Uniwersytet Wrocławski

Biblioteki w C++

Zbigniew Koza
Wydział Fizyki i Astronomii

C++/C to metajęzyki wyspecjalizowane w obsłudze bibliotek

- W czystym C++/C naprawdę niewiele można zrobić (w rozsądnym czasie)
- Popularność tych języków bierze się z łatwości tworzenia w nich i używania bibliotek
- Siła C++/C leży w dostępności wysokiej jakości bibliotek rozwijanych od blisko 50 lat

Dwa wyzwania

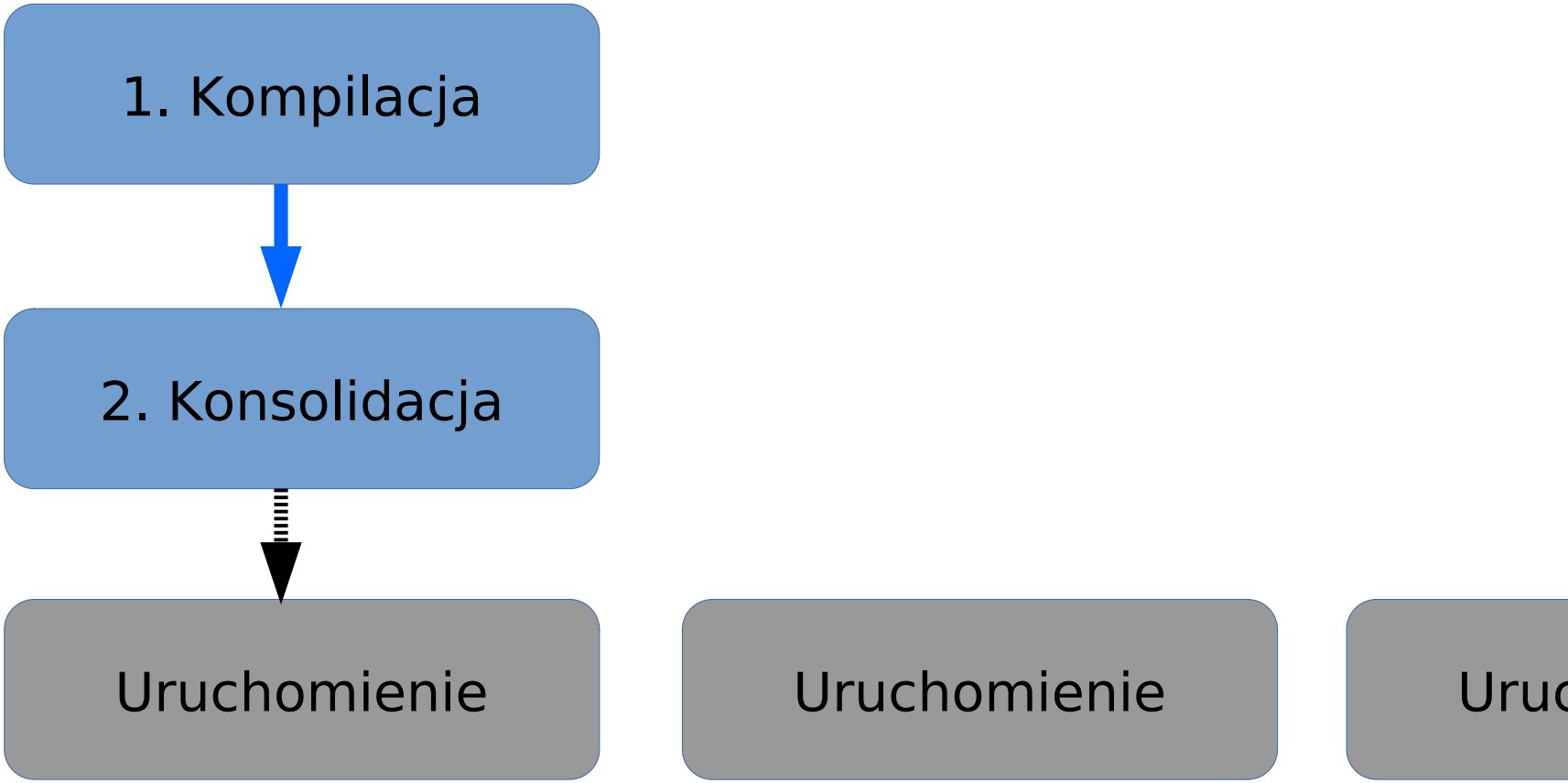
- Jak używać biblioteki?

Poziom operacyjny/ zawodowy

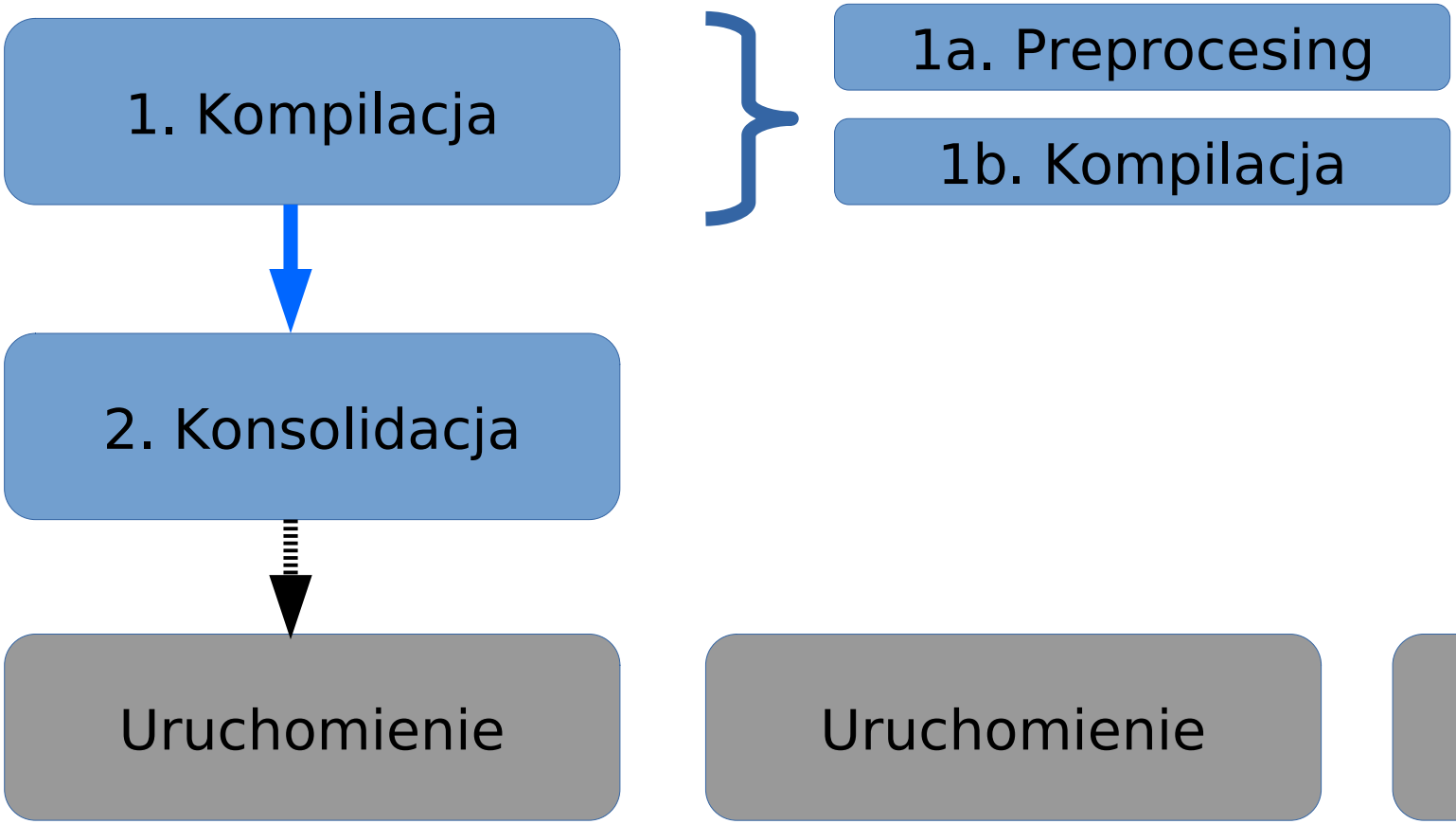
- Jak tworzyć biblioteki?

