GSC Game World

2009

Стандарт кодирования на С++

Для проектов на движке X-Ray v2.0

[GSC Game World – 2010]

# Содержание

Общее

1. [Зачем стандарт?](#CommonWhyStandard)
2. [Принципы построения стандарта](#CommonPrinciples)

Общий стандарт

1. Стиль дизайна
   * [фокус на цели](#CommonStandardFocus)

* [принцип KISS (Keep It Simple Stupid)](#CommonStandardKISS)
* [масштабируемость](#CommonStandardScaling)
* [преждевременная оптимизация](#CommonStandardPrematureOptimization)
* [ненужная пессимизация](#CommonStandardPrematurePessimization)
* [минимизируйте глобальные и общие (shared) данные](#CommonStandardMinimizeShared)
* [прячьте внутренние данные](#CommonStandardHideInternals)
* [используйте RAII (Resource Acquisition Is Initialization)](#CommonStandardRAII)

1. Стиль кодирования
   * [проверки времени компиляции](#CommonStandardCompileTime)
   * [активно используйте const](#CommonStandardConst)
   * [не используйте макросы там, где можно обойтись без них](#CommonStandardMacros)
   * [не используйте магические числа](#CommonStandardMagicNumbers)
   * [определяйте переменные, объекты как можно локальнее](#CommonStandardLocality)
   * [инициализируйте переменные](#CommonStandardInitializeVariables)
   * [избегайте длинных функций и глубоких вложений](#CommonStandardLongFunctions)
   * [не используйте статические и глобальные переменные с нетривиальными конструкторами/деструкторами](#CommonStandardGlobals)
   * [используйте форвард декларацию везде, где это возможно](#CommonStandardForwardDeclarations)
   * [включайте в заголовок все необходимые для его компиляции заголовки](#CommonStandardIncludeAllHeaders)
   * [используйте защиту от повторного включения заголовков в виде макросов стражей](#CommonStandardUseHeaderGuards)
2. Функции и операторы
   * [разделяйте входные и выходные параметры функций](#CommonStandardInputParameters)
   * [не используйте ...](#CommonStandardEllipses)
   * [переопределяйте операторы только имея существенные основания для этого и сохраняйте при этом природную семантику](#CommonStandardOperatorsOverload)
   * [если определяете a+b, то определяйте и a+=b](#CommonStandardPlusAssign)
   * [если определяете ++a, то определяйте и a++](#CommonStandardPostfixPlusPlus)
   * [не плодите большое количество ненужных временных объектов](#CommonStandardUnnecessaryTemporaries)
   * [не рассчитывайте на конкретный порядок подстановки значений в выражения](#CommonStandardValueSubstitution)
3. Дизайн классов и интерфейсов
   * [разные типы классов](#CommonStandardClassKinds)
   * [маленькие классы](#CommonStandardSmallClasses)
   * [предпочитайте композицию наследованию](#CommonStandardPreferComposition)
   * [не наследуйтесь от классов, которые не предназначены для наследования](#CommonStandardInheritCarefully)
   * [предпочитайте наследоваться от абстрактных интерфейсов](#CommonStandardInheritFromInterfaces)
   * [принцип подстановки Лисков (LSP)](#CommonStandardLSP)
   * [переопределяйте функции с умом](#CommonStandardOverriding)
   * [используйте разделение пользовательского интерфейса и интерфейса кастомизации](#CommonStandardNVI)
   * [предпочитайте явные приведения неявным](#CommonStandardPreferExplicit)
   * [размещайте все члены данных в приватной области скрытия](#CommonStandardPrivataMembers)
   * [не давайте пользователям класса возможность модифицировать внутренние данные класса](#CommonStandardDoNotModifyInternals)
   * [используйте идиому Pimpl](#CommonStandardPimpl)
   * [предпочитайте делать функции не членами класса и не друзьями — это уменьшает сцепление](#CommonStandardNonmembersNonfriends)
   * [не определяйте в классах операторы new/delete](#CommonStandardNoNewdelete)
4. Конструирование, удаление и копирование
   * [определяйте и инициализируйте члены класса в одном и том же порядке](#CommonStandardMembersInitialization)
   * [используйте список инициализации вместо оператора присваивания](#CommonStandardInitializationList)
   * [деструктор](#CommonStandardDestructor)
   * [если вы определили один из: конструктор копирования, оператор присваивания или деструктор, то определите и остальные оба](#CommonStandardDefineAll3)
   * [явно запрещайте или разрешайте копирование, либо комментируйте, почему подходит автоматически сгенерированные компилятором версии](#CommonStandardCopying)
   * [избегайте срезания объектов](#CommonStandardSlicing)
   * [при определении оператора присваивания, в котором вы используете копию переданного объекта, предпочитайте форму оператора, где объект передаётся по значению](#CommonStandardCompilerFriendly)
5. Пространства имён и модули
   * [свободные функции не члены класса должны быть определены в том же пространстве имён, что и сам класс](#CommonStandardFreeFunctions)
   * [using пространства имён](#CommonStandardUsing)
6. Шаблоны и универсальность
   * [используйте статический полиморфизм](#CommonStandardStaticPolymorphism)
   * [зряче располагайте точки кастомизации при написании шаблона](#CommonStandardCustomizationPoints)
   * [избегайте написания необщего кода](#CommonStandardNonGenericCode)
7. STL: контейнеры
   * [старайтесь размещать value-like типы в контейнерах](#CommonStandardContainerValueTypes)
   * [старайтесь использовать push\_back вместо других способов](#CommonStandardPrefer_push_back)
   * [используйте вставку диапазона вместо поэлементной вставки](#CommonStandardPreferRangeInsert)
   * [идиомы shrink-to-fit, erase-remove](#CommonStandardShrinkToFit)
8. STL: алгоритмы
   * [используйте подходящий поисковый алгоритм](#CommonStandardSuitableSearch)
   * [используйте подходящий алгоритм сортировки](#CommonStandardSuitableSorting)
   * [не разрешайте предикатам менять своё состояние во время вызова оператора(), которое влияет на результат оператора()](#CommonStandardOperatorBrackets)
   * [старайтесь передавать в качестве предикатов не функции, а объекты — они обычно производят более быстрый код](#CommonStandardPreferObjects)
9. Безопасность типов
   * [не используйте switch по типу объекта для кастомизации поведения](#CommonStandardDoNotCustomizeWithSwitch)
   * [полагайтесь на типы, а не их представление](#CommonStandardRelayOnTypes)
   * [static\_cast, static\_cast\_checked и pointer\_cast](#CommonStandard_static_cast)
   * [не используйте reinterpret\_cast](#CommonStandard_reinterpret_cast)
   * [не используйте const\_cast](#CommonStandard_const_cast)
   * [не используйте C styl cast](#CommonStandard_C_style_cast)
   * [не используйте приведение с помощью union (т.н. horrible\_cast)](#CommonStandard_horrible_cast)
   * [не используйте varargs (...)](#CommonStandard_varargs)
   * [не используйте незащищённые строковые функции](#CommonStandardUnsafeStringFunctions)
   * [не используйте memcpy и memcmp ни для чего, кроме как для сырой памяти](#CommonStandardCopyMemory)
   * [не интерпретируйте С массивы полиморфно](#CommonStandardPolymorphicC_Arrays)
10. Оформление кода
    * [разделительные символы](#CommonStandardSeparators)
    * [.h](#CommonStandardHeaders)
    * [.cpp](#CommonStandardCPPs)
    * [скобки](#CommonStandardBrackets)
    * [классы](#CommonStandardClasses)
    * [пространства имён](#CommonStandardNamespaces)
    * [шаблонные параметры](#CommonStandardTemplateParameters)
    * [if](#CommonStandardif)
    * [for](#CommonStandardfor)
    * [switch](#CommonStandardswitch)
    * [goto](#CommonStandardgoto)
11. [Динамическое приведение типов](#CommonStandardDynamicCast)
12. [Встроенные функции](#CommonStandardInlineFunctions)
13. [Внешние функции/Методы класса](#CommonStandardClassMethods)

* [идентификаторы](#CommonStandardIdentifiers)
* [параметры](#CommonStandardParameters)
* [возвращаемые значения](#CommonStandardReturnValue)
* [локальные переменные](#CommonStandardLocals)
* [уровни вложенности](#CommonStandardNestedBlocks)
* [сложные выражения](#CommonStandardComplexExpressions)
* [\_alloca](#CommonStandard_alloca)

1. Внешние функции

* [пространства имён](#CommonStandardExternalFunctions)
* [static](#CommonStandardStaticFunctions)

1. Классы

* [файлы класса](#CommonStandardClassFiles)
* [пространство имён](#CommonStandardClassNamespace)
* [идентификаторы](#CommonStandardClassIdentifiers)
* [наследование](#CommonStandardClassInheritance)
* [методы](#CommonStandardClassMethodsClass)
* [члены данных](#CommonStandardClassMembers)
* [выравнивание (padding) членов данных](#CommonStandardClassMembersPadding)
* [перечислимые типы (enum)](#CommonStandardClassEnums)
* [пользовательские типы (typedef)](#CommonStandardClassTypedefs)
* [конструктор копирования](#CommonStandardClassCopyConstructor)
* [виртуальный деструктор](#CommonStandardClassVirtualDestructor)
* [невозможность порождения](#CommonStandardFinalClass)
* [порядок описания](#CommonStandardDeclarationOrder)
* [пример декларации и реализации класса](#CommonStandardDeclarationExample)

Стандарт на проект Сталкер

1. Организационные вопросы

* [уровень ворнингов](#StalkerStandardWarningLevel)
* [система контроля версий](#StalkerStandardVersionControlSystem)
* [система review](#StalkerStandardReviewSystem)
* [автоматические билды](#StalkerStandardAutomaticBuilds)

