

CPP-Summit

# 全球 C++ 及系统软件技术大会

C++ and System Software Summit

## 嘉宾演讲稿合集

主办方

Boolan  
高端IT咨询与教育平台

8大  
技术咨询领域

Boolan依托雄厚的专家团队和丰富的行业经验，通过八大技术咨询板块，提供涵盖产品管理、架构设计、人工智能等全栈技术咨询解决方案，帮助企业  
在数字化、智能化、万物互联的变革中打造关键创新能力，加速技术和产品迭代升级，助力企业快速成长。

4大  
高端技术会议

Boolan高端技术会议汇聚国内外技术与产品大咖，分享一线实战经验，解析前沿技术趋势，探讨行业创新机遇！Boolan 四大技术会议品牌涵盖：全球产品  
经理大会 (PM-Summit) 、全球C++及系统软件技术大会 (CPP-Summit) ,全球机器学习技术大会 (ML-Summit) ,全球软件研发技术峰会 (SDCon) 。

300+  
培训课程主题

Boolan针对企业特定培训需求，通过多年的技术课程体系建设，涵盖机器学习、软件架构、产品经理、C/ C++、Java平台等14个领域，300+课程主  
题；满足不同领域，不同阶段公司差异化的人才培养需求，实现企业团队人力资本的提升。

600+  
位国内外专家

Boolan拥有遍布全球的600多位国内外专家资源。既有C++之父Bjarne Stroustrup，硅谷产品大师Marty Cagan，机器学习之父Michael.I.Jordan，全  
球软件架构大师Martin Abbott等在IT互联网领域有极强号召力的国际大师；也有来自如Goolge，Microsoft，Amazon，阿里巴巴等国内外著名互联网  
科技企业的技术骨干和行业精英。

2500+  
高端企业客户

Boolan助力企业全方位成长，迄今为止，我们已为华为、腾讯、百度、eBay、微软、SAP、HP、思科等国内外2500+企业提供高端IT技术咨询与培  
训服务，客户遍及互联网、通信、金融等各个行业领域，并且持续扩展中。



# 顾问团队



**吴咏炜** 首席软件咨询师

近25年C/C++系统级软件开发和架构经验。专注于C/C++语言、软件架构、设计模式和性能优化等。前Intel亚太研发中心资深系统架构师。



**温 昱** 首席软件咨询师

擅长代码守护、架构守护咨询等实战技能咨询。在代码质量和架构设计领域有多项原创方法论建设。多年嵌入式软件研发咨询经验著作《软件架构设计》等印刷40余次。



**赵永刚** 资深软件咨询师

16年C/C++系统级软件开发经验，系统软件架构、代码安全、性能优化专家，涉及通信、嵌入式Linux开发、自动驾驶系统软件等。前诺基亚贝尔资深架构师。



**车明君** 资深软件咨询师

前阿里高级技术专家，微店技术总监，15年以上大型互联网企业工作经验，对大规模电商系统架构、组件化体系、前端体系等有长期、有效率、有产出的技术及产品实践。



**侯 捷** 首席软件咨询师

著名C++技术专家，计算机图书作者、译者、书评人。著有《STL源码分析》等畅销书，译有众多脍炙人口的高阶技术书籍，其著作与讲座影响一代程序员。



**张银奎** 首席软件咨询师

知名系统内核专家，资深软硬件架构师，拥有13年Intel工作经历，10年GPU深度优化技术积累。著有《软件调试》《格蠹汇编》。



**冉 昕** 资深软件咨询师

长期从事嵌入式、通信行业，精于STL, Boost and C++ 11/14/17/20 llvm/clang等，在高性能、低延迟、大规模系统级软件领域，拥有丰富技术开发和架构经验。曾在朗讯，摩根斯坦利任职。



**李 锰** 资深软件咨询师

20年以上软件架构和技术开发管理、咨询经验。拥有丰富的C/C++、Java、Python开发经验。曾在阿里等公司担任技术专家，译著：《领域驱动设计精简版》等书。



**方 林** 首席AI咨询师

研究领域是图形图像处理和自然语言处理。独立开发人工智能博弈软件框架、问题求解框架。前深兰人工智能首席科学家，南京大学计算机软件博士。



**王福强** 首席软件咨询师

在软件架构、分布式系统设计与开发、并发与并行编程等领域积累了丰富的经验。曾在阿里担任高级技术专家，著有《Spring揭秘：快速构建微服务体系》等书。



**王晓涛** 资深软件咨询师

精于架构设计、系统调优、算法优化、编译构建等。曾任七牛云高级研究员，百度大搜索事业群高级系统架构师，涉及大型服务端网络、存储、搜索等多个领域的开发和架构设计工作。



**杨 武** 资深AI咨询师

人工智能专家，对人工智能与产业结合践深有研究。曾创办人工智能企业脸云FacialCloud、前房金所联合创始人兼CTO，前Cisco资深技术主管与架构师。

# 2021技术会议 预告



自1936年阿兰·图灵提出「图灵机」以及机器具备「思维」的可能性以来，以机器学习为代表的人工智能经过飞速发展，正深刻地改变着我们的世界。Boolan 特邀近40位机器学习领域的技术领袖和行业应用专家，共同探讨人工智能领域的前沿发展和行业最佳实践。官网：[www.ml-summit.org](http://www.ml-summit.org)

大会时间：1月13-14日

大会地点：金茂北京威斯汀大酒店



随着信息革命的飞速发展，软件已深入人类生活的方方面面，深刻而彻底地改造了人类世界。Boolan 特邀近40位全球软件领域的技术领袖以及一线实战专家，融合主题演讲、互动研讨、案例分享、高端培训等多种形式，共同探讨软件领域的前沿发展、最佳实践和创新应用，共同打造大师智慧+实践干货的技术盛宴。

官网：[www.sdcon.com.cn](http://www.sdcon.com.cn)

大会时间：5月      大会地点：上海



全球产品经理大会（PM-Summit）是汇聚全球互联网与科技产品领域专业人士的高端交流和分享平台。Boolan 特邀来自全球互联网与科技产品领域的专家、领袖与一线专家，与众多行业精英参会者共同开展一场面向全球产品人的高端盛宴！

官网：[www.pm-summit.org](http://www.pm-summit.org)

大会时间：8月      大会地点：北京



系统级软件是数字世界的基础设施，C++自1985年由Bjarne Stroustrup博士在贝尔实验室发明以来，一直被誉为系统级编程“皇冠上的明珠”。我们特邀全球C++和系统级软件技术领域的专家、学者，汇聚一堂，致力于将大会打造为系统软件领域规模最大、阵容最强、干货最多的高规格技术盛会！

官网：[www.cpp-summit.org](http://www.cpp-summit.org)

大会时间：12月      大会地点：北京

# 目 录

李建忠 Boolan创始人兼CEO

Bjarne Stroustrup C++之父，美国三院院士

Titus Winters Google C++代码库总负责人

Titus Winters Google C++代码库总负责人

吴咏炜 Boolan 首席咨询师，知名C++专家

赵永刚 Boolan 资深咨询师

冉 昕 Boolan 资深咨询师

Michael Wong C++嵌入式开发委员会SG14主席

Mike Spertus Amazon首席工程师

Arno Schödl Think-Cell创始人兼CTO

Charley Bay 系统架构专家

迎接软件变革的大航海时代	8
C++20与C++的持续演化	22
Google的软件工程实践与原则	80
软件工程“左移（Shifting left）”的奥秘	115
C++性能调优纵横谈	149
使用代码检查提升软件质量低延迟	227
低延迟场景下的性能优化实践	308
SYCL与oneAPI支持的C++20异构计算	358
使用现代C++来实现非凡的“面向对象”	451
C++错误处理最佳实践	505
系统架构师的三个视角：开发、架构与设计	570

# 目 录

Dori Exterman Incredibuild CTO  
IvanČukić 《C++函数式编程》作者  
Phil Nash 开发者测试专家，Catch创始人  
陈 峰 腾讯广告工程效能负责人  
冯富秋 阿里云智能系统技术负责人  
胡俊锋 阿里云智能IoT操作系统总负责人  
黄 贵 阿里云数据库技术架构总负责人  
连少华 资深架构师  
刘光聪 资深技术咨询师  
刘新铭 鉴释科技首席架构师，编译器专家  
马 骏 阿里云智能 C/C++ 编译器技术主管  
潘爱民 指令集公司CEO，著名操作系统专家

2021新趋势：如何加速大规模软件开发的迭代周期	652
如何设计优雅的API同时不牺牲性能	693
开发者测试最佳实践	789
腾讯广告系统大规模 C++ 工程实践	835
Linux内核的语言虚拟机王者-EBPF	870
AliOS Things物联网操作系统架构设计	907
数据库在云原生时代的架构演进与技术特点	932
Modern C++ 整洁代码最佳实践	966
面向领域模型的Modern C++实现模式	999
如何检测业务逻辑导致的安全漏洞	1038
C++ 协程在阿里的推广和大规模应用	1087
物联网操作系统的架构设计	1116

# 目 录

蒲俊峰 腾讯广告推荐系统负责人	实时推荐系统设计与优化	1151
茹炳晟 腾讯TEG基础架构部资深技术专家	软件研发效能提升的行业实践与探索	1192
宋铜铃 华为某软件首席架构师	基于C语言的微组件架构设计与实践	1223
谈静国 华为ICT基础软件渗透测试技术专家	渗透视角下的C_C++安全编码实践	1255
王 博 资深技术咨询	多范式融合的Modern C++ 软件设计	1282
尉刚强 软件技术咨询师	C/C++之构建设计与重构	1315
吴 锐 腾讯TEG云架构平台部高级工程师	C++ 高性能大规模服务器开发实战	1346
张 超 软件技术咨询师	C++ Modules与大规模物理设计	1377
张汉东 资深Rust专家	Rust系统级开发的优势与劣势	1413
张晓龙 中兴通讯资深架构师	中兴契约测试规模化落地实践	1441
张银奎 知名系统内核专家	从纳秒级优化谈CPU眼里的好代码	1492

# 李建忠

Boolan创始人兼首席执行官



Boolan创始人兼首席执行官，全球C++及系统软件技术大会创办人。国内知名技术专家，技术未来主义者。拥有近20年技术、产品和战略管理经验。通过汇聚全球技术专家，为包括多家世界500强公司在内的知名品牌提供高端IT技术咨询和服务。

主办方：

**Boolan**  
高端IT咨询与教育平台

C++ Summit 2020

Boolan 创始人兼CEO

李建忠

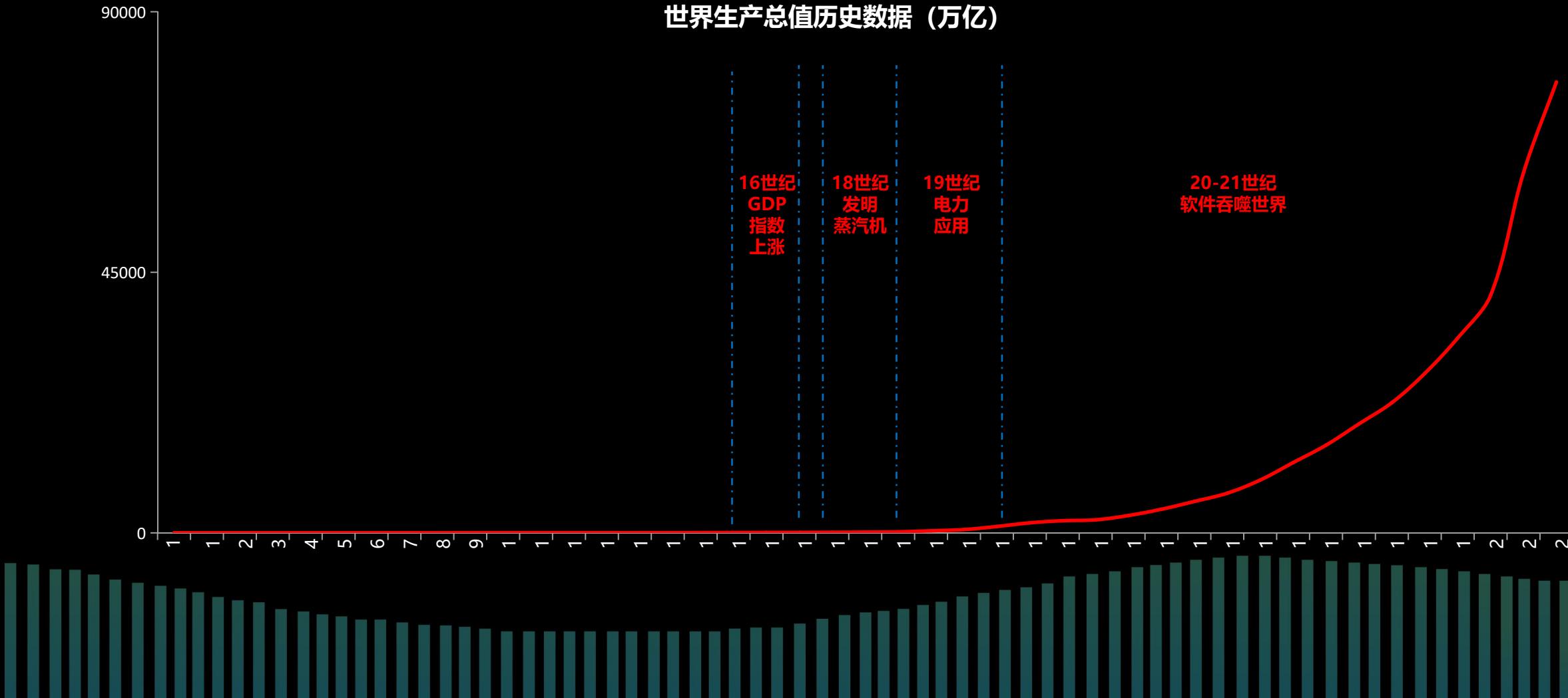
# 迎接软件变革的 大航海时代

# 软件正在吞噬整个世界

## Software Is Eating the World

Marc Andreessen

# 站到历史长河：软件是第一生产力



# 技术变革背后不变的智慧

# 时间几乎是所有学科的最大变量

物理学引入时间变量，才有了相对论的大变革...

生物学引入时间变量，才有了进化论的大变革...

.....

软件引入时间变量，才有敏捷、CI/CD、演进架构

在人类创造的所有事物中，软件越来越具有生物属性

好的软件架构必须是不断生长的

Growing

康威定律：系统的设计结构取决于组织的沟通结构

架构设计，必须关注团队之间的沟通、分工、协调

除了垂直功能团队外，横向的、跨功能的、融合团队很重要

好的软件组织是紧密协作的  
**Collaborative**

机器学习在算法 + 数据的计算架构之上开创了全新的天地

数据重新定义了算法，赋予软件以高度的智能

未来所有的软件团队都要配备大数据和机器学习专家

好的软件算法是数据驱动智能的  
**Intelligent**

# Boolan 「软件变革咨询框架GICS」

来自全球近80位顶尖软件技术专家，经过近20年软件前沿的研究和数亿行代码的积累成果。

不断生长 Growing  
数据智能 Intelligent  
紧密协作 Collaborative  
下代软件 Software

# Boolan, 高端IT咨询与教育平台



技术咨询

Tech Consulting



技术会议

Tech Conference



企业培训

Enterprise Training

## 感谢客户的长期支持



| 感谢此次大会合作伙伴



2005年，第一届C++大会，李建忠与C++之父





# Bjarne Stroustrup

C++之父，美国国家工程院、ACM、IEEE 院士

C++之父，美国国家工程院、ACM、IEEE 院士，被誉为“近 20 年来计算机工业最具影响力的 20 人”。Bjarne Stroustrup 于 1979 年获得剑桥大学计算机博士学位，毕业后加入贝尔实验室，在那里发明了 C++。他的研究领域包括：分布式系统、编程语言、软件开发工具。曾于2018年荣获美国国家工程院颁发 查尔斯·斯塔克·德拉普尔奖，被誉为工程学界的诺贝尔奖。现任摩根士丹利技术部董事总经理，哥伦比亚大学客座教授。代表作品：《C++ 程序设计语言》《C++ 语言的设计和演化》。

主办方：

**Boolan**  
高端 IT 咨询与教育平台

# The Continuing Evolution of 1979–2020 C++

Bjarne Stroustrup

Morgan Stanley, Columbia University

[www.stroustrup.com](http://www.stroustrup.com)



# Abstract

## **A short introduction to the aims and status of modern C++**

C++20 is here. How does it reflect the aims and ideals of C++? What are the major features of C++20? What is in the works for future standards? I will - necessarily briefly - touch upon type-safety, resource-safety, type deduction, modularity, and concurrency. After this introduction, the second half of the presentation will be a question and answer session.

2\*40 minutes + Q&A

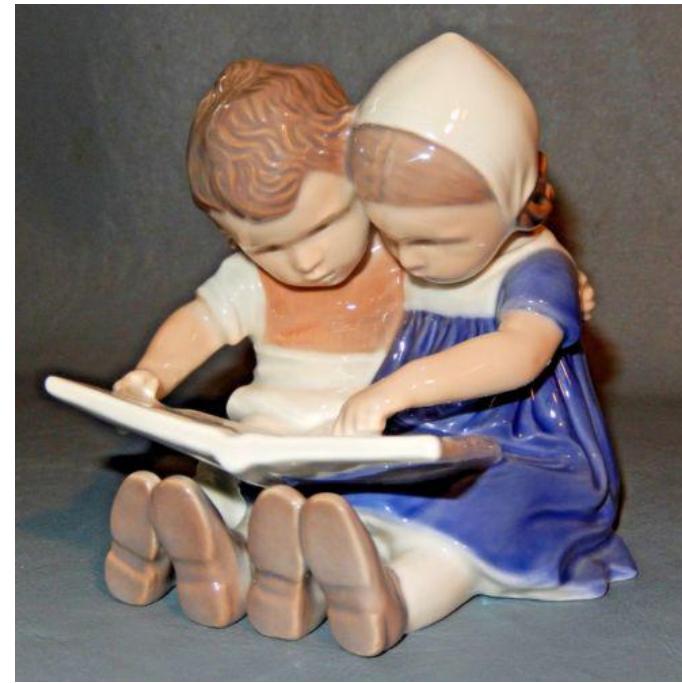
# Overview

- C++ aims and means
- C++20
  - Time to celebrate
- C++20 features
  - Making the preprocessor redundant
  - Modules
  - Compile-time programming
  - Generic Programming and Concepts
  - Concurrency and parallelism
- The future
  - C++23

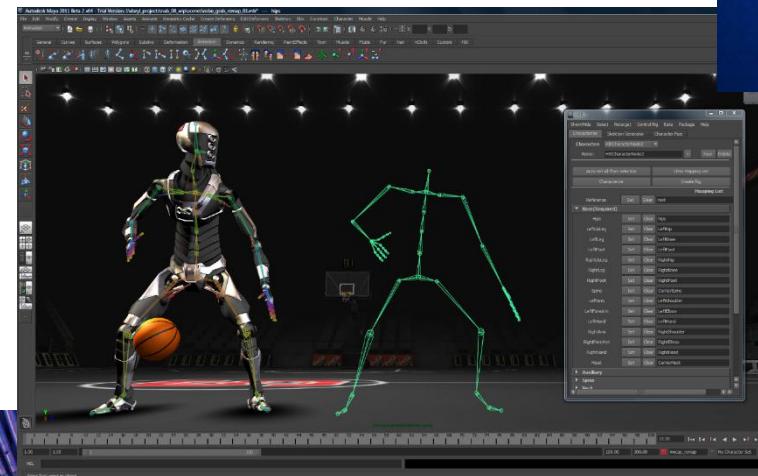
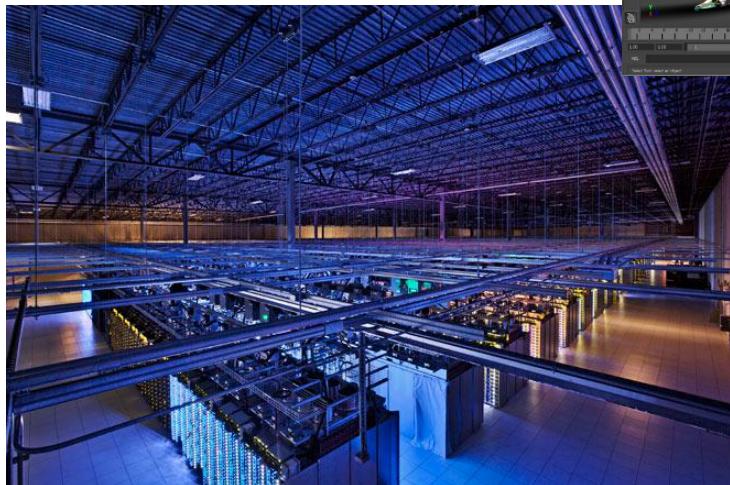


# What matters?

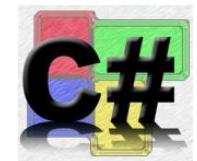
- People
- A programming language is a tool, not an end goal
  - “Ordinary people” want great systems, not programming languages
- Software developers want great tools
  - not just programming language features
- A programming language is a key part of a developer’s toolbox
  - But only one part



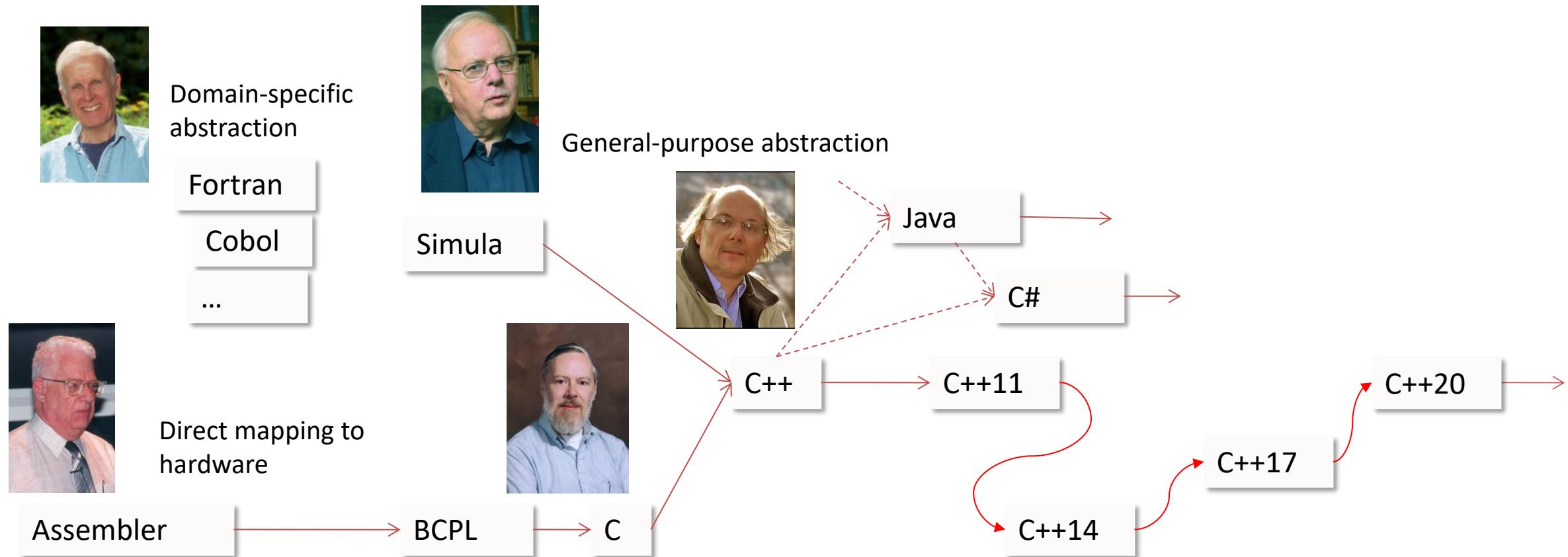
The value of a  
programming language  
is in the quality of its  
applications



amADEUS



# Programming Languages



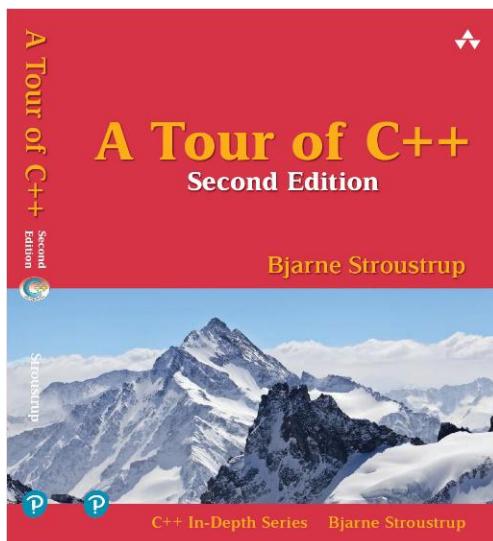
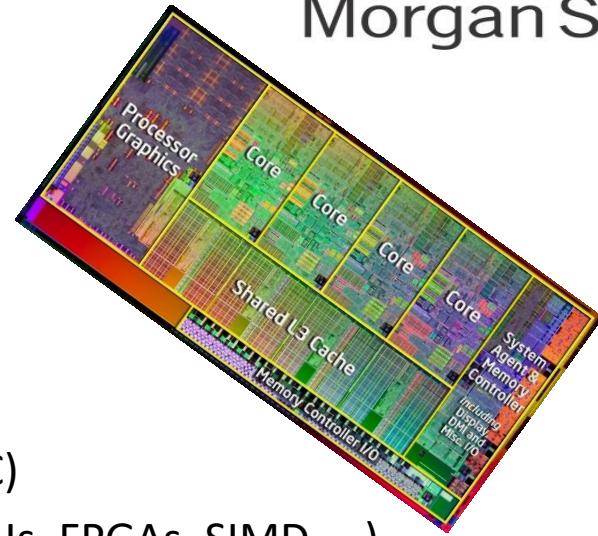
# C++ community

- About 4.5 million developers (surveys)
  - Seems to be growing by 100,000++ developers per year
  - Worldwide
    - North America, South America, Western Europe, Eastern Europe, Russia, China, India, Australia, ...
  - Most industries
    - Finance, games, Web applications, Web infrastructure, data bases, telecommunications, aerospace, automotive, microelectronics, medical, movies, graphics, imaging, scientific, embedded systems, ...



# C++ in two lines

- Direct map to hardware
  - of instructions and fundamental data types (initially from C)
  - Future: use novel hardware better (caches, multicores, GPUs, FPGAs, SIMD, ...)
- Zero-overhead abstraction
  - Classes, inheritance, templates, concepts, aliases, ... (initially inspired by Simula)
  - Future: Complete type- and resource-safety, concepts, modules, concurrency, ...



# The C++ model

- Static type system with equal support for built-in types and user-defined types
- Value and reference semantics
- Systematic and general resource management (RAII)
- Support for efficient object-oriented programming
- Support for flexible and efficient generic programming
- Support for compile-time programming
- Direct use of machine and operating system resources
- Concurrency support through libraries (often implemented using intrinsics)

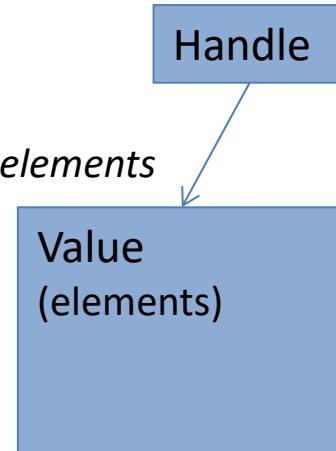
– ***Thriving in a Crowded and Changing World: C++ 2006–2020***

» <https://dl.acm.org/doi/pdf/10.1145/3386320>

# Resource management: Constructors and destructors

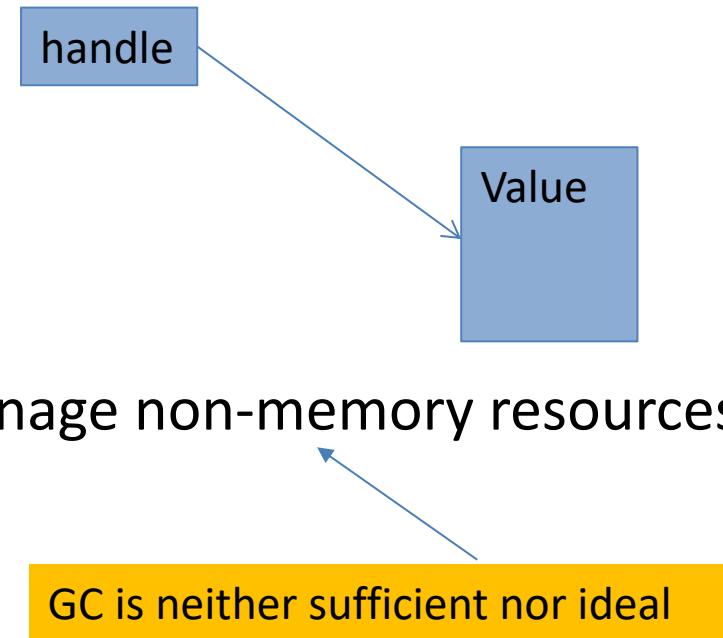
```
template<Element T>
class Vector {      // vector of Elements of type T
public:
    Vector(initializer_list<T>);    // acquire memory for list elements and initialize
    ~Vector();                      // destroy elements; release memory
    // ...
private:
    T* elem;                      // representation, e.g. pointer to elements plus #elements
    int sz;                        // #elements
};

void fct()
{
    Vector <double> v {1, 1.618, 3.14, 2.99e8};           // vector of 4 doubles
    Vector<string> vs {"Strachey", "Richards", "Ritchie"}; // vector of strings
    Vector<pair<string,jthread>> vp { {"producer",prod}, {"consumer",cons} };
    // ...
} // memory, strings, and threads released here
```



# Resources

- Make resource release implicit and guaranteed (RAII)
- All C++ standard-library containers manage their elements
  - **vector**
  - **list, forward\_list** (singly-linked list), ...
  - **map, unordered\_map** (hash table),...
  - **set, multi\_set**, ...
  - **string**
- Some C++ standard-library classes manage non-memory resources
  - **thread, shared\_lock**, ...
  - **istream, fstream**, ...
  - **unique\_ptr, shared\_ptr**, ...
- A container can hold a non-memory resource
  - This all works recursively



# Resources and Pointers

- Many uses of pointers in local scope are not exception safe

```
void f(int n, int x)
{
    Gadget* p = new Gadget{n}; // look I'm a java programmer! 😊
    // ...
    if (x<100) throw std::runtime_error("Weird!"); // leak
    if (x<200) return; // leak
    // ...
    delete p; // and I want my garbage collector! 😥
}
```

- It leaks

- Rule: No “Naked New”; no “naked pointers”!

# Resources and Pointers

- A `std::shared_ptr` implicitly releases its object at when the last `shared_ptr` to it is destroyed

```
void f(int n, int x)
{
    auto p = make_shared<Gadget>(n); // manage that pointer!
                                         // return a shared_ptr<Gadget>
    // ...
    if (x<100) throw std::runtime_error{"Weird!"}; // no leak
    if (x<200) return;                           // no leak
    // ...
}
```

- `shared_ptr` handles general resources correctly
- **But:** I don't want to create any garbage!

# Resources and Pointers

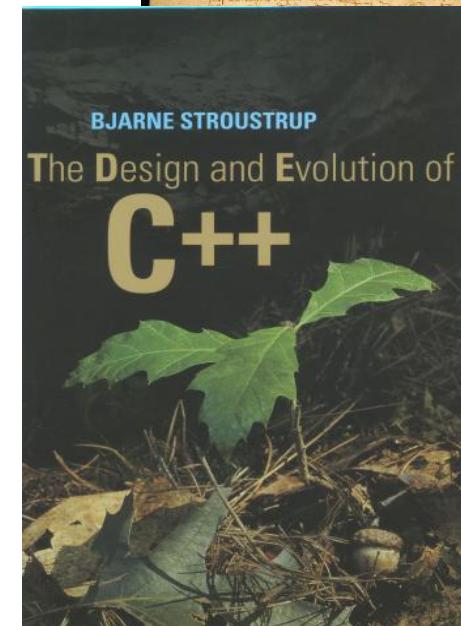
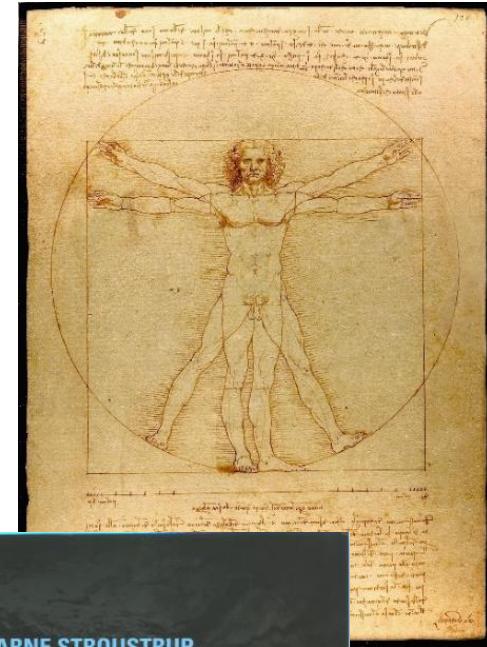
- But why use a pointer at all?
- If you can, just use a scoped variable
  - Often a handle

Keep simple things simple!

```
void f(int n, int x)
{
    Gadget g {n};
    // ...
    if (x<100) throw std::runtime_error{"Weird!"};      // no leak
    if (x<200) return;                                // no leak
    // ...
}
```

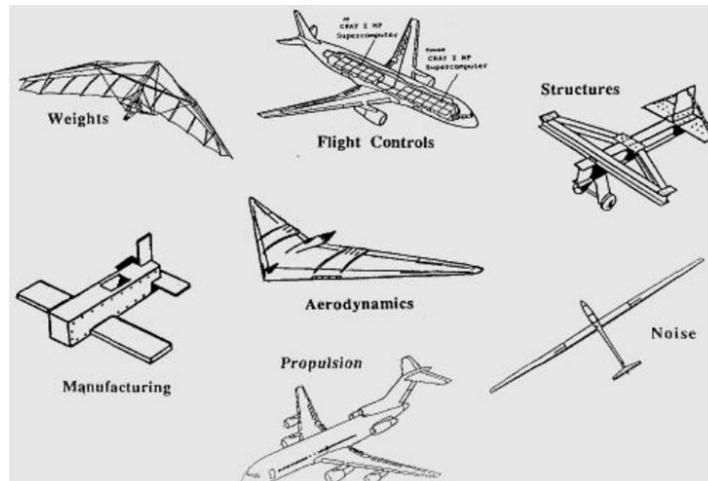
# A design philosophy is essential

- Co-evolution
  - Language features
  - Standard-library components
  - Design and programming techniques
- Fundamental aims
  - Better, faster, safer, easier-to-write code
  - Stability over decades
- Support
  - Tools (e.g., static analyzers)
  - Education (always a weak point for C++)
- Concerns must be balanced
  - Being best in one or two areas is not enough



# An engineering approach

- Principled and pragmatic design
- Progress gradually guided by feedback
- There are always many tradeoffs
  - Choosing is hard
- Design decisions have consequences
  - Determine what kind of applications can be done well



# Stability *and* evolution

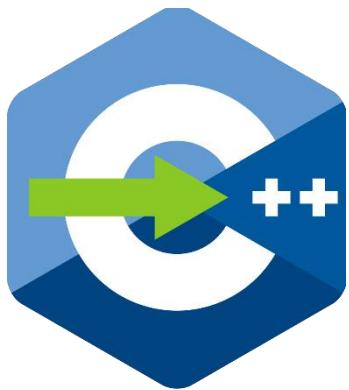
- Common attitude:
  - C++ is too large and complicated: simplify it!
  - And add these two essential features ASAP!!
  - Whatever you do, don't break my code!!!
- I agree, but how?
  - Stability/compatibility is a feature
  
- Design and Coding guidelines
  - Give people better ways to do things
    - Improve the language and standard library
  - Guide people away from unsafe and complicated habits, towards
    - Safer code
    - Easier to write
    - Faster to run



# The onion principle

- Management of complexity
  - Make simple things simple!
- Layers of abstraction
  - The more layers you peel off, the more you cry





# C++ Core Guidelines

- You can write type- and resource-safe C++
  - No leaks
  - No memory corruption
  - No garbage collector
  - No limitation of expressibility
  - No performance degradation
  - ISO C++
  - Tool enforced **(eventually)**
- Work in progress
  - C++ Core Guidelines: <https://github.com/isocpp/CppCoreGuidelines>
  - GSL: Guidelines Support Library: <https://github.com/microsoft/gsl>
  - Static analysis support tools **(work in progress: Visual Studio, Clang Tidy)**



# Standardization

- Since 1989
- A necessary evil?
  - “You can’t have a major programming language controlled by a single company”
    - Actually, you can: Java, C#, ...
- There are many kinds of standardization
  - ISO, ECMA, IEEE, W3C, ...
- Long-term stability is a feature
  - You need a standards committee
- Vendor neutral
  - Important for some major users
  - Deprives C++ or development funds
- Dangers
  - Design by committee
  - Stagnation
  - Divergent directions of design
  - Overelaboration





1990

# “the committee”



2011

2014



2017

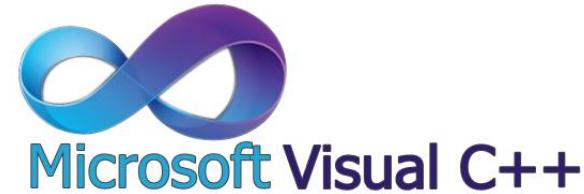


# C++20 is here



# Evolution: C++98 ... C++20

- C++98 – a solid workhorse
  - Exceptions, templates, ...
- C++11 – a major improvement
  - Libraries and language features
  - Concurrency, random numbers, regular expressions, ...
  - Lambdas, generalized constant expressions, ...
- C++14 – completes C++11
  - Technical specifications: Concepts, modules, networking, ...
- C++17 – adds many minor improvements
  - Currently shipping ☺
- C++20 – is great
  - Concepts, modules, coroutines, ...
  - Mostly shipping!



# C++20

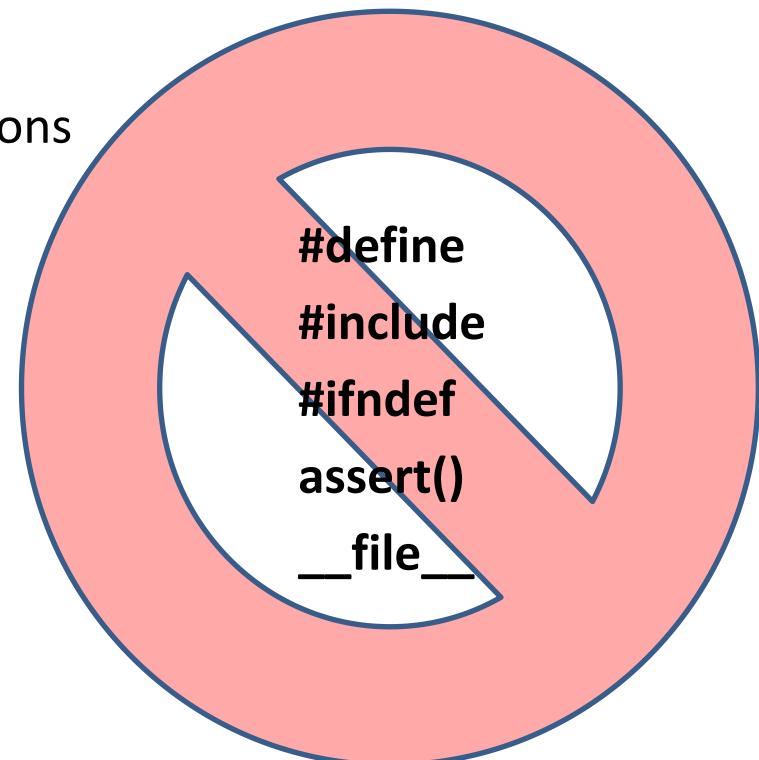
By “major”  
I mean  
“changes  
how we think”

- Major language features
  - Modules
  - Concepts
  - Coroutines
  - Improved compile-time programming support
- Major standard-library components
  - Ranges
  - Dates
  - Formats
  - Parallel algorithms
  - Span
- Many minor language features and standard-library components
- A dense web of interrelated mutually-supporting features
- Most is shipping already



# Making the preprocessor redundant

- Templates
  - Concepts
- Type deduction
- Compile-time computation
  - Constexpr and consteval functions
  - Static reflection (not yet)
  - Type traits
- Modules
- std::source\_location
- Contracts (not yet)



#define  
#include  
#ifndef  
assert()  
\_\_file\_\_

# Compile-time computation

- Not just built-in types
  - Examples from `<chrono>`



Howard Hinnant

```
cout << weekday{June/21/2016};           // Tuesday
static_assert( weekday{June/21/2016}==Tuesday ); // At compile time
```

```
static_assert(2020y/February/Tuesday[last] == February/25/2020); // true
```

```
auto tp = system_clock::now();
cout << tp;                                // 2019-11-14 10:13:40.785346
cout << zoned_time tp{current_zone(),tp}; // 2019-11-14 11:13:40.785346 CET
```

# Compile-time computation

- Compile-time computation tends to be invisible

```
auto z = sqrt(3+2.7i);           // call sqrt(complex<double>)
auto d = 5min+10s+200us+300ns;    // a duration
auto s = "This is not a pointer to char"s; // a string

// implementations:
constexpr complex<double> operator""i(long double d) { return {0,d}; }
constexpr seconds operator""s(unsigned long long s) {return s; }
constexpr string operator""s(const char* str, size_t len) { return {str, len}; }
```

# Template argument type deduction

- Help avoid range errors

```
int a[128];
// ... fill a ...
std::span s {a};
for (const auto x : s) cout << x << ' ';
```

No repeated element type  
No repeated array bound

- Help avoid data races

```
void do_something()
{
    std::scoped_lock lck {mut1,mut2};
    // ... Manipulate shared data ...
}
```

No repeated mutex type  
No mutex order dependence

# Compile-time computation

**constexpr** functions  
**consteval** functions  
Template aliases  
UDL  
Concepts  
Type deduction  
Template aliases  
Type traits



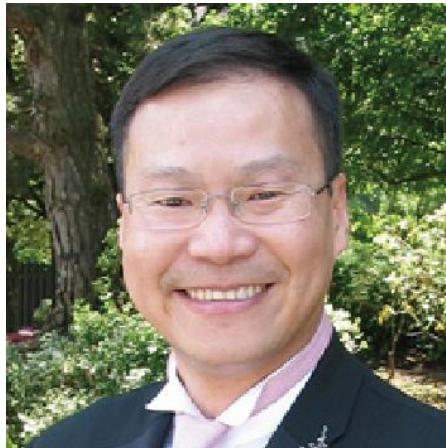
Daveed Vandevoorde



Bjarne Stroustrup



Gabriel Dos Reis



Michael Wong



Michael Spertus

# Modules

Finally!

- Order dependence

```
#include "a.h"  
#include "b.h"
```
- Can be different from

```
#include "b.h"  
#include "a.h"
```
- **#include** is textual inclusion
- Implies
  - **#include** is transitive
  - much repeated compilation
  - Subtle bugs
- Modularity

```
import a;  
import b;
```
- Same as

```
import b;  
import a;
```
- **import** is not transitive
- Implies
  - Much compilation can be done once only

# Modules history

- 1994: D&E: “Furthermore, I am of the opinion that Cpp must be destroyed.”
  - CPP == C preprocessor, not C++ 😊
  - **include** (no #) as the obvious alternative to **#include** with “module semantics”
- 2005-2012: Daveed Vandevoorde (EDG):
- 2011: Doug Gregor (Apple), C/Objective-C approach presentations
- Stroustrup and Dos Reis: IPR 2007
- Dos Reis (Microsoft): 2014
- Richard Smith (Google): 2016 (based on the C/Objective-C approach)
- 2017 Atom: revised Google design
- Gaby & Richard: 2017: Joint proposal
- Nathan Sidwell (Facebook) GCC enters the fray
- Gaby, Richard, Nathan, and Daveed: Feb 2019: Get it done now!
- Kona, Feb 2019: Voted in!!!
- 2020 staring to ship

# Modules

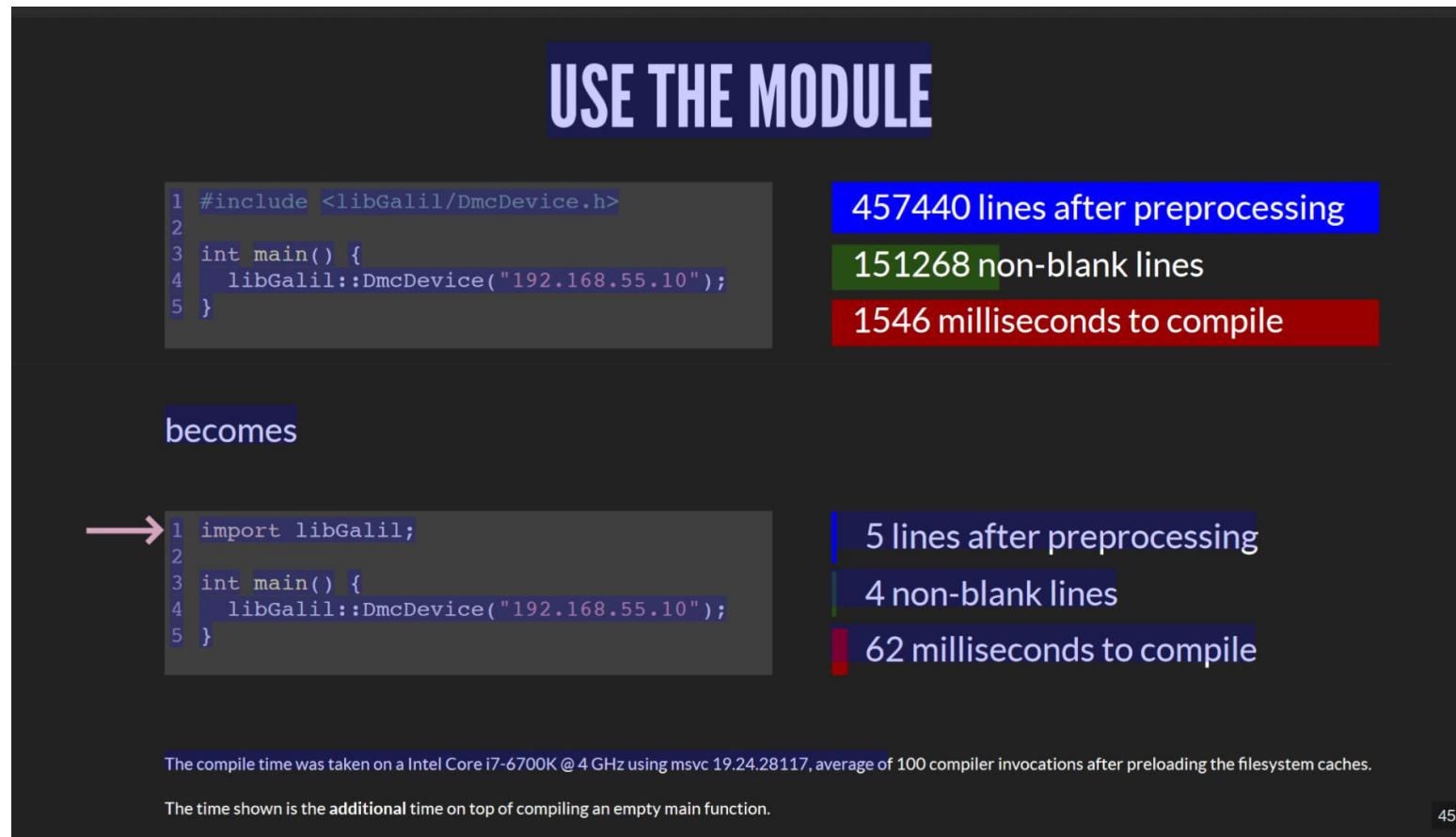
- Better code hygiene: modularity (especially protection from macros)
  - => Faster compile times (factors rather than percents)

```
export module map_printer;           // we are defining a module

import iostream;
import containers;
using namespace std;

export                      // this template is the only entity exported
template<Sequence S>
void print_map(const S& m) {
    for (const auto& [key,val] : m)    // break out key and value
        cout << key << " -> " << val << '\n';
}
```

# Compile speeds



- A 25 times speedup
  - Don't expect that in all cases

# Modularity and transition

- Support for getting from the **#include** world to the **import** world

- Global module
  - Modular headers
  - Module partitions

- A module needs not be in a single source file

```
module;
```

```
#include "xx.h"
```

*// to global module*

```
export module C;
```

```
import "a.h"
```

*// “modular headers”*

```
import "b.h"
```

```
import A;
```

```
export int f() { ... }
```

Here,  
**#include** works as ever

Potentially  
holding back progress

- Not yet: a modular standard library

- Versions exist, but not yet in the ISO standard
  - I hope for **import std**;

# Modules and transition

- Source organization
- Header file conversion
  - Header and module coexistence
- Build systems
  - Build2
  - CNUmake prototype



Nathan Sidwell



Gabriel Dos Reis



Richard Smith

# Generic Programming

- Write code that works for all suitable argument types
  - `void sort(R); // pseudo declaration`
    - R can be any sequence with random access
    - R's elements can be compared using <
  - `E* find_if(R,P); // pseudo declaration`
    - R can be any sequence that you can read from sequentially
    - P must be a predicate on R's element type
    - E\* must point to the found element of R if any (or one beyond the end)
- That's what the standard says
  - “our job” is to tell this to the compiler
  - C++20 enables that

# Generic programming:

## The backbone of the C++ standard library

- Containers
  - vector, list, stack, queue, priority\_queue, ...
- Ranges
- Algorithms
  - sort(), partial\_sort(), is\_sorted(), merge(), find(), find\_if(),...
  - Most with parallel and vectorized versions
- Concurrency support (type safe)
  - Threads, locks, futures, ...
- Time
  - time\_points, durations, calendars, time\_zones
- Random numbers
  - distributions and engines (lots)
- Numeric types and algorithms
  - complex
  - accumulate(), inner\_product(), iota(), ...
- Strings and Regular expressions
- Formats

Parameterized  
Types and algorithms

Type deduction

RAII

# Generic Programming

- Write code that works for all suitable argument types

```
void sort(Sortable_range auto& r);
```

```
vector<string> vs;  
// ... fill vs ...  
sort(vs);
```

```
array<int,128> ai;  
// ... fill ai ...  
sort(ai);
```

A concept:

- Specifies what is required of r's type

Implicit:

- Type of container
- Type of element
- Number of elements
- Comparison criteria

# Generic Programming

- Write code that works for all suitable argument types
  - Many/most algorithms have more than one template argument type
  - We need to express relationships among template arguments

```
template<input_range R, indirect_unary_predicate<iterator_t<R> Pred>
Iterator_t<R> ranges::find_if(R&& r, Pred p);
```

```
list<int> lsti;
// ... fill lsti ...
auto p = find_if(lsti, greater_than{7});
```

<ranges>

```
vector<string> vs;
// ... fill vs ...
auto q = find_if(vs, [](const string& s) { return has_vowels(s); });
```

# Overloading

- Overloading based on concepts

```
void sort(ForwardSortableRange auto&);
```

```
void sort(SortableRange auto&);
```

```
void some_code(vector<int> vec&, list<int> lst)
{
    sort(lst);           // sort(ForwardSortableRange auto&)
    sort(vec);          // sort(SortableRange auto&)
}
```

- We don't have to say
  - “**Sortable\_range** is stricter/better than **ForwardSortableRange**”
  - we compute that from their definitions

Design principles:

- Don't force the user to do what a machine does better
- Zero overhead compared to unconstrained templates

# Concepts

- A concept is a compile-time predicate
  - A function run at compile time yielding a Boolean
  - Often built from other concepts

```
template<typename R>  
concept Sortable_range =  
    random_access_range<R>  
    && sortable<iterator_t<R>>;
```

There are libraries of concepts  
<ranges>: random\_access\_range and sortable

*// has begin()/end(), ++, [], +, ...*  
*// can compare and swap elements*

```
template<typename R>  
concept ForwardSortable_range =  
    forward_range<R>  
    && sortable<iterator_t<R>>;
```

*// has begin()/end(), ++; no [] or +*  
*// can compare and swap elements*

# Concepts

- A concept is a compile-time predicate
  - A function runs at compile time yielding a Boolean
  - One or more arguments
  - Can be built from fundamental language properties: use patterns

```
template<typename T, typename U = T>
concept equality_comparable = requires(T a, U b) {
    {a==b} -> bool;
    {a!=b} -> bool;
    {b==a} -> bool;
    {b!=a} -> bool;
}
```

There are libraries of concepts  
`<concepts>: equality_comparable`

# Types and concepts

- A type
  - Specifies the set of operations that can be applied to an object
    - Implicitly and explicitly
    - Relies on function declarations and language rules
  - Specifies how an object is laid out in memory
- A ***single-argument*** concept
  - Specifies the set of operations that can be applied to an object
    - Implicitly and explicitly
    - Relies on use patterns
      - reflecting function declarations and language rules
  - Says nothing about the layout of the object

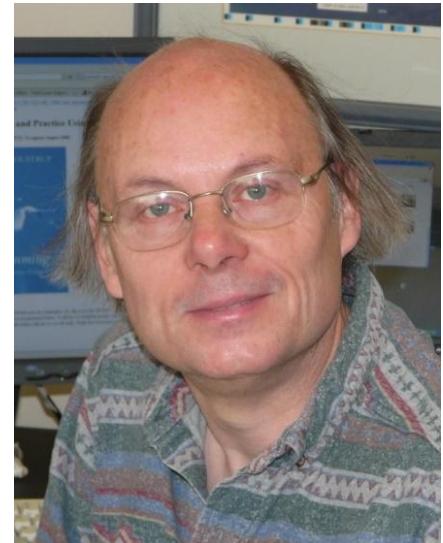
Ideal:

Use concepts where we now use types,  
except for defining layout

# Generic Programming with concepts



Gabriel Dos Reis



Bjarne Stroustrup

will change the way we think about Programming



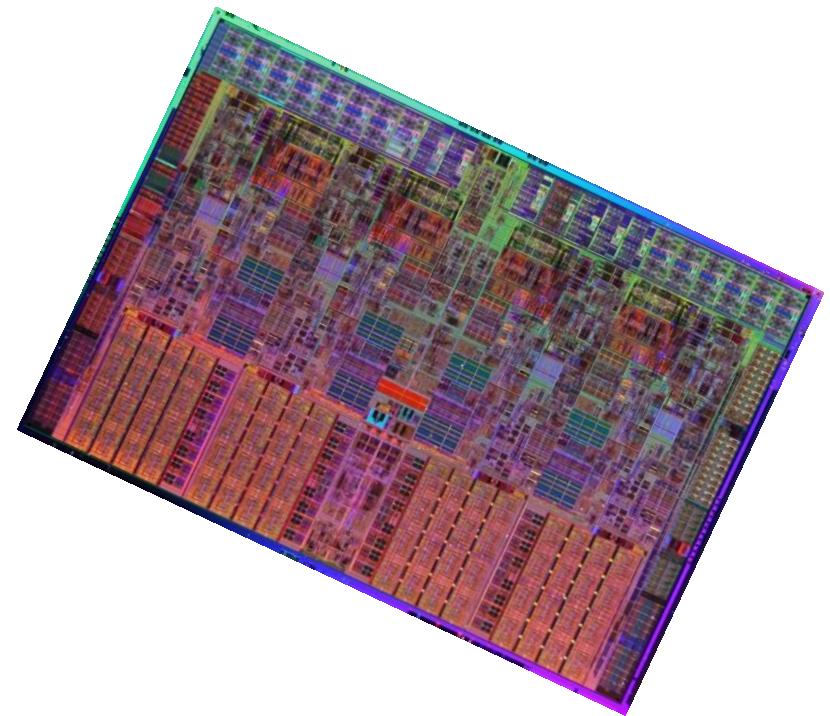
Andrew Sutton



Alex Stepanov

# Concurrency and parallelism

- What we want
  - Ease of programming
    - Writing correct concurrent code is hard
    - Modern hardware is concurrent in more ways than you imagine
  - Uncompromising performance
    - But for what?
    - Different application area really differ
  - Portability
    - Preferably portable performance
  - System level interoperability
    - C++ shares threads with other languages and with the OSs
  - Standard model for different architectures
    - CPU, GPU, FPGA, ...



