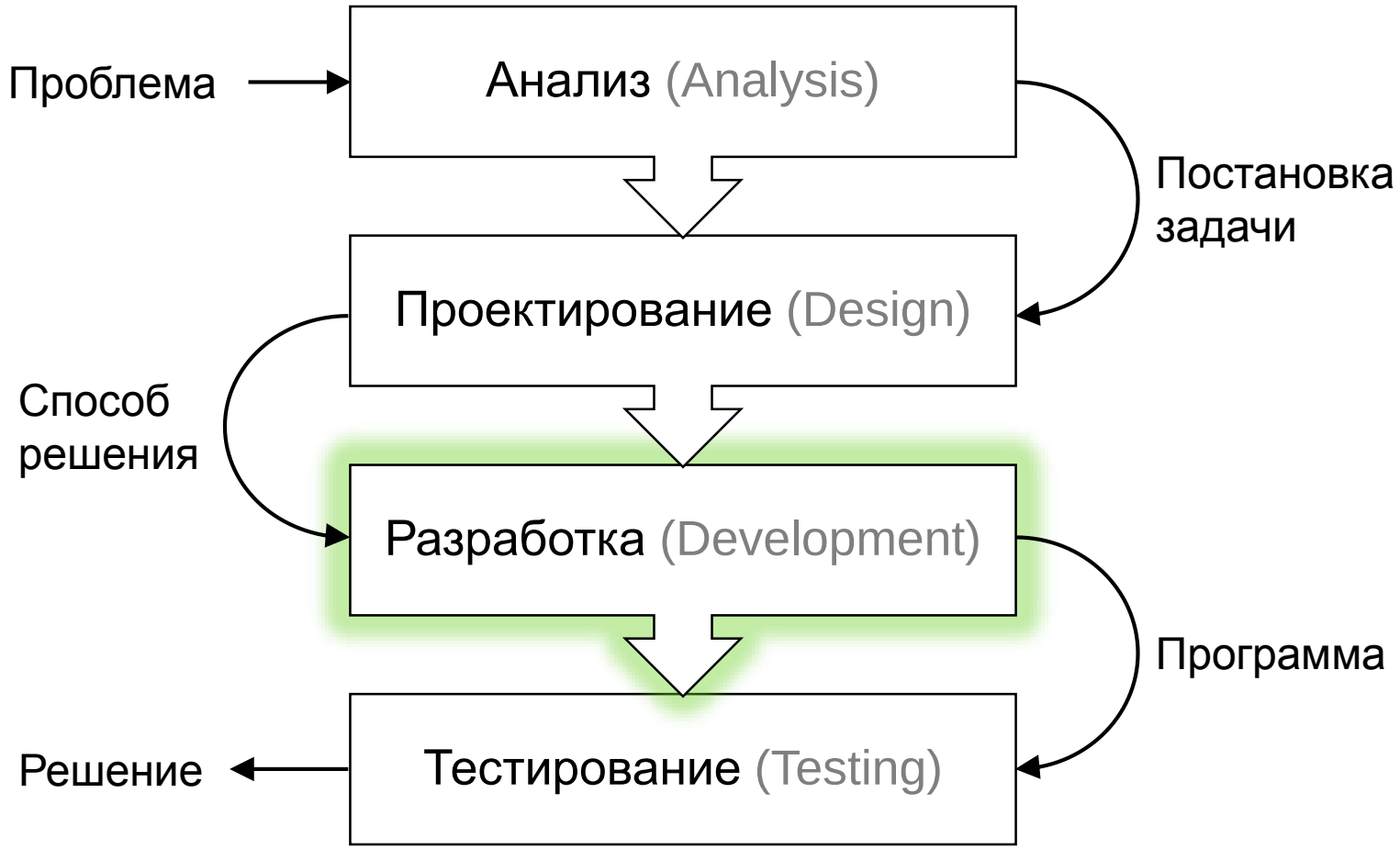
Средства автоматизации разработки программ

Курс «Технология программирования»

Кафедра управления и информатики НИУ «МЭИ»

Осень 2015 г.

Процесс создания ПО



Средства автоматизации разработки программ

1. Системы сборки
2. Анализаторы производительности
   - Иногда нужен анализ «вручную».
3. Генераторы документации
4. Статические анализаторы исходного кода
5. Системы контроля версий
   - Уже рассмотрены.

