21讲工具漫谈：编译、格式化、代码检查、排错各显身手

你好，我是吴咏炜。

现代 C++ 语言，我们讲到这里就告一段落了。今天我们正式开启了实战篇，先讲一个轻松些的话题——工具。

编译器

当然，轻松不等于不重要。毕竟，工欲善其事，必先利其器。我们做 C++ 开发，最基本的工具就是编译器，对其有些了解显然也是必要的。我们就先来看看我在专栏开头就提到的三种编译器，MSVC [1]、GCC [2] 和 Clang [3]。

MSVC

三种编译器里最老资格的就是 MSVC 了。据微软员工在 2015 年的一篇博客，在 MSVC 的代码里还能找到 1982 年写下的注释 [4]。这意味着 MSVC 是最历史悠久、最成熟，但也是最有历史包袱的编译器。

微软的编译器在传统代码的优化方面做得一直不错，但对模板的支持则是它的软肋，在 Visual Studio 2015 之前尤其不行——之前模板问题数量巨大，之后就好多了。而 2018 年 11 月 MSVC 宣布终于能够编译 range-v3 库，也成了一件值得庆贺的事 [5]。当然，这件事情是值得高兴的，但考虑我在 2016 年的演讲里就已经用到了 range-v3，不能不觉得还是有点晚了。此外，我已经提过，微软对代码的“容忍度”一直有点太高（缺省情况下，不使用 `/za` 选项），能接受 C++ 标准认为非法的代码，这至少对写跨平台的代码而言，绝不是一件好事。

MSVC 当然也有领先的地方。它对标准库的实现一直不算慢，较早就提供了比较健壮的线程（[第 19 讲]、[第 20 讲]）、正则表达式（[6]）等标准库。在并发 [7] 方面，微软也是比较领先的，并主导了协程的技术规格书 [8]。微软一开始支持 C++ 标准的速度比较慢，但慢慢地，微软已经把全面支持 C++ 标准当作了目标，并在 2018 年宣布已全面支持 C++17 标准；虽然同时也承认仍有一些重大问题影响了其编译一些重要的开源 C++ 项目 [9]。

MSVC 有一个地方我一直比较喜欢，就是代码里可以写出要求链接具体什么库，而链接什么库的命令，可以是使用的第三方代码里直接给出的。这就使得在命令行上编译使用到第三方库（如 Boost）的代码变得非常容易。在使用 GCC 和 Clang 时，用到什么库，就必须在命令行上写出来，这就迫使程序员使用更规范、也更麻烦的管理方式了。具体而言，对于下面的这个最

小的单元测试程序：

```
#define BOOST_TEST_MAIN
#include <boost/test/unit_test.hpp>

BOOST_AUTO_TEST_CASE(minimal_test)
{
  BOOST_CHECK(1 + 1 == 2);
}
```

使用 GCC 或 Clang 时你需要输入类似下面这样的命令：

```
g++ -DBOOST_TEST_DYN_LINK test.cpp -lboost_unit_test_framework
```

