Стандарт программирования на C++

Версия 0.1.1.2b
Автор: Apple
Ссылка на документ: www.kernel.pro/stdc/

Введение

Положения данного стандарта большей частью взяты из других действующих стандартов и соглашений.
Стандарт будет пополняться примерами по мере нахождения свободного времени.
Предложения, возражения и дополнения можете внести мне через обратную связь.
Данный документ можно распространять любым способом с условием указания ссылки на первоисточник.

Оглавление

Заголовочные файлы

Каждый `.c` файл кода должен иметь свой заголовочный `.h` файл. Исключения составляют только `.c` файлы, содержащие единственную функцию `main()` и некоторые другие.

Защита от повторного использования .h файла

Для защиты определяется уникальный идентификатор с помощью директивы `#define` и далее проверяется его объявление с помощью директивы `#ifdef`. Имя идентификатора получается из полного пути от корня проекта до файла. Например, файл `MyProject/device/code.h` будет иметь идентификатор `MYPROJECT_DEVICE_CODE_`

```
#ifndef MYPROJECT_DEVICE_CODE_
#define MYPROJECT_DEVICE_CODE_

// ... содержимое файла MyProject/device/code.h

#endif  // MYPROJECT_DEVICE_CODE_
```

Зависимости в заголовочных файлах

Поскольку в заголовочных файлах обычно содержатся только объявления функций, классов и т.п., не используйте `#include` с соответствующими файлами в заголовочных файлах, если это возможно (разумеется, в .c файле кода будут нужно сделать все необходимые `#include` включения файлов). Если в заголовочном файле объявлена переменная или производится наследование от класса, который определен в неком заголовочном файле, вы должны включить этот заголовочный файл в ваш заголовочный файл.

Встроенные функции (inline functions)

Применяйте только небольшие (до 10 строк кода) встроенные функции. В этом случае быстродействие программы несколько повышается. В случае использования больших встроенных функций выносите их в отдельный заголовочный файл с суффиксом `-inl.h`. Чрезмерное использование встроенных функций может привести к увеличению размера программы и снизить ее быстродействие. В случае наследуемого класса помните, что с вызовом функции могут вызываться аналогичные методы базовых классов, например, деструктор.
Не дает выигрыша в производительности применение циклов for или условий switch в таких функциях.

Порядок определения параметров функции

Первые параметры – входные, последние – выходные (`out`) параметры.

Именование и порядок включений #include

Стандартный порядок секционирования: C-библиотеки, C++ библиотеки, другие библиотеки, библиотеки текущего проекта. Внутри каждой секции файлы включений желательно располагать в алфавитном порядке.
Путь до файла должен быть написан относительно корня проекта. Архитектура расположения файлов проекта должна быть сформирована таким образом, чтобы путь до файла был написан без спец. сокращений (например, «`.`» - текущий каталог или «`..`» - каталог уровнем выше).

Границы видимости кода

Пространства имен

Пространства имен (`namespace`) помогают отойти от глобальной области видимости. Это способствует уменьшению вариантов кода, предлагаемых парсером автозавешения используемой IDE и уменьшает возможную путаницу в именах в большом проекте (где могут быть классы с одинаковым названием), помогают выстроить иерархическую структуру.
Именование namespace следует выбирать, исходя из названия проекта и пути расположения файла в дереве каталогов проекта.
В случае использования пространств имен в проекте следует использовать их во всех `.h` заголовочных файлах проекта. Возможно использование пространств имен и в `.c` файлах кода.
Пространства имен следует объявлять после определений, объявлений и предварительный объявлений классов из других пространств имен.
Используйте директиву using только внутри методов, функций, классов в `.h` заголовочном файле и где угодно – в `.c` файле кода.
Допустимо в функциях использовать локальные алиасы пространства имен:
```
namespace abc = ::a::b::c;
```

Вложенные классы

Не создавайте открытых вложенных классов там, где это возможно.

