef _direct_address<_ValueType,
                                mode,
                                other_address,
                                devWidth,
                                endian,
                                nativeWidth,
                                _BusToggle> other;
    };

    typedef _EmptyType _dynDataHolder;
    // we don't want to spend any space, so all arguments are saved
    // as types
    typedef _Int2Type<address> _BaseAddressHolder;
    typedef _Int2Type<endian>  device_endian;
    typedef _AddressInfo<sizeof(_ValueType),
                            devWidth,
                            nativeWidth,
                            _BaseAddressHolder,
                            _BusToggle> _AddressT;
};

// direct address for memory mapped registers
template <typename               _ValueType,
          hw_base::access_mode   mode,
          hw_base::address_type  address,
          hw_base::device_bus    devWidth,
          hw_base::byte_order    endian,
          hw_base::processor_bus nativeWidth>
class mm_direct_address
    : public _direct_address<_ValueType, mode, address, devWidth,
                             endian, nativeWidth, hw_base::data_bus>
{
};

// direct address for registers on I/O bus
template <typename _ValueType,
          hw_base::access_mode mode,
          hw_base::address_type address,
          hw_base::device_bus devWidth,
          hw_base::byte_order endian,
          hw_base::processor_bus nativeWidth>
class io_direct_address
    : public _direct_address<_ValueType, mode, address, devWidth,
                             endian, nativeWidth, hw_base::io_bus>
{
};
```

The `general_address` *access-specification* template uses an additional *template-parameter* `_AddressType` that has to provide a `const` member function `value()`. The return value of this function is used as the address. For the user's convenience two types `fix_address_holder` and `segmented_address_holder` are provided that can be used for simple cases. Also, the *template-parameter* `_BusToggle` can be directly provided by the user:

```
// dynamic address for memory mapped registers and address only
// known at run-time
template <typename              _ValueType,
          hw_base::access_mode   mode,
          typename              _AddressType,
          typename              _BusToggle  = hw_base::data_bus,
          hw_base::device_bus    devWidth    = hw_base::device32,
          hw_base::byte_order    endian      = hw_base::msb_high,
          hw_base::processor_bus nativeWidth = hw_base::bus32>
class general_address
{
public:
    typedef _ValueType   value_type;
    typedef _AddressType dynamic_data;
    enum constants
    {
        access_mode = mode
    };

    typedef _AddressType _BaseAddressHolder;

    typedef _Int2Type<endian> device_endian;
    typedef _AddressInfo<sizeof(_ValueType),
                         devWidth,
                         nativeWidth,
                         _BaseAddressHolder,
                         _BusToggle> _AddressT;
};
```

The `fix_address_holder` serves as `AddressType` for when the access is memory-mapped but the address is known only at run-time:

```
struct fix_address_holder
{
    explicit fix_address_holder(_ul addr) : value_(addr) {}
    _ul value() const { return value_; }
    _ul value_;
};
```

The segmented_address_holder serves as AddressType for general_address when the access is memory mapped but the address is known only at run-time and is composed from a segment and offset address. *[Note: this implementation is probably too simple and is only provided to illustrate an AddressType with two constructor parameters.]*:

```
struct segmented_address_holder
{
    explicit segmented_address_holder(unsigned short segment,
                                      unsigned short offset)
        : value_(segment << 16 + offset) {} // much simplified
    _ul value() const { return value_; }
    _ul value_;
};
```

B.2.1.3 Actual Access Implementation

The access to the device register values is divided into two parts; one group of helper classes (including _hwRead and _hwOp) does the actual register access, while another helper class _AccessHelper does the necessary adaptation between the device register value and the internal program (processor) value. This separation might not be possible or useful for all *access-specification*s (e.g. where the compiler provides combined intrinsics for both operations at once). In that case just the _AccessHelper needs to be specialized in an appropriate way.

The helper classes _hwOp and _hwRead effectively provide the functions that are eventually executed (not really called, as they are inline) when a device register is accessed. They typically use some assembler or compiler intrinsics different for all *access-specification* types, and this way, all implementation specific functionality can be provided in one place. They have no implementation for the general case, but must be specialized for all bus types that have different access operations (data_bus and io_bus in this example implementation) and the _hwOp additionally for all binary operators:

```
// helper classes for all provided operations to be specialized
// on _implTag
template <typename _RetType, typename _implTag> struct _hwRead;

// helper class for all provided binary operations
enum _binops { _write_op, _or_op, _and_op, _xor_op };
template <typename int_type, _binops, typename _implTag> struct _hwOp;
```

For the memory-mapped data_bus the access is like a normal memory access and the only thing is to do the usual int-to-pointer cast (including a volatile cast to prevent the optimizer from removing the access):

```
// implementation for hw_base::data_bus
template <typename _RetType>
struct _hwRead<_RetType, hw_base::data_bus>
{
    static _RetType r(_ul const & _addr)
    {
        return *const_cast<_RetType volatile *>
               (reinterpret_cast<_RetType *>(_addr));
    }
};

// a helper function to avoid having to write the same
// ugly cast for each op:
template <typename int_type, _binops>
struct _hwOp_data;

template <typename int_type>
struct _hwOp_data<int_type, _write_op>
{
    static void f(int_type volatile &lhs, int_type rhs)
    { lhs = rhs; }
};
template <typename int_type>
struct _hwOp_data<int_type, _or_op>
{
    static void f(int_type volatile &lhs, int_type rhs)
    { lhs |= rhs; }
};
template <typename int_type>
struct _hwOp_data<int_type, _and_op>
{
    static void f(int_type volatile &lhs, int_type rhs)
    { lhs &= rhs; }
};
template <typename int_type>
struct _hwOp_data<int_type, _xor_op>
{
    static void f(int_type volatile &lhs, int_type rhs)
    { lhs ^= rhs; }
};

// this does the casting necessary for hw_base::data_bus and
// delegates further for _op:
template <typename int_type, _binops _op>
struct _hwOp<int_type, _op, hw_base::data_bus>
{
    static void f(_ul _addr, int_type rhs)
    {
       _hwOp_data<int_type, _op>::f(
            *const_cast<int_type volatile *>
            (reinterpret_cast<int_type *>(_addr)), rhs);
    }
};
```

For the io_bus case, this implementation assumes some compiler intrinsics i_io_**xx**:

```
// implementation for hw_base::io_bus
template <typename _RetType>
struct _hwRead<_RetType, hw_base::io_bus>
{
    static _RetType r(_ul const & _addr)
    { return i_io_rd(_addr); }
};
template <typename int_type>
struct _hwOp<int_type, _write_op, hw_base::io_bus>
{
    static void f(_ul _addr, int_type rhs)
    { i_io_wr(_addr, rhs); }
};
template <typename int_type>
struct _hwOp<int_type, _or_op, hw_base::io_bus>
{
    static void f(_ul _addr, int_type rhs)
    { i_io_or(_addr, rhs); }
};
template <typename int_type>
struct _hwOp<int_type, _and_op, hw_base::io_bus>
{
    static void f(_ul _addr, int_type rhs)
    { i_io_and(_addr, rhs); }
};
template <typename int_type>
struct _hwOp<int_type, _xor_op, hw_base::io_bus>
{
    static void f(_ul _addr, int_type rhs)
    { i_io_xor(_addr, rhs); }
};
```

As the calculation of the actual address is used in quite a number of places, it is provided here as a helper function that takes the type with the actual address information as *template-parameter* (this will normally be some instantiation of _AddressInfo). The function used here is only valid where the device bus width is an exact multiple of the processor bus width:

```
// a helper function for the actual address calculation
template <class _AddressInfoT>
inline _ul _addrCalc(
              ptrdiff_t                                     idx,
              typename _AddressInfoT::_AddressHolder const& addr)
{
    return addr.value()
              + idx
                  * _AddressInfoT::_registerSize
                  * _AddressInfoT::_nativeWidth
                      / _AddressInfoT::_devWidth;
}
```

`_AccessHelper` implements the adaptation between the register value and the program value, including endian conversion and bus widths mapping. Again, there is no implementation for the general case; all supported cases must be provided by specializations:

```
// a helper class to provide all useful (partial) specializations
// for register_access
template <typename _ValueType,
          _ul      devEndian,
          class    _AddressInfoT>
struct _AccessHelper
{
    static _ValueType _read(
              _ul baseIdx,
              typename _AddressInfoT::_AddressHolder const &);
    template <_binops function>
    static void _op(_ValueType val,
                    _ul        baseIdx,
                    typename _AddressInfoT::_AddressHolder const&);
};

// no definition of the functions for the general case:
//    all valid cases must be provided as (partial) specializations
//
```

In the simplest case where no endian conversion is necessary and the device bus and processor bus have the same width, just forward to the `_hwRead` and `_hwOp` helpers:

```
// here a specialization where deviceWidth matches nativeWidth
// and ValueType
template <typename _ValueType,
          class    _AddressHolder,
          class    _implToggle>
struct _AccessHelper<_ValueType,
                     _native_endian::value_,
                     _AddressInfo<sizeof(_ValueType),
                           hw_base::device_bus(sizeof(_ValueType)),
                           hw_base::processor_bus(sizeof(_ValueType)),
                           _AddressHolder,
                           _implToggle> >
{
    typedef _AddressInfo<sizeof(_ValueType),
                           hw_base::device_bus(sizeof(_ValueType)),
                           hw_base::processor_bus(sizeof(_ValueType)),
                           _AddressHolder,
                           _implToggle>   AddressT;
    static _ValueType _read(_ul baseIdx, _AddressHolder const &addr)
    {   // the _implToggle argument selects the correct function
        return _hwRead<_ValueType, _implToggle>::r(
                                _addrCalc<_AddressT>(baseIdx, addr));
    }
    template <_binops function>
    static void _op(_ValueType            val,
                    _ul                   baseIdx,
                    _AddressHolder const& addr)
    {
        _hwOp<_ValueType, function, _implToggle>::f(
                           _addrCalc<_AddressT>(baseIdx, addr), val);
    }
};
```

Another quite simple case is where the endianness is the same and the `value_size` is an exact multiple of the device width. In other cases (e.g. when the device register is 12 bits and the value 16 bits) some padding is necessary. But here, a simple for loop (that is easily unrolled by the optimizer) does the job:

```
template <typename _ValueType,
          class    _AddressInfoT>
struct _AccessHelper<_ValueType,
                     _native_endian::value_,
                     _AddressInfoT>
{
    typedef typename _AddressInfoT::_AddressHolder _AddressHolder;
    enum constants
    {
        _wordCount = _AddressInfoT::_registerSize
                        / _AddressInfoT::_devWidth,
        _step      = _AddressInfoT::_nativeWidth
    };
    typedef typename _uint_type<_AddressInfoT::_devWidth>::ui_type
                          reg_t;
    struct buf_t
    {
       reg_t values[_wordCount];
    };

    static _ValueType _read(_ul baseIdx, _AddressHolder const &addr)
    {
        buf_t buffer;
        for (_ul idx=0; idx != _wordCount; ++idx)
        {   // uses new style casts
            buffer.values[idx] = _hwRead<reg_t,
                            typename _AddressInfoT::_BusTag>::r(
                               _addrCalc<_AddressInfoT>(baseIdx, addr)
                                  + idx
                                      * _step);
        }
        return *((_ValueType *)buffer.values);
    }
    template <_binops function>
    static void _op(_ValueType            val,
                    _ul                   baseIdx,
                    _AddressHolder const& addr)
    {
        for (_ul idx=0; idx != _wordCount; ++idx)
        {
            _hwOp<reg_t,
                  function,
                  typename _AddressInfoT::_BusTag>::f(
                      _addrCalc<_AddressInfoT> (baseIdx, addr)
                          + idx * _step,
                      reinterpret_cast<reg_t *>(val)[idx]);
        }
    }
};
```

B.2.1.4 The Interface `register_access`

`_IndexHolder` is a helper class to hold the index for the return value of the subscript operator of `register_access`:

```
template <typename T>
struct _IndexHolder
{
    _IndexHolder(T const &v) : value_(v) {}

    T value_;
    T value() const { return value_; }
};
```

The actual interface for `register_access` is realized by the *class template* `register_access`. As `register_access` provides the full interface for single registers, plus a subscript operator that returns an object that again provides the full register interface, this common interface is separated into a common base *class template* `_RAInterface`. This in turn uses a helper class `_RAImpl` to forward the operations to the correct instantiation of `_AccessHelper`.

`_RAImpl` is just a helper to save a lot of typing for all the template arguments[35]:

```
template <class _AcType, class _AddressHolder, typename _IndexType>
class _RAImpl
{
public:
    typedef typename _AcType::value_type value_type;
    static value_type _read(_AddressHolder const& _addr,
                            _IndexType const&     _idx)
    {
        return _AccessHelper<value_type,
                             _AcType::device_endian::value_,
                             typename _AcType::_AddressT>
                                 ::_read(_idx.value(), _addr);
    }

    template <_binops function>
    static void _op(_AddressHolder const& _addr,
                    _IndexType const&     _idx,
                     value_type           _val)
    {
        _AccessHelper<value_type,
                      _AcType::device_endian::value_,
                      typename _AcType::_AddressT>
                           ::_op<function>(_val, _idx.value(), _addr);
    }
};
```

[35] It would not need to be a separate class if *typedef-template*s were allowed.

`_RAInterface` provides the common interface for single register access classes as well as for the return types of the subscript operator of base register access classes. It just forwards all the interface functions to `_RAImpl`:

```
template <class _AcType, class _AddressHolder, typename _IndexType>
class _RAInterface
{
public:
    typedef typename _AcType::value_type                 value_type;
    typedef _RAImpl<_AcType, _AddressHolder, _IndexType> _Impl;

    _RAInterface() : _addr(), _idx() {}
    explicit _RAInterface(typename _AcType::dynamic_data const& _d)
    : _addr(_d), _idx() {}
    _RAInterface(_AddressHolder const& _a, _IndexType const& _i)
    : _addr(_a), _idx(_i) {}

    operator value_type() const
    {
        return _Impl::_read(_addr, _idx);
    }
    value_type read() const
    {
        return _Impl::_read(_addr, _idx);
    }
    void operator = (value_type val)
    {
        _Impl::template _op<_write_op>(_addr, _idx, val);
    }
    void operator |= (value_type val)
    {
        _Impl::template _op<_or_op>(_addr, _idx, val);
    }
    void operator &= (value_type val)
    {
        _Impl::template _op<_and_op>(_addr, _idx, val);
    }
    void operator ^= (value_type val)
    {
        _Impl::template _op<_xor_op>(_addr, _idx, val);
    }

protected:
    const _AddressHolder _addr;
    const _IndexType     _idx;
};
```

`register_access` is the final class provided for the user. As there is a major performance difference between types for which everything is known at compile time (no runtime address computations). and types with some dynamic data where the actual address can only computed at run-time, the `register_access` template comes in two versions:

- a general `template` for all cases, and
- a specialization for access types with static data only.

The actual interface is taken from _RAInterface, but as this is not intended to be a public base class, private inheritance and *using-declaration*s are used.

```
// the second template-parameter is only to provide a tag
// for partial specialization for non-dynamic data
template <class _AcType,
          class _DynData = typename _AcType::dynamic_data>
class register_access
    : private _RAInterface<_AcType,
                      typename _AcType::_BaseAddressHolder,
                      _Int2Type<0> >
{
    typedef typename _AcType::_BaseAddressHolder _AddressHolder;
    typedef _RAInterface<_AcType,
                     _AddressHolder,
                     _IndexHolder<_ul> > _RefT;
    typedef _RAInterface<_AcType,
                     typename _AcType::_BaseAddressHolder,
                     _Int2Type<0> > _Base;
    using _Base::_addr;   // this should not be necessary, but it wont
                          // compile without it

public:
    explicit register_access(typename _AcType::dynamic_data const& d)
        : _RAInterface<_AcType,
                   typename _AcType::_BaseAddressHolder,
                   _Int2Type<0> >(d) {}

    using typename _Base::value_type;
    using _Base::operator value_type;
    using _Base::read;
    using _Base::operator =;
    using _Base::operator |=;
    using _Base::operator &=;
    using _Base::operator ^=;

    _RefT operator [] (size_t index) const
    {
        return _RefT(_addr, index);
    }
    _RefT operator [] (ptrdiff_t index) const
    {
        return _RefT(_addr, index);
    }
};
```

For the static specialization, the _AddressHolder knows all necessary data in the type and contains no real object data.

```
// specialization for no dynamic data
template <class _AcType>
class register_access<_AcType, _EmptyType>
    : private _RAInterface<_AcType,
                      typename _AcType::_BaseAddressHolder,
                      _Int2Type<0> >
{
    typedef typename _AcType::_BaseAddressHolder _AddressHolder;
    typedef _RAInterface<_AcType,
                     _AddressHolder,
                     _IndexHolder<_ul> > _RefT;
    typedef _RAInterface<_AcType,
                         _AddressHolder,
                         _Int2Type<0> > _Base;

public:
//    typedef _AcType _AccessType;    // for C interface only

    register_access() {}

    using typename _Base::value_type;
    using _Base::operator value_type;
    using _Base::read;
    using _Base::operator =;
    using _Base::operator |=;
    using _Base::operator &=;
    using _Base::operator ^=;

    _RefT operator [] (size_t index) const
    {
        return _RefT(_AddressHolder(), index);
    }
    _RefT operator [] (ptrdiff_t index) const
    {
        return _RefT(_AddressHolder(), index);
    }
};
```

Appendix C: Implementing the C Interface in Terms of the C++ Interface

The implementation of the basic C register access interface on top is pretty straight-forward. As the creation of unnecessary real objects should be avoided, all `register_access` arguments are given by pointer. As this pointer is never de-referenced when all necessary access data is in the type (static access method), a (properly typed) `null` pointer can be given in that case.

For *access-specification*s with dynamic data, a real object must be created (and properly initialized) in the middle layer, and a pointer to that object given to the interface functions.

The interface functions themselves are implemented as inline *function template*s, not function-like macros as specified in the C interface. This leeway is given to the implementer in the C interface:

