الطبعة الثانية

إصدار 2026
CMake 4.4.1

إتقان CMake الحديث

الدليل المرجعي السريع لبناء مشاريع C++ قوية وقابلة للتوسع عبر المنصات

CMakeLists.txt

```
cmake_minimum_required(...)
project(... LANGUAGES CXX)
add_executable(app ...)
```

Targets

```
app (Executable)
PRIVATE core
PUBLIC fmt::fmt
```

Presets

```
configurePresets
buildPresets
testPresets
workflowPresets
```

Generators

```
Ninja
Visual Studio
Xcode
Unix Makefiles
```

- **جاهز للمشاريع** — اختبار وتكامل وتغليف
- **أمثلة عملية** — مجربة وقابلة للتشغيل
- **معلومات موثقة** — من الوثائق الرسمية للأداة
- **أفضل الممارسات الحديثة** — أهداف وإعدادات حديثة

إعداد

أيمن الحراكي

simplifycpp.org

مرجع عربي مختصر وعملي لمبرمجي البرمجيات

إتقان CMake الحديث

الدليل المرجعي السريع

الطبعة الثانية - يوليو 2026

محدّث وفق الوثائق الرسمية للإصدار CMake 4.4.1

إعداد **أيمن الحراكي**

مؤسس منصة simplifycpp.org

هذا الكتيب المختصر موجّه إلى مبرمجي C++ الذين يريدون بناء مشاريع موثوقة واختبارها وتثبيتها وتحزيمها ونشرها، من دون أن تتحول ملفات `CMakeLists.txt` إلى مشروع برمجي آخر يحتاج إلى صيانة مستقلة.

قدمت الطبعة الأولى عرضًا عامًا للأداة، أما هذه الطبعة فهي مراجعة تقنية كاملة: حُذفت التفسيرات غير الدقيقة أو الملتبسة، واستُبدلت الممارسات القديمة بالأساليب الحديثة، وأعيدت صياغة الأوامر حول مجلدي المصدر والبناء، وصُححت أمثلة تصدير الحزم، كما بُنيت المشاريع الأساسية واختُبرت ضمن حزمة تحقق آلية.

بيئة التحقق الأساسية للأمثلة القابلة للتشغيل

- CMake 3.31.6
- Ninja 1.12.1
- GCC 14.2.0
- Linux x86-64

تستخدم الأمثلة -حيثما أمكن- حدًا أدنى هو **CMake 3.25** لتظل نافعة على الأنظمة المنتشرة فعليًا. وتُذكر بوضوح الميزات التي تتطلب CMake 4.2 أو 4.3 أو 4.4. أما الشروح وملاحظات الإصدارات فمبنية على الأدلة الرسمية وملاحظات إصدار CMake 4.4.1.

المقدمة

يسهل البدء باستخدام CMake، لكن إساءة استخدامها أسهل مما يتوقع كثير من المبرمجين. قد يكون مشروع من ثلاثة أسطر صحيحًا تمامًا، في حين يظل نظام بناء إنتاجي عمره عشر سنوات هشًا؛ لأنه يعتمد على خيارات شاملة للمجلد، أو حالة خفية محفوظة في الذاكرة المخبأة، أو مسارات مكتبات منسوخة يدويًا، أو أوامر بناء لا تعمل إلا داخل صدفة طرفية محددة.

الفكرة المركزية في CMake الحديثة بسيطة:

> صِف **الأهداف** ومتطلبات استخدامها، ولا تحاول محاكاة سطر أوامر المصرّف يدويًا.

عندما تصبح هذه الفكرة طبيعية، تبدأ موضوعات تبدو متفرقة بالارتباط ضمن نموذج واحد. فمسارات التضمين، ومعايير اللغة، وتعريفات المعالج القبلي، والمكتبات المرتبطة، ومجموعات ملفات الترويسات، وقواعد التثبيت، والحزم المصدّرة، والاختبارات، وتكامل بيئات التطوير، كلها تصبح خصائص مرتبطة بالأهداف. بعد ذلك تتولى CMake تحويل هذه الخصائص إلى تعليمات مناسبة لـ Ninja أو Make أو Visual Studio أو Xcode أو FASTBuild أو مولّد آخر.

هذا الكتاب مختصر عمدًا، لكنه ليس قائمة أوامر بلا تفسير. كل فصل يجيب عن ثلاثة أسئلة:

١. ما المشكلة التي تحلها هذه الميزة؟

٢. ما الصيغة الحديثة المعتمدة على الأهداف؟

٣. ما الخطأ الذي ينبغي لمبرمج C++ تجنبه؟

الأمثلة قصيرة بما يكفي لكتابتها يدويًا، لكنها منظمة بحيث تتوسع الأفكار نفسها إلى مكتبات وتطبيقات تضم مئات الأهداف. وتحتوي حزمة الأمثلة القابلة للتنزيل على الملفات الكاملة المستخدمة في الكتاب وعلى برنامج تحقق آلي.

كيف تقرأ هذا الدليل؟

اقرأ الفصول من 1 إلى 6 بالترتيب إن كانت CMake جديدة عليك. أما المستخدم الخبير فيمكنه البدء من الفصل 4 لفهم متطلبات الاستخدام، أو الفصل 7 لإدارة التبعيات، أو الفصل 9 لبناء الحزم القابلة للتثبيت، أو الفصل 16 للانتقال من أساليب CMake القديمة.

الأوامر الموسومة بعبارة **الحد المحمول** تعمل مع CMake 3.25 أو أحدث. أما الأوامر الموسومة **CMake 4.4** فتتطلب الإصدار الرئيسي الحديث الذي توثقه هذه الطبعة.

ما الذي تغير عن الطبعة الأولى؟

تصحح الطبعة الجديدة نقاط ضعف مهمة، من أبرزها:

- تقديم مجلدي المصدر والبناء باستخدام `cmake -S` و`-B` لإزالة الالتباس المتعلق بمجلد العمل الحالي.
- الفصل الواضح بين المولّدات أحادية الإعداد والمولّدات متعددة الإعدادات.
- شرح `PUBLIC` و`PRIVATE` و`INTERFACE` بوصفها نطاقات لمتطلبات الاستخدام، لا كلمات عامة للرؤية أو الوصول.
- استخدام `install(EXPORT)` و`configure_package_config_file()` والمسارات القابلة للنقل في أمثلة الحزم، بدل إنشاء أهداف مستوردة مكتوبة يدويًا.

• حذف أسلوب Conan القديم المعتمد على `conanbuildinfo.cmake` واستبداله بتكامل Conan 2 الحديث.

• جعل التحذيرات وخيارات التحسين مرتبطة بالأهداف ومراعية لاختلاف المصرّفات.

• عدم تقديم `file(GLOB)` بوصفها الطريقة الطبيعية لجمع ملفات المصدر.

• تغطية إعدادات CMake المسبقة لخطوات الإعداد والبناء والاختبار والتحزيم وسير العمل.

• إضافة تغطية مركزة لمجموعات الملفات ووحدات C++ والقياس التشغيلي والتشخيص وCPS والتغييرات المهمة في CMake 4.2 إلى 4.4.

المحتويات

النموذج الذهني لـ CMake

CMake مولّد لأنظمة البناء

لا تقوم CMake عادةً بترجمة شيفرة C++ بنفسها. فهي تقرأ وصف المشروع، ثم تولّد تعليمات لأداة بناء أخرى. عند اختيار Ninja تنشئ ملف `build.ninja`، وعند اختيار Unix Makefiles تنشئ ملفات Makefile، أما Visual Studio وXcode فتولّد لهما معلومات المشروع التي تفهمها بيئتهما.

هذا الفرق يفسر سير العمل المعتاد:

```
CMakeLists.txt + toolchain + cache options
                  |
              configure
                  |
               generate
                  |
       Ninja / Make / MSBuild / Xcode
                  |
                build
```

تُنفّذ مرحلتا الإعداد والتوليد عادةً بأمر واحد:

```
cmake -S . -B build
```

ثم تُستدعى مرحلة البناء من خلال واجهة CMake المستقلة عن المولّد:

```
cmake --build build
```

يُفضّل استخدام `cmake --build` في الوثائق والبرامج النصية؛ لأن الأمر يعمل سواء كانت أداة البناء المولّدة Ninja أو Make أو MSBuild أو Xcode أو غيرها.

> **قاعدة عملية:** تعامل مع مجلد البناء على أنه حالة مولّدة يمكن حذفها في أي وقت. يجب أن تبقى شجرة المصدر مكتملة حتى بعد حذف جميع مجلدات البناء.

المجلدات الأربعة التي يجب التمييز بينها

تستخدم CMake كلمتي *source* و*binary* كثيرًا. والمقصود بـ *binary directory* هنا هو **شجرة البناء**، وليس فقط المجلد الذي يوجد فيه الملف التنفيذي النهائي.

- `CMAKE_SOURCE_DIR`: مجلد المصدر الأعلى لعملية CMake الحالية.
- `CMAKE_BINARY_DIR`: مجلد البناء الأعلى.
- `CMAKE_CURRENT_SOURCE_DIR`: مجلد المصدر الجاري معالجته الآن.
- `CMAKE_CURRENT_BINARY_DIR`: مجلد البناء المقابل لملف `CMakeLists.txt` الحالي.

في المشاريع الفرعية القابلة لإعادة الاستخدام تكون متغيرات المجلد الحالي أكثر أمانًا غالبًا من متغيرات المستوى الأعلى. فالمكتبة المضافة بواسطة `add_subdirectory()` لا ينبغي أن تفترض أنها تملك مجلد المصدر الأعلى للمشروع الأب.

الأهداف هي وحدة التصميم

الهدف هو ملف تنفيذي أو مكتبة أو كيان منطقي في عملية البناء. تطلب CMake الحديثة إرفاق المتطلبات بالأهداف نفسها:

```
add_library(network src/client.cpp)

target_compile_features(network PUBLIC cxx_std_20)
target_include_directories(network PUBLIC include)
target_compile_definitions(network PRIVATE NETWORK_BUILDING_LIBRARY)
```

أصبح الهدف الآن يعرف ما يحتاج إليه كي يُترجم، وما الذي يجب أن يرثه مستخدموه. أما البديل، مثل الاستدعاءات العامة include_directories() وخيارات المصرّف الشاملة، فيجعل المشروع معتمدًا على ترتيب المجلدات وعلى تأثيرات جانبية غير ظاهرة.

وقت الإعداد ووقت التوليد ووقت البناء

تنشأ أخطاء مربكة كثيرة من الخلط بين المجالات الزمنية الثلاثة.

وقت الإعداد: تنفذ أوامر لغة CMake. تعمل if() وset() وfind_package() وmessage() في هذه المرحلة.

وقت التوليد: تُقيَّم تعبيرات المولّد، مثل $<CONFIG:Debug>، أثناء كتابة CMake لنظام البناء المولّد.

وقت البناء: يعمل المصرّف والرابط والأوامر المخصصة والاختبارات وبرامج التثبيت لاحقًا.

مثال:

```
target_compile_definitions(app PRIVATE
  $<$<CONFIG:Debug>:APP_ENABLE_DIAGNOSTICS>
)
```

الشرط هنا ليس جملة if() عادية في CMake، بل يُحفظ حتى مرحلة التوليد لكي تنشئ CMake قواعد خاصة بكل إعداد، بما في ذلك قواعد المولّدات متعددة الإعدادات.

السياسات قرارات توافق

تتطور CMake من دون أن تغيّر سلوك المشاريع القديمة بصمت. وتحدد السياسات كيفية تصرف المشروع عندما تختلف التفسيرات التاريخية عن الحديثة. يضبط cmake_minimum_required() إصدار السياسات تلقائيًا.

صيغة عملية حديثة:

```
cmake_minimum_required(VERSION 3.25...4.4)
```

يمثل الحد الأدنى أقدم إصدار تدعمه، بينما يعني الحد الأعلى أن المشروع روجع وفق سلوك السياسات حتى CMake 4.4. وتفهم إصدارات CMake منذ 3.12 صيغة المجال هذه.

لا تضبط كل سياسة منفردة إلا عندما يحتاج المشروع فعلًا إلى استثناء توافق محلي.

شيفرة CMake ليست برنامج صدفة طرفية

لـ CMake قواعدها الخاصة للقوائم والاقتباس والنطاقات والدوال والشروط. الفاصلة المنقوطة فاصل بين عناصر القائمة، وقد يتوسع المتغير غير المقتبس إلى عدة وسائط، وقد تحتوي المسارات على مسافات. تعامل مع CMake كلغة، لا كنص صدفة ملفوف حول خيارات المصرّف.

```
set(files
  src/main.cpp
  src/parser.cpp
  src/lexer.cpp
)

add_executable(tool ${files})
```

استخدم التوسيع المقتبس عندما تكون القيمة مسارًا أو سلسلة واحدة منطقيا:

```
target_include_directories(tool PRIVATE "${generated_include_dir}")
```

أقصر خلاصة موثوقة

يتبع مشروع CMake القابل للصيانة هذه السلسلة:

١. أنشئ هدفًا.

٢. أرفق به ملفاته المصدرية ومتطلبات ترجمته.

٣. اربطه بأهداف أخرى بدل ربطه بمسارات مكتبات خام.

٤. صدّر متطلبات الاستخدام الضرورية فقط إلى المستهلكين.

٥. اجعل البناء خارج شجرة المصدر.

٦. استخدم الإعدادات المسبقة لتثبيت الأوامر وخيارات الذاكرة المخبأة.

٧. اختبر التثبيت والاستهلاك، لا البناء المحلي فقط.

تثبيت CMake واختيار المولّد

ثبّت إصدارًا معروفًا

مديرو الحزم في أنظمة التشغيل مريحون، لكن مستودعات التوزيعات قد تتأخر عن إصدارات CMake الحديثة. تعتمد أفضل طريقة للتثبيت على ما إذا كنت تفضّل التكامل مع نظام التشغيل أم التحكم الدقيق في الإصدار.

Windows

- استخدم مثبّت Kitware الرسمي للحصول على إصدار حديث متاح لجميع المستخدمين.
- الأمر `winget install Kitware.CMake` مناسب لأجهزة التطوير المُدارة.
- يتضمن Visual Studio تكاملًا مع CMake، لكن الإصدار المرفق قد يختلف عن إصدار CMake المثبّت بصورة مستقلة.

Linux

تصلح حزم التوزيعة عندما تحقق الحد الأدنى الذي يعلنه المشروع:

```
sudo dnf install cmake ninja-build       # Fedora/RHEL family
sudo apt install cmake ninja-build       # Debian/Ubuntu family
```

للحصول على إصدار أحدث، استخدم الحزمة الثنائية الرسمية من Kitware، أو صورة حاوية موثوقة، أو بيئة تطوير يتحكم بها المشروع.

macOS

```
brew install cmake ninja
```

تحقق دائمًا من الملف التنفيذي الذي يعثر عليه PATH:

```
cmake --version
cmake --help
```

أضاف CMake 4.3 إمكانية إخراج الإصدار بصيغة JSON:

```
cmake --version=json-v1
```

ماذا يحدد المولّد؟

يحدد المولّد نظام البناء الأصلي، وغالبًا يحدد أيضًا ما إذا كان مجلد بناء واحد يستطيع احتواء إعداد واحد أم عدة إعدادات.

المولّدات أحادية الإعداد

أمثلة شائعة:

- Ninja
- Unix Makefiles

تستخدم عادةً CMAKE_BUILD_TYPE أثناء الإعداد:

```
cmake -S . -B build/release -G Ninja -DCMAKE_BUILD_TYPE=Release
cmake --build build/release
```

لا ينبغي إعادة استخدام مجلد بناء أُعد بوصفه Debug عشوائيًا بوصفه Release. تجعل المجلدات المنفصلة أو الإعدادات المسبقة الحالة صريحة.

المولّدات متعددة الإعدادات

أمثلة شائعة:

- Visual Studio
- Xcode
- Ninja Multi-Config

تدعم عدة إعدادات داخل شجرة بناء واحدة. اختر الإعداد عند البناء والاختبار والتثبيت:

```
cmake -S . -B build -G "Ninja Multi-Config"
cmake --build build --config Release
ctest --test-dir build -C Release
cmake --install build --config Release
```

يكون تعيين `CMAKE_BUILD_TYPE=Release` غير مؤثر غالبًا مع هذه المولّدات.

> **خطأ شائع:** لا يحول `--config Release` مجلد بناء أحادي الإعداد إلى بناء Release، بل يختار إعدادًا موجودًا فقط مع المولّدات متعددة الإعدادات.

المولّدات الموصى بها

يُعد Ninja خيارًا افتراضيًا ممتازًا للتطوير من الطرفية ولأنظمة التكامل المستمر؛ فهو سريع ومتوقع السلوك ومتاح على المنصات المختلفة. يناسب Visual Studio الفرق التي تحتاج الحلول المولّدة أو التكامل المباشر مع MSBuild. أما Ninja Multi-Config فيفيد عندما تحتاج بيئة طرفية إلى ملفات Debug وRelease من شجرة واحدة.

ليس صحيحًا أن Make متسلسل بطبيعته؛ إذ تستطيع إصدارات Make الحديثة البناء بالتوازي عند تمرير خيارات المهام المناسبة. تتجسد ميزة Ninja في نموذج بناء صغير وكفء عمدًا، وفي سرعة تقييم التبعيات، وفي التكامل الجيد مع القواعد التي تولّدها CMake.

اختيار المصرّف

اختر المصرّف قبل الإعداد الأول لمجلد البناء. تحفظ CMake المصرّف وتفاصيل سلسلة الأدوات في الذاكرة المخبأة.

في الصدف الشبيهة بـ Unix:

```
CC=clang CXX=clang++ cmake -S . -B build/clang -G Ninja
```

أو مرر متغيرات الذاكرة المخبأة في الإعداد الأول:

```
cmake -S . -B build/clang -G Ninja \
  -DCMAKE_C_COMPILER=clang \
  -DCMAKE_CXX_COMPILER=clang++
```

تغيير المصرّف داخل شجرة بناء سبق إعدادها غير موثوق. أنشئ مجلد بناء جديدًا.

Cygwin وMSYS2 وMinGW ومسارات Windows الأصلية

يوفر Windows بيئات عديدة تبدو شبيهة بـ Unix، لكنها تمثل المسارات وسلاسل الأدوات بطرائق مختلفة.

- تفهم برامج **Cygwin** عادةً مسارات Cygwin مثل `/cygdrive/d/...`.
- توفر **MSYS2** صدفًا ومستودعات حزم لبيئات MSYS وMinGW وUCRT وClang.

- تنتج **MinGW-w64** ملفات Windows أصلية باستخدام أدوات GNU.
- يستخدم **Visual Studio** المصرّف MSVC وأداة MSBuild وقواعد مسارات Windows.

يجب أن ينتمي مولّد CMake والمصرّف وبرنامج Make والصدفة إلى بيئة متماسكة. قد ينجح إعداد يجمع CMake الأصلي في Windows مع Cygwin Make وترجمة المسارات ظاهريًا، لكنه يولّد تبعيات تشير إلى مسارات لا تستطيع أداة البناء المختارة حلها.

افحص الأدوات المختارة:

```
which cmake
which gmake
cmake -S . -B build -G "Unix Makefiles" --fresh
cmake --build build --verbose
```

عند تشخيص البيئات المختلطة، استخدم زوجًا كاملًا ومتسقًا: CMake الخاص بـ Cygwin مع مصرّف Cygwin وMake، أو CMake الخاص ببيئة MSYS2 MinGW مع مصرّف MinGW وNinja، أو CMake الأصلي مع MSVC أو Ninja.

التكامل مع بيئات التطوير

ينبغي لبيئات التطوير الحديثة أن تستهلك نموذج الأهداف والإعدادات المسبقة نفسه الذي تستخدمه أوامر الطرفية وأنظمة التكامل المستمر.