1. [Работа с памятью](#StalkerStandardMemory)
2. [Библиотеки](#StalkerStandardModules)

* [структура](#StalkerStandardLibrariesStructure)
* [файлы проекта](#StalkerStandardLibrariesFiles)
* [прекомпилированные заголовочные файлы](#StalkerStandardLibrariesPCH)
* [платформо-зависимая функциональность](#StalkerStandardLibrariesPlatform)

1. [Строки](#StalkerStandardStrings)

* [strings::shared](#StalkerStandardStringsShared)
* [std::string](#StalkerStandardStringsSTD)
* [buffer\_string](#StalkerStandardBufferString)
* [fixed\_string](#StalkerStandardFixedString)
* [strings::join vs sprintf\_s](#StalkerStandardSTR_JOIN)
* [STR\_JOINA](#StalkerStandardSTR_JOINA)
* [strings::duplicate](#StalkerStandardStringsDuplicate)

1. [Контейнеры](#StalkerStandardContainers)

* [vector](#StalkerStandardVector)
* [deque](#StalkerStandardDeque)
* [stack](#StalkerStandardStack)
* [list](#StalkerStandardList)
* [map](#StalkerStandardMap)
* [multimap](#StalkerStandardMultiMap)
* [set](#StalkerStandardSet)
* [multiset](#StalkerStandardMultiSet)
* [boost::array](#StalkerStandardBoostArray)
* [C style array](#StalkerStandard_C_style_array)

1. Специальные контейнеры

* [associative\_vector](#StalkerStandardAssociativeVector)
* [buffer\_vector](#StalkerStandardBufferVector)
* [fixed\_vector](#StalkerStandardFixedVector)
* [внедрённые (intrusive) контейнеры](#StalkerStandardIntrusiveContainers)

1. [Обработка ошибок и протоколирование](#StalkerStandardErrorHandling)

[Ссылки](#References)

# Общее

## Зачем стандарт?

Хороший стандарт кодирования может привнести с собой несколько взаимосвязанных преимуществ:

* улучшение качества кода: предложение разработчикам делать правильные вещи определённым образом работает на улучшение качества программного обеспечения и его поддержки
* улучшенная скорость разработки: разработчики не должны постоянно принимать решения, начиная с базовых принципов
* улучшенная командная работа: он помогает уменьшить ненужные дебаты на непоследовательные темы, и упрощают членам команды чтение и поддержку кода друг друга

В стрессовых ситуациях под давлением сроков люди делают то, чему они обучены и натренированы. Они делают по привычке. Именно поэтому в реанимационных отделениях больниц работают опытные, тренированные люди; даже обладающие знаниями новички могут запаниковать.

Разработчики программного обеспечения постоянно оказываются под сильным давлением для того, чтобы отдать завтрашнее программное обеспечение вчера. Под давлением графиков, мы делаем то, чему мы натренированы, и что мы обычно делаем. Плохие программисты, которые и в обычных условиях не знают хороших практик разработки программ (или не используют их) будут писать ещё более ужасный код с ещё большим количеством ошибок. Напротив, программисты, которые имеют хорошие привычки и которые пишут хороший код регулярно, будут более самоорганизованы в стрессовых ситуациях и будут писать качественный код быстро.

Конечно же, можно написать плохой код даже в соответствии с лучшими стандартами кодирования. Это же справедливо и для любого языка программирования, процесса или методологии. Хороший стандарт кодирования прививает хорошие привычки и дисциплину, что лучше, чем просто следование правилам. Это основа, однажды приобретённая, открывает двери на более высокие уровни. Здесь нет короткого пути; мы должны знать слова и грамматику, прежде чем заниматься поэзией. Мы лишь попытаемся сделать это проще.

## Принципы построения стандарта

Основной принцип, по которому был построен стандарт, который вы читаете, следующий: не вносить действительно вкусовые вещи, которые не относятся к читабельности или правильности. Любой профессиональный программист может с лёгкостью прочитать код, который отформатирован немного по-другому, чем так, как он привык.

В то же время, необходимо придерживаться одного стиля форматирования в пределах каждого класса.

# Общий стандарт

## Стиль дизайна

### Фокус на цели

Давайте каждой сущности (переменной, функции, классу, пространству имён, модулю) одну хорошо определённую обязанность. С ростом сущности будет расти и её область применения, но её обязанность не должна расходиться.

Пример плохого дизайна: realloc, std::basic\_string.

### Принцип KISS (Keep It Simple Stupid)

Правильное лучше быстрого, простое лучше сложного, понятное лучше крутого, надёжное лучше опасного.

Избегайте неуместной перегрузки операторов.

### Масштабируемость

Правильно выбирайте алгоритмы работы с данными: не оптимизируйте преждевременно, но всегда контролируйте асимптотическую сложность. Алгоритмы должны завершаться за предсказуемое и желательно линейное время.

### Преждевременная оптимизация

«Преждевременная оптимизация – корень всех зол» (Дональд Кнут). Не оптимизируйте раньше времени. Оптимизируйте только то, что показал профайлер. Самая быстрая, надёжная и правильно работающая функция – это функция, которая никогда не вызывается.

Преждевременная оптимизация – это использование более сложного дизайна или кода, а потому менее читабельного, во имя производительности, которая не подтверждена измерениями.

Гораздо легче сделать быстро правильную программу, чем быструю программу правильной.

Поэтому не фокусируйтесь на создании быстрого кода. Сфокусируйтесь на том, чтобы он был как можно более понятным и читабельным.

### Ненужная пессимизация

В случае, когда равноценно с точки зрения понятности и читабельности написать так или иначе, всегда выбирайте более оптимальный вариант. Это не преждевременная оптимизация – это ненужная пессимизация.

Примеры:

* передача в функцию параметров по значению, в то время, как подходит передача по ссылке
* использование постфиксной формы оператора++ вместо префиксной
* использование присваивания в теле конструктора вместо списка инициализации

Преждевременной оптимизацией не является уменьшение количества ненужных временных объектов, создаваемых в результате неявных преобразований, когда это не влечёт к увеличению сложности кода.

### Минимизируйте глобальные и общие (shared) данные

Избегайте общих данных, особенно глобальных данных. Общие данные увеличивают сцепление, что уменьшает надёжность и обычно производительность. Предпочитайте маленькую сцепление и минимизируйте взаимодействие классов. Не забывайте про безопасность с точки зрения потоков.

### Прячьте внутренние данные

Не показывайте внутреннюю информацию сущности, которая предоставляет абстракцию. Это уменьшает связь интерфейса и реализации.

### Используйте RAII (Resource Acquisition Is Initialization)

В С++ есть замечательная возможность языка, которая заключается в симметричности конструкторов/деструкторов. Она отражает симметрию захвата и освобождения ресурсов. Используйте это.

Примеры: fopen/fclose, lock/unlock, new/delete

## Стиль кодирования

### Проверки времени компиляции

Предпочитайте проверки времени компиляции и линковки проверкам времени выполнения.

### Активно используйте const

const – это ваш друг: неизменяемые значения легче понимать, отслеживать, делать выводы. Предпочитайте константы переменным. Пусть const будет по умолчанию, когда вы определяете значение — это безопасно, его можно проверить во время компиляции, он интегрирован в систему типов С++.

### Не используйте макросы там, где можно обойтись без них

«Первое правило макросов: не используйте их до тех пор, пока вы не можете обойтись без них. Практически каждый макрос демонстрирует изъян в языке программирования, программе или программисте.» (Бьерн Страуструп)

Использовать макросы можно только для условной компиляции. Пример – стражи от повторного включения заголовочных файлов.

Идентификаторы макросов пишутся заглавными буквами, символ подчёркивания используется для разделения слов. Например:

MACRO\_SAMPLE

Если макрос внешний, т.е. виден несколькими модулями, то он должен начинаться с префикса XRAY\_<MODULE>\_

Директивы препроцессора #else, #endif, #elif должны сопровождаться комментарием, который должен содержать то условие, которые находится в #if, #ifdef, #ifndef, #elif, включая и саму директиву, т.е.

#if a > 0

…

#else // #if a > 0

…

#endif // #if a > 0

или в более сложном случае

#if a > 1

...

#elif a < -1 // #if a > 1

#else // #elif a < -1

#endif // #if a > 1

Если вы используете вложенные директивы условной компиляции, используйте отступы табуляции следующего вида:

#ifdef MACRO\_0

# ifdef MACRO\_1

…

# else // #ifdef MACRO\_1

…

# endif // #ifdef MACRO\_1

…

#endif // #ifdef MACRO\_0

### Не используйте магические числа

Избегайте использования магических чисел. Используйте вместо них символические имена и выражения, например

width\*aspect\_ratio

### Определяйте переменные, объекты как можно локальнее

Переменные, время жизни которых больше, чем это необходимо, обладают следующими недостатками:

* из-за них программу сложнее понять и поддерживать
* они засоряют контекст своим именем: прямым следствием отсюда является то, что переменные в пространствах имён, которые имеют наибольшую область видимости, являются наихудшими с этой точки зрения
* они не всегда могут быть осмысленно инициализированы: никогда не объявляйте переменную до тех пор, пока вы не можете её осмысленно инициализировать. Неинициализированные переменные являются распространённым источником ошибок в С++ программах и требуют особого внимания, т.к. не всегда могут быть автоматически обнаружены компиляторами

Исключения: иногда есть смысл выносить объявление переменной за пределы цикла.

Т.к. константы не меняют своего значения, то текущий пункт к ним не относится.

### Инициализируйте переменные

Начинайте с чистого листа: неинициализированные переменные являются распространённым источником ошибок в С++ программах. Всегда инициализируйте переменные.

Исключения: в критичных к производительности местах (float3) допускается отсутствие инициализации переменных в Release конфигурации. В Debug конфигурации они должны быть инициализированы «плохими» значениями.