# Concurrency and parallelism

- C++11 – the traditional foundation
  - Memory model
  - Lock-free programming
  - Type-safe POSIX/Windows-style threads, mutex, etc.
  - Future and promise
  - Atomics
- C++14, C++17, and C++20
  - Many minor improvements (e.g., jthread, stop tokens, scoped\_lock)
  - Fences and barriers
  - Parallel algorithms
  - Coroutines (synchronous and asynchronous)
- C++23?
  - Executor model (readers and writers, push and pull)
  - Networking (now a TS)
  - SIMD support



# Threading

- A **thread** represents a system's notion of executing a task concurrently with other tasks
- You can
  - start a task on a thread
  - wait for a thread for a **specified time**
  - control access to some data by **mutual exclusion**
  - control access to some data using **locks**
  - wait for an action of another task using a **condition variable**
  - return a value from a thread through a **future**
- Threads and locks is a foundation
  - not a good model for applications
  - Type safe
  - Convenient notation

# Avoid deadlocks

- Unstructured locking is dangerous

- Simple locking (RAII)

```
mutex m1;
```

```
int sh1;      // shared data
```

```
mutex m2;
```

```
int sh2;      // some other shared data
```

```
void obvious()
```

```
{
```

```
// ...
```

```
scoped_lock lck1 {m1,m2}; // acquire both locks
```

```
// manipulate shared data:
```

```
sh1+=sh2;
```

```
} // release both locks
```

# Reader/writer locks

```
shared_mutex mx;           // a mutex that can be shared

void reader()
{
    shared_lock lck {mx};   // willing to share access with other readers
    // ... read ...
}

void writer()
{
    unique_lock lck {mx};   // needs exclusive (unique) access
    // ... write ...
}
```

# Threading and interruption

- What if you decide that the result of a thread isn't needed?
  - E.g., `find_any()` after some thread found “it”

```
auto my_task = [] (stop_token tok)
{
    while (!tok.stop_requested()) {      // is a result still needed?
        // ... do work ...
    }
};

void user()
{
    jthread t1 { my_task };           // stop_token implicitly supplied by jthread
    jthread t2 { my_task };
    // ...
    if (t1_no_longer_needed) t1.request_stop();
    // ...
}
```

# Coroutines

Preserve state between calls

```
generator<int> fibonacci() // generate 0,1,1,2,3,5,8,13 ...
{
    int a = 0; // initial values
    int b = 1;
    while (true) {
        int next = a+b;
        co_yield a;           // return next Fibonacci number
        a = b;                // update values
        b = next;
    }
}
f
int main()
{
    for (auto v: fibonacci())
        cout << v << '\n';
}
```

Fast pipelines and generators  
Simple asynchronous programming

# Coroutines – Async I/O

```
task<> start() {
    char data[1024];
    for (;;) {
        size_t n = co_await socket.async_read_some(buffer(data));
        co_await async_write(socket, buffer(data, n));
    }
}
```

- no live demo, but see
  - `cppcoro` (Windows-only)
    - <https://github.com/lewissbaker/cppcoro#socket>
  - `boost.ASIO`
    - [https://www.boost.org/doc/libs/1\\_69\\_0/doc/html/boost\\_asio/example/cpp17.coroutines\\_ts/echo\\_server.cpp](https://www.boost.org/doc/libs/1_69_0/doc/html/boost_asio/example/cpp17.coroutines_ts/echo_server.cpp)

# Parallel algorithms

- algorithms giving the option of parallel and/or vectorised execution of standard-library algorithms
  - e.g, **sort(par,b,e)** and **sort(unseq,b,e)**
- All the traditional STL algorithms, e.g., **find(seq,b,e,x)**,
  - but no **find\_all(par,b,e,x)** or **find\_any(unseq,b,e,x)**
- parallel algorithms:
  - For\_each
  - Reduce // parallel accumulate
  - Exclusive scan
  - Inclusive scan
  - Transform reduce
  - Transform exclusive scan
  - Transform inclusive scan
  - ...

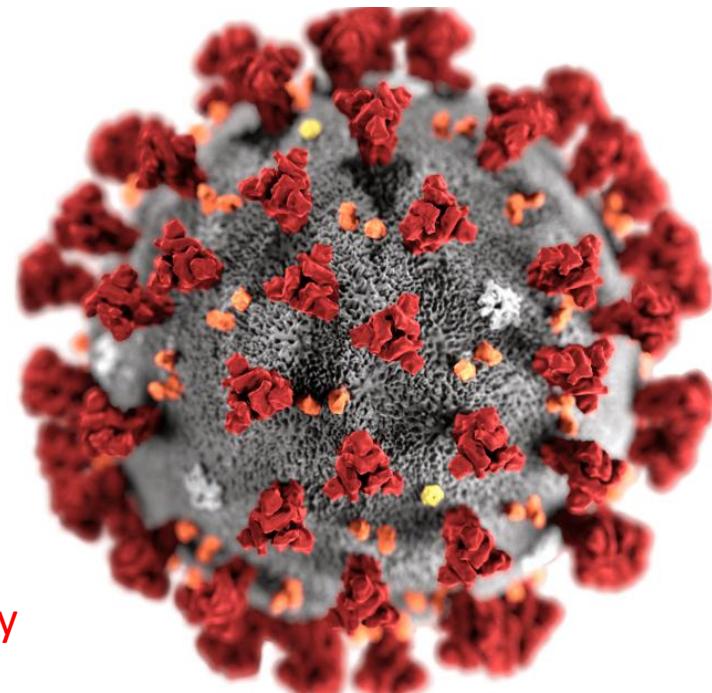
# C++23 – we have a plan

- Top priorities:
    - Library support for coroutines
    - A modular standard library
    - Executors
    - Networking
  - Also make progress on
    - Reflection
    - Pattern matching
    - Contracts
  - After that
    - Everything else
- Be careful  
Remember the Vasa!



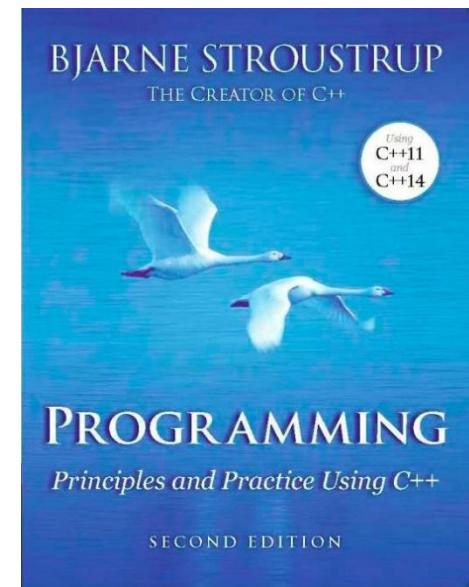
# C++23 – we have a plan

- Top priorities:
    - Library support for coroutines
    - A modular standard library
    - Executors
    - Networking
  - Also make progress on
    - Reflection
    - Pattern matching
    - Contracts
  - After that
    - Everything else
- Don't be too impatient  
Even the best plan can run into problems
- We ship a feature  
**only** when it's ready



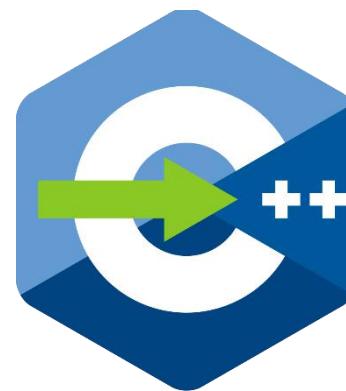
# References

- All the standards committee papers: search for WG21
- C++ Core Guidelines: GitHub
- C++ Foundation website: Isocpp.org
- Online reference: cppreference.com
- On-line compilers: C++ explorer, etc.
- C++ compilers: Clang, IBM, Intel, GCC, Microsoft, ...
- C++ libraries: <https://en.cppreference.com/w/cpp/links/libs> (incomplete, of course)
- Conference videos (hundreds: Cppcon, Meeting C++, etc.): Youtube
- ***A Tour of C++ (2<sup>nd</sup> Edition)***
- ***Programming: Principles and Practice using C++ (2<sup>nd</sup> Edition)***
- My HOPL (History of Programming Papers): [www.stroustrup.com/papers.html](http://www.stroustrup.com/papers.html)
  - In particular: ***Thriving in a crowded and changing world: C++ 2006–2020***
- More of “my stuff” (papers, interviews, videos, etc.): [www.stroustrup.com](http://www.stroustrup.com)



# C++

- C++20
  - Completed February 15, 2020
  - Most features shipping somewhere
  - Expected: essentially all features shipping by all major vendors in 2020
  - Is going to make a major difference to the way we think and program
  - Compatible / stable
- Use C++ as a modern language
  - Aim for complete type-safety and resource-safety
  - Enforce coding guidelines



# Titus Winters

Google C++代码库总负责人，C++标准库委员会主席



Titus Winters自2010年加入Google，是Google 2.5亿行C++代码库的总负责人，他领导的团队负责组织、维护、演化、重构Google C++代码库的基础组件，他在Google主导的重构项目据信是人类历史上Top 10的最大代码规模项目。他也是Google官方C++代码规范的核心制定者，是Google内部架构守护、代码一致性、自动化工具、持续集成等软件工程规范的主导者。Titus 目前还担任C++标准库（C++ Standard Library）委员会的主席。他也是《Google的软件工程实践》五星畅销书的作者。

主办方：

**Boolan**  
高端IT咨询与教育平台



# Software Engineering - Principles

aka SWE Book Theses

# What's the Secret?

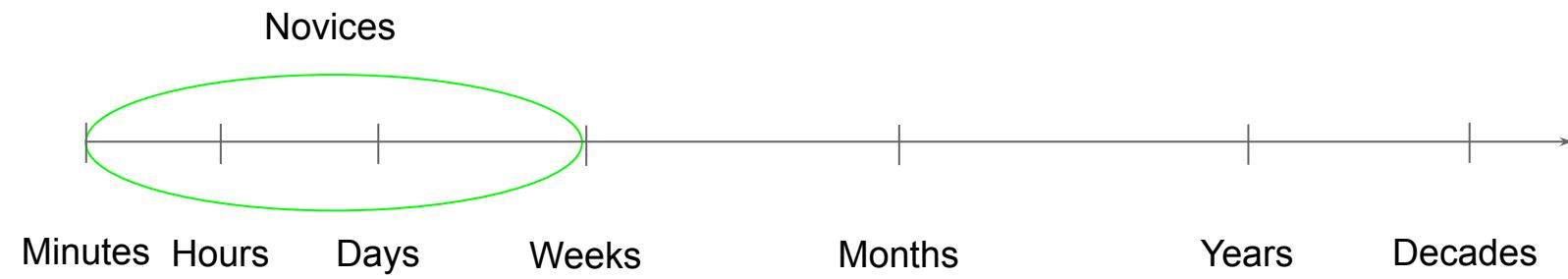
# What's the Secret?

Hint: no silver bullet

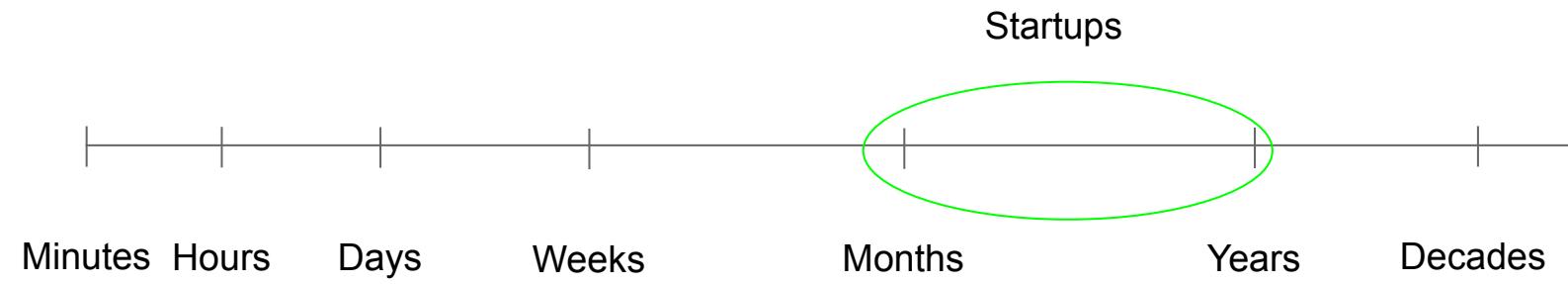
## Principle #1 - Time

What's the expected lifespan of this code?

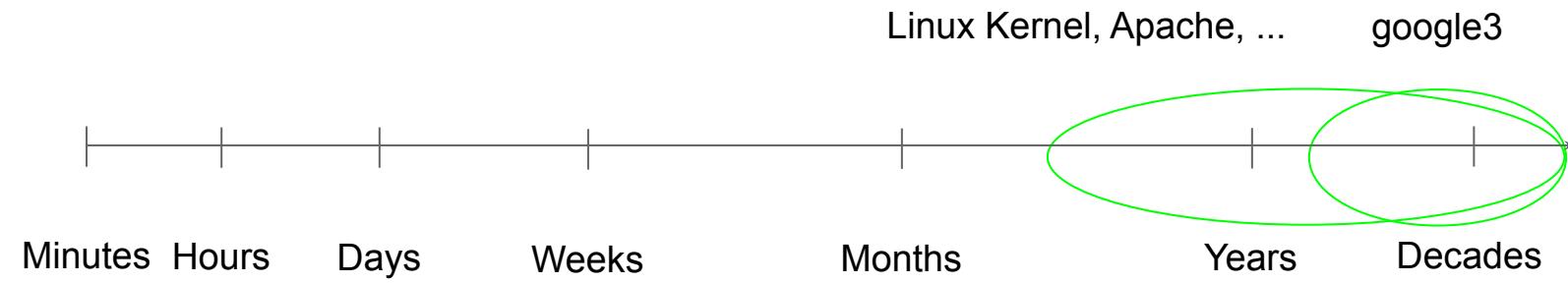
# Principle #1 - Time



# Principle #1 - Time



# Principle #1 - Time



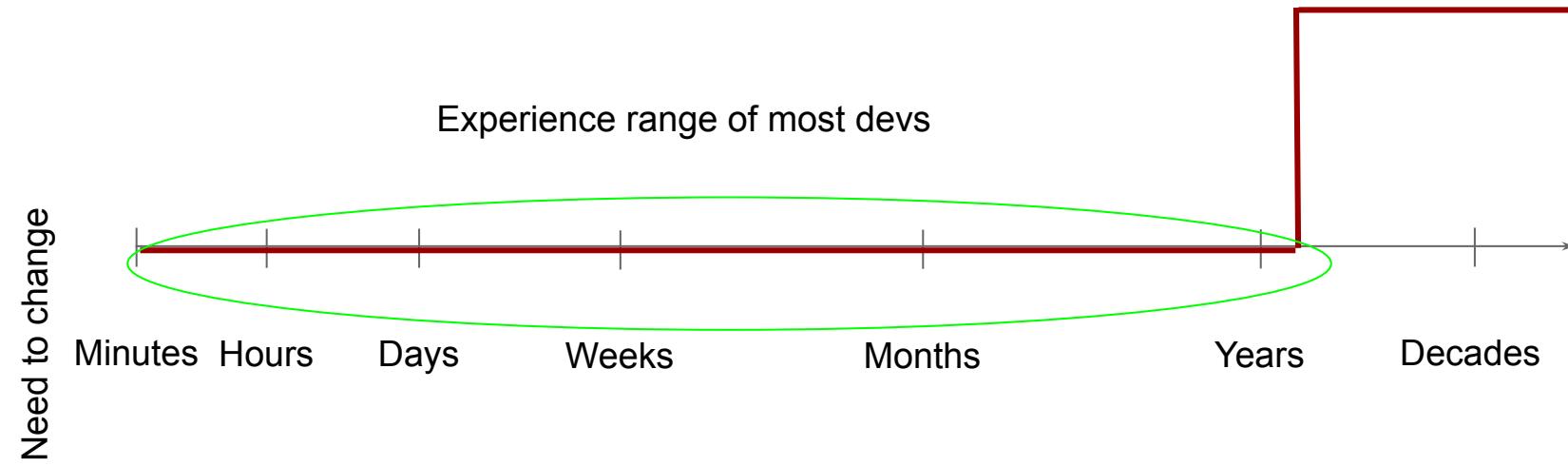
# Principle #1 - Time



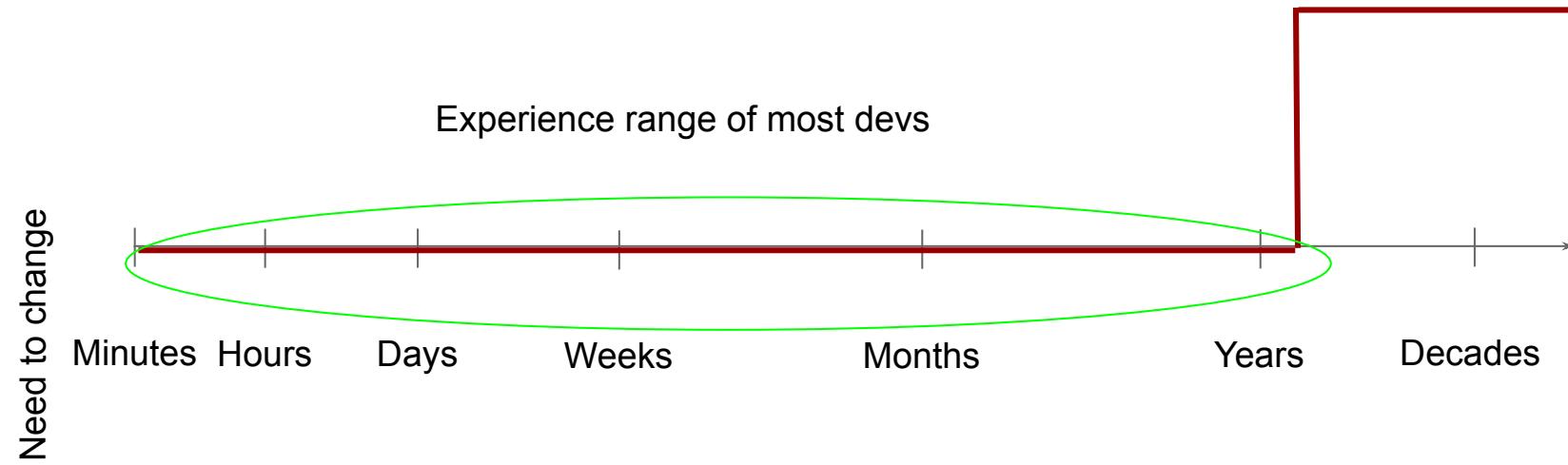
# Principle #1 - Time



# Principle #1 - Time



## Principle #1 - Time



- Performing a task that hasn't been done to this project before
- Doing it without expertise/experience
- Doing a larger-than-usual jump (e.g. v1 to v5, rather than just v5 to v6)

## Principle #1 - Time

Sustainability is the goal: for the expected lifespan of your code, you are **able** to change all of the things that you ought to change, safely.

## Principle #1 - Time

Many developers have never worked on a sustainable project  
with a recognized 5+ year lifespan.

## Principle #1 - Time

Software Engineering is not merely programming -  
it is the art of making a program resilient to change  
over time.

## Principle #1 - Time

- Keep in mind the expected lifespan
- Understand that long lifespans are rare, hard to plan for, and not well understood
- Sustainable code is capable of change - that probably means different things at different time scales.
- Sustainable is often hard to get to.

## Principles #2: Scale

When change over time leads to growth,  
where do we start to fail?

## Principles #2: Scale

- Hardware resources (CPU, RAM, Disk, Network)
- Software resources (Addresses, ports)
- Human resources

## Principles #2: Scale

Traditional Deprecation:

- **Mark the old version deprecated, introduce a new one, and call it good.**

## Principles #2: Scale

Traditional Deprecation:

- Mark the old version deprecated, introduce a new one, and call it good.
- **Mark the old version deprecated, introduce a new one, and mandate everyone update their code by some date. Delete the old one on that date.**

## Principles #2: Scale

Traditional Deprecation:

- Mark the old version deprecated, introduce a new one, and call it good.
- Mark the old version deprecated, introduce a new one, and mandate everyone update their code by some date. Delete the old one on that date.
- **Find a brave engineer to go through and build a single change that modifies the API in question and all of the callers and land that refactoring in one step.**

## Principles #2: Scale

Traditional Deprecation:

- Mark the old version deprecated, introduce a new one, and call it good.
- Mark the old version deprecated, introduce a new one, and mandate everyone update their code by some date. Delete the old one on that date.
- Find a brave engineer to go through and build a single change that modifies the API in question and all of the callers and land that refactoring in one step.

Better Deprecation:

- The team responsible does the bulk of the work.

## Principles #2: Scale

In a successful organization, everything that must be done repeatedly\* must consume sub-linear resources - especially sub-linear human resources.

# Scale: Weekly Merge Meeting

About 1 in 4 SWEs have had a regularly-scheduled meeting to discuss “merge schedule.”

- git makes it more common to have heavily-branched workflow
- long-lived dev branches are risky to merge
- manage the risk: merge carefully and rarely

How does this scale?

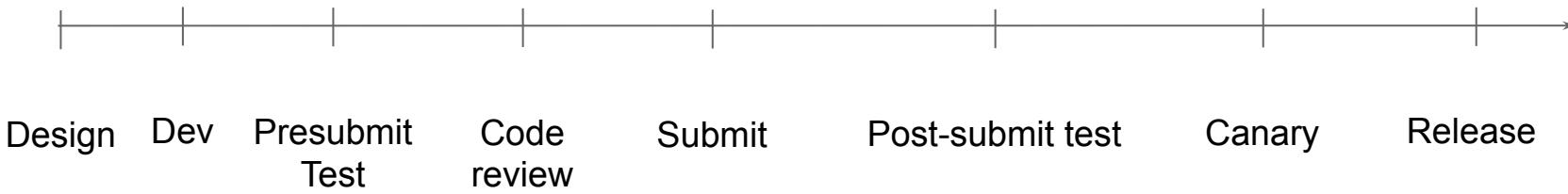
# Scale: No Weekly Merge Meeting

Published research: “Trunk-based development leads to better outcomes.”

- No long-lived dev branches
- No choices where to commit
- No choices which version to depend upon

# Time & Scale: Shifting Left

Published research: “Trunk-based development leads to better outcomes.”



## Principle #2 - Scale

- Be mindful of superlinear scaling costs
- Anything that must be done repeatedly by humans should be sub-linear
- Expertise and automation usually pay off super-linearly
- The “normal” way of doing things may have scaling problems.

## Principles #3: Tradeoffs

Make evidence-based decisions.

## Principles #3: Tradeoffs

Make evidence-based decisions.

Aim for sustainability.

## Principles #3: Tradeoffs

Make evidence-based decisions.

Aim for sustainability.

No super-linear scaling.

(Especially for humans.)

## Principles #3: Tradeoffs

Make evidence-based decisions.

Aim for sustainability.

No super-linear scaling.

(Especially for humans.)

Re-evaluate as needed.

# What's the Secret?

# Google SWE “Secrets”

1. Software Engineering is more than just programming, it's Time
  - o Especially consider the impact of time
2. Be mindful of scale
  - o Super-linear scaling is bad for required processes
  - o Expertise/specialists can provide super-linear impact in their domain
3. Make evidence-based decisions
  - o No “because I said so”
  - o Evidence will change over time, re-evaluate as needed

# Google SWE Book

- Pillars (these)
- Culture (happy devs, productive teams)
- Policies and Processes (how to make things work smoothly)
- Tools (tech)

# Google SWE “Secrets”

*It's programming if “clever” is a compliment.*

*It's software engineering if “clever” is an accusation.*



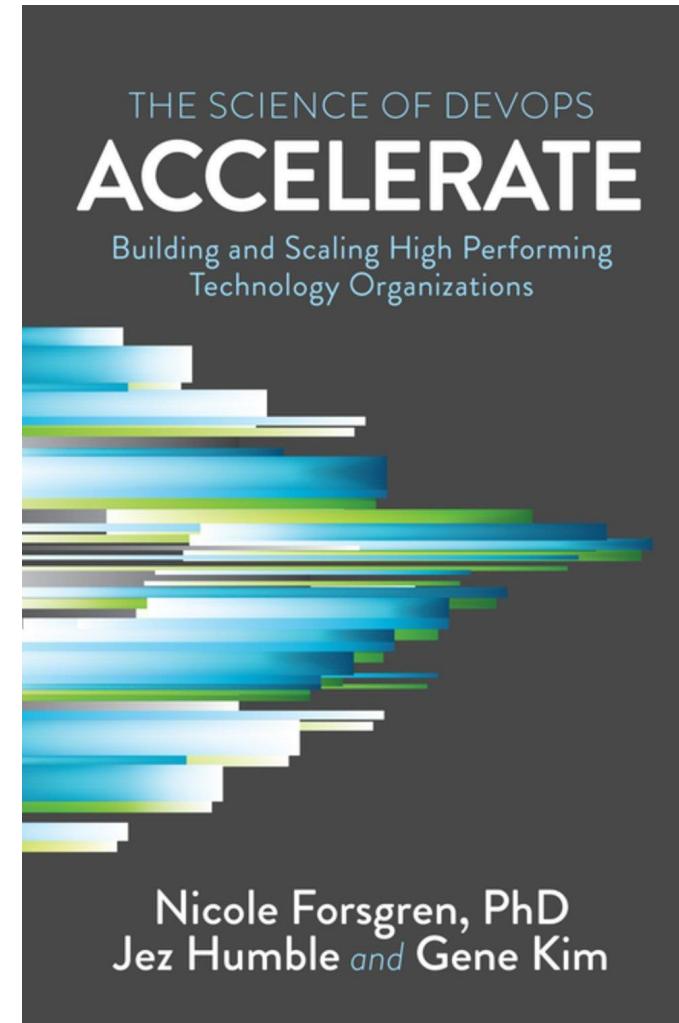
# Shifting Left

Cost vs. Fidelity, and Emerging Truths

Titus Winters

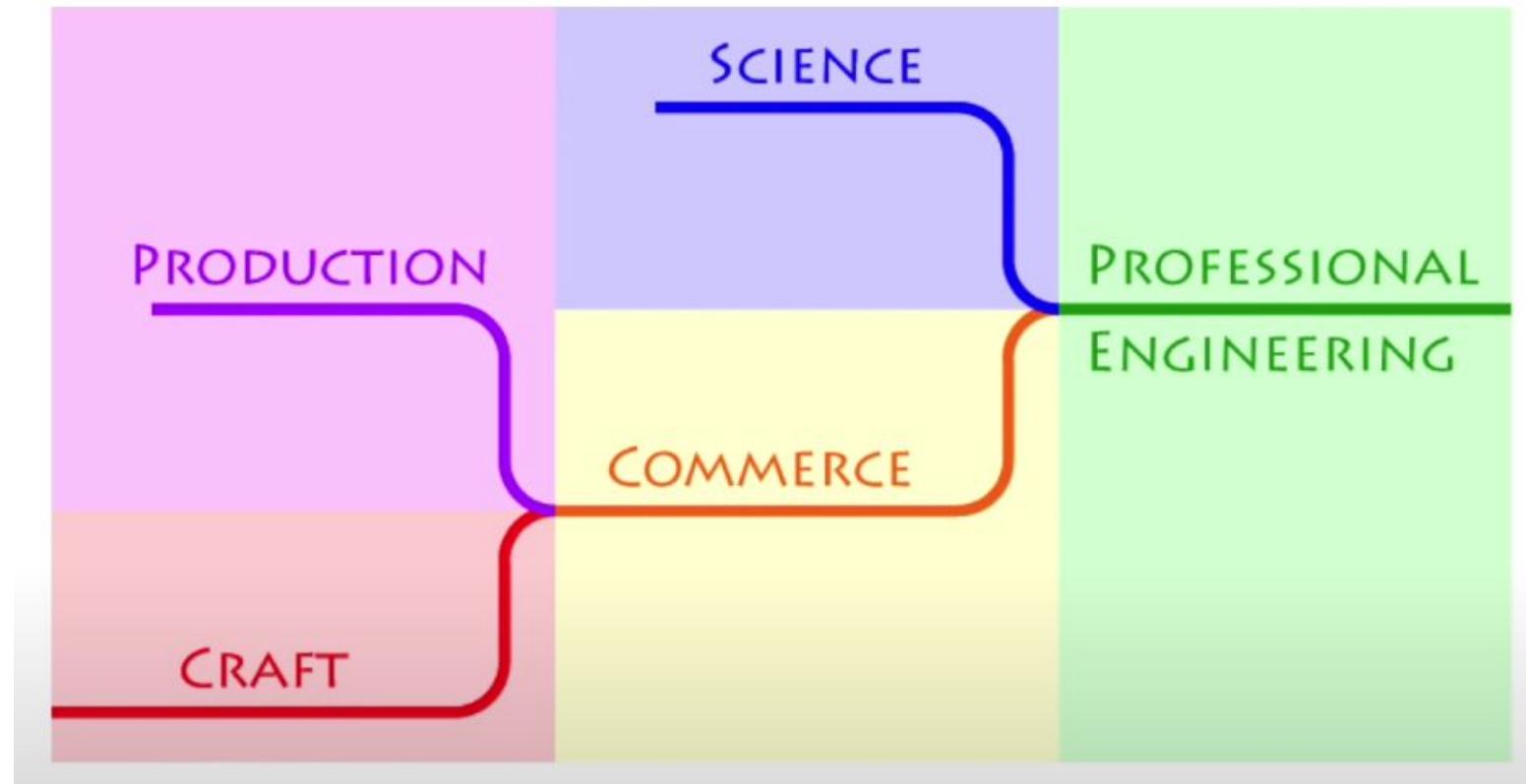
# Software Engineering

# Software Engineering?





GOTO 2015 • Progress Toward an Engineering Discipline of Software • Mary Shaw



# Software Engineering: First Contact

# Define “Software Engineering”

Dave Parnas: “The multi-person development of multi-version programs.” (Circa 1970)

Russ Cox: “Software engineering is what happens to programming when you add time and other programmers.” (Circa 2018)

“It's Programming if "clever" is a compliment.

It's Software Engineering if "clever" is an accusation.”

# Software Engineering != Programming

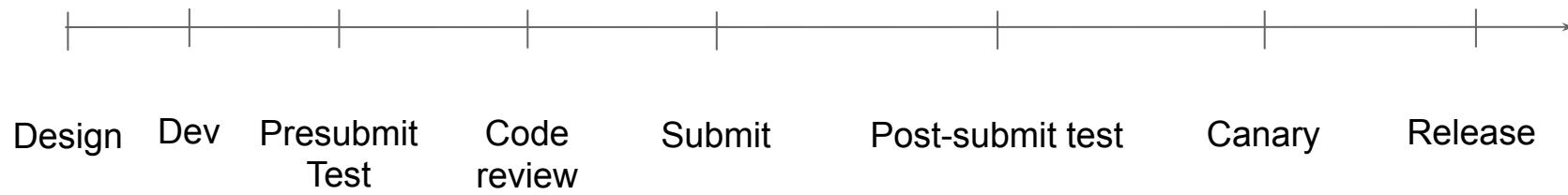
# Truth #1: Hyrum's Law

With a sufficient number of users of an API,  
it does not matter what you promise in the contract:  
all observable behaviors of your system will be depended on by somebody.

## Truth #2: Change will Happen

Over the expected lifespan of your code,  
**sustainable** code is capable of changing  
everything that *ought* to change, safely.

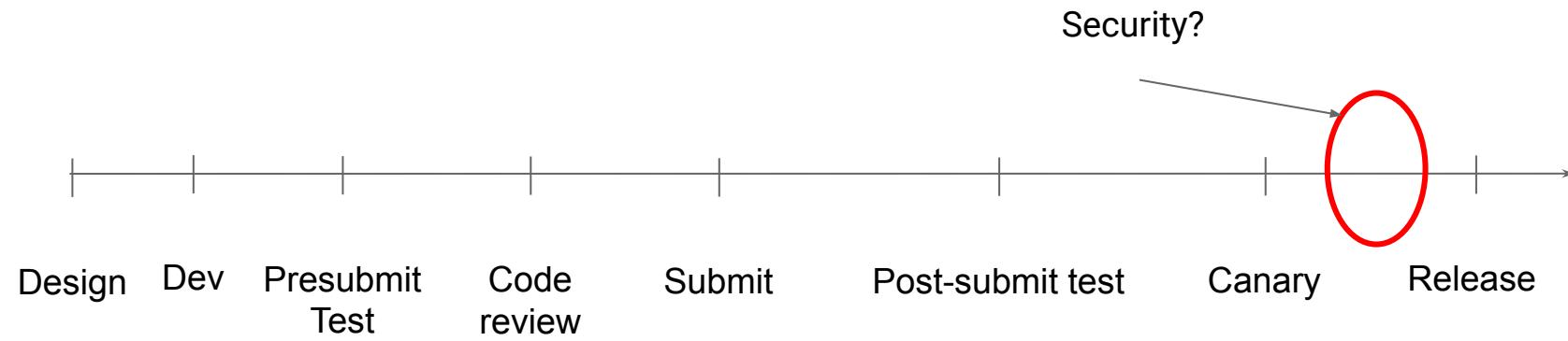
# Truth #3: Shifting Left - Earlier is Cheaper



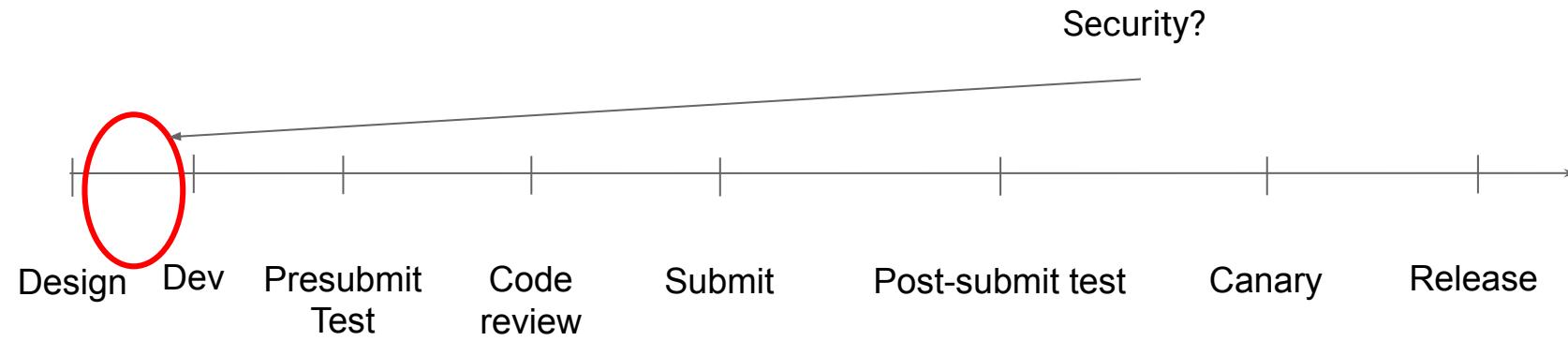
Shift Left

“Shift Left on Security”

# Shift Left



# Shift Left

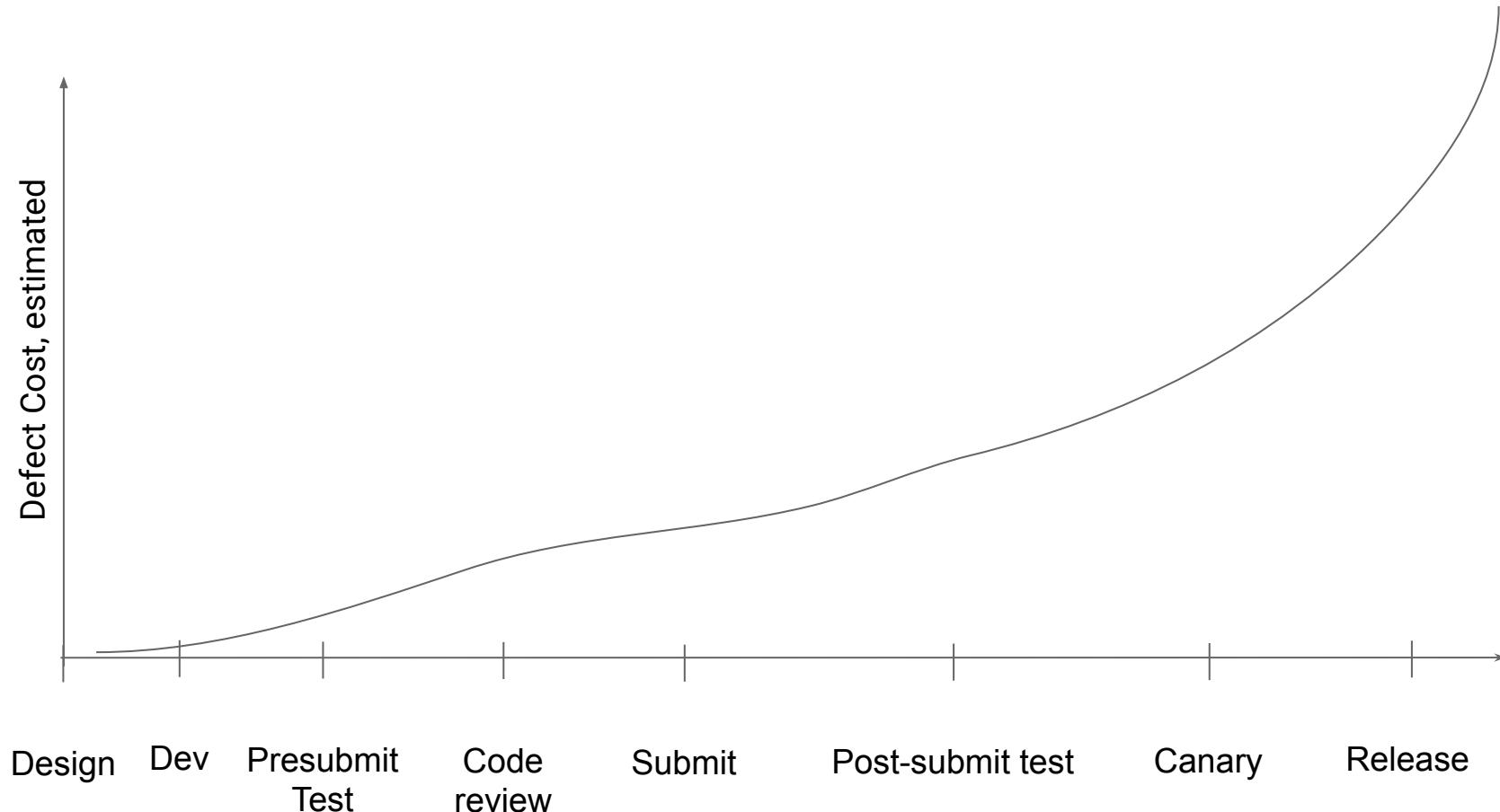


Shift Left

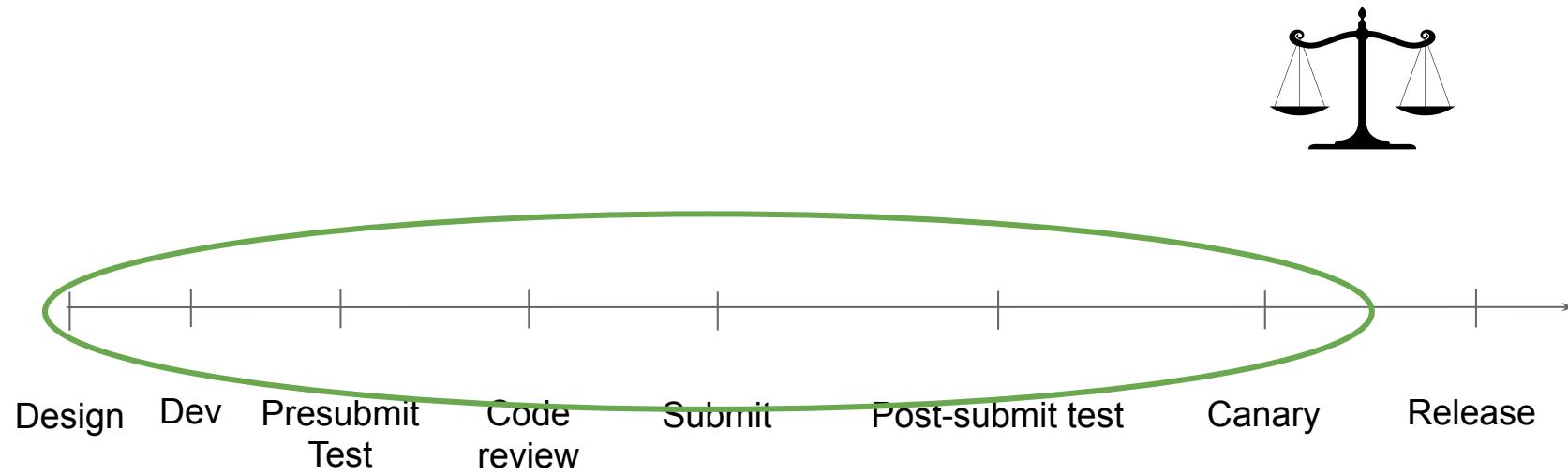
Defect Cost != Speed, Fun, Productivity

(Software Engineering != Programming)

# Shift Left



# Shift Left



Shift Left

Continuous Integration

is

Alerting

# CI is Alerting

Invariant checks:

- Tests are evaluating invariants tied to logical correctness
- Monitoring is based on application health invariants

Paging / Alerting because a poorly chosen line was crossed: brittle

Paging / Alerting off and on for unclear reason: flaky

These are (or could be) the same invariants!

# CI is Alerting

Highest ground-truth value alerts: Constant probing “Is the site up?”

Everything else: Proxies for “Is the site *healthy*? ”

(A *prediction* of whether the site will stay up.)

# CI is Alerting

Unitests

Monitored stats/metrics

Flaky / brittle tests

Flaky stats / cause-based alerts

End-to-end tests

End-to-end probers

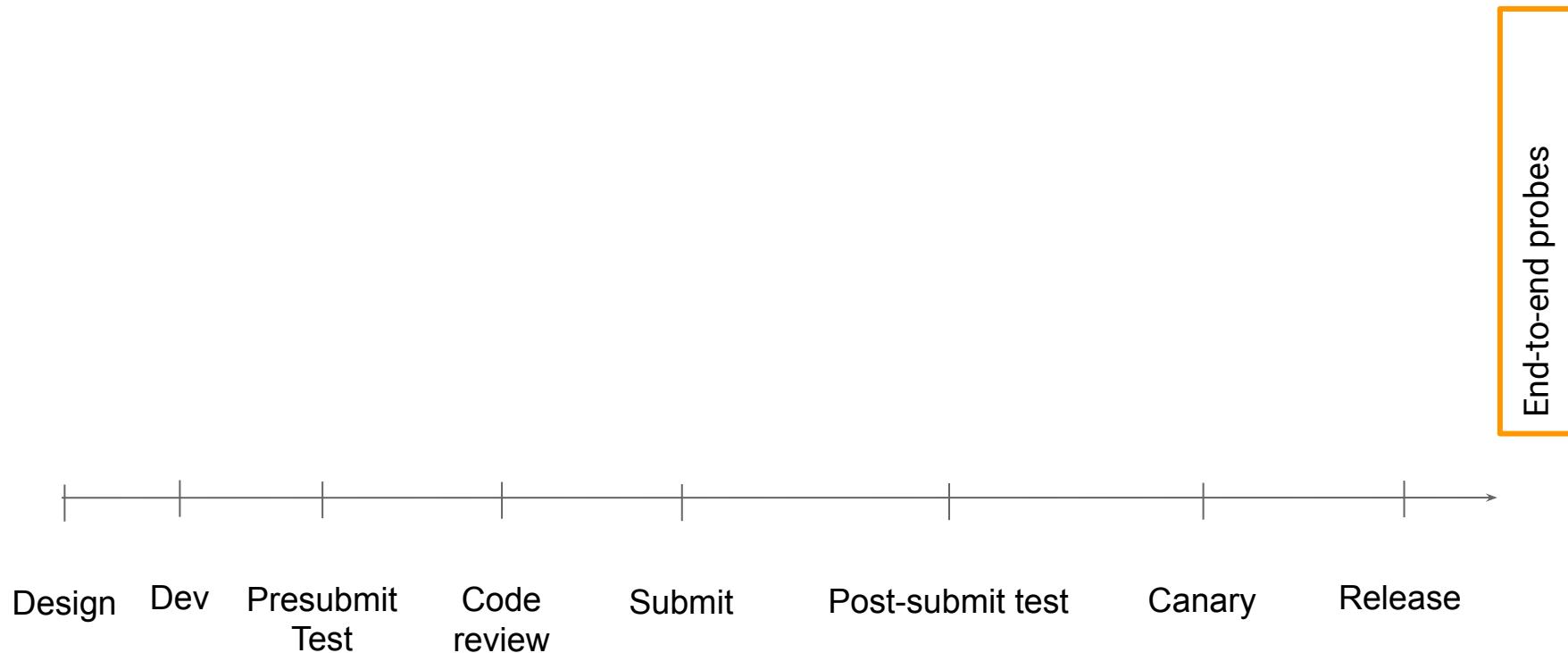
Keep trunk green

Error budgets

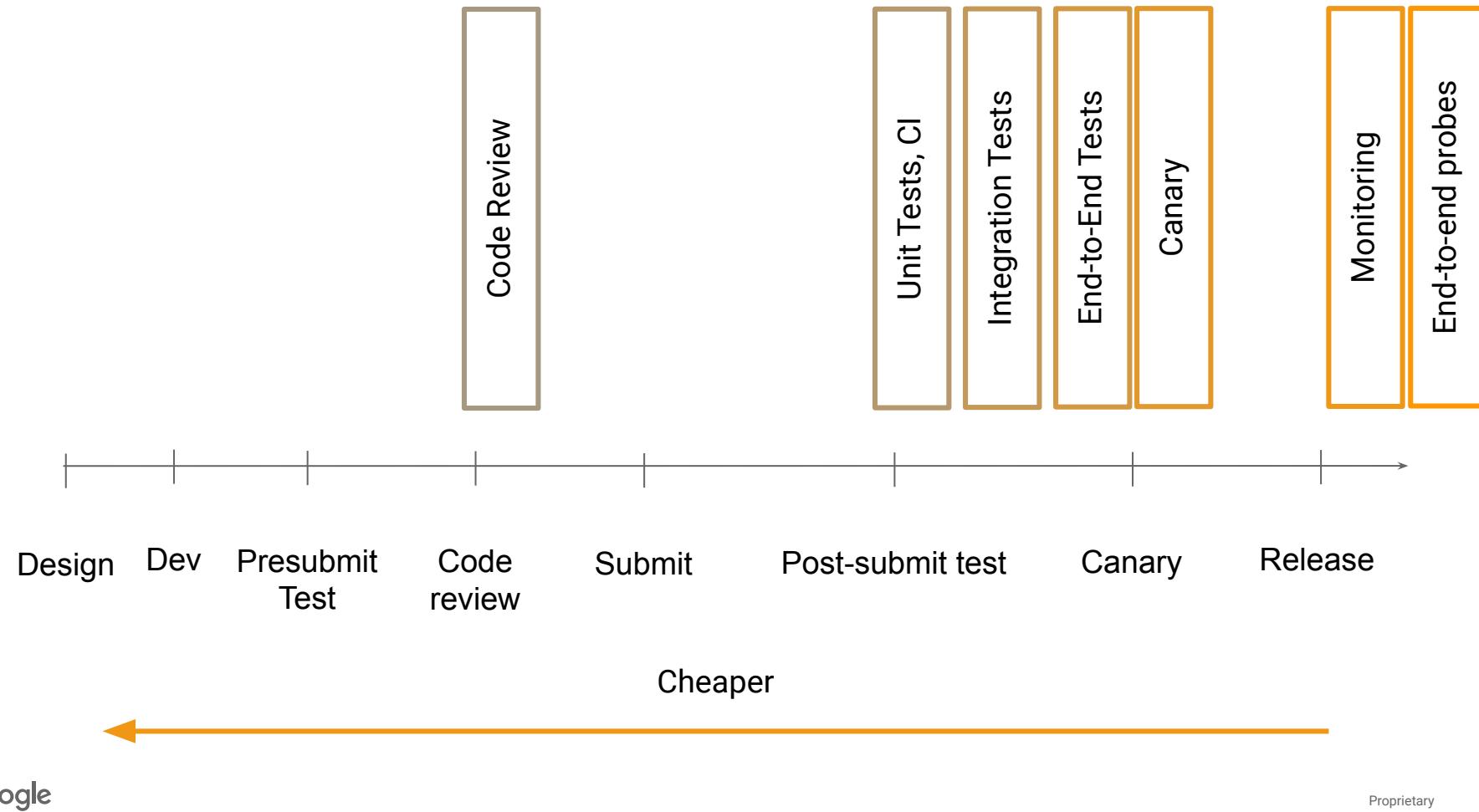
CI is Alerting

We're optimizing for “product fitness”

# Product Fitness, Proxies



# Product Fitness, Proxies



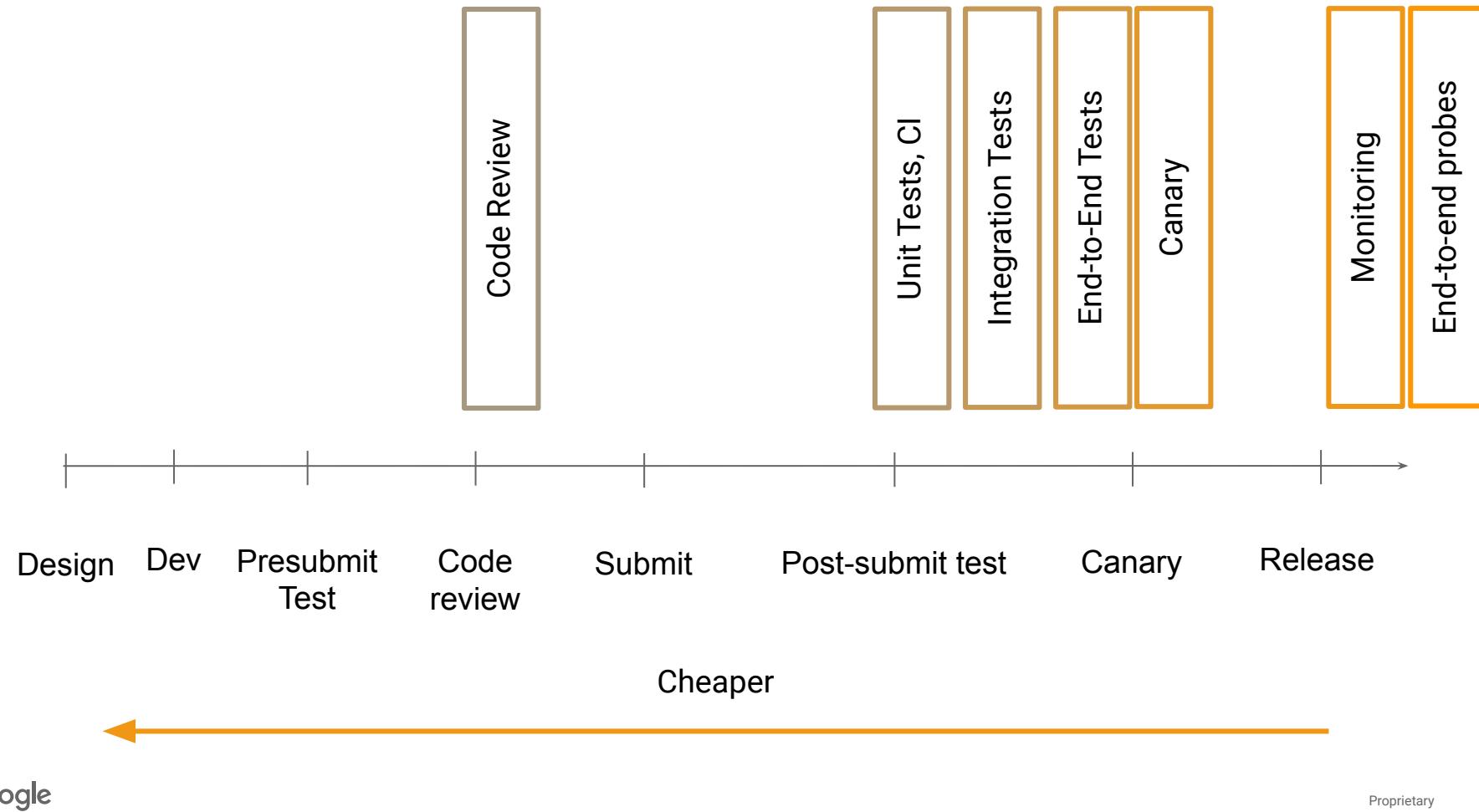
SRE Policies in the SWE Workflow

# Error Budgets

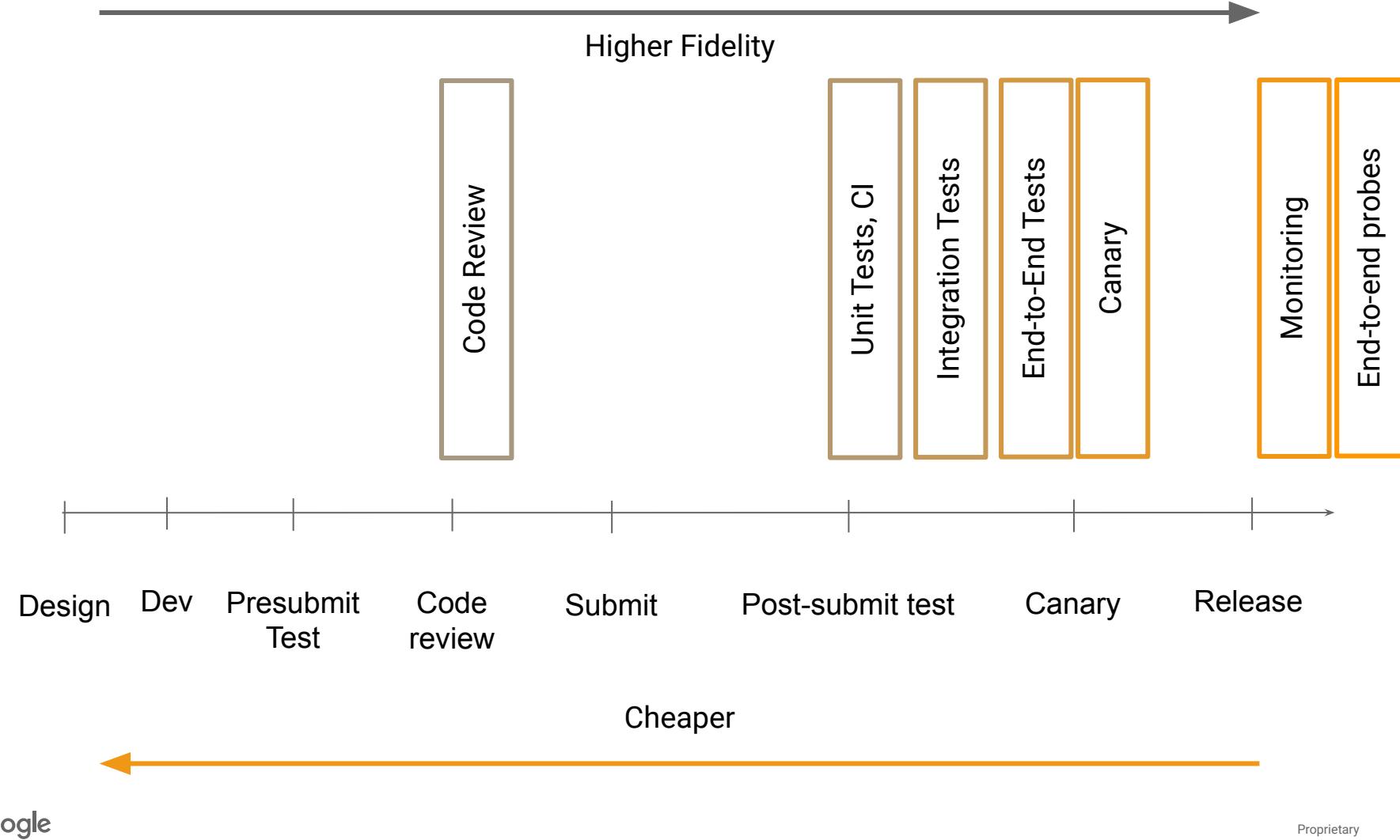
Shift Left

We're optimizing for “product fitness”

# Product Fitness, Proxies



# Product Fitness, Proxies



## Truth #3: Shifting Left

“Shifting Left” in your engineering workflow  
is trading between risk/cost and fidelity.