- يستطيع Visual Studio فتح مجلد يحتوي `CMakeLists.txt` واستخدام الإعدادات المسبقة.
- يستخدم VS Code إضافة CMake Tools.
- يقرأ CLion مشاريع CMake وإعداداتها المسبقة مباشرةً.
- يدعم Qt Creator مجموعات CMake وإعدادات البناء.
- يستطيع Xcode فتح المشاريع المولّدة أو العمل مع CMake من خلال تكاملات خارجية.

تجنب صيانة مشروع IDE مكتوب يدويًا ومنفصلًا عندما تكون CMake هي المصدر الحقيقي لتعريف المشروع.

مشروعك الصحيح الأول

هيكل المشروع

أنشئ الملفات التالية:

```
hello/
|-- CMakeLists.txt
`-- src/
    `-- main.cpp
```

الملف src/main.cpp:

```cpp
#include <iostream>

int main() {
    std::cout << "Hello from modern CMake!\n";
}
```

الملف CMakeLists.txt:

```cmake
cmake_minimum_required(VERSION 3.25)
project(HelloModernCMake VERSION 1.0.0 LANGUAGES CXX)

add_executable(hello src/main.cpp)
target_compile_features(hello PRIVATE cxx_std_20)
```

نفّذ الإعداد والبناء من جذر المشروع:

```
cmake -S . -B build -G Ninja
cmake --build build
```

شغّل البرنامج:

```
./build/hello                  # Linux/macOS with Ninja
build\hello.exe               # Windows with a single-config generator
```

مع Visual Studio أو مولّد آخر متعدد الإعدادات، يوجد الملف التنفيذي عادةً داخل مجلد الإعداد:

```
cmake -S . -B build
cmake --build build --config Debug
.\build\Debug\hello.exe
```

لماذا تهم الخياران -S و-B؟

التسلسل القديم صحيح:

```
mkdir build
cd build
cmake ..
cmake --build .
```

لكنه يعتمد على مجلد العمل الحالي، ويجبر القارئ على التفكير في معنى . و.. عند كل خطوة. الصيغة الحديثة أوضح:

```
cmake -S <source-directory> -B <build-directory>
cmake --build <build-directory>
```

تربط CMake بنفسها مجلدي المصدر والبناء أثناء الإعداد. ولا يوجد أمر إضافي يجب وضعه داخل CMakeLists.txt لإنشاء هذه الصلة.

إذا أبلغ Makefile مولّد عن عدم وجود قاعدة لإنشاء مسار مصدر، فتحقق من الأسباب الآتية:

١. ملف المصدر غير موجود في المسار الذي أُعد المشروع به.

٢. حالة الأحرف في اسم الملف تختلف عن المسار المكتوب في CMakeLists.txt.

٣. نُقل المشروع أو مجلد البناء بعد الإعداد.

٤. أُنشئت الذاكرة المخبأة في صدفة مختلفة ذات قواعد مسارات غير متوافقة.

٥. يحتوي اسم ملف المصدر على مسافة أو محرف غير مرئي بالخطأ.

٦. يحتوي مجلد البناء على حالة قديمة من تخطيط سابق.

نفّذ إعدادًا نظيفًا وصريحًا:

```
cmake -S . -B build --fresh
cmake --build build --verbose
```

في إصدارات CMake القديمة التي لا تدعم `--fresh` احذف مجلد البناء المولّد وحده، ثم أعد الإعداد.

> **مهم:** يؤدي تشغيل `cmake ..` من داخل `src/` مع استخدام `src/` مجلدًا للبناء إلى **بناء داخل شجرة المصدر**. قد يبدو أنه حل مشكلة المسار، لكنه يخلط الملفات المولّدة بملفات المصدر ويخفي السبب الأصلي.

فهم الأوامر الثلاثة

يحدد `cmake_minimum_required()` أدنى إصدار مطلوب من CMake وسلوك السياسات.

يسمي `project()` المشروع ويفعّل اللغات ويعرّف متغيرات مثل `PROJECT_NAME` و`PROJECT_VERSION` و`PROJECT_SOURCE_DIR`.

ينشئ `add_executable()` هدفًا. ويُفسر مسار المصدر نسبةً إلى ملف `CMakeLists.txt` الذي يحتوي الأمر، ما لم يكن المسار مطلقًا.

تحديد معيار اللغة بصورة محمولة

فضّل:

```
target_compile_features(hello PRIVATE cxx_std_20)
```

على إضافة `-std=c++20` أو `/std:c++20` يدويًا. تختار CMake الخيار الصحيح لمصرّفات GCC وClang وApple Clang وMSVC.

تتوفر الخصائص أيضًا عند الحاجة إلى تحكم صارم:

```
set_target_properties(hello PROPERTIES
  CXX_STANDARD 20
  CXX_STANDARD_REQUIRED YES
  CXX_EXTENSIONS NO
)
```

يكون `target_compile_features()` عادةً الخيار الأكثر تعبيرًا عندما يحتاج الهدف أو ترويساته العامة إلى مستوى محدد من اللغة.

إضافة التحذيرات من دون كسر قابلية النقل

تختلف التحذيرات بين المصرّفات. اجعلها خاصة بهدفك، واخترها بواسطة تعبيرات المولّد:

```
target_compile_options(hello PRIVATE
  $<$<CXX_COMPILER_ID:GNU,Clang,AppleClang>:-Wall;-Wextra;-Wpedantic>
  $<$<CXX_COMPILER_ID:MSVC>:/W4;/permissive->
)
```

لا تحقن تحذيرات مشروعك في التبعيات المنزلة بواسطة `FetchContent`، ولا تنشر التحذيرات كمتطلبات `PUBLIC` إلا إذا كان المستهلكون يحتاجون إليها فعلًا.

بناء هدف محدد

```
cmake --build build --target hello
```

اعرض الأهداف المولّدة عندما يدعم المولّد ذلك:

```
cmake --build build --target help
```

استخدم التوازي بصيغة محمولة:

```
cmake --build build --parallel
cmake --build build --parallel 8
```

يمكن لمتغير البيئة `CMAKE_BUILD_PARALLEL_LEVEL` تحديد قيمة افتراضية للعمليات الآلية.

التنظيف

توفر مولّدات كثيرة هدفًا للتنظيف:

```
cmake --build build --target clean
```

يحذف هدف التنظيف نواتج البناء، لكنه لا يعيد ضبط حالة الإعداد. عندما تحتاج إعدادًا جديدًا حقًا، احذف شجرة البناء أو حدّثها باستخدام `--fresh`.

الأهداف ومتطلبات الاستخدام

الفكرة الأساسية

يحتوي الهدف على نوعين من المعلومات:

١. ما يلزم لبناء الهدف نفسه.

٢. ما يجب أن ترثه الأهداف التي تستهلكه.

تسمي CMake النوع الثاني **متطلبات الاستخدام**. وتنتقل هذه المتطلبات عبر الأهداف المرتبطة.

تأمل هذه المكتبة:

```
add_library(greetings src/greetings.cpp)

target_include_directories(greetings
  PUBLIC include
)

target_compile_features(greetings
  PUBLIC cxx_std_20
)
```

تحتاج ملفات مصدر المكتبة إلى مجلد التضمين، كما يحتاج إليه أي مستهلك يضمّن <greetings/greetings.hpp>؛ لذلك يكون النطاق PUBLIC.

الفرق بين PRIVATE وPUBLIC وINTERFACE

لا تعني هذه الكلمات التحكم في الوصول كما في C++، ولا تصف ببساطة ما إذا كانت المكتبة مرئية أم لا.

- PRIVATE: يُستخدم لبناء الهدف الحالي فقط.
- PUBLIC: يُستخدم لبناء الهدف الحالي، وينتقل كذلك إلى المستهلكين.
- INTERFACE: لا يُستخدم لبناء الهدف الحالي، وإنما ينتقل إلى المستهلكين فقط.

يمكن اختصار القاعدة هكذا:

- **PRIVATE** - الهدف الحالي: نعم؛ المستهلكون: لا.
- **PUBLIC** - الهدف الحالي: نعم؛ المستهلكون: نعم.
- **INTERFACE** - الهدف الحالي: لا؛ المستهلكون: نعم.

الربط بوصفه رسمًا بيانيًا للتبعيات

```
add_library(formatting src/formatting.cpp)
add_library(protocol src/protocol.cpp)
add_executable(server src/server.cpp)

target_link_libraries(protocol PUBLIC formatting)
target_link_libraries(server PRIVATE protocol)
```

لأن protocol تربط formatting ربطًا عامًا، يحصل مستهلك protocol أيضًا على متطلبات استخدام formatting. ويكفي أن يعلن server التبعية المباشرة التي يعرفها.

هذا النموذج الانتقالي من أهم نقاط قوة CMake. وهو يفسر كذلك لماذا تفضَّل الأهداف المستوردة مثل `Threads::Threads` و`OpenSSL::SSL` و`fmt::fmt` على نسخ متغيرات مسارات التضمين والمكتبات يدويًا.

مجلدات التضمين في شجرتي البناء والتثبيت

غالبًا يختلف مسار التضمين العام أثناء البناء عنه بعد التثبيت:

```
target_include_directories(mylib
  PUBLIC
    $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
    $<INSTALL_INTERFACE:include>
)
```

تشير واجهة البناء إلى شجرة المصدر، بينما تكون واجهة التثبيت نسبية إلى بادئة التثبيت، ولذلك تظل قابلة للنقل.

يمكن لمجموعات ملفات الترويسات في المكتبات الحديثة أن تزيل قدرًا كبيرًا من إدارة المسارات اليدوية:

```
target_sources(mylib
  PUBLIC
    FILE_SET HEADERS
    BASE_DIRS include
    FILES
      include/mylib/api.hpp
      include/mylib/result.hpp
)
```

عند تثبيت الهدف مع مجموعة ملفات ترويساته، تحافظ CMake على البنية الهرمية للترويسات وتصدر متطلبات الاستخدام بصورة صحيحة.

الأهداف المستعارة

يوفر الهدف المستعار اسمًا محليًا ضمن نطاق اسمي:

```
add_library(mylib src/mylib.cpp)
add_library(MyCompany::mylib ALIAS mylib)
```

لاستخدام `MyCompany::mylib` داخليًا فائدتان: فهو يشبه اسم الهدف المستورد الذي سيستخدمه المستهلكون بعد التثبيت، كما أن الخطأ الكتابي في اسم يحتوي `::` يؤدي إلى خطأ إعداد مبكر بدل معاملته اسم مكتبة عاديًا يمرر إلى الرابط.

لا يمكن تثبيت الأهداف المستعارة أو تعديلها مباشرة. اضبط الهدف الحقيقي.

مكتبات INTERFACE

تحمل مكتبة `INTERFACE` متطلبات الاستخدام من دون إنتاج ملف مكتبة:

```
add_library(project_warnings INTERFACE)

target_compile_options(project_warnings INTERFACE
  $<$<CXX_COMPILER_ID:GNU,Clang,AppleClang>:-Wall;-Wextra;-Wpedantic>
  $<$<CXX_COMPILER_ID:MSVC>:/W4>
)

target_link_libraries(app PRIVATE project_warnings)
```

تفيد مكتبات `INTERFACE` في حزم السياسات الاختيارية، والترويسات المولّدة، والمكتبات التي تعتمد على الترويسات فقط، ومجموعات الميزات. تجنب إنشاء هدف واجهة عالمي ضخم ترثه كل تبعية خارجية من دون قصد.

مكتبات الكائنات

تترجم مكتبة الكائنات ملفات المصدر من دون أرشفتها أو ربطها في مكتبة نهائية:

```
add_library(codec_objects OBJECT
  src/decoder.cpp
  src/encoder.cpp
)
```

يمكن لأهداف أخرى استهلاك ملفات الكائنات مباشرة أو ربط هدف مكتبة الكائنات. استخدم هذا النوع عندما تحتاج عدة منتجات نهائية إلى مشاركة ملفات الكائنات المترجمة نفسها حرفيًا، لكن افهم سلوكه على المنصات المختلفة وسلوك متطلبات الاستخدام قبل اتخاذه بديلًا عن المكتبات الساكنة العادية.

الأهداف المستوردة

تمثل الأهداف المستوردة منتجات بُنيت خارج المشروع الحالي. وتعرّفها عادةً ملفات إعداد الحزم:

```
find_package(ZLIB REQUIRED)
target_link_libraries(app PRIVATE ZLIB::ZLIB)
```

قد يحمل الهدف المستورد مواقع منفصلة لـ Debug وRelease، ومجلدات تضمين، وتعريفات ترجمة، وتبعيات ربط انتقالية. وهذا أكثر أمانًا من قراءة ZLIB_LIBRARIES وZLIB_INCLUDE_DIRS مباشرةً عندما يتوفر هدف حديث.

الخصائص والأوامر

تكتب أوامر target_* كثيرة خصائص الهدف. فعلى سبيل المثال يملأ target_include_directories() خصائص مجلدات التضمين الخاصة بالبناء والواجهة. فضّل الأمر عالي المستوى لأنه يعبّر عن النية ويعالج النطاقات بصورة صحيحة.

افحص خاصية أثناء الإعداد:

```
get_target_property(public_includes mylib INTERFACE_INCLUDE_DIRECTORIES)
message(STATUS "mylib public includes: ${public_includes}")
```

للحصول على تشخيص أغنى، استخدم أدوات الطباعة المرفقة مع CMake:

```
include(CMakePrintHelpers)
cmake_print_properties(
  TARGETS mylib
  PROPERTIES TYPE SOURCES INTERFACE_INCLUDE_DIRECTORIES
)
```

قاعدة التبعية المباشرة

اربط كل مكوّن بالأهداف التي يستخدمها مباشرة فقط، ودع الواجهات العامة تمرر ما يحتاج إليه المستهلكون. ينتج عن ذلك رسم تبعيات مقروء، ويمنع الاعتماد العرضي على تبعيات انتقالية غير مرتبطة، ويجعل إعادة الهيكلة لاحقًا أكثر أمانًا.

تنظيم مشاريع C++ الحقيقية

ابدأ ببنية التبعيات لا بموضة المجلدات

يعكس تخطيط المشروع الجيد مكوناته المنطقية. يمكن لكل مجلد يملك أهدافًا أن يحتوي ملف CMakeLists.txt خاصًا به، بينما ينسق ملف المستوى الأعلى المشروع كله.

```
project/
|-- CMakeLists.txt
|-- cmake/
|-- include/
|   `-- orbit/
|       `-- orbit.hpp
|-- src/
|   `-- orbit/
|       |-- CMakeLists.txt
|       `-- orbit.cpp
|-- app/
|   |-- CMakeLists.txt
|   `-- main.cpp
`-- tests/
    |-- CMakeLists.txt
    `-- orbit_test.cpp
```

ملف CMakeLists.txt الأعلى:

```
cmake_minimum_required(VERSION 3.25...4.4)
project(Orbit VERSION 3.0.0 LANGUAGES CXX)

include(CTest)

add_subdirectory(src/orbit)
add_subdirectory(app)

if(BUILD_TESTING)
  add_subdirectory(tests)
endif()
```

ينبغي للملف الأعلى إعلان اختيارات المشروع العامة وإضافة المكونات الكبرى. أما ملفات المكونات فتعرف أهدافها ومصادرها المحلية بنفسها.

add_subdirectory() ليست تضمينًا نصيًا

تعالج add_subdirectory() ملف CMakeLists.txt آخر داخل نطاق مجلد جديد، وتنشئ له مجلد بناء مقابلًا. تكون الأهداف معروفة على مستوى عملية البناء كلها، لكن المتغيرات العادية تتبع قواعد نطاق المجلدات.

```
add_subdirectory(src/orbit)
```

تنشئ CMake المطابقة الآتية:

```
source: project/src/orbit
build:  project/build/src/orbit
```

لهذا السبب ينبغي عادةً كتابة الملفات المولّدة في CMAKE_CURRENT_BINARY_DIR لا بجانب ملفات المصدر.

ضع تعريف الهدف قرب مصادره

الملف src/orbit/CMakeLists.txt:

```
add_library(orbit orbit.cpp)
add_library(Orbit::orbit ALIAS orbit)

target_compile_features(orbit PUBLIC cxx_std_20)

target_sources(orbit
  PUBLIC
    FILE_SET HEADERS
    BASE_DIRS "${PROJECT_SOURCE_DIR}/include"
    FILES "${PROJECT_SOURCE_DIR}/include/orbit/orbit.hpp"
)
```

في مكتبة مستقلة قابلة لإعادة الاستخدام، فضّل المسارات النسبية إلى مجلد المكتبة نفسه بدل PROJECT_SOURCE_DIR؛ لأن الأخير قد يشير إلى المشروع الأب عندما تُضمَّن المكتبة داخله. ويمكن لنسخة مكتفية ذاتيًا وضع include/ داخل مكون المكتبة واستخدام CMAKE_CURRENT_SOURCE_DIR.

لا تجعل file(GLOB) الأسلوب الافتراضي لقائمة المصادر

تبدو الصيغة التالية جذابة:

```
file(GLOB sources CONFIGURE_DEPENDS "src/*.cpp")
add_library(core ${sources})
```

لكن القوائم الصريحة للمصادر لها مزايا مهمة:

- تُظهر مراجعة الشيفرة لحظة انضمام ملف إلى الهدف.
- لا تُترجم الملفات المؤقتة أو المنسوخة بالخطأ.
- تبقى عضوية الهدف حتمية وواضحة.
- تظهر عمليات إعادة التسمية والحذف في فروق نظام التحكم بالإصدارات.

يطلب CONFIGURE_DEPENDS من CMake إعادة فحص أنماط glob أثناء البناء، لكن موثوقيته وكلفته تختلفان باختلاف المولّد والمنصة. استخدم الجمع الآلي عندما تكون مجموعة الملفات فعلًا قائمة على بيانات أو مولّدة، لا لمجرد تجنب كتابة أسماء المصادر.

الدوال أفضل من الماكرو غالبًا

تنشئ الدوال نطاقًا للمتغيرات، وتتصرف بصورة أقرب إلى دوال اللغات التقليدية:

```
function(enable_project_warnings target)
  target_compile_options(${target} PRIVATE
    $<$<CXX_COMPILER_ID:GNU,Clang,AppleClang>:-Wall;-Wextra;-Wpedantic>
    $<$<CXX_COMPILER_ID:MSVC>:/W4;/permissive->
  )
endfunction()

enable_project_warnings(orbit)
```

تنفذ وحدات الماكرو استبدالًا للوسائط شبيهًا بالنص داخل نطاق المستدعي. وتظل مفيدة لبعض الأنماط اللغوية المحدودة، لكن الدوال أكثر أمانًا للمساعدات القابلة لإعادة الاستخدام.

استخدم cmake_parse_arguments() لبناء واجهات واضحة ذات كلمات مفتاحية:

```
function(add_project_tool name)
  set(options INSTALL)
  set(one_value OUTPUT_NAME)
  set(multi_value SOURCES LIBRARIES)
  cmake_parse_arguments(ARG "${options}" "${one_value}" "${multi_value}" ${ARGN})

  add_executable(${name} ${ARG_SOURCES})
  target_link_libraries(${name} PRIVATE ${ARG_LIBRARIES})

  if(ARG_OUTPUT_NAME)
    set_target_properties(${name} PROPERTIES OUTPUT_NAME "${ARG_OUTPUT_NAME}")
  endif()

  if(ARG_INSTALL)
    install(TARGETS ${name} RUNTIME DESTINATION bin)
  endif()
endfunction()
```

ضع الخيارات عند حدود ملكية واضحة

عرّف الخيارات المنطقية الموجهة إلى المستخدم بواسطة `option()`:

```
option(ORBIT_BUILD_TOOLS "Build Orbit command-line tools" ON)
option(ORBIT_BUILD_EXAMPLES "Build Orbit examples" OFF)
```

ضع بادئة خاصة بالمشروع لمتغيرات الذاكرة المخبأة لتجنب التعارض عندما يصبح مشروعك مجلدًا فرعيًا داخل مشروع آخر.

