

# **Компилирование и установка C/C++ приложений с помощью СMAKE.**

**Целикова Анна**

# C

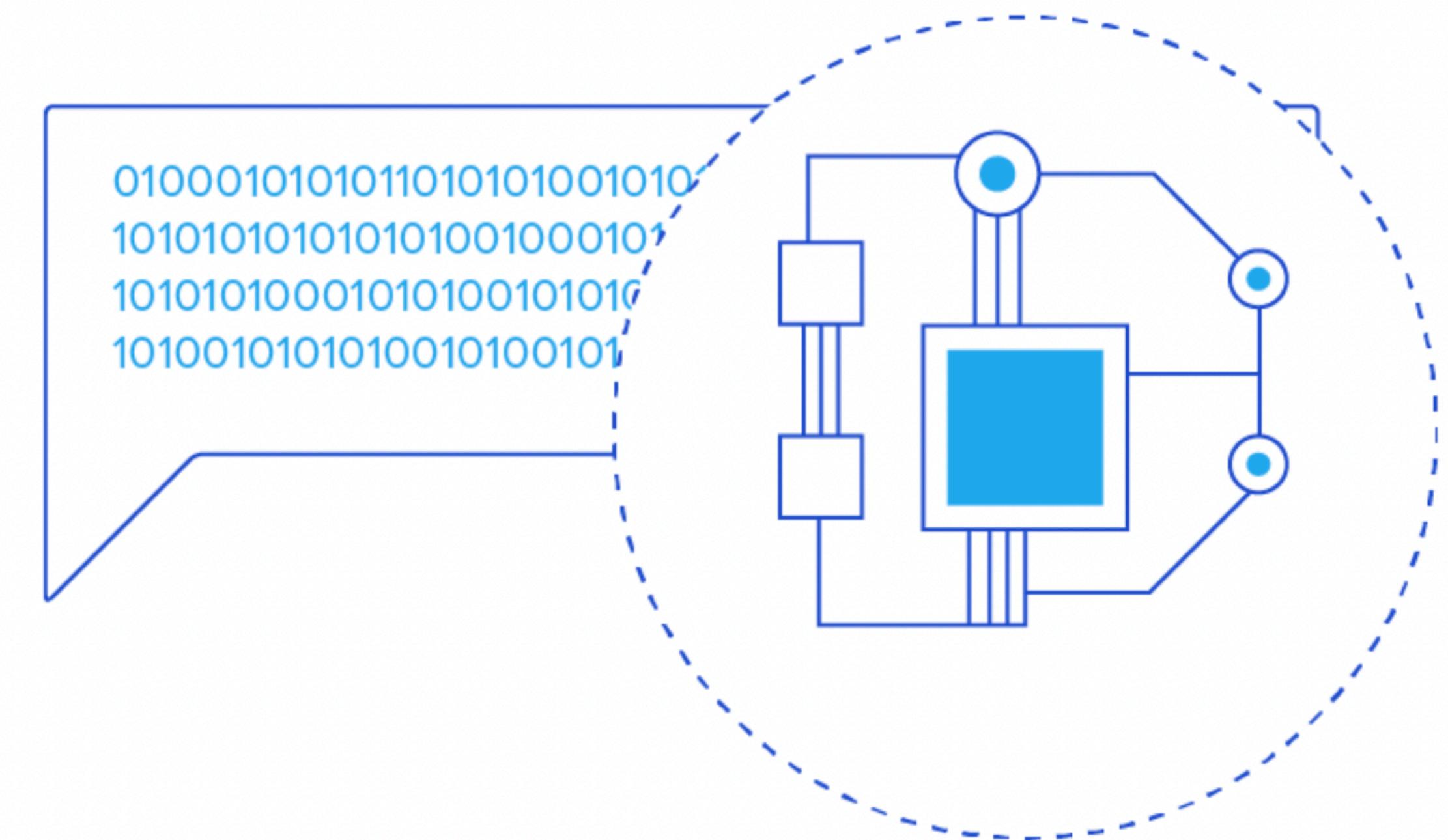
Язык "C" (произносится "си") — это универсальный язык программирования, для которого характерны экономичность выражения, современный поток управления и структуры данных, богатый набор операторов.

Язык "C" не является ни языком "очень высокого уровня", ни "большим" языком, и не предназначается для некоторой специальной области применения, но отсутствие ограничений и общность языка делают его более удобным и эффективным для многих задач, чем языки, предположительно более мощные.



## Примеры использования в GNU/Linux

Разработка операционной системы UNIX началась в 1969 году, а ее код был переписан на С в 1972 году. Язык С был фактически создан для переноса кода ядра UNIX с ассемблера на язык более высокого уровня, который будет выполнять те же задачи с меньшим количеством строк кода.