### Избегайте длинных функций и глубоких вложений

Короткое лучше длинного, мелкое лучше глубокого: чрезмерно большие функции и глубоко вложенные блоки часто являются причиной невозможности дать функции одну цельную обязанность. Предпочитайте функциональную декомпозицию, чтобы не заставлять читателя держать “в уме” слишком много. Предпочитайте сделать функцию копированию похожих кусков кода. Предпочитайте && вложенным if-ам. Не используйте switch, подумайте о полиморфном поведении.

### Не используйте статические и глобальные переменные с нетривиальными конструкторами/деструкторами

В конструкторах таких переменных нельзя использовать зависимости от других переменных (нельзя выделять или освобождать память в наших проектах). Если вам подходит явная инициализация, то в некоторых случаях можно использовать следующую конструкцию:

ALIGN( 8 ) char variable\_fake[ sizeof(T) ];

T& variable =

\*static\_cast\_checked< T\* > (

static\_cast\_checked< void\* > (

&variable\_fake[0]

)

);

void initialize( )

{

new ( &variable) T ( );

}

void finalize( )

{

variable.~T( );

}

Такая конструкция принимает, что выравнивание класса равно 8-и. В таких случаях лучше не полагаться на выравнивание по умолчанию и явно указать его в классе:

ALIGN( 8 ) class T {

…

}; // class T

### Используйте форвард декларацию везде, где это возможно

Такой подход позволяет уменьшить зависимость файлов друг от друга и повышает время компиляции. Помните, что форвард декларировать можно не только функции и классы, но и шаблонные функции и классы, а также внутренние, например:

template < typename T >

void foo( T& t );

class A {

class B;

}; // class A

…

class A::B {

}; // class A::B

### Включайте в заголовок все необходимые для его компиляции заголовки

Это правило хорошего тона. Если создать файл, в который включить только один заголовочный файл, то он должен компилироваться без ошибок и ворнингов.

### Используйте защиту от повторного включения заголовков в виде макросов стражей

Каждый .h должен быть защищён от повторного включения стражами:

#ifndef FILE\_NAME\_H\_INCLUDED

#define FILE\_NAME\_H\_INCLUDED

…

#endif // #ifndef FILE\_NAME\_H\_INCLUDED

## Функции и операторы

### Разделяйте входные и выходные параметры функций

Входные параметры:

* всегда используйте const для указателей и ссылок на параметры, которые не изменяются внутри функции
* передавайте по значению переменные маленьких по размеру типов
* остальные параметры передавайте по константной ссылке
* если функция требует копию переменной, рассмотрите возможность её передачи не по ссылке, а по значению

Выходные параметры:

* передавайте по ссылке обязательные аргументы
* передавайте по указателю опциональные параметры или если функция манипулирует владением объекта

### Не используйте ...

Функция с (…) нетипобезопасны, поэтому мы их не используем.

Исключения: функции перенаправления вывода.

### Переопределяйте операторы, только имея существенные основания для этого, и сохраняйте при этом природную семантику

Если так сделать не получается, значит, скорее всего, вам не стоит определять эти операторы. Не стоит добавлять объект в контейнер в виде

a + b

### Если определяете a+b, то определяйте и a+=b

Не забывайте при определении оператора + определять и его более оптимальную версию +=.

### Если определяете ++a, то определяйте и a++

Иногда бывает очень удобно использовать именно постфиксную версию оператора ++, поэтому не забывайте её реализовывать после реализации префиксной формы.

### Не плодите большое количество ненужных временных объектов

Неявное приведение типов может порождать множество ненужных временных объектов. Для уменьшения их количества рассмотрите возможность перегрузки функций:

void foo( std::string const& a, std::string const& b);

void foo( char const\* a, char const\* b);

### Не рассчитывайте на конкретный порядок подстановки значений в выражения

Не стоит писать такой или похожий код:

foo( ++i, ++i );

## Дизайн классов и интерфейсов

### Разные типы классов

Разные типы классов предназначены для разных целей, а потому для каждого вида свои правила.

Классы-значения смоделированы как встроенные типы:

* имеет публичный деструктор, конструктор копирования и оператор присваивания с семантикой значения
* не имеет виртуальных функций
* подразумевается его использование в качестве конкретного класса, а не базового
* экземпляра классы обычно создаются в стеке или содержатся напрямую (не через указатель или ссылку) как член класса

Базовые классы – это строительные блоки иерархий классов:

* имеет публичный виртуальный деструктор или защищённый невиртуальный
* устанавливает интерфейс посредством виртуальных функций
* экземпляры порождённых от него классов обычно создаются в куче и используются с помощью указателя

Traits-классы – шаблоны, которые содержат информацию о типах:

* содержат только typedef-ы и статические функции. Не имеют модифицируемого состояния или виртуальных функций
* экземпляры классов никогда не создаются (создание их обычно запрещается явно)

Классы-стратегии (обычно шаблоны) – фрагменты подключаемого поведения:

* могут иметь или не иметь модифицируемое состояние и виртуальные функции
* экземпляры таких классов обычно не создаются отдельно, только как член класса. Могут быть базовыми классами для других классов

Вспомогательные классы обычно поддерживают какую-то идиому (например, RAII). Они должны быть спроектированы так, чтобы их было легко использовать правильно и тяжело неправильно.

### Маленькие классы

Разделяйте и властвуйте: маленькие классы легче писать, исправлять, тестировать и использовать.

### Предпочитайте композицию наследованию

Наследование – это второй по силе тип сцепленности (coupling) в С++, уступая лишь дружбе. Сильное сцепление является нежелательным, его нужно избегать везде, где это возможно. Предпочитайте композицию наследованию, пока вы не уверены, что это действительно улучшит ваш дизайн.

### Не наследуйтесь от классов, которые не предназначены для наследования

Наследование от классов, которые не предназначены для этого, является серьёзной ошибкой. Авторы классов должны следить за этим, запрещая наследование от своих классов в отладочной конфигурации. Сделать это можно так:

#include <xray/core/debug\_make\_final.h>

…

class class\_identifier : private ::debug::make\_final< class\_identifier > {

…

}; // class class\_identifier

### Предпочитайте наследоваться от абстрактных интерфейсов

Абстрактные интерфейсы помогают сфокусироваться на получении правильной абстракции без её засорения реализацией или деталями состояния. Предпочитайте создавать иерархии, которые реализуют абстрактные интерфейсы, которые моделируют абстрактные концепции.

### Принцип подстановки Лисков (LSP)

Наследуйтесь не для того, чтобы переиспользовать (reuse), но для того, чтобы быть переиспользованными.

Публичное наследование устанавливает между классами отношение «являюсь» (“is-a”), исходя из принципа подстановки Лисков: все базовые контракты должны быть выполнены, поэтому каждая виртуальная функция должна требовать не более чем, и гарантировать не менее чем базовая функция для гарантии контрактов базового класса. Указатель или ссылка на базовый класс должна вести себя корректно даже в случае, если этот указатель или ссылка указывают на производный класс.

### Переопределяйте функции с умом

Хотя порождённые классы обычно добавляют члены данных, они моделируют концепцию подмножеств, а не надмножеств по отношению к базовым классам. При правильном наследовании порождённый класс моделирует отдельный случай более общей базовой концепции.

Не меняйте параметры виртуальной функции по умолчанию. Всегда явно указывайте virtual при переопределении.

### Используйте разделение пользовательского интерфейса и интерфейса кастомизации

Рассмотрите возможность сделать виртуальные функции непубличными, а публичные функции – невиртуальными (NVI - Nonvirtual Interface pattern). Это часто даёт возможность уменьшить затраты при изменении кода.

### Предпочитайте явные приведения неявным

Неявные приведения часто могут сделать больше вреда, чем пользы. Думайте дважды, прежде чем предоставить возможности неявного приведения. По умолчанию запрещайте неявное приведение в конструкторах с одним параметром с помощью ключевого слова explicit.

### Размещайте все члены данных в приватной области скрытия

Используйте публичность только для Си структур, но не требуйте от них инкапсуляции. Не смешивайте приватные и публичные данные в одном классе.

### Не давайте пользователям класса возможность модифицировать внутренние данные класса

Избегайте возвращения дескрипторов на внутренние данные, которыми оперирует ваш класс – тогда клиенты не смогут неконтролируемо менять состояние, которым ваш объект считает, что владеет.

### Используйте идиому Pimpl

Исключение: не используйте эту идиому только в случае явно невыгодного с точки зрения производительности дополнительного уровня косвенности.

### Предпочитайте делать функции не членами класса и не друзьями — это уменьшает сцепление

Не члены класса недружественные функции улучшают инкапсуляцию, минимизируя зависимости: тело функции не может зависеть от непубличных членов класса. Они также разбивают монолитные классы на свободно разделяемые по функциональности части, ещё более уменьшая сцепление.

Таким образом, если функции нужен доступ только к публичным членам класса, то она должна быть не членом класса и не другом.

Исключение: классы, у которых все данные публичные.

### Не определяйте в классах операторы new/delete

В наших проектах мы полагаемся на свой менеджмент памяти.

Исключение: если в процессе поиска утечек памяти будет выяснено, что какой-то из классов требует отдельный менеджмент памяти для уменьшения фрагментации, надо использовать специальный менеджмент для такого класса.

## Конструирование, удаление и копирование

### Определяйте и инициализируйте члены класса в одном и том же порядке

С компилятором надо работать в согласии: в каком бы порядке вы не инициализировали члены класса в списке инициализации, компилятор сгенерирует последовательность инициализации, в том порядке, в котором они объявлены в классе (чтобы фиксировать последовательность вызова деструкторов членов класса на деструкторе самого класса).

