Содержание

Введение

Документ описывает правила оформления исходных текстов. Стандартизация стиля преследует следующие цели:

- Сделать удобным для членов команды разработчиков изучение кода, написанного другими членами команды.
- Зафиксировать некоторое “подмножество” выразительных средств языка C++. Данный язык предоставляет чрезмерно большое количество средств, с помощью которых можно описать то или иное программное решение. Поэтому для продуктивной работы группы программистов в рамках одного проекта, весьма полезно договориться о том, какие из приемов программирования на C++ будут использоваться в проекте, а каких следует избегать.
- Унифицировать правила использования различных технологических приемов сборки проекта.
- Выдержать исходные тексты всего проекта в едином стиле.

Несмотря на то, что программирование является творческим процессом и всегда предусматривает определенную свободу самовыражения для каждого члена команды разработчиков, тем не менее, для того, чтобы в условиях промышленного производства проект был жизнеспособным, необходимо выдержать в едином стиле все модули исходных текстов, независимо от того, кем именно из участников проекта был написан тот или иной модуль.
Все без исключения участники проекта имеют полное право совещательного голоса. Любые предложения по изменению действующих правил обсуждаются командой. Если эти изменения находятся полезными для проекта, то после внесения их в настоящий документ все участники проекта обязаны следовать новым правилам.

Соглашения, принятые в документе

Ниже приведены примеры специальных знаков, которые указывают на характер информации. В тексте каждого абзаца дано соответствующее пояснение.

 “Запрещено”. Приемы и стиль программирования, описанные в таком разделе, не могут быть использованы в данном проекте.

 “Обязательно”. Некоторые правила, описанные в документе, являются обязательными в рамках конкретного проекта, т.к. от их соблюдения существенно зависит целостность и жизнеспособность всего проекта.

 “Крайне рекомендуется”. Таким знаком отмечены приемы, стиль и т.д. которые рекомендуются для использования в проекте. Соблюдение таких правил приводит к тому, что весь проект воспринимается как цельное изделие, а не как “лоскутное одеяло”.

 “Следует избегать”. С другой стороны, в документе встречаются описания популярных ранее приемов и стилей, которые в настоящее время рассматриваются как устаревшие или не соответствующие общему стилю проекта. Таких приемов следует избегать.

Общие правила

Избегайте использования глобальных переменных.

Практически всегда, в соответствии с конструкцией программы, у каждой глобальной переменной существует “хозяин”: функция, класс, модуль. Наилучшее решение – это статические данные класса.

хуже

```
int GlobalVar;

void FooToDo()
{
    GlobalVar = NewValue();
}
```

лучше

```
class Foo
{
    int m_GlobalVar;
 public:
    void GlobalVar( int val )     { m_GlobalVar = val; }
    int  GlobalVar() const       { return m_GlobalVar; }
    void ToDo()             { GlobalVar( NewValue() ); }
};
```

Если существует явная необходимость в общем поле глобальных ресурсов, то должен быть создан отдельный модуль, который и будет “хозяином”.

Запрещается прямое объявление и обращение к глобальной переменной из тела другого модуля, в обход *.h файла того модуля, где эта переменная объявлена и определена.

```
// Module1.h
extern int globalFlag;     // объявление

// Module1.cpp
#include "Module1.h”
int globalFlag = 0;        // определение и инициализация
...

// Module2.cpp
extern int globalFlag;     // Запрещено! Используйте #include "Module1.h"
void Foo()
{
    globalFlag++;
}
```

Избегайте использования #define для объявления констант. Используйте константные выражения C++.

хуже

```
#define  MAX_BUFFER_SIZE       100
```

лучше

```
const int MaxBufferSize = 100;
```

Все интерфейсные файлы (.h) должны быть прокомментированы в стиле Doxygen. Комментарии должны быть только на русском языке. Исключением являются сторонние модули и библиотеки, использованные в проекте.

