Введение

Если вы — опытный программист на языке программирования C++, как, например, я, то, наверное, первое, о чем вы подумали в связи с C++11, — "Да, да, вот и он — тот же C++, только немного улучшенный". Но познакомившись с ним поближе, вы, скорее всего, были удивлены количеством изменений. Объявления `auto`, циклы `for` для диапазонов, лямбда-выражения и rvalue-ссылки изменили лицо C++, — и это не говоря о новых возможностях параллельности. Произошли и идиоматические изменения. `0` и `typedef` уступили место `nullptr` и объявлениям псевдонимов. Перечисления получили области видимости. Интеллектуальные указатели стали предпочтительнее встроенных; перемещение объектов обычно предпочтительнее их копирования.

Даже без упоминания C++14 в C++11 есть что поизучать.

Что еще более важно, нужно очень многое изучить, чтобы использовать новые возможности *эффективно*. Если вам нужна базовая информация о "современных" возможностях C++, то ее можно найти в избытке. Но если вы ищете руководство о том, как использовать эти возможности для создания правильного, эффективного, сопровождаемого и переносимого программного обеспечения, поиск становится более сложным. Вот здесь вам и пригодится данная книга. Она посвящена не описанию возможностей C++11 и C++14, а их эффективному применению.

Информация в книге разбита на отдельные разделы, посвященные тем или иным рекомендациям. Вы хотите разобраться в разных видах вывода типов? Или хотите узнать, когда следует (а когда нет) использовать объявление `auto`? Вас интересует, почему функция-член, объявленная как `const`, должна быть безопасна с точки зрения потоков, как реализовать идиому Pimpl с использованием `std::unique_ptr`, почему следует избегать режима захвата по умолчанию в лямбда-выражениях или в чем различие между `std::atomic` и `volatile`? Ответы на эти вопросы вы найдете в книге. Более того, эти ответы не зависят от платформы и соответствуют стандарту. Это книга о *переносимом* C++.

Разделы книги представляют собой рекомендации, а не жесткие правила, поскольку рекомендации имеют исключения. Наиболее важной частью каждого раздела является не предлагаемая в нем рекомендация, а ее обоснование. Прочитав раздел, вы сможете сами определить, оправдывают ли обстоятельства вашего конкретного проекта отход от данной рекомендации. Истинная цель книги не в том, чтобы рассказать вам, как надо поступать или как поступать не надо, а в том, чтобы обеспечить вас более глубоким пониманием, как та или иная концепция работает в C++11 и C++14.

Терминология и соглашения

Чтобы мы правильно понимали друг друга, важно согласовать используемую терминологию, начиная, как ни странно это звучит, с термина "C++". Есть четыре официальные версии C++, и каждая именуется с использованием года принятия соответствующего стандарта ISO: *C++98*, *C++03*, *C++11* и *C++14*. C++98 и C++03 отличаются один от другого только техническими деталями, так что в этой книге обе версии я называю как C++98. Говоря о C++11, я подразумеваю и C++11, и C++14, поскольку C++14 является надмножеством C++11. Когда я пишу "C++14", я имею в виду конкретно C++14. А если я просто упоминаю C++, я делаю утверждение, которое относится ко всем версиям языка.

Использованный термин	Подразумеваемая версия
C++	Все
C++98	C++98 и C++03
C++11	C++11 и C++14
C++14	C++14

В результате я мог бы сказать, что в C++ придается большое значение эффективности (справедливо для всех версий), в C++98 отсутствует поддержка параллелизма (справедливо только для C++98 и C++03), C++11 поддерживает лямбда-выражения (справедливо для C++11 и C++14) и C++14 предлагает обобщенный вывод возвращаемого типа функции (справедливо только для C++14).

Наиболее важной особенностью C++11, вероятно, является семантика перемещения, а основой семантики перемещения является отличие *rvalue*-выражений от *lvalue*-выражений. Поэтому rvalue указывают объекты, которые могут быть перемещены, в то время как lvalue в общем случае перемещены быть не могут. Концептуально (хотя и не всегда на практике), rvalue соответствуют временным объектам, возвращаемым из функций, в то время как lvalue соответствуют объектам, на которые вы можете ссылаться по имени, следуя указателю или lvalue-ссылке.

Полезной эвристикой для выяснения, является ли выражение lvalue, является ответ на вопрос, можно ли получить его адрес. Если можно, то обычно это lvalue. Если нет, это обычно rvalue. Приятной особенностью этой эвристики является то, что она помогает помнить, что тип выражения не зависит от того, является ли оно lvalue или rvalue. Иначе говоря, для данного типа `T` можно иметь как lvalue типа `T`, так и rvalue типа `T`. Особенно важно помнить это, когда мы имеем дело с параметром rvalue ссылочного типа, поскольку сам по себе параметр является lvalue:

```
class Widget {
public:
    Widget(Widget&& rhs); // rhs является lvalue, хотя
    …                     // и имеет ссылочный тип rvalue
};
```

Здесь совершенно корректным является взятие адреса `rhs` в перемещающем конструкторе `Widget`, так что `rhs` представляет собой lvalue, несмотря на то что его тип — ссылка rvalue. (По сходным причинам все параметры являются lvalue.)

