Практика 6.
Doxygen

Doxygen — это кросплатформенная система для получения документации на основе исходных текстов. Система поддерживает C++, C, Java, Objective-C, Python, IDL(CORBA и MIDL), Fortran, VHDL, PHP, C# и, частично, D. Doxygen выбирает документацию из комментариев сделанных в коде, но может быть полезен и при работе с недокументированными исходными текстами, оформив структуру кода: он строит прекрасные диаграммы классов и графы зависимостей, а также оформляет код гиперссылками. Вывод можно получить в HTML или LaTeX и может легко быть сконвертирован в CHM или PDF. Кроме этого, doxygen имеет встроенную поддержку генерации документации в формате man, RTF и XML.

Установка
Способ 1: sudo apt-get install doxyg*en*

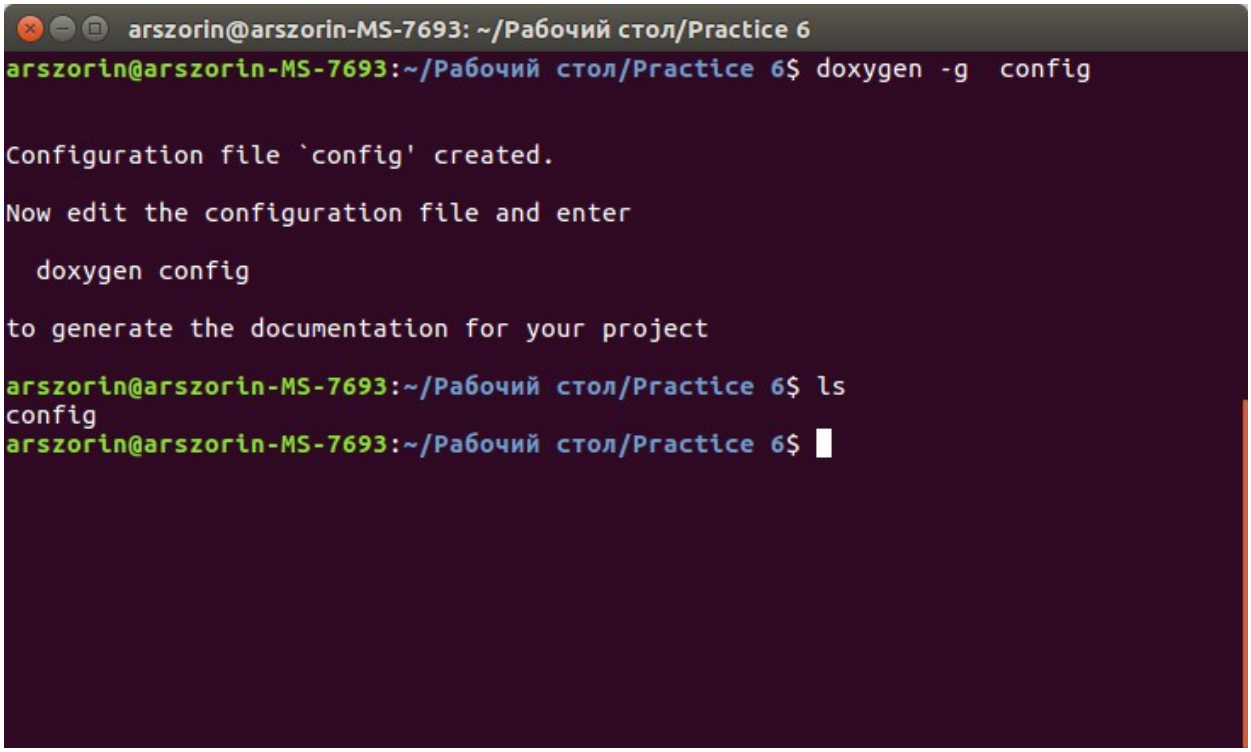
Способ 2:
1) Склонируйте репозитарий «https://github.com/doxygen/doxygen.git»
2) Перейдите в скаченный каталог и создайте в нем директорию «build».
3) Выполните команду «cmake -G "Unix Makefiles" .. » (Возможно вам потребуется скачать cmake, для этого выполните команду sudo apt-get install cmake).
4) Выполните «make» и затем «make install»

Использование Doxygen

Использовать Doxygen просто – для этого надо просто запустить программу указав ей путь к файлу с настройками. Файл с настройками представляет собой простой текстовой файл, который можно редактировать как в текстовом редакторе, так и с помощью специальных программ, например Doxygate.