Системы сборки (build systems)

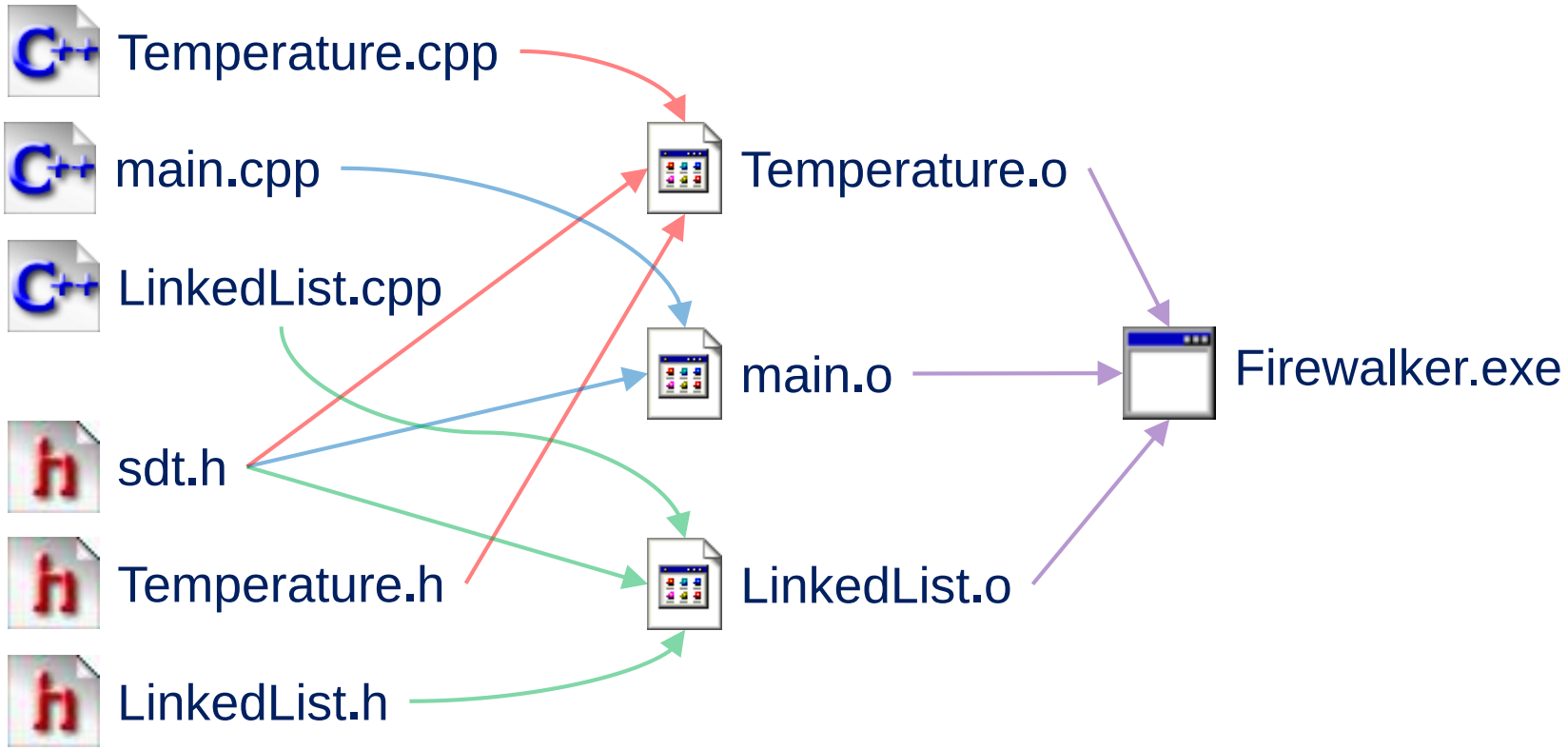
- **Задача:** собрать программу из нескольких файлов.
- Решения:
  - Компилировать файл за файлом, затем программу.
  - Использовать проект в IDE.
- Проблемы IDE как средства сборки:
  - Большое разнообразие по сравнению с компиляторами.
    - Программисты предпочитают разные IDE.
  - Малая переносимость между платформами.
  - Сложность автоматического использования.
    - Программы часто собирают на отдельном сервере (build server) без участия разработчика.
  - Собственное решение задачи сборки усложняет IDE.