而 Windows 下使用 MSVC 你只需要输入：

```
cl /DBOOST_TEST_DYN_LINK /EHsc /MD test.cpp
```

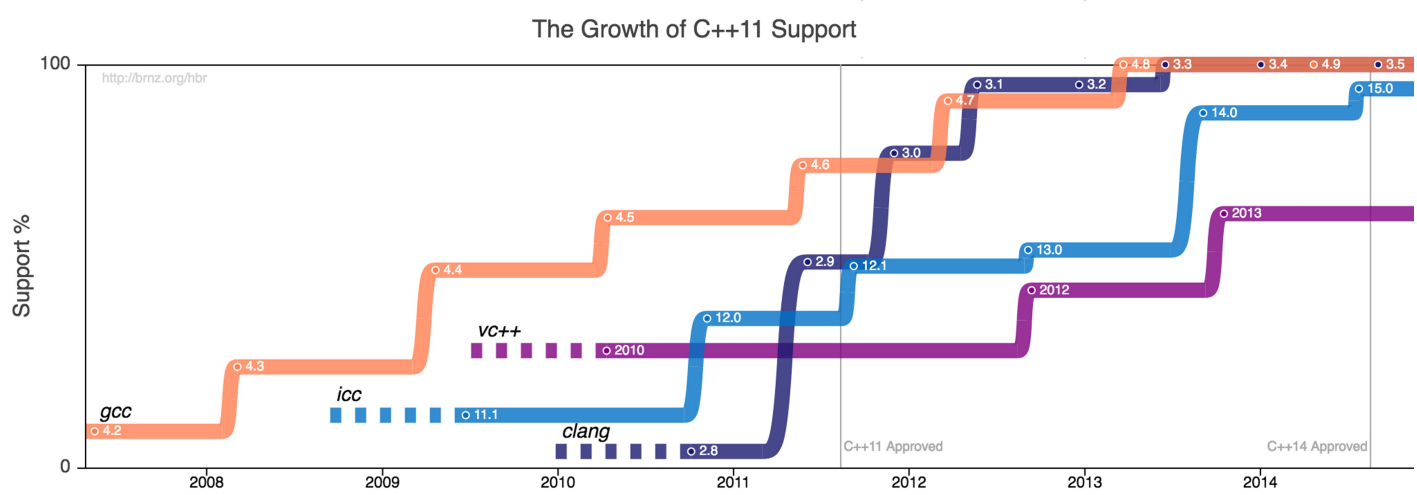
一下子就简单多了。

另外，在免费的 C++ 集成开发环境里，Visual Studio Community Edition 恐怕可以算是最好的了，至少在 Windows 上是这样。在自动完成功能和调试功能上 Visual Studio 做得特别好，为其他的免费工具所不及。如果你开发的 C++ 程序主要在 Windows 上运行，那 MSVC 就应该是首选了。

Clang

相反，在三个编译器里，最新的就是 Clang。作为 LLVM 项目的一部分，它的最早发布是在 2007 年，然后流行程度一路飙升，到现在成了一个通用的跨平台编译器。其中有不少苹果的支持——因为苹果对 GCC 的许可要求不满意，苹果把 LLVM 开发者 Chris Lattner 招致麾下（2005—2017），期间他除了为苹果设计开发了全新的语言 Swift，Clang 的 C++ 支持也得到了飞速的发展。

作为后来者，Clang 在错误信息易用性上做出了极大的改善。Clang 虽然一直在模拟 GCC 的功能和命令行，但错误信息的友好性是它的最大亮点。在语言层面，Clang 对 C++ 标准的支持也是飞速，正如下面这张图所展示的那样（[10]）：



可以看到，Clang 在 2011 异军突起，对 C++11 的支持程度在短时间甚至还超过了原先的领跑者 GCC。由于 Clang/LLVM 的模块化设计，在 Clang 上扩展新功能相当容易；而且动态库 libclang 直接向开发者暴露了分析 C++ 代码的接口，这也是

Clang 流行的一个主要原因。

即使在我主要使用 Windows 工作的时候，我在机器上也装了 Clang。我主要不是用它编译，而是利用它对 C++ 的理解，做代码的格式化（本讲下面会讲）和自动完成——对于文件数不多的项目，我还是喜欢使用 Vim [11]，那机器上能不能用 clang_complete [12] 区别就很大了。有了 clang_complete，那 Vim 里也就有个不算太笨的 C++ 自动完成引擎了。顾名思义，clang_complete 主要依赖的就是 Clang 了，更精确地说，是 libclang。

另外，当我写出在 MSVC 下编译不过的代码时，我也会看看代码能不能在 Clang 下通过。如果能过，那我就比较有信心，我写出的代码是正确的，只不过是 MSVC 处理不了而已 。

Clang 目前在 macOS 下是默认的 C/C++ 编译器。在 Linux 和 Windows 下当然也都能安装：这种情况下，Clang 会使用平台上的主流 C++ 库，也就是在 Linux 上使用 libstdc++，在 Windows 上使用 MSVC 的 C++ 运行时。只有在 macOS 上，Clang 才会使用其原生 C++ 库，libc++ [13]。顺便说一句，如果你想阅读一下现代 C++ 标准库的参考实现的话，libc++ 是可读性最好的——不过，任何一个软件产品的源代码都不是以可读性为第一考量，比起教科书、专栏里的代码例子，libc++ 肯定是要复杂多了。