Poziom ekspercki

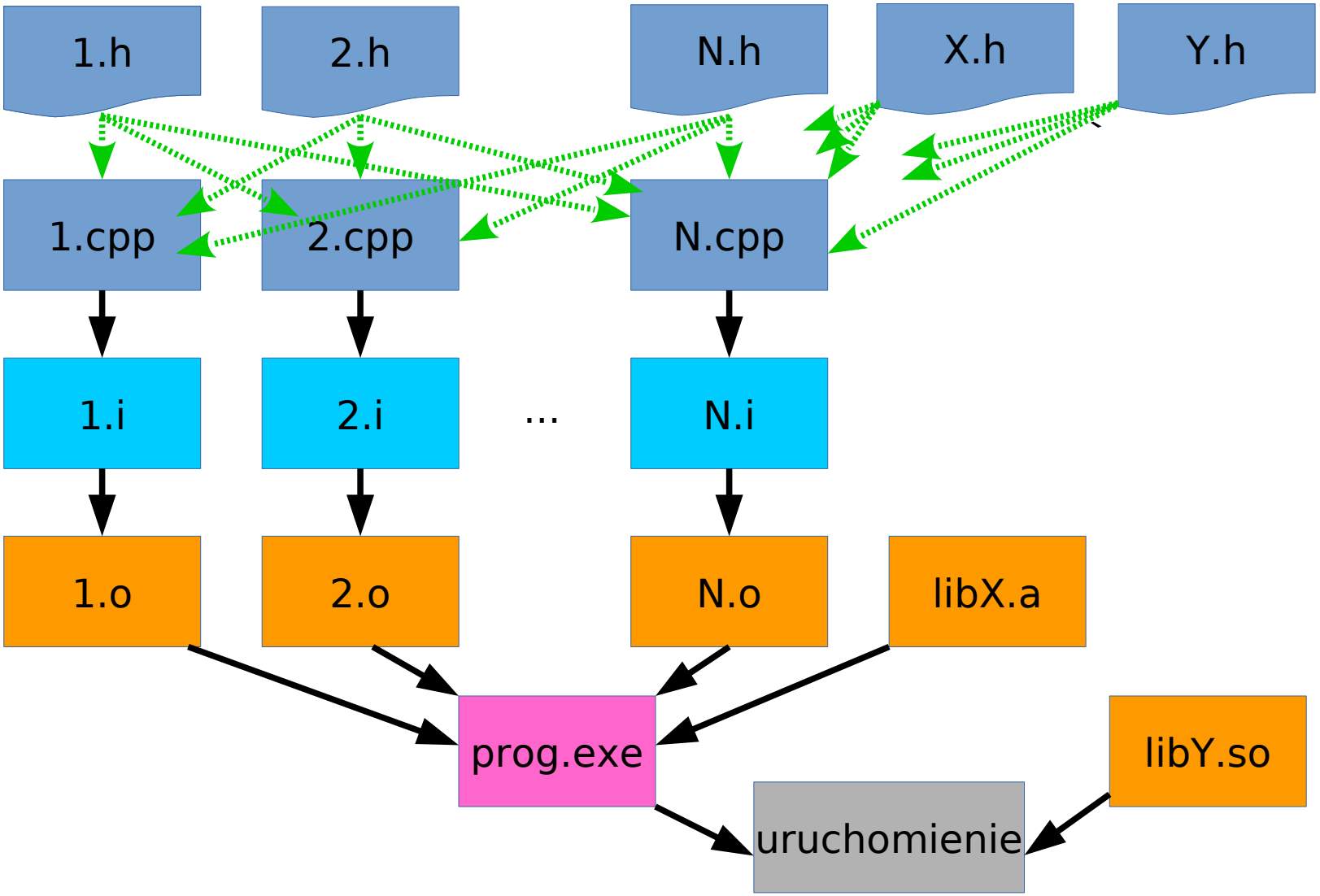
Główne etapy kompilacji programu



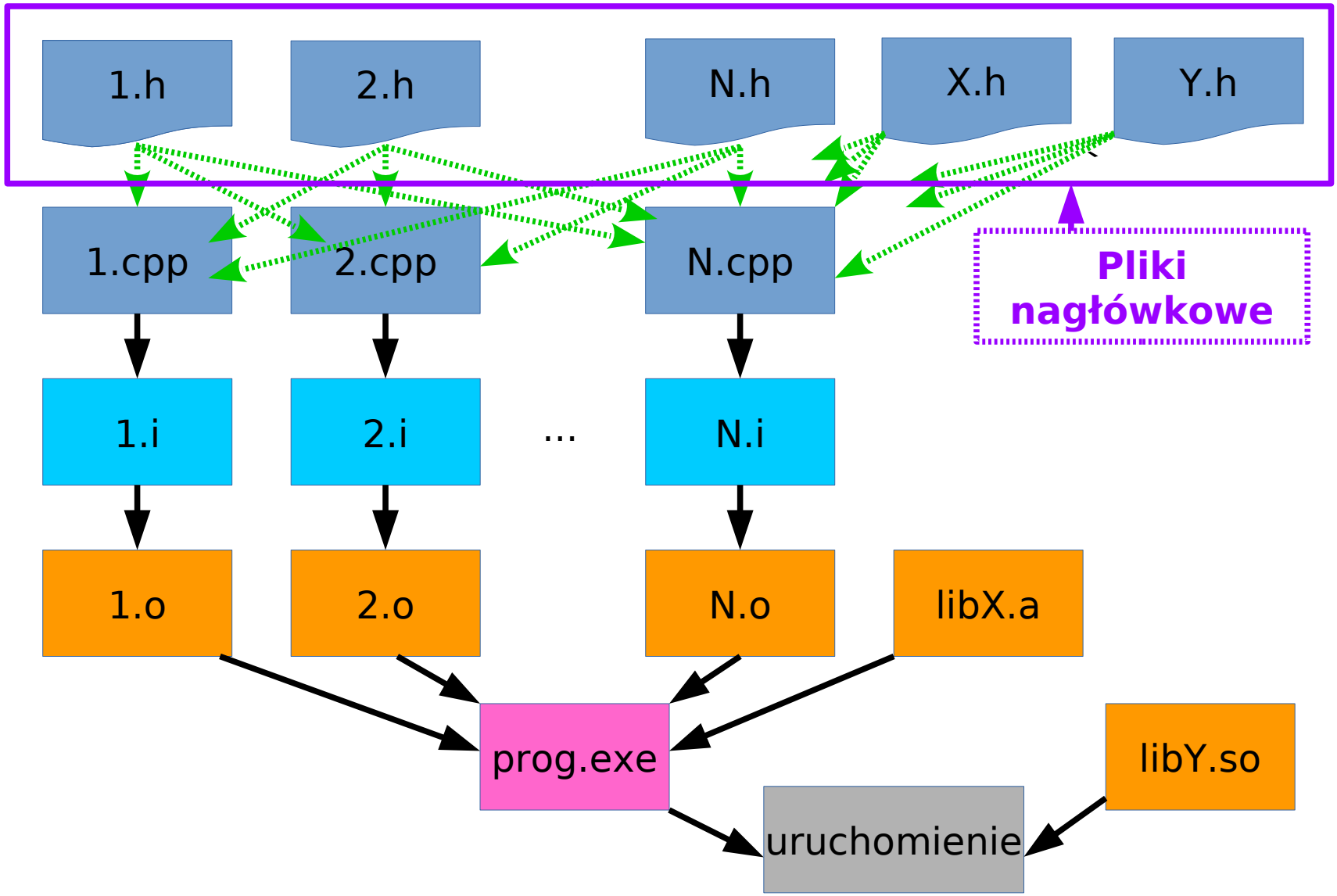
Główne etapy kompilacji programu



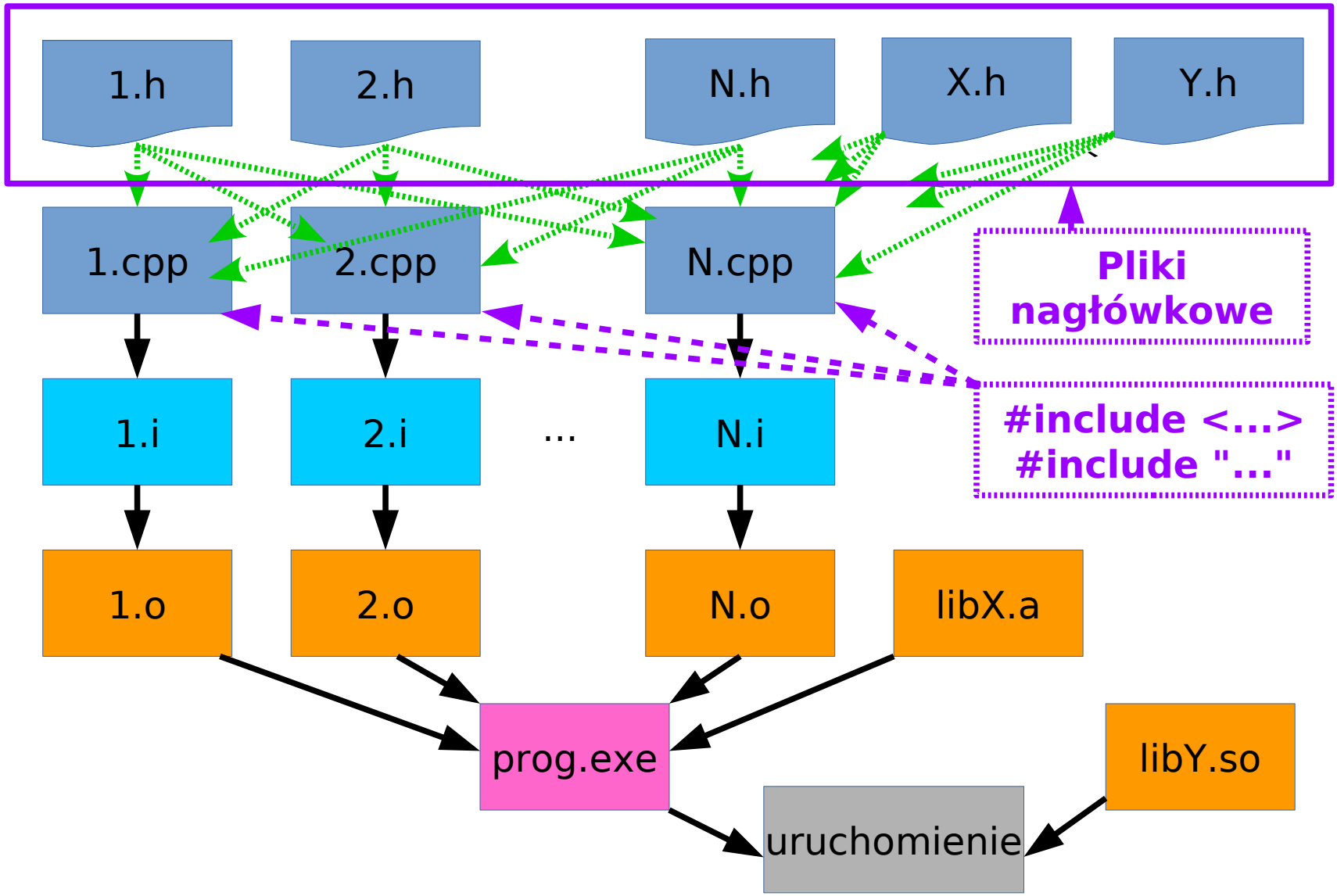
Jak przebiega proces kompilacji?



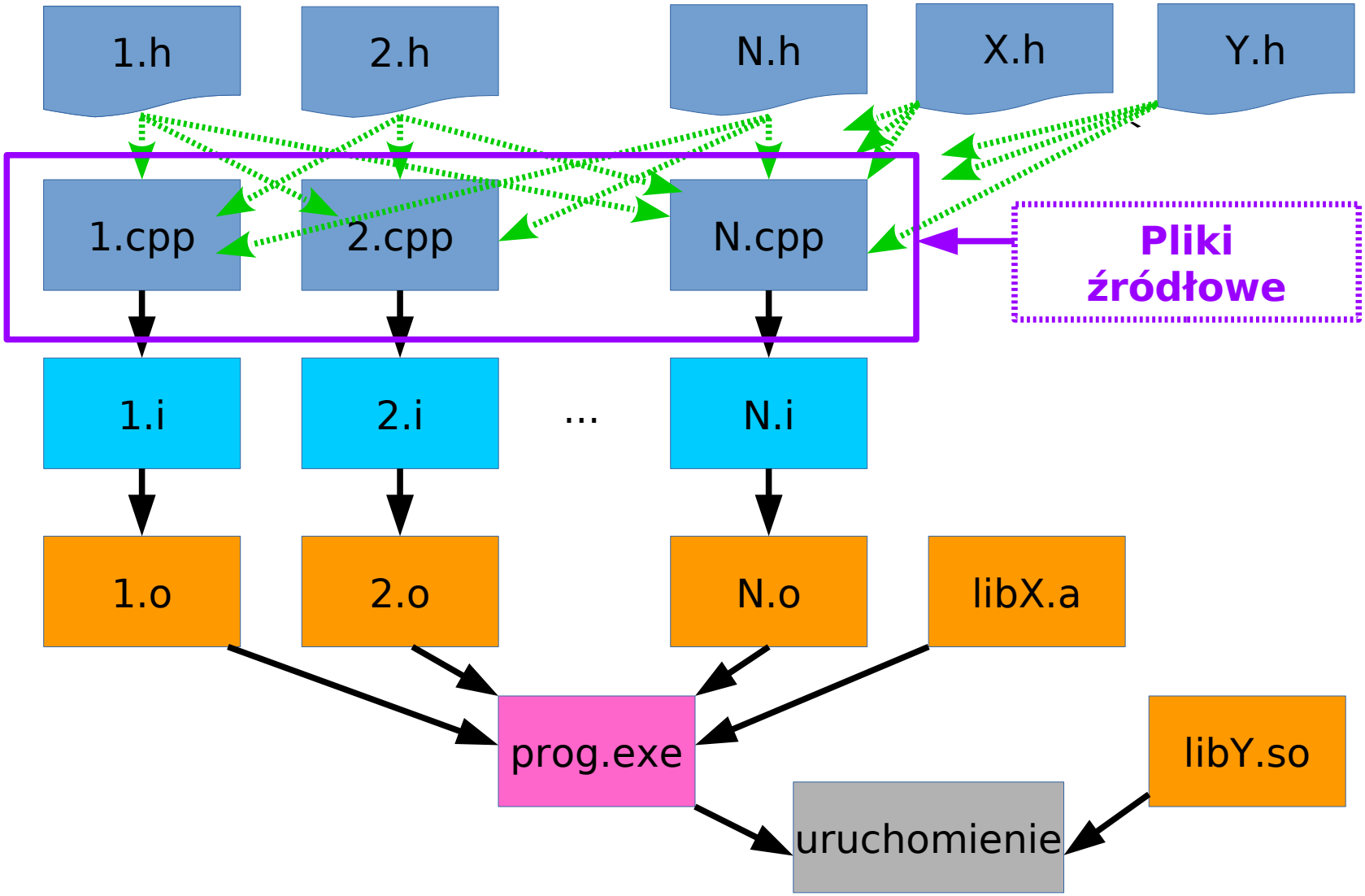
Pliki nagłówkowe



Pliki nagłówkowe są włączane...



Do plików źródłowych



Co i po co jest w plikach nagłówkowych?

- C++ to język z (dość) silną kontrolą typów danych
- Zanim cokolwiek użyjesz, musisz zadeklarować **typ** tego czegoś
- Raz zdefiniowany typ obiektu nie może być zmieniony

  → to pomaga uzyskać efektywny kod i pomaga eliminować błędy

Co i po co jest w plikach nagłówkowych?

- Dlatego zanim użyjemy cokolwiek z biblioteki zewnętrznej, musimy to coś zadeklarować
- Deklaracje umieszcza się właśnie w plikach nagłówkowych
- Pliki nagłówkowe włącza się do programu makrem preprocesora

  #include <…> lub

  #include "plik"

