C++11 FAQ

Примечание переводчика

Данный FAQ переводится на русский язык с любезного разрешения его автора Бьярна Страуструпа. На данный момент работа над этим документом еще не закончена, поэтому по мере изменения или дополнения оригинала, будет изменяться и перевод. Все пожелания/замечания по качеству перевода присылайте по почте напрямую, либо через профиль Google.

Последние изменения в оригинале: 26.11.2013

C++11 – принятый недавно новый ISO стандарт языка C++

Этот документ написан и поддерживается Бьерном Страуструпом (Bjarne Stroustrup). Конструктивные комментарии, поправки, ссылки и предложения всецело поддерживаются. Сейчас я работаю над полнотой материала и приведением в порядок ссылок.

C++11 – это стандарт языка C++ утвержденный Международной организацией по стандартизации (ISO) в 2011-м году. Предыдущие версии стандарта обычно называют C++98 или C++03. Различия между C++98 и C++03 настолько специфические и их настолько мало, что на них можно не обращать внимания.

Доступна последняя версия рабочего документа, и она близка к окончательной версии черновика стандарта, формально принятого единогласным решением (21-0) в августе 2011 года.

До официального утверждения, будущий стандарт называли C++0x. К сожалению, у меня пока что не было времени везде обновить имя стандарта, вы уж простите, и вообще, мне очень нравится название C++ 0x :-). Название “C++ 0x” осталось с тех давних пор, когда мы надеялись, что новый стандарт будет называться C++08 или C++09. Сейчас можно рассматривать “x” в имени, как признак шестнадцатеричного формата (т.е. C++0B == C++11).

Все официальные документы, связанные со стандартом C++11/C++0x можно найти на официальном веб-сайте комитета по стандартизации. Официальное имя комитета SC22 WG21.

Предупреждение: текущий документ в течение некоторого времени будет изменяться. Комментарии, вопросы, ссылки, замечания и предложения приветствуются.

Цель

Цель этого C++11 FAQ:

- Дать обзор новых возможностей (возможностей языка и стандартных библиотек), предлагаемых языком C++11 по сравнению с предыдущими версиями стандарта языка C++.
- Дать представление о целях работы комитета по стандартизации языка C++.
- Показать новые возможности языка с точки зрения программиста.
- Дать ссылки для более глубокого изучения новых возможностей.
- Назвать имена многих людей, кто оказал неоценимый вклад (в основном авторов докладов, написанных для комитета). Стандарт разработан не безликой организацией.

Обращаю внимание, что целью данного FAQ не является полноценное описание конкретных возможности языка или детальное объяснениеКакова была цель создания C++11? их использования. Цель этого документа – показать простые примеры, демонстрирующие возможности языка C++11 (и

дать дополнительные ссылки). Идеальный вариант – «максимум одна страница для описания одной возможности» не зависимо от ее сложности. Подробности всегда могут быть найдены в дополнительных ссылках.

Список вопросов

Вот некоторые высокоуровневые вопросы:

- А что вы думаете о C++11?
- Когда будет принят официальный стандарт C++0x?
- Когда компиляторы реализуют поддержку C++11?
- Когда будут доступны новые стандартные библиотеки?
- Какие новые языковые возможности появились в C++11?
- Какие новые языковые возможности появились в C++11?
- Что нового в стандартной библиотеке C++11?
- Какими конкретными целями руководствовался комитет по стандартизации?
- Как мне найти документы комитета по стандартизации?
- А где я могу найти научные и технические статьи о C++11?
- Где еще я могу почитать о C++11?
- А есть ли видео материалы о C++11?
- Сложно ли учить C++11?
- Как работает комитет по стандартизации?
- Кто является членом комитета?
- Будет ли C++1y?
- Что случилось с «концептами»?
- Есть ли возможности языка, которые вам не нравятся?

Вопросы об отдельных возможностях языка:

- __cplusplus
- Auto – вывод типа из инициализатора
- Range-for оператор
- Правые угловые скобки
- Управление поведением по умолчанию: **default** и **delete**
- Управление поведением по умолчанию: копирование и перемещение
- enum class – строготипизированные перечисления с дополнительной областью видимости
- constexpr – обобщенные гарантировано константные выражения
- decltype – тип выражения
- Списки инициализации
- Предотвращение сужения (narrowing) типов
- Делегирующие конструкторы
- Инициализация членов класса при объявлении
- Унаследованные конструкторы
- Статические утверждения (утверждения времени компиляции) – static_assert
- long long -- более длинное целое
- nullptr -- литерал для задания нулевого указателя
- template alias (известные ранее как “template typedef”)
- Variadic Templates
- Унифицированый синтаксис и семантика инициализации

- Rvalue ссылки
- Объединения
- POD типы
- Сырые строковые литералы
- Пользовательские литералы
- Атрибуты
- Лямбда-выражения
- Локальные типы в шаблонных аргументах
- noexcept – предотвращение проброса исключений
- Выравнивание (alignment)
- Управление переопределением функций: override
- Управление переопределением функций: final
- Возможности C99
- Расширенные целочисленные типы
- Динамическая инициализация и разрушение в многопоточной среде
- thread-local storage (thread_local)
- Юникодные символы
- Копирование и повторная генерация исключений
- Extern templates
- Inline namespace
- Операторы явного преобразования
- Модель памяти

Обычно я беру примеры из соответствующих предложений (proposals). Спасибо их авторам. Многие примеры заимствованы из моих собственных работ и выступлений.

Список вопросов об отдельных возможностях стандартной библиотеки:

- Улучшения алгоритмов
- Улучшения контейнеров
- Аллокаторы с дополнительным состоянием
- std::array
- std::forward_list
- Неупорядоченные контейнеры
- std::tuple
- Метапрограммирование и характеристики типов
- std::function и std::bind
- unique_ptr
- shared_ptr
- weak_ptr
- Garbage collection ABI
- Потоки
- Взаимное исключение
- Блокировки
- Условные переменные

- Работа со временем
- Атомарные операции
- std::future и std::promise
- std::async()
- Завершение процесса
- Генерация случайных чисел
- Регулярные выражения

Ниже находятся ответы на все приведенные выше вопросы.

А что вы думаете о C++11?

Это невероятно часто задаваемый (мне) вопрос. Я даже думаю, что это самый часто задаваемый вопрос. Удивительно, но C++11 воспринимается как абсолютно новый язык: разные возможности языка подогнаны друг к другу так, как никогда ранее, и более высокоуровневый стиль программирования стал более естественным и все таким же эффективным. Если вы относитесь к языку C++, как к лучшему C, или просто как к объектно-ориентированному языку, тогда вы не поняли главного. Новый язык предоставляет более гибкие и доступные абстракции, чем раньше. Вспомните старое правило: если вы думаете о *чем-то*, как об отдельном объекте или понятии, выразите это в программе; моделируйте объекты реального мира и абстракции непосредственно в коде. Теперь, это сделать намного проще: вы можете выразить свои мысли при помощи перечислений, объектов, классов (например, задавая поведение по умолчанию), иерархий классов (например, с помощью наследуемых конструкторов (inherited constructors), шаблонов, синонимов (aliases), исключений, циклов, потоков и т.д., а не использовать универсальной (“one size fits all”) механизм абстракций.

Моя мечта состоит в том, чтобы возможности языка помогали программистам думать о дизайне и реализации системы по-другому. Мне кажется, что C++11 может помочь в этом, и не только для C++ программистов, но и для тех, кто работает с другими современными языками программирования в такой обширной области как системное программирование.

Короче говоря, я все еще настроен оптимистично :)

Когда будет принят официальный стандарт C++0x?

Он уже принят! Первый черновик для формального утверждения был предоставлен в сентябре 2008 года. Окончательный Международный Черновик стандарта (FCD – Final International Draft) был единогласно одобрен комитетом по стандартизации 25-го марта 2011 года. И он был официально принят единогласным решением в августе 2011-го. Стандарт был опубликован в этом году (2011).

Следуя соглашению, новый стандарт был назван C++11 (поскольку он был опубликован в 2011-м). Лично я использую просто C++ и добавляю год только когда мне нужно различить разные версии языка C++, такие как ARM C++, C++98 и C++03. Во время переходного периода я все еще иногда использую C++0x. И вы можете рассматривать “x”, как признак шестнадцатеричного формата.

Когда компиляторы реализуют поддержку C++11?

Существующие компиляторы (например, GCC C++, Clang C++, IBM C++ и Microsoft C++) уже реализовали многие возможности C++11. Например, многие компиляторы реализуют полностью (или почти полностью) новые возможности стандартной библиотеки.

С выходом новых версий компиляторов я ожидаю появление поддержки все новых и новых возможностей. Я ожидаю увидеть первый компилятор с полной поддержкой возможностей C++11 в

2012-м году, но я не хочу гадать, когда выйдет этот компилятор или когда все компиляторы будут поддерживать все возможности C++11. Хочу заметить, что все возможности C++11 уже были кем-то и когда-то реализованы, так что уже существует определенный опыт, на который могут положиться разработчики компиляторов.

Вот некоторые ссылки о C++11 от производителей компиляторов:

- Сравнение поддержки компиляторами возможностей C++
- GCC
- IBM
- Microsoft
- EDG
- Clang

Когда будут доступны новые стандартные библиотеки?

Новые версии стандартных библиотек сейчас поставляются вместе с реализациями от GCC, Clang и Microsoft, а также доступны в boost.

Какие новые языковые возможности появились в C++11?

Нельзя улучшить язык программирования, просто добавив все возможности, которые показались кому-то интересными. На самом деле, мне предлагали включить в язык C++ практически каждую возможность современных языков программирования; представьте себе как бы выглядел коктейль из C99, C#, Java, Haskell, Lisp, Python и Ada. Кроме того, не забывайте о том, что старые возможности удалять нельзя, даже если комитет согласится, что они неудачные: как показывает опыт, пользователи заставляют поставщиков компиляторов поддерживать устаревшие или запрещенные возможности десятками лет с помощью ключей компилятора (либо эти возможности могут быть включены по умолчанию).

В попытке выбрать рациональное звено из потока предложений, мы выработали набор конкретных целей. Мы не могли следовать им в точности и они не были настолько полными, чтобы комитет мог следовать им неукоснительно (да и, IMO, это было невозможно).

В результате мы получили язык со значительно улучшенным механизмом абстракций. Был существенно расширен набор абстракций, которые могут быть выражены на языке C++ элегантно, гибко и с минимумом затрат по сравнению с самописными решениями. Когда мы говорим «абстракция», то люди обычно думают о «классах» и «объектах». C++11 идет значительно дальше: количество пользовательских типов, которые могут быть ясно и безопасно выражены на языке C++, существенно возросло после добавлении списков инициализации, обобщенной инициализации, синонимов шаблонов, rvalue ссылок, удаленных функций и функций по умолчанию (defaulted and deleted functions), а также шаблонов с переменным числом аргументов (variadic templates). Реализация абстракций упрощается с помощью таких возможностей, как auto, наследуемые конструкторы (inherited constructors) и decltype. Этих изменений достаточно, чтобы относиться к C++11, как к новому языку программирования.

Список доступных возможностей языка см. в списке новых возможностей.

Что нового в стандартной библиотеке C++11?

Я бы хотел видеть большее количество стандартных библиотек. Однако уже сейчас определение стандартной библиотеки занимает 70% текста стандарта (и это без учета стандартной библиотеки C,

которая включена в стандарт в виде ссылки). И хотя некоторые из нас хотели бы видеть в стандартных библиотеках массу других возможностей, никто не может обвинить рабочую группу стандартной библиотеки в нерасторопности. Стоит также отметить, что библиотеки C++98 были существенно улучшены путем использования новых языковых возможностей, таких как списки инициализации, rvalue ссылки, шаблоны с переменным числом аргументов, ключевого слова noexcept и constexpr. Стандартной библиотекой C++11 проще пользоваться и она работает быстрее, чем стандартная библиотека C++98.

Список доступных библиотек см. в списке библиотечных компонентов.

Какова была цель создания C++11?

C++ является языком программирования общего назначения, с некоторым уклоном в системное программирование. Этот язык:

- улучшенный C
- поддерживает абстракцию данных
- поддерживает объектно-ориентированное программирование
- поддерживает обобщенное программирование

Стандарт C++11 был призван решить следующие задачи:

- Сделать язык C++ более подходящим для системного программирования и разработки библиотек, т.е. внести вклад для всего сообщества пользователей языка C++, а не предоставить набор возможностей для конкретного под-сообщества (например, для численных вычислений или для разработчиков приложений под Windows).
- Сделать язык C++ проще для обучения и изучения путем унификации возможностей, более строгих гарантий и направленности на новичков (новичков всегда будет больше, чем экспертов).

Естественно, что все это сделано с жесткими ограничениями обратной совместимости. И лишь изредка комитету пришлось нарушить работающий код, и то, только при добавлении новых ключевых слов (таких как static_assert, nullptr и constexpr).

Подробнее об этом см. в:

- B. Stroustrup: What is C++0x?. CVu. Vol 21, Issues 4 and 5. 2009.
- B. Stroustrup: Evolving a language in and for the real world: C++ 1991-2006. ACM HOPL-III. June 2007.
- B. Stroustrup: A History of C++: 1979-1991. Proc ACM History of Programming Languages conference (HOPL-2). March 1993.
- B. Stroustrup: C and C++: Siblings. The C/C++ Users Journal. July 2002.

Какими конкретными целями руководствовался комитет по стандартизации?

Естественно, что у разных людей и у разных организаций, вовлеченных в процесс стандартизации, были несколько разные цели, особенно, когда дело касалось деталей и приоритетов. Кроме того, с течением времени изменялись и конкретные цели. Помните, что комитет даже не имеет возможности сделать все, что единогласно было признано нужным, поскольку состоит из

добровольцев с весьма ограниченными ресурсами. Однако, ниже представлен набор критериев, которыми пользовались при обсуждении того, какие возможности и библиотеки подходят для C++11:

- Поддержка стабильности и обратной совместимости; не ломать старый код, а если без этого не обойтись, то не делать это втихую.
- Отдавать предпочтение библиотекам, перед новыми языковыми возможностями – цель, которую комитет не смог достигнуть полностью; слишком многие члены комитета и сообщества предпочитают «настоящие языковые возможности».
- Отдавать предпочтение обобщению, а не специализации – сосредоточиться на улучшении механизма абстракций (классов, шаблонов и т.д.).
- Поддерживать и новичков, и экспертов; новичкам поможет улучшения библиотек и обобщения правил; экспертам же нужны более общие и эффективные возможности.
- Улучшить безопасность типов в основном за счет возможностей, которые позволяют избегать небезопасных с точки зрения типов возможностей.
- Улучшить производительность и возможность работы с оборудованием на прямую; сделать C++ еще более подходящим инструментом для встроенных (embedded) систем и высокопроизводительных вычислений.
- Соответствовать потребностям реального мира; не забывать об инструментах, стоимости реализации, проблемах миграции, проблемах бинарных интерфейсов (ABI – Application Binary Interface), проблемах обучения и изучения и т.д.

Обратите внимание, что главной и самой сложной задачей является одновременное использование старых и новых возможностей языка. Целое значительно больше простой суммы составляющих.

С другой стороны, комитет также затронул области использования и стилей использования языка:

- Аппаратная модель и многопоточность; обеспечить более строгие гарантии и облегчить использование современного аппаратного обеспечения (например, многоядерные процессоры и слабо упорядоченную модели памяти (weakly coherent memory model)). Например, ABI (Application Binary Interface) потоков, типы future, локальная память потоков и ABI атомарных операций.
- Обобщенное программирование (GP – Generic Programming); GP является одним из самых заметных успехов C++98 и мы хотим улучшить его поддержку, основываясь на полученном опыте. Например, с помощью auto и синонимов шаблонов (template aliases).
- Системное программирование; улучшить поддержку низкоуровневого программирования (например, разработку низкоуровневых встроенных систем) и улучшить производительность. Например, с помощью constexpr, std::array и обобщенных POD (Plain-Old Data) типов.
- Разработка библиотек; устранить ограничения, неэффективность и нарушения механизма абстракций. Например, с помощью встроенных пространств имен (inline namespace), наследуемых конструкторов и rvalue-ссылок.

Как мне найти документы комитета по стандартизации?

Посмотрите раздел с документами на веб-сайте комитета. Там вы наверняка увязните в деталях. Посмотрите на «список вопросов» (issues list) и списки «состояний» (например, State of Evolution (July 2008)). Ключевые группы комитета следующие:

- Core (CWG – Core Working Group) – занимается техническими вопросами языка и формулировками.

- Evolution (EWG – Evolution Working Group) – занимается языковыми возможностями и проблемами интеграции языковых возможностей и библиотек.
- Library (LWG – Library Working Group) – занимается предложениями об изменении библиотек.

Здесь вы можете найти последний черновик стандарта C++11.

А где я могу найти научные и технические статьи о C++11?

- Saeed Amrollahi: Modern Programming in New Millenium: A Technical Survey on Outstanding features of C++0x. Computer Report (Gozaresh-e Computer), No.199, November 2011 (Mehr and Aban 1390), pages 60-82. (in Persian)
- Mark Batty et al's: Mathematizing C++ concurrency, POPL 2012. // thorough, precise, and mathematical.
- Gabriel Dos Reis and Bjarne Stroustrup: General Constant Expressions for System Programming Languages. SAC-2010. The 25th ACM Symposium On Applied Computing.
- Hans-J. Boehm and Sarita V. Adve: Foundations of the C++ concurrency memory model. ACM PLDI'08.
- Hans-J. Boehm: Threads Basic. HPL technical report 2009-259 // ``what every programmer should know about memory model issues''
- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++. OOPSLA'06, October 2006. // The concept design and implementation as it stood in 2006; it has improved since, though not sufficiently to save it.
- Douglas Gregor and Jaakko Jarvi: Variadic templates for C++0x. Journal of Object Technology, 7(2):31-51, February 2008.
- Jaakko Jarvi and John Freeman: Lambda functions for C++0x. ACM SAC '08.
- Jaakko Jarvi, Mat Marcus, and Jacob N. Smith: Programming with C++ Concepts. Science of Computer Programming, 2008. To appear.
- M. Paterno and W. E. Brown : Improving Standard C++ for the Physics Community. CHEP'04. // Much have been improved since then!
- Michael Spertus and Hans J. Boehm: The Status of Garbage Collection in C++0X. ACM ISMM'09.
- Verity Stob: An unthinking programmer's guide to the new C++ -- Raising the standard. The Register. May 2009. (Humor (I hope)).
- [N1781=05-0041] Bjarne Stroustrup: Rules of thumb for the design of C++0x.
- Bjarne Stroustrup: Evolving a language in and for the real world: C++ 1991-2006. ACM HOPL-III. June 2007. (incl. slides and videos). // Covers the design aims of C++0x, the standards process, and the progress up until 2007.
- B. Stroustrup: What is C++0x?. CVu. Vol 21, Issues 4 and 5. 2009.
- Anthony Williams: Simpler Multithreading in C++0x. devx.com.

Этот список, скорее всего неполный и вероятно он станет неактуальным после выхода новых работ. Если вы знаете работу, которая достойна этого списка, но здесь ее нет, пожалуйста, пришлите мне ее. Кроме того, некоторые работы устареют после внесения в стандарт последних изменений. Я постараюсь поддерживать комментарии в актуальном состоянии.

Где еще я могу почитать о C++11?

Количество информации о C++11 увеличивается после завершения процесса стандартизации и после того, как компиляторы начинают реализовывать библиотеки и поддерживать новые языковые возможности. Вот краткий список дополнительных источников.

- Раздел с документами на веб-сайте комитета.
- Черновик стандарта C++0x.
- Страница в Википедии, посвященная C++11. Эта страница, видно, активно развивается, хотя, явно не членами комитета.

- Список поддерживаемых возможностей C++11.

А есть ли видео материалы о C++11?

(Для всех, кто знает меня лично, данный раздел является доказательством того, что это действительно FAQ, а не набор моих любимых вопросов; я не большой фанат видео материалов на техническую тему; мне кажется, что видео выступления сбивают с толку и благодаря многословному формату увеличиваются шансы появления мелких технических ошибок).

Да:

- B. Stroustrup, H. Sutter, H-J. Boehm, A. Alexandrescu, S.T.Lavavej, Chandler Carruth, and Andrew Sutter: several talks and panels from theGoing Native 2012 conference.
- Herb Sutter: Writing modern C++ code: how C++ has evolved over the years. September 2011.
- Herb Sutter: C++ and Beyond 2011: Herb Sutter - Why C++?. August 2011.
- Try Google videos.
- Lawrence Crowl: Lawrence Crowl on C++ Threads. in Sophia Antipolis, June 2008.
- Bjarne Stroustrup: The design of C++0x at U of Waterloo in 2007.
- Bjarne Stroustrup: Initialization at Google in 2007.
- Bjarne Stroustrup: C++0x -- An overview. in Sophia Antipolis, June 2008.
- Lawrence Crowl: Threads.
- Roger Orr: C++0x. January 2008.
- Hans-Jurgen Boehm: Getting C++ Threads Right. December 2007.

Сложно ли учить C++11?

Ну, поскольку мы не можем удалить ни одну существенную возможность из языка C++, не поломав огромное количество кода, то C++11 больше, чем C++98. Так что если вы хотите знать каждое правило, то изучить C++11 будет сложнее. Нам остается только две возможности, упрощающие обучение (с точки зрения учеников):

- Обобщение: заменить, скажем, три правила одним обобщенным правилом (например, универсальная инициализация, наследуемые конструкторы и потоки).
- Дать более простые альтернативы: предоставить новые возможности, которые проще использовать, по сравнению с аналогичными старыми возможностями (например, array, auto, range-for оператор и regex).

Очевидно, что стиль обучения типа «снизу вверх» нивелирует все эти преимущества, и сейчас (явно) слишком мало материалов, проповедующие другой подход к обучению. Со временем это должно измениться.

Как работает комитет по стандартизации?

Комитет по стандартизации SC22 WG21 работает по правилам работы подобных комитетов ISO. Весьма забавно, что эти правила не стандартизированы и постоянно изменяются.

Во многих странах есть свои национальные комитеты по стандартизации с действующими группами по языку C++. Эти группы участвуют в митингах, общаются посредствам веба и некоторые из них посылают представителей на заседания комитетов ISO. Канада, Франция, Германия, Швейцария и США представлены практически на каждом заседании. Дания, Нидерланды, Япония, Норвегия, Испания и другие реже присутствуют лично.

Большая часть работы выполняется между заседаниями с помощью веба, и результаты оформляются в виде многочисленных документов на вебсайте WG21.

Встречи комитета проходят два-три раза в год, длительностью в одну неделю. Основная работа этих заседаний проходит в рабочих подгруппах, таких как “Core”, “Library”, “Evolution” и “Concurrency”. В случае необходимости проводятся специальные встречи рабочих групп по определенным срочным темам, таким как «концепты» (concepts) или «модель памяти» (memory model). Голосование проводится на основном заседании. В начале, в рабочих группах проводятся «неофициальные опросы», для определения готовности обсуждения вопроса всем комитетом. Затем, голосует комитет целиком (один член комитета – один голос), и затем, если решение принято, проходит национальное голосование. Мы уделяем большое внимание, чтобы не попасть в ситуацию, когда большинство комитета согласно, а национальные комитеты – нет; продолжение заседаний в такой ситуации приведет к бесконечным спорам. Окончательное голосование национальными комитетами по официальному черновику стандарта было проведено по электронной почте.

Комитет имеет тесные связи с группой стандартизации языка C (SC22 WG14) и POSIX, и менее формальные контакты с другими группами.

Кто является членом комитета?

В состав комитета входит большое число людей (порядка 200), 60 из которых принимают участие в недельных встречах дважды или трижды в год. Кроме того, существуют национальные группы по стандартизации, которые проводят заседания в разных странах. Большая часть членов комитета вносят свой вклад либо путем участия в заседаниях, принимают участие в переписке по электронной почте или путем представления различных документов на рассмотрение комитету. У большинства участников также есть друзья и коллеги, которые им помогают. С самого начала, членами комитета были представители разных стран и на каждом заседании присутствовали представители 5-10 стран. В окончательном голосовании участвовали около 20 представителей национальных комитетов. Так что, стандартизация языка C++ - это весьма глобальное начинание, а не *затея* небольшой сплоченной команды людей, создающей идеальный язык программирования, для «таких же людей, что и они сами». Стандарт – это лучший компромисс, который устраивает все заинтересованные стороны.