В настройках описывается внешний вид документации, какие сущности и отношения между ними следует включать в нее, имя проекта, путь к анализируемым файлам и так далее.

Для создания файла конфига выполните команду «doxygen -g имя_файла_конфига». Программа сгенерирует файл с настройками по умолчанию.



Подробное описание каждой команды в конфигурационном файле можно найти тут: http://www.nrjetix.com/fileadmin/doc/publications/additional_info/DoxyGen%20guide.pdf

После создания кофига, комментирования кода, создание документация происходит посредством выполнения команды «doxygen имя_конфига»

Комментирование

Doxygen поддерживает несколько стилей комментариев:

1. JavaDoc стиль:

```
/**
 ... первая строчка ...
 ... вторая строчка ...
 */
```

Для однострочного комментирования: `/// строчка`

2. Qt стиль, в котором в начале вместо второй звёздочки ставится восклицательный знак:

```
/*!
 * ... первая строчка ...
 * ... вторая строчка ...
 */
```

Для однострочного комментирования: `//! строчка`

Так же стоит обратить внимание, что в однострочном документирующем блоке могут быть использованы специальные команды, такие как /brief, /details и тд.

Документирующие блоки, следующие друг за другом, объединяются в один (причем вне зависимости от используемого стиля и того, являются они многострочными или однострочными).

Например следующие два способа документирования дадут один и тот же результат:

```
/// \brief Краткое описание
/// \details Подробное описание

///Краткое описание
/*!
    Подробное описание
*/
```

Размещение документирующего блока после элемента
Во всех предыдущих примерах подразумевалось, что документирующий блок предварял документируемый элемент, но иногда бывают ситуации, когда удобнее разместить его после документируемого элемента. Для этого необходимо в блок добавить маркер "<", как в примере ниже:

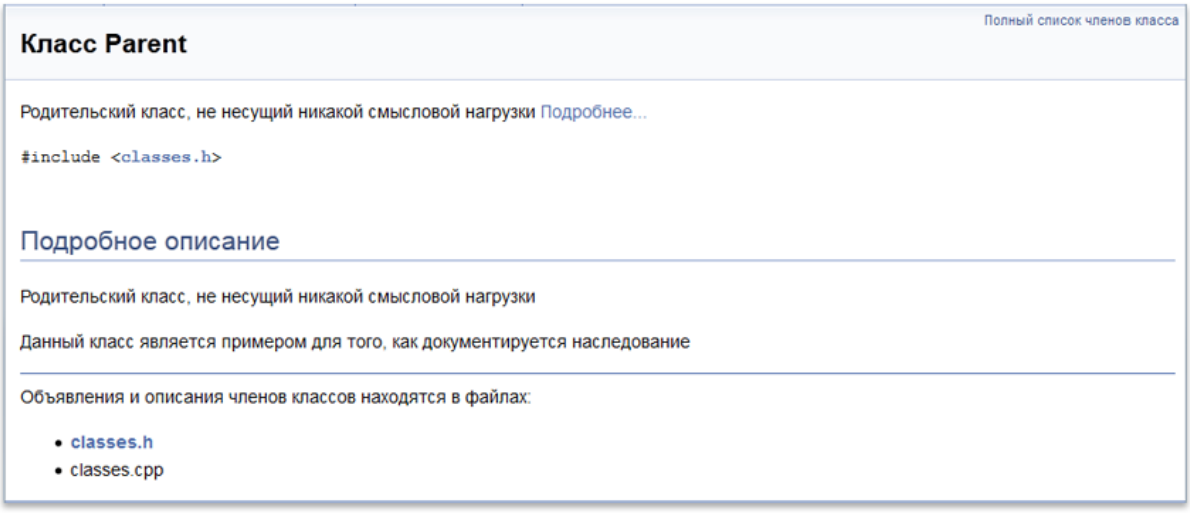
```
int variable; ///< Краткое описание
```

Пример документации

Теперь рассмотрим то, как это будет выглядеть на практике. Ниже представлен документированный код некоторого класса в соответствии с теми правилами, которые мы рассматривали ранее.

```
/*!
        \brief Родительский класс, не несущий никакой смысловой нагрузки

        Данный класс имеет только одну простую цель: проиллюстрировать то,
        как Doxygen документирует наследование
*/
class Parent {
public:
        Parent();
        ~Parent();
};
```

В итоге Doxygen сформирует на основе данных комментариев следующую красиво оформленную страничку (здесь приведена вырезка из неё):



Теперь, когда мы научились основам, пришла пора познакомиться с тем, как можно детализировать документацию. Инструментом для этого являются команды.