```
// Possible implementation for <iohw.h>
#include <hardware>

using namespace std::hardware;


template <typename reg_access>
inline typename reg_access::value_type iord(reg_access* reg)
{
    return static_cast<typename reg_access::value_type>(*reg);
}

template <typename reg_access>
inline void iowr(reg_access*                        reg,
                 typename reg_access::value_type  value)
{
    *reg = value;
}

template <typename reg_access>
inline void ioor(reg_access*                        reg,
                 typename reg_access::value_type  value)
{
    *reg |= value;
}

template <typename reg_access>
inline void ioand(reg_access*                       reg,
                  typename reg_access::value_type value)
{
    *reg &= value;
}
```

```
template <typename reg_access>
inline void ioxor(reg_access*                     reg,
                  typename reg_access::value_type value)
{
    *reg ^= value;
}

template <typename reg_access>
inline typename reg_access::value_type iordbuf(reg_access*  reg,
                                               unsigned int index)
{
    return (*reg)[index];
}

template <typename reg_access>
inline void iowrbuf(reg_access*                      reg,
                    unsigned int                     index,
                    typename reg_access::value_type  value)
{
    (*reg)[index] = value;
}

template <typename reg_access>
inline void ioorbuf(reg_access*                      reg,
                    unsigned int                     index,
                    typename reg_access::value_type  value)
{
    (*reg)[index] |= value;
}

template <typename reg_access>
inline void ioandbuf(reg_access*                     reg,
                     unsigned int                    index,
                     typename reg_access::value_type value)
{
    (*reg)[index] &= value;
}

template <typename reg_access>
inline void ioxorbuf(reg_access*                     reg,
                     unsigned int                    index,
                     typename reg_access::value_type value)
{
    (*reg)[index] ^= value;
}
```

Appendix D: Timing Code

D.1 Measuring the Overhead of Class Operations

This is the sample program discussed in 2.3.2 and following.

```
/*
    Simple/naive measurements to give a rough idea of the relative
    cost of facilities related to OOP.

    This could be fooled/foiled by clever optimizers and by
    cache effects.

    Run at least three times to ensure that results are repeatable.

    Tests:

        virtual function
        global function called indirectly
        nonvirtual member function
        global function
        inline member function
        macro
        1st branch of MI
        2nd branch of MI
        call through virtual base
        call of virtual base function

        dynamic cast
        two-level dynamic cast
        typeid()

        call through pointer to member

        call-by-reference
        call-by-value

        pass as pointer to function
        pass as function object

    not yet:

        co-variant return


    The cost of the loop is not measurable at this precision:
    see inline tests

    By default do 1000000 iterations to cout

    1st optional argument: number of iterations
    2nd optional argument: target file name
 */

//int body(int i) { return i*(i+1)*(i+2); }
```

```
class X {
    int x;
    static int st;
public:
    virtual void f(int a);
    void g(int a);
    static void h(int a);
    void k(int i) { x+=i; }    // inline
};

struct S {
    int x;
};

int glob = 0;

extern void f(S* p, int a);
extern void g(S* p, int a);
extern void h(int a);
typedef void (*PF)(S* p, int a);
PF p[10] = { g , f };
// inline void k(S* p, i) { p->x+=i; }
#define K(p,i)  ((p)->x+=(i))

struct T {
    const char* s;
    double t;

    T(const char* ss, double tt) : s(ss), t(tt) {}
    T() : s(0), t(0) {}
};

struct A {
    int x;
    virtual void f(int) = 0;
    void g(int);
};

struct B {
    int xx;
    virtual void ff(int) = 0;
    void gg(int);
};

struct C : A, B {
    void f(int);
    void ff(int);
};

struct CC : A, B {
    void f(int);
    void ff(int);
};

void A::g(int i)   { x  += i; }
void B::gg(int i)  { xx += i; }
void C::f(int i)   { x  += i; }
void C::ff(int i)  { xx += i; }
void CC::f(int i)  { x  += i; }
void CC::ff(int i) { xx += i; }
```

```
template<class T, class T2> inline T* cast(T* p, T2* q)
{
    glob++;
    return dynamic_cast<T*>(q);
}

struct C2 : virtual A {  // note: virtual base
};

struct C3 : virtual A {
};

struct D : C2, C3 {  // note: virtual base
    void f(int);
};

void D::f(int i)  { x+=i; }

struct P {
    int x;
    int y;
};

void by_ref(P& a) { a.x++; a.y++; }
void by_val(P a)  { a.x++; a.y++; }

template<class F, class V> inline void oper(F f, V val) { f(val); }

struct FO {
    void operator () (int i) { glob += i; }
};

// ---------------------------------------------------------------

#include <stdlib.h>  // Why not <cstdlib> ?
#include <iostream>
#include <fstream>
#include <time.h>    // Why not <ctime> ?
#include <vector>
#include <typeinfo>
using namespace std;

template<class T> inline T* ti(T* p)
{
    if (typeid(p) == typeid(int*))
        p++;
    return p;
}

int main(int argc, char* argv[])
{
    int i;   // loop variable here for the benefit of non-conforming
             // compilers

    int n = (1 < argc) ? atoi(argv[1]) : 10000000;  // number of
                                                    // iterations
```

```
ofstream target;
ostream* op = &cout;
if (2 < argc) {  // place output in file
    target.open(argv[2]);
    op = &target;
}
ostream& out = *op;

// output command for documentation:
for (i = 0; i < argc; ++i)
    out << argv[i] << " ";
out << endl;

X* px = new X;
X  x;
S* ps = new S;
S  s;

vector<T> v;

clock_t t = clock();
if (t == clock_t(-1)) {
    cerr << "sorry, no clock" << endl;
    exit(1);
}

for (i = 0; i < n; i++)
    px->f(1);
v.push_back(T("virtual px->f(1)            ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    p[1](ps, 1);
v.push_back(T("ptr-to-fct p[1](ps,1)       ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    x.f(1);
v.push_back(T("virtual x.f(1)              ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    p[1](&s, 1);
v.push_back(T("ptr-to-fct p[1](&s,1)       ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    px->g(1);
v.push_back(T("member px->g(1)             ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    g(ps, 1);
v.push_back(T("global g(ps,1)              ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    x.g(1);
v.push_back(T("member x.g(1)               ", clock() - t));
```

```
    t = clock();
    for (i = 0; i < n; i++)
        g(&s, 1);
    v.push_back(T("global g(&s,1)              ", clock() - t));

    t = clock();
    for (i = 0; i < n; i++)
        X::h(1);
    v.push_back(T("static X::h(1)              ", clock() - t));

    t = clock();
    for (i = 0; i < n; i++)
        h(1);
    v.push_back(T("global h(1)                 ", clock() - t));

    t = clock();
    for (i = 0; i < n; i++)
        px->k(1);
    v.push_back(T("inline px->k(1)             ", clock() - t));

    t = clock();
    for (i = 0; i < n; i++)
        K(ps, 1);
    v.push_back(T("macro K(ps,1)               ", clock() - t));

    t = clock();
    for (i = 0; i < n; i++)
        x.k(1);
    v.push_back(T("inline x.k(1)               ", clock() - t));

    t = clock();
    for (i = 0; i < n; i++)
        K(&s, 1);
    v.push_back(T("macro K(&s,1)               ", clock() - t));

    C* pc = new C;
    A* pa = pc;
    B* pb = pc;

    t = clock();
    for (i = 0; i < n; i++)
        pc->g(i);
    v.push_back(T("base1 member pc->g(i)       ", clock() - t));

    t = clock();
    for (i = 0; i < n; i++)
        pc->gg(i);
    v.push_back(T("base2 member pc->gg(i)      ", clock() - t));

    t = clock();
    for (i = 0; i < n; i++)
        pa->f(i);
    v.push_back(T("base1 virtual pa->f(i)      ", clock() - t));

    t = clock();
    for (i = 0; i < n; i++)
        pb->ff(i);
    v.push_back(T("base2 virtual pb->ff(i)     ", clock() - t));
```

```
t = clock();
for (i = 0; i < n; i++)
    cast(pa, pc);
v.push_back(T("base1 down-cast cast(pa,pc)  ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    cast(pb, pc);
v.push_back(T("base2 down-cast cast(pb,pc)  ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    cast(pc, pa);
v.push_back(T("base1 up-cast cast(pc,pa)    ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    cast(pc, pb);
v.push_back(T("base2 up-cast cast(pc,pb)    ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    cast(pb, pa);
v.push_back(T("base2 cross-cast cast(pb,pa) ", clock() - t));

