Overview of C++: Design Goals

Douglas C. Schmidt

d.schmidt@vanderbilt.edu

www.dre.vanderbilt.edu/~schmidt

Professor of Computer Science

Institute for Software
Integrated Systems

Vanderbilt University
Nashville, Tennessee, USA

Learning Objectives in this Part of the Lesson

- Recognize the key components of C++
- Know strategies for learning C++
- Understand C++ design goals

C++ Design Goals

C++ Design Goals

- As with C, run-time efficiency is important

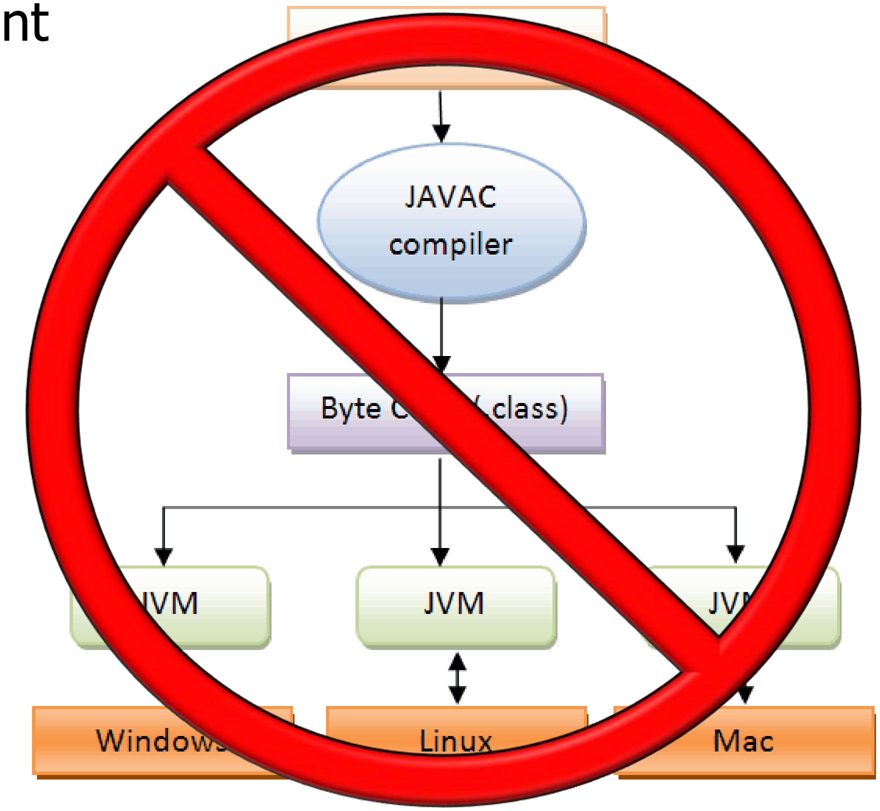
C++ Design Goals

- As with C, run-time efficiency is important
  - Zero-overhead abstraction
    - e.g., classes with constructors & destructors, inheritance, generic programming, functional programming techniques, etc.

See www.youtube.com/watch?v=G5zCGY0tkq8

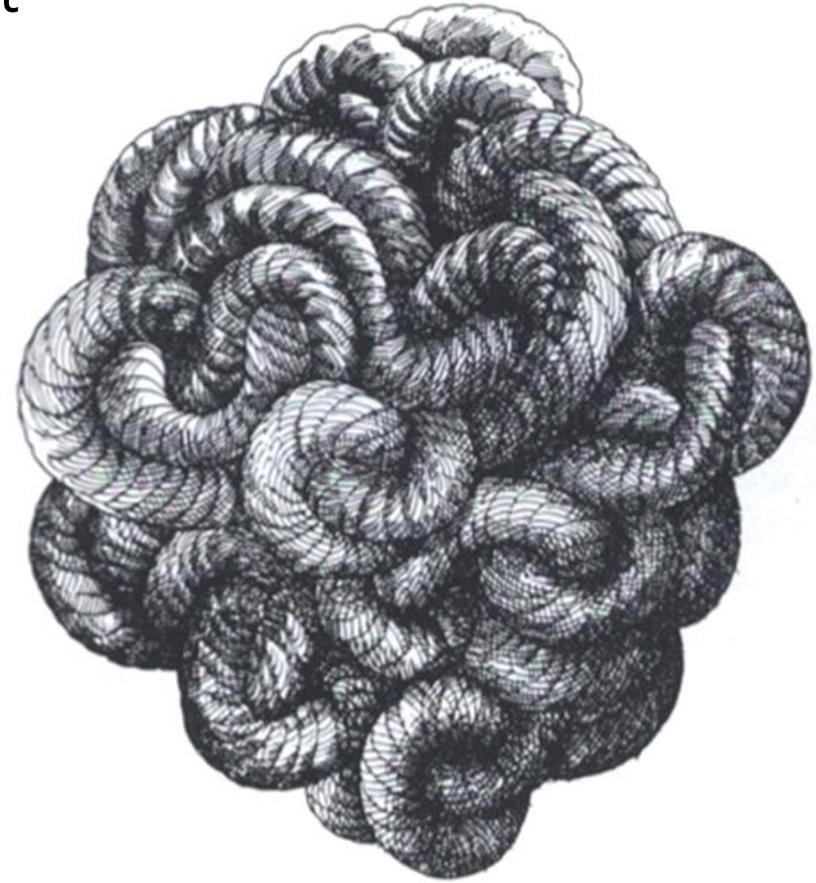
C++ Design Goals

- As with C, run-time efficiency is important
  - Zero-overhead abstraction
  - Direct mapping to hardware
    - e.g., no virtual machine overhead for instructions & native data types



