

КАК СТАТЬ АВТОРОМ

Как бессонница в час ночной, меняет промокодище облик твой

Все важнейшие IT-события здесь



Norsarium

6 мар 2015 в 08:37

Документируем код эффективно при помощи Doxygen

18 мин

346К

Программирование*, C++*, C*, C#*

Public Member Functions

```
Copy constructor. Create a new SAXParseException from a message and a Locator.
SAXParseException(const XMLCh *const message, const Locator *locator, MemoryManager *const manager=XMLPlatformUtils::getMemoryManager(), const XMLCh *const publicId, const XMLCh *const systemId, const XMLCh *const errorName, const XMLCh *const reason, const XMLFileLineColumn *const fileLineColumn, const XMLParserException &toCopy)
~SAXParseException()
Destructor.
```

Данная статья входит в получившийся цикл статей о системе документирования Doxygen:

1. Документируем код эффективно при помощи Doxygen
2. Оформление документации в Doxygen
3. Построение диаграмм и графов в Doxygen

Это первая и основная статья из упомянутого цикла и она представляет собой введение в систему документирования исходных текстов Doxygen, которая на сегодняшний день, по имеющему основания заявлению разработчиков, стала фактически стандартом для документирования программного обеспечения, написанного на языке C++, а также получила пусть и менее широкое распространение и среди ряда других языков.

В этой статье мы сначала познакомимся с самой системой и её возможностями, затем разберёмся с её установкой и базовыми принципами работы, и, наконец, завершим знакомство рассмотрением различных примеров документации, примеров того, как следует документировать те или иные части кода. Словом, познакомимся со всем тем, что позволит вам освоиться и начать работать с этой замечательной системой.

Введение

Вероятнее всего, каждый из нас сталкивался с результатами работы различных генераторов документации. Общий принцип их работы следующий: на вход такого генератора поступает специальным образом комментированный исходный код, а иногда и другие компоненты программы, а на выходе создаётся готовая документация для распространения и использования.

Рассматриваемая система Doxygen как раз и выполняет эту задачу: она позволяет генерировать на основе исходного кода, содержащего комментарии специального вида, красивую и удобную документацию, содержащую в себе ссылки, диаграммы классов, вызовов и т.п. в различных форматах: HTML, LaTeX, CHM, RTF, PostScript, PDF, мап-страницы.

Для того, чтобы составить общее впечатление о системе, ниже представлены примеры различных документаций для API, созданных при помощи Doxygen (следует обратить внимание, что в последние примеры внесены заметные изменения в сравнении со стандартной документацией, которую генерирует данная система):

1. Документация к API игрового движка CrystalSpace
2. Документация к Visualization Toolkit
3. Документация к исходникам Abiword
4. Документация к API KDE
5. Документация к API Drupal

Внимательный читатель наверняка обратил внимание на то, что в большинстве примеров Doxygen используется для документации программного обеспечения, написанного на языке C++, однако на самом деле данная система поддерживает гораздо большое число других языков: C, Objective-C, C#, PHP, Java, Python, IDL, Fortran, VHDL, Tcl, и частично D.

Впрочем, следуя сложившейся традиции, в примерах я буду использовать C++, однако это не должно смущать вас, если вы предпочитаете другой поддерживаемый язык, поскольку особой разницы на практике вы даже не заметите, и большинство сказанного далее будет справедливо и для вашего языка.



К слову, список проектов, использующих Doxygen имеется на официальном сайте, причём большинство из этих проектов свободные. Поэтому желающие могут скачать исходник того или иного проекта и посмотреть как там разработчики осуществляли документацию.

