Тулинг в C++,

и как мы до такого докатились…

План доклада

1. О спикере
2. Цель доклада (зачем мы с вами здесь собрались?)
3. Что мы понимаем под тулингом, и какое подмножество инструментов будем смотреть, что мы НЕ будем рассматривать
4. Исторический экскурс
   a. Эпоха “до C++”: Фортран, Си - 60-80-е
   b. Наше время по десятилетиям - 90-е-20-е
5. Группа тулинга и их работа
6. Потенциальное будущее?
7. Не затронутые темы (тысячи их) и кросс-ссылки на другие доклады

О себе

1. Занимаюсь опенсурсом в свободное время, и в рамках рабочих проектов
2. В основном специализируюсь на тулинге, стат/дин анализе, системах сборки и пакетных менеджерах
3. Контрибьютил в разные библиотеки, в том числе улучшая их системы сборки и работу с зависимостями
4. Контрибьючу в CMake, самая большая вещь - cmake_language(EXIT) для задания кастомного кода выхода (при ошибке и пр.)
5. Работаю с разными пакетниками (conan, vcpkg и системными)
6. Соавтор принятого в C++23 проползала на std::stacktrace

Зачем этот доклад?

- Ретроспективно посмотреть на историю развития тулинга в контексте C++ и его предков
- Подсветить новые возможности тулинга, систем сборки, пакетных менеджерах
- Подсветить потенциальные проблемы, которые могут возникать при работе с разными инструментами
- Рассказать о группе тулинга и ее текущих и прошлых работах
- Упомянуть будущие возможностях по интеропу тулинга

Что понимать под тулингом (tooling, инструментарием)

Любые технические средства (утилиты), используемые в процессе разработки:

- компиляторы
- линтеры
- системы сборки
- пакетные менеджеры
- дополнительные утилиты (ctags, clangd, perf, test launcher, gcov и пр)
- редакторы и среды разработки (их рассматривать почти не будем)
- и др. (тоже рассматривать не будем)

Зависимость (для контекста)

Любая сущность, которую нужно использовать для работы вашего проекта (сборки - сборочные зависимости, запуска - рантайм-зависимости).

Обычно это артефакт в виде исходников, или объектников, или собранный бинарник.

Для использования зависимости надо ее пробросить в процесс сборки (например, путь до заголовочных файлов через флаг -I, собранную библиотеку через -l<lib>)

Краткая история: тулинг до Си и C++

- **Станок Жаккарда и первые механические вычислительные машины**
- **1950е - зарождение компиляторов, Фортрана, магнитной ленты**
- **1970е - Си, unix и скрипты оболочки, первые редакторы (ed, emacs, vi), make,m4,lint**
- 1980е - cfront, autotools, ctags, libname-config-программы, posix
- 1990е - C++, Unicode, Borland, Visual Studio, pkg-config, qt, boost, doxygen
- 2000е - первые версии CMake, концепты пакета и find-модуля для find_package(), стат. анализ, LLVM, 64bit, valgrind, gtest
- 2010е - meson, conan, vcpkg, LSP, DAP, 3-летние релизы ISO C++, Tooling SG, первые версии cps/stdman, ninja build system и dyndep, санитайзеры, C++ Core Guidelines
- 2020е - модули C++, CPS, EcoStd, #embed и рефлексия

Станок Жаккарда

1950е: Fortran, компиляторы, магнитная лента

1970е: Си, Unix, Shell (tsh), Make, m4

- 1969г - зарождение Unix и Си
  - Процесс разработки Unix зародил концепт кросс-компиляции: Кен Томпсон писал на PDP7, но из-за отсутствия инструментов, она была пересобрана на GECOS (OS от General Electric, схожая с Multics - предком Unix) и перенесена с помощью перфоленты
- 1971г - первый Unix-shell - Thompson Shell (tsh)
  - Все еще актуален для cosmopolitan libc
  - примерно тогда сборка проектов превратилась в shell-скрипты
  - зависимости описывались скриптами или руками во флагах
- 1973г - первый интерактивный редактор ed
  - больше предназначался для телетайпов, работал построчно
  - до сих пор есть в posix
- 1976г - **Make**
  - Сборка теперь в виде набора правил (таргетов), которые образуют графы зависимостей
  - Работа с зависимостями трансформировалась - теперь можно было флаги подключения зависимости описать в мейкфайлах
- 1976г - **ex** и **Vi**
  - появился как развитие ed, чтобы можно было редактировать текст в “визуальном режиме” (во весь экран). Vi - был “визуальным режимом” для редактора ex
  - Частично вдохновлен редактором Bravo с Xerox Alto
- 1977г - m4
  - Расширенный макропроцессор, не зависящий от языка
  - Стал основой для сборки/кодогенерации
- 1978г - lint
  - Первый линтер для языка Си от Стивена Джонсона (Bell Labs)
- 1979-1982 - cb и indent - первые форматировщики кода
  - К indent приложил руку Джеймс Гослинг - в будущем создаст Java

Vi и его раскладка

Makefile: “структурированные скрипты”

- По сравнению со скриптами добавляют новую концепцию “правила сборки”, позволяя организовать их все в граф сборки, вместо полной пересборки
- Состоят из правил, которые содержат в себе имя таргета, опциональные зависимости и команда для shell
- Обычно зависимости в нем организовывались включением отдельных Make-файлов с флагами зависимости (дальше использовались потом libname-config, pkg-config - об этом позже)
- В настоящее время есть пачка эволюционировавших диалектов: GNU Make, bsd make, nmake, jom и пр.