# The SWE Workflow

- Everything is about **intra-team** communication, product fitness, or both.
- Workflow that requires 1:N communication proportional to org size is a risk.
- Shifting any process to the right will eventually be a problem - those costs often grow super-linearly with your team size.
- Arguments of the form “but we already have X that solves this, so we don’t need Y” should be ignored when X is further to the right than Y.
- Arguments of the form “X catches more than Y, so let’s drop Y” should also be ignored when X is further to the right than Y. It may catch more, but it’s still cheaper to do it earlier if you can manage - and we aren’t aiming for perfection in any given stage.

# Software Engineering?

# Software Engineering as a Discipline

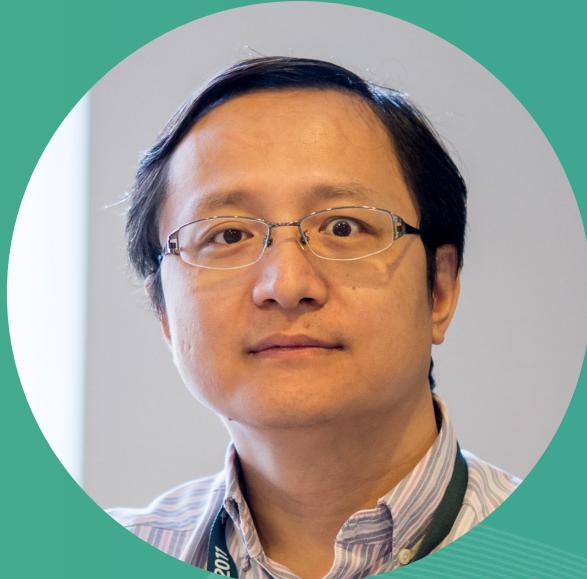
Some understanding of the boundary of the field.

Some emerging truths.

# Questions?

# 吴咏炜

Boolan首席咨询师，知名C++专家



国内知名 C++ 专家，曾任英特尔亚太研发中心资深系统架构师，近 30 年 C/C++ 系统级软件开发和架构经验。专注于 C/C++ 语言(包括 C++98/C++11/14/17/20)、软件架构、性能优化、设计模式和代码重用。对于精炼、易于维护的代码和架构有着不懈的追求，对开源平台(GNU/Linux)有深入的理解。长期担任资深技术教练，涉及 C++、软件架构、安全软件开发、开源软件等多方面。

主办方：

**Boolan**  
高端 IT 咨询与教育平台

C++ Summit 2020

吴咏炜

博览首席咨询师

# C++ 性能调优 纵横谈

# 自我介绍

- 学编程超过 35 年
- 近 30 年 C++ 老兵
- 热爱 C++ 和开源技术
- 多次在 C++ 大会上推广 C++ 新特性
- 对精炼、跨平台的代码有特别偏好
- 目前从事 C++ 方面的咨询工作

Premature optimization is the root of all evil.

We should forget about small efficiencies, say about 97 percent of the time: premature optimization is the root of all evil.

—Donald Knuth, “Structured programming with go to statements”, *ACM Computing Surveys*, **6**, 4 (December 1974), p. 268. CiteSeerX: 10.1.1.103.6084

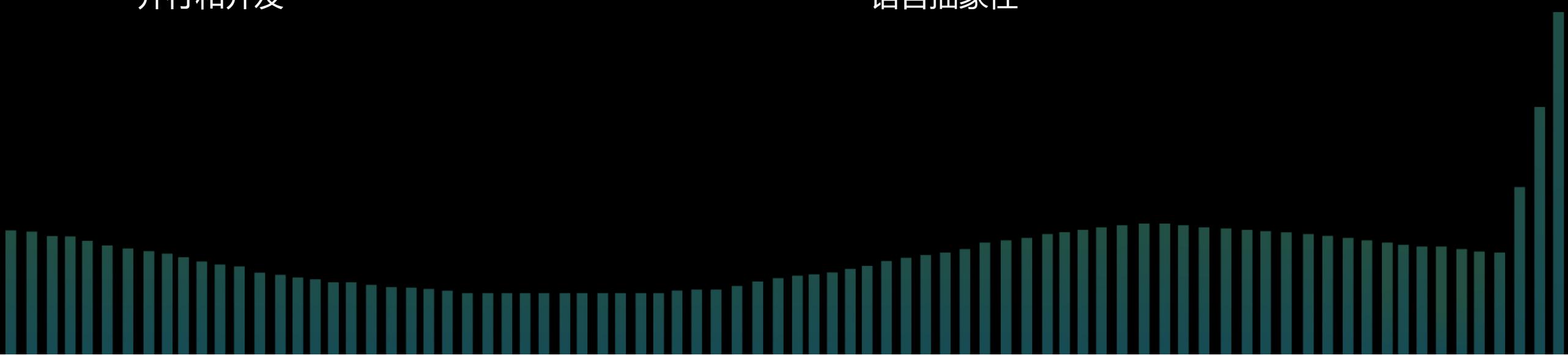
# 影响性能的架构因素

## 硬件

- 存储层次体系
- 处理器的乱序执行和流水线
- 并行和并发

## 软件

- 系统调用开销
- 编译器优化
- 语言抽象性



# 系统调用开销

- `read`
- `write`
- `open`
- `close`
- `mmap`
- `gettimeofday`
- ...

# 编译器优化

后面细说……

# 语言抽象性

C

```
Obj obj;
```

C++

```
Obj obj;
```

- 在栈上分配了 `sizeof(Obj)` 字节， $O(1)$  开销

- 在栈上分配了 `sizeof(Obj)` 字节， $O(1)$  开销
- 调用 `Obj` **构造函数**， $O(?)$
- 到达下面的 } 时调用**析构函数**， $O(?)$

# 为什么要用 C++ ?

# 优化领域的阿姆达尔定律

$$S = \frac{1}{1 - P + \frac{P}{S_P}}$$

性能需要测试！

测不准问题……

# 你知道结果吗？

```
char buffer[80];
auto t1 = clock();
for (auto i = 0; i < LOOPS; ++i) {
    memset(buffer, 0, sizeof buffer);
}
auto t2 = clock();
printf("%g\n",
       (t2 - t1) * 1.0 / CLOCKS_PER_SEC);
```

```
char buffer[80];
auto t1 = clock();
for (auto i = 0; i < LOOPS; ++i) {
    for (size_t j = 0; j < sizeof buffer;
         ++j) {
        buffer[j] = 0;
    }
}
auto t2 = clock();
printf("%g\n",
       (t2 - t1) * 1.0 / CLOCKS_PER_SEC);
```

# GCC 8 的测试结果

编译选项	memset:手工循环 ( 时间 )
-00	1:55
-01	1:5
-02	100000:1



原因：对 buffer 的写入被优化没了！

# volatile ?

# GCC 8 的测试结果 ( volatile )

编译选项	memset:手工循环 ( 时间 )
-00	1:25
-01	1:5
-02	1:5

# volatile 本身会妨碍优化……

```
volatile char buffer[80];  
...  
for (size_t j = 0; j < sizeof buffer; ++j) {  
    buffer[j] = 0;  
}
```

; GCC 10 下可能产生的汇编 (x86-64)

```
    xor    eax, eax  
.L2:  
    mov    BYTE PTR buffer[rax], 0  
    add    rax, 1  
    cmp    rax, 80  
    jne    .L2  
    ret
```

```
char buffer[80];  
...  
for (size_t j = 0; j < sizeof buffer; ++j) {  
    buffer[j] = 0;  
}
```

; GCC 10 下可能产生的汇编 (x86-64)

```
    pxor   xmm0, xmm0  
    movaps XMMWORD PTR buffer[rip], xmm0  
    movaps XMMWORD PTR buffer[rip+16], xmm0  
    movaps XMMWORD PTR buffer[rip+32], xmm0  
    movaps XMMWORD PTR buffer[rip+48], xmm0  
    movaps XMMWORD PTR buffer[rip+64], xmm0  
    ret
```

<https://godbolt.org/z/vnbEaT>

# 编译器的优化魔法

在没有同步原语的情况下，编译器可以（通常为了性能）在（当前线程）结果不变的情况下自由地调整执行顺序

- 同步原语包括互斥锁操作、内存屏障、原子操作等
- 例子：`x = a; y = 2;` 可以变为 `y = 2; x = a;` <https://godbolt.org/z/rbq1q9>
- 例子：`x = y + 1; y = x + 2;` 可以变为 `t = y; y += 3; x = t + 1;` <https://godbolt.org/z/GEqxEs>
- 局部变量可能被完全消除
- 全局变量只保证在下一个同步点到来之前写回到内存里
- `volatile` 声明会禁止编译器进行相关的优化

The image shows a debugger interface with two main panes. The left pane displays the C++ source code:

```
1 int x;
2 int y;
3 int a;
4
5 int main()
6 {
7     x = a;
8     y = 2;
9 }
```

The lines from 7 to 9 are highlighted with different colors: line 7 is light green, line 8 is yellow, and line 9 is light gray. The right pane shows the generated assembly code for an x86-64 architecture using gcc 4.4.7 with optimization level -O2:

```
1 main:
2     mov    eax, DWORD PTR a[rip]
3     mov    DWORD PTR y[rip], 2
4     mov    DWORD PTR x[rip], eax
5     xor    eax, eax
6     ret
7 x:
8     .zero 4
9 y:
10    .zero 4
11 a:
12    .zero 4
```

The screenshot shows a debugger interface with two panes. The left pane displays the C++ source code, and the right pane shows the generated assembly code.

**Left Pane (C++ Source Code):**

```
1 int x;
2 int y;
3
4 int main()
5 {
6     x = y + 1;
7     y = x + 2;
8 }
```

**Right Pane (Assembly Output):**

```
1 main:
2     mov    eax, DWORD PTR y[rip]
3     lea    edx, [rax+1]
4     add    eax, 3
5     mov    DWORD PTR y[rip], eax
6     xor    eax, eax
7     mov    DWORD PTR x[rip], edx
8     ret
9
10    .zero   4
11
12    .zero   4
```

The assembly code is color-coded by section: `main:` (light green), `x:` (yellow), and `y:` (light purple). The `main:` section contains instructions to move `y` to `eax`, calculate `y + 1` in `edx`, add 3 to `eax`, store the result back to `y`, zero `eax`, move `eax` to `x`, and then return. The `x:` and `y:` sections each have a `.zero 4` directive at the end.

# 防优化技巧

- 使用全局变量
- 使用锁来当作简单的内存屏障
- 可使用 `__attribute__((noinline))` 来防止意外内联
- 引用变量来防止其被优化掉

# don't\_optimize\_away

```
// dont_optimize_away.h

void fake_reference(char* ptr);

template <typename T>
inline void dont_optimize_away(T&& datum)
{
    fake_reference(
        reinterpret_cast<char*>(&datum));
}
```

```
// dont_optimize_away.cpp

#include "dont_optimize_away.h"
#include <stdio.h> // putchar
#include <unistd.h> // getpid

static auto pid = getpid();

void fake_reference(char* ptr)
{
    if (pid == 0) {
        putchar(*ptr);
    }
}
```

# GCC 10 的测试结果 ( dont\_optimize\_away )

编译选项	memset:手工循环 ( 时间 )
-O0	1:16
-O1	1:4
-O2	1:1

# 锁和额外函数调用的开销问题……

# Linux 的时钟函数：某平台某环境的某次测试结果

函数	精度 (微秒)	耗时 (时钟周期)
clock	1	~1800
gettimeofday	1	~69
clock_gettime	0.0265(38)	~67
std::chrono::system_clock	0.0274(38)	~68
std::chrono::steady_clock	0.0272(28)	~68
std::chrono::high_resolution_clock	0.0275(20)	~69
rdtsc	0.00965(48)	~24

# Windows 的时钟函数：某平台某环境的某次测试结果

函数	精度 (微秒)	耗时 (时钟周期)
clock	1 (毫秒)	~160
GetTickCount	15.63(49) (毫秒)	~10
GetPerformanceCounter	0.1	~61
GetSystemTimeAsFileTime	15.63(49) (毫秒)	~20
GetSystemTimePreciseAsFileTime	0.1	~100
std::chrono::system_clock	0.1	~100
std::chrono::steady_clock	0.1	~160
std::chrono::high_resolution_clock	0.1	~160
rdtsc	0.00973(93)	~25

# RDTSC 指令 ( Read Time Stamp Counter )

- x86 和 x64 系统上的的首选计时方式
- MSVC 和 GCC 都提供内置函数 `_rdtsc`，开发者不再需要自行用内联汇编实现
- 较新的 CPU 一般提供 `constant_tsc` 功能，CPU 频率变化不会影响 TSC 计时
- 较新的 CPU 一般提供 `nonstop_tsc` 功能，休眠不影响 TSC 计时
- 多核 CPU 一般能同步 TSC，但多 CPU 插槽的系统该指令可能仍有不一致问题
- TSC 频率和 CPU 参考主频不一定一致
  - 自行测量或使用 `dmesg | grep 'tsc.*MHz'`

# 实现一个小性能分析器（ profiler ）

- 在构造时记录时间
- 在析构时再记录时间，并记录这次函数调用的内部消耗时间到全局变量里
- 程序退出时，打印记录的信息
- 单个的数据记录和整体的初始化、打印都依赖于 RAII

# 小性能分析器的使用

- 使用枚举 `profiled_functions` 记录所有需要分析的函数的编号
- 使用全局变量 `g_name_map` 记录函数编号和名称的映射关系
- 在函数内用宏 `PROFILE_CHECK` 记录函数的开销
- 使用循环重复执行待测代码，操作全局变量
- 在非时间记录部分，调用锁操作，确保消除过度优化或乱序执行

# 示例：测量时钟的开销

```
enum profiled_functions { PF_MEASURE_CLOCK_OVERHEAD_SINGLE };

name_mapper g_name_map[] = {
    {PF_MEASURE_CLOCK_OVERHEAD_SINGLE, "measure_clock_overhead_single"},
    {-1, NULL}};

void measure_clock_overhead_single()
{
    PROFILE_CHECK(PF_MEASURE_CLOCK_OVERHEAD_SINGLE);
    now = std::chrono::high_resolution_clock::now();
}

...
for (int i = 0; i < LOOPS; ++i) {
    std::lock_guard guard{mutex};
    measure_clock_overhead_single();
}
```

# 示例：测量时钟的开销

```
high_resolution_clock takes 94 cycles on average
```

```
0 measure_clock_overhead_single:
```

```
Call count: 1000
```

```
Call duration: 98584
```

```
Average duration: 98.584
```

# 示例：测量函数调用和虚函数调用的额外开销

```
class space_checker {
public:
    bool isspace(char ch)
    {
        return std::isspace(ch);
    }
};

class space_checker_noinline {
public:
    bool isspace(char ch);
};

__attribute__((noinline)) bool
space_checker_noinline::isspace(char ch)
{
    return std::isspace(ch);
}
```

```
class space_checker_intf {
public:
    virtual ~space_checker_intf() = default;
    virtual bool isspace(char ch) = 0;
};

class space_checker_virt : public space_checker_intf {
public:
    bool isspace(char ch) override;
};

bool space_checker_virtual::isspace(char ch)
{
    return std::isspace(ch);
}
```

# 示例：测量函数调用和虚函数调用的额外开销

```
0 count_space:  
    Call count: 10000  
    Call duration: 1724452  
    Average duration: 172.445  
1 count_space_noinline:  
    Call count: 10000  
    Call duration: 2902176  
    Average duration: 290.218  
2 count_space_virtual:  
    Call count: 10000  
    Call duration: 4300125  
    Average duration: 430.012
```

每次函数调用的开销：

$$\frac{290 - 172}{47} \approx 2.5$$

每次虚函数调用的开销：

$$\frac{430 - 172}{47} \approx 5.5$$

# 两种性能测试方式

## 插桩测试

- 开销随测试范围而变
- 插桩本身可能影响测试结果
- 测试结果可以较为精确、稳定
- 适合对单个函数进行性能调优
- 需要修改源代码或构建过程

## 采样测试

- 总体开销可控
- 一般不影响程序“热点”
- 基于统计，误差较大
- 适合用来寻找程序的热点
- 可以完全在程序外部进行测试

# gprof 简介

## 基本使用

```
g++ -O1 -g -pg ...
```

```
./可执行程序名
```

```
gprof 可执行程序名 gmon.out > gprof.out
```

```
vim -c 'set nowrap' gprof.out
```

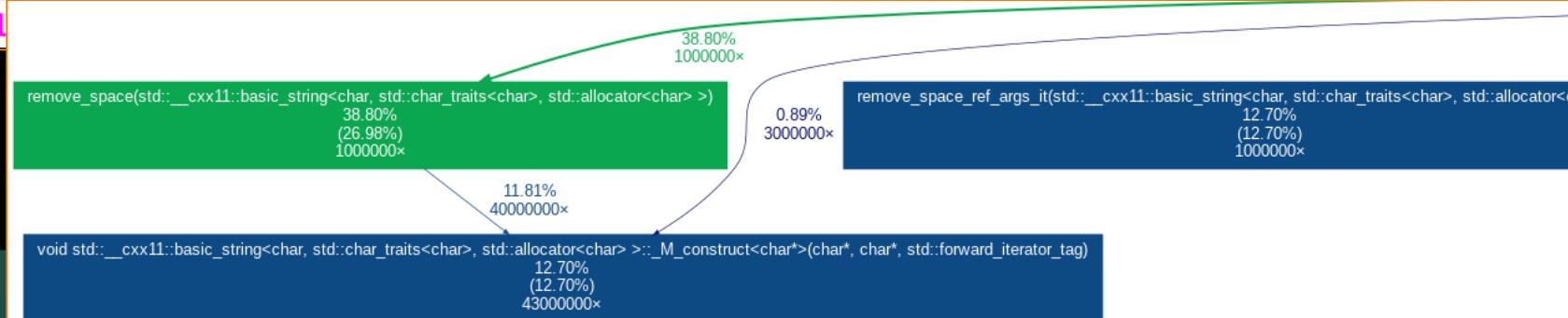
# gprof 的输出

## Flat profile:

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ns/call	total ns/call	name
27.03	0.34	0.34	1000000	340.64	489.76	remove_space(s)
12.72	0.50	0.16	43000000	3.73	3.73	void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::remove_space(std::forward<iterator_tag> it)
12.72	0.66	0.16	1000000	160.30	160.30	remove_space(args_it)
11.93	0.81	0.15	1000000	150.28	150.28	remove_space(args_it)
9.54	0.93	0.12	1000000	120.23	120.23	remove_space(args_it)
8.75	1.04	0.11	1000000	110.21	110.21	remove_space(args_it)
7.95	1.14	0.10	1000000	100.19	100.19	remove_space(args_it)
5.57	1.21	0.07	1000000	70.13	70.13	remove_space(args_it)
1.59	1.23	0.02	1000000	20.04	20.04	remove_space(args_it)
1.59	1.25	0.02				main
0.80	1.26	0.01				nvwa::fast_mutate
0.00	1.26	0.00	1	0.00	0.00	GLOBAL_sub
0.00	1	0				

index	% time	self	children	called	name
[1]	99.2	0.02	1.23		<spontaneous>
		0.34	0.15	1000000/1000000	main [1]
		0.16	0.00	1000000/1000000	remove_space
		0.15	0.00	1000000/1000000	remove_space
		0.12	0.00	1000000/1000000	remove_space
		0.11	0.00	1000000/1000000	remove_space
		0.10	0.00	1000000/1000000	remove_space
		0.07	0.00	1000000/1000000	remove_space
		0.02	0.00	1000000/1000000	remove_space
		0.01	0.00	3000000/43000000	void std::
[2]	38.8	0.34	0.15	1000000/1000000	main [1]
		0.15	0.00	40000000/43000000	remove_space(std::string)
[3]	12.7	0.16	0.00	43000000	void std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char>>::remove_space(std::forward<iterator_tag> it)
[4]	12.7	0.16	0.00	1000000	remove_space_ref
[5]	11.9	0.15	0.00	1000000	main [1]
[6]	9.5	0.12	0.00	1000000	remove_space_ref
[7]	8.7	0.11	0.00	1000000	main [1]
		0.11	0.00	1000000/1000000	remove_space_all
		0.11	0.00	1000000	main [1]
		0.11	0.00	1000000	remove_space_reserve



# gperftools 简介

## 安装 ( Ubuntu )

```
sudo apt install google-perf-tools
```

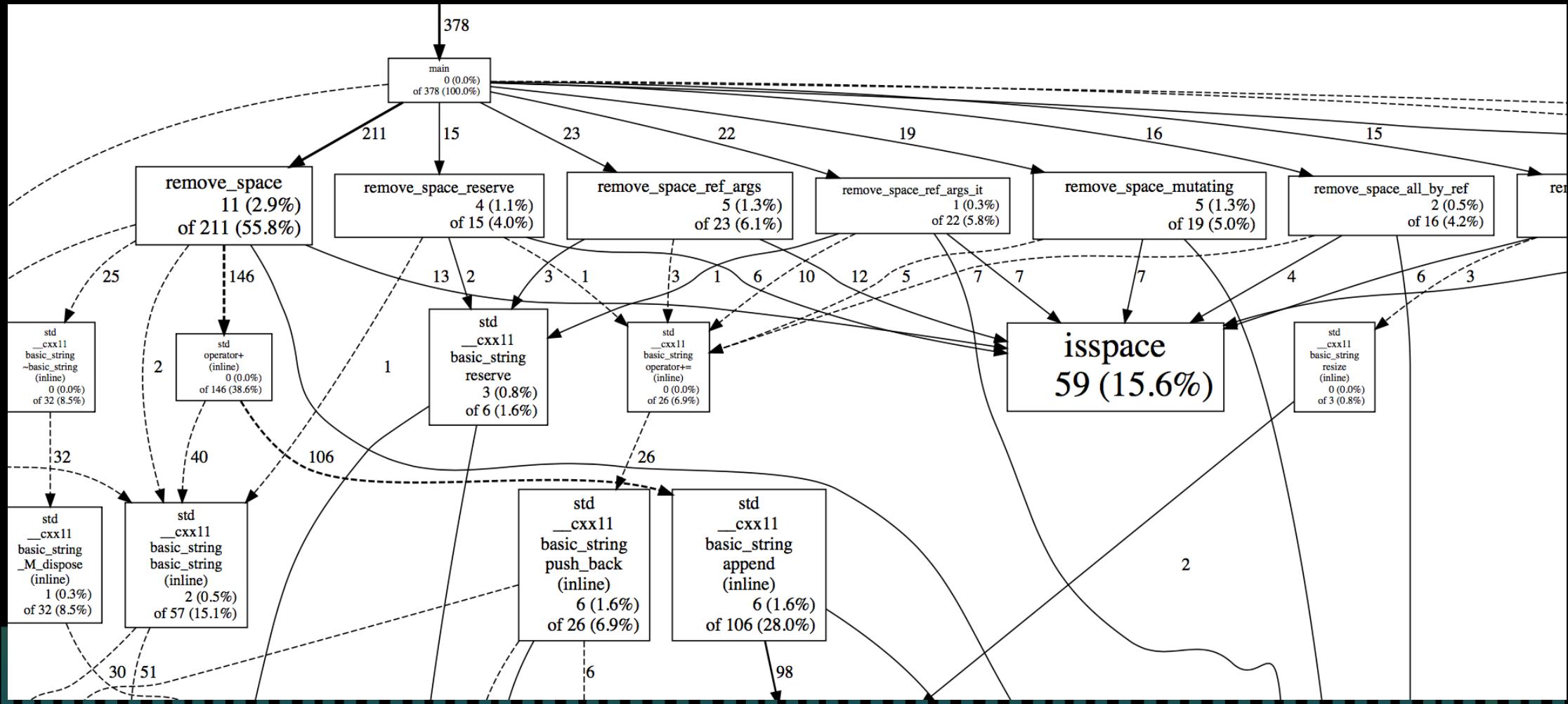
## 基本使用

```
LD_PRELOAD=/usr/lib/x86_64-linux-gnu/libprofiler.so.0 \
```

```
CPUPROFILE=test.prof ./可执行程序名
```

```
google-pprof --svg 可执行程序名 test.prof > test.svg
```

# gperftools 的 SVG 输出



# 编译器选项 – 汇编代码生成

## GCC

- -S
- -masm=intel ( 可选 )
- 使用 c++filt 转换函数名
  - Vim 里 :%!c++filt

## MSVC

- /Fa

# 浏览器里的汇编代码生成

**Compiler Explorer: <https://godbolt.org/>**

- 多种编译器 ( GCC , Clang , MSVC , 等等 )
- 多种平台 ( x86-64 , ARM , 等等 )
- 可自行设定编译 ( 优化 ) 选项
- 友好的过滤设置，一眼看到关注的内容
- 可在线运行，在浏览器里看到文本输出
- 允许使用常用 C++ 库 ( Boost , Catch2 , fmt , GSL , range-v3 , spdlog , 等等 )
- .....

The screenshot shows a debugger interface with two panes. The left pane displays the C++ source code:

```
1 int x;
2 int y;
3
4 int main()
{
    x = y + 1;
    y = x + 2;
}
```

The lines `x = y + 1;` and `y = x + 2;` are highlighted with different colors (light blue and light yellow). The right pane shows the generated assembly code for the `main` function:1 main:
2 mov eax, DWORD PTR y[rip]
3 lea edx, [rax+1]
4 add eax, 3
5 mov DWORD PTR y[rip], eax
6 xor eax, eax
7 mov DWORD PTR x[rip], edx
8 ret
9 x:
10 .zero 4
11 y:
12 .zero 4

The assembly code is color-coded to match the source code highlighting. A green checkmark icon is visible in the top right corner of the assembly pane.

未过滤的 GCC 汇编输出超过 30 行

# Compiler Explorer 示例

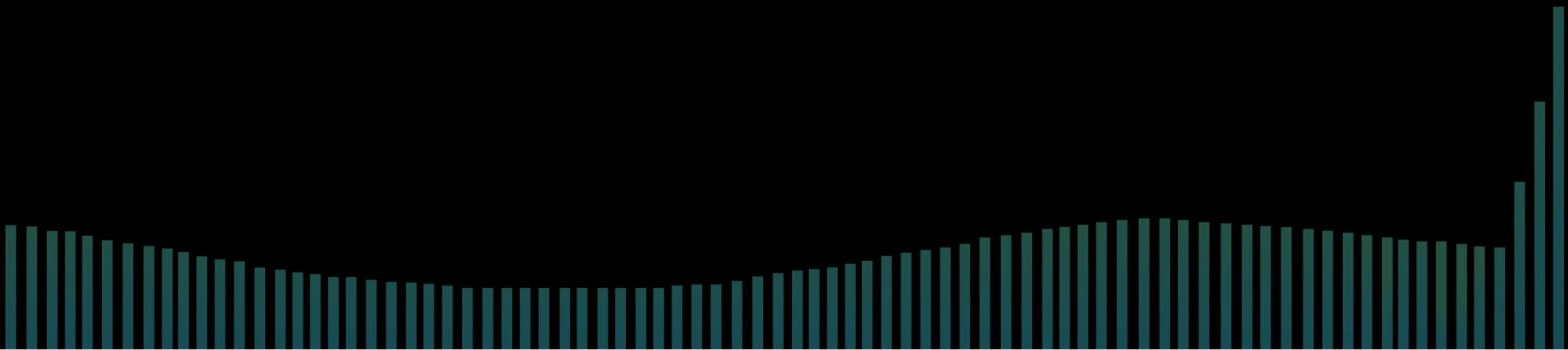
- C 的 Hello world : <https://godbolt.org/z/vns4n8>
- C++ 的 Hello world : <https://godbolt.org/z/bP9KTa>
- 简单循环和 strlen : <https://godbolt.org/z/5sb8bj>
- Ranges 和编译期优化 : <https://godbolt.org/z/foj6bq>

# 编译器选项 – 优化级别

- `-O0` (默认) : 不开启优化，方便功能调试
- `-Og` : 方便调试的优化选项 (比 `-O1` 更保守)
- `-O1` : 保守的优化选项
  - 当前 GCC 上打开了 45 个优化选项
- `-Os` : 产生小代码的优化选项 (比 `-O2` 更保守，并往产生较小代码的方向优化)
- `-O2` : 常用的发布优化选项 (对错误编码容忍度低)
  - 当前 GCC 上比 `-O1` 额外打开 48 个优化选项
  - 包括自动内联函数和严格别名规则
- `-O3` : 较为激进的优化选项 (对错误编码容忍度最低)
  - 当前 GCC 上比 `-O2` 额外打开 16 个优化选项
- `-Ofast` : 打开可导致不符合 IEEE 浮点数等标准的性能优化选项

std::sort 比 qsort 更快 !

快多少？

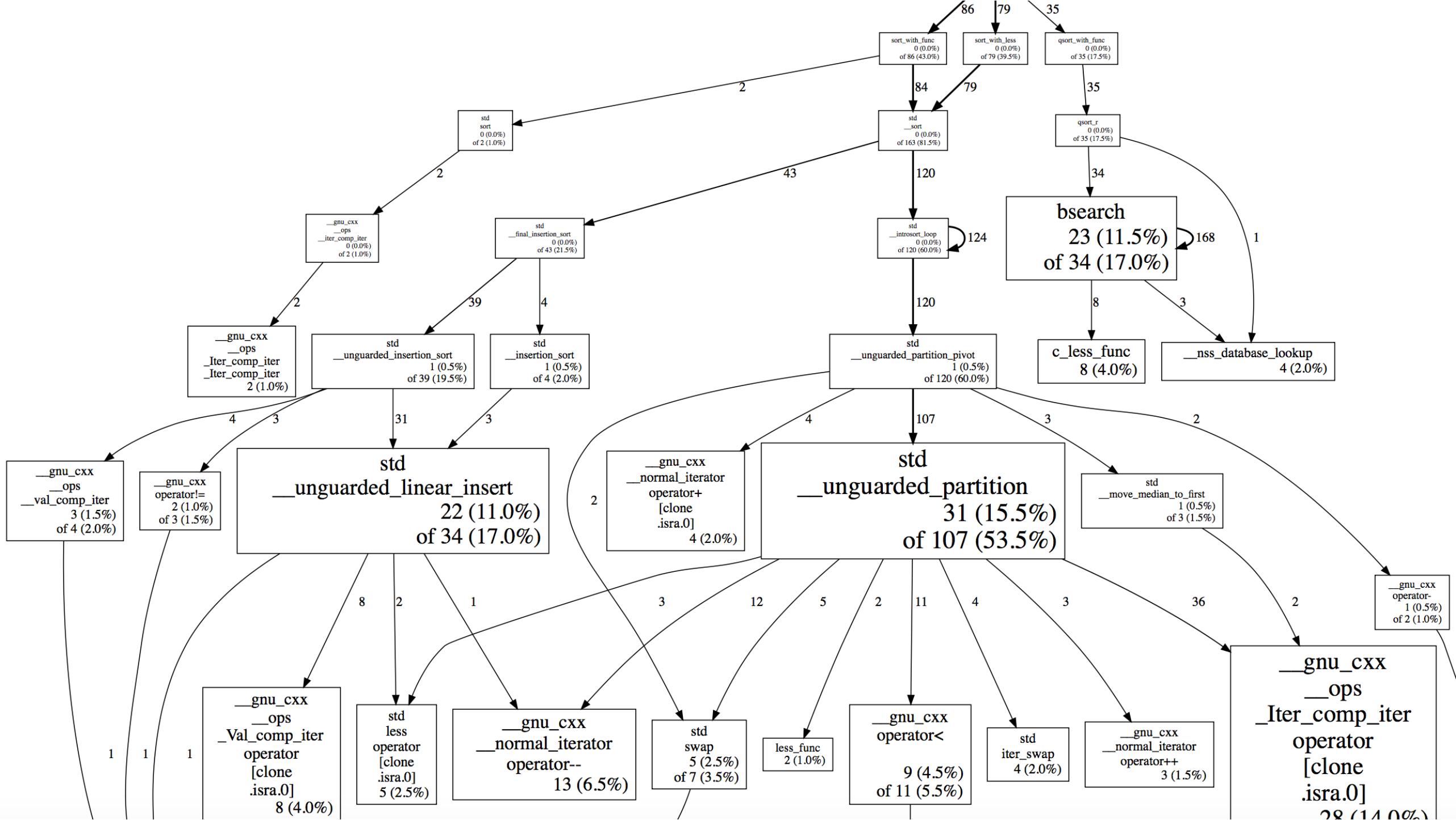


It depends. . . .

# 优化和内联的影响

	-O0	-O2 -fno-inline	-O2	-O2 对 -O0 的提升
sort_with_less	340981(1717)	197830(675)	24414(333)	14x
sort_with_func	334601(1843)	209241(609)	46384(220)	7.2x
qsort_with_func	133801(1814)	87014(790)	85323(535)	1.6x

单单内联即可产生一个数量级的性能差异！



# 循环优化（热点优化）

- 循环会放大代码中的低效率
- 把不必要的反复执行的代码提到循环外面

```
for (size_t i = 0; i < strlen(s); ++i) {  
    ...  
}
```

# 优化 I – 长度不变的情况

```
int sp_count = 0;
for (size_t i = 0,
         len = strlen(s);
      i < len; ++i) {
    if (isspace(s[i])) {
        ++sp_count;
    }
}
```

```
int sp_count = 0;
// 如果 s 是字符数组或 string
for (char ch : s) {
    if (isspace(ch)) {
        ++sp_count;
    }
}
```

## 优化 II – 长度可变的情况

```
for (size_t i = 0, len = strlen(s); i < len; ++i) {
    if (isspace(s[i])) {
        memmove(&s[i], &s[i + 1], len - i);
        len--;
    }
}
```

# 如果使用 std::string 的话

```
for (auto it = s.begin(); it != s.end(); ++it) {  
    if (isspace(*it)) {  
        s.erase(it);  
    }  
}
```

# 注意隐藏开销

- 构造
- 析构
- 值参数的拷贝构造
- .....

```
std::string remove_space(std::string s)
{
    ...
}
```

# 对象参数的一般原则

- 如果不修改对象的话，使用 `const Obj&`
- 如果需要修改对象的话，使用 `Obj&`
- 如果需要在对象的新拷贝上进行操作的话，使用 `Obj`
  - 可以利用对象移动的可能性

```
// 如果返回新对象的话  
string remove_space(const string& s);
```

```
// 如果修改既有对象的话  
void remove_space(std::string& s);
```

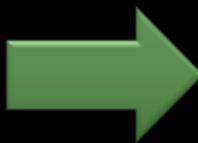
```
// 从传入的对象生成结果  
string greet(string name)  
{  
    name += ", nice to meet you!";  
    return name;  
}
```

# string 接口使用建议

- 不推荐使用 `const string&` (除非调用方确保有现成的 `string` 对象)
- 如果不需要修改字符串内容，使用 `string_view` 或 `const char*`
- 如果仅在函数内部修改字符串的内容，使用 `string`
- 如果需要修改调用者的字符串内容，使用 `string&`

# 消除不必要的反复构造和析构

```
for (...) {  
    std::string s;  
    s += "Hi, ";  
    ...  
    s += "Best regards,";  
    process(s);  
}
```



```
std::string s;  
for (...) {  
    s.clear();  
    s += "Hi, ";  
    ...  
    s += "Best regards,";  
    process(s);  
}
```

大幅减少了对内存管理器的调用次数

# 多线程优化

- 多线程加锁和竞争是性能杀手
- 能使用 `atomic` 就不要使用 `mutex`
- 如果读比写多很多，使用读写锁（`shared_mutex`）而不是独占锁（`mutex`）
- 使用线程本地（`thread_local`）变量

# volatile vs atomic

```
std::atomic<int> a;

void increment_atomic(
    std::atomic<int>& n)
{
    for (int i = 0; i < LOOPS; ++i) {
        ++n;
    }
}
```

```
volatile int v;

void increment_volatile(
    volatile int& n)
{
    for (int i = 0; i < LOOPS; ++i) {
        ++n;
    }
}
```

```
*** Incrementing on the same atomic int
Result is 200000
*** Incrementing on the same volatile int
Result is 101893
```

# 简单增一、原子增一和加锁增一

	单线程 (时钟周期数)	双线程 (时钟周期数)
increment_volatile	~7	~7
increment_atomic	~19	~116
increment_with_lock	~72	~18000

# thread\_local 成员变量

```
class Obj {  
    ...  
public:  
    static Obj& get_instance();  
private:  
    static thread_local Obj inst_;  
};  
  
thread_local Obj Obj::inst_;  
  
Obj& Obj::get_instance()  
{  
    return inst_;  
}
```

# thread\_local 变量

```
class Obj {  
    ...  
public:  
    static Obj& get_instance();  
};
```

```
Obj& Obj::get_instance()  
{  
    thread_local Obj inst;  
    return inst;  
}
```

# 算数表达式优化

$$x \cdot a \cdot b \cdot c = x \cdot (a \cdot b \cdot c)?$$

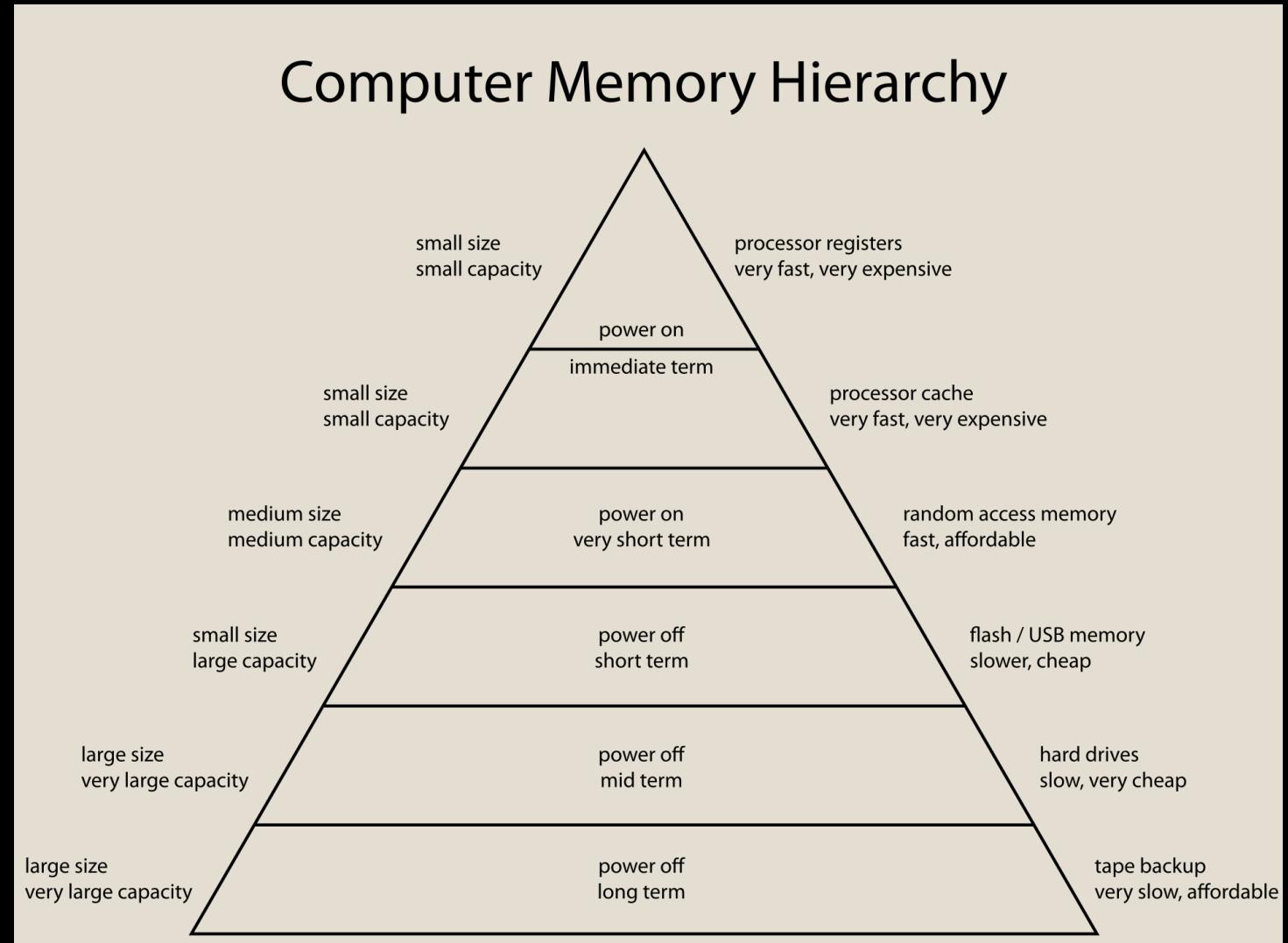
<https://godbolt.org/z/3Y15Es>

# 浮点数表达式优化

- 把常量归并到一起
  - $a \cdot x \cdot b \Rightarrow (a \cdot b) \cdot x$
- 手工简化表达式
  - $a \cdot x^2 + b \cdot x + c \Rightarrow (a \cdot x + b) \cdot x + c$
- 除法优化
  - $x / a \Rightarrow x \cdot (1 / a)$
- 注意在有硬件浮点运算单元的机器上可能 double 比 float 要快
- 关键代码上使用性能测试进行检查

# 缓存和性能

- 内存分级和访问性能
- 相邻数据访问较快
- 占用内存少 == 快
- 缓存竞争问题



# 矩阵转置中的缓存竞争

矩阵大小	内存占用 ( KB )	每元素耗时 ( 时钟周期 )	优化后耗时 ( 时钟周期 )
63x63	31	1.2	—
64x64	32	1.2	—
65x65	33	1.2	—
127x127	126	1.3	—
128x128	128	4.8	1.4
129x129	130	1.3	—
255x255	508	1.7	—
256x256	512	11.6	1.9
257x257	516	1.8	—
511x511	2040	2.5	—
512x512	2048	16.3	3.6
513x513	2056	3.9	—

# 不需要的优化

- 把  $x * 8$  写成  $x \ll 3$ ；或把  $x * 5$  写成  $(x \ll 2) + x$ 
  - 变乘为移位和加法是 GCC 在 **-O0** 时就会做的！
- 提取公共表达式
  - 编译器会自动做
- 略去本地变量的初始化
  - 编译器经常会自动做；安全编码检查要求初始化

# x \* 8 的非优化编译结果

```
int calculate(int x)
{
    return x * 8;
}
```

```
calculate(int):
    push    rbp
    mov     rbp, rsp
    mov     DWORD PTR [rbp-4], edi
    mov     eax, DWORD PTR [rbp-4]
    sal     eax, 3
    pop     rbp
    ret
```

# x \* 5 的优化编译结果

```
int calculate(int x)
{
    return x * 5;
}
```

```
calculate(int):
    lea    eax, [rdi+rdi*4]
    ret
```

# 公共表达式提取示例

```
int calculate(int x, int y, int z)
{
    int a = (x * x + y * y) + 3;
    int b = (x * x + y * y) / z;
    return a + b;
}
```

```
calculate(int, int, int):
    imul    edi, edi
    mov     r8d, edx
    imul    esi, esi
    add     esi, edi
    mov     eax, esi
    cdq
    idiv    r8d
    lea     eax, [rsi+3+rax]
    ret
```

# 消除无用的初始化 I

```
int get1();
int get2();

int test()
{
    int retcode = 0;
    retcode = get1();
    if (retcode != 0) return retcode;
    retcode = get2();
    if (retcode != 0) return retcode;
    return retcode;
}
```

```
test():
    sub    rsp, 8
    call   get1()
    test   eax, eax
    je    .L5
    add    rsp, 8
    ret

.L5:
    add    rsp, 8
    jmp   get2()
```

# 消除无用的初始化 II

```
int g = 10;
void get(int& x)
{
    x = g + 1;
}
int test()
{
    int value = 0;
    get(value);
    return value + 32;
}
```

```
get(int&):
    mov    eax, DWORD PTR g[rip]
    add    eax, 1
    mov    DWORD PTR [rdi], eax
    ret

test():
    mov    eax, DWORD PTR g[rip]
    add    eax, 33
    ret

g:
    .long 10
```

# 无法消除初始化的情况

```
void get(int& x);  
  
int test()  
{  
    int value = 0;  
    get(value);  
    return value + 32;  
}
```

```
test():  
    sub    rsp, 24  
    lea    rdi, [rsp+12]  
    mov    DWORD PTR [rsp+12], 0  
    call   get(int&)  
    mov    eax, DWORD PTR [rsp+12]  
    add    rsp, 24  
    add    eax, 32  
    ret
```

C++ 在语言上不区分出参和入参的结果

# 其他性能调优手段

- C++ : 控制流优化，输入输出优化，内存优化，算法优化，.....
- 工具：[Intel VTune Profiler](#) , [perf](#) , .....
- 编译器：[Intel C++ Compiler](#) , [Clang](#) , .....
- 并行和并发编程：多线程，并行执行策略，[OpenMP](#) , .....
- 异构计算：[CUDA](#) , [OpenCL](#) , [SYCL](#) , [Data Parallel C++ / oneAPI](#) , .....
- 开源第三方库

# 参考资料

- Agner Fog, "Software optimization resources".  
<https://www.agner.org/optimize/>.
- Kurt Guntheroth, *Optimized C++*. O'Reilly, 2016.