C++ Design Goals

- As with C, run-time efficiency is important
  - Zero-overhead abstraction
  - Direct mapping to hardware
  - No complicated run-time libraries, managed environments, or virtual machines
    - Unlike other languages, e.g., Ada, Java, C#, etc.



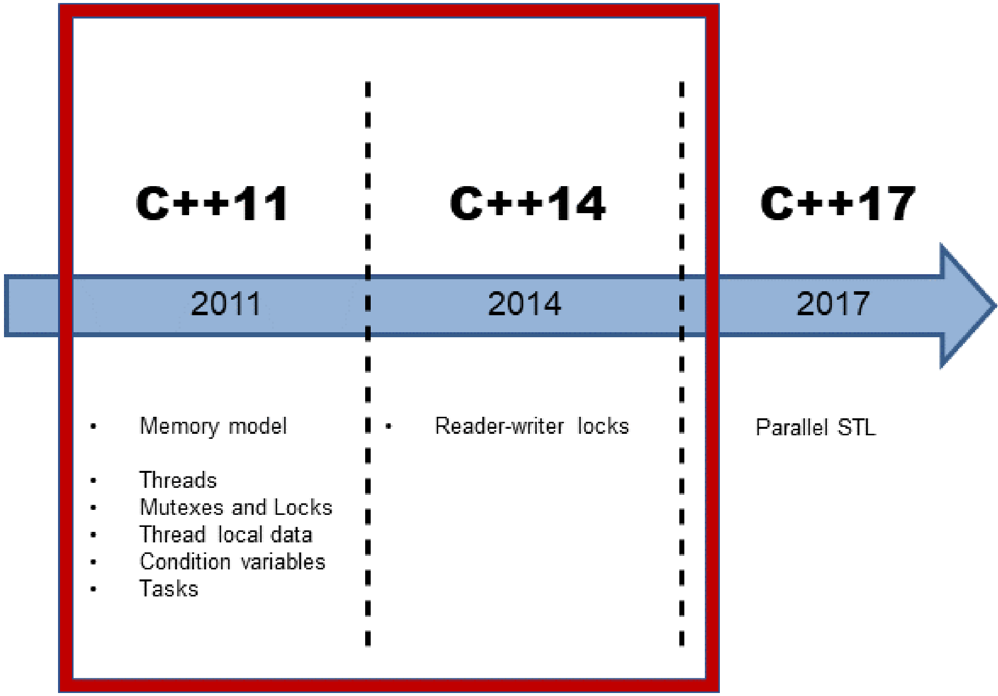
See en.wikipedia.org/wiki/Gordian_Knot

C++ Design Goals

- As with C, run-time efficiency is important
  - Zero-overhead abstraction
  - Direct mapping to hardware
  - No complicated run-time libraries, managed environments, or virtual machines
  - No language-specific support for persistence, garbage collection, or networking in C++

C++ Design Goals

- As with C, run-time efficiency is important
  - Zero-overhead abstraction
  - Direct mapping to hardware
  - No complicated run-time libraries, managed environments, or virtual machines
  - No language-specific support for persistence, garbage collection, or networking in C++
    - Additional support for threading, synchronization, & parallelism was added beginning w/C++11



See www.modernescpp.com/index.php/c-core-guidelines-rules-for-concurrency-and-parallelism