Естественно, что многие (но не все) члены комитета в своей повседневной жизни работают на C++. Членами комитета являются разработчики компиляторов, инструментальных средств, разработчики библиотек и приложений (немногие), исследователи (всего несколько), консультанты, разработчики инструментов автоматического тестирования и многие другие.

Вот лишь короткий список организаций, принимающих участие в процессе стандартизации: Adobe, Apple, Boost, Bloomberg, EDG, Google, HP, IBM, Intel, Microsoft, Red hat, Sun.

Вот короткий список имен, с которыми вы можете быть знакомы по литературе или публикациям в интернете: Dave Abrahams, Matt Austern, Pete Becker,Hans Boehm, Steve Clamage, Lawrence Crowl, Beman Dawes, Francis Glassborow, Doug Gregor, Pablo Halpern, Howard Hinnant, Jaakko Jarvi,John Lakos, Alisdair Meredith, Jens Maurer, Jason Merrill, Sean Parent, P.J. Plauger, Tom Plum, Gabriel Dos Reis, Bjarne Stroustrup, Herb Sutter,David Vandevoorde, Michael Wong. Мои извинения более двумстам текущим и прошлым членам комитета, которых я не упомянул в этом списке. Также обратите внимание на список авторов различных документов: стандарт написан множеством людей, а не безликим комитетом.

Чтобы лучше познакомиться с опытом членов комитета, можете обратиться к списку авторов на странице WG21, однако помните, что многие ключевые участники процесса стандартизации написали не так и много документов.

Будет ли C++1y?

Почти наверняка, даже несмотря на то, что комитет засиделся с C++11. Я часто слышу мнение о том, что для сокращения срока стандартизации, комитет должен начать работать над C++1y сразу же после утверждения C++11. Десятилетний интервал между стандартами – это слишком длительный период времени для текущего темпа развития технологии, поэтому некоторые предлагают трехлетний интервал. Лично я склоняюсь, что пятилетний интервал является более реалистичным. Так что, ждем C++16?

Что случилось с «концептами»?

«Концепты» (concepts) – это была возможность языка, предназначенная для спецификации точных требований к аргументам шаблона. К сожалению, комитет решил, что дальнейшая работа над «концептами» может серьезно затянуть процесс стандартизации, поэтому он проголосовал за удаление этой возможности из рабочего документа. Более подробную информацию о причинах удаления можете посмотреть в моей заметке The C++0x “Remove Concepts” Decision и A DevX interview on concepts and the implementation for C++0x.

Поскольку скорее всего, в том или ином виде «концепты» будут частью новой версии языка C++, я решил не удалять из этого документа соответствующий раздел, но перенес его в конец документа:

- Аксиомы (axioms) (семантические утверждения)
- Концепты (concepts)
- Карты концептов (concepts map)

Есть ли возможности языка, которые вам не нравятся?

Да. Также мне не нравятся некоторые возможности из C++98, например, макросы. Тут дело не в том, нравится мне та или иная возможность или нахожу ли я ее полезной или нет. Дело в том, чтобы нашелся кто-то, способный убедить остальных поддержать его идею. Или некая техника настолько укоренилась в сообществе программистов, что требует прямой реализации в языке.

__cplusplus

В C++11 макрос **__cplusplus** будет возвращать новое значение, большее текущего значения **199711L**.

Auto – вывод типа из инициализатора

Рассмотрим следующий пример:

```
auto x = 7;
```

В данном случае тип переменной **x** будет **int**, потому что именно такой тип имеет ее инициализатор. В общем случае мы можем написать:

```
auto x = expression;
```

И тип переменной **x** будет равен типу значения, полученному в результате вычисления «выражения».

Ключевое слово auto для вывода типа переменной из ее инициализатора, наиболее полезно, когда точный тип выражения не известен, либо сложен в написании. Рассмотрим пример:

```
template<class T> void printall(const vector<T>& v)
```

```
        {
                for (auto p = v.begin(); p!=v.end(); ++p)
                        cout << *p << "\n";
        }
```

В C++98, вам бы пришлось писать:

```
        template<class T> void printall(const vector<T>& v)
        {
                for (typename vector<T>::const_iterator p = v.begin();
p!=v.end(); ++p)
                        cout << *p << "\n";
        }
```

Написать код без использования ключевого слова **auto** может быть особенно сложно, когда тип переменной тесно связан с типами аргументов шаблона. Например:

```
        template<class T, class U> void multiply(const vector<T>& vt, const
vector<U>& vu)
        {
                // ...
                auto tmp = vt[i]*vu[i];
                // ...
        }
```

Тип переменной **tmp** должен зависеть от результата умножения **T** на **U**, но человеку может быть очень сложно определить конкретный результирующий тип, хотя, конечно же, компилятор легко справится с этой задачей, после того как будут известны конкретные типы **T** и **U**.

Отличительная особенность этой возможности состоит в том, что это была самая первая предложенная и реализованная возможность: у меня была ее реализация еще на Cfront в далеком 1984, но мне пришлось от нее избавиться из-за проблем совместимости с языком C. Эта проблема исчезла после того, как C++98 и C99 решили избавиться от использования **int**, в качестве неявного типа; т.е. оба языка теперь требуют, чтобы в объявлении каждой переменной или функции использовался явный тип. Старое значение ключевого слова auto (“это локальная переменная”) теперь недопустимо. Некоторые члены комитета просмотрели миллионы строк кода в поисках корректного использования, но большая часть касалась тестов или это явно были баги.

Поскольку эта возможность предназначена, прежде всего для упрощения кодирования, **auto** никак не влияет на спецификацию стандартной библиотеки.

См. также:

- Черновик C++, раздел 7.1.6.2, 7.1.6.4, 8.3.5 (для типов возвращаемых значений
- [N1984=06-0054] Jaakko Jarvi, Bjarne Stroustrup, and Gabriel Dos Reis: Deducing the type of variable from its initializer expression (revision 4).

Range-for оператор

Range-for оператор позволяет итерировать по «диапазону» (range), что позволяет «пройтись» по любой STL последовательности, заданной методами begin() и end(). Все стандартные контейнеры могут быть использованы в качестве «диапазона», в том числе std::string, список инициализаторов,

массив и любой другой класс, для которого вы определите методы begin() и end(), например, istream. Например:

```
        void f(vector<double>& v)
        {
                for (auto x : v) cout << x << '\n';
                for (auto& x : v) ++x; // использование ссылки дает возможность изменять значение
        }
```

Вы можете прочитать этот код таким образом: «для всех **x** в **v**», перебрать все элементы, начиная с **v.begin()** и заканчивая **v.end()**. Вот еще один пример:

```
        for (const auto x : { 1,2,3,5,8,13,21,34 }) cout << x << '\n';
```

Методы **begin()** и **end()** могут быть функциями членами и вызываться в форме **x.begin()** или свободными функциями, вызываемыми в форме **begin(x)**. При этом приоритет функций-членов – выше.

См. также:

- the C++ draft section 6.5.4
- [N2243==07-0103] Thorsten Ottosen: Wording for range-based for-loop (revision 2).
- [N3257=11-0027] Jonathan Wakely and Bjarne Stroustrup: Range-based for statements and ADL (Был выбран 5-й вариант).

Правые угловые скобки

Давайте рассмотрим следующий код:

```
        list<vector<string>> lvs;
```

С точки зрения C++98 приведенный код некорректен, поскольку между двумя закрывающими угловыми скобами (>) нет пробела. C++11 рассматривает подобную комбинацию угловых скобок как корректный терминатор списка из двух аргументов шаблона.

А почему это вообще было проблемой? Компилятор состоит из нескольких этапов анализа. Вот самая простая модель:

- Лексический анализ (создание лексем из символов).
- Синтаксический анализ (проверка грамматики).
- Проверка типов (поиск имен типов и выражений).

В теории, а иногда и на практике, эти этапы четко разделены, так что лексический анализатор, определяющий, что “>>” является маркером (который обычно означает правый сдвиг или ввод), понятия не имеет о его значении; в частности, он не имеет ни малейшего понятия, ни о шаблонах, ни о вложенном списке аргументов шаблона. Однако чтобы сделать этот пример «корректным», эти три этапа должны как-то взаимодействовать. Ключевое наблюдение, которое привело к решению этой проблемы заключалось в том, что компиляторы C++ выдавали подходящее сообщение об ошибке, а значит уже умели выполнять весь необходимый анализ.

См. также:

- Раздел стандарта ???
- [N1757==05-0017] Daveed Vandevoorde: revised right angle brackets proposal (revision 2).

Управление поведением по умолчанию: default и delete

Сейчас стандартная идиома «запрещения копирования» может быть явно выражена следующим образом:

```
class X {
        // ...

        X& operator=(const X&) = delete;       // Запрет копирования
        X(const X&) = delete;
};
```

И наоборот, мы можем явно сказать о том, что хотим использовать поведение копирования по умолчанию:

```
class Y {
        // ...
        Y& operator=(const Y&) = default;      // Семантика копирования по умолчанию
        Y(const Y&) = default;
        }
};
```

Явная спецификация поведения по умолчанию является избыточной. Однако являются редкостью комментарии для операций копирования и (что еще хуже) явно определенные пользователем операции копирования, которые должны вести себя аналогично операциям по умолчанию. Реализация поведения по умолчанию этих операций компилятором обычно проще, менее подвержено ошибкам и часто приводит к лучшему объектному коду.

Ключевое слово “default” может использоваться с любой функцией для которой компилятор может реализовать поведение по умолчанию. Ключевое слово “delete” может быть использовано с любой функцией. Например, так вы можете запретить нежелательное преобразование типов:

```
struct Z {
        // ...

        // может быть инициализирован с параметром типа long long
        Z(long long);
        // но ни с чем другим
        Z(long) = delete;
};
```

См. также:

- Раздел черновика стандарта C++???
- [N1717==04-0157] (Francis Glassborow and Lois Goldthwaite: explicit class and default definitions (an early proposal).
- Bjarne Stroustrup: Control of class defaults (a dead end).
- [N2326==07-0186] Lawrence Crowl: Defaulted and Deleted Functions.
- [N3174=100164] B. Stroustrup: To move or not to move. An analysis of problems related to generated copy and move operations. Approved.

Управление поведением по умолчанию: копирование и перемещение

По умолчанию класс содержит 5 операций:

- Оператор копирования
- Конструктор копирования
- Оператор перемещения (move assignment)
- Конструктор перемещения
- Деструктор

При объявлении одной из этих операций вы должны проанализировать их все, и определить явно (или указать компилятору использовать семантику по умолчанию) только те, которые вам действительно необходимы. Рассматривайте копирование, перемещение и уничтожение, как тесно связанные, а не отдельные операции, которые можно спокойно комбинировать друг с другом; вы можете использовать любые комбинации этих операций, но только некоторые из них будут иметь смысл семантически.

Если пользователь явно определил любую из операций: деструктор, операторы перемещения или копирования (объявил, определил, любо воспользовался ключевыми словами delete или default), то по умолчанию, операции перемещения сгенерированы не будут. Если же пользователь явно определил любую из операций: операторы перемещения, копирования или деструктор (объявил, определил, любо воспользовался ключевыми словами delete или default), то неоговоренные операции копирования будут определены автоматически с поведением по умолчанию. Однако такое поведение является устаревшим (deprecated), так что рассчитывать на него не стоит. Например:

```
class X1 {
        X1& operator=(const X1&) = delete;     // запрещаем копирование
};
```

Это также запрещает перемещение (moving) класса экземпляров **X1**. Конструктор копирования разрешен, но является устаревшим (deprecated).

```
class X2 {
        X2& operator=(const X2&) = default;
};
```

Это объявление также явно запрещает перемещение экземпляров класса **X2**. Конструктор копирования разрешен, но является устаревшим (deprecated).

```
class X3 {
        X3& operator=(X3&&) = delete;  // Запрещаем перемещение
}
```

Это объявление также запрещает копирование экземпляров класса **X3**.

```
class X4 {
        ~X4() = delete;// Запрещаем деструктор
}
```

Это объявление также запрещает перемещение объектов класса **X4**. Копирование разрешено, но является устаревшим.

Я очень рекомендую при определении одной из пяти этих функций, явно определять и все остальные. Например:

```
template<class T>
class Handle {
        T* p;
public:
        Handle(T* pp) : p{pp} {}

        // пользовательский деструктор: запрещается неявное копирование и перемещение
        ~Handle() { delete p; }

        // Передача владения
        Handle(Handle&& h) :p{h.p} { h.p=nullptr; };

        // Передача владения
        Handle& operator=(Handle&& h) { delete p; p=h.p; h.p=nullptr; }

        // Копирование запрещено
        Handle(const Handle&) = delete;
        Handle& operator=(const Handle&) = delete;

        // ...
};
```

См. также:

- the C++ draft section ???
- [N2326==07-0186] Lawrence Crowl: Defaulted and Deleted Functions.
- [N3174=100164] B. Stroustrup: To move or not to move. An analysis of problems related to generated copy and move operations. Approved.

---

enum class – строготипизированные перечисления с дополнительной областью видимости

enum class («новые перечисления» или «строгие перечисления» решает три проблемы обычных перечислений языка C++:

- Стандартные перечисления (**enums**) могут неявно преобразовываться к **int**, что может приводить к ошибкам, если кто-то не хочет, чтобы перечисления вели себя как целые числа.
- Стандартные перечисления экспортируют свои значения в окружающую (surrounding) область видимости (scope), что приводит к коллизиям имен.
- Невозможно указать тип, лежащий в основе стандартных перечислений (underlying type), что приводит к непониманию, проблемам совместимости и делает предварительное объявление (forward declaration) невозможным.

enum class («строгие перечисления) являются строготипизированными и с дополнительной областью видимости (scoped):

```
enum Alert { green, yellow, election, red }; // обычное перечисление

// строготипизированное перечисление с дополнительной областью видимости
// имена перечисления не экспортируются в окружающую область видимости
// отсутствует неявное преобразования имен перечисления к int
enum class Color { red, blue };

enum class TrafficLight { red, yellow, green };
```

```
        Alert a = 7;              // ошибка (как обычно C++)
        Color c = 7;              // ошибка: нет преобразования int->Color

        int a2 = red;             // OK: преобразование Alert->int
        int a3 = Alert::red;      // ошибка в C++98; OK в C++11
        int a4 = blue;            // ошибка: blue не объявлено в текущей области
видимости
        int a5 = Color::blue;     // ошибка: нет преобразования Color->int

        Color a6 = Color::blue;   // OK
```

Как показано выше, стандартные **перечисления** работают как и раньше, помимо этого появилась возможность квалифицировать имя элемента перечисления именем самого перечисления (int a3 = Alert::red; в примере выше).

Новые перечисления являются «классом перечисления» (enum class), поскольку они объединяют поведение обыкновенных перечислений (являются именованными значениями) с некоторыми возможностями классов (наличие области видимости и отсутствие преобразований).

Возможность явного указания типа, лежащего в основе перечисления, упрощает взаимодействие между различными платформами и гарантирует размер перечислений:

```
        enum class Color : char { red, blue };// компактное представление

        // типом по умолчанию является int
        enum class TrafficLight { red, yellow, green };

        // А какой размер E?
        // (вычисляется согласно старым правилам;
        // т.е. зависит от реализации "implementation defined")
        enum E { E1 = 1, E2 = 2, Ebig = 0xFFFFFFF0U };

        // теперь мы можем задать размер явно
        enum EE : unsigned long { EE1 = 1, EE2 = 2, EEbig = 0xFFFFFFF0U };
```

Это также позволяет предварительное объявление (forward declaration) перечислений:

```
        enum class Color_code : char;     // (предварительное) объявление
        void foobar(Color_code* p);       // использование этого объявления
        // ...
        enum class Color_code : char { red, yellow, green, blue }; //
определение
```

Типом, лежащим в основе перечисления должен быть знаковый или беззнаковый целочисленный тип; по умолчанию, это **int**.

В стандартной библиотеке классы перечисления используются:

- Для отражения системных кодов ошибок: **enum class errc;**
- Для определения безопасности указателей (pointer safety): **enum class pointer_safety { relaxed, preferred, strict };**
- Для ошибок потоков ввода-вывода: **enum class io_errc { stream = 1 };**
- Для обработки ошибок при асинхронном взаимодействии: **enum class future_errc { broken_promise, future_already_retrieved, promise_already_satisfied };**

Для некоторых из этих типов объявлены операторы, например оператор ==.

См. также:

- the C++ draft section 7.2
- [N1513=03-0096] David E. Miller: Improving Enumeration Types (original enum proposal).
- [N2347 = J16/07-0207] David E. Miller, Herb Sutter, and Bjarne Stroustrup: Strongly Typed Enums (revision 3).
- [N2499=08-0009] Alberto Ganesh Barbati: Forward declaration of enumerations.

---

constexpr – обобщенные гарантировано константные выражения

Механизм **constexpr**

- Предоставляет более обобщенный механизм константных выражений
- Позволяет определять константные выражения, используя типы, определенные пользователем
- Гарантирует инициализацию выражений во время компиляции

Давайте рассмотрим следующий пример:

```
        enum Flags { good=0, fail=1, bad=2, eof=4 };

        constexpr int operator|(Flags f1, Flags f2) { return
Flags(int(f1)|int(f2)); }

        void f(Flags x)
        {
                switch (x) {
                case bad:          /* ... */ break;
                case eof:          /* ... */ break;
                case bad|eof:      /* ... */ break;
                default:           /* ... */ break;
                }
        }
```

В данном случае **constexpr** говорит, что функция должна быть настолько простой, чтобы она могла быть вычислена во время компиляции с указанными константными аргументами.

Кроме того, для возможности вычисления выражения во время компиляции, мы хотим иметь возможность *требовать* вычисления выражений во время компиляции; для этого служит ключевое слово **constexpr** перед определением переменной (что неявно подразумевает константность этой переменной).

```
        constexpr int x1 = bad|eof;    // OK

        void f(Flags f3)
        {
                // ошибка: невозможно вычислить выражение во время компиляции
                constexpr int x2 = bad|f3;

                int x3 = bad|f3;               // OK
        }
```

Обычно, мы хотим получить гарантированное вычисление на этапе компиляции: глобальных объектов, объектов уровня пространства имен, и зачастую мы хотим, чтобы они располагались в хранилищах только для чтения.

Этот механизм также работает для объектов с достаточно простыми конструкторами и для выражений с этими объектами.

```
struct Point {
        int x,y;
        constexpr Point(int xx, int yy) : x(xx), y(yy) { }
};

constexpr Point origo(0,0);
constexpr int z = origo.x;

constexpr Point a[] = {Point(0,0), Point(1,1), Point(2,2) };
constexpr int x = a[1].x;      // x равняется 1
```

Обратите, пожалуйста, внимание, что **constexpr** не является механизмом общего назначения для замены ключевого слова **const** (и наоборот):

- Основное назначение ключевого слова **const** заключается в выражении того, что объект не может быть изменен через его интерфейс (хотя объект может быть спокойно изменен через другие интерфейсы). Просто так уж получается, что объявление объекта константным дает отличные возможности компилятору для оптимизации. В частности, если объект объявлен с ключевым словом **const** и его адрес не вычисляется, компилятор обычно может выполнить его инициализацию во время компиляции (хотя это и не гарантируется) и хранить этот объект в собственных таблицах вместо помещения его в сгенерированный код.
- Основное назначение ключевого слова **constexpr** заключается в расширении диапазона того, что может быть вычислено во время компиляции, делая эти вычисления типобезопасными. Инициализаторы объектов, объявленные с ключевым словом **constexpr**, выполняются на этапе компиляции; по сути - это значения, которые хранятся в таблицах компилятора и они попадают в сгенерированный код только при необходимости.

См. также:

- the C++ draft 3.6.2 Initialization of non-local objects, 3.9 Types [12], 5.19 Constant expressions, 7.1.5 The constexpr specifier
- [N1521=03-0104] Gabriel Dos Reis: Generalized Constant Expressions (original proposal).
- [N2235=07-0095] Gabriel Dos Reis, Bjarne Stroustrup, and Jens Maurer: Generalized Constant Expressions -- Revision 5.

---

decltype – тип выражения

decltype(E) – это тип («объявленный тип», declared type) имени или выражения **E**, который может быть использован в объявлениях. Например:

```
void f(const vector<int>& a, vector<float>& b)
{
        typedef decltype(a[0]*b[0]) Tmp;
        for (int i=0; i<b.size(); ++i) {
                Tmp* p = new Tmp(a[i]*b[i]);
                // ...
        }
        // ...
}
```

Эта идея, под названием “typeof” была уже давно известна в среде обобщенного (generic) программирования, но реальные реализации **typeof** всегда были неполными и несовместимыми, поэтому в текущем стандарте было решено использовать новое понятие: **decltype**.

Если вам просто нужен тип переменной, которую вы хотите проинициализировать, то более простым решением является использование ключевого слова auto. Использование **decltype** может понадобиться для чего-то другого помимо типа переменных, например, для типов возвращаемого значения.

См. также:

- the C++ draft 7.1.6.2 Simple type specifiers
- [Str02] Bjarne Stroustrup. Draft proposal for "typeof". C++ reflector message c++std-ext-5364, October 2002. (original suggestion).
- [N1478=03-0061] Jaakko Jarvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek: Decltype and auto (original proposal).
- [N2343=07-0203] Jaakko Jarvi, Bjarne Stroustrup, and Gabriel Dos Reis: Decltype (revision 7): proposed wording.

Списки инициализации

Давайте рассмотрим следующий пример:

```
        vector<double> v = { 1, 2, 3.456, 99.99 };
        list<pair<string,string>> languages = {
                {"Nygaard","Simula"}, {"Richards","BCPL"}, {"Ritchie","C"}
        };
        map<vector<string>,vector<int>> years = {
                { {"Maurice","Vincent", "Wilkes"},{1913, 1945, 1951, 1967, 2000}
},
                { {"Martin", "Ritchards"} {1982, 2003, 2007} },
                { {"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004} }
        };
```

Теперь списки инициализации могут использоваться не только для массивов. Механизмом доступа к {}-списку является функция (в большинстве случаев конструктор), принимающая в качестве аргумента **std::initializer_list<T>**. Например:

```
        void f(initializer_list<int>);
        f({1,2});
        f({23,345,4567,56789});
        f({});  // пустой список
        f{1,2}; // ошибка: пропущен вызов метода ( )

        years.insert({{"Bjarne","Stroustrup"},{1950, 1975, 1985}});
```

Список инициализации может быть произвольной длины, но должен быть однородным (все элементы должны быть такого же типа, что указан в качестве параметра шаблона (**std::initializer_list<T>)**, или же должны преобразовываться к **T**).

Контейнер может реализовывать конструктор, принимающий список инициализации следующим образом:

```
        template<class E> class vector {
        public:
                // конструктор, принимающий список инициализации
```

```
                vector (std::initializer_list<E> s)
                {
                        // выделить нужное количество памяти
                        reserve(s.size());

                        //     проинициализировать     элементы     в     диапазоне
elem[0:s.size()))
                        uninitialized_copy(s.begin(), s.end(), elem);

                        sz = s.size(); // установить размер вектора
                }

                // ... как и было ...
        };
```

Различия между непосредственной инициализацией и инициализацией копирования сохраняется и в случае использования списков инициализации, но в этом случае разница менее существенна. Например, у **std::vector** есть явный (**explicit**) конструктор, принимающий **int** и принимающий список инициализации:

```
        // OK: v1 содержит 7 элементов
        vector<double> v1(7);
        // ошибка: отсутствует преобразование из int к vector
        v1 = 9;
        // ошибка: отсутствует преобразования из int к vector
        vector<double> v2 = 9;

        void f(const vector<double>&);
        // ошибка: отсутствует преобразование из int к vector
        f(9);

        // OK: v1 содержит 1 элемент (со значением 7)
        vector<double> v1{7};
        // OK v1 теперь содержит 1 элемент (со значением 9)
        v1 = {9};
        // OK: v2 содержит 1 элемент (со значением 9)
        vector<double> v2 = {9};
        // OK: f вызывается со списком { 9 }
        f({9});

        vector<vector<double>> vs = {
                // OK: вызов явного конструктора (10 элементов)
                vector<double>(10),
                // OK: вызов явного конструктора (1 элемент со значением 10)
                vector<double>{10},
                // ошибка: конструктор класса vector явный (explicit)
                10
        };
```

Функция может использовать **initializer_list** как неизменяемую последовательность. Например:

```
        void f(initializer_list<int> args)
        {
                for (auto p=args.begin(); p!=args.end(); ++p) cout << *p << "\n";
        }
```

Конструктор, принимающий единственный аргумент типа **std::initializer_list** называется конструктором со списком инициализации (initializer-list constructor).

