Build rendszerek

Cserép Máté
ELTE Informatikai Kar
2020.

Build rendszerek

„The problem of handling the compilation of a project has been encountered well before you were born. That's why there's a single method to do it today, being a consensus since several decades.

Ha! Ha! Just kidding!"

Julien Jorge

Build rendszerek

C++ programok fordítása

- Hogyan tudunk konzolos eszközökkel lefordítani egy C++ programot (pl. GNU g++ fordítóval)?

```
g++ -c -o foo.o foo.cpp \
    -O2 -std=c++11 -pedantic -I./include/


... további fordítási egységek ...


g++ -c -o main.o main.cpp \
    -O2 -std=c++11 -pedantic -I./include/

g++ -o program.exe  foo.o ...  main.o
```

Build rendszerek

C++ programok fordítása

- Mint látjuk, már egy egyszerűbb, néhány forrás fájlból álló program esetén sem egyszerű a megoldás.

- **Problémák:**
  - A programok manuális fordítása már kisebb programoknál is könnyen nehezen kezelhetővé válik.
  - Nagyobb programoknál nem követhető mely fordítási egységek újrafordítása szükséges.
    - A teljes alkalmazás újrafordítása valós vállalati alkalmazásoknál hosszabb időt is igényelhet.

Build rendszerek

Fordító eszközök kategorizálása

- Önálló (*standalone*)
  - Projekt automatizált fordítása megadott utasítások és/vagy szabályok alapján.
  - Pl. Make, NMake, Scons, Jom, BJam, Ninja
- Fejlesztő környezetebe integrált (*integrated*)
  - Az IDE-vel együtt használható build rendszer
  - Pl. Visual Studio, Xcode, Eclipse
- Generátorok (*generators*)
  - Olyan generátorok, amelyekkel más, *standalone* build rendszerek forrásfájljai állíthatóak elő, automatizáltan.
  - CMake, Autotools, GYP

Build rendszerek – Make

Telepítés

- GNU Make
  - https://www.gnu.org/software/make/
- Telepítés:
  - Windows alatt célszerű a MinGW-t (*Minimalist GNU for Windows*) telepíteni, majd ennek *bin* könyvtárát a *PATH* környezeti változóhoz adni.
  - UNIX rendszerekre a *package repository* tartalmazza.
    - Debian/Ubuntu: `apt-get install make`
  - Jellemzően fejlesztőkörnyezetekkel együtt is telepítésre kerül (pl. JetBrains CLion, Qt Creator).

Build rendszerek – Make

Jellemzők

- A fordítást célokon (*target*) és szabályokon (*rule*) keresztül definiáljuk.
- A szabályok függőségeket (*dependency*) és utasításokat (*command*) tartalmazhatnak.
- A szabályrendszert az ún. *Makefile* tartalmazza.
- Futtatás: **`make`**
  - Amennyiben eltérő a *Makefile* neve: **`make -f MyMakefile`**
- Példa Makefile:

```
program: foo.cpp main.cpp
    g++ -o program foo.cpp main.cpp -std=c++11
```

függőség

szabály

cél

utasítás

Build rendszerek – Make

Szabályok

- A *Makefile* szabályok a következő módon épülnek fel:
  - Cél (*target*): a szabály által előállítandó, gyakran (de nem feltétlenül) bináris állomány.
  - Függőségek (*dependencies*): amennyiben a cél nem létezik vagy ha a függőségek valamelyike frissül, a célt újra kell generálni.
    - A függőségek frissülése a fájlok időbélyege alapján eldönthető.
  - Utasítások (*commands*): a cél előállításához szükséges végrehajtandó utasítások.

Build rendszerek – Make

Változók

- A *Makefile*ban definiálhatunk a szabályok mellett változókat is:

```
CXX=g++
CFLAGS=-std=c++11 -pedantic

program: foo.cpp main.cpp
  $(CXX) -o program foo.cpp main.cpp $(CFLAGS)
```

Build rendszerek – Make

Több fordítási egység kezelése

- További szabályokkal több fordítási egységet is kezelhetünk:

```
CXX=g++
CFLAGS=-std=c++11 -pedantic

foo.o: foo.cpp foo.h
	$(CXX) -c -o foo.o foo.cpp $(CFLAGS)

main.o: main.cpp
	$(CXX) -c -o main.o main.cpp $(CFLAGS)

program: foo.o main.o
	$(CXX) -o program foo.o main.o $(CFLAGS)
```

Build rendszerek – Make

Automatikus változók

- Az automatikus változók segítségével szabályainkat minták alapján általánosíthatjuk.
- Fontosabb automatikus változók:
  - $@: a cél (fájl)neve
  - $<: az első függőség neve
  - $^: az összes függőség neve
  - $?: a célnál frissebb függőségek nevei
  - $(@D): a cél nevének könyvtárrésze
  - $(@F): a cél nevének fájlrésze

Build rendszerek – Make

Automatikus változók

- Minta szabály (*pattern rule*) az object fájlok fordítására:

```
CXX=g++
CFLAGS=-std=c++11 -pedantic
DEPS=foo.h

%.o: %.cpp $(DEPS)
	$(CXX) -c -o $@ $< $(CFLAGS)

program: foo.o main.o
	$(CXX) -o program foo.o main.o $(CFLAGS)
```