Внешние функции, статические методы и глобальные функции

Предпочтение отдается использованию внешних функций внутри пространства имен или статическим методам. Реже следует пользоваться глобальными

функциями.
Если внешняя функция нужна только в текущем классе, используйте не именованное пространство имен или объявляйте ее статической.

Локальные переменные

Объявляйте локальные переменные в непосредственной близости от их использования и проводите их инициализацию во время объявления. Объявление переменных внутри цикла происходит по разному для компиляторов. Некоторые версии компиляторов позволяют видеть объявленную в цикле переменную за его пределами. Также помните, что при вхождении в область видимости переменной вызывается конструктор, а при выходе – деструктор класса. Из этого следует, что желательно объявлять переменную до начала цикла, чтобы конструктор и деструктор вызывались один раз.

Статические и глобальные переменные

Статические и глобальные переменные класса запрещены. Они могут быть только элементарных типов, таких как `int`, `char`, `float`, `void` и не могут быть инициализированы возвращенным значением функции. Не константные глобальные переменные нельзя использовать в многопоточном коде.
В случае многопоточного кода статическая переменная из основного потока может быть уничтожена деструктором, в то время как другой поток может ее использовать. Порядок вызова деструкторов может быть разный в разных версиях компилятора.

Классы

Классы – фундаментальная часть языка C++.

Выполнение операций в конструкторе класса

Выполняйте только тривиальные операции. В противном случае выносите инициализацию в отдельный метод с именем `init()`. Также полезно добавить флаг, сообщающий, что объект успешно инициализирован.
Например, при создании глобальной переменной данного типа (они запрещены) конструктор будет вызван до вызова функции `main()`, что может повлечь невозможность инициализации каких-либо переменных в коде вашего конструктора.

Конструктор по умолчанию

Необходимо самостоятельно определять конструктор по умолчанию (без

аргументов). Исключением является случай, когда класс наследуемый и не добавляет новых методов.

Явные конструкторы

В случае, если конструктор принимает один аргумент, его следует делать явным (`explicit`):

```
explicit Foo (string name)
```

То есть запрещать неявное преобразование. К примеру, если вы определили `Foo::Foo(string name)` и затем передали значенеи `string` в функцию, принимающую значение типа `Foo`, указанный явный конструктор будет вызван для преобразования `string` в `Foo` и передачи `Foo` в даную функцию.
Объявление такого конструктора влечет невозможность его использования в неявных преобразованиях.
В редких случаях конструкторы копирования нежелательно делать явными. Тогда нужно подробно описывать причину в комментариях.

Конструкторы копирования

Конструктор копирования используется при копировании одного объекта в другой. Особенно, если осуществляется передача объекта по значению. Разрешать такое копирование классов нужно ишь в тех случаях, когда вы отдаете себе отчет, зачем оно необходимо, так как оно является причиной множества ошибок, в том числе может замедлить быстродействие кода и увеличить потребляемую память.
Для блокирования копирования можно использовать макрос

```
// Макрос для запрета конструктора копирования и
// оператора «=» функций
// Должен быть объявлен в private части класса
#define DISALLOW_COPY_AND_ASSIGN(TypeName) \
  TypeName(const TypeName&);               \
  void operator=(const TypeName&)

class Sample {
 public:
  Sample(int dig);
  ~Sample();

 private:
  DISALLOW_COPY_AND_ASSIGN(Sample);
};
```

STL-контейнеры требуют, чтобы все хранимые объекты могли быть скопированы и присвоены. В этом случае необходимо сохранять в контейнерах

указатели на ваш объект.

Структуры или классы

Используйте структуры только для «пассивного» использования данных. Для остальных целей используйте классы.
Доступ к данным структуры осуществляется напрямую. Методы можно использовать только для действий над имеющимися данными (обнулить, проверить на валидность, инициализировать и т.п.).
Если нужно производить с данными более сложные действия, используйте класс.
Заметьте, что правила именования полей в структурах и классах разные.

