

## **MSBuild**

### **1. Основные настройки компилятора:**

Эти настройки включают в себя указание целевой версии .NET Framework, конфигурации (Debug/Release) и целевой платформы (x86/x64/AnyCPU).

Пример в файле .csproj:

```
<PropertyGroup>  
    <TargetFramework>net5.0</TargetFramework>  
    <Configuration>Release</Configuration>  
    <Platform>AnyCPU</Platform>  
</PropertyGroup>
```

### **2. Условные конструкции при настройке компилятора:**

Вы можете использовать условные операторы, такие как ``, ``, и ``, чтобы задавать разные параметры компиляции в зависимости от условий.

- Пример условной конструкции в файле .csproj:

```
<PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">  
    <DefineConstants>DEBUG;TRACE</DefineConstants>  
</PropertyGroup>
```

### **3. Настройка компилятора в файле csproj:**

Файл `.csproj` представляет собой XML-документ, который содержит множество различных элементов и параметров для настройки сборки и поведения проекта. Настройка компилятора в файле `.csproj` включает в себя определение различных параметров и настроек, которые влияют на процесс компиляции вашего проекта.

*Целевая версия .NET Framework:*

Вы можете указать, какую версию .NET Framework должен использовать ваш проект. Это определяется с помощью элемента `` внутри ``:

```
<PropertyGroup>
```

```
  <TargetFramework>net5.0</TargetFramework>
```

```
</PropertyGroup>
```

*Конфигурация и платформа:*

Вы можете указать конфигурацию (например, Debug или Release) и целевую платформу (например, AnyCPU, x86, x64) для вашего проекта:

```
<PropertyGroup>
```

```
  <Configuration>Release</Configuration>
```

```
  <Platform>AnyCPU</Platform>
```

```
</PropertyGroup>
```

*Компиляторные опции:*

Вы можете добавить дополнительные параметры компилятора, такие как предопределенные константы, предупреждения или отключение предупреждений:

```
<PropertyGroup>
```

```
  <DefineConstants>DEBUG;TRACE</DefineConstants>
```

```
  <WarningLevel>4</WarningLevel>
```

```
  <NoWarn>1591;1701</NoWarn>
```

```
</PropertyGroup>
```

*Ссылки на сборки (References):*

Вы можете добавить ссылки на сборки, которые ваш проект использует. Это делается с помощью элемента `<ItemGroup>` и `<Reference>`:

```
<ItemGroup>
```

```
  <Reference Include="System.Data" />
```

```
  <Reference Include="System.Xml" />
```

</ItemGroup>

*Файлы ресурсов (Resources):*

Если у вас есть ресурсы, такие как файлы ресурсов ресурсов или строковые ресурсы, вы можете добавить их в проект:

<ItemGroup>

<EmbeddedResource Include="Resources\myresource.resx" />

</ItemGroup>

*Создание исполняемых файлов:*

Если ваш проект является приложением, вы можете указать точку входа (главный класс) для вашего приложения. Это делается с помощью элемента `<Application>` внутри `<PropertyGroup>`:

<PropertyGroup>

<OutputType>Exe</OutputType>

<StartupObject>MyNamespace.MyMainClass</StartupObject>

</PropertyGroup>

*Настройка сборки (Build):*

Вы можете включить или выключить определенные этапы сборки, такие как компиляция кода, копирование файлов и др. Сделать это можно с помощью параметров `<Build>` внутри `<PropertyGroup>`:

<PropertyGroup>

<Build>>true</Build>

</PropertyGroup>

#### 4. *Настройка компилятора в файле \*.props:*

В файле ``.props`` для настройки компилятора в среде MSBuild вы можете использовать различные свойства, которые зависят от используемого компилятора и языка программирования. Вот некоторые распространенные настройки компилятора, которые можно установить в файле ``.props``:

*Свойства компилятора C# (для MSBuild с .NET и C#):*

`<LangVersion>`: Устанавливает версию языка C# (например, "latest" или "7.3").

`<PlatformTarget>`: Устанавливает целевую платформу для компиляции (например, "AnyCPU" или "x86").

`<TreatWarningsAsErrors>`: Определяет, следует ли рассматривать предупреждения как ошибки (true/false).

`<WarningLevel>`: Устанавливает уровень вывода предупреждений компилятора (например, "4" для максимального уровня).

`<DefineConstants>`: Позволяет определять символы препроцессора (например, "DEBUG" или "TRACE").