Окончание строк в файлах желательно оформлять в Unix-стиле (только символ конца строки с кодом #10, без символа перевода строки с кодом #13).

Формирование Имен

Общие правила

Выбирайте осмысленные имена, которые наиболее точно отражают назначение того или иного объекта в программе. Длина идентификатора определяется исключительно удобством при чтении кода.

Аргументы, связанные с удобством набора кода, не принимаются во внимание. Например аргумент “переменные называются b0, x2, потому что так короче писать”, не может быть использован ни при каких обстоятельствах.

Сокращайте до минимума количество аббревиатур в именах. Даже если Вам кажется, что данная аббревиатура понятна и естественна, это может не быть столь очевидным для другого человека, который будет читать Ваш код.

При формировании имен, очень осторожно используйте цифры и буквы латинского алфавита, которые имеют схожее с цифрами начертание.

неверно

```
int  lO = 10;
int  l0 = lO;
```

верно

```
int  Lo = 10;
int  L20 = Lo;
```

Используйте современный механизм “namespace” вместо префиксов имен для предотвращения возможных конфликтов между модулями.

неверно

```
class coreVectorArray;
...
class guiWindowFrame;
...

void Foo()
{
     coreVectorArray   CurrList;
     guiWindowFrame    WinXept;
     ...
}
```

верно

```
namespace Core
{
     class VectorArray;
};
...
namespace Gui
{
     class WindowFrame;
};
...

using namespace Core;
void Foo()
{
     TVectorArray          currList;
     Gui::TWindowsFrame    winFrame;
     ...
}
```

Не используйте “Венгерскую” нотацию[1] при формировании имен. Это может стать проблемой в том случае, если появится необходимость изменить тип. Кроме того, «венгерская» нотация была действительно полезна только в языке со слабым контролем типов, каким был, например, язык C Кернигана и Ричи. Исключением являются имена переменных указателей (начинаются с префикса p) и имена значений перечислимых типов (см. ниже).

[1] “Венгерская” нотация – способ формирования имен переменных, при котором с помощью префиксов давалось указание на тип. Например, iCount – имеет тип INT, dwCount – DWORD, lpcszFName – const char FAR*

Имена для препроцессора

Все имена, предназначенные для обработки препроцессором, пишутся прописными буквами. Части составного имени разделяются символом подчеркивания.

хуже	лучше
`#define er_object_NotCreated`	`#define ER_OBJECT_NOT_CREATED`

Имена классов, структур, перечислений, функций

Все имена типов и функций начинаются с прописной буквы. Составные имена пишутся слитно, каждая часть начинается с заглавной буквы. Для имен классов и структур используется префикс “T”, для имен перечислений – префикс “E”.

```
class  TCoreVectorObject;
struct SomeInternalDataStruct;
enum   ERetCodes { ... };
void   SomeMemberOrGlobalFunction();
```

При формировании имен значений перечислимого типа настоятельно рекомендуется использование префикса, показывающее принадлежность данного имени конкретному перечислимому типу.

```
enum EOpenMode { omRead, omWrite, omReadWrite };
enum EIOAction { ioSeek, ioRead, ioWrite };
```

Имена переменных и констант

Имена переменных начинаются со строчной буквы, кроме случая, когда имя является устоявшимся в проекте термином (пример, L – число разверток). Имена констант начинаются с прописной буквы.

```
int    memberVariable;
int*   pMemberPointer;
int    i, j;
```

Специальные соглашения

- **Функции доступа к закрытым переменным**

  Для формирования имен функций доступа к закрытым переменным класса используется следующий способ:

  - функции чтения и записи имеют префиксы Set и Get соответственно:

```
class Foo
{
    int objectSize;
 public:
    void SetObjectSize(int newSize);
    int  GetObjectSize() const;
    ...
};
```

- **Функции, возвращающие логические значения**

  Функциям, возвращающим логические значения, давайте имена с префиксом “Is”.

```
bool IsVisible() const;
bool IsValid() const;
```

Форматирование

Общие правила

☞ Длина строки в файле исходного текста не должна превышать **100 символов.**

☞ В среде разработки (или в используемом текстовом редакторе) должна быть настроена замена символа табуляции на пробелы. Стандартный размер отступа – **2** знака.