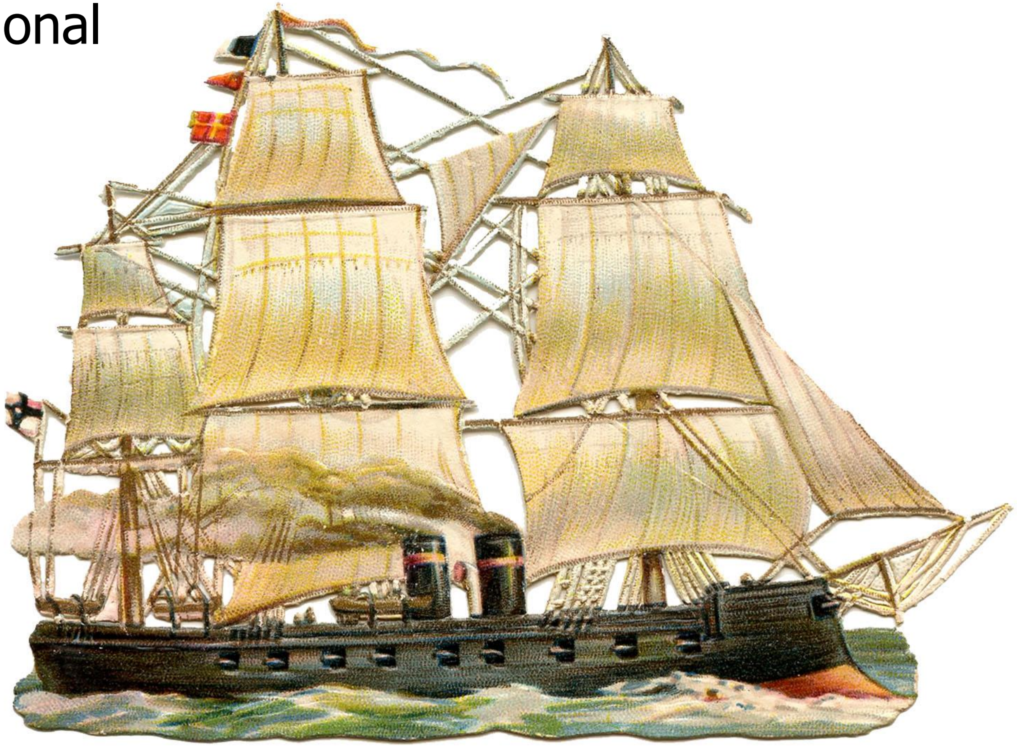
C++ Design Goals

- As with C, run-time efficiency is important
  - Zero-overhead abstraction
  - Direct mapping to hardware
  - No complicated run-time libraries, managed environments, or virtual machines
  - No language-specific support for persistence, garbage collection, or networking in C++
    - *Many* libraries exist that provide these capabilities