#include

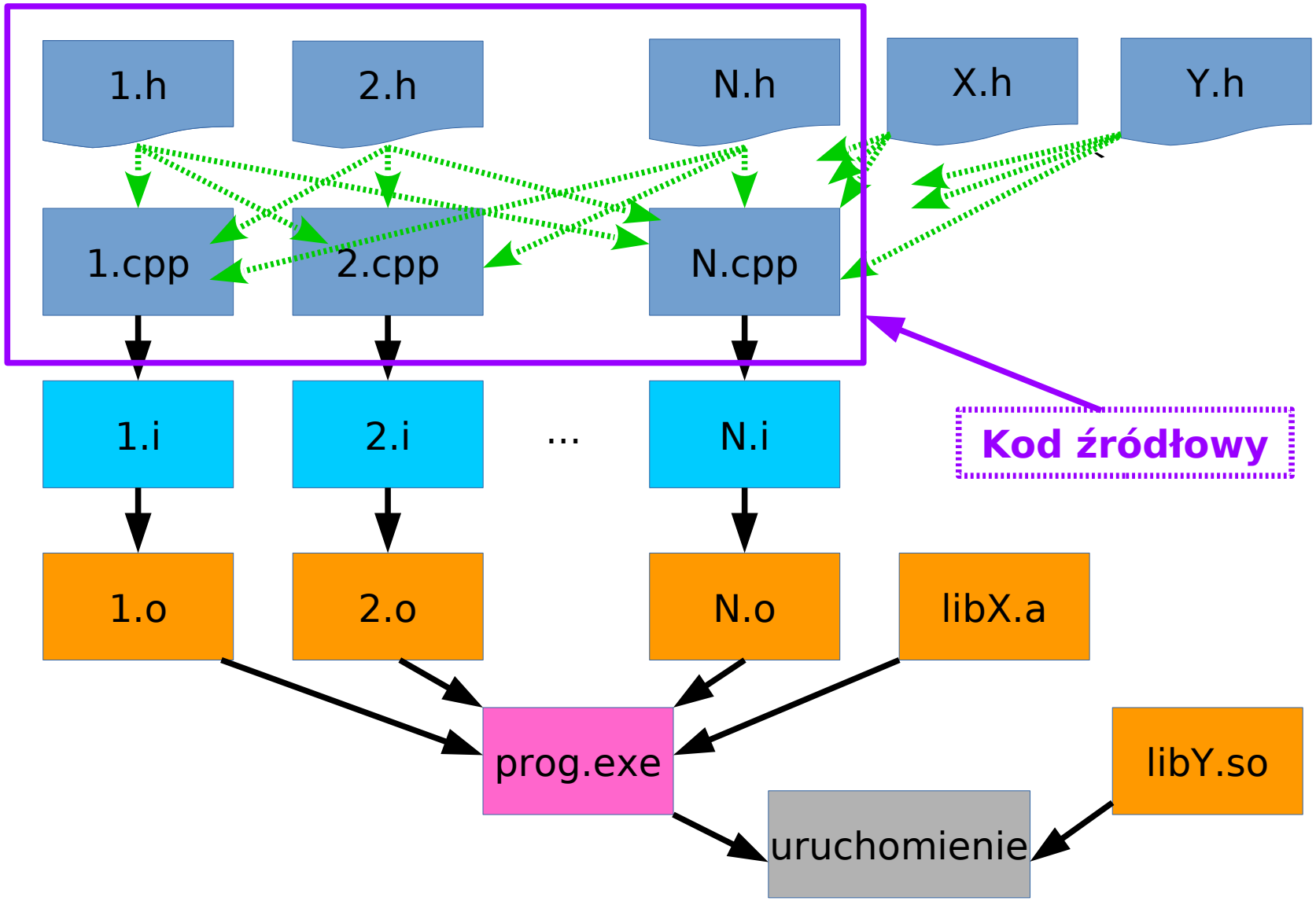
- `#include <cmath>`

  oznacza: w tym pliku korzystam z (zewnętrznej?) biblioteki `cmath`

- `#include "version.h"`

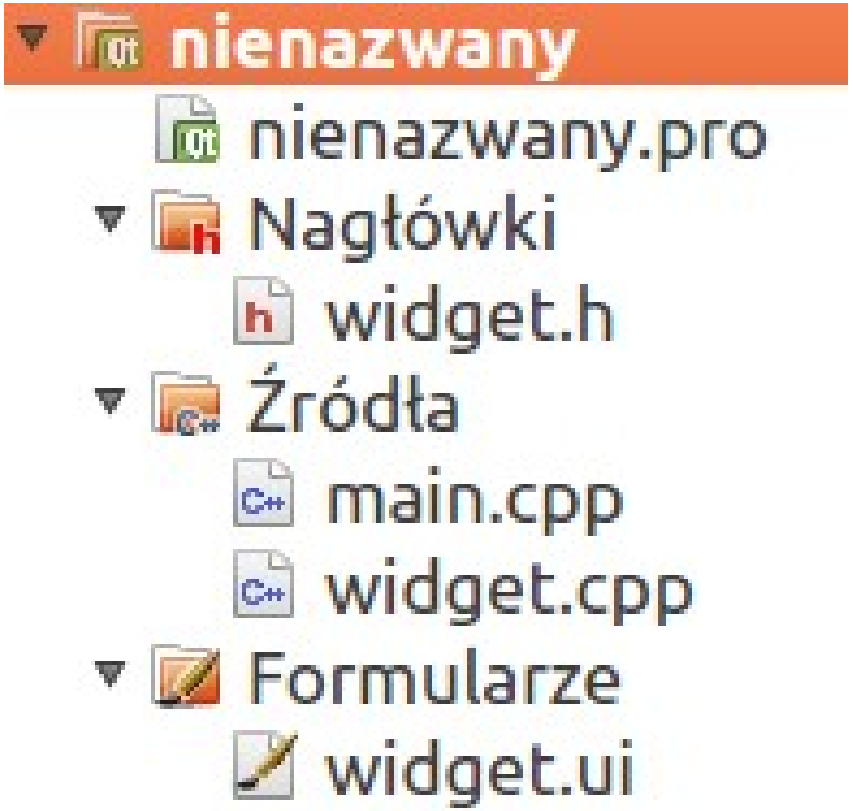
  oznacza: w tym pliku korzystam z (mojej?) biblioteki/modułu `version`

Kod źródłowy

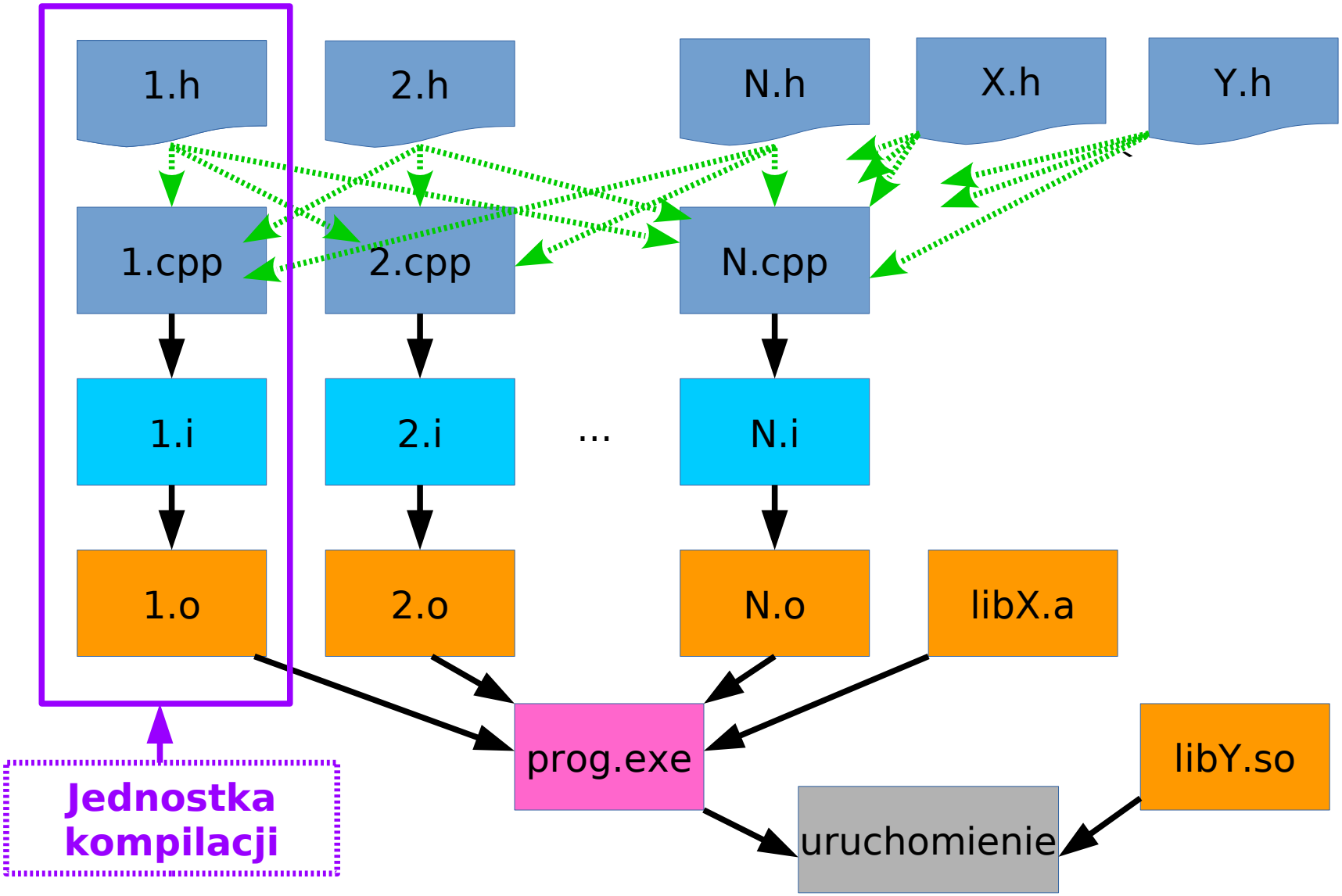


Typowa struktura kodu

- Pliki źródłowe, nagłówkowych, grafikę itp. często umieszcza się w osobnych katalogach



Jednostki kompilacji



Jednostki kompilacji są rozłączne

- Kompilację plików źródłowych można wykonać rozłącznie, niezależnie od siebie
- Jednostka kompilacji = jeden plik *.cpp
- Skoro jednak w plikach źródłowych chcemy korzystać z kodu, zdefiniowanego gdzie indziej, musimy do nich włączyć deklaracje tego zewnętrznego kodu
  → #include <...>