最后一个关于版本号的说明：苹果开发工具里带的 Clang 的是苹果自己维护的一个分支，版本号和苹果的 Xcode 开发工具版本号一致，和开源项目 Clang 的版本号没有关系，显得比较乱。目前 Apple Clang 的最新版本是 11 了，但功能上落后于官方的 LLVM Clang 9.0 [14]。要想使用最新版本的 Clang，最方便的方式是使用 Homebrew [15] 安装 llvm：

```
brew install llvm
```

安装完之后，新的 clang 和 clang++ 工具在 /usr/local/opt/llvm/bin 目录下，和系统原有的命令不会发生冲突。你如果需要使用新的工具的话，需要改变路径的顺序，或者自己创建命令的别名（alias）。

GCC

GCC 的第一个版本发布于 1987 年，是由自由软件运动的发起人 Richard Stallman（常常被缩写为 RMS）亲自写的。因而，从诞生伊始，GCC 就带着很强的意识形态，承担着振兴自由软件的任务。在 GNU/Linux 平台上，GCC 自然是首选的编译器。自由软件的开发者，大部分也选择了 GCC。由于 GCC 是用 GPL 发布的，任何对 GCC 的修改都必须以 GPL 协议发布。这就迫使想修改 GCC 的人要为 GCC 做出贡献。这对自由软件当然是件好事，但对一家公司来讲就未必了。此外，你想拆出 GCC 的一部分来做其他事情，比如对代码进行分析，也绝不是件容易的事。这些问题，实际上就是迫使苹果公司在 LLVM/Clang 上投资的动机了。

作为应用最广的自由软件之一，GCC 无疑是非常成熟的软件。某些实验性的功能，比如对概念的支持，也是最早在 GCC 上面出现的。对 C++ 标准的支持，GCC 一直跟得非常紧，但是，由于自由软件依靠志愿者的工作，而非项目经理或产品经理的管理，对不同功能的优先级跟商业产品往往不同，也造就了 GCC 和 MSVC 上各有不同的着重点，优化编译结果哪个性能更高也会依赖于具体的程序。当然 GCC 是跨平台的，这点上肯定是 MSVC 不及的。根据 GCC 的方式写出的代码，跨平台性就会更好。目前我已知的最主要例外是终端上的多语言支持：由于 GCC 在 Windows 上使用了 MSVC 的一个过时的运行库 MSVCRT.DLL，到现在为止 GCC 要在终端上显示中文经常会出现问题 [16]。

初期 GCC 在出错信息的友好程度上一直做得不太好。但 Clang 的出现刺激出了一种和 GCC 之间的良性竞争，到今天，GCC 的错误信息反而是最友好的了。我如果遇到程序编译出错在 Clang 里看不明白的话，我会试着用 GCC 再编译看看，在某些情况下，可能 GCC 的出错信息会更让人明白一些。

在可预见的将来，在自由/开源软件的开发上，GCC 一直会是编译器的标准。

格式化工具

Clang-Format

我上面提到了 Clang 有着非常模块化的设计，容易被其他工具复用其代码分析功能。LLVM 团队自己也提供一些工具，其中我个人最常用的就是 Clang-Format [17]。

在使用 Clang-Format 之前，我也使用过一些其他的格式化工具。它们和 Clang-Format 的最大区别是，它们不理解 C++ 代码，在对付简单的 C 代码时还行，遇到复杂的 C++ 代码时就很容易出问题。此外，Clang-Format 还很智能，可以像人一样，根据具体情况和剩余空间来格式化，比如：

```
void func(int arg1, int arg2,
          int arg3);

void long_func_name(int arg1,
                    int arg2,
                    int arg3);

void a_very_long_func_name(
  int arg1, int arg2, int arg3);
```