Команды

Команда	Значение
\authors	Указывает автора или авторов
\version	Используется для указания версии
\date	Предназначена для указания даты разработки
\bug	Перечисление известных ошибок
\warning	Предупреждение для использования
\copyright	Используемая лицензия
\example	Команда, добавляемая в комментарий для указания ссылки на исходник с примером (добавляется после команды)
\todo	Команда, используется для описания тех изменений, которые необходимо будет сделать (TODO).

Пример использования некоторых команд и результат приведены ниже:

```
/*!
        \brief Дочерний класс
        \author Norserium
        \version 1.0
        \date Март 2015 года
        \warning Данный класс создан только в учебных целях

        Обычный дочерний класс, который отнаследован от ранее созданного класса
Parent
*/
class Son : public Parent {
public:
        Son();
```

```
        ~Son();
};
```

Документирование файла

Хорошей практикой является добавление в начало файла документирующего блока, описывающегося его назначение. Для того, чтобы указать, что данный блок относится к файлу необходимо воспользоваться командой *file* (причём в качестве параметра можно указать путь к файлу, к которому относится данный блок, но по умолчанию выбирается тот файл, в который блок добавляется, что, как правило, соответствует нашим нуждам).

```
/*!
\file
\brief Заголовочный файл с описанием классов

Данный файл содержит в себе определения основных
классов, используемых в демонстрационной программе
*/
#ifndef CLASSES_H
#define CLASSES_H

...

#endif // CLASSES_H
```

Документирование функций и методов

При документировании функций и методов чаще всего необходимо указать входные параметры, возвращаемое функцией значение, а также возможные исключения. Рассмотрим последовательно соответствующие команды.

Параметры

Для указания параметров необходимо использовать команду *param* для каждого из параметров функции, при этом синтаксис команды имеет следующий вид:

```
\param [<направление>] <имя_параметра> {описание_параметра}
```

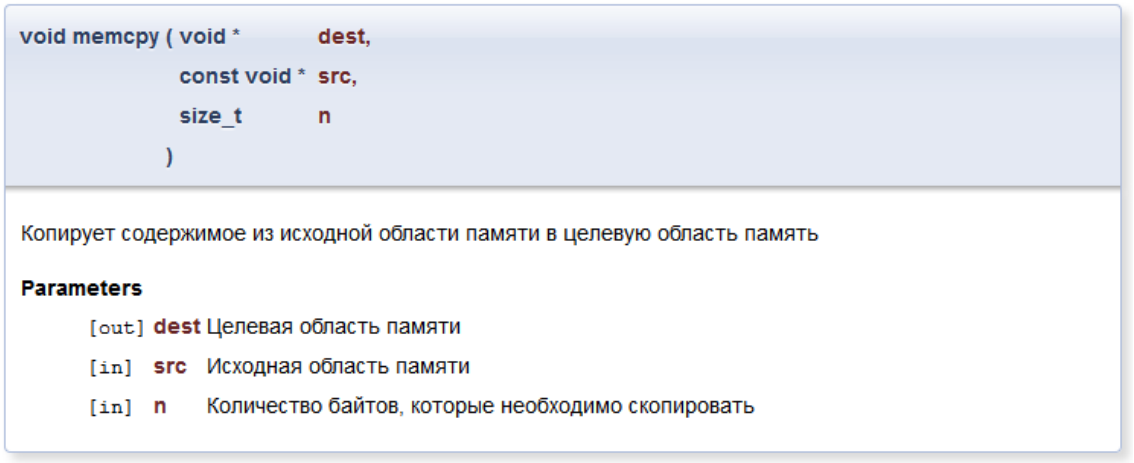
Рассмотрим значение компонентов команды:

1. Имя параметра – это имя, под которым данный параметр известен в документируемом коде;
2. Описание параметра представляет собой простое текстовое описание используемого параметра..
3. Направление – это опциональный атрибут, который показывает назначение параметра и может иметь три значения "[in]", "[out]", "[in,out]";