Контейнеры стандартной библиотеки, такие как **string** и **regex** содержат конструкторы, операторы присваивания и т.д., принимающие списки инициализации. Списки инициализации могут использоваться в качестве диапазонов значений, например, в range-for операторе.

См. также:

- the C++ draft 8.5.4 List-initialization [dcl.init.list]
- [N1890=05-0150] Bjarne Stroustrup and Gabriel Dos Reis: Initialization and initializers (an overview of initialization-related problems with suggested solutions).
- [N1919=05-0179] Bjarne Stroustrup and Gabriel Dos Reis: Initializer lists.
- [N2215=07-0075] Bjarne Stroustrup and Gabriel Dos Reis : Initializer lists (Rev. 3) .
- [N2640=08-0150] Jason Merrill and Daveed Vandevoorde: Initializer Lists -- Alternative Mechanism and Rationale (v. 2) (final proposal).

Предотвращение сужения (narrowing) типов

Проблема заключается в том, что языки C и C++ неявно обрезают некоторые типы:

```
int x = 7.3;            // Ой!
void f(int);
f(7.3);                 // Ой!
```

Однако списки инициализации C++11 не позволяют сужение (narrowing) типов:

```
int x0 {7.3};   // ошибка: сужение
int x1 = {7.3};// ошибка: сужение
double d = 7;
int x2{d};              // ошибка: сужение (double к int)
char x3{7};             // ОК: хотя 7 – это int, здесь нет сужения
vector<int> vi = { 1, 2.3, 4, 5.6 };  // ошибка: сужение double к int
```

Чтобы избежать большого количества несовместимостей, в C++11 по возможности используется не только анализ типов, но и анализ реальных значений инициализаторов (как в примере с **char x3 { 7 }**). Сужение не происходит, когда значение может быть в точности представлено целевым типом.

```
char c1{7};       // ОК: 7 – это int, но он умещается в char
char c2{77777};   // ошибка: сужение
```

Обратите внимание, что преобразование из чисел с плавающей запятой к целочисленным типам всегда рассматривается как сужение, даже 7.0 к 7.

См. также:

- the C++ draft section 8.5.4.
- [N1890=05-0150] Bjarne Stroustrup and Gabriel Dos Reis: Initialization and initializers (an overview of initialization-related problems with suggested solutions).
- [N2215=07-0075] Bjarne Stroustrup and Gabriel Dos Reis : Initializer lists (Rev. 3) .
- [N2640=08-0150] Jason Merrill and Daveed Vandevoorde: Initializer Lists - Alternative Mechanism and Rationale (v. 2) (primarily on "explicit").

Делегирующие конструкторы

В C++98, если два конструктора выполняли одно и то же, то приходилось либо дублировать этот код, либо добавлять функцию инициализации “init”. Например:

```
        class X {
                int a;
                void validate(int x) { if (0<x && x<=max) a=x; else throw
bad_X(x); }
        public:
                X(int x) { validate(x); }
                X() { validate(42); }
                X(string s) { int x = lexical_cast<int>(s); validate(x); }
                // ...
        };
```

Такая многословность ухудшает читабельность, а дублирование может приводить к ошибкам. И все это негативно сказывается на сопровождаемости. Поэтому в C++11 мы можем определить конструктор, в терминах другого конструктора:

```
        class X {
                int a;
        public:
                X(int x) { if (0<x && x<=max) a=x; else throw bad_X(x); }
                X() :X{42} { }
                X(string s) :X{lexical_cast<int>(s)} { }
                // ...
        };
```

См. также:

- the C++ draft section 12.6.2
- N1986==06-0056 Herb Sutter and Francis Glassborow: Delegating Constructors (revision 3).

Инициализация членов класса при объявлении

В C++98 только статические константные члены встроенных типов могли инициализироваться при объявлении. Эти ограничения гарантировали, что инициализация могла быть произведена во время компиляции. Например:

```
        int var = 7;

        class X {
                // OK
                static const int m1 = 7;
                // ошибка: поле не статическое
                const int m2 = 7;
                // ошибка: поле не константное
                static int m3 = 7;
                // ошибка: инициализация не константным выражением
                static const int m4 = var;
                // ошибка: не встроенный тип
                static const string m5 = "odd";
                // ...
        };
```

Идея в C++11 заключалась в том, чтобы позволить инициализировать нестатические члены данных в месте объявления (в классе). В случае необходимости, конструктор может использовать инициализатор для инициализации полей во время выполнения. Например, следующий код:

```
class A {
public:
        int a = 7;
};
```

Эквивалентен:

```
class A {
public:
        int a;
        A() : a(7) {}
};
```

Это немного экономит время набора, но главное преимущество мы получаем, когда класс содержит несколько конструкторов. Обычно все конструкторы используют одинаковый список инициализации:

```
class A {
public:
        A(): a(7), b(5), hash_algorithm("MD5"), s("Constructor run") {}
        A(int a_val) : a(a_val), b(5), hash_algorithm("MD5"),
s("Constructor run") {}
        A(D d) : a(7), b(g(d)), hash_algorithm("MD5"), s("Constructor
run") {}
        int a, b;
private:
        // Криптографическая хэш-функция, которая будет применяься
        // ко всем экземплярам класса A
        HashingFunction hash_algorithm;
        // Строка, которая представляет состояние жизненного цикла
объекта
        std::string s;
};
```

Тот факт, что **hash_algorithm** и **s** содержат одинаковые значения по умолчанию, теряются за грудой кода и легко могут привести к проблемам сопровождения. Вместо этого, мы можем разделить процесс инициализации членов:

```
class A {
public:
        A(): a(7), b(5) {}
        A(int a_val) : a(a_val), b(5) {}
        A(D d) : a(7), b(g(d)) {}
        int a, b;
private:
        // Криптографическая хэш-функция, которая будет применяься
        // ко всем экземплярам класса A
        HashingFunction hash_algorithm{"MD5"};
        // Строка, которая представляет состояние жизненного цикла
объекта
        std::string s{"Constructor run"};
};
```

Если член инициализируется в месте объявления и в списке инициализации, то выполняется только список инициализации (значения по умолчанию «затираются»). Так что предыдущий код мы можем упростить до такого:

```
        class A {
        public:
                A() {}
                A(int a_val) : a(a_val) {}
                A(D d) : b(g(d)) {}
                int a = 7;
                int b = 5;
        private:
                // Криптографическая хэш-функция, которая будет применяться
                // ко всем экземплярам класса A
                HashingFunction hash_algorithm{"MD5"};
                //  Строка,  которая  представляет  состояние  жизненного  цикла
объекта
                std::string s{"Constructor run"};
    };
```

См. также:

- the C++ draft section "one or two words all over the place"; see proposal.
- [N2628=08-0138] Michael Spertus and Bill Seymour: Non-static data member initializers.

Унаследованные конструкторы

Некоторых иногда удивляют правила областей видимости членов класса. В частности то, что члены базового класса не находится в той же области видимости, что и члены наследника:

```
        struct B {
                void f(double);
        };

        struct D : B {
                void f(int);
        };

        B b;   b.f(4.5);       // Все нормально
        D d;   d.f(4.5);       // Сюрприз: вызываем f(int) с аргументом 4
```

В C++98 мы можем «поднять» набор перегруженных функций из базового класса в наследник:

```
        struct B {
                void f(double);
        };

        struct D : B {
                // Добавляем все функции f() из области видимости B
                // в текущую область видимости
                using B::f;
                // Добавляем новую функцию с именем f()
                void f(int);
        };

        B b;   b.f(4.5);       // Все нормально
        D d;   d.f(4.5);       // Все нормально: вызываем D::f(double),
                               // которая является B::f(double)
```

Я бы сказал, что лишь историческая случайность не позволяет этой возможности работать с конструкторами так же, как с обычными функциями. C++11 предоставляет такую возможность:

```
        class Derived : public Base {
        public:
                // Поднимаем функцию f класса Base в область видимости
                // класса Derived -- работает в C++98
                using Base::f;
                // Добавляем новую функцию f
                void f(char);
                // «Предпочитаем» использовать эту функцию f вместо Base::f(int)
                void f(int);

                // Поднимаем конструктор Base в область видимости
                // класса Derived -- работает только в C++11
                using Base::Base;
                Derived(char);    // Добавляем новый конструктор
                Derived(int);           // Используем этот конструктор вместо
Base::Base(int)
                // ...
        };
```

При наследовании конструктора, вы все еще можете отстрелить себе ногу, если класс наследник содержит дополнительные члены, требующие инициализации:

```
        struct B1 {
                B1(int) { }
        };

        struct D1 : B1 {
                using B1::B1; // Неявно объявляет D1(int)
                int x;
        };

        void test()
        {
                D1 d(6);        // Ой: d.x не инициализирован
                D1 e;           //  Ошибка:  D1  не  содержит  конструктор  по
умолчанию
```

Вы можете защитить свои ноги путем использования инициализаторов членов (member-initializer):

```
        struct D1 : B1 {
                using B1::B1;   // Неявно объявляет D1(int)
                int x{0};       // поле x проинициализировано
        };

        void test()
        {
                D1 d(6);        // d.x равно 0
        }
```

См. также:

- the C++ draft section 12.9.
- [N1890=05-0150] Bjarne Stroustrup and Gabriel Dos Reis: Initialization and initializers (an overview of initialization-related problems with suggested solutions).
- [N1898=05-0158] Michel Michaud and Michael Wong: Forwarding and inherited constructors .

- [N2512=08-0022] Alisdair Meredith, Michael Wong, Jens Maurer: Inheriting Constructors (revision 4).

Статические утверждения (утверждения времени компиляции) – static_assert

Статические утверждения (утверждения времени компиляции) содержат константное выражение и строковый литерал:

```
static_assert(expression,string);
```

Компилятор вычисляет выражение, и если результат вычисления равен **false** (т.е. утверждение нарушено), выводит строку в качестве сообщения об ошибке. Например:

```
static_assert(sizeof(long)>=8,
        "64-bit code generation required for this library.");
struct S { X m1; Y m2; };
static_assert(sizeof(S)==sizeof(X)+sizeof(Y),
        "unexpected padding in S");
```

static_assert полезен для явных заданий некоторых утверждений о программе и ее поведении. Обратите внимание, что поскольку выражение **static_assert** вычисляется во время компиляции, то оно не может применяться для проверки утверждений, зависящих от значений времени выполнения. Например:

```
int f(int* p, int n)
{
        // Ошибка: выражение в static_assert() не является
        // константным выражением
        static_assert(p==0,"p is not null");
        // ...
}
```

(вместо этого следует проверить выражение и сгенерировать исключение в случае неудачи).

См. также:

- the C++ draft 7 [4].
- [N1381==02-0039] Robert Klarer and John Maddock: Proposal to Add Static Assertions to the Core Language.
- [N1720==04-0160] Robert Klarer, John Maddock, Beman Dawes, Howard Hinnant: Proposal to Add Static Assertions to the Core Language (Revision 3).

---

long long -- более длинное целое

Целочисленная переменная, размером, по крайней мере, 64 бита. Например:

```
long long x = 922337203685477807LL;
```

Нет, никаких **long long long**, и **long** нельзя рассматривать как **short long long**.

См. также:

- the C++ draft ???.
- [05-0071==N1811] J. Stephen Adamczyk: Adding the long long type to C++ (Revision 3).

---

nullptr -- литерал для задания нулевого указателя

nullptr – это литер, который задает нулевой указатель; это не целочисленное значение:

```
char* p = nullptr;
int* q = nullptr;
char* p2 = 0;          // 0 все еще работает и p==p2

void f(int);
void f(char*);

f(0);                  // вызов f(int)
f(nullptr);            // вызов f(char*)

void g(int);
g(nullptr);            // ошибка: nullptr не явлется типом int
int i = nullptr;       // ошибка: nullptr не является типом int
```

См. также:

- the C++ draft section ???
- [N1488==/03-0071] Herb Sutter and Bjarne Stroustrup: A name for the null pointer: nullptr .
- [N2214 = 07-0074] Herb Sutter and Bjarne Stroustrup: A name for the null pointer: nullptr (revision 4) .

Суффиксный синтаксис возвращаемого значения

Давайте рассмотрим пример:

```
template<class T, class U>
??? mul(T x, U y)
{
        return x*y;
}
```

Что мы должны записать в качестве типа возвращаемого значения? Конечно же, это тип выражения **x*y**, но как нам его указать. Первая мысль, с помощью **decltype**:

```
template<class T, class U>
decltype(x*y) mul(T x, U y) // Проблема с видимостью!
{
        return x*y;
}
```

Этот вариант не работает, поскольку **x** и **y** используются за пределами их области видимости. Однако мы можем записать так:

```
// Ужасно! И чревато ошибками
template<class T, class U>
```

```
decltype(*(T*)(0)**(U*)(0)) mul(T x, U y)
{
        return x*y;
}
```

И сказать, что этот вариант «не очень» - это ничего не сказать.

Решение же заключается в помещении типа возвращаемого значения на его место – после аргументов:

```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y)
{
        return x*y;
}
```

Мы используем ключевое слово **auto**, которое говорит, что «тип возвращаемого значения будет выведен или указан позднее».

На самом деле суффиксный синтаксис связан не столько с шаблонами и выводом типов, сколько с областью видимости.

```
struct List {
        struct Link { /* ... */ };
        // Удаляем p, возвращаем указатель на узел, предыдущий p
        Link* erase(Link* p);
        // ...
};

List::Link* List::erase(Link* p) { /* ... */ }
```

Первый префикс **List::** необходим только потому, что область типа **List** еще не начинается до второго **List::**. Вот более простой вариант:

```
auto List::erase(Link* p) -> Link* { /* ... */ }
```

Теперь ни одно из упоминаний **Link** не требует явной квалификации имен.

См. также:

- the C++ draft section ???
- [Str02] Bjarne Stroustrup. Draft proposal for "typeof". C++ reflector message c++std-ext-5364, October 2002.
- [N1478=03-0061] Jaakko Jarvi, Bjarne Stroustrup, Douglas Gregor, and Jeremy Siek: Decltype and auto.
- [N2445=07-0315] Jason Merrill: New Function Declarator Syntax Wording.
- [N2825=09-0015] Lawrence Crowl and Alisdair Meredith: Unified Function Syntax.

---

template alias (известные ранее как “template typedef”)

Как нам создать шаблон, «аналогичный другому шаблону», но, возможно, с несколькими указанными (привязанными, bound) шаблонными аргументами? Давайте рассмотрим пример:

```
template<class T>
```

```
// Стандартный вектор, использующий мой аллокатор
using Vec = std::vector<T,My_alloc<T>>;

// Элементы выделяются с помощью My_alloc
Vec<int> fib = { 1, 2, 3, 5, 8, 13 };

// verbose и fib одного типа
vector<int,My_alloc<int>> verbose = fib;
```

Ключевое слово **using** используется для получения линейной нотации: «имя, за которым следует то, на что оно ссылается». Мы попробовали использовать стандартное и довольно запутанное решение на основе **typedef**, но так и не смогли добиться полного и ясного решения, пока не пришли к менее запутанному синтаксису.

Специализация работает (вы можете создать синоним (alias) для набора специализаций, но не можете специализировать псевдонимы). Например:

```
template<int>
// Идея: int_exact_trait<N>::type тип в точности из N бит
struct int_exact_traits {
        typedef int type;
};

template<>
struct int_exact_traits<8> {
        typedef char type;
};

template<>
struct int_exact_traits<16> {
        typedef char[2] type;
};

// ...

// Создаем синоним для более удобного использования
template<int N>
using int_exact = typename int_exact_traits<N>::type;

// int_exact<8> является целочисленной переменной из 8 бит
int_exact<8> a = 7;
```

Помимо использования синонимов типов совместно с шаблонами, они могут использоваться в качестве альтернативного (и, ИМО более удачного) синтаксиса для синонимов обычных типов:

```
typedef void (*PFD)(double);   // C style
using PF = void (*)(double);   // using plus C-style type
using P = [](double)->void;    // using plus suffix return type
```

См. также:

- the C++ draft: 14.6.7 Template aliases; 7.1.3 The typedef specifier
- [N1489=03-0072] Bjarne Stroustrup and Gabriel Dos Reis: Templates aliases for C++.
- [N2258=07-0118] Gabriel Dos Reis and Bjarne Stroustrup: Templates Aliases (Revision 3) (final proposal).

Variadic Templates

Необходимо решить следующие задачи:

- Как создать класс с 1, 2, 3, 4, 5, 6, 7, 8, 9, … инициализаторами?
- Как избежать создания объекта по частям с последующим копированием результата?
- Как создать кортеж (tuple)?

Последний вопрос является ключевым: подумайте о создании кортежей! Если можно создать и использовать обобщенные кортежи, то все остальные вопросы уйдут сами собой.

Вот пример (из «Короткого введения в Variadic templates» (см. ссылки)) реализации обобщенной, строго типизированной версии функции **printf()**. Наверное, лучше использовать **boost::format**, но давайте посмотрим на пример:

```
        const string pi = "pi";
        const char* m =
                "The value of %s is about %g (unless you live in %s).\n";
        printf(m,  pi, 3.14159,  "Indiana");
```

В простейшем случае функция **printf()** не содержит никаких дополнительных аргументов помимо строки формата, поэтому вначале обрабатываем этот случай:

```
        void printf(const char* s)
        {
                while (s && *s) {
                        // Нужно удостовериться, что нет других аргументов
                        // %% представляет символ % внутри строки формата
                        if (*s=='%' && *++s!='%')

                          throw     runtime_error("invalid     format:     missing
arguments");
                        std::cout << *s++;
                }
        }
```

Этот случай готов и нам нужно разобраться с функцией **printf()** с несколькими аргументами:

```
        // Обратите внимание на "..."
        template<typename T, typename... Args>
        // Обратите внимание на "..."
        void printf(const char* s, T value, Args... args)
        {
                while (s && *s) {
                        // Формат указан (сам формат нам не важен)
                        if (*s=='%' && *++s!='%') {
                                //  используем  первый  аргумент,  не  являющийся
форматом
                                std::cout << value;
                                // "достаем" первый аргумент
                                return printf(++s, args...);
                        }
                        std::cout << *s++;
                }
                throw std::runtime error("extra arguments provided to printf");
        }
```

Этот код просто «достает» первый аргумент, не являющийся форматной строкой, и затем вызывает себя рекурсивно. Когда таких аргументов больше не останется, будет вызвана первая (более простая)

версия метода **printf**(). Это довольно стандартная техника из области функционального программирования, применяемая во время компиляции. Обратите внимание, как перегруженный оператор << заменяет использование (потенциально ошибочной) «подсказки» (“hint”) в спецификаторе формата.

Тип **Args…** определяет так называемую «группу параметров» (“parameter pack”). По сути, это последовательность пар тип/значение, из которых вы можете «доставать» аргументы, начиная с первого. При вызове функции **printf()** с одним аргументом, будет выбран первый метод (**printf(const char*)**). При вызове функции **printf()** с двумя или более аргументами, будет выбран второй метод (**printf(const char*, T value, Args… args)**), с первым параметром **s**, вторым – **value**, и оставшиеся параметры (если они есть) будут запакованы в группу параметров **args**, для последующего использования. При вызове:

```
printf(++s, args...);
```

Группа параметров **args** сдвигается на один, и следующий параметр может быть обработан в виде **value**. И так продолжается до тех пор, пока **args** не станет пустым (и будет вызвана первая версия метода **printf()**).

Если вы знакомы с функциональным программированием, то можете подумать, что это немного необычная запись для довольно стандартной техники. Если вы не знакомы с функциональным программированием, то вот несколько технических примеров, способных в этом помочь в ней разобраться. Прежде всего, мы можем объявить и использовать простую шаблонную функцию с переменным числом аргументов (аналогичную функции **printf()**, о которой шла речь выше):

```
// Шаблонная функция с переменным числом аргументов
// (т.е. функция, способная принимать произвольное количество
// аргументов произвольного типа)
template<class ... Types>
        void f(Types ... args);

f();            // OK: args не содержит аргументов
f(1);           // OK: args содержит один аргумент: int
f(2, 1.0);      // OK: args содержит 2 аргумента: int и double
```

Мы можем создать шаблонный тип с переменным числом аргументов:

```
template<typename Head, typename... Tail>
class tuple<Head, Tail...>
        : private tuple<Tail...> {
        // Используем рекурсию
        //  По  сути,  кортеж  (tuple)  содержит  голову  (первую  пару
(тип/значение)
        //  и  наследует  от  кортежа  с  хвостом  (остальные  пары
тип/значение).
        // Обратите внимание, что тип зашит в типе, а не хранится в виде
данных

        typedef tuple<Tail...> inherited;
public:
        tuple() { }     // Значение по умолчанию: пустой кортеж

        // Создаем кортеж по независимым аргументам:
        tuple(typename add_const_reference<Head>::type v,
                typename add_const_reference<Tail>::type... vtail)
                : m_head(v), inherited(vtail...) { }
```

```
                // Создаем кортеж по другому кортежу:
                template<typename... VValues>
                tuple(const tuple<VValues...>& other)
                :       m_head(other.head()), inherited(other.tail()) { }

                template<typename... VValues>
                tuple& operator=(const tuple<VValues...>& other)      //
присваивание
                {
                        m_head = other.head();
                        tail() = other.tail();
                        return *this;
                }

                typename add_reference<Head>::type head() { return m_head; }
                typename add_reference<const Head>::type head() const { return
m_head; }

                inherited& tail() { return *this; }
                const inherited& tail() const { return *this; }
        protected:
                Head m_head;
        }
```

С таким определением мы можем создавать кортежи (а также копировать и работать с ними):

```
        tuple<string,vector,double> tt("hello",{1,2,3,4},1.2);
        string h = tt.head();  // "hello"
        tuple<vector<int>,double> t2 = tt.tail();     // {{1,2,3,4},1.2};
```

Поскольку указывать все эти типы довольно утомительно, то обычно, мы можем вывести аргументы типов, например, с помощью метода стандартной библиотеки **make_tuple()**:

```
        // это несколько упрощенное определение (см. раздел стандарта 20.5.2.2)
        template<class... Types>
        tuple<Types...> make_tuple(Types&&... t)
        {
                return tuple<Types...>(t...);
        }

        string s = "Hello";
        vector<int> v = {1,22,3,4,5};
        auto x = make_tuple(s,v,1.2);
```

См. также:

- Standard 14.6.3 Variadic templates
- [N2151==07-0011] D. Gregor, J. Jarvi: Variadic Templates for the C++0x Standard Library.
- [N2080==06-0150] D. Gregor, J. Jarvi, G. Powell: Variadic Templates (Revision 3).
- [N2087==06-0157] Douglas Gregor: A Brief Introduction to Variadic Templates.
- [N2772==08-0282] L. Joly, R. Klarer: Variadic functions: Variadic templates or initializer lists? -- Revision 1.
- [N2551==08-0061] Sylvain Pion: A variadic std::min(T, ...) for the C++ Standard Library (Revision 2) .
- Anthony Williams: An introduction to Variadic Templates in C++0x. DevX.com, May 2009.

Унифицированый синтаксис и семантика инициализации

В C++ существует несколько способов инициализации объекта в зависимости от его типа и контекста инициализации. В случае неправильного использования, ошибка может быть неожиданной и непонятной. Давайте рассмотрим следующие примеры:

```
// ok: инициализация переменной массива
string a[] = { "foo", " bar" };
// ошибка: инициализатор списка для неагрегированного вектора
vector<string> v = { "foo", " bar" };
void f(string a[]);
// ошибка синтаксиса: блок в качестве аргумента
f( { "foo", " bar" } );
```

и

```
// "стиль присваивания"
int a = 2;
// стиль присваивания списка
int[] aa = { 2, 3 };
// инициализация "в стиле вызова функции"
complex z(1,2);
// инициализация "в стиле вызова функции" для
// конвертации/преобразования/конструирования
x = Ptr(y);
```

и

```
int a(1);       // определение переменной
int b();        // определение функции
int b(foo);     // определение переменной или объявление функции
```

Может быть сложно запомнить все правила инициализации и выбрать правильный.