此外，它也提供了完善的配置项，你可以根据自己的需要来进行配置，如这是我的一个项目使用的格式化选项：

https://github.com/adah1972/nvwa/blob/master/.clang-format

C++ 项目里放上这样一个文件，代码的格式化问题大家就不用瞎争了——大家确定这个文件的内容就行。

目前这个专栏的代码格式化选项也和上面的类似，最主要的区别就是行长限制（ColumnLimit）设成了 36，缩进宽度（IndentWidth）等选项基本减半，来适配手机的小显示屏。如果没有 Clang-Format，做代码的小屏适配就会累多了。

代码检查工具

Clang-Tidy

Clang 项目也提供了其他一些工具，包括代码的静态检查工具 Clang-Tidy [18]。这是一个比较全面的工具，它除了会提示你危险的用法，也会告诉你如何去现代化你的代码。默认情况下，Clang-Tidy 只做基本的分析。你也可以告诉它你想现代化你的代码和提高代码的可读性：

```
clang-tidy --checks='clang-analyzer-*,modernize-*,readability-*' test.cpp
```

以下面简单程序为例：

```
#include <iostream>
#include <stddef.h>

using namespace std;

int sqr(int x) { return x * x; }

int main()
{
  int a[5] = {1, 2, 3, 4, 5};
  int b[5];
  for (int i = 0; i < 5; ++i) {
    b[i] = sqr(a[i]);
  }
  for (int i : b) {
    cout << i << endl;
  }
  char* ptr = NULL;
  *ptr = '\0';
}
```

Clang-Tidy 会报告下列问题：

- <stddef.h> 应当替换成 <cstddef>
- 函数形式 `int func(…)` 应当修改成 `auto func(…) -> int`
- 不要使用 C 数组，应当改成 `std::array`
- `5` 是魔术数，应当改成具名常数
- `NULL` 应当改成 `nullptr`

前两条我不想听。这种情况下，使用配置文件来定制行为就必要了。配置文件叫 .clang-tidy，应当放在你的代码目录下或者代码的一个父目录下。Clang-Tidy 会使用最“近”的那个配置文件。下面的配置文件反映了我的偏好：

```
Checks: 'clang-diagnostic-*,clang-analyzer-*,modernize-*,readability-*,-modernize-deprecated-headers,-modernize
```

世界清静多了：我不想听到的唐僧式的啰唣就消失了。

使用 Clang-Tidy 还需要注意的地方是，额外的命令行参数应当跟在命令行最后的 `--` 后面。比如，如果我们要扫描一个 C++ 头文件 foo.h，我们就需要明确告诉 Clang-Tidy 这是 C++ 文件（默认 .h 是 C 文件）。然后，如果我们需要包含父目录下的 common 目录，语言标准使用了 C++17，命令行就应该是下面这个样子：

```
clang-tidy foo.h -- -x c++ -std=c++17 -I../common
```

你有没有注意到，上面 Clang-Tidy 实际上漏报告了些问题：它报告了一些不重要的问题，却漏过了真正严重的问题。这似乎是个实现相关的特殊问题，因为如果把前面那些行删掉的话，后面两行有问题的代码也还是会产生告警的。

Cppcheck

Clang-Tidy 还是一个比较“重”的工具。它需要有一定的配置，需要能看到文件用到的头文件，运行的时间也会较长。而 Cppcheck [19] 就是一个非常轻量的工具了。它运行速度飞快，看不到头文件、不需要配置就能使用。它跟 Clang-Tidy 的重点也不太一样：它强调的是发现代码可能出问题的地方，而不太着重代码风格问题，两者功能并不完全重叠。有条件的情况下，这两个工具可以一起使用。

以上面的例子来为例，Cppcheck 会干脆地报告代码中最严重的问题——空指针的解引用。它的开销很低，却能发现潜在的安全性问题，因而我觉得这是个性价比很高的工具。

排错工具

排错工具当然也有很多种，我们今天介绍其中两个，Valgrind 和 nvwa::debug_new。

Valgrind

Valgrind [20] 算是一个老牌工具了。它是一个非侵入式的排错工具。根据 Valgrind 的文档，它会导致可执行文件的速度减慢 20 至 30 倍。但它可以在不改变可执行文件的情况下，只要求你在编译时增加产生调试信息的命令行参数（`-g`），即可查出内存相关的错误。

