

Contents

[Документация по языку C++](#)

[Справочник по языку C++](#)

[Справочник по языку C++](#)

[Возвращение к C++ \(современный C++\)](#)

[Лексические соглашения](#)

[Лексические соглашения](#)

[Токены и кодировки](#)

[Комментарии](#)

[Идентификаторы](#)

[Ключевые слова](#)

[Символы пунктуации](#)

[Числовые, логические литералы и литералы-указатели](#)

[Строковые и символьные литералы](#)

[Определенные пользователем литералы](#)

[Основные понятия](#)

[Основные понятия](#)

[Система типов C++](#)

[Область](#)

[Файлы заголовков](#)

[Единицы трансляции и компоновка](#)

[Функция main и аргументы командной строки](#)

[Завершение программы](#)

[Значения Lvalue и Rvalue](#)

[Временные объекты](#)

[Выравнивание](#)

[Тривиальные типы, стандартная раскладка и типы POD](#)

[Типы значений](#)

[Преобразования типов и безопасность типов](#)

[Стандартные преобразования](#)

[Встроенные типы](#)

[Встроенные типы](#)

[Диапазоны типов данных](#)

[nullptr](#)

[void](#)

[bool](#)

[false](#)

[true](#)

[char, wchar_t, char8_t, char16_t, char32_t](#)

[_int8, _int16, _int32, _int64](#)

[_m64](#)

[_m128](#)

[_m128d](#)

[_m128i](#)

[_ptr32, _ptr64](#)

[Числовые пределы](#)

[Числовые ограничения](#)

[Пределы целых чисел](#)

[Пределы значений с плавающей запятой](#)

[Объявления и определения](#)

[Объявления и определения](#)

[Классы хранения](#)

[auto](#)

[const](#)

[constexpr](#)

[extern](#)

[Инициализаторы](#)

[Псевдонимы и определения типов](#)

[using - объявление](#)

[volatile](#)

[decltype](#)

[Атрибуты](#)

Встроенные операторы, приоритет и ассоциативность

Встроенные операторы, приоритет и ассоциативность

Оператор alignof

Оператор __uuidof

Операторы сложения: + и -

Оператор address-of: &

Операторы присваивания

Побитовый оператор И: &

Оператор побитового исключающего ИЛИ: ^

Оператор побитового включающего ИЛИ: |

Оператор приведения: ()

Оператор "запятая": ,

Условный оператор: ?: :

Оператор delete

Операторы равенства: == и !=

Оператор явного преобразования типа: ()

Оператор вызова функции: ()

Оператор косвенного обращения: *

Операторы сдвигов влево и вправо (>> и <<)

Оператор логического И: &&

Оператор логического отрицания: !

Оператор логического ИЛИ: ||

Операторы доступа к членам: . and ->

Операторы умножения и оператор модуля

Оператор new

Оператор дополнения до единицы: ~

Операторы указателей на члены: .* и ->*

Постфиксные операторы увеличения и уменьшения: ++ и --

Префиксные операторы увеличения и уменьшения: ++ и --

Операторы отношения: <, >, <= и >=

Оператор разрешения области: ::

Оператор sizeof

[Оператор индекса:](#)

[Оператор typeid](#)

[Унарные операторы "плюс" и "отрицание": + и -](#)

[Выражения](#)

[Выражения](#)

[Типы выражений](#)

[Типы выражений](#)

[Основные выражения](#)

[Многоточия и вариадические шаблоны](#)

[Постфиксные выражения](#)

[Выражения с унарными операторами](#)

[Выражения с бинарными операторами](#)

[Константные выражения](#)

[Семантика выражений](#)

[Приведение](#)

[Приведение](#)

[Операторы приведения](#)

[Операторы приведения](#)

[Оператор dynamic_cast](#)

[Иключение bad_cast](#)

[Оператор static_cast](#)

[Оператор const_cast](#)

[Оператор reinterpret_cast](#)

[Сведения о типах времени выполнения \(RTTI\)](#)

[Сведения о типах времени выполнения \(RTTI\)](#)

[Иключение bad_typeid](#)

[Класс type_info](#)

[Операторы](#)

[Операторы](#)

[Общие сведения об операторах в C++](#)

[Инструкции с метками](#)

[Оператор выражений](#)

Оператор выражений
Оператор NULL
Составные операторы (блоки)
Операторы выбора
Операторы выбора
if-else (Справочник по C#)
Оператор __if_exists
Оператор __if_not_exists
Оператор switch
Операторы итерации
Операторы итерации
Оператор while
Оператор do-while
Оператор for
Основанный на диапазоне оператор for
Операторы перехода
Операторы перехода
Оператор break
Оператор continue
Оператор return
Оператор goto
Передача управления
Пространства имен
Перечисления
Объединения
Функции
Функции
Функции с переменными списками аргументов
Перегрузка функций
Явно используемые по умолчанию и удаленные функции
Поиск имен функций с зависимостью от аргументов (поиск Koenig)
Аргументы по умолчанию

Встраиваемые функции

Перегрузка операторов

Перегрузка операторов

Общие правила перегрузки операторов

Перегрузка унарных операторов

Перегрузка унарных операторов

Перегрузка операторов инкремента и декремента

Бинарные операторы

Назначение

Вызов функции

Индексация ;

Доступ к членам

Классы и структуры

Классы и структуры

class

struct

Обзор членов класса

Управление доступом к членам

Управление доступом к членам

friend

private

protected

public

Инициализация фигурных скобок

Управление временем жизни и ресурсами объекта (RAII)

Идиома Pimpl для инкапсуляции времени компиляции

Переносимость на границах ABI

Конструкторы

Конструкторы

Конструкторы копий и операторы присваивания копий

Конструкторы перемещения и операторы присваивания перемещением

Делегирующие конструкторы

[Деструкторы](#)

[Общие сведения о функциях-членах](#)

[Общие сведения о функциях-членах](#)

[Описатель `virtual`](#)

[Описатель `override`](#)

[Описатель `final`](#)

[Наследование](#)

[Наследование](#)

[Виртуальные функции](#)

[Одиночное наследование](#)

[базовых классов;](#)

[Несколько базовых классов](#)

[Явное переопределение](#)

[Абстрактные классы](#)

[Общие сведения о правилах области](#)

[Ключевые слова наследования](#)

[virtual](#)

[__super](#)

[__interface](#)

[Специальные функции-члены](#)

[Статические члены](#)

[Пользовательские преобразования типов](#)

[Изменяемые элементы данных](#)

[Объявления вложенных классов](#)

[Типы анонимных классов](#)

[Указатели на члены](#)

[Указатель `this`](#)

[Битовые поля](#)

[Лямбда-выражения в C++](#)

[Лямбда-выражения в C++](#)

[Синтаксис лямбда-выражений](#)

[Примеры лямбда-выражений](#)

[Лямбда-выражения constexpr](#)

[Массивы](#)

[Ссылки](#)

[Ссылки](#)

[Декларатор ссылки Lvalue: &](#)

[Декларатор ссылки Rvalue: &&](#)

[Аргументы функции ссылочного типа](#)

[Возвращаемые значения функции ссылочного типа](#)

[Ссылки на указатели](#)

[Указатели](#)

[Указатели](#)

[Необработанные указатели](#)

[Указатели с ключевыми словами const и volatile](#)

[Операторы new и delete](#)

[Смарт-указатели](#)

[Практическое руководство. Создание и использование экземпляров unique_ptr](#)

[Практическое руководство. Создание и использование экземпляров shared_ptr](#)

[Практическое руководство. Создание и использование экземпляров weak_ptr](#)

[Практическое руководство. Создание и использование экземпляров CComPtr и CComQIPtr](#)

[Обработка исключений в C++](#)

[Обработка исключений в C++](#)

[Современные рекомендации по C++](#)

[Практическое руководство. Разработка с учетом безопасности исключений](#)

[Практическое руководство. Интерфейс между кодом с исключениями и без исключений](#)

[Операторы try, throw и catch](#)

[Проверка блоков catch](#)

[Исключения и очистка стека](#)

[Спецификации исключений \(throw\)](#)

[noexcept](#)

[Необработанные исключения C++](#)

[Сочетание исключений C \(структурированные\) и C++](#)

[Сочетание исключений C \(структурированные\) и C++](#)

[Использование setjmp-longjmp](#)

[Обработка структурированных исключений в C++](#)

[Обработка структурированных исключений \(C/C++\)](#)

[Обработка структурированных исключений \(C/C++\)](#)

[Написание обработчика исключений](#)

[Написание обработчика исключений](#)

[Оператор try-except](#)

[Написание фильтра исключений](#)

[Создание исключений программного обеспечения](#)

[Исключения оборудования](#)

[Ограничения обработчиков исключений](#)

[Написание обработчика завершения](#)

[Написание обработчика завершения](#)

[Оператор try-finally](#)

[Освобождение ресурсов](#)

[Время обработки исключений. Общие сведения](#)

[Ограничения обработчиков завершения](#)

[Перенос исключений между потоками](#)

[Утверждение и задаваемые пользователем сообщения](#)

[Утверждение и задаваемые пользователем сообщения](#)

[static_assert](#)

[Модули](#)

[Обзор модулей в C++](#)

[модуль, импорт, экспорт](#)

[Шаблоны](#)

[Шаблоны](#)

[typename](#)

[Шаблоны классов](#)

[Шаблоны функций](#)

[Шаблоны функций](#)

[Создание экземпляра шаблона функции](#)

Явное создание экземпляра
Явная специализация шаблонов функций
Частичное упорядочение шаблонов функций
Шаблоны функций-членов
Специализация шаблонов
Шаблоны и разрешение имен
Шаблоны и разрешение имен
Разрешение имен зависимых типов
Разрешение локально объявленных имен
Разрешение перегрузки вызовов шаблонов функций
Организация исходного кода (шаблоны C++)

Обработка событий
Обработка событий
__event
__hook
__raise
__unhook
Обработка событий в неуправляемом C++
Обработка событий в СОМ

Модификаторы, используемые в системах Microsoft
Модификаторы, используемые в системах Microsoft
Базовые адреса
Базовые адреса
Грамматика выражения __based
Относительные указатели

Соглашения о вызовах
Соглашения о вызовах
Передача аргументов и соглашения об именовании
Передача аргументов и соглашения об именовании
__cdecl
__clrcall
__stdcall

