TECHNICAL REPORT

ISO/IEC TR 18015

First edition
2006-09-01

Information technology — Programming languages, their environments and system software interfaces — Technical Report on C++ Performance

Technologies de l'information — Langages de programmation, leurs environnements et interfaces du logiciel système — Rapport technique sur la performance C++

Reference number
ISO/IEC TR 18015:2006(E)

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of the joint technical committee is to prepare International Standards. Draft International Standards adopted by the joint technical committee are circulated to national bodies for voting. Publication as an International Standard requires approval by at least 75 % of the national bodies casting a vote.

In exceptional circumstances, the joint technical committee may propose the publication of a Technical Report of one of the following types:

— type 1, when the required support cannot be obtained for the publication of an International Standard, despite repeated efforts;

— type 2, when the subject is still under technical development or where for any other reason there is the future but not immediate possibility of an agreement on an International Standard;

— type 3, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example).

Technical Reports of types 1 and 2 are subject to review within three years of publication, to decide whether they can be transformed into International Standards. Technical Reports of type 3 do not necessarily have to be reviewed until the data they provide are considered to be no longer valid or useful.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 18015, which is a Technical Report of type 3, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces.*

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, as has 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 Technical Report 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 Technical Report 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.

Participants

The following people contributed work to this Technical Report:

Dave Abrahams
Mike Ball
Walter Banks
Greg Colvin
Embedded C++ Technical Committee (Japan)
Hiroshi Fukutomi
Lois Goldthwaite
Yenjo Han
John Hauser
Seiji Hayashida
Howard Hinnant
Brendan Kehoe
Robert Klarer
Jan Kristofferson
Dietmar Kühl
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
Willem Wakker

Information technology — Programming languages, their environments and system software interfaces — Technical Report on C++ Performance

1 Scope

The aim of this Technical 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 library facilities to yield efficient code.

As far as run-time and space performance are 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 Technical 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 Technical 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 information (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.

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

Additional research is continuing into techniques for producing efficient C++ libraries and programs. Please see the WG21 web site at www.open-std.org/jtc1/sc22/wg21 for example code from this Technical Report and pointers to other sites with relevant information.

2 Normative References

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 14882:2003, *Programming Languages – C++*.

Mentions of "the Standard" or "IS" followed by a clause or paragraph number refer to the above International Standard for C++. Section numbers not preceded by "IS" refer to locations within this Technical Report.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1

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)

3.2

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

3.3

addressing range

portion of the total of memory addresses accessible through processor instructions

NOTE A processor has one or more addressing ranges. Program memory, data memory and I/O devices 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.

3.4

address interleave

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

3.5

cache

buffer of high-speed memory used to improve access times to medium-speed main memory or to low-speed storage devices

NOTE 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.

3.6
code bloat

generation of excessive amounts of code instructions, for instance, from unnecessary template instantiations

3.7
code size

portion of a program's memory image devoted to executable instructions

NOTE Sometimes immutable data also is placed with the code.

3.8
cross-cast

cast of an object from one base class subobject to another

NOTE This requires RTTI and the use of the `dynamic_cast<...>` operator.

3.9
data size

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

3.10
device

I/O Device

term used to mean either a discrete I/O chip or an I/O function block in a single chip processor system

NOTE The data bus bit width is significant in the access method used for the I/O device.

3.11
device bus

I/O device bus

data bus of a device

NOTE 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.

3.12

device register

I/O device register

single logical register in a device

NOTE A device may contain multiple registers located at different addresses.

3.13

device register buffer

multiple contiguous registers in a device

3.14

device register endianness

endianness for a logical register in a device

NOTE The device register endianness may be different from the endianness used by the compiler and processor.

3.15

down-cast

cast of an object from a base class subobject, to a more derived class subobject

NOTE Depending on the complexity of the object's type, this may require **RTTI** and the use of the `dynamic_cast<...>` operator.

3.16

Electrically Erasable Programmable Read-Only Memory

EEPROM

similar to **flash memory** (sometimes called flash EEPROM), the principal difference is that EEPROM requires data to be erased and written one byte at a time whereas flash memory requires data to be erased in blocks and written one byte at a time

NOTE EEPROM retains its contents even when the power is turned off, but can be erased by exposing it to an electrical charge.

3.17
endianness

describes the layout in memory of the 0 and 1 bits which together represent a value

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

3.18
embedded system

program which functions as part of a device

NOTE 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).

3.19
flash memory

non-volatile memory device type which can be read like **ROM**

NOTE 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..

3.20
heap size

portion of a program's memory image devoted to data with dynamic storage duration, associated with objects created with `operator new`

3.21
interleave

see **address interleave**

3.22
Input/Output
I/O

term used in this TR for reading from and writing to **device register**s (see §8)

3.23
I/O bus

special processor addressing range used for input and output operations on hardware registers in a **device**

3.24
I/O device

synonym for **device**

3.25
locality of reference

heuristic that most programs tend to make most memory and disk accesses to locations near those accessed in the recent past

NOTE Keeping items accessed together in locations near each other increases **cache** hits and decreases **page faults**.

3.26
logical register

device register treated as a single entity

NOTE 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.

3.27
memory bus

processor **addressing range** used when addressing data memory and/or program memory

NOTE Some processor architectures have separate data and program memory buses.

3.28
memory device

chip or function block intended for holding program code and/or data

3.29
memory mapped I/O

I/O devices connected to the processor **addressing range** which are also used by data memory

3.30

Mean-Time Between Failures

MTBF

statistically determined average time a device is expected to operate correctly without failing, used as a measure of a hardware component's reliability

NOTE The calculation takes into account the MTBF of all devices in a system. The more devices in a system, the lower the system MTBF.

3.31

non-volatile memory

memory device that retains the data it stores, even when electric power is removed

3.32

overlays

technique for handling programs that are larger than available memory, older than **virtual memory addressing**

NOTE 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.

3.33

page

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

3.34

page fault

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

3.35

Plain Old Data

POD

data type which is compatible with the equivalent data type in C in layout, initialization, and its ability to be copied with memcpy (§IS-1.8¶5)

3.36

Programmable Read Only Memory

PROM

equivalent to **ROM** in the context of this Technical Report

3.37

Random Access Memory

RAM

memory device type for holding data or code

NOTE 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.

3.38

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**)

NOTE **Hard real-time** means that every operation must meet specified timing constraints.

3.39

Read Only Memory

ROM

memory device type, normally used for holding program code, but may contain data of static storage duration as well

NOTE Content in ROM can not be modified by the processor.

3.40

ROMable

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

3.41

ROMability

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

3.42

Run-Time Type Information

RTTI

information generated by the compiler which makes it possible to determine at run-time if an object is of a specified type

3.43

stack size

portion of a program's memory image devoted to data with automatic storage duration, also with certain bookkeeping information to manage the code's flow of control when calling and returning from functions

NOTE Sometimes the data structures for exception handling are also stored on the stack (§5.4.1.1).

3.44

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

NOTE See also **working set** and **virtual memory addressing**.

3.45

System-on-Chip

SoC

embedded system where most of the functionality of the system is implemented on a single chip, including the processor(s), **RAM** and **ROM**

3.46

text size

common alternative name for **code size**

3.47

User Defined Conversion

UDC

refers to the use, implicit or explicit, of a class member conversion operator

3.48

up-cast

cast of an object to one of its base class subobjects

NOTE This does not require RTTI and can use the `static_cast<...>` operator.

3.49

Virtual Base Class

VBC

base class which exists as a single subobject in the inheritance graph, even though inherited through multiple paths (§IS-10.1¶4)

NOTE In order to share a single instance of a base class, all derived classes must use the keyword virtual in their base class specifier referring to that base.

3.50

virtual memory addressing

technique for enabling a program to address more memory space than is physically available

NOTE 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.

3.51

working set

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

3.52

Whole Program Analysis

WPA

term used to refer to the process of examining the fully linked and resolved program for optimization possibilities

NOTE Traditional analysis is performed on a single translation unit (source file) at a time.

4 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.

4.1 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.

[1] Typical systems during the year 2004.

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

- **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.

- **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 separate 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

4.2 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).

5 Language Features: Overheads and 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

5.1 Namespaces

Namespaces do not add any significant 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, 2002";

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, 2004";

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.

With all names, longer names take up more space in the program's symbol table and may add a negligible amount of time to dynamic linking. However, there are tools which will strip the symbol table from the program image and reduce this impact.

5.2 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 correct[3]. 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 (§5.3.8).

5.3 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. See "Inside the C++ Object Model" [BIBREF-17] for further information.

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 Annex 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.

[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.

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.

5.3.1 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 `struct`s). 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`.

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 class. Whole 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.

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

5.3.2 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 Annex D:

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

The compiler/machine combinations #1 and #2 match 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.

5.3.3 Virtual Functions

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

Table 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-inheritance 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 1 and Table 2, we see that implementations #1, #2, #3, and #5 confirm the obvious assumption that virtual calls (and indirect calls) are more expensive than non-virtual calls (and direct calls). Interestingly, the 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.

5.3.3.1 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.

5.3.4 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 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 explicit inlining is significant (factors of 2.7, 2.7, and 17).

5.3.5 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 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 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.

5.3.6 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 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.

5.3.7 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 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.

5.3.8 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 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.

5.4 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 Annex 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.
- ***throw-expression*s** Data and code associated with throwing an exception.

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

5.4.1 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.

[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-15.3¶13).

- ***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`*** If an exception is thrown during construction of an object with dynamic storage duration (§IS-3.7.3), after calling the destructors for the partially constructed object the corresponding `operator delete` must be called to deallocate memory.

  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, which uses compiler-generated static tables.

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

5.4.1.1 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 5.4.1 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.

5.4.1.1.1 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.

5.4.1.1.2 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 a function has an *exception-specification*, register it for checking
- As local and temporary objects are created
  - ♦ Register each one with the current exception context as it is 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 `terminate_handler`[6]
- 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.

[6] When the `terminate_handler` is called because no matching exception handler was found, it is implementation-defined whether the stack is unwound and local objects are destroyed (§IS-15.5.1).

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.

5.4.1.2 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 5.4.1 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.

5.4.1.2.1 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.

5.4.1.2.2 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[7]
- On exit from each *try-block*
    - ♦ No overhead
- 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 `terminate_handler`[8]
- 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.

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

[8] When the `terminate_handler` is called because no matching exception handler was found, it is implementation-defined whether the stack is unwound and local objects are destroyed (§IS-15.5.1).

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.

5.4.2 Predictability of Exception Handling Overhead

5.4.2.1 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.

5.4.2.2 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.

5.5 Templates

5.5.1 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[9]. 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`. (See Annex D.2 for sample code.) 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.

If the compiler supports partial specialization and member function templates, the library implementer can provide partial specializations 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.

5.5.2 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.

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

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[10]. Most of the functions were declared virtual, and so had to be overridden by derived classes[11]. 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.

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

[11] 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 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 contents[12].
- 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.

5.6 Programmer Directed Optimizations

5.6.1 General Considerations

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

[12] 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 <`.

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 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[13].

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 Nth 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.

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

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.

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 (Annex E:) provide more extensive guidance (e.g. [BIBREF-17]), but the following sections provide some suggestions for writing more efficient code.

5.6.2 Object Construction

The construction of objects, though sometimes invisible, can be more expensive than expected. Therefore some considerations about implementation can improve application performance.

- 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.
- 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[14].

[14] 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[15] `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.

- 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;
```

[15] 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.

```
};

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 A, B, X, C, D, Y, 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.

5.6.3 Temporary Objects

When the compiler creates a temporary object of a user-defined type which has a constructor, the same initialization takes place as if it were a declared variable. But with careful programming the construction of temporary objects can sometimes be avoided.

- 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.

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

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

- 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 `iterators`, 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++
{
    // ...
}
```

- 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[16]. If the only supported parameters were std::strings, then characters and pointers to character arrays would have to be converted to full std::string objects before the operator was applied.

- 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();
}

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

[16] 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.

5.6.4 Function Inlining

- 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 (§5.3.3). 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.

- 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.

[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.

5.6.5 Object-Oriented Programming

- 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” (**P**ointer to **IMPL**ementation[18]) 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(); }
```

5.6.6 Templates

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

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

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

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:

```
// 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.

5.6.7 Standard Library

- 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.

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

- 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);
```

5.6.8 Additional Suggestions

- 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++.
- 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.
- 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[20].

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

- 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 the original code when translated by a good compiler.

- 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++.

5.6.9 Compilation Suggestions

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[21] 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[22] 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.
- 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 `#include`s 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 `#include`d, it has the effect of being `#include`d 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 `#include`d again, and the macro `THIS_HEADER_H` still remains defined, then the compiler can avoid accessing the header contents.

[21] For instance the `strip` utility, which is part of the Software Development Utilities option in the IEEE Posix/Open Group Single Unix Specification/ISO/IEC 9945:2002 standards.

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

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.

6 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.

6.1 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 §6.2.

The efficiency problems come in several forms:

6.1.1 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 §6.2.2.

6.1.2 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 preprocessing, 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 §6.2.3 and §6.2.5.

6.1.3 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 §6.2.6.

6.1.4 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 §6.2.4.

6.2 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.

6.2.1 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" (§6.2.5).

- 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.

6.2.2 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.

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.

6.2.3 Preprocessing 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 preprocessing 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.

6.2.4 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 §5.6).

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.

6.2.5 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 `bool` 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 `callbacks` are ever registered, is this necessary); etc.

In order for the linker to decide which functionality is used by the application, it must follow 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 §6.2.7 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 technique is not applicable to all situations and should be used carefully where appropriate.

6.2.6 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.

6.2.7 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: in the 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).

7 Using C++ in Embedded Systems

7.1 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.

7.1.1 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.

7.1.1.1 User-Defined Objects

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

- An aggregate (§IS-8.5.1) 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-3.5¶2) 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 sub-string 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;
```

7.1.1.2 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[23] ROMable.

- Type identification tables:

  When a table is generated to identify RTTI types, the table is usually[24] ROMable.

- Exception tables:

  When exception handling is implemented using static tables, the tables are usually[25] 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:

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

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

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

```
// 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.