CC* pcc = new CC;
pa = pcc;
pb = pcc;

t = clock();
for (i = 0; i < n; i++)
    cast(pa, pcc);
v.push_back(T("base1 down-cast2 cast(pa,pcc)", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    cast(pb, pcc);
v.push_back(T("base2 down-cast  cast(pb,pcc)", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    cast(pcc, pa);
v.push_back(T("base1 up-cast cast(pcc,pa)   ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    cast(pcc, pb);
v.push_back(T("base2 up-cast2 cast(pcc,pb)  ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    cast(pb, pa);
v.push_back(T("base2 cross-cast2 cast(pa,pb)", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    cast(pa, pb);
v.push_back(T("base1 cross-cast2 cast(pb,pa)", clock() - t));
```

```
D* pd = new D;
pa = pd;

t = clock();
for (i = 0; i < n; i++)
    pd->g(i);
v.push_back(T("vbase member pd->gg(i)      ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    pa->f(i);
v.push_back(T("vbase virtual pa->f(i)      ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    cast(pa, pd);
v.push_back(T("vbase down-cast cast(pa,pd)  ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    cast(pd, pa);
v.push_back(T("vbase up-cast cast(pd,pa)    ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    ti(pa);
v.push_back(T("vbase typeid(pa)            ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    ti(pd);
v.push_back(T("vbase typeid(pd)            ", clock() - t));

void (A::* pmf)(int) = &A::f;       // virtual

t = clock();
for (i = 0; i < n; i++)
    (pa->*pmf)(i);
v.push_back(T("pmf virtual (pa->*pmf)(i)   ", clock() - t));

pmf = &A::g;                        // non virtual

t = clock();
for (i = 0; i < n; i++)
    (pa->*pmf)(i);
v.push_back(T("pmf (pa->*pmf)(i)           ", clock() - t));

P pp;

t = clock();
for (i = 0; i < n; i++)
    by_ref(pp);
v.push_back(T("call by_ref(pp)             ", clock() - t));

t = clock();
for (i = 0; i < n; i++)
    by_val(pp);
v.push_back(T("call by_val(pp)             ", clock() - t));
```

```
    FO fct;

    t = clock();
    for (i = 0; i < n; i++)
        oper(h, glob);
    v.push_back(T("call ptr-to-fct oper(h,glob)", clock() - t));

    t = clock();
    for (i = 0; i < n; i++)
        oper(fct, glob);
    v.push_back(T("call fct-obj oper(fct,glob) ", clock() - t));

    if (clock() == clock_t(-1)) {
        cerr << "sorry, clock overflow" <<endl;
        exit(2);
    }

    out << endl;
    for (i = 0; i<v.size(); i++)
        out << v[i].s << " :\t"
            << v[i].t * (double(1000000)/n)/CLOCKS_PER_SEC
            << " ms" << endl;

    if (argc < 2) {    // if output is going to cout
        cout << "press any character to finish" << endl;
        char c;
        cin  >> c;     // to placate Windows console mode
    }

    return 0;          // shut up noncompliant compilers
}

int  X::st = 0;

void X::f(int a)    { x  += a; }
void X::g(int a)    { x  += a; }
void X::h(int a)    { st += a; }

void f(S* p, int a) { p->x += a; }
void g(S* p, int a) { p->x += a; }
void h(int a)       { glob += a; }
```

D.2 Measuring Template Overheads

This is the sample code discussed in 2.5.1.

```
/* ------------------------------------------------
Test program to give a rough measure of "template bloat."
If the macro "DIFFERENT" is defined at compile time, this
 program creates a list<T*> for 100 different types of T.
Otherwise, it creates 100 instances of a list of a single
 pointer type.
A capable compiler will recognise that the binary representation
 of list<T*> is the same for all T and it need retain only a
 single copy of the instantiation code in the program.
 -------------------------------------------------*/

#include <list>

class  x0;
class  x1;
class  x2;
class  x3;
class  x4;
class  x5;
class  x6;
class  x7;
class  x8;
class  x9;
class x10;
class x11;
class x12;
class x13;
class x14;
class x15;
class x16;
class x17;
class x18;
class x19;
class x20;
class x21;
class x22;
class x23;
class x24;
class x25;
class x26;
class x27;
class x28;
class x29;
class x30;
class x31;
class x32;
class x33;
class x34;
class x35;
class x36;
class x37;
class x38;
class x39;
class x40;
class x41;
class x42;
class x43;
class x44;
class x45;
class x46;
class x47;
class x48;
class x49;
class x50;
class x51;
class x52;
class x53;
class x54;
class x55;
class x56;
class x57;
class x58;
class x59;
class x60;
class x61;
class x62;
class x63;
class x64;
class x65;
class x66;
class x67;
class x68;
class x69;
class x70;
class x71;
class x72;
class x73;
class x74;
class x75;
class x76;
class x77;
class x78;
class x79;
class x80;
class x81;
class x82;
class x83;
class x84;
class x85;
class x86;
class x87;
class x88;
class x89;
class x90;
class x91;
class x92;
class x93;
class x94;
class x95;
class x96;
class x97;
class x98;
class x99;

int main()
{
#if defined DIFFERENT  // create 100 lists of different pointer types
   std::list<x0*>  v0;
   std::list<x1*>  v1;
   std::list<x2*>  v2;
```

```
    std::list<x3*>  v3;
    std::list<x4*>  v4;
    std::list<x5*>  v5;
    std::list<x6*>  v6;
    std::list<x7*>  v7;
    std::list<x8*>  v8;
    std::list<x9*>  v9;
    std::list<x10*> v10;
    std::list<x11*> v11;
    std::list<x12*> v12;
    std::list<x13*> v13;
    std::list<x14*> v14;
    std::list<x15*> v15;
    std::list<x16*> v16;
    std::list<x17*> v17;
    std::list<x18*> v18;
    std::list<x19*> v19;
    std::list<x20*> v20;
    std::list<x21*> v21;
    std::list<x22*> v22;
    std::list<x23*> v23;
    std::list<x24*> v24;
    std::list<x25*> v25;
    std::list<x26*> v26;
    std::list<x27*> v27;
    std::list<x28*> v28;
    std::list<x29*> v29;
    std::list<x30*> v30;
    std::list<x31*> v31;
    std::list<x32*> v32;
    std::list<x33*> v33;
    std::list<x34*> v34;
    std::list<x35*> v35;
    std::list<x36*> v36;
    std::list<x37*> v37;
    std::list<x38*> v38;
    std::list<x39*> v39;
    std::list<x40*> v40;
    std::list<x41*> v41;
    std::list<x42*> v42;
    std::list<x43*> v43;
    std::list<x44*> v44;
    std::list<x45*> v45;
    std::list<x46*> v46;
    std::list<x47*> v47;
    std::list<x48*> v48;
    std::list<x49*> v49;
    std::list<x50*> v50;
    std::list<x51*> v51;
    std::list<x52*> v52;
    std::list<x53*> v53;
    std::list<x54*> v54;
    std::list<x55*> v55;
    std::list<x56*> v56;
    std::list<x57*> v57;
    std::list<x58*> v58;
    std::list<x59*> v59;
    std::list<x60*> v60;
    std::list<x61*> v61;
    std::list<x62*> v62;
    std::list<x63*> v63;
    std::list<x64*> v64;
    std::list<x65*> v65;
```

```
   std::list<x66*> v66;
   std::list<x67*> v67;
   std::list<x68*> v68;
   std::list<x69*> v69;
   std::list<x70*> v70;
   std::list<x71*> v71;
   std::list<x72*> v72;
   std::list<x73*> v73;
   std::list<x74*> v74;
   std::list<x75*> v75;
   std::list<x76*> v76;
   std::list<x77*> v77;
   std::list<x78*> v78;
   std::list<x79*> v79;
   std::list<x80*> v80;
   std::list<x81*> v81;
   std::list<x82*> v82;
   std::list<x83*> v83;
   std::list<x84*> v84;
   std::list<x85*> v85;
   std::list<x86*> v86;
   std::list<x87*> v87;
   std::list<x88*> v88;
   std::list<x89*> v89;
   std::list<x90*> v90;
   std::list<x91*> v91;
   std::list<x92*> v92;
   std::list<x93*> v93;
   std::list<x94*> v94;
   std::list<x95*> v95;
   std::list<x96*> v96;
   std::list<x97*> v97;
   std::list<x98*> v98;
   std::list<x99*> v99;

#else            // create 100 instances of a single list<T*> type
   std::list<x0*>  v0;
   std::list<x0*>  v1;
   std::list<x0*>  v2;
   std::list<x0*>  v3;
   std::list<x0*>  v4;
   std::list<x0*>  v5;
   std::list<x0*>  v6;
   std::list<x0*>  v7;
   std::list<x0*>  v8;
   std::list<x0*>  v9;
   std::list<x0*> v10;
   std::list<x0*> v11;
   std::list<x0*> v12;
   std::list<x0*> v13;
   std::list<x0*> v14;
   std::list<x0*> v15;
   std::list<x0*> v16;
   std::list<x0*> v17;
   std::list<x0*> v18;
   std::list<x0*> v19;
   std::list<x0*> v20;
   std::list<x0*> v21;
   std::list<x0*> v22;
   std::list<x0*> v23;
   std::list<x0*> v24;
   std::list<x0*> v25;
   std::list<x0*> v26;
```