`_fastcall`

`_thiscall`

`_vectorcall`

Пример вызова. Прототип и вызов функции

Пример вызова. Прототип и вызов функции

Пример результатов вызова

Вызовы функций с атрибутом naked

Вызовы функций с атрибутом naked

Правила и ограничения для функций с атрибутом naked

Особенности написания кода пролога и эпилога

Сопроцессор для вычислений с плавающей запятой и соглашения о вызовах

Устаревшие соглашения о вызовах

`restrict` (C++ AMP)

Ключевое слово `tile_static`

`_declspec`

`_declspec`

`align`

`allocate`

`allocator`

`appdomain`

`code_seg (_declspec)`

`deprecated`

`dllexport, dllimport`

`dllexport, dllimport`

Определения и объявления

Определение встроенных функций C++ с помощью `dllexport` и `dllimport`

Общие правила и ограничения

Использование `dllimport` и `dllexport` в классах C++

`jit intrinsic`

`naked`

`noalias`

`noinline`

[noreturn](#)

[no_sanitize_address](#)

[nothrow](#)

[novtable](#)

[процесс](#)

[свойство;](#)

[restrict](#)

[safebuffers](#)

[selectany](#)

[spectre](#)

[thread](#)

[uuid](#)

[_restrict](#)

[_sptr, _uptr](#)

[_unaligned](#)

[_w64](#)

[_func_](#)

[Поддержка СОМ компилятора](#)

[Поддержка СОМ компилятора](#)

[Глобальные функции СОМ компилятора](#)

[Глобальные функции СОМ компилятора](#)

[_com_raise_error](#)

[ConvertStringToBSTR](#)

[ConvertBSTRToString](#)

[_set_com_error_handler](#)

[Классы поддержки СОМ компилятора](#)

[Классы поддержки СОМ компилятора](#)

[Класс _bstr_t](#)

[_bstr_t - класс](#)

[_bstr_t::_bstr_t](#)

[_bstr_t::Assign](#)

[_bstr_t::Attach](#)

`_bstr_t::copy`
`_bstr_t::Detach`
`_bstr_t::GetAddress`
`_bstr_t::GetBSTR`
`_bstr_t::length`
`_bstr_t::operator =`
`_bstr_t::operator +=, +`
`_bstr_t::operator !`

Операторы отношения `_bstr_t`

`_bstr_t::wchar_t *, _bstr_t::char*`

Класс `_com_error`

`_com_error - класс`

Функции-члены `_com_error`

Функции-члены `_com_error`
`_com_error::_com_error`
`_com_error::Description`
`_com_error::Error`
`_com_error::ErrorInfo`
`_com_error::ErrorMessage`
`_com_error::GUID`
`_com_error::HelpContext`
`_com_error::HelpFile`
`_com_error::HRESULTToWCode`
`_com_error::Source`
`_com_error::WCode`
`_com_error::WCodeToHRESULT`

Операторы `_com_error`

Операторы `_com_error`

`_com_error::operator =`
`_com_ptr_t class`
`_com_ptr_t - класс`
Функции-члены `_com_ptr_t`

Функции-члены _com_ptr_t

_com_ptr_t::_com_ptr_t

_com_ptr_t::AddRef

_com_ptr_t::Attach

_com_ptr_t::CreateInstance

_com_ptr_t::Detach

_com_ptr_t::GetActiveObject

_com_ptr_t::GetInterfacePtr

_com_ptr_t::QueryInterface

_com_ptr_t::Release

Операторы _com_ptr_t

Операторы _com_ptr_t

_com_ptr_t::operator =

Операторы отношения _com_ptr_t

Средства извлечения _com_ptr_t

Шаблоны реляционных функций

Класс _variant_t

_variant_t - класс

Функции-члены _variant_t

Функции-члены _variant_t

_variant_t::_variant_t

_variant_t::Attach

_variant_t::Clear

_variant_t::ChangeType

_variant_t::Detach

_variant_t::SetString

Операторы _variant_t

Операторы _variant_t

_variant_t::operator =

Операторы отношения _variant_t

Средства извлечения _variant_t

Расширения Майкрософт

[Нестандартное поведение](#)

[Ограничения компилятора](#)

[Справочник по препроцессору в C/C++](#)

[Справочник по стандартной библиотеке C++](#)

Справочник по языку C++

12.11.2021 • 2 minutes to read

Эта ссылка посвящена языку программирования C++, реализованному в компиляторе Microsoft C++.

Организация основана на *справочном руководстве по C++ с заметками*, Margaret Ellis) и Бьерном

Страуструп и на стандарт ANSI/ISO C++ International (ISO/IEC фдис 14882). Включены реализации компонентов языка C++ корпорацией Microsoft.

Общие сведения о современных методиках программирования C++ см. [в статье Добро пожаловать в C++](#).

Для быстрого поиска ключевого слова или оператора обращайтесь к следующим таблицам:

- [Ключевые слова C++](#)
- [Операторы C++](#)

В этом разделе

Лексические соглашения

Основные лексические элементы программ на C++: токены, комментарии, операторы, ключевые слова, знаки пунктуации, литералы. Кроме того, трансляция файлов, приоритет и ассоциативность операторов.

Основные понятия

Область, компоновка, запуск и завершение программы, классы хранения и типы.

[Встроенные типы](#) Фундаментальные типы, встроенные в компилятор C++, и их диапазоны значений.

Стандартные преобразования

Преобразование типов между встроенными типами. Кроме того, арифметические преобразования и преобразования между типами указателей, ссылочными типами и типами указателей на члены.

[Объявления и определения](#) Объявление и определение переменных, типов и функций.

Операторы, приоритет и ассоциативность

Операторы в C++.

Выражения

Типы выражений, семантика выражений, справочные разделы по операторам, приведению типов и операторам приведения, сведения о типах времени выполнения.

Лямбда-выражения

Метод программирования, с помощью которого неявно определяется класс объекта функции и создается объект функции этого типа класса.

Операторы

Операторы выражений, пустые операторы, составные операторы, операторы выбора, операторы итераций, операторы перехода и операторы объявления.

Классы и структуры

Вводные сведения о классах, структурах и объединениях. Кроме того, функции элементов, Специальные функции элементов, элементы данных, битовые поля, `this` указатель, вложенные классы.

Объединения

Определяемые пользователем типы, в которых все члены совместно используют одно и то же

расположение в памяти.

Производные классы

Одиночное и множественное наследование, `virtual` функции, несколько базовых классов, **абстрактные** классы, правила области. Кроме того, `_super` `_interface` Ключевые слова и.

Управление доступом к членам

Управление доступом к членам класса: `public`, `private` И `protected` ключевым словам. Дружественные функции и классы.

Перегрузка

Перегруженные операторы, правила перегрузки операторов.

Обработка исключений

Обработка исключений в C++, структурированная обработка исключений (SEH), ключевые слова, используемые при написании операторов обработки исключений.

Утверждения и User-Supplied сообщения

`#error`, `static_assert` Ключевое слово, `assert` макрос.

Шаблоны

Спецификации шаблонов, шаблоны функций, шаблоны классов, `typename` Ключевое слово, шаблоны и макросы, шаблоны и смарт-указатели.

Обработка событий

Объявление событий и обработчиков событий.

Модификаторы, специфичные для Майкрософт

Модификаторы, используемые в Microsoft C++. Адресация памяти, соглашения о вызовах, `naked` функции, расширенные атрибуты класса хранения (`_declspec`), `_w64` .

Встроенный ассемблер

Использование языка ассемблера и C++ в `_asm` блоках.

Поддержка СОМ компилятором

Справочник по характерным для систем Microsoft классам и глобальным функциям, используемым для поддержки типов модели СОМ.

Расширения Майкрософт

Расширения Майкрософт для C++.

Нестандартное поведение

Сведения о нестандартном поведении компилятора Microsoft C++.

Добро пожаловать в C++

Общие сведения о современных методиках программирования на C++ для написания безэффективных и правильных программ.

Связанные разделы

Расширения компонентов для платформ среды выполнения

Справочные материалы по использованию компилятора Microsoft C++ для платформы .NET.

Справочные сведения о сборке C/C++

Параметры компилятора, параметры компоновщика и другие средства сборки.

Справочник по препроцессору C/C++

Справочный материал по pragma-директивам, директивам препроцессора, предопределенным макросам

и препроцессору.

[Библиотеки Visual C++](#)

Список ссылок на начальные страницы ссылок для различных библиотек Microsoft C++.

См. также

[Справочник по языку C](#)

Возвращение к C++ — современный C++

12.11.2021 • 7 minutes to read

С момента своего создания C++ стал одним из наиболее широко используемых языков программирования в мире. Грамотно сконструированные программы на языках C++ быстры и эффективны. Он более гибок, чем другие языки: он может работать с самыми высокими уровнями абстракции, а также на низком аппаратном уровне. C++ предоставляет стандартные библиотеки с высоким уровнем оптимизации. Он обеспечивает доступ к аппаратным функциям низкого уровня, чтобы максимально увеличить скорость и сократить потребление памяти. С помощью C++ можно создавать широкий спектр приложений. Игры, драйверы устройств и высокопроизводительное научное программное обеспечение. Встраиваемые программы. Клиентские приложения Windows. Даже библиотеки и компиляторы для других языков программирования пишутся на C++.

Одно из начальных требований для C++ — обратная совместимость с языком C. В результате программы на C++ всегда можно писать в стиле C: с необработанными указателями, массивами, символьными строками с завершающим нулем и другими функциями. Это может обеспечить высокую производительность, но также может приводить к ошибкам и увеличению сложности. Эволюция C++ концентрируется на возможностях, которые значительно снижают необходимость использования идиом в стиле C. Старые средства программирования C все еще можно использовать там, где это необходимо, но в современном C++ они нужны все реже и реже. Современный код на C++ проще, безопаснее, элегантнее и так же быстр, как и раньше.

В следующих разделах приводятся общие сведения об основных возможностях современного C++. Если не указано иное, перечисленные здесь функции доступны в C++ 11 и более поздних версиях. В компиляторе C++ от Microsoft с помощью параметра `/std` можно указать версию стандарта, используемую для проекта.