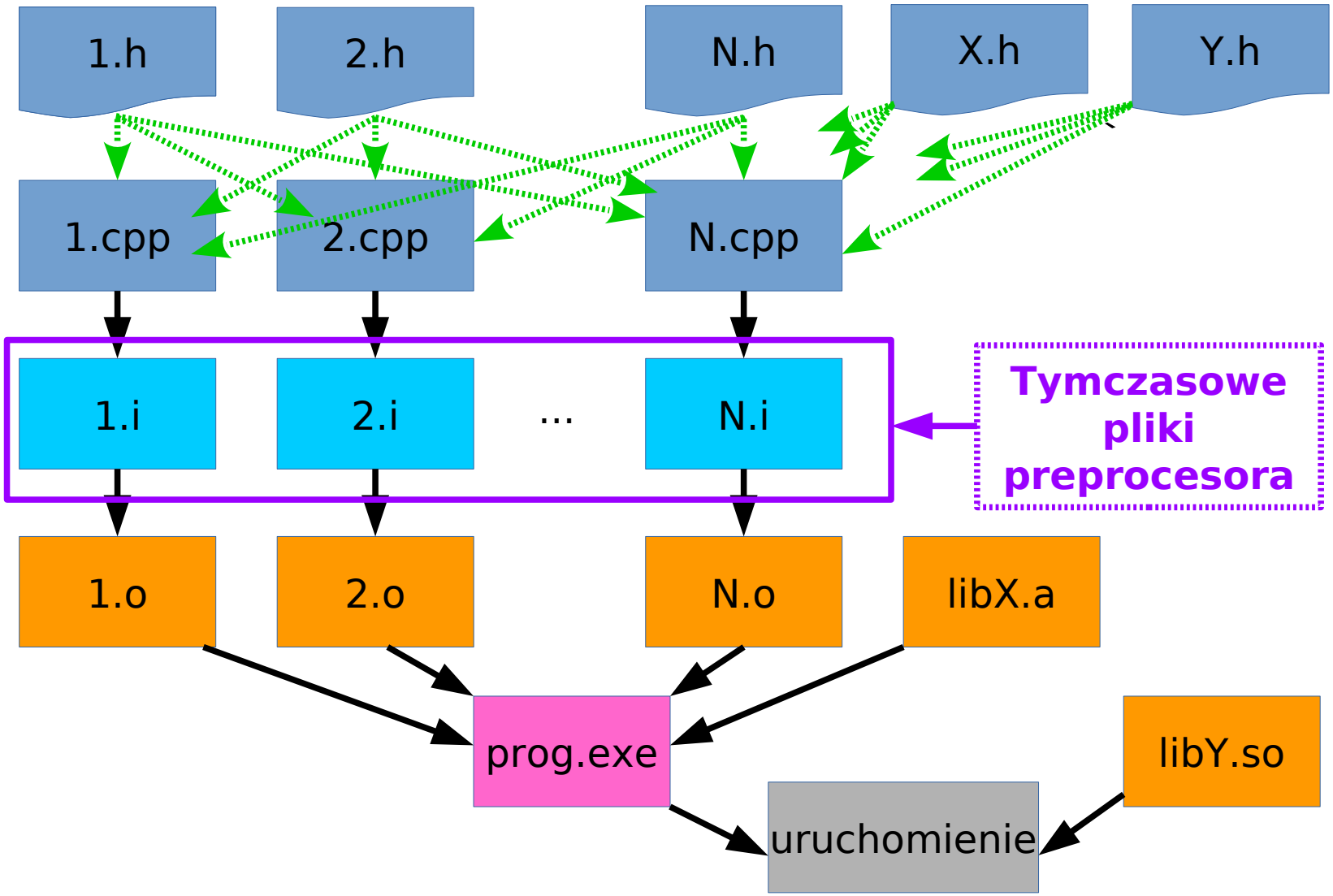
Jednostki kompilacji są niezależne

- Program podzielony na wiele plików można kompilować wieloma różnymi wersjami kompilatora, nawet różnymi kompilatorami, w różnym czasie i rożnych miejscach (jak inaczej tworzyć biblioteki?)
- Kompilator, kompilując plik `1.cpp`, nie zajrzy do treści żadnego pliku, który nie jest włączany do `1.cpp` makrem `#include`

Dwóch ich zawsze jest...

- Praktycznie każdemu plikowi `*.cpp` towarzyszy plik `*.h` (czasem kilka…)
- Częsty wyjątek: `main.cpp`
- Część plików nagłówkowych może nie mieć towarzyszącego im pliku źródłowego (np. prosty `version.h`, ale też całkiem skomplikowane pliki)

Pliki pośrednie preprocesora



-E

```cpp
#include <iostream>

int main()
{
  std::cout << "Witaj, świecie!\n";
}
```

1.cpp

```
g++ -E 1.cpp -o 1.i
wc -l 1.i
```

preprocessing

zliczenie liczby wierszy

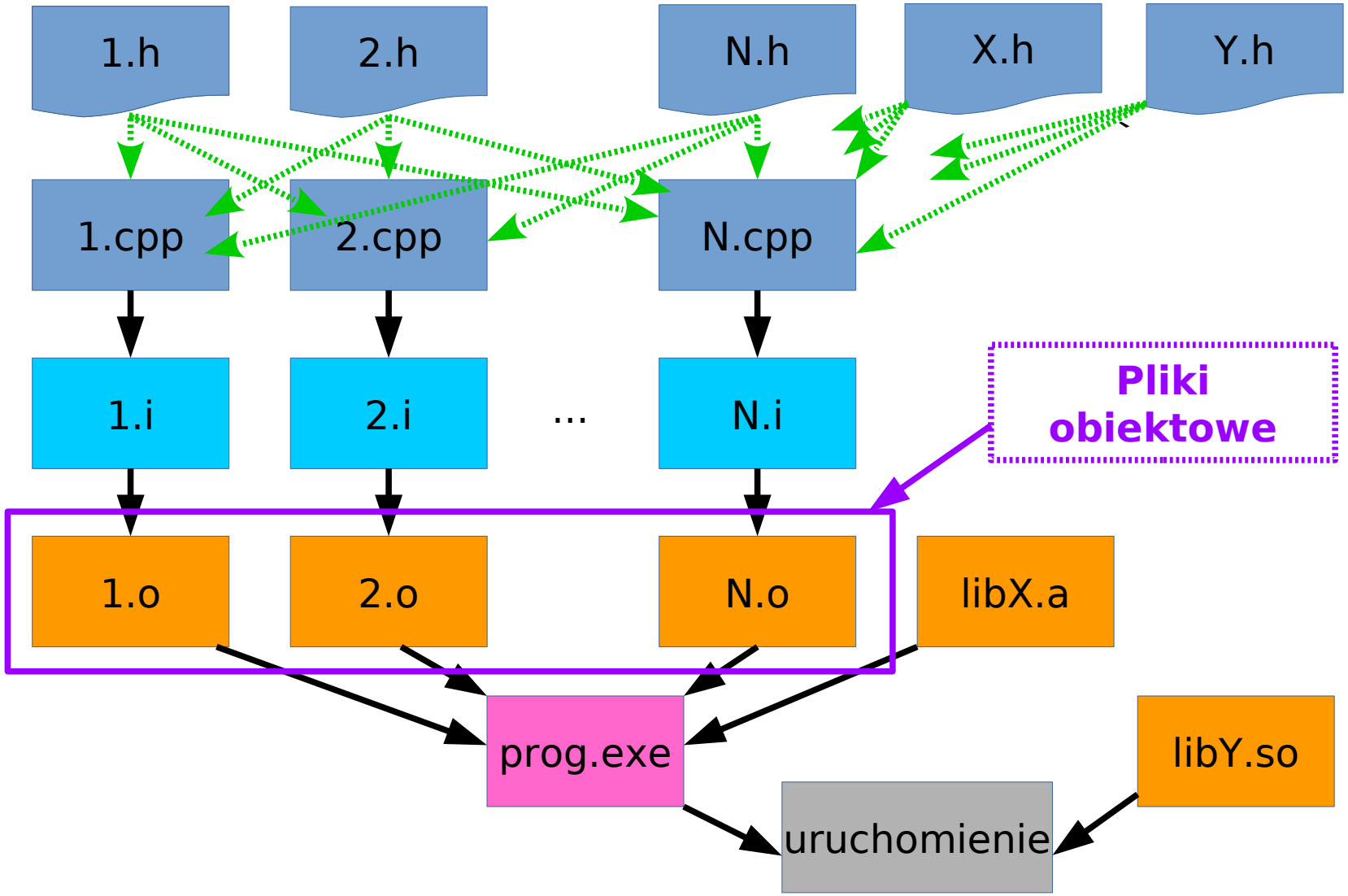
```
26779
```

wynik

Opcja -E

- Opcja -E (kompilator gcc) zatrzymuje kompilację na etapie przetworzenia pliku źródłowego przez kompilator
- Użyj jej, jeśli chcesz sprawdzić, jak interpretowane są w Twoim programie makra preprocesora: co naprawdę na wejściu dostaje kompilator?