- 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;
}
```

7.1.2 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.

7.2 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.

7.2.1 C++ Features for which Timing Analysis is Straightforward

7.2.1.1 Templates

As pointed out in detail in §5.5, 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.

7.2.1.2 Inheritance

7.2.1.2.1 Single Inheritance

Converting a pointer to a derived class to a pointer to base class[26] will not introduce any run-time overhead in most implementations (§5.3). 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.

7.2.1.2.2 Multiple Inheritance

Converting a pointer to a derived class to a pointer to base class might introduce run-time overhead (§5.3.5). This overhead is a fixed number of machine instructions (typically one).

7.2.1.2.3 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 (§5.3.6). This overhead is typically a fixed number of machine instructions for each access to a data member in the virtual base class.

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

7.2.1.3 Virtual functions

If the static type of an object can be determined at compile-time, calling a virtual function may be no more expensive than calling a non-virtual member function. If the type must be dynamically determined at run-time, the overhead will typically be a fixed number of machine instructions (§5.3.3) for each call.

7.2.2 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.

7.2.2.1 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 (§5.3.8).

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

7.2.2.2 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.

7.2.2.3 Exceptions

Enabling exceptions for compilation may introduce overhead on each function call (§5.4). 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 §5.4. 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.

7.2.3 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.

8 Hardware Addressing Interface

Embedded applications often must interact with specialized I/O devices, such as real-time sensors, motors, and LCD displays. At the lowest level, these devices are accessed and controlled through a set of special hardware registers (I/O registers) that device driver software can read and/or write.

Although different embedded systems typically have their own unique collections of hardware devices, it is not unusual for otherwise very different systems to have virtually identical interfaces to similar devices.

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/O hardware registers from the C source level. However, these extensions have not been consistent across dialects. As a growing number of C++ compiler vendors are now entering the same market, the same I/O driver portability problems become apparent for C++.

As a simple portability goal the driver source code for a given item of 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 hardware driver source code portability. The C++ interface provides definitions with a broader functionality than the C interface. It not only provides mechanisms for writing portable hardware device drivers, but also general methods to access the hardware of a given system. 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.

A header exists for the purpose of making certain names visible in the translation unit in which it is included. It may not exist as an actual file, if the compiler uses some other mechanism to make names visible. When this document mentions the "`<iohw.h>` interface" or the "`<hardware>` interface" it is referring to the collection of types and declarations made visible by the corresponding header.

This report provides:

- A general introduction and overview to the *iohw* interfaces (§8.1)
- A presentation of the common-subset interface (§8.2)
- A description of the C++ `<hardware>` interface (§8.3)
- Usage guidelines for the `<hardware>` interface (§Annex A:)
- General implementation guidelines for both interfaces (§Annex B:)
- Detailed implementation discussion for the `<hardware>` interface (§B.8)
- A discussion about techniques for implementing the common-subset interface on top of an implementation in C++ (§Annex C:)

8.1 Introduction to Hardware Addressing

The purpose of the *iohw* access functions described in this chapter is to promote portability of *iohw* driver source code and general hardware accessibility across different execution environments.

8.1.1 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 hardware 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 driver source code portability across different 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 hardware 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.)].

8.1.2 Terminology

The following is an overview of the concepts related to basic I/O hardware addressing and short definitions of the terms used in this Technical Report:

- **IO** and **I/O** are short notations for Input-Output. In the context of this chapter, these terms have no relation to C++ iostreams.
- An **I/O device** or **hardware device** is a hardware unit which uses registers to create a data interface between a processor and the external world. As the `<hardware>` interface defines a broader interface and encompasses all hardware access from processor registers to memory locations, this report uses the more general term **hardware device** and **hardware register**, though the `<iohw.h>` interface definition from WDTR 18037 and reprinted in §8.2 still uses the terms **I/O device**, **I/O register,** etc.
- A **hardware register** is the basic data unit in a *hardware device.*
- A **hardware device driver** is software which operates on *hardware registers* in a *hardware device.*

- The **logical hardware register** is the register unit as it is seen from the *hardware device*. The language data type used for holding the *hardware register* data must have a bit width equal to, or larger than, the bit width of the *logical hardware register*. The bit width of the *logical hardware register* may be larger than the bit width of the *hardware device* data bus or the processor data bus.

- **Hardware register access** is the process of transferring data between a *hardware register* and one of the compiler's native data storage objects. In a program this process is defined via a *hardware register designator specification* for the given *hardware register* or *hardware register buffer*.

- A **hardware register designator specification** specifies *hardware access properties* related to the *hardware register* itself (for instance the *hardware register bit width* and *hardware register endianness*) and properties related to the *hardware register access method* (for instance processor address space and address location).

- The `<hardware>` interface separates the *hardware access properties* into register-specific properties and (hardware) platform-specific properties. So, for the `<hardware>` interface, there is no single *hardware register designator* for a specific hardware register, but a combination of two. But there is still a single identifier that can be used in portable driver code (portable across different implementations of this interface).

- A **hardware register designator** encapsulates a *hardware register designator specification* -- the sum of all of a register's properties plus the properties of its access method – and uniquely identifies a single *hardware register* or *hardware register buffer*. The main purpose of the *hardware register designator* is to hide this information from the *hardware device driver* code, in order to make the *hardware device driver* code independent of any particular processor (or compiler).

- Multiple *hardware registers* of equal size may form a **hardware register buffer**. All registers in the *hardware register buffer* are addressed using the same *hardware register designator*. A *hardware register buffer element* is referenced with an *index* in the same manner as a C array.

- Multiple *hardware registers* may form a **hardware group**.

- A *hardware device* may contain multiple *hardware registers*. These registers can be combined into a **hardware group** which is portable as a specification for a single hardware unit (for instance an I/O chip, an FPGA cell, a plug-in board etc).

- Common hardware access properties for the hardware registers in a hardware register group are defined by the **hardware group designator**.

- Typical **hardware access properties** which are defined and encapsulated via the *hardware register designator* are the following:

  - The **access methods** used for hardware register access. *Access methods* refer to the various ways that *hardware registers* can be addressed and *hardware devices* can be connected in a given hardware platform. Typical methods are *direct addressing*, *indexed addressing*, and addressing via *hardware access drivers*. Different methods have different *hardware access properties*. Common for all access methods is that all access properties are encapsulated by the hardware register designator.

  - **Direct addressing** accesses a *hardware register* via a single constant or variable holding the static address of the register.

  - **Indexed addressing** accesses a *hardware register* by adding a constant offset to a base address initialized at runtime. (This access method is not to be confused with the `ioindex_t` used for accessing *hardware register buffers*, where the offset is not constant.)

  - A (user-supplied) **hardware access driver** may be used to encapsulate complex access mechanisms and to create virtual access spaces. Access via a user-supplied access function is common in hosted environments and when external I/O devices are connected to single-chip processors.

  - If all the *access properties* defined by the *hardware register designator specification* can be initialized at compile-time then its designator is called a **static designator**.

  - If some *access properties* defined by the *hardware register designator specification* are initialized at compile-time and others require initialization at run-time, then its designator is called a **dynamic designator**.

  - *Hardware registers* within the same *hardware group* shall share the same platform-related characteristics. Only the hardware register characteristics and address information will vary between the *hardware register designator specifications*.

  - **Direct designators** are fully initialized either at compile-time or by an `iogroup_acquire` operation using the `<iohw.h>` interface. In the `<hardware>` interface, *direct designators* are initialized by constructors that have an empty parameter list for *static designators*.

  - **Indirect designators** are fully initialized by an `iogroup_map` operation using the `<iohw.h>` interface. The `<hardware>` interface provides different means to bind *indirect designators*, e.g. template instantiation or function parameter binding.

- The hardware driver will determine whether a designator is a *direct designator* or an *indirect designator* only for the purpose of mapping (binding) a *hardware group designator*.

- If the bit width of the *logical hardware register* is larger than the bit width of the *hardware device* data bus, then (seen from the processor system) the *logical hardware register* will consist of two or more **partial hardware registers**. In such cases the **hardware register endianness** will be specified by the *designator specification*. The *hardware register endianness* is not related to any endianness used by the processor system or compiler.

- If the bit width of the *logical hardware register* is larger than the bit width of the processor data bus or the bit width of the *hardware device* data bus, then a single **logical hardware register access operation** will consist of multiple **partial hardware register access operations**. Such properties may be encapsulated by a single *hardware register designator* for the *logical hardware register*.

These concepts and terms are described in greater detail in the following sections.

8.1.3 Overview and Principles

The *iohw* access functions create a simple and platform independent interface between 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 and device bus size and endianness – from characteristics which are related to a specific execution environment, such as the hardware register address, processor bus type and endianness, address interleave, compiler access method, etc. Use of this separation principle enables driver source code itself to be portable to all platforms where the hardware device can be connected.

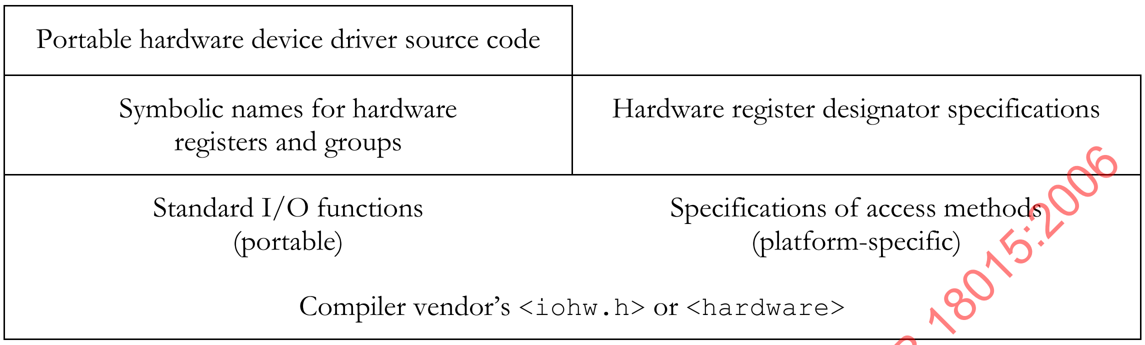
In the driver source code, a hardware register must always be referred to using a symbolic name, the hardware register designator. The symbolic name must refer to a complete hardware register designator specification of the access method used with the given register. A standardized *iohw* syntax approach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

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

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

8.1.4 The Abstract Model

The standardization of basic I/O hardware addressing is based on a three layer abstract model:


Portable hardware device driver source code
Symbolic names for hardware registers and groups
Hardware register designator specifications
Standard I/O functions (portable)
Specifications of access methods (platform-specific)
Compiler vendor's <iohw.h> or <hardware>

The top layer contains the hardware device driver code supplied by the hardware vendor or written by a driver developer. The source code in this layer is intended to be fully portable to any platform where the hardware can be connected. This code must only access hardware registers via the standardized I/O functionality described in this section. Each hardware register must be identified using a symbolic name, the hardware register designator, and referred to only by that name. These names are supplied by the author of the driver code, with the expectation that the integrator of hardware and platform will bind access properties to the names.

The middle layer associates symbolic names with complete *hardware register designator specifications* for the *hardware registers* in the given platform. The *hardware register designator* definitions in this layer are created last, and are the only part which must be updated when the *hardware driver source* code is ported to a different platform.

The bottom layer is the implementation of the `<iohw.h>` and `<hardware>` headers. They provide interfaces for the functionality defined in this section and specify the various different access methods supported by the given processor and platform architecture. This layer is typically implemented by the compiler vendor. The features provided by this layer, and used by the middle layer, may depend on intrinsic compiler capabilities.

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

§8.3 proposes a generic C++ syntax for hardware register designator specifications. 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.

8.1.4.1 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 "`platform_defs.h`" in this example):

1. Device driver module

   - The hardware driver source code
   - Portable across compilers and platforms
   - Includes `<iohw.h>` or `<hardware>` and "`platform_defs.h`"
   - Implemented by the author of the device driver

2. Hardware register designator specifications in "`platform_defs.h`"

   - Defines symbolic names for hardware register designators and their corresponding access methods
   - Specific to the execution environment
   - The header name and symbolic names are created by the author of the device driver
   - Other parts are implemented and maintained by the integrator

3. Interface header `<iohw.h>` or `<hardware>`

   - Defines hardware access functionality and access methods
   - Specific to a given compiler
   - Implemented by the compiler vendor

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

```
#include <iohw.h>
#include "platform_defs.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++:

For demonstration purposes, the hardware register designator specifications that are hidden in "`platform_defs.h`" in the above example are shown here in the unnamed namespace. For modular production code, these specifications will typically be in a separate header file. This example demonstrates various features specific to the `<hardware>` interface.

```
#include <hardware>

namespace
```

```
{
```

Middle layer (hardware register designator specifications):

```
using namespace std::hardware;
```

User-defined class used by the driver:

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

Common platform designator used by all registers in this driver:

`platform_traits` is an implementation-provided class with default definitions, so only the specifics must be provided (here only the base address).

```
struct Platform : platform_traits
{
    typedef static_address<0x34> address_holder;
};
```

Three register designators are defined here:

`register_traits` is also an implementation-provided class with default definitions.

All register designator specifications here use a static address.

```
struct PortA1_T : register_traits
{
    typedef static_address<0x1a> address_holder;
};

struct PortA2_T : register_traits
{
    typedef static_address<0x20> address_holder;
};
```

This designator specification additionally defines the `value_type` (the logical data type), as the default in this case is `uint8_t`:

```
struct PortA3_T : register_traits
{
    typedef UCharBuf value_type;
    typedef static_address<0x20> address_holder;
};

} // unnamed namespace

int main()
{
```

Writing to a single hardware register defined by the designators `PortA1_T` and `Platform`:

```
    register_access<PortA1_T, Platform> p1;
    p1 = 0x08;
```

Copying a register buffer specified by `PortA2_T`:

```
    unsigned char mybuf[10];
    register_buffer<PortA2_T, Platform> p2;
    for (int i = 0; i != 10; ++i)
    {
       mybuf[i] = p2[i];
    }
```

Essentially the same operation, but as a block read:

```
    register_access<PortA3_T, Platform> p3;
    UCharBuf myBlock;
    myBlock = p3;
}
```

The device driver 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.