Задача сборки программы

- (Возможные) элементы процесса сборки:
  - проверка наличия необходимых компонент;
  - обработка исходного кода:
    - статическое тестирование;
    - генерация кода;
    - генерация документации;
  - компиляция и компоновка;
  - запуск тестирования программы.
- Задачи систем сборки:
  - выполнение типовых этапов процесса готовыми блоками;
  - ускорение сборки за счет обработки только изменившихся файлов.

Программа make

При изменении одного файла пересобираются всё,
что зависит от него (и только это).



Структура Makefile

- Запуск make:
  - make
  - make *цель*
  - make -f 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-N

- Цель — что правило позволяет получить, т. е. какой файл будет создан в результате.
  - Либо название задачи, которое указывается make *цель*.
- Требования — 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

Простой пример Makefile

```
all : program.exe

program.exe : main.o convert.o
        g++ -o program.exe main.o convert.o

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

function.o : convert.cpp
        g++ -o convert.o -c convert.cpp
```

Особенности формата Makefile

- При вызове make без аргументов целью считается all.
- Пустые строки между правилами значимы.
- Отступы для действий — табуляции:

```
program.exe: main.cpp
	g++ -o program.exe main.cpp
```

- Есть несколько версий make (mk, gmake, nmake…), которые немного различаются.
  - На ЛР будет GNU Make (gmake).
  - Кроме mk, программы вызываются одинаково — make.

Расширенные средства make

- Специальные переменные (часть):

  $@ название цели;

  $< первое требование;

  $? все требования через пробел.

- Шаблонные правила (пример):
  - Для получения X.o нужно скомпилировать X.cpp:

```
%.o : %.cpp
        g++ –o $@ –c $<
```

- Запуск четырех процессов make параллельно (ускорение):