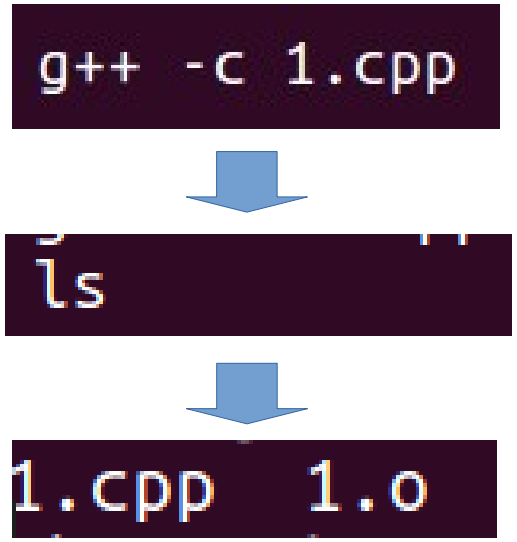
Pliki *.o = skompilowane moduły



*.o, *.obj

- Efektem kompilacji pliku źródłowego w jednostce kompilacji jest plik obiektowy
- Zwykle ma rozszerzenie *.o (linux) lub *.obj (Windows)
- Domyślnie nie jest na trwałe zapisywany na dysku
- Jest to oczywiście plik binarny!!!

Opcja -c

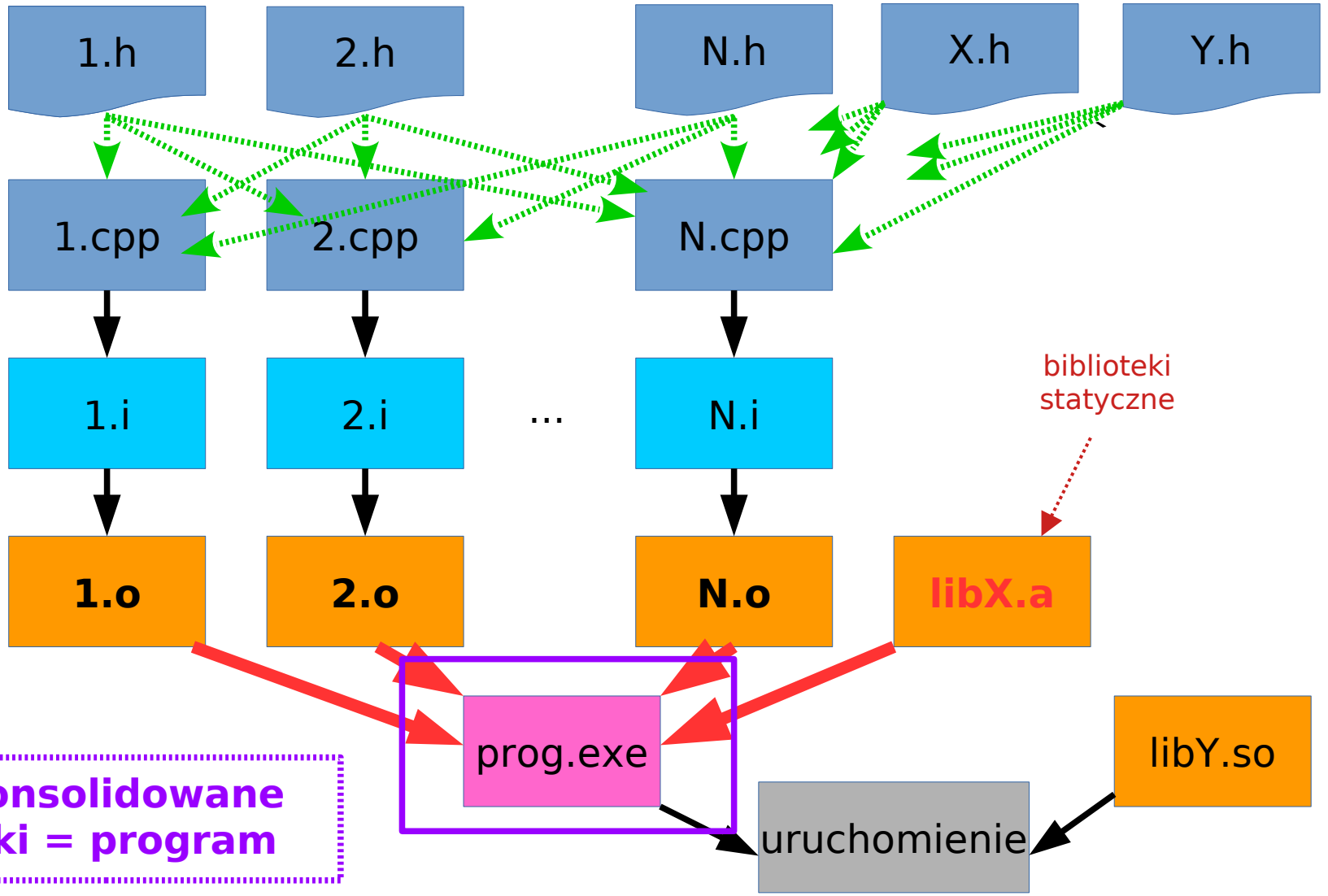


- Opcja -c powoduje przerwanie kompilacji z chwilą wygenerowania pliku obiektowego

Kompilacja zakończona...

- Z formalnego punktu widzenia kompilacja kończy się po skompilowaniu wszystkich jednostek translacji, czyli kompilacji wszystkich plików źródłowych (*.cpp) do obiektowych (*.o)
- Kolejnym etapem jest **konsolidacja programu**

Pliki *.o = skompilowane moduły



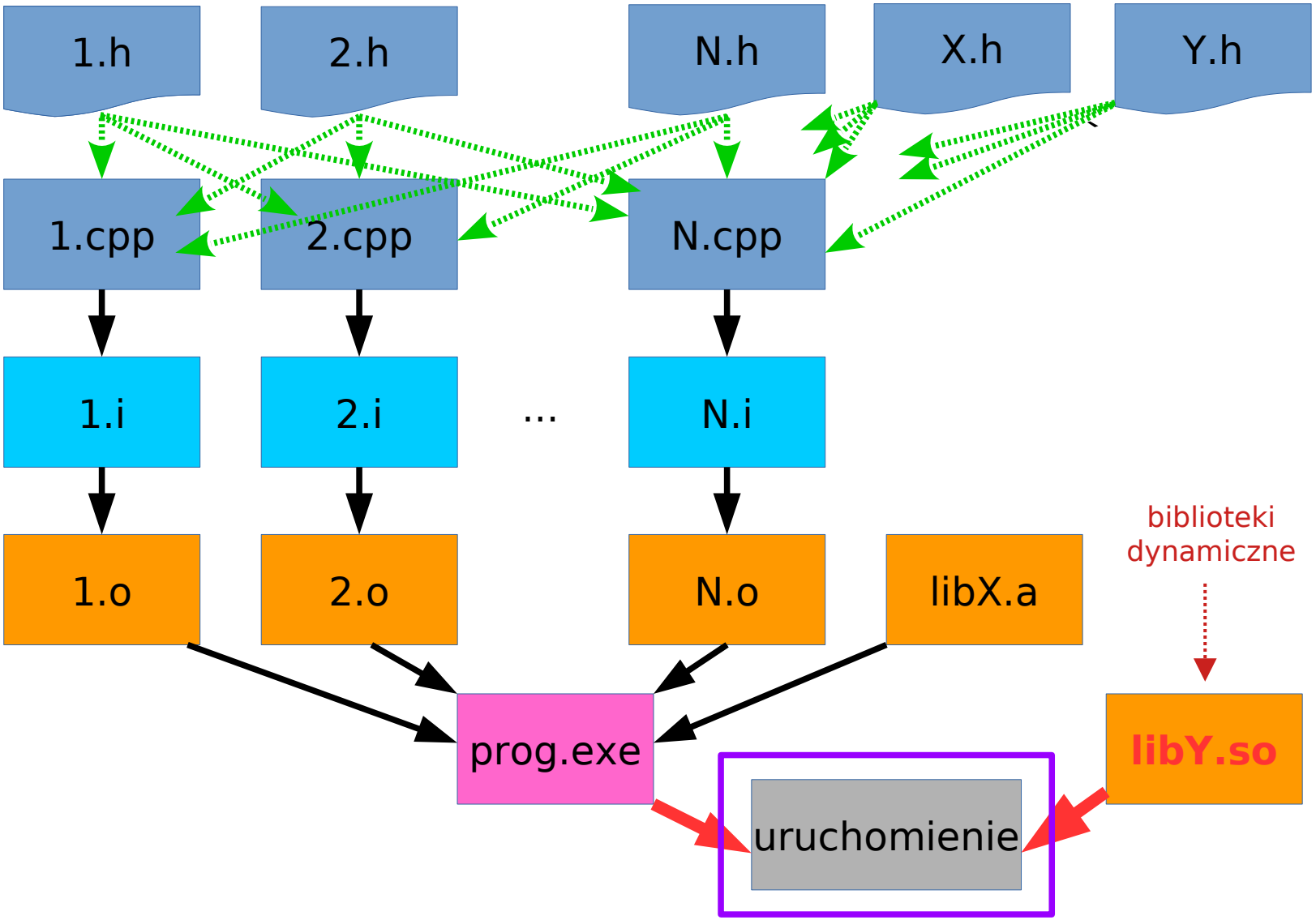
Konsolidacja

- Konsolidację programu przeprowadza konsolidator (ang. *linker*)
- Konsolidacji podlegają pliki obiektowe (*.o) i pliki bibliotek statycznych (*.a)
- Jej efektem jest plik wykonywalny
- Konsolidatora nie powinno obchodzić, w jakich językach napisano kody źródłowe