Build rendszerek – Make

Automatikus változók

- Object fájlok kiemelése (és a redundancia elkerülése):

```
CXX=g++
CFLAGS=-std=c++11 -pedantic
DEPS=foo.h
OBJ=foo.o main.o

%.o: %.c $(DEPS)
	$(CXX) -c -o $@ $< $(CFLAGS)

program: $(OBJ)
	$(CXX) -o $@ $^ $(CFLAGS)
```

Build rendszerek – Make

Projekt könyvtár struktúra kezelése

- A forrás és fordítási könyvtárszerkezet kezelése (pl. *include*, *obj*):

```
IDIR=../include
ODIR=obj
CXX=g++
CFLAGS=-std=c++11 -pedantic -I$(IDIR)
_DEPS=foo.h
DEPS=$(patsubst %, $(IDIR)/%, $(_DEPS))        → ../include/foo.h
_OBJ=foo.o main.o
OBJ = $(patsubst %, $(ODIR)/%, $(_OBJ))        → obj/foo.o obj/main.o

$(ODIR)/%.o: %.cpp $(DEPS)
	$(CXX) -c -o $@ $< $(CFLAGS)
$(ODIR)/program: $(OBJ)
	$(CXX) -o $@ $^ $(CFLAGS)
```

Build rendszerek – Make

Hamis célok

- A Makefile célok alapvetően a szabály eredményeként előálló állományt jelölik meg. Nézzünk példát, ahol ez nem teljesül:

```
clean:
  rm -f $(ODIR)/*.o $(ODIR)/program
```

- Amennyiben létezik *clean* nevű állomány, a *make clean* utasítás nem fogja végrehajtani az utasításokat. (Nincsenek függőségek.)
- Az ilyen célokat hamis (*phony*) céloknak jelöljük. Tipikusan pl.:

```
.PHONY: clean
clean:
  rm -f $(ODIR)/*.o $(ODIR)/program

.PHONY: all
all: $(ODIR)/program
```

Build rendszerek – Make

Modulok

- A GNU Make támogatja a modularizált projekteket. Ilyenkor minden alkönyvtár saját *Makefile*t tartalmaz, amelynek végrehajtását a szülő könyvtár *Makefile*ja kezdeményezi:

```
.PHONY: all

all: program

  $(MAKE) -C module_a
  $(MAKE) -C module_b
  $(CXX) -o program $(CFLAGS) \
    module_a/obj/modul_a.o \
    module_b/obj/modul_b.o
```

Build rendszerek – Make

Példa

- Minta projekt: konzolos tételszám generáló alkalmazás
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-cpp

```
1 # Programs & tools (Linux)
2 #CXX      = g++
3 #RM       = rm -f
4 #MKDIR    = mkdir -p
5 #CP       = cp -f
6
7 # Programs & tools (Windows)
8 CXX      = g++
9 RM       = rmdir /s /q
10 MKDIR    = mkdir
11 CP       = copy /y
12
13 # Flags & options
14 CFLAGS   = -O2 -std=c++11 -pedantic
15 INCLUDE  = -I./include/
16
17 # Output
18 OBJDIR   = build
19 MAIN     = thesisgenarator.exe
20 TARGET   = $(OBJDIR)/$(MAIN)
21
22 _OBJ = GeneratorModel.o main.o
23 OBJ = $(patsubst %, $(OBJDIR)/%, $(_OBJ))
24
25 # Make targets
26 .PHONY: all
27 all: $(TARGET)
28
29 $(OBJDIR):
30   $(MKDIR) $(OBJDIR)
31
32 $(OBJDIR)/GeneratorModel.o: GeneratorModel.cpp GeneratorModel.h
33   $(CXX) -c -o $@ $< $(CFLAGS) $(INCLUDE)
34
35 $(OBJDIR)/main.o: main.cpp GeneratorModel.h
36   $(CXX) -c -o $@ $< $(CFLAGS) $(INCLUDE)
37
38 $(TARGET): $(OBJDIR) $(OBJ)
39   $(CXX) -o $@ $(OBJ) $(CFLAGS)
40
41 .PHONY: clean
42 clean:
43   $(RM) $(OBJDIR)
```

Build rendszerek – CMake

Telepítés

- CMake
  - https://cmake.org/
- Platformfüggetlen és fordítófüggetlen generátor alacsonyabb szintű fordító eszközökhöz.
  - Pl. GNU Make, MSVC, Ninja.
- Telepítés:
  - A hivatalos weboldalon elérhetőek a binárisok és telepítők Windows, Linux és Mac operációs rendszerekre is. https://cmake.org/download/
  - UNIX alapú operációs rendszerekre jellemzően *package repository*-ból is telepíthető.
    - Debian/Ubuntu: `apt-get install cmake`

Build rendszerek – CMake

Használat

- A konfigurációt a *CMakeLists.txt* állományban adjuk meg.
- Futtatás: **`cmake <paraméterek>`**
- Példa CMakeLists.txt:

```
cmake_minimum_required (VERSION 3.7)
project(HelloWorld)
add_executable(HelloWorld main.cpp foo.cpp)
```