Ресурсы и интеллектуальные указатели

Одним из основных классов ошибок в программировании в стиле C является *утечка памяти*. Утечки часто возникают из-за невозможности вызвать `delete` для памяти, выделенной с помощью `new`. Современный C++ придерживается принципа *получение ресурса есть инициализация* (англ. Resource Acquisition Is Initialization (RAII)). Идея проста. Ресурсы (память кучи, дескрипторы файлов, сокеты и т. д.) должны принадлежать объекту. Этот объект создает и получает новый выделенный ресурс в конструкторе и удаляет его в его деструкторе. Принцип RAII гарантирует, что все ресурсы должным образом возвращаются операционной системе, когда объект-владелец выходит за пределы области.

Для поддержки простого внедрения принципов RAII стандартная библиотека языка C++ предоставляет три типа интеллектуальных указателей: `std::unique_ptr`, `std::shared_ptr` и `std::weak_ptr`. Интеллектуальный указатель обрабатывает выделение и удаление памяти, которой он владеет. В следующем примере показан класс с членом-массивом, который выделяется в куче в вызове `make_unique()`. Вызовы `new` и `delete` инкапсулированы в классе `unique_ptr`. Когда объект `widget` выходит из области действия, вызывается деструктор `unique_ptr` и освобождается память, выделенная для массива.

```
#include <memory>
class widget
{
private:
    std::unique_ptr<int> data;
public:
    widget(const int size) { data = std::make_unique<int>(size); }
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000); // lifetime automatically tied to enclosing scope
                      // constructs w, including the w.data gadget member
    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data
```

При выделении памяти кучи всегда, когда это возможно, используйте интеллектуальные указатели. Если необходимо явно использовать операторы new и delete, следуйте принципу RAII. Дополнительные сведения см. в разделе [Управление временем жизни и ресурсами объекта \(RAII\)](#).

`std::string` И `std::string_view`

Строки в стиле C — это еще один основной источник ошибок. Используя `std::string` И `std::wstring`, можно устранить практически все ошибки, связанные со строками в стиле C. Дополнительно вы получаете преимущества функций-членов для поиска, добавления в конец и начало и т. д. Оба эти класса оптимизированы для быстрой работы. При передаче строки в функцию, для которой требуется доступ только для чтения, в C++ 17 можно использовать `std::string_view` для еще большего выигрыша в производительности.

`std::vector` И другие контейнеры стандартной библиотеки

Все контейнеры стандартной библиотеки следуют принципу RAII. Они предоставляют итераторы для безопасного обхода элементов. И они хорошо оптимизированы для повышения производительности, а также тщательно протестированы на отсутствие ошибок. Используя эти контейнеры, можно исключить потенциальные ошибки и неэффективные приемы в пользовательских структурах данных. Вместо необработанных массивов используйте `vector` в качестве последовательного контейнера в C++.

```
vector<string> apples;
apples.push_back("Granny Smith");
```

В качестве ассоциативного контейнера по умолчанию используйте `map` (не `unordered_map`). Используйте `set`, `multimap` И `multiset` для вырожденных и множественных операторов выбора.

```
map<string, string> apple_color;
// ...
apple_color["Granny Smith"] = "Green";
```

При необходимости оптимизации производительности рассмотрите возможность использования следующих средств.

- Тип `array` важен при внедрении, например, как член класса.
- Неупорядоченные ассоциативные контейнеры, такие как `unordered_map`. Они имеют меньше

издержек на элемент и постоянный по времени поиск, но их сложно использовать правильно и эффективно.

- Сортированные `vector`. Дополнительные сведения см. в разделе [Алгоритмы](#).

Не используйте массивы в стиле языка С. Для более старых API, которым требуется прямой доступ к данным, используйте такие методы доступа, как `f(vec.data(), vec.size())`. Дополнительные сведения о контейнерах см. в разделе [Контейнеры стандартной библиотеки C++](#).

Алгоритмы стандартной библиотеки

Перед принятием решения о том, что вам нужно написать собственный алгоритм для программы, сначала ознакомьтесь с [алгоритмами](#) стандартной библиотеки C++. Стандартная библиотека содержит постоянно увеличивающийся набор различных алгоритмов для многих распространенных операций, таких как поиск, сортировка, фильтрация и рандомизация. Имеется обширная математическая библиотека. Начиная с C++ 17 предлагаются параллельные версии многих алгоритмов.

Ниже приведены некоторые важные примеры.

- `for_each`, алгоритм обхода по умолчанию (наряду с циклами `for` на основе диапазона).
- `transform`, для изменения элементов контейнера "не на месте".
- `find_if`, алгоритм поиска по умолчанию.
- `sort`, `lower_bound` и другие алгоритмы сортировки и поиска по умолчанию.

При написании операторов сравнения по возможности используйте строгие выражения `<` и [именованные лямбда-выражения](#).

```
auto comp = [](const widget& w1, const widget& w2)
    { return w1.weight() < w2.weight(); }

sort( v.begin(), v.end(), comp );

auto i = lower_bound( v.begin(), v.end(), comp );
```

auto вместо явных имен типов

В C++ 11 введено ключевое слово `auto` для использования в объявлениях переменных, функций и шаблонов. Ключевое слово `auto` предписывает компилятору определить тип объекта, чтобы не указывать его явным образом. `auto` особенно полезно, когда выводимый тип является вложенным шаблоном.

```
map<int,list<string>>::iterator i = m.begin(); // C-style
auto i = m.begin(); // modern C++
```

Циклы `for` на основе диапазона

Итерации в стиле C для массивов и контейнеров подвержены ошибкам индексирования, а также достаточно рутинные. Чтобы устранить эти ошибки и сделать код более удобочитаемым, используйте с контейнерами стандартной библиотеки и необработанными массивами циклы `for` на основе диапазона. Дополнительные сведения см. в разделе [Оператор `for` на основе диапазона](#).

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v {1,2,3};

    // C-style
    for(int i = 0; i < v.size(); ++i)
    {
        std::cout << v[i];
    }

    // Modern C++:
    for(auto& num : v)
    {
        std::cout << num;
    }
}
```

Выражения `constexpr` вместо макросов

Макросы в языках С и С++ являются токенами, которые обрабатываются препроцессором перед компиляцией. Перед компиляцией файла каждый экземпляр токена макроса заменяется определенным значением или выражением. Макросы обычно используются в программировании в стиле С для определения значений констант времени компиляции. Однако макросы подвержены ошибкам и их сложно отлаживать. В современном С++ следует отдавать предпочтение переменным `constexpr` для констант времени компиляции.

```
#define SIZE 10 // C-style
constexpr int size = 10; // modern C++
```

Унифицированная инициализация

В современном С++ можно использовать инициализацию с помощью фигурных скобок для любого типа. Такая форма инициализации особенно удобна при инициализации массивов, векторов и других контейнеров. В следующем примере `v2` инициализируется с тремя экземплярами `s`. `v3` инициализируется с тремя экземплярами `s`, которые сами по себе инициализируются с помощью фигурных скобок. Компилятор выводит тип каждого элемента на основе объявленного типа `v3`.

```

#include <vector>

struct S
{
    std::string name;
    float num;
    S(std::string s, float f) : name(s), num(f) {}
};

int main()
{
    // C-style initialization
    std::vector<S> v;
    S s1("Norah", 2.7);
    S s2("Frank", 3.5);
    S s3("Jeri", 85.9);

    v.push_back(s1);
    v.push_back(s2);
    v.push_back(s3);

    // Modern C++:
    std::vector<S> v2 {s1, s2, s3};

    // or...
    std::vector<S> v3{ {"Norah", 2.7}, {"Frank", 3.5}, {"Jeri", 85.9} };

}

```

Дополнительные сведения см. в разделе [Инициализация фигурными скобками](#).

Семантика перемещения

Современный C++ предоставляет *семантику перемещения*, что позволяет устраниить ненужное копирование памяти. В предыдущих версиях языка в определенных ситуациях копирования нельзя было избежать. Операция *перемещения* передает владение ресурсом от одного объекта к другому без создания копии. Некоторые классы владеют такими ресурсами, как память кучи, дескрипторы файлов и т. д. При реализации класса, владеющего ресурсами, можно определить для него *конструктор перемещения* и *оператор присваивания перемещения*. Компилятор выбирает эти специальные члены класса при разрешении перегрузки в ситуациях, когда копирование не требуется. Типы контейнеров стандартной библиотеки вызывают для объектов конструктор перемещения, если он определен.

Дополнительные сведения см. в разделе [Конструкторы перемещения и операторы присваивания перемещения \(C++\)](#).

Лямбда-выражения

В программировании в стиле С функцию можно передать в другую функцию с помощью *указателя функции*. Указатели функций неудобно поддерживать и сложно понимать. Функция, на которую они ссылаются, может быть определена в любом месте исходного кода, далеко от точки ее вызова. Кроме того, они не являются типобезопасными. Современный C++ предоставляет *объекты-функции* — классы, переопределяющие оператор `operator()`, который позволяет вызывать их как функцию. Наиболее удобный способ создания объектов-функций — встроенные *лямбда-выражения*. В следующем примере показано, как использовать лямбда-выражение для передачи объекта-функции, которую функция `for_each` будет вызывать для каждого элемента в векторе.

