Доклад на тему «Стандарт С++11**»**

**Содержание**

* 1 Предполагаемые изменения стандарта
* 2 Расширение ядра языка
  + 2.1Повышение производительности
    - 2.1.1Ссылки на временные объекты и семантика перемещения
    - 2.1.2Обобщённые константные выражения
    - 2.1.3Изменения в определении простых данных
  + 2.2Ускорение компиляции
    - 2.2.1Внешние шаблоны
  + 2.3Повышение удобства использования
    - 2.3.1Списки инициализации
    - 2.3.2Универсальная инициализация
    - 2.3.3Вывод типов
    - 2.3.4For-цикл по коллекции
    - 2.3.5Лямбда-функции и выражения
    - 2.3.6Альтернативный синтаксис функций
    - 2.3.7Улучшение конструкторов объектов
    - 2.3.8Явное замещение виртуальных функций и финальность
    - 2.3.9Константа нулевого указателя
    - 2.3.10Перечисления со строгой типизацией
    - 2.3.11Угловые скобки
    - 2.3.12Операторы явного преобразования
    - 2.3.13Шаблонный typedef
    - 2.3.14Снятие ограничений с union
  + 2.4Расширение функциональности
    - 2.4.1Шаблоны с переменным числом аргументов
    - 2.4.2Новые строковые литералы
    - 2.4.3Пользовательские литералы
    - 2.4.4Многопоточная модель памяти
    - 2.4.5Внутрипоточное хранилище
    - 2.4.6Явное задание по умолчанию и удаление специальных методов
    - 2.4.7Тип long long int
    - 2.4.8Статическая диагностика
    - 2.4.9Работа sizeof с элементами данных в классах без создания объекта
    - 2.4.10Управление выравниванием объектов и запросы на выравнивание
* 3 Изменения стандартной библиотеки C++
  + 3.1Изменения в имеющихся компонентах
  + 3.2Управление потоками
  + 3.3Хеш-таблицы
  + 3.4Регулярные выражения
  + 3.5Расширяемые классы генерации случайных чисел

1. Предполагаемые изменения стандарта

При разработке каждого раздела будущего стандарта комитет использовал ряд правил:

* поддержка стабильности языка и обеспечение совместимости с C++98 и, по возможности, с Си;
* предпочитается введение новых возможностей через стандартную библиотеку, а не через ядро языка;
* предпочитаются изменения, которые улучшают технику программирования;
* совершенствовать C++ с точки зрения системного и библиотечного дизайна, вместо введения новых возможностей, полезных для отдельных приложений;
* увеличивать типобезопасность для обеспечения безопасной альтернативы для нынешних опасных подходов;
* увеличивать производительность и возможности работать напрямую с аппаратной частью;
* обеспечивать решение реальных, широко распространённых проблем;
* реализовать принцип «не платить за то, что не используешь»;
* сделать C++ проще для изучения без удаления возможностей, используемых программистами-экспертами.

Уделяется внимание новичкам, которые всегда будут составлять большую часть программистов. Многие новички не стремятся углублять уровень владения C++, ограничиваясь его использованием при работе над узкими специфичными задачами[7]. Кроме того, учитывая универсальность C++ и обширность его использования (включая как разнообразие приложений, так и стилей программирования), даже профессионалы могут оказаться новичками при использовании новых парадигм программирования.

2 Расширение ядра языка

Первоочередная задача комитета — развитие ядра языка C++. Ядро значительно усовершенствовано, добавлена поддержка многопоточности, улучшена поддержка обобщённого программирования, унификация инициализации и проведены работы по повышению его производительности.

Для удобства возможности ядра и его изменения разделены на три основные части: повышение производительности, повышение удобства и новая функциональность. Отдельные элементы могут относиться к нескольким группам, но описываться будут только в одной — наиболее подходящей.

2.1Повышение производительности

Эти компоненты языка введены для уменьшения затрат памяти или увеличения производительности.

2.1.1Ссылки на временные объекты и семантика перемещения

По стандарту C++ временный объект, появившийся в результате вычисления выражения, можно передавать в функции, но только по константной ссылке (const &). Функция не в состоянии определить, можно ли рассматривать переданный объект как временный и допускающий модификацию (константный объект, который тоже может быть передан по такой ссылке, нельзя модифицировать (легально)). Это не проблема для простейших структур наподобие complex, но для сложных типов, требующих выделения-освобождения памяти, уничтожение временного объекта и создание постоянного может отнимать много времени, в то время как можно было бы просто напрямую передать указатели.

В C++11 появился новый тип ссылки — *rvalue*-ссылка (англ. *rvalue reference*). Его объявление следующее: type &&. Новые правила разрешения перегрузки позволяют использовать разные перегруженные функции для неконстантных временных объектов, обозначаемых посредством rvalues, и для всех остальных объектов. Данное нововведение позволяет реализовывать так называемую семантику перемещения.

2.1.2Обобщённые константные выражения

В C++ всегда присутствовала концепция константных выражений. Так, выражения типа *3+4* всегда возвращали одни и те же результаты, не вызывая никаких побочных эффектов. Сами по себе константные выражения предоставляют компиляторам C++ удобные возможности по оптимизации результата компиляции. Компиляторы вычисляют результаты таких выражений только на этапе компиляции и сохраняют уже вычисленные результаты в программе. Таким образом, подобные выражения вычисляются только раз. Также существует несколько случаев, в которых стандарт языка требует использования константных выражений. Такими случаями, например, могут быть определения внешних массивов или значения перечислений (enum). C++11 вводит ключевое слово constexpr, которое позволяет пользователю гарантировать, что или функция или конструктор объекта возвращает константу времени компиляции. Такое ключевое слово позволяет компилятору понять и удостовериться в том, что GiveFive возвращает константу.

Использование constexpr порождает очень жёсткие ограничения на действия функции:

1. такая функция должна возвращать значение;
2. тело функции должно быть вида return *выражение*;
3. *выражение* должно состоять из констант и/или вызовов других constexpr-функций;
4. функция, обозначенная constexpr, не может использоваться до определения в текущей единице компиляции.

В предыдущей версии стандарта в константных выражениях можно было использовать переменные только целого типа или типа перечисления. В C++11 это ограничение снято для переменных, перед определением которых стоит ключевое слово constexpr:

2.1.3Изменения в определении простых данных

В стандартном C++ только структуры, удовлетворяющие определённому набору правил, могут рассматриваться как тип простых данных (plain old data type или POD). Существуют веские причины ожидать расширения этих правил, с тем, чтобы большее число типов рассматривались как POD. Типы, удовлетворяющие этим правилам, могут использоваться в реализации объектного слоя, совместимого с C. Однако, в C++03 список этих правил чрезмерно строгий.

C++11 ослабит несколько правил, касающихся определения типов простых данных.

Класс рассматривается как тип простых данных, если он *тривиальный* (*trivial*), *со стандартным размещением* (*standard-layout*) и если типы всех его нестатических членов-данных также являются типами простых данных.

Тривиальный класс — это класс, который:

1. содержит тривиальный конструктор по умолчанию,
2. не содержит нетривиальных копирующих конструкторов,
3. не содержит нетривиальных перемещающих конструкторов,
4. не содержит нетривиальных копирующих операторов присваивания,
5. не содержит нетривиальных перемещающих операторов присваивания,
6. содержит тривиальный деструктор.