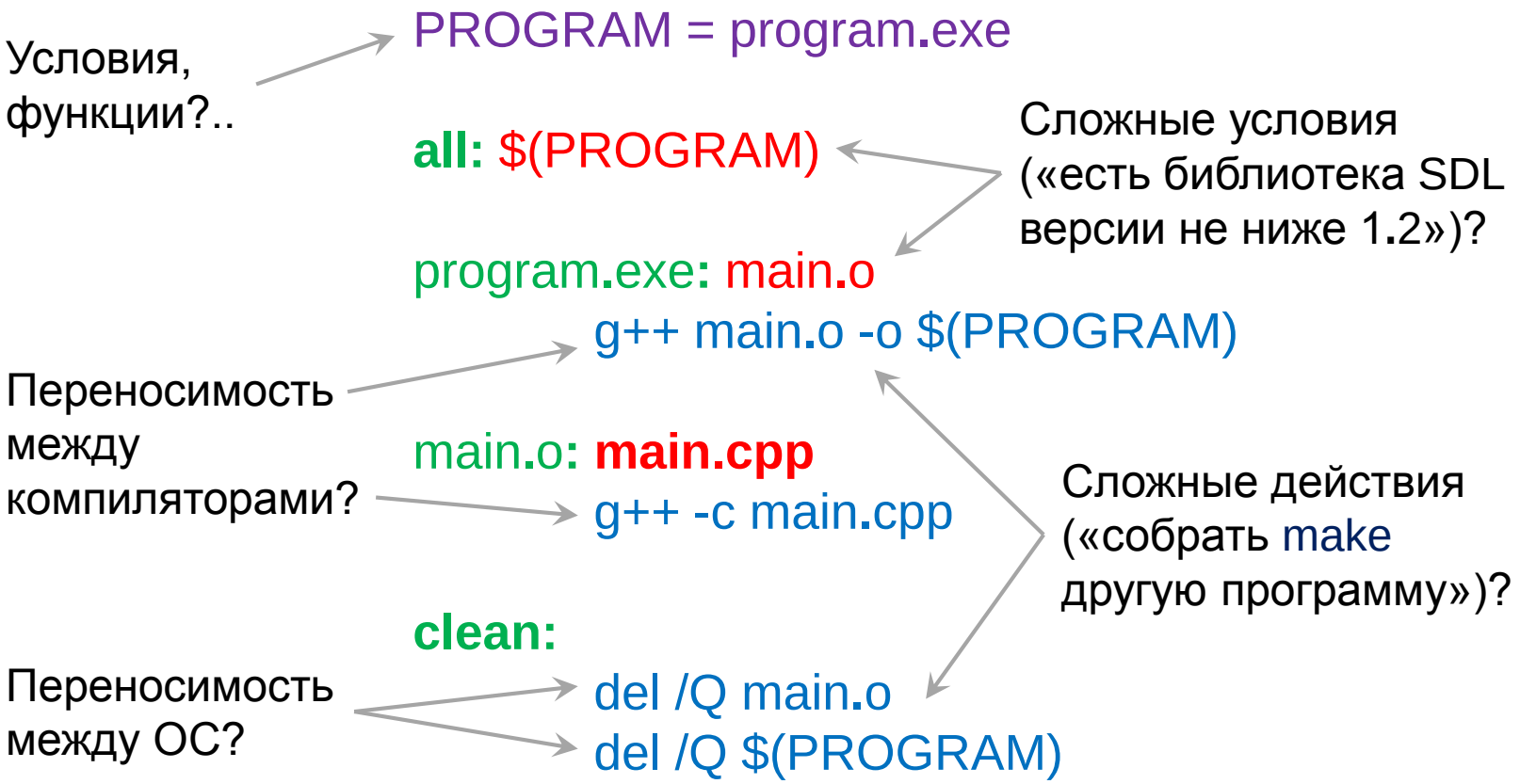
```
make –j 4
```

- В make есть много других специальных средств.

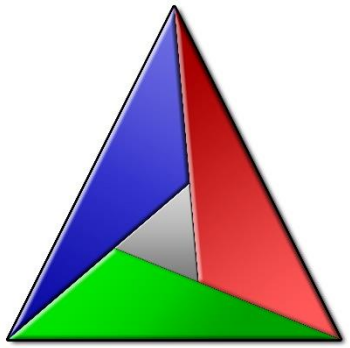
Усложненный пример Makefile

```
CC = g++ -c                 # Запуск компилятора.
LD = g++                    # Запуск компоновщика.
PROGRAM = lab3.exe          # Исполняемый файл.

all : $(PROGRAM)            # all : lab3.exe

$(PROGRAM) : main.o convert.o
        $(LD) –o $@ –static $?

%.o : %.cpp
        $(CC) –o $@ $<

clean:  # make clean удалит все файлы, кроме исходных.
        del /Q /S $(PROGRAM) *.o
```

Проблемы make



CMake — генератор Makefile



CMakeLists.txt

```
cmake_minimum_required( VERSION 2.8 )
project( Firewalker )
set( HEADERS
        Temperature.h
        LinkedList.h)
set( SOURCES
        Temperature.cpp
        LinkedList.cpp
        main.cpp)
add_executable(
        ${PROJECT_NAME}
        ${HEADERS}
        ${SOURCES})
```

Анализ производительности

- Profiling («профилирование»), performance analysis.
- Время работы блоков кода и функций.
  - Случайная величина с большой дисперсией.
  - Погрешность замеров ~ 1 мкс (10^{-6} с), до 10 нс (10^{-9} с).
  - Сильно зависит от среды:
    - аппаратная платформа, ОС и версии библиотек;
    - параллельные процессы;
    - способ тестирования:
      - современные ОС, ОЗУ и ЦП работают неочевидно.
- Анализ потребления памяти, нагрузки на диск.

Анализ производительности специальным ПО (profilers)

1. Скомпилировать программу особым образом (GCC: ключ -pg).
2. Выполнить прогон программы (анализируемый фрагмент должен отработать многократно).
3. Проанализировать сгенерированный файл со временами работы функцией.

Анализаторы производительности общего назначения (для ЦП)

	GNU Profiler (GProf)	Intel VTune Amplifier
Операционная система	любая	Windows, Linux
Центральный процессор	любой	совместимый с Intel
Стоимость	бесплатно	от 900 до 3100 $
Особенности	• универсальность • учет особенностей анализа производительности в *nix • терминальный режим для встраиваемых систем	• богатство инструментов • учет особенностей процессоров Intel • развитой интерфейс

Анализ производительности явными замерами времени

- Profiler может быть недоступен.
- Бывает нужно замерить время работы
  не блока кода, а между двумя точками.
- **Задачи и проблемы**
  - Замеры должны занимать пренебрежимо малое время,
    - как и сохранение выборки в памяти.
  - Точность замеров должна быть высокой:
    - аппаратный таймер ЦП;
    - thread CPU affinity.
  - Тестирование должно быть «адекватным».

