Embracing Modern CMake
How to recognize and use modern CMake interfaces

Stephen Kelly

Dublin C++ Meetup

September 11, 2017
Background

Qt

[Images of various software logos]
CMake - What, Why, Who

- Buildsystem Generator
- 'Cross-platform Make'
- Part of suite of productivity and quality tools
- Started by Kitware in 2000
CMake - What, Why, Who
Makefiles
Visual Studio/Xcode Project
CMake

Flexibility
Extensibility
Portability
Abstraction
Maintainability
Where CMake shines

- Finding dependencies
- Portability
- Code generation
- Multi-language support
What is Modern CMake?

• New(er) APIs and mindset of writing CMake code
• Less code
• Cleaner code
• More target-focused
## The Good News

**Mostly everything available to you already**

<table>
<thead>
<tr>
<th></th>
<th>2012</th>
<th>2013</th>
<th>2014</th>
<th>2015</th>
<th>2016</th>
<th>2017</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>October</td>
<td>May</td>
<td>October</td>
<td>June</td>
<td>December</td>
<td>March</td>
</tr>
<tr>
<td>CMake</td>
<td>&lt;= 2.8.10</td>
<td>2.8.11</td>
<td>2.8.12</td>
<td>3.0</td>
<td>3.1</td>
<td>3.2</td>
</tr>
<tr>
<td>Ubuntu</td>
<td>12.04</td>
<td>12.04</td>
<td></td>
<td></td>
<td></td>
<td>16.04</td>
</tr>
<tr>
<td>Debian</td>
<td></td>
<td></td>
<td>Wheezy</td>
<td></td>
<td>Jessie</td>
<td></td>
</tr>
<tr>
<td>Fedora</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Travis</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RHEL</td>
<td>5</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Defining a Buildsystem

```cmake
cmake_minimum_required(VERSION 3.5)
project(myproject)

add_library(libsalutation STATIC salutation.cpp)

add_executable(hello hello.cpp)
target_link_libraries(hello libsalutation)

add_executable(goodbye goodbye.cpp)
target_link_libraries(goodbye libsalutation)
```
Policies
Policies

```cmake
1 cmake_minimum_required(VERSION 2.8)
```
Policies

1. `cmake_minimum_required(VERSION 2.8)`

- Fail at runtime if version is too low
- Populate variable `CMAKE_MINIMUM_REQUIRED_VERSION`
- (Re)set runtime behavior of CMake with policies
- Should be first line of your CMake buildsystem (before project)
Policies

- Behavior deprecation mechanism
- In WARN state by default
- Set individually for fine control

<table>
<thead>
<tr>
<th>CMake 3.3</th>
<th>CMP0057</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CMP0056</td>
</tr>
<tr>
<td>CMake 3.2</td>
<td>CMP0055</td>
</tr>
<tr>
<td>CMake 3.1</td>
<td>CMP0054</td>
</tr>
</tbody>
</table>

...
Policies

```cpp
1 cmake_minimum_required(VERSION 3.0)
2 if (POLICY CMP0053)
3    cmake_policy(SET CMP0053 NEW)
4 endif()
```

```cpp
1 cmake_policy(SET CMP0053 OLD)
```
When to Set a Policy to OLD

\texttt{std::allOf(}

- You are close to your own release
- A new release of CMake just happened
- A policy warning is triggered for your code
- Your code relies on the OLD behavior
  - Or you suspect it might
- You don’t pass the setting to dependents

\texttt{)}

Create a plan to migrate to the NEW behavior!
Policy warning is slower

Cost

OLD

NEW

WARN
UseCase: Require new features (higher version) but rely on OLD behavior

Not Modern CMake

```cmake
1 cmake_minimum_required(VERSION 3.3)
2 cmake_policy(SET CMP0003 OLD)
3 target_include_directories(…)
```
UseCase: Allow old CMake but use NEW behavior where possible

```cmake
 cmake_minimum_required(VERSION 3.0)
 foreach (pol CMP0053
   CMP0063
   CMP0065
 }
   if (POLICY ${pol})
     cmake_policy(SET ${pol} NEW)
   endif()
 endforeach()
```
Some Policies do not issue warnings

as of 3.8

CMP0025 Compiler id for Apple Clang is now AppleClang
CMP0047 Use QCC compiler id for the qcc drivers on QNX
CMP0056 Honor link flags in try_compile()
CMP0060 Link libraries by full path even in implicit dirs
CMP0061 CTest does not by default tell make to ignore errors
CMP0065 ENABLE_EXPSORTS target property flags
CMP0066 Honor per-config flags in try_compile()
CMP0067 Honor language standard in try_compile()