Класс со стандартным размещением — это класс, который:

1. не содержит нестатических членов-данных, имеющих тип класса с нестандартным размещением (или массива элементов такого типа) или ссылочный тип,
2. не содержит виртуальных функций,
3. не содержит виртуальных базовых классов,
4. имеет один и тот же вид доступности (public, private, protected) для всех нестатических членов-данных,
5. не имеет базовых классов с нестандартным размещением,
6. не является классом, одновременно содержащим унаследованные и неунаследованные нестатические члены-данные, или содержащим нестатические члены-данные, унаследованные сразу от нескольких базовых классов,
7. не имеет базовых классов того же типа, что и у первого нестатического члена-данного (если таковой есть).

2.2Ускорение компиляции

2.2.1Внешние шаблоны

В стандартном C++ компилятор должен инстанцировать шаблон всякий раз, когда встречает в единице трансляции его полную специализацию. Это может существенно увеличить время компиляции, особенно в тех случаях, когда шаблон инстанцирован с одинаковыми параметрами в большом числе единиц трансляции. На данный момент не существует способа указать C++, что инстанцирования быть не должно.

В C++11 введена идея внешних шаблонов.

В C++ не хватает возможности запретить компилятору инстанцировать шаблон в единице трансляции. C++11 просто расширяет данный синтаксис:

**extern** **template** **class** **std**::vector<MyClass>;

Данное выражение говорит компилятору *не* инстанцировать шаблон в данной единице трансляции.

2.3Повышение удобства использования

2.3.1Списки инициализации

Концепция списков инициализации пришла в C++ из C. Идея состоит в том, что структура или массив могут быть созданы передачей списка аргументов в порядке, соответствующем порядку определения членов структуры. Списки инициализации рекурсивны, что позволяет их использовать для массивов структур и структур, содержащих вложенные структуры.

**struct** **Object**

{

float first;

int second;

};

Object scalar = {0.43f, 10}; *// один объект, с first=0.43f и second=10*

Object anArray[] = {{13.4f, 3}, {43.28f, 29}, {5.934f, 17}}; *// массив из трёх объектов*

Списки инициализации очень полезны для статических списков и в тех случаях, когда требуется инициализировать структуру определённым значением. C++ также содержит конструкторы, которые могут содержать общую часть работы по инициализации объектов. Стандарт C++ позволяет использовать списки инициализации для структур и классов при условии, что те соответствуют определению простого типа данных (Plain Old Data — POD). Классы, не являющиеся POD, не могут использовать для инициализации списки инициализации, в том числе это касается и стандартных контейнеров C++, таких, как векторы.

C++11 связал концепцию списков инициализации и шаблонный класс, названный std::initializer\_list. Это позволило конструкторам и другим функциям получать списки инициализации в качестве параметров. Например:

**class** **SequenceClass**

{

**public**:

SequenceClass(std::initializer\_list<int> list);

};

Данное описание позволяет создать SequenceClass из последовательности целых чисел следующим образом:

SequenceClass someVar = {1, 4, 5, 6};

Здесь демонстрируется работа особого вида конструктора для списка инициализации. Классы, содержащие подобные конструкторы, обрабатываются особым образом во время инициализации (см. ниже).

Класс std::initializer\_list<> определён в стандартной библиотеке C++11. Однако, объекты данного класса могут быть созданы компилятором C++11 только статически с использованием синтаксиса со скобками {}. Список может быть скопирован после создания, однако, это будет копированием по ссылке. Список инициализации является константным: ни его члены, ни их данные не могут быть изменены после создания.

Так как std::initializer\_list<> является полноценным типом, он может быть использован не только в конструкторах. Обычные функции могут получать типизированные списки инициализации в качестве аргумента, например:

void FunctionName(std::initializer\_list<float> list);

FunctionName({1.0f, -3.45f, -0.4f});

Стандартные контейнеры могут быть инициализированы следующим образом:

std::vector<std::string> v = { "xyzzy", "plugh", "abracadabra" };

std::vector<std::string> v{ "xyzzy", "plugh", "abracadabra" };

2.3.2Универсальная инициализация

В стандарте C++ содержится ряд проблем, связанных с инициализацией типов. Существует несколько путей инициализации типов и не все они приводят к одинаковым результатам. К примеру, традиционный синтаксис инициализирующего конструктора может выглядеть как описание функции, и нужно предпринять дополнительные меры, чтобы компилятор не ошибся при анализе. Только агрегирующие типы и POD-типы могут быть инициализированы с помощью инициализаторов агрегатов (вида SomeType var = {/\*stuff\*/};).

C++11 предоставляет синтаксис, позволяющий использовать единую форму инициализации для всех видов объектов с помощью расширения синтаксиса списков инициализации:

**struct** **BasicStruct** {

int x;

double y;

};

**struct** **AltStruct** {

AltStruct(int x, double y) : x\_(x), y\_(y) {}

**private**:

int x\_;

double y\_;

};

BasicStruct var1{5, 3.2};

AltStruct var2{2, 4.3};

Инициализация var1 работает точно так же, как и при инициализации агрегатов, то есть, каждый объект будет инициализирован копированием соответствующего значения из списка инициализации. При необходимости будет применено неявное преобразование типов. Если нужного преобразования не существует, исходный код будет считаться некорректным. Во время инициализации var2 будет вызван конструктор.

Предоставлена возможность писать подобный код:

**struct** **IdString**

{

std::string name;

int identifier;

};

IdString GetString()

{

**return** {"SomeName", 4}; *// Обратите внимание на отсутствие явного указания типов*

}

Универсальная инициализация не заменяет полностью синтаксиса инициализации с помощью конструктора. Если в классе есть конструктор, принимающий в качестве аргумента список инициализации (ИмяТипа(initializer\_list<SomeType>);), он будет иметь более высокий приоритет по сравнению с другими возможностями создания объектов. Например, в C++11 std::vector содержит конструктор, принимающий в качестве аргумента список инициализации:

std::vector<int> theVec{4};

Данный код приведёт к вызову конструктора, принимающего в качестве аргумента список инициализации, а не конструктор с одним параметром, создающий контейнер заданного размера. Для вызова этого конструктора пользователь должен будет использовать стандартный синтаксис вызова конструктора.

2.3.3Вывод типов

В стандартном C++ (и C) тип переменной должен быть явно указан. Однако, после появления шаблонных типов и техник шаблонного метапрограммирования, тип некоторых значений, в особенности возвращаемых значений функций, не может быть легко задан. Это приводит к сложностям при хранении промежуточных данных в переменных, иногда может потребоваться знание внутреннего устройства конкретной библиотеки метапрограммирования.

C++11 предлагает два способа для смягчения этих проблем. Во-первых, определение явно инициализируемой переменной может содержать ключевое слово auto. Это приведёт к тому, что будет создана переменная типа инициализирующего значения:

**auto** someStrangeCallableType = std::bind(&SomeFunction, \_2, \_1, someObject);

**auto** otherVariable = 5;

Типом someStrangeCallableType станет тот тип, который возвращает конкретная реализация шаблонной функции std::bind для заданных аргументов. Данный тип будет легко определён компилятором во время выполнения семантического анализа, а вот программисту для определения типа пришлось бы провести ряд изысканий.

