Frontier Journal
High-tech Think Tank and Action Workbench

Frontier Editorial Board
Prof. Arvind, MIT  Prof. Michael J Flynn, Stanford  Prof. Masahiro Fujita, Univ. of Tokyo
Prof. Pankaj Jalote, IIT  Carl Pixley, Snopys  Prof. Wayne Wolf, Princeton  Michael McNamara, Cadence
Prof. Robert A. Walker, Kent State University  Prof. David S. Rosenblum, Univ. College London

Kluwer Academic Publisher Editorial
Frontier Leadership Interview Column
Featured Columnists - Brian Bailey, Dr. Danny Rittmen, Prof. Orit Hazzan
Featured Industrial Column – Analog Devices, ARM

Frontier Visionary Interview - Great Minds Think Alike, Really?

Prof. Robert J. Aumann, Hebrew University of Jerusalem, Nobel Laureate in Economics
Prof. A. Michael Spence, Stanford University, Nobel Laureate in Economics
Prof. Martin L. Perl, Stanford University, Nobel Laureate in Physics
Prof. Frank Wilczek, MIT, Nobel Laureate in Physics
Steve Wozniak, Co-founder, Apple Computer
Vinton G Cerf, PhD, Turing Award Winner, VP of Google
Ann Winblad, Co-founder, Hummer Winblad Venture Partners
Prof. David Patterson, UC Berkeley

Richard Stallman, Founder of GNU Project and Free Software Foundation
Jim Rogers, No. 1 Contrarian Investor in the World
Alan Kay, PhD, Viewpoints Research Institute, Turing Award Winner
Prof. Bjarne Stroustrup, Man behind C++, Texas A & M University
Brian Behlendorf, Co-founder of Apache Project, Former Chief Engineer of Wired Magazine
Rajeev Madhavan, Co-founder, Chairman and CEO, Magma Design Automation
Jimmy Wales, Founder of Wikipedia Project, Wikimedia Foundation and Wikia, Inc
Craig Newmark, Founder, chairman of Craigslist.org
Greg Gianforte, Founder and CEO of RightNow Technologies, Inc
Grady Booch, Chief Scientist, IBM Rational
Blake Ross, Co-creator of Firefox, Co-founder of Parakey
Prof. Tom Davenport, Barson College, Accenture Fellow
Prof. Ernest S. Kuh, UC Berkeley
Prof. Costas Markides, London Business School
Don Peppers, Co-founder, Peppers & Rogers Group
Aart de Geus, PhD, Co-founder, Chairman and CEO, Synopsys
Content

1. Frontier Kluwer Academic Publish Editorial – Leading Change under Leadership .................................................. 3

2. An Overview of the C++ Programming Language ............4


4. ARM Column - The ARM Architecture Version 6 (ARMv6) .............................................................................. 49

5. Learning Standard C++ as a New Language ..................64

6. Open Application Lifecycle Management - Unlocking the Full Value of Managed Software Delivery .............. 79

7. Why C++ is not just an Object-Oriented Programming Language ................................................................. 94

Copyrights © 2004 ~ 2007, Hometown Innovation Automation Inc (“Publisher”). All rights reserved. No part of System Design Frontier (“Publication”) may be reproduced, transmitted, or translated, in any form or by any means, electronics, mechanical, manual, optical, or otherwise, without prior written permission of Publisher.

Disclaimer Publisher makes no warranty or any kind, express or implied, with regard to Publication
Frontier Kluwer Academic Publisher Editorial

Leading Change Under Leadership

Editor-In-Chief

Today is 10-yr. anniversary of the return of Hong Kong to China. I have been to Hong Kong for a couple of times, and had impression that it is a great city similar to Shanghai, very clean, very big and very dense, though as big as those big 3 in the world, namely New York, LA, and Chicago. I still remember right 10 years ago in the evening, I was watching TV on live broadcasting the ceremony of handing over Hong Kong to China, with leaders including former China President Zheming Jiang and British Prince Charles. Times runs so fast, 10 years look like 10 days.

Four days ago in the early morning I interview Prof. John Kotter at Harvard Business School, the renown Changing and Leadership expert in the world, he wrote a couple of best sellers, including Leading Change, The Heart of Chang, and The Iceberg Is Melting among others. I had to call 3 numbers in that morning, I dialed the number he personally left to me through email, and there was a voice mail left by his assistant, and then I dialed the second number she left to me, a lady answered my phone, and she gave a 3rd number, finally I reached Prof. Kotter. I have to admit Prof. Kotter is a great talker as well as a great thinker. For detail on the phone interview, please check it u at http://www.hwswworld.com/interview.php or directly at http://www.hwswworld.com/uploaddownload/interview/kotter.mp3

Change is unpredictable, change is risky, and change is painful, and most immortally, is most of time, as I mentioned during the interview, we do not know what to change, when to change, how to change to minimize loss and maximize gain, and that requires leadership. Well, leading is an art instead of science; leaders are born to be leaders. So success typically demands luck beyond anything you can do by yourself
An Overview of the C++ Programming Language

Bjarne Stroustrup
AT&T Laboratories
Florham Park, NJ07932-0971, USA

ABSTRACT

This overview of C++ presents the key design, programming, and language-technical concepts using examples to give the reader a feel for the language. C++ is a general-purpose programming language with a bias towards systems programming that supports efficient low-level computation, data abstraction, object-oriented programming, and generic programming.

1 Introduction and Overview

The C++ programming language provides a model of memory and computation that closely matches that of most computers. In addition, it provides powerful and flexible mechanisms for abstraction; that is, language constructs that allow the programmer to introduce and use new types of objects that match the concepts of an application. Thus, C++ supports styles of programming that rely on fairly direct manipulation of hardware resources to deliver a high degree of efficiency plus higher-level styles of programming that rely on user-defined types to provide a model of data and computation that is closer to a human’s view of the task being performed by a computer. These higher-level styles of programming are often called data abstraction, object-oriented programming, and generic programming.

This paper is organized around the main programming styles directly supported by C++:

§2 The Design and Evolution of C++ describes the aims of C++ and the principles that guided its evolution.

§3 The C Programming Model presents the C subset of C++ and other C++ facilities supporting traditional systems-programming styles.

§4 The C++ Abstraction Mechanisms introduces C++’s class concept and its use for defining new types that can be used exactly as built-in types, shows how abstract classes can be used to provide interfaces to objects of a variety of types, describes the use of class hierarchies in object-oriented programming, and presents templates in support of generic programming.

§5 Large-Scale Programming describes namespaces and exception handling provided to ease the composition of programs out of separate parts.

§6 The C++ Standard Library presents standard facilities such as I/O streams, strings, containers (e.g. vector, list, and map), generic algorithms (e.g. sort(), find(), for_each()) and support for numeric computation.
To round off, a brief overview of some of the tasks that C++ has been used for and some suggestions for further reading are given.

2 The Design and Evolution of C++

C++ was designed and implemented by Bjarne Stroustrup (the author of this article) at AT&T Bell Laboratories to combine the organizational and design strengths of Simula with C’s facilities for systems programming. The initial version of C++, called “C with Classes” [Stroustrup,1980], was first used in 1980; it supported traditional system programming techniques (§3) and data abstraction (§4.1). The basic facilities for object-oriented programming (§4.2-4.3) were added in 1983 and object-oriented design and programming techniques were gradually introduced into the C++ community. The language was first made commercially available in 1985 [Stroustrup,1986] [Stroustrup,1986b]. Facilities for generic programming (§4.4) were added to the language in the 1987-1989 time frame [Ellis,1990] [Stroustrup,1991].

As the result of widespread use and the appearance of several independently-developed C++ implementations, formal standardization of C++ started in 1990 under the auspices of the American National Standards Institute, ANSI, and later the International Standards Organization, ISO, leading to an international standard in 1998 [C++,1998]. During the period of standardization the standards committee acted as an important focus for the C++ community and its draft standards acted as interim definitions of the language. As an active member of the standards committee, I was a key participant in the further evolution of C++. Standard C++ is a better approximation to my ideals for C++ than were earlier versions. The design and evolution of C++ is documented in [Stroustrup,1994] [Stroustrup,1996] and [Stroustrup,1997b]. The language as it is defined at the end of the standardization process and the key design and programming techniques it directly supports are presented in [Stroustrup,1997].

2.1 C++ Design Aims

C++ was designed to deliver the flexibility and efficiency of C for systems programming together with Simula’s facilities for program organization (usually referred to as object-oriented programming). Great care was taken that the higher-level programming techniques from Simula could be applied to the systems programming domain. That is, the abstraction mechanisms provided by C++ were specifically designed to be applicable to programming tasks that demanded the highest degree of efficiency and flexibility.

These aims can be summarized:

<table>
<thead>
<tr>
<th>Aims</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++ makes programming more enjoyable for serious programmers.</td>
</tr>
<tr>
<td>C++ is a general-purpose programming language that</td>
</tr>
<tr>
<td>- is a better</td>
</tr>
<tr>
<td>- supports data abstraction</td>
</tr>
</tbody>
</table>
Support for generic programming emerged late as an explicit goal. During most of the evolution of C++, I presented generic programming styles and the language features that support them (§4.4) under the heading of “data abstraction.”

2.2 Design Principles
In [Stroustrup, 1994], the design rules for C++ are listed under the headings General rules, Design-support rules, Language-technical rules, and Low-level programming support rules:

<table>
<thead>
<tr>
<th>General rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++’s evolution must be driven by real problems.</td>
</tr>
<tr>
<td>C++ is a language, not a complete system.</td>
</tr>
<tr>
<td>Don’t get involved in a sterile quest for perfection.</td>
</tr>
<tr>
<td>C++ must be useful now.</td>
</tr>
<tr>
<td>Every feature must have a reasonably obvious implementation.</td>
</tr>
<tr>
<td>Always provide a transition path.</td>
</tr>
<tr>
<td>Provide comprehensive support for each supported style.</td>
</tr>
<tr>
<td>Don’t try to force people.</td>
</tr>
</tbody>
</table>

Note the emphasis on immediate utility in real-world applications and the respect for the skills and preferences of programmers implied by the last three points. From the start, C++ was aimed at programmers engaged in demanding real-world projects. Perfection was considered unattainable because needs, backgrounds, and problems vary too much among C++ users. Also, notions of perfection change significantly over the lifespan of a general-purpose programming language. Thus, feedback from user and implementer experience is essential in the evolution of a language.

<table>
<thead>
<tr>
<th>Design-support rules:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Support sound design notions.</td>
</tr>
<tr>
<td>Provide facilities for program organization.</td>
</tr>
<tr>
<td>Say what you mean.</td>
</tr>
<tr>
<td>All features must be affordable.</td>
</tr>
<tr>
<td>It is more important to allow a useful feature than to prevent every misuse.</td>
</tr>
<tr>
<td>Support composition of software from separately developed parts.</td>
</tr>
</tbody>
</table>

The aim of C++ was to improve the quality of programs produced by making better design and programming techniques simpler to use and affordable. Most of these techniques have their
root in Simula [Dahl,1970] [Dahl,1972] [Birtwistle,1979] and are usually discussed under the labels of object-oriented programming and object-oriented design. However, the aim was always to support a range of design and programming styles. This contrasts to a view of language design that tries to channel all system building into a single heavily supported and enforced style (paradigm).

<table>
<thead>
<tr>
<th>Language-technical rules:</th>
</tr>
</thead>
<tbody>
<tr>
<td>No implicit violations of the static type system.</td>
</tr>
<tr>
<td>Provide as good support for user-defined types as for built-in types.</td>
</tr>
<tr>
<td>Locality is good.</td>
</tr>
<tr>
<td>Avoid order dependencies.</td>
</tr>
<tr>
<td>If in doubt, pick the variant of a feature that is easiest to teach.</td>
</tr>
<tr>
<td>Syntax matters (often in perverse ways).</td>
</tr>
<tr>
<td>Preprocessor usage should be eliminated.</td>
</tr>
</tbody>
</table>

These rules must be considered in the context created of the more general aims. In particular, the desire for a high degree of C compatibility, uncompromising efficiency, and immediate real-world utility counteracts desires for complete type safety, complete generality, and abstract beauty.

From Simula, C++ borrowed the notion of user-defined types (classes, §4.1) and hierarchies of classes (§4.3). However, in Simula and many similar languages there are fundamental differences in the support provided for user-defined types and for built-in types. For example, Simula does not allow objects of userdefined types to be allocated on the stack and addressed directly. Instead, all class objects must be allocated in dynamic memory and accessed through pointers (called references in Simula). Conversely, builtin types can be genuinely local (stack-frame allocated), cannot be allocated in dynamic memory, and cannot be referred to by pointers. This difference in treatment of built-in types and user-defined types had serious efficiency implications. For example, when represented as a reference to an object allocated in dynamic memory, a user-defined type – such as complex (§4.1) – incurs overheads in run-time and space that were deemed unacceptable for the kind of applications for which C++ was intended. Also, the difference in style of usage would preclude uniform treatment of semantically similar types in generic programming (§4.4).

When maintaining a large program, a programmer must invariably make changes based of incomplete knowledge and looking at only a small part of the code. Consequently, C++ provides classes (§4), namespaces (§5.2), and access control (§4.1) to help localize design decisions. Some order dependencies are unavoidable in a language designed for one-pass compilation. For example, in C++ a variable or a function cannot be used before it has been declared. However, the rules for class member names and the rules for overload resolution were made independent of declaration order to minimize confusion and error.
C++ was designed to be source-and-link compatible with C wherever this did not seriously interfere with C++’s support for strong type checking. Except for minor details, C++ has C [Kernighan,1978] [Kernighan, 1988] as a subset. Being C-compatible ensured that C++ programmers immediately had a complete language and toolset available. It was also important that high-quality educational materials were available for C, and that C compatibility gave the C++ programmer direct and efficient access to a multitude of libraries. At the time when the decision to base C++ on C was made, C wasn’t as prominent as it later became and language popularity was a minor concern compared to the flexibility and basic efficiency offered by C.

However, C compatibility also leaves C++ with some syntactic and semantic quirks. For example, the C declarator syntax is far from elegant and the rules for implicit conversions among built-in types are chaotic. It is also a problem that many programmers migrate from C to C++ without appreciating that radical improvements in code quality are only achieved by similarly radical changes to programming styles.

3 The C Programming Model

A fundamental property of computers in widespread use has remained remarkably constant: Memory is a sequence of words or bytes, indexed by integers called addresses. Modern machines – say, designed during the last 20 years – have in addition tended to support directly the notion of a function call stack. Furthermore, all popular machines have some important facilities – such as input-output – that do not fit well into the conventional byte- or word-oriented model of memory or computation. These facilities may require special machine instructions or access to “memory” locations with peculiar semantics. Either way, from a higher-level language point of view, the use of these facilities is messy and machine-architecture-specific.

C is by far the most successful language providing the programmer with a programming model that closely matches the machine model. C provides language-level and machine-architecture-independent notions that directly map to the key hardware notions: characters for using bytes, integers for using words, pointers for using the addressing mechanisms, functions for program abstraction, and an absence of constraining language features so that the programmer can manipulate the inevitable messy hardware-specific details. The net effect has been that C is relatively easy to learn and use in areas where some knowledge of the real machine is a benefit. Moreover, C is easy enough to implement that it has become almost universally available.

3.1 Arrays and Pointers
A C array is simply a sequence of memory locations. For example:

```c
int v[10]; // an array of 10 ints
v[3] = 1; // assign 1 to v[3]
int x = v[3]; // read from v[3]
```

The subscript notation [] is used both in declarations to indicate an array and in expressions referring to elements of an array.

A C pointer is a variable that can hold an address of a memory location. For example:

```c
int *p; // p is a pointer to an int
p = &v[7]; // assign the address of v[7] to p
*p = 4; // write to v[7] through p
int y = *p; // read from v[7] through p
```

The pointer dereference ("points to") notation * is used both in declarations to indicate a pointer and in expressions referring to the element pointed to.

This can be represented graphically:

```
0 1 2 3 4 5 6 7 8 9
```

C++ adopted this inherently simple and close-to-the-machine notion of memory from C. It also adopted C’s notion of expressions, control structures, and functions. For example, we can write a function that finds an element in a vector and returns a pointer to the matching element like this:

```c
int *find(int v[], int v_size, int val) // find val in v
{
    for (int i = 0; i < v_size; i++) // loop through 0..v.size-1
        if (v[i] == val) return &v[i]; // if val is found return pointer to element
    return &v[v.size]; // if not found return pointer to one-beyond-the-end of v
}
```

The ++ operator means increment. Thus, the name C++ can be read as “one more than C,” “next C,” or “successor to C.” It is pronounced “See Plus Plus.”

The `find()` function might be used like this:

```c
int count[] = { 2, 3, 1, 9, 7, 3, 0, 2 };
icount_size = 9;
```
```cpp
void f()
{
    int * p = find(count, count_size, 7); // find 7 in count
    int * q = find(count, count_size, 0); // find 0 in count
    *q = 4;
    // ...
}
```

The C++ standard library provides more general versions of functions such as `find()`; see §6.3. A function declared `void`, as `f()` above doesn’t return a value.

### 3.2 Storage

In C and C++, there are three fundamental ways of using memory:

- **Static memory**, in which an object is allocated by the linker for the duration of the program. Global variables, `static` class members, and `static` variables in functions are allocated in static memory. An object allocated in static memory is constructed once and persists to the end of the program. Its address does not change while the program is running. Static objects can be a problem in programs using threads (shared-address-space concurrency) because they are shared and require locking for proper access.

- **Automatic memory**, in which function arguments and local variables are allocated. Each entry into a function or a block gets its own copy. This kind of memory is automatically created and destroyed; hence the name automatic memory. Automatic memory is also said “to be on the stack.”

- **Free store**, from which memory for objects is explicitly requested by the program and where a program can free memory again once it is done with it (using the `new` and `delete` operators). When a program needs more free store, `new` requests it from the operating system. Typically, the free store (also called dynamic memory or the heap) grows throughout the lifetime of a program because no memory is ever returned to the operating system for use by other programs.

For example:

```cpp
int g = 7; // global variable, statically allocated
void f()
{
    int loc = 9; // local variable, stack allocated
    int * p = new int; // variable allocated on free store
    // ...
    delete p; // return variable pointed to by p for possible re-use
}
```
As far as the programmer is concerned, automatic and static storage are used in simple, obvious, and implicit ways. The interesting question is how to manage the free store. Allocation (using `new`) is simple, but unless we have a consistent policy for giving memory back to the free store manager, memory will fill up – especially for long-running programs.

The simplest strategy is to use automatic objects to manage corresponding objects in free store. Consequently, many containers are implemented as handles to elements stored in the free store. For example, a string (§6.1) variable manages a sequence of characters on the free store:

A `string` automatically allocates and frees the memory needed for its elements. For example:

```cpp
    void g ()
    {
        strings = "Time flies when you're having fun "; // string object created here
        // ...
    }                                                                   // string object implicitly destroyed here
```

The `Stack` example in §4.2.1 shows how constructors and destructors can be used to manage the lifetime of storage for elements. All the standard containers (§6.2), such as `vector`, `list`, and `map`, can be conveniently implemented in this way.

When this simple, regular, and efficient approach isn’t sufficient, the programmer might use a memory manager that finds unreferenced objects and reclaims their memory in which to store new objects. This is usually called **automatic garbage collection**, or simply **garbage collection**. Naturally, such a memory manager is called a **garbage collector**. Good commercial and free garbage collectors are available for C++ but a garbage collector is not a standard part of a typical C++ implementation.

### 3.3 Compile, Link, and Execute

Traditionally, a C or a C++ program consists of a number of source files that are individually compiled into object files. These object files are then linked together to produce the executable form of the program.

Each separately compiled program fragment must contain enough information to allow it to be linked together with other program fragments. Most language rules are checked by the compiler as it compiles an individual source file (translation unit). The linker checks to ensure that names are used consistently in different compilation units and that every name used actually refers to something that has been properly defined. The typical C++ runtime environment
performs few checks on the executing code. A programmer who wants run-time checking must provide the tests as part of the source code.

C++ interpreters and dynamic linkers modify this picture only slightly by postponing some checks until the first use of a code fragment. For example, I might write a simple factorial program and represent it as a separate source file f a c t . c :

```c
#include "fact.h"
long fact(long f) // recursive factorial
{
    if (f>1)
        return f*fact(f-1);
    else
        return 1;
}
```

A separately compiled program fragment has an interface consisting of the minimal information needed to use it. For this simple f a c t . c program fragment, the interface consists of the declaration of f a c t () stored in a file f a c t . h :

```c
#include "fact.h"
```

The interface is included in each translation unit that uses it. I also tend to include an interface into the translation unit that defines it to give the compiler a chance to diagnose inconsistencies early.

The f a c t () function can now be used like this:

```c
#include "fact.h"
#include <iostream>
int main()
{
    std::cout << "factorial(7) is " << fact(7) << \n;
    return 0;
}
```

The function main () is the starting point for a program, i o s t r e a m is the standard C++ I/O library, and std::cout is the standard character output stream (§6.1). The operator << ("put") converts values to character strings and outputs those characters. Thus, executing this program will cause factorial(7) is 5040 to appear on output followed by a newline (the special character \n).