- Use -DCMAKE_POLICY_WARNING_CMP<NNNN>=ON to enable it
Modern CMake Guidelines

- Maintain up-to-date policy settings
Usage Requirements
Defining a Buildsystem

```cmake
cmake_minimum_required(VERSION 3.5)
project(myproject)

add_library(libsalutation STATIC salutation.cpp)

add_executable(hello hello.cpp)
target_link_libraries(hello libsalutation)

add_executable(goodbye goodbye.cpp)
target_link_libraries(goodbye libsalutation)
```
Defining a Buildsystem

```cmake
cmake_minimum_required (VERSION 3.5)
project (myproject)

add_subdirectory (libraries)

add_subdirectory (executables)
```
Defining a Buildsystem

Not Modern CMake

```
include_directories(${salutation_INCLUDES})

add_executable(hello hello.cpp)
target_link_libraries(hello libsalutation)

add_executable(goodbye goodbye.cpp)
target_link_libraries(goodbye libsalutation)
```
Defining a Buildsystem

```plaintext
add_executable(hello hello.cpp)
target_link_libraries(hello libsalutation)
target_include_directories(hello PRIVATE ${salutation_INCLUDES})

add_executable(goodbye goodbye.cpp)
target_link_libraries(goodbye libsalutation)
target_include_directories(goodbye PRIVATE ${salutation_INCLUDES})
```
2.8.11

```cmake
add_executable(hello hello.cpp)
target_link_libraries(hello
    libsalutation
)

add_executable(goodbye goodbye.cpp)
target_link_libraries(goodbye
    libsalutation
)
```
Build properties

- Target-based buildsystem definition
- Single point of dependency specification
- Targets provide information to dependers
  - Requirements to compile
  - Requirements to link
Defining a Buildsystem

Modern CMake

2.8.11

```cpp
add_library(salutation salutation.cpp)
target_include_directories(salutation
  PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include
)
```
Transitive compile dependency

```cpp
target_include_directories(<target>
    <PUBLIC|PRIVATE|INTERFACE>
    [items...] )
```
Transitive compile dependency

```cpp
#include <FooDependency>

class MyClass : public FooDependency
{
    MyClass();
};

#include <BarDependency>

MyClass::MyClass()
{
    doInitialize(BarDependency{});
}
```
Transitive compile dependency

Foo.h → MyLib → Exe
Bar.h → MyLib

Diagram shows the transitive compile dependency with Foo.h and Bar.h both depending on MyLib, which in turn depends on Exe.
Transitive compile dependency

<table>
<thead>
<tr>
<th>PRIVATE</th>
<th>Needed by me, but <strong>not</strong> my dependers</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUBLIC</td>
<td>Needed by me <strong>and</strong> my dependers</td>
</tr>
<tr>
<td>INTERFACE</td>
<td>Needed <strong>not</strong> by me, <strong>but</strong> by my dependers</td>
</tr>
</tbody>
</table>
Defining a Buildsystem

```
add_library(salutation salutation.cpp)
target_include_directories(salutation
  PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include
)
```
Transitive Scope

2.8.11

code

```cpp
target_include_directories(myTarget
    PUBLIC  "/something/public"
    PRIVATE  "/something/private"
    INTERFACE  "/something/interface"
    PUBLIC  "/another/public"
    PRIVATE  "/another/private"
)
```
include_directories command

```
include_directories(some_dir)
add_library(libA)
add_library(libB)
```
include_directories command

Not Modern CMake

```cmake
add_library(libA)
include_directories(some_dir)
add_library(libB)
```
include_directories command