В C++11 эта проблема решается с помощью списка инициализации **{}**:

```
X x1 = X{1,2};
X x2 = {1,2};  // знак равно (=) необязателен
X x3{1,2};
X* p = new X{1,2};

struct D : X {
        D(int x, int y) :X{x,y} { /* ... */ };
};

struct S {
        int a[3];
        S(int x, int y, int z) :a{x,y,z}
        { /* ... */ }; // решение старой проблемы
};
```

Очень важно, что **X{a}** создает одно и то же значение не зависимо от контекста, так что **{}**-инициализация приводит к одному и тому же результату везде, где она применима. Например:

```
X* p = new X{a};
z = X{a};          // используется преобразование типов
f({a});            // аргумент функции (типа X)
return {a};         // возвращаемое значение функции (функция возвращает
X)
```

См. также:

- the C++ draft section ???
- [N2215==07-0075] Bjarne Stroustrup and Gabriel Dos Reis: Initializer lists (Rev. 3) .
- [N2640==08-0150] Jason Merrill and Daveed Vandevoorde: Initializer Lists -- Alternative Mechanism and Rationale (v. 2) (final proposal).

Rvalue ссылки

Разница между lvalue (которые могут использоваться слева от оператора присваивания) и rvalue значениями (которые могут использоваться справа от оператора присваивания) ведут свое начало от Кристофера Страчи (Christopher Strachey) («папы» дальнего родственника C++ под названием CPL и его денотационных семантик). В языке C++ неконстантная ссылка может быть связана с lvalue, константная ссылка – с lvalue или rvalue, но не существует ничего, что может быть связано с неконстантным rvalue значением. Это сделано для защиты от изменения значений временных объектов, которые будут уничтожены до того, как новым значением можно будет воспользоваться. Например:

```
void incr(int& a) { ++a; }
int i = 0;
incr(i);        // i буде равняться 1
incr(0);        // ошибка: 0 не является lvalue
```

Если бы вызов **incr(0)** был бы разрешен, то тогда либо будет увеличено значение временной переменной, которое никто не сможет увидеть, либо, что еще хуже, 0 станет равен 1. Последний вариант звучит глупо, но подобный баг был в ранних версиях компилятора Fortran, который позволял втихую изменить ячейку памяти, хранившую 0.

Пока все хорошо, но давайте рассмотрим следующий пример:

```
// "старый добрый обмен значениями"
template<class T> swap(T& a, T& b)
{
        T tmp(a);       // теперь у нас есть две копии a
        a = b;          // теперь у нас есть две копии b
        b = tmp;        // теперь у нас есть две копии tmp (aka a)
}
```

Если для типа **T** копирование элементов является дорогой операцией, как например, для типов **string** или **vector**, операция **swap** также становится достаточно дорогой операцией (поэтому в стандартной библиотеке у нас есть специализированные версии методов **swap** для строки и вектора). Обратите внимание на интересный момент: мы вообще не хотим делать никаких копий. Мы просто хотим поменять значения **a**, **b** и **tmp**.

Для перемещения, а не копирования аргументов в C++11 мы можем определить «конструкторы перемещения» (move constructors) и «операторы перемещения» (move assignments):

```
template<class T> class vector {
        // ...
        vector(const vector&);                  // конструктор копирования
        vector(vector&&);                       // конструктор перемещения
```

```
        vector& operator=(const vector&);      // обычное присваивание
        vector& operator=(vector&&);           // оператор перемещения
};      // обратите внимание: конструктор и оператор перемещения
        // принимают неконстантные &&
        // они могут (и обычно делают) изменяют свои аргументы
```

&& означает “rvalue-ссылку”. rvalue-ссылки могут быть связаны только с rvalue (но не с lvalue).

```
X a;
X f();
X& r1 = a;              // связывает r1 с a (lvalue)
X& r2 = f();            // ошибка: f() возвращает rvalue;

X&& rr1 = f();  // ok: связывает rr1 с временным объектом
X&& rr2 = a;    // ошибка: a – это lvalue
```

Идея семантики перемещения заключается в том, что вместо создания копии, мы можем просто взять значение из источника и заменить его дешевым значением по умолчанию. Например, для выражения **s1=s2** с поддержкой перемещения, символы строки **s2** скопированы не будут; вместо этого, строка **s1** будет рассматривать символы строки **s2**, как свои собственные и удалит свои исходные данные (возможно передаст их строке **s2**, которая вскоре будет удалена).

Откуда мы знаем, что перемещение данных из источника является безопасным? Мы говорим об этом компилятору:

```
template<class T>
void swap(T& a, T& b)  // "идеальный обмен значениями" (почти)
{
        T tmp = move(a);        // может сделать a недействительным
        a = move(b);            // может сделать b недействительным
        b = move(tmp);          // может сделать tmp недействительным
}
```

move(x) означает «вы можете рассматривать **x** в качестве rvalue». Возможно, было бы лучше, если бы **move()** назывался **rval()**, но **move()** уже используется многие годы. Шаблонная функция **move()** может быть написана на C++11 (см. "Brief introduction to rvalue references") с помощью rvalue-ссылок.

Rvalue-ссылки могут использоваться для создания идеального механизма перенаправления (forwarding).

В стандартной библиотеке C++11 во все контейнеры добавлены конструкторы и операторы перемещения. Также операции добавления новых элементов, такие как **insert()** и **push_back()** содержат версии, принимающие rvalue-ссылки. Конечным результатом является то, что производительность стандартных контейнеров и алгоритмов тихо (без вмешательства пользователя) была улучшена за счет уменьшения необходимости копирования.

См. также:

- the C++ draft section ???
- N1385 N1690 N1770 N1855 N1952
- [N2027==06-0097] Howard Hinnant, Bjarne Stroustrup, and Bronek Kozicki: A brief introduction to rvalue references
- [N1377=02-0035] Howard E. Hinnant, Peter Dimov, and Dave Abrahams: A Proposal to Add Move Semantics Support to the C++ Language (original proposal).

- [N2118=06-0188] Howard Hinnant: A Proposal to Add an Rvalue Reference to the C++ Language Proposed Wording (Revision 3) (final proposal).

Объединения

В C++98 (как и в более ранних версиях языка C++), члены с пользовательским конструктором, деструктором или оператором присваивания не могли использоваться в объединениях (union):

```
        union U {
        int m1;
        // ошибка (глупая): complex содержит конструктор
        complex<double> m2;
        // ошибка (не глупая): string обладает сложным инвариантом
        // поддерживается ctor, copy и dtor
        string m3;

};
```

В частности:

```
        // какой конструктор вызывать (и вызывать ли вообще)?
        U u;
        // присваивание члену с типом int
        u.m1 = 1;
        // беда: чтение строки
        string s = u.m3;
```

Очевидно, некорректно писать один член, а затем читать другой; но, несмотря на это, пользователи постоянно это делают (обычно по ошибке).

В C++11, ограничения, накладываемые на объединения, изменены для поддержки большего количества типов; в частности, членам объединений позволяется иметь конструкторы и деструкторы. Стандарт также добавляет ограничения, чтобы уменьшить количество ошибок в наиболее гибких вариантах использования объединений, поощряя создание размеченных объединений (discriminated unions).

Типы членов объединений ограничены следующим образом:

- Отсутствие виртуальных функций (как и было)
- Отсутствие ссылок (как и было)
- Отсутствие базовых классов (как и было)
- Если члены объединений содержат пользовательский конструктор, деструктор или операторы копирования, тогда эти специальные функции удаляются из этого объединения; т.е. они не могут использоваться для объектов объединения. Это новое ограничение.

Например:

```
        union U1 {
                int m1;
                complex<double> m2;     // ok
        };

        union U2 {
```

```
        int m1;
        string m3;      // ok
};
```

Может показаться, что это решение будет приводить к ошибкам, но благодаря новым ограничениям это не так. В частности:

```
// ok
U1 u;
// ok: присваивание члена типа complex
u.m2 = {1,2};
// ошибка: деструктор string приводит к удалению деструктора U
U2 u2;
// ошибка: конструктор копирования string приводит к удалению
// конструктора копирования U
U2 u3 = u2;
```

По сути, объединение **U2** является бесполезным, пока оно не будет включено в структуру, которая отслеживает, какой из членов объединения используется в данный момент. Т.е. для создания размеченных объединений типа:

```
// Три альтернативные реализации, использующих объединения
class Widget {
private:
        // дискриминант
        enum class Tag { point, number, text } type;

        // представление
        union {
                // point содержит конструктор
                point p;
                int i;
                // string содержит конструктор по умолчанию,
                // операции копирования и деструктор
                string s;
        };
        // ...
        // необходимо для варианта, использующего string
        widget& operator=(const widget& w)
        {
                if (type==Tag::text && w.type==Tag::text) {
                        // обычное присваивание членов типа string
                        s = w.s;
                        return *this;
                }

                if (type==Tag::text) s.~string();     //          удаление
(явное!)

                switch (type==w.type) {
                case Tag::point: p = w.p; break;      //          обычное
копирование
                case Tag::number: i = w.i; break;
                case Tag::text: new(&s)(w.s); break;  //      размещающий
оператор new
                }
                type = w.type;
                return *this;
        }
};
```

См. также:

- the C++ draft section 9.5
- [N2544=08-0054] Alan Talbot, Lois Goldthwaite, Lawrence Crowl, and Jens Maurer: Unrestricted unions (Revison 2)

POD типы

POD ("Plain Old Data") могут обрабатываться, как C-структуры, т.е. копироваться с помощью **memcpy()**, инициализироваться с помощью **memset()**, и т.д. В C++98 определение POD было основано на наборе ограничений языковых конструкций, используемых при определении структуры:

```
struct S { int a; };   // S - это POD
struct SS { int a; SS(int aa) : a(aa) { } }; // SS - это не POD
struct SSS { virtual void f(); /* ... */ };
```

В C++11 **S** и **SS** являются «типами со стандартным расположением в памяти» (standard layout type) (a.k.a. POD) поскольку **SS** не содержит никакой «магии»: конструктор никак не влияет на расположение в памяти (поэтому инициализация с помощью **memcpy()** является возможной), и только лишь с инициализацией с помощью **memset()** будут проблемы, поскольку инициализация таким образом не будет обеспечивать инвариант. Однако, **SSS** будет содержать указатель на таблицу виртуальных функций vptr, и не может рассматриваться как «простая старая структура данных». В C++11 для работы с разными техническими аспектами, даются следующие определения: POD-типов, простых копируемых типов и типов со стандартным расположением в памяти. Определение POD-типа является рекурсивным:

- Если все члены и базовые типы являются POD-типами, то текущий тип является POD-типом
- Как и ранее (детали описаны в разделе 9 [10])
    - Отсутствие виртуальных функций
    - Отсутствие виртуальных базовых классов
    - Отсутствие ссылок
    - Отсутствие нескольких спецификаторов доступа

Наиболее важный аспект POD-типов в C++11 заключается в том, что добавление или удаление конструкторов не влияет на производительность или расположение объектов в памяти.

См. также:

- the C++ draft section 3.9 and 9 [10]
- [N2294=07-0154] Beman Dawes: POD's Revisited; Resolving Core Issue 568 (Revision 4).

Сырые строковые литералы

Во многих случаях, например, при написании регулярных выражений для стандартной библиотеки regex, тот факт, что обратный слэш (\) необходимо экранировать – немного раздражает (поскольку в регулярных выражениях обратный слэш является специальным символом). Давайте посмотрим, как будет выглядеть паттерн для поиска двух слов, разделенных обратным слешем (\w\\\w):

```
// я надеюсь, что не ошибся
string s = "\\w\\\\\\w";
```

Обратите внимание, что обратный слэш в регулярном выражении представлен двумя слэшами. По сути, «сырой строковый литерал» - это строковый литерал, в котором обратный слэш является обратным слэшем, так что наш пример будет выглядеть так:

```
// я практически уверен, что не ошибся
string s = R"(\w\\\w)";
```

Исходное предложение по добавлению сырых литералов в стандарт содержало следующий убедительный пример:

```
// Не является ли ошибкой 5 обратных слэшей?
// Даже эксперты легко могут допустить ошибку.
"('(?:[^\\\\']|\\\\.)*'|\"(?:[^\\\\\"]|\\\\.)*\")|"
```

R”(…)” является более тяжеловесной записью по сравнению с «простой» записью вида **“…”,** но в любом случае без экранирующих символов нам «нужно что-то еще». А как в сырую строку добавить кавычки? Очень легко, если они не идут перед символом открывающей скобки ‘**(**’:

```
R"("quoted string")"   // строка содержит "quoted string"
```

Так как нам записать в сырой строке **)”**? К счастью эта проблема возникает не часто, но **“(…)”** является всего лишь ограничительной парой по умолчанию. Мы можем добавить ограничители до и после **(…)** в **“(…)”**. Например:

```
// строка содержит "quoted string containing the usual terminator (")"
R"***("quoted string containing the usual terminator (")")***"
```

Последовательность символов после **)** должна быть идентичной последовательности до **(**. Это позволит справиться с паттернами (практически) любой сложности.

Перед символом **R** может находиться префикс, обозначающий кодировку символов: **u8, u, U** или **L**. Например, **u8R”(fdfdfa)”** представляет литерал в кодировке UTF-8.

См. также:

- Standard 2.13.4
- [N2053=06-0123] Beman Dawes: Raw string literals. (original proposal)
- [N2442=07-0312] Lawrence Crowl and Beman Dawes: Raw and Unicode String Literals; Unified Proposal (Rev. 2). (final proposal combined with the User-defined literals proposal).
- [N3077==10-0067] Jason Merrill: Alternative approach to Raw String issues. (replacing **[** with **(**);

Пользовательские литералы

В языке C++ существуют литералы для разных встроенных типов (2.14 Literals):

```
123     // int
1.2     // double
1.2F    // float
'a'     // char
1ULL    // unsigned long long
0xD0    // unsigned int в шестнадцатеричном формате
"as"    // string
```

Однако в языке C++98 нельзя определить литералы для пользовательских типов. Это раздражает, а также нарушает принцип, что поддержка пользовательских и встроенных типов должна быть аналогичной. В частности, многие разработчики просили поддержку следующих литералов:

```
"Hi!"s                  // строка, но не "массив символов,
                        // оканчивающийся нулем"
1.2i                    // мнимое число
123.4567891234df        // decimal floating point (IBM)
101010111000101b        // двоичное число
123s                    // секунды
123.56km                // не мили! (единицы измерения)
12345678901234567890123456789012345678901234567890x     // расширенная точность
```

C++11 поддерживает «пользовательские литералы» (user –defined literals) с помощью *литеральных операторов* (literal operators), которые задают соответствие литералов с определенным суффиксом для определенного пользовательского типа. Например:

```
// литерал для определения мнимого числа
constexpr complex<double> operator "" i(long double d)
{
        return {0,d};  // complex – это тип литерала
}

// литерал для std::string
std::string operator""s (const char* p, size_t n)
{
        return string(p,n);    // требуется динамическое выделение памяти
}
```

Обратите внимание на использование **constexpr** для вычисления выражения во время компиляции. При наличии указанных операторов, мы можем написать следующее:

```
template<class T> void f(const T&);
f("Hello");     // передаем указатель на char*
f("Hello"s);    // передаем объект string (из 5 символов)
f("Hello\n"s); // передаем объект string (из 6 символов)

auto z = 2+1i; // complex(2,1)
```

Основная идея (реализации) этой возможности заключается в том, что после парсинга потенциального литерала, компилятор всегда проверяет суффикс. Механизм пользовательских литералов позволяет пользователю указать новый суффикс и что будет выполнено для литерала, расположенного до этого суффикса. Пользователь не может переопределить поведение встроенных литералов или расширить синтаксис литералов. Литеральный оператор может принимать значение в кавычках (в качестве строки) или без них.

Для использования литерала без кавычек достаточно определить оператор, принимающий единственный аргумент типа **const char***:

```
Bignum operator"" x(const char* p)
{
        return Bignum(p);
}

void f(Bignum);
```

```
f(12345678901234567890123456789012345678901234567890x);
```

В **operator”” x()** передается C-строка вида "**12345678901234567890123456789012345678901234567890**". Обратите внимание, что мы не преобразуем явно это числовое значение в строку.

Пользовательские литералы можно определить для одного из четырех типов литералов:

- целочисленный литерал: литеральный оператор принимает единственный параметр типа unsigned long long или **const char***.
- значение с плавающей точкой: литеральный оператор принимает единственный параметр типа **long double** или **const char***.
- строковый литерал определяется литеральным оператором, принимающим пару аргументов **(const char*, size_t)**.
- символьный литерал определяется литеральным оператором, принимающим единственный параметр типа **char**.

Обратите внимание, что вы не можете определить литеральный оператор для строкового литерала, принимающий только параметр типа const char* (без размера). Например:

```
// предупреждение: этот код будет работать не так, как вы ожидаете
string operator"" S(const char* p);

"one two"S;     // ошибка: литеральный оператор не найден
```

Причина такого поведения заключается в том, что практически всегда, когда мы хотим определить «еще один тип строки» нам нужно знать количество символов.

Суффиксы должны быть короткими (например, **s** для строки, **i** для комплексных чисел, **m** для метров, **x** для расширенных целочисленных типов), что легко может приводить к коллизиям. Для предотвращения коллизий следует использовать пространства имен:

```
namespace Numerics {
        // ...
        class Bignum { /* ... */ };
        namespace literals {
                operator"" X(char const*);
        }
}

using namespace Numerics::literals;
```

См. также:

- Standard 2.14.8 User-defined literals
- [N2378==07-0238] Ian McIntosh, Michael Wong, Raymond Mak, Robert Klarer, Jens Mauer, Alisdair Meredith, Bjarne Stroustrup, David Vandevoorde: User-defined Literals (aka. Extensible Literals (revision 3)).

Атрибуты

«Атрибуты» являются стандартным механизмом, который должен навести порядок в том огромном наборе возможностей и/или компиляторно-зависимых механизмов добавления специальной

информации к исходному коду (например, **__attribute__**, **__declspec** и **#pragma**). Атрибуты C++11 влияют непосредственно на сущность, стоящую перед ними: этим они отличаются от обычных синтаксических конструкций языка. Например:

```
// f() никогда не возвращает управление
void f [[ noreturn ]] ()
{
        throw "error"; // OK
}

// подсказка оптимизатору
struct foo* f [[carries_dependency]] (int i);
int* g(int* x, int* y [[carries_dependency]]);
```

Как видите, атрибуты помещаются в двойные квадратные скобки: **[[** … **]]**. **noreturn** и **carries_dependency** являются парой атрибутов, определенных в стандарте.

Существует разумное опасение, что атрибуты будут использоваться для разработки диалектов языка. Рекомендуется использовать атрибуты таким образом, чтобы они не меняли семантики программы, но могли помочь в поиске ошибок (например, **[[noreturn]]**) или помочь в оптимизации (например, **[[carries_dependency]]**).

Одна из причин появления атрибутов – это улучшение поддержки OpenMP. Например:

```
for [[omp::parallel()]] (int i=0; i<v.size(); ++i) {
        // ...
}
```

Как показано выше, атрибуты могут квалифицироваться пространством имен.

См. также:

- Standard: 7.6.1 Attribute syntax and semantics, 7.6.3-4 noreturn, carries_dependency 8 Declarators, 9 Classes, 10 Derived classes, 12.3.2 Conversion functions
- [N2418=07-027] Jens Maurer, Michael Wong: Towards support for attributes in C++ (Revision 3)

Лямбда-выражения

Лямбда-выражение – это механизм определения объекта-функции. Основная цель лямбда-выражения заключается в определении некоторого действия, выполняемого некоторой функцией. Например:

```
vector<int> v = {50, -10, 20, -30};

// сортировка по умолчанию
std::sort(v.begin(), v.end());
// теперь v должен содержать { -30, -10, 20, 50 }

// сортируем по абсолютному значению:
std::sort(v.begin(), v.end(), [](int a, int b) { return abs(a)<abs(b);
});
// теперь v должен содержать { -10, 20, -30, 50 }
```

Аргумент **[](int a, int b) {return abs(a)<abs(b); }** – это «лямбда» («лямбда-выражение» или «лямбда-функция»), определяющая операцию, которая для двух целочисленных аргументов **a** и **b** возвращает результат сравнения их абсолютных значений.

Лямбда-выражение может получить доступ к локальным переменным из области видимости, в которой оно используется. Например:

```
void f(vector<Record>& v)
{
        vector<int> indices(v.size());
        int count = 0;
        generate(indices.begin(),indices.end(),[&count](){          return
++count; });

        // сортируем индексы в порядке, определяемым полем name записей:
        std::sort(indices.begin(), indices.end(),
                [&](int a, int b) { return v[a].name<v[b].name; });
        // ...
}
```

Некоторые считают эту возможность «очень классной!»; другие рассматривают ее как источник пугающе непонятного кода. ИМО, верны обе точки зрения.

[&] представляет собой «список захвата» (“capture list”), который указывает, что локальные переменные передаются в лямбда-выражение по ссылке. Мы можем явно указать, что «захватываем» по ссылке только переменную **v**: **[&v]**, или захватываем ее по значению: **[=v]**. **[]** означает, что ничего не захватывается, **[&]** – означает захват всех переменных по ссылке, а **[=]** – захват всех переменных по значению.

Если действие не является простым или распространенным, то я рекомендую использовать именованный функциональный объект или функцию. Например, приведенный выше пример может быть переписан таким образом:

```
void f(vector<Record>& v)
{
        vector<int> indices(v.size());
        int count = 0;
        fill(indices.begin(),indices.end(),[&](){ return ++count; });

        struct Cmp_names {
                const vector<Record>& vr;
                Cmp_names(const vector<Record>& r) :vr(r) { }
                bool operator()(Record& a, Record& b) const {
                        return vr[a]<vr[b];
                }
        };

        // сортируем индексы в порядке, определяемым полем name записей:
        std::sort(indices.begin(), indices.end(), Cmp_names(v));
        // ...
}
```

Для таких небольших функций, как функция сравнения записей по имени, использование функциональных объектов может быть слишком многословным, хотя сгенерированный код для обоих случаев будет скорее всего идентичным. В C++98 такие функциональные объекты должны быть нелокальными, однако в C++11 это уже не обязательно.

Для определения лямбда-выражения, вы должны указать:

- список захвата: список переменных, которые могут использоваться внутри лямбда-выражения (помимо аргументов), (**[&]** означает «все локальные переменные передаются по ссылке», как в нашем примере с типом **Record**). Если лямбда-выражение не использует никакие захваченные переменные, то оно должно начинаться с **[]**.
- (необязательно) аргументы с указанием их типов (например, **(int a, int b)**).
- Блок кода, представляющий некоторое действие (например, **{ return v[a].name<v[b].name; }**).
- (необязательно) тип возвращаемого значения с помощью нового суффиксного стиля указания типов возвращаемого значения; но обычно мы можем вывести тип возвращаемого значения по возвращаемому выражению. Если возвращаемое значение отсутствует, то выводится тип **void**.

См. также:

- Standard 5.1.2 Lambda expressions
- [N1968=06-0038] Jeremiah Willcock, Jaakko Jarvi, Doug Gregor, Bjarne Stroustrup, and Andrew Lumsdaine: Lambda expressions and closures for C++ (original proposal with a different syntax)
- [N2550=08-0060] Jaakko Jarvi, John Freeman, and Lawrence Crowl: Lambda Expressions and Closures: Wording for Monomorphic Lambdas (Revision 4) (final proposal).
- [N2859=09-0049] Daveed Vandevoorde: New wording for C++0x Lambdas.

Локальные типы в шаблонных аргументах

В C++98 локальные и неименованные типы не могли использоваться в качестве аргументов шаблона. Это было досадным ограничением, поэтому в C++11 его убрали.

```
void f(vector<X>& v)
{
        struct Less {
                bool operator()(const X& a, const X& b) {
                        return a.v<b.v;
                }
        };

        // C++98: ошибка: Less является локальным типом
        // C++11: ok
        sort(v.begin(), v.end(), Less());
}
```

В C++ 11 существует альтернатива в виде использования лямбда-выражений:

```
void f(vector<X>& v)
{
        // C++11
        sort(v.begin(), v.end(),
              [] (const X& a, const X& b) { return a.v<b.v; });
}
```

Стоит помнить, что именованные действия могут быть очень полезными с точки зрения документации и приводят к хорошему дизайну. Кроме того, нелокальные сущности (обязательно именованные) могут быть использованы повторно.