*Свойства компилятора C++ (для MSBuild с Visual C++):*

`<PlatformToolset>`: Устанавливает инструментарий компилятора (например, "v142").

`<AdditionalIncludeDirectories>`: Устанавливает дополнительные каталоги для поиска заголовочных файлов.

`<AdditionalLibraryDirectories>`: Устанавливает дополнительные каталоги для поиска библиотек.

`<TreatWarningAsError>`: Определяет, следует ли рассматривать предупреждения как ошибки (true/false).

`<WarningLevel>`: Устанавливает уровень вывода предупреждений компилятора (например, "Level4" для максимального уровня).

*Общие настройки для всех компиляторов:*

`<OutputPath>`: Устанавливает каталог для выходных файлов после компиляции.

`<IntermediateOutputPath>`: Устанавливает каталог для временных файлов, создаваемых в процессе компиляции.

*Пример файла `.props`, который настраивает компилятор C#:*

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <LangVersion>latest</LangVersion>
    <PlatformTarget>AnyCPU</PlatformTarget>
    <TreatWarningsAsErrors>true</TreatWarningsAsErrors>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
</Project>
```

*Пример файла `.props`, который настраивает компилятор C++:*

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <PlatformToolset>v142</PlatformToolset>
    <AdditionalIncludeDirectories>$(SolutionDir)include</AdditionalIncludeDirectories>
  >
    <AdditionalLibraryDirectories>$(SolutionDir)lib</AdditionalLibraryDirectories>
    <TreatWarningAsError>true</TreatWarningAsError>
    <WarningLevel>Level4</WarningLevel>
  </PropertyGroup>
</Project>
```

## 5. Настройка компилятора в файле \*.targets:

Для настройки компилятора в файле `*.targets` при использовании MSBuild, вы можете использовать различные свойства и элементы, которые определяют параметры компиляции для вашего проекта. Вот некоторые из наиболее часто используемых настроек:

`<CscToolPath>`: Указывает путь к исполняемому файлу компилятора C# (csc.exe).

`<CscToolExe>`: Указывает имя исполняемого файла компилятора C#.

`<CscArgs>`: Позволяет указать дополнительные аргументы, которые будут переданы компилятору C#.

`<WarningLevel>`: Устанавливает уровень предупреждений, который будет использоваться при компиляции.

`<TreatWarningsAsErrors>`: Указывает, следует ли рассматривать предупреждения как ошибки.

`<LangVersion>`: Устанавливает версию языка C#, которая будет использоваться при компиляции.

`<CodeAnalysisRuleSet>`: Указывает файл набора правил для статического анализа кода.

`<NoWarn>`: Позволяет отключить определенные предупреждения компилятора.

`<DefineConstants>`: Позволяет определить символы препроцессора, которые будут использоваться в коде.

`<Optimize>`: Указывает, следует ли включить оптимизацию при компиляции.

`<DebugType>`: Указывает тип информации для отладки (например, `'full'`, `'pdbonly'`, `'none'`).

`<PlatformTarget>`: Указывает целевую платформу (например, `'x86'`, `'AnyCPU'`, `'x64'`).

Файлы `.props` и `.targets` - это два типа файлов проектов, используемых в среде MSBuild для настройки и управления сборкой проектов. Они имеют разные цели и используются для разных задач:

### 1. Файлы .props:

Файлы `.props` используются для определения настроек, которые будут распространяться на все проекты, которые подключают этот файл.

Они обычно содержат свойства MSBuild, такие как параметры компилятора, определения препроцессора и другие настройки, которые нужны для сборки проектов.

Файл `.props` может быть общим для нескольких проектов и устанавливать общие настройки для всех этих проектов.

## 2. Файлы `.targets`:

Файлы `.targets` используются для определения целей сборки, которые выполняются на разных этапах сборки проекта.

Они обычно содержат инструкции по компиляции, копированию файлов, выполнению тестов и другие действия, которые должны быть выполнены в процессе сборки проекта.

Файл `.targets` может быть общим для нескольких проектов и предоставлять общие инструкции для всех этих проектов.

Обычно в проекте используется один файл `.props` и один или несколько файлов `.targets`. Файл `.props` устанавливает общие настройки для проектов, а файлы `.targets` определяют действия, которые выполняются при сборке проектов.

## ***6. Иерархическая настройка компилятора через файлы:***

В MSBuild можно настраивать компилятор и другие инструменты сборки с использованием файлов настройки. Иерархическая настройка компилятора обычно выполняется с использованием файлов проекта (например, `.csproj` для проектов на C#) и файлов конфигурации (например, `.props` и `.targets`).