需要鉴别资料中的过时信息和错误

# 代码

[https://github.com/adah1972/cpp\\_summit\\_2020](https://github.com/adah1972/cpp_summit_2020)

# 赵永刚

Boolan 资深咨询师



16年C/C++系统级软件开发经验，前诺基亚贝尔资深架构师，内部技术教练，是系统软件架构、代码安全、性能优化方面的专家。开发经验涉及通信、嵌入式Linux开发、自动驾驶系统软件产品开发的多个领域。14 年嵌入式系统开发经验，从上层应用开发到底层 硬件适配层开发均有丰富的经验。对系统级软件架构设计、重构与守护、设计模式、开发者测试、性能优化、安全编码等有深入研究和丰富经验。

主办方：

**Boolan**  
高端 IT 咨询与教育平台

C++ Summit 2020

赵永刚

Boolan 咨询师

使用  
代码检查  
提升软件质量

# 议程

- 软件质量
- 代码检查
- 代码检查的开源解决方案

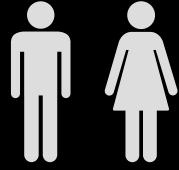
- | • 软件质量
- 代码审查
- 代码检查的开源解决方案



“软件质量是软件与明确地叙述的功能和性能需求、文档中明确描述的开发标准以及任何专业开发的软件产品都应该具有的隐含特征相一致的程度。

来自《百度百科》

# 关心软件质量的人

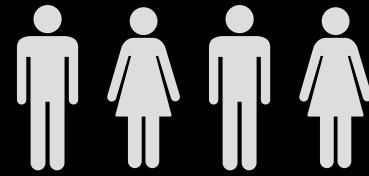


业务经理/老板



开发过程

交付的软件

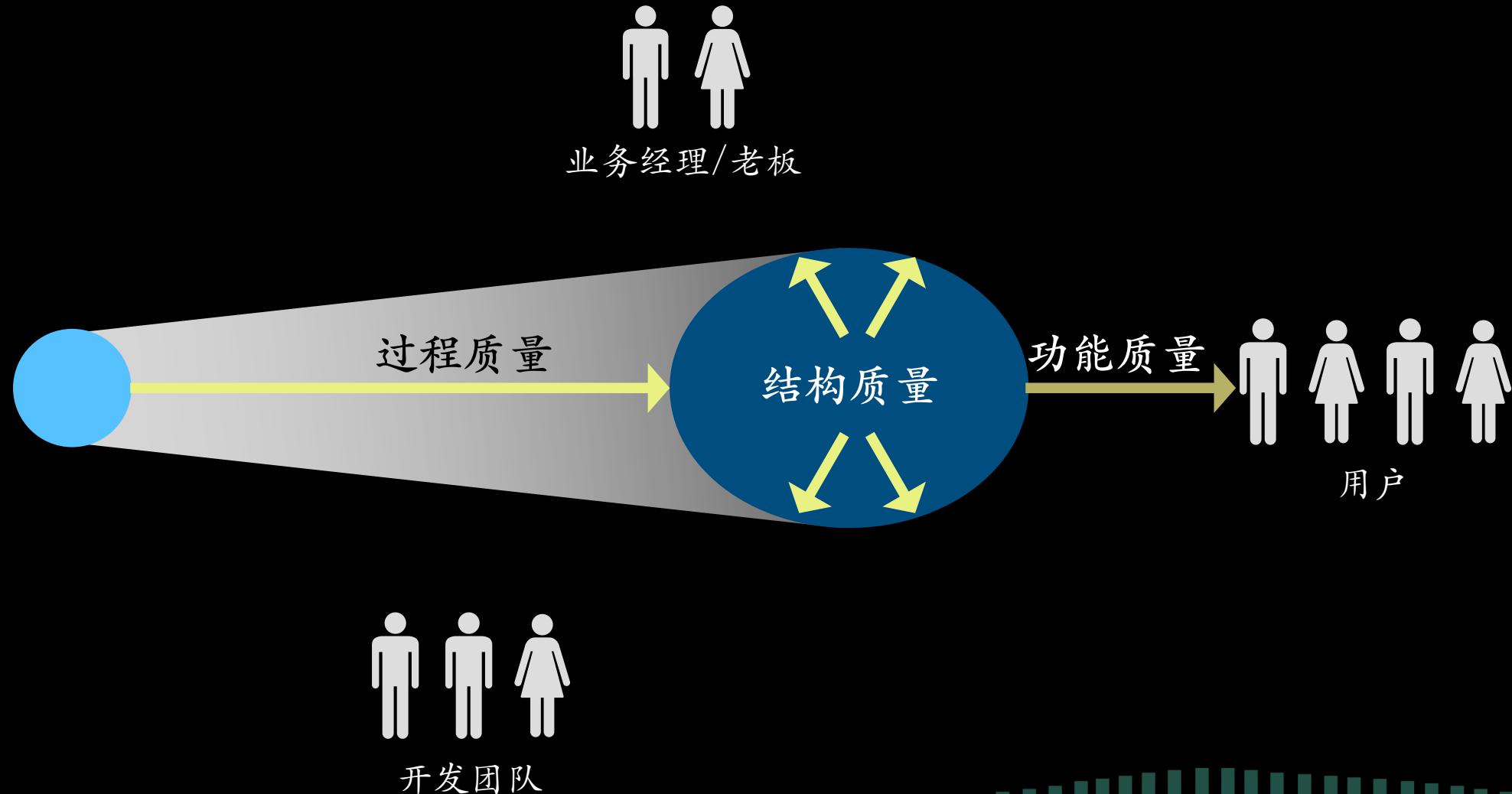


用户



开发团队

# | 软件质量的三个方面

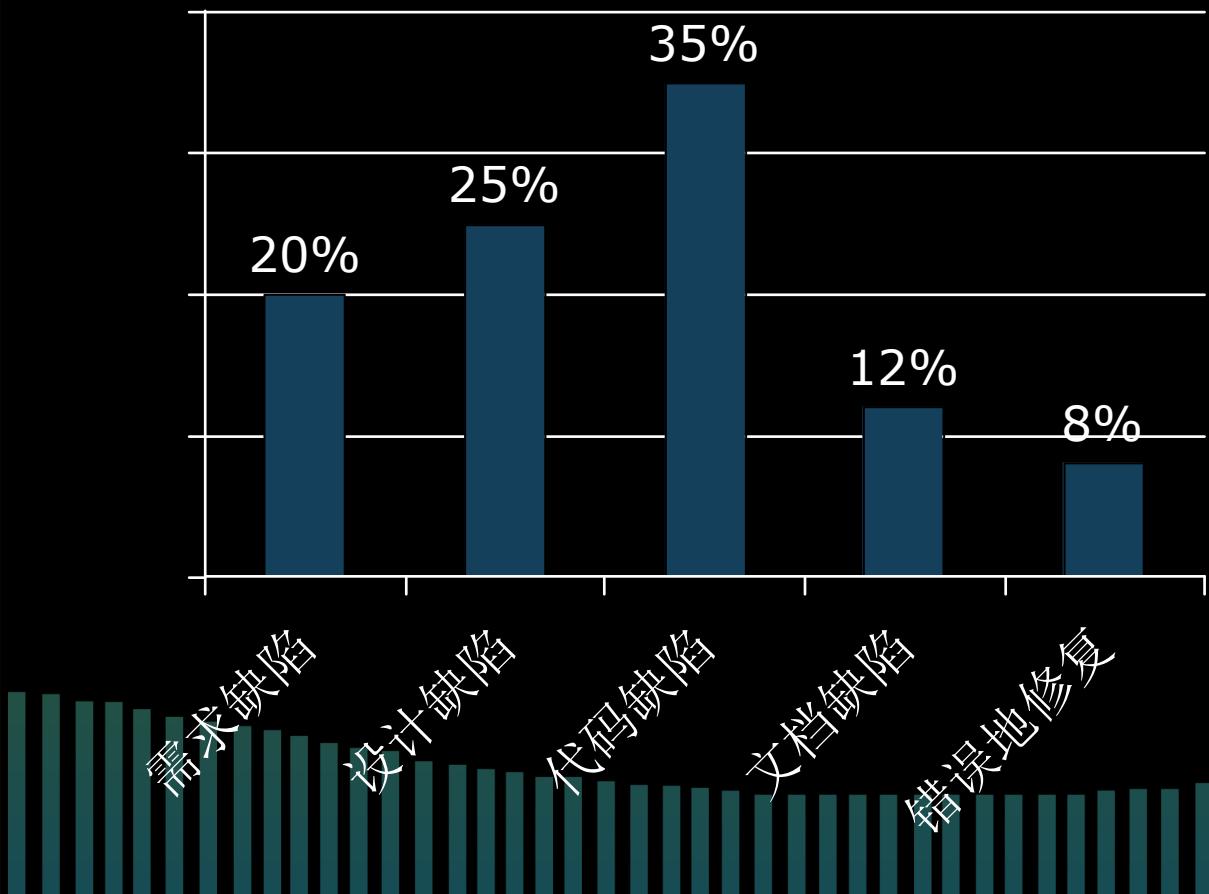


# 三个质量

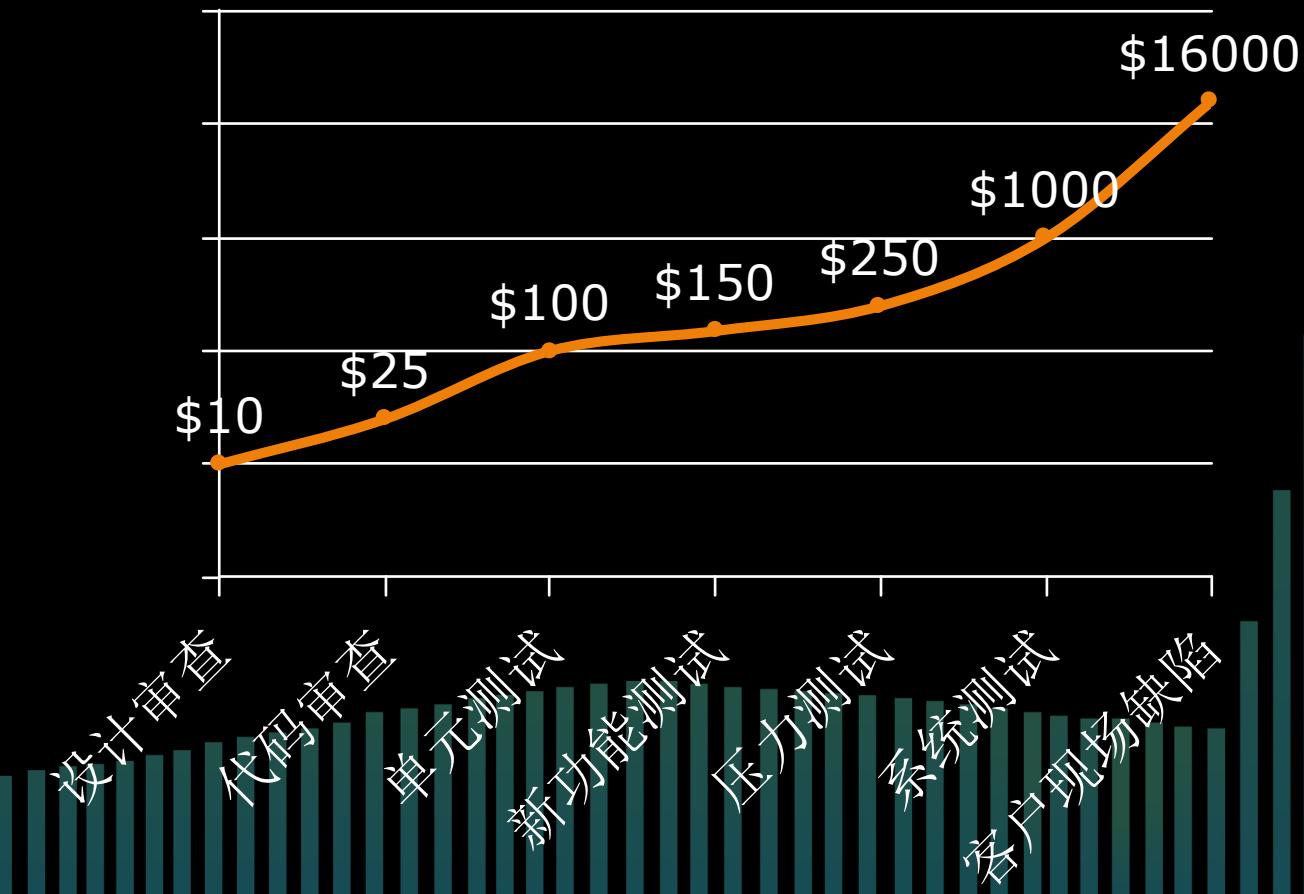


# | 缺陷的种类和不同阶段缺陷的修复成本

不同缺陷的分布



解决缺陷的成本

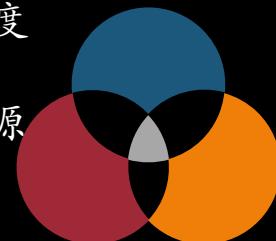


# 结构质量的最佳实践

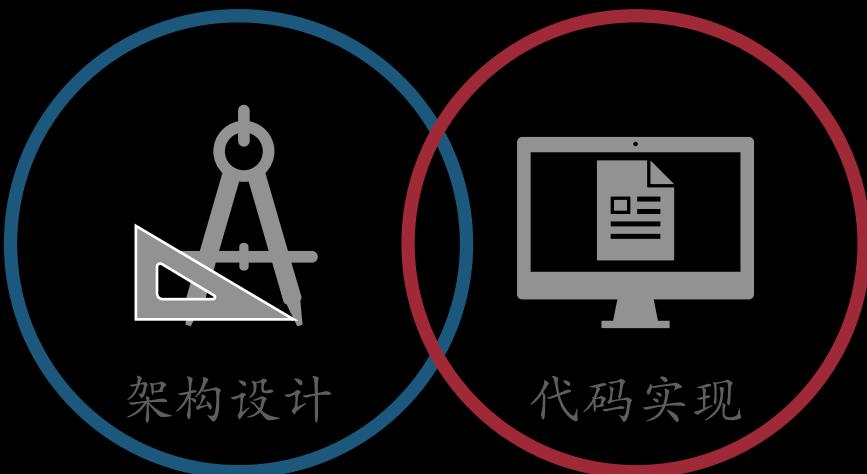
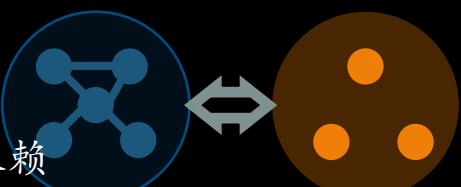
- 合理的模块层级



- 综合考虑，约束
  - 模块大小和复杂度
  - 接口大小
  - 模块之间共享资源



- 高内聚低耦合
  - 消除重复
  - 分离变化方向
  - 缩小依赖范围
  - 向稳定的方向依赖



- 统一风格
  - 代码样式统一
  - 命名风格统一
  - 词汇统一



- 整洁代码
  - 代码可读性高
  - 没有坏味道
- 使用惯用法
- 熟悉语言和标准库



- 可信编码
  - 内存安全
  - 资源安全
  - 线程安全
- 避免未定义行为



- 防止缓存溢出
- 防止堆栈溢出
- 整数安全
- 避免代码漏洞

# 提升软件质量的“捷径”

- 结构质量的提升
  - 设计审查
  - 代码审查
- 相对低成本的解决缺陷，带来功能质量的提升
  - 设计审查
  - 代码审查



- 软件质量
- | • 代码审查
- 代码检查的开源解决方案



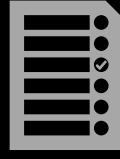
# 代码审查的几种形式



- 结对编程  
知识分享  
知识传播  
防止错误  
人力成本较高



- 正式的代码审查会议  
为试图寻找代码的缺陷提供了一种非常结构化的流程，并且，它还可以用于发现需求缺陷和设计缺陷



- 代码走查  
通常由经验丰富的工程师来对检入的代码进行检查，人力成本相对较低



- 静态代码分析
  - 基于文本和模式匹配
    - 检查对代码规范的遵守
  - 基于源代码抽象语法树的分析
    - 数据类型问题、未初始化问题
    - 进行控制流和数据流分析



- 动态代码分析
  - 根据需求构建测试用例
  - 使用覆盖率检查工具检查用例对代码的覆盖率
  - 使用动态代码检查工具检查代码问题

# 代码审查要检查的问题

- 代码规范检查
  - 代码风格统一
  - 不符合规范的代码
- 代码度量
  - 代码的复杂度
  - 嵌套深度
  - 扇入扇出度量
  - 继承树的深度
  - 类的加权方法
  - 对象间的耦合度
  - .....
- 代码缺陷检查
  - 空指针
  - 内存溢出
  - 类型转换错误
  - double free
  - 返回局部变量的引用
  - 未定义行为
  - 差一错误
  - 死锁
  - 竞争
- 代码坏味道检查
  - 过长函数
  - 过长参数列表
- 过大的类
- 哑数据类
- switch表达式惊悚现身
- 平行的继承体系
- 重复代码
- 霹雳式修改
- 代码与需求背离
- 代码不符合设计

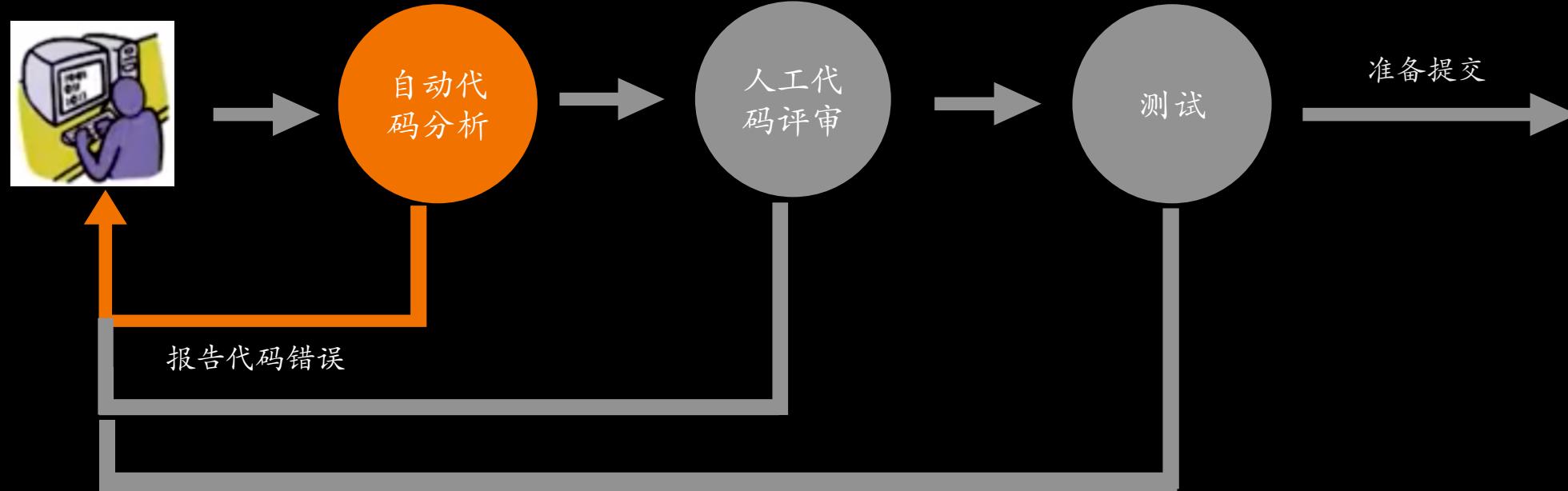
# 代码审查形式的比较

	代码走查	结对编程	正式的代码审查会议	静态代码检查	动态代码检查
代码规范检查	中低	中	中高	高	无
代码度量	低	低	低	高	无
代码缺陷	中	中高	中高	高	高
代码坏味道	中	高	高	高	无
代码与需求的背离	中	高	高	无	无
代码不符合设计	中	高	高	无	无
需求缺陷	中	中高	高	无	无
设计缺陷	中	中高	高	无	无
所需人力	低	中	高	低	低

# 使用工具检查代码

- 编译器诊断
  - 在高警告级别下干净地编译代码
- Linter风格检查工具
  - 使用同一的格式化工具对项目代码进行格式化
  - 定义团队的代码规范，使用检查工具进行检查
  - 定义团队统一的代码度量阈值，需要解决的坏味道，使用工具进行检查
- 静态代码和动态代码分析工具
  - 结合静态分析工具和动态分析工具，有效利用各种工具的长处，更早地发现代码的问题

# 自动代码分析在CI的典型应用



静态分析、语法和语义检查：

- 找出代码中没有被测试用例覆盖的错误！
- 可以分析无法测试的代码的属性（编码风格）
- 自动检查代码部分
  - 可以在CI快速反馈代码问题
  - 可以使用IDE插件帮助程序员快速反馈

- 代码质量
- 代码审查
  - 1. 代码度量分析
  - 2. 静态代码检查
  - 3. 动态代码检查
- 代码检查的开源解决方案

# 代码度量指标的作用

- 函数的圈复杂度过大，说明：
  1. 可能函数本身实现的过于复杂，或
  2. 可能因为架构设计过于复杂，导致函数过于复杂
- 函数嵌套过深，说明
  1. 函数很可能出错，需要仔细进行人工评审，并且
  2. 函数可以重构，
    - (1) 使用卫戍句优化逻辑，或
    - (2) 优化条件逻辑，或
    - (3) 提取函数，或
    - (4) 架构设计出现坏味道，需要重构架构
- 注释比例过低（注释和语句比例或注释和圈复杂度的比例）
- 模块的扇入过大
  - 模块是公共模块，要人工评审接口是否是稳定的，或
  - 模块承担过多职责，考虑遵循单一职责，分解模块职责
- 模块的扇出过大
  - 检查是否有多个模块都依赖于本模块依赖的几个模块，考虑把被依赖的多个模块合并为一个模块，重构依赖的接口
- 类的继承树过深
  - 考虑在继承树的深度上是否有新的变化方向，考虑提出新的策略类，或其他设计模式来优化继承树
- 子类过多
  - 检查子类的实现中共同的地方，先考虑提出公共的中间子类
  - 检查是否可以通过Bridge模式、装饰模式、组合模式等结构型模式重构代码

可能是有些复杂的逻辑需要添加注释说明，否则会影响代码的可理解性

- 模块的扇入过大

# 圈复杂度过高的例子

```

void Main::HandleArgs(int argc, char **argv)
{
    bool accepting_options = true;

    for (int i = 1; i < argc; i++)
    {
        string next_arg = argv[i];
        if (
            (accepting_options == false) ||
            (next_arg.substr(0, 2) != "--"))
        {
            // normally this will be a single file name, but
            // the function below also encapsulates handling of
            // globbing (only required under Win32) and
            // the conventional interpretation of '-' to mean
            // read a list of files from standard input
            AddFileArgument(next_arg);
        }
        else if (next_arg == "--")
        {
            // we support the conventional use of a bare -- to turn
            // off option processing and allow filenames that look like
            // options to be accepted
            accepting_options = false;
        }
        else if (next_arg == "--help")
        {
            PrintUsage(cout);
            exit(1);
        }
        else...
    }

    // we fill in defaults for things which have not been set
    if (outdir == "")...
    if (db_outfile == "")...
    if (html_outfile == "")...
    if (xml_outfile == "")...
    if (opt_outfile == "")...
    // the other strings all default to empty values
    if (opt_infile == "")...
    else...
}

```

```

// the options below this point are all of the form --opt=val,
// so we parse the argument to find the assignment
unsigned int assignment_pos = next_arg.find("=");
if (assignment_pos == string::npos)
{
    cerr << "Unexpected option " << next_arg << endl;
    PrintUsage(cerr);
    exit(2);
}
else
{
    string next_opt = next_arg.substr(0, assignment_pos);
    string next_val = next_arg.substr(assignment_pos + 1);

    if (next_opt == "--outdir")
    {
        outdir = next_val;
    }
    else if (next_opt == "--db_infile")
    {
        db_infile = next_val;
    }
    else if (next_opt == "--db_outfile")
    {
        db_outfile = next_val;
    }
    else if (next_opt == "--opt_infile")...
    else if (next_opt == "--opt_outfile")...
    else if (next_opt == "--html_outfile")...
    else if (next_opt == "--xml_outfile")...

    else if (next_opt == "--lang")...
    else if (next_opt == "--report_mask")...
    else if (next_opt == "--debug_mask")
    {
        // The report option may either be an integer flag vector
        // in numeric format (including hex) or may be a string of
        // characters (see HandleDebugOption for what they mean).
        HandleDebugOption(next_val.c_str());
    }
    else
    {
        cerr << "Unexpected option " << next_opt << endl;
        PrintUsage(cerr);
        exit(3);
    }
}

```

```

// we fill in defaults
if (outdir == "")
{
    outdir = ".cccc";
}
if (db_outfile == "")
{
    db_outfile = outdir + "/cccc.db";
}
if (html_outfile == "")
{
    html_outfile = outdir + "/cccc.html";
}
if (xml_outfile == "")
{
    xml_outfile = outdir + "/cccc.xml";
}
if (opt_outfile == "")
{
    opt_outfile = outdir + "/cccc.opt";
}

// the other strings all default to empty values

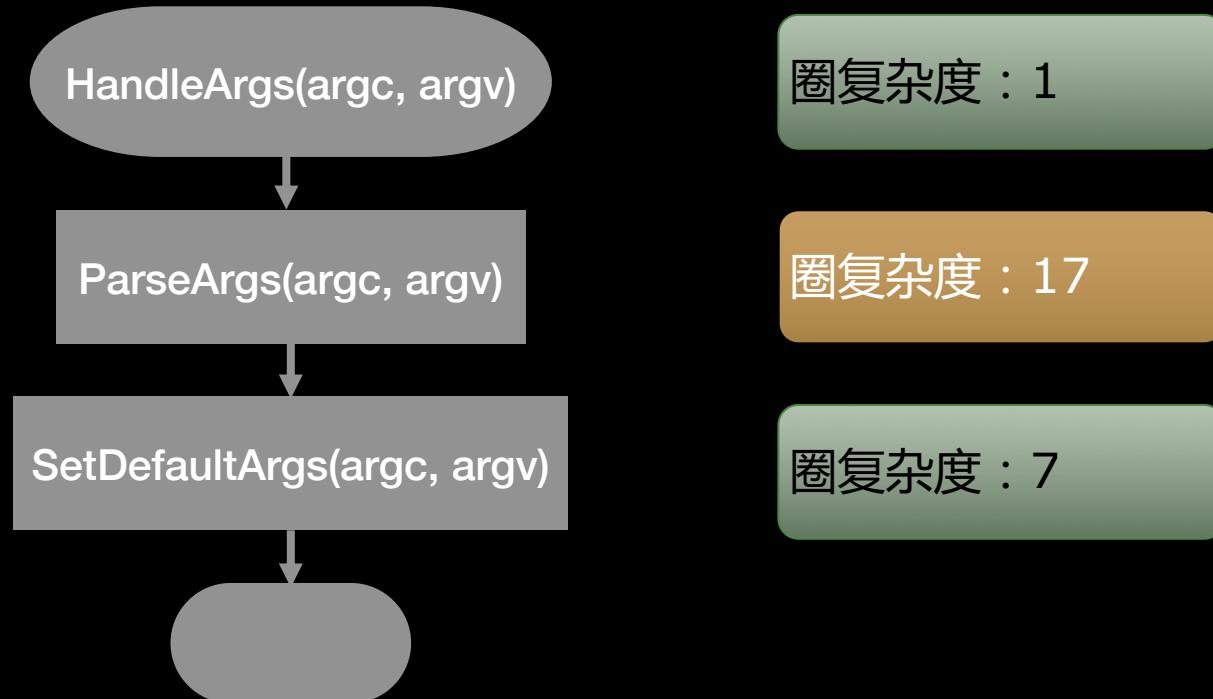
if (opt_infile == "")
{
    CCCC_Options::Load_Options();
}

// save the options so that they can be edited
CCCC_Options::Save_Options(opt_outfile);
}
else
{
    CCCC_Options::Load_Options(opt_infile);
}

```

圈复杂度 : 23

# | 使用提取函数降低复杂度



# 用表驱动的方式降低圈复杂度

```
string next_opt = next_arg.substr(0, assignment_pos);
string next_val = next_arg.substr(assignment_pos + 1);

map<string, function<void(const string&)> > opt_parse = {
    { "--outdir" , [this](const string& next_val) {outdir = next_val;} },
    { "--db_infile", [this](const string& next_val) {db_infile = next_val;} },
    { "--db_outfile", [this](const string& next_val) {db_outfile = next_val;} },
    { "--opt_infile", [this](const string& next_val) {opt_infile = next_val;} },
    { "--opt_outfile", [this](const string& next_val) {opt_infile = next_val;} },
    { "--html_outfile", [this](const string& next_val) {html_outfile = next_val;} },
    { "--xml_outfile", [this](const string& next_val) {xml_outfile = next_val;} },
    { "--lang", [this](const string& next_val) {lang = next_val;} },
    { "--report_mask", [this](const string& next_val) {HandleReportOption(next_val.c_str());} },
    { "--debug_mask", [this](const string& next_val) {HandleDebugOption(next_val.c_str());} }
};

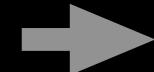
auto it = opt_parse.find(next_opt);
if(it != opt_parse.end()) {
    it->second();
}
else
{
    cerr << "Unexpected option " << next_opt << endl;
    PrintUsage(cerr);
    exit(3);
}
```

圈复杂度 : 8

# 利用多态减小圈复杂度

- 静态多态

```
void PrintShape(Shape shape) {
    switch(shape) {
        case Square:
            cout << "Square" << endl;
            break;
        case Circle:
            cout << "Circle" << endl;
            break;
        case Rectangle:
            cout << "Rectangle" << endl;
            break;
        case Arrow:
            cout << "Arrow" << endl;
            break;
        case Star:
            cout << "Star" << endl;
            break;
        case Ellipse:
            cout << "Ellipse" << endl;
    }
}
```



圈复杂度 : 7

```
template<Shape shape>
struct TypeName {
    static inline const std::string name = "Invalid Shape";
};

template<>
struct TypeName<Square> {
    static inline const std::string name = "Square";
};

template<>
struct TypeName<Circle> {
    static inline const std::string name = "Circle";
};

template<Shape shape>
void PrintShape() {
    cout << TypeName<shape>::name << endl;
}
```

圈复杂度 : 1

# 圈复杂度的问题

```
string getWords(int number) // +1
{
    switch (number)
    {
        case 1: // +1
            return "one";
        case 2: // +1
            return "a couple";
        case 3: // +1
            return "a few";
        default:
            return "lots";
    }
}
```

==

```
int sum(int max) // +1
{
    int total = 0;

    for (int i = 1; i <= max; ++i) // +1
    {
        for (int j = 2; j < i; ++j) // +1
        {
            if (i % j == 0
                || i % 2 == 0
                && j % 3 != 0) // +1
            {
                continue;
            }
        }
        total += i;
    }
    return total;
}
```

# | 认知复杂度

- 嵌套越深，复杂度越高
- 判断条件越复杂，复杂度越高
- 递归增加复杂度
- break增加复杂度

# 认知复杂度

```
string getWords(int number)
{
    switch (number)          // +1
    {
        case 1:
            return "one";
        case 2:
            return "a couple";
        case 3:
            return "a few";
        default:
            return "lots";
    }
}
```

!=

```
int sum(int max)
{
    int total = 0;

    for (int i = 1; i <= max; ++i) // +1
    {
        for (int j = 2; j < i; ++j) // +2
        {
            if (i % j == 0           // +3
                || i % 2 == 0         // +1
                && j % 3 != 0)       // +1
            {
                continue;           // +1
            }
        }
        total += i;
    }
    return total;
}
```

# 静态代码度量工具的对比

	SourceMonitor	Sonarqube sonar-cxx	cccc
度量项	代码行数, 语句数目, 分支语句比例, 注释语句比例, 函数数目 每函数平均语句数 函数圈复杂度 函数深度 类的数量	代码行数, 语句数目, 注释比例, 函数圈复杂度, 认知复杂度 函数深度	代码行, 圈复杂度, 注释行数, 注释比例, 注释和圈复杂度的比例, 扇入扇出, 继承树深度, 子类个数,
运行环境	Windows	Windows/Linux/MacOS	Windows/Linux/MacOS
支持语言	C, C++, C#, Java, Delphi, HTML	全语言	C, C++, Java
收费模式	免费	社区版免费, 开发版收费	免费

- 代码质量
- 代码审查
  - 1. 代码度量分析
  - 2. 静态代码检查
  - 3. 动态代码检查
- 代码检查的开源解决方案

# | 静态代码检查的作用

- 代码规范检查
- 代码缺陷检查
- 代码性能问题
- 代码坏味道

# 帮助检查代码缺陷

```
void test_basic1(int in, int &out) {  
    if (in > 77)  
        out++;  
    else  
        out++;  
  
    out++;  
}
```

由于拷贝粘贴造成两个分支的代码完全相同

# 帮助检查代码缺陷

```
class MyClass {  
public:  
    bool badFunc(int a, int b) { return a * b > 0; }  
    bool goodFunc(int a, int b) const { return a * b > 0; }  
  
    MyClass &operator=(const MyClass &rhs) { return *this; }  
  
    int operator-() { return 1; }  
  
    operator bool() const { return true; }  
  
    void operator delete(void *p) {}  
};  
  
int main() {  
  
    int X = 0;  
    bool B = false;  
    assert(X == 1);  
    assert(X = 1);  
  
    MyClass mc;  
    assert(mc.badFunc(0, 1));  
    assert(mc.goodFunc(0, 1));  
  
    return 0;  
}
```

这两处assert因为产生副作用，会造成调试模式和生产模式行为上的不一致

# 代码缺陷-没有用的RAII

```
int append(int data) {  
    std::lock_guard<std::mutex>(lk);  
    list.push_back(data);  
}
```

临时对象，语句结束后执行析构函数，是误用的加锁

```
int append(int data) {  
    std::lock_guard<std::mutex> lock(lk);  
    list.push_back(data);  
}
```

# 性能问题-没有必要copy构造函数

```
struct ExpensiveToCopyType {  
    ExpensiveToCopyType();  
    virtual ~ExpensiveToCopyType();  
    const ExpensiveToCopyType &reference() const;  
    void nonConstMethod();  
    bool constMethod() const;  
};  
  
void PositiveMethodCallConstParam(const ExpensiveToCopyType Obj) {  
    const auto AutoAssigned = Obj.reference();  
    // const auto& AutoAssigned = Obj.reference();  
    const auto AutoCopyConstructed(Obj.reference());  
    // const auto& AutoCopyConstructed(Obj.reference());  
    const ExpensiveToCopyType VarAssigned = Obj.reference();  
    // const ExpensiveToCopyType& VarAssigned = Obj.reference();  
    const ExpensiveToCopyType VarCopyConstructed(Obj.reference());  
    // const ExpensiveToCopyType& VarCopyConstructed(Obj.reference());  
}
```

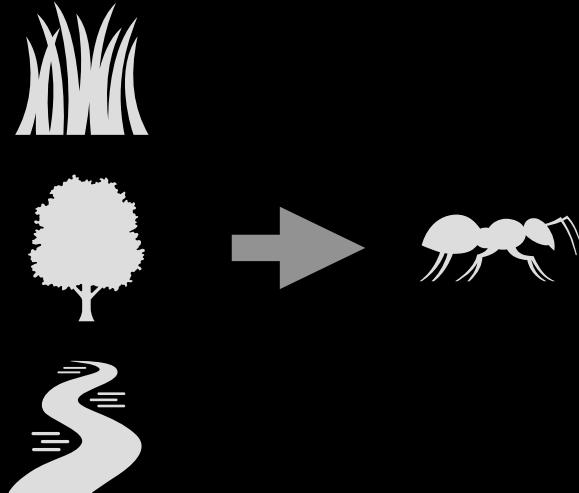
# 代码坏味道

```
void foo(int p1, int p2, int p3, int p4, int p5, int p6) {  
    int val1 = p1;  
    int val2 = p2;  
    int val3 = p3;  
    int val4 = p4;  
    int val5 = p5; // 过长的参数列表  
    int val6 = p6;  
  
    // 省略100行代码  
    // ...  
}
```

过长的参数列表

过大的函数

# 静态检查



- 静态检查不是银弹
  - 能够帮助检查代码缺陷
  - 对于检查规则覆盖的代码，能够工作得很好；对于规则没有覆盖的代码，无能为力
  - 会存在误报

# 静态分析工具的对比

	cppcheck	PC-lint	Coverity	QAC C/C++	Clang-Tidy	Clang Static Analyzer
支持语言	C/C++	C/C++	C/C++/Java/C#	C/C++	C/C++/Object C	C/C++/Object C
检查项	200+项	900	500	1000+规则	250+项	代码缺陷：40+
支持平台	Windows/Linux/MacOS	Windows/Linux/MacOS	Windows/Linux/MacOS	Windows/Linux	Windows/Linux/MacOS	Windows/Linux/MacOS
支持的使用方式	命令行 Clion/Eclipse/VsCode插件 Jenkins插件	命令行	多种使用方式， 命令行， dashboard，sonar 插件，jenkins插件	命令行、 dashboard	命令行， Clion/VsCode/VsStudio插件 Sonar插件	命令行， 编辑器插件， Sonar插件
代码规范支持	MISRA 20项	MISRA 90%	MISRA 100%	MISRA, CERT	MISRA 80+项， 可定制	MISRA规则：22项，可定制
误报率	中	高	低	低	中	中
收费模式	免费	收费 9500¥ * 5个 license	收费高	10W+	开源	开源
Devops支持	容易集成	容易集成	容易	较难	容易	容易

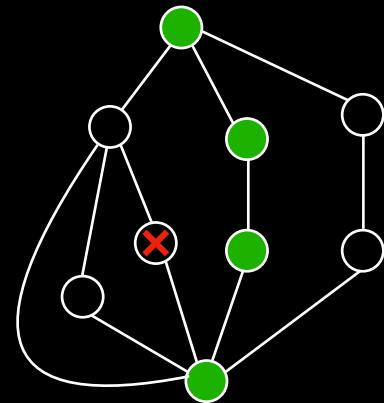
- 代码质量
- 代码审查
  - 1. 代码度量分析
  - 2. 静态代码检查
  - 3. 动态代码检查
- 代码检查的开源解决方案

# 动态分析工具

- 可以在程序运行时发现代码缺陷
  - 内存、线程、未定义行为
- 常用工具
  - GCC & Clang
    - ▶ Asan - Address Sanitizer(缓冲区溢出，内存泄露)
    - ▶ Tsan - Thread Sanitizer(并发问题)
    - ▶ Msan - Memory Sanitizer (未初始化内存)
    - ▶ Ubsan - Undefined Behavior Sanitizer (未定义行为)
    - ▶ 编译选项添加 -  
fsanitize=address/memory/thread/undefined
  - Valgrind
    - ▶ memcheck - 内存问题，包括Asan和Msan
    - ▶ helgrind - 线程和并发问题
    - ▶ cachegrind、callgrind、massif - 帮助进行性能优化
    - ▶ Valgrind --tool=memcheck/helgrind/massif [prog]
  - 与单元测试、功能测试、系统测试结合，提高覆盖率，可以帮助发现更多缺陷

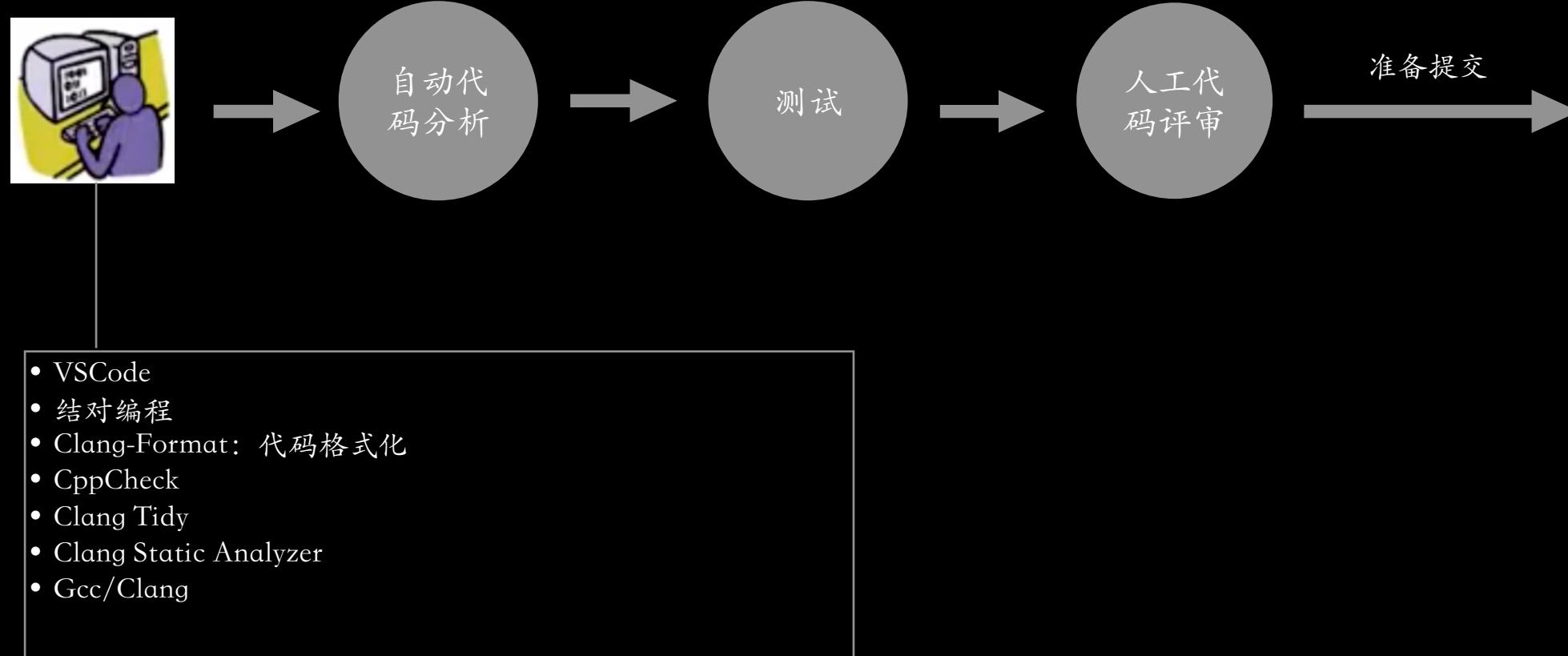
# 动态分析工具

- 能够准确定位问题，误报率低
- 和测试用例绑定在一起，查找缺陷的比例与测试用例的覆盖率有关
- 对于没有覆盖到的错误，动态分析工具无能为力
- 嵌入式系统会存在内存资源的限制
- 应该要“动静结合”，多种检查手段结合，能够更有效地提升代码质量

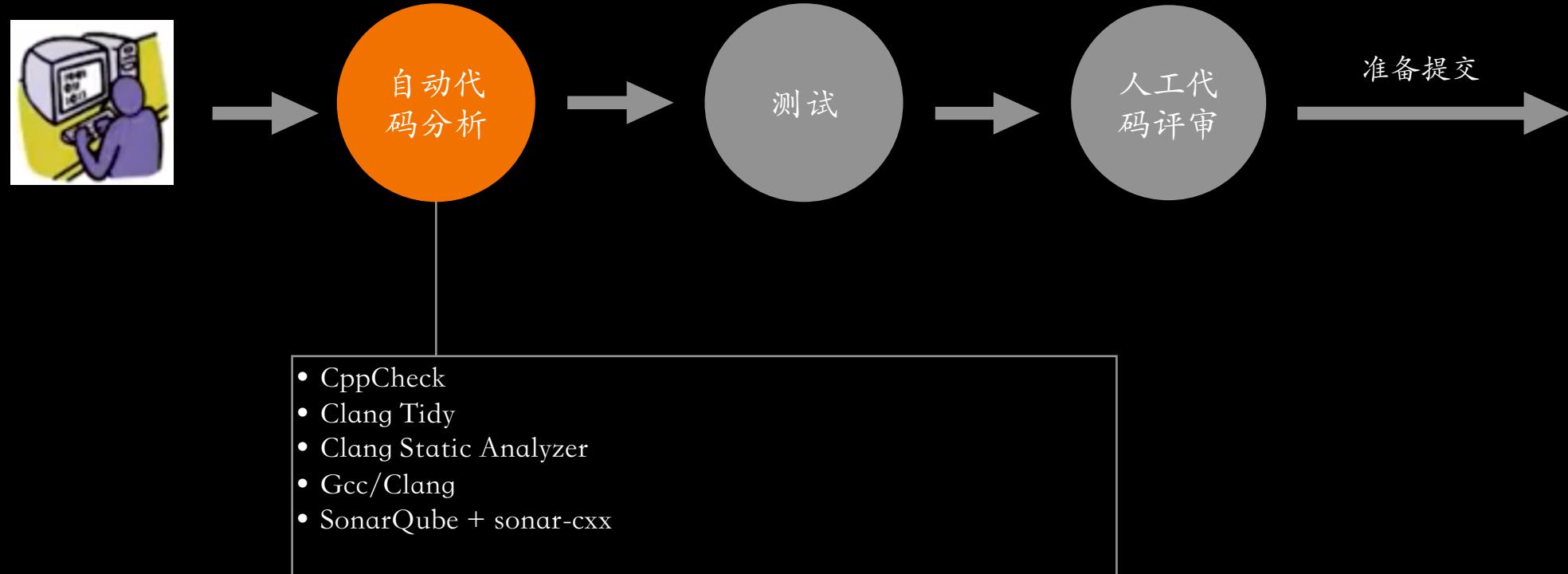


- 软件质量
- 代码审查
- | • 代码检查的开源解决方案

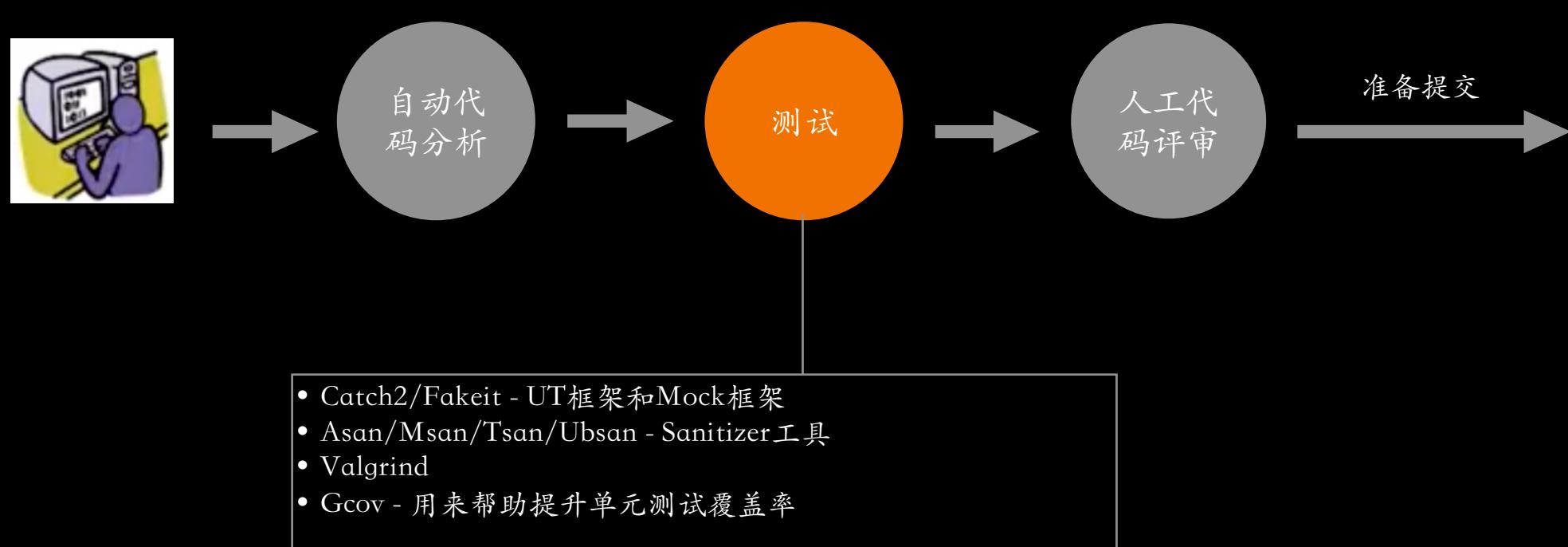
# 工程中采用的代码检查方案



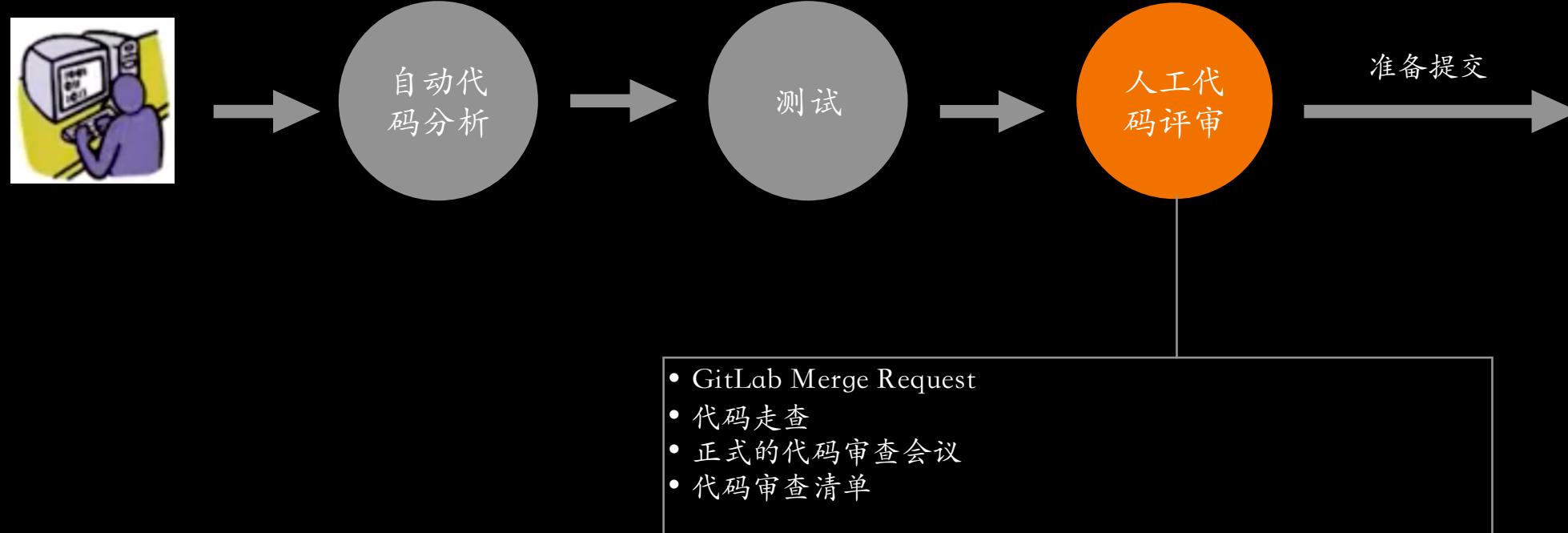
# | 工程中采用的代码检查方案



# | 工程中采用的代码检查方案



# | 工程中采用的代码检查方案



- 软件质量
- 代码审查
- 代码检查的开源解决方案

- | 1. Clang
- 2. Clang-tidy
- 3. Clang Static Analyzer
- 4. SonarQube

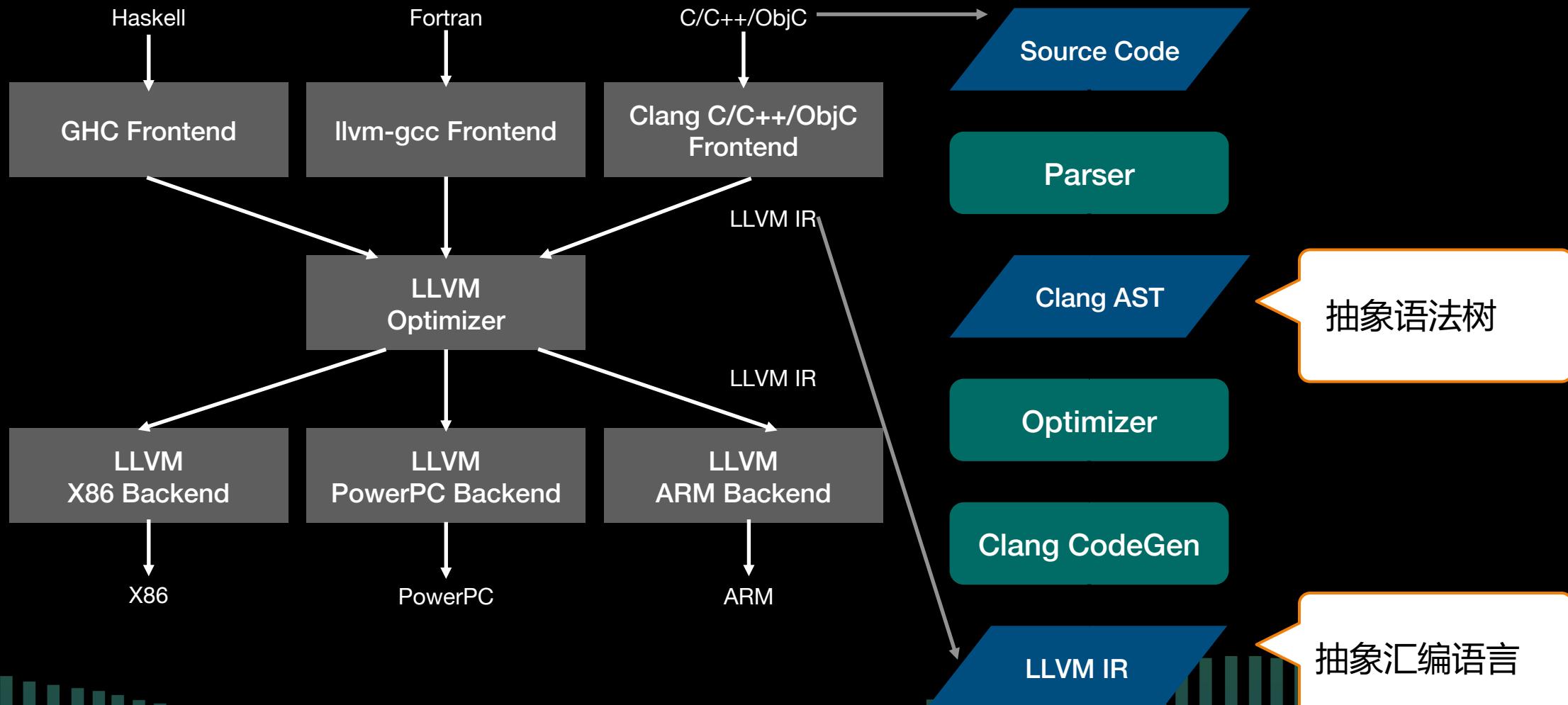
# | 用到的Clang工具

- Clang
- Clang-Format
- Clang-Tidy
- Clang Static Analyzer
- sanitizer: msan/asan/tsan/ubsan

# Gcc和Clang/LLVM

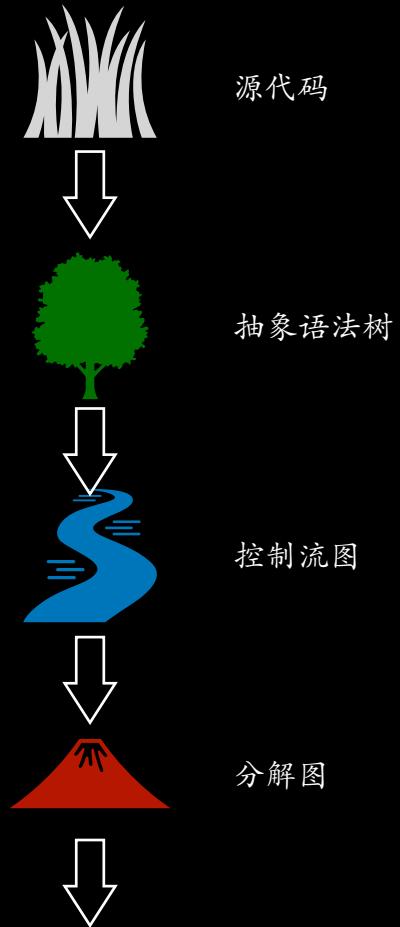
- GCC难以扩展； Clang是模块化的，更容易扩展
- GCC支持更多传统的语言， 支持更多的硬件架构， Linux内核目前还是只能用GCC编译
- Clang提供了很多有用的工具，如Clang Static Analyzer， Clang Tidy， Clang Format， 重构工具， Sanitizer工具和IDE插件等

# LLVM/Clang架构



# Clang-tidy和Static Analyzer

- Clang-tidy使用AST Matcher
  - Clang前端把源码解析成抽象语法树
  - Clang-tidy使用visitor模式访问抽象语法树
- Clang Static Analyzer使用符号执行
  - 抽象语法树转成控制流图，然后把控制流图转成分解图
  - 分解图是模拟符号在控制流图的不同路径进行计算的过程，生成不同的符号执行路径
  - 对不同路径上的执行进行分析



# AST Matcher

```
#define ZERO 0

void test(int b)
{
    int a, c;
    double *d;
    switch (b)
    {
        case 1:
            a = b / 0;
            break;
        case 2:
            a = b / ZERO;
            break;
        case 4:
            c = b - 4;
            a = b / c;
            break;
    }
}
```

```
BUILD_MATCHER() {
    return binaryOperator(
        hasOperatorName("/"),
        hasRHS(integerLiteral(equals(0)))
        .bind(KEY_NODE);
}
```

b / 0

发现

预处理指令解析，能够发现

不进行符号化执行，不能通过AST  
matcher发现

- 软件质量
- 代码审查
- 代码检查的开源解决方案
  - 1. Clang
  - 2. Clang-tidy
  - 3. Clang Static Analyzer
  - 4. SonarQube

# Clang tidy

- 是一个基于Clang的C++ lint框架
- 能够访问抽象语法树和预处理器
- Clang-tidy是可扩展的 - 可以定制检查规则
- 目前有超过200个检查规则
  - Readability, efficiency, correctness, modernization
  - 高度可配置
  - 很多规则可以自动修复代码

# | Clang-tidy使用

- 典型的clang-tidy调用
- -checks=<string>
  - "\*": 使能所有检查
  - "-\*,modernize-\*": 只使能modernize模块的检查器

```
clang-tidy test.cpp -- -Imy_project/include -DMy_DEFINES ...
```

```
clang-tidy --list-checks -checks='*' | grep "modernize"  
modernize-avoid-bind  
modernize-deprecated-headers  
modernize-loop-convert  
modernize-make-shared  
modernize-make-unique  
modernize-pass-by-value  
modernize-raw-string-literal  
modernize-redundant-void-arg  
modernize-replace-auto_ptr  
modernize-shrink-to-fit  
modernize-use-auto  
modernize-use-bool-literals  
modernize-use-default  
modernize-use-emplace  
modernize-use-nullptr  
modernize-use-override  
modernize-use-using  
...
```

# 使用clang-tidy修复代码

```
struct Base {  
    virtual void reimplementMe(int a) {}  
};  
struct Derived : public Base {  
    virtual void reimplementMe(int a) {}  
};
```



```
struct Base {  
    virtual void reimplementMe(int a) {}  
};  
struct Derived : public Base {  
    void reimplementMe(int a) override {}  
};
```

```
clang-tidy -checks='-*现代化-use-override' -fix test.cpp -- -std=c++11
```

- 不是所有的checker都有“fix”
  - 有些规则是提示程序员出现错误，但如何修正还需要程序员确定方案
  - modernize的大部分checker都有修复，能够帮助程序员快速地迁到现代C++

# checker的灵活配置

```
clang-tidy-7 -config="{Checks: '*',readability-identifier-naming', CheckOptions: [ \
{ key: readability-identifier-naming.NamespaceCase,           value: lower_case }, \
{ key: readability-identifier-naming.ClassCase,             value: CamelCase }, \
{ key: readability-identifier-naming.StructCase,            value: CamelCase }, \
{ key: readability-identifier-naming.TemplateParameterCase, value: CamelCase }, \
{ key: readability-identifier-naming.FunctionCase,          value: CamelCase }, \
{ key: readability-identifier-naming.VariableCase,          value: lower_case }, \
{ key: readability-identifier-naming.PrivateMemberSuffix,   value: _ }, \
{ key: readability-identifier-naming.ProtectedMemberSuffix, value: _ }, \
{ key: readability-identifier-naming.MacroDefinitionCase,   value: UPPER_CASE }, \
{ key: readability-identifier-naming.EnumConstantCase,      value: UPPER_CASE }, \
{ key: readability-identifier-naming.ConstexprVariableCase, value: CamelCase }, \
{ key: readability-identifier-naming.ConstexprVariablePrefix, value: k }, \
{ key: readability-identifier-naming.GlobalConstantCase,    value: CamelCase }, \
{ key: readability-identifier-naming.GlobalConstantPrefix,  value: k }, \
{ key: readability-identifier-naming.MemberConstantCase,    value: CamelCase }, \
{ key: readability-identifier-naming.MemberConstantPrefix,   value: k }, \
{ key: readability-identifier-naming.StaticConstantCase,     value: CamelCase }, \
{ key: readability-identifier-naming.StaticConstantPrefix,   value: k }, \
{ key: readability-identifier-naming.ClassMemberCase,         value: lower_case }, \
{ key: readability-identifier-naming.ClassMemberSuffix,       value: _ }]}"
\

-dump-config >.clang-tidy
```

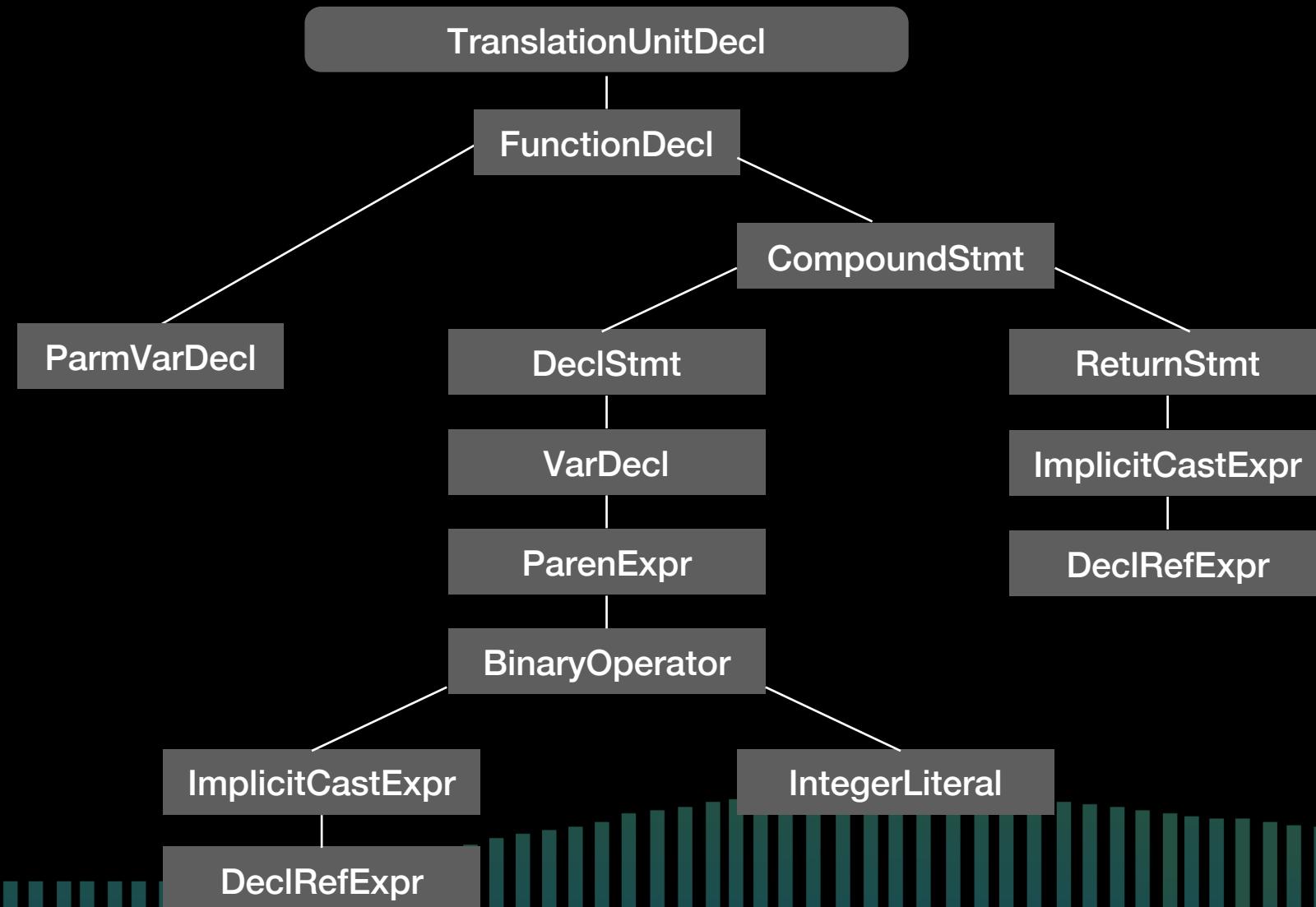
- 命名规范
  - 类型命名每个单词首字母大写，不含下划线，以名词形式
  - 变量名一律小写，单词用下划线相连
  - 常量命名：k后面跟大写字母开头的单词
  - 函数每个单词首字母大写，使用命令式语气
  - 命名空间全小写，并基于项目名称和目录结构
  - 类成员变量，以下划线做前缀，小写，单词之间用下划线相连

# | 定制clang-tidy的检查器

- 团队选择定制开发clang-tidy检查器的原因
  - 团队根据自身的编码规范，需要定制开发clang-tidy的检查器
  - 计划选择开源工具来支持C++编码规范标准，譬如
    - CERT C++
    - Autosar C++
    - C++ core guideline

# Clang AST

```
$ cat test.cc
int f(int x) {
    int result = (x / 42);
    return result;
}
```



# | Clang AST的核心类

- Decl
  - TranslationUnitDecl
  - FunctionDecl
  - CXXRecordDecl
  - VarDecl
  - ...
- Stmt
  - CompoundStmt
  - DeclStmt
  - ReturnStmt
  - ImplicitCastExpr
  - DeclRefExpr
  - BinaryOperator
- Type
  - PointerType
  - ReferentType
  - BuiltinType
  - ArrayType
  - ...