C++11 разрешает использовать значения неименованных типов в качестве шаблонных аргументов:

```
template<typename T> void foo(T const& t){}
enum X { x };
enum { y };

int main()
{
        foo(x);         // C++98: ok; C++11: ok
        foo(y);         // C++98: ошибка; C++11: ok
        enum Z { z };
        foo(z);         // C++98: ошибка; C++11: ok
}
```

См. также:

- Standard: Еще не принят: CWG issue 757
- [N2402=07-0262] Anthony Williams: Names, Linkage, and Templates (rev 2).
- [N2657] John Spicer: Local and Unnamed Types as Template Arguments.

---

noexcept – предотвращение проброса исключений

Если функция не может генерировать исключение, или программа не рассчитывает на генерацию исключений функцией, то такая функция может быть объявлена с ключевым словом **noexcept**. Например:

```
// никогда не генерирует исключений
extern "C" double sqrt(double) noexcept;

// Мы не ожидаем нехватки памяти
vector<double> my_computation(const vector<double>& v) noexcept
{
        // может сгенерировать исключение
        vector<double> res(v.size());
        for(int i; i<v.size(); ++i) res[i] = sqrt(v[i]);
        return res;
}
```

Если функция, объявленная с ключевым словом **noexcept** все-таки сгенерирует исключение (т.е. исключение попытается покинуть функцию с **noexcept)**, то программа закрывается (путем вызова функции **terminate()**). При вызове **terminate()** мы не можем рассчитывать на согласованное состояние объектов (т.е. нет гарантии того, что деструкторы вызваны, нет гарантии раскрутки стека, и нельзя возобновить выполнение программы, как будто ничего не произошло). Это намеренное поведение, которое делает **noexcept** простым, грубым и очень эффективным механизмом (значительно более эффективным по сравнению со старым механизмом **throw()**).

Спецификатор **noexcept** может быть условным. Например, некоторый алгоритм может не генерировать исключений, если (и только если) операции шаблонных параметров, используемых в этой функции также не генерируют исключения:

```
        // do_f генерирует исключение, если f(v.at(0) может генерировать исключения
        template<class T>
        void do_f(vector<T>& v) noexcept(noexcept(f(v.at(0))));
        {
                for(int i; i<v.size(); ++i)
                        v.at(i) = f(v.at(i));
        }
```

Здесь первое ключевое слово **noexcept** используется в качестве оператора: выражение **noexcept(f(v.at(0)))** равняется true, если **f(v.at(0))** не генерирует исключений, т.е. если методы **f()** и **at()** объявлены с **noexcept**.

Оператор **noexcept()** является константным выражением и не вычисляет свои операнды.

Общая форма определения **noexcept** выглядит так: **noexcept(expression)** и «просто **noexcept**» является лишь сокращенной формой для **noexcept(true)**. Каждое объявление функции должно содержать совместимую (compatible) спецификацию **noexcept**.

Деструкторы не должны генерировать исключения; сгенерированный деструктор неявно объявляется как **noexcept** (не зависимо от его тела), если деструкторы всех членов класса объявлены как **noexcept**.

Обычно операторы перемещения (move operators) не должны генерировать исключений, поэтому старайтесь объявлять их как **noexcept**. Сгенерированные компилятором операторы копирования и перемещения неявно объявлены как **noexcept**, если все члены, для которых используются операторы копирования и перемещения, содержат **noexcept** деструкторы.

В стандартной библиотеке **noexcept** широко и систематично используется для улучшения производительности и уточнения требований.

См. также:

- Standard: 15.4 Exception specifications [except.spec].
- Standard: 5.3.7 noexcept operator [expr.unary.noexcept].
- [N3103==10-0093] D. Kohlbrenner, D. Svoboda, and A. Wesie: Security impact of noexcept. (Noexcept **must** terminate, as it does).
- [N3167==10-0157] David Svoboda: Delete operators default to noexcept.
- [N3204==10-0194] Jens Maurer: Deducing "noexcept" for destructors
- [N3050==10-0040] D. Abrahams, R. Sharoni, and D. Gregor: Allowing Move Constructors to Throw (Rev. 1).

Выравнивание (alignment)

Иногда, особенно при написании кода, работающего с сырой памятью, нам приходится для некоторого выделенного участка памяти указывать желаемое расположение (alignment). Например:

```
// массив символов, выровнен для хранения тиов double
alignas(double) unsigned char c[1024];
// выравнивание по 16 байтной границе
alignas(16) char[100];
```

Существует также оператор **alignof**, который возвращает выравнивание для указанного аргумента (аргумент должен быть типом). Например:

```
// целые числа выровнены по n-байтной границе
constexpr int n = alignof(int);
```

См. также:

- Standard: 5.3.6 Alignof [expr.alignof]
- Standard: 7.6.2 Alignment specifier [dcl.align]
- [N3093==10-0083] Lawrence Crowl: C and C++ Alignment Compatibility. Aligning the proposal to C's later proposal.
- [N1877==05-0137] Attila (Farkas) Feh&#65533;r: Adding Alignment Support to the C++ Programming Language. The original proposal.

Управление переопределением функций: override

Для переопределения функции базового класса, в классе наследнике не нужны никакие ключевые слова или аннотации. Например:

```
struct B {
        virtual void f();
        virtual void g() const;
        virtual void h(char);
        // Функция не виртуальная
        void k();
};

struct D : B {
        // переопределяет B::f()
        void f();
        // не переопределяет B::g() (тип неверен)
        void g();
        // переопределяет B::h()
        virtual void h(char);
        // не переопределяет B::k() (B::k() не виртуальная)
        void k();
};
```

Такой код может приводить к неоднозначности (что имел в виду программист?) и проблемам, если компилятор не будет предупреждать о подозрительном коде. Например:

- Хотел ли программист переопределить **B::g()**? (наверняка, да).
- Хотел ли программист переопределить **B::h(char)**? (скорее всего, нет, поскольку он указал избыточное ключевое слово **virtual**).
- Хотел ли программист переопределить **B::k()**? (может быть, но это невозможно).

Теперь у нас есть ключевое слово **override**, которое позволяет программисту явно выражать свои намерения на счет переопределения функций.

```
struct D : B {
        // OK: переопределяем B::f()
        void f() override;
        // ошибка: тип неверен
        void g() override;
        // переопределяет B::h(); скорее всего будет
        // выдано предупреждение
        virtual void h(char);
        // ошибка: B::k() не виртуальная
        void k() override;
};
```

Объявление функции с ключевым словом **override** является корректным, только если существует функция для переопределения. Проблема с методом **h** не обязательно будет отловлена (поскольку согласно определению языка, это не является ошибкой), но определить такую ситуацию легко.

См. также:

- Standard: 10 Derived classes [class.derived] [9]
- Standard: 10.3 Virtual functions [class.virtual]
- [N3234==11-0004] Ville Voutilainen: Remove explicit from class-head.
- [N3151==10-0141] Ville Voutilainen: Keywords for override control. Earlier, more elaborate design.
- [N3163==10-0153] Herb Sutter: Override Control Using Contextual Keywords. Alternative earlier more elaborate design.
- [N2852==09-0042] V. Voutilainen, A. Meredith, J. Maurer, and C. Uzdavinis: Explicit Virtual Overrides. Earlier design based on attributes.
- [N1827==05-0087] C. Uzdavinis and A. Meredith: An Explicit Override Syntax for C++. The original proposal.

Управление переопределением функций: final

Иногда программист хочет предотвратить переопределение виртуальной функции. Теперь это можно сделать с помощью ключевого слова **final**. Например:

```
struct B {
        // Функция не может быть переопределена
        virtual void f() const final;
        virtual void g();
};

struct D : B {
        // ошибка: D::f пытается переопределить B::f
        void f() const;
        // OK
        void g();
};
```

Существуют разумные причины, запрещающие переопределение функций, но, к сожалению, большая часть примеров, которые я приводил для обоснования необходимости **final**, основывались на ошибочном предположении, что вызов виртуальных функций является дорогим (в основном на основе опыта работы с другими языками программирования). Так что, если вы добавляете спецификатор **final** убедитесь в том, что причины обоснованы: будет ли семантическая ошибка в том, что кто-то захочет определить класс, переопределяющий эту виртуальную функцию? Добавление модификатора final, запрещает саму возможность того, что кто-то в будущем сможет предоставить более удачную реализацию вашей функции для класса, о котором вы даже не подозреваете. Если вы не хотите давать такую возможность, то зачем вообще объявлять эту функцию виртуальной? Большинство разумных ответов, с которыми я сталкивался, сводились к следующему: это важная функция фреймворка, она переопределяется разработчиками фреймворка, но не предназначена для переопределения простыми пользователям. Я к таким заявлениям отношусь с подозрением.

Если причина кроется в производительности (возможности inlining-а) или вы просто не хотите, чтобы ее кто-либо переопределял, то значительно проще вообще не делать эту функцию виртуальной. Это не Java.

См. также:

- Standard: 10 Derived classes [class.derived] [9]
- Standard: 10.3 Virtual functions [class.virtual]

Возможности C99

Для сохранения высокой степени совместимости, при содействии комитета по стандартизации языка C, в C++ были внесены несколько небольших изменений.

- long long.
- Extended integral types (например, правила для опциональных типов, больших чем int).
- UCN changes [N2170==07-0030] ``lift the prohibitions on control and basic source universal character names within character and string literals.''
- Конкатенация узких/широких строк.
- ***Не были внесены изменения*** VLA (Variable Length Arrays; слава небесам за это).

Были добавлены несколько расширений препроцессора:

- **__func__** - максро, который разворачивается в имя лексически текущей функции
- **__STD_C_HOSTED__**
- **_Pragrma**: **_Pragma(*X*)** разворачивается в **#pragma *X***
- макрос **vararg** (перегрузка макросов с разным числом аргументов)

```
#define report(test, ...) ((test)?puts(#test):printf(_ _VA_ARGS_ _))
```

- пустые аргументы макросов

Были добавлены возможности стандартной библиотеки из C99 (по сути, все изменения C99 по сравнению с предшественником C89).

См.:

- Standard: 16.3 Macro replacement.
- [N1568=04-0008] P.J. Plauger: PROPOSED ADDITIONS TO TR-1 TO IMPROVE COMPATIBILITY WITH C99.

Расширенные целочисленные типы

Существует набор правил, как должны вести себя целочисленные типы высокой точности, если они существуют.

См.:

- [06-0058==N1988] J. Stephen Adamczyk: Adding extended integer types to C++ (Revision 1).

Динамическая инициализация и разрушение в многопоточной среде

Простите, пока не было времени написать этот раздел.

См.:

- [N2660 = 08-0170] Lawrence Crowl: Dynamic Initialization and Destruction with Concurrency (Final proposal).

---

thread-local storage (thread_local)

Простите, пока не было времени написать этот раздел.

См.:

- [N2659 = 08-0169] Lawrence Crowl: Thread-Local Storage (Final proposal).

Юникодные символы

Простите, пока не было времени написать этот раздел.

Копирование и повторная генерация исключений

Как вы перехватываете исключение и генерируете его повторно в другом потоке? Воспользуйтесь небольшой магией, описанной в разделе 18.8.5 Exception Propagation:

- **exception_ptr current_exception();** *Возвращает: объект exception_ptr, содержащий текущее обработанное исключение (15.3) или копию текущего обработанного исключения, или возвращает null, если никакое исключение не было обработано. Ссылающийся объект должен оставаться валидным как минимум до тех пор, пока объект exception_ptr на него ссылается. ...*
- **void rethrow_exception(exception_ptr p);**
- **template<class E> exception_ptr copy_exception(E e);** По сути является следующим:

```
try {
        throw e;
} catch(...) {
        return current_exception();
}
```

Это особенно полезно при передаче исключения из одного потока в другой.

Extern templates

Спецификация шаблона может явно запретить возможность множественного инстанцирования. Например:

```
#include "MyVector.h"

// Предотвращаем неявное инстанцирование --
// MyVector<int> будет явно инстанцирован где-то в другом месте
extern template class MyVector<int>;

void foo(MyVector<int>& v)
{
        // использование вектора
}
```

«Где-то в другом месте» может быть такой код:

```
#include "MyVector.h"

// Делаем MyVector доступным клиентам (например, общей библиотеке)
template class MyVector<int>;
```

По сути, это способ избежать значительного количества ненужной работы компилятору и линковщику.

См.:

- Standard 14.7.2 Explicit instantiation
- [N1448==03-0031] Mat Marcus and Gabriel Dos Reis: Controlling Implicit Template Instantiation.

Inline namespace

Inline namespace – это механизм, обеспечивающий поддержку развития библиотек, путем предоставления механизма версионирования. Рассмотрим пример:

```
// file V99.h:
inline namespace V99 {
        void f(int);    // лучше, нежели версия V98
        void f(double);// новая возможность
        // ...
}

// file V98.h:
namespace V98 {
        void f(int);    // что-то делает
        // ...
}

// file Mine.h:
namespace Mine {
#include "V99.h"
#include "V98.h"
}
```

Суть в том, что спецификатор inline заставляет объявления во вложенном пространстве имен вести себя так, как будто они объявлены в текущем.

Это поведение очень «статичное» и ориентировано на разработчика библиотеки, поскольку спецификатор **inline** указывается поставщиком пространства имен, принимая при этом решение за пользователя. Пользователь **Mine** никак не может указать, что он хочет по умолчанию использовать **V98** вместо **V99**.

См.:

- Standard 7.3.1 Namespace definition [7]-[9].

Операторы явного преобразования

В C++98 существуют явные и неявные конструкторы; преобразования, определенные с помощью конструктора со спецификатором **explicit** могут использоваться только при явном преобразовании, в то время, как другие конструкторы могут использоваться также и в неявных преобразованиях. Например:

```
// "обычный конструктор" определяет неявное преобразование
struct S { S(int); };
S s1(1);                // ok
S s2 = 1;               // ok
void f(S);
f(1);                   // ok (но это может привести к неприятным
                        // сюрпризам, что если S – это вектор?)

struct E { explicit E(int); }; // явный конструктор
E e1(1);                // ok
E e2 = 1;               // ошибка (хотя это обычно является сюрпризом)
void f(E);
f(1);                   // ошибка (защищает от сюрпризов – например,
                        //   конструктор   std::vector,   принимающий   int
является явным)
```

Однако конструктор является не единственным механизмом определения преобразования. Если мы не можем изменить класс, мы можем определить оператор преобразования из другого класса. Например:

```
struct S { S(int) { } /* ... */ };

struct SS {
        int m;
        SS(int x) :m(x) { }
        // поскольку S не содержит конструктор S(SS)
        operator S() { return S(m); }
};

SS ss(1);
S s1 = ss;      // ok; аналогично неявному конструктору
S s2(ss);       // ok; аналогично неявному конструктору
void f(S);
f(ss);          // ok; аналогично явному конструктору
```

К сожалению, не существует **явных** (explicit) операторов преобразования (поскольку количество примеров, когда это приводит к неприятностям, значительно меньше). В C++11 этот недосмотр учтен и **явные** операторы преобразования добавлены в язык. Например:

```
struct S { S(int) { } };

struct SS {
        int m;
        SS(int x) :m(x) { }
        // поскольку S не содержит конструктор S(SS)
        explicit operator S() { return S(m); }
};

SS ss(1);
S s1 = ss;      // ошибка; как и в случае явного конструктора
S s2(ss);       // ok; как и в случае явного конструктора
void f(S);
f(ss);          // ошибка; как и в случае явного конструктора
```

См. также:

- Standard: 12.3 Conversions
- [N2333=07-0193] Lois Goldthwaite, Michael Wong, and Jens Maurer: Explicit Conversion Operator (Revision 1).

Улучшения алгоритмов

Алгоритмы стандартной библиотеки частично улучшены за счет добавления новых алгоритмов, частично за счет улучшения существующих алгоритмов, которые стали возможными благодаря новым языковым возможностям, а частично за счет того, что новые языковые возможности позволили проще их использовать:

- *Новые алгоритмы:*

```
        bool all_of(Iter first, Iter last, Pred pred);
        bool any_of(Iter first, Iter last, Pred pred);
        bool none_of(Iter first, Iter last, Pred pred);

        Iter find_if_not(Iter first, Iter last, Pred pred);

        OutIter copy_if(InIter first, InIter last, OutIter result, Pred pred);
        OutIter copy_n(InIter first, InIter::difference_type n, OutIter result);

        OutIter move(InIter first, InIter last, OutIter result);
        OutIter move_backward(InIter first, InIter last, OutIter result);

        pair<OutIter1, OutIter2> partition_copy(InIter first, InIter last,
OutIter1 out_true, OutIter2 out_false, Pred pred);
        Iter partition_point(Iter first, Iter last, Pred pred);

        RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first,
RAIter result_last);
        RAIter partial_sort_copy(InIter first, InIter last, RAIter result_first,
RAIter result_last, Compare comp);
        bool is_sorted(Iter first, Iter last);
        bool is_sorted(Iter first, Iter last, Compare comp);
        Iter is_sorted_until(Iter first, Iter last);
        Iter is_sorted_until(Iter first, Iter last, Compare comp);

        bool is_heap(Iter first, Iter last);
        bool is_heap(Iter first, Iter last, Compare comp);
        Iter is_heap_until(Iter first, Iter last);
        Iter is_heap_until(Iter first, Iter last, Compare comp);

        T min(initializer_list<T> t);
        T min(initializer_list<T> t, Compare comp);
        T max(initializer_list<T> t);
        T max(initializer_list<T> t, Compare comp);
        pair<const T&, const T&> minmax(const T& a, const T& b);
        pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
        pair<const T&, const T&> minmax(initializer_list<T> t);
        pair<const T&, const T&> minmax(initializer_list<T> t, Compare comp);
        pair<Iter, Iter> minmax_element(Iter first, Iter last);
        pair<Iter, Iter> minmax_element(Iter first, Iter last, Compare comp);

        // Для каждого элемента, на который ссылается итератор i
        // в диапазоне [first,last), присваивает *i = value и
        // увеличивает value путем вызова ++value
        void iota(Iter first, Iter last, T value);
```

- *Результат использования перемещения:* перемещение (moving) может быть более эффективным, чем копирование (см. Семантика перемещения). Например, **std::sort()** и **std::st::insert()**, основанные на перемещении до 15 раз быстрее аналогичных версий, использующих копирование. На самом деле влияние не такое существенное, как кажется, поскольку операции для таких стандартных типов, как **string** и **vector**, обычно оптимизированы для использования перемещения, путем замены стандартных функций **swap** на специализированные версии, использующих перемещение. Однако, если *ваш* тип содержит операторы перемещения, вы автоматически получите повышение производительности при использовании стандартных алгоритмов. Обратите также внимание на то, что перемещение позволяет простое и эффективное использование сортировки (и другие алгоритмы) для контейнеров с «умными» указателями, в частности, с unique_ptr:

```
// сравнивает разыменованные (*P) значения
template<class P> struct Cmp<P> {
        bool operator() (P a, P b) const { return *a<*b; }
}

vector<std::unique_ptr<Big>> vb;
// заполняет vb unique_ptr-ами объектов Big

// не пытайтесь делать это с auto_ptr
sort(vb.begin(),vb.end(),Cmp<Big>());
```

- *Использование лямбда-выражений:* Многие годы люди жаловались на то, что им приходилось писать функции или (лучше) функторы для использования в таких операциях, как **Cmp<T>** для алгоритмов стандартной библиотеки (и других алгоритмов). Это было особенно болезненным в случае больших функций (которые, кстати, писать не нужно), поскольку в C++98 нельзя определить локальный функтор для использования его в качестве аргумента; сейчас вы это сделать можете. Лямбда-выражения позволяют создавать операторы «на лету».

```
sort(vb.begin(),vb.end(),
        [](unique_ptr<Big> a, unique_ptr<Big> b) { return *a<*b; });
```

Я думаю, что поначалу лямбда-выражениями, как и любыми другими мощными новыми возможностями, будут злоупотреблять.

- *Использование списков инициализации*. Иногда списки инициализаторов могут быть очень удобными для использования в качестве аргументов. Например, у нас есть несколько строковых переменных и предикат **Nocase**, для сравнения строк без учета регистра:

```
auto x = max({x,y,z}, Nocase());
```

См. также:

- 25 Algorithms library [algorithms]
- 26.7 Generalized numeric operations [numeric.ops]
- Howard E. Hinnant, Peter Dimov, and Dave Abrahams: A Proposal to Add Move Semantics Support to the C++ Language. N1377=02-0035.

Улучшения контейнеров

Что же случилось со стандартными контейнерами, после появления новых языковых возможностей и десяти лет дополнительного опыта? Прежде всего, у нас появилось несколько новых контейнеров: array (контейнер фиксированного размера), forward_list (односвязный список) и неупорядоченные контейнеры (хеш-таблицы). Потом, контейнеры начали использовать новые языковые возможности, такие как списки инициализаторов, rvalue ссылки, шаблоны с переменным числом аргументов и constexpr. Давайте в качестве примера рассмотрим **std::vector**.

- *Списки инициализации*. Наиболее видимым улучшением является возможность конструирования с помощью списка инициализации, что позволяет контейнеру принимать список инициализации в качестве аргумента конструктора:

```
vector<string> vs = { "Hello", ", ", "World!", "\n" };
for (auto s : vs ) cout << s;
```

- *Операторы перемещения*. Контейнеры теперь содержат конструктор и оператор перемещения (в дополнение к существующим операциям копирования). Наиболее важным следствием из этого, является возможность эффективного возвращения из функций контейнеров:

```
vector<int> make_random(int n)
{
        vector<int> ref(n);
        // некоторый генератор случайных чисел
        for(auto& x : ref) x = rand_int(0,255);
        return ref;
}

vector<int> v = make_random(10000);
for (auto x : make_random(1000000)) cout << x << '\n';
```

Смысл здесь заключается в том, что вектор не копируется. Перепишите этот код с выделением вектора в куче и вы столкнетесь с проблемами управления памятью. Перепишите этот код так, чтобы вектор передавался в качестве аргумента в функцию **make_random** и вы получите значительно менее понятный код (в котором, к тому же, легче допустить ошибку).

- *Улучшенные операции добавления элементов*. Моей любимой операцией контейнера является **push_back()**, которая позволяет элегантно увеличивать размер контейнеру:

```
vector<pair<string,int>> vp;
string s;
int i;
while(cin>>s>>i) vp.push_back({s,i});
```

В этом коде будет создан объект **pair<string,int>**, по значениям **s** и **i**, который затем будет перемещен в **vp**. Обратите внимание, он будет не «скопирован», а «перемещен». Существует перегруженная версия метода **push_back**, которая принимает в качестве аргумента rvalue ссылку, что позволяет использовать конструктор перемещения класса **string**. Также обратите внимание на использование унифицированного синтаксиса инициализации, что приводит к более короткому коду.

- *Устанавливающие (emplace) операции.* Метод **push_back()** использует конструктор перемещения, который значительно более эффективен по сравнению со стандартными операциями копирования, но мы можем пойти еще дальше. Зачем вообще что-то копировать/перемещать? Почему бы не выделить в векторе нужный объем памяти и просто

не создать нужное значение прямо в этой памяти? Такие операции называются “emplace” (т.е. «создающие на месте»). Давайте в качестве примера рассмотрим **emplace_back()**:

```
vector<pair<string,int>> vp;
string s;
int i;
while(cin>>s>>i) vp.emplace_back(s,i);
```

- Устанавливающие (emplace) методы принимают шаблонный параметр с переменным числом аргументов (variadic template), который используется для создания требуемого типа. Является ли метод **emplace_back()** действительно более эффективным по сравнению с методом **push_back()** зависит от используемого типа и реализации (библиотеки и шаблонов с переменным числом аргументов). Если вы считаете, что эта разница может быть существенной, то измерьте ее. В противном случае отталкивайтесь от эстетических соображений и выбирайте между **vp.push_back({s,i})** и **vp.emplace_back(s,i);** по вашему вкусу. Сейчас я предпочитаю использовать **push_back()**, но я могу изменить свою точку зрения в будущем.