Minimálisan szükséges CMake verzió

Projekt neve

Új végrehajtható bináris cél hozzáadása: *HelloWorld* bináris készítése: *main.cpp* és a *foo.cpp* forrásfájlokból

Build rendszerek – CMake

Változók

- Definiálhatók változók és feloldhatjuk a `${name}` szintaxissal:

```
cmake_minimum_required (VERSION 3.7)
project(HelloWorld)

set(SRCS \
  main.cpp \
  foo.cpp \
  foo.h)

add_executable(HelloWorld ${SRCS})
```

Build rendszerek – CMake

C++ fordító paraméterezése

- Testre szabhatjuk a C++ fordító paraméterezését:

```
cmake_minimum_required (VERSION 3.7)
project(HelloWorld)
add_executable(HelloWorld main.cpp foo.cpp)

target_compile_features(
  HelloWorld PRIVATE cxx_std_11)
target_compile_options(
  HelloWorld PRIVATE -O2 -pedantic)
```

GNU GCC specifikus kapcsolók

A láthatósági szabályozókra később térünk vissza

Build rendszerek – CMake

Programkönyvtárak fordítása

- Statikus vagy dinamikus programkönyvtár készítése:

```
cmake_minimum_required (VERSION 3.7)
project(MyStack)

add_library(my_stack_static STATIC mystack.cpp)
add_library(my_stack_dynamic SHARED mystack.cpp)
```

  - Statikus könyvtárak Windows alatt `.lib`, UNIX operációs rendszerek alatt `.a` kiterjesztéssel rendelkeznek.
  - Dinamikus könyvtárak Windows alatt `.dll`, UNIX operációs rendszerek alatt `.so` kiterjesztéssel rendelkeznek.

Build rendszerek – CMake

Include útvonal konfigurálása

- Bővítsük az *include* útvonalat az *include* könyvtárral:

```
cmake_minimum_required (VERSION 3.7)
project(HelloWorld)


target_compile_features(
  HelloWorld PRIVATE cxx_std_11)
target_compile_options(
  HelloWorld PRIVATE -O2 -pedantic)


include_directories("include")


add_executable(HelloWorld main.cpp foo.cpp)
```

Build rendszerek – CMake

Fordítási célok szerkesztése (*linkelése*)

```
cmake_minimum_required (VERSION 3.7)
project(HelloWorld)


add_executable(HelloWorld
  main.cpp foo.cpp mystack.h)
add_library(MyStack STATIC mystack.cpp mystack.h)
target_compile_options(
  MyStack PRIVATE -O2 -pedantic)


target_link_libraries(HelloWorld MyStack)
```

- Ilyenkor van szerepe a korábban említett láthatósági szabályozóknak.

Build rendszerek – CMake

Fordítási célok szerkesztése (*linkelése*)

- A következő láthatósági szabályozók érhetőek el:
  - **PRIVATE**: a kapcsolók csak az adott fordítási célra vonatkoznak. A példában így a `-O2 -pedantic` kapcsolók a `HelloWorld` cél fordítására már nem érvényesek.
  - **PUBLIC**: a kapcsolók az adott fordítási célt feldolgozó további célokra is érvényesek.
  - **INTERFACE**: a kapcsolók az adott fordítási célra nem, de az adott fordítási célt feldolgozó további célokra érvényesek.

Build rendszerek – CMake

Csomagok

- A CMake számos használt külső függőségeket kezel csomagként és tudja azt a programunkhoz fordítani és szerkeszteni.

```
cmake_minimum_required (VERSION 3.7)
project(MyProgram)

find_package(SomePackage)
include_directories(...)
link_directories(...)
link_libraries(...)
```

- Elegendő lehet a célhoz linkelni:

```
target_link_directories(mytarget ...)
target_link_libraries(mytarget ...)
```

Build rendszerek – CMake

Csomagok

- Példa az OpenCV könyvtár használatára:

```
cmake_minimum_required (VERSION 3.7)
project(MyProgram)

find_package(OpenCV REQUIRED)
link_libraries(${OpenCV_LIBS})
```

Build rendszerek – CMake

Csomagok

- Példa a Boost könyvtár használatára:

```
cmake_minimum_required (VERSION 3.7)
project(MyProgram)


set(Boost_USE_STATIC_LIBS ON)
set(Boost_USE_STATIC ON)
find_package(Boost REQUIRED
  COMPONENTS filesystem log program_options)
include_directories(${Boost_INCLUDE_DIRS})
link_libraries(${Boost_LIBRARIES})
```

Build rendszerek – CMake

Csomagok

- Példa a Qt könyvtár használatára:

```
cmake_minimum_required (VERSION 3.7)
project(QtHelloWorld)

set(CMAKE_INCLUDE_CURRENT_DIR ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTOUIC ON)

find_package(Qt5Widgets CONFIG REQUIRED)

set(SRCS mainwindow.ui mainwindow.cpp main.cpp)
add_executable(helloworld WIN32 ${SRCS})
target_link_libraries(helloworld Qt5::Widgets)
```