Самый простой пример:

```
hello:
	$(CC) hello.c -o hello
```

Эпоха тулинга до C++: резюме

- Физический тулинг стал виртуальным: от библиотек перфокарт и работы с ними до shell-скриптов, программ-трансляторов, препроцессоров, линкеров, ассемблеров и первых систем сборки, а также первые редакторы
- Эволюционировал сам выходной артефакт: от перфокарты, до виртуальных артефактов: исходников/объектников/бинарных файлов
- Сформировались первые способы описания сборки проектов: shell-скрипты, мейкфайлы
- Зарождаются первые способы автоматизации и инкапсуляции работы с зависимостями: от shell-скриптов/ручного хардкода, до мейкфайлов с параметрами, вызовами m4 с заполнением макроподстановок в шаблонах
-

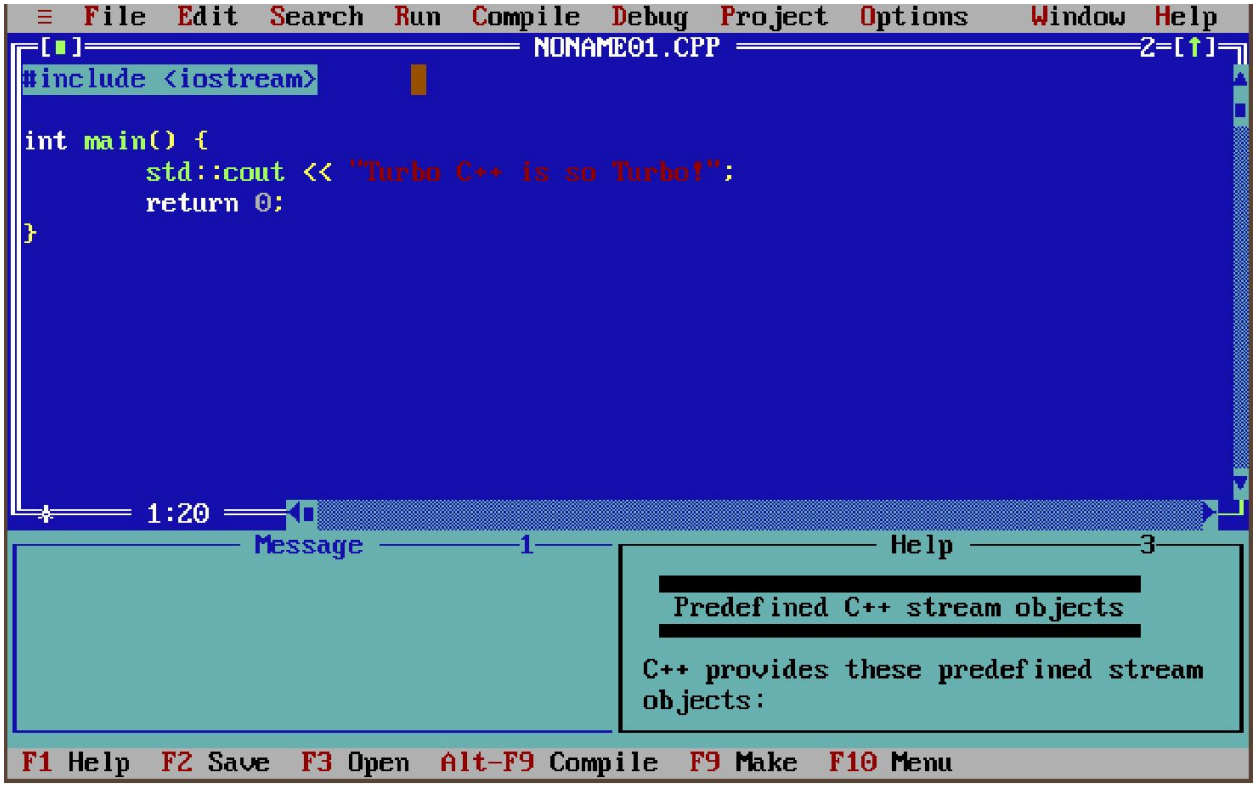
Краткая история: эпоха CFront и pre ISO C++

- Станок Жаккарда и первые механические вычислительные машины
- 1950е - зарождение компиляторов, Фортрана, магнитной ленты
- 1970е - Си, unix и скрипты оболочки, первые редакторы (ed, emacs, vi), make,m4,lint
- **1980е - cfront, autotools, ctags, libname-config-программы, posix**
- **1990е - C++, Unicode, Borland, Visual Studio, pkg-config, qt, boost, doxygen**
- 2000е - первые версии CMake, концепты пакета и find-модуля для find_package(), стат. анализ, LLVM, 64bit, valgrind, gtest
- 2010е - meson, conan, vcpkg, LSP, DAP, 3-летние релизы ISO C++, Tooling SG, первые версии cps/stdman, ninja build system и dyndep, санитайзеры, C++ Core Guidelines
- 2020е - модули C++, CPS, EcoStd, #embed и рефлексия

Этапы развития тулинга для C++: эпоха CFront

- 1980е: “Си с классами”, libname-config, ctags, первые IDE и стандарты на Unix - IEEE 1003.1-1988 (POSIX), GCC
  - Cfront - транслятор из “Си с классами” в Си
  - Aztec C - кросс-компилятор Си для CP/M, MS-DOS, Apple DOS, Amiga, Commodore 64
  - Турбо Си и Visual C и CodeView (отладчик) - первые интегрированные среды разработки, что изменят тулинг и вид IDE
    - Формат отладочных данных CodeView станет фундаментом для PDB
  - ctags - первая утилита для навигации по символам (определениям функций, переменных) в коде (BSD Unix, рубеж 1970-80х гг.)
  - **libname-config** - утилиты для инкапсуляции работы с зависимостями, получения флагов сборки, линковки
    - возникли как решение проблемы масштабируемости makefile-ов и скриптов и инкапсуляции работы с внешними зависимостями
  - Стандартизация Unix-подобных систем - IEEE 1003.1-1988 (POSIX)
    - Возникла как ответ на возросшую проблему несовместимости разношерстных *nix-подобных систем
  - 1988 г. - Первая версия отладочного формата DWARF