Файлы проекта (`.csproj`): Файл проекта содержит настройки для конкретного проекта, такие как используемая версия языка, опции компилятора и ссылки на другие библиотеки. Вы можете редактировать этот файл непосредственно, чтобы настроить компилятор для вашего проекта.

Файлы конфигурации (`.props` и `.targets`): Файлы `.props` и `.targets` позволяют вам определять общие настройки, которые можно использовать для нескольких проектов или решений. Эти файлы можно импортировать в файлы проекта для применения настроек.

.props файлы: Эти файлы обычно содержат переменные и настройки, которые вы хотите использовать в своих проектах. Вы можете импортировать .props файл в файл .csproj с помощью элемента `<Import>`.

*Импорт .props файла в .csproj:*

```
<Import Project="MySettings.props" />
```

.targets файлы: Эти файлы обычно содержат дополнительные цели (targets) сборки и действия, которые можно выполнить перед или после компиляции проекта. Они также могут быть импортированы в файлы .csproj с помощью элемента `<Import>`.

Импорт .targets файла в .csproj:

```
<Import Project="MyTargets.targets" />
```

## ***7. Работа с менеджером пакетов NuGet:***

Работа с менеджером пакетов NuGet представляет собой важную часть процесса разработки на платформе .NET. NuGet - это инструмент для управления библиотеками и зависимостями в ваших проектах.

*Добавление пакетов:*

Visual Studio: Если вы используете Visual Studio, то для добавления пакетов NuGet к вашему проекту, вы можете открыть менеджер NuGet через меню `Project > Manage NuGet Packages...`. Выберите пакеты, которые вы хотите установить, и нажмите "Install".

.NET Core и .NET 5/6: Для проектов, созданных с использованием .NET Core или .NET 5/6, вы можете добавить пакеты через командную строку с помощью инструмента `dotnet`:

```
dotnet add package PackageName
```

Это добавит пакет в ваш файл проекта (.csproj) и установит его.

*Обновление пакетов:*

Visual Studio: Если вы используете Visual Studio, то обновление пакетов можно выполнить через менеджер NuGet. Выберите вкладку "Installed" в менеджере NuGet, и там вы увидите список установленных пакетов с доступными обновлениями.

.NET Core и .NET 5/6: Вы также можете обновить пакеты через командную строку с помощью `dotnet`:



```
dotnet update package PackageName
```

*Удаление пакетов:*

Visual Studio: В менеджере NuGet выберите установленный пакет и нажмите "Uninstall".

.NET Core и .NET 5/6\*: Используйте команду `dotnet remove package`:

```
dotnet remove package PackageName
```

*Восстановление пакетов:* Важно понимать, что в проектах .NET Core и .NET 5/6 необходимо восстанавливать пакеты после клонирования проекта или после изменения файла .csproj. Для этого используйте команду:

```
dotnet restore
```

*Использование пакетов:* После установки пакетов, вы можете начать использовать их в своем коде. Например, импортировать пространство имен и вызывать функции и классы, предоставляемые пакетом.

*Настройка версий:* Вы можете настроить версии пакетов в файле .csproj, чтобы управлять тем, какие версии пакетов будут установлены и использованы вашим проектом.

*Опубликование и хостинг пакетов:* Если у вас есть собственные библиотеки, которые вы хотите опубликовать для использования другими разработчиками, вы можете создать свой собственный NuGet-пакет и опубликовать его в репозиторий NuGet.

## **8. Использование и настройка пакетов в файле csproj:**

Вы можете добавлять пакеты NuGet прямо в файл .csproj, и указывать их версии и другие параметры.

Пример в файле .csproj:

```
<ItemGroup>
```

```
  <PackageReference Include="Newtonsoft.Json" Version="13.0.1" />
```

```
</ItemGroup>
```

## **9. Использование и настройка пакетов в файле \*.props:**

Вы также можете определять общие настройки для пакетов NuGet в файлах .props.

### ***10. Работа с утилитой dotnet:***

Утилита `dotnet` представляет собой мощный инструмент для управления проектами и сборкой в .NET Core и .NET 5/6. Она интегрирована в среду разработки и используется для выполнения различных задач, включая сборку, тестирование, добавление зависимостей и управление проектами.