```
        std::list<x0*> v27;
        std::list<x0*> v28;
        std::list<x0*> v29;
        std::list<x0*> v30;
        std::list<x0*> v31;
        std::list<x0*> v32;
        std::list<x0*> v33;
        std::list<x0*> v34;
        std::list<x0*> v35;
        std::list<x0*> v36;
        std::list<x0*> v37;
        std::list<x0*> v38;
        std::list<x0*> v39;
        std::list<x0*> v40;
        std::list<x0*> v41;
        std::list<x0*> v42;
        std::list<x0*> v43;
        std::list<x0*> v44;
        std::list<x0*> v45;
        std::list<x0*> v46;
        std::list<x0*> v47;
        std::list<x0*> v48;
        std::list<x0*> v49;
        std::list<x0*> v50;
        std::list<x0*> v51;
        std::list<x0*> v52;
        std::list<x0*> v53;
        std::list<x0*> v54;
        std::list<x0*> v55;
        std::list<x0*> v56;
        std::list<x0*> v57;
        std::list<x0*> v58;
        std::list<x0*> v59;
        std::list<x0*> v60;
        std::list<x0*> v61;
        std::list<x0*> v62;
        std::list<x0*> v63;
        std::list<x0*> v64;
        std::list<x0*> v65;
        std::list<x0*> v66;
        std::list<x0*> v67;
        std::list<x0*> v68;
        std::list<x0*> v69;
        std::list<x0*> v70;
        std::list<x0*> v71;
        std::list<x0*> v72;
        std::list<x0*> v73;
        std::list<x0*> v74;
        std::list<x0*> v75;
        std::list<x0*> v76;
        std::list<x0*> v77;
        std::list<x0*> v78;
        std::list<x0*> v79;
        std::list<x0*> v80;
        std::list<x0*> v81;
        std::list<x0*> v82;
        std::list<x0*> v83;
        std::list<x0*> v84;
        std::list<x0*> v85;
        std::list<x0*> v86;
        std::list<x0*> v87;
        std::list<x0*> v88;
        std::list<x0*> v89;
```

```
   std::list<x0*> v90;
   std::list<x0*> v91;
   std::list<x0*> v92;
   std::list<x0*> v93;
   std::list<x0*> v94;
   std::list<x0*> v95;
   std::list<x0*> v96;
   std::list<x0*> v97;
   std::list<x0*> v98;
   std::list<x0*> v99;

#endif
   return 0;
}
```

D.3 The Stepanov Abstraction Penalty Benchmark

This is the sample code discussed in 2.3.1.

```
/* KAI's version of Stepanov Benchmark -- Version 1.2

   Version 1.2 -- removed some special code for GNU systems that
                 GNU complained about without -O

To verify how efficiently C++ (and in particular STL) is compiled by
the present day compilers, I composed a little benchmark. It outputs
13 numbers. In the ideal world these numbers should be the same. In
the real world, however, ...

The final number printed by the benchmark is a geometric mean of the
performance degradation factors of individual tests. It claims to
represent the factor by which you will be punished by your
compiler if you attempt to use C++ data abstraction features. I call
this number "Abstraction Penalty."

As with any benchmark it is hard to prove such a claim; some people
told me that it does not represent typical C++ usage. It is, however,
a noteworthy fact that majority of the people who so object are
responsible for C++ compilers with disproportionately large Abstraction
Penalty.

The structure of the benchmark is really quite simple. It adds 2000
doubles in an array 25000 times. It does it in 13 different ways that
introduce more and more abstract ways of doing it:

0 - uses simple Fortran-like for loop.
1 - 12 use STL style accumulate template function with plus function object.
1, 3, 5, 7 ,9, 11 use doubles.
2, 4, 6, 8, 10, 12 use Double - double wrapped in a class.
1, 2 - use regular pointers.
3, 4 - use pointers wrapped in a class.
5, 6 - use pointers wrapped in a reverse-iterator adaptor.
7, 8 - use wrapped pointers wrapped in a reverse-iterator adaptor.
9, 10 - use pointers wrapped in a reverse-iterator adaptor wrapped in a
  reverse-iterator adaptor.
11, 12 - use wrapped pointers wrapped in a reverse-iterator adaptor wrapped in a
reverse-iterator adaptor.

All the operators on Double and different pointer-like classes are
declared inline. The only thing that is really measured is the penalty for data
abstraction. While templates are used, they do not cause any performance degradation.
They are used only to simplify the code.

Since many of you are interested in the C++ performance issues, I
decided to post the benchmark here. I would appreciate if you run it
and (if possible) send me the results indicating what you have
compiled it with (CPU, clock rate, compiler, optimization level). It
is self contained and written so that it could be compiled even with
those compilers that at present cannot compile STL at all.

It takes a fairly long time to run - on a really slow machine it might take a full
hour. (For those of you who want to run it faster - give it a command line argument
that specifies the number of
iterations. The default is 25000, but it gives an accurate predictions even with 500
or a thousand.)


Alex Stepanov
stepanov@mti.sgi.com

*/
```

```
#include <stddef.h>
#include <stdio.h>
#include <time.h>
#include <math.h>
#include <stdlib.h>

template <class T>
inline int operator!=(const T& x, const T& y) {
  return !(x == y);
}

struct Double {
  double value;
  Double() {}
  Double(const double& x) : value(x) {}
  operator double() { return value; }
};

inline Double operator+(const Double& x, const Double& y) {
  return Double(x.value + y.value);
}

struct double_pointer {
    double* current;
    double_pointer() {}
    double_pointer(double* x) : current(x) {}
    double& operator*() const { return *current; }
    double_pointer& operator++() {
       ++current;
       return *this;
    }
    double_pointer operator++(int) {
       double_pointer tmp = *this;
       ++*this;
       return tmp;
    }
    double_pointer& operator--() {
       --current;
       return *this;
    }
    double_pointer operator--(int) {
       double_pointer tmp = *this;
       --*this;
       return tmp;
    }
};


inline int operator==(const double_pointer& x,
                     const double_pointer& y) {
    return x.current == y.current;
}

struct Double_pointer {
    Double* current;
    Double_pointer() {}
    Double_pointer(Double* x) : current(x) {}
    Double& operator*() const { return *current; }
    Double_pointer& operator++() {
       ++current;
       return *this;
    }
    Double_pointer operator++(int) {
       Double_pointer tmp = *this;
       ++*this;
       return tmp;
    }
    Double_pointer& operator--() {
       --current;
       return *this;
    }
    Double_pointer operator--(int) {
        Double_pointer tmp = *this;
        --*this;
        return tmp;
    }
};


inline int operator==(const Double_pointer& x,
                      const Double_pointer& y) {
    return x.current == y.current;
}

template <class RandomAccessIterator, class T>
struct reverse_iterator {
    RandomAccessIterator current;
    reverse_iterator(RandomAccessIterator x) : current(x) {}
    T& operator*() const {
      RandomAccessIterator tmp = current;
      return *(--tmp);
    }
    reverse_iterator<RandomAccessIterator, T>& operator++() {
        --current;
        return *this;
    }
    reverse_iterator<RandomAccessIterator, T> operator++(int) {
      reverse_iterator<RandomAccessIterator, T> tmp = *this;
        ++*this;
        return tmp;
    }
    reverse_iterator<RandomAccessIterator, T>& operator--() {
        ++current;
        return *this;
    }
    reverse_iterator<RandomAccessIterator, T> operator--(int) {
      reverse_iterator<RandomAccessIterator, T> tmp = *this;
        --*this;
        return tmp;
    }
};

template <class RandomAccessIterator, class T>
inline
int operator==(const reverse_iterator<RandomAccessIterator, T>& x,
        const reverse_iterator<RandomAccessIterator, T>& y) {
    return x.current == y.current;
}

struct {
  double operator()(const double& x, const double& y)  {
        return x + y;
  }
  Double operator()(const Double& x, const Double& y) {
      return x + y;
  }
} plus;


template <class Iterator, class Number>
Number accumulate(Iterator first, Iterator last, Number result) {
  while (first != last) result =  plus(result, *first++);
  return result;
}

int iterations = 25000;
#define SIZE 2000

int current_test = 0;

double result_times[20];
```

```
void summarize() {
  printf("\ntest      absolute   additions      ratio with\n");
  printf("number    time       per second     test0\n\n");
  int i;
  double millions = (double(SIZE) * iterations)/1000000.;
  for (i = 0; i < current_test; ++i)
    printf("%2i       %5.2fsec    %5.2fM         %.2f\n",
          i,
          result_times[i],
          millions/result_times[i],
          result_times[i]/result_times[0]);
  double gmean_times = 0.;
  double total_absolute_times = 0.;  // sam added 12/05/95
  double gmean_rate = 0.;
  double gmean_ratio = 0.;
  for (i = 0; i < current_test; ++i) {
    total_absolute_times += result_times[i];  // sam added 12/05/95
    gmean_times += log(result_times[i]);
    gmean_rate  += log(millions/result_times[i]);
    gmean_ratio += log(result_times[i]/result_times[0]);
  }
  printf("mean:    %5.2fsec    %5.2fM         %.2f\n",
        exp(gmean_times/current_test),
        exp(gmean_rate/current_test),
        exp(gmean_ratio/current_test));
 printf("\nTotal absolute time: %.2f sec\n",total_absolute_times);//sam added 12/05/95
  printf("\nAbstraction Penalty: %.2f\n\n", exp(gmean_ratio/current_test));
}

clock_t start_time, end_time;

inline void start_timer() { start_time = clock(); }

inline double timer() {
  end_time = clock();
  return (end_time - start_time)/double(CLOCKS_PER_SEC);
}

const double init_value = 3.;



double data[SIZE];