👍 Важно, чтобы при чтении кода хорошо просматривались отдельные функции и логические секции. Для этого используются соответствующие разделители:

- Между соседними функциями используйте разделитель вида:

```
// ---------------------------------------------------------------------------
```

- В качестве разделителя секций используйте конструкцию вида:

```
/* ========================================================================== *\
**  Constants                                                                 **
\* ========================================================================== */
```

☞ Используйте пробелы в выражениях для того, чтобы текст был более удобен для чтения. Чрезмерное увлечение пробелами также вредно, как и полное их отсутствие.

неверно

```
for(i=0;i<10;i++){
    Var+=(((i<<2)*delta)+(b/10))>>1;
}

for ( i = 0 ;  i < 10 ;  i++ )  {
    Var += ( ( ( i << 2 ) * delta ) +
           ( b / 10 ) )  >> 1 ;
}
```

верно

```
for (i = 0; i < 10; i++)
{
  Var += (((i << 2) * delta) + (b / 10)) >> 1;
}
```

☞ При написании функций не ставить пробелы ни после имени функции, ни внутри скобок.

неверно

```
void FooVoid ();
void Foo (int a, char b, double c);
```

верно

```
void FooVoid();
void Foo(int a, char b, double c);
```

Размещение ограничителей блока

☞ Согласно общему правилу, фигурные скобки – ограничители блока – должны располагаться на отдельных строках и в одной колонке, соответствующей текущему отступу (уровню вложенности блока). Общее правило подразумевается везде, где не указано иное.

```
void FooFunc()
{
    if ( ... )
    {
        while (...)
        {
            ...
        }
    }
} // FooFunc
```

- Короткие inline функции могут быть целиком написаны на одной строке. Если длина всего предложения превышает установленную ширину текста, необходимо следовать общему правилу форматирования функций.

```
struct Foo
{
     bool IsVisible() const { return m_bVisible; }
     int  MoreComplexInlinedFunction()
     {
       m_i = (m_i < 9) ? ++m_i : 0;
       return colors(m_i);
     }
};
```

Отступы и выравнивание

☞ Согласно общему правилу, каждый отступ соответствует уровню вложенности блока. Ничто, даже комментарии и операторы условной трансляции, не должны нарушать это правило.

неверно

```
void Foo() {
 int a,b;
     if( ... )
          Statement1();
 else
     Statement2();
     for( ;; ){
#ifdef _DEBUG
          TraceStatement();
#endif
// This is a comment
          if( bStop ) break;
     }
}
```

верно

```
void Foo()
{
  int a, b;

  if ( ... )
    Statement1();
  else
    Statement2();
  for( ;; )
  {
    #ifdef _DEBUG
    TraceStatement();
    #endif
    // This is a comment
    if (isStop) break;
  }
} // Foo
```

- Исключением из общего правила являются модификаторы доступа в описании класса. Ключевые слова **public, protected, private** располагаются на уровне фигурных скобок, ограничивающих определение класса.

```
class TObject
{
public:
     TObject();
     ~TObject();

private:
     bool isValid;

}; // TObject
```

Форматирование управляющих конструкций

Используйте **общие правила** размещения скобок и отступов.

Используйте указанные далее варианты оформления.

- Операторы цикла **for**, **while**, оператор условия **if.** После ключевого слова ставится пробел. В скобках пробелы не ставятся.