8.1.5 Information Required by the Interface User

In order to enable a driver library user to define the hardware register designator specifications 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 hardware register designator names and hardware group designator names used by the library (in the diagram in §8.1.4, these things comprise the left half of the middle layer).
- Device and register type information for all designators (in §8.1.4 these constitute the hardware-specific traits needed in the definitions for the right half of the middle layer):
    - ♦ Logical bit width of the logical device register.
    - ♦ The designator type – single register, a register buffer or a register group.
    - ♦ Bit width of the device data bus.
    - ♦ Endianness of registers in the device (if any register has a logical width larger than the device's data bus).
    - ♦ Relative address offset of registers in the device (if the device contains more than one register).
    - ♦ Whether the driver assumes the use of indirect designators.

8.1.6 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", "read-write," or "read-modify-write"; often `READ` and `WRITE` operations are allowed only 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.

8.1.7 Hardware Register Designators

Within a program a machine's hardware registers are specified by *hardware register designators*. A hardware register designator according to the `<iohw.h>` interface may be an identifier or some implementation-specific construct. In the `<hardware>` interface, a hardware register designator can either be the name of a type or the name of an object, depending on the context. It may also be a class template. In any case, these are C++ names that follow the normal rules for name lookup (i.e. they shall not be preprocessor macros). A C implementation must support hardware register designators in the form of identifiers; other forms of hardware register designators may be supported but might not be portable to all implementations. A C++ implementation must support hardware register designators according to the specification in §8.3.

Any unique, non-reserved identifier can be declared as a designator for a hardware register. The definition of the identifier includes the size and access method of the hardware register. The means, however, by which an identifier is defined as a hardware register designator is entirely implemented-defined for the C interface and must follow the specifications in §8.3 for the C++ interface.

By choosing convenient identifiers as designators for registers, a programmer can create device driver code with the expectation that the identifiers can be defined to refer to the actual hardware registers on a machine supporting the same interface. So long as the only important differences from one platform to another are the access methods for the registers, device driver code can be ported to a new platform simply by updating the designator definitions (the "middle layer") for the new platform.

Additional issues and recommendations concerning hardware register designators are discussed in Annexes A, B, and C of this Technical Report.

8.1.8 Accesses to Individual Hardware Registers

The header `<iohw.h>` declares a number of functions and/or macros for accessing a hardware register given a hardware register designator. Each "function" defined by the `<iohw.h>` header may actually be implemented either as a function or as a function-like macro that expands into an expression having the effects described for the function. If a function is implemented as a function-like macro, there will ordinarily not be a corresponding actual function declared or defined within the library.

`<iohw.h>` defines functions for reading from and writing to a hardware register. These functions take a hardware register designator as argument.

The header `<hardware>` defines the same functionality by defining a class template `register_access`. This class template takes two arguments that together form a hardware register designator. These template arguments are traits classes that describe the hardware register access properties. One traits class defines the register specific properties while the other defines the platform specific properties.

The class template `register_access` defines an assignment operator for the write functionality and a conversion operator to the respective logical data type for the read functionality.

Example using the `<iohw.h>` interface:

> If `dev_status` and `dev_out` are hardware register designators defined in the file `"iodriv_hw.h"`, the following is possible valid code:

```
#include <iohw.h>
#include "iodriv_hw.h"   /* Platform-specific designator
                            definitions. */

// Wait until controller is no longer busy.
while (iord(dev_status) & STATUS_BUSY) /* do nothing */;

// Write value to controller.
iowr(dev_out, ch);
```

Example using the `<hardware>` interface:

```
#include <hardware>
// includes the definitions shown in the example of §8.1.4.1:
#include "driv_defs.h"

register_access<PortA1_T, Platform> devStatus;
register_access<PortA2_T, Platform> devOut;
const uint8_t statusBusy = 0x4;
uint8_t ch = ' ';

// Wait until controller is no longer busy:
while (devStatus & statusBusy)
    ; // do nothing

// Write some value to controller:
devOut = ch;
```

Besides simple read and write operations, three read-modify-write operations are supported, corresponding to the bit-wise logical operations AND, OR, and XOR. Again, these are defined as functions in the `<iohw.h>` interface and as overloaded operators in the `<hardware>` interface.

8.1.9 Hardware Register Buffers

Besides individual hardware registers, a hardware register designator may also designate a *hardware register buffer*, which is essentially an array of hardware registers. As with a C array, an index of unsigned integer type must be supplied to access a specific register in a hardware register buffer.

The `<iohw.h>` header declares all the same functions for buffers as for single registers, with a different name and an additional index parameter, for which the `ioindex_t` type is defined.

The `<hardware>` header defines for this purpose a special class template `register_buffer` that defines an operator[], which in turn returns a reference to a normal `register_access` instantiation, so all the operations defined for single registers can be used on the result of the index operator.

Example using `<iohw.h>`:

> If `ctrl_buffer` is defined in the file "`ctrl_regs.h`" as a hardware register designator for a hardware register buffer, the following is possible valid code:

```
#include <iohw.h>
#include "ctrl_regs.h"      // Platform-specific designator
                            // definitions.

unsigned char buf[CTRL_BUFSIZE];

 // Copy buffer contents.
for (int i = 0; i < CTRL_BUFSIZE; i++)
    buf[i] = iordbuf(ctrl_buffer, i);
```

Essentially the same example using `<hardware>`:

The "middle layer" is not shown here; it is assumed to be the same as for the previous examples.

```
#include <hardware>

const size_t bufSize = 10;
int main()
{
unsigned char mybuf[bufSize];
register_buffer<PortA2_T, Platform> p2;
    for (size_t i = 0; i != bufSize; ++i)
    {
       mybuf[i] = p2[i];
    }
    return 0;
}
```

Two hardware register buffer indexes *index* and *index+1* refer to two adjacent hardware register locations in the hardware device. Note that this may be different from adjacent address locations in the underlying platform. See §B.2.2 for a more detailed discussion.

As in an ordinary array a larger index refers to a platform location at a higher address.

Unlike an ordinary array, the valid locations within a hardware register buffer might not be "dense"; any index might not correspond to an actual hardware register in the buffer. (A programmer should be able to determine the valid indices from documentation for the hardware device or the machine.) If a hardware register buffer accesses an "empty" location, the behavior is undefined.

8.1.10 Hardware Groups

A hardware group is an arbitrary collection of hardware register designators. Each hardware group is intended to encompass all the designators for a single hardware device. Certain operations are supported only for hardware groups; these operations apply to the members of a hardware group as a whole. Whether a hardware register designator can be a member of more than one group is implementation-defined.

Like hardware registers, a hardware group is specified by a hardware group designator. For the identification of this designator, the same rules apply as for the identification of normal hardware register group designators, as explained in 5.1.7, and are different for `<iohw.h>` and `<hardware>` as specified there.

8.1.11 Direct and Indirect Designators

Each hardware register designator is either direct or indirect. An indirect hardware register designator has a definition that does not fully specify the register or register buffer to which the designator refers. Before any accesses can be performed with it, an indirect designator must be mapped to refer to a specific register or register buffer. A direct hardware register designator, by contrast, has a definition that fully specifies the register or register buffer to which the designator refers. A direct designator always refers to the same register or register buffer and cannot be changed.

For the `<hardware>` interface, a direct designator specification consists of two parts, one defining the platform access properties and one defining the register access properties. An indirect hardware register designator specification only has the part defining the register access properties, and must be completed with the platform specific part by a mapping.

An indirect hardware register designator is mapped by associating it with a direct hardware register designator. Accesses to the indirect designator then occur as though with the direct designator to which the indirect designator is mapped. An indirect hardware register designator can be remapped any number of times; accesses through the designator always occur with respect to its latest mapping.

An `<iohw.h>` implementation is not required to support indirect designators. If an `<iohw.h>` implementation does support indirect designators, it may place arbitrary restrictions on the direct designators to which a specific indirect designator can be mapped. Typically, an indirect designator will be defined to be of a certain "kind," capable of mapping to some subclass of access methods. An indirect designator can be mapped to a direct designator only if the direct designator's access method is compatible with the indirect designator. Such issues are specific to an implementation.

The `<hardware>` interface defines several methods of mapping that are available in all `<hardware>` implementations, which therefore all support indirect designators.

8.1.12 Operations on Hardware Groups

8.1.12.1 Acquiring Access to a Hardware Register in a Group

For some platforms, it may be necessary to acquire a hardware register or hardware register buffer before it can be accessed. What constitutes "acquiring" a register is specific to an implementation, but acquisition performs all the initializations that are required before one can access that register.

The `<iohw.h>` header declares two functions, `iogroup_acquire` and `iogroup_release`, each taking a single direct hardware group designator as an argument and performing any initializing and releasing actions necessary for all designators in that group.

One purpose of `iogroup_acquire` is to give the I/O device driver writer control over when the hardware group designator is initialized, because certain conditions may have to be met before this can safely be done. For example, some hardware platform dependent I/O registers may need to be initialized before the given I/O device group can be safely acquired and initialized. In an implementation for a hosted environment, the initialization for a specific hardware register might call the operating system to map the physical hardware registers of the group into a block of addresses in the process's address space so that they can be accessed. In the same implementation, the releasing action would call the operating system to unmap the hardware registers, making them inaccessible to the process. Therefore hardware group designator initialization must not be something which happens automatically at program startup; it should be called explicitly.

A `<hardware>` interface implementation may handle any initializing of hardware group designators using normal C++ constructors and any releasing actions using normal destructors.

8.1.12.2 Mapping Indirect Designators

The `<iohw.h>` header declares a function `iogroup_map` taking an indirect and a direct hardware group designator as argument that binds all hardware register designators from the first group to the respective hardware register designator of the second.

The `<hardware>` interface defines several methods of mapping. In the `<hardware>` interface, a hardware register designator specification consists of two traits classes, one of them defining the register-specific access properties and one of them defining the platform-specific access properties. An indirect designator is a designator that has only the register-specific traits class. For it to become a complete, direct designator a platform-specific traits class must be added, which is called mapping.

One mapping method is simply the instantiation of a class template that puts together the register-specific traits class and the platform-specific traits class. If this method is used for static designators, it will not introduce any register-specific or platform- specific data members and any address computations for hardware register accesses can be completely resolved at compile-time.

Another method uses a simple dynamic address holder type with which an indirect designator can be augmented to make a full direct designator. This way, any dynamically acquired platform-specific data can be used to map an indirect hardware group.

Example using `<iohw.h>`:

If `"dev_hw.h"` defines two indirect I/O register designators, `dev_config` and `dev_data`, an indirect I/O group designator `dev_group` with both `dev_config` and `dev_data` as members, and two direct I/O group designators `dev1_group` and `dev2_group`, the following is possible valid code:

```
#include <iohw.h>
#include "dev_hw.h"   // Platform-specific designator
                      // definitions.

// Portable device driver function.
uint8_t get_dev_data(void)
{
   iowr(dev_config, 0x33);
   return iord(dev_data);
}

// Read data from device 1.
iogroup_map(dev_group, dev1_group);
uint8_t d1 = get_dev_data();

// Read data from device 2.
iogroup_map(dev_group, dev2_group);
uint8_t d2 = get_dev_data();
```

Example using `<hardware>`:

This example is equivalent to the `<iohw.h>` example, but for demonstration purposes, it shows the complete "middle layer" defined here in the unnamed namespace:

```
#include <hardware>

namespace
{
```

// Middle layer (hardware register designator specifications)

```
using namespace std::hardware;

struct PlatformA : platform_traits
{
    typedef static_address<0x50> address_holder;
};

struct PlatformB : platform_traits
{
    typedef static_address<0x90> address_holder;
};

struct DynPlatform : platform_traits
{
    typedef dynamic_address address_holder;
    enum { address_mode=hw_base::dynamic_address };
};

struct PortA1_T : register_traits
{
    typedef static_address<0x1a> address_holder;
};

struct PortA2_T : register_traits
{
    typedef static_address<0x20> address_holder;
};
```

// Portable device driver function using the template approach:

```
template <class PlatformSpec>
uint8_t getDevData(typename PlatformSpec::address_holder const &addr =
typename PlatformSpec::address_holder())
{
    register_access<PortA1_T, PlatformSpec> devConfig(addr);
    register_access<PortA2_T, PlatformSpec> devData(addr);

    devConfig = 0x33;
    return devData;
}
} // unnamed namespace

int main()
{
```

// static version

```
    // Read data from device 1:
    uint8_t d1 = getDevData<PlatformA>();

    // Read data from device 2:
    uint8_t d2 = getDevData<PlatformB>();
```

```
// dynamic version
        uint8_t d3 = getDevData<DynPlatform>(0x40);

        uint8_t d4 = getDevData<DynPlatform>(0x80);

        return 0;
    }
```

In this example, the mapping is done by simply instantiating register access in `getDevData()` using the template parameter of this function template. The first version shown uses a static approach that gives different static platform traits classes as template arguments. This approach will produce two different instantiations of `getDevData()`, but does all the address computations for accessing `devConfig` and `devData` at compile-time and produces in typical applications absolutely no object data for any platform or register object.

The second version uses a dynamic approach and therefore avoids the double instantiation of `getDevData`, but in turn produces a data object containing the given platform address for each of the local hardware register designators, `devConfig` and `devData`. Also, the actual address to access these registers is calculated at run-time.

The `<hardware>` interface deliberately offers both methods, as the actual trade-off judgment can only be done by the driver programmer.

8.2 The `<iohw.h>` Interface for C and C++

For the convenience of the reader, this section duplicates a portion of the Technical Report ISO/IEC WDTR 18037 "*Extensions for the programming language C to support embedded processors*" from JTC 1/SC 22/WG 14. If the description of hardware access interfaces in this report differs from that in ISO/IEC WDTR 18037, the description there takes precedence.

The header `<iohw.h>` declares a type and defines macros and/or declares functions for accessing implementation-specific I/O registers.