Build rendszerek – CMake

Modulok

- A CMake támogatja a modularizált projekteket. Ilyenkor minden alkönyvtár saját *CMakeLists.txt* fájlt tartalmaz, a szülő könyvtár hivatkozza a kezelendő alkönyvtárakat:

```
cmake_minimum_required (VERSION 3.7)
project(MyProgram)

... Konfiguráció ...

add_subdirectory(module_a)
add_subdirectory(module_b)
```

Build rendszerek – CMake

Generátorok

- Jellemzően egy külön *build könyvtárba* szeretnénk fordítani:

  `mkdir build && cd build`

  `cmake ../` vagy `cmake ../src`

- Több féle generátor közül választhatunk:
  - GNU Makefile használata:
    `cmake -G "Unix Makefiles" ../`
  - Microsoft Visual Studio:
    `cmake -G "Visual Studio 15 2017 Win64" ../`
  - Xcode:
    `cmake -G Xcode ../`
- Majd fordítjuk a programot:
  - Makefile: `make`
  - MSVC: `msbuild MyProgram.sln`

Build rendszerek – CMake

Kihelyezés

- Megadhatunk egy telepítési könyvtárat, ahova a végső binárisokat át szeretnénk másolni:

```
cmake -DCMAKE_INSTALL_PREFIX=../install ../src
```

- Példa CMakeLists.txt:

```
cmake_minimum_required (VERSION 3.7)
project(MyProgram)


set(SRCS main.cpp foo.cpp foo.h)
add_executable(HelloWorld ${SRCS})
install(TARGETS HelloWorld
  DESTINATION ${CMAKE_INSTALL_PREFIX})
```

- Makefiles: **`make install`**
- MSVC: **`msbuild INSTALL.vcxproj`**

Build rendszerek – CMake

Példa

- Minta projekt: konzolos tételszám generáló alkalmazás
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-cpp
- CMakeLists.txt:

```
cmake_minimum_required (VERSION 3.7)
project(ThesisGenerator)
include_directories(./)
add_executable(ThesisGenerator
    GeneratorModel.cpp
    GeneratorModel.h
    main.cpp)
target_compile_features(ThesisGenerator PUBLIC cxx_std_11)
install(TARGETS ThesisGenerator
    DESTINATION ${CMAKE_INSTALL_PREFIX})
```

Build rendszerek – Qt

Használat

- A Qt saját build rendszere a *Makefile*ra épít.
- Egy Qt projekt létrehozása és fordítása 3 lépésből áll:
  - `qmake -project`: új Qt projekt fájl (.pro fájl) létrehozása. Ezt a projekt fájlt manuálisan vagy IDE-vel szerkesztjük, nyilvántartja a projekt fájljait és konfigurációit.
  - `qmake`: *Makefile* generálása a projekt fájl alapján.
  - `make`: projekt fordítása GNU Make által.
- Ezt a folyamatot egy IDE, pl. a *Qt Creator* tudja gombnyomásra automatizálni.
- Minta projekt: Qt-s tételszám generáló alkalmazás
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-qt

Build rendszerek – .NET

MSBuild / XBuild

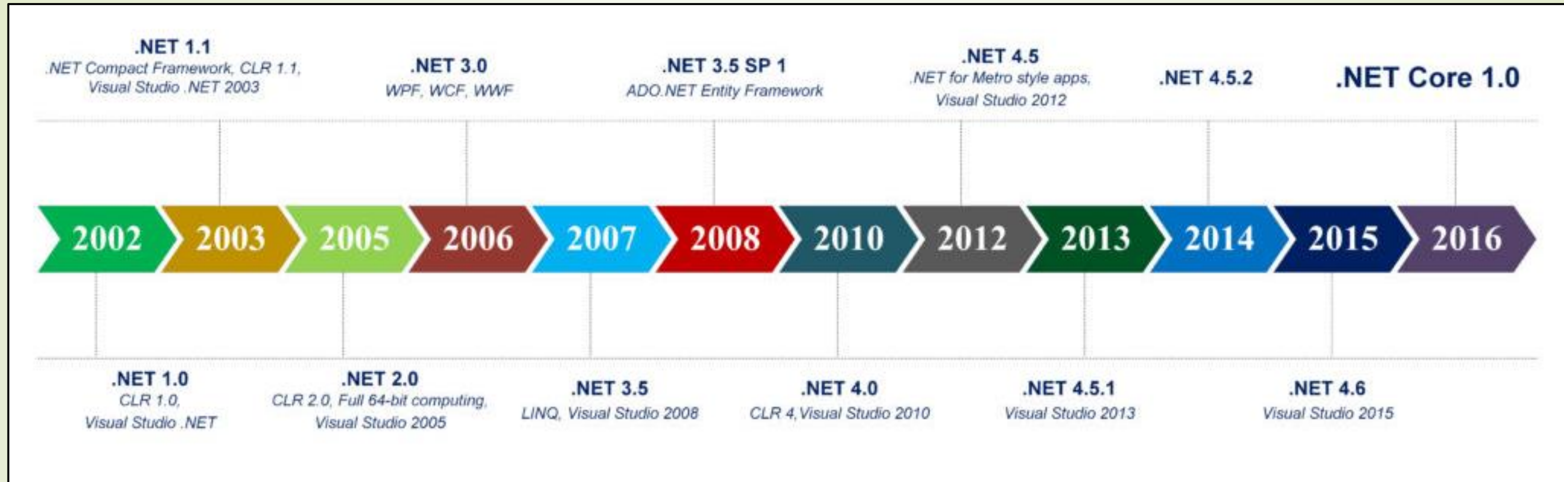
- A Microsoft saját build eszköze Visual Studio projektekre.
  - A *.sln* solution és a *.csproj/.vcxproj* projekt fájlok alapján fordítja le az alkalmazást.
  - A projekt állományok XML formátumúak, a projekthez tartozó fájlok mellett fordítási konfigurációk és egyéb beállítások is megadhatók bennük.
- Például:

```
msbuild SolutionFile.sln /t:Build /p:Configuration=Release
```