Konsolidacja jest szybka

- Kompilacja dużych programów może trwać wiele godzin, a ich konsolidacja zwykle nie dłużej niż ok. minuty
- Pozwala to efektywnie rozwijać nawet bardzo duży kod
  - (kompiluje się tylko niedawno zmienione jednostki translacji)

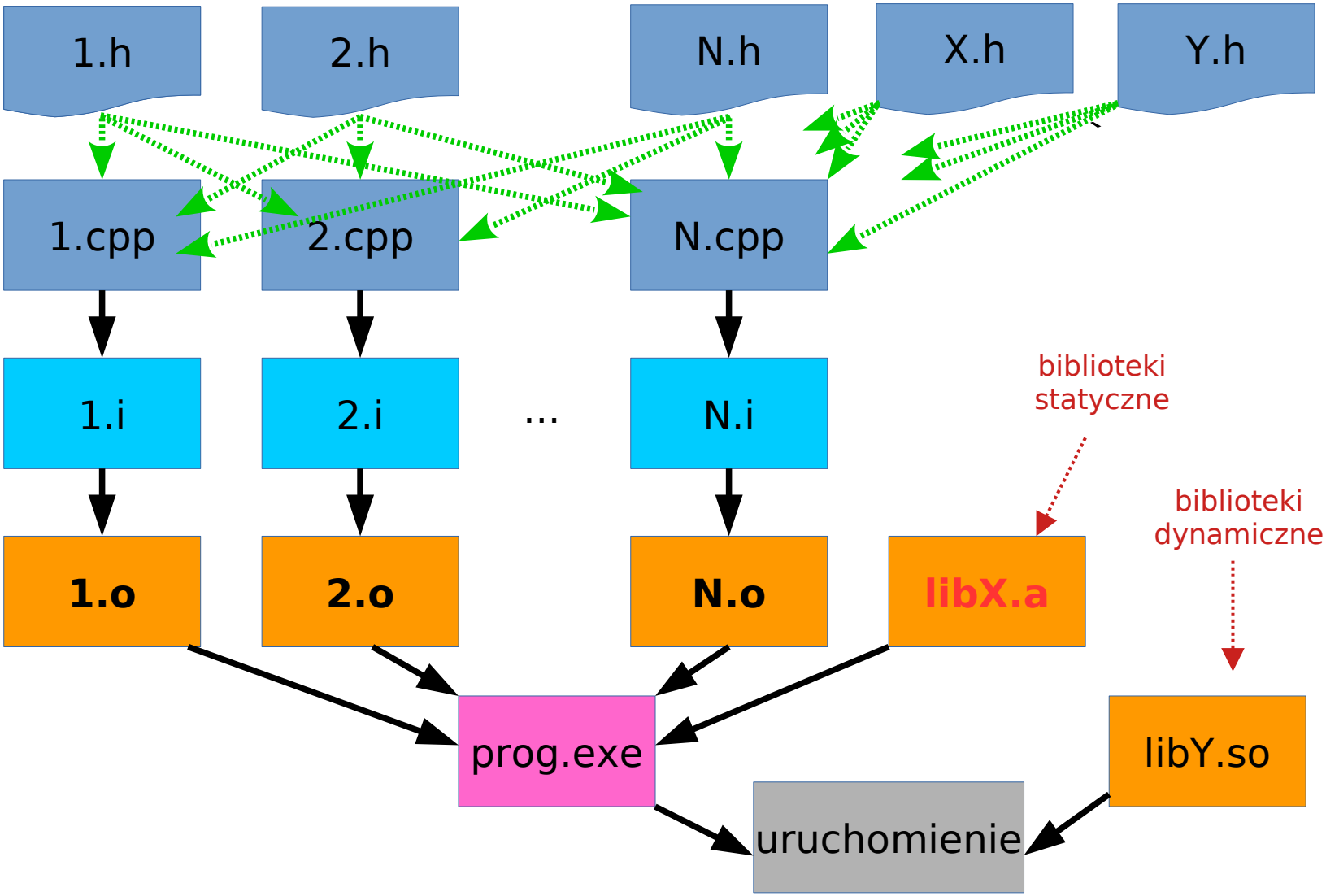
Uruchomienie programu



Program uruchomieniowy

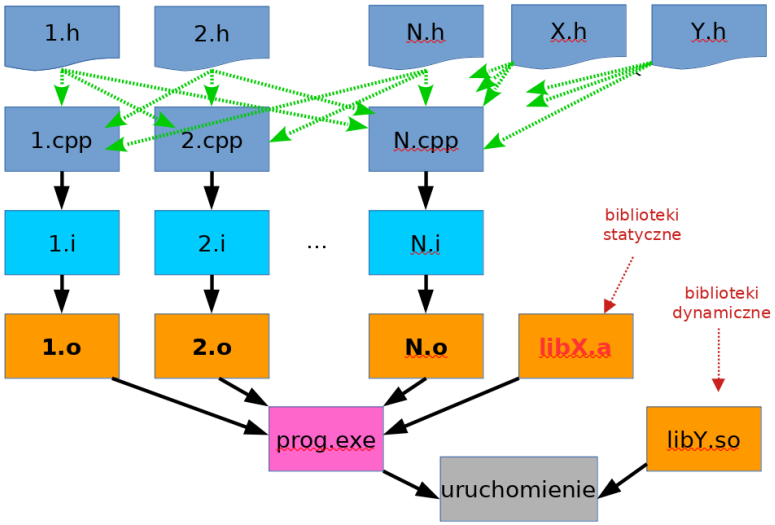
- Nigdy nie wywołałem go bezpośrednio
- Jego celem jest załadowanie obrazu programu do pamięci, połączenie go z bibliotekami współdzielonymi i uruchomienie

Biblioteki statyczne i dynamiczne



Biblioteki statyczne i dynamiczne

- Statyczne są dołączane do kodu programu
  - `*.a` (Linux), `*.obj` (Windows)
- Dynamiczne są dołączane do uruchomionego procesu
  - `*.so` (Linux), `*.dll` (Windows)
  - Ujednolicają „*user experience*”
  - Zmniejszają powielanie kodu (zapotrzebowanie na pamięć)
  - Łatwe aktualizacje



Czego potrzebuje Twój program?

- Jeśli twój program podzielony jest na pliki, to potrzebujesz dostarczyć (niemal) każdemu plikowi źródłowemu jego interfejs z pliku nagłówkowego (`*.h`)

```
// plik moje.cpp
#include "moje.h"
```

Czego potrzebujesz od bibliotek (binarnych)

1) Gdzie są interfejsy jednostek kompilacji (`*.h`)
2) Gdzie są pliki ze skompilowanymi bibliotekami statycznymi (*.a)
3) lub dynamicznymi (*.so)

   Ad 1) `-I`

   Ad 2) `-L`

   Ad 3) `LD_LIBRARY_PATH=...`

Gdzie są interfejsy bibliotek?

std::function

Defined in header <functional>

```
template< class >
class function; /* undefined */
```
(since C++11)

```
template< class R, class... Args >
class function<R(Args...)>;
```
(since C++11)

QApplication Class

The QApplication class manages the GUI a

Header: #include <QApplication>

qmake: QT += widgets

Kompilacja i linkowanie

default location of the `gsl` directory is `/usr/local/include/gsl`. A typical compilation command for a source file `example.c` with the GNU C compiler `gcc` is:

```
$ gcc -Wall -I/usr/local/include -c example.c
```

```
$ gcc -L/usr/local/lib example.o -lgsl -lgslcblas -lm
```

```
$ gcc example.o -lgsl -lcblas -lm
```