```
std::vector<int> v {1,2,3,4,5};  
int x = 2;  
int y = 4;  
auto result = find_if(begin(v), end(v), [=](int i) { return i > x && i < y; });
```

Лямбда-выражение `[=](int i) { return i > x && i < y; }` можно прочитать как "функция, которая принимает один аргумент типа `int` и возвращает логическое значение, указывающее, является ли аргумент больше `x` и меньше `y`". Обратите внимание, что переменные `x` и `y` из окружающего контекста можно использовать в лямбда-выражении. `[=]` указывает, что эти переменные *записываются* по значению, то есть лямбда-выражение имеет собственные копии этих значений.

Исключения

В современном C++ в качестве способа сообщить об ошибках и обработать их состояние отдается предпочтение исключениям, а не кодам ошибок. Дополнительные сведения см. в разделе [Современный подход к обработке исключений и ошибок в C++](#).

std::atomic

Используйте структуру и связанные типы `std::atomic` стандартной библиотеки C++ для механизмов взаимодействия между потоками.

std::variant (C++17)

Объединения обычно используются в программировании в стиле C для экономии памяти, позволяя членам разных типов занимать одно и то же расположение в памяти. Однако объединения не являются типобезопасными и могут быть подвержены ошибкам программирования. В C++ 17 появился класс `std::variant` в качестве более надежной и безопасной альтернативы объединениям. Функция `std::visit` может использоваться для доступа к членам типа `variant` типобезопасным способом.

См. также

[Справочник по языку C++](#)

[Лямбда-выражения](#)

[Стандартная библиотека C++](#)

[Соответствие стандартам языка Microsoft C/C++](#)

Лексические соглашения

12.11.2021 • 2 minutes to read

В этом разделе представлены базовые элементы программы на C++. Эти элементы (так называемые лексические элементы или токены) используются для построения операторов, определений, объявлений других компонентов, из которых состоит вся программа. В этом разделе описываются следующие лексические элементы:

- [Токены и кодировки](#)
- [Комментарии](#)
- [Идентификаторы](#)
- [Ключевые слова](#)
- [Символы пунктуации](#)
- [Числовые, логические литералы и литералы-указатели](#)
- [Строковые и символьные литералы](#)
- [Определенные пользователем литералы](#)

Дополнительные сведения о том, как выполняется синтаксический анализ исходных файлов C++, см. в разделе [этапы перевода](#).

См. также

[Справочник по языку C++](#)

[Единицы трансляции и компоновка](#)

Токены и кодировки

12.11.2021 • 3 minutes to read

Текст программы на языке C++ состоит из *маркеров* и пробелов. Токен — это наименьший элемент на C++, который имеет значение для компилятора. Средство синтаксического анализа C++ распознает следующие виды токенов:

- Ключевые слова
- Идентификаторы
- Числовые, логические литералы и константы указателей
- Строковые и символьные литералы
- Пользовательские литералы
- Инструкции
- Символы пунктуации

Маркеры обычно разделяются *пробелом*, что может быть одним или несколькими:

- Пустые значения
- Символы горизонтальной и вертикальной табуляции
- Символы перевода строки
- Веб-каналы форм
- Комментарии

Основная кодировка исходного кода

Стандарт C++ определяет основную кодировку *исходного кода*, которую можно использовать в исходных файлах. Для представления символов вне этого набора можно указывать дополнительные символы, используя *универсальные имена символов*. Реализация MSVC допускает дополнительные символы. Основная кодировка *исходного кода* состоит из 96 символов, которые могут использоваться в исходных файлах. Этот набор включает символ пробела, горизонтальной и вертикальной табуляции, управляемые символы перевода страницы и новой строки, а также следующий набор графических символов:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

```
А В С Д Е F Г Н І Ђ К Л М Н О Р І Ѓ Ђ Ђ Ђ Ђ Ђ Ђ Ђ
```

```
0 1 2 3 4 5 6 7 8 9
```

```
_ { } [ ] # ( ) < > % : ; . ? * + - / ^ & | ~ ! = , \ " '
```

Блок, относящийся только к системам Microsoft

MSVC включает `$` символ в качестве члена базовой кодировки исходного кода. MSVC также позволяет использовать дополнительный набор символов в исходных файлах в зависимости от кодировки файла. По умолчанию Visual Studio сохраняет исходные файлы, используя кодовую страницу по умолчанию. При сохранении исходных файлов с использованием кодовой страницы определенного языкового стандарта или кодовой страницы юникода MSVC позволяет использовать любой из символов этой кодовой страницы в исходном коде, за исключением кодов элементов управления, явно разрешенных в основной кодировке источника. Например, японские символы можно поместить в комментарии, идентификаторы или строковые литералы, если файл сохраняется с использованием кодовой страницы японского языка.

MSVC не допускают последовательностей символов, которые нельзя преобразовать в допустимые многобайтовые символы или кодовые точки Юникода. В зависимости от параметров компилятора не все допустимые символы могут отображаться в идентификаторах. Дополнительные сведения:
[Идентификаторы](#).

Завершение блока, относящегося только к системам Microsoft

универсальные имена символов

Поскольку программы на C++ могут использовать гораздо больше символов, чем указано в основной кодировке исходного кода, можно указать эти символы в переносимом виде, используя *универсальные имена символов*. Универсальное имя состоит из последовательности символов, представляющих кодовую точку Юникода. Оно может иметь две формы. Используйте `\UNNNNNNNN` для представления кодовой точки Юникода в форме U+NNNNNNNN, где NNNNNNNN — шестнадцатеричный номер кодовой точки из восьми цифр. Используйте код `\uNNNN` из четырех цифр для представления кодовой точки Юникода в форме U+0000NNNN.

Универсальные имена символов можно использовать в идентификаторах и в строковых и символьных литералах. Универсальное имя нельзя использовать для представления суррогатной кодовой точки в диапазоне от 0xD800 до 0xFFFF. Вместо этого используйте нужную кодовую точку: компилятор автоматически создает все необходимые суррогаты. К универсальным именам символов, которые могут использоваться в идентификаторах, применяются дополнительные ограничения. Дополнительные сведения см. в разделах [Identifiers](#) и [String and Character Literals](#).

Блок, относящийся только к системам Microsoft

Компилятор Microsoft C++ считает символ в форме универсального имени символа и литературной формы взаимозаменяемым. Например, можно объявить идентификатор, используя форму универсального имени символа, и использовать его в форме литерала:

```
auto \u30AD = 42; // \u30AD is 'ヰ'  
if (\u30AD == 42) return true; // \u30AD and ヰ are the same to the compiler
```

Формат расширенных символов в буфере обмена Windows зависит от параметров языкового стандарта приложения. Вырезание и вставка этих символов с переносом в код из другого приложения может вызвать использование непредвиденных кодировок. Это может привести к ошибкам синтаксического анализа без видимой причины в коде. Перед вставкой расширенных символов рекомендуется выбирать в качестве кодировки исходного файла кодовую страницу Юникода. Кроме того, для создания расширенных символов рекомендуется использовать IME или приложение "Таблица символов".

Завершение блока, относящегося только к системам Microsoft

Наборы символов выполнения

Наборы символов выполнения представляют символы и строки, которые могут присутствовать в скомпилированной программе. Эти наборы символов состоят из всех символов, разрешенных в исходном файле, а также управляющих символов, представляющих предупреждения, Backspace, возврат каретки и символ null. Кодировка выполнения имеет представление, определяемое языковым стандартом.

Комментарии (C++)

12.11.2021 • 2 minutes to read

Комментарий — это текст, который предназначен для программистов и не обрабатывается компилятором. Обычно комментарии используются для создания заметок к коду для дальнейшего использования. Компилятор обрабатывает их как пробел. Комментарии можно использовать для тестирования, чтобы сделать определенные строки кода неактивными. Однако `#if` / `#endif` директивы препроцессора работают лучше, поскольку можно окружить код, содержащий комментарии, но нельзя вкладывать комментарии.

Комментарии в C++ записываются одним из следующих способов:

- Символы `/*` (косая черта и звездочка), за которыми следует любая последовательность символов, включая переводы строки, после чего ставятся символы `*/`. Это тот же синтаксис, который используется в ANSI C.
- Символы `//` (две косые черты), за которыми следует любая последовательность символов. Символ перевода строки, непосредственно перед которым нет обратной косой черты, завершает комментарий, оформленный таким способом. Поэтому такие комментарии часто называют однострочными.

Символы, используемые для оформления комментариев (`/*`, `*/` и `//`), не имеют специального значения внутри символьной константы, строкового литерала, или комментария. Однако вложение комментариев, оформленных первым способом, не допускается.

См. также

[Лексические соглашения](#)

Идентификаторы (C++)

12.11.2021 • 3 minutes to read

Идентификатор — это последовательность символов, используемая для обозначения одного из следующих элементов:

- Имени объекта или переменной
- Имени класса, структуры или объединения
- Имени перечисленного типа
- Члена класса, структуры, объединения или перечисления
- Функции или функции члена класса
- Имени определения типа (typedef)
- Имени метки
- Имени макроса
- Параметра макроса

Следующие символы можно использовать в качестве любого символа идентификатора:

```
_ a b c d e f g h i j k l m  
n o p q r s t u v w x y z  
A B C D E F G H I J K L M  
N O P Q R S T U V W X Y Z
```

В идентификаторе также можно использовать определенные диапазоны универсальных имен символов. Универсальное имя в идентификаторе не может обозначать управляющий символ или символ в основной кодировке исходного кода. Дополнительные сведения см. в разделе [Character Sets](#). Следующие диапазоны номеров кодовых точек Юникода можно использовать как универсальные имена символов для любого символа в идентификаторе.

- 00A8, 00AA, 00AD, 00AF, 00B2-00B5, 00B7-00BA, 00BC-00BE, 00C0-00D6, 00D8-00F6, 00F8-00FF, 0100-02FF, 0370-167F, 1681-180D, 180F-1DBF, 1E00-1FFF, 200B-200D, 202A-202E, 203F-2040, 2054, 2060-206F, 2070-20CF, 2100-218F, 2460-24FF, 2776-2793, 2C00-2DFF, 2E80-2FFF, 3004-3007, 3021-302F, 3031-303F, 3040-D7FF, F900-FD3D, FD40-FDCF, FDF0-FE1F, FE30-FE44, FE47-FFFFD, 10000-1FFFFD, 20000-2FFFFD, 30000-3FFFFD, 40000-4FFFFD, 50000-5FFFFD, 60000-6FFFFD, 70000-7FFFFD, 80000-8FFFFD, 90000-9FFFFD, A0000-AFFFFD, B0000-BFFFFD, C0000-CFFFFD, D0000-DFFFFD, E0000-EFFFFD

Следующие символы могут быть любым символом в идентификаторе, кроме первого:

```
0 1 2 3 4 5 6 7 8 9
```

Следующие диапазоны номеров кодовых точек Юникода также можно использовать как универсальные имена символов для любого символа в идентификаторе, кроме первого:

- 0300-036F, 1DC0-1DFF, 20D0-20FF, FE20-FE2F

Значимыми являются только первые 2048 символов идентификаторов Microsoft C++. Для имен пользовательских типов компилятор создает "внутренние" имена, чтобы сохранить информацию о типе. Длина такого имени, включая информацию о типе, не может превышать 2048 символов. (Дополнительные сведения см. в разделе [декорированные имена](#).) Факторы, которые могут повлиять на длину декорированного идентификатора:

- Обозначает ли идентификатор объект пользовательского типа или типа, производного от пользовательского типа.
- Обозначает ли идентификатор функцию типа, производного от функции.
- Количество аргументов функции.

Знак доллара `$` — это допустимый символ идентификатора в компиляторе Microsoft C++ (MSVC). MSVC также позволяет использовать фактические символы, представленные допустимыми диапазонами универсальных имен символов в идентификаторах. Чтобы использовать эти символы, необходимо сохранить файл в той кодировке, которая включает эти символы. В этом примере показано, как можно взаимозаменямо использовать в коде расширенные символы и универсальные имена символов.