Установка и настройка

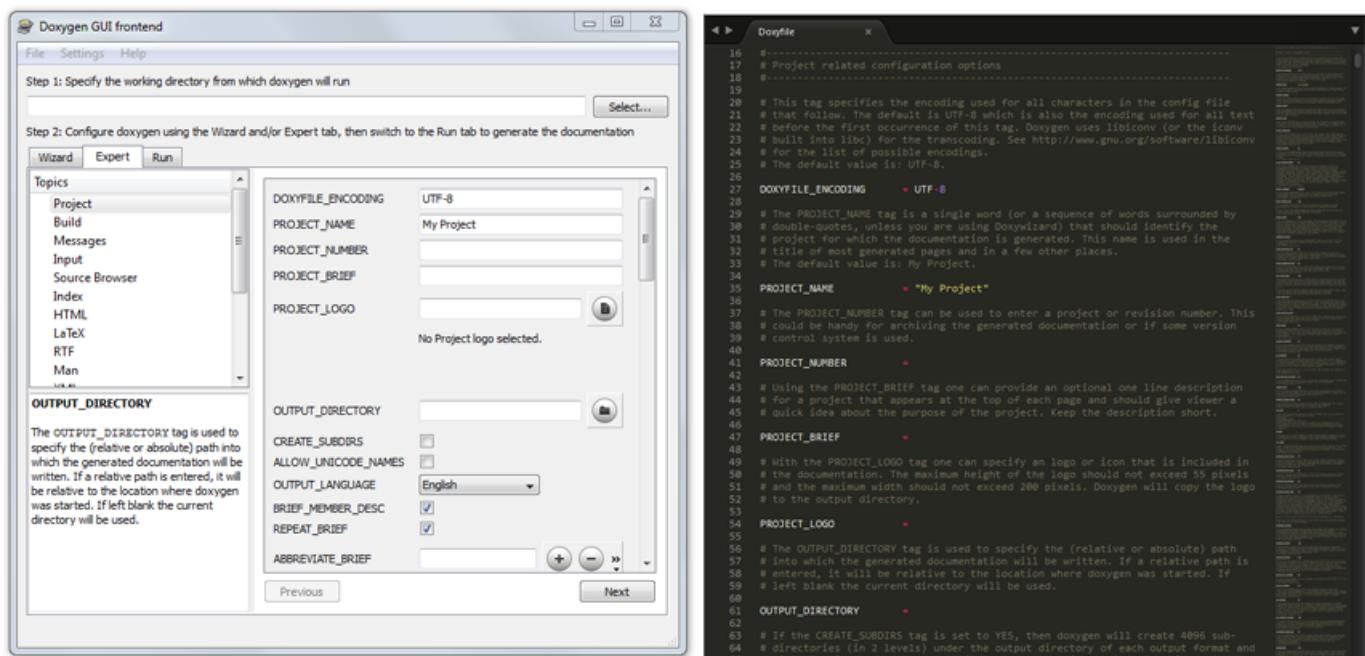
Скачать последнюю версию Doxygen можно на официальном сайте, дистрибутивы которой доступны для большинства популярных операционных систем, кроме того, вы можете воспользоваться вашим пакетным менеджером. Помимо этого для комфортной и полнофункциональной работы рекомендуется установить Graphviz.

Далее работа с Doxygen весьма тривиальна: достаточно запустить программу, указав ей путь к файлу с настройками.

```
doxygen <config_file>
```

Но в этом файле и вся тонкость. Дело в том, что каждому проекту соответствует свой файл настроек, в котором может быть прописан путь до исходников проекта, путь, по которому должна быть создана документация, а также большое число различных опций, которые подробно описаны в документации, и которые позволяют максимально настроить документацию проекта под свои нужды.

В принципе, для редактирования данного файла и, вообще, работой с Doxygen, можно воспользоваться программой Doxywizard, которая чаще всего идёт вместе с Doxygen и которая позволяет чуть удобнее работать с файлом настроек (слева – Doxywizard; справа – файл открытый в текстовом редакторе):



Итак, приступим к созданию файла с настройками. Вообще, если вы используете Doxywizard, то он будет создан автоматически, в противном случае для создания этого файла необходимо запустить программу Doxygen с ключом `-g` (от generate):

```
doxygen -g <config_name>
```

Рассмотрим основные опции, которые могут вам пригодится, чтобы создать первую вашу документацию:

Тэг	Назначение	По умолчанию
DOXYFILE_ENCODING	Кодировка, которая используется для всех символов в данном файле настроек	UTF-8
OUTPUT_LANGUAGE	Устанавливает язык, на котором будет сгенерирована документация	English
PROJECT_NAME	Название проекта, которое может представлять собой единое слово или последовательность слов (если вы редактируете вне Doxywizard, последовательность слов необходимо поместить в двойные кавычки)	My Project
PROJECT_NUMBER	Данный тэг может быть использован для указания номера проекта или его версии	–
PROJECT_BRIEF	Краткое однострочное описание проекта, которое размещается сверху каждой страницы и даёт общее представление о назначении проекта	–
OUTPUT_DIRECTORY	Абсолютный или относительный путь, по которому будет сгенерирована документация	Текущая директория
INPUT	Список файлов и/или директорий, разделенных пробелом, которые содержат в себе исходные коды проекта	Текущая директория
RECURSIVE	Используется в том случае, если необходимо сканировать исходные коды в подпапках указанных директорий	NO

После того, как мы внесли необходимые изменения в файл с настройками (например, изменили язык, названия проекта и т.п.) необходимо сгенерировать документацию.