# Clang-tidy – Clang AST Matcher

- Clang AST Matcher 分为三类
  - Node Matcher: 匹配某种特定类型的AST节点, 例如
    - `functionDecl()`, 匹配所有的函数定义
  - Narrowing Matchers: 按照节点的某个特定属性匹配AST节点, 可以帮助缩小AST节点的匹配范围, 例如
    - `functionDecl(allof(hasName("f")),unless(cxxMethodDecl()))`, 匹配名字是“f”的自由函数
  - Traversal Matchers: 指定从当前节点可到达的与其他节点的关系, 例如
    - `functionDecl(hasName("f")), hasParent(namespaceDecl(hasName("foo")))`, 命名空间foo里的名字是“f”的函数

# | 开发自定义的checker

Coding Rule : catch语句只接受常量引用变量

```
try {  
    check();  
} catch (logic_error& e) {  
}
```



```
try {  
    check();  
} catch (const logic_error& e) {  
}
```



# | 在LLVM添加自定义checker

```
$ cd [llvm project path]/clang-tools-extra/clang-tidy  
$ ./add_new_check.py misc catch-by-const-ref
```

```
void CatchByConstRefCheck::registerMatchers(MatchFinder *Finder) {  
    if(!getLangOpts().CPlusPlus) {  
        return;  
    }  
  
    Finder->addMatcher(varDecl(isExceptionVariable(),  
                                hasType(  
                                    references(  
                                        qualType(  
                                            unless(  
                                                isConstQualified() // const限定符  
                                            )  
                                            )  
                                            )  
                                ).bind("catch"),  
                            this);  
}
```

# 找到匹配的代码报错和修复

```
void CatchByConstRefCheck::check(const MatchFinder::MatchResult &Result) {
    const auto* varCatch = Result.Nodes.getNodeAs<VarDecl>("catch");
    static const char* diagMsgCatchReference =
        "catch handler catches by non const reference; "
        "catching by const-reference may be more efficient";

    // Emit error message if the type is not const (ref)s

    diag(varCatch->getBeginLoc(), diagMsgCatchReference)
        << FixItHint::CreateInsertion(varCatch->getBeginLoc(), "const ");
}
```

- 软件质量
- 代码审查
- 代码检查的开源解决方案
  - 1. Clang介绍
  - 2. Clang-tidy
  - 3. Clang Static Analyzer
  - 4. SonarQube

# Clang Static Analyzer

- 基本用法 clang --analyze -Xclang [file]
- 指定checker clang --analyze -Xclang -analyzer-checker=[checker字符串] [file]
  - 例如 clang --analyze -Xclang -analyzer-checker=alpha test.cpp
- clang-tidy 可以调用CSA的checker
  - clang-tidy test.cpp --checks="-\*,clang-analyzer-\*" --
- 目前支持的检查器
  - default checker(88项)
  - 实验checker (50项)
  - debug checker 调试CFG、Exploded Graph (分解图)

# CSA的诊断结果

Show analyzer invocation

Show only relevant lines

```
1 void test(int b) {  
2     int a, c;  
3     switch (b) {  
  
4         case 1: a = b / 0;  
5             break;  
6         case 4: c = b - 4;  
  
7             a = b / c;  
  
8             break;  
9     }  
10 }
```

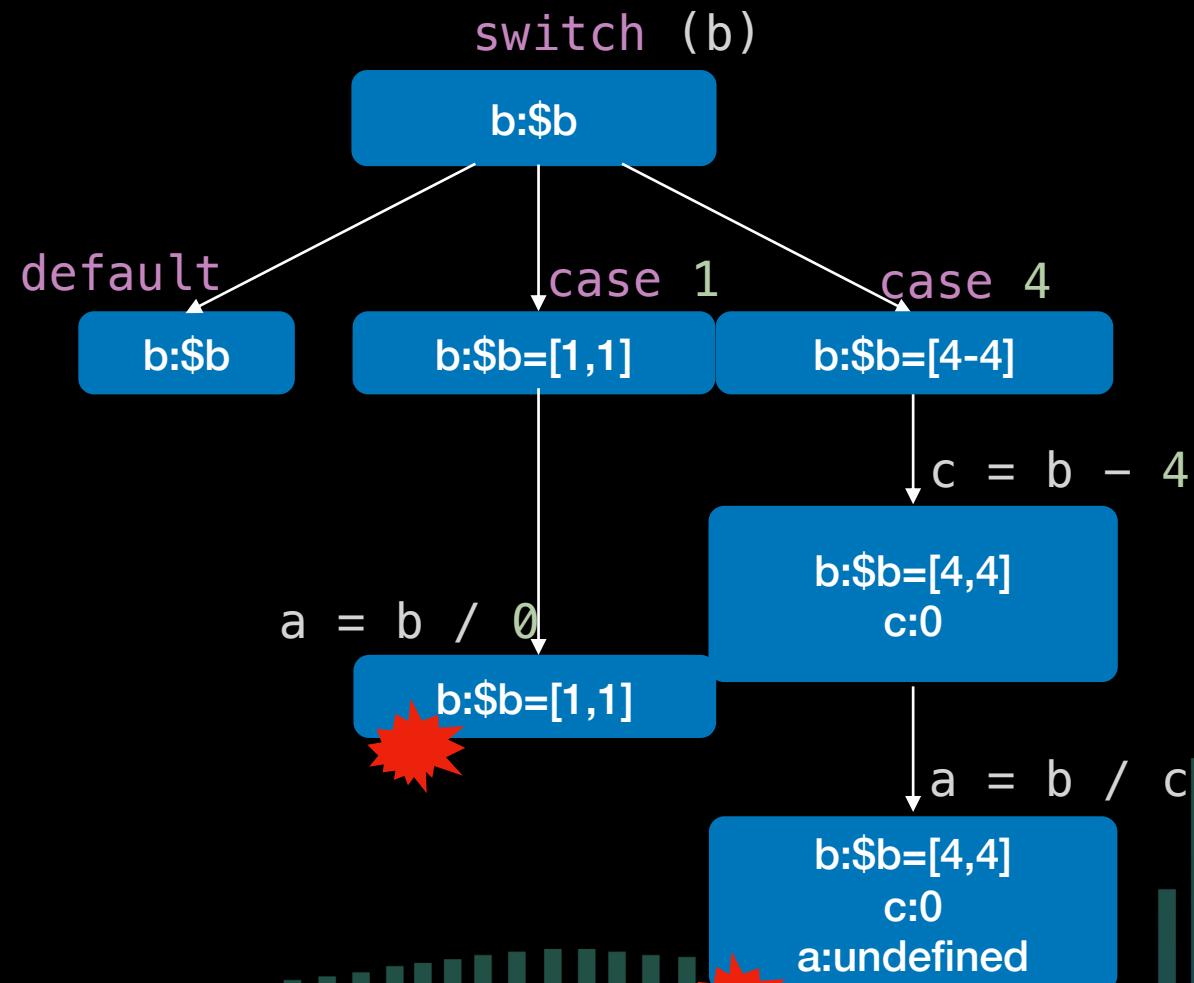
1 Control jumps to 'case 4:' at line 6 →

2 ← The value 0 is assigned to 'c' →

3 ← Division by zero

# Clang Static Analyzer的符号执行

- 不需要执行代码就能发现bug
- 基于路径的分析方法
- 用控制流图来创造出模拟不同执行路径的分解图



# 抽象语法树 vs 控制流图

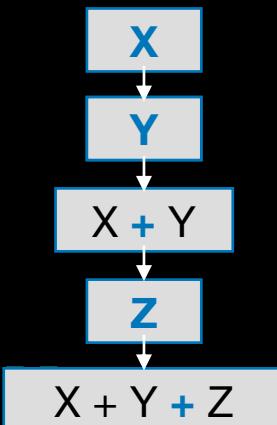
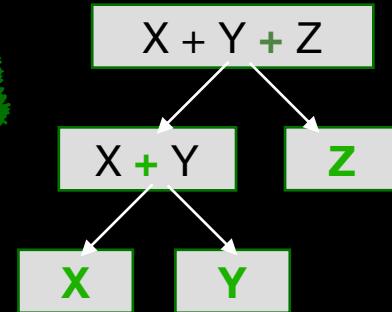
- 抽象语法树AST
- 控制流图CFG

- Node: 语句、声明和类型
- Edge: 节点之间的关系

- Node: 通常是AST语句
- Edge: 顺序执行的关系
- 按照语句执行的顺序



$X + Y + Z$



# 抽象语法树 vs 控制流图

- 抽象语法树AST

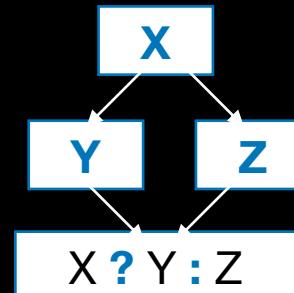
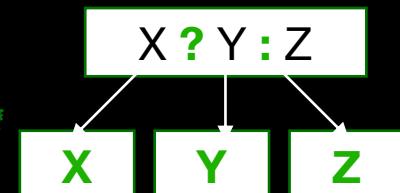
- Node: 语句、声明和类型
- Edge: 节点之间的关系

- 控制流图CFG

- Node: 通常是AST语句
- Edge: 顺序执行的关系
- 按照语句执行的顺序

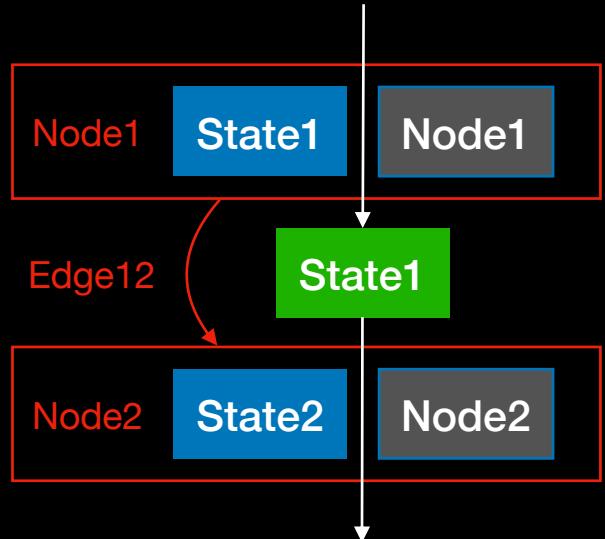


X ? Y : Z

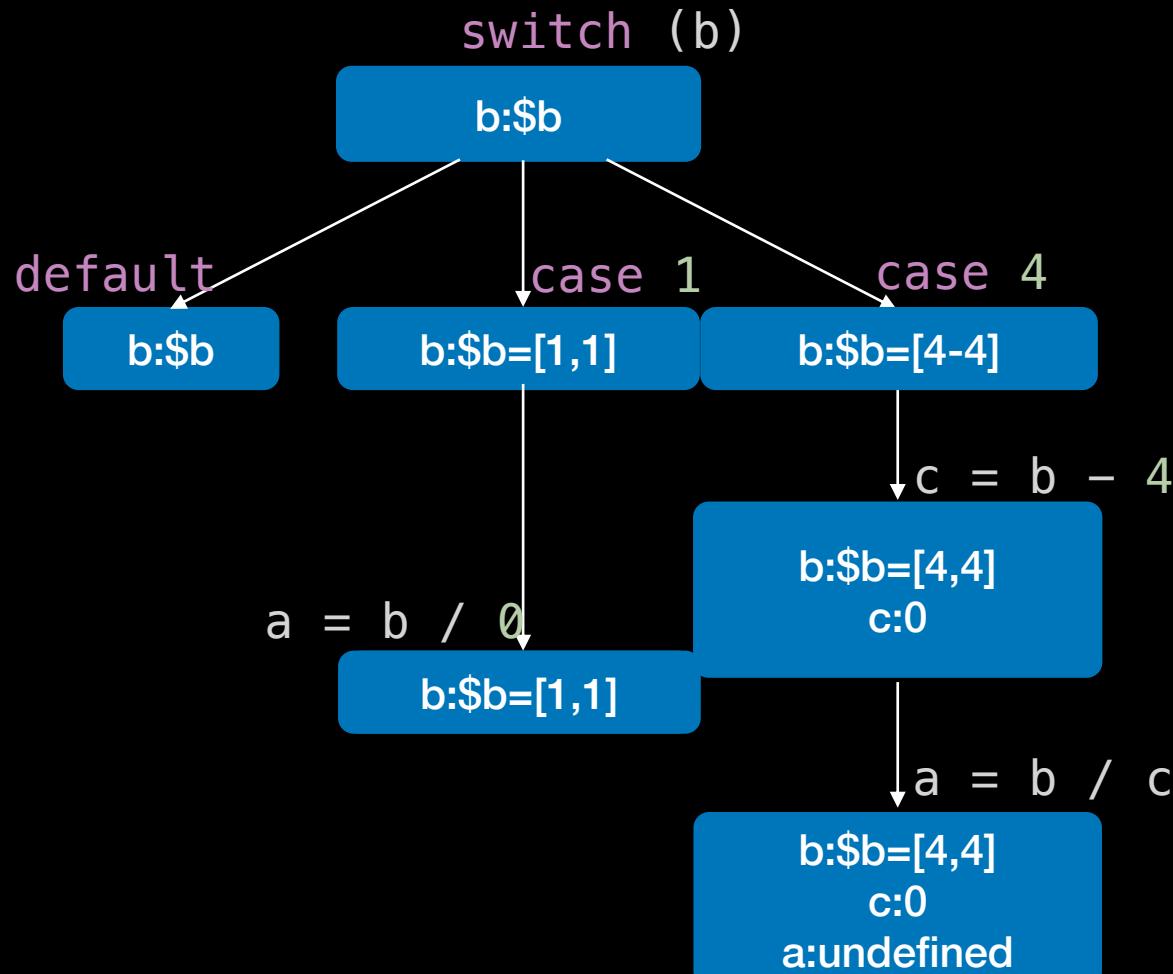


# 分解图Exploded Graph

- Node: (Point, State) pair
  - Program Point: 两个语句Statement之间的点
    - ▶ 执行位置: pre-statement、post-statement、entering a call...
    - ▶ Stack frame
  - Program State: 在当前节点之前statement模拟执行的效果的记录
    - ▶ Environment-从源代码表达式到符号值的映射
    - ▶ Store-从内存位置到符号值的映射
    - ▶ GenericDataMap-符号值的约束
- Edge: 从 (Point1, State1) 到 (Point2, State2) 的edge, 是指Point1和Point2之间的语句statement



# 分解图的示例

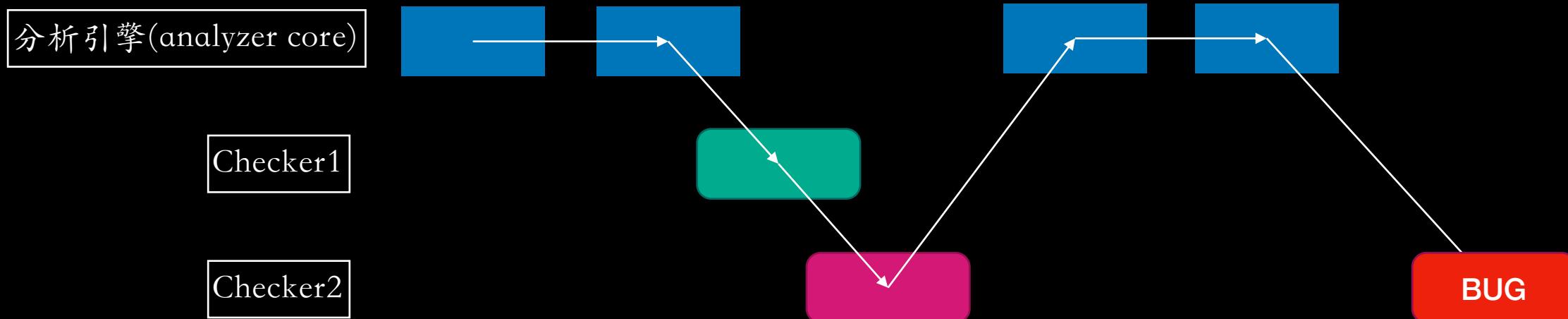


```

{
  "program_points": [
    { "kind": "Statement", "stmt_kind": "DeclRefExpr", "stmt_id": 734, "pointer": "0x7fe0ec04d6f0" },
    { "kind": "Statement", "stmt_kind": "DeclRefExpr", "stmt_id": 734, "pointer": "0x7fe0ec04d6f0" }
  ],
  "program_state": {
    "store": { "pointer": "0x7fe0ec83a338", "items": [
      { "cluster": "c", "pointer": "0x7fe0ec839e88", "items": [
        { "kind": "Direct", "offset": 0, "value": "0 S32b" }
      ]}
    ]},
    "environment": { "pointer": "0x7fe0ebd092a0", "items": [
      { "lctx_id": 1, "location_context": "#0 Call", "calling": "test", "location": null, "items": [
        { "stmt_id": 711, "pretty": "c", "value": "&c" },
        { "stmt_id": 726, "pretty": "b - 4", "value": "0 S32b" },
        { "stmt_id": 730, "pretty": "c = b - 4", "value": "&c" },
        { "stmt_id": 734, "pretty": "a", "value": "&a" },
        { "stmt_id": 738, "pretty": "b", "value": "&b" },
        { "stmt_id": 742, "pretty": "c", "value": "&c" },
        { "stmt_id": 746, "pretty": "b", "value": "4 S32b" },
        { "stmt_id": 749, "pretty": "c", "value": "0 S32b" },
        { "stmt_id": 752, "pretty": "b / c", "value": "Undefined" }
      ]}
    ]},
    "constraints": [
      { "symbol": "reg_0<int b>", "range": "{ [4, 4] }" }
    ],
    "dynamic_types": null,
    "dynamic_casts": null,
    "constructing_objects": null,
    "checker_messages": null
  }
}
  
```

# 扩展检查器

- CSA分成两部分：分析引擎（analyzer core）和检查器（checker）
  - checker在分析引擎的基础上运行
  - 增加新的检查，大多是使用分析引擎提供的功能编写新的checker
- Checker在发现bug的时候可以停止分析引擎



# Checker都是观察者

```
void checkPreStmt(const UnaryOperator *UO, CheckerContext &C) const;
void checkPreStmt(const BinaryOperator *BO, CheckerContext &C) const;
void checkPreStmt(const ArraySubscriptExpr *ASE, CheckerContext &C) const;
void checkPreStmt(const MemberExpr *ME, CheckerContext &C) const;
```

在Statement被处理前的观察点

```
void checkPreCall(const CallEvent &Call, CheckerContext &C) const
```

发生在调用call之前的观察点

```
void checkPostCall(const CallEvent &Call, CheckerContext &C) const
```

发生在调用call之后的观察点

还有很多的check点，可以参考[llvm-project/clang/include/clang/StaticAnalyzer](https://llvm-project/clang/include/clang/StaticAnalyzer)

# DivideZero Checker

```

void DivZeroChecker::checkPreStmt(const BinaryOperator *B,
                                  CheckerContext &C) const {
    BinaryOperator::Opcode Op = B->getOpcode();
    if (Op != BO_Div &&
        Op != BO_Rem &&
        Op != BO_DivAssign &&
        Op != BO_RemAssign)
        return;

    if (!B->getRHS()->getType()->isScalarType())
        return;

    SVal Denom = C.getSVal(B->getRHS());
    Optional<DefinedSVal> DV = Denom.getAs<DefinedSVal>();

    // Divide-by-undefined handled in the generic checking for uses of
    // undefined values.
    if (!DV)
        return;
}

```

判断运算符是  
"/"或"%"

```

// Check for divide by zero.
ConstraintManager &CM = C.getConstraintManager();
ProgramStateRef stateNotZero, stateZero;
std::tie(stateNotZero, stateZero) = CM.assumeDual(C.getState(), *DV);

if (!stateNotZero) {
    assert(stateZero);
    reportBug("Division by zero", stateZero, C);
    return;
}

bool TaintedD = isTainted(C.getState(), *DV);
if ((stateNotZero && stateZero && TaintedD)) {
    reportBug("Division by a tainted value, possibly zero", stateZero, C,
              std::make_unique<taint::TaintBugVisitor>(*DV));
    return;
}

// If we get here, then the denom should not be zero.
// zero denom case for now.
C.addTransition(stateNotZero);
}

```

从constraint  
取出零状态  
和非零状态

如果没有非  
零状态，就  
报错

如果可能存在  
零状态，报可  
能被0整除

- 软件质量
- 代码审查
- 代码检查的开源解决方案
  - 1. Clang介绍
  - 2. Clang-tidy
  - 3. Clang Static Analyzer
  - 4. SonarQube

# SonarQube

- SonarQube是免费的代码质量管理平台。
  - 内置了一系列C/C++代码度量工具，包括认知复杂度
  - 应用在CI/CD流程中，可以与Jenkins互通
  - 商业软件CppDepend、Coverity都提供了与SonarQube对接的插件
  - sonar-cxx是SonarQube平台的开源插件
    - 解析静态和动态代码分析工具的结果
      - **cppcheck**、**clang/gcc**告警、**VS**告警、**VS core guideline checker**、**clang static analyzer**、**clang tidy**、**Infer**、**PC-lint**、**RATS**、**Valgrind**、**Vera++**、**Dr. Memory**
    - 测试框架
      - **XUnit file format**、**Google Test**、**Boost Test ...**
    - Coverage
      - **Visual Studio coverage**、**Bullseye**、**gcov / gcovr**、**Testwell CTC++**

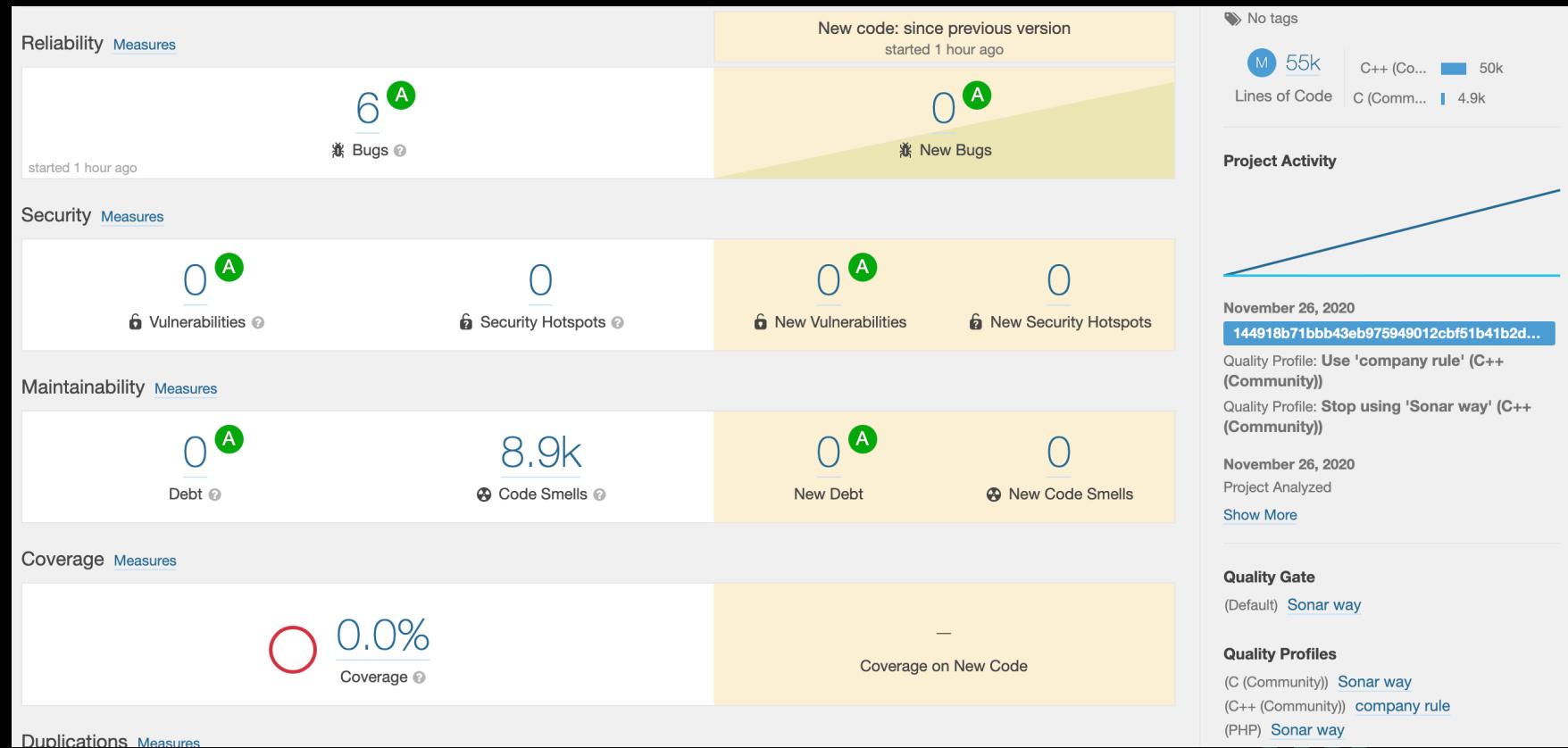
The screenshot shows the SonarQube Quality Gate report. It has a green header bar with the text "Quality Gate Passed" and "All conditions passed". Below this, there are three sections: RELIABILITY (0 Bugs), SECURITY (0 Vulnerabilities, 1 Hotspots), and MAINTAINABILITY (4 Code Smells, 5 Debt min). A central callout box highlights a major bug found in the code, which is a nullable pointer exception.

```

246     if (Provider.class == roleTypeClass) {
247         Type providedType = ReflectionUtils.getLastTypeGenericArgument(dependencyDescriptor);
248         Class providedClass = 1 ReflectionUtils.getTypeClass(providedType);
249
250         if (this.componentManager.hasComponent(providedType, dependencyDescriptor.
251             || 3 providedClass.isAssignableFrom(List.class) || providedClass.isA
252             continue;
253         }
  
```

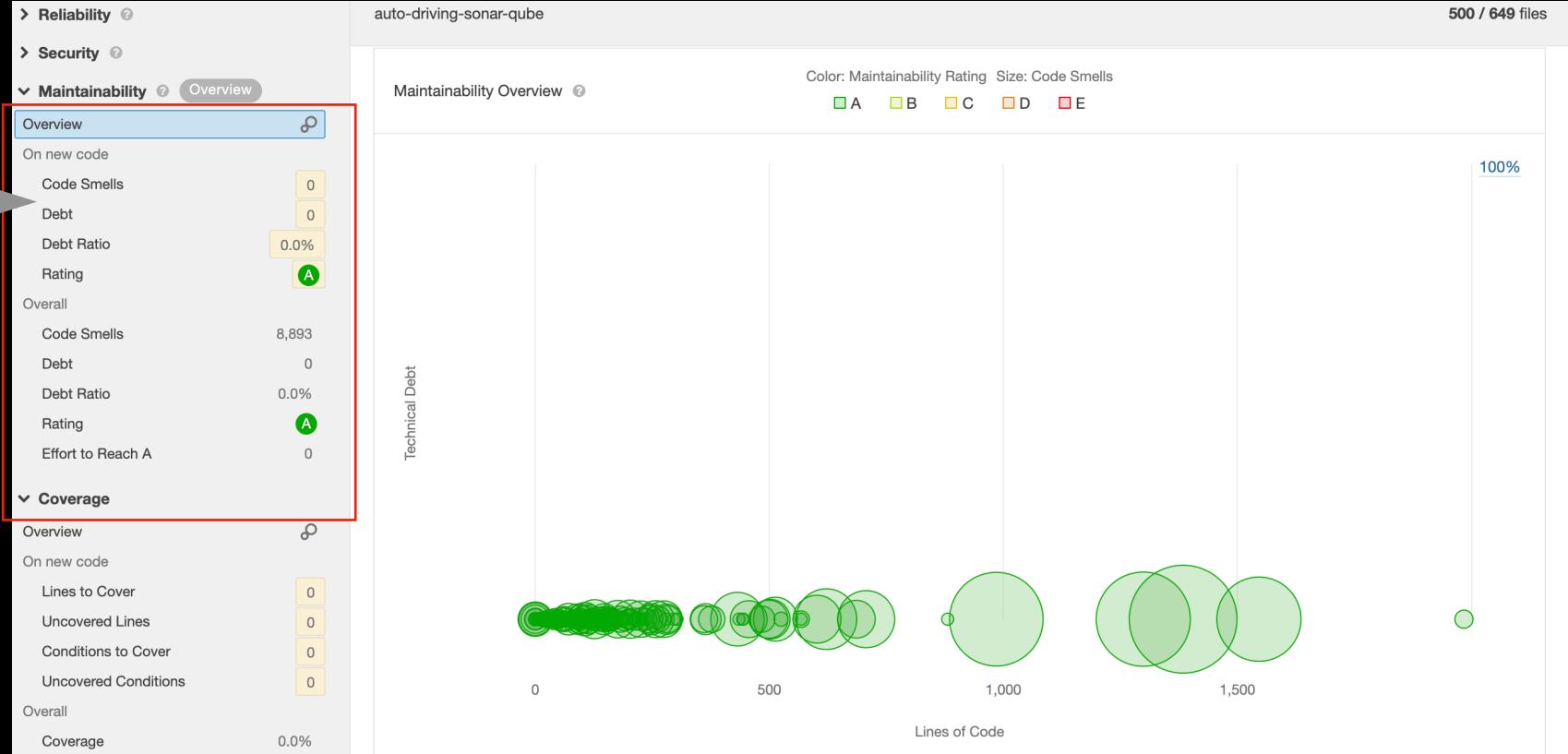
A "NullPointerException" could be thrown; "providedClass" is nullable here.  
Bug Major cert, cwe

# 项目概况

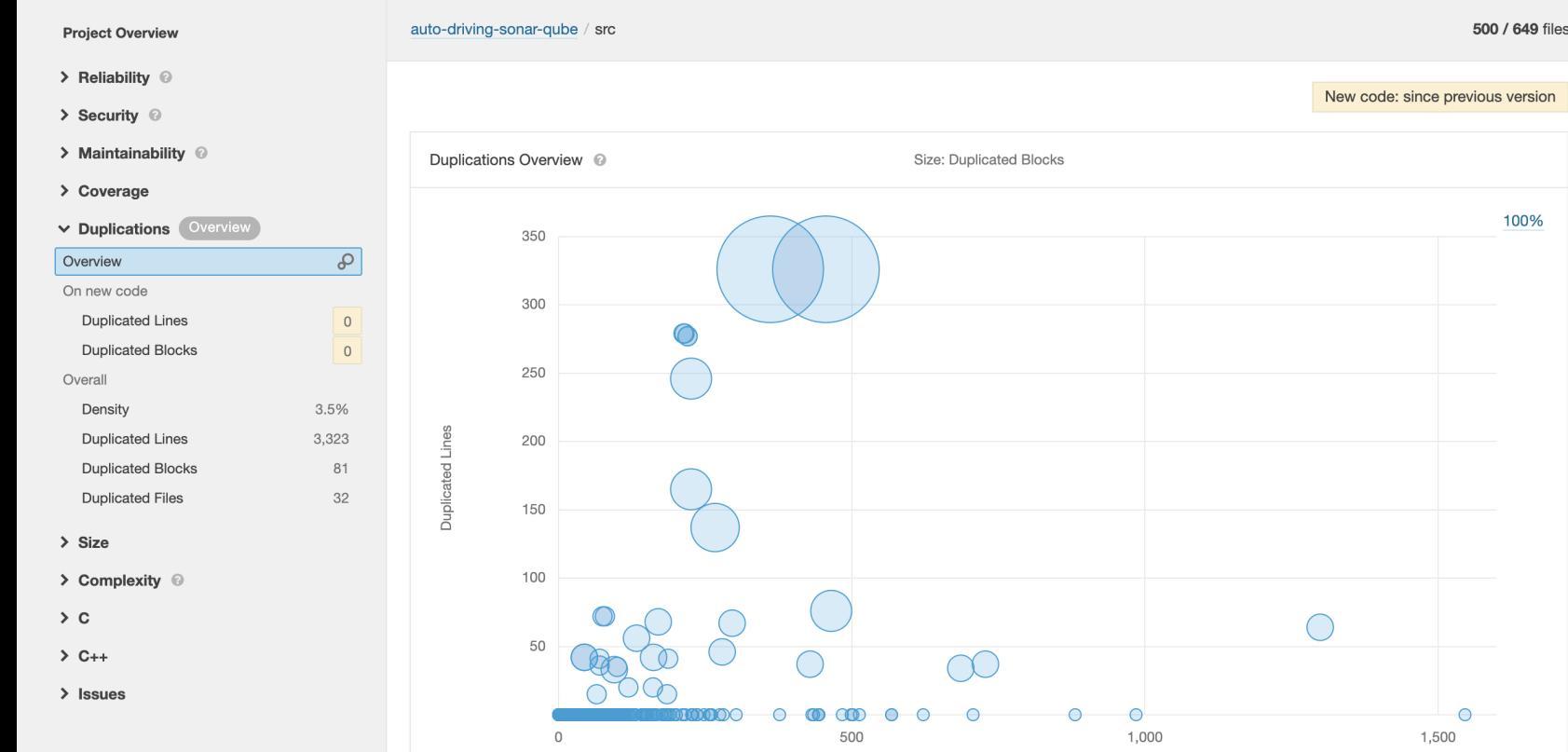


# 技术债管理

来自cppcheck、  
clang tidy/CSA的  
检查



# 重复代码



# 代码大小和代码复杂度

## 代码大小

Size	Classes
New Lines	0
Lines of Code	54,607
Lines	93,935
Statements	29,061
Functions	3,803
Classes	593
Files	649
Big Functions	247
Big Functions	39
Big Functions (%)	1.0%
Big Functions (%)	6.5%
Big Functions Lines of Code	2,163
Big Functions Lines of Code	9,821
Big Functions Lines of Code (%)	41.3%
Big Functions Lines of Code (%)	64.0%
Comment Lines	15,321
Comments (%)	21.9%
Lines of Code in Functions	23,774
Lines of Code in Functions	3,380

## 代码复杂度

Complexity	?
Cyclomatic Complexity	8,263
Cognitive Complexity	5,798
Complex Functions	61
Complex Functions	16
Complex Functions (%)	1.6%
Complex Functions (%)	0.4%
Complex Functions Lines of Code	1,466
Complex Functions Lines of Code	3,487
Complex Functions Lines of Co...	43.4%
Complex Functions Lines of Co...	14.7%

# Quality Gate

**Conditions** ?

Only project measures are checked against thresholds. Directories and files are ignored.

**Metric** ?

- Coverage on New Code
- Duplicated Lines on New Code
- Maintainability Rating on New Code
- Reliability Rating on New Code
- Security Rating on New Code

**RELIABILITY**

0 A Bugs

**SECURITY**

0 A Vulnerabilities      1 Hotspots

**MAINTAINABILITY**

4 Code Smells      5 A Debt  
min

**Quality Gate**  
**Passed**  
All conditions passed

Operator	Error
is less than	80.0%
is greater than	3.0%
is worse than	A
is worse than	A
is worse than	A

谢谢

Thank You

# 冉昕

Boolan资深咨询师



Boolan资深咨询师，曾在朗讯科技Lucent Technologies，摩根斯坦利 Morgan Stanley任职。长期从事嵌入式、通信行业，精于STL, Boost and C++ 11/14/17/20, llvm/clang, Linux kernel等，在高性能、低延迟、大规模系统级软件领域，拥有丰富的C/C++技术开发和架构经验，特别是软件架构、性能优化、编译构建等方面。

主办方：

**Boolan**  
高端IT咨询与教育平台

C++ Summit 2020

冉昕

Boolan资深咨询师

# 低延迟场景下的 性能优化实践

# Agenda

- 低延迟概述
- 低延迟系统调整
- 低延迟系统编译选项
- 低延迟软件设计与编码

## 低延迟场景

- 低延迟是第一需求
- 不追求吞吐量
- 不在意资源利用率
- 资源超配
- 案例：快速交易系统



# 低延迟优化特点

## 常用性能优化

- 压力测试
- 系统负载
  - CPU使用率
  - 内存占用
  - iowait
- Profile工具找出程序热点
- 优化热点

## 低延迟性能优化

- 系统、设计、编码需要提前考虑低延迟
- 提前规划好critical path
- 测试各单元延迟
- 优化critical path

# 常见操作时延

Operation	cpu cycle
Add, Sub, And, Or	< 1
memory write	≈ 1
"right" branch of "if"	≈ 1
Mul	3 - 6
L1 read	4
L2 read	10 - 12
"wrong" branch of "if"	10 - 20
L3 read	30 - 50
Div	20 - 100
Function call	25 - 250
Addition polymorphic function call	20 - 30
Mutex lock/unlock	50+
Main RAM read	100 - 150
NUMA: different socket L3 read	100 - 200
NUMA: different socket RAM read	200 - 300
Allocation deallocation pair	200+
User to kernel switch and back	300 - 500
Exception throw + caught	1000 - 2000
Context switch (direct cost)	2000
Context switch (total costs, including cache invalidation)	10K - 1M
Disk read	400K+

# Agenda

- 低延迟概述
- 低延迟系统调整
- 低延迟系统编译选项
- 低延迟软件设计与编码

## 硬件 & 系统

- 物理单机非集群，机器超配：
  - 单核频率高，核数有最低要求
  - X64处理器，执行效率越高越好，不需要虚拟化功能
  - 内存充足
  - NVMe SSD, Optane SSD
  - 低延迟网卡
  - 超频服务器
- 超线程
- 64位linux
  - 最小化安装
  - toolchain升级
  - Rtkernel
  - Tunning with vendor guide

# CPU相关优化

- Critical 线程 vs 普通线程
- 根据数量isolate core

/boot/grub2/grub.cfg

```
menuentry 'CentOS Linux (3.10.0-1160.6.1.el7.x86_64) 7 (Core)' --class centos ...
```

.....

```
linux16 /boot/vmlinuz-3.10.0-1127.13.1.el7.x86_64 root=UUID=..... isolcpus=3-7 nohz_full=3-7 rcu_nocbs=3-7
```

- Critical thread core binding

    sched\_setaffinity

- scheduler

- critical thread: FIFO

- sched\_setscheduler

- normal thread: default(CFS)

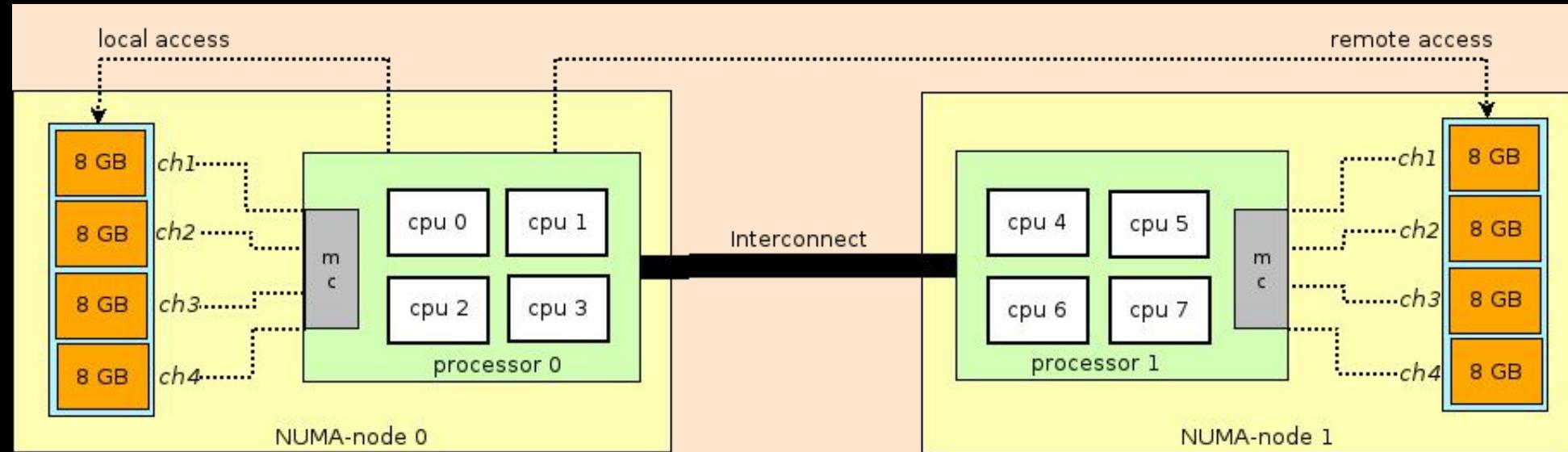
# NUMA

- 考慮NUMA

lscpu

NUMA node0 CPU(s): 0-3

NUMA node1 CPU(s): 4-7



\*Picture quoted from Boost.Fiber doc

## 中断

- irqbalance
  - systemctl disable irqbalance.service
- 设置中断affinity到非isolate核心
  - cat /proc/interrupts
    - network 及其他中断      /proc/irq/\*/smp\_affinity\_list
    - LOC                        nohz\_full
    - RES                        rcu\_nocbs
    - workqueue                /sys/devices/virtual/workqueue/\*/cpumask

# 中断

- 网卡绑定numa

```
lspci | grep -i "eth"
```

```
02:00.0 Ethernet controller: Intel Corporation I350 Gigabit Network Connection (rev 01)
```

```
cat /sys/bus/pci/devices/0000\02\00.0/numa_node
```

```
0
```

# 内存

- major fault vs minor fault
- 禁用swap
- mlock
- huge page
  - TLB
- NUMA
  - localalloc
- prefault
- 内存管理器 (ptmalloc, tcmalloc, jemalloc)

# 网络

- UDP
- TCP, 关闭Nagle和延迟确认
- 带宽越大越好
- DPDK, infiniband, RoCE
- 低延迟网卡
  - 支持kernel bypass
- FPGA

# Agenda

- 低延迟概述
- 低延迟系统调整
- 低延迟系统编译选项
- 低延迟软件设计与编码

## 编译器选择

- gcc, clang, icc
- O2 vs O3
  - -finline-functions (included in O2 since gcc10)
  - -floop-interchange
  - -funswitch-loops
  - -ftree-loop-distribution
  - -ftree-loop-distribute-patterns (included in O2 since gcc10)
  - -ftree-loop-vectorize, -ftree-slp-vectorize (clang O2)
  - -floop-unroll-and-jam
  - -fipa-cp-clone
  - void \_\_attribute\_\_((optimize("O3"))) foo() { // .. }
  - #pragma GCC optimize ("O3")

# -floop-interchange

```
for (int j = 0; j < 1024; j++) {  
    for(int i = 0; i < 1024; i++) {  
        a[i][j] = i * j;  
    }  
}
```

```
for (int i = 0; i < 1024; i++) {  
    for(int j = 0; j < 1024; j++) {  
        a[j][i] = i * j;  
    }  
}
```



```
for (int i = 0; i < 1024; i++) {  
    for(int j = 0; j < 1024; j++) {  
        a[i][j] = i * j;  
    }  
}
```

# -funswitch-loops

```
for (int i = 0; i < 1024 * 1024; ++i) {
    if (a > 0) {
        result += foo();
    }
    else {
        result += bar();
    }
}
```



```
if (a > 0) {
    for (int i = 0; i < 1024 * 1024; ++i) {
        result += foo();
    }
}
else {
    for (int i = 0; i < 1024 * 1024; ++i) {
        result += bar();
    }
}
```

# loop distribution

```
int a[length];
int b[length];

for (int i = 0; i < length; ++i) {
    a[i] = b[i];
    b[i] = 0;
}
```



```
int a[length];
int b[length];

for (int i = 0; i < length; ++i) {
    a[i] = b[i];
}

for (int i = 0; i < length; ++i) {
    b[i] = 0;
} Ⓡ
```



```
int a[length];
int b[length];

memcpy(a, b, length);
memset(a, 0, length);
```

# -ftree-loop-vectorize

Why is processing a sorted array faster than processing an unsorted array?

gcc optimization flag -O3 makes code slower than -O2

- gcc O1 & O2: cmov
- clang O1: cmov
- clang O2: sse2
- gcc -ftree-loop-vectorize: sse2
- -march

```
const unsigned arraySize = 32768;
int data[arraySize];

for (unsigned c = 0; c < arraySize; ++c)
    data[c] = std::rand() % 256;
// std::sort(data, data + arraySize);

clock_t start = clock();
long long sum = 0;

for (unsigned i = 0; i < 100000; ++i) {
    for (unsigned c = 0; c < arraySize; ++c) {
        if (data[c] >= 128)
            sum += data[c]; → sum += data[c] + data[c];
    }
}
double elapsedTime =
    static_cast<double>(clock() - start) / CLOCKS_PER_SEC;
```

## 编译选项

- O3 vs Ofast
  - -ffast-math
- Profile-Guided Optimisations
  - -fprofile-generatge, -fprofile-use & -fprofile-correction
  - -funroll-loops (clang O2)  

```
#pragma GCC unroll n
```
- -march=native
- -flto, also smaller binary size
- 其他编译选项
- irace

# loop-vectorize

```

double data[arraySize];

for (unsigned c = 0; c < arraySize; ++c)
    data[c] = std::rand() % 256;

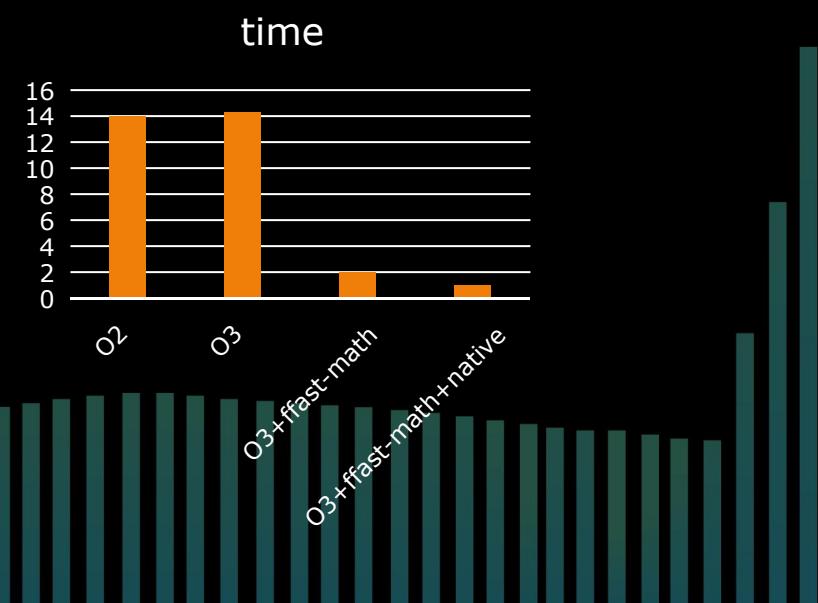
// std::sort(data, data + arraySize);

clock_t start = clock();
double sum = 0;

for (unsigned i = 0; i < 100000; ++i) {
    for (unsigned c = 0; c < arraySize; ++c) {
        if (data[c] >= 128)
            sum += data[c] + data[c];
    }
}

```

- floating运算不满足结合律
- -ffast-math
  - -funsafe-math-optimizations
  - -fassociative-math
  - -fno-signed-zeros
  - -fno-trapping-math
- 精度问题



# Agenda

- 低延迟概述
- 低延迟系统调整
- 低延迟系统编译选项
- 低延迟软件设计与编码

# 低延迟系统设计与编码

- 单进程多线程 > 多进程
- 提前创建线程
- 线程池
- 静态链接 > 动态链接
- 减少数据拷贝
- 减少数据共享

# 低延迟系统设计与编码

- 提前计算
- 增量计算
- 尽可能少的间接层
- 性能开销优先于系统的灵活可扩展
- 第三方库

# 运行时多态 vs 编译时多态

- vptr, vtable
- inline
- CRTP
- Policy based class design
- traits
- if constexpr
- SFINAE/enable\_if
- 仅有一个实现 (-fdevirtualize -fdevirtualize-speculatively)
  - 依然有vtable比较
- RTTI

# 编译时多态

```
class BaseStrategy {  
public:  
    virtual ~BaseStrategy();  
    virtual void OnTick(...);  
    .....  
private:  
    OMTYPE* _om;  
};
```

```
class ConcreteStrategy : public BaseStrategy {  
public:  
    void OnTick(...) override;  
    .....  
};
```

```
template <typename Derived, template <typename>  
typename OMTYPE = OM::OrderManager, bool Critical = true>  
class BaseStrategy : public OMTYPE<Derived> {  
public:  
    void OnTick(...) {  
        static_cast<Derived*>(this)->OnTickImpl(...);  
    }  
    .....  
};
```

```
class ConcreteStrategy : public BaseStrategy<  
ConcreteStrategy> {  
public:  
    void OnTickImpl(...);  
    .....  
};
```

# 编译时多态

```

template <typename T>
concept ControlTraits_ = requires {
    typename T::TraderApi;
    typename T::Spi;
    { T::FTDC_FCC_NonForceClose } -> std::convertible_to<char>;
    { T::FTDC_OPT_LimitPrice } -> std::convertible_to<char>;
    .....
};

template <typename Derived, ControlTraits_ Traits>
class CommonControl {
    CommonControl() {
        if constexpr (std::is_same_v<Traits,
            TradeInterface::Sgit::SgitControlTraits>){
            .....
        }
        typename Traits::TraderApi* _traderApi;
        typename Traits::Spi _spi;
    };
};

```

```

template <typename Derived, template <typename> typename OMType, bool Critical>
template <bool B>
inline typename std::enable_if_t<B> BaseStrategy<Derived, OMType, Critical>::AddMessage(const Common::TickMsg& msg) {
    _tickQueue.write(msg);
}

template <typename Derived, template <typename> typename OMType, bool Critical>
template <bool B>
inline typename std::enable_if_t<!B> BaseStrategy<Derived, OMType, Critical>::AddMessage(const Common::TickMsg& msg) {
    _tickQueue.write(msg);
    std::unique_lock lock(_lock);
    _cv.notify_all();
}

```

# 系统调用 & 日志

- 尽量避免系统调用
  - vdso支持的可以考虑排除
  - strace
- 谨慎打日志
  - 尽量避免打印cache外数据
  - format开销
  - 获取时间开销
    - time
    - clock\_gettime
    - rdtsc
      - constant\_tsc, nonstop\_tsc
  - 低开销日志库
    - 异步打印
    - 离线format

# 动态内存分配

- 尽量减少动态内存分配
  - 系统调用(可能) + page fault
  - placement new
  - memory pool
  - STL及第三方库带来的内存分配
  - 提前分配内存
    - std::array
    - ring buffer
    - vector
    - hash
      - pre add
      - 链式 vs 线性探测
    - map/set
      - 数量少用sorted array替代
  - pool allocator

# | ptmalloc 调优

- mallopt
  - M\_arena\_max
  - M\_arena\_test
  - M\_trim\_threshold
  - M\_mmap\_max
  - M\_mmap\_threshold

# vector 提前分配空间

```

const int num = 1024 * 1024;

std::vector<int> alloc() {
    std::vector<int> vec;
    vec.resize(num);
    for (size_t i = 0; i < vec.size(); i += 1024) {
        vec[i] = 0;
    }
    vec.clear();
    return vec;
}

void use(std::vector<int>& vec) {
    for (size_t i = 0; i < num; ++i) {
        vec.push_back(i);
    }
}

```

```

void* operator new(size_t n) {
    void* p = malloc(n);
    std::cout << "Allocating " << n << " bytes at "
              << p << "\n";
    return p;
}

void operator delete(void* p) {
    std::cout << "Free " << p << "\n";
    free(p);
}

perf stat -e minor-faults ./test

```

# | string

- SSO (gcc: 15, clang: 22, msvc: 15)
- char array, string\_view

```
std::string string8() {
    return s("12345678");
}
.......
```

```
static void String8(benchmark::State& state) {
    std::string str;
    for (auto _ : state)
        benchmark::DoNotOptimize(str = string8());
}
BENCHMARK(String8);
.......
```