以下面的简单程序为例：

```
int main()
{
  char* ptr = new char[20];
}
```

在 Linux 上使用 `g++ -g test.cpp` 编译之后，然后使用 `valgrind --leak-check=full ./a.out` 检查运行结果，我们得到的输出会如下所示：

```
==26423== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==26423== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==26423== Command: ./a.out
==26423==
==26423==
==26423== HEAP SUMMARY:
==26423==     in use at exit: 20 bytes in 1 blocks
==26423==   total heap usage: 1 allocs, 0 frees, 20 bytes allocated
==26423==
==26423== 20 bytes in 1 blocks are definitely lost in loss record 1 of 1
==26423==    at 0x4C2A888: operator new[](unsigned long) (vg_replace_malloc.c:42
3)
==26423==    by 0x400568: main (test.cpp:3)
==26423==
==26423== LEAK SUMMARY:
==26423==    definitely lost: 20 bytes in 1 blocks
==26423==    indirectly lost: 0 bytes in 0 blocks
==26423==      possibly lost: 0 bytes in 0 blocks
==26423==    still reachable: 0 bytes in 0 blocks
==26423==         suppressed: 0 bytes in 0 blocks
==26423==
==26423== For counts of detected and suppressed errors, rerun with: -v
==26423== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

即其中包含了内存泄漏的信息，包括内存是从什么地方泄漏的。

Valgrind 的功能并不只是内存查错，也包含了多线程问题分析等其他功能。要进一步了解相关信息，请查阅其文档。

nvwa::debug_new

在 nvwa [21] 项目里，我也包含了一个很小的内存泄漏检查工具。它的最大优点是小巧，并且对程序运行性能影响极小；缺点主要是不及 Valgrind 易用和强大，只能检查 `new` 导致的内存泄漏，并需要侵入式地对项目做修改。

需要检测内存泄漏时，你需要把 debug_new.cpp 加入到项目里。比如，可以简单地在命令行上加入这个文件：

```
c++ test.cpp \
../nvwa/nvwa/debug_new.cpp
```

下面是可能的运行时报错：

```
Leaked object at 0x100302760 (size 20, 0x1000018a4)
*** 1 leaks found
```

在使用 GCC 和 Clang 时，可以让它自动帮你找出内存泄漏点的位置。在命令行上需要加入可执行文件的名称，并产生调试信息：

```
c++ -D_DEBUG_NEW_PROGNAME=\"a.out\" \
-g test.cpp \
../nvwa/nvwa/debug_new.cpp
```

这样，我们就可以在运行时看到一个更明确的错误：