```
// extended_identifier.cpp
// In Visual Studio, use File, Advanced Save Options to set
// the file encoding to Unicode codepage 1200
struct テスト          // Japanese 'test'
{
    void トスト() {}   // Japanese 'toast'
};

int main() {
    テスト \u30D1\u30F3; // Japanese パン 'bread' in UCN form
    パン.トスト();      // compiler recognizes UCN or literal form
}
```

Диапазон разрешенных символов в идентификаторе шире, чем при компиляции кода C++/CLI. Идентификаторы в коде, скомпилированном с помощью /clr, должны соответствовать [стандарту ECMA-335: Common Language Infrastructure \(CLI\)](#).

Завершение блока, относящегося только к системам Майкрософт

Первый символ идентификатора должен быть алфавитным символом (в верхнем или нижнем регистре) или символом подчеркивания (`_`). Поскольку в идентификаторах C++ учитывается регистр, идентификаторы `fileName` и `FileName` различаются.

Идентификаторы не могут иметь то же написание и регистр, что и ключевые слова. Идентификаторы, в которых содержатся ключевые слова, являются допустимыми. Например, `Pint` является допустимым идентификатором, даже если он содержит `int` ключевое слово.

Использование двух последовательных символов подчеркивания (`__`) в идентификаторе или одиночный символ подчеркивания, за которым следует заглавная буква, зарезервировано для реализаций C++ во всех областях. В области видимости файла не следует использовать идентификаторы, начинающиеся с одного символа подчеркивания, за которым следует строчная буква. Это связано с возможными конфликтами с уже существующими или будущими зарезервированными идентификаторами.

См. также

[Лексические соглашения](#)

Ключевые слова (C++)

12.11.2021 • 3 minutes to read

Ключевые слова — это предварительно определенные зарезервированные идентификаторы, имеющие специальные значения. Их нельзя использовать в качестве идентификаторов в программе. Для Microsoft C++ зарезервированы следующие ключевые слова. Имена с начальными символами подчеркивания и именами, указанными для C++/CX и C++/CLI, являются расширениями Microsoft.

Стандартные ключевые слова C++

alignas
alignof
and^b
and_eq^b
asm объект
auto
bitand^b
bitor^b
bool
break
case
catch
char
char8_t^c
char16_t
char32_t
class
compl^b
concept^c
const
const_cast
constexpr^c
constinit^c
continue
co_await^c
co_return^c
co_yield^c
decltype
default
delete
do
double
dynamic_cast
else
enum

```
explicit
export c
extern
false
float
for
friend
goto
if
inline

int
long
mutable
namespace
new
noexcept
not6
not_eq6
nullptr
operator
or6
or_eq6
private
protected
public
register reinterpret_cast
requires c
return
short
signed
sizeof
static
static_assert

static_cast
struct
switch
template
this
thread_local
throw
true
try
typedef
typeid
typename
union
unsigned
using ПОВТОРНО using ИНСТРУКЦИ virtual
void
```

`volatile`
`wchar_t`
`while`
`xor`⁶
`xor_eq`⁶

`_asm` ключевое слово Майкрософт заменяет `asm` синтаксис C++. `asm` зарезервировано для совместимости с другими реализациями C++, но не реализовано. Используется `_asm` для встроенной сборки на целевых объектах x86. Microsoft C++ не поддерживает встроенную сборку для других целей.

^b расширенные синонимы оператора — это ключевые слова, если `/permissive-` заданы или `/Za` ([отключены языковые расширения](#)). Они не являются ключевыми словами при включении расширений Майкрософт.

^c поддерживается `/std:c++latest`, если указан.

Ключевые слова Microsoft C++

В C++ идентификаторы, содержащие два последовательных символа подчеркивания, зарезервированы для реализаций компилятора. Соглашение корпорации Майкрософт — перед ключевыми словами, зависящими от корпорации Майкрософт, с двойными символами подчеркивания. Эти слова невозможно использовать как имена идентификаторов.

Расширения Microsoft по умолчанию включены. Чтобы обеспечить полную переносимость программ, можно отключить расширения Майкрософт, указав `/permissive-` параметр или `/Za` ([Отключить расширения языка](#)) во время компиляции. Эти параметры отключают некоторые ключевые слова, относящиеся к Microsoft.

Если расширения Microsoft включены, в программах можно использовать ключевые слова, специфические для систем Microsoft. Для совместимости со стандартом ANSI эти ключевые слова начинаются с двух символов подчеркивания. Для обеспечения обратной совместимости поддерживаются версии с одним подчеркиванием многих ключевых слов с двойным подчеркиванием. `_cdecl` Ключевое слово доступно без символа подчеркивания в начале.

`_asm` Ключевое слово заменяет `asm` синтаксис C++. `asm` зарезервировано для совместимости с другими реализациями C++, но не реализовано. Используйте `_asm`.

`_based` Ключевое слово имеет ограниченное использование для целевых компиляций с 32-и 64-битным целевым объектом.

`_alignof`^д
`_asm`^д
`_assume`^д
`_based`^д
`_cdecl`^д
`_declspec`^д
`_event`
`_except`^д
`_fastcall`^д
`_finally`^д
`_forceinline`^д

`_hook`^г
`_if_exists`
`_if_not_exists`

```
_inline д
_int16 д
_int32 д
_int64 д
_int8 д
_interface
_leave д
_m128

_m128d
_m128i
_m64
_multiple_inheritance д
_ptr32 д
_ptr64 &
_raise
_restrict д
_single_inheritance &
_sptr &
_stdcall д

_super
_thiscall
_unaligned д
_unhook г
_uptr д
_uuidof д
_vectorcall д
_virtual_inheritance д
_w64 д
_wchar_t
```

Встроенная функция ^д, используемая при обработке событий.

^д. для обратной совместимости с предыдущими версиями эти ключевые слова доступны как с двумя символами подчеркивания в начале, так и с одним символом подчеркивания в начале, если включены расширения Майкрософт (по умолчанию).

Ключевые слова Майкрософт в модификаторах `_declspec`

Эти идентификаторы являются расширенными атрибутами для `_declspec` модификатора. Они считаются ключевыми словами в этом контексте.

```
align
allocate
allocator
appdomain
code_seg
deprecated

dllexport
dllimport
jitintrinsic
```

```
naked  
noalias  
noinline  
  
noreturn  
no_sanitize_address  
nothrow  
novtable  
process  
property  
  
restrict  
safebuffers  
selectany  
spectre  
thread  
uuid
```

Ключевые слова C++/CLI и C++/CX

```
__abstract f  
__box f  
__delegate f  
__gc f  
__identifier  
__nogc f  
__noop  
__pin f  
__property f  
__sealed f  
  
__try_cast f  
__value f  
abstract *  
array *  
as_friend  
delegate *  
enum class  
enum struct  
event *  
  
finally  
for each in  
gcnew *  
generic *  
initonly  
interface class *  
interface struct *  
interior_ptr *  
literal *  
  
new *
```

`property` *

`ref class`

`ref struct`

`safecast`

`sealed` *

`typeid`

`value class` *

`value struct` *

^f применим только к управляемые расширения для C++. В настоящее время использование этого синтаксиса не рекомендуется. Дополнительные сведения см. в статье [Расширения компонентов для платформ среды выполнения](#).

^g применимо к C++/CLI.

См. также

[Лексические соглашения](#)

[Встроенные операторы, приоритет и ассоциативность C++](#)

Знаки препинания (C++)

12.11.2021 • 2 minutes to read

Символы пунктуации в C++ имеют синтаксическое и семантическое значение для компилятора, однако сами по себе не указывают на операцию, которая позволяет получить значение. Некоторые символы пунктуации (по отдельности или в сочетании) могут также быть операторами C++ или иметь значение для препроцессора.

К символам пунктуации относятся следующие:

```
! % ^ & * ( ) - + = { } | ~  
[ ] \ ; ' : " < > ? , . / #
```

Символы пунктуации [], () и {} должны находиться в парах после [преобразования 4](#).

См. также раздел

[Лексические соглашения](#)

Числовые, логические литералы и литералы-указатели

12.11.2021 • 4 minutes to read

Литерал — это элемент программы, который непосредственно представляет значение. В этой статье рассматриваются литералы типа Integer, float-Point, Boolean и Pointer. Дополнительные сведения о строковых и символьных литералах см. в разделе [строковые и символьные литералы \(C++\)](#). Можно также определить собственные литералы на основе любой из этих категорий. Дополнительные сведения см. в разделе [определяемые пользователем литералы \(C++\)](#).

Литералы можно использовать во многих контекстах, но наиболее часто они используются для инициализации именованных переменных и для передачи аргументов в функции.