Наследование

Делайте наследование открытым. В базовом классе явно указывайте виртуальные (`virtual`) функции, которые могут быть переопределены в классах-потомках. Аналогично с классом-наследником. Иначе при повторном использовании кода будет тяжело разобраться, какие функции можно переопределить.
Не забывайте в классах явно задавать виртуальный деструктор.
Доступ к данным базового класса из класса-наследника должен быть закрыт.

Множественное наследование

Полезно в редких случаях. Разрешено только если родительские классы являются чистыми интерфейсами. Для гарантии того, что в будущем они останутся интерфейсами, они должны именоваться суффиксом `Interface`.

Интерфейсы

Интерфейс — это класс, отвечающий следующим условиям:
Он имеет только открытые чистые виртуальные ("`= 0`") и статические методы (см. Деструктор).
У него отсутсвуют не статические члены данных.
Не определен конструктор. Если, то в `protected` части и без аргументов.
Если у класса есть базовый класс, то он в свою очередь также должен удовлетворять этим условиям и быть именованным суффиксом `Interface`.
Из-за чистых виртуальных функций невозможно создать экземпляр интерфейса. Для того, чтобы убедиться, что все реализации интерфейса могут быть корректно освобождены, они должны также объявить деструкторы виртуальными (являясь исключением первому правилу он не должен быть чисто виртуальным). См. Страуструп, The C++ Programming Language, 3-е издание, глава 12.4.

Перегрузка операторов

Перегрузка операторов запрещена за исключением редких случаев (например, перегрузки оператора == требуют некоторые алгоритмы STL). Ее использование влечет путаницу с точки зрения использования ресурсов, работы с указателями и проч., хотя и помогает более интуитивно использовать код в стандартных случаях. Лучше определить стандартные имена методов `Add()` `Equals()`, `Copy()` и т.п.

Контроль доступа

Все члены данных должны быть закрытыми. Их изменение осуществлять через функции доступа.

Порядок деклараций

Описание класса принято начинать с `public`-части. Затем следует `protected`. И в конце — `private`.

Внутри каждой части порядок декларирования следующий:
`Typedef` и `Enum`.
Константы.
Конструкторы.
Деструкторы.
Методы, включая статические.
Члены данных (переменные), включая статические.

В конце `private`-части следует размещать макрос `DISALLOW_COPY_AND_ASSIGN`, если класс запрещено копировать и присваивать.

В файле кода желательно размещать в том порядке, в котором они декларированы в заголовочном файле.

Пишите маленькие функции

Необязательное требование. Однако, если функция занимает более 40 строчек, задумайтесь, модно ли ее разбить на несколько.

Другие инструменты C++

Ссылочные аргументы

В C, если функции требуется изменить значение переменной, ее параметр должен использовать указатель, например, `int bar(int *val)`. В C++ функция может декларировать ссылочный параметр: `int bar(int &val)`. При передаче параметра по ссылке не нужно разыменовывать указатель для изменения значения занимаемой им в ячейке памяти: `(*val)++`.
Все параметры, передаваемые по ссылке, должны быть константными, так как они имеют синтаксис обычного параметра по значению:
таким образом, входные «ссылочные» параметры функции не могут быть изменены, так как являются константами. С другой стороны выходные параметры функции должны быть изменены, следовательно, должны быть указателями. Естественно, можно использовать параметры по значению.
В соответствии с принятым нами соглашением сначала должны быть перечислены входные параметры.

```
void bar(int in_val, const int &in_val_ref, int *out_val);
```

Указатель во входном параметре необходимо использовать только в случае невозможности применения ссылочного параметра или параметра по значению. Напрмиер, STL адаптеры `bind2nd` и `mem_fun` не позволяют использовать ссылочные параметры.