Graphically, the program fragments can be represented like this:
long fact(long)  
#include "fact.h"
int main() {...}  
#include "fact.h"
long fact(longf) {...}

3.4 Type checking
C++ provides and relies on static type checking. That is, most language rules are checked by the compiler before a program starts executing. Each entity in a program has a type and must be used in accordance with its type. For example:

int f(double); // f is a function that takes a double-precision floating point  
// argument and returns an integer
float x = 2.0; // x is a single-precision floating point object
string s = "2"; // s is a string of characters
int i = f(x); // i is an integer

The compiler detects inconsistent uses and ensures that conversions defined in the language or by the user are performed. For example:

void g ()  
{
    s = "a string literal"; // ok: convert string literal to string
    s = 7; // error: can’t convert int to string
    x = "a string literal"; // error: can’t convert string literal to float
    x = 7.0; // ok
    x = 7; // ok: convert int to float
    f(x); // ok: convert float to double
    f(i); // ok: convert int to double
    f(s); // error: can’t convert string to double
    double d = f+i; // ok: add int to float
    string s2 = s +i; // error: can’t add int to string
}

For user-defined types, the user has great flexibility in defining which operations and conversions are acceptable (§6.1). Consequently, the compiler can detect inconsistent use of user-defined types as well as inconsistent use of built-in types.

4 Abstraction
In addition to convenient and efficient mechanisms for expressing computation and allocating objects, we need facilities to manage the complexity of our programs. That is, we need
language mechanisms for creating types that are more appropriate to the way we think (to our application domains) than are the low-level built-in features.

4.1 Concrete Types
Small heavily used abstractions are common in many applications. Examples are characters, integers, floating point numbers, complex numbers, points, pointers, coordinates, transforms, \((\text{pointer,offset})\) pairs, dates, times, ranges, links, associations, nodes, \((\text{value,unit})\) pairs, disc locations, source code locations, \(BCD\) characters, currencies, lines, rectangles, scaled fixed point numbers, numbers with fractions, character strings, vectors, and arrays. Every application uses several of these; a few use them heavily. A typical application uses a few directly and many more indirectly from libraries.

The designer of a general-purpose programming language cannot foresee the detailed needs of every application. Consequently, such as language must provide mechanisms for the user to define such small concrete types. It was an explicit aim of C++ to support the definition and efficient use of such user-defined data types very well. They were seen as the foundation of elegant programming. The simple and mundane is statistically far more significant than the complicated and sophisticated.

Many concrete types are frequently used and subject to a variety of constraints. Consequently, the language facilities supporting their construction emphasizes flexibility and uncompromising time and space efficiency. Where more convenient, higher-level, or safer types are needed, such types can be built on top of simple efficient types. The opposite – building uncompromisingly efficient types on top of more complicated “higher-level” types – cannot be done. Consequently, languages that do not provide facilities for efficient user-defined concrete types need to provide more built-in types, such as lists, strings, and vectors supported by special language rules.

A classical example of a concrete type is a complex number:


class complex {
    public: // interface:
        // constructors:
        complex(double r, double i) { r = r; i = i; } // construct a complex from two scalars
        complex(double r) { r = r; i = 0; } // construct a complex from one scalar
        complex() { r = i = 0; } // default complex: complex(0,0)
        // access functions:
        friend complex operator+(complex, complex);
        friend complex operator-(complex, complex); // binary minus
        friend complex operator-(complex); // unary minus
        friend complex operator*(complex, complex);
        friend complex operator/(complex, complex);
        // ...
    private:
        double r, i; // representation
    }
This defines a simple complex number type. Following Simula, the C++ term for user-defined type is class. This complex class specifies the representation of a complex number and the set of operations on a complex number. The representation is private; that is, re and im are accessible only to the functions specified in the declaration of class complex. Restricting the access to the representation to a specific set of functions simplify understanding, eases debugging and testing, and makes it relatively simple to adopt a different implementation if needed.

A member function with the same name as its class is called a constructor. Constructors are essential for most user-defined types. They initialize objects, that is, they establish the basic invariants that other functions accessing the representation can rely on. Class complex provides three constructors. One makes a complex from a double-precision floating-point number, another takes a pair of doubles, and the third makes a complex with a default value. For example:

```cpp
class a = complex (1,2);
class b = 3; // initialized by complex(3,0)
class c; // initialized by complex(0,0)
```

A friend declaration grants a function access the representation. Such access functions are defined just like other functions. For example:

```cpp
class operator+(complex a1, complex a2) // add two complex numbers
{
return complex(a1.re+a2.re, a1.im+a2.im);
}
```

This simple complex type can be used like this:

```cpp
void f()
{
complex a = 2.3;
complex b = 1/a;
complex c = a+b*complex(1,2.3);
// ...
c = -(a/b)+2;
}
```

The declaration of complex specified a representation. This is not necessary for a user-defined type (see §4.2). However, for complex, efficiency and control of data layout are essential. A simple class, such as complex, suffers no space overheads from system-provided “housekeeping” information. Because, the representation of complex is presented in its declaration, true local variables where all data is stack allocated are trivially implemented.
Furthermore, inlining of simple operations is easy for even simple compilers even in the presence of separate compilation. When it comes to supplying acceptable low-level types – such as complex, string, and vector – for high-performance systems these language aspects are essential [Stroustrup, 1994].

Often, notation is an important concern for concrete types. Programmers expect to be able to do complex arithmetic using conventional operators such as + and *. Similar programmers expect to be able to concatenate strings using some operator (often +), to subscript vectors and strings using [] or (), to invoke objects representing functions using (), etc. To meet such expectations, C++ provides the ability to define meanings of operators for user-defined types. Interestingly, the most commonly used operators and the most useful, turns out to be [] and (), rather than + and - as most people seem to expect.

The standard C++ library supplies a complex type defined using the techniques demonstrated here (§6.4.1).

4.2 Abstract Types
Concrete types, as described above, have their representation included in their declaration. This makes it trivial to allocate objects of concrete types on the stack and to inline simple operations on such objects. The resulting efficiency benefits are major. However, the representation of an object cannot be changed without recompiling code taking advantage of such optimizations. This is not always ideal. The obvious alternative is to protect users from any knowledge of and dependency on a representation by excluding it from the class declaration. For example:

```cpp
class Character_device {
  public:
    virtual int open(int opt) = 0; // =0 means "pure virtual function"
    virtual int close(int opt) = 0;
    virtual int read(char* p, int n) = 0;
    virtual int write(const char* p, int n) = 0;
    virtual int ioctl(int ...)=0;
    virtual ~Character_device() {} // destructor (see §4.2.1)
};
```

The word virtual means “may be defined later in a class derived from this one” in Simula and C++. A class derived from Character_device (see below) provides an implementation of Character_device interface. The curious =0 syntax says that some class derived from Character_device must define the function.

The Character_device is an abstract class specifying an interface only. This interface can be implemented in a variety of ways without affecting users. For example, a programmer might use this interface to device drivers on some hypothetical system like this:

```cpp
void user(Character_device * d, char * buffer, int size)
```

{
    char * p = buffer;
    while (size > chunk_size) {
        if (d->write(p, chunk_size) == chunk_size) { // whole chunk written
            size -= chunk_size; // chunk_size characters written
            p += chunk_size; // move on to next chunk
        } else { // part of chunk written
            // ...
        }
    } // ...
}

The actual drivers would be specified as classes derived from the base Character_device. For example:

class Dev1 : public Character_device {
    // representation of a Dev1
    public:
        int open(int opt); // open a Dev1
        int close(int opt); // close a Dev1
        int read(char * p, int n); // read a Dev1
        // ...
};
class Dev2 : public Character_device {
    // representation of a Dev2
    public:
        int open(int opt); // open a Dev2
        int close(int opt); // close a Dev2
        int read(char * p, int n); // read a Dev2
        // ...
};

The relationships among the classes can be represented graphically like this:

Character_device
Dev1 Dev2
base class:
derived classes:

An arrow represents the derived from relationship. The user () does not need to know whether a Dev1, or a Dev2, or some other class implementing the Character_device is actually used.
void f(Device &d1, Device &d2, char *buf, int s) {
    user(d1, buf, s); // use a Dev1
    user(d2, buf, s); // use a Dev2
}

A function declared in a derived class is said to override a virtual function with the same name and type in a base class. It is the language’s job to ensure that calls of Character_device’s virtual functions, such as write() invoke the appropriate overriding function for the derived class actually used. The overhead of doing that in C++ is minimal and perfectly predictable. The extra run-time overhead of a virtual function is a fraction of the cost of an ordinary function call.

We can represent the object layout of a typical implementation like this:

Thus, a virtual function call is simply an indirect function call. No run-time searching for the right function to call is needed.

In many contexts, abstract classes are the ideal way of representing the major internal interfaces of a system. They are simple, efficient, strongly typed, enable the simultaneous use of many different implementations of the concept represented by the interface, and completely insulate users from changes in such implementations.

4.2.1 Destructors
A constructor establishes a context for the member functions of a class to work in for a given object. Often, establishing that context requires the acquisition of resources such as memory, locks, or files. For a program to work correctly, such resources must typically be released when the object is destroyed. Consequently, it is possible to declare a function dedicated to reversing the effect of a constructor. Naturally, such a function is called a destructor. The name of a destructor for a class X is \texttt{\textasciitilde}X(); in C++, \texttt{\textasciitilde} is the complement operator.

A simple stack of characters can be defined like this:
class Stack {
    char *v;
    int max_size;
    int top;

public:
    Stack(int s) { top = 0; v = new T[max_size]; } // constructor: acquire memory
    ~Stack() { delete[] v; } // destructor: release memory
    void push(T c) { v[top++] = c; }
    T pop() { return v[--top]; }
};

For simplicity, this Stack has been stripped of all error handling. However, it is complete enough to be used like this:

void f(int n)
{
    Stack s2(n); // stack of n characters
    s2.push('a');
    s2.push('b');
    char c = s2.pop();
    // ...
}

Upon entry into f(), s2 is created and the constructor Stack::Stack() is called. The constructor allocates enough memory for n characters. Upon exit from f(), the destructor Stack::~Stack() is implicitly invoked so that the memory acquired by the constructor is freed.

This kind of resource management is often important. For example, an abstract class, such as Character_device, will be manipulated through pointers and references and will typically be deleted by a function that has no idea of the exact type of object used to implement the interface. Consequently, a user of a Character_device cannot be expected to know what is required to free a device. Conceivably freeing a device would involve nontrivial interactions with an operating system or other guardians of system resources. Declaring Character_device’s destructor virtual ensures that the removal of the Character_device is done using the proper function from the derived class. For example

void some_user(Character_device *pd)
{
    // ...
    delete pd; // implicitly invoke object’s destructor
}

4.3 Object-Oriented Programming
Object-oriented programming is a set of techniques that rely on hierarchies of classes to provide extensibility and flexibility. The basic language facilities used are the user-defined types themselves, the ability to derive a class from another, and virtual functions (§4.2). These features allow a programmer to rely on an interface (a class, often an abstract class) without knowing how its operations are implemented. Conversely, they allow new classes to be built directly on older ones without disturbing users of those older classes. As an example, consider the simple task of getting an integer value from a user to an application through some user-interface system. Assuming that we would like to keep the application independent of the details of the user-interface system we could represent the notion of an interaction needed to get an integer as a class `Ival_box`:

```cpp
class Ival_box {
public:
    virtual int get_value() = 0; // get value back to application
    virtual void prompt() = 0; // prompt the user
    // ...
};
```

Naturally, there will be a variety of `Ival_box`es:

```cpp
class Ival_dial: public Ival_box { /* ... */};
class Ival_slider: public Ival_box { /* ... */};
// ...;
```

This can be represented graphically like this:

```
Ival_box

Ival_dial  Ival_slider
```

This application hierarchy is independent of the details of an actual user-interface system. The application is written independently of I/O implementation details and then later tied into an implementation hierarchy without affecting the users of the application hierarchy.
A dashed arrow represents a protected base class. A protected base class is one that is part of 
the implementation of its derived class (only) and is inaccessible general user code. This design 
makes the application code independent of any change in the implementation hierarchy.

I have used the BB prefix for realism; suppliers of major libraries traditionally prepend some 
identifying initials. The superior alternative is to use namespaces (§5.2). The declaration of a 
class that ties an application class to the implementation hierarchy will look something like 
this:

```c
class BB_iVal_slider: public iVal_slider, protected BB_slider {
public:
    // functions overriding Ival_slider functions
    // as needed to implement the application concepts
    protected:
    // functions overriding BB_slider and BB_window functions
    // as required to conform to user interface standards
    private:
    // representation and other implementation details
};
```

This structure assumes that details of what is to be displayed by a user-interface system is 
expressed by overriding virtual functions in the BB_window s hierarchy. This may not be the 
ideal organization of a user interface system, but it is not uncommon.

A derived class inherits properties from its base classes. Thus, derivation is sometimes called inheritance. A language, such as C++, that allows a class to have more than one direct base class is said to support multiple inheritance.

### 4.3.1 Run-time Type Identification

A plausible use of the Ival_boxes defined in §4.3 would be to hand them to a system that controlled a screen and have that system hand objects back to the application program whenever some activity had occurred. This is how many user interfaces work. However, just as an application using Ival_boxes should not know about the user-interface system, the user-
interface system will not know about our *Ival_boxes*. The system’s interfaces will be specified in terms of the system’s own classes and objects rather than our application’s classes. This is necessary and proper. However, it does have the unpleasant effect that we lose information about the type of objects passed to the system and later returned to us.

Recovering the “lost” type of an object requires us to somehow ask the object to reveal its type. Any operation on an object requires us to have a pointer or reference of a suitable type for the object. Consequently, the most obvious and useful operation for inspecting the type of an object at run time is a type conversion operation that returns a valid pointer if the object is of the expected type and a null pointer if it isn’t. The `dynamic_cast` operator does exactly that. For example, assume that “the system” invokes `my_event_handler()` with a pointer to a *BB window*, where an activity has occurred:

```c
void my_event_handler(BB window *p)
{
  if (Ival_box *p = dynamic_cast<Ival_box*>(p)) { // does p point to an Ival_box?
    int i = p->get_value();
    // ...
  }
  else {
    // Oops! unexpected event
  }
}
```

One way of explaining what is going on is that `dynamic_cast` translates from the implementation-oriented language of the user-interface system to the language of the application. It is important to note what is *not* mentioned in this example: the actual type of the object. The object will be a particular kind of *Ival_box*, say an *Ival_slider*, implemented by a particular kind of *BB window*, say a *BB slider*. It is neither necessary nor desirable to make the actual type of the object explicit in this interaction between “the system” and the application. An interface exists to represent the essentials of an interaction. In particular, a well-designed interface hides inessential details.

Casting from a base class to a derived class is often called a *downcast* because of the convention of drawing inheritance trees growing from the root down. Similarly, a cast from a derived class to a base is called an *upcast*. A cast that goes from a base to a sibling class, like the cast from *BB window* to *Ival_box*, is called a *crosscast*.

### 4.4 Generic Programming

Given classes and class hierarchies, we can elegantly and efficiently represent individual concepts and also represent concepts that relate to each other in a hierarchical manner. However, some common and important concepts are neither independent of each other nor hierarchically organized. For example, the notions “vector of integer” and “vector of complex number” are related through the common concept of a vector and differ in the type of the
vector elements (only). Such abstractions are best represented through parameterization. For example, the `vector` should be parameterized by the element type.

C++ provides parameterization by type through the notion of a template. It was a crucial design criterion that templates should be flexible and efficient enough to be used to define fundamental containers with severe efficiency constraints. In particular, the aim was to be able to provide a `vector` template class that did not impose run-time or space overheads compared to a built-in array.

### 4.5 Containers

We can generalize a stack-of-characters type from §4.2.1 to a stack-of-anything type by making it a `template` and replacing the specific type `char` with a template parameter. For example:

```cpp
template <class T>
class Stack {
    T* v;
    int max_size;
    int top;

public:
    Stack(int s) { top = 0; v = new T[max_size = s]; } // constructor
    ~Stack() { delete[] v; } // destructor
    void push(T c) { v[top++] = c; }
    T pop() { v[--top]; }
};
```

The `template <class T>` prefix makes `T` a parameter of the declaration it prefixes.

We can now use stacks like this:

```cpp
Stack<char> sc(100); // stack of characters
Stack<complex> scp lx(200); // stack of complex numbers
Stack<list<int>> sl i(400); // stack of list of integers
void f() {
    sc.push('c');
    if(sc.pop() != 'c') error("impossible");
    scp lx.push(complex(1,2));
    if(scp lx.pop() != complex(1,2)) error("can't happen");
}
```

Similarly, we can define lists, vectors, maps (that is, associative arrays), etc., as templates. A class holding a collection of elements of some type is commonly called a `container class`, or simply a `container`. 
Templates are a compile-time mechanism so that their use incurs no run-time overhead compared to “hand-written code.”

4.5.1 Algorithms
Given a variety of semantically similar types – such as a set of containers that all support similar operations for element insertion and access – we can write code that works for all of those types. For example, we might count the occurrences of a value \( v a l \) in a sequence of elements delimited by \( \text{first} \) and \( \text{last} \) like this:

```cpp
template<class In, class T>
int count(In first, In last, const T& val) {
    int res = 0;
    while (first != last) if (*first++ == val) ++res;
    return res;
}
```

This code assumes only that values of type \( T \) can be compared using \( == \), that an \( In \) can be used to traverse the sequence by using \( ++ \) to get to the next element, and that \( *p \) retrieves the value of the element pointer to by an iterator \( p \). For example:

```cpp
void f(vector<complex>& vc, string s, list<int>& li) {
    int c1 = count(vc.begin(), vc.end(), complex(0));
    int c2 = count(s.begin(), s.end(), 'x');
    int c3 = count(li.begin(), li.end(), 42);
    // ...
}
```

This counts the occurrences of the complex 0 in the vector, the occurrences of \( x \) in the string, and the occurrences of 42 in the list.

A type with the properties specified for \( In \) is called an iterator. The simplest example of an iterator is a built-in pointer. Standard library containers – such as \( \text{vector} \), \( \text{string} \), and \( \text{list} \) – all provide the operations \( \text{begin}() \) and \( \text{end}() \) that returns iterators for the first element and the one-beyond-the-last element, respectively; thus, \( \text{begin}().\text{end}() \) describes a half-open sequence (§6.3). Naturally, the implementations of \( ++ \) and \( * \) differ for the different containers, but such implementation details don’t affect the way we write the code.

5 Large-scale Programming
The main part of this paper is organized around the key programming styles supported by C++. However, namespaces and exception handling mechanisms are important language features that do not fit this classification because they support large-scale programming in all styles. They are used to ease the construction of programs out of separate parts and they increase in importance as the size of programs increase.
5.1 Exceptions and Error Handling

Exceptions are used to transfer control from a place where an error is detected to some caller that has expressed interest in handling that kind of errors. Clearly, this is a mechanism that should be used only for errors that cannot be handled locally.

One might ask: “How can something that is (eventually) handled correctly so that the program proceeds as intended be considered an error?” Consequently, we often refer to exceptional events, or simply to exceptions, and the language mechanisms provided to deal with these events exception handling.