لا تفرض قيم الذاكرة المخبأة إلا لسبب توافق قوي. يجب أن يحتفظ المشروع الأب أو المستخدم بالتحكم:

```
# Avoid this pattern in normal project code:
set(BUILD_SHARED_LIBS OFF CACHE BOOL "" FORCE)
```

السلوك في المشروع الأعلى والمشروع الفرعي

قد يُعد المشروع مباشرة أو يضاف إلى مشروع آخر. افحص العلاقة هكذا:

```
if(PROJECT_IS_TOP_LEVEL)
  option(ORBIT_BUILD_TOOLS "Build tools" ON)
else()
  option(ORBIT_BUILD_TOOLS "Build tools" OFF)
endif()
```

`PROJECT_IS_TOP_LEVEL` أفضل من مقارنة متغيرات مجلد المصدر يدويًا.

ينبغي للمكتبة ألا تغيّر خيارات المصرّف العامة أو بادئة التثبيت أو افتراضات الاختبار أو إعدادات مدير الحزم لمجرد إضافتها كمجلد فرعي.

أبقِ الملفات المولّدة داخل شجرة البناء

تنتمي ترويسات الإعداد والترويسات المصدرة والمصادر المولّدة ونواتج البروتوكولات وبرامج توليد الشيفرة إلى `CMAKE_CURRENT_BINARY_DIR` أو إلى مجلد تحته.

```
configure_file(
  config.hpp.in
  "${CMAKE_CURRENT_BINARY_DIR}/generated/orbit/config.hpp"
  @ONLY
)

target_include_directories(orbit PRIVATE
  "${CMAKE_CURRENT_BINARY_DIR}/generated"
)
```

يحافظ ذلك على نظافة شجرة المصدر، ويسمح بتعايش عدة مجلدات بناء بخيارات مختلفة.

استخدم أسماء أهداف ذات نطاق اسمي باستمرار

داخل المشروع:

```
add_library(orbit orbit.cpp)
add_library(Orbit::orbit ALIAS orbit)
```

بعد التثبيت صدّر الهدف الحقيقي ضمن النطاق `Orbit::`. يستخدم المستهلكون عندها الاسم المنطقي نفسه سواء بُني Orbit داخل الشجرة أو عُثر عليه كحزمة مثبتة.

افصل سياسة المشروع عن المكتبات القابلة لإعادة الاستخدام

ينبغي عادةً أن تكون تحذيرات المصرّف والمطهّرات وتغطية الاختبارات وتفضيلات المطور المحلية أهدافًا اختيارية خاصة بالمشروع:

```
add_library(orbit_options INTERFACE)
add_library(orbit_warnings INTERFACE)

target_link_libraries(orbit PRIVATE orbit_options orbit_warnings)
```

لا تربط خيارات المطهّر علنًا بمكتبة إلا إذا كان كل مستهلك مضطرًا إلى استخدام وقت تشغيل المطهّر نفسه. قرارات مستوى التطبيق لا ينبغي أن تتسرب بصمت عبر الواجهة العامة للمكتبة.

إعدادات البناء ومتغيرات الذاكرة المخبأة والإعدادات المسبقة

الذاكرة المخبأة حالة إعداد دائمة

ينشئ الإعداد الأول الملف CMakeCache.txt داخل مجلد البناء. ويسجل المصرّفات والأدوات المكتشفة وخيارات المستخدم والمسارات ومعلومات المولّد. عند إعادة إعداد شجرة البناء نفسها يعاد استخدام هذه الحالة.

أوامر مفيدة:

```
cmake -S . -B build -L                # list cache variables
cmake -S . -B build -LAH              # include advanced values and help
cmake -S . -B build -U 'OLD_*'        # remove matching cache entries
cmake -S . -B build --fresh           # recreate cache and CMakeFiles
```

لا تعدّل CMakeCache.txt يدويًا. استخدم -D أو الإعدادات المسبقة أو cmake-gui أو ccmake.

المتغيرات العادية ومتغيرات الذاكرة المخبأة ومتغيرات البيئة

يوجد المتغير العادي داخل نطاق لغة CMake:

```
set(source_root "${CMAKE_CURRENT_SOURCE_DIR}/src")
```

أما متغير الذاكرة المخبأة فيبقى بين عمليات الإعداد ويظهر للمستخدمين:

```
set(ORBIT_DATA_DIR "share/orbit" CACHE STRING "Install location for Orbit data")
```

يكون الخيار المنطقي أوضح باستخدام option():

```
option(ORBIT_ENABLE_TRACING "Enable runtime tracing" OFF)
```

ينتمي متغير البيئة إلى بيئة العملية:

```
message(STATUS "PATH=$ENV{PATH}")
```

تفيد متغيرات البيئة في اختيار سلسلة الأدوات وفي أسرار أنظمة التكامل المستمر، لكن سلوك المشروع يكون أكثر قابلية لإعادة الإنتاج عندما تكون الخيارات المهمة مدخلات صريحة في الذاكرة المخبأة أو حقولًا في الإعدادات المسبقة.

أضاف CMake 4.2 صيغة صريحة للتعامل مع مدخلات الذاكرة المخبأة:

```
set(CACHE{ORBIT_MODE} TYPE STRING HELP "Orbit operating mode" VALUE "safe")
unset(CACHE{ORBIT_OLD_MODE})
```

تميّز هذه الصيغة عمليات الذاكرة المخبأة عن إسناد المتغيرات العادية.

إعدادات البناء

مع Ninja أو Make:

```
cmake -S . -B out/build/debug -G Ninja -DCMAKE_BUILD_TYPE=Debug
cmake -S . -B out/build/release -G Ninja -DCMAKE_BUILD_TYPE=Release
```

الإعدادات القياسية الشائعة:

- Debug: معلومات تصحيح مع تحسين منخفض أو معدوم.
- Release: تحسينات، وعادةً تعريف NDEBUG.
- RelWithDebInfo: تحسينات مع معلومات تصحيح.
- MinSizeRel: تحسين لتقليل الحجم.

تحدد CMake نموذج الإعدادات، لكن خيارات المصرّف الدقيقة تعتمد على سلسلة الأدوات. لا تفترض أن كل بناء Release يستخدم الخيارات نفسها على جميع المصرّفات.

لماذا الإعدادات المسبقة أفضل من أوامر README؟

يتقادم الأمر المكتوب في README ويسهل نسخه بطريقة خاطئة. أما CMakePresets.json فقابل للقراءة آليًا، ومحفوظ مع نظام التحكم بالإصدارات، وتفهمه CMake، وتدعمه أهم بيئات التطوير.

حد عملي يستخدم إصدار المخطط 6:

```
{
  "version": 6,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 25,
    "patch": 0
  },
  "configurePresets": [
    {
      "name": "base",
      "hidden": true,
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/out/build/${presetName}",
      "installDir": "${sourceDir}/out/install/${presetName}"
    },
    {
      "name": "dev",
      "inherits": "base",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Debug",
        "CMAKE_EXPORT_COMPILE_COMMANDS": true,
        "BUILD_TESTING": true
      }
    },
    {
      "name": "release",
      "inherits": "base",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Release",
        "BUILD_TESTING": false
      }
    }
  ],
  "buildPresets": [
    {
      "name": "dev",
      "configurePreset": "dev",
      "jobs": 8
    },
    {
      "name": "release",
      "configurePreset": "release",
      "jobs": 8
    }
  ],
  "testPresets": [
    {
      "name": "dev",
      "configurePreset": "dev",
      "output": {
        "outputOnFailure": true
      }
    }
  ]
}
```

استخدمه هكذا:

```
cmake --preset dev
cmake --build --preset dev
ctest --preset dev
```

CMakeUserPresets.json و CMakePresets.json

احفظ `CMakePresets.json` في المستودع؛ فهو يصف سير العمل المشترك للمشروع ولأنظمة التكامل المستمر.

لا تحفظ `CMakeUserPresets.json` عادةً. فقد يحتوي مسارات محلية أو مصرّفًا يفضله المطور أو إعدادات بيئة خاصة. أضفه إلى `.gitignore`.

يستخدم الملفان المخطط نفسه، ويمكنهما تضمين الإعدادات المسبقة أو الوراثة منها. وتستطيع إعدادات المستخدم أن ترث إعدادات المشروع من دون تعديل الملف المشترك.

توافق إصدارات مخطط الإعدادات المسبقة

استخدم أدنى إصدار من المخطط يحقق الحقول التي تحتاجها. قد يمنع الإصدار الأعلى مستخدمي CMake الأقدم من العمل، حتى لو كانت صيغة المشروع نفسها متوافقة.

محطات مهمة:

- المخطط 1: إعدادات الإعداد، CMake 3.19.
- المخطط 2: إعدادات البناء والاختبار، CMake 3.20.
- المخطط 3: الشروط وملف سلسلة الأدوات ومجلد التثبيت، CMake 3.21.
- المخطط 4: التضمين، CMake 3.23.
- المخطط 6: إعدادات التحزيم وسير العمل، CMake 3.25.
- المخطط 11: CMake 4.3.
- المخطط 12: CMake 4.4، ويشمل ملفات الإعدادات الخارجية وسلوك اختبار أحدث.

يستخدم الكتاب المخطط 6 في الأمثلة المحمولة، ويعزل أمثلة المخطط 12 بوضوح.

إعدادات سير العمل المسبقة

يمكن لإعداد سير عمل أن يربط خطوات الإعداد والبناء والاختبار والتحزيم:

```
{
  "workflowPresets": [
    {
      "name": "ci-release",
      "steps": [
        { "type": "configure", "name": "release" },
        { "type": "build", "name": "release" },
        { "type": "test", "name": "release" },
        { "type": "package", "name": "release" }
      ]
    }
  ]
}
```

شغّله:

```
cmake --workflow --preset ci-release
```

يجعل سير العمل التنفيذ المحلي وتنفيذ CI يتبعان التسلسل المعلن نفسه.

شروط الإعدادات المسبقة والوراثة

يمكن للإعدادات الخاصة بمنصة معينة أن ترث إعدادًا أساسيًا مخفيًا وتستخدم شروطًا:

```
{
  "name": "windows-msvc",
  "inherits": "base",
  "condition": {
    "type": "equals",
    "lhs": "${hostSystemName}",
    "rhs": "Windows"
  },
  "generator": "Visual Studio 18 2026",
  "architecture": "x64"
}
```

اجعل الوراثة قليلة العمق. فالهياكل العميقة قد تصبح صعبة الفهم مثل المتغيرات العامة في CMake.

الإعدادات المسبقة في CMake 4.4

يدعم CMake 4.4 إصدار المخطط 12، ويضيف خيارات `--presets-file`. وهذا مفيد لأنظمة CI أو حزم SDK المُدارة مركزيًا عندما توجد تعريفات الإعدادات خارج جذر المصدر:

```
cmake --presets-file ci/presets.json --preset linux-release
cmake --build build --presets-file ci/presets.json --preset linux-release
ctest --presets-file ci/presets.json --preset linux-release
```

استخدم هذه الإمكانية بحذر؛ إذ يظل ملف `CMakePresets.json` المحلي داخل المشروع هو الأسهل للمساهمين وبيئات التطوير.

المكتبات والتبعيات

أنواع المكتبات

تستطيع CMake تمثيل أنواع متعددة من المكتبات:

```
add_library(core STATIC src/core.cpp)
add_library(plugin SHARED src/plugin.cpp)
add_library(headers INTERFACE)
add_library(objects OBJECT src/a.cpp src/b.cpp)
```

إذا لم تحدد النوع، يستطيع BUILD_SHARED_LIBS اختيار المكتبة الساكنة أو المشتركة للمكتبات العادية:

```
add_library(core src/core.cpp)
```

يفيد ذلك عندما يدعم المشروع الشكلين عمدًا. لكن لا تفترض أن كل مكتبة تستطيع التبديل بينهما من دون ماكروات تصدير، أو شيفرة مستقلة عن الموضع، أو قواعد رؤية الرموز، أو اعتبارات لمسارات وقت التشغيل.

الترويسات العامة بواسطة مجموعات الملفات

```
add_library(mathlib src/add.cpp)

target_sources(mathlib
  PUBLIC
    FILE_SET HEADERS
    BASE_DIRS include
    FILES include/mathlib/add.hpp
)
```

تعبّر مجموعة الملفات من النوع HEADERS عن الترويسات العامة ومجلداتها الأساسية وصلتها بالهدف. وتحسن عرضها في بيئات التطوير، وتثبيتها وتصديرها والتحقق منها.

للترويسات الخاصة:

```
target_sources(mathlib
  PRIVATE
    FILE_SET internal_headers TYPE HEADERS
    BASE_DIRS src
    FILES src/detail/checked_add.hpp
)
```

وسع CMake 4.3 و4.4 إمكانات فحص مجموعات الملفات والتحقق منها. وأضاف CMake 4.4 أيضًا نوع مجموعة الملفات SOURCES لتنظيم أوسع لملفات المصدر.

الشيفرة المستقلة عن الموضع

تحتاج المكتبات المشتركة عادةً إلى شيفرة مستقلة عن الموضع على منصات مثل Linux. وتفعّل CMake ذلك تلقائيًا للمكتبات SHARED وMODULE.

لمكتبة ساكنة ستُربط داخل مكتبة مشتركة:

```
set_target_properties(core PROPERTIES POSITION_INDEPENDENT_CODE ON)
```

تجنب إضافة -fPIC يدويًا؛ فهو غير محمول إلى MSVC ويكرر نموذج المنصة الموجود في CMake.

رؤية الرموز وترويسات التصدير

تحتاج المكتبة المشتركة متعددة المنصات إلى ABI عام مقصود. يستطيع المكوّن GenerateExportHeader في CMake إنشاء ماكروات التصدير:

```
include(GenerateExportHeader)

generate_export_header(mylib
  EXPORT_FILE_NAME "${CMAKE_CURRENT_BINARY_DIR}/generated/mylib/export.hpp"
)
```

تستخدم التصريحات العامة بعد ذلك الماكرو المولّد. ويمكن جمعه مع إخفاء الرموز افتراضيًا عندما يلائم المشروع:

```
set_target_properties(mylib PROPERTIES
  CXX_VISIBILITY_PRESET hidden
  VISIBILITY_INLINES_HIDDEN YES
)
```

تتحكم هذه الإعدادات في الرموز المصدرة، لا في صلاحيات الوصول إلى الأصناف في C++.

اكتشاف التبعيات بواسطة find_package()

النمط المفضّل لدى المستهلك:

```
find_package(fmt 11 CONFIG REQUIRED)
target_link_libraries(app PRIVATE fmt::fmt)
```

تعمل `find_package()` في نمطين عامين.

نمط Config: يحمل ملفًا توفره الحزمة مثل `fmtConfig.cmake`. وهو التمثيل الأفضل عادةً لأن مؤلف الحزمة يعرّف الأهداف المستوردة.

نمط Module: يحمل وحدة `Find<Package>.cmake` توفرها CMake أو المشروع غالبًا. تفيد الوحدات عندما لا تقدم التبعية ملف إعداد خاصًا بها.

كن صريحًا عندما يحتمل الالتباس:

```
find_package(OpenSSL REQUIRED COMPONENTS SSL Crypto)
find_package(Qt6 6.8 REQUIRED COMPONENTS Core Widgets CONFIG)
```

لا تربط مسارات ملفات خامًا عندما يتوفر هدف

تبدو الأنماط القديمة غالبًا هكذا:

```
include_directories(${FOO_INCLUDE_DIRS})
add_executable(app main.cpp)
target_link_libraries(app PRIVATE ${FOO_LIBRARIES})
```

الهدف المستورد الحديث أفضل:

```
find_package(Foo REQUIRED)
target_link_libraries(app PRIVATE Foo::Foo)
```

يستطيع الهدف المستورد تمثيل المسارات الخاصة بكل إعداد، والتبعيات الانتقالية، وتعريفات الترجمة، وأطر المنصة، وميزات اللغة المطلوبة.

اختيار الإصدار

```
find_package(MathLib 1.4 REQUIRED CONFIG)
```

يعتمد سلوك التوافق الدقيق على ملف إصدار الحزمة. يقبل `SameMajorVersion` غالبًا الإصدارات اللاحقة ذات الرقم الرئيسي نفسه، لكن مؤلف الحزمة وحده من يستطيع تحديد معنى التوافق الثنائي أو توافق المصدر.

استخدم الإصدار المطابق تمامًا فقط عندما يحتاج المشروع إليه حقيقةً:

```
find_package(MathLib 1.4.2 EXACT REQUIRED CONFIG)
```

تجعل القيود الصارمة المفرطة تكامل النظام وتحديثات الأمان أكثر صعوبة.

التبعيات الاختيارية

يمكن لميزة أن تعتمد على حزمة اختيارية:

```
option(ORBIT_WITH_ZSTD "Enable Zstandard compression" ON)

if(ORBIT_WITH_ZSTD)
  find_package(zstd CONFIG REQUIRED)
  target_link_libraries(orbit PRIVATE zstd::libzstd_shared)
  target_compile_definitions(orbit PRIVATE ORBIT_WITH_ZSTD=1)
endif()
```

وإذا كان الرجوع التلقائي مناسبًا:

```
find_package(ZLIB QUIET)
if(ZLIB_FOUND)
  target_link_libraries(orbit PRIVATE ZLIB::ZLIB)
endif()
```

لا تعطل بصمت ميزة طلبها المستخدم صراحة. يجب أن يفشل طلب الميزة برسالة واضحة عندما تكون تبعيتها مفقودة.

مواصفة الحزم المشتركة في CMake 4.3

أضاف CMake 4.3 دعم مواصفة الحزم المشتركة **CPS**. تستطيع `find_package()` استيراد أوصاف حزم CPS المتوافقة، بينما يستطيع `install(PACKAGE_INFO)` و`export(PACKAGE_INFO)` إنشاء بيانات تعريف للحزمة.

تهدف CPS إلى تحسين قابلية التشغيل البيني بين أنظمة البناء من خلال وصف الحزم بصيغة مستقلة عن لغة CMake. وهي مهمة لمستقبل تبادل الحزم، لكن ملفات إعداد حزم CMake التقليدية لا تزال واسعة الانتشار وأساسية. تعامل مع CPS على أنها مسار حديث إضافي، لا مبررًا لإزالة دعم حزم Config العامل فورًا.

FetchContent ومديرو الحزم

اختر استراتيجية التبعيات بوعي

يمكن لمشروع C++ الحصول على التبعيات بإحدى الطرائق الآتية:

١. حزم نظام أو SDK مثبتة تعثر عليها `find_package()`.

٢. مدير حزم مثل vcpkg أو Conan.

٣. `FetchContent` لدمج مصدر التبعية في مرحلتي الإعداد والبناء.

٤. `ExternalProject` لبناء مشروع خارجي معزول.

٥. مصدر مورّد داخل المستودع نفسه.

لا توجد طريقة واحدة صحيحة دائمًا. يؤثر الاختيار في قابلية إعادة الإنتاج، والبناء دون اتصال، والمراجعة الأمنية، وإمكانية تطبيق الرقع، وزمن البناء، وقدرة توزيعات النظام على استخدام مكتباتها المثبتة.

استخدام FetchContent

```
include(FetchContent)

FetchContent_Declare(
  fmt
  GIT_REPOSITORY https://github.com/fmtlib/fmt.git
  GIT_TAG 11.2.0
  GIT_SHALLOW TRUE
)

FetchContent_MakeAvailable(fmt)

target_link_libraries(app PRIVATE fmt::fmt)
```

تجعل `FetchContent_MakeAvailable()` التبعية متاحة أثناء الإعداد، وغالبًا بإضافة شجرة مصدرها كمجلد فرعي. وتشارك أهدافها في عملية البناء الرئيسية.