- *Аллокаторы области видимости (scoped allocators).* Контейнеры теперь могут хранить «настоящие объекты-аллокаторы (с состоянием)», и использовать их для управления вложенными выделениями памяти (например, для выделения памяти под элементы контейнера).

Очевидно, что контейнеры – это не единственное место стандартной библиотеки, которое пользуется преимуществами новых языковых возможностей. Так, например:

- *Вычисления времени компиляции (compile-time evaluation)*: constexpr используется для гарантированного вычисления выражений во время компиляции для **bitset**, **duration**, **char_traits**, **complex**, array, для атомарных типов, случайных чисел и т.д.

- *Кортежи (tuples)*: Кортежи были бы невозможны без шаблонов с переменным числом аргументов.

Аллокаторы с дополнительным состоянием

Для простоты и компактности контейнеров C++98 не требует от них поддержки аллокаторов с состоянием. Аллокаторы не должны были хранится в объектах контейнеров. Такое поведение все еще используется по умолчанию в C++11, но появилась возможность использования аллокаторов с состоянием, например, можно использовать аллокатор, содержащий указатель на область памяти из которой происходит выделение. Например:

```
// Вариант для C++98 не содержит данных
template<class T> class Simple_alloc {
        // обычная реализация аллокатора
};

class Arena {
        void* p;
        int s;
public:
        Arena(void* pp, int ss);
        // выделяет из диапазона p[0..ss-1]
};
```

```
template<class T> struct My_alloc {
        Arena& a;
        My_alloc(Arena& aa) : a(aa) { }
        // обычная реализация аллокатора
};

Arena my_arena1(new char[100000],100000);
Arena my_arena2(new char[1000000],1000000);

// память выделяется аллокатором по умолчанию
vector<int> v0;

// выделяет из my_arena1
vector<int,My_alloc<int>> v1(My_alloc<int>{my_arena1});

// выделяет из my_arena2
vector<int,My_alloc<int>> v2(My_alloc<int>{my_arena2});

// выделяет с помощью Simple_alloc
vector<int,Simple_alloc<int>> v3;
```

Обычно, для облегчения синтаксиса используют typedef.

Нет никакой гарантии того, что **Simple_alloc** и аллокатор, используемый по умолчанию столкнутся с нехваткой памяти, но это можно гарантировать с помощью небольшого количества метапрограммирования шаблонов. Таким образом, использование аллокатора приводит к дополнительному расходу памяти только в том случае, когда аллокатор обладает состоянием (как **My_alloc**).

При использовании пользовательских аллокаторов с контейнерами может возникнуть одна коварная проблема: должен ли элемент располагаться в той же самой выделенной области, что и контейнер? Например, если вы используете **Your_allocator** для **Your_string** для выделения его элементов, и я использую **My_allocator** для выделения элементов в объекте **My_vector**, тогда какой аллокатор должен использоваться для элементов в типе **My_vector<Your_allocator>>**? Решение заключается в возможности указать контейнеру какой аллокатор должен передаваться его элементам. Например, предположим, что у меня есть аллокатор **My_alloc** и я хочу, чтобы **vector<string>** использовал **My_alloc** для элементов вектора и для выделения элементов строки. Прежде всего, нужно создать версию строки с **My_alloc**.

```
// строка с нужным аллокатором
using xstring = basic_string<char,
                        char_traits<char>, My_alloc<char>>;
```

Затем, мне нужно создать версию **vector**, принимающую строки и **My_alloc**, и передающий этот аллокатор строке:

```
using svec = vector<xstring,
                scoped_allocator_adaptor<My_alloc<xstring>>>;
```

Теперь мы можем создать аллокатор типа **My_alloc<xstring>**:

```
svec v(svec::allocator_type(My_alloc<xstring>{my_arena1}));
```

Теперь **svec** – это вектор строк, который использует **My_alloc** для выделения памяти для своих элементов. Новое поведение заключается в использовании «адаптера» (“wrapper”) из стандартной

библиотеки под названием **scoped_allocator_adaptor**, который указывает на то, что строки также должны использовать **My_alloc**. Обратите внимание, что адаптер может (очень легко) преобразовывать **My_alloc<xstring>**, требуемый типу **xstring**.

Так что у нас есть 4 варианта:

```
// vector и string используют свой собственный
// аллокатор (аллокатор по умолчанию):
using svec0 = vector<string>;
svec0 v0;

// vector (только) использует My_alloc, а string свой
// собственный аллокатор (аллокатор по умолчанию):
using svec1 = vector<string,My_alloc<string>>;
svec1 v1(My_alloc<string>{my_arena1});

// vector и string используют My_alloc:
using xstring = basic_string<char, char_traits<char>, My_alloc<char>>;
using svec2 = vector<xstring,scoped_allocator_adaptor<My_alloc<xstring>>>;
svec2 v2(scoped_allocator_adaptor<My_alloc<xstring>>{my_arena1});

// vector использует My_alloc и string использует My_string_alloc:
using xstring2 = basic_string<char, char_traits<char>, My_string_alloc<char>>;
using svec3 = vector<xstring2,scoped_allocator_adaptor<My_alloc<xstring>,
My_string_alloc<char>>>;
svec3 v3(scoped_allocator_adaptor<My_alloc<xstring2>,
My_string_alloc<char>>{my_arena1,my_string_arena});
```

Очевидно, что первая версия, **svec0**, будет использоваться наиболее часто, но для систем с серьезными ограничениями памяти другие версии (особенно **svec2**) могут быть очень важны. Использование нескольких **typedef** немного повысят читабельность кода, но к счастью, это не тот код, который вам придется писать каждый день. **scoped_allocator_adapter2** является разновидностью **scoped_allocator_adapter** для случаев, когда используются разные кастомные аллокаторы.

См. также:

- Standard: 20.8.5 Scoped allocator adaptor [allocator.adaptor]
- Pablo Halpern: The Scoped Allocator Model (Rev 2). N2554=08-0064.

---

std::array

array – это стандартный контейнер, определенный в **<array>**, который представляет собой последовательность фиксированного размера с возможностью случайного доступа к элементам. У него нет дополнительного расхода памяти для хранения элементов, память для элементов выделяется в куче, он может инициализироваться с помощью списка инициализации, знает свой размер (количество элементов), и не может быть неявным образом преобразован к указателю. Другими словами, этот класс очень похож на стандартный массив, но без его недостатков.

```
array<int,6> a = { 1, 2, 3 };
a[3]=4;
// x равен 0 поскольку элементы по умолчанию
// инициализируются нулями
int x = a[5];
// ошибка: std::array не может быть неявно преобразован к указателю
```

```
int* p1 = a;
// ok: получаем указатель на первый элемент
int* p2 = a.data();
```

Обратите внимание, что вы можете получить пустой **array**, однако определить его размер по списку инициализации не получится:

```
// ошибка: размер неизвестен/отсутствует
array<int> a3 = { 1, 2, 3 };
// ok: пустой массив
array<int,0> a0;
// поведение не определено; не делайте этого
int* p = a0.data();
```

Стандартные возможности **array** делают его привлекательными при разработке встроенных систем (или аналогичного ПО, к которому предъявляются высокие требования по производительности и безопасности или же наложены ограничениями по использованию памяти. Это последовательный контейнер, поэтому он предоставляет типичный набор членов и функций (как, например, и класс **vector**):

```
template<class C> C::value_type sum(const C& a)
{
        return accumulate(a.begin(),a.end(),0);
}

array<int,10> a10;
array<double,1000> a1000;
vector<int> v;
// ...
int x1 = sum(a10);
int x2 = sum(a1000);
int x3 = sum(v);
```

Кроме того, вы не получите (потенциально опасного) преобразования наследников к базовому классу:

```
struct Apple : Fruit { /* ... */ };
struct Pear : Fruit { /* ... */ };

void nasty(array<Fruit*,10>& f)
{
        f[7] = new Pear();
};

array<Apple*,10> apples;
// ...
// ошибка: невозможно преобразовать array<Apple*,10> к array<Fruit*,10>;
nasty(apples);
```

Если бы это было возможным, то в переменной **apples** содержались бы экземпляры **Pear**.

См. также:

- Standard: 23.3.1 Class template array

std::forward_list

Стандартный контейнер **forward_list** определен в файле **<forward_list>** и является по сути односвязным списком. Он поддерживает итерирование только вперед и гарантирует, что элемент не будет перемещен после вставки или удаления другого элемента. Этот класс расходует минимальный объем памяти (пустой список будет занимать скорее всего одно слово) и не предоставляет метод **size()** (поэтому ему не нужно содержать дополнительное поле с размером):

```
template <ValueType T, Allocator Alloc = allocator<T> >
        requires NothrowDestructible<T>
class forward_list {
public:
        // обычные члены контейнеров
        // нет метода size()
        // нет возможности итерироваться в обратном направлении
        // нет методов back() или push_back()
};
```

См. также:

- Standard: 23.3.3 Class template forward_list

Неупорядоченные контейнеры

Неупорядоченные контейнеры являются разновидностью хеш-таблиц. В C++11 представлены следующие типы:

- **unordered_map**
- **unordered_set**
- **unordered_multimap**
- **unordered_multiset**

Эти классы должны называться **hash_map** и т.п., но поскольку этими именами очень часто злоупотребляют, то при выборе новых имен комитет решил остановиться на **unordered_map** и т.д. как на наименьшем зле. “unordered” в названии классов говорит об отличиях между **map** и **unordered_map**: перебор элементов объекта типа **map** происходит в порядке, зависящем от оператора сравнения (по умолчанию используется <), в то время как значения в **unordered_map** не должны содержать оператор сравнения и хеш-таблица по своей природе не упорядочена. И наоборот, элементам, хранящимся в **map** не требуется хеш-функция.

Основная идея сводится к тому, чтобы использовать **unordered_map** в качестве более оптимизированной версии **map**, когда такая оптимизация возможна и обоснована. Например:

```
map<string,int> m {
        {"Dijkstra",1972}, {"Scott",1976},
        {"Wilkes",1967}, {"Hamming",1968}
};
m["Ritchie"] = 1983;
for(auto x : m) cout << '{' << x.first << ',' << x.second << '}';

unordered_map<string,int> um {
        {"Dijkstra",1972}, {"Scott",1976},
        {"Wilkes",1967}, {"Hamming",1968}
};
um["Ritchie"] = 1983;
for(auto x : um) cout << '{' << x.first << ',' << x.second << '}';
```

Итератор объекта **m** будет перебирать элементы в алфавитном порядке, а итератор объекта **um** – нет (если только не по чистой случайности). Процессы поиска элементов в **m** и **um** абсолютно различны. Поиск в **m** осуществляется за log2(m.size()) сравнений, в то время как поиск в **um** приводит к одному вызову хеш-функции и одной или более проверке равенства. Для нескольких элементов (скажем, для нескольких десятков), сложно сказать, какой из этих вариантов будет работать быстрее. Для большого количества элементов (например, если речь идет о тысячах) - поиск в **unordered_map** может быть значительно быстрее чем в **map**.

Продолжение следует.

См. также:

- Standard: 23.5 Unordered associative containers.

---

std::tuple

Определен в **<tuple>**. Представляет собой упорядоченную последовательность из N значений, где N является константой от 0 до большого значения, зависящего от реализации. Вы можете рассматривать кортеж (tuple) как неименованную структуру, члены которой соответствуют типам элементов кортежа. Следует отметить, что элементы в **tuple** хранятся компактным образом; кортеж не является связным списком.

Типы элементов кортежа могут задаваться явно, а могут выводиться (при вызове **make_tuple()**), доступ к элементам осуществляется с помощью индекса (начинается с 0) путем вызова метода **get()**:

```
tuple<string,int> t2("Kylling",123);

// t будет типа tuple<string,int,double>
auto t = make_tuple(string("Herring"),10, 1.23);
string s = get<0>(t);
int x = get<1>(t);
double d = get<2>(t);
```

Кортежи используются (явно или неявно) для хранения гетерогенного списка элементов, известного во время компиляции, и когда нет желания определять для этого именованный класс. Например, **tuple** используется для хранения элементов внутри **std::function** и **std::bind**.

Наиболее часто используется кортеж из двух элементов, т.е. пара (pair). Однако пара напрямую поддерживается классом **std::pair** из стандартной библиотеки(20.3.3 Pairs). **pair** может использоваться для инициализации кортежа, но обратное преобразование невозможно.

Операторы сравнения (==, !=, <, <=, > и >=) определены для кортежей совместимых типов.

См. также:

- Standard: 20.5.2 Class template tuple
- Variadic template paper
- Boost::tuple

Метапрограммирование и характеристики типов

Простите. Возвращайтесь к этому разделу позже.

---

std::function и std::bind

Стандартные функторы **bind** и **function** определены в **<functional>** (вместе со многими другими полезными функторами); они предназначены для работы с функциями и их аргументами. **bind** принимает функцию (или функтор, или все что угодно, что можно вызвать с помощью **(…)**) и возвращает функтор с одним или более «привязанным» (bound) аргументом, либо с другим порядком аргументов. Например:

```
int f(int,char,double);
// выводим тип возвращаемого значения
auto ff = bind(f,_1,'c',1.2);
// f(7,'c',1.2);
int x = ff(7);
```

Такая привязка аргументов обычно называется «каррированием» (currying). **_1** – это заместитель (placeholder), указывающий на первый аргумент функции **ff**, который будет передан функции **f** при вызове ее через **ff**. Первый аргумент называется **_1**, второй - **_2** и т.д. Например:

```
int f(int,char,double);
// изменяем порядок аргументов на противоположный
auto frev = bind(f,_3,_2,_1);
// f(7,'c',1.2);
int x = frev(1.2,'c',7);
```

Обратите внимание, как auto избавляет от необходимости явно указывать тип возвращаемого значения метода **bind**.

Связать аргументы перегруженной функции нельзя, для этого мы должны явно указать, какая нам нужна версия перегруженной функции:

```
int g(int);
// g() перегружена
double g(double);

// ошибка: какая версия g()?
auto g1 = bind(g,_1);
// ok (но выглядит ужасно)
auto g2 = bind((double(*)(double))g,_1);
```

Существует две версии метода **bind()**: одна версия показана выше, вторая «устаревшая» - требует явного указания типа возвращаемого значения:

```
// явно указываем тип возвращаемого значения
auto f2 = bind<int>(f,7,'c',_1);
// f(7,'c',1.2);
int x = f2(1.2);
```

Необходимость (и широкая популярность) второй версии была обусловлена тем, что первую (более удобную в использовании) версию нельзя было реализовать в C++98.

function – это тип, который может хранить практически что угодно, что можно вызвать с помощью синтаксиса вызова метода **(...)**. В частности, результат вызова метода **bind** можно присвоить типу **function**. **function** использовать очень просто. Например:

```
// создаем функтор
function<float (int x, int y)> f;

// получаем нечто, что можно вызвать с помощью ()
struct int_div {
        float operator()(int x, int y) const {
                return ((float)x)/y;
        };
};

// присваиваем
f = int_div();
// вызываем функор
cout << f(5, 3) << endl;
// спокойно передаем куда-либо
std::accumulate(b,e,1,f);
```

Функцию-член можно рассматривать как свободную функцию с одним аргументом. Мы можем рассматривать **function**, как замену функторов стандартной библиотеки C++98: **mem_fun_t**, **pointer_to_unary_function** и т.д. Аналогичным образом, мы можем рассматривать **bind()**, как замену **bind1()** и **bind2()**.

См. также:

- Standard: 20.7.12 Function template bind, 20.7.16.2 Class template function
- Herb Sutter: Generalized Function Pointers. August 2003.
- Douglas Gregor: Boost.Function.
- Boost::bind

---

unique_ptr

- **unique_ptr** (определен в **<memory>**) и обеспечивает семантику строгого владения.
  - Владеет объектом, на который хранит указатель.
  - Не CopyConstructable и не CopyAssignable, однако MoveConstructible и MoveAssignable.
  - При собственном удалении (например, при выходе из области видимости (6.7)) уничтожает объект (на который хранит указатель) с помощью заданного метода удаления (с помощью deleter-а).
- Использование **unique_ptr** дает:
  - безопасность исключений при работе с динамически выделенной памятью,
  - передачу владения динамически выделенной памяти в функцию,
  - возвращения динамически выделенной памяти из функции,
  - хранение указателей в контейнерах
- «Это то, чем должен был быть **auto_ptr**» (но который мы не могли реализовать на C++98)

unique_ptr реализован главным образом на основе rvalue-ссылок и семантике перемещения.

Вот типовой фрагмент небезопасного с точки зрения исключений кода:

```
X* f()
{
        X* p = new X;
```

```
        // делаем что-то еще, возможна генерация исключения
        return p;
}
```

Решение заключается в том, чтобы **unique_ptr** хранил указатель на память в куче:

```
X* f()
{
        // или {new X}, но не = new X
        unique_ptr<X> p(new X);
        // делаем что-то еще, возможна генерация исключения
        return p.release();
}
```

Теперь, при генерации исключения, **unique_ptr** (неявно) освободит объект, на который указывает. Это основа идиомы RAII. Однако если нет необходимости возвращать голый указатель, мы можем сделать этот код еще лучше, вернув **unique_ptr**:

```
unique_ptr<X> f()
{
        // или {new X}, но не = new X
        unique_ptr<X> p(new X);
        // делаем что-то еще, возможна генерация исключения
        return p;       // владение передается из f()
}
```

Мы можем использовать **f** следующим образом:

```
void g()
{
        // перемещаем с помощью конструктора перемещения
        unique_ptr<X> q = f();
        // используем q
        q->memfct(2);
        // копируем объект, на который указываем
        X x = *q;
        // ...
}       // q и объект, которым он владеет удаляется
```

unique_ptr обладает «семантикой перемещения», так **q** инициализируется результатом метода **f()**, путем передачи владения.

Используйте **unique_ptr** в качестве элементов контейнеров вместо стандартных указателей для обеспечения безопасности исключений и гарантированное освобождение элементов контейнера.

```
vector<unique_ptr<string>> vs {
        new string{"Doug"}, new string{"Adams"} };
```

unique_ptr представляет собой «голый» указатель и накладные расходы его использования по сравнению со встроенным указателем – минимальны. В частности, **unique_ptr** не содержит никаких проверок во время выполнения.

См. также:

- the C++ draft section 20.7.10

- Howard E. Hinnant: unique_ptr Emulation for C++03 Compilers.

---

shared_ptr

shared_ptr используется для совместного владения; т.е. когда два участка кода должны иметь доступ к некоторым данным, но ни один из них не обладает эксклюзивным владением ими (в плане ответственности за удаление объекта). **shared_ptr** – это указатель со счетчиком ссылок, который удаляет хранимый объект, когда счетчик уменьшается до нуля. Ниже представлен искусственный пример:

```
void test()
{
        shared_ptr<int> p1(new int);   // счетчик равен 1
        {
                shared_ptr<int> p2(p1);        // счетчик равен 2
                {
                        shared_ptr<int> p3(p1);        // счетчик равен 3
                }       // счетчик уменьшается до 2
        } // счетчик уменьшается до 1
}       // здесь счетчик уменьшается до 0 и int удаляется.
```

Более реалистичным примером может служить указатель на узлы в некотором графе объектов, при этом некоторый код хочет удалить указатель на узел, но не знает, держит ли кто-то еще указатель на этот узел или нет. А что если узел содержит ресурсы, требующие некоторого действия в деструкторе (например, это можем быть дескриптор файла, который должен быть закрыт при удалении узла). Можно считать **shared_ptr** неким аналогом сборщику мусора, в тех случаях, когда у вас просто не достаточно мусора, чтобы использование сборщика мусора стало экономически выгодной, ваша среда выполнения его не поддерживает, или же память – является не единственным ресурсом, которым нужно управлять (например, когда речь идет о файловых дескрипторах).. Например:

```
// обратите внимание: можем быть множество
// указателей на Node из разных мест.
struct Node {
        shared_ptr<Node> left;
        shared_ptr<Node> right;
        File_handle f;
        // ...
};
```

В данном случае, деструктор **Node** (неявно сгенерированный деструктор нас вполне устроит) удаляет подузлы; т.е. будут вызваны деструкторы для **left** и **right**. Поскольку поле **left** – это **shared_ptr**, то Node (на который указывает left), будет удален только если **left** – последний, кто хранит на него ссылку; поведение поля **right** аналогично, при этом деструктор поля **f** делает все необходимое для освобождения собственных ресурсов.

Обратите внимание, что не нужно использовать **shared_ptr** просто для передачи владения из одного места в другое; для этого предназначен unique_ptr и делает он это лучше и с меньшими накладными расходами. Если вы использовали указатели со счетчиком ссылок для возвращения значений из фабричных методов и тому подобного, то подумайте об использовании в этих местах **unique_ptr** вместо **shared_ptr**.

Пожалуйста, не заменяйте бездумно обычные указатели на **shared_ptr**, в попытке избежать утечек памяти; **shared_ptr** не является панацеей и его использование не бесплатно:

- Циклически связанные структуры указателей **shared_ptr** приведут к утечке памяти (вам потребуется усложнение логики, для разрыва циклической связи, например, с помощью **weak_ptr**).
- «Совместно используемые объекты» обычно «живут» дольше, чем объекты, ограниченные текущей областью видимостью (scoped objects), что в среднем приводит к увеличению расхода ресурсов.
- Разделяемые указатели (shared pointers) в многопоточном окружении могут быть достаточно дорогими (поскольку необходимо предотвратить гонки при использовании счетчика ссылок).
- Деструктор разделяемых объектов вызывается в непредсказуемое время.
- Допустить ошибку в логике и алгоритме обновления любого разделяемого объекта значительно легче, по сравнению с неразделяемым объектом.

shared_ptr представляет *разделяемое владение*, но с моей точки зрения разделяемое владение не является идеальным вариантом: значительно лучше, когда у объекта есть конкретный владелец и его время жизни точно определено.

См. также:

- the C++ draft: Shared_ptr (20.7.13.3)

---

weak_ptr

Необходимость слабых указателей (weak pointers) обычно объясняется необходимостью разрыва циклических зависимостей в структурах данных, управляемых с помощью shared_ptr. Я думаю, что лучше рассматривать **weak_ptr**, как указатель, который:

1. обеспечивает доступ к объекту, только когда он существует, и
2. может быть удален кем-то другим, и
3. содержит деструктор, вызываемый после его последнего использования (обычно для удаления анонимного участка памяти).

Давайте рассмотрим реализацию старой игры «астероид». «Игра» владеет всеми астероидами, но каждый астероид должен отслеживать все соседние астероиды и обрабатывать столкновения. Столкновение обычно приводит к удалению одного или более астероидов. Каждый астероид должен содержать список соседних астероидов. Обратите внимание, что наличие астероида в этом списке не должно препятствовать уничтожению астероида (так что **shared_ptr** не подходит). С другой стороны, астероид не должен удаляться, когда один астероид анализирует возможность столкновения с другим астероидом. И, очевидно, деструктор астероида должен быть вызван для освобождения ресурсов (таких, как соединения с графической системой). Так что нам нужен список астероидов, которые *могут быть* не повреждены и возможность «позаимствовать» один из них на время. **weak_ptr** предназначен именно для этого:

```
void owner()
{
        // ...
        vector<shared_ptr<Asteroid>> va(100);
        for (int i=0; i<va.size(); ++i) {
                // ...находим соседние астероиды ...
                va[i].reset(new Asteroid(
                        weak_ptr<Asteroid>(va[neighbor]));
                launch(i);
        }
        // ...
```

```
}
```

После вызова метода **reset() shared_ptr** будет указывать на новый объект.

Очевидно, я очень сильно упростил этот метод дав каждому астероиду всего лишь одного соседа. Смысл заключается в том, что астероид содержит слабый указатель на соседний астероид. Владелец содержит **shared_ptr**, что обеспечивает владение, когда астероид рассматривается другим астероидом (но не наоборот). Расчет столкновений астероидов может выглядеть таким образом:

```
void collision(weak_ptr<Asteroid> p)
{
        // p.lock возвращает shared_ptr на объект p
        if (auto q = p.lock()) {
                // ... этот Asteroid все еще жив ...
        }
        else {
                // ... Упс!: этот Asteroid уже уничтожен: мы просто
                // о нем забыли (удаляем его weak_ptr) ...
        }
}
```

Обратите внимание, что если даже владелец решит закрыть игру и удалить все астероиды (путем удаления **shared_ptr**, представляющих владение) каждый астероид, рассчитывающий в данный момент результаты столкновения, завершит свою работу корректно (поскольку после вызова **p.lock()** он получает **shared_ptr**, который не может стать недействительным).

