Стандартны оформления программного кода

Лекция по дисциплине «Программирования на С»

Стиль кодирования

- **Стандарт оформления кода** – набор правил и соглашений, используемых при написании исходного кода на некотором языке программирования. Наличие общего стиля программирования облегчает понимание и поддержание исходного кода, написанного более чем одним программистом, а также упрощает взаимодействие нескольких человек при разработке программного обеспечения.

- **Целью** создания стандарта кодирования является добиться такого положения, когда программист мог бы дать заключение о функции, выполняемой конкретным участком кода, а в идеале — также определить его корректность, изучив только сам этот участок кода или, во всяком случае, минимально изучив другие части программы. Иными словами, смысл кода должен быть виден из самого кода, без необходимости изучать контекст. Поэтому стандарт кодирования обычно строится так, чтобы за счёт определённого визуального оформления элементов программы повысить информативность кода для человека.

Стандарт

Обычно, стандарт оформления кода описывает:

- способы выбора названий и используемый регистр символов для имён переменных и других идентификаторов:
- запись типа переменной в её идентификаторе (венгерская нотация);
- регистр символов (нижний, верхний, «верблюжий», «верблюжий» с малой буквы), использование знаков подчёркивания для разделения слов;
- стиль отступов при оформлении логических блоков — используются ли символы табуляции, ширина отступа;
- способ расстановки скобок, ограничивающих логические блоки;
- использование пробелов при оформлении логических и арифметических выражений;
- стиль комментариев и использование документирующих комментариев.

Вне стандарта подразумевается:

- отсутствие магических чисел;
- ограничение размера кода по горизонтали (чтобы помещался на экране) и вертикали (чтобы весь код файла держался в памяти), а также функции или метода в размер одного экрана.

Венгерская нотация

- **Венгерская нотация** – соглашение об именовании переменных, констант и прочих идентификаторов в коде программ. Своё название венгерская нотация получила благодаря программисту компании *Microsoft* венгерского происхождения *Чарльзу Симони*. Эта система стала внутренним стандартом Майкрософт.

- **Суть** венгерской нотации сводится к тому, что имена идентификаторов предваряются заранее оговорёнными префиксами, состоящими из одного или нескольких символов. При этом, как правило, ни само наличие префиксов, ни их написание не являются требованием языков программирования, и у каждого программиста (или коллектива программистов) они могут быть своими.

Таблица префиксов

Префикс	Сокращение от	Смысл	Пример
s	string	строка	sClientName
sz	zero-terminated string	строка, ограниченная нулевым символом	szClientName
n, i	int	целочисленная переменная	nSize, iSize
l	long	длинное целое	lAmount
b	boolean	булева переменная	bIsEmpty
a	array	массив	aDimensions
p	pointer	указатель	pBox
h	handle	дескриптор	hWindow
m_	member	переменная-член	m_sAddress
g_	global	глобальная переменная	g_nSpeed
T	type	тип	TObject
I	interface	интерфейс	IDispatch
v	void	отсутствие типа	vReserved

Стили написания составных слов

- **CamelCase** — стиль написания составных слов, при котором несколько слов пишутся слитно без пробелов, при этом каждое слово пишется с заглавной буквы. Стиль получил название *CamelCase*, поскольку заглавные буквы внутри слова напоминают горбы верблюда.
- Альтернативным способом создания имён в программном код **spinal_case** (несколько слов разделяются символом нижнего подчеркивания '_') и **snake-case** (используется тире '-').

ПРИМЕР

Используемый язык

- Имена файлов, проектов, папок, сам текст программы в том числе и комментарии должны быть записаны и использованием букв латинского (английского) алфавита.
- Не допускается использование транслита — передача текста с помощью чужого алфавита. Все должно быть написано с использованием правил английского языка.
- Почему?

Имена файлов и папок

- Имена файлов должны отражать реализацию/объявление модулей/функций. Используется CamelCase. Имена файлов должны быть уникальными в пределах проекта. Используйте следующие расширения:

  **.cpp – C++;*

  **.c – C;*

  **.h – заголовочный файл;*

  **.inl – inline файл;*

- Имена папок должны быть в стиле **snake_case**, но старайтесь использовать одно слово в именах, чтобы не использовать знак подчеркивания. Почему нет побелов?