ثبّت دائمًا معرّفًا ثابتًا غير قابل للتغير. يؤدي استخدام فرع متحرك مثل `main` إلى تغير البناء من دون تغير مراجعة المصدر في مشروعك.

> **قاعدة أمنية:** تنزيل الشيفرة في وقت الإعداد يعني تنفيذ منطق CMake الخاص بمشروع آخر على جهاز المطور أو خادم CI. ثبّت الإصدارات، وراجع المصدر، واستخدم مرايا مُدارة عندما يتطلب نموذج التهديد ذلك.

تجاوزات FetchContent المحلية

توضح حزمة الاختبار استخدام `FetchContent` من دون شبكة:

```
include(FetchContent)

FetchContent_Declare(
  tinydep
  SOURCE_DIR "${CMAKE_CURRENT_SOURCE_DIR}/third_party/tinydep"
)

FetchContent_MakeAvailable(tinydep)
```

يستطيع المستخدمون ومديرو الحزم أيضًا تجاوز مصدر التبعية بواسطة متغيرات الذاكرة المخبأة الموثقة FETCHCONTENT_SOURCE_DIR_<NAME>. وهذا يسمح بمصادر محلية أو معدلة من دون تحرير المشروع.

امنع خيارات التبعية من السيطرة على المشروع

قد تكشف التبعيات المضافة كمجلدات فرعية خيارات ذاكرة مخبأة وقواعد تثبيت وتحذيرات واختبارات. تراعي مشاريع حديثة كثيرة استخدامها كمشاريع فرعية، لكن ليس كلها.

طرائق نافعة:

- اضبط خيارات التبعية الموثقة قبل إتاحتها، ومن دون FORCE متى أمكن.
- استخدم EXCLUDE_FROM_ALL عندما يكون مدعومًا.
- عطّل اختبارات التبعية وأمثلتها بواسطة خياراتها الموثقة.
- فضّل مدير الحزم أو الحزمة المثبتة عندما يسبب الدمج تعارضات.
- أبقِ تحذيراتك ومطهّراتك مرتبطة بأهدافك.

لا تعدّل CMAKE_CXX_FLAGS العامة لحل مشكلة في تبعية.

استخدام ExternalProject_Add()

```
include(ExternalProject)

ExternalProject_Add(zlib_external
  URL https://zlib.net/zlib-1.3.1.tar.gz
  URL_HASH SHA256=<verified-hash>
  CMAKE_ARGS
    -DCMAKE_INSTALL_PREFIX=<INSTALL_DIR>
    -DBUILD_SHARED_LIBS=OFF
)
```

ينفذ ExternalProject خط تنزيل وإعداد وبناء وتثبيت مستقلًا، عادةً في وقت البناء. وهو مناسب للمشاريع الفوقية، أو سلاسل الأدوات غير المتوافقة، أو التبعيات التي يجب عزلها.

لا تصبح أهداف المشروع الخارجي تلقائيًا أهداف CMake عادية داخل المشروع الرئيسي. وقد تحتاج إلى أهداف مستوردة وتبعيات صريحة لتمثيل المنتجات الناتجة. وهذه التعقيدات هي ثمن العزل.

vcpkg

يتكامل vcpkg عادةً من خلال ملف سلسلة أدوات CMake:

```
cmake -S . -B build -G Ninja \
  -DCMAKE_TOOLCHAIN_FILE=/path/to/vcpkg/scripts/buildsystems/vcpkg.cmake
```

ثم تُستهلك الحزم بالطريقة العادية:

```
find_package(fmt CONFIG REQUIRED)
target_link_libraries(app PRIVATE fmt::fmt)
```

يسجل ملف البيان vcpkg.json التبعيات والإصدارات وفق خط أساس vcpkg وإعدادات السجلات المختارة. ضع إعداد سلسلة الأدوات في إعداد مسبق بدل مطالبة كل مطور بحفظه.

```
{
  "name": "vcpkg-dev",
  "inherits": "dev",
  "toolchainFile": "$env{VCPKG_ROOT}/scripts/buildsystems/vcpkg.cmake"
}
```

Conan 2

تولّد سير العمل الحديثة في Conan 2 ملفات تكامل CMake بواسطة `CMakeDeps` و`CMakeToolchain`. أما النمط القديم المعتمد على `conanbuildinfo.cmake` و`conan_basic_setup()` فأصبح متجاوزًا.

تسلسل نموذجي:

```
conan install . --output-folder=build/conan --build=missing -s build_type=Release
cmake -S . -B build/release \
  --toolchain build/conan/conan_toolchain.cmake \
  -DCMAKE_BUILD_TYPE=Release
cmake --build build/release
```

ينبغي لشيفرة CMake في المشروع أن تستمر في استخدام الأهداف المستوردة من `find_package()`:

```
find_package(fmt CONFIG REQUIRED)
target_link_libraries(app PRIVATE fmt::fmt)
```

يجهز مدير الحزم بيانات الحزم القابلة للاكتشاف، بينما يبقى نموذج أهداف CMake ثابتًا.

مزودو التبعيات

تدعم CMake مزودي تبعيات يستطيعون اعتراض استدعاءات مثل `find_package()` و`FetchContent_MakeAvailable()`. ويمكن لمديري الحزم استخدام هذه الآلية لتوفير التبعيات مع بقاء شيفرة المشروع محايدة تجاه مدير الحزم.

لا ينبغي لمشروع مكتبة جيد فرض مدير حزم واحد إلا إذا اختار نظامه البيئي هذا القيد صراحة. صِف تبعياتك بمصطلحات CMake القياسية، ووثق مسارات التكامل المتاحة.

قائمة تحقق لقابلية إعادة الإنتاج

- ثبّت إصدارات التبعيات أو مراجعات المصدر غير القابلة للتغير.
- سجّل خطوط الأساس وبيانات القفل التي يدعمها مدير الحزم.
- تحقق من تجزئات أرشيفات التنزيل.
- أبقِ الوصول إلى الشبكة خارج بناء الإصدارات النهائية متى أمكن.
- حافظ على إمكانية استخدام الحزم المثبتة مسبقًا لتوزيعات الأنظمة.
- تجنب اكتشاف التبعيات خفية عبر مسارات بيئة عشوائية.
- اختبر في بيئة نظيفة، لا على جهاز مطور تراكمت فيه المكتبات سنوات.

تثبيت الحزم وتصديرها

شجرة البناء ليست شجرة تثبيت

نجاح بناء الهدف لا يعني تلقائيًا أن مشروعًا آخر يستطيع استهلاكه. يعرّف التثبيت تخطيطًا ثابتًا للملفات التنفيذية والمكتبات والترويسات والبيانات الوصفية والموارد.

استخدم `GNUInstallDirs` للحصول على وجهات تقليدية تراعي المنصة:

```
include(GNUInstallDirs)
```

يوفر متغيرات مثل:

- `CMAKE_INSTALL_BINDIR`
- `CMAKE_INSTALL_LIBDIR`
- `CMAKE_INSTALL_INCLUDEDIR`
- `CMAKE_INSTALL_DATADIR`

استخدم وجهات نسبية حتى تتمكن `CMAKE_INSTALL_PREFIX` وأدوات التحزيم من نقل التثبيت.

تثبيت الهدف وترويساته

```
install(
  TARGETS mathlib
  EXPORT MathLibTargets
  FILE_SET HEADERS DESTINATION "${CMAKE_INSTALL_INCLUDEDIR}"
  ARCHIVE DESTINATION "${CMAKE_INSTALL_LIBDIR}"
  LIBRARY DESTINATION "${CMAKE_INSTALL_LIBDIR}"
  RUNTIME DESTINATION "${CMAKE_INSTALL_BINDIR}"
)
```

تعرف CMake المنتجات المناسبة لكل منصة. تُعد ملفات DLL في Windows منتجات وقت تشغيل، بينما تُعد مكتبات الاستيراد الخاصة بها منتجات أرشيفية.

تصدير الأهداف

```
install(
  EXPORT MathLibTargets
  NAMESPACE MathLib::
  DESTINATION "${CMAKE_INSTALL_LIBDIR}/cmake/MathLib"
)
```

يكتب ذلك تعريفات الأهداف التي يستطيع المستهلكون استيرادها. لا تكتب مسار المكتبة المثبتة يدويًا داخل ملف `MathLibConfig.cmake` مخصص.

إنشاء ملف إعداد حزمة قابل للنقل

الملف `cmake/MathLibConfig.cmake.in`:

```
@PACKAGE_INIT@
include("${CMAKE_CURRENT_LIST_DIR}/MathLibTargets.cmake")
check_required_components(MathLib)
```

ولإنشائه وتثبيته:

```
include(CMakePackageConfigHelpers)

configure_package_config_file(
  cmake/MathLibConfig.cmake.in
  "${CMAKE_CURRENT_BINARY_DIR}/MathLibConfig.cmake"
  INSTALL_DESTINATION "${CMAKE_INSTALL_LIBDIR}/cmake/MathLib"
)

write_basic_package_version_file(
  "${CMAKE_CURRENT_BINARY_DIR}/MathLibConfigVersion.cmake"
  VERSION "${PROJECT_VERSION}"
  COMPATIBILITY SameMajorVersion
)

install(
  FILES
    "${CMAKE_CURRENT_BINARY_DIR}/MathLibConfig.cmake"
    "${CMAKE_CURRENT_BINARY_DIR}/MathLibConfigVersion.cmake"
  DESTINATION "${CMAKE_INSTALL_LIBDIR}/cmake/MathLib"
)
```

يستطيع المستهلك الآن كتابة:

```
find_package(MathLib 1.4 REQUIRED CONFIG)
add_executable(consumer main.cpp)
target_link_libraries(consumer PRIVATE MathLib::mathlib)
```

اختبر الحزمة المثبتة

لا تفترض صحة التصدير لأن المشروع الأصلي يُبنى بنجاح. اختبر مشروعًا مستهلكًا مستقلًا مقابل شجرة التثبيت:

```
cmake -S . -B build -G Ninja \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_INSTALL_PREFIX="$PWD/stage"
cmake --build build
cmake --install build

cmake -S consumer -B consumer-build -G Ninja \
  -DCMAKE_PREFIX_PATH="$PWD/stage"
cmake --build consumer-build
```

يكشف ذلك مسارات التضمين غير القابلة للنقل، والتبعيات الانتقالية الناقصة، والترويسات المنسية، والنطاقات الاسمية الخاطئة.

التبعيات العامة داخل ملفات إعداد الحزم

إذا كانت `MathLib::mathlib` تعتمد علنًا على حزمة أخرى، فيجب أن يكتشفها ملف الإعداد قبل تحميل الأهداف المصدرة:

```
@PACKAGE_INIT@
include(CMakeFindDependencyMacro)
find_dependency(fmt 11 CONFIG)
include("${CMAKE_CURRENT_LIST_DIR}/MathLibTargets.cmake")
```

استخدم `find_dependency()` للتبعيات المكشوفة عبر واجهة الحزمة. وقد تصبح التبعيات الخاصة لمكتبة ساكنة متطلبات ربط أيضًا وفق المنصة وتركيب الأرشيف؛ لذا اختبر المستهلك المثبت بدل الاعتماد على قاعدة مبسطة.

المكونات

قد تثبت الحزم الكبيرة مكونات منفصلة لوقت التشغيل والتطوير والأدوات والوثائق:

```
install(TARGETS orbit_cli
  RUNTIME DESTINATION "${CMAKE_INSTALL_BINDIR}"
  COMPONENT Runtime
)

install(TARGETS orbit
  EXPORT OrbitTargets
  FILE_SET HEADERS DESTINATION "${CMAKE_INSTALL_INCLUDEDIR}"
  ARCHIVE DESTINATION "${CMAKE_INSTALL_LIBDIR}"
  COMPONENT Development
)
```

ثبّت مكونًا محددًا:

```
cmake --install build --component Runtime
```

يسمح CMake 4.4 بتحديد عدة مكونات ضمن استدعاء واحد لـ `cmake --install`.

RPATH وتبعيات وقت التشغيل

يتطلب نشر المكتبات المشتركة أكثر من مجرد الربط. يستخدم Linux وmacOS مسارات بحث وقت التشغيل، بينما يحتاج Windows إلى اكتشاف ملفات DLL في مجلد الملف التنفيذي أو عبر `PATH`.

توفر CMake خصائص أهداف مثل `BUILD_RPATH` و`INSTALL_RPATH` وآليات تثبيت تراعي المنصة لتبعيات وقت التشغيل. لا تضبط `$ORIGIN` أو مسارات البناء المطلقة عالميًا من دون فهم. عرّف سياسة نشر للتطبيق واختبرها داخل بيئة نظيفة.

تساعد `install(RUNTIME_DEPENDENCY_SET)` و`file(GET_RUNTIME_DEPENDENCIES)` في جمع مكتبات وقت التشغيل، لكن تصفية مكتبات النظام ومراجعة التراخيص تحتاجان عناية.

التحزيم باستخدام CPack

حزمة أرشيفية بسيطة:

```
set(CPACK_PACKAGE_VENDOR "simplifycpp.org")
set(CPACK_PACKAGE_DESCRIPTION_SUMMARY "Orbit command-line tools")
set(CPACK_GENERATOR "ZIP;TGZ")
include(CPack)
```

نفّذ الإعداد والبناء أولًا ثم التحزيم:

```
cmake -S . -B build/release -G Ninja -DCMAKE_BUILD_TYPE=Release
cmake --build build/release
cpack --config build/release/CPackConfig.cmake
```

أو استخدم إعداد تحزيم مسبقًا:

```
cpack --preset release
```

يحزّم CPack ما تصفه قواعد التثبيت؛ ولذلك يكون التثبيت الصحيح أساس التحزيم الصحيح.

لا تثبّت في وجهة مطلقة من دون قصد واضح

تتجاوز `DESTINATION` المطلقة بادئة التثبيت وتعقّد التحزيم. أضاف CMake 4.4 فئة تشخيص خاصة بوجهات التثبيت المطلقة.

فضّل:

```
install(FILES orbit.conf DESTINATION "${CMAKE_INSTALL_DATADIR}/orbit")
```

بدلًا من:

```
install(FILES orbit.conf DESTINATION "/etc/orbit")
```

قد تتعمد حزم تكامل النظام تثبيت ملفات الإعداد في مواقع مطلقة، لكن هذا قرار ضمن سياسة تحزيم المنصة ويجب أن يكون صريحًا.

الاختبار باستخدام CTest وGoogleTest

CTest مشغّل الاختبارات المتكامل مع CMake

لا يتطلب CTest إطار اختبار C++ محددًا. فهو يشغّل الأوامر المسجلة كاختبارات، ويتابع خصائصها، ويرسل نتائجها للمستخدم المحلي أو لأنظمة التكامل المستمر.

فعّل الاختبار قرب المستوى الأعلى:

```
include(CTest)
```

تستدعي وحدة CTest الأمر enable_testing() وتعرّف الخيار القياسي BUILD_TESTING. وهذا أفضل عادةً من استدعاء enable_testing() بلا شرط داخل مشروع قابل لإعادة الاستخدام.

```
if(BUILD_TESTING)
  add_subdirectory(tests)
endif()
```

تسجيل اختبار بسيط

```
add_executable(greetings_test greetings_test.cpp)
target_link_libraries(greetings_test PRIVATE Tutorial::greetings)

add_test(
  NAME greetings.basic
  COMMAND greetings_test
)
```

استخدم اسم الهدف في COMMAND. تستبدل CMake المسار الصحيح للملف التنفيذي، بما فيه الإعداد المختار للمولّدات متعددة الإعدادات.

شغّل الاختبارات:

```
ctest --test-dir build --output-on-failure
```

للمولّدات متعددة الإعدادات:

```
ctest --test-dir build -C Debug --output-on-failure
```

خصائص الاختبارات

```
set_tests_properties(greetings.basic PROPERTIES
  TIMEOUT 10
  LABELS "unit;fast"
  WORKING_DIRECTORY "${CMAKE_CURRENT_BINARY_DIR}"
)
```

خصائص مفيدة:

- TIMEOUT: إيقاف الاختبار المعلّق.
- LABELS: تجميع الاختبارات.
- ENVIRONMENT: ضبط مدخلات بيئة كاملة.
- ENVIRONMENT_MODIFICATION: إضافة قيم البيئة في البداية أو النهاية أو إعادة ضبطها.
- WILL_FAIL: عكس معنى النجاح لاختبار يُفترض أن يفشل.
- PASS_REGULAR_EXPRESSION وFAIL_REGULAR_EXPRESSION: فحص المخرجات.

• RESOURCE_LOCK: منع الاختبارات من مشاركة مورد مسمى في الوقت نفسه.

لا تستخدم التعبيرات المنتظمة على المخرجات بديلًا عن عبارات التحقق الصحيحة عندما تملك شيفرة برنامج الاختبار.

اختيار الاختبارات وتشغيلها بالتوازي

```
ctest --test-dir build -R greetings
ctest --test-dir build -L unit
ctest --test-dir build --parallel 8
ctest --test-dir build --repeat until-pass:3
```

يكون --output-on-failure أفضل إعداد افتراضي غالبًا في CI. ويقدم -V ثم -VV تشخيصًا أكثر تفصيلًا.

تجعل إعدادات الاختبار المسبقة أسطر الأوامر قابلة للتكرار:

```
{
  "name": "dev",
  "configurePreset": "dev",
  "output": {
    "outputOnFailure": true
  },
  "execution": {
    "jobs": 8,
    "noTestsAction": "error"
  }
}
```

GoogleTest مع FetchContent

```
include(FetchContent)

FetchContent_Declare(
  googletest
  GIT_REPOSITORY https://github.com/google/googletest.git
  GIT_TAG v1.17.0
)

set(INSTALL_GTEST OFF CACHE BOOL "" FORCE)
FetchContent_MakeAvailable(googletest)

add_executable(unit_tests test_math.cpp)
target_link_libraries(unit_tests PRIVATE GTest::gtest_main)

include(GoogleTest)
gtest_discover_tests(unit_tests)
```

تطلب gtest_discover_tests() من ملف الاختبار التنفيذي المبني عرض اختباراته الفردية، ثم تسجلها في CTest. وهذا أغنى من تسجيل الملف التنفيذي كله اختبارًا واحدًا.

في البيئات المقيدة أو عند البناء المتقاطع، يهم توقيت الاكتشاف وإعداد المحاكي. أما gtest_add_tests() فتفحص ملفات المصدر بدلًا من تشغيل الملف التنفيذي، ولها مفاضلات مختلفة.

الاختبارات التي تحتاج إلى ملفات

لا تعتمد على مجلد العمل الحالي للصدفة. مرر المسارات كوسائط أو عيّن مجلد العمل صراحة.

```
add_test(
  NAME parser.fixture
  COMMAND parser_test
          "${CMAKE_CURRENT_SOURCE_DIR}/data/valid-input.txt"
)
```

لا ينبغي للاختبار تعديل بيانات مصدر مشتركة. انسخ الملفات القابلة للكتابة إلى شجرة البناء:

```
configure_file(
  data/template.json
  data/template.json
  COPYONLY
)
```

تذهب الوجهة النسبية إلى مجلد البناء الحالي.

الاختبارات والأهداف المولّدة في CMake 4.4

مع مولّدات Ninja المدعومة يستطيع CMake 4.4 إنشاء أهداف باسم <test_prep/<test-name> تبني التبعيات التي يحتاج إليها اختبار منفرد. تتحكم بالميزة CMAKE_TEST_BUILD_DEPENDS وتبعيات الاختبارات المرتبطة بها.