Тип otherVariable также чётко определён, однако, так же легко может быть определён и программистом. Этот тип — int, такой же как у целочисленной константы.

Кроме того, для определения типа выражения во время компиляции может быть использовано ключевое слово decltype. Например:

int someInt;

**decltype**(someInt) otherIntegerVariable = 5;

Использование decltype наиболее полезно совместно с auto, так как тип переменной, описанной как auto, известен только компилятору. Кроме того, использование decltype может быть весьма полезным в выражениях, использующих перегрузку операторов и специализацию шаблонов.

auto также может быть использован для уменьшения избыточности кода. Например, вместо:

**for** (vector<int>::const\_iterator itr = myvec.cbegin(); itr != myvec.cend(); ++itr)

программист сможет написать:

**for** (**auto** itr = myvec.cbegin(); itr != myvec.cend(); ++itr)

Разница становится особенно заметной, когда программист использует большое число различных контейнеров, несмотря на то, что и сейчас существует хороший путь для уменьшения избыточного кода — использование typedef.

Тип, помеченный как decltype, может отличаться от типа выведенного с помощью auto.

#include *<vector>*

int main()

{

**const** std::vector<int> v(1);

**auto** a = v[0]; *// тип a - int*

**decltype**(v[0]) b = 1; *// тип b - const int& (возвращаемое значение*

*// std::vector<int>::operator[](size\_type) const)*

**auto** c = 0; *// тип c - int*

**auto** d = c; *// тип d - int*

**decltype**(c) e; *// тип e - int, тип сущности, именованной как c*

**decltype**((c)) f = c; *// тип f - int&, так как (c) является lvalue*

**decltype**(0) g; *// тип g - int, так как 0 является rvalue*

}

2.3.4For-цикл по коллекции

В стандартном C++ для перебора элементов коллекции требуется масса кода. В некоторых языках, например, в C#, есть средства, предоставляющие «foreach»-инструкцию, которая автоматически перебирает элементы коллекции от начала до конца. C++11 вводит подобное средство. Инструкция for позволит проще осуществлять перебор коллекции элементов:

int my\_array[5] = {1, 2, 3, 4, 5};

**for**(int &x : my\_array)

{

x \*= 2;

}

Эта форма for, называемая в английском языке «range-based for», посетит каждый элемент коллекции. Это будет применимо к C-массивам, спискам инициализаторов и любым другим типам, для которых определены функции begin() и end(), возвращающие итераторы. Все контейнеры стандартной библиотеки, имеющие пару begin/end, будут работать с for-инструкцией по коллекции.

Такой цикл будет работать и, например, с C-like массивами, т.к. C++11 вводит искусственно для них необходимые псевдометоды (begin, end и некоторые другие).

*// range-based обход классического массива*

int arr1[] = { 1, 2, 3 };

**for** (**auto** el : arr1);

2.3.5Лямбда-функции и выражения

В стандартном C++, например, при использовании алгоритмов стандартной библиотеки C++ sort и find, часто возникает потребность в определении функций-предикатов рядом с местом, где осуществляется вызов этого алгоритма. В языке существует только один механизм для этого: возможность определить класс функтора (передача экземпляра класса, определенного внутри функции, в алгоритмы запрещена (Meyers, Effective STL)). Зачастую данный способ является слишком избыточным и многословным и лишь затрудняет чтение кода. Кроме того, стандартные правила C++ для классов, определённых в функциях, не позволяют использовать их в шаблонах и таким образом делают их применение невозможным.

Очевидным решением проблемы явилось разрешение определения лямбда-выражений и лямбда-функций в C++11. Лямбда-функция определяется следующим образом:

[](int x, int y) { **return** x + y; }

Тип возвращаемого значения этой безымянной функции вычисляется как decltype(x+y). Тип возвращаемого значения может быть опущен только в том случае, если лямбда-функция представлена в форме return *expression*. Это ограничивает размер лямбда-функции до одного выражения.

Тип возвращаемого значения может быть указан явно, например:

[](int x, int y) -> int { int z = x + y; **return** z; }

В этом примере создаётся временная переменная z для хранения промежуточного значения. Как и в нормальных функциях, это промежуточное значение не сохраняется между вызовами.

Тип возвращаемого значения может быть полностью опущен, если функция не возвращает значения (то есть тип возвращаемого значения — void)

Также возможно использование ссылок на переменные, определённые в той же области видимости, что и лямбда-функция. Набор таких переменных обычно называют замыканием. Замыкания определяются и используются следующим образом:

std::vector<int> someList;

int total = 0;

std::for\_each(someList.begin(), someList.end(), [&total](int x) {

total += x;

});

std::cout << total;

Это отобразит сумму всех элементов в списке. Переменная total хранится как часть замыкания лямбда-функции. Так как она ссылается на стековую переменную total, она может менять её значение.

Переменные замыкания для локальных переменных могут быть также определены без использования символа ссылки *&*, что означает, что функция будет копировать значение. Это вынуждает пользователя заявлять о намерении сослаться на локальную переменную или скопировать её.

Для лямбда-функций, гарантированно исполняемых в области их видимости, возможно использование всех стековых переменных без необходимости явных ссылок на них:

std::vector<int> someList;

int total = 0;

std::for\_each(someList.begin(), someList.end(), [&](int x) {

total += x;

});

Способы внутренней реализации могут различаться, но предполагается, что лямбда-функция сохранит указатель на стек функции, в которой она создана, а не будет работать с отдельными ссылками на переменные стека.

Если вместо [&] используется [=], все используемые переменные будут скопированы, что позволяет использовать лямбда-функцию вне области действия исходных переменных.

Способ передачи по умолчанию можно также дополнить списком отдельных переменных. Например, если необходимо передать большинство переменных по ссылке, а одну по значению, можно использовать следующую конструкцию:

int total = 0;

int value = 5;

[&, value](int x) { total += (x \* value); } (1); *//(1) вызов лямбда-функции с передачей значения 1*

Это вызовет передачу total по ссылке, а value — по значению.

Если лямбда-функция определена в методе класса, она считается дружественной этому классу. Такие лямбда-функции могут использовать ссылку на объект типа класса и обращаться к его внутренним полям:

[](SomeType \*typePtr) { typePtr->SomePrivateMemberFunction(); }

Это будет работать только если областью создания лямбда-функции является метод класса SomeType.

Особым образом реализована работа с указателем this на объект, с которым взаимодействует текущий метод. Он должен быть явно обозначен в лямбда-функции:

[**this**]() { **this**->SomePrivateMemberFunction(); }

Использование формы [&] или [=] лямбда-функции делает this доступным автоматически.