  - a *SolutionFile.sln* fájlban adott solutionra
  - a *Build* target végrehajtása
  - a *Release* konfiguráció szerint
- Az MSBuild Mono projekt alatti implementációja az *XBuild*.

Build rendszerek – .NET

A .NET Framework keretrendszer



- *Problémák a .NET Framework keretrendszerrel:*
  - Windows-központú megközelítés
  - Monolitikus, nem megfelelően modularizált felépítés
  - Zárt forráskód

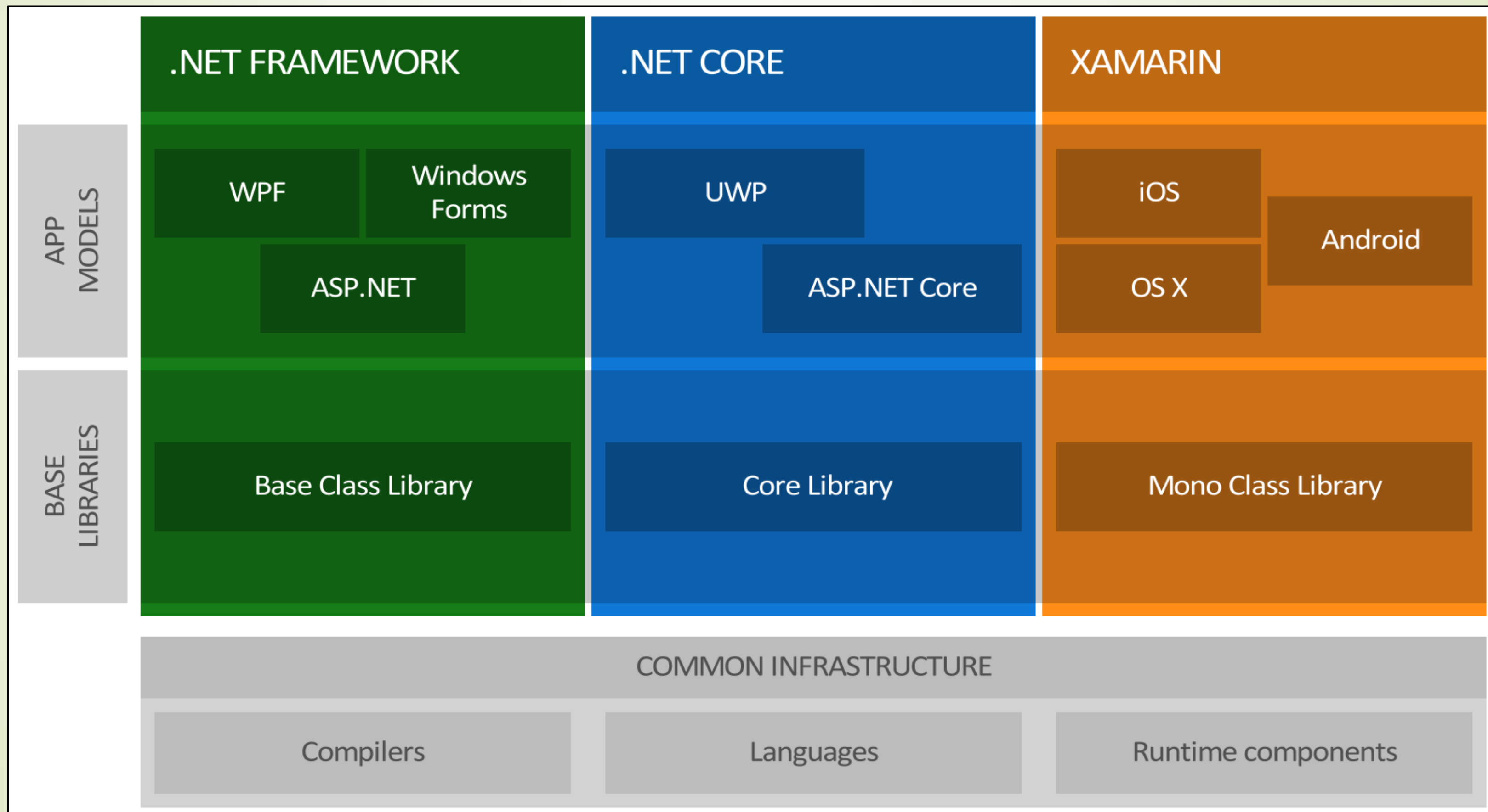
Build rendszerek – .NET

A .NET Core keretrendszer

- *A .NET Core ezekre nyújt megoldást:*
  - Cross-platform (Windows, Linux, macOS)
  - Modularizált felépítés, csak az alkalmazáshoz szükséges komponenseknek kell jelen lennie.
  - Nyílt forráskód (https://github.com/dotnet/core)
- *Érdemes .NET Core-t használni:*
  - Platformfüggetlen és/vagy open source projekteknél