See www.dre.vanderbilt.edu/ACE & www.boost.org

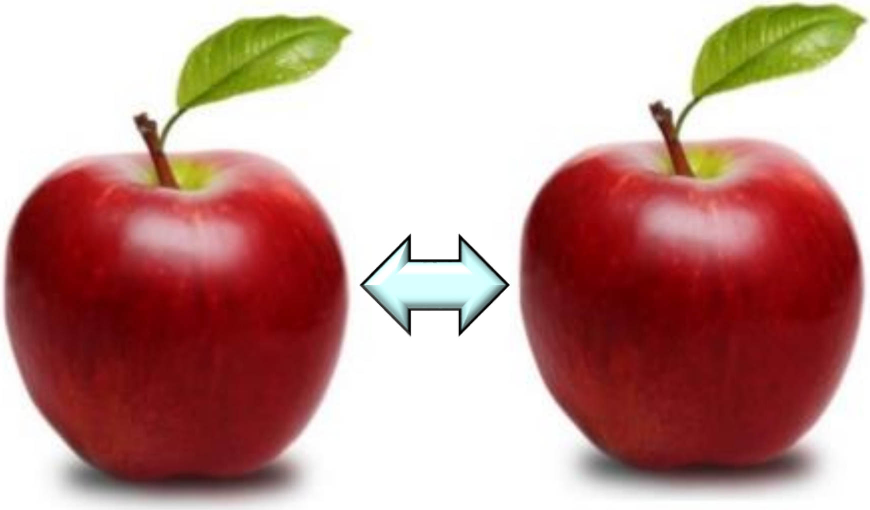
C++ Design Goals

- Compatibility w/C libraries & traditional development tools is emphasized



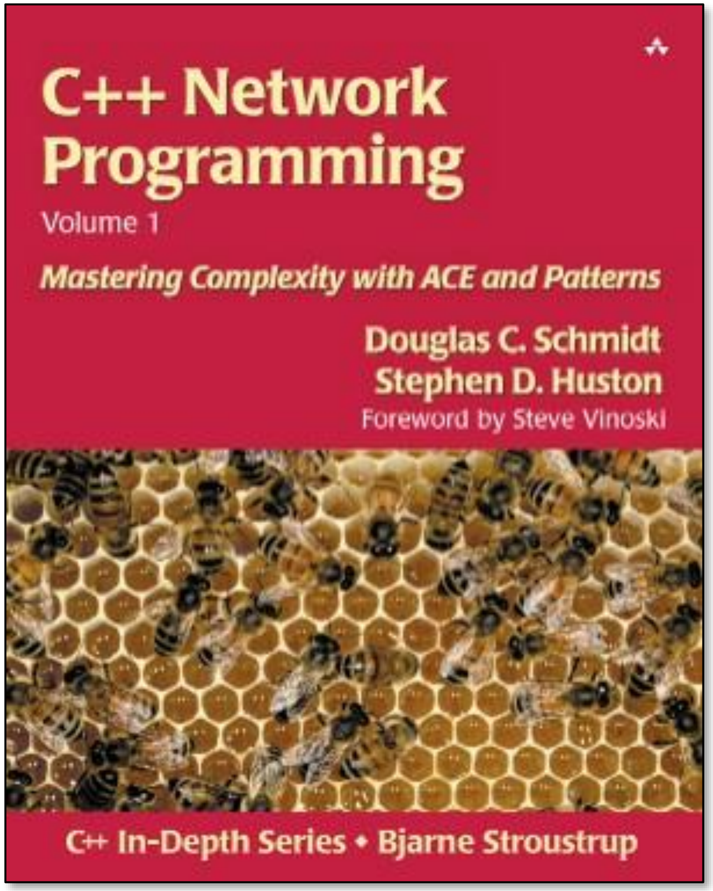
C++ Design Goals

- Compatibility w/C libraries & traditional development tools is emphasized, e.g.,
  - Object code reuse
    - e.g., the storage layout of structs is compatible with C



C++ Design Goals

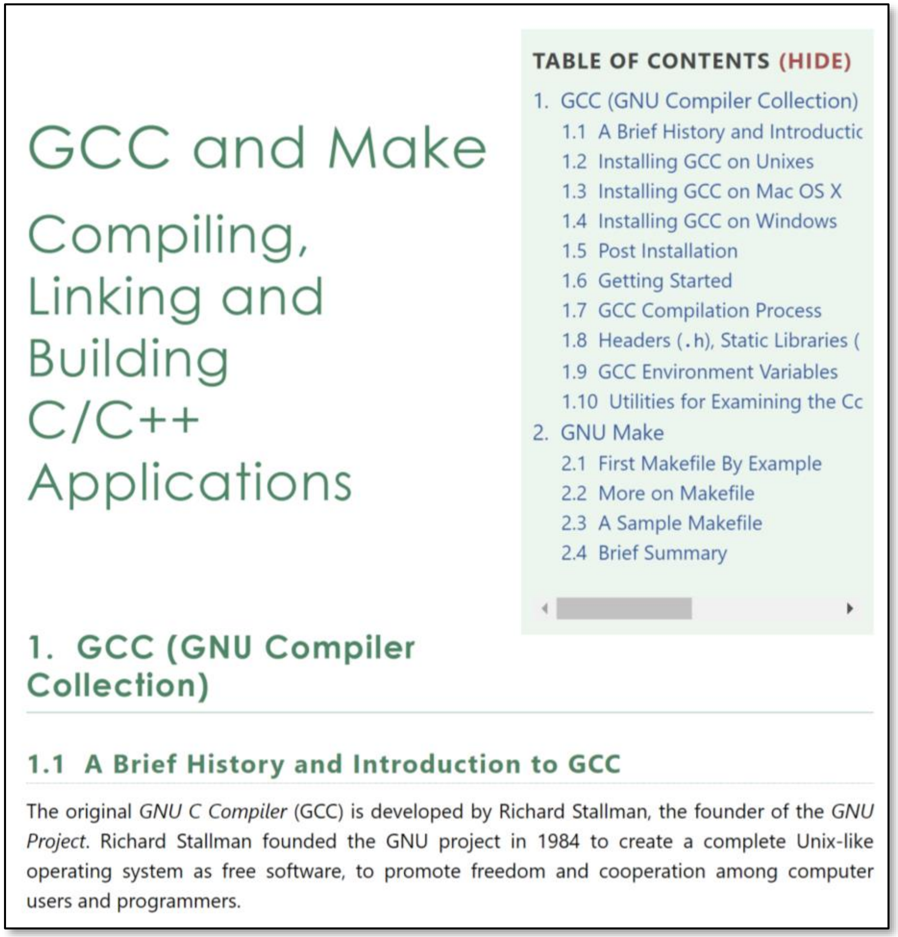
- Compatibility w/C libraries & traditional development tools is emphasized, e.g.,
  - Object code reuse
    - e.g., the storage layout of structs is compatible with C
      - Supports the standard ANSI C library, UNIX & Windows system calls via extern blocks, etc.


C++ Network Programming
Volume 1
Mastering Complexity with ACE and Patterns
Douglas C. Schmidt
Stephen D. Huston
Foreword by Steve Vinoski
C++ In-Depth Series • Bjarne Stroustrup

See www.dre.vanderbilt.edu/~schmidt/ACE/book1

C++ Design Goals

- Compatibility w/C libraries & traditional development tools is emphasized, e.g.,
  - Object code reuse
  - C++ works with the "make" family of (re)compilation build tools



GCC and Make

Compiling, Linking and Building C/C++ Applications

TABLE OF CONTENTS (HIDE)

1. GCC (GNU Compiler Collection)

1.1 A Brief History and Introduction to GCC

The original *GNU C Compiler* (GCC) is developed by Richard Stallman, the founder of the *GNU Project*. Richard Stallman founded the GNU project in 1984 to create a complete Unix-like operating system as free software, to promote freedom and cooperation among computer users and programmers.