неверно

```
if( ... ){
     statement;

for( ... )
     statement;
```

верно

```
if (...)
     statement;

for (...)
     statement;
```

- Если оператор **if** имеет ветвления **else if**, используйте следующий вариант размещения скобок:

неверно

```
if( ... ){
     statement;
}else if( ... ){
     statement;
}else{
     statement;
}
```

верно

```
if (...)
{
     statement;
     statement;
}
else if (...)
{
     statement;
     statement;
}
else
{
     statement;
     statement;
}
```

- В операторе **do-while** размещайте выражение **while** на одной строке с закрывающей скобкой

```
do
{
     ...
} while( ... );
```

- Форматируйте оператор **switch** так, чтобы были видны варианты **case**. Рекомендуется использовать отступ для слов **case** и двойной отступ для тел соответствующих вариантов.

неверно

```
switch (eventHandler)
{
  case evCreate:
  statement; statement;
  break;
  case evDestroy:
  statement; statement;
  break;
  default:
  statement; statement;
} // switch
```

верно

```
switch (eventHandler)
{
  case evCreate:
    statement;
    statement;
    break;
  case evDestroy:
    statement;
    statement;
    break;
  default:
    statement;
} // switch
```

☞ Все операторы обязаны иметь скобки, ограничивающие тело блока. Исключением является единственный случай, когда тело оператора *гарантированно* состоит из одного предложения.

неверно

```
for( i = 0; i < 10; i++ )
     if( i % 2 )
          if( i % 8 )
               c = a + b;
          else
               return;
```

верно

```
for (i = 0; i < 10; i++)
{
     if ((i % 2) == 0)
          continue;
     if ((i % 8) == 0)
          return;
     c = a + b;
}
```

Форматирование блока namespace

Если блок достаточно велик (как это обычно бывает – весь файл заголовков), то нет необходимости сдвигать все содержимое файла. Иначе придерживайтесь общих правил.

локальный блок

```
namespace ModuleName
{
  class Foo
  {
    ...
  };
  ...
}; // namespace
```

интерфейсный файл

```
#ifndef __MODULENAME_H__
#define __MODULENAME_H__

namespace ModuleName
{

class Foo
{
  ...
};
....

}; // namespace
#endif
```

Форматирование функций

Скобки, ограничивающие тело функции, всегда располагаются в одной колонке на отдельных строках.

неверно

```
void Foo() {
    ...
}
```

верно

```
void Foo()
{
    ...
}
```

Старайтесь разместить весь заголовок функции на одной строке. Если функция имеет длинный список параметров, то продолжение списка должно иметь стандартный отступ в 2 пробела.

Если функция имеет достаточно длинное тело (> 15–20 строк), то у закрывающей скобки желательно повторить (в комментарии) имя функции.

неверно

```
void Foo(
    int a,
    char b
        ){
    statement(s);
}
```

```
BOOL
EXTAPI
AFunc(int aVaryLongParameterName,
    int aSecondParameterName,
    char* pAThirdParameterName ){

    statement(s);
    statement(s);
}
```

верно

```
void Foo(int a, char b)
{
    statement(s);
}
```

```
BOOL EXTAPI AFunc(int   aVeryLongParameterName,
  int   aSecondParameterName,
  char* pAThirdParameterName )
{
    statement(s);
    statement(s);
} // AFunc
```

Если в начале тела функции следует описание локальных переменных, рекомендуется отделять этот список от первого исполняемого оператора пустой строкой. По возможности выравнивайте колонки описания типа и имен переменных.

хуже

```
void Foo()
{
     int i,j;
     HMODULE hInst = NULL;
     dword count = 100;
     while(count > 0){
          ...
          ...
     }
}
```

лучше

```
void Foo()
{
     int     i, j;
     HMODULE hInst = NULL;
     dword   count = 100;

     while (count > 0)
     {
          ...
          ...
     }
}// Foo
```

При описании шаблонов используйте отдельную строку для конструкции template<>.

