Министерство образования Республики Беларусь

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

**Факультет прикладной математики и информатики**

ПАНЬКОВА АННА ВЛАДИМИРОВНА

**Дополнительные возможности языка С++:**

Реферат  
студента 1 курса 2 группы

|  |  |  |
| --- | --- | --- |
|  |  |  |

Минск 2017

**СОДЕРЖАНИЕ**

[ВВЕДЕНИЕ 3](#_Toc480638337)

[1 Использование ключевого слова auto 4](#_Toc480638338)

[2 Автоматическое определение типа возвращаемого значения для обычных функций 5](#_Toc480638339)

[3 Константа нулевого указателя nullptr 5](#_Toc480638340)

[4 range-based циклы 6](#_Toc480638341)

[5 Явное замещение виртуальных функций и финальность 6](#_Toc480638342)

[6 Rvalue-ссылки и семантика перемещения 8](#_Toc480638343)

[7 Новые строковые литералы 9](#_Toc480638344)

[7.1 Новые типы литералов 9](#_Toc480638345)

[7.2 Пользовательские литералы 9](#_Toc480638346)

[7.3 Обособленные строки 10](#_Toc480638347)

[7.4 Пользовательские литералы для типов стандартной библиотеки 10](#_Toc480638348)

[7.5 Двоичные литералы 10](#_Toc480638349)

[8 Новые типы 11](#_Toc480638350)

[8.1 Изменения в имеющихся компонентах 11](#_Toc480638351)

[8.2 Хеш-таблицы 11](#_Toc480638352)

[9 Выражения 11](#_Toc480638353)

[9.1 Регулярные выражения 11](#_Toc480638354)

[9.2 Обобщённые константные выражения 12](#_Toc480638355)

[8.3 Лямбда-выражения 13](#_Toc480638356)

[10. Операторы явного преобразования 15](#_Toc480638357)

[10.1 Ключевое слово explicit 15](#_Toc480638358)

[ЛИТЕРАТУРА 16](#_Toc480638359)

**РЕФЕРАТ**

Реферат, 14с., источника.

**Ключевые слова:** ТИП, УКАЗАТЕЛЬ, ЦИКЛ, ВИРТУАЛЬНАЯ ФУНЦИЯ, СТРУКТУРА, СЕМАНТИКА ПЕРЕМЕЩЕНИЯ, ССЫЛКА, ЛИТЕРАЛ, ВЫРАЖЕНИЯ, ОПЕРАТОРЫ ПРЕОБОРАЗОВАНИЯ.

**Объект исследования** - дополнительные возможности языка С++.

**Цель работы** – изучить дополнительные возможности языка С++, представленные в виде С++11, С++14.

**Методы исследования** – использование стандартных библиотек языка С++.

**Область применения** – программирование на языке С++.

# ВВЕДЕНИЕ

Любой язык программирования постоянное развивается, появляются новые возможности усовершенствования старых принципов. С++ не является исключением. В 2011 году выходит новая версия языка С++. Новый стандарт включает дополнения в ядре языка и расширение его стандартной библиотеки.

В изменениях, принятых в новом стандарте было уделено много внимания “новичкам” в программировании, т.е. людям, которые не стремятся углубить знания языка С++, а лишь используют его в работе над специфичными и узкими задачами.

В 2014 году выходит следующая версия языка С++, а именно С++14. Его можно рассматривать как небольшое расширение над уже существующим C++11, которое содержит в основном исправления ошибок и небольшие улучшения.

В данном реферате раскрыты основные новые возможности С++, выпущенные в 2011 и 2014 годах и принятые в стандарт.

# 1 Использование ключевого слова auto

Ключевое слово auto не является совершенно новым для языка C++, оно уже было использовано раньше, однако, в качестве спецификатора хранения переменной (так же, как register, static, extern). Теперь же слово auto позволяет не указывать тип переменной явно, а сообщает компилятору о том, что он тип объявляемой переменной будет определен на основе типа инициализирующего значения. Это может быть использовано при объявлении переменных, которые находятся в различный областях видимости. Например, пространства имен, блоки и т.д.

auto a = 5;

auto b = 5LL;

auto p = new foo();

Здесь переменная «а» имеет тип int, переменная «b» именно тип long long, переменная «р» имеет тип foo\*.