See www3.ntu.edu.sg/home/ehchua/programming/cpp/gcc_make.html

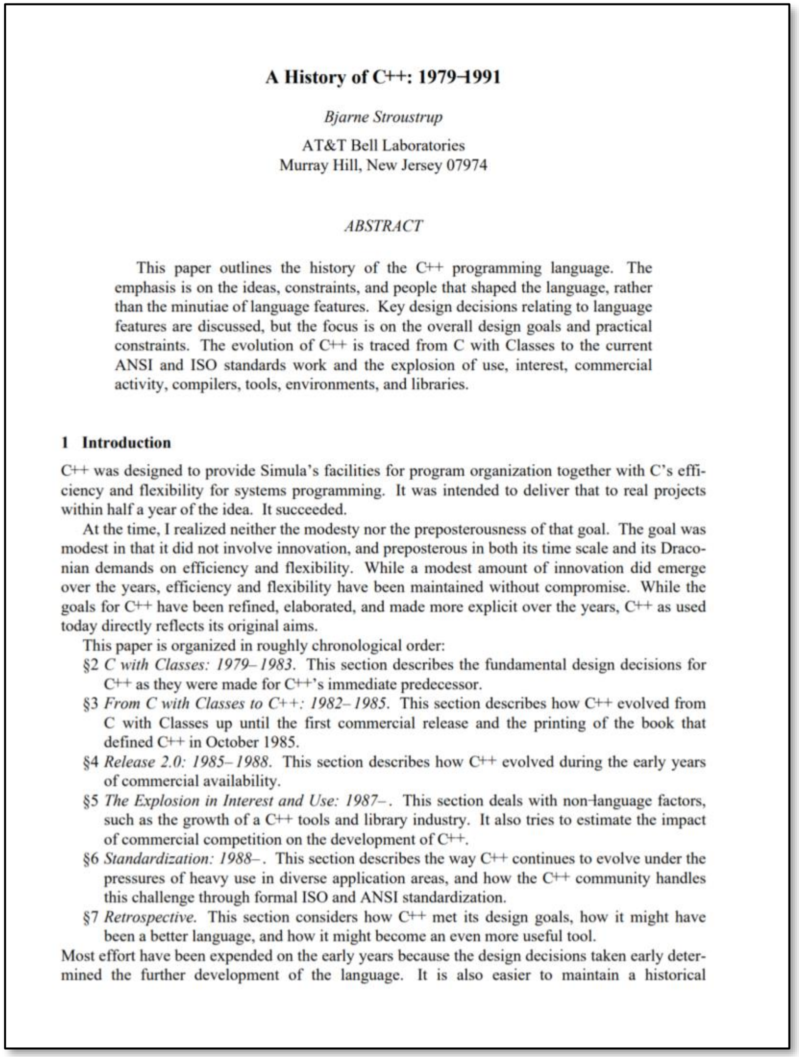
C++ Design Goals

- An initial design goal was for C++ to be “as close to C as possible, but no closer”

Don’t Stand So Close to Me
Music and Lyrics by Sting
Bright Rock
Verse
1. Young teacher, the subject of schoolgirl fantasy
2., 3. See additional lyrics
4. Instrumental
She wants him so badly. Knows what she wants to be.
Inside her there’s longing. This girl’s an open page.
Book marking, she’s so close now. This girl is half his age.
To Coda
musicnotes.com
© 1980 G.M. SUMNER
Administered by EMI MUSIC PUBLISHING LIMITED
All Rights Reserved International Copyright Secured Used by Permission

C++ Design Goals

- An initial design goal was for C++ to be “as close to C as possible, but no closer”
  - i.e., C++ is not a proper superset of C
    - Backwards compatibility with C is not entirely maintained



A History of C++: 1979−1991

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

This paper outlines the history of the C++ programming language. The emphasis is on the ideas, constraints, and people that shaped the language, rather than the minutiae of language features. Key design decisions relating to language features are discussed, but the focus is on the overall design goals and practical constraints. The evolution of C++ is traced from C with Classes to the current ANSI and ISO standards work and the explosion of use, interest, commercial activity, compilers, tools, environments, and libraries.

1 Introduction

C++ was designed to provide Simula’s facilities for program organization together with C’s efficiency and flexibility for systems programming. It was intended to deliver that to real projects within half a year of the idea. It succeeded.

At the time, I realized neither the modesty nor the preposterousness of that goal. The goal was modest in that it did not involve innovation, and preposterous in both its time scale and its Draconian demands on efficiency and flexibility. While a modest amount of innovation did emerge over the years, efficiency and flexibility have been maintained without compromise. While the goals for C++ have been refined, elaborated, and made more explicit over the years, C++ as used today directly reflects its original aims.

This paper is organized in roughly chronological order:

§2 *C with Classes: 1979–1983*. This section describes the fundamental design decisions for C++ as they were made for C++’s immediate predecessor.

§3 *From C with Classes to C++: 1982–1985*. This section describes how C++ evolved from C with Classes up until the first commercial release and the printing of the book that defined C++ in October 1985.

§4 *Release 2.0: 1985–1988*. This section describes how C++ evolved during the early years of commercial availability.