Для её генерации можно воспользоваться Doxywizard (для этого необходимо указать

рабочую директорию, из которой будут браться исходные коды, перейти на вкладку «Run» и нажать «Run doxygen») или запустив программу Doxygen, указав ей в качестве параметра путь к файлу с настройками:

```
doxygen <config_file>
```

Основы документирования на Doxygen

Теперь, когда мы разобрались с тем, как настраивать Doxygen и работать с ним, пора разобраться с тем, как необходимо документировать код, основными принципами и подходами.

Документация кода в Doxygen осуществляется при помощи документирующего блока. При этом существует два подхода к его размещению:

1. Он может быть размещён перед или после объявления или определения класса, члена класса, функции, пространства имён и т.д.;
2. Либо его можно располагать в произвольном месте (и даже другом файле), но для этого потребуется явно указать в нём, к какому элементу кода он относится. Мы не будем рассматривать этот подход, поскольку даже разработчики рекомендуют его избегать, но если интересно, то подробнее о нём можно прочитать в документации.

Структурно, любой документирующий блок является комментарием, просто оформленным специальным образом, поэтому естественно, что его вид зависит от используемого языка (подробнее об этом можно прочитать в соответствующем разделе документации). Поэтому далее мы остановимся на рассмотрении синтаксиса для С-подобных языков (C/C++/C#/Objective-C/PHP/Java).

Сразу отметим, что, вообще, всего существует два основных типа документирующих блоков: многострочный блок и односрочный блок.

Разница между ними чуть более сильная, чем между односрочным и многострочным комментарием. Дело в том, что текст, написанный в односрочном блоке относится к краткому описанию документируемого элемента (сродни заголовку), а текст, написанный в многострочном блоке относится к подробному описанию. Про эту разницу не следует забывать.

Многострочный блок

Мы сказали, что любой блок – это комментарий, оформленный специальным образом.

Поэтому необходимо определить каким таким «специальным образом». Вообще, существует целый ряд способов для описания многострочного блока, и выбор конкретного способа зависит от ваших предпочтений:

1. JavaDoc стиль (напоминает обычный С комментарий, но начинающийся с двух звездочек):

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

При этом звездочки не обязательно ставить на каждой строке. Такая запись будет эквивалентной:

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

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

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

Сказанное о необязательности промежуточных звездочек также остаётся справедливым. Помимо названных двух стилей есть ещё ряд, но на них пока мы не будем останавливаться.

При этом ещё раз обратите внимание на то, что текст написанный в таком комментарии относится к подробному описанию.

Для указания краткого описания может быть использована команда `\brief`. Указанный после команды текст, вплоть до конца параграфа будет относится к краткому описанию, и для отделения подробного описания и краткого описания используется пустая строка.

```
/*!  
 \brief Краткое описание и  
 его продолжение.  
  
 Подробное описание  
 */
```

Однострочный блок

Для описания однострочного блока опять же существует целый ряд способов оформления, рассмотрим два из них:

1. Можно использовать специальный комментарий в C++ стиле:

```
// Краткое описание
```

2. Можно использовать аналогичный предыдущему комментарий, только вместо дополнительного слеша в нем ставится восклицательный знак:

```
//! Краткое описание
```

При этом хотелось бы обратить внимание на два момента:

1. Для указания подробного описания в однострочном документирующем блоке может быть использована команда `\details`:

```
// \details Подробное описание
```

2. 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  
*/
```

Да, Doxygen крайне гибок в плане способов документирования, однако не стоит этим злоупотреблять, и в рамках одного проекта всегда придерживайтесь заранее оговоренного единообразного стиля

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):

Класс Parent

Полный список членов класса

Родительский класс, не несущий никакой смысловой нагрузки [Подробнее...](#)