Тип лямбда-функций зависит от реализации; имя этого типа доступно только компилятору. Если необходимо передать лямбда-функцию в качестве параметра, она должна быть шаблонного типа, либо сохранена с использованием std::function. Ключевое слово auto позволяет локально сохранить лямбда-функцию:

**auto** myLambdaFunc = [**this**]() { **this**->SomePrivateMemberFunction(); };

Кроме того, если функция не принимает аргументов, то () можно опустить:

**auto** myLambdaFunc = []{ std::cout << "hello" << std::endl; };

2.3.6Альтернативный синтаксис функций

Иногда возникает потребность в реализации шаблона функции, результатом применения которого являлось бы выражение, имеющее тот же тип и ту же категорию значения (value category), что и у некоторого другого выражения.

**template** <**typename** **LHS**, **typename** **RHS**>

RETURN\_TYPE AddingFunc(**const** LHS &lhs, **const** RHS &rhs) *// каким должен быть RETURN\_TYPE?*

{

**return** lhs + rhs;

}

Для того, чтобы выражение AddingFunc(x, y) имело тот же тип и ту же категорию значения, что и выражение lhs + rhs при передаче данных аргументов x и y, в рамках C++11 можно было бы использовать следующее определение:

**template** <**typename** **LHS**, **typename** **RHS**>

**decltype**(std::declval<**const** LHS &>() + std::declval<**const** RHS &>())

AddingFunc(**const** LHS &lhs, **const** RHS &rhs)

{

**return** lhs + rhs;

}

Данная запись несколько громоздка, и было бы хорошо иметь возможность вместо std::declval<const LHS &>() и std::declval<const RHS &>() использовать соответственно lhs и rhs. Однако в следующем варианте

**template** <**typename** **LHS**, **typename** **RHS**>

**decltype**(lhs + rhs) AddingFunc(**const** LHS &lhs, **const** RHS &rhs) *// Не допустимо в C++11*

{

**return** lhs + rhs;

}

выглядящем более удобочитаемым, идентификаторы lhs и rhs, используемые в операнде decltype, не могут обозначать параметры, объявленные позже. Для решения этой проблемы в C++11 представлен новый синтаксис объявления функций с указанием возвращаемого типа в конце:

**template** <**typename** **LHS**, **typename** **RHS**>

**auto** AddingFunc(**const** LHS &lhs, **const** RHS &rhs) -> **decltype**(lhs + rhs)

{

**return** lhs + rhs;

}

Следует отметить, однако, что в более обобщённой реализации AddingFunc, приведённой ниже, новый синтаксис не даёт выигрыша в плане краткости:

**template** <**typename** **LHS**, **typename** **RHS**>

**auto** AddingFunc(LHS &&lhs, RHS &&rhs) ->

**decltype**(std::forward<LHS>(lhs) + std::forward<RHS>(rhs))

{

**return** std::forward<LHS>(lhs) + std::forward<RHS>(rhs);

}

**template** <**typename** **LHS**, **typename** **RHS**>

**auto** AddingFunc(LHS &&lhs, RHS &&rhs) ->

**decltype**(std::declval<LHS>() + std::declval<RHS>()) *// эффект такой же, как и с std::forward выше*

{

**return** std::forward<LHS>(lhs) + std::forward<RHS>(rhs);

}

**template** <**typename** **LHS**, **typename** **RHS**>

**decltype**(std::declval<LHS>() + std::declval<RHS>()) *// эффект такой же, как и с помещением типа в конец*

AddingFunc(LHS &&lhs, RHS &&rhs)

{

**return** std::forward<LHS>(lhs) + std::forward<RHS>(rhs);

}

Новый синтаксис может использоваться в более простых объявлениях и описаниях:

**struct** **SomeStruct**

{

**auto** FuncName(int x, int y) -> int;

};

**auto** SomeStruct::FuncName(int x, int y) -> int

{

**return** x + y;

}

Использование ключевого слова «auto» в этом случае означает только позднее указание возвращаемого типа и не связано с его автоматическим выведением.

2.3.7Улучшение конструкторов объектов

Стандартный C++ не допускает вызова одних конструкторов класса из других конструкторов этого же класса; каждый конструктор должен полностью инициализировать все члены класса либо вызывать для этого методы класса. Неконстантные члены класса не могут быть инициализированы в месте объявления этих членов.

C++11 избавляет от этих проблем.

Новый стандарт позволяет вызывать одни конструкторы класса из других (так называемая делегация). Это позволяет писать конструкторы, использующие поведение других конструкторов без внесения дублирующего кода.

Пример:

**class** **SomeType** {

int number;

**public**:

SomeType(int new\_number) : number(new\_number) {}

SomeType() : SomeType(42) {}

};

Из примера видно, что конструктор SomeType без аргументов вызывает конструктор того же класса с целочисленным аргументом для инициализации переменной number. Похожего эффекта можно было добиться, указав инициализирующее значение 42 для этой переменной прямо при её объявлении.

**class** **SomeType** {

int number = 42;

**public**:

SomeType() {}

**explicit** SomeType(int new\_number) : number(new\_number) {}

};

Любой конструктор класса будет инициализировать number значением 42, если он сам не присваивает ей другое значение.

Примером языков, которые так же решают эти проблемы служат Java, C# и D.

Следует заметить, что если в C++03 объект считается до конца созданным когда его конструктор завершает выполнение, то в C++11 после выполнения хотя бы одного делегирующего конструктора остальные конструкторы будут работать уже над полностью сконструированным объектом. Несмотря на это объекты производного класса начнут конструироваться только после выполнения всех конструкторов базовых классов.

2.3.8Явное замещение виртуальных функций и финальность

Возможна ситуация, когда сигнатура виртуального метода изменена в базовом классе или изначально неправильно задана в производном классе. В таких случаях данный метод в классе-наследнике не будет замещать соответствующий метод базового класса. Так что если программист должным образом не изменит сигнатуру метода во всех классах-наследниках, метод может быть вызван некорректно в процессе выполнения программы. Например:

**struct** **Base** {

**virtual** void some\_func();

};

**struct** **Derived** : Base {

void sone\_func();

};

Здесь в имени виртуальной функции, объявленной в производном классе, допущена ошибка, поэтому такая функция не будет замещать Base::some\_func и, соответственно, не будет вызываться полиморфно через указатель или ссылку на базовый подобъект.

В C++11 будет добавлена возможность отследить подобные проблемы на этапе компиляции (а не на этапе выполнения). Для обратной совместимости данная возможность является опциональной. Новый синтаксис представлен ниже:

**struct** **B**

{

**virtual** void some\_func();

**virtual** void f(int);

**virtual** void g() **const**;

};

**struct** **D1** : **public** B

{

void sone\_func() **override**; *// ошибка: неверное имя функции*

void f(int) **override**; *// OK: замещает такую же функцию в базовом классе*

**virtual** void f(long) **override**; *// ошибка: несоответствие типа параметра*

**virtual** void f(int) **const** **override**; *// ошибка: несоответствие cv-квалификации функции*

**virtual** int f(int) **override**; *// ошибка: несоответствие типа возврата*

**virtual** void g() **const** **final**; *// OK: замещает такую же функцию в базовом классе*

**virtual** void g(long); *// OK: новая виртуальная функция*

};

**struct** **D2** : D1

{

**virtual** void g() **const**; *// ошибка: попытка замещения финальной функции*

};

Наличие у виртуальной функции спецификатора final означает, что её дальнейшее замещение невозможно. Кроме того, класс, определённый со спецификатором final, не может использоваться в качестве базового класса:

**struct** **F** **final**

{

int x, y;

};

**struct** **D** : F *// ошибка: наследование от final классов запрещено*

{

int z;

};

Идентификаторы override и final имеют специальное значение только при использовании в определённых ситуациях. В остальных случаях они могут использоваться в качестве нормальных идентификаторов (например, как имя переменной или функции).