# Какие компании используют?

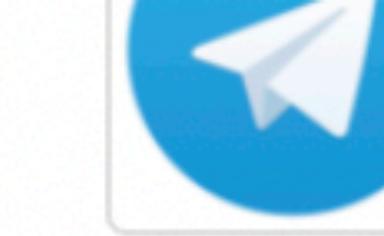
Узнать больше: [stackshare.io/c](https://stackshare.io/c)



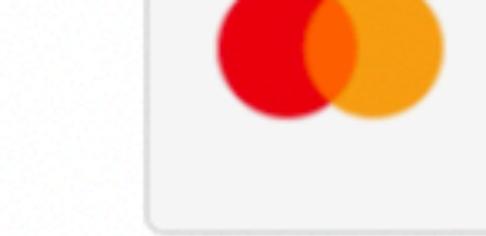
Twitch



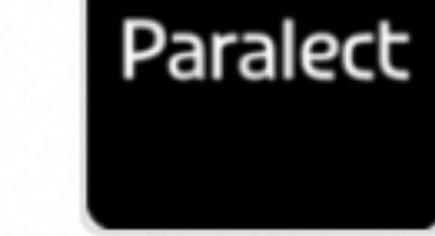
Github



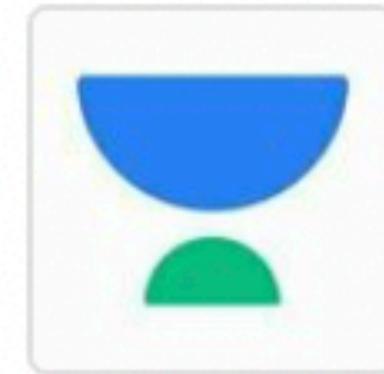
Telegram  
Messenger



MasterCard



Paralect



Unacademy



Venmo



SparkPost



Scopeland  
Technology



StockX



Backend



Cisco



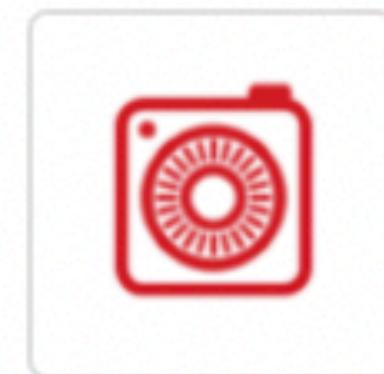
Crunchyroll



Glassdoor



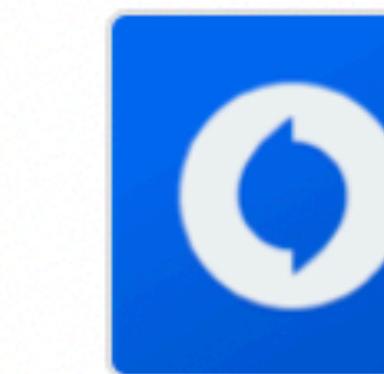
bet365



Carousell



Ubidreams



Juspay



CrunchBase



Esri



myfitnesspal



Kumho



Yum!