```
#include <classes.h>
```

Подробное описание

Родительский класс, не несущий никакой смысловой нагрузки

Данный класс является примером для того, как документируется наследование

Объявления и описания членов классов находятся в файлах:

- [classes.h](#)
- [classes.cpp](#)

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

Команды

С некоторыми из команд в Doxygen мы успели познакомиться (речь идёт о `\brief` и `\details`), однако на самом деле их значительно больше. Полный их список приведён в [официальной документации](#).

Вообще, любая команда в Doxygen представляет собой слово на английском языке предваренное символом "\\" или "@" (обе записи тождественны) и таких команд очень много, порядка двухсот. Приведём для примера несколько таких команд:

Команда	Значение
<code>\authors</code>	Указывает автора или авторов
<code>\version</code>	Используется для указания версии
<code>\date</code>	Предназначена для указания даты разработки
<code>\bug</code>	Перечисление известных ошибок
<code>\warning</code>	Предупреждение для использования
<code>\copyright</code>	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();  
};
```

Класс Son Полный список членов класса

Дочерний класс [Подробнее...](#)

```
#include <classes.h>
```

Подробное описание

Дочерний класс

Автор
Norserium

Версия
1.0

Дата
Март 2015 года

Предупреждения
Данный класс создан только в учебных целях

Обычный дочерний класс, который отнаследован от ранее созданного класса Parent

Объявления и описания членов классов находятся в файлах:

- classes.h
- classes.cpp

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 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.

Документирование основных элементов исходного кода

Теперь мы можем рассмотреть специфичные особенности документирования различных элементов исходного кода, начиная от файлов в целом и заканчивая классами, структурами, функциями и методами.

Документирование файла

Хорошей практикой является добавление в начало файла документирующего блока, описывающего 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 `\ragam` для каждого из параметров функции, при этом синтаксис команды имеет следующий вид:

```
\ragam [<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
\ragam[out] dest Целевая область памяти
\ragam[in] src Исходная область памяти
\ragam[in] n Количество байтов, которые необходимо скопировать
*/
void memcpuy(void *dest, const void *src, size_t n);
```

В результате мы получим такую вот аккуратную документацию функции:

```
void memcpv ( 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
\params a,b Складываемые числа
\return Сумму двух чисел, переданных в качестве аргументов
*/
double sum(const double a, const double b);
```

Получаем следующий результат:

Функции

```
double sum ( const double a,  
             const double b  
           )
```

Находит сумму двух чисел

Аргументы

a, b Складываемые числа

Возвращает

Сумму двух чисел, переданных в качестве аргументов

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 автоматически 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 неопределено
    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  
 */
```

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 файлам 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.

Код внутри документации

Зачастую внутри пояснения к документации необходимо для примера добавить какой-то код, например для иллюстрации работы функции.

Команды `\code` и `\endcode`

Один из наиболее простых и универсальных 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 a;  
}
```

```
b = r;  
}  
return r;  
}  
\endcode  
*/  
int gcd(int a, int b);
```

Результат будет иметь следующий вид:

Примеры кода

Иногда удобнее приводить примеры использования кода хранить в отдельных файлах. Для этого эти файлы необходимо разместить в отдельной директории и прописать к ней путь в настройках:

```
EXAMPLE_PATH = путь_к_директории
```

Рассмотрим, некоторые способы того, как примеры кода могут быть использованы в документации.

Команда \example

Данная команда показывает, что документирующий блок относится к примеру кода.

```
\example <имя_файла>
```

Текст исходного кода будет добавлен в раздел «примеры», а исходный код примера будет проверен на наличие документированных элементов, и если такие будут найдены, то к ним в описание будет добавлена ссылка на пример.

Титульная страница	Файлы	Примеры	<input type="text"/> Поиск
--------------------	-------	---------	----------------------------

main.cpp

Пример того, как использовать функцию

```
void main()
{
    int result = gcd(52, 106);
```

Функции

```
int gcd ( int a,
          int b
      )
```

Алгоритм Евклида

Аргументы

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.

Примеры:

[main.cpp](#).

▶ [Файл gcd.h](#)

▶ [Файл examples/main.cpp](#)

Команда \include

Для того, чтобы добавить в описание к документируемому элементу код примера используется 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 оформлению кода в блок начинающейся командой `\code` и заканчивающейся командой `\endcode`).