Consider how we might report underflow and overflow errors from a Stack:

```cpp
template <class T> class Stack {
  T* v;
  int max_size;
  int top;
  public:
    class Underflow { }; // type used to report underflow
    class Overflow { }; // type used to report overflow
    Stack(int); // constructor
    ~Stack(); // destructor
    void push(T c) {
      if (top == max_size) throw Overflow(); // check for error
      v[top++] = c; // raise top and place c on top
    }
    T pop() {
      if (top == 0) throw Underflow(); // check for error
      return v[--top]; // lower top
    }
};
```

An object thrown can be caught by a caller of the function that threw. For example:

```cpp
void f() {
  Stack<string> ss(10);
  try {
    ss.push("Quiz");
    string s = ss.pop(); // pop "Quiz"
    ss.pop(); // try to pop an empty string: will throw Underflow
  }
  catch (Stack<string>::Underflow) { // exception handler
    
  }
}```
cerr << "error : Stack underflow";
return;
}
catch (Stack <string>::Overflow) { // exception handler
cerr << "error : Stack overflow";
return;
} // ...
}

Exceptions thrown within a try { ... } block or in code called from within a try block will be caught by a catch-clause for their type.

Exceptions can be used to make error handling more stylized and regular. In particular, hierarchies of exception classes can be used to group exceptions so that a piece of code need deal only with errors at a suitable level of detail. For example, assume that open_file_error is a class derived from io_error:

```cpp
void use_file (const char * fn)
{
    File_ptr fp (fn,"r"); // open fn for reading; throw open_file_error if that can't be done
    // use fp
}
```

Here, some_fct need not know the details of what went wrong; that is, it need not know about open_file_error. Instead, it deals with errors at the io_error level of abstraction.

Note that use_file didn't deal with exceptions at all. It simply opens a file and uses it. Should the file not be there to open, control is immediately transferred to the caller some_fct. Similarly, should a read error occur during use, the file will be properly closed by the File_ptr destructor before control returns to some_fct.
5.2 Namespaces

A namespace is a named scope. Namespaces are used to group related declarations and to keep separate items separate. For example, two separately developed libraries may use the same name to refer to different items, but a user can still use both:

```cpp
namespace Mylib {
    template <class T> class Stack { /* ... */ };
    // ...
}
namespace Yourlib {
    class Stack { /* ... */ };
    // ...
}
void f(int max)
{
    Mylib::Stack<int> s1(max); // use my stack
    Yourlib::Stack s2(max); // use your stack
    // ...
}
```

Repeating a namespace name can be a distraction for both readers and writers. Consequently, it is possible to state that names from a particular namespace are available without explicit qualification. For example:

```cpp
void f(int max)
{
    using namespace Mylib; // make names from Mylib accessible
    Mylib::Stack<int> s1(max); // use my stack
    Yourlib::Stack s2(max); // use your stack
    // ...
}
```

Namespaces provide a powerful tool for the management of different libraries and of different versions of code. In particular, they offer the programmer alternatives of how explicit to make a reference to a nonlocal name.

6 The C++ Standard Library

The standard library provides:

1. Basic run-time language support (e.g., for allocation and run-time type information).
2. The C standard library (with very minor modifications to minimize violations of the type system).
3. Strings and I/O streams (with support for international character sets and localization).
4. A framework of containers (such as vector, list, and map) and algorithms using containers (such as general traversals, sorts, and merges).
[5] Support for numerical computation (complex numbers plus vectors with arithmetic operations, BLAS-like and generalized slices, and semantics designed to ease optimization).

The main criterion for including a class in the library was that it would somehow be used by almost every C++ programmer (both novices and experts), that it could be provided in a general form that did not add significant overhead compared to a simpler version of the same facility, and that simple uses should be easy to learn. Essentially, the C++ standard library provides the most common fundamental data structures together with the fundamental algorithms used on them.

The framework of containers, algorithms, and iterators is commonly referred to as the STL. It is primarily the work of Alexander Stepanov [Stepanov, 1994].

6.1 Strings and I/O

Strings and input/output operations are not provided directly by special language constructs in C++. Instead, the standard library provides string and I/O types. For example:

```cpp
#include <string>  // make standard strings available
#include <iostream>  // make standard I/O available

int main()
{
    using namespace std;
    string name;
    cout << "Please enter your name: ";  // prompt the user
    cin >> name;  // read a name
    cout << "Hello, \n" << name;  // output the name followed by a newline
    return 0;
}
```

This example uses the standard input and output streams, `cin` and `cout` with their operators `>>` (‘‘get from’’) and ‘‘<< (‘‘put to’’).

The I/O streams support a variety of formatting and buffering facilities, and strings support common string operations such as concatenation, insertion, and extraction of characters and strings. Both streams and strings can be used with characters of any character set.

The standard library facilities can be – and usually are – implemented using only facilities available to all users. Consequently, where the standard facilities happen to be inadequate, a user can provide equally elegant alternatives.

6.2 Containers

The standard library provides some of the most general and useful container types to allow the programmer to select a container that best serves the needs of an application:
These containers are designed to be efficient yet have interfaces designed to ensure that they can be used interchangeably wherever reasonable. For example, like `list`, `vector` provides efficient operations for adding elements to its end (back). This allows data that eventually needs to be efficiently accessed by subscripting to be constructed incrementally. For example:

```cpp
vector<Point> cities;
void add_points(Point sentinel)
{
    Point buf;
    while (cin >> buf) { // read points from input
        if (buf == sentinel) return; // check new point
        cities.push_back(buf); // add point to (end of) vector
    }
}
```

The containers are nonintrusive; that is, essentially every type can be used as a container element type. In particular, built-in types – such as `int` and `char*` – and C-style data structures (`struct`s) can be container elements.

### 6.3 Algorithms

The standard library provides dozens of algorithms. The algorithms are defined in namespace `std` and presented in the `<algorithm>` header. Here are a few I have found particularly useful:

<table>
<thead>
<tr>
<th>Selected Standard Algorithms</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for_each()</code></td>
</tr>
<tr>
<td><code>find()</code></td>
</tr>
<tr>
<td><code>find_if()</code></td>
</tr>
<tr>
<td><code>count()</code></td>
</tr>
<tr>
<td><code>count_if()</code></td>
</tr>
</tbody>
</table>

Invoke function for each element
Find first occurrence of arguments
Find first match of predicate
Count occurrences of element
Count matches of predicate
These algorithms, and many more, can be applied to of standard containers, string s, and built-in arrays. In fact, they can be applied to any sequence as described in §4.5.1:

As mentioned, operator * is used to mean ‘‘access an element through an iterator’’ and operator ++ to mean ‘‘make the iterator refer to the next element.’’ For example, we can define a general algorithm that finds the first element of a sequence that matches a predicate like this:

```cpp
template<class In, class Predicate>
In find_if(In first, In last, Predicate pred)
{
    while (first!=last && !pred(*first)) ++first;
    return first;
}
```

Simple definitions, the ability to chose the right algorithm at compile time based on the type of the input sequence, and the ability to inline simple operations (such as ==, <, and simple user-defined predicates) mean that these very generic and general algorithms outperform most conventional alternatives.

6.4 Numerics
Like C, C++ wasn’t designed primarily with numerical computation in mind. However, a lot of numerical work is done in C++, and the standard library reflects that.

6.4.1 Complex Numbers
The standard library supports a family of complex number types along the lines of the complex class described in §4.1. To support complex numbers where the scalars are single-precision floating-point numbers (floats), double precision numbers (doubles), etc., the standard library complex is a template:

```cpp
template<class scalar> class complex {
public:
    complex(scalar real, scalar imag);
};
```
// ...
};

The usual arithmetic operations and the most common mathematical functions are supported for complex numbers. For example:

template <class C> complex <C> pow (const complex <C>&, int ); // exponentiation

void f (complex <float> fl, complex <double> db)
{
    complex <long double> ld = fl + sqrt (db);
    db += fl * 3;
    fl = pow (1/fl, 2);
    // ...
}

Thus, complex numbers are supported approximately to the degree that floating-point numbers are.

6.4.2 Vector Arithmetic

The standard vector from §6.2 was designed to be a general mechanism for holding values, to be flexible, and to fit into the architecture of containers, iterators, and algorithms. However, it does not support mathematical vector operations. Adding such operations to vector would be easy, but its generality and flexibility precludes optimizations that are often considered essential for serious numerical work. Consequently, the standard library provides a vector, called valarray, that is less general and more amenable to optimization for numerical computation:

```cpp
template <class T> class valarray {
    // ...
    T& operator [] (size_t);
    // ...
};
```

The type size_t is the unsigned integer type that the implementation uses for array indices. The usual arithmetic operations and the most common mathematical functions are supported for valarrays. For example:

```cpp
template <class T> valarray <T> abs (const valarray <T>&); // absolute value