Я думаю, что **weak_ptr** будет использоваться значительно реже «обычного» **shared_ptr**, и я надеюсь, что **unique_ptr** станет популярнее **shared_ptr**, поскольку он представляет более простую (и более эффективную) идею владения и, как результат, проще в использовании.

См. также:

- the C++ draft: Weak_ptr (20.7.13.3)

Garbage collection ABI

Сборка мусора (garbage collection) (автоматическое освобождение регионов памяти, на которые никто не ссылается) является опциональной в C++; т.е. сборка мусора не является обязательной частью реализации. Однако, C++11 предоставляет определение того, что может делать сборщик мусора и также содержит ABI (Application Binary Interface) для управления его действиями.

Правила для указателей и их времени жизни определены в терминах «безопасного указателя» (“safely derived pointer”) (3.7.4.3); грубо говоря - это «указатель на объект или его подобъект, размещенный при помощи оператора new». Вот несколько примеров «небезопасных указателей» известных также как «замаскированные указатели» (disguised pointers), а также то, чего не стоит делать в понятных простым смертным и нормально работающих программах:

- Изменять указатель, чтобы он указывал некоторое время «в другое место»

```
int* p = new int;
p+=10;
// ... здесь может запуститься сборка мусора ...
p-=10;
// можем ли мы рассчитывать, что int еще не удален?
```

```
*p = 10;
```

- Преобразовывать указатель в int

```
int* p = new int;
// не переносимо
int x = reinterpret_cast<int>(p);
p=0;
// ... здесь может запуститься сборка мусора ...
p = reinterpret_cast<int*>(x);
// можем ли мы рассчитывать, что int еще не удален?
*p = 10;
```

- Существует множество еще более хитроумных трюков. Подумайте о вводе выводе, о битовых операциях и т.п.

Существуют законные основания использовать низкоуровневую работу с указателями (например, трюк с «исключающим или» в приложениях с сильными ограничениями по памяти), но они не так часто встречаются, как думают многие программисты.

Программист может явно указать места, в которых нельзя найти указателей (например, в изображении) и какую память нельзя освобождать, даже если сборщик не найдет указателей на нее:

```
// участок памяти, начиная с p
// (выделенный некоторым аллокатором,
// который знает размер выделенной памяти)
// и который не должен быть собран сборщиком мусора
void declare_reachable(void* p);

template<class T> T* undeclare_reachable(T* p);

// p[0..n] не содержит указателей
void declare_no_pointers(char* p, size_t n);
void undeclare_no_pointers(char* p, size_t n);
```

Программист может выяснить текущие правила безопасности указателей и освобождения памяти:

```
enum class pointer_safety {relaxed, preferred, strict };
pointer_safety get_pointer_safety();
```

Раздел 3.7.4.3[4]: результат разыменования или освобождения небезопасного указателя не определен, если объект указателя располагается в динамической памяти и указатель не был помечен как доступный (20.7.13.7).

- **relaxed**: безопасные (safely-derived) и небезопасные (not safely-derived) указатели анализируются одинаково; это поведение аналогично поведению в C и в C++98, но я преследовал не эту цель; я хотел позволить сборку мусора, даже если пользователь не хранит корректные указатели.
- **preferred**: аналогично **relaxed**, но сборщик мусора может работать для определения утечек памяти и/или для определения разыменования «плохих указателей».
- **strict**: безопасные (safely-derived) и небезопасные (not safely derived) указатели могут анализироваться по разному; т.е. сборщик мусора может игнорировать небезопасные указатели.

Не существует стандартного способа выбрать правильный вариант. Это можно рассматривать как проблему «качества реализации» и «программного окружения».

См. также:

- the C++ draft 3.7.4.3
- the C++ draft 20.7.13.7
- Hans Boehm's GC page
- Hans Boehm's Discussion of Conservative GC
- final proposal
- Michael Spertus and Hans J. Boehm: The Status of Garbage Collection in C++0X. ACM ISMM'09.

Модель памяти

Модель памяти (memory model) – это соглашение между аппаратной архитектурой и разработчиками компиляторов, которое позволяет большинству разработчиков не думать о деталях аппаратного обеспечения современных компьютеров. Без модели памяти лишь некоторые аспекты, связанные с многопоточностью, блокировками и lock-free программированием имели бы смысл.

Ключевая гарантия здесь следующая: два потока исполнения могут обновлять и получать доступ к разным ячейкам памяти, не влияя друг на друга. Но что такое «ячейка памяти» (memory location)? Ячейка памяти – это либо объект скалярного типа, либо максимальная последовательность непрерывных битовых полей с ненулевой шириной. Например, структура **S** содержит в точности четыре ячейки памяти:

```
struct S {
        // область #1
        char a;
        // область #2
        int b:5,
        int c:11,
        // обратите внимание: :0 – это "особый" случай
        int :0,
        // область #3
        int d:8;
        // область #4
        struct {int ee:8;} e;
};
```

Почему это так важно? Почему так неочевидно? Было ли это всегда так? Проблема заключается в том, что если несколько вычислений могут реально выполняться параллельно, т.е. (видимо) несвязанные инструкции могут выполняться в одно и то же время, могут вылезать наружу особенности оборудования по работе с памятью. На самом деле, без поддержки компилятором, проблемы исполнения инструкций и конвейерной обработки данных, а также использование кэша, они *обязательно* вылезут наружу в совершенно неуправляемом виде для прикладного программиста. Это так, даже при отсутствии двух потоков, работающих с общими данными! Давайте рассмотрим два отдельных скомпилированных «потока»:

```
// поток 1:
char c;
c = 1;
int x = c;

// поток 2:
char b;
b = 1;
int y = b;
```

Для большего реализма я мог бы использовать раздельную компиляцию (каждого потока), чтобы гарантировать, что компилятор/оптимизатор не смогут избавиться от доступа к памяти, и просто проигнорировать переменные **c** и **b**, и напрямую проинициализировать **x** и **y** единицей. Чему могут равняться **x** и **y**? Согласно спецификации языка C++11 единственным корректным и очевидным ответом является следующий: 1 и 1. Причина, по которой это важно заключается в том, что если вы возьмете обычный хороший компилятор C или C++ домногопоточной эры, то возможными ответами на предыдущий вопрос могут быть: 0 и 0 (маловероятно), 1 и 0, 0 и 1, и 1 и 1. И такое поведение происходило на практике. Как? Линковщик мог выделить память для **c** и **b** рядом друг с другом (в том же слове) и ничего в стандартах C и C++ 90-х годов не говорили о том, что так делать нельзя. В этом вопросе C++ похож на другие языки, разработанные без учета параллелизма. Однако, большинство современных процессоров не могут читать и писать по одному символу, они должны прочитать или записать целое слово, так что присваивание переменной **c** на самом деле выглядит так: «прочитать слово, в котором содержится **c**, заменить часть, которая содержит **c** и записать слово целиком назад». Поскольку присваивание **b** аналогично, то существует масса возможностей для двух потоков помешать друг другу, даже если ни один из потоков (согласно исходному тексту) не использует общие данные!

Итак, C++11 гарантирует, что такое невозможно для «разных ячеек памяти». Если быть более точным, то помимо операций чтения ячейка памяти не может использоваться двумя потоками без некоторой формы блокировки. Обратите внимание, что разные битовые поля внутри одного слова не являются разными ячейками памяти, поэтому структуры с битовыми полями не следует раздельно использовать между потоками без некоторой формы блокировки. Если не вдаваться в детали, то модель памяти в C++ именно «наиболее ожидаемая».

Однако, низкоуровневые проблемы многопоточности не всегда очевидны. Давайте рассмотрим пример:

```
// изначально x==0 и y==0

if (x) y = 1;   // Поток 1

if (y) x = 1;   // Поток 2
```

Если ли проблема с этим кодом? А если точнее, есть ли здесь гонка (data race)? (Нет, ее нет).

К счастью, мы уже привыкли к новым временам и любой C++ компилятор с поддержкой многопоточности (который я знаю) выдает единственный правильный ответ уже многие годы. И так происходит с большинством (к сожалению, не со всеми) коварными вопросами. В конце концов, C++ использовался в разработке сложных многопоточных систем очень долго. Новый стандарт должен еще улучшить эту ситуацию.

См. также:

- Standard: 1.7 The C++ memory model [intro.memory]
- Paul E. McKenney, Hans-J. Boehm, and Lawrence Crowl: C++ Data-Dependency Ordering: Atomics and Memory Model. N2556==08-0066.
- Hans-J. Boehm: Threads basics, HPL technical report 2009-259. ``what every programmer should know about memory model issues.''
- Hans-J. Boehm and Paul McKenney: A slightly dated FAQ on C++ memory model issues.

Потоки

Поток (thread) – это способ представления исполнения/вычисления в программе. В C++11, как и в большинстве современных языков, поток может (и обычно так и делает) разделять адресное пространство других потоков. В этом он отличается от процессов (process), который обычно не разделяет данные с другими процессами. C++ в прошлом неоднократно применялся для реализации потоков для многих аппаратных платформ и операционных систем, новшество же заключается в появлении потоков в стандартной библиотеке.

О многопоточности, параллелизме и конкрурентности написано множество толстых книг и десятки тысяч статей, в этом FAQ дается лишь краткий обзор. Четко понимать понятие многопоточности *сложно*. Если вы хотите заниматься параллельным программированием, как минимум прочитайте книгу. Не рассчитывайте только лишь на руководства, стандарт или FAQ.

Запуск потока осуществляется путем создания объекта **std::thread**, в который передается функция или функтор (включая лямбда-выражение):

```
#include <thread>

void f();

struct F {
        void operator()();
};

int main()
{
        // f() исполняется в отдельном потоке
        std::thread t1{f};
        // F()() исполняется в отдельном потоке
        std::thread t2{F()};
}
```

К сожалению, это едва ли приведет к полезным результатам, чтобы ни делали **f()** и **F()**. Проблема заключается в том, что программа завершится еще до того, как **t1** исполнит **f()** и до того, как **t2** исполнит **F()**. Нам нужно дождаться выполнения этих двух задач:

```
int main()
{
        // f() исполняется в отдельном потоке
        std::thread t1{f};
        // F()() исполняется в отдельном потоке
        std::thread t2{F()};

        t1.join();      // ждем завершения t1
        t2.join();      // ждем завершения t2
}
```

Вызов метода **join()** гарантирует, что мы не завершим выполнение приложения пока потоки не завершаться. “join” означает “дождаться завершения потока”.

Обычно мы передаем некоторые аргументы для выполнения задачи (выполнение некоторого кода в другом потоке я называю задачей (task)). Например:

```
void f(vector<double>&);

struct F {
        vector<double>& v;
```

```
        F(vector<double>& vv) :v{vv} { }
        void operator()();
};

int main()
{
        // f(some_vec) исполняется в отдельном потоке
        std::thread t1{std::bind(f,some_vec)};
        // F(some_vec)() исполняется в отдельном потоке
        std::thread t2{F(some_vec)};

        t1.join();
        t2.join();
}
```

Стандартная функция bind по сути создает функтор из переданных аргументов.

В общем случае, после выполнения задачи мы хотим получить результаты ее исполнения. В простом случае возвращаемого значения не существует. Если результат существует, то я рекомендую использовать std::future. В противном случае, можно передать в задачу аргумент, в который будет записан результат после окончания ее исполнения. Например:

```
// поместить результаты в res
void f(vector<double>&, double* res);

struct F {
        vector& v;
        double* res;
        F(vector<double>& vv, double* p) :v{vv}, res{p} { }
        void operator()();     // поместить результат в res
};

int main()
{
        double res1;
        double res2;

        // f(some_vec,&res1) исполняется в отдельном потоке
        std::thread t1{std::bind(f,some_vec,&res1)};
        // F(some_vec,&res2)() исполняется в отдельном потоке thread
        std::thread t2{F(some_vec,&res2)};

        t1.join();
        t2.join();

        std::cout << res1 << ' ' << res2 << '\n';
}
```

А как насчет ошибок? Что если задача генерирует исключение? Если задача генерирует исключение, и оно не обрабатывается в блоке catch, то вызывается метод **std::terminate()**. А это обычно означает закрытие приложения, чего обычно мы очень сильно хотим избежать. std::future может передать исключение в родительский/вызывающий поток; это одна из причин, почему мне нравится эта возможность. В противном случае, можно вернуть некоторый код ошибки.

Когда **thread** покидает область видимости приложение завершается вызовом метода **std::terminate()**, если текущая задача еще не завершена. Очевидно, этого следует избегать.

Не существует возможности запросить поток завершить исполнение (например, потребовать его завершиться как можно быстрее и насколько возможно безопасно), или принудительно прервать его

исполнение (т.е. убить его). Мы оставили следующие возможности для реализации такого поведения:

- Разработать собственный механизм кооперативный отмены исполнения (с помощью разделяемых данных, вызывающий поток может установить значение, которое будет прочитано вызываемым потоком для быстрого и безопасного завершения),
- С помощью «нативного подхода» (получив доступ к дескриптору операционной системы с помощью **thread::native_handle()**),
- Завершить процесс (с помощью **std::quick_exit()**),
- Завершить программу (с помощью **std::terminate()**).

Это все, о чем смогли договориться члены комитета. В частности, представители POSIX были ярыми противниками любой формы «отмены исполнения», хотя многие модели ресурсов в C++ основаны на деструкторах. Не существует идеального решения для любой системы и для всех возможных приложений.

Главная проблема при использовании потоков заключается в гонках (data races); т.е. два потока исполняются в одном адресном пространстве и могут независимо обращаться к объекту таким образом, что это приведет к непредвиденным результатам. Если один (или оба) потока будут изменять данные объекта и другой (или оба) потока будут читать данные объекта, то произойдет «гонка» за то, какая из операций будет выполнена первой. Результаты могут быть не просто неопределены; они обычно совершенно непредсказуемы. Во избежание гонок C++11 предоставляет программисту некоторые правила/гарантии:

- Функции стандартной библиотеки не должны прямо или косвенно обращаться к объектам, доступным из других потоков, если только эти объекты прямо или косвенно не передаются через аргументы функций (включая **this**).
- Функции стандартной библиотеки не должны прямо или косвенно изменять объекты, доступные из других потоков, если только эти объекты прямо или косвенно не передаются через аргументы функции (включая **this**).
- Реализации стандартной библиотеки должны избегать гонок за разные элементы в одной последовательности, которые могут изменяться параллельно.

Параллельный доступ к потоку ввода-вывода (stream) объектов, буферизированному потоку объектов или потоку из стандартной библиотеки C может привести к гонкам, если не указано обратное. Так, не следует разделять доступ к потокам ввода-вывода между двумя потоками исполнения, пока вы каким-либо образом не контролируете к ним доступ.

Вы можете:

- Ждать поток указанное время.
- Контролировать доступ к данным с помощью взаимного исключения (mutual exclusion).
- Контролировать доступ к данным с помощью блокировок.
- Ожидать действия другой задачи с помощью условных переменных.
- Возвращать значение из потока с помощью future.

См. также:

- Standard: 30 Thread support library [thread]
- 17.6.4.7 Data race avoidance [res.on.data.races]
- ???

- H. Hinnant, L. Crowl, B. Dawes, A. Williams, J. Garland, et al.: Multi-threading Library for Standard C++ (Revision 1) N2497==08-0007
- H.-J. Boehm, L. Crowl: C++ object lifetime interactions with the threads API N2880==09-0070.
- L. Crowl, P. Plauger, N. Stoughton: Thread Unsafe Standard Functions N2864==09-0054.
- WG14: Thread Cancellation N2455=070325.

Взаимное исключение

mutex – это примитивный объект, используемый для управления доступом в многопоточной системе. Вот наиболее простой способ использования:

```
std::mutex m;
int sh; // разделяемые данные
// ...
m.lock();
// выполняем операции над разделяемыми данными:
sh+=1;
m.unlock();
```

Только один поток исполнения (thread) может находиться в регионе между **lock()** и **unlock()** (обычно его называют критическим регионом или критической секцией, critical region). Если второй поток попытается вызвать **m.lock()**, в то время, когда исполнение другого потока находится внутри этого региона, второй поток будет заблокирован до тех пор, пока первый поток не вызовет метод **m.unlock()**. Это простой случай. Но вот что непросто, так это использовать мьютекс таким образом, чтобы это не привело к серьезным неприятностям. А что, если поток «забудет» вызвать **unlock()**? А что если поток попытается вызвать **lock()** для того же самого мьютекса второй раз? А что если пройдет очень много времени, прежде чем поток вызовет **unlock()**? А что если потоку нужно захватить два мьютекса? Для ответов на эти вопросы пишутся целые книги. В этом разделе (и в разделе о блокировках) описаны лишь самые базовые случаи.

Помимо метода **lock() mutex** содержит метод **try_lock()**, который может использоваться для попытки попасть в критическую секцию без риска блокировки:

```
std::mutex m;
int sh; // разделяемые данные
// ...
if (m.try_lock()) {
        // работа с разделяемыми данным:
        sh+=1;
        m.unlock();
else {
        // возможно, делаем что-то еще
}
```

reqursive_mutex – это мьютекс, который может быть захвачен одним и тем же потоком более одного раза:

```
std::recursive_mutex m;
int sh; // разделяемые данные
// ...
void f(int i)
{
        // ...
        m.lock();
        // работа с разделяемыми данными:
```

```
        sh+=1;
        if (--i>0) f(i);
        m.unlock();
        // ...
}
```

Здесь я просто вызываю метод **f()** рекурсивно. Однако обычно код может быть более запутанным. Рекурсивный вызов может быть неявным, например, **f()** вызывает **g()**, который вызывает **h()**, который, в свою очередь вызывает **f()**.

А что если мне нужно захватить мьютекс на десять секунд? Класс **timed_mutex** предназначен для этого. Он содержит специализированные версии метода **try_lock()** с ограничением по времени:

```
std::timed_mutex m;
int sh; // разделяемые данные
// ...
if (m.try_lock_for(std::chrono::seconds(10))) {
        // работа с разделяемыми данными:
        sh+=1;
        m.unlock();
}
else {
        // мы не захватили мьютекс, делаем что-то еще
}
```

Метод **try_lock_for()** принимает относительно время (duration), в качестве аргумента. Если вы хотите ожидать до фиксированного момента времени, то можете использовать time_point в методе **try_lock_until()**:

```
std::timed_mutex m;
int sh; // разделяемые данные
// ...
if (m.try_lock_until(midnight)) {
        // работа с разделяемыми данными:
        sh+=1;
        m.unlock();
}
else {
        // мы не захватили мьютекс, делаем что-то еще
}
```

Ожидание до полуночи – это моя слабая попытка пошутить: для такого низкоуровневого механизма, как мьютексы, время обычно исчисляется миллисекундами, а не часами.

И, конечно же, существует **recursive_timed_mutex**.

Мьютекс является ресурсом (обычно он содержит реальные ресурсы) и для нормального использования должен быть доступен как минимум двум потокам. Поэтому мьютекс не может быть скопирован или перемещен (вы не можете просто так сделать копию аппаратных регистров).

Бывает удивительно сложным сопоставить все вызовы методов **lock()** и **unlock()**. Подумайте о сложных ветвлениях, ошибках и исключениях. Если у вас есть выбор, то используйте блокировки для управления мьютексами; это сэкономит вам и вашим пользователям массу усилий.

См. также:

- Standard: 30.4 Mutual exclusion [thread.mutex]
- H. Hinnant, L. Crowl, B. Dawes, A. Williams, J. Garland, et al.: Multi-threading Library for Standard C++ (Revision 1)
- ???

Блокировки

Lock – это объект, который содержит ссылку на мьютекс и вызывает метод **unlock()** мьютекса в своем деструкторе (например, при выходе из области видимости). Поток может использовать блокировку (lock) для управления владением мьютексом потокобезопасным образом. Другими словами блокировка реализует идиому Захвата ресурсов во время инициализации (Resource Acquisition Is Initialization) для взаимоисключающего доступа. Например:

```
std::mutex m;
int sh; // разделяемые данные
// ...
void f()
{
        // ...
        std::unique_lock lck(m);
        // работа с разделяемыми данными:
        sh+=1;
}
```

Блокировка может быть перемещена (цель блокировки заключается в предоставлении локального владения нелокального ресурса), но не скопирована (какая из копий будет владеть ресурсом/мьютексом?).

Это простой способ использования блокировки с помощью **unique_lock** обладает теми же возможностями, что и мьютекс, но делает это безопасным образом. Например, мы можем использовать блокировку для попытки захвата блокировки:

```
std::mutex m;
int sh; // разделяемые данные
// ...
void f()
{
        // ...
        // создаем блокировку, но не захватываем мьютекс
        std::unique_lock lck(m,std::defer_lock);
        // ...
        if (lck.try_lock()) {
                // работа с разделяемыми ресурсами:
                sh+=1;
        }
        else {
                // возможно делаем что-то еще
        }
}
```

Аналогично, **unique_lock** поддерживает методы **try_lock_for()** и **try_lock_until()**. Использование блокировок по сравнению с ручным использованием мьютекса дает безопасность с точки зрения исключений, и не дает забыть вызвать метод **unlock()**. В параллельном программировании нам пригодится любая помощь.

А что если нам нужны два ресурса, представленные двумя мьютексами? Простой способ заключается в захвате двух мьютексов по очереди:

```
std::mutex m1;
std::mutex m2;
int sh1;        // разделяемые данные
int sh2
// ...
void f()
{
        // ...
        std::unique_lock lck1(m1);
        std::unique_lock lck2(m2);
        // работа с разделяемыми ресурсами:
        sh1+=sh2;
}
```

Этот код может привести к серьезной проблеме, если другой поток попытается захватить **m1** и **m2** в противоположном порядке, тогда каждый из них будет держать блокировку, необходимую другому потоку для продолжения своей работы, в результате они будут ожидать друг друга вечно (это называется взаимоблокировкой (deadlock)). При большом количестве блокировок в системе это становится реальной опасностью. Именно поэтому стандартные блокировки предоставляют две функции для безопасной попытки захвата двух или более блокировок:

```
void f()
{
        // ...
        // создаем блокировки, но пока не захватываем мьютексы
        std::unique_lock lck1(m1,std::defer_lock);
        std::unique_lock lck2(m2,std::defer_lock);
        std::unique_lock lck3(m3,std::defer_lock);
        lock(lck1,lck2,lck3);
        // работа с разделяемыми данными
}
```

Очевидно, реализация **lock()** должна быть тщательно продумана, чтобы избежать взаимоблокировок. По сути, это эквивалентно использованию нескольких вызовов **try_lock()**. Если во время **lock()** не удастся захватить ни одной блокировки, то будет сгенерировано исключение. На самом деле, метод **lock()** может принимать любые аргументы, которые содержат методы **lock()**, **try_lock()** и **unlock()** (как, например, класс **mutex**), поэтому мы не можем точно говорить о том, какие исключения может генерировать метод **lock()**; это зависит от его аргументов.

Если вы предпочитаете использовать **try_lock()**, то вам может помочь метод, эквивалентный методу **lock()**:

```
void f()
{
        // ...
        // создаем блокировки, но пока не захватываем мьютексы
        std::unique_lock lck1(m1,std::defer_lock);
        std::unique_lock lck2(m2,std::defer_lock);
        std::unique_lock lck3(m3,std::defer_lock);
        int x;
        // добро пожаловать в мир C
        if ((x = try_lock(lck1,lck2,lck3))==-1) {
                // работаем с разделяемыми данными
        }
        else {
```

```
                        // x содержит индекс мьютекса, который не удалось
захватить
                        // например, если lck2.try_lock() завершится неудачно, то
x==1
                }
        }
```

См. также:

- Standard: 30.4.3 Locks [thread.lock]
- ???

Условные переменные

Условные переменные (conditional variables) являются примитивами синхронизации, используемыми для блокировки потока до получения уведомления от другого потока о выполнении некоторого условия или же до достижения определенного системного времени.

Простите, но у меня пока не было времени написать этот раздел. Вернитесь к нему позже.