§5 *The Explosion in Interest and Use: 1987–*. This section deals with non-language factors, such as the growth of a C++ tools and library industry. It also tries to estimate the impact of commercial competition on the development of C++.

§6 *Standardization: 1988–*. This section describes the way C++ continues to evolve under the pressures of heavy use in diverse application areas, and how the C++ community handles this challenge through formal ISO and ANSI standardization.

§7 *Retrospective*. This section considers how C++ met its design goals, how it might have been a better language, and how it might become an even more useful tool.

Most effort have been expended on the early years because the design decisions taken early determined the further development of the language. It is also easier to maintain a historical

See www.stroustrup.com/hopl2.pdf

C++ Design Goals

- An initial design goal was for C++ to be "as close to C as possible, but no closer"
  - i.e., C++ is not a proper superset of C
    - Backwards compatibility with C is not entirely maintained

Valid in C, but not in C++

```
void *ptr;
/* Implicit conversion
   from void* to int* */
int *i = ptr;

/* Implicit conversion
   from void* to int* */
int *j =
  malloc(5 * sizeof *j);
```

See en.wikipedia.org/wiki/Compatibility_of_C_and_C++

C++ Design Goals

- An initial design goal was for C++ to be "as close to C as possible, but no closer"
  - i.e., C++ is not a proper superset of C
    - Backwards compatibility with C is not entirely maintained

Valid in C, but not in C++

```
void *ptr;
/* Implicit conversion
   from void* to int* */
int *i = ptr;

/* Implicit conversion
   from void* to int* */
int *j =
  malloc(5 * sizeof *j);
```

Valid in C++ & C

```
void *ptr;
int *i = (int *)ptr;
int *j = (int *)
  malloc(5 * sizeof *j);
```

See en.wikipedia.org/wiki/Compatibility_of_C_and_C++

C++ Design Goals

- An initial design goal was for C++ to be "as close to C as possible, but no closer"
  - i.e., C++ is not a proper superset of C
    - Backwards compatibility with C is not entirely maintained

Preferred in C++

```
void *ptr;
auto i =
  reinterpret_cast<int *>
    (ptr);
auto j = new int[5];
```

Valid in C, but not in C++

```
void *ptr;
/* Implicit conversion
   from void* to int* */
int *i = ptr;

/* Implicit conversion
   from void* to int* */
int *j =
  malloc(5 * sizeof *j);
```

Valid in C++ & C

```
void *ptr;
int *i = (int *)ptr;
int *j = (int *)
  malloc(5 * sizeof *j);
```

See en.wikipedia.org/wiki/Compatibility_of_C_and_C++

C++ Design Goals

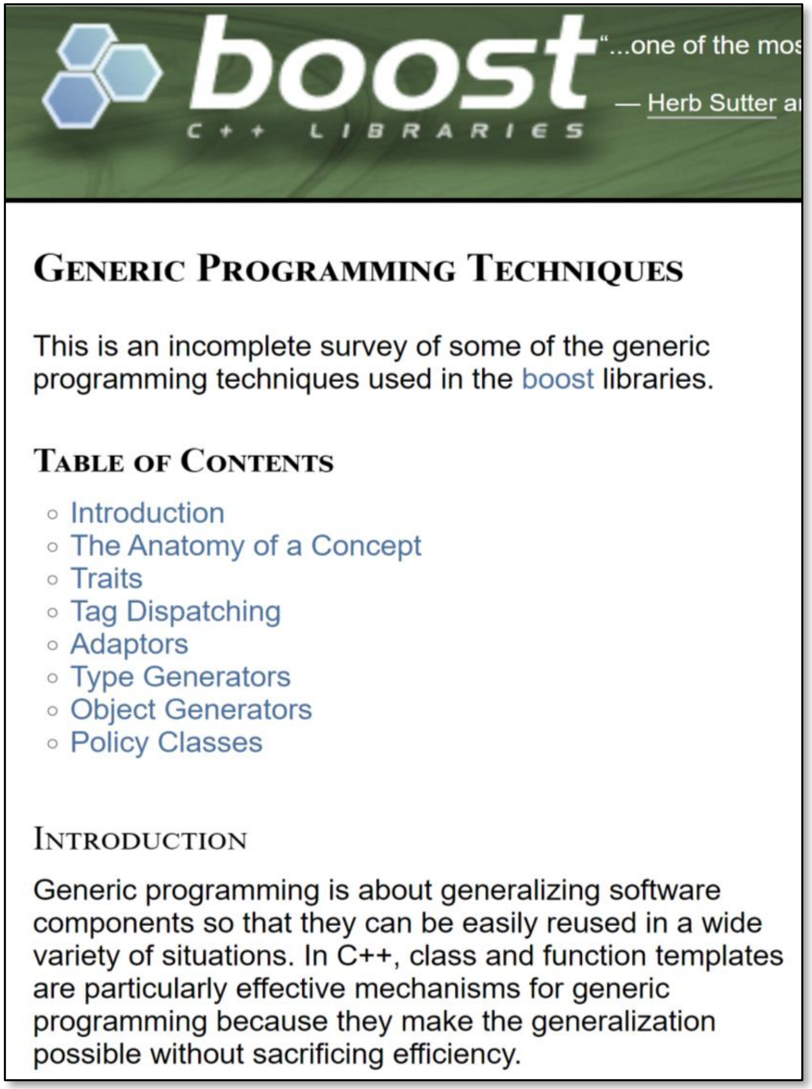
- An initial design goal was for C++ to be "as close to C as possible, but no closer"
  - i.e., C++ is not a proper superset of C
    - Backwards compatibility with C is not entirely maintained
    - Typically not a problem in practice...