### Используйте список инициализации вместо оператора присваивания

Инициализация членов класса в списке инициализации имеет ту же читабельность и аналогичное время работы, что и присваивание в теле конструктора, поэтому стоит использовать более эффективный вариант, иначе это будет пессимизация.

Исключение: избегайте зависимости между членами данных. Если она всё же есть, то выносите инициализацию в тело конструктора. Также выносите в тело конструктора сложные вычисления инициализационных значений (их лучше выносить во внешние функции, но если нужны значения других членов класса, то тогда надо производить вычисления в теле конструктора).

### Деструктор

Отсутствие виртуального деструктора может привести к неправильной последовательности удаления объекта и, как следствие, к утечкам памяти. Поэтому стоит внимательно следить за классами в иерархии классов.

* если вы можете удалять экземпляры класса, имея указатель на какой-то из классов в иерархии, то вы должны определить в базовом классе виртуальный деструктор
* иначе вы должны в каждом классе, кроме финальных определить защищённый невиртуальный деструктор, а финальный класс защитить от порождения

Исключение: интерфейсы, одиночные классы (предикаты).

### Если вы определили один из: конструктор копирования, оператор присваивания или деструктор, то определите и остальные оба

Иначе поведение вашего класса может быть неправильным в случае использования этих функций, сгенерированных по умолчанию.

Если подходит поведение сгенерированных компилятором функций, то опишите этот факт комментарием (можно стандартным).

### Явно запрещайте или разрешайте копирование, либо комментируйте, почему подходит автоматически сгенерированные компилятором версии

По умолчанию каждый класс должен быть защищён от копирования своих экземпляров путём приватного порождения от **noncopyable** (аналог boost::noncopyable). Только в случае, если вы действительно хотите разрешить копировать экземпляры своего класса, вы не защищаться от копирования.

Замечание: если вы решили написать предикат, который может быть скопирован алгоритмом, и потому не предполагает защиту от копирования, помните [об этом](#CommonStandardOperatorBrackets).

Примечание: комментарий может быть стандартным.

Исключение: интерфейсы.

### Избегайте срезания объектов

class B {/\*…\*/};

class D : public B {/\*…\*/};

void bar ( B b )

{

}

void foo( B& b ) {

bar ( b );

}

D d;

foo( d );

### При определении оператора присваивания, в котором вы используете копию переданного объекта, предпочитайте форму оператора, где объект передаётся по значению

Такая функция будет более дружественна компилятору и может помочь сгенерировать более оптимальный код.

## Пространства имён и модули

### Cвободные функции не члены класса должны быть определены в том же пространстве имён, что и сам класс

В противном случае можно получить множество непонятных ошибок компиляции из-за применения правила Кёнига поиска имён.

Пример:

namespace N {

struct X {

/\*…\*/

}; // struct X

template < typename T >

int\* operator+ ( T, unsigned int )

{

/\*…\*/

}

} // namespace N

int main()

{

std::vector< N::X > v( 5 );

v[0];

}

На некоторых реализациях стандартной библиотеки выражение v[0] не компилируется (если operator[] использует выражение begin() + index).

### using пространства имён

Используйте его для удобства, а не для конфликта имён. Никогда не делайте using пространства имён перед включением заголовка. Никогда не делайте using пространства имён в хедере — явно указывайте, кому вы хотите сделать using.

Исключение: разрешается делать using пространства имён в прекомпилированном заголовочном файле.

## Шаблоны и универсальность

### Используйте статический полиморфизм

Статический и динамический полиморфизм дополняют друг друга. Понимайте их преимущества и недостатки, комбинируйте ими для получения лучшего из возможного.

### Зряче располагайте точки кастомизации при написании шаблона

Не стоит делать шаблон «излишне общим». Точки кастомизации шаблона должны быть тщательно продуманы, чтобы не получить ситуации, когда шаблон был кастомизирован необычным для автора способом и в итоге дал неправильный результат.

### Избегайте написания необоснованно необщего кода

Если вам необходимо в шаблоне использовать !=, то используйте его, а не конкретный оператор <, >. Проверяйте на пустоту контейнера container.empty(), а не container.size() == 0.

## STL: контейнеры

### Старайтесь размещать в контейнерах типы-значения

Иначе вы потеряете уйму времени на конструирование объектов

### Старайтесь использовать push\_back вместо других способов

Исключение: конечно же, не забывайте о reserve и resize в случае добавления множества элементов.

### Используйте вставку диапазона вместо поэлементной вставки

Это гораздо оптимальнее.

### Идиомы shrink-to-fit, erase-remove

Для того, чтобы контейнер занимал ровно столько места, сколько ему необходимо, используйте

container( other ).swap( other );

container( ).swap( other );

Никогда не сохраняйте итератор, возвращённый алгоритмом std::remove или std::remove\_if, удаляйте элементы сразу же в одном выражении:

container.erase(

std::remove\_if(

container.begin( ),

container.end( ),

predicate( )

),

container.end( )

);

## STL: алгоритмы

### Используйте подходящий поисковый алгоритм

Ищите «ровно столько, сколько нужно», не делайте лишней работы. Для поиска в несортированных данных используйте **find**/**find\_if**, **count**/**count\_if**. Для поиска в сортированных данных используйте **lower\_bound**, **upper\_bound**, **equal\_range** или (редко) **binary\_search** (несмотря на своё название binary\_search обычно редко бывает правильным выбором). Предпочитайте для сортированных данных

p = std::equal\_range( first, last, value);

distance( p.first, p.second );

вместо

count( first, last, value );

### Используйте подходящий алгоритм сортировки

Не всегда нужна именно сортировка, обычно нужно меньше, реже – больше. В порядке от самых дешёвых до самых дорогих по сложности алгоритмов: **partition**, **stable\_partition**, **nth\_element**, **partial\_sort**, **sort**, **stable\_sort**

### Не разрешайте предикатам менять своё состояние во время вызова оператора(), которое влияет на результат оператора()

Некоторые алгоритмы копируют предикаты, поэтому использование таких возможностей ведёт к нетривиальным ошибкам.

### Старайтесь передавать в качестве предикатов не функции, а объекты — они обычно производят более быстрый код

## Безопасность типов

### Не используйте switch по типу объекта для кастомизации поведения

Используйте для этого один из видов полиморфизма.

Исключение: в местах, где нужна максимальная производительность разрешается использовать такую конструкцию.

### Полагайтесь на типы, а не их представление

Следующие пункты стандарт не гарантирует:

* int не равен ни 32 битам, ни какому либо другому фиксированному размеру
* указатели и int-ы не обязательно имеют один и тот же размер и не могут быть свободно преобразованы один в другой
* расположение экземпляра класса в памяти необязательно размещает базовые классы и члены данных в порядке их определения (исключение – POD-типы)
* offsetof работает только для POD-типов, хотя компилятор может и не генерировать ошибки
* класс может иметь скрытые поля
* указатели могут вообще не иметь свойств целых чисел. Если два указателя упорядочены, и вы приведёте их к целым числам, то порядок при этом может быть другим
* ничего нельзя предполагать о расположении автоматических переменных или о направлении роста стэка
* указатели на функции могут иметь размер, отличный от void\*
* нельзя записать любой объект в любое адрес памяти, даже если достаточно памяти для его размещения из-за выравнивания

Просто определяйте типы. Читайте и пишите данные, используя и думая в типах, а не в битах, словах и адресах.

### static\_cast, static\_cast\_checked и pointer\_cast

Не используйте static\_cast напрямую, используйте защищённую версию static\_cast\_checked, которая в случае полиморфных типов проверит их на приводимость.

В случае несвязных типов используйте pointer\_cast, который приведёт static\_cast через void\*, но проверит несвязность перед этим.

На границе API для приведения несвязных типов можно использовать псевдоним static\_cast\_checked-a – cast.

### Не используйте reinterpret\_cast

Если вы обманываете компилятор – он отмстит вам (Генри Спенсер).

В случае приведения T1\* в T2\*, когда типы не связаны, вместо

T1\* p1 = …;

T2\* p2 = reinterpret\_cast< T2\* >( p1 );

используйте

T1\* p1 = …;

T2\* p2 = pointer\_cast< T2\* >( p1 );

### Не используйте const\_cast

Применение const\_cast иногда приводит к неопределённому поведению, и это знак плохого стиля программирования, даже если оно легально.

Исключения: использование const\_cast в неправильно спроектированных API сторонних производителей.

### Не используйте C styl cast

Никогда не используйте это приведение, т.к. оно в себе сочетает и const\_cast, и reinterpret\_cast.

Исключение: разрешено приведение для встроенных типов.

### Не используйте приведение с помощью union (т.н. horrible\_cast)

Никогда не используйте это приведение, т.к. оно самое сильное из всех предыдущих.

Исключение: если у вас 2 POD типа, и они начинаются с одинаковых полей, то можно записать одно из общих полей и прочитать другое общее поле.

### Не используйте varargs (...)

Избегайте небезопасную к типам передачу параметров. Используйте шаблоны и перегруженные функции.

### Не используйте незащищённые строковые функции

Используйте только те строковые функции, которых есть размер буфера, с которым производятся манипуляции (strcpy\_s, например).

### Не используйте memcpy и memcmp ни для чего, кроме как для сырой памяти

memcpy и memcmp нарушают систему типов. Использовать memcpy для создания объектов, это как печатать деньги с помощью фотокопировального устройства. Может даже показаться, что инструменты и методы работают, но они слишком грубы, чтобы сделать это приемлемо.

### Не интерпретируйте С массивы полиморфно

Необходимо помнить, что при статическом приведении типов значения указателя может измениться.

## Оформление кода

### Разделительные символы

В качестве разделительных символов следует использовать табуляцию и пробел. Табуляция используется для отступов выравнивания (indentation). Пробел используется для всего остального. Размер табуляции должен быть равен 4-м символам пробела.