The type declared is

```
    ioindex_t
```

which is the unsigned integer type of an index into an I/O register buffer.

Any "function" declared in `<iohw.h>` as described below may alternatively be implemented as a function-like macro defined in `<iohw.h>`. (If a function in `<iohw.h>` is implemented as a function-like macro, there need not be an actual function declared or defined as described, despite the use of the word function.) Any invocation of such a function-like macro shall expand to code that evaluates each of its arguments exactly once, fully protected by parentheses where necessary, so it is generally safe to use arbitrary expressions as arguments.

8.2.1 I/O registers

An I/O register is a storage location that is addressable within some address space. An I/O register has a size and an access method, which is the method by which an implementation accesses the register at execution time. An I/O register is accessed (read or written) as an unsigned integer. An I/O register may need to be acquired before it can be accessed. (I/O registers are acquired with the `iogroup_acquire` function described in §8.2.3.1)

Accesses to an I/O register may have unspecified side effects that may be unknown to the implementation, and an I/O register may be modified in ways unknown to the implementation. Accesses to I/O registers performed by functions declared in `<iohw.h>` are therefore treated as side effects which respect sequence points[27].

An I/O register buffer is a collection of I/O registers indexed by an integer of type `ioindex_t` and otherwise sharing a common size and access method. The set of valid indices for the I/O registers in an I/O register buffer may be any subset of the values of type `ioindex_t`; the set of valid indices need not be contiguous and need not include zero.

An I/O register designator refers (except as stipulated below) to a specific individual I/O register or a specific I/O register buffer. Functions that access I/O registers take an I/O register designator argument to determine the register to access. An implementation shall support at least one of the following as a valid I/O register designator for any individual I/O register or I/O register buffer:

- any ordinary identifier that is not a reserved identifier, defined by some implementation-defined means; and/or

- any object-like macro name that is not a reserved identifier, defined in accordance with some implementation-defined convention.

An implementation may optionally support other, implementation-defined forms of I/O register designators.

Each I/O register designator is either direct or indirect. A direct I/O register designator refers to a specific I/O register or I/O register buffer as determined by the designator's definition. An indirect I/O register designator does not refer to a specific I/O register or I/O register buffer until the designator has been mapped to a direct I/O register designator. Once mapped, an indirect I/O register designator can subsequently be remapped (mapped again) to the same or a different direct I/O register designator. An indirect I/O register designator refers to the same I/O register or I/O register buffer as the direct designator to which it was last mapped. (I/O register designators are mapped with the `iogroup_map` function described in §8.2.3.2.)

[27] And therefore I/O register access must always be qualified as `volatile`.

An indirect I/O register designator is compatible with a direct I/O register designator if it is possible to map the indirect designator to the direct designator. An I/O register designator that refers to an individual I/O register is not compatible with an I/O register designator that refers to an I/O register buffer, and vice versa. Otherwise, whether a specific indirect I/O register designator is compatible with a specific direct I/O register designator is implementation-defined.

An implementation need not support a means for indirect I/O register designators to be defined.

An I/O register designator covers an I/O register if it refers to the I/O register or it refers to an I/O register buffer that includes the register.

8.2.2 I/O groups

An I/O group is a collection of I/O register designators. It is intended that each I/O group encompass all the designators for a single hardware controller or device.

The members of an I/O group shall be either all direct designators or all indirect designators. An I/O group is direct if its members are direct. An I/O group is indirect if its members are indirect.

An I/O group shall not have as members two or more I/O register designators that cover the same I/O register. Whether an I/O register designator can be a member of more than one I/O group at the same time is implementation-defined.

An I/O group designator specifies an I/O group. An implementation shall support at least one of the following as a valid I/O group designator for any supported I/O group:

- any ordinary identifier that is not a reserved identifier, defined by some implementation-defined means; and/or
- any object-like macro name that is not a reserved identifier, defined in accordance with some implementation-defined convention.

8.2.3 I/O group functions

8.2.3.1 The `iogroup_acquire` and `iogroup_release` functions

Synopsis

```
#include <iohw.h>
void iogroup_acquire( iogroup_designator );
void iogroup_release( iogroup_designator );
```

Description

The **`iogroup_acquire`** function acquires a collection of I/O registers; the **`iogroup_release`** function releases a collection of I/O registers. Releasing an I/O register undoes the act of acquiring the register. The functions acquire or release all the I/O registers covered by the I/O register designators that are members of the I/O group designated by **`iogroup_designator`**. If the I/O group is indirect, the behavior is undefined.

An I/O register is only said to be acquired between an invocation of **`iogroup_acquire`** that acquires the register and the next subsequent invocation of **`iogroup_release`**, if any, that releases the register. If `iogroup_release` releases an I/O register that is not at the time acquired, or if `iogroup_acquire` acquires an I/O register that is at the time already acquired, the behavior is undefined.

Acquiring or releasing an I/O register is treated as a side effect which respects sequence points.

If an implementation can access a particular I/O register without needing it to be first acquired, the act of acquiring and the act of releasing the register may have no real effect.

8.2.3.2 The `iogroup_map` function

Synopsis

```
#include <iohw.h>
void iogroup_map( iogroup_designator, iogroup_designator );
```

Description

The `iogroup_map` function maps the indirect I/O register designators in the I/O group designated by the first `iogroup_designator` to corresponding direct I/O register designators in the I/O group designated by the second `iogroup_designator`. The first I/O group shall be indirect, and the second I/O group shall be direct. The correspondence between members of the two I/O groups is implementation-defined and shall be one-to-one. If an indirect I/O register designator is mapped to a direct I/O register designator with which it is not compatible, the behavior is undefined.

8.2.4 I/O register access functions

If a register is accessed (read or written) when it is not acquired, the behavior is undefined. If an indirect I/O register designator is given as an argument to one of the functions below and the designator has not been mapped, the behavior is undefined.

8.2.4.1 The `iord` functions

Synopsis

```
#include <iohw.h>
unsigned int iord( ioreg_designator );
unsigned long iordl( ioreg_designator );
```

Description

The functions `iord` and `iordl` read the individual I/O register referred to by `ioreg_designator` and return the value read. The I/O register is read as an unsigned integer of its size; the read value is then converted to the result type, and this converted value is returned.

8.2.4.2 The `iordbuf` functions

Synopsis

```
#include <iohw.h>
unsigned int iordbuf( ioreg_designator, ioindex_t ix );
unsigned long iordbufl( ioreg_designator, ioindex_t ix );
```

Description

The functions `iordbuf` and `iordbufl` read one of the I/O registers in the I/O register buffer referred to by `ioreg_designator` and return the value read. The functions are equivalent to `iord` and `iordl`, respectively, except that the I/O register read is the one with index `ix` in the I/O register buffer referred to by `ioreg_designator`. If `ix` is not a valid index for the I/O register buffer, the behavior is undefined.

8.2.4.3 The `iowr` functions

Synopsis

```
#include <iohw.h>
void iowr( ioreg_designator, unsigned int a );
void iowrl( ioreg_designator, unsigned long a );
```

Description

The functions `iowr` and `iowrl` write the individual I/O register referred to by `ioreg_designator`. The unsigned integer `a` is converted to an unsigned integer of the size of the I/O register, and this converted value is written to the I/O register.

8.2.4.4 The `iowrbuf` functions

Synopsis

```
#include <iohw.h>
void iowrbuf( ioreg_designator, ioindex_t ix, unsigned int a );
void iowrbufl( ioreg_designator, ioindex_t ix, unsigned long a );
```

Description

The functions `iowrbuf` and `iowrbufl` write one of the I/O registers in the I/O register buffer referred to by `ioreg_designator`. The functions are equivalent to `iowr` and `iowrl`, respectively, except that the I/O register written is the one with index `ix` in the I/O register buffer referred to by `ioreg_designator`. If `ix` is not a valid index for the I/O register buffer, the behavior is undefined.

8.2.4.5 The `ioor`, `ioand`, and `ioxor` functions

Synopsis

```
#include <iohw.h>
void ioand( ioreg_designator, unsigned int a );
void ioor( ioreg_designator, unsigned int a );
void ioxor( ioreg_designator, unsigned int a );

void ioorl( ioreg_designator, unsigned long a );
void ioandl( ioreg_designator, unsigned long a );
void ioxorl( ioreg_designator, unsigned long a );
```

Description

The functions `ioand`, `ioandl`, `ioor`, `ioorl`, `ioxor`, and `ioxorl` modify the individual I/O register referred to by `ioreg_designator`. The function `ioand` has a behavior equivalent to

```
iowr( ioreg_designator, iord( ioreg_designator ) & a )
```

except that the `ioreg_designator` is not evaluated twice (assuming it is an expression).

Likewise, the function `ioor` has a behavior equivalent to

```
iowr( ioreg_designator, iord( ioreg_designator ) | a )
```

and the function `ioxor` has a behavior equivalent to

```
iowr( ioreg_designator, iord( ioreg_designator ) ^ a )
```

Corresponding equivalencies apply for `ioandl`, `ioorl`, and `ioxorl`, but with the unsigned long functions `iordl` and `iowrl` replacing `iord` and `iowr`.

8.2.4.6 The `ioorbuf`, `ioandbuf`, and `ioxorbuf` functions

Synopsis

```
#include <iohw.h>
void ioandbuf( ioreg_designator, ioindex_t ix, unsigned int a );
void ioorbuf( ioreg_designator, ioindex_t ix, unsigned int a );
void ioxorbuf( ioreg_designator, ioindex_t ix, unsigned int a );

void ioandbufl( ioreg_designator, ioindex_t ix, unsigned long a );
void ioorbufl( ioreg_designator, ioindex_t ix, unsigned long a );
void ioxorbufl( ioreg_designator, ioindex_t ix, unsigned long a );
```

Description

The functions ioandbuf, ioorbuf, ioxorbuf, ioorbufl, ioandbufl, and ioxorbufl modify one of the I/O registers in the I/O register buffer referred to by ioreg_designator. The functions are equivalent to ioand, ioandl, ioor, ioorl, ioxor, and ioxorl, respectively, except that the I/O register modified is the one with index ix in the I/O register buffer referred to by ioreg_designator. If ix is not a valid index for the I/O register buffer, the behavior is undefined.

8.3 The **<hardware>** Interface for C++

The programming model behind these definitions is described in §8.1.4. The header <hardware> defines an interface for two layers of that model, the top layer for the portable source code and parts of the middle layer for the device register definitions. This is notably different to the C interface <iohw.h> described in §8.2.

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 §8.3.3

        struct hw_base { ... };

        // required address holder types
        template <hw_base::address_type val>
        struct static_address;

        struct dynamic_address;
        // [others may be provided by an implementation]

        struct platform_traits;
        struct register_traits;

        template <class RegTraits, class PlatformTraits>
        class register_access;

        template <class RegTraits, class PlatformTraits>
        class register_buffer;
    } // namespace hardware
} // namespace std
```

8.3.1 The Class Template `register_access`

Synopsis

```
template <class RegTraits, class PlatformTraits>
class register_access
{
public:
    typedef typename RegTraits::value_type value_type;

    // constructors
    register_access
        (typename RegTraits::address_holder const &rAddr,
         typename PlatformTraits::address_holder const &pAddr);
    register_access
        (typename PlatformTraits::address_holder const &pAddr);
    register_access();


    // operator interface
    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);

};
```

Description

`class register_access<...>`

- Provides direct access to hardware registers. This defines the interface for the top layer as described in §8.1.4.

`class RegTraits`

- The argument to the first template parameter `RegTraits` must be a class or instantiation of a class template that is a derived class of `register_traits` and specify the register-specific access properties of the hardware register.

`class PlatformTraits`

- The argument to the second template parameter `PlatformTraits` must be a class or instantiation of a class template that is a derived class of `platform_traits` and specify the platform-specific access properties of the hardware register.

An implementation may add additional template parameters with default values.

```
typedef value_type
```

- Names the `value_type` of the `RegTraits`.

Constructors:

```
register_access
    (typename RegTraits::address_holder const &rAddr,
     typename PlatformTraits::address_holder const &pAddr);

register_access
    (typename PlatformTraits::address_holder const &pAddr);

register_access();
```

- The constructors take references to the respective address holder classes of the access specification traits template parameters. If an address holder is marked as `is_static` in its traits class, the respective constructor argument shall not be given.

```
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`.

```
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 type for all assignment operators is `void` to prevent assignment chaining that could inadvertently cause considerable harm with device registers.

Note: The class interface provides both member functions and overloaded operators to perform `READ` and `WRITE` accesses to the register. The redundancy is intentional, to accommodate different programming styles. One set of operations can trivially be implemented in terms of the other.

8.3.2 *The Class Template* `register_buffer`

Synopsis

```
template <class RegTraits, class PlatformTraits>
class register_buffer
{
public:
    typedef register_access<RegTraits, PlatformTraits> ref_type;
    typedef typename RegTraits::value_type value_type;

    // constructors
    register_buffer
        (typename RegTraits::address_holder const &rAddr,
         typename PlatformTraits::address_holder const &pAddr);
    register_buffer
        (typename PlatformTraits::address_holder const &pAddr);
    register_buffer();

    // operator interface
    ref_type operator[](size_t index) const;
    // function-style interface
    ref_type get_buffer_element(size_t index);

};
```

Description

`class register_buffer<...>`

- Provides direct access to hardware register buffers. This defines the interface for the top layer as described in §8.1.4.

`class RegTraits`

- The argument to the first template parameter `RegTraits` must be a class or instantiation of a class template that is a derived class of `register_traits` and specify the register-specific access properties of the hardware register.

`class PlatformTraits`

- The argument to the second template parameter `PlatformTraits` must be a class or instantiation of a class template that is a derived class of `platform_traits` and specify the platform-specific access properties of the hardware register.

An implementation may add additional template parameters with default values.

`typedef ref_type`

- Names the return type of the index operator which is equivalent to the corresponding `register_access` class. (It might be a nested class that can be used like the corresponding `register_access` class.)

`typedef value_type`

- Names the `value_type` of the `RegTraits`.

Constructors:

```
register_buffer
    (typename RegTraits::address_holder const &rAddr,
     typename PlatformTraits::address_holder const &pAddr);
register_buffer
    (typename PlatformTraits::address_holder const &pAddr);
register_buffer();
```

- The constructors take references to the respective address holder classes of the access specification traits template parameters. If an address holder is marked as `is_static` in its traits class, the respective constructor argument shall not be given.