Double Data[SIZE];

inline void check(double result) {
  if (result != SIZE * init_value) printf("test %i failed\n", current_test);
}

void test0(double* first, double* last) {
  start_timer();
  for(int i = 0; i < iterations; ++i) {
    double result = 0;
    for (int n = 0; n < last - first; ++n) result += first[n];
    check(result);
  }
  result_times[current_test++] = timer();
}


template <class Iterator, class T>
void test(Iterator first, Iterator last, T zero) {
  int i;
  start_timer();
  for(i = 0; i < iterations; ++i)
    check(double(accumulate(first, last, zero)));
  result_times[current_test++] = timer();
}

template <class Iterator, class T>
```

```
void fill(Iterator first, Iterator last, T value) {
  while (first != last) *first++ = value;
}


double d = 0.;
Double D = 0.;
typedef double* dp;
dp dpb = data;
dp dpe = data + SIZE;
typedef Double* Dp;
Dp Dpb = Data;
Dp Dpe = Data + SIZE;
typedef double_pointer dP;
dP dPb(dpb);
dP dPe(dpe);
typedef Double_pointer DP;
DP DPb(Dpb);
DP DPe(Dpe);
typedef reverse_iterator<dp, double> rdp;
rdp rdpb(dpe);
rdp rdpe(dpb);
typedef reverse_iterator<Dp, Double> rDp;
rDp rDpb(Dpe);
rDp rDpe(Dpb);
typedef reverse_iterator<dP, double> rdP;
rdP rdPb(dPe);
rdP rdPe(dPb);
typedef reverse_iterator<DP, Double> rDP;
rDP rDPb(DPe);
rDP rDPe(DPb);
typedef reverse_iterator<rdp, double> rrdp;
rrdp rrdpb(rdpe);
rrdp rrdpe(rdpb);
typedef reverse_iterator<rDp, Double> rrDp;
rrDp rrDpb(rDpe);
rrDp rrDpe(rDpb);
typedef reverse_iterator<rdP, double> rrdP;
rrdP rrdPb(rdPe);
rrdP rrdPe(rdPb);
typedef reverse_iterator<rDP, Double> rrDP;
rrDP rrDPb(rDPe);
rrDP rrDPe(rDPb);

int main(int argv, char** argc) {
  if (argv > 1) iterations = atoi(argc[1]);
  fill(dpb, dpe, double(init_value));
  fill(Dpb, Dpe, Double(init_value));
  test0(dpb, dpe);
  test(dpb, dpe, d);
  test(Dpb, Dpe, D);
  test(dPb, dPe, d);
  test(DPb, DPe, D);
  test(rdpb, rdpe, d);
  test(rDpb, rDpe, D);
  test(rdPb, rdPe, d);
  test(rDPb, rDPe, D);
  test(rrdpb, rrdpe, d);
  test(rrDpb, rrDpe, D);
  test(rrdPb, rrdPe, d);
  test(rrDPb, rrDPe, D);
  summarize();
  return 0;
}
```

D.4 Comparing Function Objects to Function Pointers

2.6 mentions that optimizers work better with function objects than function pointers. This program attempts to measure any benefit.

```
//=============================================================================
// This is a program to measure the relative efficiency of qsort vs std::sort
// and of function objects vs function pointers.
//
// Optional Arguments: number of iterations to repeat
//                     size of array of doubles to sort
//                     name of output file
//
// In all cases, an array of doubles is filled with random numbers.
// This array is sorted in ascending order, then the same random numbers are
// reloaded into the array and sorted again. Repeat ad libitum.
//
//
// What is measured:
// These measurements operate on an array of doubles
// 1. Using qsort + user-defined comparison function to sort array
// 2. Using std::sort + a function pointer (not a function object)
// 3. Using std::sort + user-defined function object, out-of-line code
// 4. Using std::sort + user-defined function object, inline code
// 5. Using std::sort + std::less
// 6. Using std::sort + native operator <
//
// These measurements operate on an std::vector of doubles
//
// 7. Using std::sort + std::less,
// 8. Using std::sort + native operator <,
//         and a vector instead of a primitive array
// 9. Using std::sort + function pointer from test 2
//
//
// Since qsort's comparison function must return int (less than 0, 0, greater than 0)
// and std::sort's must return a bool, it is not possible to test them with each
// other's comparator.

//=============================================================================
// struct to hold identifier and elapsed time
struct T {
    const char* s;
    double t;

    T(const char* ss, double tt) : s(ss), t(tt) {}
    T() : s(0), t(0) {}
};

// ---------  helper functions -------------------------------------------
// qsort passes void * arguments to its comparison function,
// which must return negative, 0, or positive value

int
less_than_function1( const void * lhs, const void * rhs )
{
    int retcode = 0;
    if( *(const double *) lhs < *(const double *) rhs ) retcode = -1;
    if( *(const double *) lhs > *(const double *) rhs ) retcode = 1;
    return retcode;
}

// std::sort, on the other hand, needs a comparator that returns true or false
bool
less_than_function2( const double lhs, const double rhs )
{
    if( lhs < rhs ) return true;
    else return false;
```

```
}


// the comparison operator in the following functor is defined out of line
struct less_than_functor
{
    bool operator()( const double& lhs, const double& rhs ) const;
};

bool
less_than_functor::operator()( const double& lhs, const double& rhs ) const
{
    return( lhs < rhs? true : false );
}

// the comparison operator in the following functor is defined inline
struct inline_less_than_functor
{
    bool operator()( const double& lhs, const double& rhs ) const
    {
       return( lhs < rhs? true : false );
    }
};


// -------------------------------------------------------------------------------
#include <vector>
#include <functional>
#include <algorithm>
#include <iostream>
#include <fstream>
#include <ctime>
#include <stdlib.h>

using namespace std;

int main(int argc, char* argv[])
{

    int i;

    int iterations = (1 < argc) ? atoi(argv[1]) : 1000000;  // number of
                                                           // iterations
    int tablesize = (2 < argc) ? atoi(argv[2]) : 1000000;  // size of
                                                    // array

    ofstream target;
    ostream* op = &cout;
    if (3 < argc) {  // place output in file
        target.open(argv[3]);
        op = &target;
    }
    ostream& out = *op;


    // output command for documentation:
    for (i = 0; i < argc; ++i)
        out << argv[i] << " ";
    out << endl;

    vector<T> v;                       // holds elapsed time of the tests

    // seed the random number generator
    srand( clock() );
    clock_t t = clock();
    if (t == clock_t(-1))
    {
        cerr << "sorry, no clock" << endl;
        exit(1);
    }
```

```
// initialize the table to sort. we use the same table for all tests,
// in case one randomly-generated table might require more work than
// another to sort
double * master_table = new double[tablesize];
for( int n = 0; n < tablesize; ++n )
{
    master_table[n] = static_cast<double>( rand() );
}

double * table = new double[tablesize];                // working copy

// here is where the timing starts
// TEST 1: qsort with a C-style comparison function
copy(master_table, master_table+tablesize, table);
t = clock();
for (i = 0; i < iterations; ++i)
{
    qsort( table, tablesize, sizeof(double), less_than_function1 );
    copy(master_table, master_table+tablesize, table);
}
v.push_back(T("qsort array with comparison function1         ", clock() - t));


//TEST 2: std::sort with function pointer
copy(master_table, master_table+tablesize, table);
t = clock();
for (i = 0; i < iterations; ++i)
{
    sort( table, table + tablesize, less_than_function2 );
    copy(master_table, master_table+tablesize, table);
}
v.push_back(T("sort array with function pointer              ", clock() - t) );

// TEST 3: std::sort with out-of-line functor
copy(master_table, master_table+tablesize, table);
t = clock();
for (i = 0; i < iterations; ++i)
{
    sort( table, table + tablesize, less_than_functor() );
    copy(master_table, master_table+tablesize, table);
}
v.push_back(T("sort array with user-supplied functor         ", clock() - t));

// TEST 4: std::sort with inline functor
copy(master_table, master_table+tablesize, table);
t = clock();
for (i = 0; i < iterations; ++i)
{
    sort( table, table + tablesize, inline_less_than_functor() );
    copy(master_table, master_table+tablesize, table);
}
v.push_back(T("sort array with user-supplied inline functor  ", clock() - t));