Оформление логических блоков

- Все исходные файлы (файлы кода) должны иметь следующую структуру:
- - Авторство (c*opyright*);
- - Описание (*documentation*);
- - Подключаемые заголовочные файлы (*include headers*);
- - Предварительное объявление (*forward declarations*);
- - Локальное объявление типов (*locally declared types*);
- - Локальные константы и макросы (*local constants and macros*);
- - Локальные статически и глобальные переменные (*local static / global variables*).
- - Реализация функций (*functions implementation*).
- Подключайте минимально требуемое количество заголовочных файлов. Не руководствуйтесь словами: "на всякий случай". Почему?

```
/*
 * Copyright Student Company.
 * All rights reserved.
 */

/// Documentation ////////////////////////////////////////////////////////

/*
 * This module is used for bla-bla
 * ...
 */

// Headers
#include "Module.h"

// Global headers
#include "Macro.h"

// Standard headers
#include <stdlib.h>
#include <string.h>

#define MACRO_ADD(a,b)   ((a) + (b))

typedef struct
{
        ...
} SDataStruct;


uint32 g_globalVariable = 0;
static uint32 s_staticVariable = 123;

static void InternalFunction( void );

int32 ExternalFunction( int32 a, double b )
{
        ...
}

void InternalFunction( void )
{
        ...
}
```

Оформление заголовочного файла

Структура заголовочного файла:

- Информация о модуле;
- Проверка множественного включения (***#ifdef***);
- Препроцессорные директивы (***#include, #define***);
- Предварительное объявление (***forward declarations***);
- Локальное объявление типов/структур (***locally declared types***);
- Проверка множественного включения (***#endif***).

```c
/*
 * Copyright Student Company. 2016.
 * All rights reserved.
 */

/*
 * Example C-header file
 */

#ifndef __HEADER_EXAMPLE_H
#define __HEADER_EXAMPLE_H

//// Includes ////////////////////////////////////////////////////////////////

#include "Base.h"

#define SQR(x) ((x)*(x))

#include "Module.h"

//// Forward Declarations ////////////////////////////////////////////////////

struct SNextStruct;

//// Definitions and Declarations ////////////////////////////////////////////

typedef struct SInfoStruct
{
    int counter;
    struct SNextStruct * next;
} SInfoStruct;

// Example function
int ExampleFunc( const char* param1, int param2 );


#endif //   HEADER_EXAMPLE_H
```

Типы переменных

- Используйте следующие базовые типы:

Type	Basic type	Type	Basic type
int8	знаковое 8-битное целое число	float	вещественное число
uint8	беззнаковое 8-битное целое число	double	вещественное число с двойной точностью
int16	знаковое 16-битное целое число	bool	логический
uint16	беззнаковое 16-битное целое число	char	Символ
int32	знаковое 32-битное целое число	const char *	константная строка
uint32	беззнаковое 32-битное целое число	void*, uint32*,...	указатель
int64	знаковое 64-битное целое число		
uint64	беззнаковое 64-битное целое число		
int	знаковое целое число		

Имена структур должны начинаться с буквы **'S'** – structure с использование *CamelCase* (например, **SPoint**) ; **'E'** - enumiration, перечисление; **'U'** - union.

Имена переменных

- В именах переменных возможно присутствие префиксов *'g_'* для обозначение глобальных переменных и *'s_'* – для статических.
- Имена переменных должны отражать содержащую информацию.
- Используемый стиль ***camelCase*** (первая буква всегда маленькая).
- Не используйте знак подчеркивания '_', кроме тех случаев, где это явно разрешено.
- Не используйте придуманные аббревиатуры, разрешено только применять только известные, например, ***rgb***, ***lcd***.

Константы

Все буквы в имени констант должны быть в верхнем регистре (большими) и в качестве разделителей должны быть использованы знаки подчеркивания '_'. Данное правило применяется только к макросам и базовым типам.

```
#define DPOF_TOTAL_SIZE 100
static const uint32 BUFFERS_COUNT = 2;
```

Во избежание создания магических чисел (с комментариями или без) используйте смысловое имя константы.