Сразу же перейдём к примеру:

```
/*!
Копирует содержимое из исходной области памяти в целевую область память
\param[out] dest Целевая область памяти
\param[in] src Исходная область памяти
\param[in] n Количество байтов, которые необходимо скопировать
*/
void memcpy(void *dest, const void *src, size_t n);
```

В результате мы получим такую вот аккуратную документацию функции:



void memcpy (void * dest,
const void * src,
size_t n
)

Копирует содержимое из исходной области памяти в целевую область память

Parameters

[out] **dest** Целевая область памяти

[in] **src** Исходная область памяти

[in] **n** Количество байтов, которые необходимо скопировать

Возвращаемое значение

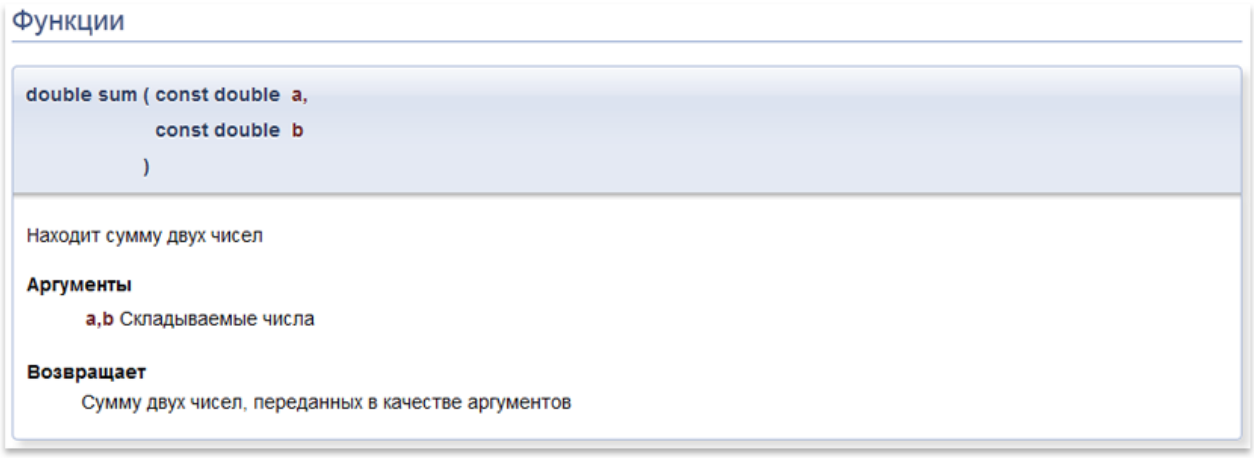
Для описание возвращаемого значения используется команда *\return* (или её аналог *\returns*). Её синтаксис имеет следующий вид:

```
\return {описание_возвращаемого_значения}
```

Рассмотрим пример с описанием возвращаемого значения (при этом обратите внимание на то, что параметры описываются при помощи одной команды и в результате они в описании размещаются вместе):

```
/*!
Находит сумму двух чисел
\param a,b Складываемые числа
\return Сумму двух чисел, переданных в качестве аргументов
*/
double sum(const double a, const double b);
```

Получаем следующий результат:



Исключения

Для указания исключения используется команда *throw* (или её синонимы: *throws*, *exception*), которая имеет следующий формат:

```
\throw <объект-исключение> {описание}
```

Простейший пример приведён ниже:

```
\throw std::bad_alloc В случае возникновения ошибки при выделении памяти
```

Документирование классов

Классы также могут быть задокументированы просто предварением их документирующим блоком. При этом огромное количество информации Doxygen получает автоматически, учитывая синтаксис языка, поэтому задача документирования классов значительно упрощается. Так при документировании Doxygen автоматические определяет методы и члены класса, уровни доступа к функциям, дружественные функции и т.п.

Если ваш язык не поддерживает явным образом определенные концепции, такие как например уровни доступа или создание методов, но их наличие подразумевается и его хотелось бы как-то выделить в документации, то существует ряд команд (например, *public*, *private*, \protected, *memberof*), которые позволяют указать явно о них Doxygen.