Документирование кода

- Полезно разработчикам:
  - Перечень сущностей (функции, типы данных)
  - Каким *в точности* заявлено поведение функций?
  - Обзор и анализ структуры кода
- Нужно руководителям в ИТ:
  - Как часть отчетности по проекту или продукта-библиотеки.
  - Для принятия решения об использовании библиотеки:
    - определение круга задач, решаемых библиотекой;
    - оценка стоимости внедрения:
      - объем и сложность структуры библиотеки,
      - наличие и качество документации *(замкнутый круг).*

Требования к документации

- Актуальность:
  - простота поддержания.
- Доступность:
  - время на создание;
  - понятность языка для читателя;
  - формат документов.
- Удобство работы:
  - алфавитный и предметный указатель;
  - перекрестные ссылки.
- Оформление:
  - логичное;
  - эргономичное;
  - стандартное;
  - иллюстрации и схемы:
    - диаграмма классов,
    - зависимости модулей.

Классические подходы

Написание вручную

- Содержательность, ёмкость, логичное изложение.
- Специальные иллюстрации.
- Необходимость актуализации вручную при изменениях кода.
- Высокая трудоемкость.

Чтение исходного кода

- Всегда актуальная информация.
- Комментарии уже имеются.
- Нужна квалификация:
  - сложно для руководителя;
  - привычно разработчику,
    - но надолго отвлекает.
- Трудно получить общее представление.
- Оформление отсутствует.

Генераторы документации

- По исходному коду можно сформировать документацию:
  - оглавление, алфавитный указатель, поиск;
  - диаграммы классов и вызовов (структура библиотеки).
- В коде есть комментарии —
  - их можно включить в документацию;
  - в них можно указывать команды генератору документации:
    - управление форматированием,
    - перекрестные ссылки,

      (частично генерируются автоматически);
    - разнесение сущностей по логическим разделам.

Характеристики генерируемой документации

- Удовлетворительная структура и оформление:
  - Приходится вносить много информации в комментарии,
    - **код становится труднее читать;**
    - IDE может формировать контекстные подсказки по ним.
  - Излишняя подробность, громоздкость.
  - Специальные иллюстрации все равно выполняются вручную.
- Актуализируется автоматически.
- Малая трудоемкость:
  - достаточно поддерживать комментарии в коде и соблюдать формат комментариев;
  - можно формировать документы, файлы справки, версии для web.