تساعد الميزة سير العمل الذي يريد بناء ما يحتاج إليه اختبار واحد فقط، لكن التسلسل التقليدي يظل واضحًا ومحمولًا:

```
cmake --build build
ctest --test-dir build
```

إخراج JUnit

تفهم أنظمة CI كثيرة XML بصيغة JUnit:

```
ctest --test-dir build \
  --output-on-failure \
  --output-junit test-results.xml
```

احتفظ أيضًا بسجل الطرفية المقروء للبشر. تفيد ملفات XML لوحات المتابعة، لكنها أقل راحة للتشخيص الفوري.

المطهّرات وتشغيل الاختبارات

المطهّرات ميزات في المصرّف ووقت التشغيل، وليست ميزات في CTest. اربط الخيارات بالأهداف المختبرة عبر مكتبة واجهة أو دالة مساعدة، ثم شغّل الاختبارات الناتجة بواسطة CTest.

```
add_library(project_sanitizers INTERFACE)

target_compile_options(project_sanitizers INTERFACE
  $<$<CXX_COMPILER_ID:GNU,Clang>:-fsanitize=address,undefined>
)
target_link_options(project_sanitizers INTERFACE
  $<$<CXX_COMPILER_ID:GNU,Clang>:-fsanitize=address,undefined>
)

target_link_libraries(unit_tests PRIVATE project_sanitizers)
```

اجعل المطهّرات اختيارية ومراعية لإعداد البناء. ويجب أن يتوفر وقت تشغيل المطهّر عند تنفيذ الاختبار.

الملفات المولّدة والأوامر المخصصة

هل يتم التوليد في وقت الإعداد أم وقت البناء؟

استخدم `configure_file()` عندما يعتمد الناتج على قيم الإعداد فقط، ويمكن إنشاؤه أثناء قيام CMake بالإعداد:

```
configure_file(
  config.hpp.in
  "${CMAKE_CURRENT_BINARY_DIR}/generated/orbit/config.hpp"
  @ONLY
)
```

واستخدم `add_custom_command(OUTPUT ...)` عندما يجب تشغيل أداة خارجية ضمن عملية البناء، ويكون الناتج مرتبطًا بتبعيات وقت البناء.

اختيار المجال الزمني الصحيح يمنع عمليات إعادة البناء غير الضرورية، ويمنع فقدان روابط التبعية داخل الرسم البياني للبناء.

ترويسات الإعداد

الملف `config.hpp.in`:

```
#pragma once

#define ORBIT_VERSION "@PROJECT_VERSION@"
#cmakedefine01 ORBIT_WITH_ZSTD
```

وفي CMake:

```
option(ORBIT_WITH_ZSTD "Enable Zstandard support" OFF)

configure_file(
  config.hpp.in
  "${CMAKE_CURRENT_BINARY_DIR}/generated/orbit/config.hpp"
  @ONLY
)

target_include_directories(orbit PRIVATE
  "${CMAKE_CURRENT_BINARY_DIR}/generated"
)
```

ينتج `#cmakedefine01` قيمة رقمية `0` أو `1`، وهي سهلة الاستخدام داخل شروط `#if`.

مولّد صحيح يعمل في وقت البناء

```
find_package(Python3 REQUIRED COMPONENTS Interpreter)

set(generated_header
  "${CMAKE_CURRENT_BINARY_DIR}/generated/build_stamp.hpp"
)

add_custom_command(
  OUTPUT "${generated_header}"
  COMMAND "${CMAKE_COMMAND}" -E make_directory
          "${CMAKE_CURRENT_BINARY_DIR}/generated"
  COMMAND "${Python3_EXECUTABLE}"
          "${CMAKE_CURRENT_SOURCE_DIR}/tools/generate_header.py"
          "${generated_header}"
  DEPENDS
          "${CMAKE_CURRENT_SOURCE_DIR}/tools/generate_header.py"
  VERBATIM
)

add_executable(generated_demo
  src/main.cpp
  "${generated_header}"
)

target_include_directories(generated_demo PRIVATE
  "${CMAKE_CURRENT_BINARY_DIR}/generated"
)
```

يظهر الملف المولّد في قائمة مصادر الهدف، فيصبح للأمر المخصص مستهلك داخل رسم البناء. يحدد DEPENDS متى يجب إعادة التوليد، بينما يطلب VERBATIM من CMake اقتباس وسائط الأمر بطريقة صحيحة للمنصة والمولّد.

الفرق بين OUTPUT وBYPRODUCTS

تسمي OUTPUT المنتجات الأساسية المولّدة التي تحدد طوابعها الزمنية ما إذا كان الأمر يحتاج إلى التشغيل. أما BYPRODUCTS فتسجل الملفات الإضافية التي ينشئها الأمر.

```
add_custom_command(
  OUTPUT generated/api.cpp
  BYPRODUCTS generated/api.hpp generated/schema.bin
  COMMAND api_generator ...
  DEPENDS schema.idl
  VERBATIM
)
```

يهم تسجيل المنتجات الجانبية خصوصًا مع Ninja، وعند التنظيف، وللتبعيات اللاحقة.

لا تعتمد على صيغة الصدفة بلا حاجة

الأمر التالي هش:

```
add_custom_command(
  OUTPUT combined.txt
  COMMAND cat a.txt b.txt > combined.txt
)
```

إعادة التوجيه > جزء من لغة الصدفة، في حين يشغّل COMMAND عملية مباشرة عادةً. تختلف صدف Windows، وتصبح قواعد الاقتباس صعبة.

فضّل برنامج CMake نصيًا أو Python أو عملية محمولة من cmake -E:

```
COMMAND "${CMAKE_COMMAND}" -E copy_if_different
        "${input}" "${output}"
```

أضاف CMake 4.2 الأمرين `copy_if_newer` و`copy_directory_if_newer` للنسخ المعتمد على الطابع الزمني، ووسع CMake 4.4 أوامر النسخ في `cmake -E` بخيارات لمجلد الهدف.

الأهداف المخصصة

لا يملك الهدف المخصص ناتجًا رئيسيًا، ويُعد دائمًا قديمًا عندما يُطلب بناؤه صراحة:

```
add_custom_target(format
  COMMAND clang-format -i ${project_sources}
  WORKING_DIRECTORY "${PROJECT_SOURCE_DIR}"
  VERBATIM
)
```

لا تضف أهداف التنسيق أو الوثائق إلى `ALL` إلا إذا كان ينبغي لكل بناء عادي تشغيلها. يكون الأمر المخصص ذو المخرجات المعلنة أفضل للمنتجات المولّدة ضمن البناء.

اعتمد على الأهداف لا على الملفات فقط

```
add_custom_command(
  OUTPUT embedded_resources.cpp
  COMMAND resource_packer
          "$<TARGET_FILE:resource_tool>"
          embedded_resources.cpp
  DEPENDS resource_tool resource_manifest.json
  VERBATIM
)
```

تنتج تعبيرات المولّد الواعية بالأهداف، مثل `$<TARGET_FILE:...>`، المسار الصحيح للملف التنفيذي. كما ينشئ `DEPENDS` علاقة ترتيب البناء `resource_tool`.

الملفات المولّدة مع الإعدادات المتعددة

ينبغي للأمر المخصص الذي ينتج ملفًا خاصًا بكل إعداد أن يضع اسم الإعداد في المسار:

```
set(output_dir "${CMAKE_CURRENT_BINARY_DIR}/generated/$<CONFIG>")
```

يختلف دعم تعبيرات المولّد بين مواضع الوسائط وإصدارات CMake؛ لذلك راجع توثيق موضع الأمر المحدد. لا تسمح لبناء Debug وRelease بالكتابة فوق ملف واحد داخل شجرة متعددة الإعدادات.

مولّدات الشيفرة التي تعمل على المضيف أثناء البناء المتقاطع

عند البناء المتقاطع يُبنى الهدف للآلة المستهدفة، وقد لا يعمل على المضيف. ينبغي بناء أدوات المضيف بصورة مستقلة، أو استيرادها، أو توفيرها عبر بناء من مرحلتين.

بنية شائعة:

١. ينتج بناء أصلي أداة توليد الشيفرة.

٢. يستقبل البناء المتقاطع مسارها من متغير في الذاكرة المخبأة أو من هدف تنفيذي مستورد.

٣. تستخدم الأوامر المخصصة أداة المضيف لتوليد مصادر الهدف.

لا تفترض أن `CMAKE_CROSSCOMPILING_EMULATOR` يحل كل حالات أدوات المضيف. يفيد عندما يستطيع الملف التنفيذي المستهدف العمل بواسطة محاكي، لكن بناء أدوات مضيف مستقلًا يكون أوضح في كثير من المشاريع.

البناء متعدد المنصات وملفات سلسلة الأدوات

تبدأ قابلية النقل من التعبير عن نية الهدف

ينبغي للمشروع المحمول أن يصف:

- ميزات اللغة المطلوبة.
- تبعيات الأهداف المرتبطة.
- مجلدات التضمين.
- تعريفات الترجمة.
- وجهات التثبيت.
- الملفات المولّدة.
- ملفات المصدر الخاصة بالمنصة عند الحاجة فقط.

ولا ينبغي له تخمين كل خيار مصرّف أو لاحقة اسم ملف يدويًا.

```
add_library(core src/core.cpp)
target_compile_features(core PUBLIC cxx_std_20)
```

تحدد CMake لواحق ملفات الكائنات وامتدادات الملفات التنفيذية وأوامر الأرشفة واستدعاء الرابط الاعتيادي.

اختبارات المنصة

توفر CMake متغيرات مثل `WIN32` و`APPLE` و`UNIX` و`MSVC` ومعرّفات المصرّفات. استخدم أضيق شرط يمثل الحاجة الفعلية.

```
if(WIN32)
  target_sources(core PRIVATE src/platform/windows_registry.cpp)
elseif(APPLE)
  target_sources(core PRIVATE src/platform/macos_keychain.mm)
else()
  target_sources(core PRIVATE src/platform/posix_store.cpp)
endif()
```

تكون `UNIX` صحيحة على macOS أيضًا. لذلك لا تكتب `if(UNIX) ... elseif(APPLE)` عندما تحتاج Apple إلى فرع منفصل؛ لأن الشرط الأول سيطابقها بالفعل.

فضّل معرّفات المصرّفات على افتراضات المنصة عند اختيار الخيارات:

```
target_compile_options(core PRIVATE
  $<$<CXX_COMPILER_ID:MSVC>:/W4>
  $<$<CXX_COMPILER_ID:GNU,Clang,AppleClang>:-Wall;-Wextra>
)
```

يستطيع Clang استهداف Windows، كما يستطيع `clang-cl` استخدام سطر أوامر بأسلوب MSVC. وقد تهم متغيرات واجهة المصرّف والمحاكاة في الحالات المتقدمة.

ملفات سلسلة الأدوات

يُقرأ ملف سلسلة الأدوات مبكرًا قبل أن يفعّل المشروع اللغات. وهو يختار المصرّفات ويصف المنصة المستهدفة.

```
# toolchains/linux-aarch64.cmake
set(CMAKE_SYSTEM_NAME Linux)
set(CMAKE_SYSTEM_PROCESSOR aarch64)

set(CMAKE_C_COMPILER aarch64-linux-gnu-gcc)
set(CMAKE_CXX_COMPILER aarch64-linux-gnu-g++)

set(CMAKE_FIND_ROOT_PATH /usr/aarch64-linux-gnu)
set(CMAKE_FIND_ROOT_PATH_MODE_PROGRAM NEVER)
set(CMAKE_FIND_ROOT_PATH_MODE_LIBRARY ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_INCLUDE ONLY)
set(CMAKE_FIND_ROOT_PATH_MODE_PACKAGE ONLY)
```

الإعداد:

```
cmake -S . -B build/aarch64 -G Ninja \
  --toolchain toolchains/linux-aarch64.cmake
```

أو ضع ملف سلسلة الأدوات داخل إعداد مسبق.

لا تستخدم متغيرات المشروع الأعلى داخل ملف سلسلة الأدوات

قد يُقيّم ملف سلسلة الأدوات أكثر من مرة وفي سياقات مثل `try_compile()`. وقد تشير `CMAKE_SOURCE_DIR` و`CMAKE_BINARY_DIR` إلى مشاريع فحص مؤقتة.

استخدم `CMAKE_CURRENT_LIST_DIR` للعثور على الملفات المجاورة لملف سلسلة الأدوات:

```
set(SDK_ROOT "${CMAKE_CURRENT_LIST_DIR}/../sdk")
```

try_compile() وفحوص الميزات

قد تترجم CMake ووحداتها برامج اختبار صغيرة لتحديد قدرات المصرّف والمنصة. أثناء البناء المتقاطع قد يتعذر ربط ملف تنفيذي للاختبار حتى عندما تنجح الترجمة.

يمكن لملف سلسلة الأدوات تعيين:

```
set(CMAKE_TRY_COMPILE_TARGET_TYPE STATIC_LIBRARY)
```

يطلب ذلك من الفحوص الملائمة ترجمة مكتبة ساكنة بدل ربط ملف تنفيذي. استخدمه فقط عندما تحتاجه سلسلة الأدوات؛ فبعض الفحوص تحتاج فعلًا إلى الربط أو التنفيذ.

فحص الميزات والتويسات والرموز

استخدم وحدات الفحص في CMake بدل تشغيل المصرّف يدويًا:

```
include(CheckCXXSourceCompiles)

check_cxx_source_compiles([[
  #include <version>
  int main() {
  #if !defined(__cpp_lib_expected)
    #error expected unavailable
  #endif
  }
]] ORBIT_HAS_STD_EXPECTED)
```

لكن اسأل أولًا: هل الفحص ضروري؟ تعبّر `target_compile_features()` وأهداف الحزم غالبًا عن المتطلبات بصورة أوضح من متغيرات الفحص.

Emscripten

أضاف CMake 4.2 دعمًا مبسطًا للبناء المتقاطع باستخدام Emscripten عبر ملفات سلسلة الأدوات. عمليًا يستخدم المطورون غالبًا ملف سلسلة الأدوات المرفق مع Emscripten SDK:

```
emcmake cmake -S . -B build/wasm -G Ninja
cmake --build build/wasm
```

أو يمكن تحديد مسار ملف سلسلة أدوات SDK صراحة. وقد تحتاج نواتج المتصفح وNode.js إلى خيارات ربط خاصة بالهدف، مثل الدوال المصدرة أو إعدادات الذاكرة أو الوحدات؛ أبقِ هذه الخيارات مرتبطة بالهدف.

Android وiOS

تستخدم عمليات بناء Android عادةً سلسلة أدوات Android NDK ومتغيرات مثل ABI ومستوى المنصة واختيار STL. أما iOS فيستخدم غالبًا مولّد Xcode مع `CMAKE_SYSTEM_NAME=iOS` وإعدادات SDK والمعمارية وهدف النشر.

تفرض هذه الأنظمة البيئية قيودًا للتحزيم والتوقيع وبيئات التطوير تتجاوز الترجمة. تمثل CMake عملية البناء، لكن أدوات النشر الأصلية للمنصة تظل ضرورية.

اختيار مكتبة وقت تشغيل Windows

استخدم `MSVC_RUNTIME_LIBRARY` بدل تعديل `/MD` و`/MT` ونسخ Debug يدويًا:

```
set_property(TARGET core PROPERTY
  MSVC_RUNTIME_LIBRARY "MultiThreaded$<$<CONFIG:Debug>:Debug>DLL"
)
```

تحتاج الخاصية إلى سلوك السياسة الحديثة المناسب، وتنطبق على سلاسل الأدوات المتوافقة مع ABI الخاص بـ MSVC.

الخيوط

لا تضف `-pthread` يدويًا:

```
find_package(Threads REQUIRED)
target_link_libraries(app PRIVATE Threads::Threads)
```

يمثل الهدف المستورد متطلبات الترجمة والربط الصحيحة للمنصة.

أطر المنصة ومكتبات النظام

فضّل أهداف الحزم أو المنصة عندما تكون متاحة. على منصات Apple يمكن التعبير عن ربط الأطر بواسطة الروابط والخيارات المدعومة للأهداف. وعلى Windows يمكن ربط مكتبات النظام بالاسم، لكن اجمعها داخل هدف واجهة محلي عندما تحتاج عدة أهداف إلى خدمة منصة متسقة:

```
add_library(platform_sockets INTERFACE)
if(WIN32)
  target_link_libraries(platform_sockets INTERFACE ws2_32)
endif()
```

يربط المستهلكون بعد ذلك `platform_sockets` بدل تكرار الشروط.

الأداء وتسريع البناء

قِس قبل أن تغيّر نظام البناء

يتكون أداء البناء من عناصر متعددة:

- زمن الإعداد والتوليد.
- فحص التبعيات.
- زمن الترجمة.
- زمن الربط.
- زمن الاختبارات.
- زمن التثبيت والتحزيم.
- زمن الشبكة ومدير الحزم.

قد يحسن حل عنصرًا ويضر عنصرًا آخر. قد تقلل عمليات Unity زمن البناء الكامل، لكنها تزيد كلفة إعادة البناء الجزئي. وقد يحسن LTO أداء البرنامج أثناء التشغيل، لكنه يبطئ الربط. ويفيد مخبأ المصرّف عمليات البناء المتكررة، لكنه لا يفيد أول بناء على مخبأ جديد.

اختر Ninja واستخدم التوازي

```
cmake -S . -B build -G Ninja
cmake --build build --parallel
```

يوفر Ninja غالبًا جدولة سريعة للبناء الجزئي ومخرجات مختصرة. ويجنب خيار `--parallel` في CMake الحاجة إلى صيغة مهام خاصة بالمولّد.

قد يكون التوازي غير المحدود أبطأ أو يؤدي إلى الفشل داخل CI محدود الذاكرة. عيّن عددًا مناسبًا من المهام عبر الإعدادات المسبقة أو `CMAKE_BUILD_PARALLEL_LEVEL`.

مشغّلات المصرّف وأدوات التخزين المؤقت

توفر CMake نقاط تهيئة عامة وخاصة بالأهداف لمشغلات المصرّف:

```
find_program(CCACHE_PROGRAM ccache)
if(CCACHE_PROGRAM)
  set(CMAKE_CXX_COMPILER_LAUNCHER "${CCACHE_PROGRAM}")
endif()
```

يهيئ ذلك الأهداف التي تُنشأ بعده. ويتوفر تحكم لكل هدف بواسطة `CXX_COMPILER_LAUNCHER`.

تكون مخابئ المصرّف أكثر فاعلية عندما تكون أسطر الأوامر ومسارات المصدر قابلة لإعادة الإنتاج. تقلل الطوابع الزمنية المولّدة غير الثابتة، واختلاف المسارات المطلقة، وتغير خيارات المصرّف من معدل الإصابات الناجحة في المخبأ.

على Windows قد تعمل أدوات مثل `sccache` مع MSVC وClang وبيئات GCC. اختر أداة التخزين وفق المصرّف وبنية المؤسسة.

الترويسات المترجمة مسبقًا

```
target_precompile_headers(core PRIVATE
  <algorithm>
  <memory>
  <string>
  <vector>
)
```

يمكن لـ PCH تسريع الأهداف الكبيرة ذات الترويسات الثابتة والمكلفة. لكنها قد تخفي تضمينات ناقصة إذا كانت الملفات المصدرية تترجم فقط لأن PCH وفرت تصريحات بالصدفة.

نادراً ما يكون نشر PCH علنًا مناسبًا لمكتبة مثبتة. استخدمها تحسينًا خاصًا للبناء، وأبقِ كل ملف مصدر مكتفيًا بتضميناته من الناحية الدلالية.

بناء Unity

```
set_target_properties(core PROPERTIES UNITY_BUILD ON)
```

تجمع CMake عدة ملفات مصدر داخل وحدات ترجمة Unity مولّدة. وقد يقلل ذلك تكرار تحليل الترويسات في البناء النظيف.