```
Leaked object at 0x100302760 (size 20, main (in a.out) (test.cpp:3))
*** 1 leaks found
```

这个工具的其他用法可以参见文档。

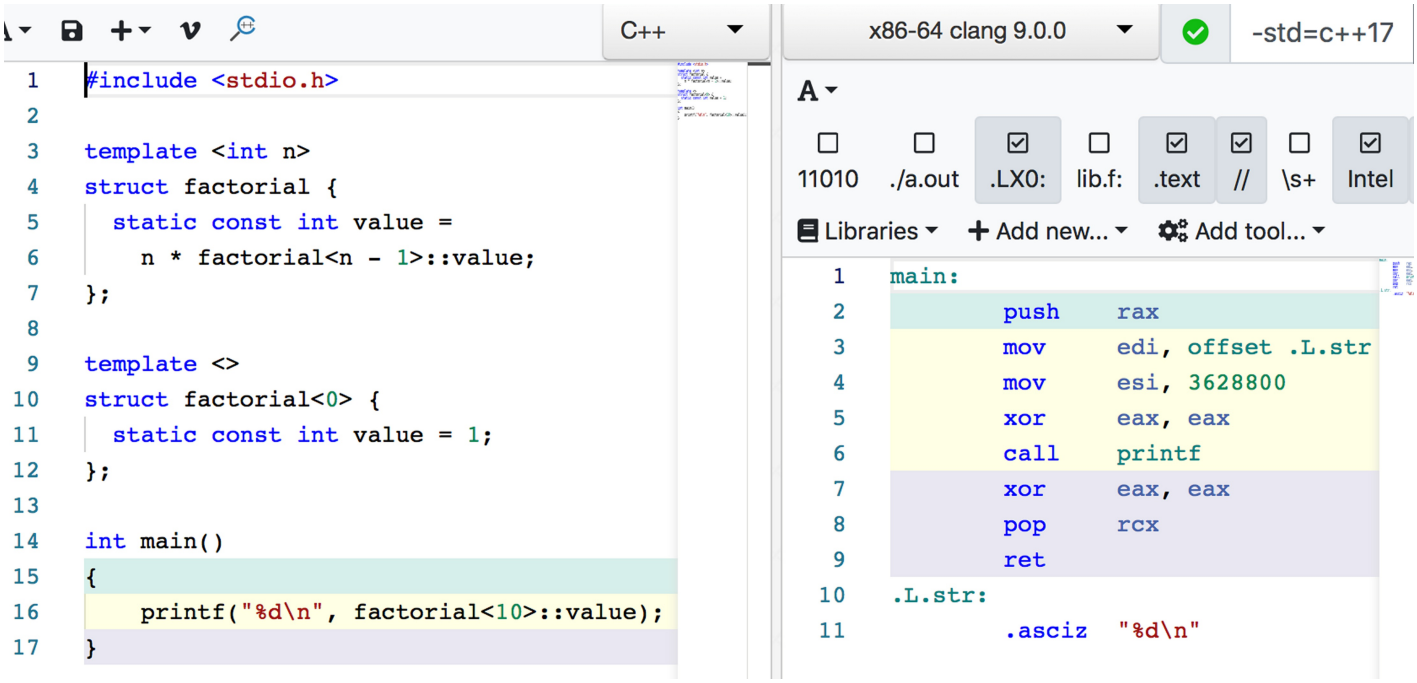
网页工具

Compiler Explorer

编译器都有输出汇编代码的功能：在 MSVC 上可使用 `/Fa`，在 GCC 和 Clang 上可使用 `-S`。不过，要把源代码和汇编对应起来，就需要一定的功力了。在这点上，godbolt.org [22] 可以提供很大的帮助。它配置了多个不同的编译器，可以过滤掉编译器产生的汇编中开发者一般不关心的部分，并能够使用颜色和提示来帮助你关联源代码和产生的汇编。使用这个网站，你不仅可以快速查看你的代码在不同编译器里的优化结果，还能快速分享结果。比如，下面这个链接，就可以展示我们之前讲过的一个模板元编程代码的编译结果：

https://godbolt.org/z/zPNEJ4

网页截图示意如下：



当然，作为一个网站，godbolt.org 对代码的复杂度有一定的限制，也不能任意使用你在代码里用到的第三方库（不过，它已经装了不少主流的 C++ 库，如我们后面会讲到的 Boost、Catch2、range-v3 和 cppcoro）。要解决这个问题，你可以在你自己的机器上本地安装它背后的引擎，compiler-explorer [23]。如果你的代码较复杂，或者有安全、隐私方面的顾虑的话，可以考虑这个方案。

C++ Insights

如果你在上面的链接里点击了“CppInsights”按钮的话，你就会跳转到 C++ Insights [24] 网站，并且你贴在 godbolt.org 的代码也会一起被带过去。这个网站提供了另外一个编译器目前没有提供、但十分有用的功能：展示模板的展开过程。

回想我们在模板编程时的痛苦之一来自于我们需要在脑子中想象模板是如何展开的，而这个过程非常容易出错。当编译器出错时，我们得通过冗长的错误信息来寻找出错原因的蛛丝马迹；当编译器成功编译了一段我们不那么理解的模板代码时，我们在感到庆幸的同时，也往往会仍然很困惑——而使用这个网站，你就可以看到一个正确工作的模板是如何展开的。以 [第 18 讲] 讨论的 `make_index_sequence` 为例，如果你把代码完整输入到网站上去、然后尝试展开 `make_index_sequence<5>`，你就会看到 `index_sequence_helper` 是这样展开的：