SALSIFY

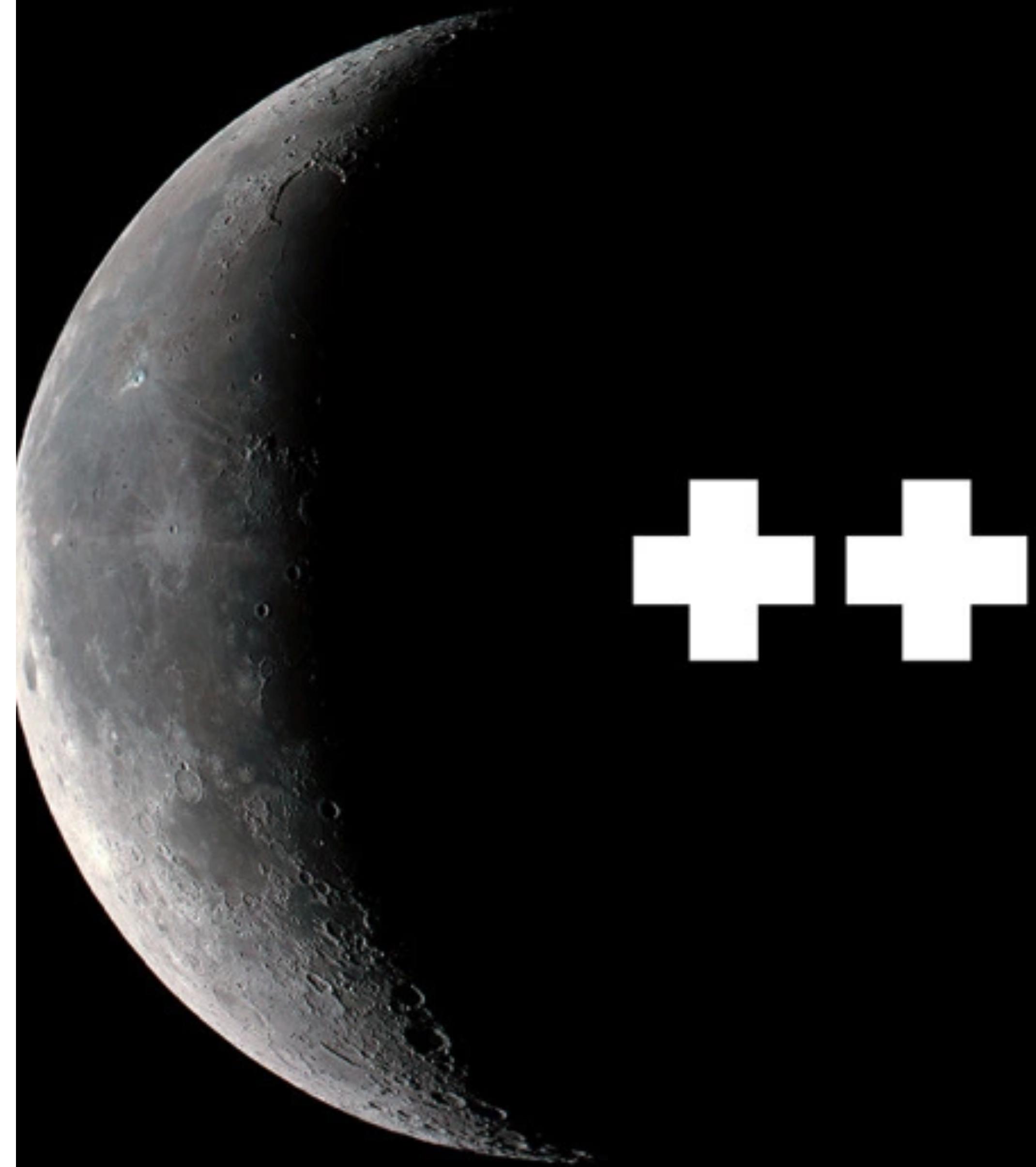


AMD

# C++

В начале 80-х годов прошлого века сотрудник Bell Labs Бьёрн Страуструп после долгих мучений с существующими языками программирования провёл эксперимент со скрещиванием C и Simula. Он даже не рассчитывал, что его детище, получившее название C++, привлечёт столько внимания.

Однако тогда язык произвёл настоящий фурор: компилируемый, структурированный, объектно-ориентированный, невероятно упрощающий работу с большими программами и при этом имеющий огромный потенциал для развития. Такой, что ещё почти десятилетие потребовалось Страуструпу, чтобы наделить C++ всеми характерными особенностями. Развитие же продолжается до сих пор.