C++ Design Goals

- Later C++ design goals focus on generic programming & helping developers to use modern C++ effectively

C++ Design Goals

- Later C++ design goals focus on generic programming & helping developers to use modern C++ effectively
  - Generic programming generalizes software components so that they can be easily reused in many situations



GENERIC PROGRAMMING TECHNIQUES

This is an incomplete survey of some of the generic programming techniques used in the boost libraries.

TABLE OF CONTENTS

- Introduction
- The Anatomy of a Concept
- Traits
- Tag Dispatching
- Adaptors
- Type Generators
- Object Generators
- Policy Classes

INTRODUCTION

Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations. In C++, class and function templates are particularly effective mechanisms for generic programming because they make the generalization possible without sacrificing efficiency.

See www.boost.org/community/generic_programming.html

C++ Design Goals

- Later C++ design goals focus on generic programming & helping developers to use modern C++ effectively
  - Generic programming generalizes software components so that they can be easily reused in many situations
    - C++ templates enable generic programming since they generalize without sacrificing efficiency

```
template
   <typename InputIterator,
     typename OutputIterator>
OutputIterator
copy(InputIterator first,
         InputIterator last,
         OutputIterator result) {
   while (first != last)
     *result++ = *first++;
   return result;
}
```

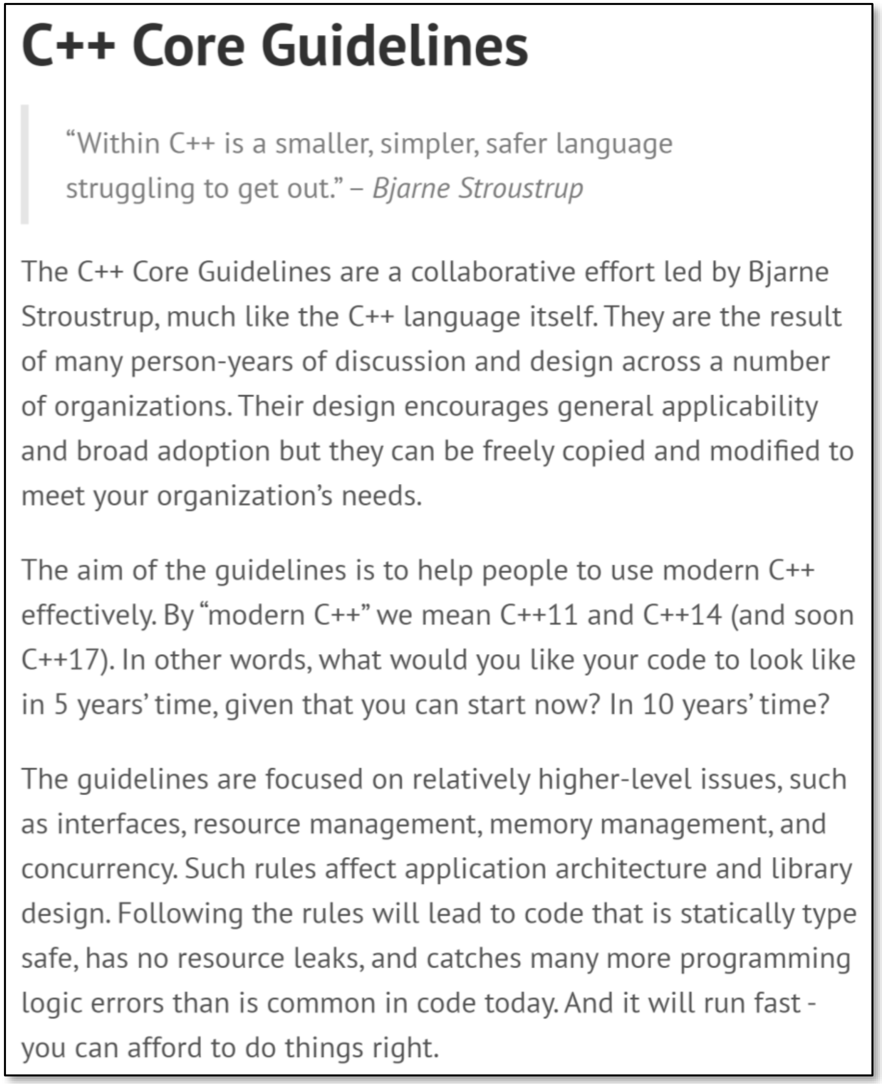
```
int a[] = {1, 2, 3, ...};
vector<int> v = {1, 2, 3, ...};

copy(a, a + sizeof(a)/sizeof(*a), ostream_iterator<int>(cout));
copy(v.begin(), v.end(), ostream_iterator<int>(cout));
```

