

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)

AVA Documents
Document Sharing Hub
© ISO/IEC 2006

PDF disclaimer

This PDF file may contain embedded typefaces. In accordance with Adobe's licensing policy, this file may be printed or viewed but shall not be edited unless the typefaces which are embedded are licensed to and installed on the computer performing the editing. In downloading this file, parties accept therein the responsibility of not infringing Adobe's licensing policy. The ISO Central Secretariat accepts no liability in this area.

Adobe is a trademark of Adobe Systems Incorporated.

Details of the software products used to create this PDF file can be found in the General Info relative to the file; the PDF-creation parameters were optimized for printing. Every care has been taken to ensure that the file is suitable for use by ISO member bodies. In the unlikely event that a problem relating to it is found, please inform the Central Secretariat at the address given below.

© ISO/IEC 2006

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying and microfilm, without permission in writing from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Case postale 56 • CH-1211 Geneva 20
Tel. + 41 22 749 01 11
Fax + 41 22 749 09 47
E-mail copyright@iso.org
Web www.iso.org

Published in Switzerland

Contents

Contents	iii
Foreword	v
Introduction	vi
Participants	vii
1 Scope	1
2 Normative References	3
3 Terms and definitions.....	4
4 Typical Application Areas.....	13
4.1 Embedded Systems	13
4.2 Servers	15
5 Language Features: Overheads and Strategies.....	16
5.1 Namespaces.....	16
5.2 Type Conversion Operators	17
5.3 Classes and Inheritance.....	18
5.4 Exception Handling	27
5.5 Templates.....	37
5.6 Programmer Directed Optimizations	41
6 Creating Efficient Libraries	63
6.1 The Standard <i>IOStreams</i> Library – Overview	63
6.2 Optimizing Libraries – Reference Example: "An Efficient Implementation of <i>Locales</i> and <i>IOStreams</i> "	64
7 Using C++ in Embedded Systems.....	77
7.1 ROMability	77
7.2 Hard Real-Time Considerations	81
8 Hardware Addressing Interface	85
8.1 Introduction to Hardware Addressing	86
8.2 The <code><iohw.h></code> Interface for C and C++	102
8.3 The <code><hardware></code> Interface for C++	108

Annex A: Guidelines on Using the <hardware> Interface	118
A.1 Usage Introduction.....	118
A.2 Using Hardware Register Designator Specifications.....	118
A.3 Hardware Access	121
Annex B: Implementing the <i>iohw</i> Interfaces.....	124
B.1 General Implementation Considerations	124
B.2 Overview of Hardware Device Connection Options	125
B.3 Hardware Register Designators for Different Device Addressing Methods.....	128
B.4 Atomic Operation.....	130
B.5 Read-Modify-Write Operations and Multi-Addressing.....	130
B.6 Initialization.....	131
B.7 Intrinsic Features for Hardware Register Access.....	133
B.8 Implementation Guidelines for the <hardware> Interface	134
Annex C: A <hardware> Implementation for the <iohw.h> Interface	149
C.1 Implementation of the Basic Access Functions.....	149
C.2 Buffer Functions.....	150
C.3 Group Functionality	151
C.4 Remarks.....	155
Annex D: Timing Code.....	156
D.1 Measuring the Overhead of Class Operations	156
D.2 Measuring Template Overheads.....	165
D.3 The Stepanov Abstraction Penalty Benchmark	171
D.4 Comparing Function Objects to Function Pointers.....	177
D.5 Measuring the Cost of Synchronized I/O	181
Annex E: Bibliography	184

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 **Abstract Base Class** – 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 registers** (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 follows¹:

- **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 ROM².

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 operators*, using syntax that looks like *function templates*, for example:

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

The four syntactic forms are:

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

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

The first three forms of *type conversion operator* have no size or speed penalty versus the equivalent *cast notation*. Indeed, it is typical for a compiler to transform *cast notation* into one of the other *type conversion operators* 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 structs`). In particular, a non-virtual function does not take up any space in an object of its `class`, and neither does a static data or function member of the `class`.