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 документируемого объекта

Наконец, 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
```

Формулы с использованием LaTeX

Doxygen позволяет использовать TeX формулы прямо в документации, это очень удобно и результат получается весьма достойным. Однако стоит отметить, что при этом имеются ограничения: на данный момент формулы могут быть вставлены только в HTML и LaTeX документацию, но этого, как правило, вполне достаточно.

На данный момент существует два подхода к отображению формул:

1. Отображение формул при помощи MathJax, для этого необходимо в файле настроек установить соответствующую опцию:

```
USE_MATHJAX = YES
```

2. Генерация соответствующих изображений и вставка их в документацию. Всё это будет сделано автоматически, но вам потребуется следующие инструменты: *Latex*, *dvips*,

gs. По умолчанию формулы отображаются именно этим способом.

Способы добавление формул в документацию

Существуют три способа добавления формул в документацию. Последовательно рассмотрим каждый из них с примерами из документации:

- Использование строчных формул, которые обрамляются в начале и в конце при помощи команды "\f\$". Пример представлен ниже:

```
расстояние между \f$(x_1,y_1)\f$ и \f$(x_2,y_2)\f$ равно \f$\sqrt{(x_2-x_1)^2}
```

Результатом будет строка следующего вида: расстояние между и равно

- Использование выносных формул, которые начинаются на отдельной строке и центрируются. В отличие от предыдущих формул они обрамляются в начале командой "\f[", а в конце командой "\f]". Пример представлен ниже:

```
\f[
|I_2|=\left| \int_{\psi(t)}^{\psi(a)} \left( \gamma(t) - \frac{d\theta}{dt} k(\theta, t) \right) dt \right|
```

Результатом будет строка следующего вида:

- Существует команда "\f{environment}", где *environment* – это название определенного окружения в LaTeX. Она позволяет использовать указанное окружение будто бы оно было указано в обычном LaTeX документе. Пример приведён ниже:

```
\f{eqnarray*}{

g &=& \frac{Gm_2}{r^2} \\
```

```

&=& \frac{(6.673 \times 10^{-11} \cdot \text{m}^3 \cdot \text{kg}^{-1} \cdot
      \text{s}^{-2})(5.9736 \times 10^{24} \cdot \text{kg})}{(6371.01 \cdot \text{m})^2
&=& 9.82066032 \cdot \text{m/s}^2
\}

```

В итоге мы получим следующий результат (заметим, что окружение `eqnarray*` – это ненумерованное окружение для размещения нескольких формул):

Пример внедрения формул в документацию

Рассмотрим конкретный пример документации с использованием формул LaTeX:

```

/*
\brief Вычисление факториала числа \f$ n \f$
\param n - число, чей факториал необходимо вычислить
\return \f$ n! \f$

Данная функция вычисляет значение факториала числа \f$ n \f$, определяемое по формуле:
\f[
n! = \prod_{i = 1}^n i
\f]
*/
int factorial(int n);

```

Результат представлен ниже:

Кратко о Markdown

Markdown – это облегчённый язык разметки (почитать о нём можно, например, здесь, а также в специальном разделе в документации). Начиная с версии 1.8.0. Doxygen обеспечивает его пока ограниченную поддержку и он служит одним из способов оформить документацию (альтернативой могут быть, например, команды для оформления документации или HTML вставки, которые, впрочем, не универсальны).

Не хотелось бы сейчас расписывать подробности и принципы данного языка, поэтому ограничимся рассмотрением того, как данный язык позволяет «украсить» нашу

документацию:

```
/*
Функция генерирующая псевдослучайное число
```

Изначально планировалось реализовать в данной функции один из следующих методов генерации псевдо-

- Линейный конгруэнтный метод;
- Метод Фибоначчи;
- Линейный регистр сдвига с обратной связью;
- Вихрь Мерсенна.

Но разработчики вспомнили про одну замечательную цитату:

> Есть два способа создания дизайна программы. Один из них, это сделать его настолько простым,
> – С.А. Р. Ноаге

И выбрали первый путь.