Константы "сложных" типов следует писать в стиле **camelCase**, и применение префиксов *'g_'* и *'s_'* нецелесообразно, в силу того, что эти данные не могут быть измены, поэтому на программиста не возлагается работа по потокобезопасности

```
typedef struct
{
    int32 x;
    int32 y;
} SPoint;

const SPoint upperLeftCorner =
{
    .x = 0,
    .y = 0
};
```

Макросы

Все буквы в имени макросов должны быть в верхнем регистре (большими) и в качестве разделителей должны быть использованы знаки подчеркивания '_'.

```
#define CALC_ADD(a,b)                    ( (a) + (b) )
```

Многострочные макросы следует писать столбцом для увеличения читабельности:

```
#define CALC(a)          ( (input) + 10 + \
                           (input) * 20 + \
                           (input) / 30 )
```

Многострочные макросы, которые могут быть использованы как один оператор (одно действие) следует писать в блоке ***'do {…} while (0)'***.

```
#define MULTI_LINE_MACRO( )                \
                        do                 \
                        {                  \
                            ...            \
                        } while( 0 )
```

Макросы *(продолжение)*

Это позволит использовать макрос в следующей конструкции:

```
if ( ... )
    MULTI_LINE_MACRO();
else
    MULTI_LINE_MACRO();
```

Функции (размер, имена)

- Размер функции не должен быть большим, это осложняет понимания. Если функция слишком большая следует применить **рефакторинг** для уменьшения размера.
- Имена функций должны иметь префик, который обозначает модуль/библиотеку, к которому он относится. Также это требуется для исключения одинаковых имен функций в программе. Исключением является стандартные функции и функции из сторонних библиотек.
- Также как и имя переменных имена функции несут смысловую нагрузку. Стиль *CamelCase*.

Форматирование кода

Длина строк кода

Длина строк кода не должна превышать **100** символов. В случае слишком длинной строки она может быть разделена на несколько строк, что влечет за собой уменьшения читабельности.

Табуляция

Не используйте знак табуляции, только в качестве **4** пробельных символов. Почему?

Отступы

Код должен быть всегда отформатирован – код в каждом блоке должен быть смещен относительно символа начала и конца блока. В случает, если вложенность достигает **4-5**, то возможно следует провести **рефакторинг**.

Пробелы

Используйте пробелы для разделения операторов и их операндов.

```
count = 100 + total;
```

Не пишите/печатайте пробелы в конце строк. А также перед запятой и точкой с запятой.

Рекомендуется, вставлять пробел после открывающей круглой скобки и перед закрывающей скобкой в конструкция *if, switch, while, for.*

Разделяйте ключевые слова (конструкции *if, switch, while, for*) от условий:

```
// OK
if ( condition )
{
}

// BAD
if( condition )
{
}
```

Пробелы *(продолжение)*

Не разделяйте имя функции от скобок:

```
// OK
Func( int arg );

// BAD
Func ( int arg );
```

Комментарии

- Комментарии использовать нужно. Комментировать можно отдельные куски кода, функции, всякие оригинальные трюки используемые вами, но стоит избегать комментирования всего подряд.
- Используйте ***TODO***-комментарий для описания временного или недостаточного хорошо реализованного программного кода, к которому Вы планирует вернуться для доработки:

```
// TODO: Refactor this code in the future
```

Комментарии *(продолжение)*

- Другие типы тегов-комментариев:

  FIXME – код должен быть исправлен.

  HACK – обходной приём (workaround).

  UNDONE – откат ("roll back") на предыдущую версию кода.

  XXX – предупреждение для программистов об сложно или опасном коде

  UX – уведомление о нетривиальном коде

Замечание. Обходной приём, workaround, паллиатив, на техническом жаргоне — «костыль» — относительно быстрое и простое решение проблемы, применяемое для срочного устранения её последствий, но не влияющее на причины её возникновения. Обходной приём обычно является временным, или неполным решением, не отвечающим требованиям к дальнейшему развитию системы, требующим в дальнейшем замены на окончательное, более полное (**wikipedia**).