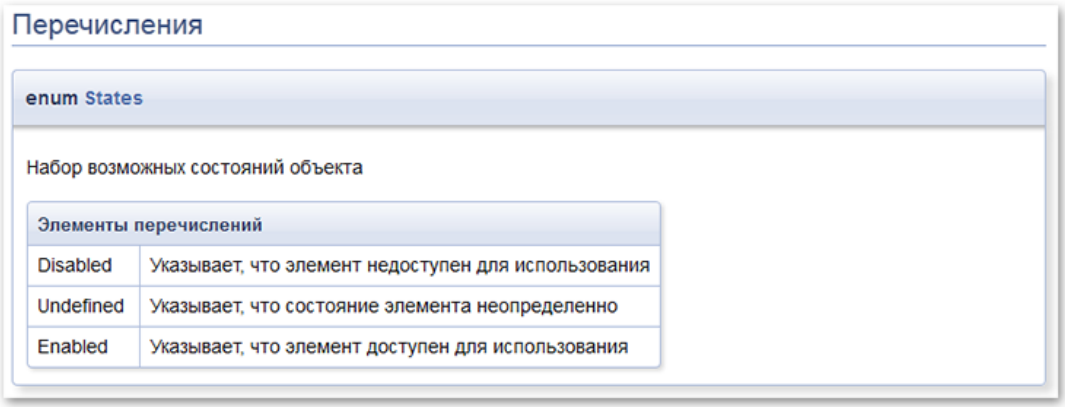
Документирование перечислений

Документирование перечислений не сильно отличается от документирования других элементов. Рассмотрим пример, в котором иллюстрируется то, как можно удобно документировать их:

```
/// Набор возможных состояний объекта
enum States {
        Disabled, ///< Указывает, что элемент недоступен для использования
        Undefined, ///< Указывает, что состояние элемента неопределенно
        Enabled, ///< Указывает, что элемент доступен для использования
}
```

То есть описание состояний указывается, собственно, после них самих при помощи краткого или подробного описания (в данном случае роли это не играет).

Результат будет иметь следующий вид:



Модули

Отдельное внимание следует обратить на создание модулей в документации, поскольку это один из наиболее удобных способов навигации по ней и действенный инструмент её структуризации. Пример хорошей группировки по модулям можете посмотреть здесь.

Далее мы кратко рассмотрим основные моменты и приведём пример, однако если вы хотите разобраться во всех тонкостях, то тогда придётся обратиться к соответствующему разделу документации.

Создание модуля

Для объявления модуля рекомендуется использовать команду *\defgroup*, которую необходимо заключить в документирующий блок:

```
\defgroup <идентификатор> (заголовок модуля)
```

Идентификатор модуля представляет собой уникальное слово, написанное на латинице, который впоследствии будет использован для обращения к данному модулю; заголовок модуля – это произвольное слово или предложение (желательно краткое) которое будет отображаться в документации.

Обратите внимание на то, что при описании модуля можно дополнить его кратким и подробным описанием, что позволяет раскрыть назначение того или иного модуля, например:

```
/*!
    \defgroup maze_generation Генерация лабиринтов
    \brief Данный модуль, предназначен для генерации лабиринтов.

     На данный момент он поддерживает следующие алгоритмы генерации лабиринтов:
Eller's algorithm, randomized Kruskal's algorithm, cellular automaton algorithm,
randomized Prim's algorithm.
*/
```

Размещение документируемого элемента в модуле

Для того, чтобы отнести тот или иной документируемый элемент в модуль, существуют два подхода.

Первый подход – это использование команды *ingroup*:

```
\ingroup <идентификатор> (заголовок модуля)
```

Его недостатком является то, что данную команду надо добавлять в документирующие блоки каждого элемента исходного кода, поскольку их в рамках одного модуля может быть достаточно много.

Поэтому возникает необходимость в другом подходе, и второй подход состоит в использовании команд начала и конца группы: *@{* и *@}*. Следует отметить, что они используются наряду с командами *defgroup*, *addtogroup* и *weakgroup*.

Пример использования приведён ниже:

```
/*! \defgroup <идентификатор> (заголовок модуля)
    @{
*/
  документируемые_элементы
/*! @} */
```

Смысл примера должен быть понятен: мы объявляем модуль, а затем добавляем к ней определенные документируемые элементы, которые обрамляются при помощи символов начала и конца модуля.