See www.boost.org/community/generic_programming.html

C++ Design Goals

- Later C++ design goals focus on generic programming & helping developers to use modern C++ effectively
  - Generic programming generalizes software components so that they can be easily reused in many situations
  - The C++ core guidelines are a set of idioms documented to help developers efficiently and consistently write type & resource safe C++ programs



C++ Core Guidelines

> "Within C++ is a smaller, simpler, safer language struggling to get out." – *Bjarne Stroustrup*

The C++ Core Guidelines are a collaborative effort led by Bjarne Stroustrup, much like the C++ language itself. They are the result of many person-years of discussion and design across a number of organizations. Their design encourages general applicability and broad adoption but they can be freely copied and modified to meet your organization's needs.

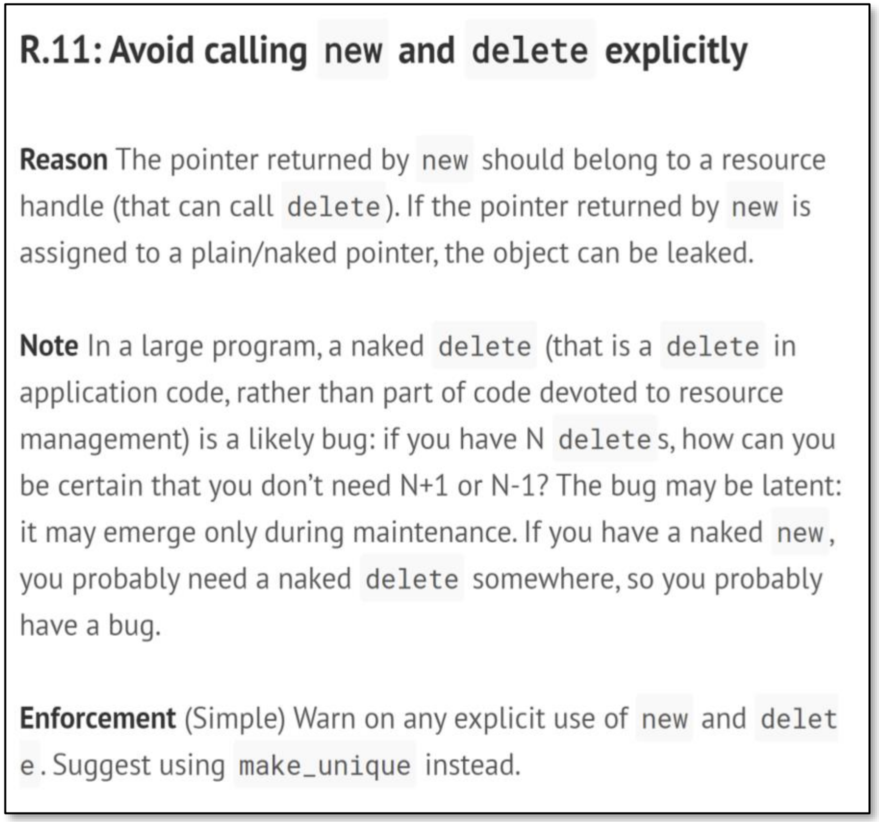
The aim of the guidelines is to help people to use modern C++ effectively. By "modern C++" we mean C++11 and C++14 (and soon C++17). In other words, what would you like your code to look like in 5 years' time, given that you can start now? In 10 years' time?

The guidelines are focused on relatively higher-level issues, such as interfaces, resource management, memory management, and concurrency. Such rules affect application architecture and library design. Following the rules will lead to code that is statically type safe, has no resource leaks, and catches many more programming logic errors than is common in code today. And it will run fast - you can afford to do things right.

See isocpp.github.io/CppCoreGuidelines

C++ Design Goals

- Later C++ design goals focus on generic programming & helping developers to use modern C++ effectively
  - Generic programming generalizes software components so that they can be easily reused in many situations
  - The C++ core guidelines are a set of idioms documented to help developers efficiently and consistently write type & resource safe C++ programs



R.11: Avoid calling `new` and `delete` explicitly

Reason The pointer returned by `new` should belong to a resource handle (that can call `delete`). If the pointer returned by `new` is assigned to a plain/naked pointer, the object can be leaked.

Note In a large program, a naked `delete` (that is a `delete` in application code, rather than part of code devoted to resource management) is a likely bug: if you have N `delete`s, how can you be certain that you don't need N+1 or N-1? The bug may be latent: it may emerge only during maintenance. If you have a naked `new`, you probably need a naked `delete` somewhere, so you probably have a bug.

Enforcement (Simple) Warn on any explicit use of `new` and `delete`. Suggest using `make_unique` instead.

See isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.html#Rr-newdelete

End of C++ Design Goals