//TEST 5: std::sort with std::<less> functor
copy( master_table, master_table+tablesize, table );
t = clock();
for (i = 0; i < iterations; ++i)
{
    sort( table, table + tablesize, less<double>() );
    copy(master_table, master_table+tablesize, table);
}
v.push_back(T("sort array with standard functor              ", clock() - t));

//TEST 6: std::sort using native operator <
copy( master_table, master_table+tablesize, table );
t = clock();
for (i = 0; i < iterations; ++i)
{
    sort( table, table + tablesize );
    copy(master_table, master_table+tablesize, table);
}
```

```
    v.push_back(T("sort array with native < operator             ", clock() - t));


    //TEST 7: std::sort with std::less functor,
    //     on a vector rather than primitive array
    vector<double> v_table(  master_table, master_table+tablesize );
    t = clock();
    for (i = 0; i < iterations; ++i)
    {
        sort( v_table.begin(), v_table.end(), less<double>() );
        copy( master_table, master_table+tablesize, v_table.begin() );
    }
    v.push_back(T("sort vector with standard functor             ", clock() - t));

    //TEST 8: std::sort vector using native operator <
    v_table.assign(  master_table, master_table+tablesize );
    t = clock();
    for (i = 0; i < iterations; ++i)
    {
        sort( v_table.begin(), v_table.end() );
        copy( master_table, master_table+tablesize, v_table.begin() );
    }
    v.push_back(T("sort vector with native < operator            ", clock() - t));


    //TEST 9: std::sort vector using function pointer from test 2
    v_table.assign(  master_table, master_table+tablesize );
    t = clock();
    for (i = 0; i < iterations; ++i)
    {
        sort( v_table.begin(), v_table.end(), less_than_function2 );
        copy( master_table, master_table+tablesize, v_table.begin() );
    }
    v.push_back(T("sort vector with function pointer             ", clock() - t));


    if (clock() == clock_t(-1))
    {
        cerr << "sorry, clock overflow" <<endl;
        exit(2);
    }

    // output results
    out << endl;
    for (i = 0; i<v.size(); i++)
        out << v[i].s << " :\t"
            << v[i].t /CLOCKS_PER_SEC
            << " seconds" << endl;
     delete[] table;
     return 0;
}
```

D.5 Measuring the Cost of Synchronized I/O

2.6 discusses using `sync_with_stdio(false)` to improve I/O performance. This program attempts to measure any benefit.

```
/* -------------------------------------------------

  Test program to
 (1) compare the performance of classic iostreams,
  standard iostreams, and C-style stdio for output, and
 (2) test any overhead of sync_with_stdio(true). Standard
 iostreams by default are synchronized with stdio streams;
 the opposite was true of classic iostreams.

 optional command line argument:
 - how many numbers to output (default 1,000,000)

  When compiling, define CLASSIC or STDIO to enable
 those options; otherwise the default is to use
 standard iostreams.
 --------------------------------------------------*/
#include <vector>
#include <ctime>

#if defined (STDIO)
   #include <stdio.h>
#elif defined (CLASSIC)
   #include <iostream.h>
#else
   #include <iostream>                  // use standard iostreams
#endif

   using namespace std;


//=============================================================================
// struct to hold identifier and elapsed time
struct T {
    const char* s;
    double t;

    T(const char* ss, double tt) : s(ss), t(tt) {}
    T() : s(0), t(0) {}
};



int main (int argc, char *argv[])
{
    const int n = (1 < argc) ? atoi(argv[1]) : 1000000;  // number of
                                                          // iterations
    int i;                              // loop variable


    vector<T> v;                        // holds elapsed time of the tests

    #if defined (CLASSIC)
    // non-synchronized I/O is the default
    #else
    cout.sync_with_stdio (false);       // must be called before any output
    #endif

    // seed the random number generator
    srand( clock() );
    clock_t t = clock();
    if (t == clock_t(-1))
    {
#if defined( STDIO )
        fprintf( stderr, "sorry, no clock\n" );
#else
```

```
        cerr << "sorry, no clock" << endl;
#endif
        exit(1);
    }


#if defined( STDIO )
    t = clock();
    for (i = 0; i != n; ++i)
    {
       printf ("%d ", i);
    }
    v.push_back(T("output integers to stdio                      ", clock() - t));

    t = clock();
    for ( i = 0; i != n; ++i)
    {
       printf ("%x ", i);
    }
    v.push_back(T("output hex integers to stdio                  ", clock() - t));

    if (clock() == clock_t(-1))
    {
        fprintf ( stderr, "sorry, clock overflow\n" );
        exit(2);
    }

    // output results
    fprintf ( stderr, "\n" );
    for (i = 0; i<v.size(); i++)
        fprintf( stderr, "%s :\t%f seconds\n", v[i].s, v[i].t /CLOCKS_PER_SEC );

#else
    t = clock();
    for ( i = 0; i != n; ++i)
    {
            cout << i << ' ';
    }
    v.push_back(T("output integers to cout (sync = false)       ", clock() - t));

    cout << hex;
    t = clock();
    for ( i = 0; i != n; ++i)
    {
            cout << i << ' ';
    }
    v.push_back(T("output hex integers to cout (sync = false)   ", clock() - t));

    #if defined (CLASSIC)
    cout.sync_with_stdio();             // synchronize -- no argument needed
    #else
    cout.sync_with_stdio (true);
    #endif

    cout << dec;
    t = clock();
    for ( i = 0; i != n; ++i)
    {
            cout << i << ' ';
    }
    v.push_back(T("output integers to cout (sync = true)        ", clock() - t));

    cout << hex;
    t = clock();
    for ( i = 0; i != n; ++i)
    {
            cout << i << ' ';
    }
    v.push_back(T("output hex integers to cout (sync = true)     ", clock() - t));

    if (clock() == clock_t(-1))
    {
        cerr << "sorry, clock overflow" <<endl;
        exit(2);
    }

    // output results
    cerr << endl;
    for (i = 0; i<v.size(); i++)
        cerr << v[i].s << " :\t"
            << v[i].t /CLOCKS_PER_SEC
            << " seconds" << endl;
#endif

    return 0;