```
const int answer = 42;           // integer literal
double d = sin(108.87);         // floating point literal passed to sin function
bool b = true;                 // boolean literal
MyClass* mc = nullptr;          // pointer literal
```

Иногда важно указывать компилятору, как следует обрабатывать литерал или какой конкретный тип ему предоставить. Это делается путем добавления префиксов или суффиксов к литералу. Например, префикс `0x` указывает компилятору интерпретировать число, следующее за ним, как шестнадцатеричное значение, например `0x35`. `ULL` Суффикс указывает компилятору обрабатывать значение как `unsigned long long` тип, как в `5894345ULL`. Полный список префиксов и суффиксов для каждого типа литерала см. в следующих разделах.

Целочисленные литералы

Целочисленные литералы начинаются с цифры и не имеют дробных частей или экспонент.

Целочисленные литералы можно указать в десятичной, двоичной, восьмеричной или шестнадцатеричной форме. При необходимости можно указать целочисленный литерал как неподписанный, а в качестве типа `long` или `long long` — с помощью суффикса.

Если префикс или суффикс отсутствует, компилятор выдает целочисленный тип литерального значения `int` (32 бит), если значение подходит, в противном случае присваивает ему тип `long long` (64 бит).

Чтобы указать десятичный целочисленный литерал, начинайте спецификацию с любой цифры, кроме нуля. Пример:

```
int i = 157;           // Decimal literal
int j = 0198;          // Not a decimal number; erroneous octal literal
int k = 0365;          // Leading zero specifies octal literal, not decimal
int m = 36'000'000 // digit separators make large values more readable
```

Чтобы указать восьмеричный целочисленный литерал, начинайте спецификацию с нуля, за которым следует ряд цифр в диапазоне от 0 до 7. Цифры 8 и 9 при указании восьмеричного литерала будут ошибками. Пример:

```
int i = 0377; // Octal literal
int j = 0397; // Error: 9 is not an octal digit
```

Чтобы указать шестнадцатеричный целочисленный литерал, начните спецификацию с `0x` или `0X` (регистр «х не важен»), за которым следует последовательность цифр в диапазоне от `0` до `9` и `a` (или `A`) до `f` (или `F`). Шестнадцатеричные цифры от `a` (или `A`) до `f` (или `F`) представляют собой значения в диапазоне от 10 до 15. Пример:

```
int i = 0x3fff; // Hexadecimal literal
int j = 0X3FFF; // Equal to i
```

Чтобы указать тип без знака, используйте либо `u` суффикс, либо `U`. Чтобы указать тип `long`, используйте либо суффикс `l` либо `L`. Для указания 64-разрядного целочисленного типа используется суффикс `LL` или `ll`. Суффикс `l64` по-прежнему поддерживается, но не рекомендуется. Он относится только к Microsoft и не является переносимым. Пример:

```
unsigned val_1 = 328u; // Unsigned value
long val_2 = 0xFFFFFL; // Long value specified
// as hex literal
unsigned long val_3 = 0776745uL; // Unsigned long value
auto val_4 = 108LL; // signed long long
auto val_4 = 0x800000000000000ULL << 16; // unsigned long long
```

Разделители цифр. символ одинарной кавычки (апостроф) можно использовать для разделения значений в больших числах, чтобы облегчить чтение людям. Разделители не влияют на компиляцию.

```
long long i = 24'847'458'121
```

Литералы с плавающей запятой

Литералы с плавающей запятой задают значения, которые должны иметь дробную часть. Эти значения содержат десятичные разделители (`.`) и могут содержать экспоненты.

Литералы с плавающей запятой имеют **значащим** (иногда называется **мантиссой**), который указывает значение числа. Они имеют **показатель степени**, который указывает величину числа. И имеют необязательный суффикс, указывающий тип литерала. Значащим указывается как последовательность цифр, за которыми следует точка, за которой следует дополнительная последовательность цифр, представляющая дробную часть числа. Пример:

```
18.46
38.
```

Если указан показатель степени, он задает порядок числа в виде степени 10, как показано в следующем примере:

```
18.46e0 // 18.46
18.46e1 // 184.6
```

Показатель степени можно указать с помощью `e` или `E`, который имеет то же значение, за которым следует необязательный знак (+ или -) и последовательность цифр. Если указан показатель степени, десятичная точка в конце целых чисел не требуется, например `18E0`.

Литералы с плавающей запятой по умолчанию имеют тип `double`. Используя суффиксы или OR `f` `l` `F` `L` (суффикс не учитывает регистр), литерал можно указать как `float` или `long double`.

Хотя `long double` и `double` имеют одинаковое представление, они имеют разные типы. Например, можно использовать перегруженные функции, такие как

```
void func( double );
```

и

```
void func( long double );
```

логические литералы

Логические литералы: `true` и `false`.

Литерал-указатель (C++11)

C++ вводит `nullptr` литерал для указания нулевого инициализированного указателя. В переносимом коде `nullptr` следует использовать вместо целочисленного типа нуля или макросов, таких как `NULL`.

Двоичные литералы (C++14)

Двоичный литерал можно задать с помощью префикса `0b` или `0B` и последовательности, состоящей из 1 и 0:

```
auto x = 0B001101 ; // int
auto y = 0b000001 ; // int
```

Избегайте использования литералов как «магических констант»

Несмотря на то что это не всегда является хорошим стилем программирования, можно использовать литералы непосредственно в выражениях и операторах:

```
if (num < 100)
    return "Success";
```

В предыдущем примере рекомендуется использовать именованную константу, которая передает ясное значение, например `"MAXIMUM_ERROR_THRESHOLD"`. Если конечные пользователи видят возвращаемое значение `Success`, возможно, лучше использовать именованную строковую константу. Строковые константы можно хранить в одном месте в файле, который может быть локализован на другие языки. Использование именованных констант помогает обоим и другим пользователям понять смысл кода.

См. также

[Лексические соглашения](#)

[Строковые литералы C++](#)

[Определяемые пользователем литералы C++](#)

Строковые и символьные литералы (C++)

12.11.2021 • 16 minutes to read

В C++ поддерживаются различные типы строк и символов, а также доступны различные способы выражения значений литералов каждого из этих типов. В исходном коде содержимое символьных и строковых литералов выражается с помощью кодировки. Универсальные имена символов и escape-символы позволяют представить любую строку, используя только основную кодировку исходного кода. Необработанные строковые литералы позволяют не использовать escape-символы и могут применяться для выражения всех типов строковых литералов. Можно также создавать `std::string` литералы без необходимости выполнения дополнительных действий по созданию или преобразованию.

```
#include <string>
using namespace std::string_literals; // enables s-suffix for std::string literals

int main()
{
    // Character literals
    auto c0 = 'A'; // char
    auto c1 = u8'A'; // char
    auto c2 = L'A'; // wchar_t
    auto c3 = u'A'; // char16_t
    auto c4 = U'A'; // char32_t

    // Multicharacter literals
    auto m0 = 'abcd'; // int, value 0x61626364

    // String literals
    auto s0 = "hello"; // const char*
    auto s1 = u8"hello"; // const char*, encoded as UTF-8
    auto s2 = L"hello"; // const wchar_t*
    auto s3 = u"hello"; // const char16_t*, encoded as UTF-16
    auto s4 = U"hello"; // const char32_t*, encoded as UTF-32

    // Raw string literals containing unescaped \ and "
    auto R0 = R"("Hello \ world")"; // const char*
    auto R1 = u8R"("Hello \ world")"; // const char*, encoded as UTF-8
    auto R2 = LR"("Hello \ world")"; // const wchar_t*
    auto R3 = uR"("Hello \ world")"; // const char16_t*, encoded as UTF-16
    auto R4 = UR"("Hello \ world")"; // const char32_t*, encoded as UTF-32

    // Combining string literals with standard s-suffix
    auto S0 = "hello"s; // std::string
    auto S1 = u8"hello"s; // std::string
    auto S2 = L"hello"s; // std::wstring
    auto S3 = u"hello"s; // std::u16string
    auto S4 = U"hello"s; // std::u32string

    // Combining raw string literals with standard s-suffix
    auto S5 = R"("Hello \ world")"s; // std::string from a raw const char*
    auto S6 = u8R"("Hello \ world")"s; // std::string from a raw const char*, encoded as UTF-8
    auto S7 = LR"("Hello \ world")"s; // std::wstring from a raw const wchar_t*
    auto S8 = uR"("Hello \ world")"s; // std::u16string from a raw const char16_t*, encoded as UTF-16
    auto S9 = UR"("Hello \ world")"s; // std::u32string from a raw const char32_t*, encoded as UTF-32
}
```

Строковые литералы могут не иметь префикса или включать префиксы `u8`, `L`, `u` и `U` для обозначения кодировок обычных символов (однобайтовых или многобайтовых), UTF-8, расширенных символов (UCS-2 или UTF-16), UTF-16 и UTF-32, соответственно. Необработанный строковый литерал может иметь `R` `u8R`

`LR` префиксы „`uR`“ и `UR` для необработанных эквивалентов этих кодировок. Чтобы создать временные или статические `std::string` значения, можно использовать строковые литералы или необработанные строковые литералы с `s` суффиксом. Дополнительные сведения см. в разделе [строковые литералы](#) ниже. Дополнительные сведения о базовой кодировке исходного кода, универсальных именах символов и использовании символов из расширенных кодовых страниц в исходном коде см. в разделе [наборы символов](#).

Символьные литералы

Символьный литерал состоит из символьной константы. Он представляется символом, заключенным в одинарные кавычки. Существует пять типов символьных литералов:

- Обычные символьные литералы типа `char`, например `'a'`
- Символьные литералы UTF-8 типа `char` (`char8_t` в C++ 20), например `u8'a'`
- Литералы с расширенными символами типа `wchar_t`, например `L'a'`
- Символьные литералы UTF-16 типа `char16_t`, например `u'a'`
- UTF-32 символьные литералы типа `char32_t`, например `u'a'`

Символ, используемый для символьного литерала, может быть любым символом, за исключением символов обратной косой черты (`\`), одинарной кавычки (`'`) или новой строки. Зарезервированные символы можно указывать с помощью escape-последовательности. Символы можно указывать с помощью универсальных имен символов, при условии что тип является достаточно крупным для размещения символа.

Кодирование

Символьные литералы кодируются по-разному в соответствии с их префиксом.

- Символьный литерал без префикса является обычным символьным литералом. Значение обычного символьного литерала, содержащего один символ, escape-последовательность или универсальное имя символа, которое может быть представлено в наборе символов выполнения, имеет значение, равное числовому значению его кодировки в наборе символов выполнения. Обычный символьный литерал, содержащий более одного символа, escape-последовательности или универсального имени символа, является **многосимвольным литералом**. Многосимвольный литерал или обычный символьный литерал, который не может быть представлен в наборе символов выполнения `int`, имеет тип, а его значение определяется реализацией. дополнительные MSVC см. в разделе, [относящемся к корпорации майкрософт](#) ниже.
- Символьный литерал, начинающийся с `L` префикса, является литералом расширенных символов. Значение литерала расширенных символов, содержащего один символ, escape-последовательность или универсальное имя символа, имеет значение, равное числовому значению его кодировки в наборе расширенных символов выполнения, если только символьный литерал не имеет представления в наборе расширенных символов выполнения, в этом случае значение определяется реализацией. Значение литерала расширенных символов, содержащего несколько символов, escape-последовательностями или универсальных имен символов, определяется реализацией. дополнительные MSVC см. в разделе, [относящемся к корпорации майкрософт](#) ниже.
- Символьный литерал, начинающийся с `u8` префикса, является символьным литералом UTF-8. Значение символьного литерала UTF-8, содержащего один символ, escape-последовательность или универсальное имя символа, имеет значение, равное значению его кодовой точки ISO 10646, если оно может быть представлено в одной единице кода UTF-8 (соответствующее элементам управления C0 и основному регистру символов латиницы). Если значение не может быть

представлено одной единицей кода UTF-8, программа неправильно сформирована. Символьный литерал в кодировке UTF-8, содержащий более одного символа, escape-последовательности или универсального имени символа, имеет неправильный формат.

- Символьный литерал, начинающийся с `\u` префикса, является символьным литералом UTF-16. Значение символьного литерала UTF-16, содержащего один символ, escape-последовательность или универсальное имя символа, имеет значение, равное значению его кодовой точки ISO 10646, если оно может быть представлено одной единицей кода UTF-16 (соответствующей базовой многоязыковой плоскости). Если значение не может быть представлено одной единицей кода UTF-16, программа неправильно сформирована. Символьный литерал UTF-16, содержащий более одного символа, escape-последовательности или универсального имени символа, имеет неправильный формат.
- Символьный литерал, начинающийся с `\u` префикса, является символьным литералом UTF-32. Значение символьного литерала UTF-32, содержащего один символ, escape-последовательность или универсальное имя символа, имеет значение, равное значению кодовой точки ISO 10646. Символьный литерал в кодировке UTF-32, содержащий более одного символа, escape-последовательности или универсального имени символа, имеет неправильный формат.

Escape-последовательности

Существует три вида escape-последовательностей: простая, восьмеричная и шестнадцатеричная. Escape-последовательностями могут быть следующие значения:

ЗНАЧЕНИЕ	ESCAPE-ПОСЛЕДОВАТЕЛЬНОСТЬ
новая строка	<code>\n</code>
обратная косая черта	<code>\\"</code>
горизонтальная табуляция	<code>\t</code>
вопросительный знак	? или <code>\?</code>
вертикальная табуляция	<code>\v</code>
одинарная кавычка	<code>\'</code>
BACKSPACE	<code>\&</code>
двойная кавычка	<code>\"</code>
Возврат каретки	<code>\Серверный</code>
нуль-символ	<code>\0</code>
Смена страницы	<code>\ж</code>
восьмеричный	<code>\ooo</code>
оповещение (колокольчик)	<code>\a</code>
шестнадцатеричный	<code>\ксххх</code>

Восьмеричная escape-последовательность — это обратная косая черта, за которой следует

последовательность из одной до трех восьмеричных цифр. Восьмеричная escape-последовательность завершается на первом символе, который не является восьмеричной цифрой, если он встречается раньше, чем третья цифра. Наибольшее возможное восьмеричное значение — `\377`.

Шестнадцатеричная escape-последовательность — это обратная косая черта, за которой следует символ `x`, за которым следует последовательность из одной или нескольких шестнадцатеричных цифр.

Начальные нули пропускаются. В обычном или `U8` символьном литерале самое высокое шестнадцатеричное значение — `0xFF`. В расширенном символьном литерале с префиксом `L` или `U` максимальное шестнадцатеричное значение — `0xFFFF`. В расширенном символьном литерале с префиксом `U` максимальное шестнадцатеричное значение — `0xFFFFFFFF`.

В этом примере кода показаны некоторые примеры экранированных символов с помощью обычных символьных литералов. Один и тот же синтаксис escape-последовательности допустим для других типов символьных литералов.