```
![Описание функции](image.png)
*/
int getRandomNumber();
```

Результат представлен ниже:

Подводя итоги

На этой ироничной ноте я решил остановиться. Я прекрасно понимаю, что ещё многое не было описано и затронуто, но, надеюсь, что главные свои цели статья выполнила: познакомить с понятием генератора документации, познакомиться с системой Doxygen, объяснить основные принципы и подходы к документации, а также мельком затронуть вопросы, связанные с её оформление и детализацией, подготовив задел для вашей дальнейшей работы.

Спасибо за внимание!

Литература и ссылки для дальнейшего изучения



1. Основным источником, который был использован при написании статьи была официальная документация;
2. На большое количество вопросов, связанных с Doxygen, ответы были получены здесь (там есть и создатель Doxygen).

3. Продолжение цикла: оформление внешнего вида документации, построение графов и диаграмм в Doxygen.

Теги: doxygen, документирование кода, документация, C++, C#, Python, Java, PHP

Хабы: Программирование, C++, C, C#

Редакторский дайджест

Присыпаем лучшие статьи раз в месяц



Электропочта



52

0

Карма Рейтинг

@Norsegium

Пользователь

Подписаться



Github



Комментарии 39

Публикации

ЛУЧШИЕ ЗА СУТКИ ПОХОЖИЕ



Cloud4Y

18 часов назад

Как запустить Windows 95 на одноразовом вейпе

Простой

15 мин

8.2K

Туториал

Перевод

+79

30

25



pokrovsk

19 часов назад

Как я тестировал российские фоторедакторы, полжизни проработав в Фотошопе



Простой



9 мин



15K

Обзор



+55



32



75



sendelust

18 часов назад

Как начать писать на Java в VSCode



Простой



11 мин



5.1K

Туториал



+51



120



34



Stefanio

19 часов назад

Как потреблять API с ограничением по RPS в .NET приложениях



Простой



11 мин



2.2K

Туториал



+42



49



1



ru_vds

15 часов назад

Нельзя предполагать, что все используют UTF-8



Средний



6 мин



4K

Мнение

Перевод



+34



18



25



SloNN

22 часа назад

Как Яндекс создал свою шину данных, чтобы передавать сотни гигабайт в секунду

 Простой 7 мин 11К

Роадмэп

 +33 50 19**technono_mot**

18 часов назад

Шаг за шагом: разработка 3D-игры в Godot 4.2 для начинающих

 7 мин 2.4К +32 26 3**pkolt**

21 час назад

Метеостанция на ионисторе

 Простой 10 мин 4.9К

Из песочницы

 +31 40 24**MaFrance351**

20 часов назад

Оживляем раритетный домофон с магнитным ключом

 Простой 9 мин 2.3К

Обзор

 +29 9 8**asolovskyev24**

17 часов назад

Удаленка. Мой путь к выгоранию (и обратно)

 7 мин 5.8К +21 33 29

Проблемы со связью и боязнь облаков: а при чём тут DevOps?

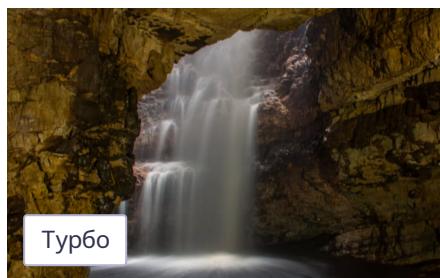
Турбо

[Показать еще](#)

МИНУТОЧКУ ВНИМАНИЯ