}
```

Appendix E: Bibliography

These references may serve as a starting point for finding more information about programming for performance.

[BIBREF-1] 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 optimize the space and time consumed by an application (often by trading one resource to minimize 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."

[BIBREF-2] 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."

[BIBREF-3] C++ ABI Group

C++ ABI for Itanium (Draft)

`http://www.codesourcery.com/cxx-abi/abi.html`

Although this document contains processor-specific material for the Itanium 64-bit Application Binary Interface, it is intended as a generic specification, to be usable by C++ implementations on a variety of architectures. It discusses implementation details of virtual table layout, exception handling support structures, Run-Time Type Information, name mangling, stack unwinding, and template instantiation.

[BIBREF-4] Cusumano, Michael A., and David B. Yoffie

What Netscape Learned from Cross-Platform Software Development
Communications of the ACM, October 1999.

Faster run-time performance brings commercial advantage, sometimes enough to outweigh other considerations such as portability and maintainability (an argument also advanced in the Bulka-Mayhew book [BIBREF-2]).

[BIBREF-5] de Dinechin, Christophe

C++ Exception Handling

IEEE Concurrency, October-December 2000

[BIBREF-6] 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)
- run-time type identification (RTTI)
- templates
- multiple inheritance (MI)
- namespaces

[BIBREF-7] 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."

[BIBREF-8] 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.

[BIBREF-9] 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.

[BIBREF-10] Hewlett-Packard Corp.

CXperf User's Guide

`http://docs.hp.com/hpux/onlinedocs/B6323-96001/B6323-96001.html`

This guide describes the CXperf Performance Analyzer, an interactive run-time performance analysis tool for programs compiled with HP ANSI C (c89), ANSI C++ (aCC), Fortran 90 (f90), and HP Parallel 32-bit Fortran 77 (f77) compilers. This guide helps you prepare your programs for profiling, run the programs, and analyze the resulting performance data.

Vendors of development tools often provide guidance on programming for maximum performance. This is one of such documents available.

[BIBREF-11] Knuth, Donald E.

The Art of Computer Programming, Volume 1, Reissued 3rd Edition
Addison-Wesley

Fundamental Algorithms	[1997]
Semi-numerical Algorithms	[1998]
Sorting and Searching	[1998]

The definitive work on issues of algorithmic efficiency.

[BIBREF-12] Koenig, A., and B. Stroustrup

Exception Handling for C++ (revised)
Proceedings of the 1990 Usenix C++ Conference, pp149-176, San Francisco, April 1990.

This paper discusses the two approaches to low-overhead exception handling.

[BIBREF-13] 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]

Because of the interaction of many factors, measuring the run-time performance of a program can be surprisingly difficult.

“The most important way to obtain good performance is to use good algorithms.”

[BIBREF-14] 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.

[BIBREF-15] 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.

[BIBREF-16] Levine, John R.

Linkers & Loaders
Morgan Kaufmann Publishers, 2000

This book explains the mechanisms which enable static and dynamic linking to create executable programs from multiple translation units.

[BIBREF-17] 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.

[BIBREF-18] Liu, Yanhong A., and Gustavo Gomez

Automatic Accurate Cost-Bound Analysis for High-Level Languages
IEEE Transactions on Computers, Vol. 50, No. 12, December 2001

This paper describes a language-independent approach to assigning cost parameters to various language constructs, then through static analysis and transformations automatically calculating the cost bounds of whole programs. Example programs in this article are written in a subset of Scheme, not C++. The article discusses how to obtain cost bounds in terms of costs of language primitives, though it does not really discuss how to obtain such costs.

However, it includes a list of references to other resources discussing how to perform respective measurements for different hardware architectures and programming languages.

"It is particularly important for many applications, such as real-time systems and embedded systems, to be able to predict accurate time bounds and space bounds automatically and efficiently and it is particularly desirable to be able to do so for high-level languages."

[BIBREF-19] Meyers, Scott

Effective C++: 50 Specific Ways to Improve Your Programs and Design
Second Edition, Addison-Wesley, 1997.

More Effective C++: 35 New Ways to Improve Your Programs and Designs
Addison-Wesley, 1995.

Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library
Addison-Wesley, 2001.

In keeping with the philosophy of the Standard Library, this book carefully documents the performance implications of different choices in design and coding, such as whether to use `std::map::operator[]` or `std::map::insert`.

*"The fact that function pointer parameters inhibit inlining explains an observation that long-time C programmers often find hard to believe: C++'s sort virtually always embarrasses C's `qsort` when it comes to speed. Sure, C++ has function and class templates to instantiate and funny-looking `operator()` functions to invoke while C makes a simple function call, but all that C++ "overhead" is absorbed during compilation... It's easy to verify that when comparing function objects and real functions as algorithm parameters, there's an abstraction **bonus**."*

[BIBREF-20] Mitchell, Mark

Type-Based Alias Analysis
Dr. Dobbs' Journal, October 2000.

Some techniques for writing source code that is easier for a compiler to optimize.

"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."

[BIBREF-21] Moss, Darren G.

Embedded Systems Conference Proceedings

Efficient C/C++ Coding Techniques
Boston, 2001
`http://www.esconline.com/db_area/01boston/304.pdf`

The objective of this entire treatment is to determine if the speed and size disadvantages of C/C++ can be minimized for a range of compiler/microprocessor platforms. This study resoundly says: yes. The assembly output of common C/C++ constructs demonstrate that the correct selection of coding techniques does guide the compiler to produce efficient code.

[BIBREF-22] Musser, David R., Gillmer J. Derge, and Atul Saini

STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library
Addison-Wesley, 2001.

Among the tutorial material and example code is a chapter describing a class framework for timing generic algorithms.

[BIBREF-23] Noble, James, and Charles Weir

Small Memory Software: Patterns for Systems with Limited Memory
Addison-Wesley, 2001

A book of design patterns illustrating a number of strategies for coping with memory constraints.

"But what is small memory software? Memory size, like riches or beauty, is always relative. Whether a particular amount of memory is small or large depends on the requirements the software should meet, on the underlying software and hardware architecture, and on much else. A weather-calculation program on a vast computer may be just as constrained by memory limits as a word-processor running on a mobile phone, or an embedded application on a smart card. Therefore:

Small memory software is any software that doesn't have as much memory as you'd like!"

[BIBREF-24] Prechelt, Lutz

Technical Opinion: Comparing Java vs. C/C++ Efficiency Differences to Interpersonal Differences
Communications of the ACM, October 1999.

This article compares the memory footprint and run-time performance of 40 implementations of the same program, written in C++, C, and Java. The difference between individual programmers was more significant than the difference between languages.

“The importance of an efficient technical infrastructure (such as language/compiler, operating system, or even hardware) is often vastly overestimated compared to the importance of a good program design and an economical programming style.”

[BIBREF-25] Quiroz, César A.

Embedded Systems Conference Proceedings

Using C++ Efficiently In Embedded Applications
San Jose, CA, Nov. 1998
`http://esconline.com/db_area/98fall/pdf/401.pdf`

[BIBREF-26] Saks, Dan

C++ Theory and Practice
C/C++ Users Journal

Standard C++ as a High-Level Language?	[Nov ‘99]
Replacing Character Arrays with Strings, Part 1	[Jan ‘00]
Replacing Character Arrays with Strings, Part 2	[Feb ‘0]

These articles are part of a series on migrating a C program to use the greater abstraction and encapsulation available in C++. The run-time and executable size are measured as more C++ features are added, such as Standard strings, *IOStreams*, and containers.

“A seemingly small change in a string algorithm [such as reserving space for string data, or erasing the data as an additional preliminary step,] might produce a surprisingly large change in program execution time.”

The conclusion is that you should "program at the highest level of abstraction that you can afford”.

[BIBREF-27] Saks, Dan

Embedded Systems Conference Proceedings Reducing Run-Time Overhead in C++ Programs
San Francisco, March 2002
`http://www.esconline.com/db_area/02sf/405.pdf`

Representing and Manipulating Hardware in Standard C and C++
San Francisco, March 2002
`http://www.esconline.com/db_area/02sf/465.pdf`

Programming Pointers
Embedded Systems Programming

Placing Data into ROM	[May 1998]
Placing Data into ROM with Standard C	[Nov. 1998]
Static vs. Dynamic Initialization	[Dec. 1998]
Ensuring Static Initialization in C++	[March 1999]

[BIBREF-28] Schilling, Jonathan

Optimizing Away C++ Exception Handling
ACM SIGPLAN Notices, August 1998

> This article discusses ways to measure the overhead, if any, of the exception handling mechanisms. A common implementation of EH incurs no run-time penalty unless an exception is actually thrown, but at a cost of greater static data space and some interference with compiler optimizations. 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."*

[BIBREF-29] Stepanov, Alex

The Standard Template Library

Byte Magazine, October 1995, also at

`http://www.byte.com/art/9510/sec12/art3.htm`

The originator of the Standard Template Library discusses the emphasis on efficiency which motivated its design.

[H]ow do you know that a generic algorithm is efficient? An algorithm is called relatively efficient if it's as efficient as a nongeneric version written in the same language, and it's called absolutely efficient if it's as efficient as a nongeneric assembly language version.

For many years, I tried to achieve relative efficiency in more advanced languages (e.g., Ada and Scheme) but failed. My generic versions of even simple algorithms were not able to compete with built-in primitives. But in C++ I was finally able to not only accomplish relative efficiency but come very close to the more ambitious goal of absolute efficiency. To verify this, I spent countless hours looking at the assembly code generated by different compilers on different architectures.

I found that efficiency and generality were not mutually exclusive. In fact, quite the reverse is true. If a component is not efficient enough, it usually means that it's not abstract enough. This is because efficiency and abstractness both require a clean, orthogonal design.

[BIBREF-30] Stroustrup, Bjarne

The C++ Programming Language, Special 3rd Edition
Addison-Wesley, 2000

This definitive work from the language's author has been extensively revised to present Standard C++.

[BIBREF-31] 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."*

[BIBREF-32] 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."

[BIBREF-33] Sutter, Herb

Exceptional C++
Addison-Wesley, 2000.

This book includes a long discussion on minimizing compile-time dependencies using compiler firewalls (the PIMPL idiom), and how to compensate for the space and run-time consequences.

[BIBREF-34] Tribolet, Chuck, and John Palmer

Embedded Systems Conference Proceedings
available on CD from `http://www.esconline.com`

Embedded C and C++ Compiler Evaluation Methodology
Fall 1999
`http://www.esconline.com/db_area/99fall/443.pdf`

Be aggressive about trying compiler options. The compilers each have many options, and it is important to arrive at the best set of the options for each compiler.... A thorough tweaking of compiler options will frequently generate an improvement on the order of 30% over an initial decent set of options. If the initial set is truly abysmal, the improvement could be in excess of 100%.

[BIBREF-35] Veldhuizen, Todd

Five compilation models for C++ templates
Proceedings of the 2000 Workshop on C++ Template Programming

`http://www.oonumerics.org/tmpw00`

This paper describes a work in progress on a new C++ compiler. Type analysis is removed from the compiler and replaced with a type system library, which is treated as source code by the compiler.

"By making simple changes to the behavior of the partial evaluator, a wide range of compilation models is achieved, each with a distinct trade-off of compile-time, code size, and execution speed. ... This approach may solve several serious problems in compiling C++: it achieves separate compilation of templates, allows template code to be distributed in binary form by deferring template instantiation until run-time, and reduces the code bloat associated with templates."

[BIBREF-36] Vollmann, Detlef

Exception Handling Alternatives
Published by ACCU – Overload, Issues 30 and 31 (February 1999)

`http://www.accu.org/c++sig/public/Overload.html`
`http://www.vollmann.ch/en/pubs/cpp-excpt-alt.html`

This article shows some pros and cons of the C++ exception handling mechanism and outlines several possible alternative approaches.

[BIBREF-37] 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."

[BIBREF-38]