```
ref_type operator [] (size_t index)
ref_type 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 or read, and the bitwise `OR`, `AND` and `XOR` can be applied to it.

  Note: The purpose of providing both `operator[]` and a member function `get_buffer_element` is to accommodate different programming styles. One can be implemented in terms of the other.

8.3.3 Header "`stdint.h`"

The header `<stdint.h>` is specified by C99 (IS 9899-1999), and is not part of the C++ Standard (ISO/IEC 14882:2003 (Second Edition)). Instead, some implementation specific mechanism introduces the fixed size integer types described by `<stdint.h>` of the C standard into namespace `std::hardware` as if the header `<stdint.h>` were included by `<hardware>`.

No names are introduced into global namespace.

8.3.4 The `struct hw_base`

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 };  // possibly more
           enum processor_bus { bus8, bus16, bus32, bus64 };

           // identifiers for bus types as type names
           enum data_bus {};
           enum io_bus {};
           // only identifiers should be present that are supported
           // by the underlying implementation -- diagnostic required

           enum address_kind { is_static, is_dynamic };

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

Description

`struct hw_base`

- Provides the names for the supported hardware characteristics. Only those names that are supported by the hardware shall be present. Additional names that define additional or different functionality may be defined by an implementation.

`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. An implementation may define additional byte orders.

`enum processor_bus`

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

`enum data_bus, io_bus`

- Defines a type name for each bus at which accessible devices can be connected. data_bus specifies a bus that addresses attached devices like normal memory cells (memory-mapped). io_bus specifies a bus that addresses attached devices by using special instructions (e.g. in/out or port instructions). An implementation may define additional type names for additional buses. Only names shall be defined in an implementation for which a respective bus actually exists in the underlying hardware architecture.

`enum address_kind`

- Defines the names is_static and is_dynamic to mark address holders in register and platform traits. An address holder that is marked as is_static holds an address that is known at compile time. An address holder that is marked as is_dynamic holds an address that might only be known at run-time.

`typedef address_type`

- Is a type specified by the implementation to hold a hardware address. If the underlying hardware platform supports it, this type shall be an integral type. If the underlying hardware platform supports more than one type of hardware address (as is usually the case where more than one bus exists), an implementation shall define additional types for those addresses with implementation defined names.

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

8.3.5 Common Address Holder Types

This Technical Report defines the names and properties for the two address holder types `static_address` and `dynamic_address`. An implementation may define additional address holder types.

8.3.5.1 The Class Template `static_address`

Synopsis

```
template <hw_base::address_type val>
struct static_address
{
    enum { value_=val };
    hw_base::address_type value() const;
};
```

Description

`static_address`

- Holds an address known at compile time.

`hw_base::address_type val`

- Provides the statically known address. If in an implementation hw_base::address_type is not a basic type, the implementation can define different template parameters.

`value_`

- Provides the address value through a name. If in an implementation hw_base::address_type is not an integral type, the implementation must provide the address by the name value_ using a different mechanism than enum.

`hw_base::address_type value()`

- Provides the address value through a function.

8.3.5.2 The Class `dynamic_address`

Synopsis

```
struct dynamic_address
{
    dynamic_address(hw_base::address_type addr);
    hw_base::address_type value() const;
    hw_base::address_type value_;
};
```

Description

`struct dynamic_address`

- Holds an address that can be set at run-time.

`dynamic_address(hw_base::address_type addr)`

- A (converting) constructor to set the address.

`hw_base::address_type value()`

- Provides the address value through a function.

`value_`

- Provides the address value through a name.

8.3.6 Basic Hardware Register Designator Traits Classes

An implementation shall provide at least one traits class for the platform-specific access properties and one traits class for register-specific access properties for hardware registers (see §8.1.3). These traits classes specify the most common access properties for a given implementation. If there is no most common case, the implementation shall provide respective traits classes for all common cases.

The traits classes must be provided in a way that they can easily be used as base classes where all names that are not overridden in the derived class are inherited from the base class.

8.3.6.1 Traits Class `platform_traits`

Synopsis

```
struct platform_traits
{
    typedef implementation-defined address_holder;
    typedef implementation-defined processor_bus;
    enum
    {
        address_mode,
        processor_endianness,
        processor_bus_width
    };
};
```

Description

`struct platform_traits`

- Provides names that specify the most common platform-specific access properties for an implementation.
- Names that are not meaningful in an implementation shall be omitted. An implementation can also define additional names that specify additional access properties that are meaningful for that implementation.

`typedef address_holder`

- Specifies the type for the address that is part of the platform-specific access properties (e.g. the base address of a hardware group, see §8.1.10). [Note: This can be the static_address<0> where there is no platform-specific address and will probably be dynamic_address for group base addresses to be initialized at run-time).]

`typedef processor_bus`

- Specifies the bus where hardware registers are attached. The choice of the bus for an implementation can be arbitrary, but shall be one of the bus type names in hw_base. If an implementation provides only one bus where hardware devices can be attached, this name can be omitted.

`address_mode`

- One of the values from hw_base::address_kind. Specifies whether the address held by address_holder is known at compile-time or only known at run-time.

`processor_endianness`

- One of the values from hw_base::byte_order. Specifies the endianness of the processor or processor_bus.

`processor_bus_width`

- One of the values from hw_base::processor_bus. Specifies the width in bytes of the processor_bus.

8.3.6.2 Traits Class `register_traits`

Synopsis

```
struct register_traits
{
    typedef implementation-defined value_type;
    typedef implementation-defined address_holder;
    enum
    {
        address_mode,
        access_mode,
        endianness,
        device_bus_width
    };
};
```

Description

`struct register_traits`

- Provides names that specify the most common register-specific access properties for an implementation.
- Names that are not meaningful in an implementation shall be omitted. An implementation can also define additional names that specify additional access properties that are meaningful for that implementation.

`typedef value_type`

- Specifies the type of the hardware register. This shall be an Assignable and CopyConstructible type.

`typedef address_holder`

- Specifies the type for the address of the hardware register. [Note: If the address (offset) of the register is known at compile-time (as is usually the case), this can be omitted, as users will override this with static_address<register_addr> in the register-specific derivation. If the register address is to be specified at run-time, which sometimes might be useful even if it is known at compile-time, this will probably be dynamic_address.]

`address_mode`

- One of the values from hw_base::address_kind. Specifies whether the address held by address_holder is known at compile-time or only known at run-time.

`access_mode`

- One of the values from hw_base::access_mode. Specifies what access operations (read/write) are allowed on the hardware register.

`endianness`

- One of the values from hw_base::byte_order. Specifies the endianness of the device bus where the hardware register is attached.

`device_bus_width`

- One of the values from hw_base::device_bus. Specifies the width in bytes of the device bus where the hardware register is attached.

Annex A: Guidelines on Using the `<hardware>` Interface

A.1 Usage Introduction

The design of the C++ `<hardware>` interface follows two lines of separation between:

- The definition of hardware register designator specifications and the device driver code.
- What is known at compile-time and what is known only at run-time.

Unfortunately, these two lines of separation are neither orthogonal nor identical; for example, a `dynamic_address` is only known at run-time, but is part of the hardware register designator specifications.

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 type names or `enum` constants in traits classes) and what is only known at run-time (which becomes function (especially constructor) arguments or operator operands to the interface of `register_access` and `register_buffer`).

A.2 Using Hardware Register Designator Specifications

Hardware register designator specifications specify how a given device register can be accessed. These specifications are separated into two parts: the register-specific part and the (hardware) platform-specific part. Both parts are defined as traits classes in the middle layer of the abstract model (§8.1.4). These traits classes are then used as template parameters to the class templates `register_access` and `register_buffer`.

The actual details of these traits classes are mainly implementation defined, as these specify access details that can vary widely over different platforms. An implementation provides at least two generic traits classes: `platform_traits` and `register_traits`. These traits classes specify which definitions and names are required for a platform and give meaningful default values for them. So, these implementation provided traits classes can be used as a guide to what information must be provided, and they also serve as base classes such that only those names which differ from the default must be (re-)defined in the user's own traits classes.

Though the details of these traits classes are implementation defined by nature, there are some aspects that these traits classes have in common:

- `platform_traits` and `register_traits` contain the `typedef address_holder` that actually holds the hardware address. The `address_holder` for the register often holds an offset address that specifies the offset of this specific register inside a hardware device. To form the final address, this offset is added to the base address of the hardware device that is specified in the respective `platform_traits` class. But for simpler cases, the `address_holder` for the register simply holds the final address of the register and the respective address holder of the `platform_traits` class holds a null address.

- `address_holder` can hold either an address that is statically known at compile- time or an address that is initialized at run-time. What kind of address the `address_holder` actually holds is specified by the value of the `enum` constant `address_mode` and can either be `hw_base::is_static` or `hw_base::is_dynamic`.

- `register_traits` contains a `typedef value_type` that specifies the type of the data held in that register.

- `register_traits` contains an `enum` constant `access_mode` that contains a value from `hw_base::access_mode` and specifies whether a register is read-only, write-only or read/write.

Other information that must be specified in the `platform_traits` class often includes:

- A `typedef processor_bus` that specifies to which bus the device is connected if a processor has more than one bus.

- An `enum` constant `processor_bus_width` if that can vary for a given platform.

- An `enum` constant `processor_endianness` to specify the order of bytes for multiple byte bus widths.

Other information that must be specified in the `register_traits` class often includes:

- An `enum` constant `endianness` to specify the order of bytes for multiple byte wide registers.

- An `enum` constant `device_bus_width` to specify width of the device bus to which the register is connected.

As already said, the actual requirement details of `platform_traits` and `register_traits` are platform dependent and can vary widely for more exotic platforms. It is the purpose of the middle layer of the abstract model (§8.1.4) to cope with such requirements and to isolate them from the device driver code.

A.2.1 Using `address_holder`s

An implementation typically provides two pre-defined `address_holder` definitions: a class `static_address` and a class `dynamic_address`, to hold address information known at compile-time and address information that can be initialized at run-time, respectively.

For addresses that are known at compile time, the class template `static_address` defines the actual address through a template argument (there can be more than one template parameter if the address is not a simple scalar value). A simple offset address of a register might be specified as `static_address<0x0b>`.

For address information that must be initialized at run-time, the class `dynamic_address` is provided. This class provides a constructor that accepts as many arguments as necessary for a platform.

For more complex cases it might be necessary to provide a user-specified `address_holder` class. This class must provide a public member function `value()` with the return type `hw_base::address_type` (or something similar, if an implementation provides more than one address type in `hw_base`). If the address is known at compile-time (and therefore marked `is_static` in its traits class), this member function must be `static`.

For example, an implementation might provide a `general_address` for which the dynamic data type is `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;
};
```

Here the initialization of the address information is provided by some global variable. In a different case, the constructor might require an argument, and therefore some initialization code must provide that argument. But the mechanics of the initialization are always left to the user to choose the most suitable method.

A.2.2 Traits Specifications

As already said, the actual requirements details of the register and platform traits are implementation defined by nature. But as the classes `register_traits` and `platform_traits` are provided, in most cases it is quite easy to define the traits for a specific application. Sometimes the `platform_traits` can even be used directly, without any modifications. More often, the provided `platform_traits` is used as a base class with overrides specific to the application:

```
struct DynMM : platform_traits
{
    typedef dynamic_address address_holder;
    typedef hw_base::data_bus processor_bus;
    enum { address_mode=hw_base::is_dynamic };
};
```

In this example, the derived class uses a dynamic base address and the (memory mapped) data bus of the processor.

Here is another example, using the `DynAddressPortDA` address holder from above:

```
struct MySpecialDyn : platform_traits
{
    typedef DynAddressPortDA address_holder;
    enum { address_mode=hw_base::is_dynamic };
};
```

Register traits nearly always have a static address, so it is often useful to provide class templates to cover common cases:

```
template <typename ValType, hw_base::address_type addr>
struct Dev16Reg : public register_traits
{
    typedef ValType value_type;
    typedef static_address<addr> address_holder;
    enum
    {
        address_mode=hw_base::is_static,
        access_mode=hw_base::random,
        endianness=hw_base::msb_high,
        device_bus_width=hw_base::device16
    };
};
```

It is then simple to use this class to define traits for concrete registers:

```
typedef Dev16Reg<uint8_t, 0x04> ControlPort;
```

A.3 Hardware Access

All hardware access is provided through the class templates `register_access` and `register_buffer`. For access traits 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 register traits with ValueType = uint8_t:
//   InPort, OutPort and ControlPort
register_access<InPort, platform_traits>      ip;
register_access<OutPort, platform_traits>     op;
register_access<ControlPort, platform_traits> 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
```

Because the `register_access` object is empty, there is no real need to define these objects, as it is 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, platform_traits>      ip;
typedef register_access<OutPort, platform_traits>     op;
typedef register_access<ControlPort, platform_traits> 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

A `register_buffer` is used to access a block of hardware registers, rather than a single register. In this case the `value_type` definition of the `register_traits` denotes 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
// assume platform traits IObus for a device on the I/O bus
register_buffer<PortBuffer, IObus> 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 as a unit, an appropriate `value_type` can be defined:

```
struct Buffer32 { uint8_t data[32]; };
struct XYBlock : public register_traits
{
    typedef Buffer32 value_type;
    typedef static_address<0x35800> address_holder;
    enum
    {
        address_mode=hw_base::is_static
    };
};
register_access<XYBlock, IObus> blockBuf;
Buffer32 tmpBlock;

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

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

A.3.2 Initialization of register_access

For access traits with static address_holders that are fully specified at compile-time, register_access and register_buffer provide only a default constructor (in these cases there is nothing to construct). But if one of the traits contains an address_holder with dynamic data, this must be initialized at run-time. For those cases, register_access and register_buffer provide a constructor that takes a respective address_holder object as argument. How the address_holder type is initialized is under control of the user, as explained above. As there are two traits arguments for register_access and register_buffer, in theory there can be two dynamic address_holders, though in practice the address_holder of the register_traits is nearly always static. So, regarding the examples from above, the initialization can be:

```
// using default constructor of DynAddressPortDA
register_access<ControlPort, MySpecialDyn> portA(DynAddressPortDA());
```

or in very special cases:

```
// using conversion constructors of the respective address_holders
register_access<SpecialDynReg, DynBase>
    portDB(0x1234,          // dynamic register offset
           0xa0b165);       // dynamic base address
```

Annex B: Implementing the *iohw* Interfaces

B.1 General Implementation Considerations

The `<hardware>` header defines a standardized syntax for basic hardware register addressing. This header should normally be created by the compiler vendor.

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

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

B.1.1 Recommended Steps

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

- Get an overview of all the possible and relevant ways the hardware device is typically connected with the given bus hardware architectures, plus an overview of the basic software methods typically used to address such hardware registers.
- Define specializations of `register_access` and `register_buffer` which support the relevant hardware register access methods for the intended compiler market.
- Provide `platform_traits` and `register_traits` as a way to select the right `register_access` and `register_buffer` specializations at compile time and generate the right machine code based on the hardware register access properties related to the hardware register designators (the traits classes).

B.1.2 Compiler Considerations

In practice, an implementation will often require that very different machine code is generated for different hardware register 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++ memory model.

Selection between different code generation alternatives must be determined solely from the hardware register designator definition 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. Alternatively, inline assembler can be used to produce the required machine instructions.

With a conforming C++ compiler, an efficient, all-round implementation of both the `<iohw.h>` and `<hardware>` interface headers can usually be achieved using C++ template functionality (see also §Annex C:). 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.

B.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.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 hardware register is located at the *lowest* processor bus address. (Note that the hardware register endianness has nothing to do with the endianness of the underlying processor hardware architecture).

[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.]

Table B-1: Logical hardware register / hardware device addressing overview[28]

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.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)

[28] 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.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 hardware registers in buffer

Register width	**Device bus**			**Processor data bus width**			
	Width	*LSB MSB*	*No. Oper-ations.*	*Width=8* *size 1*	*Width=16* *size 2*	*Width=32* *size 4*	*Width=64* *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.2.3.1 Generic Buffer Index

The interleave distance between two logically adjacent registers in a device register array can be calculated from[29]:

- The size of the logical register in bytes

[29] For systems with byte-aligned addressing.

- The processor data bus width in bytes
- The device data bus width in bytes

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

```
Address_offset = index *
                  sizeof( logical_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_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.3 Hardware Register Designators for Different Device Addressing Methods

A processor may have more than one addressing range[30]. 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 hardware register designator consists of a platform traits class with a dynamic address (address of base register) and a register traits class with a static address (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.

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

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 hardware register designator 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 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 buses, 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 traits classes of the hardware register designator contain a user-defined `address_holder`.

  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.

To adapt to the different requirements and interface properties of the different addressing modes, the `<hardware>` interface uses different combinations of platform and register traits classes in the hardware register designators of the different addressing methods.

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

B.4 Atomic Operation

It is a requirement of the *iohw* implementation that in each *iohw* function a given (partial[31]) 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 access 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.5 Read-Modify-Write Operations and Multi-Addressing

In general `READ`-modify-`WRITE` operations should do a complete `READ` of the hardware register, followed by the operation, followed by a complete `WRITE` to the hardware register.

It is therefore recommended that an implementation of multi-addressing cases should not use `READ`-modify-`WRITE` machine instructions during partial register addressing operations.

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

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

B.6 Initialization

Some parts of hardware register designators may require some initialization at run-time, which is done using normal C++ constructors. But quite often, the compiler-known lifetime of such objects is not identical with the logical lifetime that is important for initialization and de-initialization. In such cases, the constructor (through placement `new` syntax) and destructor must be called explicitly.

With respect to the abstract model in §8.1.4, 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 hardware device register operation:

- hardware (device) initialization
- device access initialization of hardware register designators
- device selector (or hardware group) initialization[32] of platform traits

Here only device access initialization and device selector initialization are relevant for basic hardware register addressing:

- **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 hardware addressing. Hardware initialization is therefore not a topic for the standardization process.

- **device access initialization:** This concerns the definition of hardware register designator objects. The actual functionality of this initialization is inherently implementation-defined. It depends both on the platform and processor architecture and also on which underlying access methods are supported by the `<hardware>` implementation. While the functionality is implementation-defined, the syntax for this initialization is the normal C++ syntax of object constructors.

  If runtime initialization is needed, this can easily be done by providing a platform traits class with a dynamic `address_holder`. The register traits class can in most cases still use a static `address_holder`.

- **device selector (or hardware group) initialization of platform traits:** This is used if the platform-specific part of the address information is only available at run-time. In this case the platform traits class contains a dynamic `address_holder`, which must be initialized using normal C++ constructors.

  This can also be used if, for instance, the same hardware device driver code needs to service multiple hardware devices of the same type. But if the addresses of the

[32] 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.

different hardware devices are known at compile time, it is also possible to implement the hardware device driver code as a function template on the `platform_traits` class and call this function with different platform traits with static `address_holder`. Here is an example that demonstrates both options:

```
#include <hardware>

namespace
{
```

// middle layer (hardware register designator specifications):

```
using namespace std::hardware;

struct DeviceA : platform_traits
{
    typedef static_address<0x50> address_holder;
};

struct DeviceB : platform_traits
{
    typedef static_address<0x90> address_holder;
};

struct DynDevice : platform_traits
{
    typedef dynamic_address address_holder;
    enum { address_mode=hw_base::dynamic_address };
};

struct PortA1_T : register_traits
{
    typedef static_address<0x1a> address_holder;
};

struct PortA2_T : register_traits
{
    typedef static_address<0x20> address_holder;
};
```

Portable device driver function using the template approach:

```
template <class PlatformSpec>
uint8_t getDevData(typename PlatformSpec::address_holder const &addr =
            typename PlatformSpec::address_holder())
{
    register_access<PortA1_T, PlatformSpec> devConfig(addr);
    register_access<PortA2_T, PlatformSpec> devData(addr);

    devConfig = 0x33;
    return devData;
}
} // unnamed namespace

int main()
{
```

static version:

```
        // Read data from device 1:
        uint8_t d1 = getDevData<DeviceA>();

        // Read data from device 2:
        uint8_t d2 = getDevData<DeviceB>();
```

dynamic version:

```
        uint8_t d3 = getDevData<DynDevice>(0x40);

        uint8_t d4 = getDevData<DynDevice>(0x80);

        return 0;
    }
```

With most free-standing environments and embedded systems the platform hardware is well defined, so all hardware group designators for device registers used by the program can be completely defined at compile-time. For such platforms run-time device selector initialization is not an issue.

With larger processor systems the base address of a hardware device is often assigned dynamically at run-time. Here only the `register_traits` of the hardware group designator can be defined at compile-time, while the `platform_traits` part of it must be initialized at run-time.

When designing the hardware group designator object a compiler implementer should therefore make a clear distinction between the static information in the `register_traits` class and the dynamic information in the `platform_traits` class; i.e. the register traits class should contain a static `address_holder` that can be defined and initialized at compile time, while the platform traits class should contain a dynamic `address_holder` that must be initialized at runtime.

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

B.7 Intrinsic Features for Hardware Register Access

The implementation of hardware device access operations may require for many platforms the 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 (§IS-7.4) 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.

The implementation should completely encapsulate any intrinsic functionality.

B.8 Implementation Guidelines for the `<hardware>` Interface

There are two main design alternatives in implementing `register_access` and `register_buffer` for the different hardware register designators:

- Using the information in the traits classes of the hardware register designators to implement the `register_access` and `register_buffer` functionality (this is the approach chosen in the sample implementation).

- Using the traits classes of the hardware register designators as mere labels that also hold the address information and specializing `register_access` and `register_buffer` for each of the meaningful combinations of platform and register traits (this is a useful approach if there are very few commonalities between the implementations for the different traits).

In any case, carefully implemented specializations of helper classes used in `register_access` and `register_buffer` together with an optimizing compiler can provide resulting object 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.8.1 Annotated Sample Implementation

The sample implementation implements the `<hardware>` interface for a very simple 8-bit processor. This processor supports only 8-bit buses, but has a memory bus and an I/O bus. This simplifies the implementation, but the necessary steps for a more general implementation are also mentioned. Also, as is typical for such small systems, all address information is assumed to be known at run-time; i.e. dynamic `address_holders` are not supported.

A note on the style: 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.

B.8.1.1 Common Definitions – `struct hw_base`

`hw_base` defines all the constants that are necessary in the hardware register designators' traits classes. Of course, this is highly dependent 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 enumerators.

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 in platform traits classes. Otherwise, as the chosen example platform is pretty simple, `hw_base` is quite small:

```
struct hw_base
{
    enum access_mode {random, read_write, write, read};
    enum device_bus {device8=1};
    enum byte_order {msb_low, msb_high};
    enum processor_bus {bus8=1};

    typedef _ul address_type;
    enum address_kind { is_static, is_dynamic };

    // type names for different bus types
    enum data_bus {};
    enum io_bus {};

};
```

`_ul` is used as shorthand for the type that holds an address and is defined as 16-bit type:

```
typedef uint16_t _ul;
```

And the other required types are defined as well:

```
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned long uint32_t;

typedef unsigned char size_t;
typedef signed char ptrdiff_t;
```

These definitions are inside of `std::hardware`, so the `size_t` and `ptrdiff_t` types can be differently defined as the respective global types.

The width definitions for `device_bus` and `processor_bus` are not really necessary, as this platform supports only 8-bit buses. Therefore, any endianness doesn't matter and the definition of `byte_order` could also be omitted. But they are shown here for demonstration purposes.

B.8.1.2 Access Traits Classes

In this sample implementation the traits classes of the hardware register designators hold all necessary access property information and provide them to the implementation of `register_access` and `register_buffer`. To produce as little overhead as possible in cases where the address information is known at compile- time, no object data is produced. The address value is kept in the type information of the `address_holder` `static_address`:

```
template <_ul val>
struct static_address
{
    enum { value_=val };
    static _ul value() { return value_; }
};
```

As this platform only supports statically known addresses, only this `address_holder` is required. Where dynamically initialized addresses are also supported, a respective dynamic `address_holder` is required:

```
// a class to hold address information dynamically
struct dynamic_address
{
    dynamic_address(_ul _addr) : value_(_addr) {}
    _ul value() const { return value_; }
    _ul value_;
};
```

The default traits classes don't define much more than the `address_holder`, as everything else is fixed for this platform. Only the `platform_traits` define the I/O bus as the default bus:

```
struct platform_traits
{
    typedef static_address<0> address_holder;
    typedef hw_base::io_bus processor_bus;
    enum
    {
        address_mode=hw_base::is_static
    };
};

struct register_traits
{
    typedef uint8_t value_type;
    typedef static_address<0> address_holder;
    enum
    {
        address_mode=hw_base::is_static,
        access_mode=hw_base::random
    };
};
```

In a more flexible environment, these classes would provide more information:

```
struct platform_traits
```

```
{
    typedef static_address<0> address_holder;
    typedef hw_base::data_bus processor_bus;
    enum
    {
       address_mode=hw_base::is_static,
       processor_endianness=hw_base::msb_high,
       processor_bus_width=hw_base::bus32
    };
};

struct register_traits
{
    typedef uint8_t value_type;
    typedef static_address<0> address_holder;
    enum
    {
       address_mode=hw_base::is_static,
       access_mode=hw_base::random,
       endianness=hw_base::msb_high,
       device_bus_width=hw_base::device16
    };
};
```

B.8.1.3 The Interface `register_access` and `register_buffer`

The actual interface for `register_access` is realized by the class template `register_access`. This provides the full interface for single registers. As the sample platform supports only static addresses, only a default constructor is required:

```
template <class _RegTraits, class _PlatformTraits>
class register_access
{
public:
    register_access() {}

    typedef typename _RegTraits::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);
// functional interface omitted for brevity
};
```

This template needs no data members, as all address information is held in the type definitions. If a platform supports dynamic addresses as well, the template would have to hold respective data members:

```
private:
    const typename _RegTraits::address_holder _addrR;
    const typename _PlatformTraits::address_holder _addrP;
```

This would not cause any overhead for static address holders, as those static types have no data members and are simply empty types, which are completely optimized away by the compiler. But there is another problem: `register_access` instantiations on traits classes with dynamic address holders require appropriate constructors. One option is simply to provide all constructors in the same class template:

```
register_access();

explicit register_access
        (typename _RegTraits::address_holder const &_rAddr);

explicit register_access
        (typename _PlatformTraits::address_holder const &_pAddr);

register_access
    (typename _RegTraits::address_holder const &_rAddr,
     typename _PlatformTraits::address_holder const &_pAddr);
```

But this would allow for construction with two arguments even if both address holders are static. To avoid this, a common base class template `_RAInterface` can be introduced with all interface functions. Then the `register_access` class template can inherit (privately) from `_RAInterface` and import the functions from `_RAInterface` with *using-declaration*s. The `register_access` class template is then specialized on the address_mode of the traits classes to provide only that constructor for each class that is meaningful:

```
template <class _RegTraits, class _PlatformTraits,
          int = _RegTraits::address_mode,
          int = _PlatformTraits::address_mode>
class register_access
    : private _RAInterface<_RegTraits, _PlatformTraits>
{
    typedef typename _RegTraits::address_holder _AddressHolderR;
    typedef typename _PlatformTraits::address_holder _AddressHolderP;
    typedef _RAInterface<_RegTraits, _PlatformTraits> _Base;
public:
    register_access(_AddressHolderR const &rAddr,
                    _AddressHolderP const &pAddr)
        : _Base(rAddr, pAddr) {}

    using typename _Base::value_type;
    using _Base::operator value_type;

    using _Base::operator=;
    using _Base::operator|=;
    using _Base::operator&=;
    using _Base::operator^=;
};
```

And the specialization for static addresses for both traits provides only the default constructor:

```
// specialization for static platform and register address
template <class _RegTraits, class _PlatformTraits>
class register_access<_RegTraits, _PlatformTraits,
                        hw_base::is_static, hw_base::is_static>
    : private _RAInterface<_RegTraits, _PlatformTraits>
{
    typedef _RAInterface<_RegTraits, _PlatformTraits> _Base;
public:
    register_access() : _Base() {}
// the using declarations...
};
```

But the sample platform doesn't support `dynamic_address` and therefore `_RAInterface` can be omitted.