Этот фрагмент кода демонстрирует несколько соглашений, которым я обычно следую.

- Имя класса — `Widget`. Я использую слово `Widget`, когда хочу сослаться на произвольный пользовательский тип. Если только мне не надо показать конкретные детали класса, я использую имя `Widget`, не объявляя его.
- Я использую имя параметра `rhs` ("right-hand side", правая сторона). Это предпочитаемое мною имя параметра для *операций перемещения* (например, перемещающего конструктора и оператора перемещающего присваивания) и *операций копирования* (например, копирующего конструктора и оператора копирующего присваивания). Я также использую его в качестве правого параметра бинарных операторов:

  ```
  Matrix operator+(const Matrix& lhs, const Matrix& rhs);
  ```

  Я надеюсь, для вас не станет сюрпризом, что `lhs` означает "left-hand side" (левая сторона).
- Я использую специальное форматирование для частей кода или частей комментариев, чтобы привлечь к ним ваше внимание. В перемещающем конструкторе `Widget` выше я подчеркнул объявление `rhs` и часть комментария, указывающего, что `rhs` представляет собой lvalue. Выделенный код сам по себе не является ни плохим, ни хорошим. Это просто код, на который вы должны обратить внимание.
- Я использую "...", чтобы указать "здесь находится прочий код". Такое "узкое" троеточие отличается от широкого "`...`", используемого в исходных текстах шаблонов с переменным количеством параметров в C++11. Это кажется запутанным, но на самом деле это не так. Вот пример.

  ```
  template<typename... Ts>                // Эти троеточия
  void processVals(const Ts&... params)   // в исходном
  {                                       // тексте C++
      …                                   // Это троеточие озна-
                                          // чает какой-то код
  }
  ```

  Объявление `processVals` показывает, что я использую ключевое слово `typename` при объявлении параметров типов в шаблонах, но это просто мое личное предпочтение; вместо него можно использовать ключевое слово `class`. В тех случаях, когда я показываю код, взятый из стандарта C++, я объявляю параметры типа с использованием ключевого слова `class`, поскольку так делает стандарт.

Когда объект инициализирован другим объектом того же типа, новый объект является *копией* инициализирующего объекта, даже если копия создается с помощью перемещающего конструктора. К сожалению, в C++ нет никакой терминологии, которая позволяла бы различать объекты, созданные с помощью копирующих и перемещающих конструкторов:

```
void someFunc(Widget w);  // Параметр w функции someFunc
                          // передается по значению

Widget wid;               // wid — объект класса Widget

someFunc(wid);            // В этом вызове someFunc w
                          // является копией wid, созданной
                          // копирующим конструктором

someFunc(std::move(wid)); // В этом вызове SomeFunc w
                          // является копией wid, созданной
                          // перемещающим конструктором
```

Копии rvalue в общем случае конструируются перемещением, в то время как копии lvalue обычно конструируются копированием. Следствием является то, что если вы знаете только то, что объект является копией другого объекта, то невозможно сказать, насколько дорогостоящим является создание копии. В приведенном выше коде, например, нет возможности сказать, насколько дорогостоящим является создание параметра `w`, без знания того, какое значение передано функции `someFunc` — rvalue или lvalue. (Вы также должны знать стоимости перемещения и копирования `Widget`.)

В вызове функции выражения, переданные в источнике вызова, являются *аргументами* функции. Эти аргументы используются для инициализации *параметров* функции. В первом вызове `someFunc`, показанном выше, аргументом является `wid`. Во втором вызове аргументом является `std::move(wid)`. В обоих вызовах параметром является `w`. Разница между аргументами и параметрами важна, поскольку параметры являются lvalue, но аргументы, которыми они инициализируются, могут быть как rvalue, так и lvalue. Это особенно актуально во время *прямой передачи*, при которой аргумент, переданный функции, передается другой функции так, что при этом сохраняется его "правосторонность" или "левосторонность". (Прямая передача подробно рассматривается в разделе 5.8.)

Хорошо спроектированные функции *безопасны с точки зрения исключений*, что означает, что они обеспечивают как минимум *базовую гарантию*, т.е. гарантируют, что, даже если будет сгенерировано исключение, инварианты программы останутся нетронутыми (т.е. не будут повреждены структуры данных) и не будет никаких утечек ресурсов. Функции, обеспечивающие *строгую гарантию*, гарантируют, что, даже если будет сгенерировано исключение, состояние программы останется тем же, что и до вызова функции.

Говоря о *функциональном объекте*, я обычно имею в виду объект типа, поддерживающего функцию-член `operator()`. Другими словами, это объект, действующий, как функция. Иногда я использую термин в несколько более общем смысле для обозначения чего угодно, что может быть вызвано с использованием синтаксиса вызова функции, не являющейся членом (т.е. `function Name(arguments)`). Это более широкое определение охватывает не только объекты, поддерживающие `operator()`, но и функции и указатели на функции в стиле C. (Более узкое определение происходит из C++98, более широкое — из C++11.) Дальнейшее обобщение путем добавления указателей на функции-члены дает то, что известно как *вызываемый объект* (callable object). Вообще говоря,

можно игнорировать эти тонкие отличия и просто рассматривать функциональные и вызываемые объекты как сущности в C++, которые могут быть вызваны с помощью некоторой разновидности синтаксиса вызова функции.

