

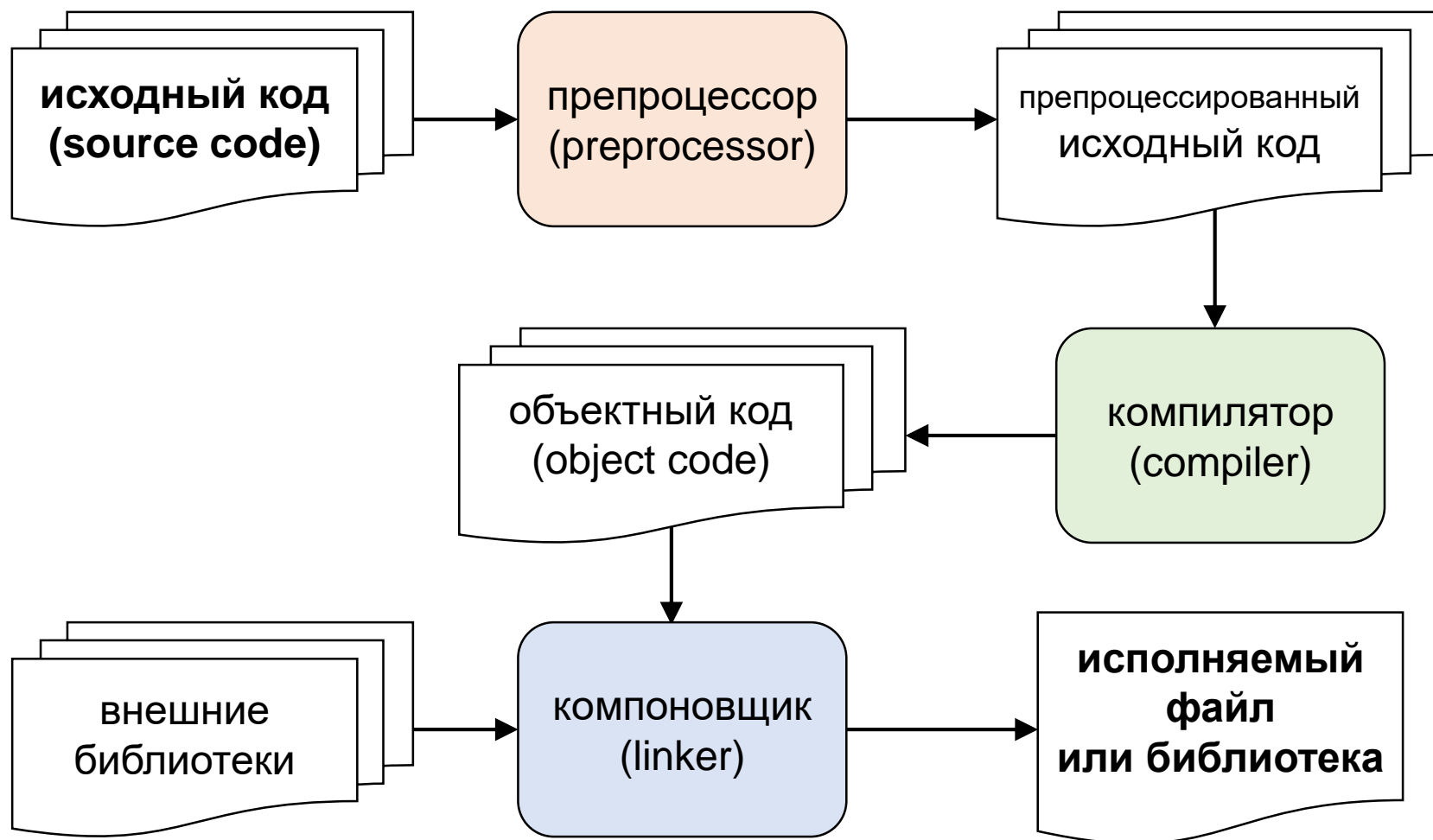
Процесс трансляции и системы сборки программ

Курс «Технология программирования»

Кафедра управления и информатики НИУ «МЭИ»

Осень 2016 г.