  - Grafikus felülettel nem rendelkező alkalmazások esetében, tipikusan ilyenek a szerveralkalmazások (*.NET Core 3 a grafikus felületre is megoldást nyújt*)
  - Microservicek készítésekor (modularizáltság), konténerek használatakor (pl. Docker)
  - Magas teljesítmény és skálázhatóság esetén (a .NET Core és az ASP.NET Core teljesítménye jelentősen jobb)

Build rendszerek – .NET

A .NET Framework és Core kapcsolata



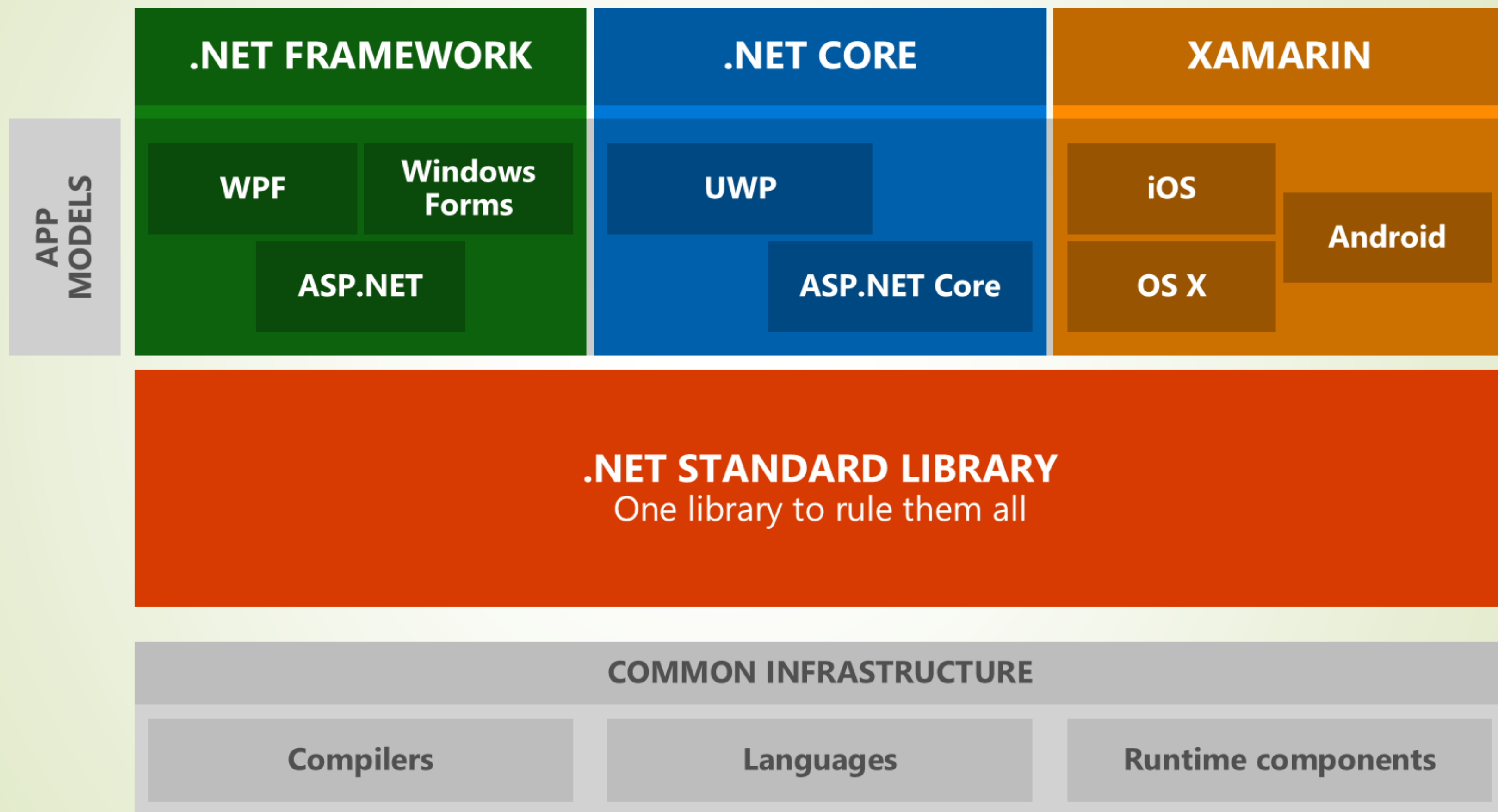
Build rendszerek – .NET

A .NET Standard

- *Felmerülő problémák:*
  - Különböző keretrendszerben írt alkalmazások integrálása
  - Általános felhasználható programkönyvtárak fejlesztése
- *.NET Standard:*
  - Közös, megosztott API az egyes keretrendszer BCL-ek (Base Class Library) felett
  - Felváltja a PCL (Portable Class Library) projekteket