Not Modern CMake

```
1
2
3
4
5
add_library(libA)
add_library(libB)
include_directories(some_dir)
```
include_directories command

```
add_library(libA)
add_subdirectory(dir1)
include_directories(some_dir)
add_library(libB)
add_subdirectory(dir2)
```
include_directories command

```cmake
add_library(libA)
subdirs(dir1)
include_directories(some_dir)
add_library(libB)
add_subdirectory(dir2)
```
Lack of transitivity

```
set (app_INCLUDES
    ${lib1_INCLUDES}
    ${lib2_INCLUDES}
    ${lib3_INCLUDES})

set (app_LIBRARIES ...)

include_directories (${app_INCLUDES})

add_subdirectory (dir1)
add_subdirectory (dir2)
add_subdirectory (dir3)
add_subdirectory (app)
```
add_definitions command has all the same problems as include_directories command.

```cpp
add_library(salutation salutation.cpp)
add_definitions(
  -DUSE_INTERNAL_SIMD
  -DUSE_MULTITHREADING)
```
2.8.11

```cmake
add_library(salutation salutation.cpp)
target_include_directories(salutation
    PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include
)
target_compile_definitions(salutation
    PRIVATE USE_INTERNAL_SIMD
    PUBLIC USE_MULTITHREADING
)
```
## Build properties

Support `<PRIVATE|PUBLIC|INTERFACE>` and transitivity

<table>
<thead>
<tr>
<th>Include Directories (-I/foo/bar)</th>
<th>target_include_directories</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compile Definitions (-DSOMEDEF)</td>
<td>target_compile_definitions</td>
</tr>
<tr>
<td>Compile Options (-fPIC)</td>
<td>target_compile_options</td>
</tr>
<tr>
<td>Link Libraries (-l/path/to/lib)</td>
<td>target_link_libraries</td>
</tr>
<tr>
<td>Sources</td>
<td>target_sources</td>
</tr>
</tbody>
</table>
Modern CMake Guidelines

- Maintain up-to-date policy settings
- Write target-centric code
  - Use `target_` command variants
  - Specify usage requirements for targets
Lack of transitivity

set(app_INCLUDES
    ${lib1_INCLUDES}
    ${lib2_INCLUDES}
    ${lib3_INCLUDES})

set(app_LIBRARIES ...)  

include_directories(${app_INCLUDES})

add_subdirectory(dir1)
add_subdirectory(dir2)
add_subdirectory(dir3)

add_subdirectory(app)
Reliance on variables

Not Modern CMake

```cmake
set(main_SRCS
  main.cpp)

add_executable(app ${main_SRCS})
target_include_directories(app
  PRIVATE ${app_INCLUDES})
target_compile_definitions(app
  PRIVATE ${app_DEFINES})
target_link_libraries(app
  ${app_LIBRARIES})
```
Reliance on variables

Not Modern CMake

```cmake
set(main_SRCS
   main.cpp)

add_executable(app ${main_SRCS})
target_include_directories(app
   PRIVATE)
target_compile_definitions(app
   PRIVATE)
target_link_libraries(app
   )
```
Reliance on variables

Eschew obfuscation; Espouse elucidation

Modern CMake

```cpp
add_executable(app main.cpp)
target_link_libraries(app
    lib2 lib3
)
```
Problems with Variables

- Variables are fragile
- Variables leak to other contexts
- Variables don’t express scope of dependencies
- Variables are not checked for correctness or content
Modern CMake Guidelines

- Maintain up-to-date policy settings
- Write target-centric code
  - Use `target_<command>` variants
  - Specify usage requirements for targets
- Avoid unnecessary variables
Generator Expressions
Conditions

```cmake
set(main_SRCS
    main.cpp
)
if (WIN32)
    list(APPEND main_SRCS helper_win.cpp)
else()
    list(APPEND main_SRCS helper_posix.cpp)
endif()
add_executable(hello ${main_SRCS})
```
add_executable(hello main.cpp)
if (WIN32)
    target_sources(hello PRIVATE
        helper_win.cpp
    )
else()
    target_sources(hello PRIVATE
        helper_posix.cpp
    )
endif()
3.1

```cmake
add_executable(hello main.cpp
  $<$<BOOL:${WIN32}>:helper_win.cpp>
  $<$<NOT:$<BOOL:${WIN32}>>:helper_posix.cpp>
)
```
Warning: this code is buggy and non-portable!

```cmake
set (main_SRCS
    main.cpp
)
if (CMAKE_BUILD_TYPE STREQUAL DEBUG)
  list (APPEND main_SRCS helper_debug.cpp)
else()
  list (APPEND main_SRCS helper_rel.cpp)
endif()
add_executable (hello ${main_SRCS})
```
Conditions