Перегрузка функций

Не используйте перегрузку функций для симулирования исполнения функции с параметром по умолчанию.
В остальных случаях перегрузку следует использовать, если могут принимать входные данные разных типов. Или же для уточнения обработки данных.

```
int Add(int val);
int Add(int val, int round); //указывается система счисления
int Add(double val);
```

Однако, в некоторых случаях это может привести к путанице. Например, если напишем

```
int Add(char val);
```

будет не совсем ясно, что как такая функция добавляет параметр char. В этом случае проще присваивать функциям разные имена:

```
int AppendInt(int val);
int AppendInt(int val, int round); //указывается система счисления
```

```
int AppendDouble(double val);
int AppendChar(char val);
```

Параметры функций по умолчанию

Параметры функции по умолчанию запрещены.
Все параметры, применяемые в функции, должны быть явно заданы при ее вызове.
Данное требование заставляет более качественно продумывать архитектуру функции. Также оно связано с возможными проблемами при повторном использовании кода в другом проекте.

Массивы переменной длины и alloca()

Массивы переменной длины и `alloca` запрещены. В плане ресурсов такие массивы и `alloca()` эффективны. Имеют дружелюбный синтаксис. Однако, не являются частью стандартного языка C++. Они размещают зависимое от данных количество памяти в стеке, что может вызвать трудно определимые ошибки перезаписи памяти. Используйте безопасное размещение памяти, такое как `scoped_ptr`/`scoped_array`.

Дружественные классы

Дружественные классы обычно определяются в одном файле. Обычно применяются для определения состояния класса из-вне без открытия его методов.

Исключения

Следует везде, где это возможно, писать код без использования исключений. В проектах, состоящих более чем из одного класса и имеющих возможность масштабирования, исключения запрещены.

Приведения типов

В C++ для приведения типов используйте `static_cast<>()`. Нежелательно использовать другие типы приведения. Например, C-подобные, такие как `(int)x` или `int(x)`.
Для удаления спецификатора `const` используйте `static_const`:

```
const char *str = "hello";
char *str1 = const_cast<char*>(str);
```

Приведение указателя к указателю, к целому или целого к указателю

осуществляется с помощью `reinterpret_cast`

```
reinterpret_cast<whatever *>(some *)
reinterpret_cast<integer_expression>(some *)
reinterpret_cast<whatever *>(integer_expression)
```

При использовании данного приведения не осуществляется никакких проверок. Поэтому используйте его только если уверены, что знаете больше, сем компилятор.
Запрещено использование `dynamic_cast`.

Потоки (стандартный ввод/вывод)

Для вывода используйте разновидности функций `printf()` и `scanf()`.

Константные переменные (const)

Используйте `const` везде, где это имеет смысл. Это поможет другим людям быстрее понять алгоритм класса/функции и назначение переменной.

```
class Foo
{
    int Bar(char c) const;
};
```

Видно, что функция `Bar()` не может изменить значения переменных.

Целые числа

Из всех стандартных целых типов языка C++ используйте `int`. У разных компиляторов размерность целых чисел может быть разной. Для использования других типов целых чисел включите в проект файл `<stdint.h>`.
В данном файле определены типы `int16_t`, `uint32_t`, `int64_t`.
Использовать данные типы предпочтительнее, чем `short`, `unsigned long`, `long` и т.п.

Беззнаковые целые числа

Не используйте беззнаковые целые числа для документирования факта, что число не может иметь отрицательных значений (пользуйтесь assert макросом). Например, данный код не завершится:

```
for (unsigned int i = foo.Length()-1; i >= 0; --i) ...
```

Макросы препроцессора

Используйте макросы только в том случае, если задачу нельзя решить с помощью стандартных функций/классов/переменных. Не определяйте макросы в заголовочном файле (не относится к макросу защиты от множественного включения).

0 и NULL

Используйте `0` для целых, `0.0` для действительных, `NULL` для указателей и `'\0'` для символьных переменных.

sizeof
Поскольку тип переменной, для которой нужно узнать размерность, может измениться, для предотвращения ошибок передавайте переменную в аргумент функции `sizeof()`, а не тип:

```
MyStruct data;
memset(&data, 0, sizeof(data));
```

C++0x

C++0x – это следующий ISO стандарт языка C++. Его поддержка уже присутсвует в последних версиях компиляторов (например, gcc 4.6).
Однако, в настоящее время нельзя сказать со 100%-й уверенность, как поведет себя приложение в других версиях компиляторов. Поэтому пока лучше воздержаться от использования C++0x.