IT-события, которые ты ищешь, тоже ищут тебя



Вода, которую мы не знаем: что скрывается в грунтовых потоках



Глупым вопросам и ошибкам — быть! IT-менторство на ХК

ЗАКАЗЫ

Переписать бота тг с питона на Go по ветке из гита
20000 руб./за проект · 3 отклика · 20 просмотров

Установщик на INNO SETUP
1000 руб./за проект · 1 отклик · 16 просмотров

Переписать скрипт Python на Node.js
1500 руб./в час · 7 откликов · 42 просмотра

Разработать Telegram mini app
30000 руб./за проект · 40 откликов · 355 просмотров

Обучение модели нейронной сети процессу примерки одежды
25000 руб./за проект · 5 откликов · 31 просмотр

Больше заказов на Хабр Фрилансе

bank.yandex.ru РЕКЛАМА

**Ноль причин
откладывать покупку**

[Перейти на сайт](#)

ЧИТАЮТ СЕЙЧАС

Попросил нейросети собрать игровой ПК за 100 000 рублей. Вот что из этого получилось

 143K  122

Как я тестировал российские фоторедакторы, полжизни проработав в Фотошопе

 15K  75

В Steam стартовал первый фестиваль российских игр

 2.9K  6

Число пользователей Тог в РФ снизилось двукратно

 11K  21

Как Яндекс создал свою шину данных, чтобы передавать сотни гигабайт в секунду

 11K  19

Проблемы со связью и боязнь облаков: а при чём тут DevOps?

Турбо

ИСТОРИИ



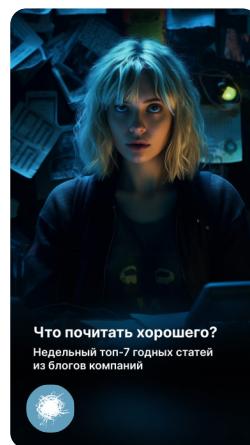
Как приходят идеи крутых статей



Зовём участвовать в UX-тестах



Активность найма в 1 квартале 2024



Годнота из блогов компаний



Как продвинуть машину времени?

РАБОТА

Программист C
38 вакансий

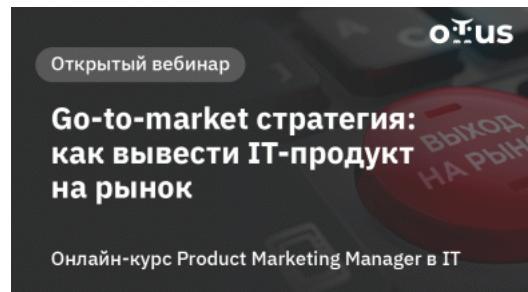
QT разработчик
9 вакансий

Программист C# удаленно
100 вакансий

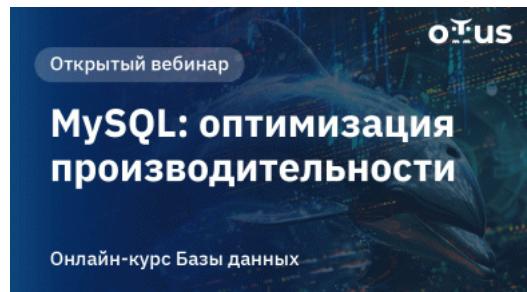
Программист C++
125 вакансий

[Все вакансии](#)

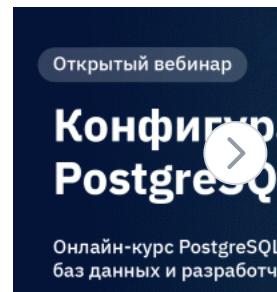
БЛИЖАЙШИЕ СОБЫТИЯ



Вебинар «Go-to-market стратегия: как вывести IT-



Вебинар «MySQL: оптимизация производительности



Практический у конфигурации PostgreSQL

03.05.2024, 11:10

продукт на рынок»

Документируем код эффективно при помощи Doxygen / Хабр

производительности»

14 мая 20

13 мая 20:00

14 мая 19:00

Онлайн

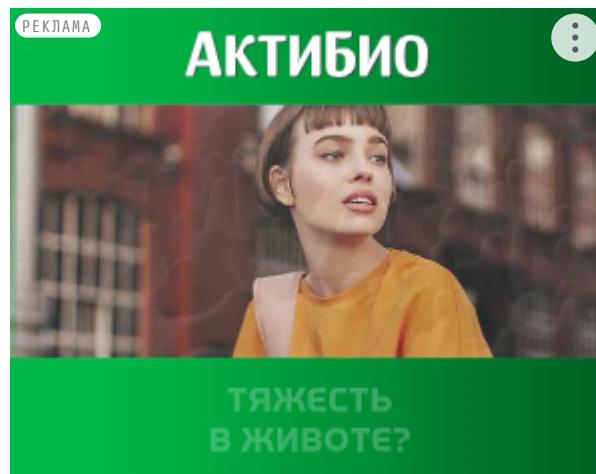
Онлайн

Онлайн

[Подробнее в календаре](#)

[Подробнее в календаре](#)

[Подробнее в кален.](#)



Ваш аккаунт

[Войти](#)

[Регистрация](#)

Разделы

[Статьи](#)

[Новости](#)

[Хабы](#)

[Компании](#)

[Авторы](#)

[Песочница](#)

Информация

[Устройство сайта](#)

[Для авторов](#)

[Для компаний](#)

[Документы](#)

[Соглашение](#)

[Конфиденциальность](#)

Услуги

[Корпоративный блог](#)

[Медийная реклама](#)

[Нативные проекты](#)

[Образовательные](#)

[программы](#)

[Стартапам](#)



[Настройка языка](#)

[Техническая поддержка](#)

© 2006–2024, Habr