void f (const valarray <double>& al, const valarray <double>& a2)
{
    valarray <double> a = al * 3.14 + a2 / al;
    a += a2 * 3.14;
    valarray <double> a a = a * a (a);
```
double d = a[7];
// ...
}

The `valarray` type also supports BLAS-style and generalized slicing. More complicated numeric types, such as `Matrix`, can be constructed from `valarray`.

**7 Use of C++**

C++ is used by hundreds of thousands of programmers in essentially every application domain. This use is supported by about a dozen independent implementations, hundreds of libraries, hundreds of textbooks, several technical journals, many conferences, and innumerable consultants. Training and education at a variety of levels are widely available.

Early applications tended to have a strong systems programming flavor. For example, several major operating systems have been written in C++ [Campbell,1987] [Rozier,1988] [Hamilton,1993] [Berg,1995] [Parrington,1995] and many more have key parts done in C++. C++ was designed so that every language feature is usable in code under severe time and space constraints [Stroustrup,1994]. This allows C++ to be used for device drivers and other software that rely on direct manipulation of hardware under real-time constraints. In such code, predictability of performance is at least as important as raw speed. Often, so is compactness of the resulting system.

Most applications have sections of code that are critical for acceptable performance. However, the largest amount of code is not in such sections. For most code, maintainability, ease of extension, and ease of testing is key. C++’s support for these concerns has led to its widespread use where reliability is a must and in areas where requirements change significantly over time. Examples are banking, trading, insurance, telecommunications, and military applications. For years, the central control of the U.S. long-distance telephone system has relied on C++ and every 800 call (that is, a call paid for by the called party) has been routed by a C++ program [Kamath,1993]. Many such applications are large and long-lived. As a result, stability, compatibility, and scalability have been constant concerns in the development of C++. Million-line C++ programs are not uncommon.

Like C, C++ wasn’t specifically designed with numerical computation in mind. However, much numerical, scientific, and engineering computation is done in C++. A major reason for this is that traditional numerical work must often be combined with graphics and with computations relying on data structures that don’t fit into the traditional Fortran mold [Budge,1992] [Barton,1994]. Graphics and user interfaces are areas in which C++ is heavily used.

All of this points to what may be C++’s greatest strength: its ability to be used effectively for applications that require work in a variety of application areas. It is quite common to find an application that involves local and wide-area networking, numerics, graphics, user interaction, and database access. Traditionally, such application areas have been considered distinct, and they have most often been served by distinct technical communities using a variety of
programming languages. However, C++ has been widely used in all of those areas. Furthermore, it is able to coexist with code fragments and programs written in other languages.

C++ is widely used for teaching and research. This has surprised some who – correctly – point out that C++ isn’t the smallest or cleanest language ever designed. However, C++ is

– clean enough for successful teaching of basic concepts,
– realistic, efficient, and flexible enough for demanding projects,
– available enough for organizations and collaborations relying on diverse development and execution environments,
– comprehensive enough to be a vehicle for teaching advanced concepts and techniques, and
– commercial enough to be a vehicle for putting what is learned into non-academic use.

Thanks to the ISO standards process (§2), C++ is also well-specified, stable, and supported by a standard library.

8 Further Reading
There is an immense amount of literature on C++, Object-oriented Programming, and Object-Oriented Design. Here is a short list of books that provides information on key aspects of C++ and its use.

[Stroustrup,1997] is a tutorial for experienced programmers and user-level reference for C++ and its standard library; it presents a variety of fundamental and advanced design and programming techniques. [Stroustrup,1994] describes the rationale behind the design choices for C++.

[Koenig,1997] is a collection of essays discussing ways of using C++ effectively. [Barton,1994] focuses on numeric computation and presents some advanced uses of templates. [Cline,1995] gives practical answers to many questions that occur to programmers starting to use C++.

Other books discuss C++ primarily in the context of design. [Booch,1994] presents the general notion of Object-Oriented Design, and [Martin,1995] gives detailed examples of Booch’s design method, [Gamma,1994] introduces the notion of design patterns. These three books all provide extensive examples of C++ code.

These books are all aimed at experienced programmers and designers. There is also a host of C++ books aimed at people with weak programming experience, but the selection of those varies so rapidly that a specific recommendation would be inappropriate.

9 Acknowledgements
This paper was written in grateful memory of my CRC Standard Mathematical Tables, 17th edition: an essential tool, status symbol, and security blanket for a young Math student.

10 References


Analog Devices Column

Reducing Ground Bounce in Dc-to-DC Converters – Some Grounding Essentials

Jeff Barrow

Electrical ground looks simple on a schematic; unfortunately, the actual performance of a circuit is dictated by its printed-circuit-board (PCB) layout. What’s more, ground-node analysis is difficult, especially for dc-to-dc converters, such as buck and boost circuits, which pound the ground node with large, fast-changing currents. When the ground node moves, system performance suffers and the system radiates EMI. But a well-“grounded” understanding of the physics of ground noise can provide an intuitive sense for reducing the problem.

Ground bounce can produce transients with amplitudes of volts; most often changing magnetic flux is the cause. A loop of wire carrying current is essentially an electromagnet whose field strength is proportional to the current. Magnetic flux is proportional to the magnetic field passing through the loop area,

Magnetic Flux $\propto$ Magnetic Field $\times$ Loop Area

or more precisely,

$$\Phi_B = BA \cos \phi$$

Where the magnetic flux, $\Phi_B$, is the magnetic field, B, passing through a surface loop area, A, at an angle, $\phi$, to the area’s unit vector.

A look at Figure 1 gives meaning to the magnetic flux associated with an electric current. A voltage source pushes current through a resistor and around a loop of wire. This current is associated with magnetic flux encircling the wire. To relate the different quantities, think of grabbing the wire with your right hand (applying the right-hand rule). If you point your thumb in the direction of current flow, your fingers will wrap around the wire in the direction of the magnetic field lines. As those field lines pass through the loop, their product is magnetic flux, directed in this case into the page.
Change either the magnetic field strength or the loop area, and the flux will change. As the flux changes, a voltage is induced in the wire, proportional to the rate of change of the flux, $\frac{d\Phi_B}{dt}$. Notice that either a fixed loop and changing current or a constant current and a changing loop area—or both—will change the flux.

Suppose, for example, that the switch in Figure 2 is suddenly opened. When current stops flowing, the magnetic flux collapses, which induces a momentarily large voltage everywhere along the wire. If part of the wire is a ground return lead, voltage that is supposed to be at ground will spike, thus producing false signals in any circuitry using it as a ground reference.

Generally, voltage drops in printed-circuit-board sheet resistance are not a major source of ground bounce. 1-oz copper has a resistivity of about 500 $\mu$ohm/square, so a 1-A change in
current produces a bounce of 500 µV/square—a problem only for thin, long, or daisy-chained grounds, or precision electronics.

Charging and discharging of parasitic capacitors provides a path for large transient currents to return to ground. The change in magnetic flux from those changing currents induces ground bounce.

The best way to reduce ground bounce in a switching dc-to-dc converter is to control changes in magnetic flux—by minimizing both current loop areas and changes in loop area.

In some cases, as in Figure 3, the current remains constant, but the switching produces a change of loop area, hence a change of flux. In switch Case 1, an ideal voltage source is connected by ideal wires to an ideal current source. Current flows in a loop that includes a ground return.

In Case 2, when the switch changes position, the same current flows in a different path. The current source is dc and does not change, but loop area does change. The change in loop area means a change in magnetic flux, so voltage is induced. Since a ground return is part of that changing loop, its voltage will bounce.
Figure 3. Buck Converter Ground Bounce
For the purpose of discussion, the simple circuit in Figure 3 is similar to—and can be morphed into—the buck converter in Figure 4.

![Buck Converter Diagram]

Figure 4. To a high frequency switch, an enormous $C_{VIN}$ and $L_{BUCK}$ look like a voltage and current source.

At high frequencies, a large capacitor—such as the buck input capacitor, $C_{VIN}$—looks like a dc voltage source. Similarly, the large output buck inductor, $L_{BUCK}$, looks like a dc current source. These approximations are made to help foster intuition.

Figure 5 displays how magnetic flux changes as the switch alternates between the positions.

![Switching Effect Diagram]

Figure 5. The effect of switching on loop area.

The large $L_{BUCK}$ inductor holds the output current roughly constant. Similarly, $C_{VIN}$ maintains a voltage approximately equal to $V_{IN}$, so the input current is also more or less constant due to the unchanging voltage across the input lead inductance.
Although the input and output currents are roughly constant, as the switch moves from Position 1 to Position 2, the total loop area rapidly changes in the middle portion of the circuit. That change means a rapid change in magnetic flux, which in turn induces ground bounce along the return wire.

Actual buck converters are made with pairs of semiconductor switches, as shown in Figure 6. Although the complexity has increased with each figure, the analysis of ground bounce induced by changing magnetic flux remains simple and intuitive.

![Figure 6. The basic principles are unchanged with semiconductor switching.](image)

The fact that a change in magnetic flux will induce voltage everywhere along a ground return brings up the interesting question: where is true ground? Because ground bounce means a voltage on the ground return trace is bouncing with respect to some ideal point called ground, that point needs to be identified.

In the case of power-regulating circuits, true ground needs to be at the low end of the load. After all, a dc-to-dc converter’s purpose is to deliver quality voltage and current to the load. All other points along the current return are not ground, just part of the ground return.

Since ground is at the low end of the load, and since changing loop area is the cause for ground bounce, Figure 7 shows how careful placement of $C_{VIN}$ reduces ground bounce by reducing the portion of loop area that changes.
Figure 7. Careful placement of CVIN greatly reduces ground bounce.

Capacitor $C_{VIN}$ bypasses the top of the high-side switch directly to the bottom of the low side switch, thereby shrinking the changing loop area and isolating it from the ground return. From the bottom of $V_{IN}$ to the bottom of the load, no loop-area or switch-current changes occur from one case to the next. Consequently, the ground return does not bounce.

Figure 8. A bad layout results in a large change in current loop area from one switch case to the next.

The PCB layout itself actually determines the performance of the circuit. Figure 8 is a PCB layout of the buck schematic in Figure 6. In the switch position shown in Case 1, with the high-side switch on, dc flow follows the outer red loop. In the switch position shown in Case 2, with the low-side switch on, dc flow now follows the blue loop. Notice the changing loop area, and hence, the changing magnetic flux. So, voltage is induced and the ground bounces.

The layout is realized on a single PCB layer for clarity, but using a second layer of solid ground plane would not fix the bounce. Before showing an improved layout, Figure 9 gives a quick example of where a solid ground plane may not be such a good idea.
Figure 9. A solid ground plane is not always a good idea.

Here, a 2-layer PCB is constructed so that a bypass capacitor is attached at right angles to a top-layer supply line. In the example on the left, the ground plane is solid and uncut. Top trace current flows through the capacitor, down the via, and out the ground plane.

Because ac always takes the path of least impedance, ground return current rounds the corner on its way back to the source. So the current’s magnetic field and the associated loop area change when either magnitude or frequency of the current changes, hence the changing flux. The tendency of current to flow along the easiest path means that even a solid-sheet ground plane can bounce—irrespective of its conductivity.

In the example on the right, a well-planned cut in the ground plane will constrain the return current to a minimum loop area and greatly reduce the bounce. Any residual bounce voltage that is developed in the cut return line is isolated from the general ground plane.

The PCB layout in Figure 10 uses the principle illustrated in Figure 9 to reduce ground bounce. A 2-layer PCB is designed so that the input capacitor and both switches are built over an island in the ground plane.

This layout is not necessarily the best, but it works well and illustrates a key principle. Notice that the loop area enclosed by the red (Case 1) and blue (Case 2) currents is large. However, the difference between the two loops is small. The small change in loop area means a small change in magnetic flux—and so, a small ground bounce. (In general, however, also keep the loop area small—this figure strives to illustrate the importance of matching ac current paths.)

Additionally, in the ground-return island, where magnetic fields and loop area do change, any ground-return bounce is contained by the cut.

Also of interest, the input capacitor, $C_{VIN}$, may not at first glance appear to be located between the top of the high-side switch and the bottom of the low-side switch, as discussed in Figure 7, but closer perusal will reveal that it is. Although physical proximity can be good, what really matters is the electrical closeness that is achieved by minimizing the area of the loop.
Figure 10. A good buck layout has a small change in loop area as between Case 1 and Case 2.

Boost Converter Ground Bounce

A *boost converter* is essentially a reflection of a buck converter, so—as seen in Figure 11—it is the *output* capacitor that must be placed between the top of the high-side switch and the bottom of the low-side switch to minimize the change in loop area.
Figure 11. Boost converter means CVOUT placement is critical in the same way that buck converter’s CVIN placement is critical. a) Bad design. b) Good design.

Review
Ground-bounce voltage is induced principally by a change in magnetic flux. In a dc-to-dc switching power supply, the flux changes because high speed switches direct current between different current-loop areas. But careful placement of the buck/boost input/output capacitor and a surgical cut to a ground plane can isolate bounce. However, it is important to be watchful when cutting a ground plane, to avoid possibly increasing the loop area for some other return current in the circuit.

Also, a good layout locates true ground at the bottom of the load, with no changing loop areas or changing currents. Any other conductively associated point may be called “ground,” but it is just a point along the return path.

Other Useful Concepts for Ground Analysis
If you keep the following basic ideas in mind, you’ll have a good feeling for what will and will not cause ground bounce. Figure 12 shows that conductors that cross at a right angle do not suffer magnetic interaction.
Figure 12. Conductors that cross at a right angle do not interact magnetically.

Magnetic field lines around parallel wires carrying equal currents flowing in the same direction cancel everywhere between the wires, so the total stored energy is less than what would be found for the individual wires. For this reason, wide PCB traces have less inductance than narrow traces.

Figure 13. Parallel wires with currents flowing in the same direction.

Magnetic field lines around parallel conductors carrying equal currents flowing in opposite directions cancel everywhere outside of the conductors and add everywhere between them. If the inside loop area can be made small, then the total magnetic flux, and therefore the
inductance, will also be small. This behavior explains why ac ground plane return current always flows under the top trace conductor.

**Figure 14. Parallel conductors with currents flowing in opposite directions.**

Figure 15 shows why corners increase inductance. A straight conductor sees its own magnetic field, but at a corner, it also sees the magnetic field from the right-angled conductor. As a result, corners store more magnetic energy, and so, have more inductance than straight lines.

**Figure 15. Why corners increase inductance.**

Figure 16 shows that interruptions to the ground plane under conductors carrying current can increase loop area by diverting the return current, thus increasing loop size and facilitating ground bounce.
Figure 16. Return current takes the path of least impedance.

Component orientation does matter, as shown in Figure 17.

Figure 17. Effects of component orientation.

Summary
Ground bounce is always a potential problem. For a monitor or TV, it can mean a noisy picture—for an audio device, background noise. In a digital system, it can lead to computation errors—even a system crash.

A careful estimation of parasitic elements followed by detailed simulation is a rigorous way to predict the magnitude of ground bounce. But to guide circuit-design intuition, it is necessary to understand the physics underlying its origin.

First, design the PCB so that the low end of the load is the true ground point.
Then, simplify the circuit dynamics by replacing large inductors and capacitors with current- and voltage sources. Look for the current loops in each switching combination. Make the loops overlap; where that is impossible, carefully cut out a small island of ground return such that only dc flows into and out of the opening.

In most cases, these efforts will give acceptable ground performance. If they don’t, consider ground-plane resistance, then the displacement currents flowing in parasitic capacitors across all switches and down into the return path.

No matter what the circuit, the basic grounding principles are the same—changing magnetic flux needs to be minimized and/or isolated.

ENDNOTE

ARM Column

The ARM Architecture Version 6 (ARMv6)

David Brash
Architecture Program Manager, ARM Ltd.

A microprocessor’s architecture defines the instruction set and programmer’s model for any processor that will be based on that architecture. Different processor implementations may be built to comply with the architecture. Each processor may vary in performance and features, and be optimized to target different applications.

Future processors, based on the new ARMv6 architecture will provide developers of embedded systems with higher levels of system performance, whilst maintaining excellent power and area efficiency.

The Evolution of the ARM Architecture
The ARM architecture has evolved steadily to respond to the changing needs of ARM’s partners, and of the design community in general. At each major revision of the ARM architecture, significant features have been added.

Between major architecture revisions, new features have been included as variants on the architectures. The key letters appended to the core names indicate specific architecture enhancements within each implementation.

- V3 introduced 32-bit addressing, and architecture variants:
  - M – long multiply support (32 x 32 => 64 or 32 x 32 + 64 => 64). This feature became standard in architecture V4 onwards.
- V4 added halfword load and store.
- V5 improved ARM and Thumb interworking, count leading-zeroes (CLZ) instruction, and architecture variants:
  - E – enhanced DSP instructions including saturated arithmetic operations and 16-bit multiply operations
  - J – support for new Java state, offering hardware and optimized software acceleration of bytecode execution.
All of the ‘TEJ’ enhancements above become part of the new ARMv6 architecture specification.

In order to maintain backwards compatibility, ARMv6 also includes ARMv5 compliant memory management and exception handling. This enables the significant third-party developer community to exploit existing development effort, and supports the reuse of existing software and design experience.

The introduction of a new architecture does not replace existing architectures, or make them redundant. Where the provisions of ARMv4 or ARMv5 meet market needs, new cores and derivative products will continue to be based on these architectures, whilst tracking technology and process trends. For example, the ARM7TDMI core based on the V4T architecture is still being ‘designed-in’ to many new products, where a performance level of 100MIPS or so is adequate. Processors based on the ARMv5 architecture continue in development.

The ARM architecture will of course continue to evolve with appropriate enhancements in the future.

**Figure 1. ARM Architecture Revisions**

Implementations of the ARMv6 architecture are primarily driven by ARM’s partner development activity. The first ARM implementations of ARMv6 are underway; more information will be released with the product rollout during 2002.

**Driving Architecture Development**

Next generation architectures have been driven by the needs of emerging products and evolving markets. The key design constraints are predictable. The function, performance, speed, power,
area and cost parameters must be balanced to meet the requirements of each application. ARMv6 offers better ways of optimizing these constraints across a number of vertical market segments.

Delivering leading performance/power (MIPS/Watt) has been fundamental to ARM’s success in the past, and will continue to be a critical benchmark for future applications.

Functionality is growing dramatically as computing and communications continue to converge in many consumer products. Increasingly, consumers expect features such as advanced user interfaces, multimedia capability and improved product quality. ARMv6 will enable more efficient support for all of these new features and technologies across a number of market segments.

A number of specific market drivers for ARMv6 have been identified. ARMv6 will benefit developers targeting wireless, networking, automotive and consumer entertainment markets. ARM has worked with architecture licensees and key partners such as Intel, Microsoft, Symbian and Texas Instruments in specifying the requirements for ARMv6.

As well as taking into account changing market requirements, key improvements in software, synthesis and process technology also influence the architecture specification. The development of ARMv6 will enable partners to better exploit these, and other technological advances.

**Key ARMv6 Improvements**

In developing the ARMv6 architecture, effort has been focused on five key areas:

**Memory Management**

System design and performance is heavily affected by the way that memory is managed. The memory management architectural enhancements improve the overall processor performance significantly – especially for platform-type applications where operating systems need to manage frequent task changes. With the changes in ARMv6, average instruction fetch and data latency is greatly reduced; the processor has to spend less time waiting for instructions or data cache misses to be loaded. The memory management improvements will provide a boost in overall system performance by as much as 30%. In addition, the memory management enhancements will enable more efficient bus usage. Less bus activity will yield significant power savings as a result of reduced memory access.

**Multiprocessing**

Application convergence is driving system implementations towards the need for multiprocessor systems. Wireless platforms, especially for 2.5G and 3G, are typical applications that demand integration between ARM processors, ARM and DSPs, or other application accelerators.

Multiprocessor systems share data efficiently by sharing memory. New ARMv6 capabilities in data sharing and synchronization will make it easier to implement multiprocessor systems, as
well as improving their performance. New instructions enable more complex synchronization schemes, greatly improving system efficiency.

**Multimedia Support**
Single Instruction Multiple Data (SIMD) capabilities enable more efficient software implementation of high-performance media applications such as audio and video encoders. Over sixty SIMD instructions are added to the ARMv6 Instruction Set Architecture (ISA).

Adding the SIMD instructions will provide performance improvements of between 2x and 4x, depending on the multimedia application. The SIMD capabilities will enable developers to implement high-end features such as video codecs, speaker-independent voice recognition and 3D graphics, especially relevant for next generation wireless applications.

**Data Handling**
A system’s endianism refers to the way data is referenced and stored in a processor’s memory.

With increasing system on a chip (SoC) integration, a single chip is more likely to contain little-endian OS environments and interfaces (such as USB, PCI), but with big-endian data (TCP/IP packets, MPEG streams). With ARMv6, support for mixed-endian systems has been improved. As a result, handling data in mixed-endian systems under ARMv6 is far more efficient.

Unaligned data is data that is not aligned to its natural size boundary. For example, within DSP applications there is sometimes a requirement to treat words with half-word data alignment. For a processor to handle this situation efficiently requires that it be able to load a word aligned to any half-word boundary.

Current versions of the architecture require a number of instructions to manage unaligned data. ARMv6 compliant architectures will manage unaligned data more efficiently in hardware. In algorithms that rely heavily on DSP operations with unaligned data, ARMv6 implementations will have a performance advantage and may also benefit from reduced code size. Unaligned support also makes it more efficient for ARM to emulate other processors, such as Motorola’s 68000 family.

Similar to recent ARMv5 implementations such as ARM10 and XScale™, ARMv6 is based on a 32-bit processor. ARMv6 will support implementations based on bus widths of 64-bits and above - ARM10 and XScale support 64-bit buses today. This provides bus throughput equivalent to, or even better than a 64-bit machine, but without the power and area overhead of a full 64-bit CPU.

**Exceptions and Interrupts**
For implementations targeted at real-time systems, efficient handling of interrupts can be critical. Examples include systems such as hard disk controllers, and engine management applications, where the consequences can be severe if a critical interrupt does not get serviced.
in time. More efficient handling of exception and interrupt conditions also improve overall
system performance. This is especially important in reducing system latency.

In ARMv6, new instructions have been added to the ISA to improve the implementation of
interrupts and exceptions. These provide the ability to efficiently nest exception handling onto a
different privileged mode.

Each of these architectural advances is described in more detail in the following sections.

**Programmer’s Model**

Six new status bits have been added to the programmer’s model. Four bits are associated with
providing “greater than or equal to” status for the new multimedia instructions. The E-bit
indicates the current load/store endian setting for the core, and the A-bit is used to mask
imprecise data aborts.

- **GE[3:0] bits**
  - SIMD status bits - greater than or equal to for each 8/16-bit slice
- **E-bit**
  - Indicates the current load/store endian setting of the core
  - Can be set/cleared with the SETEND instruction
- **A-bit**
  - Indicates if imprecise data abort exceptions are masked

**Compatibility**

ARMv6 maintains 100% backward compatibility at the binary level for operating systems and
applications. The ARMv6 architecture requires that all Thumb and ‘E’ instructions be
implemented for backwards compatibility with ARMv5.

Some of the newly introduced ARMv6 instructions also have Thumb equivalents – for example
the new ‘REV*’ instructions. The BXJ instruction is also a requirement within ARMv6 for
consistent Java support – regardless of whether Jazelle technology is implemented or not.

**Improved Memory Management**

Memory management is primarily concerned with two issues. First, the translation of virtual
addresses into physical addresses within a system. Second, ensuring appropriate levels of
protection between different processes and tasks.

The ARM architecture is a load-store architecture, where the ARM core instructions can only
operate on data in registers that form part of the core. Load and store instructions are used to
transfer data to and from this register file.

A multi-level memory system is part of normal system design hierarchy. Closer coupled
memory systems tend to run faster, with level 1 memory systems ideally having no wait states.
In practical terms, this limits the size of memories that can be supported at core clock speeds.
Many high performance systems are now supporting additional (larger) L2 caches with some wait states, but less latency than if the memory was located off-chip. L3 cache may be provided as fast off-chip SRAM, with "normal" DRAM a level behind that.

ARM first introduced cores (e.g. ARM7TDMI), then developed and offered cached cores with MMU's (e.g. ARM720/920). ARMv6 is a logical progression on this - providing a complete definition of the L1 memory system, and to a lesser extent how memory levels beyond this need to behave for overall system correctness.

L1 memory will run synchronized to the core. Where different clock domains are introduced into a design, memory synchronization becomes dependent on the implementation.

**Figure 2. ARMv6 Memory Model**

ARM Virtual Memory System Architecture
The ARM Virtual Memory System Architecture v6 (VMSAv6) fully specifies the new Level 1 cache system – that most tightly coupled to the processor. The VMSA also specifies a Tightly-Coupled Memory (TCM) and DMA system. The architecture permits a range of implementations of these systems, with software-visible configuration registers to allow identification of the resources that exist. V6 supports hierarchy and memory ordering rules to ensure system correctness for additional levels of cache in both single processor and
multiprocessor systems. Memory ordering rules define the architecture, without constraining the implementation.

Version 6 now supports physically tagged caches, reducing software overhead on context switches. This can save up to 20% of the processor utilization by eliminating the need to perform cache flushing by the OS.

**ARM v6 L1 Cache**
The L1 cache is architected to reduce the requirement for cache clean and invalidation on a context switch. The cache may be organized as a Harvard system with separate instruction and data caches, or as a single unified von Neumann cache. The TCM is a physically-addressed area of scratchpad memory, which is implemented alongside the L1 cache. Similarly, the TCM can be organized as a Harvard or von Neumann system. The L1 DMA subsystem is designed to allow background transfers to and from the TCM.

<table>
<thead>
<tr>
<th>SBZ</th>
<th>0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Coarse page table page address</td>
<td>P</td>
</tr>
<tr>
<td>Section base address</td>
<td>SBZ</td>
</tr>
<tr>
<td>Reserved</td>
<td>1 1</td>
</tr>
</tbody>
</table>

**Page Table Formats**

The XP bit in Coprocessor 15 is used to enable this format, otherwise an ARMv5 legacy mode is invoked for backwards compatibility.

New features include:
- an execute never bit (XN)
- a “not Global” (nG) bit for address matching

Application Space Identifier - or ASID - support is another key feature in this area. When the nG-bit is set, address translation uses the virtual address and ASID for translation matching. This provides a significant saving in software overhead on context switches, avoiding the need to flush on-chip translation buffers in most cases. The result is improved performance. The architecture also supports its use in task-aware debugging. The ASID forms part of a process ID that can be used in task aware debugging. Type extension, shared, and access permission bits are used to provide all the attributes necessary for the ARMv6 memory model. A P-bit, which is compatible with the mechanism already available on Intel’s XScaleTM product, has been added for memory protection.
Additional Translation Table Base Register

To improve page table handling, a second translation table base register has been added; CP15 now supports TTBR0 and TTBR1. A control register is used to program N, the number of leading zeroes (most significant address bits) in virtual addresses that use TTBR0; 0 < N < 7. The device resets with N equal to zero, meaning all virtual addresses use TTBR0, otherwise the address space 0-232-N will use TTBR0 and other addresses will use TTBR1. The size of the first level page table required for TTBR0 will vary from 128 bytes to 16kB depending on the value of N, offering additional scope for memory savings in resource critical systems, particularly where multiple tables are held in memory and swapped on a context switch by updating the translation base register.

Multiprocessing

While many ARM processors today are used in isolation, or with simple communications links to another resource with its own memory, there are increasing requirements for unified memory models, and closer coupling of processors in general.

Systems consisting of multiple processors – either multiple ARM processors or a mixture of ARMs and DSPs, are becoming more common. Improvements to the ARMv6 memory management unit (MMU) are important in ensuring that processors get predictable and consistent (coherent) views of memory when it is shared between multiple processors.

Improvements include defining the level1 memory system, and the memory order model - how loads and stores to memory relate to each other.

As well as memory improvements to facilitate multiprocessing, Load and Store Exclusive instructions have been added in version 6 to support semaphores in multiprocessor systems (used to synchronise tasks). These instructions provide a more powerful and flexible mechanism over the current swap instructions.

- **LDREX{<cond>} <Rd>, [<Rn>]**
  This performs a load, then sets a monitor to “watch” the address
- **STREX {<cond>} <Rd>, <Rm>, [<Rn>]**
  This performs a store and returns “success” in Rd if no intervening access detected by the monitor.

Exceptions and Interrupts

The desire to implement more efficient processing of exception and interrupt conditions has led to several architectural enhancements in ARMv6. A low interrupt latency mode allows implementations to modify or switch off features. This is enabled by the FI bit in CP15 register 1 (the CPU control and configuration register). This facility enables designers to make performance versus latency tradeoffs, and support both in the design. For example, Load
Multiple or Store Multiple instructions (LDM/STM) can be made interruptible where low latency is important. Normally, these instructions would run to completion.

ARMv6 provides for vectored interrupt support. The Vectored Interrupt Controller (VIC) is enabled by the VE bit in CP15 register 1. The VE bit is used to enable returning vectored interrupts directly to the core. VIC support is currently provided through an external system peripheral. This requires an IRQ or FIQ system interrupt, and then the interrupt handler needs to perform a memory mapped read of a register for the vector address.

Imprecise external aborts are supported in ARMv6. The A-bit added to the program status register (CPSR), provides an abort mask for this - like the I and F bit masks for IRQ and FIQ.

**Stack Handling and Mode Change support**

New stack handling capabilities in ARMv6 avoid the need for multiple stacks. The ARMv6 register model supports separate stacks in the different modes. Many operating systems like to nest all their state saving and restoring onto a single stack. Version 6 makes this much more efficient. The stack handling capabilities are based on new crossmode state-saving instructions:

- **SRS #Mode** - Save Return State onto stack belonging to ‘Mode’
- **RFE** - Return From Exception

The SRS instruction allows register 14 and the SPSR (Saved Processor Status Register) for the current mode to be saved to a stack in a different mode. The RFE instruction loads the PC and CPSR (Current Processor Status Register) from the saved state.

New instructions support fast mode changes in privileged modes. Instructions cannot be used in user mode for security reasons.

- **CPSID #Mode** (and disable interrupts)
- **CPSIE #Mode** (and enable interrupts)

The CPS instructions allow software to move efficiently to a different mode while enabling or disabling interrupts.

Table 1a and 1b show code extracts, including entry code and exit code, comparing stack handling with SRS/CPS/RFE usage in ARMv6 with ARMv5. The two sections of code are exact equivalents for the context:

- An FIQ entry - FIQ2 - from a VIC is to be processed in ABORT mode (there is a higher priority FIQ - FIQ1 - which uses FIQ mode directly)
- In ARMv5, the handler needs to use the FIQ_stack as a scratchpad for R0-R3 to provide the necessary workspace
- The target (abort mode) stack has R2, R3, R14 (Link register) and SPSR (the saved status captured in FIQ mode needed for the eventual return) added to the
ABORT_stack

R0 and R1 are transferred with their context intact

<table>
<thead>
<tr>
<th>ARMv5</th>
<th>ARMv6</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIQ2handler. FIQs are now re-enabled, with original R2, R3, R14, SPSR on stack. Includes code to stack any more registers required, process the interrupt and unstack extra registers.</td>
<td></td>
</tr>
<tr>
<td>STMIA R13, {R0-R3}</td>
<td>SUB R14, R14, #4</td>
</tr>
<tr>
<td>MOV R0, LR</td>
<td>SRSFD R13_abt!</td>
</tr>
<tr>
<td>MRS R1, SPSR</td>
<td>CPSIE f, #0x1B ; = Abort mode</td>
</tr>
<tr>
<td>ADD R2, R13, #8</td>
<td>STMFD R13!, {R2,R3}</td>
</tr>
<tr>
<td>MRS R3, CPSR</td>
<td></td>
</tr>
<tr>
<td>BIC R3, R3, #0x1F</td>
<td></td>
</tr>
<tr>
<td>ORR R3, R3, #0x1B ; = Abort mode No.</td>
<td></td>
</tr>
<tr>
<td>MSR CPSR_c, R3</td>
<td></td>
</tr>
<tr>
<td>STMFD R13!, {R0,R1}</td>
<td></td>
</tr>
<tr>
<td>LDMIA R2, {R0,R1}</td>
<td></td>
</tr>
<tr>
<td>STMFD R13!, {R0,R1}</td>
<td></td>
</tr>
<tr>
<td>LDMDDB R2, {R0,R1}</td>
<td></td>
</tr>
<tr>
<td>BIC R3, R3, #0x40 ; = F bit</td>
<td></td>
</tr>
<tr>
<td>MSR CPSR_c, R3</td>
<td></td>
</tr>
<tr>
<td>Exit code including the LDR/STR instructions needed to acknowledge the VIC</td>
<td></td>
</tr>
<tr>
<td>ADR R2, #VICaddress MRS R3, CPSR</td>
<td>LDMFD R13!, {R2,R3} ADR R14, #VICaddress</td>
</tr>
<tr>
<td>ORR R3, R3, #0x40 ; = F bit</td>
<td>CPSID f</td>
</tr>
<tr>
<td>MSR CPSR_c, R3 STR R0, [R2,#AckFinished]</td>
<td>STR R0, [R14,#AckFinished] RFEFD R13!</td>
</tr>
<tr>
<td>LDR R14, [R13,#12] ; Original SPSR value</td>
<td></td>
</tr>
<tr>
<td>MSR SPSR_fsxc, R14</td>
<td></td>
</tr>
<tr>
<td>LDMDFD R13!, {R2,R3,R14}</td>
<td></td>
</tr>
<tr>
<td>ADD R13, R13, #4</td>
<td></td>
</tr>
<tr>
<td>SUBS PC, R14, #4</td>
<td></td>
</tr>
<tr>
<td>Approximate cycles: 35</td>
<td>Approximate cycles: 11</td>
</tr>
</tbody>
</table>

Table 1a. Efficient code handling in ARMv6
The code illustrates a different stack mechanism for FIQ-mode and ABORT-mode:

- **FIQ-mode:** "Empty ascending" stack; uses STMIA and LDMDB
- **ABORT-mode:** "Full descending" stack; uses STMDB and LDMIA
  - **DB == decrement before**
  - **IA == increment after**
- "FD" is a stack-orientated suffix for the Full Descending stack model, supported in the ARM assembler. STMFD and LDMFD translate to STMDB and LDMIA.
- **ADR** is a pseudo assembler instruction used to load an address

In the ARMv5 case, it was necessary to save R2 and R3, as the registers were required for the algorithm. In the ARMv6 case they were stored for equivalence reasons.

<table>
<thead>
<tr>
<th>Entry code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>add R2, R3, R14 and SPSR to the target (ABORT) stack</td>
</tr>
<tr>
<td>switch mode =&gt; ABORT</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>exit code:</th>
</tr>
</thead>
<tbody>
<tr>
<td>recover R2 and R3 context</td>
</tr>
<tr>
<td>return from handler (pop values from the ABORT stack)</td>
</tr>
<tr>
<td>-&quot;LR&quot; =&gt; PC</td>
</tr>
<tr>
<td>-&quot;SPSR&quot; =&gt; CPSR</td>
</tr>
</tbody>
</table>

**Table 1b. Entry/Exit code handling in ARMv6**

For ARMv5 the FIQs are disabled for some time at the start of the lower-priority FIQs. The worst-case interrupt latency for the FIQ1 interrupt occurs if a lower-priority FIQ2 has just fetched its handler address, and is approximately:

- 3 cycles for the pipeline refill after the LDR PC instruction fetches the handler address
- + 24 cycles to get to and execute the MSR instruction that re-enables FIQs
- + 3 cycles to re-enter the FIQ exception
- + 5 cycles for the LDR PC instruction at FIQhandler
- or about 35 cycles.

For ARMv6, the worst-case interrupt latency for a FIQ1 now occurs if the FIQ1 occurs during a FIQ2’s interrupt entry sequence, just after it disables FIQs, and is approximately:

- 3 cycles for the pipeline refill for the FIQ2’s exception entry sequence
- + 5 cycles to get to and execute the CPSIE instruction that re-enables FIQs
- + 3 cycles to re-enter the FIQ exception
- or about 11 cycles.
The underlying mechanism illustrated can be used from any privileged mode, to stack and swap state to a different privileged mode, then return from this mode using the stack values.

**Data Handling**

Version 6 has introduced two features for mixed-endian support:

**E-bit**

A state bit (E-bit) is set and cleared under program control using the SETEND instruction. The E-bit defines which endian to load and store data. Figure 4 illustrates the functionality associated with the E-bit for a word load or store operation.

![Data bytes in memory](image)

**Figure 4. Endian support - Word Load/Store with E-bit**

This mechanism enables efficient dynamic data load/store for system designers who know they need to access data structures in the opposite endianness to their OS/environment. Note that the address of each data byte is fixed in memory. However, the byte lane in a register is different.

**Unaligned Data Support**

In ARMv5 (ARM state), an access will abort in all unaligned cases when the A-bit in CP15 register 1 is set, otherwise:

- an unaligned word load (LDR) will rotate right by addr[1:0] x 8 bits
- unaligned word stores (STR) will ignore addr[1:0] and treat them as zero
- unaligned halfword loads and stores are UNPREDICTABLE
- Dword (64-bit) loads and stores LDRD/STRD are implementation dependent as to whether they require Dword or word alignment to execute correctly.

Version 6 introduces unaligned data support for 32-bit words and 16-bit halfwords, the behavior controlled by a new (U-bit) in CP15 register 1. The A-bit will still cause unaligned
errors to abort in all cases; Dwords if not word aligned. When the U-bit is set and the A-bit is clear, LDR, STR, LDRH and STRH support unaligned accesses in hardware. All other unaligned accesses will data abort and require handling in software.

**REV Instructions**

Three byte reverse instructions are available in both ARM and Thumb states. The byte reverse (REV) instructions can be used to improve byte-swap routines present in many code bases today typically replacing four instructions with a single instruction (Figure 5).

New instructions (ARM and Thumb variants)

- REV - byte reverse a word
- REV16 - byte reverse packed (2 x) halfwords
- REVSH - byte reverse + sign extend halfword

![Figure 5. ARMv6 Byte Reverse Instructions](image)

**Media Extensions**

The media extensions were announced during 2000, and will be implemented for the first time in ARMv6 designs. They include a set of Single Instruction Multiple Data (normally known as SIMD) instructions, as well as new multiplier and Sum-of-Absolute-Differences support. The SIMD instructions use the GE-bits added to the programmer’s model.

The new instructions support 8 and 16-bit SIMD arithmetic, including four 8-bit and two 16-bit operations, parallel add and subtract, selection, packing and unpacking. Advanced multiplier options include dual 16-bit multiply-accumulate, and a new long multiply instruction, useful for cryptographic applications. Table 2 demonstrates the efficiency of the complex multiply in ARMv6 architectures.
### ARMv5TE: 5 cycles in a single-cycle implementation

- \( \text{SMULTT} \) Real, Ra, Rb ; Real = Ra.real*Rb.real
- \( \text{SMULBB} \) Temp, Ra, Rb ; Temp = Ra.imag*Rb.imag
- \( \text{SUB} \) Real, Real, Temp ; Real = Ra.real*Rb.real - Ra.imag*Rb.imag
- \( \text{SMULTB} \) Imag, Ra, Rb ; Imag = Ra.real*Rb.imag
- \( \text{SMLABT} \) Imag, Ra, Rb ; Imag = Ra.real*Rb.imag + a.imag*Rb.real

### ARMv6: 2 cycles in a single-cycle implementation

- \( \text{SMUSD} \) Real, Ra, Rb ; Real = Ra.real*Rb.real - Ra.imag*Rb.imag
- \( \text{SMUADX} \) Imag, Ra, Rb ; Imag = Ra.real*Rb.imag + a.imag*Rb.real

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Cycles/4 pixels</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARMv5TE</td>
<td>18 cycles</td>
</tr>
<tr>
<td>ARMv6</td>
<td>3 cycles</td>
</tr>
</tbody>
</table>

**Table 2. Example 16-bit Complex Multiply**

ARMv6 provides better support for the sum of absolute differences calculation, with the inclusion of the USAD8 (sum of differences) and USADA8 (sum of differences and accumulate) instructions. These are particularly useful for video encoding and motion estimation applications.

Table 3 shows the relative performance of the sum of absolute differences. The comparison with version 5TE relates to a software implementation in ARM registers. This can also be accelerated with the MOVE coprocessor.

**Table 3. Implementing Sum of Absolute Differences**

Architectural provision in ARMv6 yields a choice between hardware and software implementation, giving similar performance results. A single instruction in software, with the register usage overhead, or the MOVE coprocessor hardware option with dedicated resource, which leaves the ARM processor free for other tasks.

**Summary**

The introduction of the ARMv6 architecture brings a new set of features and a performance leap that will meet the needs of ARM’s partners as they design next-generation products across a range of target markets.
ARMv6 consolidates the developments in ARMv5, and provides 100% backwards compatibility. It also adds significant enhancements for next-generation applications. New multimedia support provides 4x-processing improvements in some media applications. The new VMSA provides faster context switches enhancing performance of platform processors hosting complex operating systems. Improved multiprocessor support eases development and enhances the performance of systems based on multiple ARM cores, or ARM plus DSP core configurations.

ARM will be working with a growing number of partners during 2002 to ensure the successful introduction of the ARMv6 architecture. As well as ARMv6-compliant silicon product introduction, considerable effort will also be devoted to development support – examples are improved AMBA support used for on-chip connectivity, platform design support, code generation and debug tools, as well as operating system porting.
Learning Standard C++ as a New Language

Bjarne Stroustrup

ABSTRACT

To get the most out of Standard C++ [C++, 1998], we must rethink the way we write C++ programs. An approach to such a "rethink" is to consider how C++ can be learned (and taught). What design and programming techniques do we want to emphasize? What subsets of the language do we want to learn first? What subsets of the language do we want to emphasize in real code?

This paper compares a few examples of simple C++ programs written in a modern style using the standard library to traditional C-style solutions. It argues briefly that lessons from these simple examples are relevant to large programs. More generally, it argues for a use of C++ as a higher-level language that relies on abstraction to provide elegance without loss of efficiency compared to lower-level styles.

1 Introduction

We want our programs to be easy to write, correct, maintainable, and acceptably efficient. It follows that we ought to use C++ – and any other programming language – in ways that most closely approximate this ideal. It is my conjecture that C++ community has yet to internalize the facilities offered by Standard C++ so that major improvements relative to the ideal can be obtained from reconsidering our style of C++ use. This paper focuses on the styles of programming that the facilities offered by Standard C++ support – not the facilities themselves.

The key to major improvements is a reduction of the size and complexity of the code we write through the use of libraries. Below, I demonstrate and quantify these reductions for a couple of simple examples such as might be part of an introductory C++ course.

By reducing size and complexity, we reduce development time, ease maintenance, and decrease the cost of testing. Importantly, we also simplify the task of learning C++. For toy programs and for students who program only to get a good grade in a nonessential course, this simplification would be sufficient. However, for professional programmers efficiency is a major issue. Only if efficiency isn’t sacrificed can we expect our programming styles to scale to be usable in systems dealing with the data volumes and real-time requirements regularly encountered by modern services and businesses. Consequently, I present measurements that demonstrate that the reduction in complexity can be obtained without loss of efficiency.
Finally, I discuss the implications of this view on approaches to learning and teaching C++

2 Complexity
Consider a fairly typical second exercise in using a programming language:

```
write a prompt "Please enter your firstname"
read the name
write out "Hello <name>"
```

In Standard C++, the obvious solution is:
```
#include <iostream>  // get standard I/O facilities
#include <string>   // get standard string facilities
int main()
{
    using namespace std;  // gain access to standard library
    cout << "Please enter your firstname: \n";
    string name;
    cin >> name;
    cout << "Hello " << name << \n;
}
```

For a real novice, we need to explain the ‘scaffolding:’ What is `main()`? What does `#include` mean? What does `using` do? In addition, we need to understand all the ‘small’ conventions, such as what `\n` does, where semicolons are needed, etc.

However, the main part of the program is conceptually simple and differs only notationally from the problem statement. We have to learn the notation, but doing so is relatively simple: `string` is a string, `cout` is output, `<<` is the operator we use write to output, etc.

To compare, consider a traditional C-style solution†:
```
#include <stdio.h>  // get standard I/O facilities
int main()
{
    const int max = 20;  // maximum name length is 19 characters
    char name[max];
    printf("Please enter your firstname: \n");
    scanf("%s ", name);  // read characters into name
    printf("Hello %s \n", name);
    return 0;
}
```

Objectively, the main logic here is slightly – but only slightly – more complicated than the C++-style version because we have to explain about arrays and the magic `%s`' . The main
problem is that this simple C-style solution is shoddy. If someone enters a “first name” that is longer than the magic number 19 (the stated number 20 minus one for a C-style string terminating zero), the program is corrupted.

It can be argued that this kind of shoddiness is harmless as long as a proper solution is presented “later on.” However, that line of argument is at best “acceptable” rather than “good.” Ideally, a novice user isn’t presented with a program that brittle.

What would a C-style program that behaved as reasonably as the C++-style one look like? As a first attempt we could simply prevent the array overflow by using `scanf()` in a more appropriate manner:

```c
#include <stdio.h> // get standard I/O facilities
int main()
{
    const int max = 20;
    char name[max];
    printf("Please enter your first name: \n");
    scanf("%19s \n", name); // read at most 19 characters into name
    printf("Hello %s \n", name);
    return 0;
}
```

There is no standard way of directly using the symbolic form of the buffer size, `max`, in the `scanf()` format string, so I had to use the integer literal. That is bad style and a maintenance hazard. The expert-level alternative is not one I’d care to explain to novices:

```c
char fmt[10];
printf(fmt,"%d %s",max-1); // create a format string: plain %s can overflow
scanf(fmt,name); // read at most max-1 characters into name
```

Furthermore, this program throws “surplus” characters away. What we want is for the string to expand to cope with the input. To achieve that, we have to descend to a lower level of abstraction and deal with individual characters:

```c
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
void quit() // write error message and quit
{
    fprintf(stderr,"memory exhausted \n");
    exit(1);
}
int main()
{
    int max = 20;
    char *name = (char*)malloc(max); // allocate buffer
    if(name == 0) quit();
```
```c
printf("Please enter your first name:\n ");
while (true) { // skip leading whitespace
    int c = getchar();
    if (c == EOF) break; // end of file
    if (isspace(c)) {
        ungetc(c, stdin);
        break;
    }
}
int i = 0;
while (true) {
    int c = getchar();
    if (c == '\n' || c == EOF) { // at end; add terminating zero
        name[i] = 0;
        break;
    }
    name[i] = c;
    if (i == max - 1) { // buffer full
        max = max + max;
        name = (char*) realloc(name, max); // get a new and larger buffer
        if (name == 0) quit();
    }
    i++;
}
printf("Hello %s\n", name);
free(name); // release memory
return 0;
```

Compared to the previous versions, this seems rather complex. I feel a bit bad adding the code for skipping whitespace because I didn’t explicitly require that in the original problem statement. However, skipping initial whitespace is the norm and the other versions of the program skip whitespace.

One could argue that this example isn’t all that bad. Most experienced C and C++ programmers would – in a real program – probably (hopefully?) have written something equivalent in the first place. We might even argue that if you couldn’t write that program, you shouldn’t be a professional programmer. However, consider the added conceptual load on a novice. This variant uses nine different standard library functions, deals with character-level input in a rather detailed manner, uses pointers, and explicitly deals with free store. To use `realloc()` while staying portable, I had use `malloc()` (rather than `new`). This brings the issues of sizes and casts† into the picture. It is not obvious what is the best way to handle the possibility of memory exhaustion in a small program like this. Here, I simply did something obvious to avoid the discussion going off on another tangent. Someone using the C-style approach would have to
carefully consider which approach would form a good basis for further teaching and eventual use.

To summarize, to solve the original simple problem, I had to introduce loops, tests, storage sizes, pointers, casts, and explicit free-store management in addition to whatever a solution to the problem inherently needs. This style is also full of opportunity for errors. Thanks to long experience, I didn’t make any of the obvious off-by-one or allocation errors. Having primarily worked with stream I/O for a while, I initially made the classical beginner’s error of reading into a char (rather than into an int) and forgetting to check for EOF. In the absence of something like the C++ standard library, it is no wonder that many teachers stick with the ‘‘shoddy’’ solution and postpone these issues until later. Unfortunately, many students simply note that the shoddy style is ‘‘good enough’’ and quicker to write than the (non-C++ style) alternatives. Thus they acquire a habit that is hard to break and leave a trail of buggy code behind.

This last C-style program is 41 lines compared to 10 lines for its functionally equivalent C++-style program. Excluding ‘‘scaffolding,’’ the difference is 30 lines vs 4. Importantly, the C++-style lines are also shorter and inherently easier to understand. The number and complexity of concepts needed to be explained for the C++-style and C-style versions are harder to measure objectively, but I suggest a 10-to-1 advantage for the C++-style version.

3 Efficiency
Efficiency is not an issue in a trivial program like the one above. For such programs, simplicity and (type) safety is what matters. However, real systems often have parts where efficiency is essential. For such systems, the question becomes ‘‘can we afford a higher level of abstraction?’’

Consider a simple example of the kind of activity that occurs in programs where efficiency matters:
read an unknown number of elements
do something to each element
do something with all elements

The simplest specific example I can think of is a program to find the mean and median of a sequence of double precision floating-point numbers read from input. A conventional C-style solution would be:
/ / C-style solution:
#include <stdlib.h>
#include <stdio.h>
int compare (const void * p, const void * q ) // comparison function for use by qsort()
{
    register double p0 = *(double *)p; // compare doubles
    register double q0 = *(double *)q;

if (p0 > q0) return 1;
if (p0 < q0) return -1;
return 0;
}
void quit() // write error message and quit
{
    fprintf(stderr,"memory exhausted \n");
    exit(1);
}
int main(int argc, char* argv[])
{
int res = 1000; // initial allocation
char* file = argv[2];
double* buf = (double*)malloc(size of (double)*res);
if (buf == 0) quit();
double median = 0;
double mean = 0;
int n = 0; // number of elements
FILE* fin = fopen(file,"r"); // open file for reading
doubled;
while (scanf(fin,"%lg",&d) == 1) { // read number, update running mean
    if (n == res)
    {
        res += res;
        buf[n++] = d;
        mean = (n == 1) ? d : mean + (d - mean) / n; // prone to rounding errors
    }
}
qsort(buf, n, size of (double), compare);
if (n)
{
    int mid = n / 2;
    median = (n % 2) ? buf[mid] : (buf[mid - 1] + buf[mid]) / 2;
}
printf("number of elements = %d, median = %g, mean = %g \n", n, median, mean);
free(buf);
}

To compare, here is an idiomatic C++ solution:
// Solution using the Standard C++ library:
#include <vector>
#include <fstream>
#include <algorithm>

int main(int argc, char* argv[])
{
    std::vector<double> data;
    std::ifstream fin(argv[2], std::ios::in);
    if (!fin)
    {
        std::cerr << "Error opening file \n";
        return 1;
    }
    double d;
    while (fin >> d)
    {
        data.push_back(d);
    }
    std::sort(data.begin(), data.end());
    int n = data.size();
    double median = (n % 2) ? data[n / 2] : (data[n / 2 - 1] + data[n / 2]) / 2;
    double mean = 0;
    for (int i = 0; i < n; ++i)
    {
        mean += data[i];
    }
    mean /= n;
    std::cout << "number of elements = \n";
```cpp
typedef char* file;

using namespace std;

t void main (int argc, char* argv[])
{
    char* file = argv[2];
    vector<double> buf;
    double median = 0;
    double mean = 0;
    ifstream fin (file, ios::in); // open file for input
    double d;
    while (fin >> d) {
        buf.push_back(d);
        mean = ((buf.size() == 1) ? d : mean + (d - mean) / buf.size()); // prone to rounding errors
    }
    sort (buf.begin(), buf.end());
    int mid = buf.size() / 2;
    median = (buf.size() % 2) ? buf[mid] : (buf[mid - 1] + buf[mid]) / 2;
}
```

Published in the May 1999 issue of "The C/C++ Users Journal". All rights reserved

```
cout << "number of elements = " << buf.size() << 
      " median = " << median << n
```

The size difference is less dramatic than in the previous example (43 vs 24 non-blank lines). Excluding, irreducible common elements such as the declaration of main() and the calculation of the median (13 lines) the difference is 20 lines vs 11. The critical input-and-store loop and the sort are both significantly shorter in the C++-style program (9 vs 4 lines for the read-and-store loop, and 9 lines vs 1 line for the sort). More importantly, their logic is far simpler in the C++ version – and therefore far easier to get right.

Again, memory management is implicit in the C++-style program; a vector grows as needed when elements are added using push_back(). In the C-style program, memory management is explicit using realloc(). Basically, the vector constructor and push_back() in the C++-style program does what malloc(), realloc(), and the code tracking the size of allocated memory does in the C-style program. In the C++ style program, I rely on the exception handling to report memory exhaustion. In the C-style program, I added explicit tests to avoid the possibility of memory corruption.

Not surprisingly, the C++ version was easier to get right. I constructed this C++-style version from the C-style version by cut-and-paste. I forgot to include <algorithm>, I left n in place rather than using buf.size() twice, and my compiler didn’t support the local using-
directive so I had to move it outside main(). One the other hand, after fixing these four errors, the program ran correctly first time.

To a novice, \texttt{q sort()} is “odd.” Why do you have to give the number of elements? (because the array doesn’t know it). Why do you have to give the size of a \texttt{double}? (because \texttt{q sort()} doesn’t know that it is sorting doubles). Why do you have to write that ugly function to compare \texttt{double}s? (because \texttt{q sort()} needs a pointer to function because it doesn’t know the type of the elements that it is sorting). Why does \texttt{q sort()}’s comparison function take \texttt{const void*} arguments rather than \texttt{char*} arguments? (because \texttt{q sort()} can sort based on non-string values). What is a \texttt{void*} and what does it mean for it to be \texttt{const}? (“Eh, hmmm, we’ll get to that later”). Explaining this to a novice without getting a blank stare of wonderment over the complexity of the answer is not easy. Explaining, \texttt{sort(v.begin(), v.end())} is comparatively easy: “Plain \texttt{sort(v)} would have been simpler in this case, but sometimes we want to sort part of a container so it’s more general to specify the beginning and end of what we want to sort.”

To compare efficiencies, I first determined how much input was needed to make an efficiency comparison meaningful. For 50,000 numbers the programs ran in less than half a second each, so I choose to compare runs with 500,000 and 5,000,000 input values:

<table>
<thead>
<tr>
<th>Measurability</th>
<th>• Enabling the definition of systems of measures around quality, productivity, progress and risk • Reporting and analyzing such metrics throughout project execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment</td>
<td>• Aligning LOB and IT priorities • Aligning project outcome with expectations of end users</td>
</tr>
<tr>
<td>Discipline</td>
<td>• Defining, deploying and tracking compliance with software processes • Introducing more rigor to the process of managing change and predicting its impact</td>
</tr>
</tbody>
</table>

The key numbers are the ratio; a ratio larger than one means that the C++-style version is faster. Comparisons of languages, libraries, and programming styles are notoriously tricky, so please do not draw sweeping conclusions from these simple tests. The numbers are averages of several runs on an otherwise quiet machine. The variance between different runs of an example was less than 1%. I also ran strictly ISO C conforming versions of the C-style programs. As expected there were no performance difference between those and their C-style C++ equivalents.

I had expected the C++-style program to be only slightly faster. Checking other C++ implementations, I found a surprising variance in the results. In some cases, the C-style version even outperformed the C++-style version for small data sets. However, the point of this example is that a higher level of abstraction and a better protection against errors can be
affordable given current technology: The implementation I used is widely available and cheap – not a research toy. Implementations that claim higher performance are also available.

It is not unusual to find people being willing to pay a factor of 3, 10, or even 50 for convenience and better protection against errors. Getting the benefit together with a doubling or quadrupling of speed is spectacular. These figures should be the minimum that a C++ library vendor would be willing to settle for.

To get a better idea of where the time was spent, I ran a few additional tests:

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
 & \multicolumn{2}{|c|}{unoptimized} & \multicolumn{2}{|c|}{optimized} \\
\hline
 & C & C/C++ ratio & C & C/C++ ratio \\
\hline
read & 2.1 & 1.33 & 2.0 & 1.40 \\
generate & 0.6 & 0.5 & 0.4 & 0.75 \\
read&sort & 3.5 & 1.75 & 2.5 & 2.04 \\
generate&sort & 2.0 & 1.75 & 0.9 & 2.89 \\
\hline
\end{tabular}
\end{table}

Naturally, “read” simply reads the data and “read&sort” reads the data and sorts it but doesn’t produce output. To get a better feel for the cost of input, “generate” produces random numbers rather than reading.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|c|}
\hline
 & \multicolumn{2}{|c|}{unoptimized} & \multicolumn{2}{|c|}{optimized} \\
\hline
 & C & C/C++ ratio & C & C/C++ ratio \\
\hline
read & 21.5 & 1.35 & 21.3 & 1.34 \\
generate & 7.2 & 0.57 & 5.2 & 0.69 \\
read&sort & 38.4 & 4.49 & 27.4 & 4.62 \\
generate&sort & 24.4 & 6.03 & 11.3 & 8.90 \\
\hline
\end{tabular}
\end{table}

From other examples and other implementations, I had expected streamio to be somewhat slower than stdio. That was actually the case for a previous version of this program where I use \texttt{cin} rather than a \texttt{file stream}. It appears that on some C++ implementations, file I/O is much faster than \texttt{cin}. The reason is at least partly poor handling of the tie between \texttt{cin} and \texttt{cout}. However, these numbers demonstrate that C++-style I/O can be as efficient as C-style I/O.

Changing the programs to read and sort integers instead of floating-point values did not change the relative performance – though it was nice to note that making that change was much simpler in the C++ style program (2 edits as compared to 12 for the C-style program). That is a good omen for maintainability.

The differences in the “generate” tests reflect a difference in allocation costs. A \texttt{vector} plus \texttt{push_back()} ought to be exactly as fast as an array plus \texttt{malloc()} / \texttt{free()}, but it
wasn’t. The reason appears to be failure to optimize away calls of initializers that do nothing. Fortunately, the cost of allocation is (always) dwarfed by the cost of the input that caused the need for the allocation.

As expected, \texttt{sort()} was noticeably faster than \texttt{qsort()}. The main reason is that \texttt{sort()} inlines its comparison operations whereas \texttt{qsort()} must call a function.

It is hard to choose an example to illustrate efficiency issues. One comment I had from a colleague was that reading and comparing numbers wasn’t realistic. I should read and sort strings. So I tried this program:

```c
#include <vector>
#include <fstream>
#include <algorithm>
#include <string>

using namespace std;

int main(int argc, char* argv[])
{
    char* file = argv[2]; // input file name
    char* o file = argv[3]; // output file name
    vector<string> buf;
    fstream fin (file, ios::in);
    string d;
    while (getline(fin, d)) { buf.push_back(d); // add line from input to buf
        sort(buf.begin(), buf.end());
        fstream fout (ofile, ios::out);
        copy(buf.begin(), buf.end(), ostream_iterator<string>(fout, "\n")); // copy to output
    }
}
```

I transcribed this into C and experimented a bit to optimize the reading of characters. The C++-style code performs well even against hand-optimized C-style code that eliminates copying of strings. For small amounts of output there is no significant difference and for larger amounts of data \texttt{sort()} again beats \texttt{qsort()} because of its better inlining:

<table>
<thead>
<tr>
<th>Read, sort, and write strings</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>500,000 elements</td>
</tr>
<tr>
<td>2,000,000 elements</td>
</tr>
</tbody>
</table>

I used 2 million strings because I didn’t have enough main memory to cope with 5 million strings without paging.

To get an idea of what time was spent where, I also ran the program with the \texttt{sort()} omitted:
The strings were relatively short (seven characters on average).

Note that string is a perfectly ordinary user-defined type that just happens to be part of the standard library. What we can do efficiently and elegantly with a string, we can do efficiently and elegantly with many other user-defined types.

Why do I discuss efficiency in the context of programming style and teaching? The styles and techniques we teach must scale to real-world problems. C++ is – among other things – intended for large-scale systems and systems with efficiency constraints. Consequently, I consider it unacceptable to teach C++ in a way that leads people to use styles and techniques that are effective for toy programs only; that would lead people to failure and to abandon what was taught. The measurements above demonstrate that a C++ style relying heavily on generic programming and concrete types to provide simple and type-safe code can be efficient compared to traditional C styles. Similar results have been obtained for object-oriented styles.

It is a significant problem that the performance of different implementations of the standard library differ dramatically. For a programmer who wants to rely on standard libraries (or widely distributed libraries that are not part of the standard), it is often important that a programming style that delivers good performance on one system give at least acceptable performance on another. I was appalled to find examples where my test programs ran twice as fast in the C++ style compared to the C style on one system and only half as fast on another. Programmers should not have to accept a variability of a factor of four between systems. As far as I can tell, this variability is not caused by fundamental reasons, so consistency should be achievable without heroic efforts from the library implementors. Better optimized libraries may be the easiest way to improve both the perceived and actual performance of Standard C++. Compiler implementers work hard to eliminate minor performance penalties compared with other compilers. I conjecture that the scope for improvements is larger in the standard library implementations.

Clearly, the simplicity of the C++-style solutions above compared to the C-style solutions was made possible by the C++ standard library. Does that make the comparison unrealistic or unfair? I don’t think so. One of the key aspects of C++ is its ability to support libraries that are both elegant and efficient. The advantages demonstrated for the simple examples hold for every application area where elegant and efficient libraries exist or could exist. The challenge to the C++ community is to extend the areas where these benefits are available to ordinary programmers. That is, we must design and implement elegant and efficient libraries for many more application areas and we must make these libraries widely available.

4 Learning C++
Even for the professional programmer, it is impossible to first learn a whole programming language and then try to use it. A programming language is learned in part by trying out its facilities for small examples. Consequently, we always learn a language by mastering a series of subsets. The real question is not “Should I learn a subset first?” but “Which subset should I learn first?”

One conventional answer to the question “Which subset of C++ should I learn first?” is “The C subset of C++.” In my considered opinion, that’s not a good answer. The C-first approach leads to an early focus on low-level details. It also obscures programming style and design issues by forcing the student to face many technical difficulties to express anything interesting. The examples in §2 and §3 illustrate this point. C++’s better support of libraries, better notational support, and better type checking are decisive against a “C first” approach. However, note that my suggested alternative isn’t “Pure Object-Oriented Programming first.” I consider that the other extreme.

For programming novices, learning the programming language should support the learning of effective programming techniques. For experienced programmers who are novices at C++, the learning should focus on how effective programming techniques are expressed in C++ and on techniques that are new to the programmer. For experienced programmers, the greatest pitfall is often to concentrate on using C++ to express what was effective in some other language. The emphasis for both novices and experienced programmers should be concepts and techniques. The syntactic and semantic details of C++ are secondary to an understanding of design and programming techniques that C++ supports.

Teaching is best done by starting from well-chosen concrete examples and proceeding towards the more general and more abstract. This is the way children learn and it is the way most of us grasp new ideas. Language features should always be presented in the context of their use. Otherwise, the programmer’s focus shifts from producing systems to delight over technical obscurities. Focussing on language-technical details can be fun, but it is not effective education.

On the other hand, treating programming as merely the handmaiden of analysis and design doesn’t work either. The approach of postponing actual discussion of code until every high-level and engineering topic has been thoroughly presented has been a costly mistake for many. That approach drives people away from programming and leads many to serious underestimate the intellectual challenge in the creation of production-quality code.

The extreme opposite to the “design first” approach is to get a C++ implementation and start coding. When encountering a problem, point and click to see what the online help has to offer. The problem with this approach is that it is completely biased towards the understanding of individual features and facilities. General concepts and techniques and not easily learned this way. For experienced programmers, this approach has the added problem of reinforcing the tendency to think in a previous language while using C++ syntax and library functions. For the novice, the result is a lot of if-then-else code mixed with code snippets inserted using cut-and-paste from vendor-supplied examples. Often the purpose of the inserted code is obscure to the
novice and the method by which it achieves its effect completely beyond comprehension. This is the case even for clever people. This “poking around approach” can be most useful as an adjunct to good teaching or a solid textbook, but on its own it is a recipe for disaster.

To sum up, I recommend an approach that
– proceeds from the concrete to the abstract,
– presents language features in the context of the programming and design techniques that they exist to support,
– presents code relying on relatively high-level libraries before going into the lower-level details (necessary to build those libraries),
– avoids techniques that do not scale to real-world applications,
– presents common and useful techniques and features before details, and
– focus on concepts and techniques (rather than language features).

No. I don’t consider this particularly novel or revolutionary. Mostly, I see it as common sense. However, common sense often gets lost in heated discussion about more specific topics such as whether C should be learned before C++, whether you must write Smalltalk to really understand Object-Oriented programming, whether you must start learning programming in a pure-OO fashion (whatever that means), and whether a thorough understanding of the software development process before trying to write code.

Fortunately, there is some experience with approaches that meet my criteria. My favorite approach is to start teaching the basic language concepts such as variables, declarations, loops, etc. together with a good library. The library is essential to enable students to concentrate on programming rather than the intricacies of, say C-style strings. I recommend the use of the C++ standard libraries or a subset of those. This is the approach taken by the Computer Science Advanced Placement course taught in American high schools [Horwitz,1999]. A more advanced version of that approach aimed at experienced programmers has also proved successful; for example, see [Koenig,1998].

A weakness of these specific approaches is the absence of a simple graphics and graphical user interfaces early on. This could (easily?) be compensated for by a very simple interface to commercial libraries. By “very simple,” I mean usable by students on day two of a C++ course. However, no such simple graphics and graphical user interface C++ library is widely available.

After the initial teaching/learning that relies on libraries, a course can proceed in a variety of ways based on the needs and interests of the students. At some point, the messier and lower-level features of C++ will have to be examined. One way of teaching/learning about pointers, casting, allocation, etc. is to examine the implementation of the classes used to learn the basics. For example, the implementation of \textit{string}, \textit{vector}, and \textit{list} classes are excellent contexts for discussions of language facilities from the C subset of C++ that are best left out of the first part of a course.
Classes, such as `vector` and `string`, that manage variable amounts of data require the use of free store and pointers in their implementation. Before introducing those, classes that doesn’t require that (concrete classes), such as a `Date`, a `Point`, and a `Complex` types can be used to to introduce the basics of class implementation.

I tend to present abstract classes and class hierarchies after the discussion of containers and the implementation of containers, but there are many alternatives here. The actual ordering of topics should depend on the libraries used. For example, a course using a graphics library relying on class hierarchies will have to explain the basics of polymorphism and the definition of derived classes relatively early.

Finally, please remember there is no one right way to learn and teach C++ and its associated design and programming techniques. The aims and backgrounds of students differ and so does the backgrounds and experience of their teachers and textbook writers.

5 Summary
We want our C++ programs to be easy to write, correct, maintainable, and acceptably efficient. To do that, we must design and program at a higher level of abstraction than has typically been done with C and early C++. Through the use of libraries, this ideal is achievable without loss of efficiency compared to lowerlevel styles. Thus, work on more libraries, on more consistent implementation of widely-used libraries (such as the standard library), and on making libraries more widely available can yield great benefits to the C++ community.

Education must play a major role in this move to cleaner and higher-level programming styles. The C++ community doesn’t need another generation of programmers who by default use the lowest level of language and library facilities available out of misplaced fear of inefficiencies. Experienced C++ programmers as well as C++ novices must learn to use Standard C++ as a new and higher-level language as a matter of course and descend to lower levels of abstraction only where absolutely necessary. Using Standard C++ as a glorified C or glorified C with Classes only would be to waste the opportunities offered by Standard C++.

6 Acknowledgements
Thanks to Chuck Allison for suggesting that I wrote a paper on learning Standard C++. Thanks to Andrew Koenig and Mike Yang for constructive comments on earlier drafts. My examples were compiled using Cygnus’ EGCS1.1 and run on a Sun Ultrasparc 10. The programs I used can be found on my homepages: http://www.research.att.com/~bs.

7 References


Open Application Lifecycle Management (ALM)  
Unlocking the Full Value OF Managed Software Delivery

Executive Summary

Application Lifecycle Management (ALM) is emerging as a promising approach to improving the software delivery process. However, "traditional" ALM hasn't been able to reach its full potential for delivering business value. Why? Because vendors are aggressively pushing restrictive, end-to-end ALM solutions that aim to lock customers into proprietary IT platforms. Customers soon find that these solutions don't integrate well with their existing development processes, tools and platforms. Unfortunately, this leaves software development teams with disconnected processes and silos of ALM data, which in turn prevents them from realizing the full value of ALM.

To overcome this challenge, a new approach is required, one that enables customers to deliver software on top of a mixed development environment. With Borland's Open ALM solutions, organizations can leverage their existing software development assets and IP and achieve visibility, traceability and discipline across the complete software delivery cycle. Customers can now benefit from an optimized ALM platform, with the advantages of a fully connected, managed and measurable software delivery process.

Predictable Software Delivery: Mission Impossible?
Software development is an intrinsically complicated undertaking. Delivering reasonably defined software within acceptable quality, budget and time-to-market constraints requires constant coordination of a vast number of activities among many professionals. The complexity of managing and tracking software delivery projects increases when organizations decide to leverage distributed development models, such as offshore development or outsourcing. As a result, project cancellations and failures are ubiquitous. Cost overruns, schedule slippages, low quality and poor reliability are disturbing norms in the software industry.

Consequently, software development organizations have been increasingly pressured to become more mature and to adopt well-orchestrated, systematic and process-centric approaches that follow the steps of more traditional engineering disciplines. With growing standardization and adoption of enterprise development platforms, the challenges facing the industry have become less technical in nature.

Rather, the ability to achieve consistent and predictable value from software development has become a top priority for many software executives who need the confidence that their teams
will be effective in their delivery. With this in mind, companies like Borland have designed ALM platforms to address the demand for consistency and predictability of software delivery.

The Emergence of ALM
As the application development tools industry responds to the need for predictable software delivery, it has expanded its focus beyond tools for individual developer productivity. Vendors have expanded the breadth of their portfolios to address additional roles in the delivery process, and integrated existing and new capabilities into their offerings. These suites of products, often marketed and sold as team-based development platforms, have marked the emergence of Application Lifecycle Management, or ALM, as a new market category and as a software development discipline.

ALM platforms specifically address the challenge of increasing the consistency and predictability of software delivery. They do that by providing integration and automation for each of the major roles that participates in the process, and by automating the following capabilities:

<table>
<thead>
<tr>
<th>Measurability</th>
<th>• Enabling the definition of systems of measures around quality, productivity, progress and risk • Reporting and analyzing such metrics throughout project execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alignment</td>
<td>• Aligning LOB and IT priorities • Aligning project outcome with expectations of end users</td>
</tr>
<tr>
<td>Discipline</td>
<td>• Defining, deploying and tracking compliance with software processes • Introducing more rigor to the process of managing change and predicting its impact</td>
</tr>
</tbody>
</table>

These capabilities enable IT managers to balance and prioritize their software project portfolios, while achieving increased levels of control over their teams and much better visibility into project execution. With ALM, executives can also be assured that the software development process is far more auditable, which supports better corporate governance and helps the organization to demonstrate compliance with various regulations.

The ALM Industry
Initially, Borland and IBM Rational were among the few innovators that recognized the importance of the ALM trend, and shifted their product strategies to explicitly support it. Reacting to the evident opportunity, more companies such as Microsoft, Telelogic, Mercury and Serena jumped on the ALM bandwagon.

Today ALM is an established trend and a growing industry, which is recognized by industry analysts. ALM vendors provide a wide array of tools and technologies to support the process of software development. These tools go well beyond the traditional focus on individual developer productivity, and attempt to deliver a team-oriented methodology and tooling for software delivery.
To deliver a viable ALM solution, vendors must address the “extended” application development team, and include roles that participate in the wider process:

- Executive needs are addressed with portfolio-level dashboards that surface important project metrics such as risk, progress and quality.

- Project managers’ needs are addressed with tools for project planning and tracking, tradeoff analysis and resource allocation.

- Analysts’ needs are addressed with tools to facilitate requirements definition, interaction with end users and other stakeholders, and the management of requirements throughout the project lifecycle, including changes over time.

- Architects’ needs are addressed with tools to facilitate visual modeling of various application aspects (components, data, process) as well as tools for describing design patterns and enterprise architectures.

- Developers’ needs are addressed with sophisticated coding environments, as well as with code-level quality tools, such as performance profilers, unit-testing frameworks and automated code audits.

- Quality assurance engineers’ needs are met with tools for test creation and management, automated regression and functional testing, and automated performance testing.

- The needs of the overall team are addressed with team-wide infrastructure that provides facilities for collaboration, process guidance, change management and version control.

- Software process managers’ needs are addressed with tools for modeling and deploying a set of enterprise-wide process standards.

- The needs of end users and business stakeholders are addressed with tools that automate demand management and provide self-service capabilities around communicating requirements, reporting defects and tracking their delivery status.

ALM is widely recognized as a huge leap forward for the application development tools industry and for its customers. Interestingly, the latest Chaos report from the Standish Group indicates that failure rates of software projects have decreased to about half compared with a decade ago, an improvement that can be partly attributed to the emergence of ALM. However, deeper investigation of customer needs reveals that despite the obvious benefits of ALM, its full potential is still difficult to realize without changing the fundamental approach used to integrate processes and tools that are used across the software lifecycle.

**ALM Business Value is Largely Unrealized**

To better understand why current solutions make it difficult to unlock the full business value of ALM, let’s take a closer look at typical software development and operation environments. We
will examine how software is produced and deployed in terms of processes, development tools and runtime platforms. Ultimately, this discussion explains why software delivery remains one of the last business processes not being performed—let alone automated—in a consistent and predictable fashion.

**The Enterprise IT Environment: A Case Study in Heterogeneity**

The introduction of the Internet and its adoption as a major commerce platform, as well as constant pressures to operate in a lean and agile manner, have caused major changes in the average enterprise IT organization. The crux of these changes revolves around an architectural evolution, which is designed to progress IT responsiveness, level of service and efficiency, through migration from legacy technologies into modern application platforms. The key areas of evolution are:

- Migration from monolithic, mainframe-based custom applications to new development done on enterprise distributed platforms, namely J2EE™ and .NET
- Migration from packaged enterprise applications built on legacy architectures to process execution and composite application frameworks, such as SAP NetWeaver and Oracle® Fusion
- Adoption of specialized platforms for specific needs, such as scripting languages for dynamic, database-centric Web applications (PHP, Ruby, and so on) or platforms for the development of rich Internet and media applications (for example, Adobe® Flash®/Flex™)

Each of these technologies is associated with specific application development tools (often offered by multiple vendors), which cover analysis, design, coding, quality assurance, version control and configuration management.

It is reasonable to assume, especially for medium to large-size corporations, that in the foreseeable future every enterprise IT environment will include a combination of at least three of these deployment targets: mainframe, distributed (J2EE or .NET) and business process runtimes (SAP or Oracle). It is also likely (as is becoming increasingly evident) that some organizations deploy software to both J2EE and .NET.

**Conflicting Agendas**

It is interesting to note that for obvious reasons some IT vendors attempt to influence the heterogeneous nature of enterprise IT as much as they can. These vendors aspire to completely “own” the IT organization by pushing cradle-to-grave solutions, which include software development tools, application runtime environments, as well as network and system management tools. The largest vendors include the operating system or even the hardware as part of their solution. It goes without saying that such solutions include a significant component of professional services.

Despite this massive promotion of comprehensive single-vendor stacks, the reality is that many customers simply cannot adopt this approach. Such organizations promote heterogeneity at all
levels, and therefore have a different set of priorities, which emphasize objectives critical to the customer (rather than to the vendor):

- Maximize competitive advantage—organizations that strive to deliver the best product or service tend to cherry-pick best-of-breed platforms and development tools based on project fit, to gain specific end-user advantages provided by each platform. This often happens in separate projects, but may as well happen in the context of a single project, resulting in “hybrid” applications that span multiple technology domains. Some relevant examples include:
  - Composite applications or services that wrap mainframe, packaged applications and homegrown distributed applications
  - J2EE/.NET hybrids that leverage the power and UI experience of .NET on the client side, and the scalability, manageability and security of J2EE on the server side. This architectural pattern is particularly common in the financial vertical, and is used for high-performance trading platforms, given that Windows® is the de facto standard desktop of Wall Street
  - Flash/J2EE hybrids that combine the power of Adobe Flash as a RIA and video streaming platform and the J2EE server-side advantages to achieve highly scalable, rich multimedia experience

- Cut development costs—organizations attempt to reduce the cost of software development and deployment by utilizing a combination of homegrown and open source tools and runtimes. In this context, it is worth mentioning the growing popularity of the LAMP stack (Linux, Apache, MySQL, PHP), and its increasing adoption in the enterprise.

- Decrease time to market—organizations may prefer certain development tools based on specific productivity enhancements that they incorporate. These have the potential to dramatically reduce time to market.

- Leverage legacy investments—any rip-and-replace approach has a significant barrier, since most organizations are unwilling to give up the significant investments made in older runtimes and tools.

- Reduce risk—some IT vendors provide nonstandard proprietary support to their platforms, which are viewed as risky in the eyes of their customers. Getting locked into your IT vendor platform may result in a significant business risk, especially if that IT vendor is or will become a competitor.

**IT Heterogeneity: The Biggest Challenge for ALM**

To summarize, many IT organizations view heterogeneity as the only alternative because of the many business advantages associated with it. More often than not, development teams use a wide variety of tools that were not designed to interoperate. There is no single vendor that provides tooling to cover all activities necessary in the context of a single software project. Further, there is no single vendor that can completely cover the three primary domains of legacy maintenance and modernization, packaged applications extension and customization,
and new development of distributed applications. Therefore, it is likely that organizations will continue to use diverse development tools within the same project and across different technology domains.

For this reason, ALM’s biggest challenge is development tool heterogeneity. To recall, ALM strives to achieve consistent and predictable software delivery through automated measurability, alignment and discipline. However, these qualities of software delivery become much harder to achieve in a highly heterogeneous environment:

• While measurability requires harvesting metrics across disparate application development tools and repositories, there is no adopted standard that facilitates such data aggregation. Since no common information schema is available for all tools that participate in the process, it also becomes essential to “normalize” harvested metrics and to be able to correlate them to the context of specific projects.

• Alignment requires tracing deliverables and activities all the way from IT strategies down to deployed modules. This degree of traceability is very hard to accomplish when process assets and activities reside in disparate tools and repositories. There is no standard facility that enables automatic definition, collection, management and utilization of traceability information.

• Discipline requires deploying, enacting and monitoring multiple overarching processes to govern software delivery. This becomes much harder when sub-processes reside as “process islands” within a variety of process-enabled tools. No standard mechanism exists to provide choreography of such sub-processes (according to a higher-level process) or to deploy process components into these tools. There is also no common terminology to describe processes across disparate tools, which all use their own languages of “Items,” “Artifacts,” “Projects,” etc. The other aspect of discipline calls for rigorous change management and impact analysis; however, these capabilities require end-to-end traceability to be properly realized. As already discussed, end-to-end traceability is much harder to achieve in a heterogeneous development environment.

To overcome these challenges, organizations that practice ALM often end up developing multiple ad hoc point-to-point integrations, which fill the process gaps between the various development tools that they use. Such integrations are fragile, break with upgrades or changes of tools, and are costly to build and maintain. They also result in software processes that cannot be easily managed, measured and audited. Obviously, this approach is unsustainable and not cost-effective.

Therefore, most IT organizations pose big challenges for ALM vendors. These organizations would like to get the huge value associated with ALM, namely a dramatically improved software delivery process that yields the required consistency and predictability. However, on top of that ALM customers also want:
• To be able to use a mix of runtime platforms in a manner optimized to their business objectives

• The freedom to use a mix of commercial and open source application development tools, which are optimized to the deployment targets that they decide to utilize

• The freedom to use a variety of commercial or custom software development processes that are optimized to organization culture, project types and underlying technology

To address this challenging set of requirements, a new approach for ALM is needed, an approach that enables customers to unlock the full value of ALM on top of a heterogeneous IT environment. Borland’s recently announced Open ALM vision and product strategy is directly aimed at addressing this challenge. It is the only ALM solution that is fundamentally designed to enable IT organizations to predictably deliver software on their own terms.

**Conquering Heterogeneity: The Final Frontier of ALM**

Open ALM advances Borland’s established vision and product strategy. It represents a major architectural shift that is unique in the commercial ALM market. In fact, when fully realized, the Borland Open ALM platform and its associated applications could bring tremendous value to customers that don’t use even a single Borland application development tool.

To be sure, Borland views its tools business as vital, and will continue to innovate and deliver best-of-breed tools to the extended software development team. Borland’s tools will be gradually migrated to support the Open ALM strategy, which will enable them to participate in an Open ALM–based orchestration of software delivery. However Borland’s tools could be replaced, if customers see fit, with any third-party or open source tool that supports their development needs. This level of modularity and flexibility is what makes Borland’s product strategy exceptional among ALM vendors, many of whom attempt to own the complete software delivery chain.

**Benefits of Open ALM**

Open ALM provides the functional value of ALM while introducing unprecedented levels of flexibility at the process, tools and platform levels. More specifically, Open ALM customers would:

• Be free to choose any combination of platforms and runtime environments in the context of a single software project or across different projects, based on business priorities and project fit

• Be free to choose the best development tools for chosen platforms, per economic considerations, specific productivity enhancement and technical fit

• Be free to choose or design development processes that are best fit for their projects and chosen platforms, as well as match their organizational culture and time-to-market needs
The Open ALM platform and its supporting tools will, for the first time, enable IT organizations that deploy heterogeneous application development environments to:

- Gain unparalleled, multidimensional and customizable visibility into progress, quality and risk metrics of projects and portfolios, to support project management and process improvement initiatives

- Reach the holy grail of full lifecycle traceability to support true alignment of business objectives and development activities, better correlation between end-user expectations and project outcome as well as better project management through accurate and comprehensive impact analysis

- Achieve a new level of control over software delivery through automated process-driven orchestration of practitioners and tools that participate in the lifecycle

These capabilities enable superior team productivity, support quality initiatives and ease the burden of compliance with internal and external regulations. They will be delivered in a set of infrastructure-level components and enterprise ALM management tools. On top of that, customers can also use Borland’s best-of-breed integrated application development and portfolio management tools to realize the value of four core process areas:

- **Project Portfolio Management (PPM)** – tools and automated processes to control the development of a top-down software delivery strategy, and to manage the execution of a portfolio of software development projects

- **Requirements Definition and Management (RDM)** – a set of tools and best practices to ensure that project requirements are accurate and complete, can be effectively traced back to business objectives and are optimally covered by software tests

- **Lifecycle Quality Management (LQM)** – discipline and tools to govern the definition and measurement of quality across all phases of software delivery. It is designed to detect and prevent quality problems early in the process, when the cost of fixing is relatively low, and to enable QA teams to ensure their tests are complete and based on end-user requirements

- **Change Management (CM)** – infrastructure and tools to help predict the impact of changes and to facilitate management of assets and activities related to the changes across the lifecycle, in single-site or multisite delivery models

**Borland’s Open ALM Solution**

As previously mentioned, the major objective of ALM is to achieve predictable and managed software delivery through automated measurability, alignment and discipline. We have seen that each of the three dimensions of ALM becomes much more difficult to accomplish in a heterogeneous application development environment, and therefore presents a set of specific problem areas to ALM customers.
Borland’s Open ALM platform is architected as a set of three solution areas, each of which is intended to specifically address one of the major ALM problem domains. Every Open ALM solution area is based on a highly modular and extensible infrastructure layer and delivers its business value through a set of dedicated applications. The purpose of the infrastructure layer is to enable the Open ALM platform and applications to work with any combination of commercial or open source development tools and processes, regardless of vendor or expected runtime technology. The diagram on the following page provides a conceptual decomposition of Borland’s ALM solution.
Open Business Intelligence for ALM (OBI4ALM) is based on a standard infrastructure and applications to increase measurability of progress, quality productivity or any other custom metric of software projects in a heterogeneous application development environment.

OBI4ALM provides infrastructure for unobtrusive, distributed collection of data, correlation and analysis of metrics from any application development tool that registers itself with it. By pulling predefined metrics from its data sources, the OBI4ALM infrastructure unifies information silos that are scattered across the software delivery cycle. Such consolidation provides powerful capabilities, such as an aggregated project view of metrics and the definition of new project metrics that combine several lower-level ones.

The OBI4ALM infrastructure employs a data warehouse, which stores current and historical information harvested from tools that participate in the various phases of the software delivery process, using a structure that is optimized for querying and analysis.

OBI4ALM applications are capable of transforming the collected metrics into actionable information, enabling decision making and early awareness of problems:

• Real-time dashboards—customizable views of key performance indicators, showing trends over any time period
• Metrics-based alerts—configurable notifications that get triggered under certain conditions (i.e., when a certain threshold is crossed). Alerts help to accelerate management responsiveness to various project problems, such as slow progress, low quality, insufficient productivity or any other problem that can be quantified using metrics
• Decision-making tools—analytical tools that use historical project and cross-project information to facilitate various project management decisions

Open Process Management for ALM
In the final analysis, process is the most important concept that permeates the software lifecycle. Far more than sharing information structures across tools used by various roles, or providing UI-level integration of features, it is the process that has the true potential of coordinating activities of humans and systems that participate in the delivery of software, while at the same time ensuring conformance to agreed-upon policies and monitoring quality of execution.

Open Process Management for ALM (OPM4ALM) provides infrastructure components and a set of applications used to model, deploy and enact multiple software processes across an entire heterogeneous application development environment. Going way beyond providing guidance and allocating tasks to process practitioners, OPM4ALM also uses the process automation layer as the primary "glue" for integrating client side, server side and methodware, according to rules captured in the process models. From this point of view, the integration among application development tools is in fact driven by the underlying processes, which becomes the fundamental enabler for effective team productivity.
OPM4ALM Infrastructure is built around a distributed process engine, which enables the modeling, tailoring, deployment, orchestration and choreography of multiple software delivery processes, across a heterogeneous environment of development tools. Process events definition and management is an important part of OPM4ALM infrastructure, as Open ALM tools can subscribe to “listen” to such events and receive notification when they occur. The process engine also enables the flexible definition and evaluation of rules, which help to describe and enforce application development policies.

OPM4ALM applications surface the business value of the process infrastructure layer. They provide:

- Tools for process modeling, customization, tailoring and reuse, which enable effective design of commercial or custom software processes using a rich and capable software development meta-model

- Enterprise-wide software process console that shows a consolidated bird’s-eye view of entire software processes that are deployed in various projects, across disparate development tools

- Process compliance dashboard that exposes process deviations and their potential implications, and provides reporting features useful for compliance initiatives

- Measurement and reporting based on metrics specific to each process

**Open Traceability for ALM**

End-to-end process traceability supports many important ALM benefits. To name a few, it is a critical enabler for business-driven development, requirements-based development and testing, and accurate change impact analysis.

Open Traceability for ALM (OT4ALM) provides infrastructure to create and classify relationships between assets created in the software delivery process, regardless of the tool that hosts them, establishing a flexible graph of asset links. It also provides the means to navigate the asset relationship graph and to optimally query and mine data that is captured in it.

OT4ALM provides applications that transform captured traceability data into actionable information:

- Automated planning, change impact analysis, accurate cost/budget predictions

- Scope monitors—early alerts for scope deviations (i.e., assets that do not trace to requirements) and unimplemented requirements

- Reuse analyzer—enables reuse of complete asset trees (from requirements to models to code and tests) rather than simple reuse of code modules
• TraceView—cross-project interactive traceability viewers that help locate every process asset and relate it to other assets

Common Platform Infrastructure
The Open ALM infrastructure includes two components that are shared across all solution areas:

• ALM meta-model – a common language to describe software processes, process asset relationships (traceability) and measurement units (metrics). The ALM meta-model provides a rich conceptual model for the software delivery domain. It is essential for describing the standard vocabulary that all Open ALM–compliant tools must understand in order to effectively participate in the Open ALM platform.

• ALM integration layer – an extensible and pluggable integration mechanism and SDK, which defines the standard manner in which ALM tools can be invoked, ALM metrics can be harvested and asset traceability graphs can be navigated. To support and participate in the ALM platform, a tool needs to provide a platform plug-in that conforms to the Open ALM–standard API, or utilize a custom adapter that connects it to the rest of the application development environments through processes orchestrated by the Open ALM platform.

The Road to Open ALM
Over the next 24 months, Borland will incrementally roll out the infrastructure, applications and tools that comprise its Open ALM platform. Borland also intends to round out its product offering with a comprehensive set of professional service programs, designed to accelerate the deployment and ensure the success of enterprise Open ALM implementations.

Some of the advantages of Open ALM can be enjoyed by customers today. Organizations that seek to improve their quality, change management and project management processes, will find the current Borland solution extremely compelling. This solution provides highly automated and integrated support for four critical application development process areas:

• Project and Portfolio Management (PPM)

• Requirements Definition and Management (RDM)

• Lifecycle Quality Management (LQM)

• Change Management (CM)

These solutions are delivered through tight integration between Borland tools and third-party tools, which gives customers the desired flexibility that they seek while significantly improving their ability to manage software delivery projects today.

Why Borland?
Throughout its long history of innovation, Borland has consistently partnered with its customers to enable them to build software the way they see fit. With its uncompromising adherence to standard-based development and broad multiplatform support, Borland has offered IT organizations the flexibility and freedom to choose what they require. With Open ALM, Borland elevates its traditional values to a whole new level, which clearly separates it from the pack of ALM vendors and non-commercial ALM initiatives.

When it comes to the largest ALM vendors, IBM Rational and Microsoft, it can hardly be claimed that serving the customer agenda is of top priority, as both of these vendors continuously attempt to leverage their development tools to lock customers to their middleware and system management platforms.

In contrast, Borland has always insisted on standard Java™ and J2EE support, had strong and integrated support for Microsoft’s platform, languages and development tools, and continues to be very committed to extending Microsoft’s ALM solution in meaningful ways. The investment made by Borland to support the latest Microsoft® technologies is very significant. For example, CaliberRM™, which is the first fully integrated requirements management solution for Team System, is recommended by Microsoft to complement the basic requirements functionality delivered by VSTS. Borland plans to continue to enhance the synergies between the Java and .NET platforms by providing additional capabilities such as UML to C# code generation and support for Microsoft Domain Specific Languages (Microsoft’s alternative to UML).

The open source movement has also identified the challenge that heterogeneity creates for ALM. The objectives of several Eclipse initiatives, Application Lifecycle Framework (ALF), Corona, and the Eclipse Process Framework (EPF), are somewhat aligned with those of Borland Open ALM. While Borland understands and identifies with the motivation behind these projects, it feels that their approach is insufficient. Both ALF and Corona are attempting to deliver components of the Open ALM infrastructure only. However, Open ALM represents a more holistic approach, since it also enables customers to extract business value from such infrastructure out-of-the-box, through a set of value-add applications.

In its quest toward Open ALM, Borland goes further than any other ALM vendor, and has recently expanded its horizons to cover additional application development domains. Borland is also investigating the best approach to support packaged application development projects on the SAP NetWeaver and Oracle Fusion platforms.

Conclusion
Borland is uniquely positioned to help ALM customers build software on their own terms. The Open ALM vision and product strategy clearly differentiates Borland from other ALM vendors as well as from open source initiatives. Borland is the only major ALM vendor that genuinely accepts the reality of IT heterogeneity, and attempts to enable ALM adopters to leverage their existing investments in development processes, runtimes and tools. Borland’s process-driven
integration approach further separates Borland from its peers, enabling it to deliver ALM-wide visibility, traceability and discipline.

As Borland begins to roll out the Open ALM infrastructure, applications and compliant development tools, customers will be able, for the first time, to unlock the full value of ALM, and experience the benefits of a fully connected, managed and measurable software delivery process.

Safe Harbor Statement
This release contains “forward-looking statements” as defined under the U.S. Federal Securities Laws, including the Private Securities Litigation Reform Act of 1995 and is subject to the safe harbors created by such laws. Forward-looking statements may relate to, but are not limited to, the features available in, and the potential benefits to be derived from, Borland products and solutions, plans and market acceptance of such products and solutions, including the Borland Open Application Lifecycle Management solution. Such forward-looking statements are based on current expectations that involve a number of uncertainties and risks that may cause actual events or results to differ materially. Factors that could cause actual events or results to differ materially include, among others, the following: rapid technological change that can adversely affect the demand for Borland products, shifts in customer demand, shifts in strategic relationships, delays in Borland’s ability to deliver its products and services, software errors or announcements by competitors. These and other risks may be detailed from time to time in Borland periodic reports filed with the Securities and Exchange Commission, including, but not limited to, its latest Annual Report on Form 10-K and its latest Quarterly Report on Form 10-Q, copies of which may be obtained from www.sec.gov. Borland is under no obligation to (and expressly disclaims any such obligation to) update or alter its forward-looking statements whether as a result of new information, future events or otherwise. Information contained in our website is not incorporated by reference in, or made part of this press release.
Why C++ is not just an Object-Oriented Programming Language

Bjarne Stroustrup
AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

C++ directly supports a variety of programming styles. In this, C++ deliberately differs from languages designed to support a single way of writing programs. This paper briefly presents key programming styles directly supported by C++ and argues that the support for multiple styles is one of its major strengths. The styles presented include: traditional C-style, concrete classes, abstract classes, traditional class hierarchies, abstract classes and class hierarchies, and generic programming. To provide a context for this overview, I discuss criteria for a reasonable and useful definition of “object-oriented programming.”

1 Introduction

There are many tools and techniques that can help in our effort to build useful, economical, and maintainable systems. To complete ambitious and complex projects, we rely on a wide variety of techniques and tools that must work together.

The title of this paper singles out a programming language†. However, the real topic is programming, or if you prefer a longer formulation, the design and implementation of systems. A programming language is just one of the means by which we try to achieve our goals.

The definition of “object-oriented programming” is no longer a popular topic of discussion at major conferences. A practical definition of “object-oriented programming,” “object-oriented analysis,” “object-oriented design,” “object-oriented technology,” etc., is, however, a burning issue for people who want to turn the oft-repeated promises made for techniques and languages called “object-oriented” into reality in everyday projects. It has become a practical rather than academic topic of discussion. What is “object-oriented technology?,” what benefits can be expected from it? At what risks?, how do those techniques, benefits, and risks compare with those associated with alternatives?

A systems builder trying to explain to an accountant why money should be spent for tools supporting object-oriented techniques needs more than a statement to the effect that “object-oriented is great” or that “really great techniques are really object-oriented.” You simply cannot ask someone to bet their company’s future on vague promises phased in ill-defined
terms. Nor is a well-polished and logically coherent semi-mathematical treatment of the subject of direct practical use.

We need to define “object-oriented” to be something specific so that we can point out specific benefits and risks associated with its use. We must also be specific about what is not object-oriented, and what benefits and lack of benefits we can expect from various non-object-oriented techniques.

Consequently, this paper starts out discussing what makes a good definition of “object-oriented.” Next, I present a range of useful techniques which may or may not be object-oriented and discuss their advantages and disadvantages.

2 Defining “Object-oriented”

To be useful and intellectually honest, a definition of “object-oriented” must

[1] not be a mere synonym for “good,”

[2] not exclude most accepted meanings,

[3] have a firm historical basis,


Not everything good is object-oriented, and not everything object-oriented is good. I think I can support both claims from experience. I have seen examples of the latter often enough: it is not uncommon to find programs that apply techniques usually deemed object-oriented extensively or even exclusively, yet are hard to comprehend, hard to maintain, and perform abysmally. Such examples occur in every programming language. But then, of course, some people respond “that just proves that the program wasn’t truly object-oriented.” To which the answer must be that either the term has become meaningless or there must be something good beyond what is called “object-oriented.”

On the other hand, when we define “object-oriented,” we must not be too exclusive. Object-oriented programming is a broad intellectual discipline, not the mere use of specific language features. Attempts to define “object-oriented” to mean “what I’m selling” are not uncommon, but are fundamentally sleazy.

Any definition of “object-oriented” should be historically reasonable. Words are only useful for communication, really only mean something, if we agree on a meaning for them. There are several plausible, logically coherent, and mutually contradictory definitions of “object-oriented” in use. However, the mainstream usage stems directly from the ideas pioneered by programming language Simula and the design techniques it was developed to support. The communities of programmers and designers centered around languages such as C++, CLOS, Eiffel, Object Pascal, and Smalltalk have contributed much to this tradition.

A meaningful definition of any concept must exclude something.

3 A Broad Definition of “Object-oriented”
Given these general criteria for a definition of “object-oriented” you can find several plausible candidates, and several communities have their own definitions. However, I suggest we stick to the traditional definition of object-oriented used within broad communities of programmers. A language or technique is object-oriented if and only if it directly supports:

1. Abstraction – providing some form of classes and objects.
2. Inheritance – providing the ability to build new abstractions out of existing ones.
3. Runtime polymorphism – providing some form of runtime binding.

This definition includes all major languages commonly referred to as Object-oriented: Ada95, Beta, C++, CLOS, Eiffel, Simula, Smalltalk, and many other languages fit this definition. Classical programming languages without classes, such as C, Fortran4, and Pascal, are excluded. Languages that lack direct support for inheritance or runtime binding, such as Ada88 and ML are also excluded. ML is a good example of something that is good but not object-oriented. I like ML; it is an interesting, innovative, and powerful language, but it is functional rather than object-oriented and its polymorphism is resolved at compile time rather than at runtime. Thus, saying that ML isn’t object-oriented is not a criticism, it’s an observation about definitions and the nature of ML.

Techniques and tools are object-oriented if and only if they support the use of object-oriented programming. For example, a design method is object-oriented if its regular and proper use leads to programs that exploit abstraction, inheritance, and polymorphism where appropriate. I strongly prefer design methods that directly and naturally support the use of at least one of the major object-oriented languages supporting in ways that exploit its features in an idiomatic way.

For example, it is often possible to simplify application code by hiding objects with different representations and different implementation details behind a common “abstract” interface (see §6.4 and §6.6). Conversely, the implementation of related concepts can often be greatly simplified by exploiting commonality through inheritance (see §6.5 and §6.6). A major purpose of design methods and the CASE tools that commonly support them is to make design simpler, more regular, and more predictable. Thus, to earn the label “object-oriented,” a design method must regularly and predictably help the discovery of commonality that can be exploited in these ways. Ideally, an object-oriented design method must strongly encourage the expression of this commonality using the most appropriate facilities in one or more of the languages supporting object-oriented programming. Minimally, the method and its supporting tools must not be a hindrance to the use of object-oriented facilities in the programming language used to implement the design. Much confusion arise because not every design method that claims to be object-oriented does that.

Please remember that I’m looking for a practical understanding of the notion of “object-oriented” rather than a formal definition. A formal definition is useful, indeed it may be essential. However, to be relevant, a formal definition must match a coherent view of what the formal definition is meant to specify precisely.
4 Purity
There has been much debate about "purity," in the context of languages supporting object-oriented programming. In my opinion, much of that discussion is confused by the – often unstated – assumption that not only does "object-oriented" imply "good," but also by the further assumption that only "object-oriented" features are good. Consequently, it is – wrongly – assumed that a language that provides features deemed non-object-oriented must be worse than a language that does not. People who like a language to support classes as part of a hierarchy only and functions/methods attached to one specific class only often call such a language "a pure object-oriented language." If we don’t like the idea of restricting the definition of classes and functions that way we can call such a language "just an object-oriented programming language."

I prefer to have more facilities available than can be provided by methods defined on classes within a single hierarchy. A lot of good design goes beyond that relatively narrow domain. Incidentally, I have come to dislike the adjective "hybrid" as used to distinguish "pure" object-oriented systems from others. Too often, "hybrid" is used in a prejudicial manner. If I must apply a descriptive label, I use the phrase "multi-paradigm language" to describe C++.

4.1 Use of Language Features
Even when all the features required to support object-oriented programming are available, you don’t need to use them all the time. Some classes just don’t belong in a hierarchy and some functions don’t belong to any particular object.

The key to maintainable, efficient, and evolvable programs isn’t particular language features. It is the ability to develop concepts needed for a solution and to express them clearly in a program. Language features exist to make such expression simple and direct.

Object-oriented programming can be done in a language lacking one or more of the features required to directly support object-oriented programming. However, doing so is unnecessarily difficult, very difficult to support with tools, and often prohibitively expensive.

Furthermore, there are things that can’t be expressed directly using only the "pure" object-oriented constructs mentioned above. For example, some entities belong together, but their relationships are not hierarchical. Some entities simply do not obey the rules of a particular object-oriented language. Some things that you build in an object-oriented world are manipulated from the outside so that it is difficult to make guarantees about the way they are used.

5 C++ Design Ideals
I felt the need for facilities outside what is conventionally called "object-oriented," so I supplied some in C++. However, C++ isn’t meant to be everything to everybody. No one programming language and no one view of how to write programs is sufficient for everything. Constraints-based programming, logic programming, functional programming, and various
forms of concurrent programming are examples of good and useful styles of programming not supported by C++.

No single language can support every style. However, a variety of styles can be supported within the framework of a single language. Where this can be done, significant benefits arise from sharing a common type system, a common toolset, etc. These technical advantages translate into important practical benefits such as enabling groups with moderately differing needs to share a language rather than having to apply a number of specialized languages.

C++ was designed to support a range of styles that I considered fundamentally good and useful. Whether they were object-oriented, and in which sense of the word, was either irrelevant or a minor concern:

[1] Abstraction – the ability to represent concepts directly in a program and hide incidental details behind welled-fined interfaces – is the key to every flexible and comprehensible system of any significant size.

[2] Encapsulation – the ability to provide guarantees that an abstraction is used only according to its specification – is crucial to defend abstractions against corruption.

[3] Polymorphism – the ability to provide the same interface to objects with differing implementations – is crucial to simplify code using abstractions.

[4] Inheritance – the ability to compose new abstractions from existing one – is one of the most powerful ways of constructing useful abstractions.

[5] Genericity – the ability to parameterize types and functions by types and values – is essential for expressing type-safe containers and a powerful tool for expressing general algorithms.


[8] Static type safety – an integral property of languages of the family to which C++ belongs and valuable both for guaranteeing properties of a design and for providing runtime and space efficiency.

These facilities and general properties can be supported in several alternative ways. For example, one programming language may support a facility in its core language where another supports it in a library. Similarly, a facility provided by a runtime mechanism in one language may be provided by a Compile-time mechanism in another.

The requirement for coexistence is essential for any language claiming to be general-purpose. Looking at the world from the perspective of a given programming language, we find that almost every real-life system contain parts that are written in others languages and designed according to principles foreign to that language. To be general-purpose, a language must somehow take the unpredictable, ugly, and constantly changing demands of program fragments written in “other languages” into account.
To be genuinely general purpose, a language must possess facilities that allow it to share data with program fragments written in other languages, to invoke code fragments written in other languages, and have code invoked by code written in other languages. For example, systems relying on callbacks can be rather ugly to program, but not being able to use such systems in a direct and idiomatic way would be crippling for a language as a tool for real-world programming. In many languages, a common use of “foreign” code is exactly to violate the languages rules: to do things that can’t be done – or can’t be done efficiently – in the language itself.

Alternatively, access to facilities in ”the outside world” could be carefully fitted into the framework of the object-oriented programming language through special facilities in the runtime environment or in libraries. However, accessing facilities in the manner they were meant to be used is often easier and less awkward than to fit them into our language framework. A general mechanism for accessing “foreign” code also leads to more extensible systems than a requirement to fit each individual “foreign” facility into the language framework.

Over the years, we have seen spectacular improvements both in hardware performance and in compilation techniques. However, runtime efficiency and compact representation is still absolutely essential to many people.

Static type safety is an essential part in my view of both design and implementation (see, for example [Stroustrup,1991]). The guarantees provided and the discipline of design imposed have been found extremely valuable by many people working in a wide range of application areas. Static type checking is of course not a panacea, but it is something I would not attempt major projects without.

The fundamental ideal of C++ is actually the fundamental ideal for a lot of languages: center box; c. Represent concepts and relationships between concepts directly and affordably.

Naturally, there are many ways of approaching this ideal. It is worth remembering that all of the languages usually mentioned in a discussion of practical use of object-oriented techniques are suitable vehicles for good design. A rational discussion of languages is one of relative merits, applicability to specific problem areas, and personal preferences, rather than one of absolutes. Representing concepts directly is a restatement and possibly a generalization of ideas relating to data abstraction and information hiding. Representing hierachical relationships is the traditional key to object-oriented programming. There are, however, clean and useful relationships that are not hierarchical yet can still be represented directly in a program (for example, see §6.3 and §6.7).

Being more concerned with producing good software than with finding the most elegant expression of ideas in the abstract, I insist on affordability. Affordability is a multi-facetted issue that involves not only runtime efficiency, but also availability of suitable hardware, availability of designers and programmers comfortable with new techniques, etc.
6 Programming Style and Language Features
I will now give examples of programming styles and language features supporting them. Some are commonly referred to as “object-oriented,” some are not, but that doesn’t prevent me from recommending them in some contexts.

6.1 Conventional Notation
There are aspects of conventional code and conventional notation that I would like to see maintained even in a strictly object-oriented overall design. Being able to say plain square root of two, sqrt(2) is nice, and so is the ability to write x+y*z and know that it means add x to the product of y and z. We have about 400 years of experience with such notation and it is deeply ingrained in our technical culture.

6.2 Concrete Types
Very simple concepts, such as integers, floating point numbers, complex numbers, points, lines, pairs, dates, disk locations, bcd characters, error messages, currency, are usually not considered suitable topics for discussion in academic articles or at conferences. These are usually considered too simple to merit discussion. However, the mundane is often statistically more significant than the sophisticated.

Provided they can be implemented in a way that is simple, elegant, efficient and flexible enough, I consider such simple concepts excellent candidates for independent proper types – as opposed to presenting them to users as plain data structures or as parts of a larger class hierarchy. Consequently, part of a design effort should focus on these little abstractions. These very concrete types should be designed carefully and supported well.

To illustrate why, I’ll contrast this approach to the use of a plain data structure and to the use of class hierarchies. To make this discussion concrete, I’ll use the example of a date.

6.2.1 Structures and Functions
The simplest way of presenting a date in a program is simply to specify its data layout. For example:

```c
struct date {
    // representation
};
```

Given that, programmers can do anything at all with dates. That “anything at all” is the strength and weakness of this idea. Naturally, a “standard” set of functions is usually provided to manipulate a structure such as date. However, such a set of functions is rarely complete, and even when it is, programmers find reasons to manipulate dates directly. Consequently, it is usually not possible to change the definition of date after the initial release of the software; it is simply too difficult to track down every use of a date and modify it use to the new definition. The reason that the set of functions is rarely complete is that there is no incentive to make it so.
A programmer can always write new functions accessing the date structure, and the dominant culture encourages the programmer to do so. Writing a new function that is “just right” for the job, carries no overhead, and relies on no potentially untrustworthy code is often considered better than improving a standard and general-purpose set of access functions and using it. Often, it is also far easier. This trend is typically reinforced by poor documentation.

6.2.2 A Concrete Class
A simple Date type can remedy most of the problems related to using a data structure directly. Consider:

```cpp
class Date {
public:
    // public interface, consisting
    // of nonvirtual
    functions
private:
    // representation and other
    // implementation details
};
```

Such a Date type will provide
[1] constructors specifying how objects of the type are to be initialized;
[2] functions for examining a Date; these functions will be explicitly declared not to modify the value of the object;
[3] functions for manipulating Dates without actually having to know the details of the representation or fiddle with the intricacies of the implementation.

In addition, Dates can be freely copied.

This set of member functions supplied as members of Date should be those that provide a basic semantics for a Date and also requires direct access to the representation of Date to be implemented.

The set of member functions should be almost minimal; many operations that users would find convenient can be supplied separately (see §6.3). I dislike classes with dozens or even hundreds of member functions. Such a class does not represent a well-thought-out concept; it’s a glorified data structure produced by somebody who couldn’t decide on what was really wanted.

The member functions are declared non-virtual to ensure that there is no time or space overheads involved in using this Date, and to ensure that the semantics of Date cannot be modified later. Similarly, the representation of Date is declared private to prevent access by any function not explicitly mentioned in the class itself.
The representation of a concrete type should be compact. Sometimes millions of objects of such classes exist, and even with modern memory sizes space overheads can be a burden. If nothing else, reading and writing objects with bloated representations can be a nuisance.

The use of concrete types must be fast. In my world at least, programmers are prone to represent something as plain data structure out of fear of overheads supposedly associated with abstractions.

There are no time or space overheads associated with the Date class as defined above. The size is identical to that of the plain date structure, and inlining is done for simple member functions to make these as fast as the code a programmer would write accessing a plain structure directly.

Often, it is important that simple concrete types, such as Date, be layout-compatible with simple data structures, such as date, as used in traditional languages. This allows simple exchange or sharing of information with code written in traditional languages. This can be a major convenience if your operating system, your database, or your high-performance numeric library is written in a traditional language and requires manipulation of data of a specific layout.

The Date class is very simple and very basic. It requires no elaborate framework, no class hierarchies, no clever dispatcher to mediate access, etc. It doesn’t affect the overall structure of a program much; it just provides a lot of help at the detailed programming level – below the level of detail of interest to most managers and to many designers.

If such types are that simple, why bother spending time on them?
[1] The concepts best represented by such simple types are common; most applications can use a few dozen or a few hundred such types. Thus, any benefits we get from a single concrete type, we get many times over.
[2] The problems relating to lack of encapsulation of plain structures (§6.2.1) are eliminated.
[3] The replication of effort writing simple access functions is eliminated.
[4] Making a concrete type the subject of a conscious design effort typically results in a better thought out, more comprehensive, and better documented concept. In principle, this could equally well be done for the plain structure approach, but in practice that typically doesn’t happen.
[5] Writing the basic functions of a concrete type is not difficult, but it is not trivial either. For example, adding a year to a date requires us to handle leap years. By relying on common access functions, we eventually achieve an implementation that has been better thought out and has fewer errors.
[6] Since more of the implementation is documented and shared, user code becomes more uniform. Thus, code written by others become easier to comprehend.

These are classical reuse benefits and I don’t think we should decline them just because they are easy to obtain.

6.2.3 Re-using Concrete Types
For many concrete types, derivation doesn’t make sense. Consider, deriving a new class from Date:

```cpp
class MyDate : public Date {
  // ...
};
```

Is it ever valid for MyDate to be used as a plain Date? Well, that depends on what MyDate is, but in my experience it is rare to find a concrete type that makes a good base class without modification.

Derivation from a concrete type is almost always a mistake. A concrete type is a self-contained Entity that can’t easily added to in a way that makes sense. A concrete type is “reused” unmodified in the same way as built-in types such as int are. For example:

```cpp
class Date_and_time {
  public:
    // ...
  private:
    Date d;
    Time t;
};
```

This form of use (reuse?) is usually simple, effective, and efficient.

Maybe it was a mistake not to design Date to be easy to modify through derivation? It is sometimes asserted that every class should be open to modification by overriding and by access from derived class member functions. This view leads to a variant of Date along these lines:

```cpp
class Date2 {
  public:
    // public interface, consisting
    // primarily of virtual functions
    protected:
    // representation and other
    // implementation details
};
```

Here, the functions are declared virtual, meaning that a class derived from Date2 (in the style of MyDate above) can provide its own versions. To make it possible to write such overriding functions easily and efficiently, the representation is declared protected. A protected member of a class is accessible not just to the classes’ own members, but also to the member functions of derived classes.
This achieves the objective of making Date2 arbitrarily malleable by derivation, yet keeping its user interface unchanged. However, there are costs:

[1] Efficiency of basic operations – a C++ virtual function call is a fraction slower than an ordinary function call, virtual functions cannot be inlined as often as non-virtual functions, and a class with virtual functions typically incurs a one word space overhead.

[2] Need to use free store – the aim of Date2 is to allow objects of different classes derived from Date2 to be used interchangeably. Because the sizes of these derived classes differ, the obvious thing to do is to allocate them on the free store and access them through pointers or references. Thus, the use of genuine local variables dramatically decreases.

[3] Inconvenience to users – to benefit from the polymorphism provided by the virtual functions, accesses to Date2s must be through pointers or references.

Naturally, these costs are not always significant, and as we will see in §6.4 the behavior of a class defined in this way is often exactly what we want. However, for a simple concrete type, such as Date2, the costs are unnecessary and can be significant.

Please note that the costs are fundamental; different languages present the facilities differently, but every language that provides runtime polymorphism incur these costs in some way or other.

Finally, a well-designed concrete type is often the ideal representation for a more malleable type. For example:

```cpp
class Date3 {
    public:
    // public interface, consisting
    // primarily of virtual functions
    protected:
    Date d;
};
```

### 6.3 Namespaces

Class hierarchies express (hierarchical) relationships, but not every relationship in a program can or should be expressed as a hierarchical relationship between classes. For example, if a class is intended for use only in the context of another class it can be declared a member of that class exactly the way a function can be:

```cpp
class Date {
    public:
    enum Month {
        jan, feb, mar,
        apr, may, jun,
        jul, aug, sep,
        oct, nov, dec
    }
};
```
More generally, C++ provides namespaces for grouping declarations [Stroustrup, 1994]. For example, many operations on Dates shouldn’t be members of class Date because they don’t need direct access to the representation of a Date. Providing such functions as nonmember functions leads to a cleaner Date class, but we would still like to make the association between the functions and the class explicit:

```cpp
namespace Chrono {
    // facilities for dealing with time:
    enum Month {
        // ...
    };
    class Date {
        // ...
    };
    int diff(Date a, Date b);
    bool leapyear(int y);
    Date next_weekday(Date d);
    Date next_saturday(Date d);
    // ...
}
```

A namespace is not a module; it is not an object. A namespace is a general scope mechanism to support a variety of techniques related to modularity. Not incidentally, namespaces provides a way of avoiding name clashes in software composed out of libraries from different suppliers. For example:

```cpp
namespace LibA {
    class String {
        // Astyle
        string
    };
    // ...
}
namespace LibB {
    class String {
        // Bstyle
        string
    };
    // ...
}
```
6.4 Abstract Classes

It is possible to completely disassociate implementation and interface. For example, we might implement a set using either an array or a list in such a way that the two kinds of sets can be used interchangeably:

```cpp
class set {
    // ...
};
class v_set : public set,
    private vector {
    // ...
};
class l_set : public set,
    private list {
    // ...
};
```

Or graphically:
```
vector set list
v_set l_set
```

I use the dotted lines to show that private inheritance is an implementation issue that does not affect the interface of the derived class.

Importantly, a common interface (here, set) can be provided long after the design and implementation of implementation classes (here, vector and list). I find that when people design things, they typically first invent something fairly concrete. They design an array, they invent a list, and only later do they discover an abstraction that covers both in a given context. Using abstract classes as shown above, we use (reuse?) vector and list without the foresight (and cost) necessary to design them as part of a common hierarchy.

As a matter of fact, you can do this “late abstraction” several times. Say, I want to represent the notion of “something you could read from.” This is a very different abstraction from set, yet I can provide such an interface to arbitrary sets as well as for lists, vectors, files, and input streams much in the way I provided set as an interface to vector and list.

Late abstraction using abstract classes allows us to provide different implementations of a concept even when there is no significant similarity between the implementations.

6.5 Classical Hierarchies
Sometimes we do have sufficient foresight to design a classical hierarchy. More importantly, sometimes the various implementations of a concept have a high degree of commonality so that there is significant benefit in organizing these implementations into a hierarchy. For example, consider a class hierarchy that one might find in an application relying on a windows system:

```
window
  ↓
 dial  slider
  ↑
  ↓ival_dial  ival_slider
```

Presumably, the implementations of the application classes ival_dial and ival_slider are greatly facilitated by code and data provided by the ”‘system classes” window, dial, and slider. That is, you build your code incrementally and your interfaces incrementally.

### 6.6 Hierarchies and Abstract Classes

A classical hierarchy is a nice way of providing a variety of related concepts and a nice way of minimizing the effort of building their implementations. However, you do get the classes in the hierarchy tightly coupled. If anything significant in a base class changes, all of the derived classes must change (or at least be recompiled) to match. In particular, any significant change to “system classes” at the base of a hierarchy, such as window, will affect application classes, such as ival_dial. Worse, the choice of a foundation library representing system resources determines the structure of the application class hierarchy and permeates the application code.

There are quite a few ways of dealing with this. For example, some languages have a solution (at its associated costs) mandated, and some implementations of C++ allow major changes to base classes without requiring recompilation of derived classes. However, here I’ll show a solution using abstract classes to make dependencies explicit. Logically, it closely parallels the way the abstract class set was used to insulate users from the details of vectors and lists.

Here, an application hierarchy

```
ival_box
  ↓ival_dial  ival_slider
```

is written independently of implementation details and then later tied into an implementation hierarchy without affecting the users of the application hierarchy:
This expresses the design in such a way that the application code becomes independent of any change in the implementation hierarchy. I have used the BB prefix for realism; suppliers of major libraries invariably prepend some identifying initials. In the future, I expect namespaces (§6.3) to be used instead.

The declaration of a class that ties an application class to the implementation hierarchy will look something like this:

```cpp
class BB_ival_slider
 : public ival_slider,
  protected BB_slider {
  public:
  // functions overriding ival_slider
  // functions as needed to implement
  // the application concepts
  protected:
  // functions overriding BB_slider
  // and BB_window functions as
  // required to conform to user
  // interface standards
  private:
  // representation and other
  // implementation details
  }
```

This structure assumes that details of what is to be displayed by a windows system is expressed by overriding virtual functions in the BB_windows hierarchy. This may not be the ideal organization of a user interface system, but it is not uncommon.

I use protected members and protected inheritance to allow classes derived from BB_ival_slider to use information about its implementation.

### 6.7 Generic Programming

A major theme in the C++ community over the last few years has been the development of techniques exploiting the template mechanism.
6.7.1 Parameterization
Independent concepts should be independently represented and should be combined only when needed.

Where this principle is violated, you either bundle unrelated concepts together or create unnecessary dependencies in the implementation of classes and functions. In particular, fitting weakly related class into a single hierarchy can be a source of unnecessary and problem-causing dependencies.

Consider the concepts of sorting, character, string, and collating sequence. A sort algorithm is independent from the concept of a character. The concept of a string is independent of any particular kind of a character. Finally, the collating sequence which you use when you sort strings of characters is independent of these other three concepts.

This independence can be expressed directly. Here is a string parameterized by the kind of characters contained so that we can make strings of both built-in and user-defined character types:

```cpp
template<class C>
class string {
    // ...
};
class Jchar {
    // Japanese characters
};
string<char> s1, s2;
string<unsigned char> us1, us2;
string<Jchar> js1, js2;
```

Independently, we can define the notion of a collating sequence and a string comparison function:

```cpp
template<class C>
class std_coll {
    public:
    bool eq(C a, C b) {
        return a==b;
    }
    bool lt(C a, C b) {
        return a<b;
    }
};
template<class C, class Coll = std_coll<C> >
int cmp(
    string<C>& s1,
    string<C>& s2
```
The `cmp` template function takes two template arguments:
- the type of characters in the strings, and
- the collator supplying the character comparison operations.

The ability to pass operations as template parameters is a very powerful expressive mechanism. It is also important for efficiency. For example, it is trivial for a compiler to inline all uses of `eq()` and `lt()`. This can be a significant advantage compared to C where operations can only be passed as pointers to functions so that function call overheads are incurred.

The second template parameter has a default so we need only specify it if we want a nonstandard collating sequence. The first template parameter can usually be deduced from the arguments to `cmp()`. For example:

```cpp
cmp(s1,s2);
cmp(js1,js2);
cmp(us1,us2)
cmp<char,no_case>(s1,s2);
```

Here, `no_case` is a collator defining `eq()` and `lt()` not to be case sensitive.

Typically, the string class, the `cmp()` function, the collator classes, and the character classes will be written by different people. Only the final user puts all of the independently developed pieces together.

This style of design relying on templates and additional template arguments (in this example, Coll) to express policies, is the basis of much of the C++ standard library. The result is exceptional flexibility and unsurpassed run-time efficiency.

### 6.7.2 Containers and Algorithms

I want to have algorithms written once and used for objects of many different types. I want these algorithms to run as fast as functions written for a single argument type. I want this to be compile-time checked so that I can be more confident of my code. I don’t want to be forced to fit my types into a hierarchy simply to be able to use them for the generic algorithms.

The containers and algorithms in the C++ standard library use a variant of the philosophy of keeping independent concepts separate. Much of the library is based on the notion of a sequence. Examples of sequences are arrays, sets, lists, maps, files.
A sequence has a beginning and an end. The end is one beyond the last element of the sequence. Positions in a sequence are represented by iterators.

```
begin          end
|                   |
v                   v
XXX -> .... -> XXX -> 0
```

Given an iterator for an element of a sequence, we can get to the next element using the `++` (increment) operator. Given an iterator for an element, we can access the element itself using the `*` (dereference) operator.

Given this simple notion, a surprising number of useful algorithms can be expressed. For example, this template function writes all elements of a container to output:

```cpp
template<class C>
void print(C& s)
{
    C::iterator p=s.begin();
    while ( p!=s.end() ) {
        cout << *p; // output
        p++; // next
    }
}
```

The C++ standard library containers and algorithms are primarily the work of Alex Stepanov. A surprising number of containers and algorithms can be expressed using just a few kinds of iterators. Importantly, the resulting generic algorithms are efficient even compared to handcrafted assembly code. For example, the C++ standard library algorithm, sort() is for many simple and realistic examples several times faster than the C standard library qsort() function. For more information about the C++ standard library and the principles underlying its design see [Koenig,1995] [Stepanov,1994].

7 Closing Remarks
Are the various facilities presented above Object-oriented or not? Which ones? Using what definition of Object-oriented? In most contexts, I think these are the wrong questions. What matters is what ideas you can express clearly, how easily you can combine software from different sources, and how efficient and maintainable the resulting programs are. In other words, how you support good programming techniques and good design techniques matters more than labels and buzz words.
The fundamental idea is simply to improve design and programming through abstraction. You want to hide details, you want to exploit any commonality in a system, and you want to make this affordable.

I would like to encourage you not to make Object-oriented a meaningless term. The notion of ‘Object-oriented’ is too frequently debased
– by equating it with good,
– by equating it with a single language, or
– by accepting everything as Object-oriented.

I have argued that there are – and must be – useful techniques beyond Object-oriented programming and design. However, to avoid being totally misunderstood, I would like to emphasize that I wouldn’t attempt a serious project using a programming language that didn’t at least support the classical notion of Object-oriented programming. In addition to facilities that support Object-oriented programming, I want – and C++ provides – features that go beyond those in their support for direct expression of concepts and relationships.

Several of the themes related to C++ programming style in this paper have been developed further in [Koenig,1995b]. The design and evolution of C++, including its most recent features, is discussed in [Stroustrup,1994].

8 Acknowledgements
Thanks to the OOPSLA’95 program committee for inviting me to give the talk upon which this paper is based, and especially to May Loomis for encouraging me to get this paper written. Carolyn Heaps transcribed the audio tape of my talk to produce the first draft of this paper. Tim Griffin and Christos Polyzois made many constructive comments.

9 References