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 compilers⁴ 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
<i>Non-virtual:</i>	<code>px->f(1)</code>	0.019	0.002	0.016	0.085	0
	<code>g(ps,1)</code>	0.020	0.002	0.016	0.067	0
<i>Non-virtual:</i>	<code>x.g(1)</code>	0.019	0.002	0.016	0.085	0
	<code>g(&s,1)</code>	0.019	0	0.016	0.067	0.001
<i>Static member:</i>	<code>X::h(1)</code>	0.014	0	0.013	0.069	0
	<code>h(1)</code>	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
<i>Virtual:</i>	<code>px->f(1)</code>	0.025	0.012	0.019	0.078	0.059
<i>Ptr-to-fct:</i>	<code>p[1](ps,1)</code>	0.020	0.002	0.016	0.055	0.052
<i>Virtual:</i>	<code>x.f(1)</code>	0.020	0.002	0.016	0.071	0
<i>Ptr-to-fct:</i>	<code>p[1](&s,1)</code>	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-parameters* 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
<i>Non-inline:</i>	<code>px->g(1)</code>	0.019	0.002	0.016	0.085	0
<i>Non-inline:</i>	<code>x.g(1)</code>	0.019	0.002	0.016	0.085	0
<i>Inline:</i>	<code>ps->k(1)</code>	0.007	0.002	0.006	0.005	0
<i>Macro:</i>	<code>K(ps,1)</code>	0.005	0.003	0.005	0.006	0
<i>Inline:</i>	<code>x.k(1)</code>	0.005	0.002	0.005	0.006	0
<i>Macro:</i>	<code>K(&s,1)</code>	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
<i>SI, non-virtual:</i>	<code>px->g(l)</code>	0.019	0.002	0.016	0.085	0
<i>Base1, non-virtual:</i>	<code>pc->g(i)</code>	0.007	0.003	0.016	0.007	0.004
<i>Base2, non-virtual:</i>	<code>pc->gg(i)</code>	0.007	0.004	0.017	0.007	0.028
<i>SI, virtual:</i>	<code>px->f(l)</code>	0.025	0.013	0.019	0.078	0.059
<i>Base1, virtual:</i>	<code>pa->f(i)</code>	0.026	0.012	0.019	0.082	0.059
<i>Base2, virtual:</i>	<code>pb->ff(i)</code>	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
<i>SI, non-virtual:</i>	<code>px->g(1)</code>	0.019	0.002	0.016	0.085	0
<i>VBC, non-virtual:</i>	<code>pd->gg(i)</code>	0.010	0.010	0.021	0.030	0.027
<i>SI, virtual:</i>	<code>px->f(1)</code>	0.025	0.013	0.019	0.078	0.059
<i>VBC, virtual:</i>	<code>pa->f(i)</code>	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
<i>Global:</i>	<code>h(1)</code>	0.014	0	0.013	0.071	0.001
<i>On base:</i>	<code>typeid(pa)</code>	0.079	0.047	0.218	0.365	0.059
<i>On derived:</i>	<code>typeid(pc)</code>	0.079	0.047	0.105	0.381	0.055
<i>On VBC:</i>	<code>typeid(pa)</code>	0.078	0.046	0.217	0.379	0.049
<i>VBC on derived:</i>	<code>typeid(pd)</code>	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
<i>Virtual call:</i>	<code>px->f(1)</code>	0.025	0.013	0.019	0.078	0.059
<i>Up-cast to base1:</i>	<code>cast(pa,pc)</code>	0.007	0	0.003	0.006	0
<i>Up-cast to base2:</i>	<code>cast(pb,pc)</code>	0.008	0	0.004	0.007	0.001
<i>Down-cast from base1:</i>	<code>cast(pc,pa)</code>	0.116	0.148	0.066	0.640	0.063
<i>Down-cast from base2:</i>	<code>cast(pc,pb)</code>	0.117	0.209	0.065	0.632	0.070
<i>Cross-cast:</i>	<code>cast(pb,pa)</code>	0.305	0.356	0.768	1.332	0.367
<i>2-level up-cast to base1:</i>	<code>cast(pa,pcc)</code>	0.005	0	0.005	0.006	0.001
<i>2-level up-cast to base2:</i>	<code>cast(pb,pcc)</code>	0.007	0	0.006	0.006	0.001
<i>2-level down-cast from base1:</i>	<code>cast(pcc,pa)</code>	0.116	0.148	0.066	0.641	0.063
<i>2-level down-cast from base2:</i>	<code>cast(pcc,pb)</code>	0.117	0.203	0.065	0.634	0.077
<i>2-level cross-cast:</i>	<code>cast(pa,pb)</code>	0.300	0.363	0.768	1.341	0.377
<i>2-level cross-cast:</i>	<code>cast(pb,pa)</code>	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 termination⁵. 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-blocks*** 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-expressions*** 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-specifications* 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-specifications* 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 *handlers* 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-specifications* 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 table⁷
- 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-specifications* 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-specifications* 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-parameters*. This can lead to an unexpectedly large

amount of code and data⁹. 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 objects¹⁰. Most of the functions were declared virtual, and so had to be overridden by derived classes¹¹. 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 contents¹².
- 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 ints will occupy approximately 4,000 bytes, whereas a list of 1,000 ints 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 container¹³.

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

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

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