Именования

Общие правила именования

Имена функций, переменных, файлов дожны быть описательными, воздердитесь от аббревиатур.
Типы и переменные — существительные. Функции — глаголы (императивные «командные»).

Именование кода

Не беспокойтесь о длине имени. Главное, чтобы код был понятен любому человеку. Не выбрасывайте гласные буквы из слов и не используйте двусмысленные неизвестные другим людям аббреваитуры.

```
int num_errors;                  // Хорошо.
int num_completed_connections;   // Хорошо.
```

```
int num_dns_servers;              // Хорошо, если dns - это
                                  // Domain Name System -
                                  // известная аббревиатура.

int n;                            // Плохо — не несет смысла.
int nerr;                         // Плохо — неизвестная
                                  // аббревиатура.
int n_pc_cnt;                     // Плохо — неизвестная
                                  // аббревиатура cnt (возможно,
                                  // от count), pc также может
                                  // обозначать что угодно.
```

Именование файлов

Имена файлов должны быть только в нижнем регистре, использовать только стандартную ASCII таблицу символов.
Желательно использовать только буквы и символы тире «-» и нижнего подчеркивания «_»
Файлы кода должны иметь расширение .c, заголовочные файлы — расширение `.h`.
Объявления класса — в заголовочном файле, определения — в файле кодаэ
При этом имя файла должно быть одинаковым.
Не используйте имена существующих стандартных файлов C/C++. Например, `db.h`
Лучше используйте специфичные имена файлов, чем общие. К примеру, `http_parser.h` лучше, чем `parser.h`.
Встроенные функции, если они очень короткие, можно размещать в заголовочном файле `.h`. Если они занимают больше 10 строчек, создавайте новый заголовоный файл с суффиксом `-inl`
Например,
http_parser.c — объявление класса;
http_parser.h — определение класса;
http_parser-inl.h — встроенные функции.

Имена типов

Имена типов (`class`, `struct`, `typedef`, `enum`) начинаются с заглавной буквы. Если слово составное — каждое новое слово начинается с заглавной буквы (`HttpParser`). Нижнее подчеркивание недопустимо.

Имена переменных

Имена переменных пишутся только в нижнем регистре. Слова разделяются символом нижнего подчеркивания «_».

Переменные экземпляра класса

Пишутся только в нижнем регистре. Слова разделяются символом нижнего подчеркивания «_». Должны всегда заканчиваться символом нижнего подчеркивания:

```
int num_errors_ = 0;
```

Переменные структур

Переменные пишутся только в нижнем регистре. Слова разделяются символом нижнего подчеркивания «_».

Глобальные переменные

Переменные пишутся только в нижнем регистре.Слова разделяются символом нижнего подчеркивания «_». Всегда начинаются с приставки `g_`

```
int g_num_errors = 0;
```

Константы (const) и члены перечислений (enum)

Переменные пишутся в смешанном регистре. Начинаются с приставки `k`, за которой сразу начинается слово с заглавной буквы. Если слово составное — каждое новое слово начинается с заглавной буквы.

```
const int kMonthsOfYear = 12;
```

Имена функций

Имена открытых функций (методов) пишутся в смешанном регистре. Начинаются с заглавной буквы. Если слово составное — каждое новое слово начинается с заглавной буквы.
Нижнее подчеркивание допустимо только в вспомогательных `protected` и `private` функциях. В этом случае имя функции пишется только в нижнем регистре.

Имена функций доступа

Имя функции получения значения аналогично имени переменной но начинается с заглавной буквы. Если функция доступа открытая, нижнее подчеркивание, разделяющее слова, заменяется на смешанный регистр. В противном случае допустимо не менять название функции.
Функция записи значения — с префиксом `set`

```
int count_windows_ = 0;  // закрытый член класса

void setCountWindows(int count_windows) // открытая функция
                                        // установки значения
{
    count_windows_ = count_windows;
}

int CountWindows() const // открытая функция получения значения
{
    return count_windows_;
};
```

Имена макросов

Все буквы заглавные. Слова разделаются знаком нижнего подчеркивания «_». `MY_MACROS`.