```
index_sequence_helper<5>
index_sequence_helper<4, 4>
index_sequence_helper<3, 3, 4>
index_sequence_helper<2, 2, 3, 4>
index_sequence_helper<1, 1, 2, 3, 4>
index_sequence_helper<0, 0, 1, 2, 3, 4>
```

如果我更早一点知道这个工具的话，我就会在讲编译期编程的时候直接建议大家用了，应该会更有助于模板的理解……

内容小结

在今天这一讲中，我们对各个编译器和一些常用的工具作了简单的介绍。用好工具，可以大大提升你的开发效率。

课后思考

哪些工具你觉得比较有用？哪些工具你已经在用了（除了编译器）？你个人还会推荐哪些工具？

欢迎留言和我分享。

参考资料

[1] Visual Studio. https://visualstudio.microsoft.com/

[2] GCC, the GNU Compiler Collection. https://gcc.gnu.org/

[3] Clang: a C language family frontend for LLVM. https://clang.llvm.org/

[4] Jim Springfield, “Rejuvenating the Microsoft C/C++ compiler”. https://devblogs.microsoft.com/cppblog/rejuvenating-the-microsoft-cc-compiler/

[5] Casey Carter, “Use the official range-v3 with MSVC 2017 version 15.9”.https://devblogs.microsoft.com/cppblog/use-the-official-range-v3-with-msvc-2017-version-15-9/

[6] cppreference.com, “std::regex”. https://en.cppreference.com/w/cpp/regex/basic_regex

[7] Microsoft, “Concurrency Runtime”. https://docs.microsoft.com/en-us/cpp/parallel/concrt/concurrency-runtime

[8] ISO/IEC JTC1 SC22 WG21, “Programming languages—C++extensions for coroutines”. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4680.pdf

[9] Ulzii Luvsanbat, “Announcing: MSVC conforms to the C++ standard”. https://devblogs.microsoft.com/cppblog/announcing-msvc-conforms-to-the-c-standard/

[10] Jonathan Adamczewski, “The growth of modern C++ support”.http://brnz.org/hbr/?p=1404

[11] Vim Online. https://www.vim.org/

[12] Xavier Deguillard, clang_complete. https://github.com/xavierd/clang_complete

[13] “libc++” C++ Standard Library .https://libcxx.llvm.org/

[14] cppreference.com, “C++ compiler support”. https://en.cppreference.com/w/cpp/compiler_support

[15] Homebrew. https://brew.sh/

[16] 吴咏炜, “MSVCRT.DLL console I/O bug”. https://yongweiwu.wordpress.com/2016/05/27/msvcrt-dll-console-io-bug/

[17] ClangFormat. https://clang.llvm.org/docs/ClangFormat.html

[18] Clang-Tidy. https://clang.llvm.org/extra/clang-tidy/

[19] Daniel Marjamäki, Cppcheck. https://github.com/danmar/cppcheck

[20] Valgrind Home. https://valgrind.org/

[21] 吴咏炜, nvwa. https://github.com/adah1972/nvwa/

[22] Matt Godbolt, “Compiler Explorer”. https://godbolt.org/

[23] Matt Godbolt, compiler-explorer. https://github.com/mattgodbolt/compiler-explorer

[24] Andreas Fertig, “C++ Insights”. https://cppinsights.io/

精选留言

Geek_71d4ac