2.3.9Константа нулевого указателя

Со времён появления Си в 1972, константа 0 играла двойную роль целого числа и нулевого указателя. Одним из способов борьбы с этой неопределённостью, свойственной языку Си, служит макрос NULL, который обычно осуществляет подстановку ((void\*)0) или 0. C++ в этом плане отличается от Си, позволяя использовать только 0 в качестве константы нулевого указателя. Это приводит к плохому взаимодействию с перегрузкой функций:

void foo(char \*);

void foo(int);

Если макрос NULL определён как 0 (что является обычным для C++), строка foo(NULL); приведёт к вызову foo(int), а не foo(char \*), как можно предположить при беглом просмотре кода, что почти наверняка не совпадает с планами программиста.

Одним из новшеств C++11 является новое ключевое слово для описания константы нулевого указателя — nullptr. Данная константа имеет тип std::nullptr\_t, который можно неявно конвертировать в тип любого указателя и сравнить с любым указателем. Неявная конверсия в целочисленный тип недопустима, за исключением bool. В исходном предложении стандарта не допускалось неявной конверсии в булевый тип, но рабочая группа разработчиков стандарта разрешила такое преобразования в целях совместимости с обычными типами указателей. Предлагаемая формулировка была изменена после единогласного голосования в июне 2008[1].

В целях обеспечения обратной совместимости, константа 0 также может использоваться в качестве нулевого указателя.

char \*pc = **nullptr**; *// верно*

int \*pi = **nullptr**; *// верно*

bool b = **nullptr**; *// верно. b = false.*

int i = **nullptr**; *// ошибка*

foo(**nullptr**); *// вызывает foo(char \*), а не foo(int);*

Часто конструкции, где указатель гарантированно пуст, проще и безопаснее, чем остальные — потому можно сделать перегрузку с nullptr\_t.

**class** **Payload**;

**class** **SmartPtr** {

SmartPtr() = **default**;

SmartPtr(nullptr\_t) {} *// <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<*

**explicit** SmartPtr(Payload\* aData) : fData(aData) {}

*// конструкторы копирования и op= опустим*

~SmartPtr() { **delete** fData; }

**private**:

Payload\* fData = **nullptr**;

}

SmartPtr getPayload1() { **return** **nullptr**; } *// будет вызвана перегрузка SmartPtr(nullptr\_t).*

2.3.10Перечисления со строгой типизацией

В стандартном C++ перечисления не являются типобезопасными. В действительности они представлены целыми числами, несмотря на то, что сами типы перечислений различны между собой. Это позволяет производить сравнения между двумя значениями из разных перечислений. Единственной возможностью, которую предлагает C++03 для защиты перечислений, является запрет на неявное преобразование целых чисел или элементов одного перечисления в элементы другого перечисления. Кроме того, способ представления в памяти (целочисленный тип) зависит от реализации и поэтому не является переносимым. Наконец, элементы перечислений имеют общую область видимости, что приводит к невозможности создания элементов с одинаковым именем в разных перечислениях.

C++11 предлагает специальную классификацию этих перечислений, свободную от вышеописанных недостатков. Для описания таких перечислений используется объявление enum class (также возможно использование enum struct в качестве синонима):

**enum** **class** **Enumeration** {

Val1,

Val2,

Val3 = 100,

Val4, */\* = 101 \*/*

};

Такое перечисление является типобезопасным. Элементы классового перечисления невозможно неявно преобразовать в целые числа. Как следствие, сравнение с целыми числами также невозможно (выражение Enumeration::Val4 == 101 приводит к ошибке компиляции).

Тип классового перечисления теперь не зависит от реализации. По умолчанию, как в случае выше, таким типом является int, но в иных случаях тип может быть задан вручную следующим образом:

**enum** **class** **Enum2** : unsigned int {Val1, Val2};

Область действия элементов перечислений определяется областью действия имени перечисления. Использование имён элементов требует указания имени классового перечисления. Так, например, значение Enum2::Val1 определено, а значение Val1 — не определено.

Кроме того, C++11 предлагает возможность указания явной области видимости и низлежащего типа и для обычных перечислений:

**enum** **Enum3** : unsigned long {Val1 = 1, Val2};

В данном примере имена элементов перечисления определены в пространстве перечисления (Enum3::Val1), но для обеспечения обратной совместимости имена элементов также доступны в общей области видимости.

Также в C++11 возможно предварительное объявление перечислений. В предыдущих версиях C++ это было невозможным, поскольку размер перечисления зависел от его элементов. Такие объявления можно использовать только в тех случаях, когда размер перечисления указан (явно или неявно):

**enum** **Enum1**; *// неверно для C++ и C++11; низлежащий тип не может быть определён*

**enum** **Enum2** : unsigned int; *// верно для C++11, низлежащий тип указан явно*

**enum** **class** **Enum3**; *// верно для C++11, низлежащий тип — int*

**enum** **class** **Enum4** : unsigned int; *// верно для C++11.*

**enum** **Enum2** : unsigned short; *// неверно для C++11, поскольку Enum2 ранее объявлен с иным низлежащим типом*

2.3.11Угловые скобки

Парсеры стандартного C++ всегда определяют комбинацию символов «>>» как оператор правого сдвига. Отсутствие пробела между закрывающими угловыми скобками в параметрах шаблона (если они вложены) воспринимается как синтаксическая ошибка.

C++11 улучшает поведение анализатора в этом случае так, что несколько правых угловых скобок будут интерпретироваться как закрытие списков аргументов шаблонов.

Описанное поведение может быть исправлено в пользу старого подхода с помощью круглых скобок.

**template**<**class** **T**> **class** **Y** { */\* ... \*/* };

Y<X<1>> x3; *// Правильно, то же, что и "Y<X<1> > x3;".*

Y<X<6>>1>> x4; *// Синтаксическая ошибка. Нужно писать "Y<X<(6>>1)>> x4;".*

Как было показано выше, данное изменение не совсем совместимо с предыдущим стандартом.

2.3.12Операторы явного преобразования

Стандарт C++ предлагает ключевое слово explicit как модификатор конструкторов с одним параметром, чтобы такие конструкторы не функционировали как конструкторы неявного преобразования. Однако это никак не влияет на действительные операторы преобразования. Например, класс умного указателя может содержать operator bool() для имитации обычного указателя. Такой оператор можно вызвать, например, так: if(smart\_ptr\_variable) (ветка выполняется, если указатель ненулевой). Проблемой является то, что такой оператор не защищает от других непредвиденных преобразований. Поскольку в C++ тип bool объявлен как арифметический, возможно неявное преобразование в любой целочисленный тип или даже в тип числа с плавающей точкой, что в свою очередь может привести к непредвиденным математическим операциям.