Использование auto позволяет сократить объем кода. Это не относится к тем случаем, когда можно написать вместо «int a = 5;» - «auto b = 5;» Однако часто используемо при объявлении итераторов в библиотеке STL, которые могут быть использованы, например, для прохода по контейнерам. В этом случае можно использовать typedef, но auto вытеснило его использование из-за своей простоты. Сравним два ваприанта:

for (auto it = container.begin(); it != container.end(); ++it)

{

// do smth

}

и

for (std::vector<std::map<int, std::string>>::const\_iterator it = container.begin(); it != container.end(); ++it)

{

// do smth

}

Нельзя не отметить, что возвращаемое значение не может быть auto. Но можно использовать auto вместо типа возвращаемого значения функции. В этом случае, auto не сообщает компилятору, что он должен определить тип, оно лишь дает ему команду искать возвращаемый тип в конце функции. В примере указанном ниже, возвращаемый тип функции compose — это возвращаемый тип оператора сложения, который суммирует два значения типа А и В. Здесь decltype позволяет определить тип на основе входного параметра

template <typename А, typename В>

auto compose(А a, В b) -> decltype(a + b) {

return a + b;

}

auto c = compose(2, 3.14);

В итоге c будет иметь тип double.

# 2 Автоматическое определение типа возвращаемого значения для обычных функций

C++11 позволяет выводить тип возвращаемых значений для лямбда-функций из типа возвращаемого выражения. Новый стандарт C++14 расширяет эту возможность на все функции. Для того, чтобы можно было использовать автоматический вывод типа возвращаемого значения, функция обязана быть объявлена с типом auto, указанного в качестве типа возвращаемого функцией значения.

auto MyFunction();

Но есть несколько требований к таким функциям. Если в теле функции в разных местах происходит возврат нескольких выражений, то все эти выражения должны иметь общий тип. Кроме того, функции, которые используют автоматическое выведение типа возвращаемого значения, могут быть объявлены заранее, но использовать их можно только после определения. В таких функциях даже может быть использована рекурсия, однако рекурсивный вызов должен быть выполнен хотя бы после одного возврата значения в этой функции. Ниже представлены две функции. Одна из них определена верно (“Correct”), другая – неверно (“Wrong”).

auto Correct(int i) {

if (i == 1)

return i; // в качестве типа возвращаемого значения выводится int

else

return Correct(i - 1) + i; // теперь можно вызывать

}

auto Wrong(int i) {

if (i != 1)

return Wrong(i - 1) + i; // неподходящее место для рекурсии. Нет предшествующего возврата.

else

return i; // в качестве типа возвращаемого значения выводится int

}

# 3 Константа нулевого указателя nullptr

Еще со времён появления С 1972, константа «0» играла двойную роль целого числа и нулевого указателя. Эта неопределенность требовала решения. Одним из способов борьбы, свойственному языку Си, служит макрос NULL.  C++ в этом плане отличается от С, он позволяет использовать только «0» в качестве константы нулевого указателя. И это приведет к плохому взаимодействию с перегрузкой следующих функций:

void foo(char \*);

void foo(int);

Если макрос NULL определён как 0 (а это является нормой для C++), строка foo(NULL); приведёт к вызову функции foo(int), но не функции foo(char \*), как можно было предположить. Это почти наверняка не совпадает с планами программиста.

Одним из новшеств C++11 является новое ключевое слово для описания константы нулевого указателя — nullptr. Эта константа имеет тип std::nullptr\_t, его можно неявно преобразовать к типу указателя и сравнить с любым другим указателем. Однако, неявное преобразование к в целочисленному типу недопустима. Исключением является преобразование к типу bool. Интересно, что в исходном предложении стандарта неявная конверсии в булевый тип была также недопустима, но рабочая группа разработчиков все таки разрешила ее, для совместимости с обычными типами указателей. И тогда формулировка была изменена после голосования в июне 2008, которая была принята единогласно.