3.1

```cpp
add_executable(hello
  main.cpp
  $<$CONFIG:Debug>:helper_debug.cpp>
  $<$NOT:$<CONFIG:Debug>>:helper_rel.cpp>
)
```
Generator Expressions

Configure
if() / else() / endif()

Compute
$<1:...>$

Generate
### Generator Expressions basics

<table>
<thead>
<tr>
<th>$&lt;1:...&gt;$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt;0:...&gt;$</td>
<td></td>
</tr>
<tr>
<td>$&lt;$<a href="">Config:Debug$</a></td>
<td>1 (in Debug config)</td>
</tr>
<tr>
<td>$&lt;$<a href="">Config:Debug$</a></td>
<td>0 (in Debug config)</td>
</tr>
<tr>
<td>$&lt;$$<a href="">Config:Debug</a>:....&gt;$</td>
<td>... (in Debug config)</td>
</tr>
<tr>
<td>$&lt;$$<a href="">Config:Debug</a>:....&gt;$</td>
<td>(in Debug config)</td>
</tr>
</tbody>
</table>
Truthiness conversion

\$<$<BOOL:${WIN32}>:...> \textit{at configure time produces} \$<$<BOOL:1>:...> \textit{or} \$<$<BOOL:>:...> \textit{at generate-time!}

```cpp
add_executable(hello
    main.cpp
    \$<$<BOOL:${WIN32}>:helper_win.cpp>
    \$<$<NOT:$<BOOL:${WIN32}>}:helper_posix.cpp>
)
```
Support for Generator Expressions

- `target_` commands
- `file(GENERATE)` command
- `add_executable/add_library` commands
- `install` command (partial)
- `add_custom_target` command (partial)
- `add_custom_command` command (partial)

There are others, but these are the most important
# Compile with USE_THREADS if the
# WITH_THREADS property is ON
get_property(buildWithThreads TARGET hello
  PROPERTY WITH_THREADS)
if (buildWithThreads)
  target_compile_definitions(hello PRIVATE
    USE_THREADS)
endif()

set_property(TARGET hello
  PROPERTY WITH_THREADS ON)
Modern CMake

2.8.11

```cpp
# Compile with USE_THREADS if the 
# WITH_THREADS property is ON

# target_compile_definitions

```

```cpp
target_compile_definitions(hello PRIVATE

  $<$<TARGET_PROPERTY:WITH_THREADS>:USE_THREADS>>)

```

```cpp
set_property(TARGET hello

  PROPERTY WITH_THREADS ON)

```
Conditions

- “Generator Expression” conditions at Generate-time
- Test content after configure-time
  - Configuration
  - TARGET_PROPERTY
  - TARGET_POLICY
  - COMPILE_FEATURES
  - LOCATION
- Never use CMAKE_BUILD_TYPE in if()
• Maintain up-to-date policy settings
• Write target-centric code
  • Use `target_` command variants
  • Specify usage requirements for targets
• Avoid unnecessary variables
• Use generate-time conditions correctly
target_link_libraries
Defining a Buildsystem

**Modern CMake** 2.8.11

```cpp
add_executable(hello hello.cpp)
target_link_libraries(hello
    libsalutation
)

add_executable(goodbye goodbye.cpp)
target_link_libraries(goodbye
    libsalutation
)
```
target_link_libraries

<item> can be:

- A CMake target
- A library name on disk
- A full library path
- A linker flag
target_link_libraries

1 \texttt{target\_link\_libraries}(\texttt{someTarget aTargetName})