Сборка программы (build)







Сборка вручную

- Компиляция (и препроцессирование):

- Препроцессировать отдельным шагом можно.





- `g++ -c -std=c++14 main.cpp -o main.o`

- `g++ -c -std=c++14 input.cpp -o input.o`

			
ВЫЗОВ компилятора	флаги компиляции	ВХОД: исходный код	ВЫХОД: объектный код

- Компоновка:

- `g++ -static main.o input.o -o program.exe`

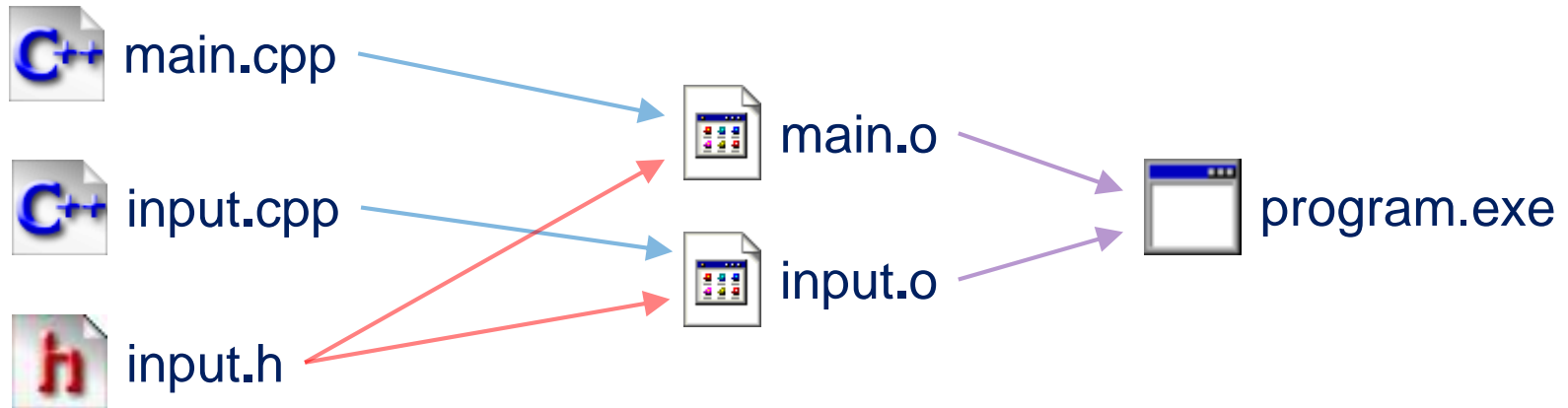
			
ВЫЗОВ КОМПОНОВЩИКА	флаги КОМПОНОВКИ	ВХОД: объектный код	ВЫХОД: ИСПОЛНЯЕМЫЙ КОД

Системы сборки программ

- Причины использования:
 - Сборка на build-серверах, где нет IDE.
 - Переносимость:
 - можно собрать программу без IDE;
 - по конфигурации сборки можно создавать проекты для разных IDE.
 - Сложные сценарии и схемы сборки.
- Задачи систем сборки:
 - описание и решение типовых задач готовыми блоками;
 - полностью автоматическая работа;
 - ускорение сборки за счет обработки только изменившихся файлов.

Программа **make**

- Описывается, **из каких файлов** и **как именно** получить **каждый файл**.
- **Make** проверяет, какие файлы изменились или отсутствуют, и *перерабатывает только необходимую работу*.



Структура Makefile

- Запуск `make`:

- › `make` конфигурация в `Makefile`, цель `all`
- › `make цель` альтернативная цель
- › `make -f config.mk` альтернативное имя для `Makefile`

- Элементы `Makefile`:

Комментарии в любом месте файла.

- Присваивание значений переменным.
 - `EXE_NAME = program.exe`
 - `EXE_PATH = bin/$(EXE_NAME) # bin/program.exe`
 - Доступны действия со строками, арифметика, простые условия.
- Список правил.