Турбо Си++


File Edit Search Run Compile Debug Project Options Window Help
NONAME01.CPP
#include <iostream>
int main() {
std::cout << "Turbo C++ is so Turbo!";
return 0;
}
1:20
Message 1
Help 3
Predefined C++ stream objects
C++ provides these predefined stream objects:
F1 Help F2 Save F3 Open Alt-F9 Compile F9 Make F10 Menu

libname-config-утилиты

- Позволяет инкапсулировать флаги для зависимости в логику исполняемого файла, по сравнению с прошлыми разными способами
- печатает нужные флажки в stdout, и обычно использовался в скриптах сборки (Makefile, configure.ac)
- Разные утилиты сначала предлагали разные параметры командной строки для запросов, постепенно унифицировались
- Встречаются до сих пор, см llvm-config, magick-config
- Для кросскомпиляции работают плохо (надо уметь исполнять файл кросс-компилируемой системы)

Пример вывода llvm-config

$ llvm-config --cxxflags

-I/usr/lib/llvm-19/include -std=c++17 -fno-exceptions -funwind-tables
-D_GNU_SOURCE -D__STDC_CONSTANT_MACROS
-D__STDC_FORMAT_MACROS -D__STDC_LIMIT_MACROS

Этапы развития тулинга для C++: pre-ISO C++

- 1990е: Всплеск разных диалектов Си, стандартизация C++, Юникод, pkg-config, рождение и смерть разных ОС, кроссплатформенность
  - Появляются первые версии IDE Borland и Visual Studio - автокомплит/IntelliSense/браузер объектов (классов, функций, и пр.)
  - Унификация тулинга для сборки и отладки с помощью кросс-компиляции
- Конец 1980-х - начало 1990х - **Autotools**
- 1995г. - Появилась Java
  - переиспользовала и трансформировала концепт класса
  - начала унифицировать “enterprise”-подходы к проектированию и тулингу, в частности, javadoc
- 1995-1996г - Qt
  - один из первых кроссплатформенных gui
  - интроспекция и рефлексия через дополнительный тулинг - утилиту moc (Meta Object Compiler)
  - qDoc - формат дальше будет взят doxygen-ом (оба написаны одним и тем же человеком: Димитри ван Хееш)
- 1997г - Doxygen: javadoc + qDoc
- 1998г - AStyle - “артистичное” форматирование кода
- 1999г - Boost
- 1999-2000 - **pkg-config**
  - Решает проблему кросс-компиляции с помощью декларативных .pc-файлов

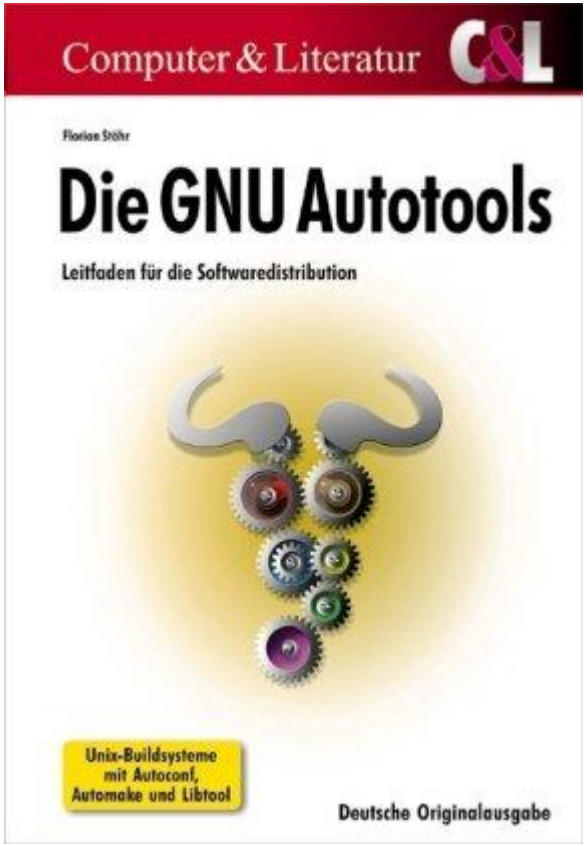
GNU Autotools

- Одна из первых массово распространенных мета-билдсистем: генерирует makefile-ы, уменьшая бойлерплейт
- Унифицировали базовые флаги для задания параметров сборки, такие как CC, cflags, ldflags (а также CXX, cxxflags)
- Унифицировали “двухэтапный” подход к сборке C/C with classes-проектов: ./configure && make
  - На первом этапе Autotools на основе своего DSL генерирует Makefile
  - А на втором этапе make собирает все цели (служебная цель all)
- За генерацию configure-скрипта отвечает команда **autoreconf -i**, работающая с файлом configure.ac**,** описание исходников - Makefile.am

Пример configure.ac

```
AC_PREREQ([2.69])

AC_INIT([myproject], [1.0], [you@example.com])

AM_INIT_AUTOMAKE([-Wall -Werror foreign])

AC_PROG_CC


AC_CONFIG_FILES([Makefile src/Makefile])

AC_OUTPUT
```