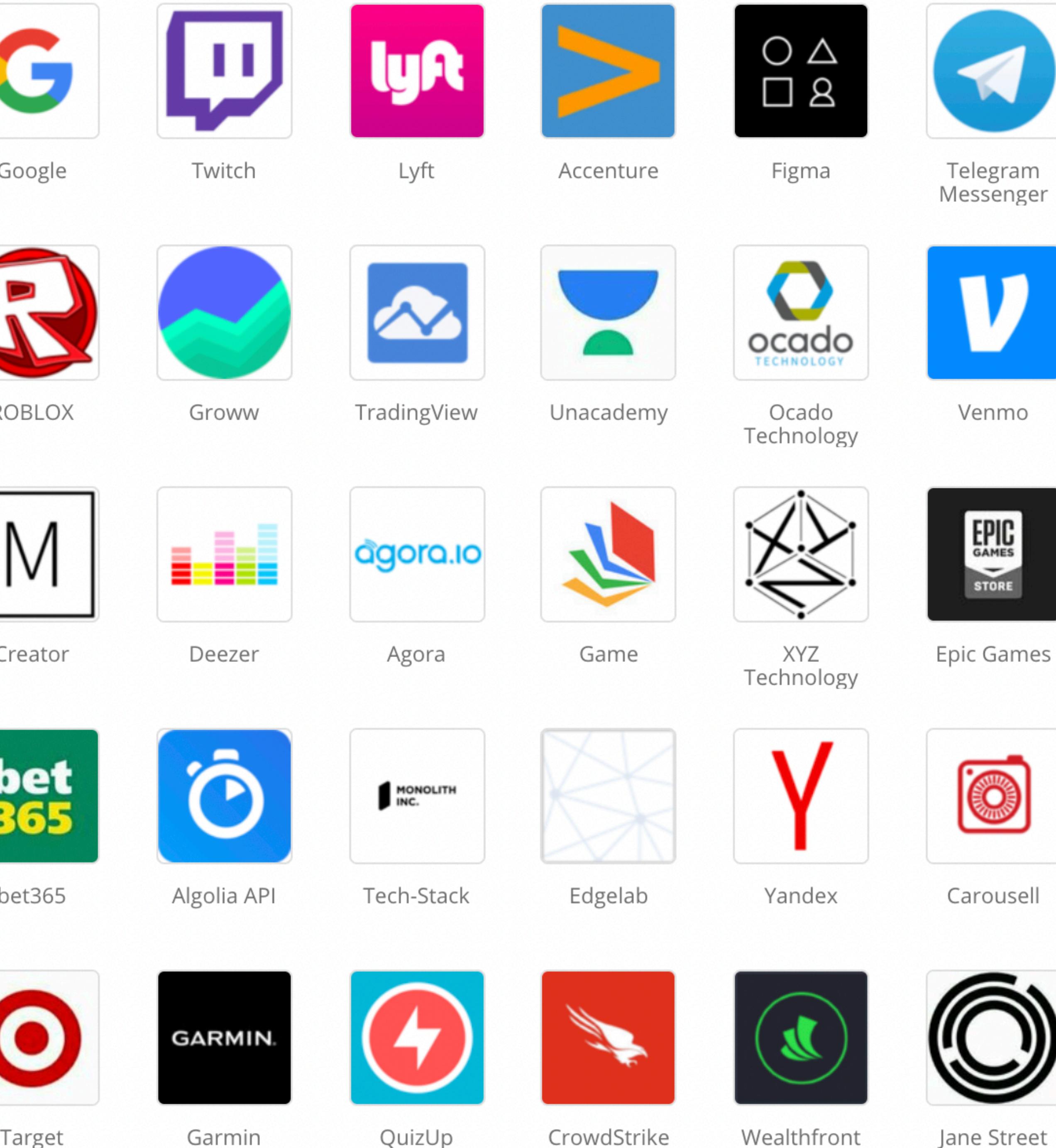


# Преимущества С/С++

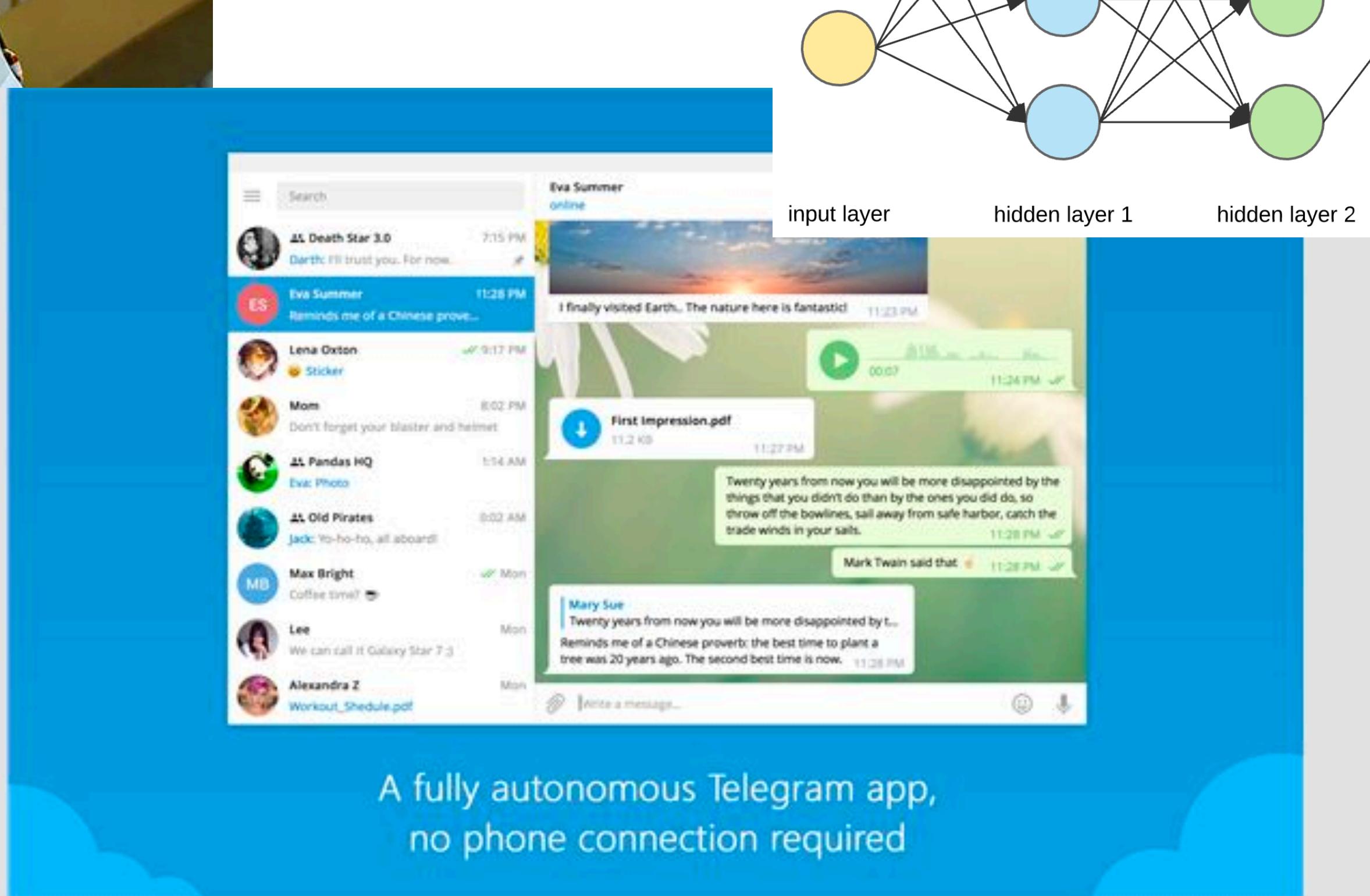
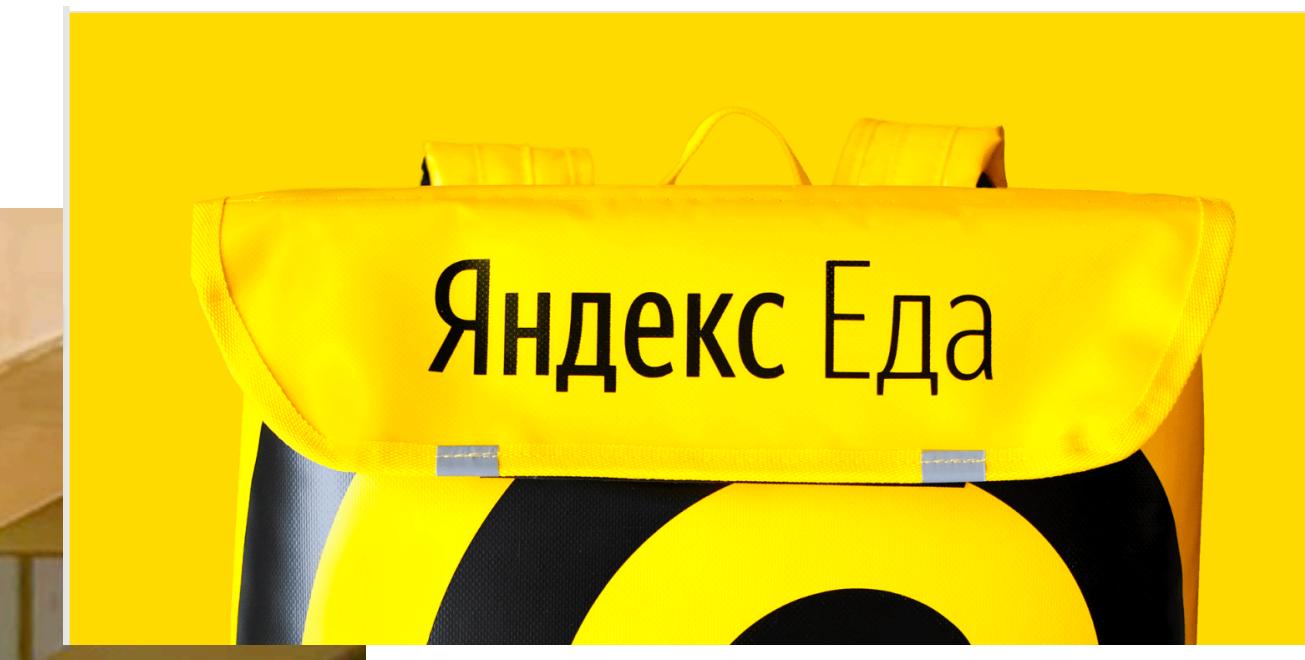
- Быстр
- Универсален
- Активно поддерживается
- Полезен в качестве фундамента для обучения
- Востребован
- Огромное количество старых, но необходимых для поддержки программ написано именно на нем