المخاطر تشمل:

- تعارض الأسماء أو الماكروات المحلية للملفات.
- تغير ترتيب التضمين.
- بناء جزئي أسوأ.
- استهلاك ذاكرة أكبر في المصرّف.
- إخفاء مشكلات نظافة المصدر.

فعّل Unity بوصفه نمط أداء اختياريًا، واختبر بناء Unity والبناء العادي معًا.

يمكن استثناء ملفات منفردة بواسطة `SKIP_UNITY_BUILD_INCLUSION`، وقد وسع CMake 4.4 هذه الضوابط لتشمل مجموعات الملفات.

التحسين بين الإجراءات

```
include(CheckIPOSupported)
check_ipo_supported(RESULT ipo_supported OUTPUT ipo_error)

if(ipo_supported)
  set_property(TARGET app PROPERTY
    INTERPROCEDURAL_OPTIMIZATION_RELEASE TRUE
  )
endif()
```

تعبّر هذه الصيغة عن IPO/LTO بصورة محمولة. لا تفترض أن `flto-` يعمل بالطريقة نفسها مع كل مصرّف ورابط وأداة أرشفة وتركيبة تبعيات.

طبّق IPO على الإعدادات الشبيهة بالإصدار النهائي عندما تبرر فائدته في وقت التشغيل كلفة البناء الإضافية.

معلومات التصحيح وأدوات الربط

يفيد `RelWithDebInfo` غالبًا عند تحليل أداء شيفرة قريبة من الإنتاج. وفي مشاريع Linux الكبيرة قد تقلل روابط بديلة مثل `lld` أو `mold` أو الروابط التي تدعمها CMake من زمن الربط. استخدم آليات اختيار الرابط الموثقة لإصدار CMake والمصرّف لديك بدل إدخال `-fuse-ld` عالميًا من دون اختبار.

أضاف CMake 4.4 دعمًا وتعريفًا لسلاسل ربط إضافية، ومنها الرابط `wild`. ولا تهم هذه الميزات إلا إذا كانت سلسلة أدوات المنصة توفر الرابط وكان المشروع قد اختبره.

الانضباط في الترويسات

أسرع ترويسة هي التي لا تحتاج إلى التحليل. لا تستطيع إعدادات نظام البناء تعويض تصميم يضمّن فيه كل ملف مصدر ترويسة شاملة ضخمة.

ممارسات نافعة في تصميم C++:

- قلل تبعيات الترويسات العامة.
- استخدم التصريحات المسبقة عندما تكون صحيحة.
- انقل تفاصيل التنفيذ خارج الترويسات.
- تجنب التوسع غير الضروري للقوالب داخل ترويسات واسعة الاستخدام.
- لا تستخدم وحدات C++ قبل التحقق من دعم سلسلة الأدوات.
- تحقق من كل ترويسة عامة بصورة مستقلة.

تستطيع CMake المساعدة في التحقق من مجموعات الترويسات:

```
set_target_properties(mylib PROPERTIES
  VERIFY_INTERFACE_HEADER_SETS ON
)
```

أضاف CMake 4.3 ضوابط للتحقق من مجموعات الترويسات الخاصة، وحسن CMake 4.4 سلوك أهداف التحقق.

وحدات C++20

تدعم CMake مجموعات ملفات وحدات C++ مع المصرّفات والمولّدات المتوافقة:

```
target_sources(math
  PUBLIC
    FILE_SET CXX_MODULES
    FILES math.cppm
)
```

يعتمد دعم الوحدات بدرجة كبيرة على إصدار CMake وإصدار المصرّف ونمط الواجهة والمولّد وفاحص التبعيات. وسع CMake 4.4 دعم المصرّفات، بما فيه الوحدات المسماة مع `clang-cl` خارج مولّدات Visual Studio.

لا تدّعِ أن المشروع محمول لمجرد نجاح إعداد `CXX_MODULES` على جهاز واحد. اختبر مصفوفة المصرّف والمولّد الفعلية المستخدمة لدى المساهمين وCI.

القياس التشغيلي للبناء

أضاف CMake 4.3 نظام instrumentation لجمع بيانات الزمن والأهداف والتشخيص عبر خطوات الإعداد والتوليد والبناء والاختبار والتثبيت. ويمكنه إصدار بيانات للاستدعاءات الراجعة وCDash وعرض التتبعات.

وسع CMake 4.4 النظام بإمكانية التقاط المخرجات وجمع تتبعات المصرّف. وهذه الميزة قيمة عند تحسين بناء معقد؛ لأنها تستبدل التخمين بأدلة منظمة.

للإصدارات السابقة استخدم، حيث يكون مدعومًا:

```
cmake -S . -B build --profiling-output=cmake-profile.json \
  --profiling-format=google-trace
```

واجمع ذلك مع سجلات Ninja وتقارير زمن المصرّف وبيانات توقيت CI.

لا تثبّت خيارات تحسين عدوانية يدويًا

الصيغة التالية ليست إعدادًا افتراضيًا محمولًا:

```
set(CMAKE_CXX_FLAGS_RELEASE "-O3 -march=native -DNDEBUG")
```

تشمل المشكلات: الكتابة فوق افتراضات سلسلة الأدوات، وكسر المصرّفات غير المبنية على GNU، وإنتاج ملفات لا تعمل على أجهزة أخرى، والتأثير في التبعيات.

إذا قدم المشروع تحسينًا خاصًا بالمعمارية، فاجعله صريحًا ومقيدًا بالهدف:

```
option(ORBIT_NATIVE_ARCH "Optimize for the build machine" OFF)
if(ORBIT_NATIVE_ARCH)
  target_compile_options(orbit PRIVATE
    $<$<CXX_COMPILER_ID:GNU,Clang>:-march=native>
  )
endif()
```

لا تفعّل `-march=native` لحزم إصدار محمولة أو منتجات CI عامة إلا إذا كان ذلك هو الغرض المعلن.

التكامل المستمر وسير العمل القابل لإعادة الإنتاج

ينبغي لـ CI استخدام الواجهة نفسها التي يستخدمها المطورون

يُبنى سير العمل الأكثر موثوقية حول الإعدادات المسبقة:

```
cmake --preset ci-linux
cmake --build --preset ci-linux
ctest --preset ci-linux
cpack --preset ci-linux
```

يمنع ذلك ملف YAML الخاص بـ CI من التحول إلى لغة إعداد بناء ثانية مختلفة عن المشروع.

مثال بسيط متعدد المنصات باستخدام GitHub Actions

```
name: build

on:
  push:
  pull_request:

jobs:
  test:
    strategy:
      fail-fast: false
      matrix:
        os: [ubuntu-latest, windows-latest, macos-latest]
    runs-on: ${{ matrix.os }}

    steps:
      - uses: actions/checkout@v4

      - name: Configure
        run: cmake --preset ci

      - name: Build
        run: cmake --build --preset ci

      - name: Test
        run: ctest --preset ci
```

تحتاج المصفوفة الحقيقية عادةً إلى إعدادات مسبقة خاصة بكل نظام تشغيل أو إلى مصرّفات شرطية. اجعل YAML مسؤولًا عن تجهيز البيئة واستدعاء سير العمل، وأبقِ إعداد CMake داخل الإعدادات المسبقة.

ثبّت البيئة

تحتاج قابلية إعادة الإنتاج إلى أكثر من تثبيت صيغة CMake. سجّل أو تحكم في:

- إصدار CMake.
- إصدار المصرّف والرابط.
- المولّد.
- إصدارات التبعيات.
- خط أساس مدير الحزم أو بيانات القفل.
- إصدارات SDK وsysroot.
- متغيرات البيئة التي تؤثر في الاكتشاف.
- إعدادات البناء وخيارات الميزات.

تساعد الحاويات على Linux، لكنها لا تستبدل الاختبار الأصلي على Windows وmacOS والهواتف والمنصات المرتبطة بعتاد محدد.

البناء النظيف والبناء الجزئي

شغّل إعدادات نظيفة دوريًا لاكتشاف التبعيات غير المعلنة والملفات المولّدة الناقصة. وشغّل كذلك البناء الجزئي لحماية إنتاجية المطور.

يتضمن تصميم CI جيد:

١. بناء سريعًا لطلبات الدمج مع إعادة استخدام المخبأ.

٢. بناءً نظيفًا مجدولًا.

٣. اختبارات التثبيت واستهلاك الحزمة.

٤. إعدادات للمطهّرات والتحليل الساكن.

٥. اختبارات أولية للتحزيم.

٦. عدة مصرّفات عندما يعد المشروع بدعمها.

استخدم التخزين المؤقت بحذر

خزّن تنزيلات التبعيات ومخرجات مخبأ المصرّف، لا مجلدات بناء قديمة عشوائية من دون مفاتيح مرتبطة بالإصدارات.

قد يشمل مفتاح المخبأ:

- نظام التشغيل.
- هوية المصرّف وإصداره.
- إصدار CMake.
- تجزئة ملف قفل التبعيات.
- إعداد البناء.
- خيارات الميزات المهمة.

قد يؤدي استرجاع `CMakeCache.txt` غير متوافق إلى أعطال دقيقة. تخزّن فرق كثيرة مجلدات مخبأ المصرّف وتنزيلات مدير الحزم، بينما تعيد إنشاء شجرة بناء CMake.

معاملة التحذيرات كأخطاء

تفيد معاملة التحذيرات كأخطاء في الشيفرة التي يملكها المشروع، لكنها قد تجعل التبعيات الخارجية تفشل بعد تحديث المصرّف.

```
option(ORBIT_WARNINGS_AS_ERRORS "Treat Orbit warnings as errors" OFF)

if(ORBIT_WARNINGS_AS_ERRORS)
  set_property(TARGET orbit PROPERTY COMPILE_WARNING_AS_ERROR ON)
endif()
```

تسمح `COMPILE_WARNING_AS_ERROR` لـ CMake باختيار سلوك المصرّف المناسب. طبّقها على أهدافك لا عالميًا.

فعّلها في إعدادات CI مسيطر عليها، واترك البناء المحلي وبناء المستهلكين مرنًا.

التحليل الساكن

تستطيع خصائص أهداف CMake دمج الأدوات مع مولّدات Makefile وNinja:

```
set_target_properties(orbit PROPERTIES
  CXX_CLANG_TIDY "clang-tidy;--warnings-as-errors=*"
)
```

تدعم خصائص مرتبطة أدوات مثل include-what-you-use وcpplint، وفي إصدارات CMake الحديثة تكامل PVS-Studio.

قد يبطئ التحليل الساكن البناء كثيرًا. خصص له إعدادًا مسبقًا أو مهمة CI بدل تحميل كل بناء عادي هذه الكلفة.

تغطية الاختبارات

تحتاج التغطية إلى خيارات مصرّف وتنفيذ الاختبارات وأداة لإعداد التقرير. اعزلها في إعداد خاص:

```
add_library(project_coverage INTERFACE)

target_compile_options(project_coverage INTERFACE
  $<$<CXX_COMPILER_ID:GNU,Clang>:--coverage;-O0;-g>
)
target_link_options(project_coverage INTERFACE
  $<$<CXX_COMPILER_ID:GNU,Clang>:--coverage>
)
```

اربط الهدف داخل إعداد التغطية فقط. ولا تخلط بيانات تغطية ناتجة من عمليات بناء مختلفة.

التحقق من التثبيت والتحزيم

يفوّت بناء CI الذي يترجم شجرة البناء فقط عيوبًا كثيرة في التحزيم. أضف خطوات مثل:

```
cmake --install build --prefix stage
cmake -S tests/consumer -B consumer-build \
  -DCMAKE_PREFIX_PATH="$PWD/stage"
cmake --build consumer-build
```

ثم أنشئ الحزمة وافحص الأرشيف أو المثبّت. تحقق من احتوائه على التراخيص وملفات وقت التشغيل والترويسات العامة وبيانات الإصدار، ومن خلوه من مسارات بناء مطلقة غير مقصودة.

إعدادات سير عمل CMake المسبقة

منذ CMake 3.25 يستطيع إعداد سير العمل توثيق خط التنفيذ:

```
{
  "workflowPresets": [
    {
      "name": "verify",
      "steps": [
        { "type": "configure", "name": "ci" },
        { "type": "build", "name": "ci" },
        { "type": "test", "name": "ci" },
        { "type": "package", "name": "ci" }
      ]
    }
  ]
}
cmake --workflow --preset verify
```

لا يستبدل ذلك تنسيق CI أو رفع المنتجات أو التوقيع أو النشر، لكنه يوفر تسلسل بناء متسقًا يملكه المشروع.

تشخيص مشكلات CMake

اقرأ أول خطأ ذي معنى

قد تطبع CMake وأدوات البناء الأصلية سلسلة طويلة من الأعطال. وغالبًا لا يقول السطر الأخير إلا إن هدفًا أعلى قد فشل. ابدأ بأول رسالة تسمي الملف المفقود، أو الأمر الفاشل، أو الرمز غير المحلول، أو مشكلة السياسة.

حدد المرحلة أولًا:

- خطأ الإعداد: لغة CMake، أو اكتشاف الحزم، أو فحوص المصرّف، أو الذاكرة المخبأة.
- خطأ التوليد: رسم أهداف غير صالح، أو تعبير مولّد، أو قاعدة تصدير.
- خطأ البناء: المصرّف، أو الرابط، أو ناتج مولّد مفقود، أو أداة البناء الأصلية.
- خطأ الاختبار: سلوك وقت التشغيل أو بيئة الاختبار.
- خطأ التثبيت أو التحزيم: منتج أو وجهة أو مكوّن أو تبعية وقت تشغيل مفقودة.

اجعل الأوامر صريحة

استخدم مخرجات بناء تفصيلية:

```
cmake --build build --verbose
```

ولتتبع منطق إعداد CMake نفسها:

```
cmake -S . -B build --trace-expand
```

تتبع كل شيء كثير الضوضاء. ضيّق النطاق إلى ملفات المشروع عندما يكون ذلك ممكنًا:

```
cmake -S . -B build \
  --trace-source=cmake/Dependencies.cmake \
  --trace-expand
```

أضاف CMake 4.2 الأمر `cmake_language(TRACE)` للتحكم البرمجي في التتبع أثناء تنفيذ البرامج النصية.

مستويات السجل

```
message(STATUS "Configuring Orbit ${PROJECT_VERSION}")
message(VERBOSE "Search root: ${ORBIT_SDK_ROOT}")
message(DEBUG "Candidate list: ${candidates}")
```

اختر مستوى السجل أثناء الإعداد:

```
cmake -S . -B build --log-level=DEBUG
```

لا تترك مخرجات حالة ضخمة وغير مشروطة داخل مكتبة تستهلكها مشاريع أخرى.

فحص المتغيرات والخصائص

```
include(CMakePrintHelpers)

cmake_print_variables(
  CMAKE_CXX_COMPILER
  CMAKE_GENERATOR
  CMAKE_BUILD_TYPE
)

cmake_print_properties(
  TARGETS orbit
  PROPERTIES
    TYPE
    SOURCES
    INCLUDE_DIRECTORIES
    INTERFACE_INCLUDE_DIRECTORIES
    LINK_LIBRARIES
    INTERFACE_LINK_LIBRARIES
)
```

تحتوي شجرة البناء كذلك معلومات مولّدة مفيدة:

- `CMakeCache.txt`
- `CMakeFiles/CMakeConfigureLog.yaml`
- `compile_commands.json` عند تفعيله.
- ملفات المولّد الأصلي.
- بيانات الاختبارات.
- نواتج إعداد الحزم.

تصدير أوامر الترجمة

لمولّدات Ninja وMakefile:

```
cmake -S . -B build -DCMAKE_EXPORT_COMPILE_COMMANDS=ON
```

يعرض `compile_commands.json` استدعاء المصرّف الفعلي لكل وحدة ترجمة، ويدعم clangd وأدوات التحليل والتشخيص القابل لإعادة الإنتاج.

لا تعامله على أنه أمر الربط الكامل أو ميزة عامة لكل مولّد.

تشخيص اكتشاف الحزم

عندما تعجز `find_package()` عن العثور على حزمة:

```
cmake -S . -B build --debug-find-pkg=Foo
```

أو استخدم التشخيص الأوسع:

```
cmake -S . -B build --debug-find
```

تحقق من:

- اسم الحزمة وحالة أحرفه.
- نمط Config مقابل Module.

- `.CMAKE_PREFIX_PATH`

- تفعيل سلسلة أدوات مدير الحزم.

- المعمارية وإعداد البناء.

- تثبيت حزمة التطوير، لا وقت التشغيل فقط.

- عدم احتفاظ مخبأ قديم بقيمة `Foo_DIR-NOTFOUND` أو بمسار سابق.

في حزم Config يجب أن يشير `Foo_DIR` إلى المجلد الذي يحتوي `FooConfig.cmake`، لا إلى مجلد التضمين أو المكتبة.

تشخيص فشل مسارات المصدر

تعني رسالة مثل:

```
No rule to make target '.../src/main.cpp'
```

أن نظام البناء المولّد يعتقد أن الملف دخل في البناء، لكنه لا يستطيع العثور عليه. تحقق:

```
ls -l src/main.cpp
cmake -S . -B build --fresh
cmake --build build --verbose
```

على Windows مع Cygwin أو MSYS، قارن صيغة المسار التي تطبعها CMake بما يفهمه المصرّف وأداة البناء المختاران. وتأكد أن جميع الأدوات تأتي من البيئة المقصودة.

ابحث في الملفات المولّدة للتشخيص فقط:

```
grep -R "main.cpp" build
```

لا ترقع ملفات Makefile المولّدة. صحح مسار المصدر أو البيئة أو وصف CMake، ثم أعد التوليد.

أخطاء الربط

تعني المراجع غير المعرفة عادةً أحد الأسباب الآتية:

- الهدف غير مربوط بتبعيته المباشرة.

- ملف مصدر مطلوب غير موجود ضمن هدف المكتبة.

- أُعلنت تبعية `PRIVATE` بينما يجب أن ينتقل متطلب ربطها إلى المستهلكين.

- ترتيب المكتبات الساكنة أو التبعيات الدائرية يكشف مشكلة تصميم.

- خلط ملفات Debug وRelease.

- اختلاف أسماء الربط بين C و++C.

- عدم توافق ABI أو مكتبة وقت التشغيل.

افحص أمر الربط التفصيلي ورسم الأهداف. لا تحاول حل المشكلة بنسخ كل مكتبة إلى كل استدعاء `target_link_libraries()`.

تعبيرات المولّد

تُقيّم تعبيرات المولّد متأخرًا، وقد يصعب طباعتها بواسطة `message()`. أنشئ ملفات تشخيص أو أهدافًا مخصصة عند الحاجة:

```
file(GENERATE
  OUTPUT "${CMAKE_CURRENT_BINARY_DIR}/debug/$<CONFIG>/includes.txt"
  CONTENT "$<JOIN:$<TARGET_PROPERTY:orbit,INCLUDE_DIRECTORIES>,\n>"
)
```

يُنتج الملف في مرحلة التوليد لكل إعداد مناسب.

Graphviz

تستطيع CMake إنشاء رسم للتبعيات:

```
cmake --graphviz=build/targets.dot build
```

ثم تحويله بواسطة Graphviz:

```
dot -Tsvg build/targets.dot -o build/targets.svg
```

قد تحتاج المشاريع الكبيرة إلى خيارات تستبعد أهداف الخدمات أو الأهداف الخارجية أو أهدافًا محددة.

نظام التشخيص في CMake 4.4

أضاف CMake 4.4 نظام تشخيص منظمًا تتحكم به `cmake_diagnostic()` وخيارات التحذير في سطر الأوامر وحقول التحذيرات داخل الإعدادات المسبقة. ويضيف فئات مثل تشخيص وجهات التثبيت المطلقة.