Функциональные объекты, создаваемые с помощью лямбда-выражений, известны как *замыкания* (closures). Различать лямбда-выражения и замыкания, ими создаваемые, приходится редко, так что я зачастую говорю о них обоих как о *лямбдах* (lambda). Точно так же я редко различаю *шаблоны функций* (function templates) (т.е. шаблоны, которые генерируют функции) и *шаблонные функции* (template functions) (т.е. функции, сгенерированные из шаблонов функций). То же самое относится к *шаблонам классов* и *шаблонным классам*.

Многие сущности в C++ могут быть как объявлены, так и определены. *Объявления* вводят имена и типы, не детализируя информацию о них, такую как их местоположение в памяти или реализация:

```
extern int x;               // Объявление объекта

class Widget;               // Объявление класса

bool func(const Widget& w); // Объявление функции

enum class Color;           // Объявление перечисления
                            // с областью видимости
                            // (см. раздел 3.4)
```

Определение предоставляет информацию о расположении в памяти и деталях реализации:

```
int x;                        // Определение объекта

class Widget {                // Определение класса
    …
};
bool func(const Widget& w)
    { return w.size() < 10; } // Определение функции

enum class Color
{ Yellow, Red, Blue };        // Определение перечисления
```

Определение можно квалифицировать и как объявление, так что, если только то, что нечто представляет собой определение, не является действительно важным, я предпочитаю использовать термин “объявление”.

Сигнатуру функции я определяю как часть ее объявления, определяющую типы параметров и возвращаемый тип. Имена функции и параметров значения не имеют. В приведенном выше примере сигнатура функции `func` представляет собой `bool(const Widget&)`. Исключаются элементы объявления функции, отличные от типов ее параметров и возвращаемого типа (например, `noexcept` или `constexpr`, если таковые имеются). (Модификаторы `noexcept` и `constexpr` описаны в разделах 3.8

и 3.9.) Официальное определение термина “сигнатура” несколько отличается от моего, но в данной книге мое определение оказывается более полезным. (Официальное определение иногда опускает возвращаемый тип.)

Новый стандарт C++ в общем случае сохраняет корректность кода, написанного для более старого стандарта, но иногда Комитет по стандартизации *не рекомендует* применять те или иные возможности. Такие возможности находятся в “камере смертников” стандартизации и могут быть убраны из новых версий стандарта. Компиляторы могут предупреждать об использовании программистом таких устаревших возможностей (но могут и не делать этого), но в любом случае их следует избегать. Они могут не только привести в будущем к головной боли при переносе, но и в общем случае они ниже по качеству, чем возможности, заменившие их. Например, `std::auto_ptr` не рекомендуется к применению в C++11, поскольку `std::unique_ptr` выполняет ту же работу, но лучше.

Иногда стандарт гласит, что результатом операции является *неопределенное поведение* (undefined behavior). Это означает, что поведение времени выполнения непредсказуемо, и от такой непредсказуемости, само собой разумеется, следует держаться подальше. Примеры действий с неопределенным поведением включают использование квадратных скобок (`[]`) для индексации за границами `std::vector`, разыменование неинициализированного итератора или гонку данных (т.е. когда два или более потоков, как минимум один из которых выполняет запись, одновременно обращаются к одному и тому же месту в памяти).

Я называю встроенный указатель, такой как возвращаемый оператором `new`, *обычным указателем* (raw pointer). Противоположностью обычному указателю является *интеллектуальный указатель* (smart pointer). Интеллектуальные указатели обычно перегружают операторы разыменования указателей (`operator->` и `operator*`), хотя в разделе 4.3 поясняется, что интеллектуальный указатель `std::weak_ptr` является исключением.

Замечания и предложения

Я сделал все возможное, чтобы книга содержала только ясную, точную, полезную информацию, но наверняка есть способы сделать ее еще лучшей. Если вы найдете в книге ошибки любого рода (технические, разъяснительные, грамматические, типографские и т.д.) или если у вас есть предложения о том, как можно улучшить книгу, пожалуйста, напишите мне по адресу `emc++@aristeia.com`. В новых изданиях книги ваши замечания и предложения обязательно будут учтены.

Список исправлений обнаруженных ошибок можно найти по адресу `http://www.aristeia.com/BookErrata/emc++-errata.html`.

От редакции

Редакция выражает признательность профессору университета Иннополис Е. Зуеву за обсуждения и советы при работе над переводом данной книги.

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@williamspublishing.com`
WWW: `http://www.williamspublishing.com`

Наши почтовые адреса:

в России: 127055, Москва, ул. Лесная, д. 43, стр. 1
в Украине: 03150, Киев, а/я 152