# Какие компании используют?

Узнать больше: [stackshare.io/  
cplusplus](https://stackshare.io/cplusplus)



# Из интересного полезны в: высоконагруженных системах, нейронных сетях (быстрые вычисления), беспилотниках, играх



# Еще немного о применении

- Ядро Linux, Microsoft Windows, Mac OS полностью написаны на языке C, а Android и iOS – частично.
- Являясь самым быстрым на сегодняшний день языком программирования, C++ оказывается одним из лучших для игр в 3D, многопользовательских и других. Counter-Strike, StarCraft: Brood War, Diablo I, World of Warcraft – все эти игры написаны на C++. Не говоря уже о консолях Xbox и PlayStation, в основе которых лежит программирование C++.
- В ядре игрового движка Unity – самого популярного движка для создания видеоигр под несколько операционных систем одновременно – также использовался C++.
- Photoshop, Illustrator и Adobe Premiere целиком написаны на C++.
- В Facebook перевели часть кода из PHP на язык C++, чтобы сократить затраты электроэнергии в расчёте на одного пользователя. Возможно, облачные системы хранения, базы данных, драйверы устройств и другие виды ПО тоже используют C++.

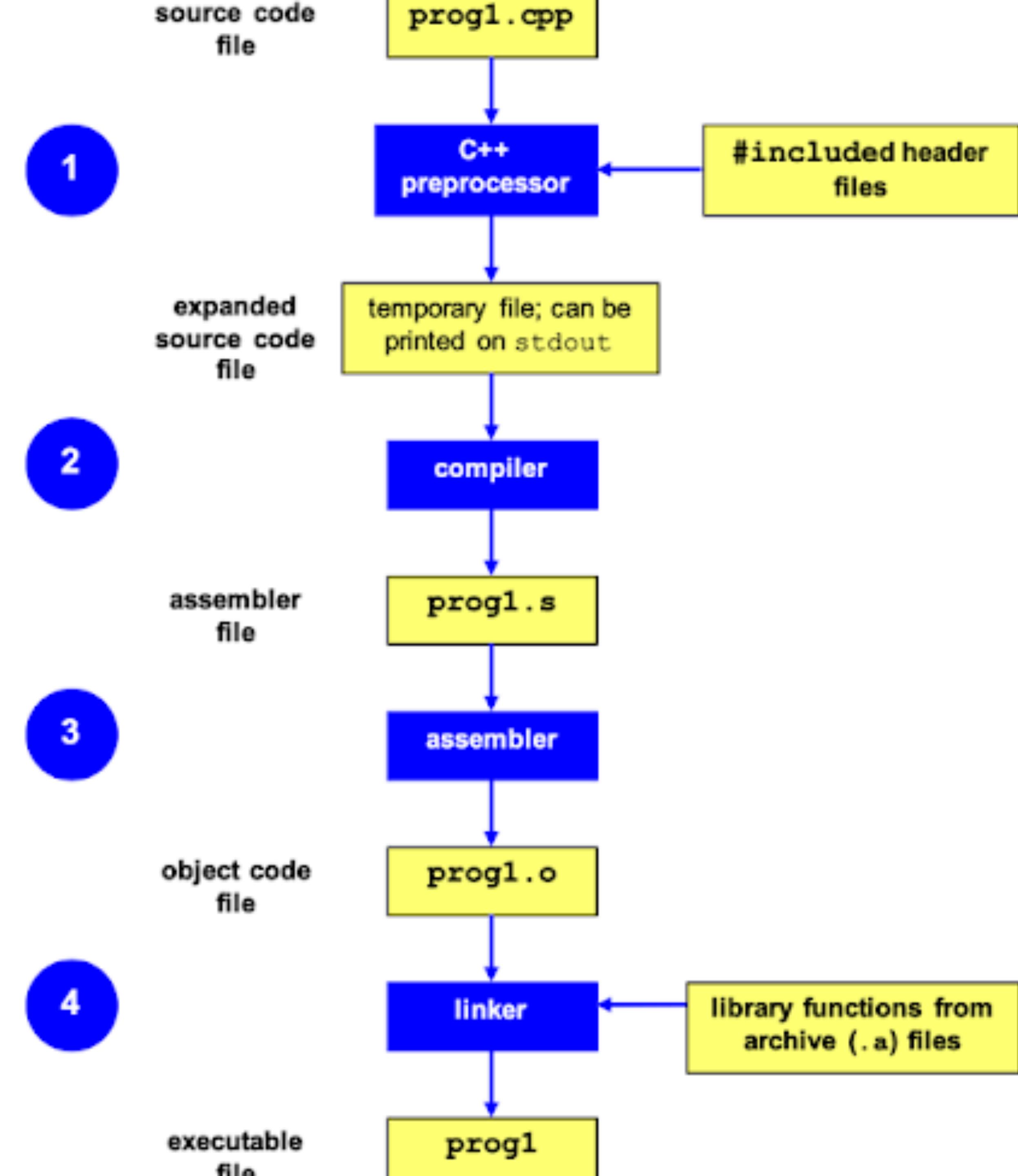
CMAKE



**CMake**  
*Cross-platform Make*

# Процесс компиляции на C/C++

1. **Препроцессинг.** С++ препроцессор копирует содержимое включенных (`#include`) заголовочных файлов в файл исходного кода, генерирует макрокод и заменяет символические константы, определенные с помощью `#define` их значениями.
2. **Компиляция.** Расширенный файл исходного кода, созданный С++ препроцессором, компилируется в ассемблере для платформы.
3. **Ассемблирование.** Ассемблерный код, генерируемый компилятором, собирается в объектный код платформы.
4. **Компоновка (линковка).** Генерируемый ассемблером объектный файл кода объединяется с функциями из стандартных архивных файлов библиотеки компоновщика для получения исполняемого файла. По умолчанию этот исполняемый файл называется `a.out`. В данном случае файл назван как `prog1.o`



# Что это и зачем нужно

CMake – кроссплатформенная автоматизированная система сборки проектов. Генерирует конфигурацию сборки (Unix системы - Makefiles), а сборка выполняется средствами системы (Unix - утилита make).

CMake может проверять наличие необходимых библиотек и подключать их, собирать проекты под разными компиляторами и операционными системами. Т.е. у вас есть куча кода и файлик, содержащий информацию для стаке, и чтобы скомпилиить это дело где-нибудь еще, вам нужно просто запустить там стаке, который сделает всё сам. Удобно, полезно, просто.

# Описание процесса

1. **CMake**. CMake читает проекты для сборки из файлов CMakeLists.txt, написанных на собственном языке. Из них он генерирует несколько Makefiles, которые можно использовать для сборки проекта.
2. **Make**. Make - утилита на unix-like операционных системах, обычно автоматизирующая процесс перевода файлов из одной формы в другую. Чаще всего, например, это процесс компиляции исходного кода.
3. **G++**. Make вызывает компилятор (G++), который непосредственно продуцирует нам исполняемый файл.

# Документация

- Можно найти здесь:  
[cmake.org/documentation/](https://cmake.org/documentation/)



CMake » 3.20.0



Documentation » CMake Referen

## Table of Contents

[Introduction](#)  
[Command-Line Tools](#)  
[Interactive Dialogs](#)  
[Reference Manuals](#)  
[Guides](#)  
[Release Notes](#)  
[Index and Search](#)

## Next topic

[cmake\(1\)](#)

## This Page

[Show Source](#)

## Quick search

Go

## Introduction

CMake is a tool to manage dialects of [Makefile](#), to such as Visual Studio and

CMake is widely used too.