#### **1. Создание нового проекта:**

Например, чтобы создать новое приложение на C# с использованием шаблона консольного приложения, выполните следующую команду:

```
dotnet new console -n MyConsoleApp
```

#### **2. Сборка проекта:**

Для сборки проекта используйте команду `dotnet build`. Она автоматически вызывает инструмент `msbuild` для выполнения сборки проекта. Просто перейдите в каталог с проектом и выполните:

```
dotnet build
```

Запуск приложения:

Чтобы запустить ваше приложение, используйте команду `dotnet run`:

```
dotnet run
```

Добавление зависимостей (пакетов NuGet):

Вы можете добавить зависимости к вашему проекту с помощью `dotnet add package`. Например, чтобы добавить пакет Entity Framework Core, выполните следующую команду:

```
dotnet add package Microsoft.EntityFrameworkCore
```

Тестирование проекта:

Для запуска тестов используйте команду `dotnet test`:

```
dotnet test
```

Публикация приложения:

Чтобы опубликовать приложение, чтобы оно было готово к развертыванию на сервере или другом месте, выполните команду ``dotnet publish``:

```
dotnet publish -c Release
```

Это создаст собранный и оптимизированный код приложения в папке ``bin/Release/netcoreappX.X/publish``.

Использование ``msbuild`` в ``dotnet``:

Вы также можете использовать ``msbuild`` напрямую, если у вас есть особые требования или настройки для сборки проекта. Например:

```
msbuild MyProject.csproj
```

Это позволит вам более тонко настроить процесс сборки.

``dotnet`` и ``msbuild`` хорошо интегрированы и взаимодействуют друг с другом в большинстве случаев. Вы можете выбирать тот инструмент, который лучше соответствует вашим потребностям в сборке и управлении проектом в .NET Core и .NET 5%.

### ***11. Интеграция скриптовых действий в компиляторе при сборке:***

Вы можете использовать задачи MSBuild, такие как ``<Exec>``, для выполнения скриптовых действий во время сборки проекта.

Пример выполнения скрипта PowerShell в файле .csproj:

```
<Target Name="RunCustomScript" BeforeTargets="Build">  
    <Exec Command="powershell -File CustomScript.ps1" />  
</Target>
```

## **Clang**

Компилятор Clang - это популярный компилятор для языка C и C++. Он предоставляет множество опций и настроек для управления процессом компиляции и оптимизации кода.

### ***1. Основные настройки компилятора Clang:***

Для компиляции наиболее простых программ вам может понадобиться всего лишь указать имя исходного файла и имя выходного файла. Например:

```
clang source.c -o executable
```

Можно указать уровень оптимизации с помощью флага `-O`, например:

```
clang -O2 source.c -o executable
```

### ***2. Условные конструкции при настройке компилятора:***

Вы можете использовать флаги компилятора Clang, чтобы включать или выключать определенные опции на основе условий. Например, использование `-D` для определения макросов препроцессора:

```
clang -DDEBUG_MODE source.c -o executable
```

### ***3. Настройка компилятора в файле CMakeLists.txt:***

CMake - это инструмент для автоматизации сборки проектов. Вы можете настроить компилятор Clang в файле CMakeLists.txt следующим образом:

```
project(MyProject)

set(CMAKE_C_COMPILER "clang")

set(CMAKE_CXX_COMPILER "clang++")

add_executable(my_executable source.cpp)
```

### ***4. Иерархическая настройка компилятора через файлы:***

Вы можете создать файл с настройками компилятора (например, `clang-config.cmake`) и использовать его в CMakeLists.txt следующим образом:

```
project(MyProject)

include("clang-config.cmake")
```

```
add_executable(my_executable source.cpp)
```

В файле `clang-config.cmake` вы можете определить дополнительные параметры компилятора.

### ***5. Интеграция скриптовых действий в компиляторе при сборке:***

Вы можете использовать скрипты или пользовательские команды в CMake для выполнения дополнительных действий при сборке. Например, создайте пользовательскую цель и добавьте к ней команды:

```
add_custom_target(my_target
    COMMAND echo "Running custom build commands"
    COMMAND ./my_script.sh
)
```

## Cmake

Указание необходимой версии cmake

```
cmake_minimum_required(VERSION 2.6)
```

Указывайте высокую минимальную версию CMake. Если используемая версия cmake меньше 2.6, он не захочет работать. Писать эту команду всегда - хороший стиль (cmake будет пыхтеть и обижаться, если вы не укажете версию, но собирать всё равно всё будет).