Пробел требуется для разделения списка аргументов через запятую, например:

a, b, c

а не

a,b,c

При вызове функций не регламентируется, стоит ли ставить пробел после открывающей скобки. Это же касается объявления шаблонных параметров и специализаций шаблонов. Зато фиксируется последовательность автора класса в использовании таких пробелов. Т.о. не должно быть спешения разных стилей, вроде

foo ( a, b );

…

bar(a, b);

### .h

Каждый .h должен иметь заголовок следующего вида:

////////////////////////////////////////////////////////////////////////////

// Created : 30.01.2009

// Author : Dmitriy Iassenev

// Copyright(C) GSC Game World - 2009

////////////////////////////////////////////////////////////////////////////

Такой заголовок поможет при переходе на другие программы контроля версий, а также будет служить защитой от копирования файлов внутри проекта. Для того чтобы постоянно не писать этот код вручную, можно сделать трафарет или даже визард для всех.

Каждый .h должен быть защищён от повторного включения стражами:

#ifndef SMART\_COVER\_MANAGER\_H\_INCLUDED

#define SMART\_COVER\_MANAGER\_H\_INCLUDED

…

#endif // #ifndef SMART\_COVER\_MANAGER\_H\_INCLUDED

Оптимальный размер .h файла — около 80 строк (включая заголовок и стражи), оптимальная длина строки - не более 96-и символов. Если файл сильно превышает оптимальные величины — рассмотрите возможности форматирования и рефакторинга.

Описанные в .h файле функции/методы классов должны иметь следующее вертикальное выравнивание:

* тип функции: статическая/встроенная/виртуальная
* возвращаемые значения + конвенция вызова
* идентификаторы функций/методов класса
* параметры функций/методов класса

Например:

class manager {

public:

manager ( );

virtual ~manager ( );

virtual void foo ( ) const;

inline bool actual ( ) const;

private:

static void process ( bool something ) const;

}; // class manager

Если какой-то метод не помещается в общий строй, то можно его выравнивать хоть по какой-то из вертикалей, например

class manager {

public:

manager ( );

virtual ~manager ( );

virtual void foo ( ) const;

inline bool actual ( ) const;

very\_very\_long\_type bar ( ) const;

private:

static void process ( bool something ) const;

}; // class manager

Допускается применение разных выравниваний для разных групп функций.

### .сpp

Каждый .сpp должен иметь заголовок следующего вида:

////////////////////////////////////////////////////////////////////////////

// Created : 30.01.2009

// Author : Dmitriy Iassenev

// Copyright (C) GSC Game World - 2009

////////////////////////////////////////////////////////////////////////////

Оптимальный размер .cpp файла - не более 200 строк (включая заголовок), оптимальный размер строки — около 64-х символов. Если файл сильно превышает оптимальные величины — рассмотрите возможности рефакторинга и форматирования.

Рекомендуется для описанных и реализованных в .cpp файле функций/методов классов следующее вертикальное выравнивание:

* параметры функций/методов класса
* левая и правая стороны выражений в реализации функций/методов класса

Например:

manager::manager ( )

{

}

manager::~manager ( )

{

}

void manager::foo ( ) const

{

int const b = 0;

process (true);

}

void manager::process\_static ( bool const something ) const

{

int const n = 10;

bool found = false;

for ( int i=0; i<n; ++i ) {

if (!condition)

continue;

process ( false );

found = true;

break;

}

}

### Скобки

Где именно размещать скобки стандарт не регламентирует, но он регламентирует последовательность автора.

### Классы

Каждый класс должен заканчивать свою декларацию следующим комментарием:

class class\_identifier {

…

}; // class class\_identifier

### Пространства имён

Каждое пространство имён класс должен заканчивать свою декларацию следующим комментарием:

namespace namespace\_identifier {

…

}; // namespace namespace\_identifier

### Шаблонные параметры

Шаблонные параметры нужно писать, начиная слова с больших букв, без разделения между ними, например.

template < typename ClassName >

class policy : public ClassName {…};

Допускается использование стандартного названия шаблонного параметра: T или же множества параметров: T0, T1, …

template < typename T >

void foo ( )

{

…

}

### if

После каждого оператора if, должна следовать пустая строка.

{

if ( a )

b ( );

c ( );

}

Нельзя располагать действие в той же строке, что и сам оператор или его else. Выражение в скобках должно быть выделено пробелами с обеих сторон для улучшения читабельности. Выражение внутри форматируется по усмотрению автора в наиболее читабельный вид.

{

if ( a )

b ( );

else

c ( );

}

Допускается использование следующей конструкции:

{

if ( condition0 )

b ( );

else if ( condition1 )

c ( );

else if ( condition2 )

d ( );

}

### for

После каждого цикла for, должна следовать пустая строка:

{

for ( int i=0; i<n; ++i )

foo ( i );

bar ( );

}

Избегайте использования слишком большого количества пробелов в описании цикла, как вот здесь:

{

for ( int i = 0; i < n; ++i )

foo ( i );

}

### switch

Для оформления switch можно использовать следующие варианты:

switch ( expression ) {

case value0 : {

foo ( );

break;

}

case value1 : {

bar ( );

break;

}

default : NODEFAULT;

}

Там, где это возможно, не используйте поведение по умолчанию, лучше перечислите все значения, для которых вам это нужно:

switch ( expression ) {

case value0 :

case value1 :

case value2 :

case value3 : {

default\_behaviour ( );

break;

}

default : NODEFAULT;

}

Таким образом, при добавлении нового типа, вы получите гарантированный вылет во время тестирования, а не странное поведение по умолчанию.

### goto

Не используйте этот оператор. Разрешается использовать этот оператор только для оптимизации функций, которые профайлер показал, как критические с точки зрения производительности.

## Динамическое приведение типов

Старайтесь не использовать динамическое приведение типов, обычно это свидетельствует об ошибках в иерархии классов. Чтобы обойтись без динамического приведения типов иногда хватает полиморфного поведения классов (виртуальные функции), в случаях сложных устоявшихся иерархий классов можно рассмотреть возможность использования паттерна проектирования «визитор».

Если вы всё же решили, что вам необходимо использовать именно динамическое приведение типов, обратите внимание на иерархию классов, внутри которой будет происходить это приведение, т.к. от этого существенно зависит скорость приведения.

## Встроенные функции

Встроенные функции (**inline**) нужно использовать **только** в следующих случаях:

* для accessors (функциям доступа к приватным членам класса)
* для шаблонных функций
* для функций шаблонных классов
* для функций, которые профилировщик показал как такие, которые вызываются большое количество раз, и которые имеет смысл встраивать (т.е. для которых генерируется небольшое количество ассемблерных инструкций)

Во всех вышеперечисленных случаях встраивать функции не **можно**, а **обязательно**.

В случае приватной встроенной функции, которая используется в одной единице компиляции (.cpp файле), нужно реализовывать встроенную функцию непосредственно в единице компиляции.

В случае однострочных функций-accessor-ов можно не писать inline в заголовочном файле.

Можно также не писать inline в шаблонных функциях и функциях шаблонных классов.

Реализация встроенных функций должна описываться в файлах, которые называются следующим образом:

<namespace\_id0>\_<namespace\_id1>\_...\_<namespace\_idN>\_<class\_id>\_inline.h

В случае, если шаблонный класс помещается на один экран (40 строк), можно не создавать файл \_inline.h

В случае, когда реализация встроенной функции требует включения дополнительных заголовков, её реализацию следует вынести в файл с названием

<namespace\_id0>\_<namespace\_id1>\_...\_<namespace\_idN>\_<class\_id>\_impl.h

в начале которого включить все необходимые заголовки, а его самого включать только в модулях, которые используют эти встроенные функции.

## Внешние функции/Методы класса

Данная секция в равной степени относится как ко внешним/свободным функциям, так и к методам классов. Далее в этой секции мы под функцией будем понимать, как внешнюю/свободную функцию, так и метод класса.

### Идентификаторы

Идентификаторы функций пишутся прописными буквами, символ подчёркивания используется для разделения слов. Например:

function\_sample

### Параметры

Параметры функций должны отвечать следующим требованиям:

* передача по значению (**void foo ( float parameter )**) – параметр может передаваться по значению тогда и только тогда, когда его значение меняется внутри функции или вызываемых ею функций или же, когда размер типа не более 8 байт. Во втором случае в реализации в случае, если параметр не меняется внутри функции, стоит указать const перед ним.
* передача по указателю (**void foo (raw<float>::ptr parameter)**) – параметр может передаваться через указатель тогда и только тогда, когда необходимо, чтобы изменённое функцией значение по указателю передалось наружу, и нулевой указатель допустим в качестве параметра. Если сам указатель не может меняться внутри функции, то в реализации необходимо указать const перед ним.
* передача по ненулевому указателю (**void foo (non\_null<float>::ptr parameter)**) – параметр может передаваться через указатель тогда и только тогда, когда необходимо, чтобы изменённое функцией значение по указателю передалось наружу, и нулевой указатель недопустим в качестве параметра. если сам указатель не может меняться внутри функции, то в реализации необходимо указать const перед ним. Этим методом (константным ненулевым указателем) стоит пользоваться, когда идёт преобразование из сырых указателей (float\*) в защищённые указатели, т.е. когда есть риск того, что будет безнаказанно разыменован нулевой указатель. Это обычно происходит на стыке модулей движка с другими модулями. Если это происходит внутри модуля, то стоит пользоваться неконстантными ссылками — они нагляднее и удобнее.
* передача по неконстантной ссылке (**void foo (float& parameter)**) – параметр может передаваться через неконстантную ссылку тогда и только тогда, когда необходимо, чтобы изменённое функцией значение передалось наружу, нулевой указатель не допустим в качестве параметра, и этот указатель не может быть изменён внутри функции.
* аналогичные правила стоит применять и для константных типов ( **raw<float const>::ptr, non\_null<float const>::ptr, float const&**)

### Возвращаемый результат

Для возвращаемого результата лучше всего использовать константную ссылку там, где это возможно, т.е., например, если вы в методе класса возвращаете значение члена класса, то верните его через константную ссылку. Это позволяет избегать ненужного копирования. Для небольших типов данных (до 8 байт) имеет смысл использовать именно копирование.