# string sso

Benchmark	Time	CPU	Iterations
String6	3.45 ns	3.45 ns	202897921
String7	4.07 ns	4.07 ns	172152608
String8	2.82 ns	2.82 ns	248720421
String9	3.47 ns	3.46 ns	201992970
String10	3.55 ns	3.55 ns	197528242
String11	4.10 ns	4.10 ns	170848552
String12	3.45 ns	3.45 ns	203180117
String13	4.09 ns	4.08 ns	171481778
String14	4.07 ns	4.07 ns	171863748
String15	4.69 ns	4.69 ns	149172560
String16	17.6 ns	17.5 ns	39915270
String17	17.7 ns	17.7 ns	39468941

# string sso

```
string7[abi:cxx11]():
```

```
    leaq  16(%rdi), %rdx
    movb $55, 22(%rdi)
    movq  %rdi, %rax
    movq  %rdx, (%rdi)
    movl  $13877, %edx
    movl  $875770417, 16(%rdi)
    movw  %dx, 20(%rdi)
    movq  $7, 8(%rdi)
    movb  $0, 23(%rdi)
    ret
```

```
string8[abi:cxx11]():
```

```
    leaq  16(%rdi), %rdx
    movq  $8, 8(%rdi)
    movq  %rdi, %rax
    movabsq $4050765991979987505, %rcx
    movq  %rdx, (%rdi)
    movq  %rcx, 16(%rdi)
    movb  $0, 24(%rdi)
    ret
```

```
string9[abi:cxx11]():
```

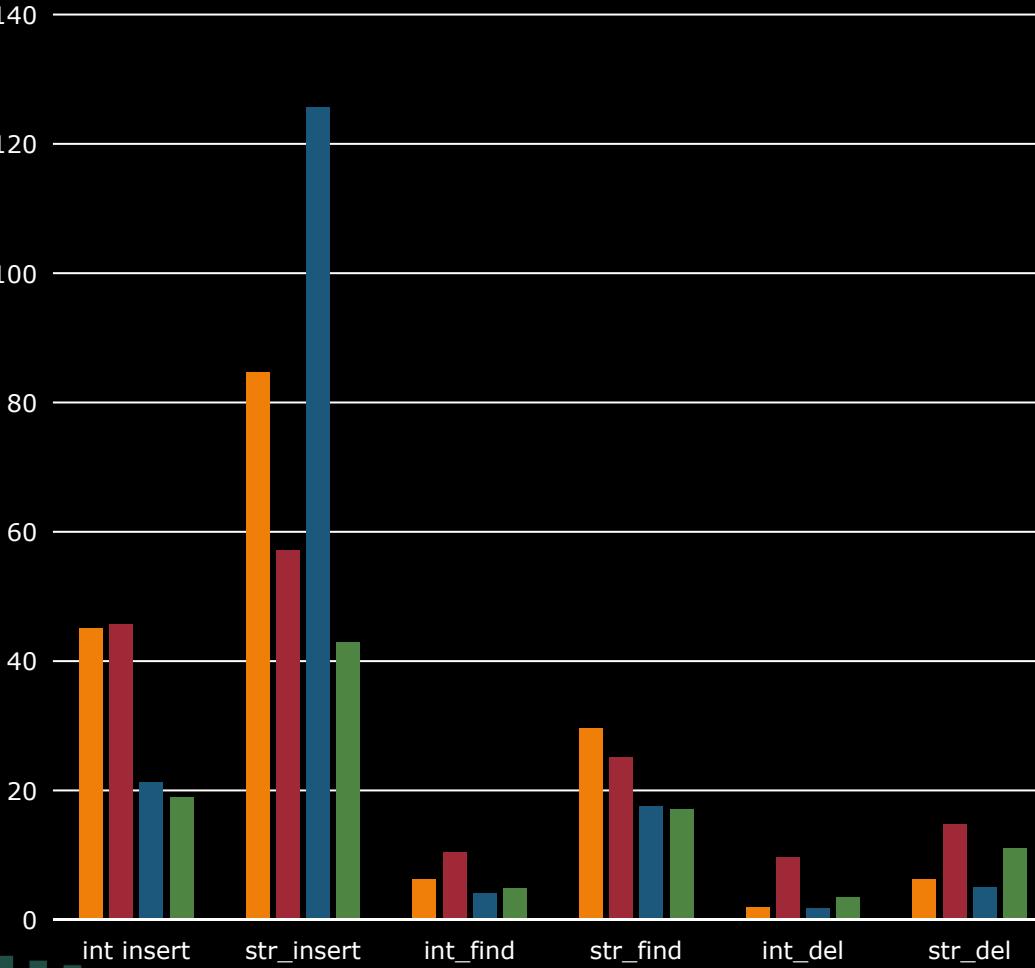
```
    leaq  16(%rdi), %rdx
    movb $57, 24(%rdi)
    movq  %rdi, %rax
    movabsq $4050765991979987505, %rcx
    movq  %rdx, (%rdi)
    movq  %rcx, 16(%rdi)
    movq  $9, 8(%rdi)
    movb  $0, 25(%rdi)
    ret
```

```
string16[abi:cxx11]():
```

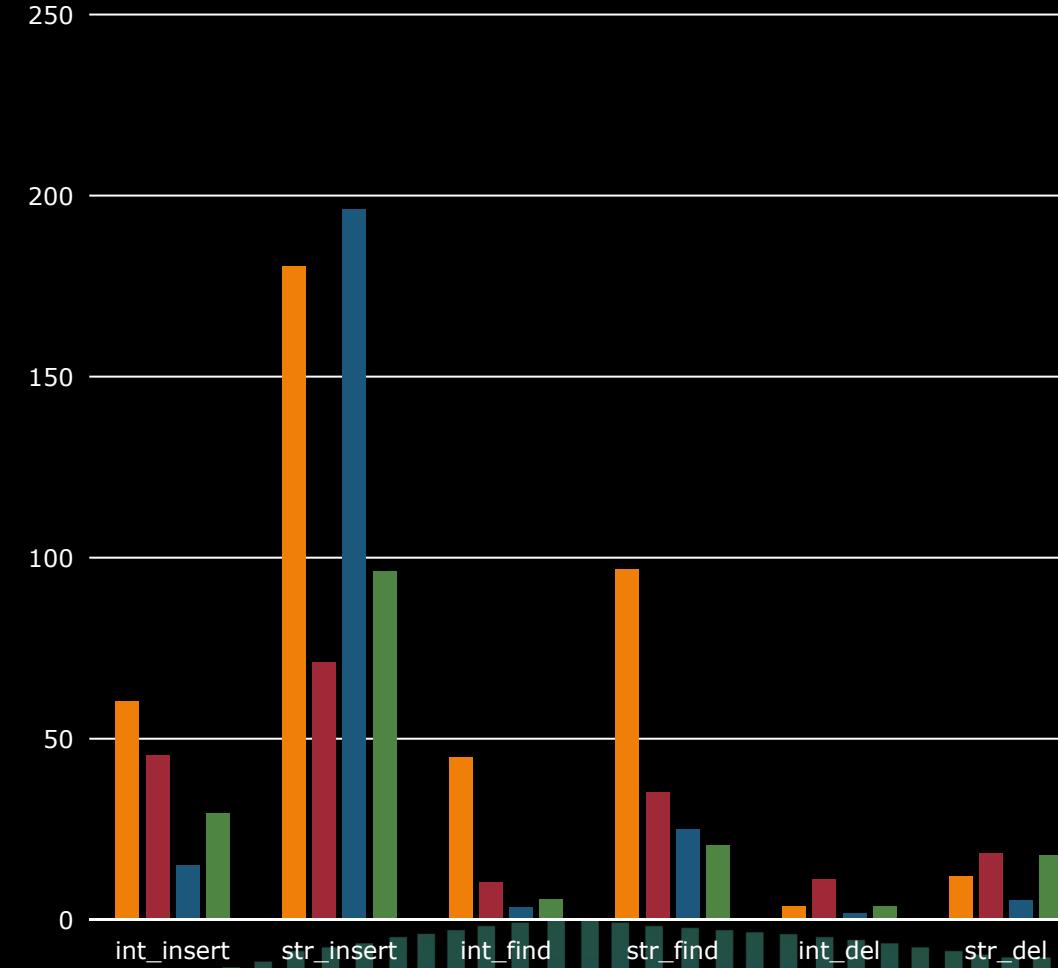
```
    pushq  %r12
    leaq   16(%rdi), %rax
    xorl  %edx, %edx
    movq  %rdi, %r12
    subq  $16, %rsp
    movq  %rax, (%rdi)
    movq  $16, 8(%rsp)
    leaq   8(%rsp), %rsi
    call   std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<ch
ar> >::__M_create(unsigned long&, unsigned long)
    movq  8(%rsp), %rdx
    movdqa .LC0(%rip), %xmm0
    movq  %rax, (%r12)
    movq  %rdx, 16(%r12)
    movups %xmm0, (%rax)
    movq  8(%rsp), %rax
    movq  (%r12), %rdx
    movq  %rax, 8(%r12)
    movb  $0, (%rdx,%rax)
    addq  $16, %rsp
    movq  %r12, %rax
    popq  %r12
    ret
```

# hashmap

50 elements



3000 elements

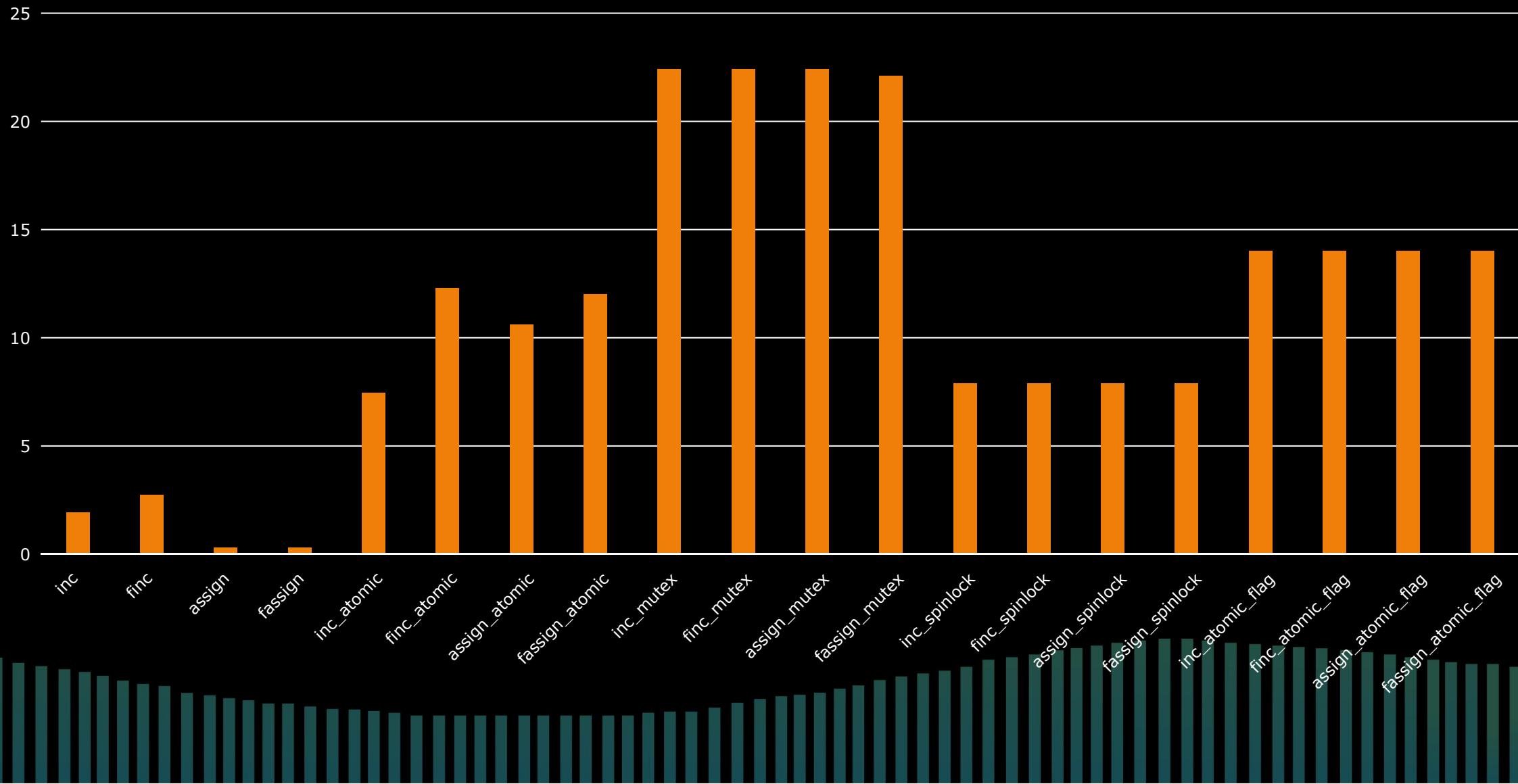


# 无锁编程

- 消息传递
- lockfree queue
  - spsc
  - mpsc
  - mpmc
- atomic
- atomic\_flag
- compare\_and\_exchange
- spinlock > mutex/semaphore
  - 内核态
  - futex

# 同步开销

time



## 分支处理

- 代码尽可能少用branch
- cmov
- setcc
- sse, avx
- lookup table
- jump table

```
extern void func0();
extern void func1();
// func2, func3, ...

void (*F[])() = { func1, func2, ..... };

void test(int a) {
    switch (a) {
        case 0:
            func0();
            break;
        case 1:
            func1();
            break;
        // .....
    }
}

void test2(int a) {
    F[a]();
}
```

# 分支处理

```
// probability: condA > condB > condC

if (condA()) {
    return;
}

..... // sectionA

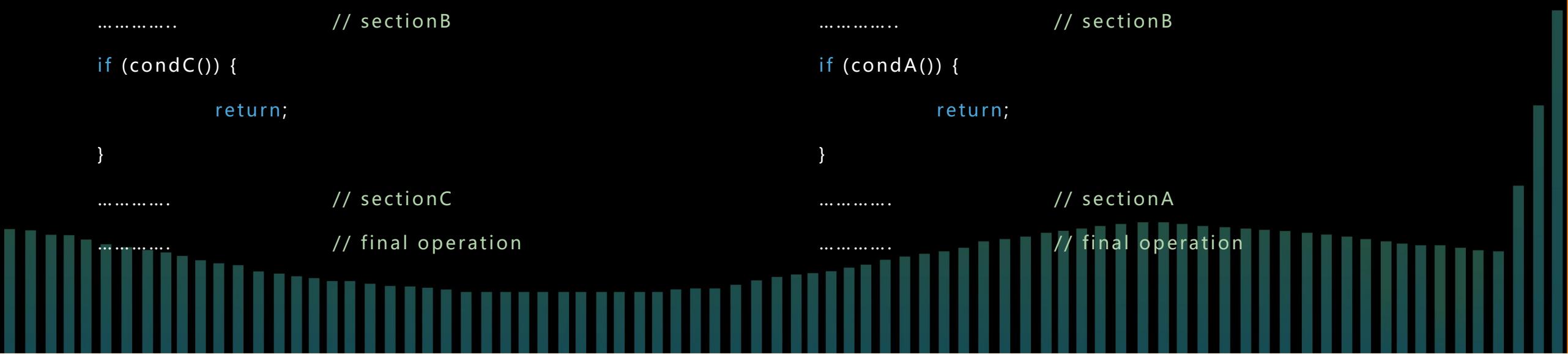
if (condB()) {
    return;
}

..... // sectionB

if (condC()) {
    return;
}

..... // sectionC

// final operation
```



# 分支处理

```
if (condA()) {  
    return;  
}  
..... // sectionA ..... // sectionA  
if (condB()) {  
    return;  
}  
..... // sectionB ..... // sectionB  
if (condC()) {  
    return;  
}  
..... // sectionC ..... // sectionC  
..... // final operation ..... // final operaton
```

# branch prediction

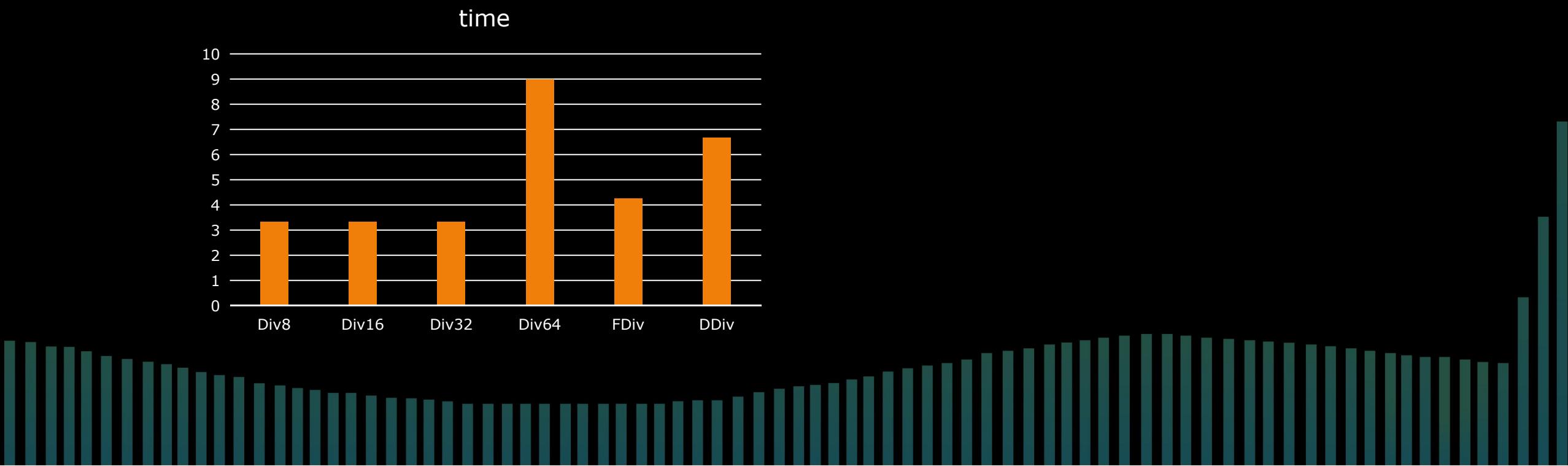
- `__builtin_expect (!!x, 1), __builtn_expect (!!x, 0)`
- `[[likely]], [[unlikely]]`
- reference for compiler
- static vs dynamic
- take expected branch with fake flag

# cache优化

- 减少多线程数据写
- alignas(64)
- Padding, 避免false sharing
- struct data arrangement
- static, global数据访问
- 代码组织
- conflict miss
- prefetch
- cache warming

## 运算开销

- Integral除法开销大，尤其是64位
- floating除法优于int64\_t
- float vs double
- Packed vs Scalar



# 类型转换

- signed/unsigned
- Integral: int64\_t, long, int, short, char
- float <-> integral
- string <-> integral

```
template <typename F, typename T>
__attribute__((noinline))
T convert(F f)
{
    return static_cast<T>(f);
}
```

```
int convert<signed char, int>(signed char):
    movsbl %dl, %eax
    ret
int convert<short, int>(short):
    movswl %di, %eax
    ret
signed char convert<int, signed char>(int):
    movl %edi, %eax
    ret
short convert<int, short>(int):
    movl %edi, %eax
    ret
long convert<signed char, long>(signed char):
    movsbq %dl, %rax
    ret
long convert<short, long>(short):
    movswq %di, %rax
    ret
long convert<int, long>(int):
    movslq %edi, %rax
    ret
int convert<long, int>(long):
    movq %rdi, %rax
    ret
```

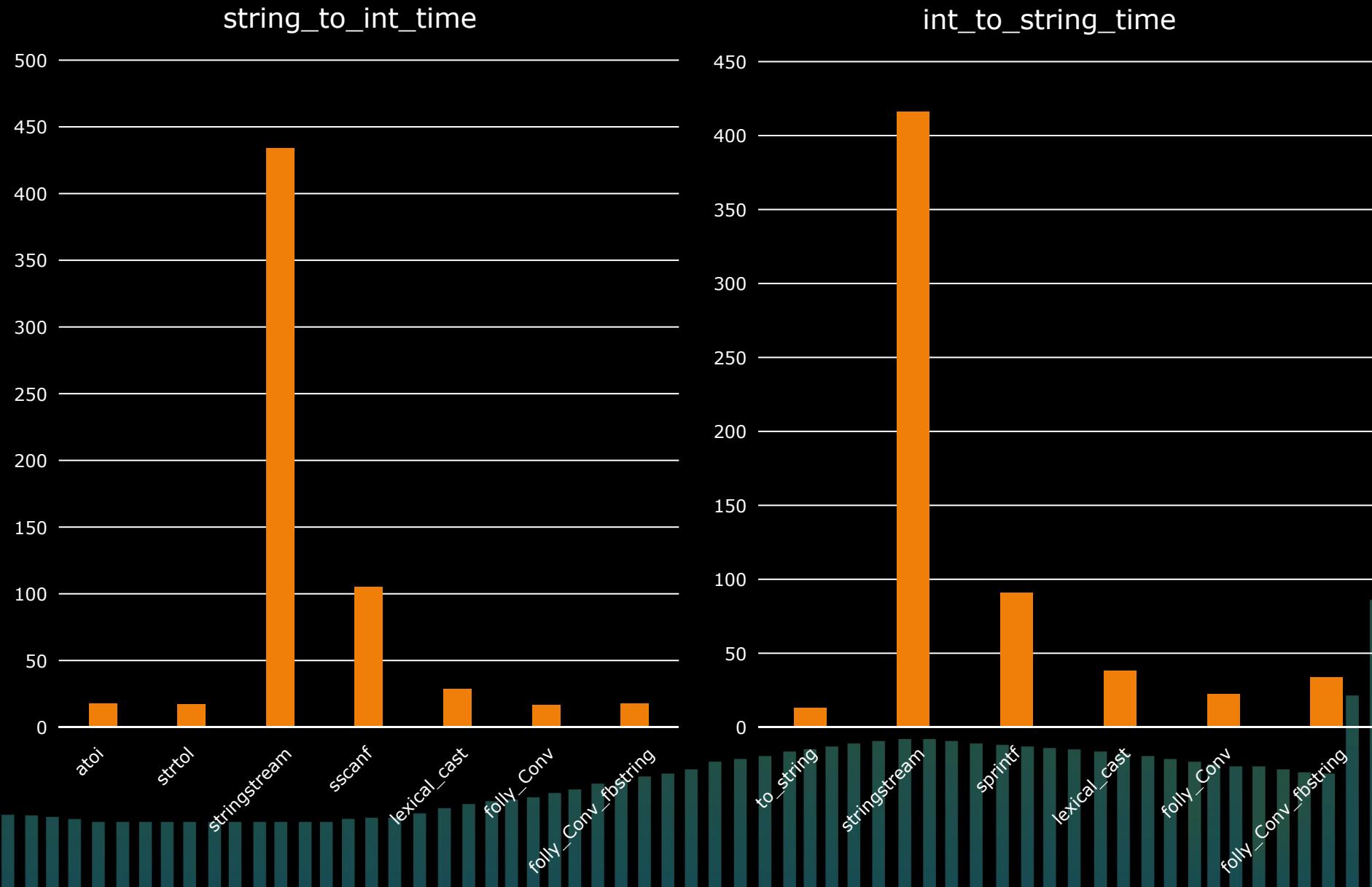
# 类型转换

```
int convert<float, int>(float):
    cvttss2sil    %xmm0, %eax
    ret
int convert<double, int>(double):
    cvttsd2sil    %xmm0, %eax
    ret
float convert<int, float>(int):
    pxor    %xmm0, %xmm0
    cvtsi2ssl    %edi, %xmm0
    ret
double convert<int, double>(int):
    pxor   %xmm0, %xmm0
    cvtsi2sdl    %edi, %xmm0
    ret
float convert<long, float>(long):
    pxor   %xmm0, %xmm0
    cvtsi2ssq    %rdi, %xmm0
    ret
double convert<long, double>(long):
    pxor   %xmm0, %xmm0
    cvtsi2sdq    %rdi, %xmm0
    ret
long convert<float, long>(float):
    cvttss2siq    %xmm0, %rax
    ret
long convert<double, long>(double):
    cvttsd2siq    %xmm0, %rax
    ret
```

- Intel® Core™ Processor instruction throughput and latency  
<https://software.intel.com/content/www/us/en/develop/download/10th-generation-intel-core-processor-instruction-throughput-and-latency-docs.html>
- Instruction tables  
[https://www.agner.org/optimize/instruction\\_tables.ods](https://www.agner.org/optimize/instruction_tables.ods)

## 类型转换

- atoi
- strtol, to\_string
- stringstream
- sscanf, sprintf
- lexical\_cast
- folly Conv



# 低延迟系统设计与编码

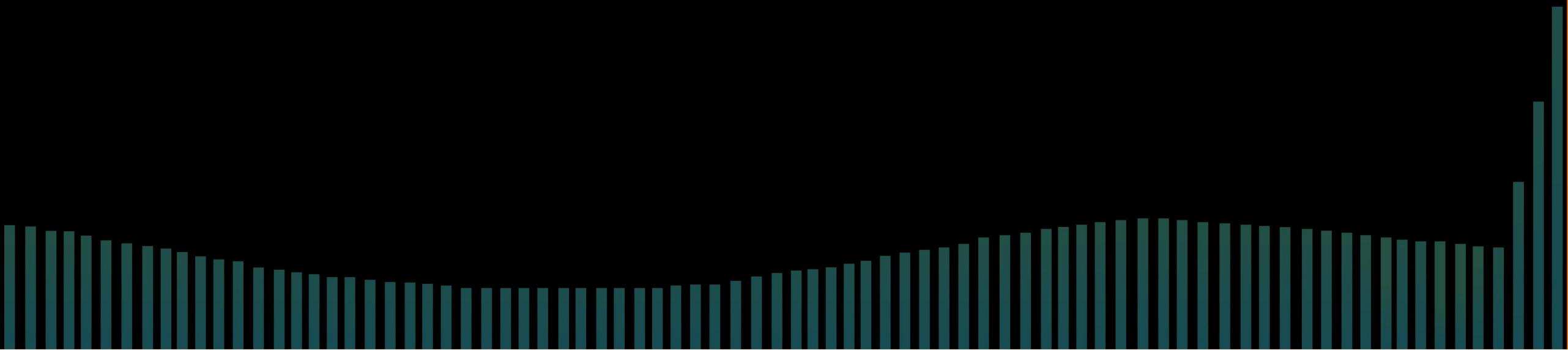
- 注意异常开销
  - 编译器打开异常选项
  - 正常路径不应该用异常
  - 不触发几乎没有开销
  - noexcept
- 作用域尽可能小
- 尽可能使用const
  - 有利于编译器优化
- 无连接 > 内连接 > 外连接
  - static
  - 匿名namespace
  - const
  - inline
  - -fipa-icf

# 低延迟系统设计与编码

- 智能指针
  - unique\_ptr
  - shared\_ptr
- C++20
  - atomic\_float
  - atomic\_ref
  - [[likely]], [[unlikely]]
  - consteval, constinit
  - atomic shared\_ptr

# 2020

# THANK YOU





# Michael Wong

C++ 嵌入式开发委员会SG14主席, Codeplay研发副总裁

C++ 嵌入式开发委员会SG14与机器学习委员会SG19主席, 同时担任C++语言方向演化委员会主席, Codeplay研发副总裁, C++标准委员会加拿大代表团团长。Michael在C++并行计算、高性能计算、机器学习领域拥有丰富工作经验, 他领导制订了应用于GPU应用开发C++异构编程语言(SYCL)标准.对Tensorflow底层性能优化有着深刻的研究和见解。其具体工作涵盖并行编程、神经网络、计算机视觉、自动驾驶等领域。Michael 曾任 IBM 高级技术专家, 领导 IBM XL C++ 编译器、XL C 编译器的开发工作。

主办方:

**Boolan**  
高端IT咨询与教育平台



# C++ 20 parallelism and concurrency features with heterogenous programming using SYCL and oneAPI

Michael Wong  
Codeplay Software Ltd.  
Distinguished Engineer

## Products

### Acoran

Integrates all the industry standard technologies needed to support a very wide range of AI and HPC

### ComputeAorta™

The heart of Codeplay's compute technology enabling OpenCL™, SPIR-V™, HSA™ and Vulkan™

## Company

Leaders in enabling high-performance software solutions for new AI processing systems

Enabling the toughest processors with tools and middleware based on open standards

Established 2002 in Scotland with ~80 employees

### ComputeCpp™

C++ platform via the SYCL™ open standard, enabling vision & machine learning e.g. TensorFlow™



High Performance Compute (HPC)  
Automotive ADAS, IoT, Cloud Compute  
Smartphones & Tablets  
Medical & Industrial

Technologies: Artificial Intelligence  
Vision Processing  
Machine Learning  
Big Data Compute

## Customers

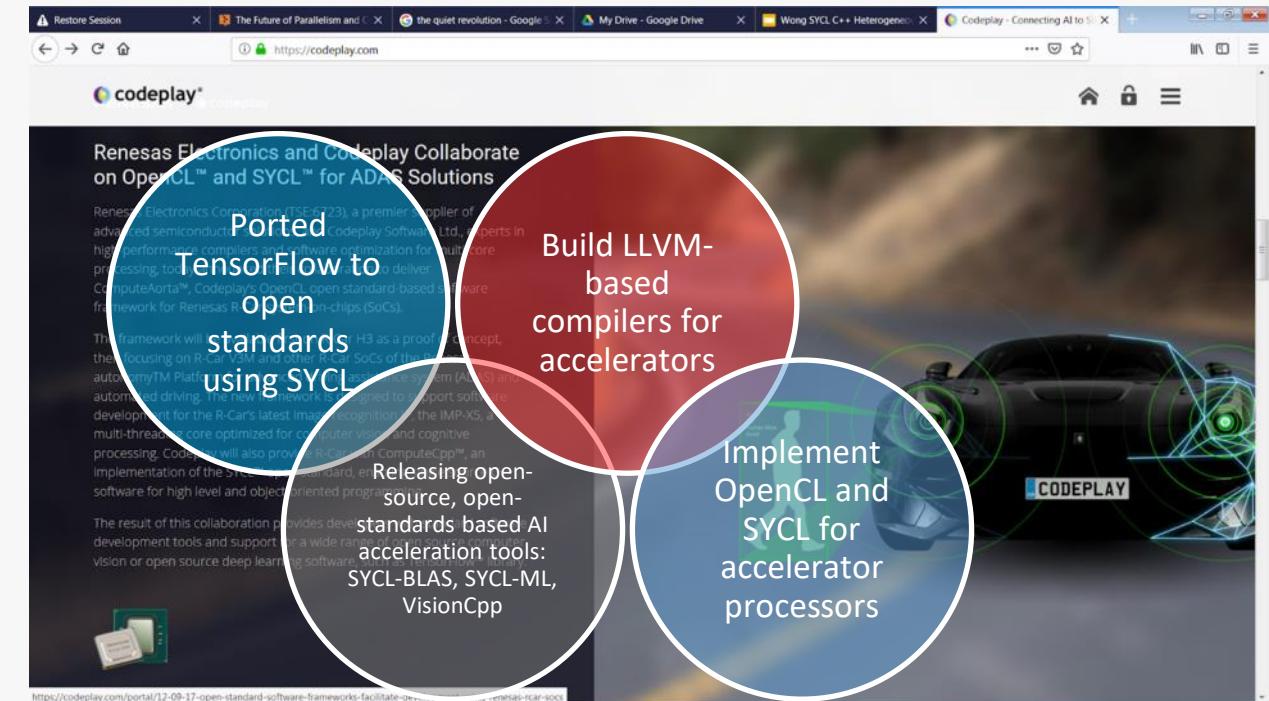


And many more!

# Distinguished Engineer

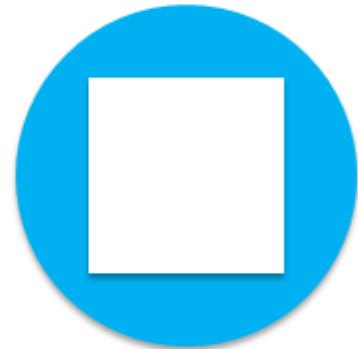
- Chair of SYCL Heterogeneous Programming Language
- C++ Directions Group
- ISOCPP.org Director, VP  
<http://isocpp.org/wiki/faq/wg21#michael-wong>
- Head of Delegation for C++ Standard for Canada
- Chair of Programming Languages for Standards Council of Canada
- Chair of WG21 SG19 Machine Learning
- Chair of WG21 SG14 Games Dev/Low Latency/Financial Trading/Embedded
- Editor: C++ SG5 Transactional Memory Technical Specification
- Editor: C++ SG1 Concurrency Technical Specification
- MISRA C++ and AUTOSAR
- Chair of Standards Council Canada TC22/SC32 Electrical and electronic components (SOTIF)
- Chair of UL4600 Object Tracking

# Michael Wong



We build GPU compilers for semiconductor companies  
**Now working to make AI/ML heterogeneous acceleration safe for autonomous vehicle**

# Acknowledgement and Disclaimer



THIS WORK REPRESENTS THE  
VIEW OF THE AUTHOR AND DOES  
NOT NECESSARILY REPRESENT  
THE VIEW OF CODEPLAY.

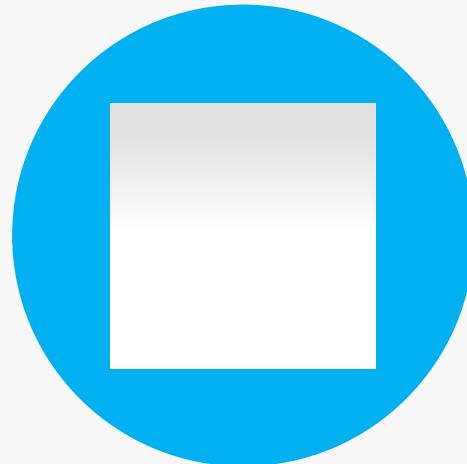


OTHER COMPANY, PRODUCT, AND  
SERVICE NAMES MAY BE  
TRADEMARKS OR SERVICE MARKS  
OF OTHERS.

Numerous people internal and external to the original C++/Khronos group/OpenMP, in industry and academia, have made contributions, influenced ideas, written part of this presentations, and offered feedbacks to form part of this talk. These include Bjarne Stroustrup, Joe Hummel, Botond Ballo, Simon McIntosh-Smith, as well as many others.

But I claim all credit for errors, and stupid mistakes. **These are mine, all mine! You can't have them.**

# Legal Disclaimer



THIS WORK REPRESENTS THE VIEW OF THE  
AUTHOR AND DOES NOT NECESSARILY  
REPRESENT THE VIEW OF CODEPLAY.



OTHER COMPANY, PRODUCT, AND SERVICE  
NAMES MAY BE TRADEMARKS OR SERVICE  
MARKS OF OTHERS.

## Disclaimers

NVIDIA, the NVIDIA logo and CUDA are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and/or other countries

Codeplay is not associated with NVIDIA for this work and it is purely using public documentation and widely available code

# 3 Act Play

1. What excites me?
2. Parallelism and  
Concurrency in C++  
20
3. SYCL and C++ future  
Heterogeneity



# Act 1

What gets me up every morning?



# A tale of two cities

PPL

stands for

Parallel Patterns Library



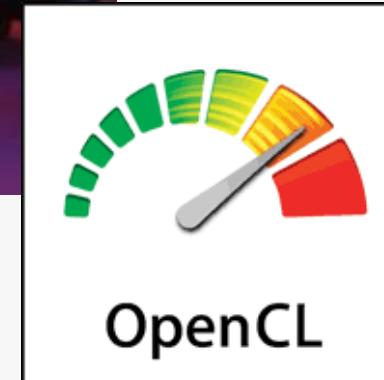
Abbreviations.com

Offloading C++ for Multi-core Processor Parallelism



O'REILLY®

James Reinders  
Reviewed by Alexander Repenning





OH, East is East, and West is West,  
and never the twain shall meet...

-Rudyard Kipling

OpenCL

OpenMP

SYCL

CUDA



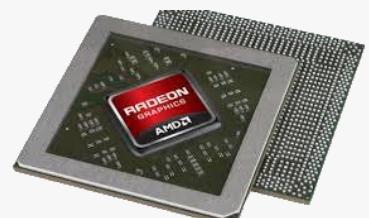
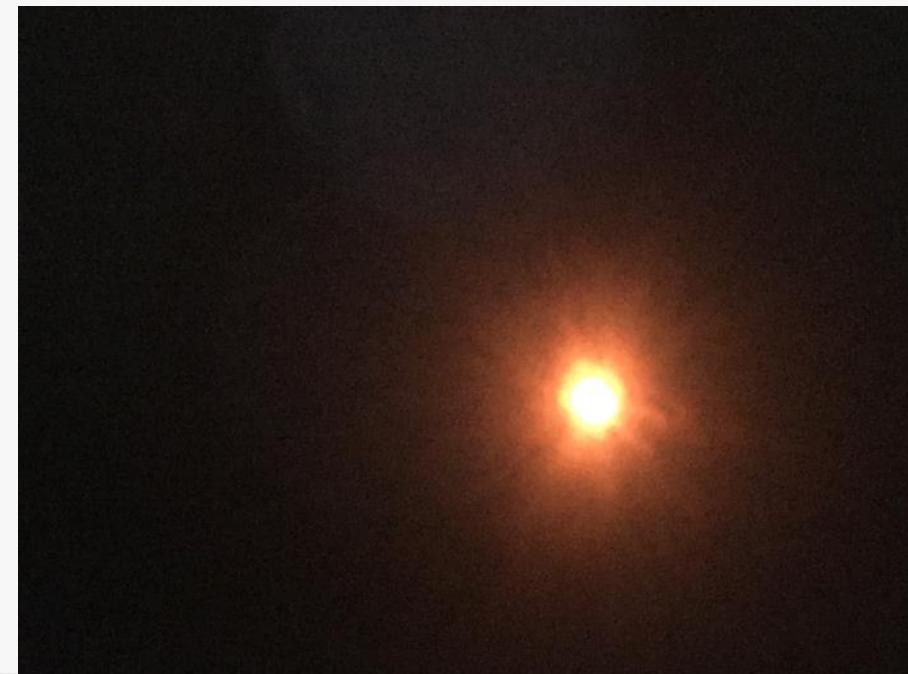
Kokkos

HPX

Raja

Boost.Compute

# The Quiet Revolution



# Parallel/concurrency before C++11 (C++98)

	Asynchronous Agents	Concurrent collections	Mutable shared state	Heterogeneous (GPUs, accelerators, FPGA, embedded AI processors)
summary	tasks that run independently and communicate via messages	operations on groups of things, exploit parallelism in data and algorithm structures	avoid races and synchronizing objects in shared memory	Dispatch/offload to other nodes (including distributed)
examples	GUI, background printing, disk/net access	trees, quicksorts, compilation	locked data(99%), lock-free libraries (wizards), atomics (experts)	Pipelines, reactive programming, offload,, target, dispatch
key metrics	responsiveness	throughput, many core scalability	race free, lock free	Independent forward progress,, load-shared
requirement	isolation, messages	low overhead	composability	Distributed, heterogeneous
today's abstractions	POSIX threads, win32 threads, OpenCL, vendor intrinsic	openmp, TBB, PPL, OpenCL, vendor intrinsic	locks, lock hierarchies, vendor atomic instructions, vendor intrinsic	OpenCL, CUDA

# Parallel/concurrency after C++11

	Asynchronous Agents	Concurrent collections	Mutable shared state	Heterogeneous (GPUs, accelerators, FPGA, embedded AI processors)
summary	tasks that run independently and communicate via messages	operations on groups of things, exploit parallelism in data and algorithm structures	avoid races and synchronizing objects in shared memory	Dispatch/offload to other nodes (including distributed)
examples	GUI,background printing, disk/net access	trees, quicksorts, compilation	locked data(99%), lock-free libraries (wizards), atomics (experts)	Pipelines, reactive programming, offload,, target, dispatch
key metrics	responsiveness	throughput, many core scalability	race free, lock free	Independent forward progress,, load-shared
requirement	isolation, messages	low overhead	composability	Distributed, heterogeneous
today's abstractions	C++11: thread,lambda function, TLS, Async	C++11: Async, packaged tasks, promises, futures, atomics	C++11: locks, memory model, mutex, condition variable, atomics, static init/term	C++11: lambda

# Parallel/concurrency after C++14

	Asynchronous Agents	Concurrent collections	Mutable shared state	Heterogeneous
summary	tasks that run independently and communicate via messages	operations on groups of things, exploit parallelism in data and algorithm structures	avoid races and synchronizing objects in shared memory	Dispatch/offload to other nodes (including distributed)
examples	GUI,background printing, disk/net access	trees, quicksorts, compilation	locked data(99%), lock-free libraries (wizards), atomics (experts)	Pipelines, reactive programming, offload,, target, dispatch
key metrics	responsiveness	throughput, many core scalability	race free, lock free	Independent forward progress,, load-shared
requirement	isolation, messages	low overhead	composability	Distributed, heterogeneous
today's abstractions	C++11: thread,lambda function, TLS, async  <b>C++14: generic lambda</b>	C++11: Async, packaged tasks, promises, futures, atomics,	C++11: locks, memory model, mutex, condition variable, atomics, static init/term,  <b>C++ 14:</b> <b>shared_lock/shared_timed_mutex, OOTA,</b>  <b>atomic_signal_fence,</b>	C++11: lambda  <b>C++14: none</b>

# Parallel/concurrency after C++17

	Asynchronous Agents	Concurrent collections	Mutable shared state	Heterogeneous (GPUs, accelerators, FPGA, embedded AI processors)
summary	tasks that run independently and communicate via messages	operations on groups of things, exploit parallelism in data and algorithm structures	avoid races and synchronizing objects in shared memory	Dispatch/offload to other nodes (including distributed)
today's abstractions	C++11: thread, lambda function, TLS, async  C++14: generic lambda	C++11: Async, packaged tasks, promises, futures, atomics,  <b>C++ 17: ParallelSTL, control false sharing</b>	C++11: locks, memory model, mutex, condition variable, atomics, static init/term,  C++ 14: shared_lock/shared_timed_mutex, OOTA, atomic_signal_fence,  <b>C++ 17: scoped_lock, shared_mutex, ordering of memory models, progress guarantees, TOE, execution policies</b>	C++11: lambda  C++14: generic lambda  <b>C++17: progress guarantees, TOE, execution policies</b>

# Act 2

## Parallelism and Concurrency in C++ 20



# Concurrency vs Parallelism

**What makes parallel or concurrent programming harder than serial programming? What's the difference? How much of this is simply a new mindset one has to adopt?**



# Parallel/concurrency for C++11, 14, 17, C++20

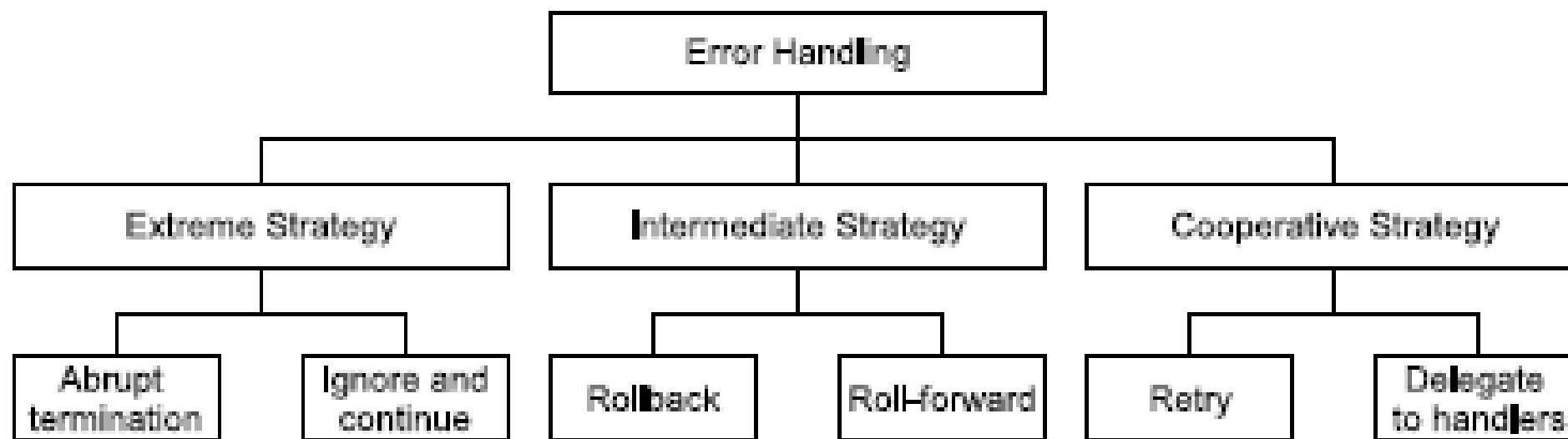
CPP-Summit

	Asynchronous Agents	Parallel collections	Mutable shared state	Heterogeneous/Distributed
<b>abstractions from C++11, 14, 17, 20</b>	<p>C++11: thread,lambda function, TLS, async</p> <p><b>C++ 20: Jthreads +interrupt _token, coroutines</b></p>	<p>C++11: packaged tasks, promises, futures,</p> <p>C++ 17: ParallelSTL, control false sharing</p> <p><b>C++20 : Vec execution policy, Algorithm unsequenced policy</b></p>	<p>C++11: locks, memory model, mutex, condition variable, atomics, static init/term,</p> <p>C++ 14: shared_lock/shared_timed_mutex, OOTA, atomic_signal_fence, C++ 17: scoped_lock, shared_mutex, ordering of memory models, progress guarantees, TOE, execution policies</p> <p><b>C++20: atomic_ref, Latches and barriers, atomic&lt;shared_ptr&gt;</b></p> <p><b>Atomics &amp; padding bits</b></p> <p><b>Simplified atomic init</b></p> <p><b>Atomic C/C++ compatibility</b></p> <p><b>Semaphores and waiting</b></p> <p><b>Fixed gaps in memory model , Improved atomic flags, Repair</b></p> <p><b>memory model</b></p>	<p>C++11: lambda</p> <p>C++14: generic lambda</p> <p>C++17: , progress guarantees, TOE, execution policies</p> <p><b>C++20: atomic_ref</b></p>

# C++20 asynchronous, concurrency, parallelism, heterogeneous programming

- cooperative cancellation of threads
- new synchronization facilities
- updates to atomics
- coroutines

# Error handling classification



# Violence is NEVER the answer

	1. Kill	2. Tell, don't take no for an answer	3. Ask politely, and accept rejection	4. Set flag politely, let it poll if it wants
Tagline	Shoot first, check invariants later	Fire him, but let him clean out his desk	Tap him on the shoulder	Send him an email
Summary	A time-honored way to randomly corrupt your state and achieve undefined behavior	Interrupt at well-defined points and allow a handler chain (but target can't refuse or stop)	Interrupt at well-defined points and allow a handler chain, but request can be ignored	Target actively checks a flag – can be manual, or provided as part of #2 or #3
Pthreads	<code>pthread_kill</code> <code>pthread_cancel (async)</code>	<code>pthread_cancel (deferred mode)</code>	<i>n/a</i>	Manual
Java	<code>Thread.destroy</code> <code>Thread.stop</code>	<i>n/a</i>	<code>Thread.interrupt</code>	Manual, or <code>Thread.interrupted</code>
.NET	<code>Thread.Abort</code>	<i>n/a</i>	<code>Thread.Interrupt</code>	Manual, or <code>Sleep(0)</code>
C++0x	<i>n/a</i>	<i>n/a</i>	<i>n/a</i>	Manual
Guidance	Avoid, almost certain to corrupt transaction(s)	OK for languages without exceptions and unwinding...	Good, conveniently automated	Good, but requires more cooperative effort (can be a plus!)

# Interrupt politely

- Cooperative thread cancellation

4. Set flag politely,  
let it poll if it wants

Send him an email

Target actively checks  
a flag – can be  
manual, or provided  
as part of #2 or #3

Manual

Manual, or  
`Thread.interrupted`

Manual, or `Sleep(0)`

Manual

Good, but requires  
more cooperative  
effort (can be a plus!)

# Stop\_token

**n4835.pdf - Adobe Acrobat Pro Extended**

File Edit View Document Comments Forms Tools Advanced Window Help

Create Combine Collaborate Secure Sign Forms Multimedia Comment

Bookmarks

- 18 Concepts library
- 19 Diagnostics library
- 20 General utilities library
- 21 Strings library
- 22 Containers library
- 23 Iterators library
- 24 Ranges library
- 25 Algorithms library
- 26 Numerics library
- 27 Time library
- 28 Localization library
- 29 Input/output library
- 30 Regular expressions library
- 31 Atomic operations library
- 32 Thread support library
  - 32.1 General
  - 32.2 Requirements
  - 32.3 Stop tokens**
  - 32.4 Threads
  - 32.5 Mutual exclusion
  - 32.6 Condition variables
  - 32.7 Semaphore
  - 32.8 Coordination types
    - 32.8.1 Latches
    - 32.8.2 Barriers
  - 32.9 Futures
- A Grammar summary
- B Implementation quantities
- C Compatibility
- D Compatibility features
  - Bibliography
  - Cross references
  - Cross references from ISO C++ 2017
- Index
- Index of grammar productions
- Index of library headers
- Index of library names
- Index of library concepts

176% Find

**32.3.3 Class stop\_token** [stoptoken]

1 The class `stop_token` provides an interface for querying whether a stop request has been made (`stop_requested`) or can ever be made (`stop_possible`) using an associated `stop_source` object (32.3.4). A `stop_token` can also be passed to a `stop_callback` (32.3.5) constructor to register a callback to be called when a stop request has been made from an associated `stop_source`.

```
namespace std {
    class stop_token {
public:
    // 32.3.3.1, constructors, copy, and assignment
    stop_token() noexcept;

    stop_token(const stop_token&) noexcept;
    stop_token(stop_token&&) noexcept;
    stop_token& operator=(const stop_token&) noexcept;
    stop_token& operator=(stop_token&&) noexcept;
    ~stop_token();
    void swap(stop_token&) noexcept;

    // 32.3.3.2, stop handling
    [[nodiscard]] bool stop_requested() const noexcept;
    [[nodiscard]] bool stop_possible() const noexcept;

    [[nodiscard]] friend bool operator==(const stop_token& lhs, const stop_token& rhs) noexcept;
    [[nodiscard]] friend bool operator!=(const stop_token& lhs, const stop_token& rhs) noexcept;
    friend void swap(stop_token& lhs, stop_token& rhs) noexcept;
};

}
```

**32.3.3.1 Constructors, copy, and assignment** [stoptoken.cons]

# C++ Cooperative cancellation

- std::stop\_source and std::stop\_token to handle cooperative cancellation.
- the target task needs to check
- std::condition\_variable\_away so the wait can be interrupted by a stop request
- use std::stop\_callback to provide your own cancellation mechanism.

```
Data read_file(  
    std::stop_token st,  
    std::filesystem::path filename  
{  
    auto  
        handle=open_file(filename);  
    std::stop_callback cb(  
        st,[=]{ cancel_io(handle);});  
    return read_data(handle); //  
    blocking  
}
```

# Work flow

1. Create a std::stop\_source
2. In your background task, Obtain a std::stop\_token from the std::stop\_source
3. Pass the std::stop\_token to a new thread or task
4. When you want the background operation to stop call source.request\_stop()
5. Periodically Background task call token.stop\_requested() to check

# jthread

n4835.pdf - Adobe Acrobat Pro Extended  
 File Edit View Document Comments Forms Tools Advanced Window Help  
 Create Combine Collaborate Secure Sign Forms Multimedia Comment  
 Bookmarks  
 32.2 Exceptions  
 32.2.3 Native handles  
 32.2.4 Timing specifications  
 32.2.5 Requirements for Cpp17Lockable types  
 32.3 Stop tokens  
 32.3.1 Introduction  
 32.3.2 Header <stop\_token> synopsis  
 32.3.3 Class stop\_token  
 32.3.4 Class stop\_source  
 32.3.5 Class template stop\_callback  
 32.4 Threads  
 32.4.1 Header <thread> synopsis  
 32.4.2 Class thread  
 32.4.3 Class jthread  
 32.4.4 Namespace this\_thread  
 32.5 Mutual exclusion  
 32.6 Condition variables  
 32.7 Semaphore  
 32.8 Coordination types  
 32.8.1 Latches  
 32.8.2 Barriers  
 32.9 Futures  
 32.9.1 Overview  
 32.9.2 Header <future> synopsis  
 32.9.3 Error handling  
 32.9.4 Class future\_error  
 32.9.5 Shared state  
 32.9.6 Class template promise  
 32.9.7 Class template future  
 32.9.8 Class template shared\_future  
 32.9.9 Function template async  
 32.9.10 Class template

Effects: As if by `x.swap(y)`.

### 32.4.3 Class `jthread`

[`thread.jthread.class`]

The class `jthread` provides a mechanism to create a new thread of execution. The functionality is the same as for class `thread` (32.4.2) with the additional abilities to provide a `stop_token` (32.3) to the new thread of execution, make stop requests, and automatically join.

```
namespace std {
    class jthread {
public:
    // types
    using id = thread::id;
    using native_handle_type = thread::native_handle_type;

    // 32.4.3.1, constructors, move, and assignment
    jthread() noexcept;
    template<class F, class... Args> explicit jthread(F&& f, Args&&... args);
    ~jthread();
    jthread(const jthread&) = delete;
    jthread(jthread&&) noexcept;
    jthread& operator=(const jthread&) = delete;
    jthread& operator=(jthread&&) noexcept;

    // 32.4.3.2, members
    void swap(jthread&) noexcept;
    [[nodiscard]] bool joinable() const noexcept;
    void join();
    void detach();
    [[nodiscard]] id get_id() const noexcept;
    [[nodiscard]] native_handle_type native_handle(); // see 32.2.3

    // 32.4.3.3, stop token handling
    [[nodiscard]] stop_source get_stop_source() noexcept;
    [[nodiscard]] stop_token get_stop_token() const noexcept;
    bool request_stop() noexcept;

    // 32.4.3.4, specialized algorithms
    friend void swap(jthread& lhs, jthread& rhs) noexcept;

    // 32.4.3.5, static members
    [[nodiscard]] static unsigned int hardware_concurrency() noexcept;

private:
    stop_source ssouce; // exposition only
};
```

# jthread is an RAII Style thread

- std::jthread integrates with std::stop\_token to support cooperative cancellation.
- Destroying a std::jthread calls source.request\_stop() and thread.join().
- The thread needs to check the stop token passed in to the thread function.
- Stop\_token is passed as a first parameter
- Backwards compatible: If you don't support having a stop\_token, it would not stop.

```
void thread_func(  
    std::stop_token st,  
    std::string arg1,int arg2){  
    while(!st.stop_requested()){  
        do_stuff(arg1,arg2);  
    }  
}  
  
void foo(std::string s){  
    std::jthread t(thread_func,s,42);  
    do_stuff();  
} // destructor requests stop and  
  joins
```

# New Synchronization

- Latches
- Barriers
- Semaphores

# Latch

n4835.pdf - Adobe Acrobat Pro Extended  
File Edit View Document Comments Forms Tools Advanced Window Help  
Create Combine Collaborate Secure Sign Forms Multimedia Comment  
Bookmarks  
18 Concepts library  
19 Diagnostics library  
20 General utilities library  
21 Strings library  
22 Containers library  
23 Iterators library  
24 Ranges library  
25 Algorithms library  
26 Numerics library  
27 Time library  
28 Localization library  
29 Input/output library  
30 Regular expressions library  
31 Atomic operations library  
32 Thread support library  
  32.1 General  
  32.2 Requirements  
  32.3 Stop tokens  
  32.4 Threads  
  32.5 Mutual exclusion  
  32.6 Condition variables  
  32.7 Semaphore  
  32.8 Coordination types  
    32.8.1 Latches  
    32.8.2 Barriers  
  32.9 Futures  
A Grammar summary  
B Implementation quantities  
C Compatibility  
D Compatibility features  
Bibliography  
Cross references  
Cross references from ISO C++ 2017  
Index  
Index of grammar productions  
Index of library headers  
Index of library names  
Index of library concepts  
150% Find [thread.latch.class]  
**32.8.1.2 Class latch** § 32.8.1.2 [thread.latch.class] 1571  

```
namespace std {
    class latch {
public:
    constexpr explicit latch(ptrdiff_t expected);

    ~latch();

    latch(const latch&) = delete;
    latch& operator=(const latch&) = delete;

    void count_down(ptrdiff_t update = 1);
    bool try_wait() const noexcept;
    void wait() const;
    void arrive_and_wait(ptrdiff_t update = 1);

private:
    ptrdiff_t counter; // exposition only
    };
}
```

© ISO/IEC N4835

1 A latch maintains an internal counter that is initialized when the latch is created. Threads can block on the latch object, waiting for counter to be decremented to zero.