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);

# 4 range-based циклы

В стандартном C++ для перебора элементов коллекции требуется большой объем кода. Коллекция — программный объект, содержащий в себе, тем или иным образом, набор значений одного или различных типов, и позволяющий обращаться к этим значениям. Коллекция позволяет записывать в себя значения и извлекать их. Назначение коллекции — служить хранилищем объектов и обеспечивать доступ к ним. Т.е. речь идет о массивах, стеке, векторах и т.д.

В С++11 добавлена поддержка средства foreach для итерации по набору. Но итерации таким образом можно выполнять лишь в том случае, если для объекта этой итерации перегружены методы begin() и end().

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

for (int &x : my\_array)

{

x \*= 2;

}

Это может оказаться очень полезным, когда необходимо получить элементы из коллекции и сделать с ними что - то, не заботясь об индексах или кол - ве элементов.

std::map<std::string, std::vector<int>> map;

std::vector<int> v;

v.push\_back(1);

v.push\_back(2);

v.push\_back(3);

map["one"] = v;

for (const auto &kvp : map)

{

std::cout << kvp.first << std::endl;

for (auto v : kvp.second)

std::cout << v << std::endl;

}

# 5 Явное замещение виртуальных функций и финальность

Сейчас речь пойдет о виртуальных функциях. В языке С++ слово virtual опционально и поэтому очень сильно затрудняет чтение кода. Оно заставляет просматривать всю иерархию классов наследования, чтобы проверить объявлен ли тот или иной метод виртуальным. На этой почве часто возникают ошибки:

class B

{

public: virtual void f(short) { std::cout << "B::f" << std::endl; }

};

class D : public B

{

public: virtual void f(int) { std::cout << "D::f" << std::endl; }

};

Здесь D::f переопределяет B::f. Однако они имеют разную сигнатуру, поэтому в данном случае В::f это просто другой метод. Он перегружен, но не переопределен. Т.е. работая через указатель на базовый класс, Вы можете вызвать f() и ожидать вывода «переопределенного» вами метода: «D::f», однако вывод будет «B::f». Возможно и другая распространенная ошибка: параметры одни и те же, но в базовом классе метод константный, а в производном — нет.

С++11 предоставляет способ для решения этих проблем и отследить подобные проблемы еще на этапе компиляции. Были добавлены два новых идентификатора (не ключевые слова). Оverride и final. Override используется для указания того, что метод является переопределением виртуального метода в базовом классе, а final указывает, что производный класс не может переопредить виртуальный метод базового класса. А класс, определенный с таким спецификатором, не может быть использован в качестве базового класса. 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

};

Ошибка в наследовании:

struct F final

{

int x, y;

};

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

{

int z;

};

# 6 Rvalue-ссылки и семантика перемещения

Rvalue ссылки –техническое расширение языка C++. Такие ссылки позволяют программистам избегать логически ненужного копирования и обеспечивают возможность идеальной передачи (perfect forwarding). Прежде всего они предназначены для использования в высоко производительных проектах и библиотеках. В то время как обычно копирование через конструкторы копии очень сильно снижают производительность и могут сильно увеличить время работы программы, если речь идет о копировании больших объемов данных. Это нормальный процесс, если программе действительно требуются копии объектов, но чаще всего программисту нужно лишь переместить, например, один контейнер в другой и удалить старую копию.

Rvalue ссылка – это составной тип, очень похожий на традиционную ссылку в C++. Обычная ссылка или lvalue ссылка по семантике объявляется таким образом A& a\_ref1 = a; а rvalue ссылка так - A&& a\_ref2 = a; Rvalue ссылка ведет себя точно так же, как и lvalue ссылка, за исключением того, что она может быть связана с временным объектом, тогда как lvalue связать с временным (не константным) объектом нельзя.

Оказывается, что комбинация rvalue ссылок и lvalue ссылок — это то, что необходимо для лёгкой реализации семантики перемещения (move semantics). Большинству программистов rvalue ссылки дают возможность создавать более производительные библиотеки.