Структура правила в Makefile

цель : требование-1 ... требование-N
 действие-1
 действие-M

- **Цель** — что правило позволяет получить, т. е. какой файл будет создан в результате.
 - Либо название задачи, которое указывается в `make цель`.
- **Требования (зависимости)** — какие цели должны быть уже достигнуты, т. е. какие файлы должны иметься.
- **Действия** — команды на запуск программ, которые нужно выполнить для достижения цели.
- Пример: `program.exe : source1.cpp source2.cpp`
 `g++ -o program.exe source1.cpp source2.cpp`

Пример Makefile для ЛР № 1

all: program.exe

program.exe: main.o input.o

g++ main.o input.o -o program.exe

main.o: main.cpp input.h

g++ main.cpp -o main.o

input.o: input.cpp input.h

g++ input.cpp -o input.o

	✗
Makefile	
main.cpp	
input.cpp	
input.h	
main.o	
input.o	
program.exe	

Особенности формата **Makefile**

- При вызове **make** без аргументов целью считается **all**.
- Пустые строки между правилами значимы.
- Отступы для действий — табуляции (не пробелы!):

```
program.exe: main.cpp  
    ──────────▶ g++ main.cpp -o program.exe
```
- Есть несколько версий **make** (**mk**, **gmake**, **nmake**...), которые немного различаются.
 - На ЛР будет GNU Make (**gmake**).
 - Кроме **mk**, программы называются одинаково — **make**.
- Цели без зависимостей с ключевым словом **.PHONY** (см.)

Расширенные средства **make**

- Специальные переменные (приведены некоторые):

`$@` название цели

`$<` первое требование

`$?` все требования через пробел

- Шаблонные правила (пример):

- Для получения `X.o` нужно скомпилировать `X.cpp`:

```
%.o: %.cpp
```

```
g++ -c $< -o $@
```

- Запуск четырех процессов **make** параллельно (ускорение):

```
make -j4
```

- Генерация файла `*.d`, описывающего зависимости от `#include` (GCC облегчает написание **Makefile**):

- `g++ -c -MMD main.cpp -o main.o`

Пример Makefile (2)

PROGRAM = program.exe

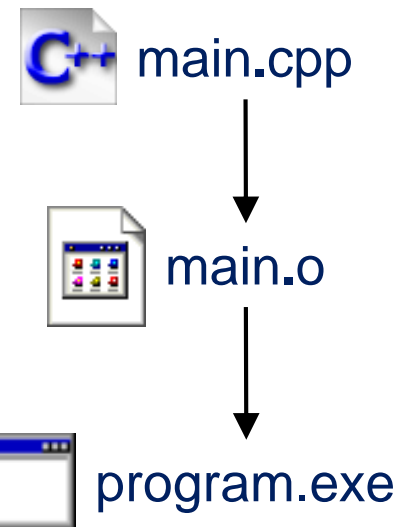
.PHONY: all clean

all: \$(PROGRAM) # all: program.exe

program.exe: main.o
g++ main.o -o \$(PROGRAM)

main.o: main.cpp
g++ -c main.cpp

clean:
del /Q main.o
del /Q \$(PROGRAM)



Командная строка

Сборка программы:

> make

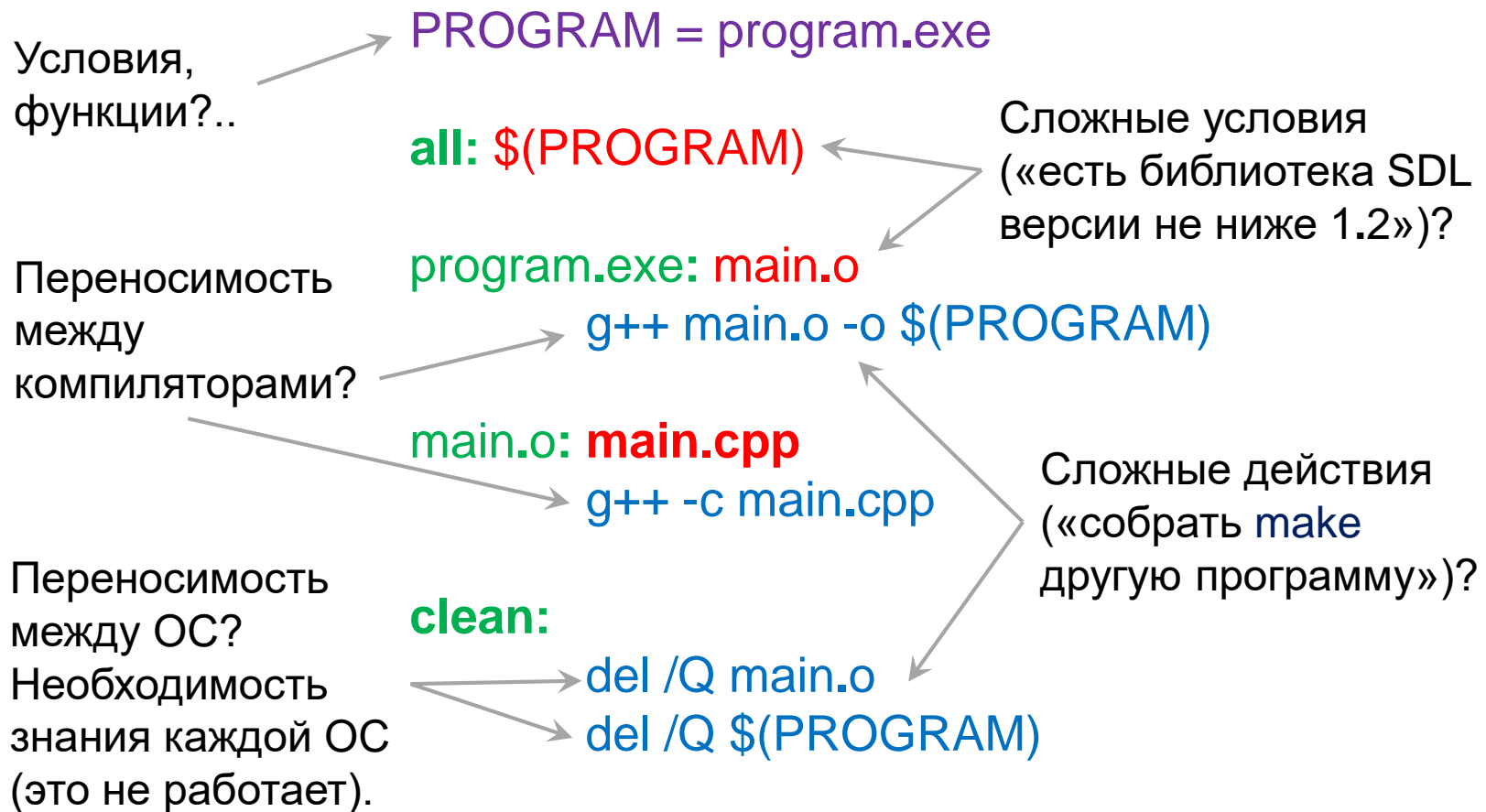
> make all

Удаление всего,

кроме исходного кода.

> make clean

Проблемы make



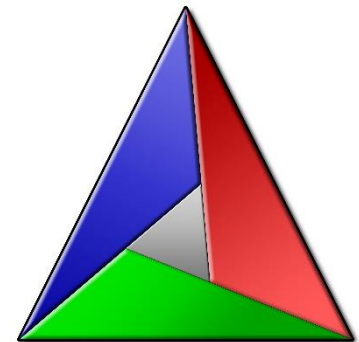
CMake — генератор Makefile

(и не только!)

CMakeLists.txt

```
cmake_minimum_required( VERSION 2.8 )
project( program )
set ( CMAKE_CXX_FLAGS
    "${CMAKE_CXX_FLAGS} -std=c++14" )
set( HEADERS
    input.h)
set( SOURCES
    main.cpp
    input.cpp)
add_executable(
    ${PROJECT_NAME}
    ${HEADERS}
    ${SOURCES})
```

Задание флагов
компиляции
для современного C++.