Wersje alternatywne

- Możesz wybrać bibliotekę, z którą linkujesz swój program

Plik `.bashrc`

export **LD_LIBRARY_PATH=$LD_LIBRARY_PATH:**/opt/My/lib

- Plik **`.bashrc`** zawiera skrypt inicjalizujący każdą interaktywną sesję powłoki **bash**
- **`LD_LIBRARY_PATH`** jest zmienną środowiskową zawierającą listę niestandardowych katalogów, w których program uruchomieniowy poszukuje potrzebnych mu bibliotek dynamicznych

Biblioteki „czasu kompilacji”

- Szablony C++ umieszcza się w plikach nagłówkowych
- Biblioteki oparte na szablonach nie wymagają kompilacji do plików obiektowych
  - „Biblioteki czasu kompilacji”
  - STL, Boost i wiele innych

Co i gdzie? Język C

- Pliki nagłówkowe: **deklaracje**

```
extern int N;
int f(int n);
struct X;
```

- Pliki źródłowe: **definicje**

```
int N = 100;
int f(int n)
{
    return n*n;
}
struct X{
  int i, j;
};
```

Co i gdzie? Język C++

- Pliki nagłówkowe: **deklaracje + definicje funkcji inline**

```
extern int N;
int f(int n);
struct X;
inline int g() {
  return 10;
}
```

- Pliki źródłowe: **definicje**

```
int N = 100;
int f(int n)
{
   return n*n;
}
struct X{
  int i, j;
};
```

ODR = *One definition rule*

- In any *translation unit*, a template, type, function, or object can have **no more than one definition**. Some of these can have any number of declarations. **A definition provides an instance.**
- In the *entire program*, an object or non-inline function cannot have more than one definition; **if an object or function is used, it must have exactly one definition**. You can declare an object or function that is never used, in which case you don't have to provide a definition. In no event can there be more than one definition.
- Some things, like **types, templates, and extern inline functions, can be defined in more than one translation unit.** For a given entity, each definition must have the same sequence of tokens. **Non-extern** objects and functions in different translation units are different entities, even if their names and types are the same.

Pliki nagłówkowe

https://en.wikipedia.org/wiki/One_Definition_Rule

Biblioteki statyczne

- `ar cr libfajne.a 1.o 2.o`

Tworzenie (z plików *.o)

- `g++ main.o -l fajne -L.`

Użycie

Automatyzacja kompilacji

- make + Makefile
  - cmake / ccmake
  - qmake (Qt)
- ninja
- GNU autotools
  - ./configure

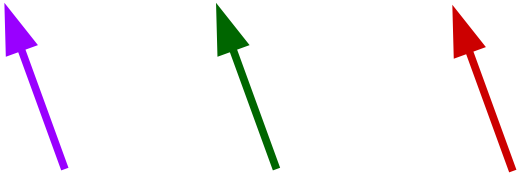
- Kompilują tylko te jednostki translacji, które są niezbędne
- Mogą automatycznie dostosować kompilację do platformy, na którą kompilujesz

Biblioteki statyczne

- `ar cr libfajne.a 1.o 2.o`

Tworzenie (z plików *.o)

- `g++ main.o -l fajne -L.`



Użycie Co? Gdzie szukać libfajne.a?

Biblioteki dynamiczne

- ```
  g++ -fPIC -c 1.cpp
  g++ -fPIC -c 2.cpp
  …
  g++ -shared -o libfajne.so 1.o 2.o
  ```

- ```
  g++ main.o -l fajne -L.
  ```

Uruchomienie programu

```
export LD_LIBRARY_PATH
/path/to/library:${LD_LIBRARY_PATH}
```

(lub **setenv**…)

> ./a.out

Automatyzacja

Dlaczego automatyzacja?

- Google Chrome: 30 000 plików źródłowych
- Gdyby kompilacja każdego z nich trwała tylko sekundę, to kompilacja całego programu trwałaby
  30 000/3600 = 8 godzin i 20 minut
- A przecież chcielibyśmy kompilować rozwijane przez siebie programy
  nawet co kilka minut

Prosty Makefile

```
a.exe: main.o my_sin.o my_cos.o
	g++ main.o my_sin.o my_cos.o -o a.exe
main.o: main.cpp
	g++ main.cpp -c
my_sin.o: my_sin.cpp my_sin.h
	g++ my_sin.cpp -c
my_cos.o: my_cos.cpp my_cos.h
	g++ my_cos.cpp -c -O2
clean:
	rm -f a.exe *.o
```

Uruchomienie make

```
> make
g++ main.cpp -c
g++ my_sin.cpp -c
g++ my_cos.cpp -c -O2
g++ main.o my_sin.o my_cos.o -o a.exe
```

Prosty CMakeLists.txt

```
cmake_minimum_required(VERSION 2.8)

project(pierwszy)
add_executable(${PROJECT_NAME} "main.cpp")
```

Bardziej złożony CMakeLists.txt

```
cmake_minimum_required(VERSION 3.10)

set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

project(goooo)
add_executable(${PROJECT_NAME} main.cpp)
add_executable(histogram  histogram.cpp)
add_executable(test_histogram tests/test1.cpp)

find_package(GSL REQUIRED)
target_link_libraries(histogram GSL::gsl GSL::gslcblas)

find_package(GTest REQUIRED)

target_include_directories(tests_histogram PRIVATE ${GTEST_INCLUDE_DIRS})
target_link_libraries(tests_histogram GTest::GTest GTest::Main)
```

Uruchomienie `cmake`

```
> cmake .
> make
```

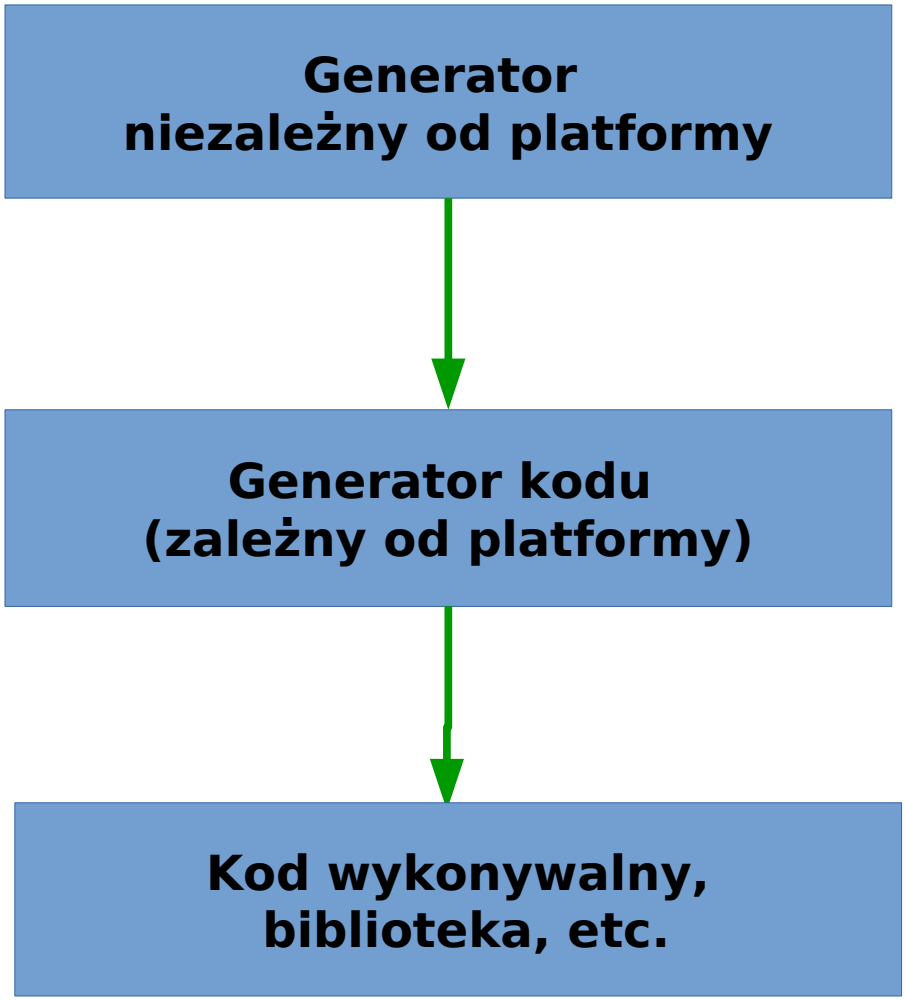
Prosty build.ninja

- Prosty plik `build.ninja` nie istnieje
- Użyj generatora, np.
  `cmake -G ninja`

Uruchomienie ninja

```
> cmake -G ninja ../my_src_dir
> ninja
```

Podsumowanie



- autotools
- cmake
- qmake
- meson
- …

- make
- ninja
- xcode
- MSBuild
- …