يحسن ذلك القدرة على إظهار فئات محددة من السلوك المشكوك فيه أو جعلها قاتلة، من دون الاعتماد على متغيرات غير مرتبطة. استخدم التشخيص لفرض جودة المشروع، لكن لا تحول كل تحذير للمطور إلى خطأ عالميًا من دون تقييم توافق الإصدارات والتبعيات المدعومة.

صغّر حالة الفشل

عندما تكون المشكلة غير واضحة، أنشئ مشروعًا أدنى يحتوي:

- ملف `CMakeLists.txt` واحدًا.
- ملف مصدر واحدًا أو اثنين.
- المولّد والمصرّف نفسيهما.
- أصغر علاقة أهداف تفشل.

يميّز المثال المصغر بين سلوك CMake وبين دوال المشروع الخاصة، وربط مدير الحزم، وحالة IDE، وسنوات من تاريخ الذاكرة المخبأة.

الممارسات الخاطئة والانتقال من CMake القديمة

مجلدات التضمين العامة

الأسلوب القديم:

```
include_directories(include ${FOO_INCLUDE_DIRS})
```

الأسلوب الحديث:

```
target_include_directories(core
  PUBLIC include
)
target_link_libraries(core PRIVATE Foo::Foo)
```

تجعل أوامر المجلد العامة ترتيب المعالجة مؤثرًا، وقد تنقل الإعداد إلى أهداف لا تحتاج إليه أصلًا. أما ربط مسارات التضمين والتبعيات بالهدف نفسه فيوضح نطاقها ويمنع التأثيرات الجانبية غير المقصودة.

خيارات المصرّف العامة

الأسلوب القديم:

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -O3 -march=native")
```

تسبب هذه الطريقة مشكلات عدة:

- قد تكرر الإعدادات الافتراضية لسلسلة الأدوات أو تستبدلها.
- تفترض أن المصرّف يفهم خيارات بأسلوب GNU.
- تؤثر في التبعيات التي لا تملكها أنت.
- تخلط سياسة التحذيرات بسياسة التحسين.
- تجعل الملفات الناتجة مرتبطة بمعالج جهاز البناء عند استخدام `-march=native`.

الأسلوب الحديث:

```
target_compile_options(core PRIVATE
  $<$<CXX_COMPILER_ID:GNU,Clang>:-Wall;-Wextra>
)
```

استخدم خصائص CMake المخصصة لـ IPO وPIC ومعيار اللغة وتحويل التحذيرات إلى أخطاء ومكتبة وقت تشغيل MSVC كلما وُجد تجريد جاهز للغرض. لا تحول خيارات المصرّف الخام إلى آلية عامة إن كانت CMake قادرة على التعبير عن المقصود بطريقة مستقلة عن الأداة.

مجلدات الربط

الأسلوب القديم:

```
link_directories(/opt/foo/lib)
target_link_libraries(app PRIVATE foo)
```

يطلب هذا الأسلوب من الرابط البحث داخل مجلد واختيار ملف اعتمادًا على الاسم، وقد يختار بنية معالج أو إعداد بناء غير صحيح.

فضّل هدفًا مستوردًا توفره `find_package()`. وإن اضطررت إلى نمذجة ملف مبني مسبقًا بنفسك فاكتب:

```
add_library(Foo::Foo UNKNOWN IMPORTED)
set_target_properties(Foo::Foo PROPERTIES
  IMPORTED_LOCATION "/opt/foo/lib/libfoo.a"
  INTERFACE_INCLUDE_DIRECTORIES "/opt/foo/include"
)
target_link_libraries(app PRIVATE Foo::Foo)
```

ينبغي للحزمة القابلة لإعادة الاستخدام أن تعرّف هذا الهدف داخل وحدة Find أو حزمة Config، لا أن تكرر التعريف داخل كل مشروع مستهلك.

البناء داخل شجرة المصدر

تشغيل الأمر:

```
cmake .
```

داخل جذر المصدر يضع ملفات الذاكرة المخبأة وملفات المولّد بجوار ملفات المشروع. عندها يصعب الاحتفاظ بإعدادات متعددة معًا، وتتلوث إدارة الإصدارات بالملفات المولّدة، ويصبح التنظيف محفوفًا بخطر حذف ملفات حقيقية.

استخدم بدلًا من ذلك:

```
cmake -S . -B out/build/dev
```

ويمكن للمشروع رفض البناء داخل المصدر صراحةً:

```
if(CMAKE_SOURCE_DIR STREQUAL CMAKE_BINARY_DIR)
  message(FATAL_ERROR "In-source builds are not supported. Use cmake -S . -B build")
endif()
```

تعديل الملفات المولّدة

لا تتعامل مطلقًا مع مشاريع Visual Studio أو ملفات Makefile أو ملفات Ninja المولّدة على أنها ملفات مصدر. ستختفي تعديلاتك عند إعادة التوليد، كما لن تنتقل إلى مولّد آخر.

ضع التغيير في الأهداف أو الخصائص أو ملفات سلسلة الأدوات أو الإعدادات المسبقة أو بيانات الحزمة الوصفية.

استدعاء project() في كل مجلد فرعي

يصلح `project()` للمشاريع المتداخلة الحقيقية التي تحتاج بيانات مشروع مستقلة أو تفعيل لغات خاصة بها. لكنه ليس مطلوبًا لمجرد وجود ملف `CMakeLists.txt` داخل المجلد.

قد تعيد الاستدعاءات الزائدة ضبط متغيرات مرتبطة بالمشروع وتربك المنطق الذي يميز المشروع الأعلى من المشروع الفرعي.

استخدام aux_source_directory() والجمع التلقائي للمصادر

يخفي الجمع التلقائي عضوية الملفات في الأهداف، مثلما تفعل أنماط glob الواسعة. تبقى قوائم المصادر الصريحة الخيار الطبيعي القابل للصيانة في معظم المشاريع.

ملفات حزم مكتوبة يدويًا تحوي مسارات شجرة البناء

الحزمة التي تخزن:

```
set(MYLIB_INCLUDE_DIR "/home/user/project/include")
```

ليست قابلة للنقل. استخدم الأهداف المصدّرة، و`configure_package_config_file()`، وواجهات تثبيت نسبية، واختبر الحزمة بواسطة مشروع مستهلك مستقل.

التحذيرات العامة والتبعيات الخاصة

يتكرر خطآن متعاكسان في تحديد النطاق:

١. تُوسم التحذيرات بـ PUBLIC، فيُجبر المستهلكون على وراثة سياسة الجودة المحلية للمكتبة المنتجة.

٢. تُستخدم مكتبة داخل ترويسة عامة، لكنها تُربط بـ PRIVATE، فلا يحصل المستهلكون على معلومات التضمين أو الربط المطلوبة.

اختر النطاق من الواجهة الحقيقية. اسأل: هل يجب على المستهلك أن يترجم أو يربط بهذا المتطلب لأنه يستخدم هدفي العام؟ إن كانت الإجابة نعم فهو جزء من متطلبات الاستخدام، وإلا فليبقَ خاصًا.

تكامل Conan القديم

احذف الأسلوب الآتي:

```
include(${CMAKE_BINARY_DIR}/conanbuildinfo.cmake)
conan_basic_setup()
```

ينشئ Conan 2 ملف سلسلة أدوات وملفات إعداد للتبعيات. اضبط المشروع باستخدام سلسلة الأدوات المولّدة، ثم استخدم find_package() مع الأهداف المستوردة.

فرض قيم الذاكرة المخبأة

تفرض بعض المشاريع القديمة خيارات التبعيات هكذا:

```
set(BUILD_TESTING OFF CACHE BOOL "" FORCE)
```

قد يلغي ذلك اختيار المشروع الأب. فضّل أسماء خيارات خاصة بالتبعية، أو إعدادًا محدود النطاق، أو تكامل مدير الحزم. وإن تطلب مشروع خارجي فرض خيار كي يمكن تضمينه بثبات، فاعزل الاستثناء ووثقه بوضوح.

معاملة CMAKE_BUILD_TYPE على أنه شامل لكل المولّدات

استخدم تعبيرات المولّد للسلوك المعتمد على الإعداد:

```
target_compile_definitions(core PRIVATE
  $<$<CONFIG:Debug>:ORBIT_DEBUG_BUILD>
)
```

فحص وقت الإعداد مثل:

```
if(CMAKE_BUILD_TYPE STREQUAL "Debug")
```

لا يعبّر عن المولّدات متعددة الإعدادات، لأن Debug وRelease قد يوجدان معًا داخل شجرة البناء نفسها.

ضبط متغيرات المصرّف بعد project()

يجب اختيار المصرّف قبل تفعيل اللغة، ويفضل أن يتم ذلك من البيئة أو ملف سلسلة أدوات أو أمر الإعداد الأول. تغيير CMAKE_CXX_COMPILER لاحقًا ليس وسيلة مدعومة لتبديل المصرّف داخل شجرة بناء موجودة.

تسلسل الانتقال

حدّث المشروع تدريجيًا:

١. أعلن حدًا أدنى صريحًا ومدعومًا لإصدار CMake.

٢. انقل جميع عمليات البناء إلى خارج شجرة المصدر.

٣. استبدل مجلدات التضمين العامة بمتطلبات تضمين مرتبطة بالأهداف.

٤. استبدل خيارات المصرّف الخام بأوامر الأهداف وخصائصها.

٥. استبدل مسارات التبعيات الخام بالأهداف المستوردة.

٦. أضف الأسماء البديلة ومساحات الأسماء.

٧. افصل المتطلبات العامة عن الخاصة.

٨. أضف إعدادات مسبقة لسير العمل المدعوم.

٩. صحح قواعد التثبيت وتصدير الحزم.

١٠. اختبر الحزمة المثبتة بواسطة مشروع مستهلك مستقل.

١١. أضف مصفوفات تكامل مستمر نظيفة، ولا تحذف شيفرة التوافق إلا عندما تدعم بيانات الاستخدام ذلك.

ينبغي لعملية التحديث أن تقلل الحالة الضمنية. أما إعادة الكتابة التي تخفي كل المنطق خلف دوال مخصصة ضخمة فقد تستبدل تعقيدًا قديمًا بتعقيد جديد فحسب.

ما المهم في CMake 4.0 إلى 4.4؟

CMake 4.0: حد فاصل في التوافق

يمثل CMake 4.0 إصدارًا رئيسيًا أزال بعض مسارات التوافق المهملة منذ زمن، وقدم سياسات ودعم منصات أحدث. لا ينبغي للمشروع رفع الحد الأدنى لمجرد إظهار رقم جديد؛ بل عليه مراجعة ملاحظات الإصدار والسياسات، واختبار المولّدات المدعومة، وتحديد السلوك الحديث الذي يعتمد عليه فعلًا.

يوصل نطاق السياسات معنى المراجعة من دون التخلي عن التثبيتات الأقدم:

```
cmake_minimum_required(VERSION 3.25...4.4)
```

يبقى المشروع قابلًا للتشغيل على 3.25، بينما يطلب سلوك السياسات الذي اختُبر حتى 4.4 عندما يعالجه إصدار أحدث من CMake.

أبرز إضافات CMake 4.2

أضاف CMake 4.2 ميزات مهمة للمطورين ومؤلفي الأدوات، منها:

- مولّد Visual Studio 18 2026.
- مولّد FASTBuild.
- تبسيط دعم البناء المتقاطع لـ Emscripten.
- الأمران `cmake -E copy_if_newer` و`copy_directory_if_newer`.
- `cmake_language(TRACE)`.
- الصيغة الصريحة `set(CACHE{...})` و`unset(CACHE{...})`.
- `string(REGEX QUOTE)`.
- توسيع معلومات File API للأهداف المستوردة وأهداف الواجهة.

الدرس العملي ليس إعادة كتابة المشاريع العادية حول كل أمر جديد. استخدم الميزة حين تحسن الصحة والوضوح: عمليات ذاكرة مخبأة صريحة في شيفرة الأطر، أو النسخ المعتمد على الطابع الزمني في عمليات التوليد الكبيرة، أو المولّد الجديد بعد أن تعتمد المؤسسة أداته.

أبرز إضافات CMake 4.3

قدم CMake 4.3 إمكانات استراتيجية عدة:

المواصفة المشتركة للحزم (CPS)

تستطيع CMake استيراد أوصاف حزم وتصديرها بصيغة مستقلة عن نظام البناء، وهو اتجاه مهم لتحسين التشغيل البيني.

القياس التشغيلي Instrumentation

يوفر دليل القياس التشغيلي وأمره الجديد بيانات منظمة تمتد عبر الإعداد والبناء والاختبار والتثبيت.

مخطط الإعدادات المسبقة 11

تستمر قدرات الإعدادات المسبقة في التوسع مع الحفاظ على توافق يحدده رقم المخطط.

إخراج الإصدار بصيغة JSON

يجعل `cmake --version=json-v1` اكتشاف الأداة أكثر موثوقية من تحليل نص موجّه للبشر.

تحسين أدوات سطر الأوامر

تعزز إمكانات الأرشفة والتحقق من المجاميع و`bin2c` دور CMake كأداة أتمتة محمولة.

تحسين مجموعات الملفات والتحقق من الترويسات

يساعد التحقق من الترويسات الخاصة ومعلومات File API الأغنى المكتبات الكبيرة على الحفاظ على واجهات أنظف.

أبرز إضافات CMake 4.4

أضاف CMake 4.4 ميزات تؤثر مباشرة في سير العمل الحديث:

- تشخيصًا منظمًا والأمر `cmake_diagnostic()`.
- مخطط الإعدادات المسبقة 12.
- الخيار `--presets-file` لعمليات `cmake` و`ctest` و`cpack`.
- تمرير وسائط اختبار إضافية من إعدادات الاختبار المسبقة.
- التقاط مخرجات القياس التشغيلي والتكامل مع تتبع المصرّف.
- أهدافًا لتحضير الاختبارات مع Ninja.
- دعم وحدات C++20 المسماة مع تركيبات إضافية من Clang وWindows.
- تثبيت مكونات متعددة بأمر واحد.
- `discover_tests()` لاكتشاف الاختبارات المخصص.
- التحكم في البيئة داخل `execute_process()`.
- مسندات ومقارنات وتحويلات قوائم يعرّفها المستخدم.
- مجموعات ملفات من نوع `SOURCES` وخصائص أغنى لمجموعات الملفات.
- أنماط توافق إضافية لإصدارات الحزم، ومنها التعامل مع الإصدارات الدلالية.

هذه إضافات قوية، لكن الجوهر الثابت يظل: تصميم يعتمد على الأهداف، ومتطلبات استخدام صحيحة، ومجلدا مصدر وبناء صريحان، وصادرات قابلة للتثبيت، وإعدادات مسبقة قابلة لإعادة الإنتاج.

اختيار الحد الأدنى في 2026

الحد الأدنى المناسب قرار مرتبط بالمنتج:

- اختر CMake 3.25 عندما تريد إعدادات الحزم وسير العمل المسبقة وميزات الأهداف الحديثة القوية مع توفر واسع.
- اختر 3.28 أو أحدث عندما يعتمد المشروع على قدرات أحدث للوحدات أو سلاسل الأدوات.
- اختر CMake 4.x فقط عندما يستخدم المشروع فعلًا سلوكًا من x.4، أو عندما تستطيع بيئة النشر ضمانه.

ينبغي للمكتبات الموزعة على أنظمة Linux أن تكون متحفظة في الحد الأدنى. أما التطبيقات الداخلية ذات سلسلة الأدوات المضبوطة فيمكنها التقدم أسرع.

وثق فكرتين مختلفتين:

١. **الإصدار الأدنى المطلوب:** أقدم إصدار يدعمه وصف المشروع وقت التشغيل.

٢. **إصدار المراجعة التوثيقية:** الإصدار الحالي الذي روجعت التوصيات وملاحظات الميزات الجديدة في ضوئه.

تستهدف هذه الطبعة وثائق CMake 4.4.1 الرسمية، مع إبقاء معظم الأمثلة متوافقة مع CMake 3.25.

اختبر تغييرات السياسات لا الصياغة وحدها

قد ينجح إعداد المشروع بإصدار CMake جديد، لكنه يتصرف بصورة مختلفة لأن سياسة ما غيّرت تفسير الربط أو المسارات المولّدة أو معالجة الأرشيف أو سلوك الحزم.

استخدم التكامل المستمر لاختبار الإصدار الأدنى المدعوم وإصدار حالي. تعامل مع تحذيرات المطورين على أنها معلومات للانتقال، لا رسائل يجب إسكاتها آليًا.

اعزل الميزات الجديدة

عندما تتطلب ميزة الإصدار 4.4، اجعل الشرط محليًا:

```
if(CMAKE_VERSION VERSION_GREATER_EQUAL 4.4)
  # Optional 4.4 enhancement
endif()
```

لا تنشئ رسمَي أهداف مختلفين إلا عند الضرورة. الأفضل الحفاظ على خط أساس محمول، ثم إضافة تحسين أو تشخيص محلي، بدل صيانة نظامَي بناء منفصلين داخل الملف نفسه.

هيكل مشروع بأسلوب إنتاجي

الهدف

يجمع المشروع الختامي الممارسات المركزية في الكتاب:

- حدًا أدنى صريحًا ونطاق سياسات محددًا.
- تعريفات للمكتبات والملفات التنفيذية تتمحور حول الأهداف.
- مجموعات ملفات للترويسات العامة.
- أدوات واختبارات اختيارية.
- تحذيرات خاصة تراعي نوع المصرّف.
- قواعد تثبيت وتصدير حزمة.
- إعدادات مسبقة للتطوير والإصدار.
- مساحة أسماء مستقرة للمستهلكين.

هيكل المشروع:

```
orbit/
|-- CMakeLists.txt
|-- CMakePresets.json
|-- cmake/
|   `-- OrbitConfig.cmake.in
|-- include/
|   `-- orbit/
|       `-- orbit.hpp
|-- src/
|   `-- orbit/
|       |-- CMakeLists.txt
|       `-- orbit.cpp
|-- app/
|   |-- CMakeLists.txt
|   `-- main.cpp
`-- tests/
    |-- CMakeLists.txt
    `-- orbit_test.cpp
```

المشروع الأعلى

```
cmake_minimum_required(VERSION 3.25...4.4)
project(
  Orbit
  VERSION 3.0.0
  DESCRIPTION "A production-style CMake skeleton"
  LANGUAGES CXX
)

include(CTest)
include(GNUInstallDirs)
include(CMakePackageConfigHelpers)

option(ORBIT_BUILD_TOOLS "Build command-line tools" ON)
option(ORBIT_ENABLE_WARNINGS "Enable strict warnings for Orbit targets" ON)

add_subdirectory(src/orbit)

if(ORBIT_BUILD_TOOLS)
  add_subdirectory(app)
endif()

if(BUILD_TESTING)
  add_subdirectory(tests)
endif()

install(
  EXPORT OrbitTargets
  NAMESPACE Orbit::
  DESTINATION "${CMAKE_INSTALL_LIBDIR}/cmake/Orbit"
)

configure_package_config_file(
  cmake/OrbitConfig.cmake.in
  "${CMAKE_CURRENT_BINARY_DIR}/OrbitConfig.cmake"
  INSTALL_DESTINATION "${CMAKE_INSTALL_LIBDIR}/cmake/Orbit"
)

write_basic_package_version_file(
  "${CMAKE_CURRENT_BINARY_DIR}/OrbitConfigVersion.cmake"
  VERSION "${PROJECT_VERSION}"
  COMPATIBILITY SameMajorVersion
)

install(
  FILES
    "${CMAKE_CURRENT_BINARY_DIR}/OrbitConfig.cmake"
    "${CMAKE_CURRENT_BINARY_DIR}/OrbitConfigVersion.cmake"
  DESTINATION "${CMAKE_INSTALL_LIBDIR}/cmake/Orbit"
)
```

هدف المكتبة

محتوى `src/orbit/CMakeLists.txt`:

```
add_library(orbit orbit.cpp)
add_library(Orbit::orbit ALIAS orbit)

target_compile_features(orbit PUBLIC cxx_std_20)

target_sources(orbit
  PUBLIC
    FILE_SET HEADERS
    BASE_DIRS "${PROJECT_SOURCE_DIR}/include"
    FILES "${PROJECT_SOURCE_DIR}/include/orbit/orbit.hpp"
)

if(ORBIT_ENABLE_WARNINGS)
  target_compile_options(orbit PRIVATE
    $<$<CXX_COMPILER_ID:GNU,Clang,AppleClang>:-Wall;-Wextra;-Wpedantic>
    $<$<CXX_COMPILER_ID:MSVC>:/W4;/permissive->
  )
endif()

set_target_properties(orbit PROPERTIES
  VERSION "${PROJECT_VERSION}"
  SOVERSION "${PROJECT_VERSION_MAJOR}"
)

install(
  TARGETS orbit
  EXPORT OrbitTargets
  FILE_SET HEADERS DESTINATION "${CMAKE_INSTALL_INCLUDEDIR}"
  ARCHIVE DESTINATION "${CMAKE_INSTALL_LIBDIR}"
  LIBRARY DESTINATION "${CMAKE_INSTALL_LIBDIR}"
  RUNTIME DESTINATION "${CMAKE_INSTALL_BINDIR}"
)
```

ملاحظة حول PROJECT_SOURCE_DIR: يملك هذا الهيكل المشروع الأعلى. إذا كان المقصود نسخ مجلد المكتبة أو تضمينه مستقلًا، فضع ترويساته العامة داخل المكوّن نفسه واستخدم CMAKE_CURRENT_SOURCE_DIR بدلًا منه.

هدف التطبيق

محتوى app/CMakeLists.txt:

```
add_executable(orbit_cli main.cpp)
target_link_libraries(orbit_cli PRIVATE Orbit::orbit)

install(
  TARGETS orbit_cli
  RUNTIME DESTINATION "${CMAKE_INSTALL_BINDIR}"
)
```

يرتبط التطبيق بالاسم البديل ذي مساحة الأسماء، ولا يكرر مجلدات تضمين المكتبة أو معيار C++ الخاص بها.

هدف الاختبار

محتوى tests/CMakeLists.txt:

```
add_executable(orbit_test orbit_test.cpp)
target_link_libraries(orbit_test PRIVATE Orbit::orbit)
add_test(NAME orbit.name COMMAND orbit_test)
```

يستهلك الاختبار المكتبة بالطريقة نفسها التي يستهلكها بها أي تطبيق.

قالب إعداد الحزمة

محتوى cmake/OrbitConfig.cmake.in:

```
@PACKAGE_INIT@
include("${CMAKE_CURRENT_LIST_DIR}/OrbitTargets.cmake")
check_required_components(Orbit)
```

إن اكتسبت Orbit تبعيات عامة من حزم أخرى، فأضف CMakeFindDependencyMacro واستدعاءات find_dependency() هنا.

الإعدادات المسبقة

```
{
  "version": 6,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 25,
    "patch": 0
  },
  "configurePresets": [
    {
      "name": "ninja-base",
      "hidden": true,
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/out/build/${presetName}",
      "installDir": "${sourceDir}/out/install/${presetName}",
      "cacheVariables": {
        "CMAKE_EXPORT_COMPILE_COMMANDS": true
      }
    },
    {
      "name": "dev",
      "inherits": "ninja-base",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Debug",
        "BUILD_TESTING": true
      }
    },
    {
      "name": "release",
      "inherits": "ninja-base",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Release",
        "BUILD_TESTING": false
      }
    }
  ],
  "buildPresets": [
    {
      "name": "dev",
      "configurePreset": "dev",
      "jobs": 8
    },
    {
      "name": "release",
      "configurePreset": "release",
      "jobs": 8
    }
  ],
  "testPresets": [
    {
      "name": "dev",
      "configurePreset": "dev",
      "output": {
        "outputOnFailure": true
      }
    }
  ]
}
```

الأوامر اليومية

```
cmake --preset dev
cmake --build --preset dev
ctest --preset dev

cmake --preset release
cmake --build --preset release
cmake --install out/build/release
```

أسئلة تحقق لمشروعك

قبل اعتبار تصميم CMake مكتملًا، اسأل:

- هل يمكن بناء شجرة المصدر من مجلد جديد وفارغ؟

- هل يمكن أن يتعايش Debug وRelease من دون حذف أحدهما الآخر؟
- هل يعلن كل هدف تبعياته المباشرة؟
- هل تنتقل المتطلبات العامة ويبقى الخاص منها محليًا؟
- هل يمكن إضافة المشروع كمجلد فرعي من دون تغيير المشروع الأب عالميًا؟
- هل يعمل التثبيت تحت بادئة غير افتراضية؟
- هل يستطيع مشروع مستهلك منفصل استخدام الحزمة المثبتة؟
- هل يستطيع التكامل المستمر استدعاء الإعدادات المسبقة نفسها التي يستخدمها المطورون؟
- هل تبعيات توليد الملفات صريحة؟
- هل تُختبر إصدارات CMake الحالية والدنيا المدعومة؟

إذا كانت الإجابات نعم، فقد تجاوز المشروع مرحلة مجموعة الأوامر وأصبح واجهة بناء موثوقة.

مرجع سريع للأوامر

الإعداد

```
cmake -S . -B build
cmake -S . -B build -G Ninja
cmake -S . -B build --fresh
cmake -S . -B build -DNAME=value
cmake -S . -B build --toolchain toolchains/target.cmake
cmake --preset dev
```

البناء

```
cmake --build build
cmake --build build --parallel 8
cmake --build build --target app
cmake --build build --config Release
cmake --build --preset dev
```

الاختبار

```
ctest --test-dir build --output-on-failure
ctest --test-dir build -C Release
ctest --test-dir build -R parser
ctest --test-dir build -L unit
ctest --test-dir build --parallel 8
ctest --preset dev
```

التثبيت

```
cmake --install build
cmake --install build --prefix stage
cmake --install build --config Release
cmake --install build --component Runtime
```

التحزيم

```
cpack --config build/CPackConfig.cmake
cpack --preset release
```

اكتشاف المساعدة

```
cmake --help
cmake --help-command target_link_libraries
cmake --help-property INTERFACE_INCLUDE_DIRECTORIES
cmake --help-variable CMAKE_BUILD_TYPE
cmake --help-module CMakePackageConfigHelpers
cmake --help-manual cmake-buildsystem
```

أوامر المشروع الأساسية

- `cmake_minimum_required()` - يحدد الحد الأدنى للإصدار وسلوك السياسات.
- `project()` - يعرّف بيانات المشروع ولغاته.
- `add_executable()` - ينشئ هدفًا تنفيذيًا.
- `add_library()` - ينشئ هدف مكتبة.
- `add_subdirectory()` - يعالج مجلد مكوّن فرعي.
- `target_sources()` - يربط الملفات المصدرية ومجموعات الملفات بالهدف.
- `target_link_libraries()` - يربط التبعيات وينقل متطلبات الاستخدام.
- `target_include_directories()` - يضيف مسارات التضمين إلى الهدف.

- `target_compile_features()` - يطلب ميزات اللغة أو معيارها.
- `target_compile_definitions()` - يضيف تعريفات المعالج القبلي.
- `target_compile_options()` - يضيف خيارات خاصة بالمصرّف.
- `target_link_options()` - يضيف خيارات الرابط.
- `set_target_properties()` - يضبط عدة خصائص للهدف.
- `find_package()` - يكتشف تبعية مثبتة.
- `configure_file()` - يولد ملفًا في وقت الإعداد.
- `add_custom_command()` - يولد مخرجات في وقت البناء.
- `install()` - يعرّف قواعد التثبيت.
- `add_test()` - يسجل اختبارًا لدى CTest.

تعبيرات المولّد

تستخدم تعبيرات المولّد الصيغة $<...>، وتُقيّم في مرحلة التوليد أو لاحقًا ضمن سياق نظام البناء الأصلي. وهي أساسية للمتطلبات التي تعتمد على الإعداد أو المصرّف أو المنصة أو الهدف.

الإعداد

```
$<$<CONFIG:Debug>:ORBIT_DEBUG>
$<$<CONFIG:Release>:-O3>
$<CONFIG>
```

تعامل بحذر مع فواصل القوائم داخل التعبيرات. يقبل أمر الهدف قائمة CMake، وغالبًا تمثل الفواصل المنقوطة عدة خيارات:

```
target_compile_options(app PRIVATE
  $<$<CXX_COMPILER_ID:MSVC>:/W4;/permissive->
)
```

ضع التعبيرات المركبة بين علامتي اقتباس عندما يمكن للمسافات أو سلوك القوائم أن يقسمها بصورة غير مقصودة.

المصرّف والمنصة

```
$<CXX_COMPILER_ID:GNU,Clang,AppleClang>
$<PLATFORM_ID:Windows>
$<CXX_COMPILER_VERSION:14.2>
```

ادمج الشروط:

```
$<$<AND:$<CONFIG:Debug>,$<CXX_COMPILER_ID:GNU,Clang>>:-fno-omit-frame-pointer>
```

ملفات الأهداف

```
$<TARGET_FILE:orbit_cli>
$<TARGET_FILE_DIR:orbit_cli>
$<TARGET_LINKER_FILE:orbit>
$<TARGET_PROPERTY:orbit,INTERFACE_INCLUDE_DIRECTORIES>
```

استخدم تعبيرات ملفات الأهداف داخل الأوامر المخصصة بدل تركيب أسماء ملفات المخرجات يدويًا.

واجهتا البناء والتثبيت

```
target_include_directories(mylib PUBLIC
  $<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
  $<INSTALL_INTERFACE:include>
)
```

يكون تعبير البناء فعالًا عندما يستهلك هدف آخر المكتبة من شجرة البناء. أما تعبير التثبيت فيصبح جزءًا من الواجهة المثبتة المصدّرة.

الاقتباس والقوائم

يجمع التعبير التالي قائمة لتصبح وسيطة أمر:

```
"$<JOIN:$<TARGET_PROPERTY:app,INCLUDE_DIRECTORIES>,;-I>"
```

يمكن تعشيق تعبيرات المولّد، وهي شديدة القوة، لكن الإفراط في التعشيق يجعلها عسيرة القراءة. انقل المنطق المتكرر إلى هدف واجهة صغير أو دالة مساعدة واضحة الاسم عندما يزيد ذلك الفهم.

حدود التقييم

لا تدعم كل وسيطة في كل أمر جميع تعبيرات المولّد، كما تُقيّم بعض الخصائص ضمن سياق الهدف المستهلك. راجع توثيق الأمر أو الخاصية المحددة. لا يستطيع `message()` في وقت الإعداد إظهار قيمة لا توجد قبل مرحلة التوليد.

قوالب الإعدادات المسبقة

قالب Ninja محمول

```
{
  "version": 6,
  "cmakeMinimumRequired": {
    "major": 3,
    "minor": 25,
    "patch": 0
  },
  "configurePresets": [
    {
      "name": "base",
      "hidden": true,
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/out/build/${presetName}",
      "installDir": "${sourceDir}/out/install/${presetName}"
    },
    {
      "name": "debug",
      "inherits": "base",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Debug",
        "BUILD_TESTING": true
      }
    },
    {
      "name": "release",
      "inherits": "base",
      "cacheVariables": {
        "CMAKE_BUILD_TYPE": "Release",
        "BUILD_TESTING": false
      }
    }
  ],
  "buildPresets": [
    { "name": "debug", "configurePreset": "debug" },
    { "name": "release", "configurePreset": "release" }
  ],
  "testPresets": [
    {
      "name": "debug",
      "configurePreset": "debug",
      "output": { "outputOnFailure": true }
    }
  ]
}
```

قالب متعدد الإعدادات

```
{
  "version": 6,
  "configurePresets": [
    {
      "name": "multi",
      "generator": "Ninja Multi-Config",
      "binaryDir": "${sourceDir}/out/build/multi"
    }
  ],
  "buildPresets": [
    {
      "name": "multi-debug",
      "configurePreset": "multi",
      "configuration": "Debug"
    },
    {
      "name": "multi-release",
      "configurePreset": "multi",
      "configuration": "Release"
    }
  ],
  "testPresets": [
    {
      "name": "multi-debug",
      "configurePreset": "multi",
      "configuration": "Debug",
      "output": { "outputOnFailure": true }
    }
  ]
}
```

قالب سلسلة أدوات

```
{
  "name": "aarch64-release",
  "inherits": "release",
  "toolchainFile": "${sourceDir}/toolchains/linux-aarch64.cmake"
}
```

قالب المخطط 12 في CMake 4.4

```
{
  "version": 12,
  "cmakeMinimumRequired": {
    "major": 4,
    "minor": 4,
    "patch": 0
  },
  "configurePresets": [
    {
      "name": "dev",
      "generator": "Ninja",
      "binaryDir": "${sourceDir}/out/build/dev"
    }
  ],
  "testPresets": [
    {
      "name": "dev",
      "configurePreset": "dev",
      "execution": {
        "testPassthroughArguments": ["--color", "always"]
      }
    }
  ]
}
```

لا تستخدم المخطط 12 إلا عندما يحتاج سير العمل إلى ميزاته ويستطيع جميع المشاركين تشغيل CMake 4.4 أو أحدث.

قائمة فحص لاستكشاف الأخطاء

فشل الإعداد

- تحقق من `cmake --version` ومن الملف التنفيذي المختار.
- تحقق من أن مجلد المصدر يحوي ملف `CMakeLists.txt` الأعلى المتوقع.
- استخدم مجلد بناء جديدًا.
- أكد مسارات المصرّفات قبل الإعداد الأول.
- افحص `CMakeFiles/CMakeConfigureLog.yaml`.
- شغل `--debug-find-pkg=<name>` لتشخيص اكتشاف الحزم.
- تحقق مما إذا كان إعداد مسبق يختار سلسلة أدوات أو بيئة على نحو غير متوقع.

مصدر مفقود أو رسالة "no rule to make target"

- تحقق من مسار الملف الدقيق ومن حالة الأحرف.
- ابحث عن مسافات خفية داخل أسماء الملفات ووسائط CMake.
- لا تنقل شجرة مصدر أو بناء جرى إعدادها سابقًا.
- احذف الحالة المولّدة القديمة أو حدّثها.
- تحقق من أن CMake والمصرّف وأداة البناء تستخدم صيغ مسارات متوافقة في Windows وCygwin وMSYS.
- ابنِ باستخدام `--verbose` وافحص أول مُدخل مفقود.

فشل المصرّف

- افحص `compile_commands.json` أو المخرجات التفصيلية.
- تحقق من إرفاق معيار C++ المطلوب بالهدف الصحيح.
- تحقق من مجلدات التضمين العامة ومسارات الترويسات المولّدة.
- تحقق من التعريفات الخاصة بكل إعداد.
- اختبر من دون Unity Build أو PCH لكشف مشكلات ترتيب التضمين.

فشل الرابط

- اربط التبعيات المباشرة للهدف.
- تحقق من أن الملفات المصدرية جزء من المكتبة المقصودة.
- افحص انتقال `PUBLIC` مقابل `PRIVATE`.
- تجنب خلط البنى أو مكتبات وقت التشغيل أو ملفات Debug وRelease.

- افحص مواقع الأهداف المستوردة.
- تحقق من رؤية الرموز وماكروهات التصدير في المكتبات المشتركة.

فشل الاختبار

- شغّل الملف التنفيذي الفاشل يدويًا.
- استخدم `ctest -VV -R <name>`.
- اضبط مجلد عمل صريحًا أو مرر مسارات بيانات الاختبار.
- افحص مسارات البحث عن المكتبات وقت التشغيل ومتغيرات البيئة.
- أضف مهلة زمنية.
- تحقق من إعداد البناء الصحيح باستخدام `-C`.

فشل التثبيت أو التحزيم

- اختبر تحت بادئة مرحلية.
- تحقق من انضمام جميع الترويسات العامة إلى مجموعات ملفات مثبتة.
- ابنِ مشروعًا مستهلكًا مستقلًا من شجرة التثبيت.
- ابحث داخل ملفات CMake المثبتة عن مسارات مطلقة لشجرتي المصدر أو البناء.
- تحقق من تبعيات وقت التشغيل والتراخيص.
- تذكر أن CPack يتبع قواعد التثبيت.

المراجع

تعطي هذه الطبعة الأولوية لوثائق CMake وKitware الرسمية.

١. **وثائق CMake 4.4.1**

/https://cmake.org/cmake/help/latest

٢. **دليل CMake التعليمي**

/https://cmake.org/cmake/help/latest/guide/tutorial

٣. **دليل سطر أوامر CMake**

https://cmake.org/cmake/help/latest/manual/cmake.1.html

٤. **دليل نظام البناء في CMake**

https://cmake.org/cmake/help/latest/manual/cmake-buildsystem.7.html

٥. **دليل لغة CMake**

https://cmake.org/cmake/help/latest/manual/cmake-language.7.html

٦. **دليل تعبيرات المولّد**

https://cmake.org/cmake/help/latest/manual/cmake-generator-expressions.7.html

٧. **دليل إعدادات CMake المسبقة**

https://cmake.org/cmake/help/latest/manual/cmake-presets.7.html

٨. **دليل حزم CMake**

https://cmake.org/cmake/help/latest/manual/cmake-packages.7.html

٩. **دليل سلاسل أدوات CMake**

https://cmake.org/cmake/help/latest/manual/cmake-toolchains.7.html

١٠. **دليل وحدات C++ في CMake**

https://cmake.org/cmake/help/latest/manual/cmake-cxxmodules.7.html

١١. **دليل القياس التشغيلي في CMake**

https://cmake.org/cmake/help/latest/manual/cmake-instrumentation.7.html

١٢. **دليل التشخيص في CMake**

https://cmake.org/cmake/help/latest/manual/cmake-diagnostics.7.html

١٣. **ملاحظات إصدار CMake 4.2**

https://cmake.org/cmake/help/latest/release/4.2.html

١٤. **ملاحظات إصدار CMake 4.3**

https://cmake.org/cmake/help/latest/release/4.3.html

١٥. **ملاحظات إصدار CMake 4.4**

https://cmake.org/cmake/help/latest/release/4.4.html

١٦. دليل **CTest**

https://cmake.org/cmake/help/latest/manual/ctest.1.html

١٧. دليل **CPack**

https://cmake.org/cmake/help/latest/manual/cpack.1.html

١٨. الوحدة **CMakePackageConfigHelpers**

https://cmake.org/cmake/help/latest/module/CMakePackageConfigHelpers.html

١٩. الوحدة **FetchContent**

https://cmake.org/cmake/help/latest/module/FetchContent.html

٢٠. الوحدة **ExternalProject**

https://cmake.org/cmake/help/latest/module/ExternalProject.html

رؤية ختامية

تصبح CMake قابلة للإدارة عندما يتوقف المشروع عن معاملتها كمكان لتخزين خيارات المصرّف، ويبدأ في معاملتها بوصفها نموذجًا لمكونات البرمجيات. تعبّر الأهداف عن البنية، وتعبر متطلبات الاستخدام عن الواجهات، وتعبّر الإعدادات المسبقة عن مسارات العمل المدعومة، وتعبّر صادرات التثبيت عن الطريقة التي تستهلك بها المشاريع الأخرى الناتج.

ليست أكثر ميزات CMake تقدمًا أمرًا جديدًا، بل وصف بناء يظل معناه واضحًا عندما يتغير المصرّف أو المولّد، وعندما يصبح المشروع تبعية، وعندما يضبطه مطور جديد داخل مجلد فارغ.

ابنِ الهدف. انشر الواجهة. واجعل شجرة البناء قابلة للحذف دائمًا.