См. также:

- Standard: 30.5 Condition variables [thread.condition]
- ???

Работа со временем

Нам часто приходится иметь дело со временем или выполнять некоторые операции, в зависимости от времени. Например, примитивы синхронизации из стандартной библиотеки – мьютексы и блокировки, предоставляют возможность потоку ожидать некоторый период времени (**duration**) или ожидать достижения определенного времени (**time_point**).

Для получения текущего времени (в виде **time_point**) можно вызвать метод **now()** для одних из трех часов (clocks): **system_clock**, **monotonic_clock**, **high_resolution_clock**. Например:

```
monotonic_clock::time_point t = monotonic_clock::now();
// некоторые операции
monotonic_clock::duration d = monotonic_clock::now() - t;
// операция заняла d времени
```

clock возвращает **time_point**, а **duration** – это разница двух экземпляров **time_point** для одного и того же объекта **clock**. И, как обычно, для простоты вы можете воспользоваться ключевым словом **auto**:

```
auto t = monotonic_clock::now();
// некоторые операции
auto d = monotonic_clock::now() - t;
// операция заняла d времени
```

Утилиты по работе со временем предназначены для эффективной реализации низкоуровневых возможностей; они не предоставляют удобных возможностей для работы с вашим социальным календарем. На самом деле, эти возможности разрабатывались с учетом потребностей физики высоких энергий. Чтобы иметь возможность выразить время в любых градациях (будь то век или

пикосекунда), избежать неоднозначностей в единицах измерения, а так же ошибок при округлении, типы **duration** и **time_point** используют пакет работы с рациональными числами времени компиляции. Тип **duration** состоит из двух частей: количества «тиков» и «периода», который характеризует длительность одного тика (соответствует ли он секунде или миллисекунде?); период (period) является частью типа **duration**. Следующая таблица, взята из заголовочного файла стандартной библиотеки **<ration>**; она определяет периоды в системе СИ и может дать понятия об области применения:

```
        // синонимы для периодов в системе СИ:
        typedef   ratio<1,   1000000000000000000000000>   yocto;      //   частичная
поддержка
        typedef  ratio<1,        1000000000000000000000>  zepto;     //  частичная
поддержка
        typedef ratio<1,       1000000000000000000> atto;
        typedef ratio<1,          1000000000000000> femto;
        typedef ratio<1,             1000000000000> pico;
        typedef ratio<1,                1000000000> nano;
        typedef ratio<1,                   1000000> micro;
        typedef ratio<1,                      1000> milli;
        typedef ratio<1,                       100> centi;
        typedef ratio<1,                        10> deci;
        typedef ratio<                        10, 1> deca;
        typedef ratio<                       100, 1> hecto;
        typedef ratio<                      1000, 1> kilo;
        typedef ratio<                   1000000, 1> mega;
        typedef ratio<                1000000000, 1> giga;
        typedef ratio<             1000000000000, 1> tera;
        typedef ratio<          1000000000000000, 1> peta;
        typedef ratio<       1000000000000000000, 1> exa;
        typedef  ratio<      1000000000000000000000,  1>  zetta;     //  частичная
поддержка
        typedef   ratio<1000000000000000000000000,   1>   yotta;      //   частичная
поддержка
```

Рациональные числа времени компиляции (compile time rational numbers) предоставляют обычные арифметические операции (+, -, * и /), и операции сравнения (==, !=. <, <=, >, >=) для всех разумных комбинаций типов **duration** и **time_point** (например, вы не можете сложить два экземпляра **time_point**). В этих операциях также выполняется проверка переполнения и деления на ноль. Вам не стоит задумываться о производительности, поскольку это возможности времени компиляции. Кроме того, для типов **duration**, вы можете использовать операции ++, --, -=, *= и /=, а также операции **tp += d**, **tp -= d** для **time_point tp** и **duration d**.

Вот несколько примеров использования стандартных типов **duration**, определенных в **<chrono>**:

```
microseconds mms = 12345;
milliseconds ms = 123;
seconds s = 10;
minutes m = 30;
hours h = 34;

// явно указываем пространство имен
auto x = std::chrono::hours(3);
// предполагаем наличие необходимых директив "using"
auto x = hours(2)+minutes(35)+seconds(9);
```

Вы не можете инициализировать **duration** дробным значением. Например, не пытайтесь создать период, равный 2.5 секунды, используйте вместо этого 2500 миллисекунд. Связано это с тем, что период реализован в виде некоторого количества «тиков». Каждый тик представляет собой набор

«периодов», таких как **milli** или **kilo**. В качестве периода по умолчанию используются секунды; т.е. интервал с периодом, равным 1, «тик» будет равен одной секунде. Мы можем явно указать единицы измерения для **duration**:

```
// секунды (значение по умолчанию)
duration<long> d0 = 5;
// килосекунды!
duration<long,kilo> d1 = 99;
// d1 и d2 одинакового типа ("kilo" означает "*1000")
duration<long,ratio<1000,1>> d2 = 100;
```

Если мы захотим сделать что-то с **duration**, например, распечатать, то мы должны задать единицу измерения, **minutes** или **microseconds**. Например:

```
auto t = monotonic_clock::now();
// некоторые операции
auto d = monotonic_clock::now() - t;
cout << "something took " << duration<double>(d).count() << "seconds\n";
```

Метод **count()** возвращает количество «тиков».

См. также:

- Standard: 20.9 Time utilities [time]
- Howard E. Hinnant, Walter E. Brown, Jeff Garland, and Marc Paterno: A Foundation to Sleep On. N2661=08-0171. Including ``A Brief History of Time'' (With apologies to Stephen Hawking).

Атомарные операции

Простите, но у меня пока не было времени написать этот раздел. Возвращайтесь к этому разделу позже.

См. также:

- Standard: 29 Atomic operations library [atomics]
- ???

---

std::future и std::promise

Многопоточное программирование может быть очень *сложным*, особенно при попытках хитроумного использования потоков и блокировок. И оно может стать еще сложнее, если вам нужно использовать условные переменные и std::atomics (для lock-free программирования). C++11 предоставляет классы **future** и **promise** для получения значения из задачи, запущенной в другом потоке, а также класс **packaged_task** для запуска этих задач. Важное замечание о классах **future** и **promise** заключается в том, что они позволяют передать значение между двумя задачами без явного использования блокировки; «система» эффективным образом реализует эту передачу. Основная мысль очень проста: если задача хочет вернуть значение в породивший ее поток, то она помещает значение в **promise**. Каким-то образом, значение помещается в объект **future**, связанный с объектом **promise**. Вызывающий код (обычно это код, запустивший данную задачу) может затем прочитать это значение. Для дополнительного упрощения, см. async.

Стандарт описывает три типа future: **future** – для самых простых случаев и **shared_future** и **atomic_future** для более сложных случаев. Для начала давайте рассмотрим **future**, поскольку это самый простой тип и он делает все, что мне нужно. Если у нас есть **future<X>** с именем **f**, мы можем получить значение типа **X** путем вызова метода **get()**:

```
// ожидаем завершения выполнения, в случае необходимости
X v = f.get();
```

Если значения еще нет, то выполнение текущего потока будет заблокировано до тех пор, пока значение не будет получено. Если значение не может быть получено, то метод **get()** может сгенерировать исключение (исключение может передаваться из задачи, или генерироваться инфраструктурой).

В некоторых случаях мы не хотим ожидать получение результата, поэтому можем узнать, готов результат или нет:

```
// значение для get() готово
if (f.wait_for(0)) {
        // выполняем некоторые операции
}
else {
        // делаем что-то еще
}
```

Однако, главная задача типа **future** заключается в простом предоставлении результата методом **get()**.

Основная задача типа **promise** заключается в предоставлении возможности «поместить» (“put”) значение, которое будет затем получено с помощью метода **get()** объекта типа **future**. Имена “future” и “promise” являются историческими, так что не вините меня за них. Эти имена являются богатым источником разных каламбуров[1].

Если у вас есть объект типа **promise**, и вы хотите передать результаты типа **X** (обратно) во **future**, то вы можете сделать это двумя способами: передать значение или передать исключение.

```
try {
        X res;
        // вычисляем значение res
        p.set_value(res);
}
catch (...) {  // Ой: не могу получить res
        p.set_exception(std::current_exception());
}
```

Это все хорошо, но как мне получить пару соответствующих друг другу объектов **future**/**promise** – один объект в моем потоке, а другой – в каком-то другом? Ну, поскольку объекты **future** и **promise** могут перемещаться (но не копироваться), то решить это можно самыми разными способами. Наиболее очевидный подход заключается в следующем: при запуске задачи передать ей объект

[1] Примечание переводчика. Promise в переводе с английского означает «обещание», поэтому многие предложения имеют дополнительный оттенок, не передаваемый при переводе на русский язык. Например, следующее предложение звучит так: “If you have a **promise** and need to send a result of type X (back) to a **future**…”, что можно было бы перевести так: «Если бы вы дали **обещание** передать результаты, типа X (обратно) в **будущее**…”.

promise и оставить вызывающему коду соответствующий объект **future**, куда будет помещен результат. Использование async() является наиболее экстремальным и элегантным способом использования этого подхода.

Тип **package_task** предоставляет простой способ запуска потока для выполнения задачи. В том числе, он заботится об установке объекта **future**, связанного с соответствующим объектом **promise** и предоставляет обертку для помещения результата или исключения из задачи в **promise**. Например:

```
double comp(vector<double>& v)
{
        // упаковываем задачи:
        // (в качестве задачи мы используем стандартный
        // метод accumulate() для массива double):
        packaged_task<double(double*,double*,double)> pt0{
                std::accumulate<double*,double*,double>};
        packaged_task<double(double*,double*,double)> pt1{
                std::accumulate<double*,double*,double>};

        auto f0 = pt0.get_future();    // получаем future
        auto f1 = pt1.get_future();

        pt0(&v[0],&v[v.size()/2],0);   // запускаем потоки
        pt1(&v[v.size()/2],&v[size()],0);

        return f0.get()+f1.get();      // получаем результаты
}
```

См. также:

- Standard: 30.6 Futures [futures]
- Anthony Williams: Moving Futures - Proposed Wording for UK comments 335, 336, 337 and 338. N2888==09-0078.
- Detlef Vollmann, Howard Hinnant, and Anthony Williams An Asynchronous Future Value (revised) N2627=08-0137.
- Howard E. Hinnant: Multithreading API for C++0X - A Layered Approach. N2094=06-0164. The original proposal for a complete threading package..

---

std::async()

Метод **async()**, предназначен для простого запуска задач, является единственной возможностью, которая не утверждена в черновом варианте стандарта. Я ожидаю, что она будет принята в октябре, после переработки двух немного отличающихся предложений.

Вот пример того, как программист может подняться над всеми этими непонятными потоками и блокировками в многопоточном программировании.

```
// простой функтор аккумулятора
template<class T, class V> struct Accum  {
        T* b;
        T* e;
        V val;
        Accum(T* bb, T* ee, const V& v) : b{bb}, e{ee}, val{vv} {}
        V operator() () { return std::accumulate(b,e,val); }
};

double comp(vector<double>& v)
        // запускаем несколько задач, если v содержит
```

```
        // довольно много элементов
{
        if (v.size()<10000)
                return std::accumulate(v.begin(),v.end(),0.0);

        auto f0 {async(Accum{&v[0],&v[v.size()/4],0.0})};
        auto f1 {async(Accum{&v[v.size()/4],&v[v.size()/2],0.0})};
        auto f2 {async(Accum{&v[v.size()/2],&v[v.size()*3/4],0.0})};
        auto f3 {async(Accum{&v[v.size()*3/4],&v[v.size()],0.0})};

        return f0.get()+f1.get()+f2.get()+f3.get();
}
```

Это очень простой способ использования многопоточного программирования (обратите внимание на использования «магических чисел»), но обратите внимание на отсутствие явного использования потоков, блокировок, буферов и т.п. Тип переменных **f**x определяется типом возвращаемого значения функции стандартной библиотеки **async()**, которая возвращает future. В случае необходимости, вызов метода **get()** объекта типа **future** ожидает завершения потока. В данном случае, обязанностью метода **async()** является порождение потоков, а задачей объектов **future** является вызов **join()** для ожидания завершения соответствующих потоков. «Простота» является самым важным аспектом дизайна **async()**/**future**; тип **future** может быть использован вручную при работе с потоками, но даже и не *думайте* использовать **async()** для запуска задач, выполняющих ввод/вывод, использующих мьютексы или каким-то другим способом взаимодействующих с другими задачами. Идея создания **async()** аналогична идее, лежащей в основе range-for оператора: предоставить простой механизм обработки самых простых и довольно распространенных случаев, и оставить базовые механизмы для более сложных случаев.

Вызывающий код может указать, чтобы метод **async()** запускал новый поток, использовал любой поток, кроме вызывающего или же запускал задачу в новом потоке, только если **async()** «считает», что это того стоит. Последний вариант является самым простым с точки зрения пользователя и потенциально самым эффективным ((только) для *простых* задач).

См. также:

- Standard: ???
- Lawrence Crowl: An Asynchronous Call for C++. N2889 = 09-0079.
- Herb Sutter : A simple async() N2901 = 09-0091 .

Завершение процесса

Простите, но у меня не было времени для написания этого раздела. Вернитесь к этому разделу позже.

См. также:

- Abandoning a process

Генерация случайных чисел

Случайные числа полезны во многих сферах, таких как тестирование, игры, симуляция и безопасность. Многообразие приложений отражается в многообразии генераторов случайных чисел, представленных в стандартной библиотеке. Генератор случайных чисел состоит из двух частей:

движка (engine), генерирующего последовательность случайных или псевдо-случайных значений и *распределения* (distribution), которое находит соответствие этих значений математическому распределению в некотором диапазоне. Примерами распределений являются **uniform_int_distribution** (в котором вероятность генерации целочисленного значения является одинаковой) и **normal_distribution** (распределение в форме «колокола»); каждое для определенного диапазона. Например:

```
// распределение, соответствующее распределению целых
// чисел в диапазоне 1..6
uniform_int_distribution<int> one_to_six {1,6};
// движок по умолчанию
default_random_engine re {};
```

Для получения случайного значения нужно вызвать метод распределения с указанием движка:

```
// x является значением в диапазоне [1:6]
int x = one_to_six(re);
```

Постоянная передача движка может оказаться утомительной, поэтому мы можем «связать» (bind) этот аргумент для получения функтора, вызываемого без аргументов:

```
// создаем генератор
auto dice {bind(one_to_six,re)};

// бросаем кости: x является значением в диапазоне [1:6]
int x = dice();
```

Благодаря бескомпромиссному вниманию к обобщению и производительности, кое-кто считает компонент генерации случайных чисел стандартной библиотеки «тем, чем хочет стать любая библиотека по работе со случайными числами, когда вырастет». Однако ее сложно назвать «простой для новичков» (novice friendly). Я никогда не видел, чтобы интерфейс генерации случайных чисел был узким местом в плане производительности, но я никогда не обучал новичков (с любым опытом), без того чтобы у меня не возникла необходимость в простом генераторе случайных чисел. Этого было бы достаточно:

```
// генерируем случайное значение равномерного распределения
// в диапазоне [low:high]
int rand_int(int low, high);
```

Но как нам это сделать? Внутри метода **rand_int()** нам нужно использовать код, аналогичный в примере с игральными костями:

```
int rand_int(int low, int high)
{
        static default_random_engine re {};
        using Dist = uniform_int_distribution<int>;
        static Dist uid {};
        return uid(re, Dist::param_type{low,high});
}
```

Эта реализация все еще требует «экспертного уровня», но *использование* метода **rand_int()** будет доступно студенту на первой неделе обучения языку C++.

Просто, ради нетривиального примера, вот пример кода, который генерирует и печатает значения, распределенные по нормальному закону распределения:

```
default_random_engine re;   // движок по умолчанию
normal_distribution<int> nd(31 /* мат. ожидание */,8 /* СКО */);

auto norm = std::bind(nd, re);

vector<int> mn(64);

int main()
{
        for (int i = 0; i<1200; ++i) ++mn[round(norm())]; // генерация

        for (int i = 0; i<mn.size(); ++i) {
                cout << i << '\t';
                for (int j=0; j<mn[i]; ++j) cout << '*';
                cout << '\n';
        }
}
```

Результат выполнения:

```
0
1
2
3
4       *
5
6
7
8
9       *
10      ***
11      ***
12      ***
13      *****
14      *******
15      ****
16      **********
17      **********
18      ****************
19      ******************
20      ******************
21      *************************
22      ********************************
23      ******************************************
24      ****************************************
25      ***************************************
26      *********************************************
27      ********************************************************
28      ************************************************
29      ************************************************************
30      *********************************************
31      ************************************************************************
32      *********************************************
33      ************************************************************
34      ****************************************************************
35      ************************************
36      *********************************************
37      *********************************************
38      *******************************************
39      ******************************
40      ******************************************
```

```
41      **********************
42      ************************
43      ******************************
44      ****************
45      *************
46      *********
47      ********
48      *****
49      *****
50      ****
51      ***
52      ***
53      **
54      *
55      *
56
57      *
58
59
60
61
62
63
```

См. также:

- Standard 26.5: Random number generation

Регулярные выражения

- 28 Regular expressions library

Простите, но у меня не было времени на написание этого раздела. Вернитесь к этому разделу позже.

См. также:

- Standard: ???
- ???

Концепты

«Концепты» (concepts) – это механизм, описывающий требования для типов, комбинаций типов или комбинаций типов и целых чисел. Эта возможность особенно полезна для ранней проверки использования шаблонов. И наоборот, она также помогает находить ошибки в теле шаблона на ранней стадии. Давайте рассмотрим алгоритм **fill** из стандартной библиотеки:

```
// типы типов
template<ForwardIterator Iter, class V>
        // взаимоотношения между аргументами типов
        requires Assignable<Iter::value_type,V>
// лишь объявление, а не определение
void fill(Iter first, Iter last, const V& v);

// Iter имеет тип int; ошибка: int не является ForwardIterator-ом
// int не содержит *
fill(0, 9, 9.9);

// Iter имеет тип int; ok: int* является ForwardIterator-ом
```

```
fill(&v[0], &v[9], 9.9);
```

Обратите внимание, что мы только объявили метод **fill()**; мы не определяли его реализацию. С другой стороны, мы точно определили, что метод **fill()** требует от аргументов:

- Аргументы **first** и **last** должны быть типом **ForwardIterator** (и они должны быть одного типа).
- Третий аргумент **v** должен быть типом, который можно присвоить типу **value_type** типа **ForwardIterator**.

Конечно, мы все это знаем, ведь мы читали стандарт. Однако, компиляторы не умеют читать документы с требованиями, поэтому мы должны сказать ему в коде, в виде концептов **ForwardIterator** и **Assignable**. В результате ошибки использования метода **fill()** могут быть пойманы в месте использования, и текст ошибок может быть значительно яснее. Теперь у компилятора есть информация о намерениях программиста, что позволяет обеспечить хороший уровень проверок и диагностических сообщений.

Концепты также помогают в реализации шаблонов. Давайте рассмотрим пример:

```
template<ForwardIterator Iter, class V>
        requires Assignable<Iter::value_type,V>
void fill(Iter first, Iter last, const V& v)
{
        while (first!=last) {
                *first = v;
                // ошибка: + не определен для Forward_iterator
                // (нужно использовать ++first)
                first=first+1;
        }
}
```

Эта ошибка будет отловлена сразу же, устраняя необходимость трудоемкого тестирования (хотя и не избавляет от тестирования полностью).

С возможностью классифицировать и различать разные типы типов, мы можем перегружать методы, основываясь на виде передаваемых типов. Например:

```
// стандартный алгоритм сортировки, на основе
// итераторов (с концептами):
template<Random_access_iterator Iter>
        requires Comparable<Iter::value_type>
// используем обычную реализацию
void sort(Iter first, Iter last);

// сортировка на основе контейнера:
template<Container Cont>
        requires Comparable<Cont::value_type>
void sort(Cont& c)
{
        // просто вызываем версию на основе итераторов
        sort(c.begin(),c.end());
}

void f(vector<int>& v)
{
        sort(v.begin(), v.end());    // один способ
        sort(v);                     // еще один способ
        // ...
}
```

Вы можете определить свои собственные концепты, но для начала, в стандартной библиотеке определен большой набор полезных концептов, таких как **ForwardIterator**, **Collable**, **LessThanComparable** и **Regular**.

Обратите внимание, стандартные библиотеки C++11 предусматривают использование концептов.

См. также:

- the C++ draft 14.10 Concepts
- [N2617=08-0127] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek: Proposed Wording for Concepts (Revision 5) (Final proposal).
- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++. OOPSLA'06, October 2006.

Карты концептов

int* является **ForwardIterator**; мы сказали об этом при обсуждении концептов, и в стандартной библиотеке это всегда было именно так; даже в первой версии STL указатели использовались в качестве итераторов. Однако, мы также говорили о члене **value_type** типа **ForwardIterator**. Но **int*** не содержит члена с именем **value_type**; на самом деле, он вообще не содержит никаких членов. Так как **int*** может быть **ForwardIterator**? Поскольку мы сами это указали. С помощью **concept_map** мы можем сказать, что при использовании **T***, где требуется **ForwardIterator**, мы будем рассматривать **T** в качестве **value_type**:

```
template<Value_type T>
// value_type типа T* - это T
concept_map ForwardIterator<T*> {
        typedef T value_type;
};
```

concept_map позволяет указать способ представления нашего типа, предотвращая от модификации существующий тип или от создания нового типа в качестве обертки. «Карты концептов» (concept maps) являются гибким и обобщенным механизмом для адаптации независимо разработанных программ для совместного использования.

См. также:

- the C++ draft 14.10.2 Concept maps
- [N2617=08-0127] Douglas Gregor, Bjarne Stroustrup, James Widman, and Jeremy Siek: Proposed Wording for Concepts (Revision 5) (Final proposal).
- Douglas Gregor, Jaakko Jarvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, Andrew Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++. OOPSLA'06, October 2006.

Аксиомы

Аксиома (axiom) – это набор предикатов, определяющих семантику концепта. Главным сценарием использование аксиом являются внешние инструменты (например, действия, нетипичные для компилятора), такие как, инструменты для предметно-ориентированных оптимизаций (domain-specific optimizations) (языки для описания программных изменений стали важным поводом

появления аксиом). Аксиомы также могут быть полезны для некоторых оптимизаций (выполняемых компилятором и обычными оптимизаторами), но компиляторы *не должны* обращать внимания на пользовательские аксиомы; их работа основана на семантиках, определенных в стандарте.

Аксиомы перечисляют пару вычислений, которые могут трактоваться как эквивалентные. Например:

```
concept Semigroup<typename Op, typename T> : CopyConstructible<T> {
        T operator()(Op, T, T);
        axiom Associativity(Op op, T x, T y, T z) {
                // можно предположить, что оператор типа T является ассоциативным
                op(x, op(y, z)) <=> op(op(x, y), z);
        }
}

// monoid – это полугруппа с единичным элементом
concept Monoid<typename Op, typename T> : Semigroup<Op, T> {
        T identity_element(Op);
        axiom Identity(Op op, T x) {
                op(x, identity_element(op)) <=> x;
                op(identity_element(op), x) <=> x;
        }
}
```

<=> - это оператор эквивалентности, используемый только в аксиомах. Обратите внимание, что вы не можете (в общем случае) доказать аксиому; мы используем аксиомы для указания того, что мы не можем доказать, но что программист может назвать приемлемым допущением. Обратите внимание, что обе стороны оператора эквивалентности могут быть некорректными для некоторых значений, например использование NaN (not a number) для типов с плавающей запятой: если обе стороны эквивалентности используют NaN, то обе стороны (явно) некорректны и эквивалентны (независимо от того, что говорится в аксиоме), но если NaN используется только с одной стороны, то существует возможность получить выгоды от использования аксиомы.

Аксиома это последовательность выражений эквивалентности (<=>) и условных выражений (вида “if (something) тогда мы можем предполагать следующую эквивалентность”):

```
// в концепте TotalOrder:
axiom Transitivity(Op op, T x, T y, T z)
{
        // условная эквивалентность
        if (op(x, y) && op(y, z)) op(x, z) <=> true;
}
```

См. также:

- the C++ draft 14.9.1.4 Axioms
- ???.