Название проекта

```
project(visualization)
```

Указывает, что этот cmake-файл является корневым для некоторого проекта. С проектами связаны определенные переменные и поведение cmake (читайте документацию).

Переменные

В cmake можно создавать текстовые переменные. Команда

```
set(VARIABLE The variable's value)
```

запишет в переменную "VARIABLE" значение "The variable's value". Чтобы где-либо использовать значение этой переменной, нужно написать `${VARIABLE}`.

Чтобы добавить к переменной некий текст, можно сделать так:

```
set(VARIABLE "${VARIABLE} new text")
```

Как видите, использовать значение можно и внутри кавычек. Переменные активно используются различными библиотеками - для установки флагов, параметров сборки/линковки и прочих вкусностей, об этом чуть-чуть попозже.

Пример коше'гного проекта со списком сорцов в отдельной переменной:

```
cmake_minimum_required(VERSION 2.6)
```

```
set(SOURCES test.cpp lib1.cpp lib2.cpp)
```

```
add_executable(test ${SOURCES})
```

Устанавливаем команды компилятору

```
add_definitions(-DSOME_IMPORTANT_DEFINITION)
```

Эта команда используется для установки дефайнов, которые можно проверить в коде через, например, `#ifdef SOME_IMPORTANT_DEFINITION`.

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11 -Wall")
```

Эта команда добавит к флагам, используемым при сборке с++-кода, флаги `-std=c++11` и `-Wall`.

Кто не знает: `"-std=c++11"` включает в gcc поддержку стандарта с++11, `"-Wall"` говорит gcc выводить все предупреждения (очень советую, помогает отловить много глупых багов и писать аккуратный код).

Если ваша версия GCC меньше, чем 4.7.0, вместо `-std=c++11` нужно использовать `-std=c++0x`.

В GCC 4.8.0 появился флаг `-std=c++1y`, в котором начинают реализовывать фичи следующего стандарта.

### Папка с хедерами

Допустим, Вы хотите, чтобы хедеры (файлики, подключаемые через `#include`) искались еще и в каталогах `"headers/"` и `"more_headers/"`:

```
include_directories("headers/" "more_headers/")
```

Надеюсь, и это понятно.

### Самое важное - подключение библиотек

Научимся искать и подключать библиотеки при помощи cmake на примере Boost. Для начала установим переменные для буста:

```
set(Boost_USE_STATIC_LIBS OFF)
```

```
set(Boost_USE_MULTITHREADED ON)
```

Первое - мы не хотим, чтобы буст подключался к нам статически (т.е. хотим динамическую линковку). Второй флаг разрешает бусту внутри своих магических реализаций использовать треды для распараллеливания и прочих радостей.

Итак, мы установили флаги. Давайте найдем буст!

Допустим, нам нужны компоненты буста под названием `chrono` (библиотека для работы со временем) и `filesystem` (библиотека для работы с файловой системой):

```
find_package(Boost COMPONENTS chrono filesystem REQUIRED)
```

Win, будут искаться только нужные библиотеки, и их расположение будет записано в переменную `Boost_LIBRARIES`.

Опция `"REQUIRED"` говорит о том, что библиотека необходима проекту. Без нее cmake решит, что отсутствие данной библиотеки - не так уж и страшно, и будет собирать дальше.

Добавим директории с хедерами буста для поиска в них хедеров:

```
include_directories(${Boost_INCLUDE_DIRS})
```

Итак, осталось найденные библиотеки подключить к исполняемому файлу.

```
target_link_libraries(test ${Boost_LIBRARIES})
```

В качестве библиотек нужно указать пути к необходимым собранным библиотекам. cmake нашел указанные нами библиотеки и записал в переменную, чем мы и пользуемся.

Заметим, что эту команду нужно вызывать после того, как создан target сборки (через add\_executable).

Как создать библиотеку в поддиректории и слинковать ее с основной программой

Пусть в ./ лежит основной проект, а в ./subdir мы хотим сделать либу, а в ./build построить проект.