Однако, модуль должен определяться один раз, причём это объявление будет только в одном файле, а часто бывает так, что элементы одного модуля разнесены по разным файлым и потому возникает необходимость использования команды *addtogroup*, которая не переопределяет группу, а добавляет к ней тот или иной элемент:

```
/*! \addtogroup <идентификатор> [(заголовок модуля)]
    @{
*/
   документируемые_элементы
/*! @} */
```

Название модуля указывать необязательно. Дело в том, что данная команда может быть использована как аналог команды *\defgroup*, и если соответствующий модуль не был определена, то она будет создана с соответствующим названием и идентификатором.

Наконец, команда *\weakgroup* аналогична команде *\addtogroup*, отличие заключается в том, что она просто имеет меньший приоритет по сравнению с ней в случае если возникают конфликты, связанные с назначение одного и того же элемента к разным модулям.

Создание подмодуля

Для создания подмодуля достаточно при его определении отнести его к тому или иному подмодулю, подобно любому другому документируемому элементу.

Пример приведён ниже:

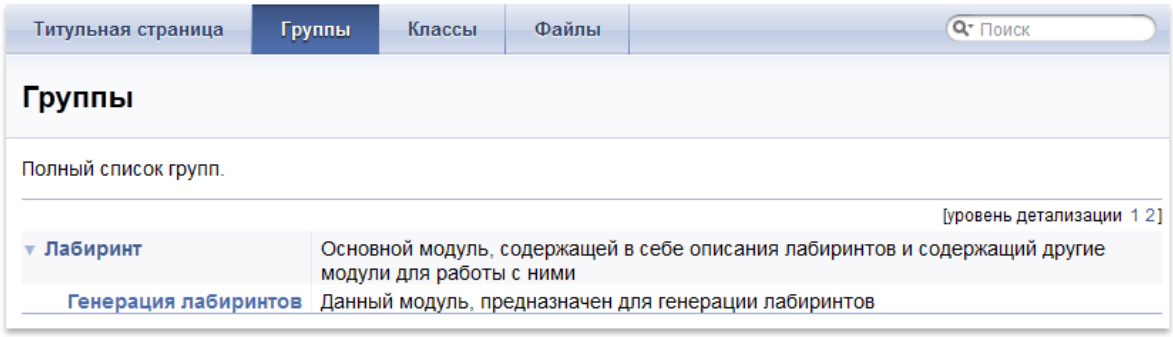
```
/*! \defgroup main_module Главный модуль */

/*! \defgroup second_module Вложенный модуль
     \ingroup main_module
*/
```

Пример создания нескольких модулей

Далее представлен подробный пример создания модуля:
Файл generate_maze.h
Файл maze.h

Обратите внимание на то, что вид документирующих блоков несколько изменился по сравнению с приведёнными ранее примерами, но как мы говорили ранее, принципиального значения это не играет и выбирайте тот вариант, который вам ближе.

В результате мы получим следующую документацию:



Оформления текста документации

Теперь, после того, как мы в общих чертах разобрались с тем как документировать основные элементы кода, рассмотрим то, как можно сделать документацию более наглядной, выразительной и полной.

Команды \code и \endcode

Один из удобных способов сделать это – команды *\code* и *\endcode*, которые применяются следующим образом:

```
\code [ {<расширение>}]
     ...
\endcode
```