Build rendszerek – .NET

A .NET Standard



Build rendszerek – .NET

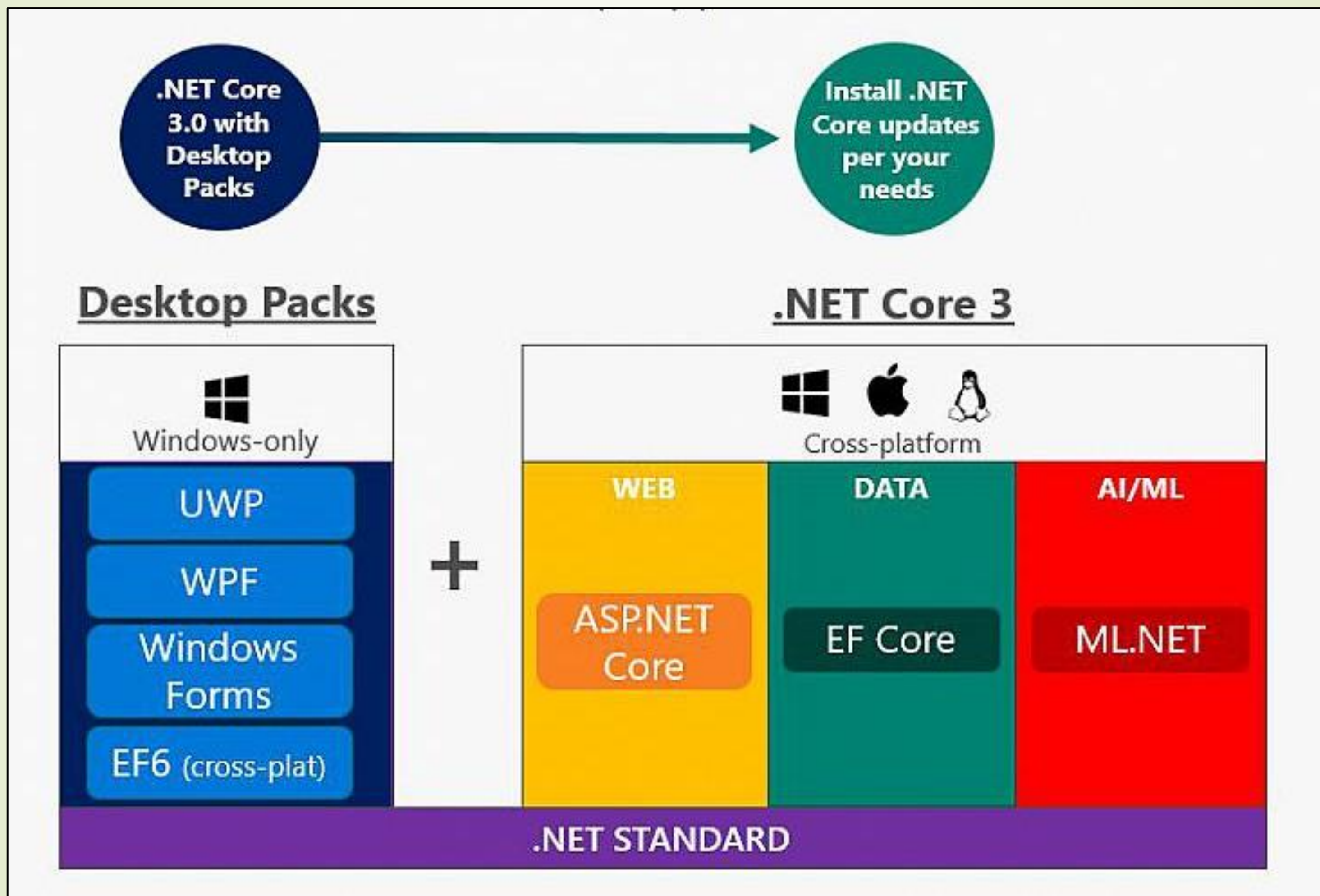
A .NET Standard

.NET Standard	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
.NET Core	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.1	4.6.1	4.6.1	*N/A*
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4
Xamarin.iOS	10.0	10.0	10.0	10.0	10.0	10.0	10.0	10.14	12.16
Xamarin.Mac	3.0	3.0	3.0	3.0	3.0	3.0	3.0	3.8	5.16
Xamarin.Android	7.0	7.0	7.0	7.0	7.0	7.0	7.0	8.0	10.0
Universal Windows Platform	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	*TBD*
Unity	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	2018.1	*TBD*