# Latches are for single use

- std::latch is a single-use counter that allows threads to wait for the count to reach zero.
- Any waiting threads become unblocked and carry on

- 1 Create the latch with a non-zero count
- 2 One or more threads decrease the count
- 3 Other threads may wait for the latch to be signalled.
- 4 When the count reaches zero it is permanently signalled and all waiting threads are woken.

```
void foo(){
    unsigned const thread_count=...
    std::latch done(thread_count);
    my_data data[thread_count];
    std::vector<std::jthread> threads;
    for(unsigned i=0;i<thread_count;++i)
        threads.push_back(std::jthread{[&,i]{
            data[i]=make_data(i);
            done.count_down();
            do_more_stuff();
        }});
    done.wait();
    process_data();
}
```

# Using a latch is great for multithreaded

## tests

1. Set up the test data
2. Create a latch
3. Create the test threads: The first thing each thread does is `test_latch.arrive_and_wait()`
4. When all threads have reached the latch they are unblocked to run their code

# Barrier

n4835.pdf - Adobe Acrobat Pro Extended  
File Edit View Document Comments Forms Tools Advanced Window Help  
Create + Combine Collaborate Secure Sign Forms Multimedia Comment  
Bookmarks  
ISO/IEC  
N4835

}

### 32.8.2.2 Class template barrier

[thread.barrier.class]

```
namespace std {
    template<class CompletionFunction = see below>
    class barrier {
public:
    using arrival_token = see below;

    constexpr explicit barrier(ptrdiff_t expected,
                               CompletionFunction f = CompletionFunction());
    ~barrier();

    barrier(const barrier&) = delete;
    barrier& operator=(const barrier&) = delete;

    [[nodiscard]] arrival_token arrive(ptrdiff_t update = 1);
    void wait(arrival_token&& arrival) const;

    void arrive_and_wait();
    void arrive_and_drop();

private:
    CompletionFunction completion; // exposition only
    };
}
```

1 Each *barrier phase* consists of the following steps:

# Barriers is reusable for loop synchronization between parallel tasks

- std::barrier<> is a reusable barrier with completion function.
  - Barriers are great for loop synchronization between parallel tasks.
  - The **completion function** allows you to do something between loops: pass the result on to another step, write to a file, etc.
- i.** Synchronization is done in **phases**:
1. Construct a barrier, with a non-zero count and a **completion function**
  2. One or more threads arrive at the barrier
  3. These other threads wait for the barrier to be signalled
  4. When the count reaches zero, the barrier is signalled, the **completion function** is called on one of the thread and the count is reset

```
unsigned const num_threads=...;  
void finish_task();  
std::barrier<std::function<void()>> b(  
num_threads,finish_task);  
void worker_thread(  
    std::stop_token st,unsigned i){  
    while(!st.stop_requested()){  
        do_stuff(i);  
        b.arrive_and_wait();  
    }  
}
```

# Semaphore

n4835.pdf - Adobe Acrobat Pro Extended

File Edit View Document Comments Forms Tools Advanced Window Help

Create Combine Collaborate Secure Sign Forms Multimedia Comment

Bookmarks

- 21 Strings library
- 22 Containers library
- 23 Iterators library
- 24 Ranges library
- 25 Algorithms library
- 26 Numerics library
- 27 Time library
- 28 Localization library
- 29 Input/output library
- 30 Regular expressions library
- 31 Atomic operations library
- 32 Thread support library
  - 32.1 General
  - 32.2 Requirements
  - 32.3 Stop tokens
  - 32.4 Threads
  - 32.5 Mutual exclusion
  - 32.6 Condition variables
  - 32.7 Semaphore
    - 32.7.1 Header `<semaphore>` synopsis
    - 32.7.2 Class template `counting_semaphore`
  - 32.8 Coordination types
    - 32.8.1 Latches
    - 32.8.2 Barriers
  - 32.9 Futures
- A Grammar summary
- B Implementation quantities
- C Compatibility
- D Compatibility features
- Bibliography
- Cross references
- Cross references from ISO C++ 2017
- Index
- Index of grammar productions
- Index of library headers
- Index of library names
- Index of library concepts

176% Find

```
using binary_semaphore = counting_semaphore;
}
```

**32.7.2 Class template `counting_semaphore`** [thread.sema.cnt]

```
namespace std {
    template<ptrdiff_t least_max_value = implementation-defined>
    class counting_semaphore {
public:
    static constexpr ptrdiff_t max() noexcept;

    constexpr explicit counting_semaphore(ptrdiff_t desired);
    ~counting_semaphore();

    counting_semaphore(const counting_semaphore&) = delete;
    counting_semaphore& operator=(const counting_semaphore&) = delete;

    void release(ptrdiff_t update = 1);
    void acquire();
    bool try_acquire() noexcept;
    template<class Rep, class Period>
        bool try_acquire_for(const chrono::duration<Rep, Period>& rel_time);
    template<class Clock, class Duration>
        bool try_acquire_until(const chrono::time_point<Clock, Duration>& abs_time);

private:
    ptrdiff_t counter;           // exposition only
};
```

<sup>1</sup> Class template `counting_semaphore` maintains an internal counter that is initialized when the semaphore is created. The counter is decremented when a thread acquires the semaphore, and is incremented when a thread releases the semaphore. If a thread tries to acquire the semaphore when the counter is zero, the

# Semaphore is a very low level machine to build anything

- Manage a shared count
  - A semaphore represents a number of available “slots”. If you **acquire** a slot on the semaphore then the count is decreased until you **release** the slot.
  - Acquire decrease count, release increase count, instead of a countdown
  - Attempting to acquire a slot when the count is zero will either block or fail.
  - A thread may release a slot without acquiring one and vice versa.
  - Semaphores can be used to build just about any synchronization mechanism, including latches, barriers and mutexes.
  - A **binary semaphore** has 2 states: 1 slot free or no slots free.
  - It can be used as a mutex.
  - C++20 has `std::counting_semaphore<max_count>`
  - `std::binary_semaphore` is an alias for `std::counting_semaphore<1>`.
  - As well as **blocking** `sem.acquire()`, there are also `sem.try_acquire()`, `sem.try_acquire_for()` and `sem.try_acquire_until()` functions that fail instead of blocking
- ```
std::counting_semaphore<5> slots(5);
void func(){
    slots.acquire();
    do_stuff(); // at most 5 threads can be here
    slots.release();
}
```

# Atomics

- Low-level waiting for atomics
- Atomic Smart Pointers
- `std::atomic_ref`

# Atomic waiting and notifying

n4835.pdf - Adobe Acrobat Pro Extended  
 File Edit View Document Comments Forms Tools Advanced Window Help  
 Create Combine Collaborate Secure Sign Forms Multimedia Comment  
 176% Find

**Bookmarks**

- 21 Strings library
- 22 Containers library
- 23 Iterators library
- 24 Ranges library
- 25 Algorithms library
- 26 Numerics library
- 27 Time library
- 28 Localization library
- 29 Input/output library
- 30 Regular expressions library
- 31 Atomic operations library
  - 31.1 General
  - 31.2 Header <atomic> synopsis
  - 31.3 Type aliases
  - 31.4 Order and consistency
  - 31.5 Lock-free property
  - 31.6 Waiting and notifying**
  - 31.7 Class template atomic\_ref
  - 31.8 Class template atomic
  - 31.9 Non-member functions
  - 31.10 Flag type and operations
  - 31.11 Fences
- 32 Thread support library
  - 32.1 General
  - 32.2 Requirements
  - 32.3 Stop tokens
  - 32.4 Threads
  - 32.5 Mutual exclusion
  - 32.6 Condition variables
  - 32.7 Semaphore
    - 32.7.1 Header <semaphore> synopsis
    - 32.7.2 Class template counting\_semaphore
  - 32.8 Coordination types
    - 32.8.1 Latches
    - 32.8.2 Barriers

## 31.6 Waiting and notifying

[**atomics.wait**]

- 1 *Atomic waiting operations* and *atomic notifying operations* provide a mechanism to wait for the value of an atomic object to change more efficiently than can be achieved with polling. An atomic waiting operation may block until it is unblocked by an atomic notifying operation, according to each function's effects. [Note: Programs are not guaranteed to observe transient atomic values, an issue known as the A-B-A problem, resulting in continued blocking if a condition is only temporarily met. — *end note*]
- 2 [Note: The following functions are atomic waiting operations:
  - (2.1) — `atomic<T>::wait`,
  - (2.2) — `atomic_flag::wait`,
  - (2.3) — `atomic_wait` and `atomic_wait_explicit`,
  - (2.4) — `atomic_flag_wait` and `atomic_flag_wait_explicit`, and
  - (2.5) — `atomic_ref<T>::wait`.
 — *end note*]
- 3 [Note: The following functions are atomic notifying operations:
  - (3.1) — `atomic<T>::notify_one` and `atomic<T>::notify_all`,
  - (3.2) — `atomic_flag::notify_one` and `atomic_flag::notify_all`,
  - (3.3) — `atomic_notify_one` and `atomic_notify_all`,
  - (3.4) — `atomic_flag_notify_one` and `atomic_flag_notify_all`, and
  - (3.5) — `atomic_ref<T>::notify_one` and `atomic_ref<T>::notify_all`.
 — *end note*]
- 4 A call to an atomic waiting operation on an atomic object M is *eligible to be unblocked* by a call to an atomic notifying operation on M if there exist side effects X and Y on M such that:

# Waiting for atomics

- Replaces spin mutex with an atomic flag test and set with complicated waiting and exponential backoffs
- `std::atomic<T>` now provides a `var.wait()` member function to wait for it to change.
- `var.notify_one()` and `var.notify_all()` wake one or all threads blocked in `wait()`.
- Like a low level `std::condition_variable`.
- Possible implementations
  - Futex, conditional variables, contention tables, timed back-off, spinlock
- Also now have test method on atomic flag

# atomic<shared\_ptr<T>>

n4835.pdf - Adobe Acrobat Pro Extended  
File Edit View Document Comments Forms Tools Advanced Window Help  
Create Combine Collaborate Secure Sign Forms Multimedia Comment  
Bookmarks  
20.5 Topics  
20.6 Optional objects  
20.7 Variants  
20.8 Storage for any type  
20.9 Bitsets  
20.10 Memory  
20.11 Smart pointers  
20.11.1 Class template unique\_ptr  
20.11.2 Class bad\_weak\_ptr  
20.11.3 Class template shared\_ptr  
20.11.4 Class template weak\_ptr  
20.11.5 Class template owner\_less  
20.11.6 Class template enable\_shared\_from\_this  
20.11.7 Smart pointer hash support  
20.11.8 Atomic specializations for smart pointers  
20.11.8.1 Atomic specialization for shared\_ptr  
20.11.8.2 Atomic specialization for weak\_ptr  
20.12 Memory resources  
20.13 Class template scoped\_allocator\_adaptor  
20.14 Function objects  
20.15 Metaprogramming and type traits  
20.16 Compile-time rational arithmetic  
20.17 Class type\_index  
20.18 Execution policies  
20.19 Primitive numeric conversions  
20.20 Formatting  
21 Strings library

### 20.11.8.1 Atomic specialization for shared\_ptr [util.smartptr.atomic.shared]

```
namespace std {
    template<class T> struct atomic<shared_ptr<T>> {
        using value_type = shared_ptr<T>;
        static constexpr bool is_always_lock_free = implementation-defined;

        bool is_lock_free() const noexcept;
        void store(shared_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
        shared_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;
        operator shared_ptr<T>() const noexcept;

        shared_ptr<T> exchange(shared_ptr<T> desired,
                             memory_order order = memory_order::seq_cst) noexcept;

        bool compare_exchange_weak(shared_ptr<T>& expected, shared_ptr<T> desired,
                                  memory_order success, memory_order failure) noexcept;
        bool compare_exchange_strong(shared_ptr<T>& expected, shared_ptr<T> desired,
                                    memory_order success, memory_order failure) noexcept;

        bool compare_exchange_weak(shared_ptr<T>& expected, shared_ptr<T> desired,
                                  memory_order order = memory_order::seq_cst) noexcept;
        bool compare_exchange_strong(shared_ptr<T>& expected, shared_ptr<T> desired,
                                    memory_order order = memory_order::seq_cst) noexcept;

        void wait(shared_ptr<T> old, memory_order order = memory_order::seq_cst) const noexcept;
        void notify_one() noexcept;
        void notify_all() noexcept;

        constexpr atomic() noexcept = default;
        atomic(shared_ptr<T> desired) noexcept;
        atomic(const atomic&) = delete;
        void operator=(const atomic&) = delete;
        void operator=(shared_ptr<T> desired) noexcept;

    private:
        shared_ptr<T> p; // exposition only
    };
}
```

# atomic<weak\_ptr<T>>

n4835.pdf - Adobe Acrobat Pro Extended

File Edit View Document Comments Forms Tools Advanced Window Help

Create + Combine Collaborate Secure Sign Forms Multimedia Comment

150% Find

Bookmarks

- 20.5 Topics
- 20.6 Optional objects
- 20.7 Variants
- 20.8 Storage for any type
- 20.9 Bits
- 20.10 Memory
- 20.11 Smart pointers
  - 20.11.1 Class template unique\_ptr
  - 20.11.2 Class bad\_weak\_ptr
  - 20.11.3 Class template shared\_ptr
  - 20.11.4 Class template weak\_ptr
  - 20.11.5 Class template owner\_less
  - 20.11.6 Class template enable\_shared\_from\_this
  - 20.11.7 Smart pointer hash support
  - 20.11.8 Atomic specializations for smart pointers
    - 20.11.8.1 Atomic specialization for shared\_ptr
    - 20.11.8.2 Atomic specialization for weak\_ptr
- 20.12 Memory resources
- 20.13 Class template scoped\_allocator\_adaptor
- 20.14 Function objects
- 20.15 Metaprogramming and type traits
- 20.16 Compile-time rational arithmetic
- 20.17 Class type\_index
- 20.18 Execution policies
- 20.19 Primitive numeric conversions
- 20.20 Formatting
- 21 Strings library

20.11.8.2 Atomic specialization for weak\_ptr [util.smartptr.atomic.weak]

```
namespace std {
    template<class T> struct atomic<weak_ptr<T>> {
        using value_type = weak_ptr<T>;
        static constexpr bool is_always_lock_free = implementation-defined;

        bool is_lock_free() const noexcept;
        void store(weak_ptr<T> desired, memory_order order = memory_order::seq_cst) noexcept;
        weak_ptr<T> load(memory_order order = memory_order::seq_cst) const noexcept;
        operator weak_ptr<T>() const noexcept;

        weak_ptr<T> exchange(weak_ptr<T> desired,
                             memory_order order = memory_order::seq_cst) noexcept;

        bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired,
                                   memory_order success, memory_order failure) noexcept;
        bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired,
                                    memory_order success, memory_order failure) noexcept;

        bool compare_exchange_weak(weak_ptr<T>& expected, weak_ptr<T> desired,
                                   memory_order order = memory_order::seq_cst) noexcept;
        bool compare_exchange_strong(weak_ptr<T>& expected, weak_ptr<T> desired,
                                    memory_order order = memory_order::seq_cst) noexcept;

        void wait(weak_ptr<T> old, memory_order order = memory_order::seq_cst) const noexcept;
        void notify_one() noexcept;
        void notify_all() noexcept;

        constexpr atomic() noexcept = default;
        atomic(weak_ptr<T> desired) noexcept;
        atomic(const atomic&) = delete;
        void operator=(const atomic&) = delete;
        void operator=(weak_ptr<T> desired) noexcept;

    private:
        weak_ptr<T> p; // exposition only
    };
}
```

42

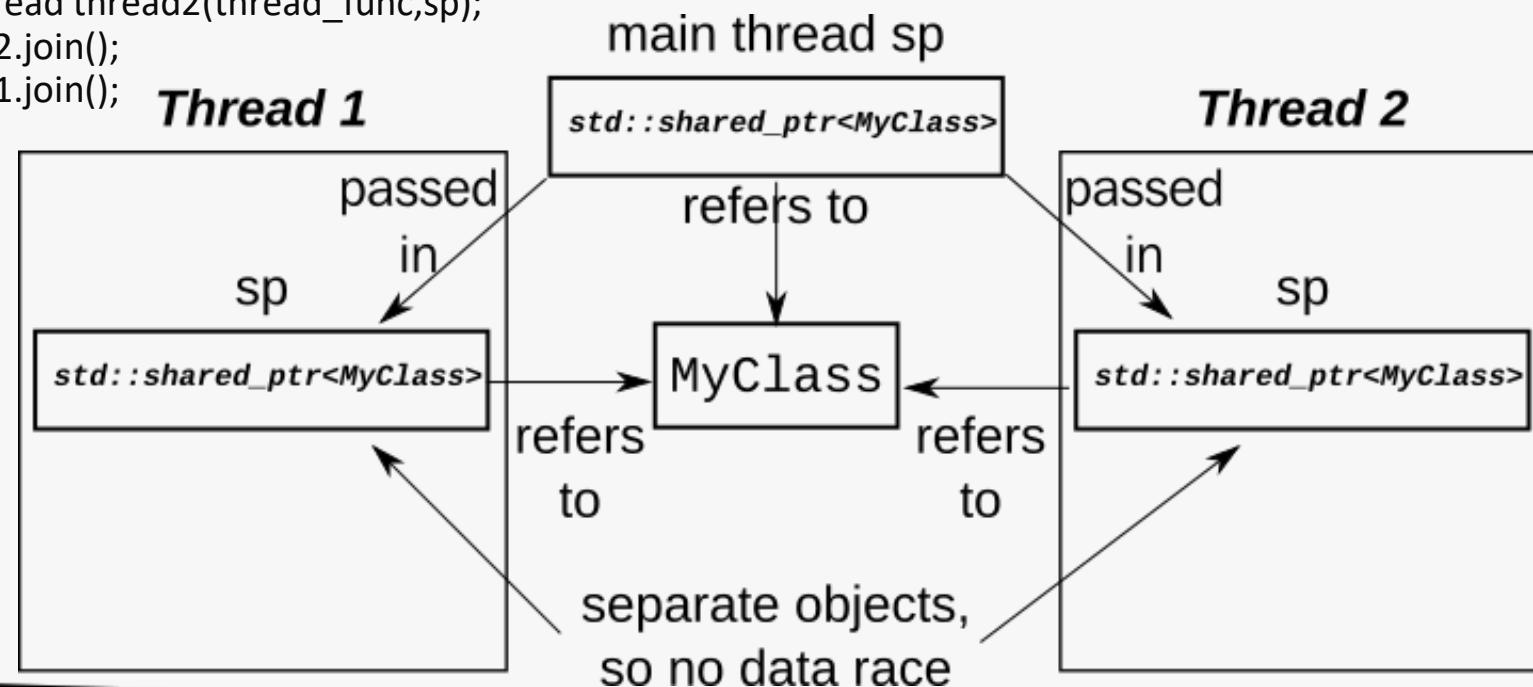
# C++11 Smart Pointers

```
class MyList{
    shared_ptr<Node> head;
    ...
    void pop_front(){
        std::shared_ptr<Node> p=head;
        while(p &&
            !atomic_compare_exchange_strong(&head, &p,
            p));
    }
};
```

- Error-prone: all access to *head* must go through *atomic\_xxx*.
- Inefficient:  
*atomic\_compare\_exchange\_strong* is a free function taking regular *shared\_ptr*, we don't want extra synchronization in *shared\_ptr*!

# Std::shared\_ptr with multiple threads

```
class MyClass;
void thread_func(std::shared_ptr<MyClass> sp){
    sp->do_stuff();
    std::shared_ptr<MyClass> sp2=sp;
    do_stuff_with(sp2);
}
int main(){
    std::shared_ptr<MyClass> sp(new MyClass);
    std::thread thread1(thread_func,sp);
    std::thread thread2(thread_func,sp);
    thread2.join();
    thread1.join();
}
```



- `std::shared_ptr` works great in multiple threads, *provided each thread has its own copy or copies.*
  - changes to the reference count are synchronized,
  - everything just works, if your shared data is correctly synchronized.
- you need to ensure that it is safe to call `MyClass::do_stuff()` and `do_stuff_with()` from multiple threads concurrently on the same instance, but the reference counts are handled OK.

## Sharing a std::shared\_ptr instance between threads

If we're going to access this from 2 threads, then we have a choice:

1. We could wrap the whole object with a mutex, so only one thread is accessing the list at a time, or
2. We could try and allow concurrent accesses. But there are problems:
  - a. removing from the front of the list
  - b. Race condition on head
  - c. Multiple threads calling pop\_front

# atomic<shared\_ptr<T>>

```
class MyList{
    atomic<shared_ptr<Node>> head;
    ...
    void pop_front(){
        std::shared_ptr<Node> p=head;
        while(p &&
            !head.compare_exchange_strong(p,p->next));
    }
};
```

- Guaranteed atomic access
- Can be implemented more efficiently

# atomic<shared\_ptr<T>>

- implementations *may* use a mutex to provide the synchronization in atomic\_shared\_ptr
- may also manage to make it lock-free
- can be tested using the is\_lock\_free() member function
- with a lock-free atomic<shared\_ptr<t> using a split reference count for atomic<shared\_ptr<T>>
  - double -word compare and swap operation
  - the shared\_ptr control block holds a count of "external counters" in addition to the normal reference count, and each atomic\_shared\_ptr instance that holds a reference has a local count of threads accessing it concurrently.

# Atomic smart pointer

- C++20 provides `std::atomic<std::shared_ptr<T>>` and `std::atomic<std::weak_ptr<T>>` specializations.
- May or may not be **lock-free**
- If lock-free, can simplify lock-free algorithms.
- If not lock-free, a better replacement for `std::shared_ptr<T>` and a mutex.
- Can be slow under high contention.

```
template<typename T> class stack{
    struct node{
        T value;
        shared_ptr<node> next;
        node(){} node(T&& nv):value(std::move(nv)){}
    };
    std::atomic<shared_ptr<node>> head;
public:
    stack():head(nullptr){}
    ~stack(){ while(head.load()) pop(); }
    void push(T);
    T pop();
};
```

# atomic\_ref

n4835.pdf - Adobe Acrobat Pro Extended  
File Edit View Document Comments Forms Tools Advanced Window Help  
Create Combine Collaborate Secure Sign Forms Multimedia Comment  
Bookmarks  
17 Language support library  
18 Concepts library  
19 Diagnostics library  
20 General utilities library  
21 Strings library  
22 Containers library  
23 Iterators library  
24 Ranges library  
25 Algorithms library  
26 Numerics library  
27 Time library  
28 Localization library  
29 Input/output library  
30 Regular expressions library  
31 Atomic operations library  
31.1 General  
31.2 Header <atomic>  
synopsis  
31.3 Type aliases  
31.4 Order and consistency  
31.5 Lock-free property  
31.6 Waiting and notifying  
31.7 Class template atomic\_ref  
31.7.1 Operations  
31.7.2 Specializations for integral types  
31.7.3 Specializations for floating-point types  
31.7.4 Partial specialization for pointers  
31.7.5 Member operators common to integers and pointers to objects  
31.8 Class template atomic  
31.9 Non-member functions  
31.10 Flag type and operations  
31.11 Fences  
32 Thread support library  
32.1 General  
32.2 Requirements  
32.2.1 Templates

125% Find

31.7 Class template **atomic\_ref** [atomics.ref.generic]

```
namespace std {
    template<class T> struct atomic_ref {
private:
    T* ptr;           // exposition only
public:
    using value_type = T;
    static constexpr size_t required_alignment = implementation-defined;

    static constexpr bool is_always_lock_free = implementation-defined;
    bool is_lock_free() const noexcept;

    explicit atomic_ref(T&);
    atomic_ref(const atomic_ref&) noexcept;
    atomic_ref& operator=(const atomic_ref&) = delete;

    void store(T, memory_order = memory_order_seq_cst) const noexcept;
    T operator=(T) const noexcept;
    T load(memory_order = memory_order_seq_cst) const noexcept;
    operator T() const noexcept;

    T exchange(T, memory_order = memory_order_seq_cst) const noexcept;
    bool compare_exchange_weak(T&, T,
                               memory_order, memory_order) const noexcept;
```

§ 31.7 1510

© ISO/IEC N4835

```
bool compare_exchange_strong(T&, T,
                             memory_order, memory_order) const noexcept;
bool compare_exchange_weak(T&, T,
                           memory_order = memory_order_seq_cst) const noexcept;
bool compare_exchange_strong(T&, T,
                             memory_order = memory_order_seq_cst) const noexcept;
```

© 2018 Codeplay Software Ltd.

# atomic\_ref <T>

- std::atomic\_ref allows you to perform atomic operations on non-atomic objects.
- This can be important when sharing headers with C code, or where a struct needs to match a specific binary layout so you can't use std::atomic, or if you have distinctive non-atomic parts of your program and you only need to do atomic access in a few places
  - this is where atomic\_ref<T> is superior to atomic<T> and is more efficient
- **If you use std::atomic\_ref to access an object, all accesses to that object must use std::atomic\_ref within that scope.**

```
struct my_c_struct{  
    int count;  
    data* ptr;  
};  
void do_stuff(my_c_struct* p){  
    std::atomic_ref<int> count_ref(p->count);  
    ++count_ref;  
    // ...  
}
```

# coroutines

n4835.pdf - Adobe Acrobat Pro Extended  
File Edit View Document Comments Forms Tools Advanced Window Help  
Create Combine Collaborate Secure Sign Forms Multimedia Comment  
Bookmarks  
8.9 Ambiguity resolution  
9 Declarations  
9.1 Preamble  
9.2 Specifiers  
9.3 Declarators  
9.4 Initializers  
9.5 Function definitions  
9.6 Structured binding declarations  
9.7 Enumerations  
9.8 Namespaces  
9.9 The using declaration  
9.10 The asm declaration  
9.11 Linkage specifications  
9.12 Attributes  
10 Modules  
10.1 Module units and purviews  
10.2 Export declaration  
10.3 Import declaration  
10.4 Global module fragment  
10.5 Instantiation context  
10.6 Reachability  
11 Classes  
11.1 Preamble  
11.2 Properties of classes  
11.3 Class names  
11.4 Class members  
11.5 Unions  
11.6 Local class declarations  
11.7 Derived classes  
11.8 Member name lookup  
11.9 Member access control  
11.10 Initialization  
11.11 Comparisons  
11.12 Free store  
12 Overloading  
12.1 Preamble  
12.2 Overloadable declarations

Otherwise, `compare_partial_order_lattice(E, T)` is informed.

## 17.12 Coroutines [support.coroutine]

1 The header `<coroutine>` defines several types providing compile and run-time support for coroutines in a C++ program.

### 17.12.1 Header `<coroutine>` synopsis [coroutine.syn]

```
namespace std {
    // 17.12.2, coroutine traits
    template<class R, class... ArgTypes>
        struct coroutine_traits;

    // 17.12.3, coroutine handle
    template<class Promise = void>
        struct coroutine_handle;

    // 17.12.3.6, comparison operators
    constexpr bool operator==(coroutine_handle<> x, coroutine_handle<> y) noexcept;
    constexpr strong_ordering operator<=>(coroutine_handle<> x, coroutine_handle<> y) noexcept;

    // 17.12.3.7, hash support
    template<class T> struct hash;
    template<class P> struct hash<coroutine_handle<P>>;

    // 17.12.4, no-op coroutines
    struct noop_coroutine_promise;
```

§ 17.12.1 518

# coroutines

- Not Pre-emptive but cooperative
- A **coroutine** is a function that can be **suspended** mid execution and **resumed** at a later time.
- Resuming a coroutine continues from the suspension point;
- local variables have their values from the original call
- C++20 provides **stackless coroutines**
- Only the locals for the current function are saved
- Everything is localized
- Minimal memory allocation — can have millions of in-flight coroutines
- Whole coroutine overhead can be eliminated by the compiler — Gor's “disappearing coroutines”

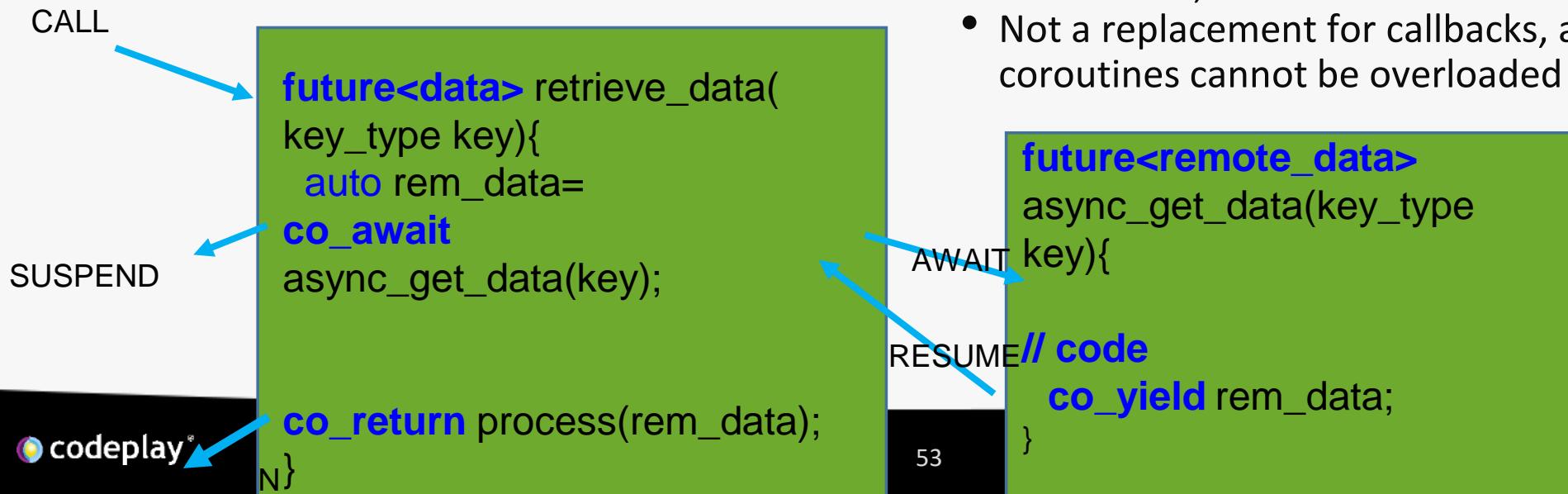
```
future<remote_data>
async_get_data(key_type key);

future<data> retrieve_data(
key_type key){
    auto rem_data=
    co_await    async_get_data(key);
    co_return process(rem_data);
}
```

# Cooperative instead of preemption

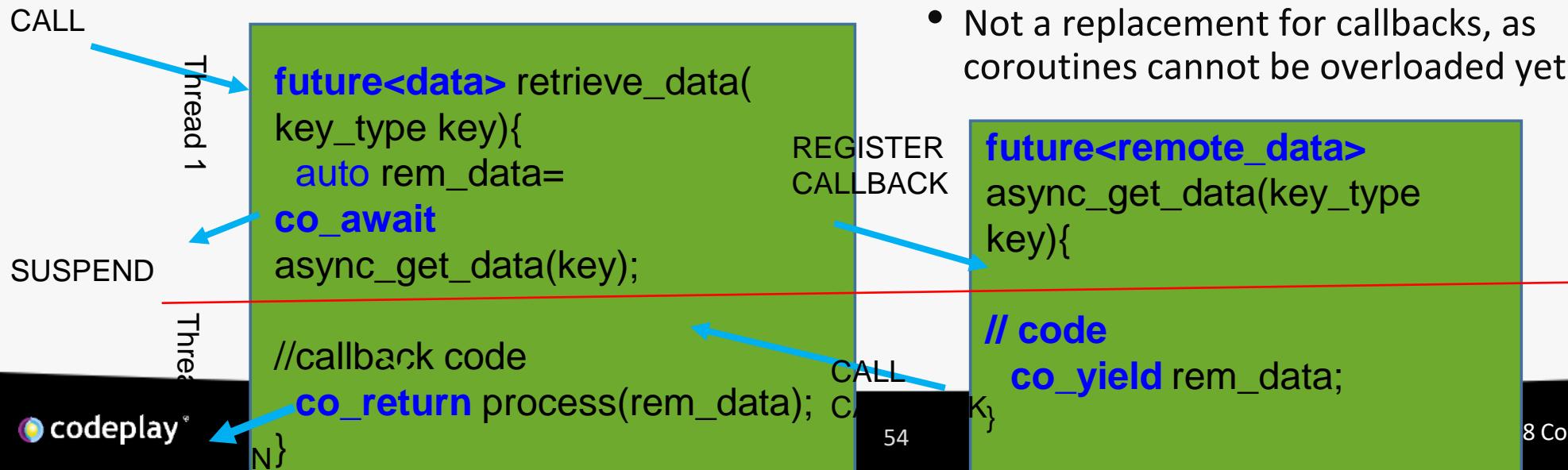
- `co_await` to suspend execution until resumed
- `co_yield` to suspend +returning a value
- `co_return` to complete+return value

- You can't tell from the signature
- Only if body uses the special keywords
- Just an implementation detail
- Dangling references
- No plain or placeholder return yet
- No `std::generator<T>`
- No `constexpr`, constructor, destructors, main as coroutines
- Not a replacement for callbacks, as coroutines cannot be overloaded yet



# Like Callback but not a replacement

- `co_await` to suspend execution until resumed
- `co_yield` to suspend +returning a value
- `co_return` to complete+return value
- You can't tell from the signature
- Only if body uses the special keywords
- Just an implementation detail
- Dangling references
- No plain or placeholder return yet
- No `std::generator<T>`
- No `constexpr`, constructor, destructors, main as coroutines
- Not a replacement for callbacks, as coroutines cannot be overloaded yet



# More for Everyone

## In C++ 20

- Better lambda
- atomic\_ref
- [[likely]][[unlikely]]
- Better constexpr
- Better Class Template Argument Deduction (CTAD)
- Better library
  - Span
  - NTTP
  - Calendars
  - constint
  - [[nodiscard()]]
  - ...

## In future C++

- Linear Algebra
- executors
- machine learning
- data affinity
- data layout
- data locality
- data movement
- mds span
- mdarray

# C++

- Good use of hardware
  - originally from C
  - CPU, GPU, FPGA, AI/ML chips, ...
- Zero-overhead abstraction
  - originally from Simula
  - performant libraries
  - simplified programming
  - control of complexity

“if you don’t need the right answer, I can make it as fast as you like”

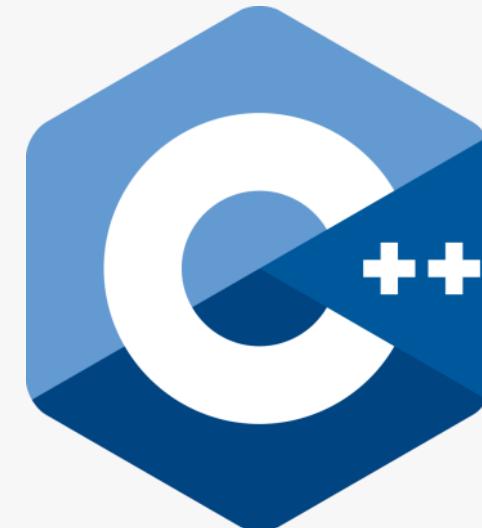
“if you can afford to waste 98% of your CPU, I can make programming much simpler”

# C++20 Take Away

- C++20 is a major release, maybe even bigger than C++11
- Less verbose code
- Solves Error Novel problem with Concepts
- Solves Constant Recompilation Problem with Modules
- Improves STL with Ranges
- Better Lazy Cooperative function Control with Coroutine and `atomic_ref`
- Works well to improve HPC workloads to make them
  - compile faster,
  - safer,
  - do more at compile time,
  - less verbose and
  - run faster.

# Act 3

# SYCL and C++ future Heterogeneity



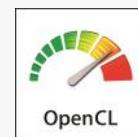


>150 Members ~ 40% US, 30% Europe, 30% Asia



Khronos is an **open, non-profit, member-driven industry consortium** developing **royalty-free standards**, and vibrant ecosystems, to harness the power of **silicon acceleration** for demanding **graphics rendering** and **computationally intensive applications** such as **inferencing** and **vision processing**

### Some Khronos Standards Relevant to Embedded Vision and Inferencing



Inferencing



Compute

59

© 2018 Codeplay Software Ltd.

# Khronos Asian Members



**Khronos warmly welcomes Chinese and Asian company participation!!**

# Khronos Active Initiatives and where does SYCL fit

**3D Graphics**  
Desktop, Mobile, Web  
Embedded and Safety Critical

**KHRONOS<sup>®</sup>  
SAFETY CRITICAL  
ADVISORY FORUM | SC™**

**3D Assets**  
Authoring  
and Delivery


**Portable XR**  
Augmented and  
Virtual Reality

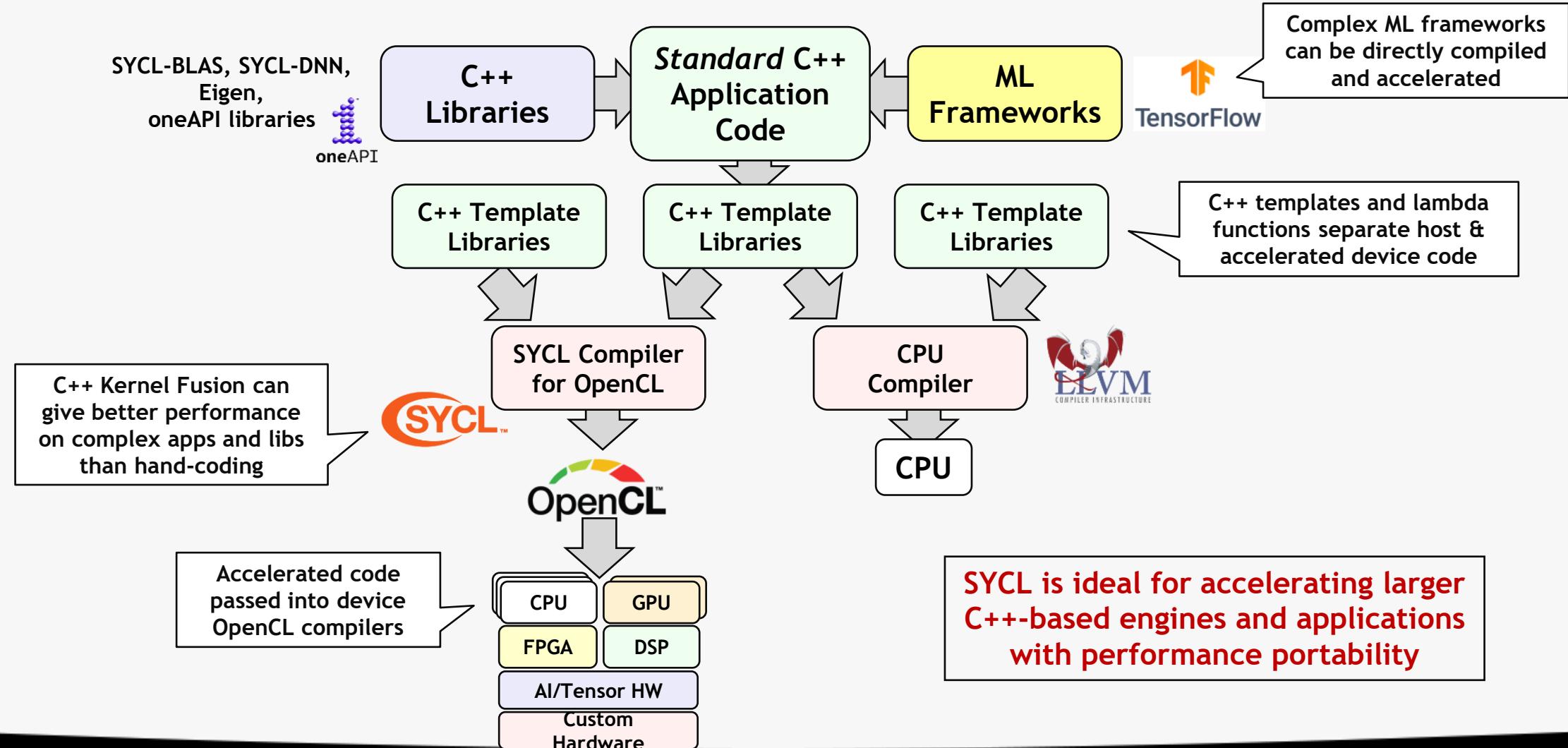


**Parallel Computation**  
Vision, Inferencing,  
Machine Learning



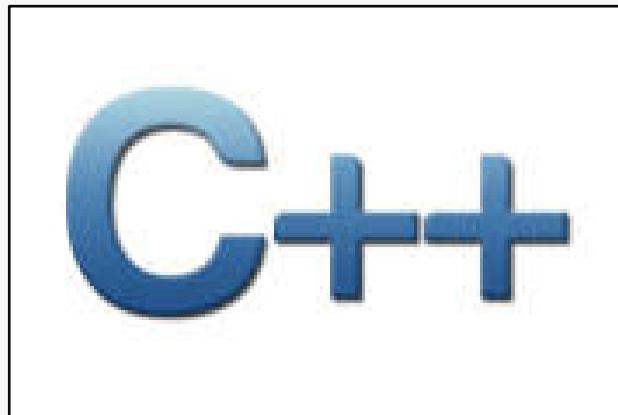
Guidelines for creating APIs to streamline system safety certification

# SYCL Single Source C++ Parallel Programming



# SYCL: A New Approach to Heterogeneous Programming in C++

# SYCL for OpenCL

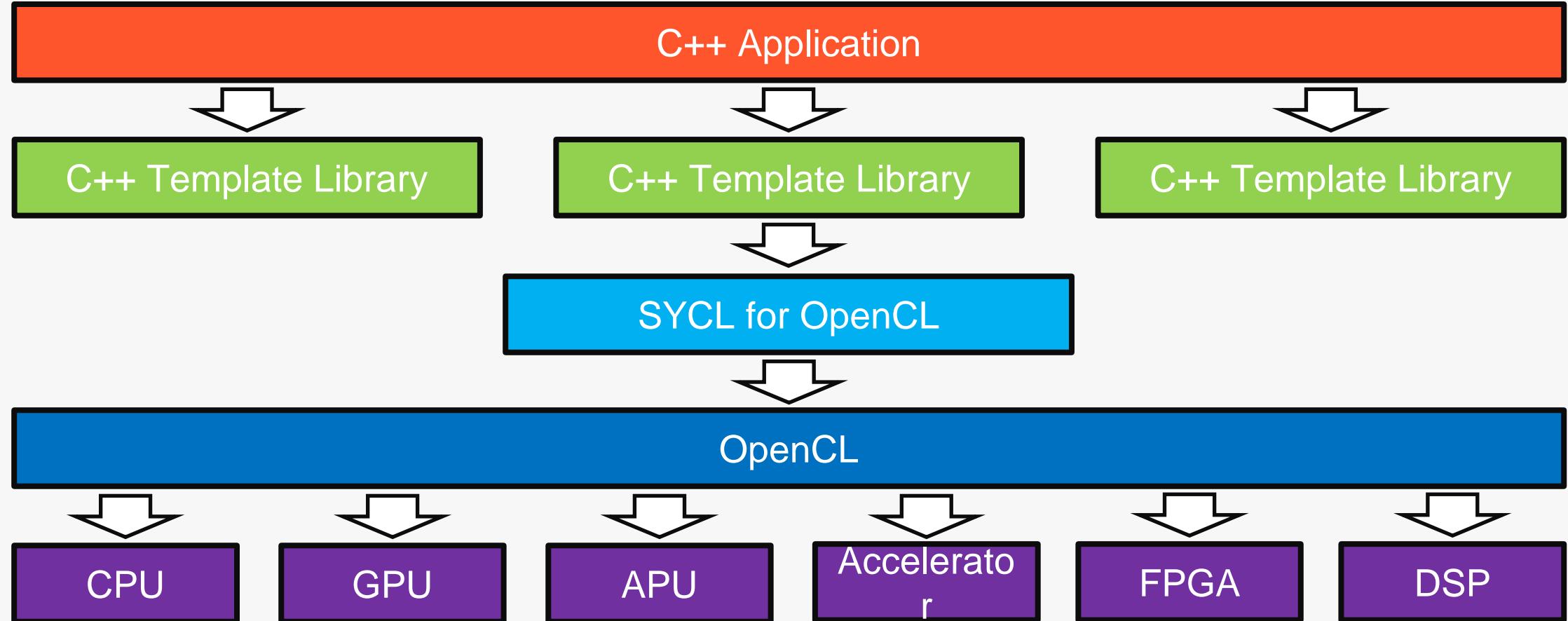


- Cross-platform, single-source, high-level, C++ programming layer
  - Built on top of OpenCL and based on standard C++17

# What Makes SYCL Different?

- SYCL separates the **what** from the **how**
  - Separates the work that is done from how it is executed
- SYCL allows data dependency optimizations
  - Separates storage and access of data
  - Allows SYCL to move data efficiently
- SYCL provides a C++ for OpenCL ecosystem
  - Open standard
  - Performance and portability of OpenCL
  - Based entirely on standard standard C++
- SYCL provides a high-level single source model
  - Provides a high-level abstraction over OpenCL boiler plate code
  - Allows C++ template libraries to target wide range of heterogeneous devices
  - Allow type safety across host and device

# The SYCL Ecosystem



## SYCL aims to easily integrate with existing C++ libraries

- SYCL is completely standard C++ with no language extensions
- SYCL provides a limited subset of C++ features

# Standard C++ + heterogeneous

```
__global__ vec_add(float *a, float *b, float *c)
{
    return c[i] = a[i] + b[i];
}

float *a;
array_view<float> a, b, c;
vec_add<float>(&a, &b, &c);
#pragma parallel_for
parallel_for_each(extent, [=] (index<2> idx) restrict(amp)
{
    c[idx] = a[idx] + b[idx];
});
```

```
cgh.parallel_for<class vec_add>(range, [=] (cl::sycl::id<2> idx) {
    c[idx] = a[idx] + b[idx];
}));
```

## SYCL aims to be open, portable and flexible

- SYCL offers a single source programming model with multi pass compilation

# Example SYCL Code

```
#include <CL/sycl.hpp>

void func (float *array_a, float *array_b, float *array_c,
           float *array_r, size_t count)
{
    buffer<float, 1> buf_a(array_a, range<1>(count));
    buffer<float, 1> buf_b(array_b, range<1>(count));
    buffer<float, 1> buf_c(array_c, range<1>(count));
    buffer<float, 1> buf_r(array_r, range<1>(count));
    queue myQueue (gpu_selector);

    myQueue.submit([&](handler& cgh)
    {
        auto a = buf_a.get_access<access::read>(cgh);
        auto b = buf_b.get_access<access::read>(cgh);
        auto c = buf_c.get_access<access::read>(cgh);
        auto r = buf_r.get_access<access::write>(cgh);

        cgh.parallel_for<class three_way_add>(count, [=](id<1> i)
        {
            r[i] = a[i] + b[i] + c[i];
        });
    });
}
```

#include the SYCL header file

Encapsulate data in SYCL *buffers* which be mapped or copied to or from OpenCL devices

Create a *queue*, preferably on a GPU, which can execute *kernels*

Submit to the queue all the work described in the handler lambda that follows

Create *accessors* which encapsulate the type of access to data in the buffers

Execute in parallel the work over an *ND range* (in this case 'count')

This code is executed in parallel on the device

# SYCL Ecosystem, Research and Benchmarks

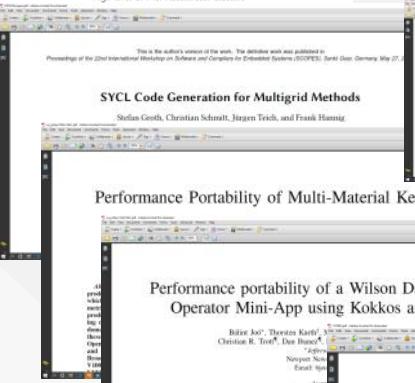
## Implementations



oneAPI



## Research

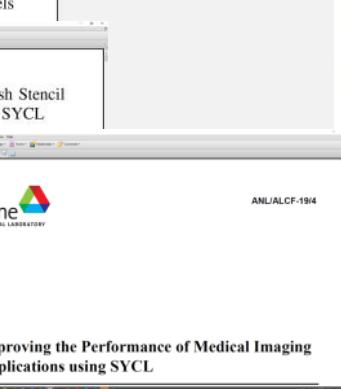


INSTITUT FÜR TECHNISCHE INFORMATIK  
PROFESSUR FÜR RECHNERARCHITEKTUR  
PROF. DR. WOLFGANG E. NÄGEL

SYCL Code Generation for Multigrid Methods  
Stefan Gottsch, Christian Schmitt, Jürgen Teich, and Frank Hammig

Performance Portability of Multi-Material Kernels

Innovative Spracherweiterungen für  
Beschleunigerkarten am Beispiel von SYCL, HC, HIP  
und CUDA: Untersuchung zu Nutzbarkeit und  
Performance



Improving the Performance of Medical Imaging  
Applications using SYCL

## Benchmarks



BabelStream

Background Parallel Research Kernels

Create test suite to study behavior of parallel systems



Include automatic verification test (analytical solution)  
Ensure enough exploitable concurrency (can be load balanced)  
Make trivially statically load balanced

RSBench

## Linear Algebra Libraries

Eigen

## Machine Learning Libraries and Parallel Acceleration Frameworks

SYCL-ML



SYCL-DNN

oneMKL

SYCL Parallel STL

K H R O N O S<sup>®</sup>  
C O R P O R A T I O N

STREAM  
HPC

Argonne  
NATIONAL LABORATORY

QUALCOMM

University of  
BRISTOL

AMD

arm

SYCL™

intel®

XILINX®

codeplay®

© 2018 Codeplay Software Ltd.

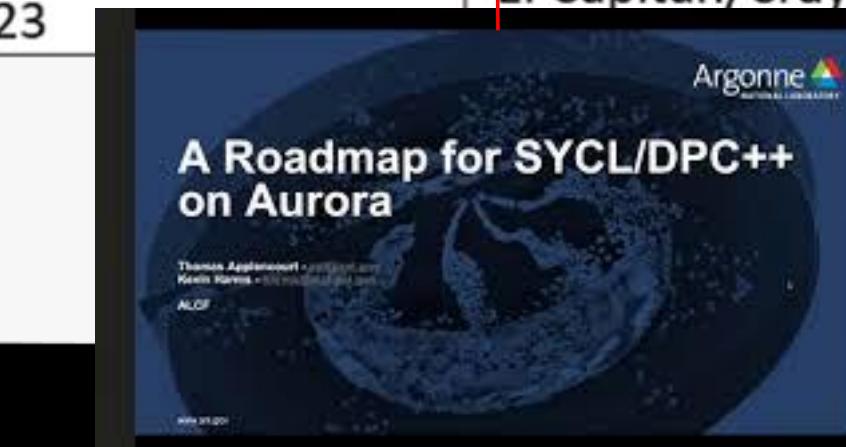
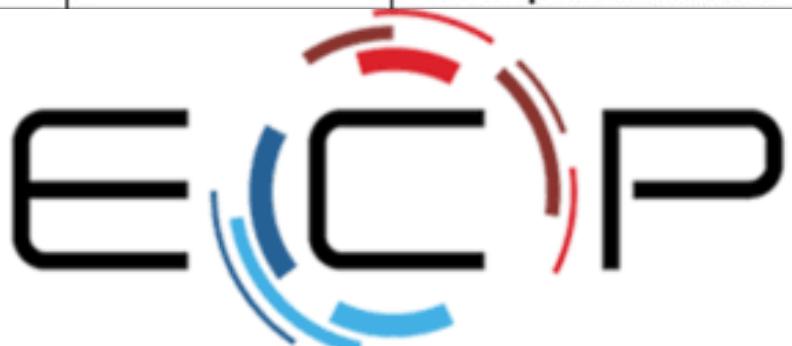


Active Working Group Members

# SYCL, Aurora and Exascale computing

| Program | Laboratory | Timeline/Projected timeline                             | System Name/Prime Contractor | System Architecture                                                                                                                              |
|---------|------------|---------------------------------------------------------|------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| CORAL-1 | ANL        | System delivered in late 2021 and accepted in 2022      | Aurora/Intel                 | Cray Shasta with Intel Xeons and Intel Xe GPUs                |
| CORAL-2 | ORNL       | System delivered in late 2021 and accepted in 2022      | Frontier/Cray                | Cray Shasta with AMD future Epyc CPUs and future Radeon GPUs  |
| CORAL-2 | LLNL       | System delivered in late 2022 and accepted in late 2023 | El Capitan/Cray              | Cray Shasta with CPUs and GPUs                               |

K H R O N



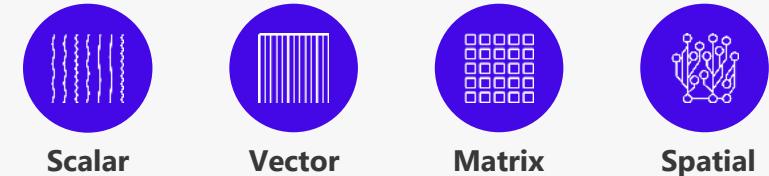
2018 Codeplay Software Ltd.

SYCL can  
run on AMD  
ROCM

# Programming Challenges for Multiple Architectures

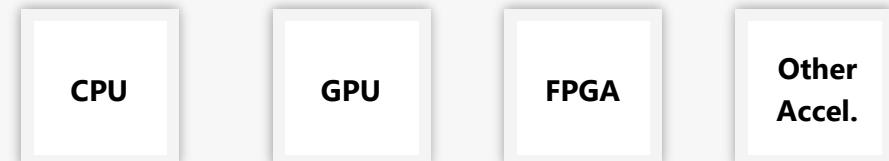
- Growth in specialized workloads
- No common programming language or APIs
- Inconsistent tool support across platforms
- Each platform requires unique software investment
- Diverse set of data-centric hardware required

Application Workloads Need Diverse Hardware



Middleware / Frameworks

Languages & Libraries



**X**PUs

# Introducing oneAPI

Unified programming model to simplify development across diverse architectures

- Unified and simplified language and libraries for expressing parallelism
- Uncompromised native high-level language performance
- Based on industry standards and open specifications
- Interoperable with existing HPC programming models

Refer to <http://software.intel.com/en-us/articles/optimization-notice> for more information regarding performance and optimization choices in Intel software products.