GNU Autotools все еще используются…



Эпоха Cfront-pre-ISO C++: Резюме

- Инструментарий сделал резкий скачок вперед: от текстовых редакторов в текстовом режиме (vi, emacs, ed, edit) до интегрированных сред в текстовом графическом режиме (Turbo C, Visual Studio, Borland C++ Builder) с контекстной справкой, автокомплитом и пр.
- C++ прошел большой путь развития от CFront до полноценных компиляторов (gcc, edg, msvc и прочие)
- Начали зарождаться и развиваться современные C++-фреймворки, повлиявшие на тулинг языка и на сам язык в целом: boost (bjam/b2 и часть стд. библиотеки), Qt (moc)
- Некоторые конвенции описания окружения унифицировались (в основном в unix-подобных системах и окружениях): CC, CXX, cflags
- Возникшее разнообразие различных ОС подняло проблему кросс-компиляции

pkg-config

Первая распространенная утилита, предложившая декларативный путь подключения зависимостей в виде текстовых файлов с расширением .pc

Типичная структура .pc-файла состоит из блока переменных и пар “ключ: значение”, и он генерируется по шаблону (например, через m4)

Пример файла .pc для pkg-config

```
prefix=/usr/local

exec_prefix=${prefix}

includedir=${prefix}/include

libdir=${exec_prefix}/lib


Name: foo

Description: The foo library

Version: 1.0.0

Cflags: -I${includedir}/foo

Libs: -L${libdir} -lfoo
```

pkg-config: недостатки

- Прибиты гвоздями к префиксу установки, что привязывает к FHS-иерархии
- Подразумевает, что все используют один компилятор и линкер, что не всегда бывает правда (gcc + clang например)
- Требуют совместимости по флагам, что приводит к несовместимости с разными фронтэндами компиляторов
- Плохо масштабируются на фреймворки библиотек с наборами компонентов типа Qt, Boost (много бойлерплейта)

00-е: развитие других языков, IDE и ожидание C++0x

Этапы развития C++-тулинга: новое тысячелетие

- 00-е годы: CMake, LLVM, Visual Studio, 64 бита и развитие статического анализа
  - **2000 г. - CMake** - упрощение кроссплатформы, абстрагирование сборки между разными генераторами
  - 2001г. - Eclipse - повлиял на фичи рефакторинга в MSVC и в последующем тулинге
    - использовал тулинг java (JDT), и создал свои инструменты, привязанные к IDE
  - 2003 г. - LLVM - исследовательский проект, ставший конструктором для компиляторов и C++-тулинга (clangd, clang-tidy и пр.)
  - 1997-2003г. - Visual Assist X - проприетарное расширение, “чинящее” рефакторинг и навигацию кода в Visual Studio
  - 2004 г. - Resharper - мощный плагин для рефакторинга кода на C#
    - в 2015-м добрался и до C++
  - 2005 г. - Uncrustify - экстремально-настраиваиваемый форматировщик C/C++-кода
  - 2005 г. - Visual Studio 2005 - поддержка 64 бита
    - Новые диагностики для миграции
    - Пришлось править даже заголовочники winapi (LONG -> LONG_PTR)
  - 2006 г. - Viva64 - первый (проприетарный) стат. анализатор, подсветивший проблемы миграции на 64 бита
  - 2006 г. - Rust. Nuff said.
  - 2007 г. - cppcheck - развитие идей линтинга языка

CMake

CMake: find-модули, таргеты и наследование

- Первоначально зародился в Kitware как результат публикации внутренней кроссплатформенной билд-системы (с поддержкой windows)
- Развивает концепцию двухэтапной сборки из autotools (configure/make) и некоторые другие вещи
- Расширяет концепцию таргета мейкфайлов - теперь таргет может хранить различные свойства, которые имеют семантическую роль (INCLUDE_DIRECTORIES, LIBRARIES, DEFINITIONS)
- В середине нулевых появилась концепция “usage requirement” - наследования свойств целей (PRIVATE, PUBLIC, INTERFACE)
- Развил концепцию pc-файлов для подключения зависимостей - **Find-модули** и **cmake-пакеты**

CMake и find_package(): Find-модули

Специальный CMake-скрипт (с именем `Find<libname>.cmake`), который ищет артефакты библиотеки по заданным путям (например, заголовок uuid.h или librt), и запаковывает это в CMake-цели (и в переменные вида <libname>_INCLUDES, <libname>_LIBS и пр., что не рекомендуется)

- пишется автором проекта для адаптации не-CMake-библиотек;
- Для некоторых распространенных библиотек есть Find-модули в самом CMake;
- для добавления своего каталога с модулями надо добавить путь до него в переменную CMAKE_MODULE_PATH: list(APPEND CMAKE_MODULE_PATH ./cmake/modules)

Пример Find-модуля

```
# 1. Поиск библиотеки и сохранения пути в переменную UUID_LIBRARY
find_library(UUID_LIBRARY uuid REQUIRED)
# 2. Поиск ее заголовочного файла для передачи путей в include-ы
find_file(UUID_H uuid.h PATHS /usr/include REQUIRED)
# 3. Получение каталога из имени файла
file(DIRECTORY UUID_H UUID_INCLUDE_DIR)
# 4. Создаем специальный таргет, который не будет собираться компилятором
add_library(uuid::uuid UNKNOWN IMPORTED)
# 5. Добавляем путь до найденного инклуда к цели. этот путь будет наследоваться другими
target_include_directories(uuid::uuid INTERFACE ${UUID_INCLUDE_DIR})
# 6. Говорим CMake'у, что у таргета есть предсобранный артефакт, и устанавливаем путь
set_target_properties(uuid::uuid PROPERTIES IMPORTED_LOCATION ${UUID_LIBRARY})
```