Семантика перемещений (move semantics)

Рассмотрим функцию swap

template <class T> swap(T& a, T& b)

{

T tmp(a); // сейчас мы имеем две копии объекта a

a = b; // теперь у нас есть две копии объекта b

b = tmp; // а теперь у нас две копии объекта tmp (т.е. a)

}

Но, как уже упоминалось ранее, нам не нужны две копии одного объекта. А нужно лишь поменять их местами.

template <class T> swap(T& a, T& b)

{

T tmp(std::move(a));

a = std::move(b);

b = std::move(tmp);}

В данном примере вызов функции move() возвращает значение объекта, переданного в качестве параметра, однако не отвечает за сохранность этого объекта. Т.е., например, если передать параметром в move() тип vector, то вполне можно ожидать, что после окончания работы функции от переданного параметра останется лишь вектор нулевой длины, потому что все элементы будут перемещены, а не скопированы. Поэтому можно утверждать, что перемещение – это считывание со стиранием (destructive read).

Но для этого примера В данном случае мы специализировали swap, сделали его более оптимальным. Но ведь специализация каждой функции, которая скопирует большой объект перед тем, как она удалит или перезапишет его была бы неконструктивной.

Главная задача rvalue ссылок состоит в том, чтобы позволять нам реализовывать перемещение без переписывания кода и издержек времени выполнения (runtime overhead).

# 7 Новые строковые литералы

## 7.1 Новые типы литералов

Ранее C++ предлагал только два типа строковых литералов. Первым типом была строка, заключавшаяся в двойные кавычки. Она представляла собой массив с завершающим нулём (null-terminated) типа const char. Вторым типом, определённым как L"", был массив с завершающим нулём типа const wchar\_t, где wchar\_t является широким символом неопределённых размеров и семантики. Но ни один из типов литералов не предполагал поддержку таких строковых литералов, как UTF-8, UTF-16, или любого другого типа Unicode кодировок.

Теперь 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 предназначаются соответственно для хранения UTF-16 и UTF-32 символов.

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

u"This is a UTF-16 string."

U"This is a UTF-32 string."

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

Часто бывает полезно вставить Unicode код прямо в строку. С++11 позволяет это сделать слелующим образом: u8"This is a Unicode Character: \u2018." Число после \u обязательно должно быть шестнадцатеричным, не требуется использовать префикс 0x. Идентификатор \u будет обозначать 16-битный Unicode код; для ввода 32-битного кода используется \U и соответсвенно 32-битное шестнадцатеричное число. Вводить можно только действительные Unicode коды. А вот коды в диапазоне U+D800-U+DFFF запрещены, так как они уже зарезервированы для суррогатных пар в кодировке UTF-16.

## 7.2 Пользовательские литералы

Пользовательские литералы реализуются с использованием перегрузок оператора operator"". Они могут быть с квалификатором inline или constexpr. Весьма желательно, чтобы это литерал начинался с символа нижнего подчёркивания, так как может возникнуть коллизия с будущими стандартами. К примеру, литерал i уже принадлежит комплексным числам из std::complex.

Литералы принимают только один из следующих типов: const char \* , unsigned long long int , long double , char , wchar\_t , char16\_t , char32\_t. Вполне достаточно перегрузки литерала только для типа const char \*. Если не найдется более подходящего кандидата, то будет вызван оператор именно с этим типом. Пример преобразования миль в километры:

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

{

return 1.6 \* i;

}

## 7.3 Обособленные строки

Как известно, в С++, если мы попытаемся занести строку original типа string, состоящую из двух и более слов, например, в поток stringstream, через оператор <<, то переменная ss типа stringstream, будет содержать только лишь первое слово, до символа пробела. Эта проблема удачно решена в С++14 с помощью двойных кавычек, добавленных в начало и конец строки соответсвенно.

ss << quoted(original);

## 7.4 Пользовательские литералы для типов стандартной библиотеки

В C++11 описывается синтаксис литеральных суффиксов, которые определяются пользователем, однако ни один из них не использовался в стандартной библиотеке. C++14 добавил следующие стандартные литералы

1. «s» для создания различных std::basic\_string типов.
2. «h», «min», «s», «ms», «us» и «ns» для создания соответствующих временных интервалов std::chrono::duration.

string str = "hello world"s;

chrono::duration dur = 60s;

Здесь два литерала не будут вступать в конфликт, ибо строковой литерал будет работать только со строками, а секундный, соответственно, только с числами.