People encountering C package downloaded from run the [cmake\(1\)](#) or [cmake](#)

The [Using Dependencies](#)

For developers starting [buildsystem\(7\)](#) manual coming familiar with the how to create package



```
1 # In CMake, this is a comment
2
3 # To run our code, please perform the following commands:
4 # - mkdir build && cd build
5 # - cmake ..
6 # - make
7 #
8 # With those steps, we will follow the best practice to compile into a subdir
9 # and the second line will request to CMake to generate a new OS-dependent
10 # Makefile. Finally, run the native Make command.
11
12 #-----
13 # Basic
14 #-----
15 #
16 # The CMake file MUST be named as "CMakeLists.txt".
17
18 # Setup the minimum version required of CMake to generate the Makefile
19 cmake_minimum_required(VERSION 2.8)
20
21 # We define the name of our project, and this changes some directories
22 # naming convention generated by CMake. We can send the LANG of code
23 # as the second param
24 project(learn cmake C)
25
26 # Set the project source dir (just convention)
27 set( LEARN_CMAKE_SOURCE_DIR ${CMAKE_CURRENT_SOURCE_DIR} )
28 set( LEARN_CMAKE_BINARY_DIR ${CMAKE_CURRENT_BINARY_DIR} )
29
30 # Include Directories
31 # In GCC, this will invoke the "-I" command
32 include_directories( include )
```

## Конфигурационный файл



```
1 # IMPORTANT
2 # Where are the additional libraries installed? Note: provide includes
3 # path here, subsequent checks will resolve everything else
4 set( CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH} "${CMAKE_SOURCE_DIR}/CMake/modules/" )
5
6 message (STATUS "DONE")
7
8 # Lists
9 # Setup the list of source files
10 set( LEARN_CMAKE_SOURCES
11     src/main.c
12     src/imagem.c
13     src/pather.c
14 )
15
16 # IMPORTANT
17 # Calls the compiler
18 # ${PROJECT_NAME} refers to Learn_CMake
19 add_executable( ${PROJECT_NAME} ${LEARN_CMAKE_SOURCES} )
20
21 # TRANSLATES ROUGHLY TO
22 # g++ -o learnmake src/main.c src/imagem.c
23
24 # Compiler Condition (gcc ; g++)
25 if ( "${CMAKE_C_COMPILER_ID}" STREQUAL "GNU" )
26     message( STATUS "Setting the flags for ${CMAKE_C_COMPILER_ID} compiler" )
27     add_definitions( --std=c99 ) # IMPORTANT
28 endif()
```

## Конфигурационный файл

# Установка



```
1 # install cmake
2 sudo apt-get install cmake
3
4 # check
5 cmake --version
```

**Пример работы с самым  
простым проектом**

# Пример

1. Рассмотрим пример кода на C++, который складывает два числа. Нам нужен cpp файл.
2. Создадим CMakeLists.txt для дальнейшей работы с CMake.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main() {
6     int a, b;
7     cin >> a >> b;
8     cout << a + b;
9     return 0;
10 }
```

```
1 # Setup the minimum version required of CMake
2 # to generate the Makefile
3 cmake_minimum_required(VERSION 3.10)
4
5 # We define the name of our project
6 project(aplusb sum.cpp)
7
8 # Calls the compiler
9 add_executable(aplusb sum.cpp)
```

# Пример

3. В нашей папке всего 2 файла: CMakeLists.txt и sum.cpp. Наш конфигурационный файл для CMake готов, поэтому вызовем CMake для текущей директории.

'cmake.'

По выводу в консоли мы можем понять, что CMake создал Build-файлы для make.

```
user@playground:~/suffer-cmake$ ls  
CMakeLists.txt  sum.cpp  
user@playground:~/suffer-cmake$ cmake .  
-- The C compiler identification is GNU 8.3.0  
-- The CXX compiler identification is GNU 8.3.0  
-- Check for working C compiler: /usr/bin/cc  
-- Check for working C compiler: /usr/bin/cc -- works  
-- Detecting C compiler ABI info  
-- Detecting C compiler ABI info - done  
-- Detecting C compile features  
-- Detecting C compile features - done  
-- Check for working CXX compiler: /usr/bin/c++  
-- Check for working CXX compiler: /usr/bin/c++ -- works  
-- Detecting CXX compiler ABI info  
-- Detecting CXX compiler ABI info - done  
-- Detecting CXX compile features  
-- Detecting CXX compile features - done  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /home/user/suffer-cmake  
user@playground:~/suffer-cmake$ █
```

# Пример

После предыдущей команды  
наша директория пополнилась  
новыми файлами.

```
user@playground:~/suffer-cmake$ ls
CMakeCache.txt  cmake_install.cmake  Makefile
CMakeFiles       CMakeLists.txt    sum.cpp
user@playground:~/suffer-cmake$ █
```

# Пример

4. Вызовем make, т.к. у нас теперь есть Makefile.

‘make’

5. Результат работы make - исполняемый файл “aplusb”.

6. Запускаем исполняемый файл.

‘./aplusb’

Все работает!