хуже

```
template<class T, class O> void Foo(T value)
{
     O obj;
     ...
}
```

лучше

```
template<class T, class O>
void Foo(T value)
{
     O obj;
     ...
}
```

Если возможно, размещайте список инициализаторов на одной строке с заголовком конструктора. Если список слишком длинный и не помещается на одной строке, то начинайте список инициализаторов с новой строки. При этом двоеточие должно располагаться на строке заголовка, а список инициализации идти со стандартным отступом в 2 символа.

хуже

```
TObj::TObj( int a, int b ) :TBase(a),
     localValue(b)
{
     ...
}
```

лучше

```
TObj::Tobj(int a, int b):
  TBase(a), localValue(b)
{
     ...
}
```

Организация классов

- Реализация класса состоит из двух частей: спецификация и реализация. Спецификация находится в заголовочном файле (.h). Реализация класса находится в файле (.cpp). Для классов-шаблонов и классов, у которых все методы реализованы как **inline**, файл (.cpp) отсутствует.
- Описывайте классы в следующей последовательности:
  - друзья класса,
  - приватные члены класса,
  - приватные конструкторы, деструкторы, операторы,
  - защищенные члены класса,
  - защищенные конструкторы, деструкторы, операторы,
  - открытые члены класса.

Если у класса есть хотя бы один виртуальный метод, то деструктор должен быть объявлен виртуальным.

В общем случае, данные класса должны быть приватными или защищенными.

Реализация коротких **inline** методов может быть размещена в спецификации класса. Если тело **inline** метода превышает 3–5 строк, выносите реализацию за пределы спецификации.

неверно

```
class TFoo
{
 public:
     TFoo() { SomeInitialization(); }
     bool IsVisible() const { return m_bVis; }
     void MoreComplexInlined()
     {
          if( ... ){
               statement;
          }
          statement;
          statement;
          ...
     }
     void MoreMemberFunction( ... );

}; //TFoo
```

верно

```
class TFoo
{
public:
     TFoo() { SomeInitialization(); }
     bool IsVisible() const { return isVis; }
     void MoreComplexInlined();
     void MoreMemberFunction(...);
}; // TFoo

inline void TFoo::MoreComplexInlined()
{
     if (...)
     {
          statement;
     }
     statement;
     statement;
}
```

Если класс реализует структуру данных с некоторой “встроенной автоматизацией”, используйте следующие правила:

- Поля данных могут быть открытыми, можно использовать ключевое слово “struct” вместо комбинации “class / public:”.
- Вначале описывайте поля данных, затем конструктор(ы), деструктор, затем методы.

Повторяйте в комментарии имя описываемого класса у закрывающей скобки.

хуже

```
class TVeryComplexClass
{
     friend TFriendClass;
     ....
public:
     TVeryComplexClass();
     int GetSomeInformation() const;
     ...
};
```

лучше

```
class TVeryComplexClass
{
     friend TFriendClass;
     ....
public:
     TVeryComplexClass();
     int GetSomeInformation() const;
     ...
}; // TVeryComplexClass
```

Указатели и Ссылки

Не объявляйте в одной строке переменные и указатели.

Давайте по одному определению локальной переменной на каждой строке. Исключение могут составлять переменные скалярного типа, например счетчики циклов (i, j). Используйте “цельное” определение типа переменной, указывая символ указателя или ссылки непосредственно вслед за именем базового типа.

```
int* p;
int* pA;
int  i, j;
```

Организация Модулей

Понятие Модуль

Модуль – это единица компиляции. Каждый модуль состоит из пары файлов с одинаковым именем: включаемого файла заголовков (.h) и файла исходного текста (.cpp). Включаемый файл (.h) представляет собой интерфейсную часть модуля, т.е. содержит всю необходимую информацию для того, чтобы воспользоваться услугами классов и /или функций, реализованных в данном модуле. Существуют следующие типы модулей:

- **Реализация отдельного класса**

  В этом случае рекомендуется давать файлам (.h) и (.cpp) имена, совпадающие с именем класса, но без префикса "T". При этом в файле заголовков располагается описание класса и реализация **inline** методов, а в файле .cpp – реализация **outline** методов.

```
//FileBrowser.h
#include "ObjBase.h"

class TFileBrowser : public TObjBase
{
    ....
};
```

```
//FileBrowser.cpp
#include "FileBrowser.h"

TFileBrowser::TFileBrowser()
{
    ....
}
```

- **Реализация функционального слоя системы**

  В этом случае модуль реализует набор классов и/или функций, объединенных общей функциональной нагрузкой. В этом случае имя модулю дается по имени наиболее важного класса или по общему назначению классов/функций.

Общие правила

☞ Каждый модуль состоит из двух файлов: включаемого интерфейсного файла (.h) и файла реализации (.cpp). Оба файла имеют одинаковые имена. Допускается использование длинных имен файлов. Имена файлов не должны содержать пробелов. Если модуль реализует отдельный класс, то имена файлов совпадают с именем класса, но без префикса "T".

неверно

```
//Common Header.h
class TFileSystem ... ;
class TUserInterface ... ;

//filesyst.cpp
#include "Common Header.h"
TFileSystem::TFileSystem() ...

//User Interf.cpp
#include "Common Header.h"
TUserInterface::TUserInterface() ...
```

верно

```
//FileSystem.h
class TFileSystem ... ;

//FileSystem.cpp
#include "FileSystem.h"
TFileSystem::TFileSystem() ...

//UserInterface.h
class TUserInterface ... ;

//UserInterface.cpp
#include "UserInterface.h"
TUserInterface::TUserInterface() ...
```

Рекомендуется располагать реализацию методов класса и/или тела функций точно в том порядке, в котором они перечислены в интерфейсном файле. Это значительно облегчает изучение модуля.

☞ Каждый интерфейсный файл должен содержать полную информацию о связях с другими модулями, т.е. в нем должны быть перечислены все интерфейсные файлы тех модулей, услугами которых пользуется данный модуль. Другими словами, каждый интерфейсный файл должен быть успешно скомпилирован без дополнительных включений.

неверно

```
//Base.h
class TBase ... ;

//Object.h
class TObject : public TBase ... ;

//Object.cpp
#include "Base.h”
#include "Object.h”

TObject::TObject() : TBase()
{
    ...
}
```

верно

```
//Base.h
class TBase ... ;

//Object.h
#include "Base.h”
class TObject : public TBase ... ;

//Object.cpp
#include "Object.h”

TObject::TObject() : TBase()
{
    ...
}
```

👍 В общем случае файл реализации имеет один единственный включаемый файл – это его собственный интерфейс. Однако допускается включение файлов заголовков для объявления средств, необходимых исключительно для реализации и которые не нужно (или даже вредно) «показывать» в интерфейсной части модуля.

хуже

```
//Object.h
#include "Base.h”
#include <stdio.h>
#include <malloc.h>
#include <direct.h>

class TObject :public TBase ... ;

//Object.cpp
#include "Object.h”

TObject::TObject() :TBase() ...
```

лучше

```
//Object.h
#include "Base.h”

class TObject : public TBase ... ;

//Object.cpp
#include "Object.h”
#include <stdio.h>
#include <malloc.h>
#include <direct.h>

TObject::TObject() : TBase() ...
```

☞ Символы, предотвращающие повторное включение интерфейсного файла, должны быть записаны так: «*двойное подчеркивание – имя файла заглавными буквами – подчеркивание – расширение имени файла заглавными буквами – двойное подчеркивание*».

неверно

```
//FileSystem.h
#ifndef _INC_FileSystem_h
#define _INC_FileSystem_h
....
#endif
```

верно

```
//FileSystem.h
#ifndef __FILESYSTEM_H__
#define __FILESYSTEM_H__
....
#endif
```