CMake: Резюме

- Трансформация некоторых концептов: из “таргет” - “выполни эти команды” к “таргет” - набор наследуемых свойств и выполняемых команд
  - Старые “таргеты” с вызовом команд можно делать через add_custom_target()
  - Для подключения предсобранных библиотек есть импортированные таргеты
- Приход к кроссплатформенности, появление генераторов в родные системы сборки
- Наследование свойств через линковку таргетов, вместо работы с глобальными переменными
- Инкапсуляция работы с зависимостями в цепочки install() <-> find_package() или ExternalProject/fetch_content
- Первоначальная абстракция от флагов сборки через “compile_features” (похожее было в msbuild-проектах)

CMake и find_package(): пакеты CMake

Автосгенерированные скрипты самим CMake-ом на этапе установки библиотеки с помощью команды install()

- пишутся авторами библиотек путем добавления пары команд для экспорта целей библиотек
- ищутся в CMAKE_PREFIX_PATH
- также можно указать точный путь до КАТАЛОГА нужной либы через <libname>_DIR
- С CMake 3.11 можно также указать корневой каталог, куда была установлена библиотека через <libname>_ROOT

Генерация CMake-пакетов: функция install() и наборы экспортов

- install() позволяет устанавливать файлы, артефакты сборки, задавать кастомную логику и **генерировать пакеты:**
  - install(TARGETS <targets> *EXPORT* <*export-name*> DESTINATION <path>)
  - install(EXPORT <export-name> DESTINATION <path>)
- Для портабельных путей рекомендуется использовать встроенный модуль **GNUInstallDirs**, и его переменные CMAKE_INSTALL_<smt>DIR (smt - LIB, BIN, INCLUDE)

Установка библиотек и CMake-пакета

```
# 1. Подключаем модуль, задающий стандартные пути для установки, совместимые с FHS и GNU
include(GNUInstallDirs)
# 2. Говорим CMake'у, что таргет lib1 будет лежать в наборе экспортов lib1-config
#    и что бинарные артефакты надо установить в стандартный путь CMAKE_INSTALL_LIB_DIR
install(TARGETS lib1 EXPORT lib1-config DESTINATION ${CMAKE_INSTALL_LIBDIR})
# 3. Устанавливаем инклуды нашей либы
install(FILES ./include/lib1/lib1.h DESTINATION ${CMAKE_INSTALL_INCLUDEDIR}/lib1)
# 4. Генерация и установка ПАКЕТА. Те, кто будут подключать библиотеку через find_package(),
#    смогут ее использовать потом через имя с пространством имен lib1::
install(EXPORT lib1-config NAMESPACE lib1:: DESTINATION ${CMAKE_INSTALL_LIBDIR}/cmake/lib1)

# пример потребления:
# find_package(lib1 REQUIRED)
# target_link_libraries(myapp lib1::lib1)
```

CMake: недостатки

- Документация до некоторого момента была “reference-”only и experts friendly
- DSL довольно bugprone, и близок по сути к shell, autotools, чем к скриптовым языкам вида python, perl, ruby
- Слишком много примеров, написанных с использованием старых версий, что приводит к менее поддерживаемому коду и копипасту старых подходов
- Трудно отлаживать (до 3.29, в котором появился отладчик через Debug Adapter Protocol)
- Последняя версия 4.0+ дропнула поддержку до CMake 3.5 (неудобство для мейнейнеров, лечится переменной окружения CMAKE_MINIMUM_POLICY_REQUIRED=3.5)

2010е: пакетники, санитайзеры, свежие стандарты

- 2011г.: санитайзеры для ловли проблем с памятью
  - были интегрированы в gcc и clang в ~2012-2013гг., MSVC - 2019
  - подробнее - доклады Алексея Веселовского на cppconf
- C++11 и дальше - переход на трехлетнюю модель релизов стандартов и создание разных групп по работе с разными областями языка
  - зарождение **SG Tooling**
- LLVM набирает популярность
  - развиваются инструменты на libtooling для стат. анализа, форматирования (2013 г.) и пр.
- 2010г.: **meson** - новая (мета)система сборки, распространившаяся в GNOME/linux-проектах
  - Исправляет условный “недостаток” CMake - его DSL
- 2011г. **ninja-build** - система сборки от Google
  - была придумана для ускорения сборки Chromium
  - DSL больше предназначен для кодгена метасистемами сборки
  - в 2013г. научилась в dyn deps - механизм для поиска динамических зависимостей исходника, например, import module (в C++20)

2010е: пакетники, санитайзеры, свежие стандарты (ч.2)

- 2016 г: Conan v0.x - пакетный менеджер для C++
  - умеет работать с разными системами сборки
  - до этого был biicode от тех же авторов, наработки которых пошли в новый
- 2016г.: LSP - language server protocol
  - позволяет отцепить логику автодополнения, рефакторингов от логики IDE/редакторов
  - Работает поверх JSON-RPC-подобного протокола
- 2017г.: CPS - манифест описания зависимостей
  - рассмотрим позже
- 2018г.: vcpkg
  - основан на cmake-портах
  - имеет декларативную структуру
  - для подключения нужен cmake-тулчейн и манифест vcpkg.json (пример будет далее)

Meson

- Переиспользует часть концепций CMake (таргеты, наследования свойств)
- Использует свой DSL, похожий на python
- Используется в экосистеме GNOME и в некоторых linux-библиотеках
- Постепенно эволюционировала в мультиязыковую билд-систему (есть биндинги для go, rust)
- Все еще есть некоторые детские ошибки, которые в CMake уже давно починили (некоторый хардкод флагов, компонентов и т.п.)
- Хуже работает с Windows

Meson: dependency()