```
user@playground:~/suffer-cmake$ ls
CMakeCache.txt  cmake_install.cmake  Makefile
CMakeFiles      CMakeLists.txt      sum.cpp
user@playground:~/suffer-cmake$ make
Scanning dependencies of target aplusb
[ 50%] Building CXX object CMakeFiles/aplusb.dir/sum.cpp.o
[100%] Linking CXX executable aplusb
[100%] Built target aplusb
user@playground:~/suffer-cmake$ ls
aplusb          CMakeFiles           CMakeLists.txt  sum.cpp
CMakeCache.txt  cmake_install.cmake  Makefile
```

```
user@playground:~/suffer-cmake$ ./aplusb
2 4
6
user@playground:~/suffer-cmake$ █
```

# Пример работы с большим Open-Source проектом

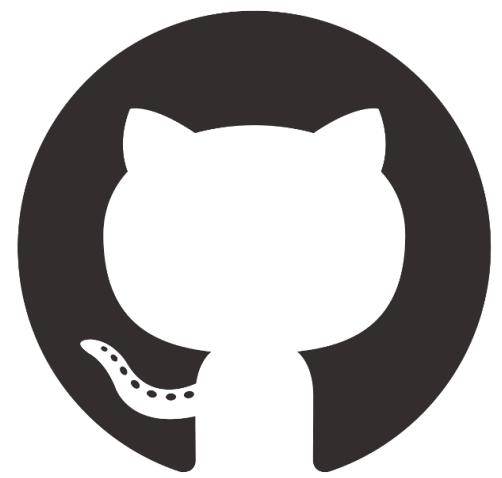
# Ultralight

“Render HTML at the Speed of Light  
Ultra-fast, ultra-light, standards-compliant HTML renderer for applications and games.”



# Ultralight

Next-Generation HTML Renderer



# ultralight-ux/Ultralight

Search or jump to... / Pull requests Issues Marketplace Explore

ultralight-ux / Ultralight Watch 91 Star 3.2k Fork 145

Code Issues 154 Pull requests 1 Actions Wiki Security Insights

master 1 branch 4 tags Go to file Add file Code

adamjs	Fix repaint bug on Windows (thanks for the report Dark Empath!)	5011dbf 18 days ago	115 commits
license	Update third-party license notices.	8 months ago	
media	Update README and logo.	5 months ago	
packager	Generate SDK for UWP, update links in README. Remove debug bins.	10 months ago	
samples	Merging in new 1.3 branch changes (API breaking changes!), adds mu...	18 days ago	
.gitignore	Pruned repo history to get rid of 200MB of stale binaries. Repo has n...	2 years ago	
CMakeLists.txt	Fix Linux build errors, update dependencies.	10 months ago	
Deps.cmake	Fix repaint bug on Windows (thanks for the report Dark Empath!)	18 days ago	
Jenkinsfile	Disable UWP build.	6 months ago	
README.md	Update README	5 months ago	

About

Next-generation HTML renderer for apps and games

ultralight

windows macos linux gamedev  
games opengl metal  
cross-platform gpu  
game-development webkit  
html-renderer desktop-apps  
directx-11 ultralight

Readme

Releases 4

# Процесс сборки

Ссылка на документацию по сборке: [docs.ultralig.ht/docs/trying-the-samples](https://docs.ultralig.ht/docs/trying-the-samples)

1. `git clone https://github.com/ultralight-ux/Ultralight.git`

Клонируем репозиторий проекта. В папке проекта есть нужный нам конфигурационный файл CMakeLists.txt.

2. `sudo apt-get install libx11-dev`

`sudo apt-get install xorg-dev libglu1-mesa-dev`

Нужно установить дополнительные библиотеки.

3. `mkdir build`

`cd build`

`cmake ..`

`cmake --build . --config Release`

Считается хорошей практикой хранить build-файлы в папке build. ‘`cmake ..`’ - указывает на папку, где лежит наш CMakeLists.txt. Он как раз лежит уровнем выше относительно папки build.

‘`cmake --build . --config Release`’ – на самом деле под капотом вызывает `make` (на Unix системах). ‘`--build .`’ указывает, где лежат конфигурационные файлы для сборки (системозависимо). То есть, эта команда говорит `cmake`’у: “разберись, какая в системе сборочная утилита, и вызови её”.

# Процесс сборки

Ссылка на документацию по сборке: <docs.ultralig.ht/docs/trying-the-samples>

4. *cd /build/samples/<Sample Name>/*

В этой директории будут располагаться папки с исполняемыми файлами.

# Полезные ссылки

- [cmake.org/documentation/](https://cmake.org/documentation/)
- [cmake.org/cmake/help/v3.20/guide/tutorial/index.html](https://cmake.org/cmake/help/v3.20/guide/tutorial/index.html)
- [derekmolloy.ie/hello-world-introductions-to-cmake/](https://derekmolloy.ie/hello-world-introductions-to-cmake/)
- [gist.github.com/martinbinna/c61dbb39bca0e4fb7d1f73b0d66a4fd1](https://gist.github.com/martinbinna/c61dbb39bca0e4fb7d1f73b0d66a4fd1)



**Таким образом, мы создали  
платформонезависимый скрипт сборки  
проекта (за исключением установки  
зависимостей :)**