- Link to \texttt{aTargetName}
- Determine build order
- Consume usage requirements
  - Compiling
  - Linking
- Determine compatibility
target_link_libraries

1 \texttt{target\_link\_libraries(someTarget oopsItsATypo)}

- Check if it is a CMake target name
- Check if it is a link flag (starts with ‘-’)
- Check if it is a path
- Assume it is a libraryname (add -loopsItsATypo)
CMake Target types

<table>
<thead>
<tr>
<th>Type</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executables</td>
<td>add_executable</td>
</tr>
<tr>
<td>Shared libraries</td>
<td>add_library(SHARED)</td>
</tr>
<tr>
<td>Static libraries</td>
<td>add_library(STATIC)</td>
</tr>
<tr>
<td>Object libraries</td>
<td>add_library(OBJECT)</td>
</tr>
<tr>
<td>Interface libraries</td>
<td>add_library(INTERFACE)</td>
</tr>
<tr>
<td>Alias libraries</td>
<td>add_library(ALIAS)</td>
</tr>
</tbody>
</table>
Interface targets

Suitable for header-only libraries

```cpp
1  add_library(boost_mpl INTERFACE)
2  target_compile_definitions(boost_mpl INTERFACE BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS)
3  target_include_directories(boost_mpl INTERFACE "3rdparty/boost/mpl")
4
5  add_executable(my_exe)
6  target_link_libraries(my_exe boost_mpl)
```
Interface targets

Suitable for header-only libraries

```cpp
add_library(boost_mpl INTERFACE)
target_compile_definitions(boost_mpl INTERFACE BOOST_MPL_CFG_NO_PREPROCESSED_HEADERS)
target_include_directories(boost_mpl INTERFACE "3rdparty/boost/mpl")

add_library(boost_icl INTERFACE)
target_link_libraries(boost_icl INTERFACE boost_mpl)
target_include_directories(boost_icl INTERFACE "3rdparty/boost/icl")
```

```cpp
add_executable(my_exe)
target_link_libraries(my_exe boost_icl)
```
Group build properties for convenient consumption

```cpp
target_link_libraries(windows_specific INTERFACE directX)

target_compile_definitions(windows_specific INTERFACE USE_DIRECTX)

target_sources(windows_specific INTERFACE network_win.cpp)

add_library(platform_specific INTERFACE)

target_link_libraries(platform_specific INTERFACE
  $<$<BOOL:${WIN32}>:windows_specific>
  $<$<NOT:$<BOOL:${WIN32}>:posix_specific>
)
```
Interface targets

```cmake
1 target_link_libraries(mytarget
2 platform_specific
3 helper_library
4 )
```
2.8.12

```cpp
add_library(detail::platform_specific
  ALIAS platform_specific)
)

target_link_libraries(mytarget
detail::platform_specific
)
```
Alias targets

```cmake
add_library(boost::mpl
    ALIAS boost_mpl)
)
```

```cmake
target_link_libraries(mytarget
    boost::mpl
)
```
Dependencies
External dependencies