В C++11 ключевое слово explicit применимо и к операторам преобразования. По аналогии с конструкторами, оно защищает от непредвиденных неявных преобразований. Однако ситуации, когда язык по контексту ожидает булевый тип (например, в условных выражениях, циклах и операндах логических операторов), считаются явными преобразованиями и оператор явного преобразования в bool вызывается непосредственно.

2.3.13Шаблонный typedef

В стандартном C++ ключевое слово typedef можно использовать только как определение синонима для другого типа, в том числе, как синоним для спецификации шаблона с указанием всех его параметров. Но невозможно создание шаблонного синонима. Например:

**template**< **typename** **First**, **typename** **Second**, int third>

**class** **SomeType**;

**template**< **typename** **Second**>

**typedef** SomeType<OtherType, Second, 5> TypedefName; *// Невозможно в C++*

Это не будет компилироваться.

В C++11 добавлена эта возможность со следующим синтаксисом:

**template**< **typename** **First**, **typename** **Second**, int third>

**class** **SomeType**;

**template**< **typename** **Second**>

**using** TypedefName = SomeType<OtherType, Second, 5>;

В C++11 директива using также может использоваться для создания псевдонима типа данных.

**typedef** void (\*OtherType)(double); *// Старый стиль*

**using** OtherType = void (\*)(double); *// Новый синтаксис*

2.3.14Снятие ограничений с union

В предыдущих стандартах C++ существует ряд ограничений по использованию элементов классовых типов внутри объединений. В частности, объединения не могут содержать объекты с нетривиальным конструктором. C++11 снимает часть таких ограничений.[2]

Вот простой пример объединения, разрешённого в C++11:

*//для placement new*

#include *<new>*

**struct** **Point** {

Point() {}

Point(int x, int y): x\_(x), y\_(y) {}

int x\_, y\_;

};

**union** **U** {

int z;

double w;

Point p; *// Неверно для C++03, поскольку Point имеет нетривиальный конструктор. Однако код работает корректно в C++11.*

U() { **new**( &p ) Point(); } *// Для объединения не определяются нетривиальные методы.*

*// При необходимости они могут быть удалены, чтобы работало ручное определение*

};

Изменения не затрагивают существующий код, поскольку лишь ослабляют существующие ограничения.

2.4Расширение функциональности

2.4.1Шаблоны с переменным числом аргументов

До появления C++11 шаблоны (классов или функций) могли принимать только заданное число аргументов, определяемых при первоначальном объявлении шаблона. C++11 позволяет определять шаблоны с переменным числом аргументов любого типа.

**template**<**typename**... Values> **class** **tuple**;

Например, шаблонный класс *tuple* (кортеж) принимает любое количество имён типов в качестве параметров шаблона:

**class** **tuple**<int, std::vector<int>, std::map<std::string, std::vector<int>>> some\_instance\_name;

Аргументы могут и отсутствовать, так что вариант class tuple<> some\_instance\_name также будет работать.

Чтобы запретить инстанцирование шаблона без аргументов, можно использовать следующее определение:

**template**<**typename** **First**, **typename**... Rest> **class** **tuple**;

Шаблоны с переменным числом аргументов также применимы к функциям, что позволяет использовать их в типобезопасных вариантах функций с переменным числом аргументов (таких как printf), а также для обработки нетривиальных объектов.

**template**<**typename**... Params> void printf(**const** std::string &str\_format, Params... parameters);

Оператор ... играет здесь две роли. Слева от Params оператор объявляет о необходимости упаковать параметры. Использование запакованных параметров позволяет связать 0 или более аргументов с шаблоном. Запакованные параметры могут использоваться не только для передачи имён типов. Оператор ... справа в свою очередь осуществляет распаковку параметров в отдельные аргументы (см. args... в теле функции в примере ниже).

Также возможно рекурсивное использование шаблонов с переменным числом аргументов. Одним из примеров может быть типобезопасная замена *printf*:

void printf(**const** char \*s)

{

**while** (\*s) {

**if** (\*s == '%' && \*(++s) != '%')

**throw** std::runtime\_error("invalid format string: missing arguments");

std::cout << \*s++;

}

}

**template**<**typename** **T**, **typename**... Args>

void printf(**const** char \*s, T value, Args... args)

{

**while** (\*s) {

**if** (\*s == '%' && \*(++s) != '%') {

std::cout << value;

++s;

printf(s, args...); *// продолжаем обработку аргументов, даже если \*s == 0*

**return**;

}

std::cout << \*s++;

}

**throw** std::logic\_error("extra arguments provided to printf");

}

Этот шаблон является рекурсивным. Обратите внимание, что функция printf вызывает результаты инстанциирования самой себя или базовую функцию printf в случае, если *args...* пустой.

Нет простого механизма обхода параметров в шаблоне с переменным числом аргументов. Несмотря на это, использование оператора распаковки аргументов позволяет обойти эту проблему.

Например, класс может быть определён следующим образом:

**template** <**typename**... BaseClasses> **class** **ClassName** : **public** BaseClasses... {

**public**:

ClassName (BaseClasses&&... base\_classes) : BaseClasses(base\_classes)... {}

};

Оператор распаковки продублирует все типы родительских классов ClassName таким образом, что класс будет наследован от всех типов, указанных в шаблонных параметрах. Кроме того, конструктор должен принимать ссылку на все базовые классы чтобы осуществлялась инициализация каждого базового класса-родителя ClassName.

Параметры шаблона могут быть перенаправлены. В сочетании с ссылками на rvalue (см. выше) можно выполнить перенаправление:

**template**<**typename** **TypeToConstruct**> **struct** **SharedPtrAllocator** {

**template**<**typename** ...Args> std::shared\_ptr<TypeToConstruct> construct\_with\_shared\_ptr(Args&&... params) {

**return** std::shared\_ptr<TypeToConstruct>(**new** TypeToConstruct(std::forward<Args>(params)...));

};

};

Данный код осуществляет распаковку списка аргументов в конструктор TypeToConstruct. Синтакс std::forward<Args>(params) позволяет абсолютно прозрачно перенаправить аргументы к конструктору, вне зависимости от их rvalue-характера. Функция автоматически оборачивает указатели в std::shared\_ptr для обеспечения защиты от утечек памяти.

Также есть возможность определить число упакованных аргументов следующим образом:

**template**<**typename** ...Args> **struct** **SomeStruct** {

**static** **const** int size = **sizeof**...(Args);

};

Здесь SomeStruct<Type1, Type2>::size равен 2, а SomeStruct<>::size равен 0.

2.4.2Новые строковые литералы

C++03 предлагал два типа строковых литералов. Первый тип, строка, заключённая в двойные кавычки, представляет собой массив с завершающим нулём (null-terminated) типа const char. Второй тип, определённый как L"", представляет собой массив с завершающим нулём типа const wchar\_t, где wchar\_t является широким символом неопределённых размеров и семантики. Ни один из типов литералов не предполагает поддержку строковых литералов UTF-8, UTF-16, или любого другого типа Unicode кодировок[en].

Определение типа char было модифицировано до явного высказывания, что он по меньшей мере размера, необходимого для хранения восьми-битной кодировки UTF-8, и достаточно большой, чтобы содержать любой символ из набора символов времени выполнения. Ранее в стандарте этот тип определялся как один символ, позже, следуя стандарту языка Си, он стал гарантированно занимать как минимум 8 бит.