- Команда dependency() позволяет искать зависимости, используя:
  - pkg-config
  - системные зависимости
  - cmake
  - внешний subproject() из wrapdb.build
  - internal-зависимости - концептуально близки к встроенным Find-модулям cmake
    - некоторые из них умеют использовать libname-config
- Пример подключения зависимостей:
  - zdep = dependency(‘zlib’, optional : True) # опциональная зависимость
  - llvm_dep = dependency('llvm', version : '>=4.0') # может использовать libname-config
  - exe = executable('zlibprog', 'prog.c', dependencies : zdep)

Meson и wrapDB

- Позволяет адаптировать сборку зависимости в поддерживаемую нативно Meson-ом
  - Подключается как subproject() мезона, что позволяет их аккуратно пробросить внутрь проекта
  - В отличие от других пакетов, позволяет полностью кастомизировать сборку и подключение зависимости
  - Устанавливается через meson wrap install <package-name>, и появляется в подкаталоге “subprojects”
- Файл пакета wrapDB - имеет декларативную структуру INI файла

Пример пакета wrapDB для внешней зависимости

[wrap-file]

directory = libfoobar-1.0

source_url = https://example.com/foobar-1.0.tar.gz

source_filename = foobar-1.0.tar.gz

source_hash = 5ebeea0dfb75d090ea0e7ff84799b2a7a1550db3fe61eb5f6f61c2e971e57663

Пакетники для C++: Conan

Появился в 2016 г, на основе прошлого пакетника biicode

Для подключения зависимостей предлагает декларативный подход (conanfile.txt) и императивный (conanfile.py)

Умеет генерировать пакеты/конфиги/скрипты для разных билд-систем из *рецепта* для библиотеки, написанного на Python;

Позволяет описывать окружение с помощью *профилей сборки* (в том числе и кросскомпиляцию)

Имеет открытый репозиторий пакетов (*рецептов*) - conan-center-index

Пример conanfile.txt

```
[requires]
boost/1.86.0
reflect-cpp/0.18.0
msgpack-cxx/7.0.0
structopt/0.1.3
spdlog/1.15.0

[test_requires]
benchmark/1.9.1
gtest/1.15.0

[generators]
CMakeDeps

[options]
reflect-cpp/*:with_yaml=True
reflect-cpp/*:with_json=True
reflect-cpp/*:with_msgpack=True

spdlog/*:header_only=True
```

Пример рецепта (немного сокращен)

```
class FmtConan(ConanFile):
    name = "fmt"
    description = "A safe and fast alternative to printf and IOStreams."
    license = "MIT"
    url = "https://github.com/conan-io/conan-center-index"
    homepage = "https://github.com/fmtlib/fmt"
    topics = ("format", "iostream", "printf")
    package_type = "library"
    settings = "os", "arch", "compiler", "build_type"
    options = { "header_only": [True, False], }
    default_options = { "header_only": False, }

    def source(self):
        get(self, **self.conan_data["sources"][self.version], strip_root=True)

    def build(self):
        cmake = CMake(self)
        cmake.configure()
        cmake.build()

    def package(self):
        cmake = CMake(self)
        cmake.install()

    def package_info(self):
        target = "fmt-header-only" if self.options.header_only else "fmt"
        self.cpp_info.set_property("cmake_file_name", "fmt")
        self.cpp_info.set_property("cmake_target_name", f"fmt::{target}")
```

Пакетники для C++: vcpkg

- Появился примерно в 2018-м году как опенсурсный проект Microsoft
- Использует CMake для описания работы с зависимостями (“*порты*”)
- Переиспользует CMake-возможности работы с пакетами, переопределяя пути поиска пакетов через свой тулчейн,
- Имеет два режима работы: классический (в стиле системных пакетников) и режим манифеста (vcpkg.json)
- В декларативном режиме умеет в переключабельные наборы библиотек (“фичи”)

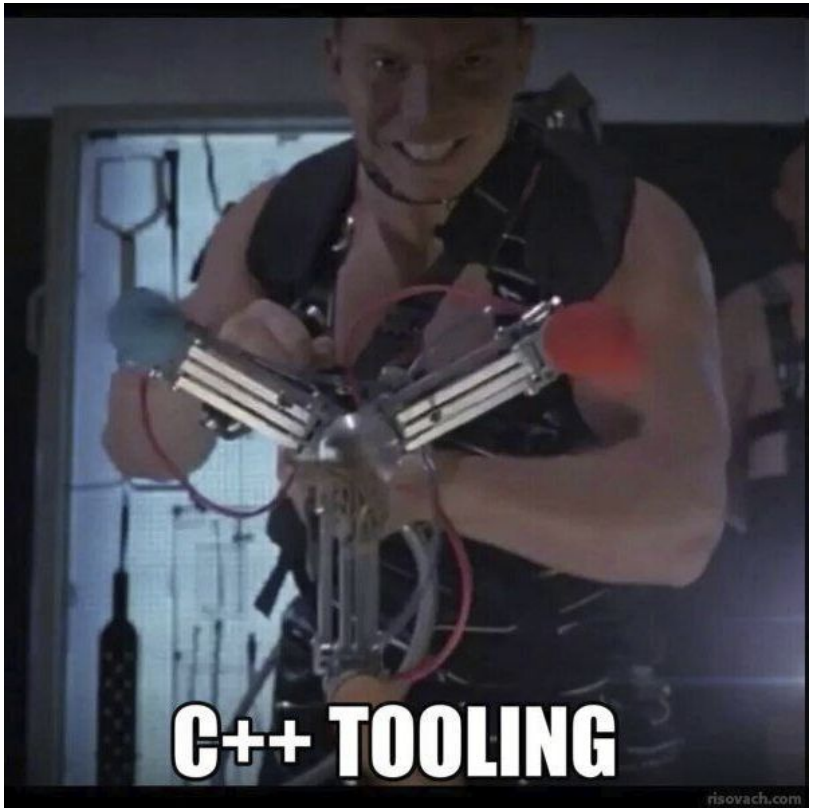
Пример vcpkg.json

```
{
    "name": "tracy",
    "version-semver": "0.8.0",
    "description": "C++ frame profiler",
    "homepage": "https://github.com/wolfpld/tracy",
    "dependencies": [
        { "name": "capstone", "features":[ "arm", "arm64", "x86" ] },
        "freetype",
        "glfw3"
    ]
}
```