```
./subdir/CMakeLists.txt
project(MegaLibrary)
set(SOURCES lib.cpp)
set(HEADERS lib.h)
add_library(lib
$
${SOURCES}
$
${HEADERS})
target_include_directories(lib PUBLIC
$
${CMAKE_CURRENT_SOURCE_DIR})
./CMakeLists.txt
project(MainProject)
set(MAIN_PROJECT_SRC_LIST main)
# Other stuff
add_executable(main
$
${MAIN_PROJECT_SRC_LIST})
add_subdirectory(subdir)
target_link_libraries(main lib)
```

Теперь можно в файлах основного проекта делать #include "lib.h" (см. документацию по target\_include\_directories).

В ./build запускаем "cmake .. && make" и получаем собранный проект.



## ***1. Основные настройки компилятора:***

CMake - это инструмент для автоматизации процесса сборки программного обеспечения, который позволяет настраивать компилятор и другие инструменты сборки для проекта. Настройка компилятора в CMake обычно выполняется с использованием переменных окружения и команды `CMAKE\_CXX\_COMPILER` (для C++) или `CMAKE\_C\_COMPILER` (для C).

*Установка компилятора по умолчанию:*

```
set(CMAKE_C_COMPILER "gcc")    # Для компилятора C (GCC)
set(CMAKE_CXX_COMPILER "g++")  # Для компилятора C++ (g++)
```

*Установка версии компилятора:*

```
set(CMAKE_CXX_STANDARD 11)     # Настройка стандарта C++ (например,
C++11)
set(CMAKE_C_STANDARD 99)       # Настройка стандарта C (например,
C99)
```

*Установка флагов компилятора:*

```
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -O2")  #
Добавление флагов компилятора для C++
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wall -O2")       # Добавление
флагов компилятора для C
```

*Использование альтернативных компиляторов:*

```
set(CMAKE_C_COMPILER "/путь/к/компилятору/gcc")      # Указание пути к
компилятору C
set(CMAKE_CXX_COMPILER "/путь/к/компилятору/g++")    # Указание пути к
компилятору C++
```

*Передача опций компилятора через командную строку:*

```
add_compile_options(-std=c++11)  # Передача опций компилятора через
командную строку (для C++)
```

*Установка переменных окружения:*

Если вы хотите установить переменные окружения для компилятора, вы можете сделать это перед запуском CMake:

```
export CC=/путь/к/компилятору/gcc  
export CXX=/путь/к/компилятору/g++  
cmake /путь/к/вашему/проекту
```

## ***2. Условные конструкции при настройке компилятора:***

CMake позволяет использовать условные конструкции для настройки компилятора в зависимости от определенных условий, таких как операционная система, версия компилятора и т. д. Пример условной настройки:

```
if(WIN32)  
  
    set(CMAKE_CXX_COMPILER cl) # Если Windows, используем компилятор  
    Visual C++  
  
else()  
  
    set(CMAKE_CXX_COMPILER g++) # В противном случае используем  
    компилятор GNU C++  
  
endif()
```

Этот код устанавливает компилятор в зависимости от операционной системы (Windows или не Windows).

## ***3. Настройка компилятора в файле CMakeLists.txt:***

Внутри файла `CMakeLists.txt`, который находится в корневой директории вашего проекта, вы также можете устанавливать компилятор. Например:

```
set(CMAKE_CXX_COMPILER g++) # Установка компилятора C++
```

Это позволяет настраивать компилятор для конкретного проекта, не затрагивая глобальные настройки.

## ***4. Иерархическая настройка компилятора через файлы:***

CMake позволяет организовывать проекты с иерархической структурой каталогов. Вы можете создавать файлы `CMakeLists.txt` в каждом подкаталоге проекта для настройки компилятора и других параметров. Например:

project/

|— CMakeLists.txt (главный файл)

|— src/

| |— CMakeLists.txt (конфигурация для исходных файлов)

| |— main.cpp

|— lib/

| |— CMakeLists.txt (конфигурация для библиотеки)

| |— mylib.cpp

|— build/

Каждый `CMakeLists.txt` может содержать настройки компилятора и другие параметры, специфичные для этой части проекта.

### ***5. Интеграция скриптовых действий в компиляторе при сборке:***

CMake позволяет выполнять произвольные скрипты или команды в процессе сборки с использованием команды `add\_custom\_command`. Например, вы можете использовать эту команду для выполнения скрипта перед компиляцией:

```
add_custom_command(  
    OUTPUT output_file  
    COMMAND your_script.sh  
    DEPENDS input_file  
)
```

Здесь `your\_script.sh` - это скрипт, который будет выполнен перед компиляцией, и `input\_file` - зависимость, указывающая, что скрипт должен быть выполнен, если `input\_file` изменился.

Это может быть полезным, если вам нужно выполнить дополнительные действия перед сборкой, такие как генерация файлов кода, копирование ресурсов или другие пользовательские действия.