Если в функции для возврата результата заводится переменная, она должна называться **result**.

### Локальные переменные

Правила именования локальных переменных совпадает с правилом именования параметров функции.

К локальным переменным применяются те же правила, что и для параметров функции.

### Уровни вложенности

Оптимальный уровень вложенности конструкций внутри функции равен двум:

void foo ( int const n )

{

for ( int i=0; i<n; ++i ) {

if ( condition ) {

…

continue;

}

…

}

}

Избегайте использования else в if-ах, стройте функцию таким образом, чтобы она после условия сразу же делала выход.

Вместо

void foo()

{

for ( int i=0; i<n; ++i ) {

if ( a ) {

…

}

else {

if ( b ) {

}

else {

if ( c )

;

else

;

}

}

}

}

используйте

void foo( )

{

for ( int i=0; i<n; ++i ) {

if ( a ) {

…

continue;

}

if ( b ) {

…

continue;

}

if ( c ) {

…

continue;

}

…

}

}

или даже:

void foo\_impl( int index )

{

if ( a ) {

…

return;

}

if ( b ) {

…

return;

}

if ( c ) {

…

return;

}

…

}

void foo( )

{

for ( int i=0; i<n; ++i )

foo\_impl ( i );

}

Исключение: если код с else понятнее, то стоит использовать именно его.

### Сложные выражения

Рекомендация: не используйте сложные выражения в условных операторах, разбивайте их и заменяйте на более простые, вплоть до выноса в отдельную функцию. Оптимальная сложность выражения:

if ( pointer && (pointer->foo( ) == value) )

Во всех остальных случаях выражения стоит дробить на более мелкие. Например, вместо

if ( ( (a == b) && (c() == d ) ) || (e() != f() ) ) {

…

}

можно рассмотреть такой вариант

static bool class\_name::condition (type0 const& a, type1 const& b, type2 const& d) const

{

if ( a != b )

return e( ) == f( );

if ( c( ) != d );

return e( ) == f( );

return true;

}

…

if (condition(a, b, d)) {

}

Препятствуйте расползанию кода вправо. Всегда старайтесь после условий выполнять небольшое количество кода и выходить из функции, продолжать или прерывать цикл. Старайтесь избегать else в if по этой же причине.

Однако не забывайте, что главный критерий хорошего кода — его понятность. Т.о. если код с else понятнее, то стоит использовать именно его.

### \_alloca

Там, где это возможно пытайтесь выделять память не в куче (heap), а в стеке с помощью функции **\_alloca**. Часто это убирает ненужные ограничения и динамические выделения памяти. Например, если необходимо скопировать строку, не стоит пользоваться буфером символов фиксированного размера, т.к. он всегда может переполниться. Также не стоит выделять память в куче, т.к. это медленно и может привести к фрагментации (например, в случае многопоточности или выделений памяти функциями, которые вы вызываете). Вместо этого вы можете выделить необходимое количество памяти в стеке и скопировать туда строку. Тут важно отметить, что очень большие объекты могут переполнить отведённый размер стека. Если такое случится, то есть два способа решения проблемы:

* не выделять крупные объекты в стеке
* увеличить размер стека для исполняемого потока (нежелательно)

## Внешние функции

### Пространства имён

Каждая внешняя функция должна быть в своём пространстве имён.

### static

Если функция используется только в одном модуле, она должна быть объявлена как статическая. Старайтесь локализовать функции как можно сильнее. Чем выше локальность — тем ниже сцепление.

## Классы

### Файлы класса

Декларация класса должна находиться в файле

<namespace\_id0>\_<namespace\_id1>\_...\_<namespace\_idN>\_<class\_id>.h

В одном файле должно находиться определение одного класса. Допускается определение вспомогательных POD структур и внутренних классов. Важно, чтобы при этом сохранялась читабельность кода. Также помните [об этом](#CommonStandardForwardDeclarations).

Реализация методов класса должна находиться в файле

<namespace\_id0>\_<namespace\_id1>\_...\_<namespace\_idN>\_<class\_id>.cpp

Для встроенных функций, не требующих для себя включения дополнительных заголовков необходимо использовать файл

<namespace\_id0>\_<namespace\_id1>\_...\_<namespace\_idN>\_<class\_id>\_inline.h

Для встроенных функций, требующих для себя включения дополнительных заголовков необходимо использовать файл

<namespace\_id0>\_<namespace\_id1>\_...\_<namespace\_idN>\_<class\_id>\_impl.h

### Пространство имён

Каждый класс должен быть помещён в соответствующее ему пространство имён.

### Идентификаторы

Идентификаторы классов пишутся, в нижнем регистре каждое, символ подчёркивания служит разделением слов, например:

сlass\_identifier

### Наследование

При наследовании классов, список базовых классов нужно писать вот так:

class A :

public B,

protected C,

private noncopyable

{

}; // class A

Исключение: если базовый класс один, то его можно записать в той же строке:

class A : private noncopyable {

}; // class A

### Методы

Идентификаторы методов класса пишутся в нижнем регистре, символ подчёркивания служит разделением слов, например

class\_method

Методы класса, также как и наследование, должны быть максимально защищены от внешнего вмешательства.

Поэтому лучший метод класса – это статическая функция, определённая в модуле, где она используется и не видна снаружи (т.е. даже не фигурирующая в описании класса).

Если же этой функции необходим доступ к приватным или защищённым типам/нумераторам класса/статическим членам, тогда она должна быть приватной статической функцией класса.

Если же этой функции нужен экземпляр класса (this), тогда она должна быть константной приватной функцией класса.

Если же эта функция меняет значение неизменяемых (non mutable) членов класса, тогда она должна быть неконстантной приватной функцией класса.

Если же эта функция нужна наследникам этого класса, она становится неконстантной защищённой функцией класса.

Если же эта функция нужна пользователям экземпляров этого класса, она становится неконстантной публичной функцией класса.

Если же эта функция переопределяет или предполагает своё переопределение в наследниках класса, и при этом, кто-то из пользователей класса может иметь указатель не на финальный класс, либо же сам класс подразумевает полиморфизм в этой функции, тогда эта функция должна быть неконстантной публичной виртуальной функцией.

Обратите внимание на то, сколько необходимо условий для того, чтобы метод класса был публичным неконстантым и виртуальным. Следите крайне внимательно за тем, насколько сильно вы защитили свой код от внешнего вмешательства.

По умолчанию аксессоры (accessors) должны быть встроенными функциями.

Аксессоры должны называться по имени члена данных, доступ к которому они предоставляют, используйте префиксы get\_... и set\_..., например:

class A : private noncopyable {

public:

inline type const& get\_class\_member ( ) const;

inline void set\_class\_member ( type const& value );

private:

type m\_class\_member;

}; // class A

### Члены данных

Идентификаторы членов данных класса пишутся в нижнем регистре, символ подчёркивания служит разделением слов. Каждый идентификатор класса должен начинаться с m\_, например m\_class\_member. Если все члены данных класса являются публичными, то они пишутся без префикса m\_: class\_member.

Члены данных также должны быть максимально защищены.

Не используйте статические члены данных с нетривиальными конструкторами и деструкторами ([смотрите здесь](#CommonStandardGlobals)).

По умолчанию член данных должен быть приватным.

Если член данных нужен наследникам, он должен быть защищённым.

Если член данных нужен пользователям экземпляров класса – только тогда он должен быть публичным.

Если член данных – это указатель или ссылка, то по умолчанию она должна быть константная, только если класс должен уметь изменять её данные, тогда она должна быть неконстантная.

Публичные члены данных допускаются только в типах, которые имеют только публичные члены данных. Во всех остальных случаях они должны быть доступны только через аксессоры (accessors).

Все члены данных должны быть инициализированы в конструкторе в списке инициализации. Если у вас возникла ситуация, когда такое поведение нежелательно в виду зависимостей членов класса друг от друга или исходя из соображений производительности, стоит воспользоваться инициализацией в теле конструктора.

### Выравнивание (padding) членов данных

Если вы не группируете члены данных по функциональности, то располагайте их всегда в порядке уменьшения их размера.

### Перечислимые типы (enum)

На перечислимые типы действуют всё те же правила сокрытия, что и на члены данных, поэтому правила на области объявления private/protected/public являются такими же.

Правила именования идентификаторов являются следующими: сам тип должен использовать нижний регистр, используя символ подчёркивание для разделения слов, члены типа должны начинаться с префикса типа, например:

enum enum\_simple\_name {

simple\_name\_id0,

}; // enum enum\_simple\_name

Для сложных названий можно использовать сокращения в виде акронимов (аббревиатура из первых букв слов, cn в данном случае, это сокращение от complex\_name)

enum enum\_complex\_name {

cn\_id0,

}; // enum enum\_complex\_name

Безымянные перечислимые типы использовать можно, в этом случае, префикс типа не используется.

### Пользовательские типы (typedef)

На пользовательские типы действуют всё те же правила сокрытия, что и на члены данных.

Идентификаторы пользовательских типов данных класса пишутся в нижнем регистре, символ подчёркивания служит разделением слов, а заканчиваться на \_type, например:

typedef complex\_expression complex\_expression\_type;

### Конструктор копирования

По умолчанию каждый класс должен быть защищён от копирования своих экземпляров путём приватного порождения от **noncopyable**. Только в случае, если у вы действительно хотите разрешить копировать экземпляры своего класса, вы не защищаться от копирования.