Есть три Unicode кодировки, которые поддерживаются в стандарте C++11: UTF-8, UTF-16, и UTF-32. В дополнение к вышеуказанным изменениям встроенного символьного типа char, в C++11 добавлено два новых символьных типа: char16\_t и char32\_t. Они предназначены для хранения UTF-16 и UTF-32 символов соответственно.

Ниже показано, как создать строковые литералы для каждой из этих кодировок:

u8"I'm a UTF-8 string."

u"This is a UTF-16 string."

U"This is a UTF-32 string."

Типом первой строки является обычный const char[]. Тип второй строки — const char16\_t[]. Тип третьей строки — const char32\_t[].

При построении строковых литералов в стандарте Unicode, часто полезно вставить Unicode код прямо в строку. Для этого C++11 предлагает следующий синтаксис:

u8"This is a Unicode Character: **\u2018**."

u"This is a bigger Unicode Character: **\u2018**."

U"This is a Unicode Character: **\U00002018**."

Число после \u должно быть шестнадцатеричным; не нужно использовать префикс 0x. Идентификатор \u означает 16-битный Unicode код; для ввода 32-битного кода используется \U и 32-битное шестнадцатеричное число. Могут быть введены только действительные Unicode коды. Например, коды в диапазоне U+D800-U+DFFF запрещены, так как они зарезервированы для суррогатных пар в кодировке UTF-16.

Также иногда полезно избегать экранирования строк вручную, особенно при использовании литералов XML файлов, скриптовых языков программирования, или регулярных выражений. Для этих целей C++11 поддерживает «сырые» строковые литералы:

R"(The String Data \ Stuff " )"

R"delimiter(The String Data \ Stuff " )delimiter"

В первом случае всё между "( и )" является частью строки. Символы " и \ не нужно экранировать. Во втором случае "delimiter( начинает строку, и она заканчивается только при достижении )delimiter". Строка delimiter может быть любой строкой длиной до 16 символов, включая пустую строку. Эта строка не может содержать пробелы, управляющие символы, '(', ')', или символ '\'. Использование этой строки-разделителя позволяет использовать символ ')' в «сырых» строковых литералах. Например, R"delimiter((a-z))delimiter" эквивалентно "(a-z)"[3].

«Сырые» строковые литералы могут быть объединены с литералом из расширенного набора (префикс L"") или любыми префиксами Unicode литералов.

LR"(Raw wide string literal \t (without a tab))"

u8R"XXX(I'm a "raw UTF-8" string.)XXX"

uR"\*(This is a "raw UTF-16" string.)\*"

UR"(This is a "raw UTF-32" string.)"

2.4.3Пользовательские литералы

Пользовательские литералы реализуются с помощью перегрузки оператора operator"". Литералы могут быть с квалификатором inline или constexpr. Желательно, чтобы литерал начинался с символа нижнего подчёркивания, так как может возникнуть коллизия с будущими стандартами. Например, литерал i уже принадлежит комплексным числам из std::complex.

Литералы могут принимать только один из следующих типов: const char \* , unsigned long long int , long double , char , wchar\_t , char16\_t , char32\_t. Достаточно перегрузить литерал только для типа const char \*. Если не найдено более подходящего кандидата, то будет вызван оператор с этим типом. Пример преобразования миль в километры:

**constexpr** int **operator** "" \_mi (unsigned long long int i)

{ **return** 1.6 \* i;}

Строковые литералы принимают вторым аргументом std::size\_t, а первым один из: const char \* , const wchar\_t \*, const char16\_t \* , const char32\_t \*. Строковые литералы применяются к записям, записанным в двойных кавычках.

2.4.4Многопоточная модель памяти

C ++ 11 стандартизирует поддержку многопоточного программирования. Здесь задействованы две части: модель памяти, которая позволяет нескольким потокам сосуществовать в программе, и библиотека, поддерживающая взаимодействие между потоками.

Модель памяти определяет, как несколько потоков могут обращаться к одной и той же ячейке памяти, и определяет, когда изменения, произведенные одним потоком, становятся видимыми для других потоков.

2.4.6Явное задание по умолчанию и удаление специальных методов

Спецификаторы default и delete можно указывать вместо тела метода.

**class** **Foo**

{

**public**:

Foo() = **default**;

Foo(int x) {*/\* ... \*/*}

};

Спецификатор default означает реализацию по умолчанию и может применяться только к специальным функциям-членам:

* конструктор по-умолчанию;
* конструктор копий;
* конструктор перемещения;
* оператор присваивания;
* оператор перемещения;
* деструктор.

Спецификатор delete помечают те методы, работать с которыми нельзя. Раньше приходилось объявлять такие конструкторы в приватной области класса.

**class** **Foo**

{

**public**:

Foo() = **default**;

Foo(**const** Foo&) = **delete**;

void bar(int) = **delete**;

void bar(double) {}

};

*// ...*

Foo obj;

obj.bar(5); *// ошибка!*

obj.bar(5.42); *// ok*

2.4.7Тип long long int

Целый тип long long int специфицирован в C99 и де-факто широко применяется в C++. Теперь его официально включили в стандарт.

2.4.8Статическая диагностика

C++11 имеет два механизма статической диагностики:

* Ключевое слово **static\_assert** выдаёт ошибку компиляции, если выражение в скобках ложно.
* Библиотека **type\_traits**, содержащая шаблоны, выдающие информацию о типе прямо во время компиляции.

#include *<type\_traits>*

**template**<**class** **T**>

void run(T \*aData, size\_t n)

{

**static\_assert**(std::is\_pod<T>::value, "Тип T должен быть простым.");

...

}

2.4.9Работа sizeof с элементами данных в классах без создания объекта

* C++03 позволял использовать оператор sizeof для простых типов и объектов. Но следующая конструкция была недопустима:
* **struct** **SomeType** { OtherType member; };
* **sizeof**(SomeType::member); *//Не работает в C++03, но верно для C++11.*
* Результатом этого вызова должен быть размер OtherType. C++03 не поддерживает такой вызов и данный код не будет скомпилирован. C++11 разрешает подобные конструкции.