Forrás: https://docs.microsoft.com/en-us/dotnet/standard/net-standard

Build rendszerek – .NET

Grafikus alkalmazások .NET Core 3 keretrendszerrel

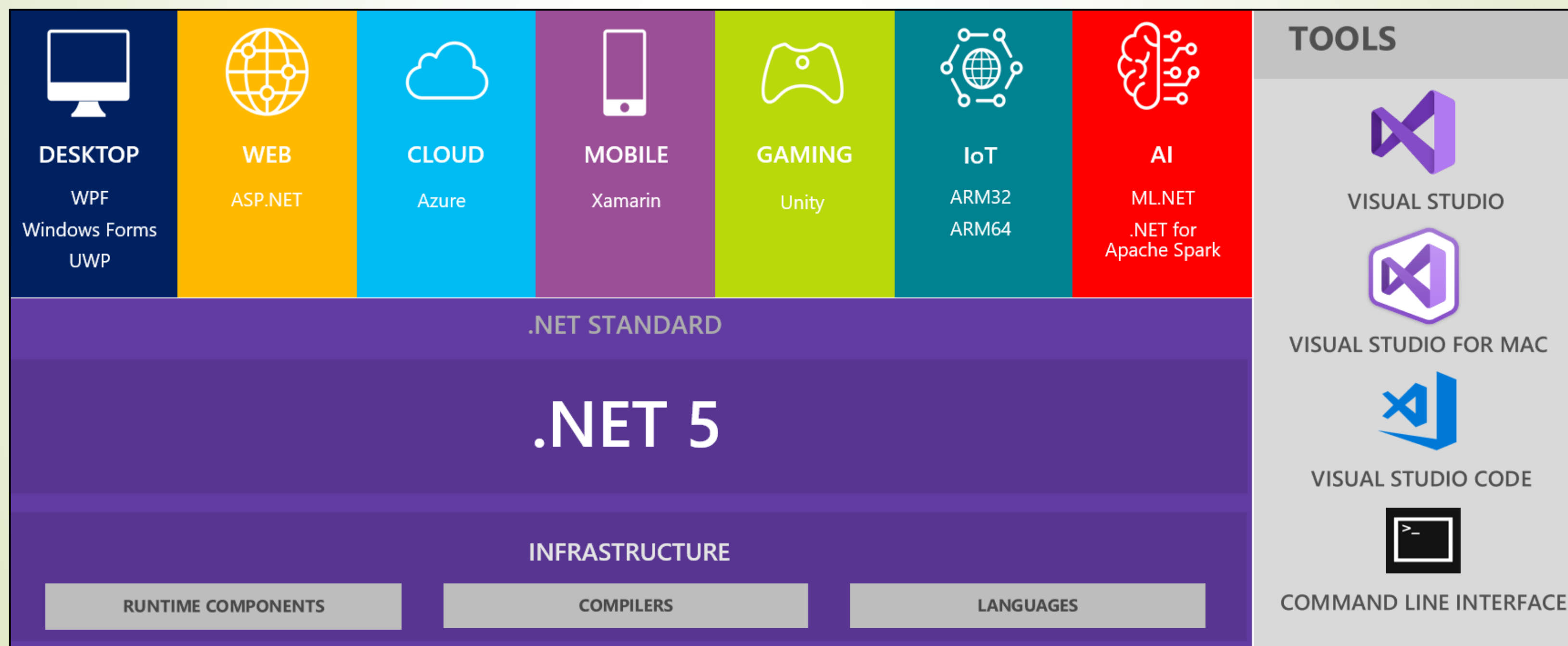


Build rendszerek – .NET

A .NET keretrendszer jövője

- A .NET Framework 4.8 az utolsó verzió
- A .NET Core 3.1 után a .NET Core vNext = .NET 5 következik
  - Tervezett kiadás: 2020. november

Ez mostanra már a .NET világ jelene lett.



Build rendszerek – .NET

Fordítás .NET Core keretrendszer alatt

- A platformfüggetlen .NET Core keretrendszer alatt a *dotnet* konzolos utasítással érhetjük el az SDK-t.
- A projekteket változatlanul solutionökre (.sln fájl) és projektekre (*.csproj*/*.vcxproj* fájlok) osztjuk, amelyek XML formátumúak.
- Telepítés Debian/Ubuntu: `apt-get install dotnet`
- A .NET Core build rendszerének használatára példa:
  - NuGet csomagok visszaállítása: `dotnet nuget restore`
  - Fordítás: `dotnet build path/to/SolutionFile.sln`
    Kurrens könyvtárra: `dotnet build`
  - Fordítás a *Release* konfiguráció szerint:
    `dotnet build --configuration Release`
  - Futtatatás: `dotnet run path/to/SolutionFile.sln`
    Már előállított binárisra: `dotnet path/to/Binary.exe`

Build rendszerek – .NET

Javasolt felépítés a gyakorlati projektekhez

- Model projekt: felületfüggetlen, így egy .NET Standard könyvtárként implementálható. Függőségként használható lesz .NET Core és .NET Framework projektekben is.
- View projekt: specifikus (WinForms vagy WPF) projekt .NET Framework alatt. .NET Core 3 alatt a szükséges *desktop pack* használatával is megvalósítható, de ekkor is platform specifikus (Windows) lesz a termék.

Idén már .NET Core 3.1 (vagy akár .NET 5) használata javasolt a nézethez is.

- Test projekt: .NET Core projekt, így a folyamatos integráció (CI) Linux operációs rendszeren is elvégezhető lesz. Használhatjuk az *MSTest*, az *NUnit* vagy x*Unit* keretrendszert.

- Minta projekt: konzolos tételszám generáló alkalmazás
  - https://szofttech.inf.elte.hu/mate/thesisgenerator-net