### Виртуальный деструктор

Отсутствие виртуального деструктора может привести к неправильной последовательности удаления объекта и, как следствие, к утечкам памяти. Поэтому, стоит внимательно следить за базовыми классами иерархии классов ([смотрите здесь](#CommonStandardDestructor)).

### Невозможность порождения

В предыдущем пункте в качестве защиты от неправильной последовательности удаления, упоминалось защита класса от порождения. Для того, чтобы она не влияла на производительность в релизной конфигурации, существует шаблон debug::make\_final, который используется следующим образом:

#include <xray/core/debug\_make\_final.h>

namespace class\_namespace {

class class\_identifier : private core::debug::make\_final< class\_identifier > {

…

}; // class class\_identifier

} // namespace class\_namespace

Допускается не защищаться от порождения в одиночных классах, которые ни от кого не наследованы, и от которых никто не наследован.

### Порядок описания

Порядок описания методов и членов данных класса следующий: сначала идут публичные члены, затем – защищённые, затем – приватные.

### Пример декларации и реализации класса

([смотрите на следующей странице](#ExamplesClass))

file: smart\_cover\_manager.h

////////////////////////////////////////////////////////////////////////////

// Created : 06.11.2007

// Author : Dmitriy Iassenev

// Copyright (C) GSC Game World

////////////////////////////////////////////////////////////////////////////

#ifndef SMART\_COVER\_MANAGER\_H\_INCLUDED

#define SMART\_COVER\_MANAGER\_H\_INCLUDED

#include “script\_export\_space.h”

#include <xray/core/debug\_make\_final.h>

namespace smart\_cover {

class manager :

private noncopyable,

private core::debug::make\_final< manager >

{

public:

manager ( );

virtual ~manager ( );

inline bool member ( ) const;

inline void member ( bool value ) const;

private:

bool m\_member;

}; // class manager

} // namespace smart\_cover

#include “smart\_cover\_manager\_inline.h”

#endif // #ifndef SMART\_COVER\_MANAGER\_H\_INCLUDED

file: smart\_cover\_manager\_inline.h

////////////////////////////////////////////////////////////////////////////

// Created : 06.11.2007

// Author : Dmitriy Iassenev

// Copyright (C) GSC Game World

////////////////////////////////////////////////////////////////////////////

#ifndef SMART\_COVER\_MANAGER\_INLINE\_H\_INCLUDED

#define SMART\_COVER\_MANAGER\_INLINE\_H\_INCLUDED

namespace smart\_cover {

inline bool manager::member ( ) const

{

return ( m\_member );

}

inline void manager::member ( bool const value )

{

m\_member = value;

}

} // namespace smart\_cover

#endif // #ifndef SMART\_COVER\_MANAGER\_INLINE\_H\_INCLUDED

file: smart\_cover\_manager.cpp

////////////////////////////////////////////////////////////////////////////

// Created : 06.11.2007

// Author : Dmitriy Iassenev

// Copyright (C) GSC Game World

////////////////////////////////////////////////////////////////////////////

#include “pch.h”

#include “smart\_cover\_manager.h”

using smart\_cover::manager;

manager::manager ( ) :

m\_member ( false )

{

}

manager::~manager ( )

{

}

# Стандарт на проект Сталкер

## Организационные вопросы

### Уровень ворнингов

Во всех проектах мы используем самый высокий уровень ворнингов. Мы не подавляем ворнинги с помошью #pragma warning(disable:...), мы изменяем код таким образом, чтобы ворнинги исчезали.

Исключения:

* в случае если мы используем стороннюю библиотеку и понимаем причину ворнингов, мы можем отключить ворнинги для включения хедера сторонней библиотеки
* в случае, когда компилятор генерирует ворнинг, и нет никакой возможности его отключить, изменив код.

У всех проектов во всех конфигурациях уже настроены эти параметры.

### Система контроля версий

Мы используем систему контроля версий Subversion.

Не держите файлы в незакоммиченном состоянии долго.

Делайте коммиты часто. Если вы планируете начать новую задачу, которая требует более дня работы, заведите отдельную ветку.

Не переносите файлы с места на место без помощи системы контроля версий — таким образом, мы теряем историю изменений файлов.

Для предотвращения бессмысленных коммитов, установите в ОС, которую вы используете знак десятичной запятой ‘.’ (точка).

### Система review

Мы не используем централизированную систему review, но в системе контроля версий Subversion есть возможность подписаться на изменения. По умолчанию каждый автор подписывается на свои файлы, но вы можете подписываться и на другие файлы. Не пропускайте изменения, на которые вы подписались без внимательного их изучения. Часто ошибки можно найти даже до начала тестирования.

### Автоматические билды

Мы не используем централизированную систему автоматических билдов, но планируем её использование на будущее. Система будет уметь собирать солюшин во всех конфигурациях и в случае успеха коммитить его в сабвершин тестерам/дизайнерам.

## Работа с памятью

В проекте мы не используем ни кучу процесса, ни кучу библиотеки времени выполнения. Мы используем память, выделенную в самом начале программы. Именно поэтому выделять и освобождать память в статических/глобальных переменных нельзя.

Для локальности выделений памяти мы используем несколько арен. Сейчас есть функциональное разделение (по модулям), планируется в будущем добавить и разделение по данным (уровень, персонажи, меню и т.д.()ь функциональное разделение (удущем разделениесколько арен. 5454545454545454545454545454545454545454545454545454545454545454545454545454545454545454545454545454545454545454545454545454545454).

Для модулей, которым требуется динамическая память, существуют стандартные контейнеры, находящиеся в пространстве имён модуля. Именно их и нужно использовать в сущностях модуля. Также для модуля определяются макросы выделения памяти: NEW, DELETE, MALLOC, REALLOC, FREE, ALLOC.

Динамические выделения/освобождения памяти фрагментируют память. Поэтому мы пытаемся максимально избегать частых выделений памяти. Для этого мы используем внедрённые контейнеры, контейнеры фиксированного размера (во время компиляции или во время выполнения). Если мы всё же выбрали контейнер, который использует динамические выделения памяти, то желательно при этом зарезервировать предполагаемое пространство для элементов, ведь у многих контейнеров есть метод reserve.

Не забывайте и [об этом](#CommonStandardShrinkToFit).

## Библиотеки

### структура

Библиотеки проекта должны иметь чёткую структуру, и сделаны по образу и подобию специального проекта default, который является примером библиотеки. Все файлы библиотеки, которые должны видеть другие проекты, должны находиться в директории include/xray/<library>. Всё, что определяет библиотека, должно находиться в пространстве имён xray::<library>. Библиотека должна определить интерфейсы xray::<library>::engine и xray::<library>::world.

Библиотека должна иметь как минимум следующие файлы:

* + xray/<library>/api.h – определяет интерфейс библиотеки – **единственные** внешние функции библиотеки create\_world, destroy\_world и memory\_allocator
  + xray/<library>/engine.h – определяет интерфейс, через который библиотека будет общаться с движком
  + xray/<library>/world.h – определяет интерфейс, через который движок будет общаться с библиотекой
  + xray/<library>/library\_linkage.h – при включении этого заголовочного файла линкует на себя движок

### файлы проекта

Библиотека должна содержать следующие файлы, которые не должны быть видны извне:

* + pch.h/pch.cpp – прекомпилированные заголовочные файлы, файл находится в фильтре “core” проекта
  + <module>\_memory.h – здесь включено всё, что нужно для менеджмента памяти (в том числе и контейнеры) , файл находится в фильтре “core” проекта
  + <module>\_library\_linkage.cpp – сюда включается заголовки других библиотек, которые использует данная библиотека (например, #include <xray/core/library\_linkage.h>), файл находится в фильтре “core” проекта
  + <module>\_entry\_point.cpp – здесь определены все интерфейсные функции библиотеки, файл находится в корне проекта

Все включения заголовочных файлов из директории include должны выглядеть вот так:

#include <xray/core/library\_linkage.h>

Для разделения директориев допускается только прямой слэш, т.е. символ ‘/’.

Не разрешается добавлять директории в опции проекта Include Directories.

Расположение файлов проекта на диске должно быть «плоским», т.е. без поддиректориев. Это сделано для облегчения включений файлов, избегания коллизий объектных файлов. В качестве исключения разрешено использовать небольшое количество поддиректориев, но запрещено менять свойство Include Directories в свойствах проекта.

В то же время, структура директориев внутри директория include может содержать поддиректории. Там могут быть только заголовочные файлы, причём все свои включения они должны делать только через угловые скобки, во избежание коллизий при включении из разных папок.

Не разрешается включать файлы из директории include в обычных кавычках, а не угловых скобках – это замедляет время компиляции.

Каждый проект должен иметь фильтр include, в котором должен содержать все файлы, которые находятся в директории include/<module>, дублируя структуру поддиректориев, если таковые имеются.

Пространства имён должны совпадать с директориями и поддиректориями. Однако, от автора требуется предоставлять доступ к сущностям своего модуля с использованием не более двух пространств имён. Вложенность директориев более двух уровней не обязывает автора создавать аналогичные пространства имён, но обязывает писать префиксы в названии классов, например: include/xray/render/base/game\_renderer.h может содержать класс xray::render::base\_game\_renderer.

Исключение: проект core создан для обеспечения базовой функциональности всего движка, и состоит из разнообразных подсистем. Поэтому он (и только он ) экспортирует свои сущности в пространство имён xray, при этом экспортируя и часть непосредственно своей функциональности в своё пространство имён (xray::core). Его заголовочные файлы находятся в директориях include/xray и include/xray/core.