2.4.10Управление выравниванием объектов и запросы на выравнивание

C++11 позволяет выравнивать переменные с помощью операторов alignof и alignas.

alignof принимает тип и возвращает число байт, на которые можно смещать объект. Например для struct X { int n; char c; };, размер которой равен 8 байтам, alignof вернёт значение 4. Для ссылок он возвращает значение для типа ссылки; для массивов — значение для элемента массива

alignas управляет выравниванием объекта в памяти. Например, можно указать, чтобы массив char должен был быть выравнен соответствующим образом, чтобы хранить тип float:

**alignas**(float) unsigned char c[**sizeof**(float)]

3 Изменения стандартной библиотеки C++

3.1Изменения в имеющихся компонентах

* При вставке в std::set программист иногда знает, в какую позицию попадёт новый элемент. Для этого служит необязательный параметр — «хинт»; если догадка верна, временнáя оценка будет амортизированной константой, а не O(log n). Смысл «хинта» в C++11 изменили: раньше он означал элемент *до* текущего, что не совсем правильно: непонятно, что делать, если вставка в первую позицию. Теперь это элемент *после* текущего.
* Написан удобный шаблон, вызывающий конструкторы без выделения памяти — std::allocator\_traits<>::construct(). Во все контейнеры добавили метод emplace, создающий объект на месте.
* Добавлены новые языковые возможности C++11.
* Добавлены методы cbegin и cend, гарантированно создающие константные итераторы. Удобно для метапрограммирования, для задания типов через auto.
* В контейнерах, которые заводят память с запасом, появилась функция shrink\_to\_fit.
* В std::list поставлены более жёсткие рамки на то, что выполняется за *O*(*n*), а что — за константное время.
* В std::vector добавился прямой доступ к памяти через data().
* Запретили нескольким std::string ссылаться на одну и ту же память. Благодаря этому появился и прямой доступ через front(), удобный, например, для взаимодействия string’а и WinAPI.

3.2Управление потоками

В то время, как язык C++ 03 предоставляет модель памяти, поддерживающую многопоточность, основная поддержка для фактического использования многопоточности обеспечивается стандартной библиотекой C ++ 11.

Предоставляется класс потока (std::thread), который принимает объект функции (и необязательный список аргументов для передачи ему) для запуска в новом потоке. Можно заставить поток остановиться до завершения другого исполняемого потока, обеспечив поддержку объединения потоков через функцию-член std::thread::join(). Если это возможно, предоставляется доступ к нативному дескриптору потока для специфичных для платформы операций с помощью функции-члена std::thread::native\_handle().

Для синхронизации между потоками в библиотеку добавляются соответствующие мьютексы (std::mutex, std::recursive\_mutex и т. д.) и условные переменные (std::condition\_variable и std::condition\_variable\_any). Они доступны через блокировки при инициализации ресурса (RAII) (std::lock\_guard и std::unique\_lock) и алгоритмы блокировки для простоты использования.

Для высокопроизводительной низкоуровневой работы иногда требуется общение между потоками без накладных расходов мьютексов. Это делается с использованием атомарных операций в ячейках памяти. Они могут дополнительно указывать минимальные ограничения видимости памяти, необходимые для операции. Также могут быть использованы для этой цели явные барьеры памяти.

Библиотека потоков C++ 11 также включает в себя фьючерсы (futures) и обещания (promise) для передачи асинхронных результатов между потоками и класс std::packaged\_task для упаковки вызова функции, которая может генерировать такой асинхронный результат. Предложение по фьючерсам было подвергнуто критике, поскольку ему не хватает способа комбинировать фьючерсы и проверять выполнение одного обещания в наборе обещаний.

Дополнительные средства потоковой обработки высокого уровня, такие как пулы потоков, были помещены в будущий технический отчет C++. Они не являются частью C++ 11, но предполагается, что их возможная реализация будет построена полностью поверх функций библиотеки потоков.

Новая функция std::async предоставляет удобный способ запуска задач и привязки результата их выполнения к объекту класса std::future. Пользователь может выбрать, будет ли задание выполняться асинхронно в отдельном потоке или синхронно в текущем потоке, ожидающем значение.

3.3Хеш-таблицы

std::hash\_set и std::hash\_map давно были нестандартным расширением STL, по факту реализованным в большинстве компиляторов. В C++11 они стали стандартом, под именами std::unordered\_set и std::unordered\_map. Хотя по факту это хеш-таблицы и стандарт не оставляет много пространства для манёвра, имена даны в стиле C++: не «как они реализованы», а «что они собой представляют».

3.4Регулярные выражения

Новая библиотека, объявленная в заголовочном файле <regex>, содержит в себе несколько новых классов:

* Регулярные выражения представлены в виде экземпляров класса std::regex;
* результаты поиска представлены в виде экземпляров шаблона std::match\_results.

Функция std::regex\_search используется для поиска, для операции ‘найти и заменить’ используется функция std::regex\_replace. Функция вернёт строку после выполнения замены. Алгоритмы std::regex\_search и std::regex\_replace получают на вход регулярное выражение и строку и возвращают найденные результаты в виде экземпляра std::match\_results.

Пример использования std::match\_results:

**const** char \*reg\_esp = "[ ,.**\\**t**\\**n;:]"; *// Список символов-разделителей.*

*// то же самое можно сделать, используя "сырые" строки:*

*// const char \*reg\_esp = R"([ ,.\t\n;:])";*

std::regex rgx(reg\_esp); *// 'regex' - это экземпляр шаблонного класса*

*// 'basic\_regex' с шаблонным параметром 'char'.*

std::cmatch match; *// 'cmatch' - это экземпляр шаблонного класса*

*// 'match\_results' с шаблонным параметром 'const char \*'.*

**const** char \*target = "Unseen University - Ankh-Morpork";

*// Фиксирует все слова строки 'target' разделённые символами из 'reg\_esp'.*

**if**( std::regex\_search( target, match, rgx ) ) {

*// Если слова, разделённые заданными символами присутствуют в строке.*

**const** size\_t n = match.size();

**for**( size\_t a = 0; a < n; a++ ) {

std::string str( match[a].first, match[a].second );

std::cout << str << "**\n**";

}

}

Обратите внимание, что требуется использование двойной обратной косой черты, так как C++ использует обратную косую черту для экранирования символов. Можно использовать «сырые строки» — еще одно нововведение стандарта C++11.

Библиотека <regex> не требует ни изменения существующих заголовочных файлов, ни установки дополнительных расширений языка.

3.5Расширяемые классы генерации случайных чисел

Стандартная библиотека Си позволяла генерировать псевдослучайные числа с помощью функции rand. Однако её поведение могло варьироваться в зависимости от реализации.

Данная функциональность разделена на две части: движок генератора, который содержит текущее состояние генератора случайных чисел и производит псевдослучайные числа, и распределение, которое определяет диапазон и математическое распределение результата. Комбинация этих двух объектов создает генератор случайных чисел.

Движки генераторов:

* linear\_congruential\_engine,
* subtract\_with\_carry\_engine,
* mersenne\_twister\_engine.

Распределения:

* uniform\_int\_distribution,
* uniform\_real\_distribution,
* bernoulli\_distribution,
* binomial\_distribution,
* geometric\_distribution,
* negative\_binomial\_distribution,
* poisson\_distribution,
* exponential\_distribution,
* gamma\_distribution,
* weibull\_distribution,
* extreme\_value\_distribution,
* normal\_distribution,
* lognormal\_distribution,
* chi\_squared\_distribution,
* cauchy\_distribution,
* fisher\_f\_distribution,
* student\_t\_distribution,
* discrete\_distribution,
* piecewise\_constant\_distribution and
* piecewise\_linear\_distribution.

Пример:

#include *<random>*

#include *<functional>*

std::uniform\_int\_distribution<int> distribution(0, 99);

std::mt19937 engine; *// Вихрь Мерсенна MT19937*

**auto** generator = std::bind(distribution, engine);

int random = generator(); *// Получить случайное число от 0 до 99.*

int random2 = distribution(engine); *// Получить случайное число, используя движок и распределение напрямую.*