The class template `register_buffer` has the same problem (and the same or a similar solution), but the sample platform implementation again is simple. The only problem is the `ref_type` definition. Clearly it is some instantiation of `register_access`, but the problem is that the result of the subscript operator needs some knowledge about the index to perform the hardware access to the correct address. But this index might not be known at compile time, and currently `register_access` has no way to hold a dynamic address.

The solution is the same as the one above for dynamic addresses in the traits: an additional template parameter for the type of the index holder and a data member of that type (which is empty if there is no index). For the holder of the index itself a class similar to `dynamic_address` above is used, but with `size_t` as the value type:

```
// a class to hold an index value dynamically
struct _IdxHolder
{
    _IdxHolder(size_t _i) : value_(_i) {}
    size_t value() const { return value_; }
    size_t value_;
};
```

With that, the definitions for `register_buffer` and `register_access` can be completed:

```
template <class _RegTraits, class _PlatformTraits,
          class _IndexHolder = static_address<0> >
class register_access
{
public:
    register_access() {}
    explicit register_access(_IndexHolder const &_i) : _idx(_i) {}

    typedef typename _RegTraits::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);

private:
    _IndexHolder _idx;
};
```

```
template <class _RegTraits, class _PlatformTraits>
class register_buffer
{
    typedef register_access<_RegTraits,
                            _PlatformTraits,
                            _IdxHolder> ref_type;
public:
    register_buffer() {}

    ref_type operator[](size_t _idx) const
    {
        return ref_type(_idx);
    }
};
```

The constructor for the index type in `register_access` can be private, and `register_buffer` can be declared as friend, but this is omitted here for brevity.

Instead of directly implementing the functions in `register_access`, to save some typing and better separate the different tasks some helper classes are introduced: `_RAImpl` combines the different assignment functions and performs the address calculation, while `_AccessHelper` is concerned with different specializations for register value types that are larger than the connecting bus. Finally, `_hwRead` and `_hwOp` provide the actual hardware access functionality for the different processor buses.

To combine the different assignment functions (at least for the intermediate steps), an enumeration for the different assignment operations is defined:

```
enum _binops { _write_op, _or_op, _and_op, _xor_op };
```

Using that, `register_access` can delegate the functions to `_RAImpl`:

```
template <class _RegTraits, class _PlatformTraits,
          class _IndexHolder = static_address<0> >
class register_access
{
    typedef _RAImpl<_RegTraits, _PlatformTraits, _IndexHolder> _Impl;

public:
    operator value_type() const
    {
        return _Impl::_read(_idx);
    }
    void operator=(value_type _val)
    {
        _Impl::template _op<_write_op>(_idx, _val);
    }
    void operator|=(value_type _val)
    {
        _Impl::template _op<_or_op>(_idx, _val);
    }
    // etc.
};
```

`_RAImpl` then performs the actual address calculation and then delegates further to `_AccessHelper`:

```
template <class _RegTraits, class _PlatformTraits, class _IndexHolder>
class _RAImpl
{
public:
    typedef typename _RegTraits::value_type _ValT;

    static _ul _addrCalc(_IndexHolder const &_idx)
    {
        return _PlatformTraits::address_holder::value()
            + _RegTraits::address_holder::value()
            + _idx.value() * sizeof(_RegTraits::value_type);
    }

    static _ValT _read(_IndexHolder const &_idx)
    {
        return _AccessHelper<_ValT,
                             typename _PlatformTraits::processor_bus,
                             sizeof(_ValT)>
            ::_read(_addrCalc(_idx));
    }

    template <_binops function>
    static void _op(_IndexHolder const &_idx, _ValT _val)
    {
        _AccessHelper<_ValT,
                      typename _PlatformTraits::processor_bus,
                      sizeof(_ValT)>
            ::template _op<function>(_val, _addrCalc(_idx));
    }
};
```

_addrCalc is simple for the sample platform, but is a bit more complex in the general case (see §B.2.3.1), but all required information is in the traits classes that are still template parameters for `_RAImpl`.

Apart from the address calculation, `_RAImpl` simply delegates further to `_AccessHelper`. The purpose of `_AccessHelper` is to separate the single hardware accesses from the ones where the register's value type is larger than the connecting bus and therefore multiple accesses are required. For the sample implementation on an 8-bit platform any access to registers with more than one byte requires multiple accesses, so the specialization can be done on `sizeof(value_type)`:

```
// general case that uses a for-loop
template <typename _ValT, typename _BusTag, size_t _s>
struct _AccessHelper
{
    static _ValT _read(_ul _addr)
    {
        uint8_t buffer[_s];
        for (uint8_t _i=0; _i != _s; ++_i)
        {
            buffer[_i] = _hwRead<_BusTag>::r(_addr+_i);
        }
        return *((_ValT *)buffer);
    }
    template <_binops _func>
```

```
        static void _op(_ValT _val, _ul _addr)
        {
            for (uint8_t _i=0; _i != _s; ++_i)
            {
                _hwOp<_func, _BusTag>
                    ::f(_addr+_i, ((uint8_t *)&_val)[_i]);
            }
        }
    };

    // here the specialization for size==1
    template <typename _ValT, typename _BusTag>
    struct _AccessHelper<_ValT, _BusTag, 1>
    {
        static _ValT _read(_ul _addr)
        {
            return (_ValT)_hwRead<_BusTag>::r(_addr);
        }
        template <_binops _func>
        static void _op(_ValT _val, _ul _addr)
        {
            _hwOp<_func, _BusTag>::f(_addr, (uint8_t)_val);
        }
    };
```

For a more flexible platform, _AccessHelper must be specialized for each valid pair matching the size of the value_type and the width of the device bus, with additional specializations for different endiannesses. To achieve that, `_AccessHelper` needs the complete traits classes as template arguments.

The final separation is done on the processor bus type: different access instructions are necessary for the (memory mapped) data bus than for the I/O bus. This is done by a specialization on `_hwRead` and `_hwOp` based on the bus.

B.8.1.4 Actual Access Implementation

The actual hardware access method depends on the processor architecture and the type of the bus where a hardware device is connected. For the memory mapped case normal C++ expressions together with some (completely machine-dependent) casts can do the access:

The general declaration:

```
    template <typename _BusTag> struct _hwRead;
    template <_binops, typename _BusTag> struct _hwOp;
```

The cast:

```
    template <_binops _op> struct _hwOp< _op, hw_base::data_bus>
    {
        static void f(_ul _addr, uint8_t _rhs)
        {
            _hwOp_data<_op>
                ::f(*const_cast<uint8_t volatile *>
                        (reinterpret_cast<uint8_t *>(_addr)),
                      _rhs);
            }
        };
```

The address (which is of integer type) is first cast to a pointer (to `uint8_t` in the sample case, in the more general case the `value_type` must be transferred as a further template argument), and that pointer is then cast to a pointer to volatile to tell the compiler not to make any assumptions on the memory cell. That cast done, the access is accomplished by a specialization of yet another helper class:

```
        // helper class declaration
        template <_binops> struct _hwOp_data;

        // and one specialization for each operation:
        struct _hwOp_data<_write_op>
        {
            static void f(uint8_t volatile &_lhs, uint8_t _rhs)
            {
                 _lhs = _rhs;
            }
        };
        struct _hwOp_data<_or_op>
        {
            static void f(uint8_t volatile &_lhs, uint8_t _rhs)
            {
                _lhs |= _rhs;
            }
        };
        struct _hwOp_data<_and_op>
        {
            static void f(uint8_t volatile &_lhs, uint8_t _rhs)
            {
                _lhs &= _rhs;
            }
        };
        struct _hwOp_data<_xor_op>
        {
            static void f(uint8_t volatile &_lhs, uint8_t _rhs)
            {
                _lhs ^= _rhs;
            }
        };
```

And for the read, the same cast sequence is required:

```
        struct _hwRead<hw_base::data_bus>
        {
            static uint8_t r(_ul const & _addr)
            {
                return
                    *const_cast<uint8_t volatile *>
                        (reinterpret_cast<uint8_t *>(_addr));
            }
        };
```

For registers that are attached to the I/O bus, special machine instructions must be generated. For this, some compiler specific extensions are necessary. The sample implementation uses the `asm` extensions of GCC.

Using these extensions, the basic access functions can be defined:

```
inline uint8_t i_io_rd(uint8_t _port)
{
    uint8_t _ret;
    asm volatile ("in %0,%1" : "=r" (_ret) : "i" (_port));
    return _ret;
}

inline void i_io_wr(uint8_t _port, uint8_t _val)
{
    asm volatile ("out %0,%1" : : "i" (_port), "r" (_val));
}

inline void i_io_and(uint8_t _port, uint8_t _val)
{
    uint8_t _tmp;
    asm volatile ("in %0,%1\n\tand %0,%2\n\tout %1,%0"
                  : "=&r" (_tmp) : "i" (_port), "r" (_val));
}

inline void i_io_or(uint8_t _port, uint8_t _val)
{
    uint8_t _tmp;
    asm volatile ("in %0,%1\n\tor %0,%2\n\tout %1,%0"
                  : "=&r" (_tmp) : "i" (_port), "r" (_val));
}

inline void i_io_xor(uint8_t _port, uint8_t _val)
{
    uint8_t _tmp;
    asm volatile ("in %0,%1\n\teor %0,%2\n\tout %1,%0"
                  : "=&r" (_tmp) : "i" (_port), "r" (_val));
}
```

These basic functions can then be used to implement the specializations for the I/O bus:

```
struct _hwRead<hw_base::io_bus>
{
    static uint8_t r(_ul const & _addr) { return i_io_rd(_addr); }
};
struct _hwOp<_write_op, hw_base::io_bus>
{
    static void f(_ul _addr, uint8_t _rhs) { i_io_wr(_addr, _rhs); }
};
struct _hwOp<_or_op, hw_base::io_bus>
{
    static void f(_ul _addr, uint8_t _rhs) { i_io_or(_addr, _rhs); }
};
struct _hwOp<_and_op, hw_base::io_bus>
{
    static void f(_ul _addr, uint8_t _rhs) { i_io_and(_addr, _rhs); }
};
struct _hwOp<_xor_op, hw_base::io_bus>
```

```
{
    static void f(_ul _addr, uint8_t _rhs) { i_io_xor(_addr, _rhs); }
};
```

B.8.1.5 Usage and Overhead

Using that implementation, the `<hardware>` interface can be used as specified:

```
namespace
{
// middle layer (hardware register designators)
using namespace std::hardware;

typedef platform_traits IObus;

struct MMbus : platform_traits
{
    typedef hw_base::data_bus processor_bus;
};

template <typename ValType, std::hardware::size_t addr>
struct StaticReg : public register_traits
{
    typedef ValType value_type;
    typedef static_address<addr> address_holder;
    enum
    {
        address_mode=hw_base::is_static
    };
};

} // anonymous namespace

// test
int main()
{
    register uint8_t v, i;

    register_access<StaticReg<uint8_t, 0x23>, IObus> port1;
    i = port1;          // (1)

    register_access<StaticReg<uint8_t, 0x24>, IObus> port2;
    port2 &= 0xaa;      // (2)

    register_access<StaticReg<uint8_t, 0x25>, IObus> port3;
    port3 = 0x17;       // (3)

    register_access<StaticReg<uint8_t, 0xab>, MMbus> mem1;
    v = mem1;           // (4)
    mem1 &= 0x55;       // (5)
    mem1 = v;           // (6)

    register_buffer<StaticReg<uint8_t, 0x0a>, MMbus> memBuf;
    v = memBuf[i];      // (7)
    memBuf[4] &= 0x03;  // (8)

    return 0;
}
```

The compiler output for this small program looks very different depending on the optimization level. Without optimization, the generated code is horrible as none of the many intermediate functions is inlined. The result is lots of function calls (and related stack handling).

With low optimization (-O1) the resulting code is essentially what one could expect:

(1) results in two machine instruction, one for the port read and one to store the result in a separate register).

C++:

```
    i = port1;        // (1)
```

Assembler:

```
    5a:    83 b5         in     r24, 0x23     ; R24 is used as accumulator
    5c:    48 2f         mov    r20, r24      ; r20 is 'i' variable
```

(2) results in four machine instructions, one register load and the `in/and/out` sequence.

C++:

```
    port2 &= 0xaa;        // (2)
```

Assembler:

```
    5e:    8a ea         ldi    r24, 0xAA
    60:    94 b5         in     r25, 0x24
    62:    98 23         and    r25, r24
    64:    94 bd         out    0x24, r25
```

(3) results in two machine instructions, one register load and the `out` instruction.

C++:

```
    port3 = 0x17;        // (3)
```

Assembler:

```
    66:    87 e1         ldi    r24, 0x17
    68:    85 bd         out    0x25, r24
```

(4) results in two machine instruction, one access to the memory and one register move for the resulting `v`.

C++:

```
    v = mem1;            // (4)
```

Assembler:

```
    6a:    80 91 ab 00   lds    r24, 0x00AB
    6e:    28 2f         mov    r18, r24
    70:    33 27         eor    r19, r19      ; superfluous nulling of R19
```

(5) results in three machine instructions, again one read access to the memory (that was declared as `volatile`), the `and`, and a store to the memory.

C++:

```
mem1 &= 0x55;       // (5)
```

Assembler:

```
72:    80 91 ab 00  lds    r24, 0x00AB
76:    85 75        andi   r24, 0x55
78:    80 93 ab 00  sts    0x00AB, r24
```

(6) results in one machine instruction, the value of `v` that is still in a register is stored to the memory.