Application Workloads Need Diverse Hardware



Scalar



Vector



Matrix



Spatial

Middleware / Frameworks



XPU

# Vision for oneAPI Industry Initiative

A top-to-bottom ecosystem around oneAPI specification

oneAPI Specification

oneAPI Open Source Projects

oneAPI Commercial Products

Applications powered by oneAPI

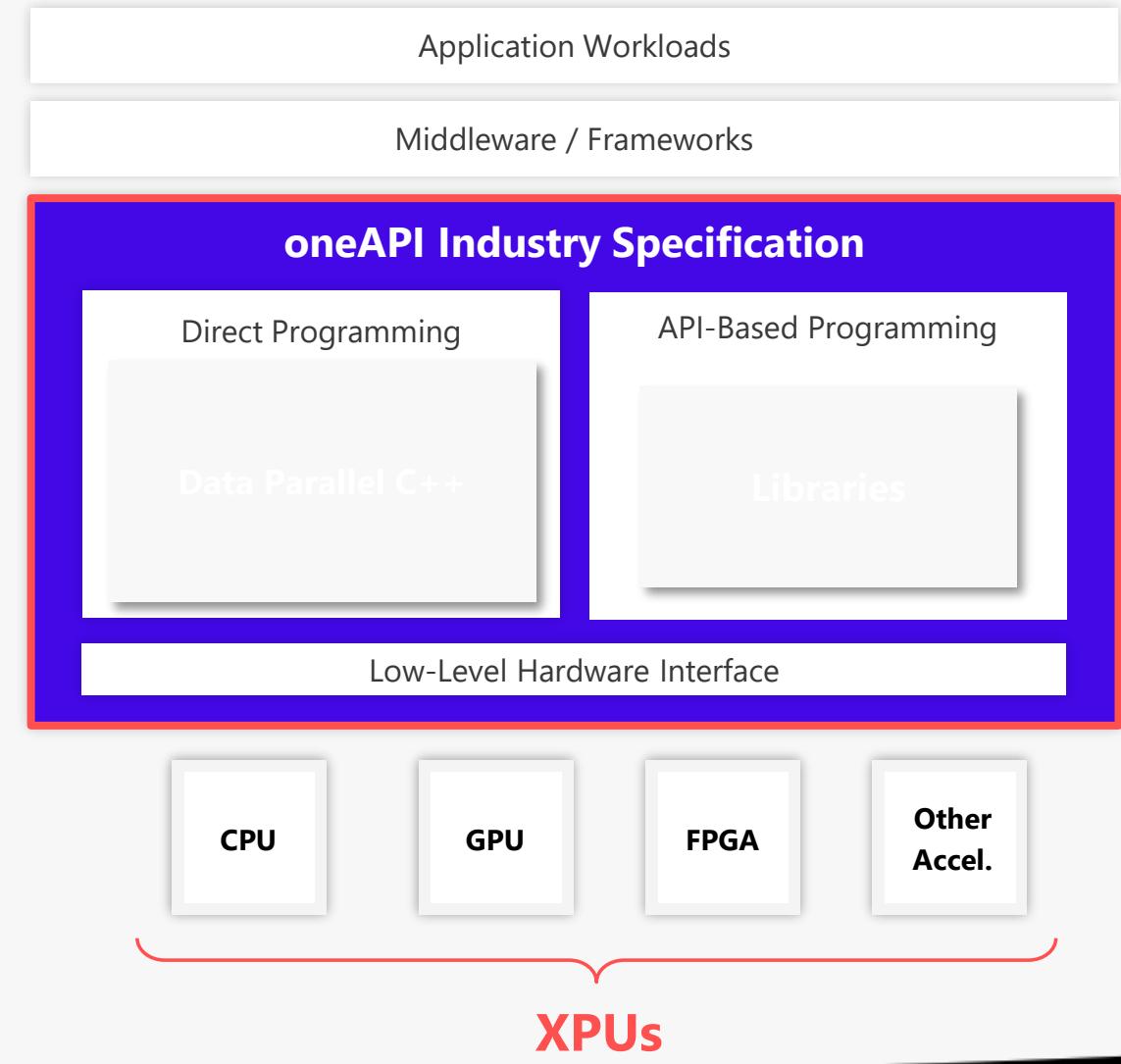
# oneAPI Industry Initiative

## — oneAPI Industry Specification

- A standards based cross-architecture language, DPC++, based on C++ and SYCL
- Powerful APIs designed for acceleration of key domain-specific functions
- Low-level hardware interface to provide a hardware abstraction layer to vendors
- Enables code reuse across architectures and vendors
- Open standard to promote community and industry support

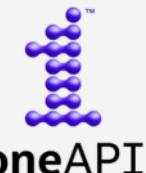
## — Technical Advisory Board

## — oneAPI Industry Brand



Some capabilities may differ per architecture and custom-tuning will still be required.  
 Refer to <http://software.intel.com/en-us/articles/optimization-notice> for more information regarding performance and optimization choices in Intel software products.

# oneAPI Specification Feedback Process



To promote compatibility and enable developer productivity and innovation, the oneAPI specification builds upon industry standards and bring a complete open cross platform developer stack.

[SEE SPEC NOW](#) [DOWNLOAD PDF](#)

**Feedback Process: oneAPI Specification Is Available For Your Input**

The time and cost of developing applications that can be easily ported across architectures has been a barrier to innovation, and has kept developers from taking advantage of new technologies that can speed up their code.

It's time to change that. We invite radical collaboration from across the ecosystem to create a shared industry spec that gives developers the power and flexibility to create fast, innovative, data-centric solutions.

oneapi-src / oneapi-spec

Code Issues 16 Pull requests 0 Actions Projects 0 Wiki Security Insights

oneAPI Specification source files <https://spec.oneapi.com>

accelerator parallel-programming linear-algebra video-processing analytics sycl deep-learning oneapi

85 commits 3 branches 0 packages 1 release 10 contributors View license

Branch: master New pull request Create new file Upload files Find file Clone or download

rscohn2 warnings: empty container (#105) Latest commit sec7b10 2 hours ago

- .github/workflows Full names include oneAPI, add spellchecker (#89) 4 days ago
- docker initial 2 months ago
- scripts add imagemagick, disable mkl tarball (#96) 3 days ago
- site-root fix links (#24) 2 months ago
- source warnings: empty container (#105) 2 hours ago
- .gitignore Initial version of K-Means spec 11 days ago
- .gitlab-ci-onetbb.yml Change how standalone oneTBB and oneDPL specs are built (#97) 2 days ago

**CONTENTS:**

- Introduction
  - Target Audience
  - Goals of the specification
  - Definitions
  - Supersets and subsets
  - Contributing
  - Legal
- Software Architecture
  - Library Interoperability
  - oneAPI Elements
  - DPC++: Data Parallel C++
  - oneDPC: Data Parallel C++ library
  - oneDNN: Deep Learning Math Library
  - oneCCL: Collective Communications Library
  - oneAAL: Accelerator Abstraction Layer
  - oneDAL: Data Analytics Library
  - oneTBB: Threading Building Blocks
  - oneVPL: Video Processing Library
  - oneMKL: Math Kernel Library
- Authors

Introduction

oneAPI is an open, free, and standards-based programming system that provides portability across platforms with different accelerators and across different generations of hardware. oneAPI provides the potential to achieve full platform performance with minimal platform-specific tuning. oneAPI consists of nine elements:

1. oneAPI Data Parallel C++ (DPC++): C++ with data parallel programming model
2. oneAPI Data Parallel C++ Library (oneDPC): DPC++ library that extends STL for parallelism
3. oneAPI Deep Neural Network Library (oneDNN): Library of optimized building blocks for deep learning
4. oneAPI Collective Communications Library (oneCCL): Efficient implementation of communications patterns for deep learning
5. Accelerator Abstraction Layer (AAL): System interface for oneAPI languages and libraries
6. oneAPI Data Analytics Library (oneDAL): Optimized algorithmic building blocks for data analysis
7. oneAPI Threading Building Blocks (oneTBB): Library for parallel programming and heterogeneous computing
8. oneAPI Video Processing Library (oneVPL): Algorithms for video processing
9. oneAPI Math Kernel Library (oneMKL): Library of optimized math routines for science, engineering and financial applications

Many modern applications are data parallel: they process tremendous amounts of data, performing similar computations on each data element. Data parallel applications include AI, machine learning, data analytics, visual computing, and scientific computing. Data parallel applications can be accelerated with parallel hardware and a programming system, such as oneAPI, that allows the programmer to express parallelism.

We encourage feedback on the oneAPI Specification from organizations and individuals

# Data parallel C++

## Standards-based, Cross-architecture Language

### Language to deliver uncompromised parallel programming productivity and performance across CPUs and accelerators

DPC++ = ISO C++ and Khronos SYCL and Extensions

Allows code reuse across hardware targets, while permitting custom tuning for a specific accelerator

Open, cross-industry alternative to single architecture proprietary language

### Based on C++

Delivers C++ productivity benefits, using common and familiar C and C++ constructs

Incorporates SYCL\* from the Khronos Group to support data parallelism and heterogeneous programming

### Community Project to drive language enhancements

Extensions to simplify data parallel programming

Open and cooperative development for continued evolution

DPC++ extensions including Unified Shared Memory are being incorporated into upcoming versions of the Khronos SYCL standard.

oneAPI Industry Specification

Language

Libraries

Low Level Hardware Interface

Direct Programming:  
Data Parallel C++

Community Extensions

Khronos SYCL

ISO C++

# oneAPI Specification Libraries

Key domain-specific functions to accelerate compute intensive workloads

Custom-coded for supported architectures

oneAPI Industry Specification

Language

Libraries

Low Level Hardware Interface

| Library Name                             | Description                                                                            | Short name |
|------------------------------------------|----------------------------------------------------------------------------------------|------------|
| oneAPI DPC++ Library                     | Key algorithms and functions to speed up DPC++ kernel programming                      | oneDPC     |
| oneAPI Math Kernel Library               | Math routines including matrix algebra, fast Fourier transforms (FFT), and vector math | oneMKL     |
| oneAPI Data Analytics Library            | Machine learning and data analytics functions                                          | oneDAL     |
| oneAPI Deep Neural Network Library       | Neural networks functions for deep learning training and inference                     | oneDNN     |
| oneAPI Collective Communications Library | Communication patterns for distributed deep learning                                   | oneCCL     |
| oneAPI Threading Building Blocks         | Threading and memory management template library                                       | oneTBB     |
| oneAPI Video Processing Library          | Real-time video decoding, encoding, transcoding, and processing functions              | oneVPL     |

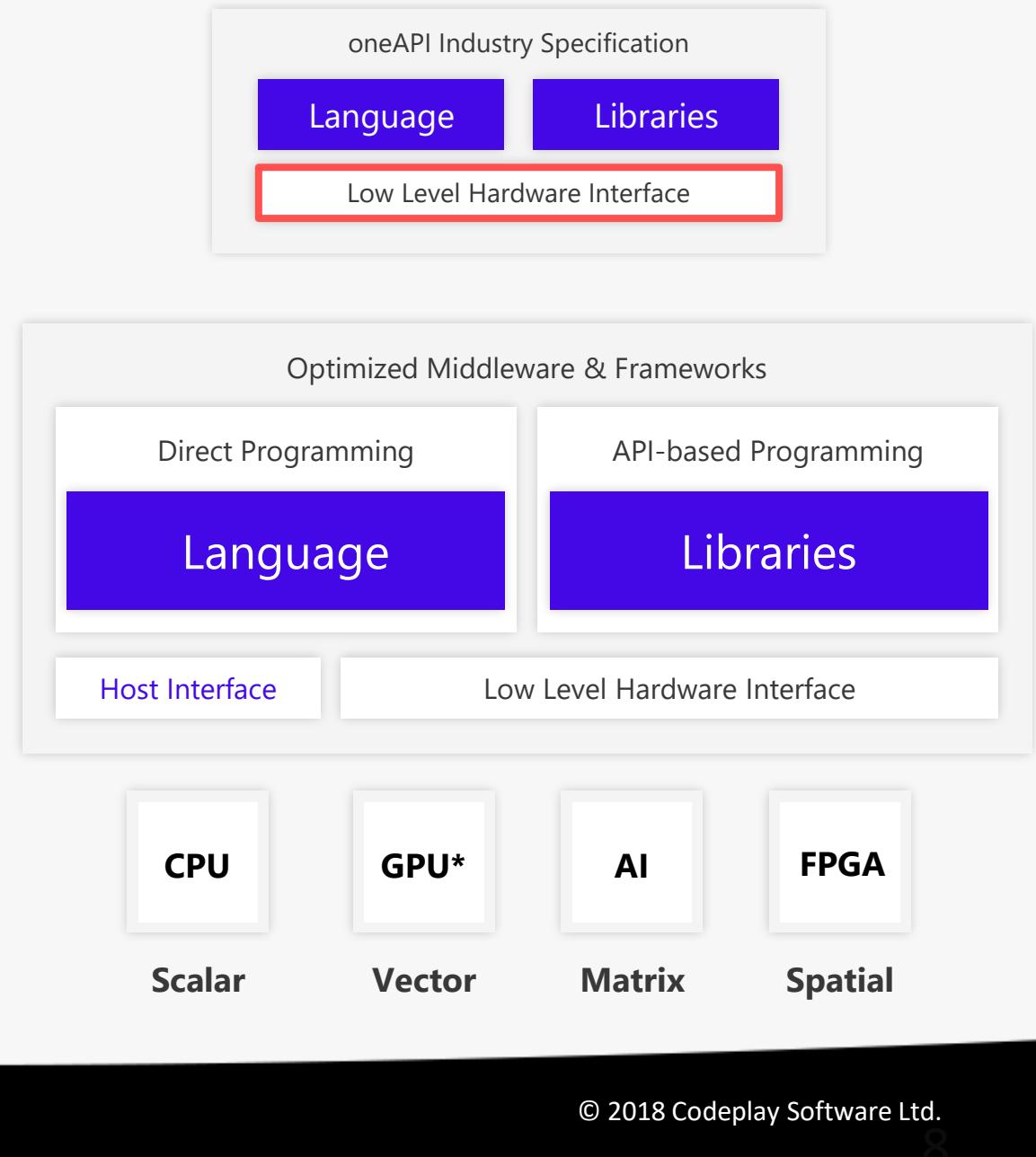
# oneAPI Level Zero

Hardware abstraction layer for low-level low-latency accelerator programming control

Target: Hardware and OS vendors who would like to implement oneAPI specification; as well as runtime developers for other languages

Refer to <http://software.intel.com/en-us/articles/optimization-notice> for more information regarding performance and optimization choices in Intel software products

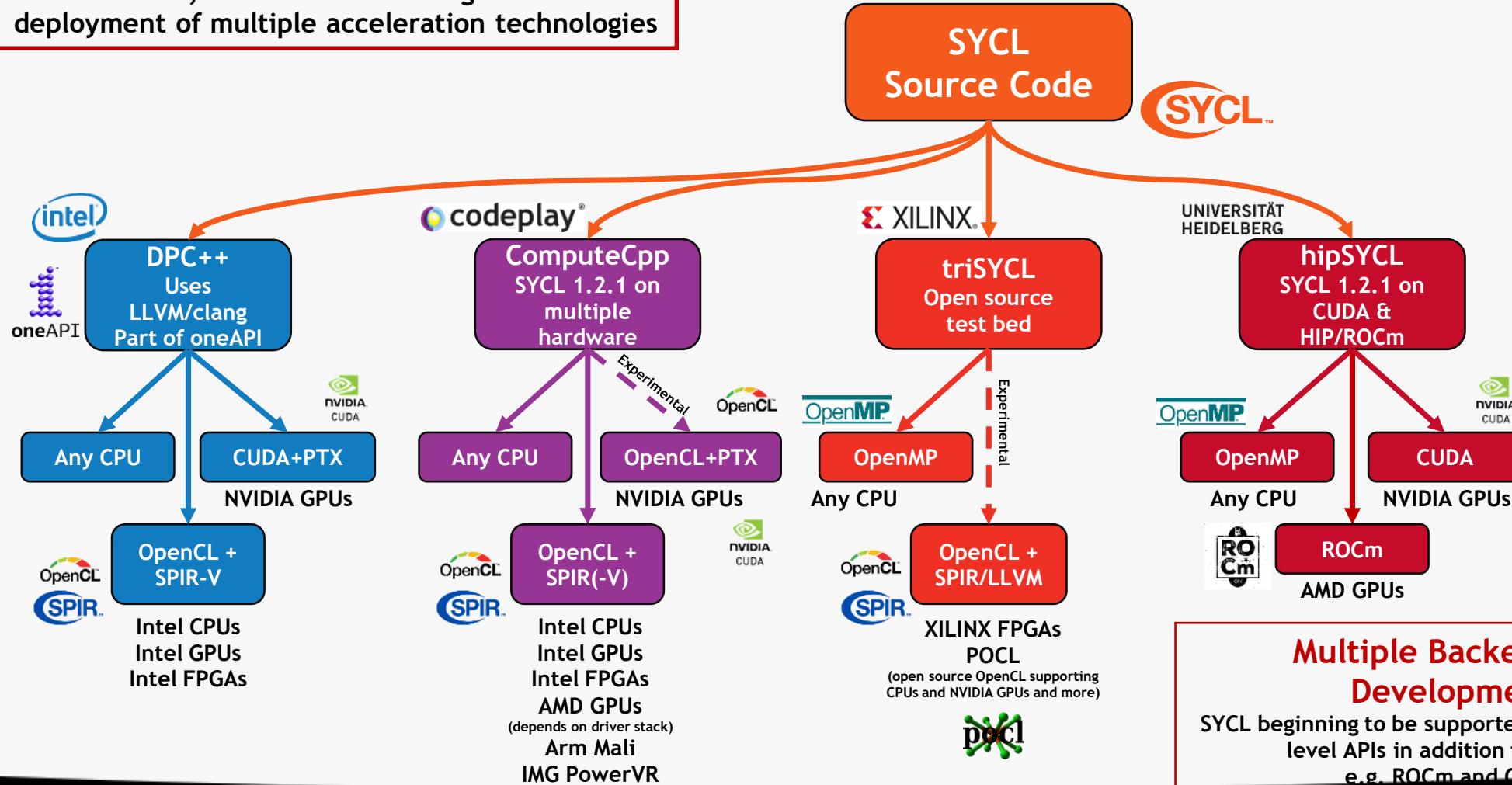
\* Current version supports GPU



# SYCL Implementations

SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies

Multiple members active in both SYCL and ISO C++ committees



## Multiple Backends in Development

SYCL beginning to be supported on multiple low-level APIs in addition to OpenCL  
e.g. ROCm and CUDA

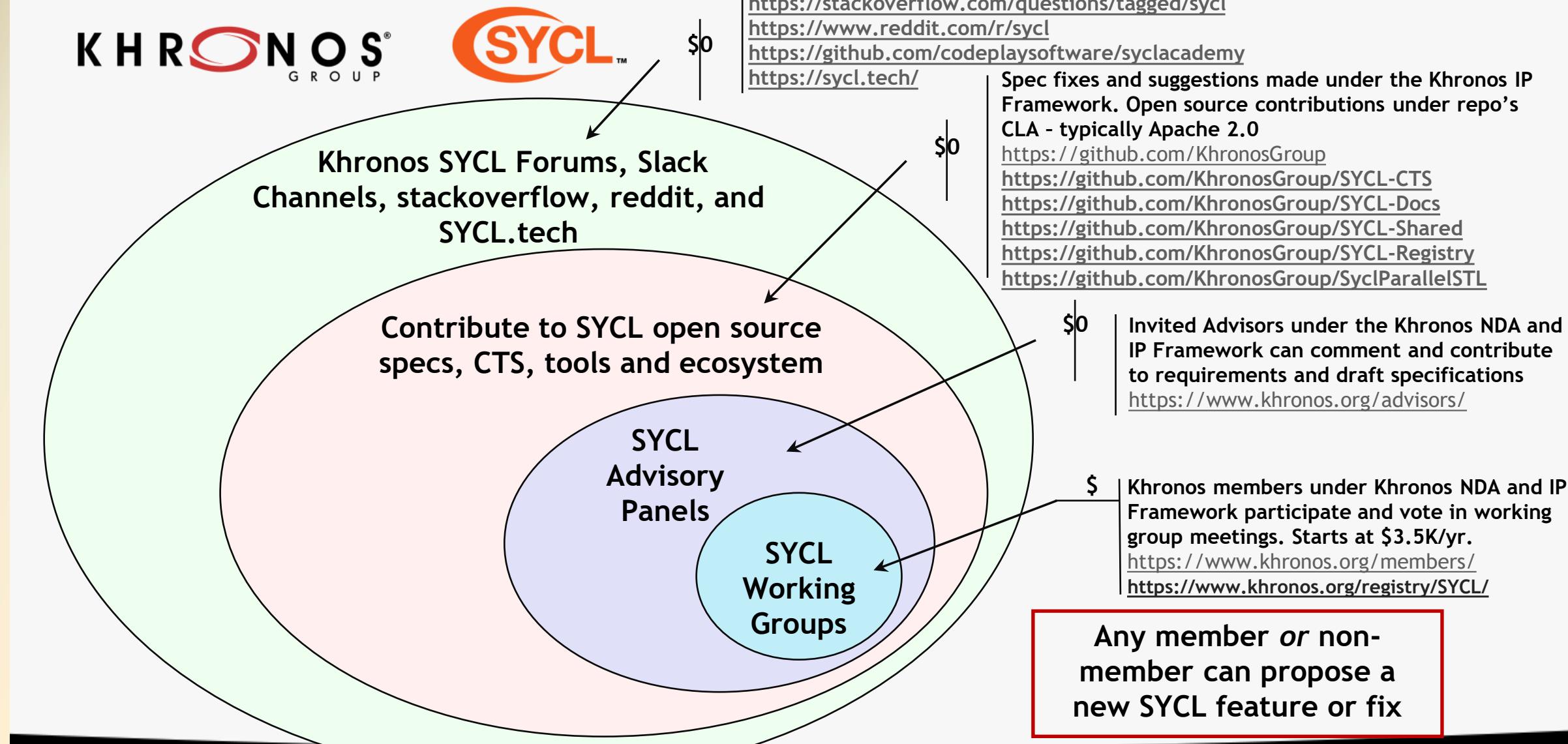
<http://sycl.tech>

© 2018 Codeplay Software Ltd.

# SYCL 2020 Provisional is here

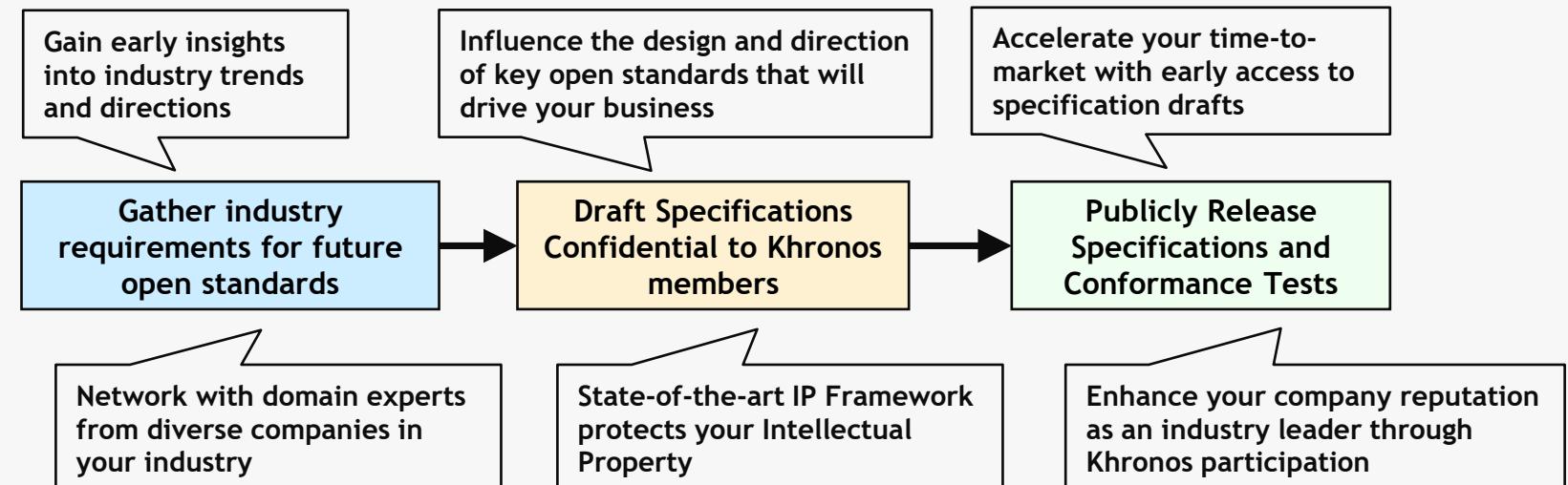
- SYCL 2020 provisional is released and final in Q4
- We need your feedback asap
  - <https://app.slack.com/client/TDMDFS87M/CE9UX4CHG>
  - <https://community.khronos.org/c/sycl>
  - <https://sycl.tech>
- What features are you looking for in SYCL 2020?
- What feature would you like to aim for in future SYCL?
- How do you join SYCL?

## Engaging with the Khronos SYCL Ecosystem



# Thank You!

- Khronos SYCL is creating cutting-edge royalty-free open standard
  - For C++ Heterogeneous compute, vision, inferencing acceleration
- Information on Khronos SYCL Standards:
- Any entity/individual is welcome to join Khronos SYCL:
- Join the SYCLCon Tutorial Monday and Wednesday Live panel : Wednesday Apr 29 15:00-18:00 GMT
  - Have your questions answered live by a group of SYCL experts



## Benefits of Khronos membership

# SYCL Ecosystem

- ComputeCpp - <https://codeplay.com/products/computesuite/computecpp>
- triSYCL - <https://github.com/triSYCL/triSYCL>
- SYCL - <http://sycl.tech>
- SYCL ParallelSTL - <https://github.com/KhronosGroup/SyclParallelSTL>
- VisionCpp - <https://github.com/codeplaysoftware/visioncpp>
- SYCL-BLAS - <https://github.com/codeplaysoftware/sycl-blas>
- TensorFlow-SYCL - <https://github.com/codeplaysoftware/tensorflow>
- Eigen <http://eigen.tuxfamily.org>

# Eigen Linear Algebra Library

**SYCL backend in mainline**

**Focused on Tensor support, providing  
support for machine learning/CNNs**

**Equivalent coverage to CUDA**

**Working on optimization for various  
hardware architectures (CPU, desktop and  
mobile GPUs)**

**<https://bitbucket.org/eigen/eigen/>**



# TensorFlow

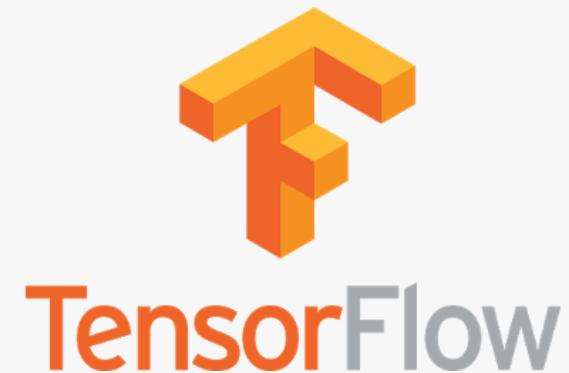
**SYCL backend support for all major CNN operations**

**Complete coverage for major image recognition networks**

GoogLeNet, Inception-v2, Inception-v3,  
ResNet, ....

**Ongoing work to reach 100% operator coverage and optimization for various hardware architectures (CPU, desktop and mobile GPUs)**

<https://github.com/tensorflow/tensorflow>



TensorFlow, the TensorFlow logo and any related marks are trademarks of Google Inc.

# SYCL Ecosystem

- Single-source heterogeneous programming using STANDARD C++
  - Use C++ templates and lambda functions for host & device code
  - Layered over OpenCL
- Fast and powerful path for bring C++ apps and libraries to OpenCL
  - C++ Kernel Fusion - better performance on complex software than hand-coding
  - Halide, Eigen, Boost.Compute, SYCLBLAS, SYCL Eigen, SYCL TensorFlow, SYCL GTX
  - Clang, triSYCL, ComputeCpp, VisionCpp, ComputeCpp SDK ...
- More information at <http://sycl.tech>

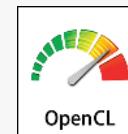
## Developer Choice

The development of the two specifications are aligned so code can be easily shared between the two approaches

### C++ Kernel Language

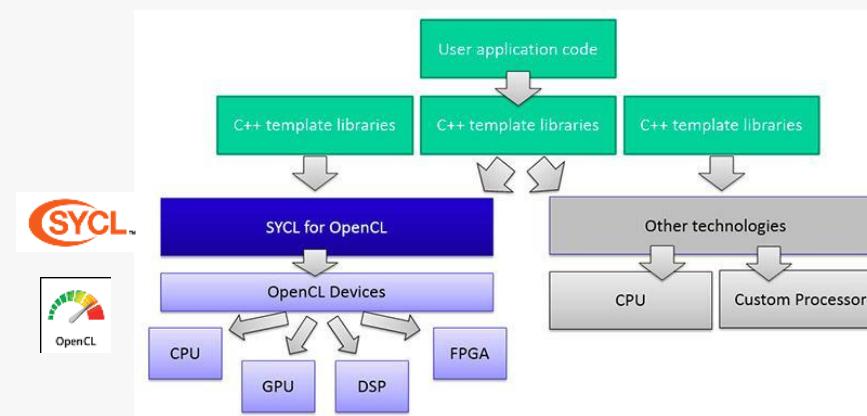
Low Level Control

'GPGPU'-style separation of device-side kernel source code and host code

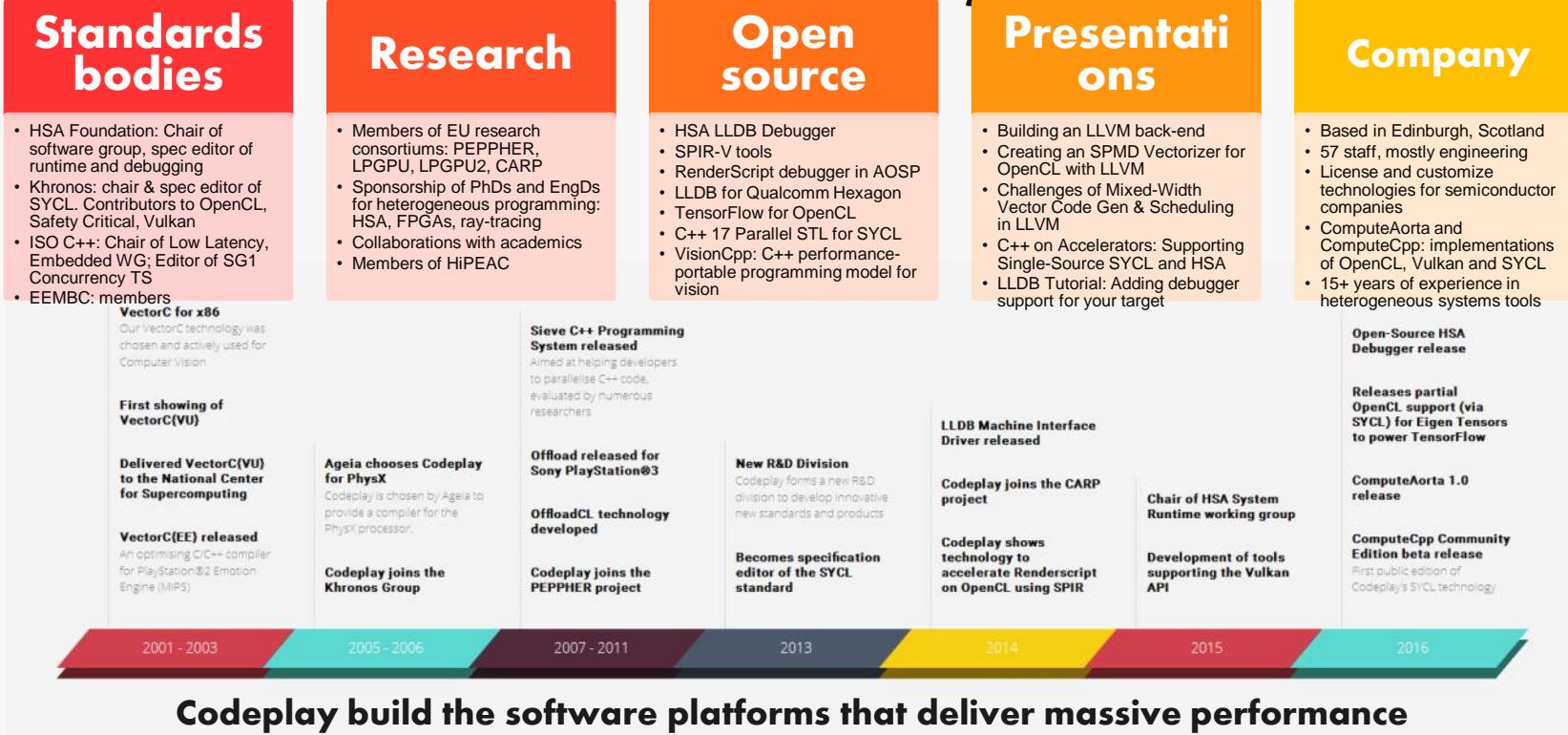


### Single-source C++

Programmer Familiarity  
Approach also taken by C++ AMP and OpenMP



# Codeplay



# What our ComputeCpp users say about us

Benoit Steiner – Google  
TensorFlow engineer



*"We at Google have been working closely with Luke and his Codeplay colleagues on this project for almost 12 months now. Codeplay's contribution to this effort has been tremendous, so we feel that we should let them take the lead when it comes down to communicating updates related to OpenCL. ... we are planning to merge the work that has been done so far... we want to put together a comprehensive test infrastructure"*

ONERA

ONERA  
THE FRENCH AEROSPACE LAB

*"We work with royalty-free SYCL because it is hardware vendor agnostic, single-source C++ programming model without platform specific keywords. This will allow us to easily work with any heterogeneous processor solutions using OpenCL to develop our complex algorithms and ensure future compatibility"*

Hartmut Kaiser - HPX



*"My team and I are working with Codeplay's ComputeCpp for almost a year now and they have resolved every issue in a timely manner, while demonstrating that this technology can work with the most complex C++ template code. I am happy to say that the combination of Codeplay's SYCL implementation with our HPX runtime system has turned out to be a very capable basis for Building a Heterogeneous Computing Model for the C++ Standard using high-level abstractions."*

WIGNER Research Centre  
for Physics



*It was a great pleasure this week for us, that Codeplay released the ComputeCpp project for the wider audience. We've been waiting for this moment and keeping our colleagues and students in constant rally and excitement. We'd like to build on this opportunity to increase the awareness of this technology by providing sample codes and talks to potential users. We're going to give a lecture series on modern scientific programming providing field specific examples."*

# Further information

- OpenCL <https://www.khronos.org/opencl/>
- OpenVX <https://www.khronos.org/openvx/>
- HSA <http://www.hsafoundation.com/>
- SYCL <http://sycl.tech>
- OpenCV <http://opencv.org/>
- Halide <http://halide-lang.org/>
- VisionCpp <https://github.com/codeplaysoftware/visioncpp>



**Community Edition**

Available now for free!



- Open source SYCL projects:
  - ComputeCpp SDK - Collection of sample code and integration tools
  - SYCL ParallelSTL – SYCL based implementation of the parallel algorithms
  - VisionCpp – Compile-time embedded DSL for image processing
  - Eigen C++ Template Library – Compile-time library for machine learning

All of this and more at: <http://sycl.tech>



# Thanks



# Michael Spertus

Amazon 首席工程师、C++标准委员会资深委员

Michael Spertus是世界级C++技术权威。作为ISO C++标准委员会资深成员，Michael 曾递交过50多项标准提案，是内存管理与性能调优方面公认的技术权威。他目前是Symantec的技术院士与首席科学家，负责云端安全服务。同时在芝加哥大学任教。

Michael 自1980开始沉迷软件设计，是IBM PC第一个商用C语言编译器的作者，并曾创办 Geodesic，后被VERITAS收购。

主办方：

**Boolan**  
高端IT咨询与教育平台

C++ Summit 2020

---

Mike Spertus

# When OO is not “OO”

# OO = Inheritance + Virtuals?

- We've all heard that inheritance and virtual functions are how C++ supports object orientation
- That's not exactly wrong
- But it's only a small piece of the C++ OO Story

# What do I mean by OO?

- Polymorphism
  - A Dog “isA” Animal
- Dispatch
  - Choose what code to run based on type
- **Note:** I know this is a vast oversimplification of a huge topic, but it will be good enough for this talk

# Typical basic example

```
// https://godbolt.org/z/7bj1h1

struct Animal {
    virtual string name() = 0;
    virtual string eats() = 0;
};

class Cat : public Animal {
    string name() override { return "cat"; }
    string eats() override { return "delicious mice"; }
};

int main() {
    unique_ptr<Animal> a = make_unique<Cat>();
    cout << "A " << a->name() << " eats " << a->eats();
}
```

# This does some good things

- `a` can be any kind of `Animal`, and the code in `main()` will work properly based on the type of the animal
- It also makes good use of modern C++ features like `make_unique` and `override`
- In many cases, this can be exactly the right approach

# Possible downsides

- Performance
  - `make_unique` uses operator `new` to allocate memory, which can take thousands of clock cycles
  - Virtual functions can have some runtime cost
- Lack of Flexibility
  - The client may wish the class had different virtual functions
  - Inheritance hierarchies can become hard to maintain
    - “Spaghetti Inheritance”

# VIRTUAL METHOD PERFORMANCE

# What are virtual functions and are they slow?

- Review: A virtual function is called based on the runtime type of the object even if it is accessed through a base class pointer
- You may have heard that virtual functions only have one more indirection, and you don't need to worry about the performance
- This is actually true most of the time
- But not always (this is not well-known)
- Let's look at an example

# Benchmark 1

```
class A {  
public:  
    // 15.83s if virtual 15.82 if not virtual  
    virtual int f(double i1, int i2) {  
        return static_cast<int>(i1*log(i1))*i2;  
    }  
};  
int main()  
{  
    boost::progress_timer t;  
    A *a = new A();  
    int ai = 0;  
    for(int i = 0; i < 100000000; i++) {  
        ai += a->f(i, 10);  
    }  
    cout << ai << endl;  
}
```

## Benchmark 2:

```
class A {  
public: // 15.36s if virtual 0.22 if not virtual  
    virtual int f(double d, int i) {  
        return static_cast<int>(d*log(d))*i;  
    }  
};  
int main()  
{  
    boost::progress_timer t;  
    auto a = new A();  
    int ai = 0;  
    for(int i = 0; i < 100000000; i++) {  
        ai += a->f(10, i); // Only change!  
    }  
    cout << ai << endl;  
}
```

# What happened?

- Changing one line made `virtuals` 70 times slower than regular methods!
- The main performance cost of virtual functions is the loss of inlining and function level optimization
  - Not the overhead of the indirection
  - In the second benchmark, `log(10)` only needed to be calculated once in the non-virtual case.
- Usually not significant, but this case is good to understand
  - The more virtual functions you use, the less the compiler can understand your code to optimize it

# Virtual methods: Best Practices

- Usually it is fine to make methods virtual, the performance cost is "almost always" minimal and virtual methods are powerful in object-oriented code
- But don't use virtual functions unthinkingly, especially in performance critical or low level code
  - Virtual functions can impede compiler code optimization and unpredictably modify memory layout

# THE VISITOR PATTERN

# The Problem

- As mentioned above, you might wish that the class hierarchy you are using had a different virtual method
- After all, the class designer doesn't understand your application
- But you may not be able to add them
  - Maybe they're not your classes
  - Maybe the virtuals you want only apply to your particular program, and it's not worth cluttering up a general interface with the particulars of your app

# The Visitor Pattern

- The Visitor Pattern is a way to make your class hierarchies extensible
- Suppose, as a user of the Animal class, I wished that it had a `lifespan()` method, but the class designer did not provide one
- We will fix that with the visitor pattern

# Class Creator: Make Your Classes Extensible

- Create a visitor class that can be overriden
- ```
struct AnimalVisitor {  
    virtual void visit(Cat &) const = 0;  
    virtual void visit(Dog &) const = 0;  
};
```
- Add an “accept” method to each class in the hierarchy
- ```
struct Animal {  
    virtual void accept(AnimalVisitor const &v) = 0;  
};  
struct Cat : public Animal {  
    virtual void accept(AnimalVisitor const &v)  
    { v.visit(*this); }  
    /* ... */  
};
```

# Class User: Create a visitor

- Now, I create a visitor that adds the methods I wish were there
- ```
struct LifeSpanVisitor
    : public AnimalVisitor {
    LifeSpanVisitor(int &i) : i(i) {}
    void visit(Dog &) const { i = 10; }
    void visit(Cat &) const { i = 12; }
    int &i;
};
```

# Using the visitor

- Now, I can get the lifespan of the Animal a I created above
- ```
int years;
a->accept(LifeSpanVisitor(years));
cout << "lives " << years;
```
- <https://godbolt.org/z/WbGe4z>

# Best practice

- If you are designing a class hierarchy where the best interface is unclear, add an `accept()` method as a customization point

# TEMPLATE AND OO

Generic Programming  
Should “Just” Be  
Normal Programming

— Bjarne Stroustrup

# Using templates instead of inheritance and virtuals

- OO and templates can be used for many of the same problems
- In the spirit of Bjarne's quote, new C++ features, like C++20 Concepts, were intentionally designed to be familiar for replacing object-oriented code
- Let's look at some examples

# Redoing Animal with concepts

- <https://godbolt.org/z/zreMKW>
- Instead of having an `Animal` base class, we now have an `Animal` concept

```
template<typename T>
concept Animal = requires(T a) {
    { a.eats() } -> convertible_to<string>;
    { a.name() } -> convertible_to<string>;
};
```

# Now define classes without inheritance

```
struct Cat {  
    string eats() { return "delicious mice"; }  
    string name() { return "cat"; }  
};  
  
struct Dog {  
    string eats() { return "sleeping cats"; }  
    string name() { return "dog"; }  
};
```

Usage is almost identical to  
before

```
Animal auto a = Cat();  
cout << "A " << a.name() << " eats " << a.eats();
```

# How does this compare?

- Performance is better
  - Objects created on stack
  - No virtual dispatch
- No inheritance
  - Makes it easier to adapt classes to our code without risking “spaghetti inheritance”
  - On the other hand, it weakens type safety
    - Pacman is not an animal but eats and has a name

# A real world example

- Suppose C++ didn't have mutexes
  - It didn't until C++11
- How would we design them?
- Let's look at how Java does it
- Java uses inheritance and virtual methods

# Java-style mutexes

```
struct lockable {  
    virtual void lock() = 0;  
    virtual void unlock() = 0;  
};  
struct mutex: public lockable {  
    void lock() override;  
    void unlock() override;  
};  
struct lock_guard {  
    lock_guard(lockable &m) : m(m) { m.lock(); }  
    ~lock_guard() { m.unlock(); }  
    lockable &m;  
};
```

# Using our Java-style mutex

```
mutex m;

void f() {
    lock_guard lk(m);
    // do stuff
}
```

# Wait, that's not how C++ mutexes work!

- The C++ committee decided to use templates instead of the virtual override approach taken by Java
- Since mutexes are frequently used in performance-critical code, this was undoubtedly the right choice
- Let's take a look

# C++-style mutexes

```
struct mutex {  
    void lock() override;  
    void unlock() override;  
};
```

```
template<typename T>  
struct lock_guard {  
    lock_guard(T &m) : m(m) { m.lock(); }  
    ~lock_guard() { m.unlock(); }  
    T &m;  
};
```

# Using our C++-style mutex is typically unchanged

```
// Exactly the same as before!  
mutex m;  
  
void f() {  
    lock_guard lk(m);  
    // do stuff  
}
```

# Class Template Argument Deduction

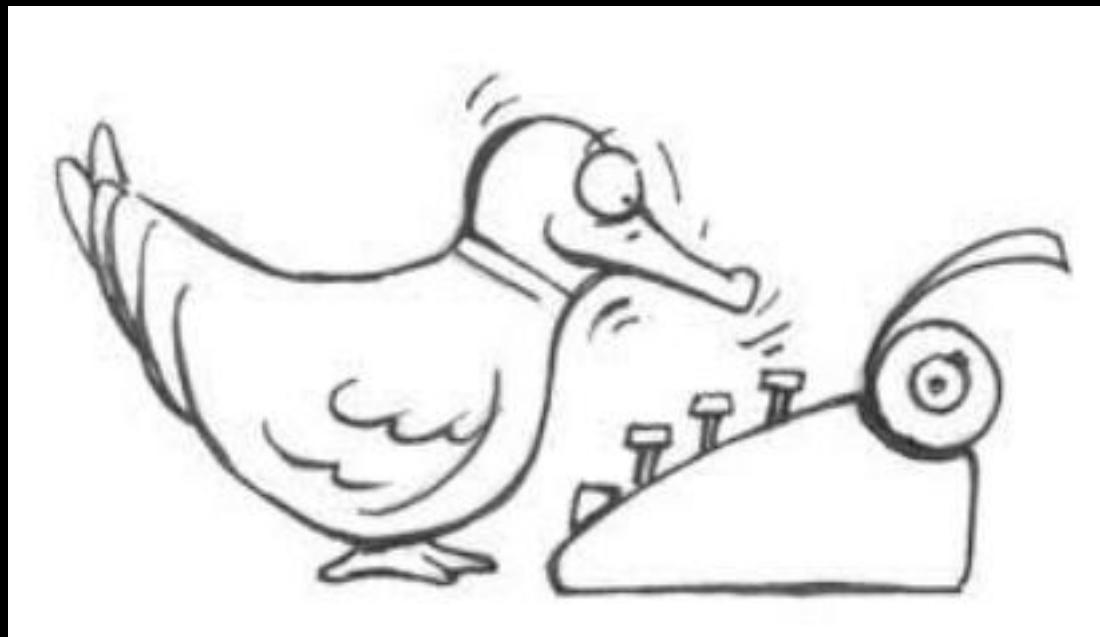
- Note the following line depended on C++17 CTAD
  - `lock_guard lk(m);`
- CTAD infers the template arguments for `lock_guard` from the constructor similarly to how Function Template Argument Deduction infers the template arguments for function templates
- CTAD can often be useful in this way when using templates instead of virtuals. E.g., if `tp1` and `tp2` are of type `time_point<C, duration<R>>`
  - `duration d = tp1 - tp2; // duration<R>`

# Warning: Consider whether you need a common type

- In our OO version of Animal, we can create a `set<unique_ptr<Animal>>`, which would be useful for describing a zoo
  - The set can contain many different kinds of animals that may not be known until runtime
- There is no easy equivalent for the version when `Animal` is a concept, because concepts can't be template arguments
- Templates' compile-time dispatch gives great performance but means the type always needs to be known at compile-time
- If this is important, you may be better off using inheritance-based OO
- Or our next idiom!

# USING DUCK-TYPED VARIANTS FOR OO

# Duck Typing



# Duck Typing

- OK. Not that
- There is a saying

If it walks like a duck and quacks  
like a duck, then it is a duck

- Let's see if we can apply this to types

# Inheritance models “isA”

- As we've mentioned, inheritance is a model of the “isA” concept
- Duck typing gives a different notion of “isA”
- If a class has a walk() method and a quack() method, let's not worry about inheritance and call it a duck

If it walks like a duck and quacks like a duck, then it isA duck

# Templates use duck typing

- ```
template<typename T>
T square(T x) { return x*x; }
```
- `square(5); // OK`
- We don't require that `T` inherits from a `HasMultiplication` class

# Concepts

- C++20 Concepts improve duck typing
- We could have a HasMultiplication concept that would do the trick without requiring any complex inheritance hierarchies

# Dynamic dispatch

- While C++ templates have always been duck typed, templates are used for compile-time dispatch
- By contrast, OO is used for dynamic dispatch
- Because duck typing is flexible and forgiving while remaining statically typesafe, people have asked whether we could use dynamically-dispatched duck typing as an alternative to inheritance

# Using Duck-Typed Variants in place of OO

- Suppose we knew (at compile-time) all of the Animal classes
  - E.g., Cat and Dog
- However, we don't know the type of a particular animal until run-time
- Instead of inheriting from an abstract animal base class, we can have an animal variant
- using `Animal = variant<Cat, Dog>;`

# How do I call a method on a variant?

- While `variant<Cat, Dog>` is a great way to store either a `Cat` or a `Dog`
- How can I simulate virtual functions and call the right `name()` method for the type it is holding?

# The C++ standard library has a solution: `std::visit`

- **Warning:** Do not confuse with the Visitor Pattern we discussed earlier
- If `v` is a variant, and `c` is a callable, `visit(v, c)` calls `c` with whatever is stored in `v` as its argument
- Does this solve our problem of making variants behave like virtuals?
- Let's see

# Dynamically calling our Animal's name() method

- Animal a = Cat(); // a is a cat
- cout << visit(  
    [] (auto &x) { x.name(); },  
    a);
- Prints "Cat", just like we want
- How does it compare to using virtuals or templates?

In some ways, it's the best of both world

- Almost as fast as templates
  - Since `Animal` is a single type that can hold a `Cat` or a `Dog`, we can just use animals by value instead of having to do memory allocations
- As dynamic as traditional OO
  - A `set<Animal>` works great
  - Unlike our Concepts version

# Malleable (mallard?) typing

- Virtual functions need to exactly match what they override, so we couldn't, say, give Cat's eat() a defaulted argument
- With variants, that is not a problem
- As long as eats() is callable, we don't care about the rest
- `visit([](auto &x) { x.eats(); }, a); // OK Now`

# Users can add methods

- Just like we discussed with the Visitor Pattern, users of the Animal type can add their own methods
- To do this, we will use the “overload pattern”

# The overload pattern

- Define an overload class (you only need to do this once)
- ```
template<class... Ts>
struct overload : Ts... { using Ts::operator()...; };
```
- This inherits the function call operator of everything it is constructed with
- Let's make this clear by creating a `lifeSpan()` "method" like we did before
- ```
overload lifeSpan([](Cat &) { return 12 },
                     [](Dog &) { return 10 });
// Get life span without knowing whether
// a is a Cat or a Dog
cout << visit(lifeSpan, a); // Prints 12
```
- Note that this idiom relies on CTAD deducing the template arguments for you

# Problems with Duck Typing

- The notation is much uglier than calling a virtual function
- While the flexibility is nice, duck typing reduces type safety
  - It cannot tell that a Shape's draw() method for drawing a picture is different than a Cowboy's draw() method for drawing a gun
- Variants always uses as much space as the biggest type
- Whenever we create a new kind of Animal, we have to add it to the variant, which can create maintenance problems

# Best Practice

- Because it is ugly and not well-known, only prefer variant-based polymorphism over virtuals or templates when there is a clear benefit
  - In practice, I use it a lot, but less than I do virtual functions or templates

# Summary

- Many of the same problems can be addressed by Inheritance/Virtual, Template/Concepts, and duck-typed variants
- They all have different tradeoffs, and I use them all frequently in my programs
- Which one should I use?
  - That is why programming is not just a science, but an art, where you have to use your judgment
  - I hope you learned some good new tools for your toolbox in this lecture and some good ways of thinking about when to use them

# QUESTIONS



# Arno Schödl

Think-Cell创始人兼CTO

Arno负责所有Think-Cell产品的设计、体系结构和开发。他是Think-Cell的研发和质量团队的负责人。在创立 Think-Cell 之前，Arno曾在 Microsoft Research 和 McKinsey 工作。Arno学习了计算机科学和管理，拥有乔治亚理工学院计算机图形专业的博士学位。

主办方：

**Boolan**  
高端IT咨询与教育平台

# A Practical Approach to Error Handling

- Errors can happen anywhere
- Want reliable program
- No time to write error handling

What do we do?

# Options for Error Handling

```
file f("file.txt");
```

# Options for Error Handling

```
file f("file.txt");
```

What happens if the file does not exist?

# Options for Error Handling

```
file f("file.txt");
```

What happens if the file does not exist?

- return value

```
file f;
bool b0k=f.open("file.txt");
if( !b0k ) {...}
```

- not for ctor

# Options for Error Handling

```
file f("file.txt");
```

What happens if the file does not exist?

- return value

```
file f;
bool b0k=f.open("file.txt");
if( !b0k ) {...}
```

- not for ctor
- out parameter

```
bool b0k;
file f("text.txt",b0k);
if( !b0k ) {...}
```

- clutter code with checks
  - can forget check - `[ [nodiscard] ]` for return values

# Options for Error Handling (2)

- status: bad flag on first failure
  - single control path
  - good if checking at the very end is good enough
  - writing a file - ok
  - reading a file - maybe not
  - default for C++ iostreams

- status: bad flag on first failure
  - single control path
  - good if checking at the very end is good enough
    - writing a file - ok
    - reading a file - maybe not
  - default for C++ iostreams
- monad
  - goal: same code path for success and error case
  - like `std::variant<result, error>` + utilities
  - P0323R7 `std::expected`

# Options for Error Handling: Exception



- exception

CPP-Summit

- exception
  - Catch exception objects always by reference
  - Slicing
  - Copying of exception may throw -> `std::terminate`

```
struct A {...};
struct B : A {...};

try {
    throw B();
} catch( A a ) { // B gets sliced and copied into a
    ...
    throw; // throws original B
};
```

- exception
  - Catch exception objects always by reference
  - Slicing
  - Copying of exception may throw -> `std::terminate`

```
struct A {...};
struct B : A {...};

try {
    throw B();
} catch( A const& a ) { // B gets sliced and copied into a
    ...
    throw; // throws original B
};
```

# Options for Error Handling: Exception (2)

- work like multi-level return/goto
- add invisible code paths
  - one reason some code bases do not allow exceptions

```
auto inc(int i)->int { // throw(char const*)
    if(3==i) throw "Hello";
    return i+1;
}

auto main()->int {
    try {
        int n=3;
        inc(n); // throw(char const*)
        n=42;
    } catch( char const* psz ) {
        std::cout << psz;
    }
    return 0;
}
```

# Options for Error Handling: Exception (2)

- work like multi-level return/goto
- add invisible code paths
  - one reason some code bases do not allow exceptions

```
auto inc(int i)->int { // throw(char const*)
    if(3==i) throw "Hello";
    return i+1;
}

auto main()->int {
    try {
        int n=3;
        inc(n); // throw(char const*)
        n=42;
    } catch( char const* psz ) {
        std::cout << psz;
    }
    return 0;
}
```

# Options for Error Handling: Exception (3)

```
auto inc(int i, char const* & pszException )->int {
{
    if(3==i) {
        pszException="Hello";
        goto exception;
    }
    return i+1;
}
exception:
    return 0;
}
```

# Options for Error Handling: Exception (4)

```
auto main() -> int {
    char const* pszException=nullptr;
{
    int n=3;
    inc(n,pszException);
    if( pszException ) goto exception;
    n=42;
    return 0;
}
exception:
{
    std::cout << pszException;
    return 0;
}
}
```

# Options for Error Handling: Exception (4)

```
auto main() -> int {
    char const* pszException=nullptr;
{
    int n=3;
    inc(n,pszException);
    if( pszException ) goto exception;
    n=42;
    return 0;
}
exception:
{
    std::cout << pszException;
    return 0;
}
}
```

Stop whining! Of course must write exception-safe code!

# Exception Safety Guarantees



(not really exception-specific)

Part of function specification

- Never Fails