2000-2010е гг.:Резюме

- C++ стал развиваться быстрее (3-годичный интервал между стандартами)
- Развитие новых ОС, не совместимых с POSIX требовало нового тулинга для работы с ними
- Появление попыток в декларативные, повторно используемые конфигурационные файлы - pkg-config, Find<lib>.cmake, <lib>-config.cmake
- Появление билд-систем, изменивших работу со сборкой и зависимостями, эволюция которых продолжается по сей день
- Зарождение первых пакетных менеджеров для C/C++: Conan, vcpkg
- Развитие тулинга и наследование концепций из других языков программирования

Тулинг: тогда и сейчас



Tooling SG

- Долгое время работали над бумагами с модулями
- Часть других бумаг переносилось в пользу модулей и import std
- В 2023-2024 гг. трансформировались в C/C++ Ecosystem Standard (EcoStd) - более открытый формат разработки спецификации, не зависящий от процессов ISO и самого C++
- Примеры бумаг:
  - Поддержка SARIF для тулинга (структированный формат логов)
  - виды модулей (см. доклад Антона Полухина про фичи C++20 и борьбу за модули)
  - интероп с системами сборки (Kitware, **P1689**)
  - файлы ответа компиляторов (rsp-файлы) для унифицированной передачи длинных контекстов/флагов сборки
  - Метаинформация об инструменте (tools introspection): –std-info
    - позволяет узнать у компилятора, поддерживает ли он такой-то флаг
    - позволяет заставить компилятор использовать “мета-фичу” - концептуально близко к compile features’
  - Описание наборов окружений как “tuplets” (p2073)

EcoStd сейчас

- работа над частями SBOM (purl - package url)
- Работа над общей открытой спецификацией тулинга
  - ecostd.github.io
  - Процесс добавления предложений - открытый - через RFC
- Работа над CPS и различными проблемами интеропа систем сборки и пакетных менеджеров
- Проработка некоторых проползалов стандартного C++
  - std::breakpoint, std::is_debugger_present, std::embed

Ближайшее будущее: CPS (Common Package Spec.)

- Декларативный формат, основан на формате JSON, и имеет JSON-схему
- Прорабатывает недостатки pkg-config (привязку к флагам компилятора, и др), и cmake-пакетов (привязка к CMake, и сложный код сгенерированных пакетов)
- Переиспользует концепции из CMake:
  - Компоненты и транзитивное наследование (boost::threads)
  - Фичи сборки
- Расширяем на другие языки: Фортран, Java;
- Рассчитан на генерирование и потребление тулингом
- Поддерживается в conan (пока экспериментально), CMake v4.3+, Meson (экспериментально)

Пример CPS-файла

```
{
  "name": "sample",
  "description": "Sample CPS",
  "license": "BSD",
  "version": "1.2.0",
  "compat_version": "0.8.0",
  "platform": {
    "isa": "x86_64",
    "kernel": "linux",
  },
  "configurations": [ "optimized", "debug" ],
  "default_components": [ "sample" ],
  "components": {
    "sample-core": {
      "type": "interface",
      "definitions": [ "SAMPLE" ],
      "includes": [ "@prefix@/include" ]
    },
    "sample": {
      "type": "dylib",
      "requires": [ ":sample-core" ],
      "configurations": {
        "optimized": {
          "location": "@prefix@/lib64/libsample.so.1.2.0"
```

Работа с CPS: подключение зависимостей

1. CMake: прозрачно работает через find_package(libname)
2. Meson: dependency() - экспериментально (на момент мая 2026г)
3. Makefile/shell-скрипты/прочее: вызов утилиты cps-config (drop-in для pkg-config)

Работа с CPS: генерация манифеста в CMake

install(**PACKAGE_INFO lib1** EXPORT **lib1-config**)

- Также есть кастомизация тех полей, что мы видели раньше, но они наследуются от project()

Работа с CPS в Conan: WIP

- Генератор CMakeConfigDeps: следующая итерация для интеропа с CMake
  - Все еще пока экспериментальный
- Для CPS используйте Conan 2.25+ (вышла в январе 2026 года)
- CPSDeps - незадокументированный генератор
  - Conan 2.26+
  - Не требует cmake со стороны потребителя
  - пока не готов :(

Другие системы сборки, что мы не посмотрели:

Bazel - использует свои модули для зависимостей, и имеет свой реестр тут: https://registry.bazel.build/ . Также можно использовать conan-интеграцию

MSBuild - старые проекты могут вынести всякие флаги зависимостей в .props-файлы, новые проекты могут использвать vcpkg-интеграцию или conan-интеграцию

Gradle - может использовать cmake-интеграцию свою, в которую можно затащить conan/vcpkg

Заключение

Очень много инструментов, что актуальны до сих пор, зародились во времена хиппи и эпохи юникс (и не всегда подходят к другим окружениям)

Новые инструменты возникают, улучшая в целом DX, но многие из них отрицают старое, что приводит к сосуществованию нескольких тулзов

Другие языки в какой-то мере “учатся” на ошибках экосистемы C/C++ и выпускают сразу нужный инструментарий (пакетники, системы сборки, линтеры, форматировщики). Удачные идеи адаптируются плюсовым тулингом обратно (doxygen, clang-tidy, conan, vcpkg, bazel)