想知道在vim上写C++，有没有比较好的插件推荐，谢谢

2020-01-13 08:01

作者回复

看我的 Vim 配置，我用到的这些和 C++ 开发有关：

clang_complete

nerdcommenter（注释）

vim-fugitive（git）

vim-gitgutter（git）

code_complete

echofunc

另外，我在 .vimrc 里加了下面几句来集成 clang-format：

" Key mappings to use clang-format

noremap <silent> <Tab> :pyxf /usr/local/opt/llvm/share/clang/clang-format.py<CR>

inoremap <silent> <C-F> <ESC>:pyxf /usr/local/opt/llvm/share/clang/clang-format.py<CR>i

2020-01-13 13:48

Vackine

怪不得看课程的代码，手机可以不用左右划

2020-02-12 09:14

tech2ipo

我看到公司的算法库都用intel的icc编译器编译的，算法的同事说icc编译器编译的代码性能会好一些，对于计算密集型的程序是否可以用icc编译器代替gcc?

2020-02-02 03:00

作者回复

你可以对照下面链接看一下，你需要的 C++ 功能在 ICC 里是不是支持了：

https://en.cppreference.com/w/cpp/compiler_support

初看之下，似乎问题不大。但性能问题，还是需要实测的。

如果发现有问题（性能或功能），混合编译器也是可以考虑的，虽然管理上会复杂点。可以考虑把需要 ICC 编译的东西放单独的项目，编译成库，供其他项目使用。

2020-02-02 12:25

tt

嗯，我说老师的代码怎么是我订阅的课程里显示最友好的呢！

2020-01-13 08:03

作者回复

工具很重要。

2020-01-13 09:58

王小白白白

老师，c++项目用clang编译耗时10min，gcc 20min，在使用gcc的前提下有什么办法提高速度呢，最近研究使用预编译头，并且拆分头文件，快了三四分钟感觉也不理想。另外clang和gcc编译速度为什么差这么多呀

2020-02-03 18:14

作者回复

clang 本来就是为这方面做了很多优化的……但产生优化的二进制文件，似乎 gcc 仍然要强些。

可以考虑并发编译。首先是 make 本身的 -j 参数。其次网上你能找到工具做多机并发编译的。

2020-02-04 10:27

睡在床板下

windbg + application verifier + pclint

2020-01-17 16:37

阿太

老师，在vscode用了您的clangformat配置，为啥好多配置选项报错呢。比如 regx 这些选项

2020-01-14 21:28

作者回复

具体什么错误？在命令行上直接运行有问题吗？

这些选项我从 Clang 3.x 开始用的，现在是 Clang 9，不应该有兼容性问题的……

2020-01-14 23:00

花晨少年

一直在ubuntu用clion+bazle进行项目编译，感觉挺好用的，但是macbook这套方案没法用，很蛋疼。

2020-01-13 20:40

作者回复

Bazel？那你只能找Bazel和JetBrains的人解决这个兼容性问题了，我可没办法。

或者改用CMake？我看了下Bazel的介绍，没觉得它比CMake好。特别是，CMake有不少特别的对C++的支持，包括对不同的编译器、Boost的库链接选项、C++标准，等等。

2020-01-14 14:25

三味

cpp中第三方库的管理超麻烦，x86或x64，release或debug，从vc6到最新的vs2019，windows下第三方库管理起来太麻烦了。

那么，有没有一种好用的第三方库管理呢?

如果用vs2015以上，超推荐使用vcpkg啊！就连生成汇编参考的那个网页工具，都大大咧咧的横幅推荐vcpkg，好用到爆啊！

如何使用不多说了，如何好用提一下。命令行安装对应的库，工具会进行漫长的编译。要引用这个库，只需要#include对应的头文件，lib库会自动加载，dll会在运行的时候拷贝。就这样。

2020-01-13 11:25

作者回复

名气是挺响了。我没有使用经验……

2020-01-13 13:44

廖熊猫

看到C++ Insights这个工具了，跟大家分享一下这段总结的关于递归模板的几个心得吧。

根据老师讲的这几种，我把这几种分类成: 1.数值、2.拉取、3.打包。

1. 数值型：操作数值（一般是减去），然后到达一个终止条件（一般是等于0），例如factorial

2. 拉取型：从...中每次拉取一个数值跟第一个参数进行操作，合并成一个参数，继续拉取过程，直到后面...中的参数被用光，终止条件就是指剩一个参数，例如： sum

3. 打包型： 从第一个参数中分离出一个参数进入...参数包中，然后继续这个过程，直到第一个参数达到终止条件，例如: make_index_sequence

只是个人见解，希望能对大家有点帮助。

2020-01-13 09:08

作者回复

挺好。这确实是最常见的几种用法。

2020-01-13 14:20

tt

文末有彩蛋呦——C++ Insights

2020-01-13 08:14

作者回复

嗯，这个是最近发现的，赶忙补进工具篇。

2020-01-13 09:58