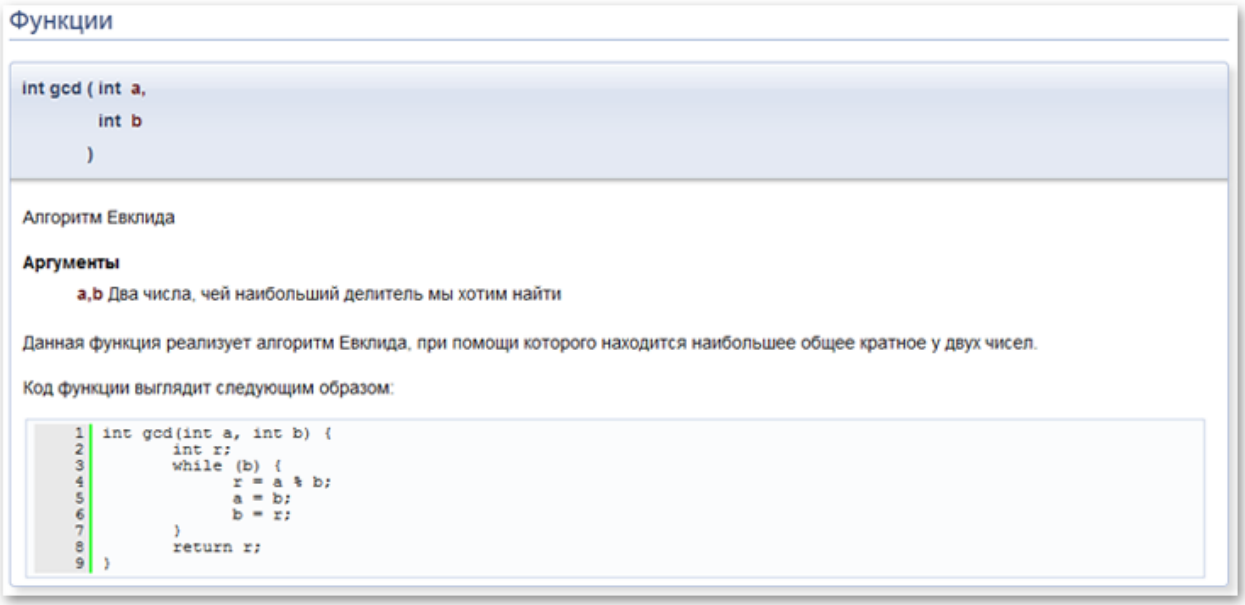
Используемый язык определяется автоматически в зависимости от расширения файла, в котором располагается документирующий блок, однако в случае, если такое поведение не соответствует ожиданиям расширение можно указать явно.

Рассмотрим пример использования:

```
/*!
     \brief Алгоритм Евклида
     \param a,b Два числа, чей наибольший делитель мы хотим найти

     Данная функция реализует алгоритм Евклида, при помощи которого
     находится наибольшее общее кратное у двух чисел.

     Код функции выглядит следующим образом:
     \code
     int gcd(int a, int b) {
            int r;
            while (b) {
                  r = a % b;
                  a = b;
                  b = r;
            }
            return r;
     }
     \endcode
```

```
*/
int gcd(int a, int b);
```

Результат будет иметь следующий вид:



Команда \include

Как альтернатива данному способу существует команда *include*, общий формат которой имеет следующий вид:

```
\include <имя_файла>
```

Она полностью копирует содержимое файла и вставляет его в документацию как блок кода (аналогично предыдущей рассмотренной команде.

Команда \snippet

Команда *snippet* аналогична предыдущей команде, однако она позволяет вставлять не весь файл, а его определенный фрагмент. Неудивительно, что её формат несколько другой:

```
\snippet <имя_файла> ( имя_фрагмента )
```

Для выделения определенного фрагмента кода необходимо в начале и в конце его разместить документирующий блок с указанием имени фрагмента:

```
//! [ имя_фрагмента ]
    ...
//! [ имя_фрагмента ]
```

Автоматическое внедрение кода

Наконец рассмотрим последний, наверно наименее гибкий метод. Дело в том, что Doxygen поддерживает возможность автоматической вставки тела функций, методов, классов, структур и т.п., в их подробное описание. Для этого используется следующая опция:

```
INLINE_SOURCES = YES
```

Далее будут использоваться следующие обозначения при описании аргументов команды, когда будет приводиться её общий формат:

Обозначение	Значение
<...>	Угловые скобки показывают, что аргумент представляет собой одно слово
(...)	Круглые скобки показывают, что аргументом является весь текст вплоть до конца строки, на которой размещена команда
{...}	Фигурные скобки показывают, что аргументом является весь текст вплоть до до следующего параграфа. Параграфы разделяются пустой строкой или командой-разделителем

Кроме того, обратите внимание на то, что вы можете создавать и собственные команды. Подробно об этом можно прочитать в соответствующем разделе документации.

Задание:

1. Склонируйте репозитарий https://github.com/arizorin/Practice6-Doxygen-.git и разберите код программы, конфиг Doxygen.
2. Создайте документацию к своей собственной программе.

Список литературы:

1. http://www.devexp.ru/2010/02/ispolzovanie-doxygen-dlya-dokumentirovaniya-koda/
2. https://habrahabr.ru/sandbox/26539/
3. http://courses.graphicon.ru/files/courses/cg/2008/prac/doxygen.pdf
4. https://www.stack.nl/~dimitri/doxygen/manual/