Комментарии

Комментирование классов должно состоять из нескольких повествовательных предложений.
Комментирование функцию следует делать императивными (глагол действия), например «получить данные через HTTP» или «Добавить значение в массив».
При комментировании переменных избегайте, где это возможно, использование глаголов.
Все комментирование выполняйте в едином синтаксическом стиле — от единого лица.

Именование сборки и контроль версий

Именования файлов сборки (скомпилированного проекта и вспомогательных файлов)

При именовании файлов сборки можно использовать только символы ASCII (латиницу). Ярлыки (ссылки) на файлы, однократно сохраняемые сборкой файлы и файлы, сохраняемые пользователем, могут именоваться другим набором символов. Желательно применять UTF, если это поддерживает файловая система.

Жизненный цикл релизов проекта

Pre-Alpha (pa) – назначается в начале разработки проекта. Характеризуется постоянным добавлением функционала без проведения детального тестирования. Также данный статус обычно назначается проекту в случае

увеличения мажорного номера версии (MAJOR_VERSOIN).

Alpha (a) – при включении в проект запланированного функционала Pre-Alpha версия передаетяс на внутренне тестирование. На данном этапе возможно наращивание функционала, однако предпочтение отдается тестированию.

Beta (b) – этап публичного тестирования. Расширение функционала на данном этапе не осуществляется (только планируется — заносится в список TODO). Возможно улучшение GUI.

Release Candidate (rc) – присваивается проекту, когда устранены все известные ошибки, найденные на предыдущих этапах и реализован весь функционал проекта. Можно изменять только файлы конфигурации и документацию.

Release to manufacturing [marketing] (rtm) – проект готов для распространения.

General availability (ga) – создана инфраструктура для продажи и распространения на физических носителях и в электронном виде.

Long term support (lts) – проект с длительной поддержкой (выпуском обновлений, заплаток), то есть на данном этапе меняется только BUILD_VERSION. Обычно данный статус дается тем проектам, поддержка которых запланирована минимум на 3 года и одновременно планируется выпуск других версий проекта (MAJOR_VERSION и/или MINOR_VERSION)

End on life (eol) – последняя версия проекта, не предусматривающая дальнейшей модернизации и поддержки.

Именование версии релиза проекта

Версия проекта состояит из пяти составляющих:

```
<MAJOR_VERSION>.<MINOR_VERSION>.<BUILD_VERSION>.<REVISION>-<LIFE_CYCLE>
```

например: `MyProject 1.0.32.54-rc`

где:
MAJOR_VERSION – старший номер версии. В начале разработки устанавливается «0». Первый раз увеличивается до «1», начиная с этапа RTM. В последующем увеличивается только при внесении больших измененииий в функционал проекта. Например, при смене ядра проекта. Также увеличивается в случае смены команды разработчиков проекта.

`MINOR_VERSION` – младший номер версии. Увеличивается при смене функционала проекта. Например, добавлении новых форм, модулей и т.п. Решение об увеличении принимает руководитель проекта.

`BUILD_VERSION` – версия сборки. Увеличивается при каждом компилировании проекта.

`REVISION` – версия проекта, хранящаяся в репозитарии. Увеличивается при каждом изменении содержимого репозитария независимо от типа изменений.

`LIFE_CYCLE` – этап разработки (см. «Жизненный цикл релиза проекта»).

Примечание: при наличии плагинов и модулей решение о типе именования их версий принимается на начальной стадии разработки проекта и не меняется в течение жизненного цикла проекта.

Контроль версий

В качестве системы контроля версий используейтся GIT (www.git-scm.com). Обызательно указывать свое имя (псевдоним) и email дл связи. Каждый коммит должен иметь комментарии, отражающие суть изменений.