## 7.5 Двоичные литералы

Числовые литералы в C++14 теперь можно указывать в двоичной форме. Префиксы 0b или 0B используются в данном синтаксисе. Схожий синтаксис также используется в Java, Python, Perl и D.

В C++14 так же возможно использование апостроф, который произвольным образом разделяется разряды в числовых литералах. Часто это улучшает читабельность кода упрощает восприятие больших числовых констант в нем.

# 8 Новые типы

Целый тип long long int был специфицирован еще в C99 и широко применяется в C++. Теперь его официально включили в стандарт.

## 8.1 Изменения 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.
* 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.

## 8.2 Хеш-таблицы

std::hash\_set и std::hash\_map давно были нестандартным расширением STL, но по факту они реализовывались в большинстве компиляторов. Но вот 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. Их имена даны в стиле C++, а именно не «как они реализованы», а «что они собой представляют».

# 9 Выражения

## 9.1 Регулярные выражения

Новая библиотека объявляется в заголовочном файле <regex>. Она содержит в себе несколько новых классов

Регулярные выражения представлены в виде экземпляров класса std::regex, а 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. Она вернет строку после выполнения соответствующей замены. Алгоритмы std::regex\_search и std::regex\_replace получают на вход регулярное выражение и строку и возвращают результаты, которые они нашли в виде экземпляра 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' разделенные символами из 'reg\_esp'.

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

// Если слова, разделенные 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";

}

}

## 9.2 Обобщённые константные выражения

В C++ присутствует концепция константных выражений. Так, например, выражения типа 6+4 всегда возвращают одни и те же результаты, не вызывают никаких побочных эффектов. Сами по себе выражения, являющиеся константными, предоставляют компиляторам C++ возможности для оптимизации результата компиляции. Так как компиляторы вычисляют результаты таких выражений уже на этапе компиляции и сохраняют эти вычисленные результаты в программе. Так что подобные выражения вычисляются только один 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).

int GiveFive() { return 5; }

int some\_value[GiveFive() + 7]; // создание массива 12 целых; запрещено в C++

Вышеуказанный код запрещён в C++, потому что выражение GiveFive() + 7 формально не является константным выражением, оно неизвестно на этапе компиляции. Поэтомуч компилятор не знает, что функция на самом деле возвращает константу во время исполнения. А причиной данных рассуждений компилятора является то, что эта функция может повлиять на состояние глобальной переменной, ведб она может вызвать другую неконстантную функцию времени исполнения и т. д.

Стандарт C++11 вводит ключевое слово constexpr, которое позволяет пользователю гарантировать, что функция или конструктор объекта вернет. Тогда вышеуказанный код выше может быть переписан следующим образом:

constexpr int GiveFive() { return 5; }

int some\_value[GiveFive() + 7]; // создание массива 12 целых; разрешено в C++11

Однако использование ключевого слова constexpr порождает очень жёсткие ограничения на действия функции:

1. такая функция обязана 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 используется 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 сняли для переменных перед определением которых стоит ключевое слово constexpr:

constexpr double accelerationOfGravity = 9.8;

constexpr double moonGravity = accelerationOfGravity / 6;

Эти переменные считаются обозначенными словом const, но неявно. В них содержатся только результаты константных выражений, либо конструкторы таких выражений.

При необходимости конструирования константных значений из типов, которые определил пользователь, конструкторы этих типов описываются с помощью constexpr. Конструктор константных выражений, как и константные функции, должен быть определён до момента первого его использования. У этого конструктора тело должно быть пусто, и он может инициализировать члены своего типа только лишь константами.

## 8.3 Лямбда-выражения

В стандартном C++ очень часто используются алгоритмы стандартной библиотеки C++. Алгоритмов достаточно много и они разнообразны, но у всех есть одно обзее свойство. Они требуют определения функций-предикатов рядом с местом, где осуществляется вызов этого алгоритма. И, в принципе, поддерживает один механизм для этого, а именно возможность определить класс функтора. Но слишком часто этот способ является слишком избыточным, занимает много места, он лишь затрудняет чтение кода. Более того, стандартные правила C++ для классов, которые определены в функциях, не позволяют использование их в шаблонах и таким образом делают их применение просто невозможным.