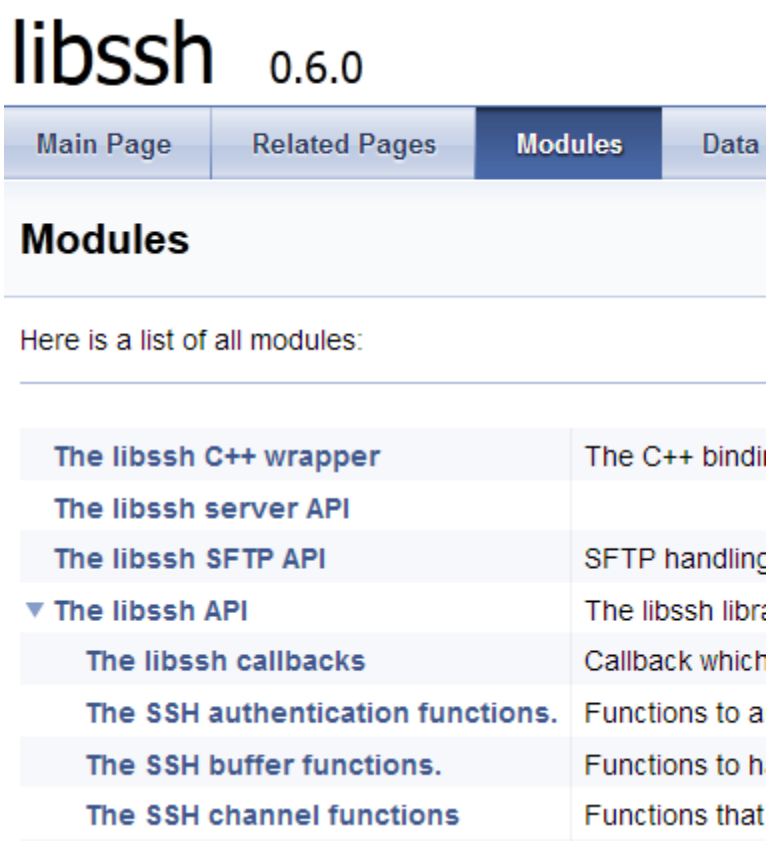
Популярные генераторы документации

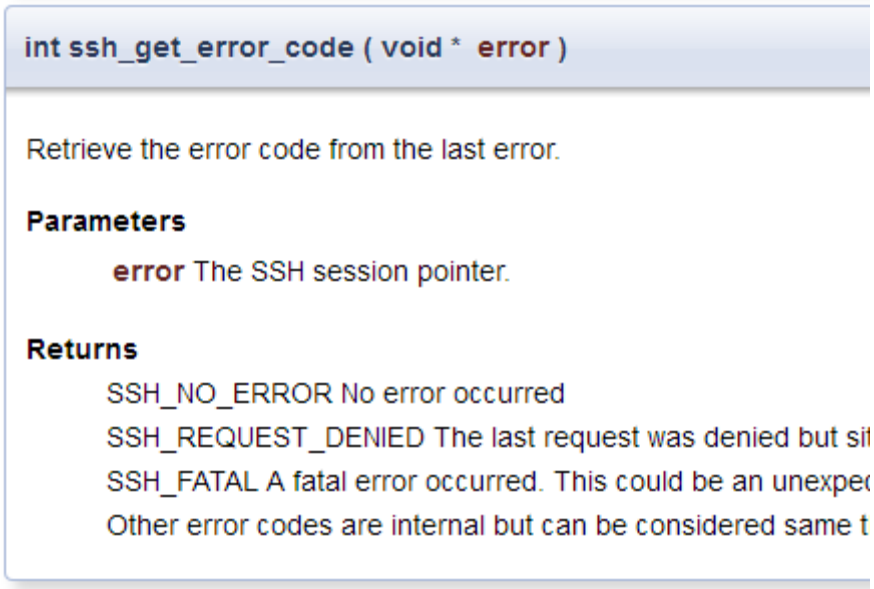
- Doxygen (C-подобные языки)
  - DoxyBlocks — расширение Code::Blocks
    (индустриальный стандарт для C++).
- JavaDoc (Java)
  (индустриальный стандарт для Java)
  - PhpDocumentor (PHP), JSDoc (Java Script)
- Sphinx (Python и другие)
  - Python имеет встроенные в язык """docstrings""".
- Doc-O-Matic

Документирующий комментарий

```
/** @brief Производит снятие средств со счета.
  *
  * Снятие средств со счета возможно, если сумма к снятию
  * положительна и не превышает баланса на счете, а сам счет
  * не заморожен (@see isFrozen). После успешного снятия сумма
  * средств на счете (@see getBalance) уменьшается на величину
  * @paramref amount.
  *
  * @param amount Сумма средств к снятию.
  * @returns Удалось ли произвести снятие (@code true),
  *                  или возникла ошибка (@code false).
  */
bool withdraw(double amount);
```

Примеры сгенерированной документации (doxygen)

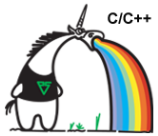




Статическое тестирование (анализ исходного кода)

- Поиск ошибок в коде без запуска программы.
- **Проблемы в исходном коде**
  - Потенциальные ошибки:
    - явные опечатки;
    - всегда истинные и всегда ложные условия, дублирование условий;
    - отсутствие обработки ошибок;
    - неправильная работа со стандартными компонентами;
    - использование устаревшей функциональности.
  - Нарушение правил кодирования.

Популярные статические анализаторы кода (C++)

	 **PVS-Studio**	**CppCheck**	 **Coverity**
Операционная система	Windows	любая	любая
Стоимость	5—10 тыс. €	бесплатно (свободное ПО)	• по договоренности • для некоторых проектов OSS — бесплатно
Интеграция со средами разработки	Microsoft Visual Studio, Embarcadero C++ Builder	Code::Blocks, CodeLite, Eclipse, средства сборки	средства сборки для C++ и Java; continuous integration
Выдающиеся преимущества	богатство методов анализа	• коммерческая бесплатность • простота интеграции	• анализ, кроме C++, кода на C# и Java • комплексное решение для QA