C++:

```
mem1 = v;           // (6)
```

Assembler:

```
7c:    20 93 ab 00  sts    0x00AB, r18
```

(7) takes a number of machine instructions, as the address calculation is in 16-bit, which takes several instructions on an 8-bit processor.

C++:

```
v = memBuf[i];    // (7)
```

Assembler:

```
80:    8a e0        ldi    r24, 0x0A    ; loading base address LSB
82:    90 e0        ldi    r25, 0x00    ; loading base address MSB
84:    f9 2f        mov    r31, r25     ; move to address register
86:    e8 2f        mov    r30, r24
88:    e4 0f        add    r30, r20     ; add the index
8a:    f1 1d        adc    r31, r1      ; add 0 as MSB
8c:    20 81        ld     r18, Z       ; load using address register
```

(8) finally takes six machine instructions, two for the move of the 16-bit base address (0x0a, still in the register), one for the add of the index, one for the load of the memory value, one for the `and` and a last one for the store back to the memory.

C++:

```
memBuf[4] &= 0x03;  // (8)
```

Assembler:

```
8e:    f9 2f        mov    r31, r25;    ; base address still in R24/25
90:    e8 2f        mov    r30, r24
92:    34 96        adiw   r30, 0x04    ; add index (as 16-bit value)
94:    80 81        ld     r24, Z       ; load using address register
96:    83 70        andi   r24, 0x03
98:    80 83        st     Z, r24       ; store using address register
```

The most annoying case is (8), as the base address and the index are both known at compile-time, but the computation is done at run-time.

But with optimization one level higher (-O2) that is also solved:

(7) knows the base address from compile-time and comes down to four machine instructions.

Assembler:

```
80:    e9 2f         mov    r30, r25      ; move 'i' to address register
82:    ff 27         eor    r31, r31;     ; zeor MSB of address register
84:    3a 96         adiw   r30, 0x0a     ; add base address
86:    80 81         ld     r24, Z        ; load using address register
```

(8) comes down to three machine instructions as the final address 0x0e is completely computed at compile-time and therefore does only one load, the `and` and the store.

Assembler:

```
88:    80 91 0e 00   lds    r24, 0x000E   ; load final address
8c:    83 70         andi   r24, 0x03
8e:    80 93 0e 00   sts    0x000E, r24   ; store using R24 as address
```

So for this platform and for optimization level -O2 the goal of a non-overhead implementation is reached.

Annex C: A `<hardware>` Implementation for the `<iohw.h>` Interface

The implementation of the basic `<iohw.h>` hardware register access interface on top of the `<hardware>` interface is mainly straightforward. This section provides an example of how such an implementation can be achieved.

The purpose of using C++ at the lowest level is to take advantage of compile-time evaluation of template code to yield object code specialized for specific hardware. For a good implementation of the basic templates that perform the lowest-level hardware access operations, this approach typically leads to code that maps directly to machine instructions as efficient as code produced by an expert programmer. Additionally, the type safety of the C++ interface minimizes debugging and errors.

The sample implementation presented here uses the sample `<hardware>` implementation presented in §B.8.

C.1 Implementation of the Basic Access Functions

The sample implementation here avoids the creation of unnecessary objects and instead generally passes *ioreg-designator* arguments in the form of (properly typed) null pointers. But it would also be possible to pass them as normal objects, as long as they contain no data members, as the compiler typically optimizes them away. Though the null pointer is syntactically de-referenced in the access functions, it is never actually de-referenced, as the objects do not contain any data members.

The access functions are implemented as function templates on the *ioreg-designator*, which must be an instantiation of `register_access`:

```
template <typename _RegAcc>
inline typename _RegAcc::value_type
iord(_RegAcc * _reg)
{
    return static_cast<typename _RegAcc::value_type>(*_reg);
}
template <typename _RegAcc>
inline void
iowr(_RegAcc * _reg, typename _RegAcc::value_type _val)
{
    *_reg = _val;
}
template <typename _RegAcc>
inline void
ioor(_RegAcc * _reg, typename _RegAcc::value_type _val)
{
    *_reg |= _val;
}
// etc.
```

The `iord` implementation calls the conversion operator of `register_access` explicitly by using a `static_cast`, but this is not really necessary.

This can be used by providing a middle layer that is essentially the same as for the `<hardware>` interface as presented in the previous chapter:

```
typedef platform_traits IObus;

template <typename ValType, size_t addr>
struct Dev8Reg : public register_traits
{
    typedef ValType value_type;
    typedef static_address<addr> address_holder;
    enum
    {
        address_mode=hw_base::is_static,
        access_mode=hw_base::random,
        device_bus_width=hw_base::device8
    };
};

register_access<Dev8Reg<uint8_t, 0x06>, IObus > *myPort1 = 0;
```

The only difference to the middle layer of the `<hardware>` interface is that the final designator is defined as a pointer (and initialized to null).

The device driver code itself refers only to functions named in the `<iohw.h>` interface:

```
uint8_t val8;

val8 = iord(myPort1); // read single register
iowr(myPort1, 0x9); // write single register
```

C.2 Buffer Functions

The buffer functions are analogous to the single register functions. They are also implemented as function templates and their template argument must be an instantiation of `register_buffer`:

```
template <typename _RegBuf>
inline typename _RegBuf::value_type
iordbuf(_RegBuf * _reg, ioindex_t _idx)
{
    return (*_reg)[_idx];
}
template <typename _RegBuf>
inline void
iowrbuf(_RegBuf * _reg, ioindex_t _idx, typename _RegBuf::value_type
_val)
{
    (*_reg)[_idx] = _val;
}
template <typename _RegBuf>
```

```
inline void
ioorbuf(_RegBuf * _reg, ioindex_t _idx, typename _RegBuf::value_type
_val)
{
    (*_reg)[_idx] |= _val;
}
```

Here, the `iordbuf` implementation uses the conversion operator of `register_access` implicitly.

Again, the respective middle layer is similar to the middle layer of the `<hardware>` interface:

```
struct MMbus : platform_traits
{
    typedef hw_base::data_bus processor_bus;
};

template <typename ValType, size_t addr>
struct Dev8Reg : public register_traits;  // as above

register_buffer<Dev8Reg<uint16_t, 0x04>, MMbus > *myBuf = 0;
```

And again, the device driver code uses only `<iohw.h>` functionality:

```
uint16_t buffer[10];
uint8_t val8;

for (ioindex_t i = 0; i != 10; ++i )
{
    buffer[i] = iordbuf(myBuf, i); // read register array
    iowrbuf(myBuf, i, buffer[i]); // write register array
    ioorbuf(myBuf, val8, buffer[i]); // or register array
}
```

C.3 Group Functionality

Up to this point, the implementation of `<iohw.h>` has used only the interface of `<hardware>`, not its implementation. However, this might not be possible for the grouping functionality of `<iohw.h>`. The sample implementation here uses the internal implementation of the `<hardware>` interface, but it does not require any changes to that implementation.

While "normal" hardware register designators always use the combination of register traits and platform traits together, for hardware register groups these are separated. The indirect designators contain only the register traits, while the direct designators contain the platform traits. Only through a call to `iogroup_map` are they combined to make a fully usable designator. But when working with groups, the device driver code syntactically uses the indirect designators for the access functions, so they need to know which direct designator is currently mapped to them. And the access function must combine the address information from both designators to form the final address that is accessed.

The actual designators for indirect groups must be an instantiation of the class template `_IOGroupRegister`. This is just a type holder for the used register traits and the direct designator that can be mapped to this group:

```
template <class _RegTraits, class _GrpBase>
struct _IOGroupRegister {};
```

The `_GrpBase` template argument must be an instantiation of a platform traits class with a special `address_holder` that provides a static `value_` member that can be modified and the `value()` function must also be `static`. That way, the class `_IOGroupRegister` does not need a pointer to keep the connection to the mapped direct designator. Such a pointer would introduce a major overhead in space and time for the hardware accesses. The implementation provides such an address holder `_BaseAddress` as a class template that can be used by the middle layer of an application to define respective platform traits:

```
template <int _id>
struct _BaseAddress
{
    static _ul value() { return value_; }
    static _ul value_;
};
template <int _id>
_ul _BaseAddress<_id>::value_;
```

The `_id` template parameter serves to differentiate the different direct designators if more than one is used in an application. Such an address holder can still be declared as `is_static` in the traits where it is used, as it offers exactly the same interface as "normal" compile-time `static_address` holders.

The access functions of `<iohw.h>` must be redefined for indirect designators, through a set of overloads. For the purpose of implementation, the overload set presented here uses the `_RAImpl` helper class presented in §B.8.1.3[33]:

```
template <class _RegTraits, class _GrpBase>
inline typename _RegTraits::value_type
iord(_IOGroupRegister<_RegTraits, _GrpBase> _reg)
{
    return _RAImpl<_RegTraits, _GrpBase>::_read(0);
}

template <class _RegTraits, class _GrpBase>
inline void
iowr(_IOGroupRegister<_RegTraits, _GrpBase> _reg,
     typename _RegTraits::value_type _val)
{
    _RAImpl<_RegTraits, _GrpBase>::template _op<_write_op>(0, _val);
}

// etc.
```

[33] The `_RAImpl` template has been slightly modified in this implementation for typing convenience: the `_IndexHolder` template parameter is assumed to have a default argument of `_IdxHolder`.

Internally `_RAImpl` uses `_GrpBase::address_holder::value()`, which is exactly the interface provided by `_BaseAddress`.

The implementation for the buffer functions is quite similar to that of the basic access functions: an empty class template `_IOGroupBuffer` is defined to provide the necessary type information, and that is used to instantiate `_RAImpl` and call its member functions directly:

```
template <class _RegTraits, class _GrpBase>
struct _IOGroupBuffer {};

template <class _RegTraits, class _GrpBase>
inline typename _RegTraits::value_type
iordbuf(_IOGroupBuffer<_RegTraits, _GrpBase> _reg,
        ioindex_t _i)
{
    return _RAImpl<_RegTraits, _GrpBase>::_read(_i);
}

template <class _RegTraits, class _GrpBase>
inline void
iowrbuf(_IOGroupBuffer<_RegTraits, _GrpBase> _reg,
        typename _RegTraits::value_type _val,
        ioindex_t _i)
{
    _RAImpl<_RegTraits, _GrpBase>::template _op<_write_op>(_i, _val);
}
```

The middle layer in this implementation is similar to the middle layer for the `<hardware>` interface:

```
// the platform traits to be used for group designators
template <int baseId>
struct DynMM : platform_traits
{
    typedef _BaseAddress<baseId> address_holder;
    typedef hw_base::data_bus processor_bus;
    enum { address_mode=hw_base::is_static };
};

// the designators
typedef DynMM<1> DevGroupT;
_IOGroupRegister<Dev8Reg<uint8_t, 0x00>, DevGroupT> devConfig;
_IOGroupBuffer<Dev8Reg<uint8_t, 0x04>, DevGroupT> devData;

DevGroupT *devGroup = 0;
```

`devConfig` and `devData` are the indirect designators to be used as arguments to the access functions. `devGroup` or `DevGroupT` is not really a direct designator; it is merely a place holder to define the group. The actual direct designators must provide the functionality to be used in the group functions of `<iohw.h>`, which are `iogroup_acquire()`, `iogroup_release()` and `iogroup_map()`. Therefore the real direct designators must provide the member functions `init()` and `release()` and the data member `value` that has the same type as the `value_` member of the address holder for the group. Not every direct designator needs non-trivial initialization and release functions, so a helper class is provided for convenience to save defining unnecessary functions:

```
struct _EmptyGroup
{
    void init() {}
    void release() {}
};
```

Using that, the middle layer can provide the direct designators for the group:

```
struct Dev1GroupT : _EmptyGroup
{
    void init() { value = 0x0020; }
    hw_base::address_type value;
} dev1Group;

struct Dev2GroupT : _EmptyGroup
{
    void init() { value = 0x0120; }
    hw_base::address_type value;
} dev2Group;
```

Of course, for real life applications the `init()` function will typically be a bit more complex.

Based on that interface, the implementation of the group functions is easy:

```
template <class _Grp>
inline void iogroup_acquire(_Grp &_g)
{
    _g.init();
}
template <class _Grp>
inline void iogroup_release(_Grp &_g)
{
    _g.release();
}

template <class _GrpBase, class _Grp>
inline void iogroup_map(_GrpBase *, _Grp const &_g)
{
    _GrpBase::address_holder::value_ = _g.value;
}
```

The device driver code again uses only `<iohw.h>` functionality:

```
uint8_t get_dev_data(void)
{
    iowr(devConfig, 0x33);
    return iordbuf(devData, 3);
}

// ...
iogroup_acquire(dev1Group);
// Read data from device 1
iogroup_map(devGroup, dev1Group);
```

```
uint8_t d1 = get_dev_data();
iogroup_release(dev1Group);

iogroup_acquire(dev2Group);
// Read data from device 2
iogroup_map(devGroup, dev2Group);
uint8_t d2 = get_dev_data();
iogroup_release(dev2Group);
```

C.4 Remarks

The implementation here does not completely conform to the `<iohw.h>` interface in WDTR 18037. That definition requires a value type for the access functions of `unsigned int` and a second set of access functions with the suffix `'l'` with a value type of `unsigned long`. That is not only unnecessarily constraining (in general, the *iohw* interface allows transfers of non-integer number types as well as any POD `struct` type), but also introduces a major overhead for many real-life devices where registers are only 8 bits wide. Therefore this implementation allows for generic value types.

In general, using C++ for the implementation of `<iohw.h>` introduces no overhead, but allows for a common implementation of the `<iohw.h>` interface and the more generic `<hardware>` interface.

The implementation summarized here is not the only possible C++ implementation. Complete code for this implementation and some alternatives can be found on the WG21 web site, www.open-std.org/jtc1/sc22/wg21.

Annex D: Timing Code

D.1 Measuring the Overhead of Class Operations

This is the sample program discussed in §5.3.2 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
```