Характеристики CMake

Преимущества:

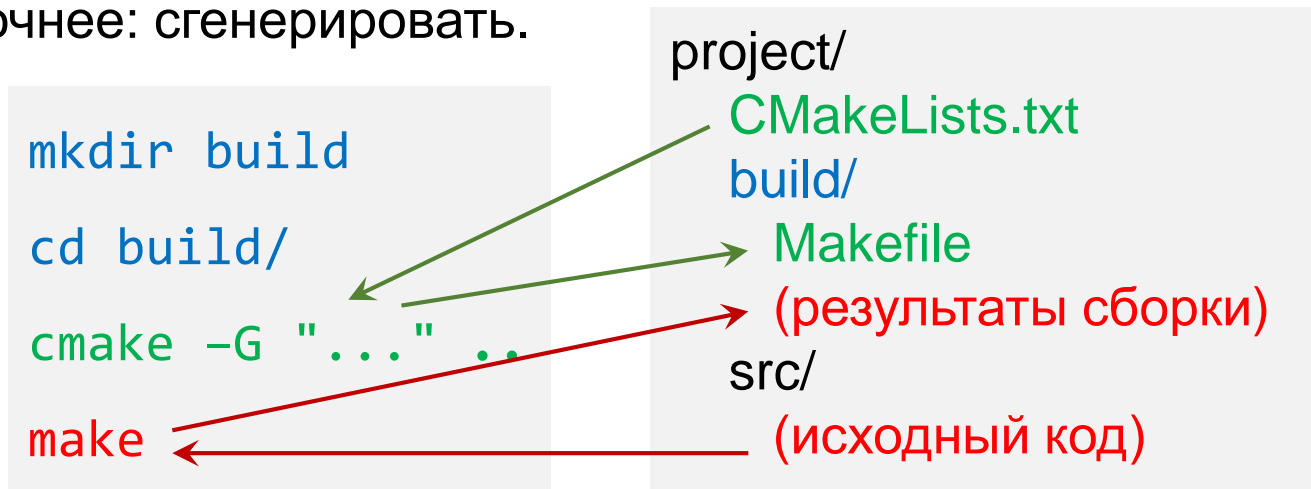
- Декларативность описания
(что нужно получить, а не что сделать).
- Переносимость между ОС и компиляторами.
- Возможность генерации по `CMakeLists.txt` не только `Makefile`, но и проектов для IDE.
- Широкая автоматизация («найти библиотеку»).
- Возможность out-of-source build.

Недостатки:

- Сравнительно запутанная система (ср.: `make`).
- Конфигурация — не программа, не всё доступно.

Out-of-source build

- Сборка в каталоге с исходным кодом засоряет его.
- Код может одновременно собираться под разные платформы, ОС — будут разные файлы на выходе.
- Решение: создавать их в отдельном каталоге.
 - Конкретно: написать **Makefile**, который обращался бы к файлам в ином, не в текущем, каталоге.
 - Точнее: сгенерировать.



Другие системы сборки

■ С и C++:

- Ninja — высокопроизводительный аналог make.
- Waf, Scons — конфигурация на Python,
- Shake — конфигурация на Haskell.
- MSBuild — в MS Visual Studio.

■ В других языках:

- Современные системы сборки не только собирают программу, но и загружают нужные библиотеки.
- Некоторые системы могут наблюдать изменения в файлах и перезапускаться автоматически.
- Интерпретируемым языкам нужна не компиляция, но выполнение ряда шагов — системы сборки актуальны.

Литература к лекции

- Документация к GNU make (<https://cmake.org/>).
- Статья «Просто о make» (аналог этой лекции) (<https://habrahabr.ru/post/211751/>).
- Сайт CMake (<https://cmake.org/>).
- Более подробное описание процесса сборки с примерами команд (<http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html>).