### прекомпилированные заголовочные файлы

Все прекомпилированные заголовочные файлы должны называться либо pch.h, либо pch\_<something>.h, а соответствующие им создателя прекомпилированных заголовочных файлов pch.cpp, либо pch\_<something>.cpp соответственно.

Никогда не включайте прекомпилированные заголовочные файлы в другие заголовочные файлы. Такое включение возможно только в другие прекомпилированные заголовочные файлы. Т.е. можно включить pch.h в заголовочный файл pch\_seomthing.h

Все включения прекомпилированных заголовочных файлов должны следовать сразу же после шапки файла, следующим включением должен быть соответствующий заголовчный файл. Т.е. в файле something.cpp, должны быть включены по очереди pch.h и something.h

### платформо-зависимая функциональность

Для использования платформо-зависимых вещей разделяйте код по разным файлам, например: engine.cpp, engine\_win32.cpp, engine\_xbox360.cpp и т.д. Только в таких файлах можно включать заголовочный файл <xray/core/os\_include.h>. Не включайте <windows.h> напрямую. Файлы для разных платформ надо называть единообразно: file\_win32.cpp/h, file\_xbox360.cpp/.h, file\_ps3.cpp/.h

Если используются всем нужные платформо-зависимые механизмы, которые не относятся только к самой платформе (функции big\_endian/little\_endian), то их стоит вынести в пространство имён xray::core::platform.

Есть несколько методов разделения платформо-зависимого кода:

* один интерфейс – несколько реализаций в разных классах
* один класс – разные реализации в разных файлах
* условные константные выражения внутри реализации
* условная компиляция внутри реализации

Эти методы отсортированы по степени читабельности и поддерживаемости кода - от лучших к худшим. Используйте последние два пункта только в случае крайней необходимости. Код, реализованный с помощью таких техник трудно читаемый.

Рекомендуется избегать использования условной компиляции для разделения кода для разных платформ. Рекомендуется также избегать условных выражений с константными условиями для разделения кода.

## Строки

Интенсивное использование строк может привести к существенной фрагментации памяти и потерям производительности.

### strings::shared

Этот класс предназначен для строк, которые повторяются, и для которых мы хотим сэкономить память и выиграть в скорости сравнения, в случае, когда нам нужно ввести хоть какое-то отношение порядка на множестве строк.

Его имеет смысл использовать тогда, когда вы уверены, что строк с таким содержимым будет несколько.

Также его имеет смысл использовать в виде ключей различных контейнеров, но при этом в качестве предиката сравнения надо брать оператор меньше.

**std::string**

Не используйте этот класс.

### buffer\_string

Класс строки, использующий переданный на конструкторе буфер для содержания строки и поддерживающий различные операции с ней.

### fixed\_string

Класс строки, использующий буфер статического размера для содержания строки и поддерживающий различные операции с ней.

### strings::join vs sprintf\_s

strings::join конкатенирует несколько строк в одну и записывает результат в переданный буфер. Всегда предпочитайте её функции sprintf\_s там, где это возможно, т.к. она гораздо эффективнее.

Кроме того, предпочитайте этот макрос для конкатенации строк нескольким вызовам функции strcat\_s, т.к. он работает гораздо эффективнее.

### STR\_JOINA

STR\_JOINA конкатенирует несколько строк в одну, буфер для которой выделяется в стеке. Всегда предпочитайте использовать этот вариант функции strings::join, т.к. он никогда не переполнит буфер, разве что не хватит стека.

### strings::duplicate

Функция strings::duplicate выделяет память с помощью переданного аллокатора под новую строку и копирует туда содержимое переданной строки. Не забывайте, что память, выделенную этой функцией необходимо освободить с помощью функции free соответствующего аллокатора.

## Контейнеры

Проект повсеместно использует стандартную библиотеку шаблонов, поэтому самим не нужно писать новые типы контейнеров, они все уже написаны, если нужен какой-то особенный – подходите за консультацией к ведущему программисту. В проекте используется собственный аллокатор памяти, поэтому использовать контейнеры со стандартным аллокатором запрещено, для всех видов контейнеров есть их эквиваленты с нужным аллокатором.

### module::vector

Использовать этот контейнер надо стараться для value-классов в случае, если удаления из середины вектора редки. Очень хорошо, если вы заранее знаете количество элементов – обязательно зарезервируйте место для них, перед заполнением вектора.

### module::deque

Редко используемый контейнер, по сути, то же, что и вектор, но позволяет удалять относительно бесплатно начальный элемент, к тому же, лучше вектора работает с удалениями/вставками в середину.

### module::stack

Редко используемый, но полезный контейнер, используя его вместо вектора, вы подсказываете о способе работы с ним.

### module::list

Очень редко используемый контейнер, практически неприменим в движке из-за большого количества выделений памяти. Кроме того, память может быть выделена в разных адресах памяти, что увеличивает промахи по кэшу данных.

### module::map

Часто используемый контейнер, рекомендуется использовать при частых удалениях/вставках, при необходимом быстром поиске по ключу. Однако, в случае, когда порядок элементов не важен, рекомендуется его использовать только для большого количества элементов (более 128-256), т.к. из-за особенностей архитектуры нынешних компьютеров, линейный поиск не уступит поиску по дереву из-за промахов по памяти, а занимать место в памяти будет меньше, и итерация по элементам будет на порядок быстрее. В случае, когда элементов немного, и работа с ними ведётся неинтенсивно, рекомендуется использовать associative\_vector (о нём читайте ниже).

### module::multimap

То же, что и для map, в случае, если ключи неуникальны.

### module::set

Полезный, но редко используемый контейнер, т.к. в нём тяжело искать по ключу, не имея самого элемента. Тем не менее, если ключ агрегируется самим элементом, то он должен быть предпочтён map.

### module::multiset

То же, что и для set, в случае агрегации ключа самим элементом должен быть предпочтён multimap

### boost::array

Используйте этот контейнер, как замену C style array, не забывайте, что есть fixed\_vector.

### C style array

Не используйте эти массивы.

## Специальные контейнеры

### associative\_vector

Контейнер используется для небольшого количества элементов (128-256) вместо map. Также может использоваться и для большого количества элементов, если они заполняются один раз на загрузке, и потом не меняются, а только используются. Полностью повторяет интерфейс module::map, хотя по сути своей является вектором.

### buffer\_vector

Этот контейнер ведёт себя как стандартный вектор, за исключением того, что он не выделяет память, а использует тот буфер, который ему передали на конструкторе. Стандартное его использование подразумевает первоначальное выделение буфера, затем вы можете работать с ним, как с обычным вектором. Удобно использовать для векторов, для которых память выделена в стеке (например, с помощью \_alloca).

void some\_class::foo ( module::vector< u32 > const& items )

{

u32 const count = items.size( );

buffer\_vector< u32 > temp(

\_alloca( count\*sizeof( u32 ) ),

count,

items.begin( ),

items.end( )

);

temp.erase (

std::remove\_if(

temp.begin( ),

temp.end( ),

some\_predicate( this )

),

temp.end( )

);

m\_items.insert ( m\_items.end( ), temp.begin( ), temp.end( ) );

std::sort ( m\_items.begin( ), m\_items.end( ) );

m\_items.erase (

std::unique(

m\_items.begin( ),

m\_items.end( )

),

m\_items.end( )

);

}

### fixed\_vector

Этот контейнер ведёт себя как стандартный вектор, за исключением того, что он не выделяет память, а использует буфер на заданное количество элементов. Такие векторы можно использовать в стеке.

**Внедрённые (intrusive) контейнеры**

Если возникла ситуация, когда объекты предназначены для хранения в одном единственном контейнере, имеет смысл рассмотреть вариант внедрённого контейнера. Он имеет ряд преимуществ:

* + не выделяет дополнительную память
  + часто доступ к элементам получается быстрее за счёт того, что контейнер не выделяет узлы в памяти, а содержит их внутри объектов

Такой контейнер имеет и свои ограничения: использовать объекты можно только в одном контейнере.

## Обработка ошибок и протоколирование

Всегда проверяйте все пред- и постусловия функций с помощью следующих макросов:

* COMPILE\_ASSERT – проверка времени компиляции
* ASSERT – проверка времени выполнения в Debug конфигурациях
* R\_ASSERT – проверка времени компиляции в Debug и Release конфигурациях, если макрос MASTER\_GOLD не определён
* NODEFAULT – достигнуто место, которое не должно было быть достигнуто
* FATAL – сообщение о фатальной ошибке (без условия)
* CHECK\_OR\_EXIT – сообщение об ошибке для пользователя игры с выходом
* CHECK\_AND\_CORRECT – проверка условия, выполнения кода в случае невыполнения его и продолжение работы

Примеры использования:

void foo ( int parameter )

{

ASSERT( parameter >= 0, “invalid parameter passed %d”, parameter);

}

Для логгирования мы используем макросы LOG\_TRACE, LOG\_DEBUG, LOG\_INFO, LOG\_WARNING и LOG\_ERROR. В них необходимо передавать формат и параметры, как это делается для функции printf. Также есть макросы LOGI\_TRACE, LOGI\_DEBUG, LOGI\_INFO, LOGI\_WARNING, LOGI\_ERROR, которые выводят в лог с дополнительной информацией о том, кто выводит, например:

LOGI\_WARNING ( “systems/subsystem” ) ( … )

Макрос LOG\_TRACE работает только в Debug конфигурациях.

Макрос LOG\_DEBUG работает только в Debug и Release конфигурациях.

Все остальные макросы работают во всех остальных конфигурациях, включая и конфигурации, в которых определён макрос MASTER\_GOLD.

# Ссылки

1. Bjarne Stroustrup, “The C++ Programming Language”
2. Herb Sutter, Andrei Alexandrescu, “C++ Coding Standards: 101 Rules, Guidelines, and Best Practices”