Решением проблемы стало разрешение определения лямбда-выражений и лямбда-функций в C++11. Определение лямбда-функции:

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

Если требуется найти тип возвращаемого значения у этой безымянной функции, то его можно определить как decltype(x+y). Кроме того, он 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 можно указать явно, например:

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

Здесь создаётся временная переменная z для хранения промежуточного значения. Как и в обычных функциях, это промежуточное значение не сохранится между вызовами.

Кроме того, тип возвращаемого значения можно совсем опустить, если функция не возвращает значения (тип возвращаемого значения — void)

Также возможно использование ссылок на переменные, определённые в той же области видимости, что и лямбда-функция. Набор этих переменных обычно называют замыканием. Определение и использование замыканий:

std::vector<int> someList;

int total = 0;

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

total += x;

});

std::cout << total;

Переменная total использутеся как часть замыкания лямбда-функции. Из-за того, что она ссылается на переменную total, она может менять её значение.

Если объявить переменные замыкания для локальных переменных без использования символа ссылки &, то это будет означать, что функция скопирует значение.

Для лямбда-функций, гарантированно исполняемых в области их видимости, можно использовать все стековые переменные без необходимости явных ссылок на них:

[&](int x) {total += x; }

А вот 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:

[&, value](int x) { total += (x \* value); }

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

Отдельным образом реализована работа с указателем this. Он обязан быть явно обозначен в лямбда-функции, например, следующим образом

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

Но использование [&] или [=] лямбда-функции делает this доступным автоматически.

Кроме всего прочего лямбда-функцию можно передать в качестве параметра. Но тогда она должна быть шаблонного типа или может быть сохранена с использованием std::function. А вот ключевое слово auto дает возможность локально сохранить лямбда-функцию

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

Если наша лямбда-функция не принимает аргументов, то круглые скобки можно опустить

В С++14 разрешено использование лямбда-выражений со спецификатором типа , auto, то есть снято ограничение на указание конкретных типов.

auto lambda = [](auto x, auto y) {return x + y; };

# 10. Операторы явного преобразования

## 10.1 Ключевое слово explicit

Ключевое слово explicit – модификатор конструкторов с одним параметром. Запрещает конструкторам функционировать как конструкторам неявного преобразования. Но оно не влияет на действительные операторы преобразования. Рассмотрим класс умного указателя. Он может содержать operator bool(). Это имитирует работу обычного указателя. Этот указатель может попасть в ветку if(smart\_ptr). Проблемы заключается в том, что такой оператор не защитит от непредвиденных преобразований. Так как bool – это арифметический тип, то вполне может произойти неявное преобразование в любой целочисленный тип или в тип числа с плавающей точкой, а это привело бы к непредвиденным математическим операциям.

В C++11 ключевое слово explicit применяется также к операторам преобразования. По аналогии с конструкторами, оно способно защитить от непредвиденных неявных преобразований. Но, например, ситуации, когда язык по контексту ожидает булевый тип (например, в условных выражениях, циклах и операндах логических операторов), являются явными преобразованиями и оператор явного преобразования в bool будет вызван непосредственно.

# ЛИТЕРАТУРА

1. Стенли, Б.Л. Язык программирования C++. Базовый курс, 5-е издание = C++ Primer (5th Edition) / Б.Л. Стенли, Б.Э. Му — М.: Вильямс, 2014. — 1120 с.

2. Сиддхартха Р. Освой самостоятельно C++ за 21 день, 7-е издание = Sams Teach Yourself C++ in One Hour a Day, 7th Edition. — М.: Вильямс, 2013. — 688 с.

3. Прата С. Язык программирования C++ (C++11). Лекции и упражнения, 6-е издание = C++ Primer Plus, 6th Edition (Developer's Library). — М.: Вильямс, 2012. — 1248 с.

4. Meyers S. Effective Modern C++ : 42 Specific Ways to Improve Your Use of C++11 and C++14, O'Reilly Media – 2014. – 334 c.

5. Джосаттис Н.М. Стандартная библиотека C++. Справочное руководство - М.: Вильямс, 2014. – 1136 c.