Унификация инструментов, несмотря на большое число легаси, продолжается, и в ближайшем будущем будет ускоряться

Сцена после титров: список бумаг, связанных с EcoStd

2018, P1040R0, std::embed, SG15, Evolution, Library Evolution

2018, P1052R0, Modules, Macros, and Build Systems, SG15, Evolution

2018, P1067R0, C++ Dependency Management: Package Consumption vs Development, SG15

2018, P1177R1, Package Ecosystem Plan, SG15

2018, P1178R0, C++ Compile, SG15, Library Evolution

2018, P1204R0, Canonical Project Structure, SG15

2018, P1254R0, Notes on C++ Package Management, SG15

2018, P1275R0, Desert Sessions: Improving hostile environment interactions, SG15, SG16, Library Evolution

2018, P1281R0, Feature Presentation, SG15

2018, P1300R0, Remember the FORTRAN, SG15, Evolution

2018, P1313R0, Let's Talk About Package Specification, SG15

2019, P1482R0, Modules Feedback, SG15, Evolution

2019, P1484R1, A uniform and predefined mapping from modules to filenames, SG15

2019, P1634R0, Naming guidelines for modules, SG15

2019, P1687R1, Summary of the Tooling Study Group's Modules Ecosystem Technical Report Telecons, SG2, SG15, Evolution

2019, P1688R0, Towards a C++ Ecosystem Technical Report, SG15, Evolution

2019, P1767R0, Packaging C++ Modules, SG15

2019, P1788R3, Reuse of the built modules (BMI), SG15, Core

2019, P1832R0, Improving Debug Builds Inline With User Expectation, SG2, SG14, SG15

2019, P1842R0, Generalized Module (Dependency?) Mapper, SG15

2019, P1845R0, 2019-09-21 Denver Tooling Meeting, SG15

2019, P1876R0, All The Module Names, SG15

2019, P1905R0, In-Source Mechanism to Identify Importable Headers, SG15, Evolution

2019, P1948R0, Modules: Keep the dot, SG2, SG15, Evolution

2020, P1184R2, A Module Mapper, SG15

2020, P1838R0, Modules User-Facing Lexicon and File Extensions, SG15

2020, P1857R3, Modules Dependency Discovery, SG2, SG15, Core

2020, P1864R0, Defining Target Tuplets, SG15

2020, P2073R0, Debugging C++ coroutines, SG15

Продолжение большого списка проползалов

2020, P2074R0, Asynchronous callstacks & coroutines, SG15

2021, P2409R0, Requirements for Usage of C++ Modules at Bloomberg, SG15

2021, P2473R1, Distributing C++ Module Libraries, SG15

2022, P1689R5, Format for describing dependencies of source files, SG15, SG16

2022, P2429R0, Concepts Error Messages for Humans, SG15

2022, P2514R0, std::breakpoint, SG15, Library Evolution

2022, P2515R0, std::is_debugger_present, SG15, Library Evolution

2022, P2536R0, Distributing C++ Module Libraries with dependencies json files., SG15

2022, P2546R0, Debugging Support, SG15, Library Evolution

2022, P2565R0, Supporting User-Defined Attributes, SG15

2022, P2577R2, C++ Modules Discovery in Prebuilt Library Releases, SG15

2022, P2581R2, Specifying the Interoperability of Built Module Interface Files, SG15

2022, P2673R0, Common Description Format for C++ Libraries and Packages, SG15

2022, P2701R0, Translating Linker Input Files to Module Metadata Files, SG15

2022, P2702R0, Specifying Importable Headers, SG15

2023, P2656R1, C++ Ecosystem International Standard, SG15

2023, P2717R5, Tool Introspection, SG15 Tooling

2023, P2791R0, mandate concepts for new features, SG15 Tooling

2023, P2800R0, Dependency flag soup needs some fiber, SG15 Tooling

2023, P2803R0, std::simd Intro slides, SG15 Tooling

2023, P2898R1, Build System Requirements for Importable Headers, SG15 Tooling

2023, P2962R0, Communicating the Baseline Compile Command for C++ Modules support, SG15 Tooling

2023, P2978R0, A New Approach For Compiling C++, SG15 Tooling

2023, P2990R0, C++ Modules Roadmap, SG15 Tooling

2023, P3033R0, Should we import function bodies to get the better optimizations?, SG15 Tooling

2023, P3034R0, Module Declarations Shouldn't be Macros, SG15 Tooling,Evolution

2023, P3041R0, Transitioning from "#include" World to Modules, SG15 Tooling

2023, P3057R0, Two finer-grained compilation model for named modules, SG15 Tooling

2024, P2977R2, Build database files, SG15 Tooling

2024, P3051R1, Structured Response Files, SG15 Tooling

Вообще огромный список (часть 3)

2024, P3081R0, Core safety Profiles: Specification, adoptability, and impact, SG15 Tooling,SG23 Safety and Security,EWG Evolution

2024, P3092R0, Modules ABI requirement, SG15 Tooling,ARG ABI Review Group

2024, P3267R1, Approaches to C++ Contracts, SG15 Tooling,SG21 Contracts

2024, P3286R0, Module Metadata Format for Distribution with Pre-Built Libraries, SG15 Tooling

2024, P3321R0, Contracts Interaction With Tooling, SG15 Tooling,SG21 Contracts,EWG Evolution

2024, P3335R3, Structured Core Options, SG15 Tooling

2024, P3358R0, SARIF for Structured Diagnostics, SG15 Tooling

2024, P3470R0, Interface-Unit-Only Module Library Support, SG15 Tooling,EWG Evolution

2025, P3696R0, Discovering Header Units via Module Maps