```
#include <iostream>
using namespace std;

int main() {
    char newline = '\n';
    char tab = '\t';
    char backspace = '\b';
    char backslash = '\\';
    char nullChar = '\0';

    cout << "Newline character: " << newline << "ending" << endl;
    cout << "Tab character: " << tab << "ending" << endl;
    cout << "Backspace character: " << backspace << "ending" << endl;
    cout << "Backslash character: " << backslash << "ending" << endl;
    cout << "Null character: " << nullChar << "ending" << endl;
}
/* Output:
Newline character:
ending
Tab character: ending
Backspace character:ending
Backslash character: \ending
Null character: ending
*/
```

Обратная косая черта (`\`) — это символ продолжения строки, когда он помещается в конец строки. Если требуется, чтобы символ обратной косой черты отображался в виде символьного литерала, необходимо ввести две обратные косые черты в строке (`\\\`). Дополнительные сведения о символе продолжения строки см. в разделе [Phases of Translation](#).

Специально для систем Майкрософт

Чтобы создать значение из короткого многосимвольного литерала, компилятор преобразует символ или последовательность символов между одинарными кавычками в 8-битные значения в пределах 32-разрядного целого числа. Несколько символов в литерале заполняют соответствующие байты по мере необходимости от высокого до низкого порядка. Затем компилятор преобразует целое число в целевой тип после обычных правил. Например, чтобы создать `char` значение, компилятор принимает байт нижнего порядка. Чтобы создать `wchar_t` значение или `char16_t`, компилятор принимает слово низкого порядка. Компилятор выдает предупреждение о том, что результат усекается, если какие-либо биты заданы выше назначенного байта или слова.

```
char c0      = 'abcd';    // C4305, C4309, truncates to 'd'
wchar_t w0 = 'abcd';    // C4305, C4309, truncates to '\x6364'
int i0       = 'abcd';    // 0x61626364
```

Восьмеричная escape-последовательность, которая содержит более трех цифр, рассматривается как восьмеричная последовательность из 3 цифр, за которой следуют последующие цифры как символы в многосимвольном литерале, что может привести к неудивительному результату. Пример:

```
char c1 = '\100'; // '@'  
char c2 = '\1000'; // C4305, C4309, truncates to '0'
```

Escape-последовательности, которые содержат невосьмеричные символы, вычисляются в виде восьмеричной последовательности вплоть до последнего восьмеричного символа, за которыми следуют оставшиеся символы в виде последующих символов в многосимвольном литерале. Предупреждение C4125 создается, если первый невосьмеричный символ является десятичной цифрой. Пример:

```
char c3 = '\009'; // '9'  
char c4 = '\089'; // C4305, C4309, truncates to '9'  
char c5 = '\qrs'; // C4129, C4305, C4309, truncates to 's'
```

Восьмеричная escape-последовательность, которая имеет большее значение, чем [\377](#) Ошибка C2022: "значение-in-Decima": слишком большое для символа.

Escape-последовательность, которая содержит шестнадцатеричные и нешестнадцатеричные символы, вычисляется как многосимвольный литерал, содержащий шестнадцатеричную escape-последовательность вплоть до последнего шестнадцатеричного символа, за которыми следуют нешестнадцатеричные символы. Шестнадцатеричная escape-последовательность, которая не содержит шестнадцатеричных цифр, приводит к ошибке компилятора C2153: "шестнадцатеричные литералы должны содержать по крайней мере одну шестнадцатеричную цифру".

```
char c6 = '\x0050'; // 'P'  
char c7 = '\x0pqr'; // C4305, C4309, truncates to 'r'
```

Если в расширенном символьном литерале с префиксом `L` содержится последовательность из множества символов, значение берется из первого символа, а компилятор выдает предупреждение C4066. Последующие символы игнорируются, в отличие от поведения эквивалентного обычного многосимвольного литерала.

```
wchar_t w1 = L'\100'; // L'@'  
wchar_t w2 = L'\1000'; // C4066 L'@', 0 ignored  
wchar_t w3 = L'\009'; // C4066 L'\0', 9 ignored  
wchar_t w4 = L'\089'; // C4066 L'\0', 89 ignored  
wchar_t w5 = L'\qrs'; // C4129, C4066 L'q' escape, rs ignored  
wchar_t w6 = L'\x0050'; // L'P'  
wchar_t w7 = L'\x0pqr'; // C4066 L'\0', pqr ignored
```

Раздел, **относящийся к корпорации Майкрософт**, заканчивается здесь.

Универсальные имена символов

В символьных литералах и машинных (не являющихся необработанными) строковых литералах любой символ может быть представлен универсальным именем символа. Универсальные имена символов формируются с помощью префикса, `\u` за которым следует 8-значная кодовая точка Юникода или префикс, `\u` за которым следует 4-значная кодовая точка Юникода. Все восемь или четыре знака, соответственно, должны присутствовать для создания корректного универсального имени символа.

```
char u1 = 'A';           // 'A'
char u2 = '\101';        // octal, 'A'
char u3 = '\x41';        // hexadecimal, 'A'
char u4 = '\u00041';      // \u UCN 'A'
char u5 = '\U00000041'; // \U UCN 'A'
```

Суррогатные пары

Универсальные имена символов не могут кодировать значения в суррогатном диапазоне кодовых точек D800–DFFF. Для суррогатных пар Юникода укажите универсальное имя символа, используя `\UNNNNNNNN`, где NNNNNNNN — восьмизначная кодовая точка для символа. При необходимости компилятор создает суррогатную пару.

В C++03 языком допускалось, чтобы универсальными именами символов представлялось лишь определенное подмножество символов. Также могли существовать универсальные имена символов, не представляющие никаких допустимых символов Юникода. Эта ошибка была исправлена в стандарте C++ 11. В C++11 в символьных и строковых литералах и идентификаторах можно использовать универсальные имена символов. Дополнительные сведения об универсальных именах символов см. в разделе [Character Sets](#). Дополнительные сведения о Юникоде см. в статье [Unicode](#). Дополнительные сведения о суррогатных парах см. в статье [Surrogate Pairs and Supplementary Characters](#)(Суррогатные пары и дополнительные символы).

Строковые литералы

Строковый литерал представляет последовательность символов, которые вместе образуют строку с завершающим нулем. Символы должны быть заключены в двойные кавычки. Существуют следующие типы строковых литералов.

Узкие строковые литералы

Узким строковым литералом является нефиксированный, разделенный символами двойной кавычки массив типа `const char[n]`, где *n* — это длина массива в байтах. Обычный строковый литерал может содержать любые графические символы, за исключением двойных кавычек (`" "`), обратной косой черты (`\`) или символа новой строки. Обычный строковый литерал также может содержать перечисленные выше escape-последовательности и универсальные имена символов, которые помещаются в байте.