3.1

```cmake
cmake_minimum_required(VERSION 3.5)
project(myproject)

find_package(Qt5Widgets REQUIRED)
find_package(Qt53D REQUIRED)

add_executable(hello main.cpp)
target_link_libraries(hello
  Qt5::Widgets Qt5::3DCore
)
```
External dependencies

```cmake
# 3.1

cmake_minimum_required(VERSION 3.5)
project(myproject)

find_package(Qt5Widgets REQUIRED)

add_library(locallib STATIC locallib.cpp)
target_link_libraries(locallib PUBLIC
  Qt5::Widgets
)
add_executable(hello main.cpp)
target_link_libraries(hello
  locallib
)
```
Legacy pattern

```cmake
# Not Modern CMake

cmake_minimum_required(VERSION 3.5)
project(myproject)

find_package(Qt5Core REQUIRED)
find_package(Qt5Gui REQUIRED)
find_package(Qt5Widgets REQUIRED)

add_definitions(
    
    ${Qt5Core_DEFINITIONS}
    ${Qt5Gui_DEFINITIONS}
    ${Qt5Widgets_DEFINITIONS}
)
```
Legacy pattern

```
include_directories(
    ${Qt5Core_INCLUDE_DIRS}
    ${Qt5Gui_INCLUDE_DIRS}
    ${Qt5Widgets_INCLUDE_DIRS}
)

set (CMAKE_CXX_FLAGS
    "${CMAKE_CXX_FLAGS} ${Qt5Core_CXX_FLAGS} \n
    ${Qt5Gui_CXX_FLAGS} ${Qt5Widgets_CXX_FLAGS}")
```
Legacy pattern

```cpp
add_library(locallib SHARED
    locallib.cpp
)

target_link_libraries(locallib
    ${Qt5Core_LIBRARIES} ${Qt5Gui_LIBRARIES}
    ${Qt5Widgets_LIBRARIES}
)

set(locallib_LIBRARIES locallib
    ${Qt5Core_LIBRARIES} ${Qt5Gui_LIBRARIES}
    ${Qt5Widgets_LIBRARIES}
)
```
Old style packages populate variables

```cpp
find_package(Foo REQUIRED)

add_executable(hello main.cpp)
target_include_directories(hello PRIVATE ${Foo_INCLUDE_DIRS})
target_compile_definitions(hello PRIVATE ${Foo_COMPILE_DEFINITIONS})
target_link_libraries(hello ${Foo_LIBRARIES})
```
Modern CMake packages define IMPORTED targets.

```cmake
find_package(Foo REQUIRED)

add_executable(hello main.cpp)
target_link_libraries(hello Foo::Core)
```

- Compare with syntax using ALIAS libraries.
- CMake code for `hello` doesn’t change under refactoring.
Modern CMake Guidelines

- Maintain up-to-date policy settings
- Write target-centric code
  - Use `target_` command variants
  - Specify usage requirements for targets
- Avoid unnecessary variables
- Use generate-time conditions correctly
- Use `IMPORTED` targets for external dependencies
Creating Packages

```cpp
add_library(mylib ...)

install(TARGETS mylib
    EXPORT exp
    ARCHIVE DESTINATION lib)

install(EXPORT exp
    NAMESPACE ns:: DESTINATION share/cmake)
install(FILES mylibConfig.cmake
    DESTINATION share/cmake)
```
Modern CMake Guidelines

• Maintain up-to-date policy settings
• Write target-centric code
  • Use `target_<command>` variants
  • Specify usage requirements for targets
• Avoid unnecessary variables
• Use generate-time conditions correctly
• Use `IMPORTED` targets for external dependencies
• Install 'rich' targets and export packages
Where to get more information

- Avoid the wiki
- Use the documentation
- Use the cmake mailing list
- Use stack overflow
Thanks & Questions

Special thanks:
• Brad King
• Ben & Robert
• Reddit <3
Compatible interfaces

```cpp
find_package(qjson REQUIRED)

add_executable(main main.cpp)

target_link_libraries(main
    Qt5::Widgets qjson::qjson)
```
Compatible interfaces

```
Qt4Core           QJson
               /    \
              /      \
Qt5Core     MyApp     Qt5Widgets
               |       |
              |       |
Qt5Gui
```
Compatible interfaces

```cpp
set_property(TARGET Qt4::QtCore PROPERTY INTERFACE_QT_MAJOR_VERSION 4)

set_property(TARGET Qt4::QtCore APPEND PROPERTY COMPATIBLE_INTERFACE_STRING QT_MAJOR_VERSION)

# ...

set_property(TARGET Qt5::Core PROPERTY INTERFACE_QT_MAJOR_VERSION 5)

set_property(TARGET Qt5::Core APPEND PROPERTY COMPATIBLE_INTERFACE_STRING QT_MAJOR_VERSION)
```
Creating build-dir Packages

```cmake
add_library(mylib ...)

install(TARGETS mylib
    EXPORT exp
    ARCHIVE DESTINATION lib)

export(EXPORT exp
    NAMESPACE ns::
    FILE buildDirTargets.cmake)
```

- Non-relocatable (use `install(EXPORT)` for that)
- Suitable in cross-compiles or superbuilds.