```
const char *narrow = "abcd";  
  
// represents the string: yes\no  
const char *escaped = "yes\\no";
```

Строки в кодировке UTF-8

Строка в кодировке UTF-8 — это U8 с двойной кавычкой, разделенный нулем массив типа `const char[n]`, где *n* — это длина закодированного массива в байтах. Строковый литерал с префиксом `u8` может содержать любые графические символы, за исключением двойных кавычек (`" "`), обратной косой черты (`\`) или символа новой строки. Строковый литерал с префиксом `u8` может также содержать перечисленные выше escape-последовательности и любые универсальные имена символов.

```
const char* str1 = u8"Hello World";
const char* str2 = u8"\U0001F607 is O:-)";
```

Широкие строковые литералы

Широкий строковый литерал — это массив констант, заканчивающийся нулем `wchar_t`, который имеет префикс `L` и содержит любой графический символ, кроме двойных кавычек (`" "`), обратной косой черты (`\`) или символа новой строки. Расширенный строковый литерал может содержать

перечисленные выше escape-последовательности и любые универсальные имена символов.

```
const wchar_t* wide = L"zyxw";
const wchar_t* newline = L"hello\ngoodbye";
```

char16_t и char32_t (C++11)

В C++ 11 введены переносимые `char16_t` (16-разрядные Юникод) и `char32_t` (32-разрядные Юникод) символы типа:

```
auto s3 = u"hello"; // const char16_t*
auto s4 = U"hello"; // const char32_t*
```

Необработанные строковые литералы (C++ 11)

Необработанный строковый литерал — это массив с завершающим нулем (любой символьный тип), содержащий любой графический символ, включая двойные кавычки (" "), обратную косую черту (\) или символ новой строки. Необработанные строковые литералы часто применяются в регулярных выражениях, которые используют классы символов, а также в строках HTML и XML. Примеры см. в следующей статье: [Bjarne Stroustrup's FAQ on C++11](#)(Вопросы и ответы о C++11 от Бьерна Страуструпа).

```
// represents the string: An unescaped \ character
const char* raw_narrow = R"(An unescaped \ character)";
const wchar_t* raw_wide = LR"(An unescaped \ character)";
const char* raw_utf8 = u8R"(An unescaped \ character";
const char16_t* raw_utf16 = uR"(An unescaped \ character)";
const char32_t* raw_utf32 = UR"(An unescaped \ character);
```

Разделитель — это определяемая пользователем последовательность длиной до 16 символов, которая непосредственно предшествует открывающей скобке необработанного строкового литерала и сразу после закрывающей скобки. Например, в `R"abc(Hello"\()abc"` последовательность разделителей — `abc`, а содержимое строки — `Hello"\()`. Разделители можно использовать для различия необработанных строк, содержащих двойные кавычки и круглые скобки. Этот строковый литерал вызывает ошибку компилятора:

```
// meant to represent the string: )
const char* bad_parens = R"()""; // error C2059
```

Однако ошибку можно устранить с помощью разделителя:

```
const char* good_parens = R"xyz()"")xyz";
```

Можно создать необработанный строковый литерал, содержащий символ новой строки (не экранированный символ) в источнике:

```
// represents the string: hello
//goodbye
const wchar_t* newline = LR"(hello
goodbye)";
```

литералы std:: String (C++ 14)

`std::string` литералы являются реализациями определяемых пользователем литералов в стандартной библиотеке (см. ниже), которые представлены как `"xyz"s` (с `s` суффиксом). Этот тип строкового

литерала создает временный объект типа `std::string`, `std::wstring`, `std::u32string` ИЛИ `std::u16string`, в зависимости от указанного префикса. Если префикс не используется, то создается `std::string`.

`L"xyz"s` создает `std::wstring`. `u"xyz"s` создает `std::u16string` И `U"xyz"s` создает `std::u32string`.

```
//#include <string>
//using namespace std::string_literals;
string str{ "hello"s };
string str2{ u8"Hello World" };
wstring str3{ L"hello"s };
u16string str4{ u"hello"s };
u32string str5{ U"hello"s };
```

Суффикс можно также использовать для необработанных строковых литералов:

```
u32string str6{ UR"(She said \"hello.\")"s };
```

`std::string` литералы определяются в пространстве имен `std::literals::string_literals` в `<string>` файле заголовка. Поскольку `std::literals::string_literals` и `std::literals` объявляются как **встроенные пространства имен**, `std::literals::string_literals` автоматически обрабатывается так, как если бы он принадлежал непосредственно в пространстве имен `std`.

Размер строковых литералов

Для `char*` строк ANSI и других однобайтовых кодировок (но не UTF-8) размер строкового литерала (в байтах) — это число символов плюс 1 для завершающего нуль-символа. Для всех других типов строк размер не строго связан с числом символов. UTF-8 использует до четырех `char` элементов для кодирования некоторых единиц кода, а `char16_t` или `wchar_t` кодирования UTF-16 может использовать два элемента (всего четыре байта) для кодирования одной единицы кода. В примере ниже показан размер расширенного строкового литерала в байтах.

```
const wchar_t* str = L"Hello!";
const size_t byteSize = (wcslen(str) + 1) * sizeof(wchar_t);
```

Обратите внимание, что `strlen()` и `wcslen()` не включайте размер завершающего нуль-символа, размер которого равен размеру элемента строкового типа: один байт в `char*` `char8_t*` строке или, два байта `wchar_t*` или `char16_t*` строки и четыре байта в `char32_t*` строках.

Максимальная длина строкового литерала составляет 65 535 байт. Это ограничение применимо как к узким, так и к расширенным строковым литералам.

Изменение строковых литералов

Поскольку строковые литералы (не включая `std::string` литералы) являются константами, попытка их изменить, например, `str[2] = 'A'` приводит к ошибке компилятора.

Специально для систем Майкрософт

В Microsoft C++ можно использовать строковый литерал для инициализации указателя на `non-const char` или `wchar_t`. Эта неконстантная инициализация разрешена в коде C99, но не рекомендуется в C++ 98 и удалена в C++ 11. Попытка изменить строку вызовет нарушение прав доступа, как показано в следующем примере:

```
wchar_t* str = L"hello";
str[2] = L'a'; // run-time error: access violation
```

Если задать параметр компилятора `/Zc:strictStrings` (отключить преобразование типов строковых

[литералов](#) , то при преобразовании строкового литерала в указатель неконстантного символа компилятор может выдать ошибку. Рекомендуется использовать его для создания переносимого кода, соответствующего стандартам. Также рекомендуется использовать `auto` ключевое слово для объявления инициализированных указателей строкового литерала, так как он разрешается в правильный (`const`) тип. В следующем примере кода перехватывается во время компиляции попытка записать в строковый литерал:

```
auto str = L"hello";
str[2] = L'a'; // C3892: you cannot assign to a variable that is const.
```

В некоторых случаях идентичные строковые литералы могут быть объединены в пул для экономии места в исполняемом файле. При объединении строковых литералов в пулы компилятор делает так, что все ссылки на определенный строковый литерал указывают на одну и ту же область в памяти, вместо того чтобы каждая ссылка указывала на отдельный экземпляр строкового литерала. Чтобы включить объединение строк, используйте `/GF` параметр компилятора.

Раздел, [относящийся к корпорации Майкрософт](#) , заканчивается здесь.

Сцепление смежных строковых литералов

Все смежные расширенные и узкие строковые литералы соединяются. Данное объявление:

```
char str[] = "12" "34";
```

идентично следующему объявлению:

```
char atr[] = "1234";
```

и следующему объявлению:

```
char atr[] = "12\
34";
```

Использование внедренных шестнадцатеричных escape-кодов для задания строковых литералов может привести к непредвиденным результатам. В следующем примере выполняется попытка создать строковый литерал, содержащий символ ASCII 5, за которым следуют символы f, i, v и e:

```
"\x05five"
```

Фактический результат (шестнадцатеричное значение 5F) является кодом ASCII для символа подчеркивания, за которым следуют символы i, v и e. Чтобы получить правильный результат, можно использовать одну из следующих escape-последовательностей:

```
"\005five"      // Use octal literal.
"\x05" "five"   // Use string splicing.
```

`std::string` литералы, так как они `std::string` являются типами, могут быть объединены с `+` оператором, определенным для `basic_string` типов. Эти литералы также можно соединить аналогично смежным строковым литералам. В обоих случаях кодировка строки и суффикс должны совпадать:

```
auto x1 = "hello" " " " world"; // OK
auto x2 = U"hello" " " L"world"; // C2308: disagree on prefix
auto x3 = u8"hello" " "s u8"world"s; // OK, agree on prefixes and suffixes
auto x4 = u8"hello" " "s u8"world"z; // C3688, disagree on suffixes
```

Строковые литералы с универсальными именами символов

Машинные (не являющиеся необработанными) строковые литералы могут использовать универсальные имена символов для представления любого символа, при условии что универсальные имена можно кодировать как один или несколько символов в строковом типе. Например, универсальное имя символа, представляющее расширенный символ, не может быть закодировано в виде короткой строки с помощью кодовой страницы ANSI, но может быть закодировано в виде узких строк в некоторых многобайтовых кодовых страницах или в строках UTF-8 или в расширенной строке. В C++ 11 Поддержка Юникода расширена с помощью `char16_t*` `char32_t*` строковых типов и.

```
// ASCII smiling face
const char*     s1 = ":-)";

// UTF-16 (on Windows) encoded WINKING FACE (U+1F609)
const wchar_t*   s2 = L"\u0001F609 is ;-)";

// UTF-8 encoded SMILING FACE WITH HALO (U+1F607)
const char*     s3 = u8"\u0001F607 is O:-)";

// UTF-16 encoded SMILING FACE WITH OPEN MOUTH (U+1F603)
const char16_t*  s4 = u"\u0001F603 is :-D";

// UTF-32 encoded SMILING FACE WITH SUNGLASSES (U+1F60E)
const char32_t*  s5 = U"\u0001F60E is B-)";
```

См. также:

[Наборы символов](#)

[Числовые, логические литералы и константы указателей](#)

[Пользовательские литералы](#)

Определенные пользователем литералы

12.11.2021 • 6 minutes to read

В C++ имеется шесть основных категорий литералов: целое число, символ, число с плавающей запятой, строка, логическое значение и указатель. Начиная с C++ 11, можно определить собственные литералы на основе этих категорий, чтобы предоставить синтаксические сочетания клавиш для общих идиом и повышения безопасности типов. Например, предположим, что у вас есть `Distance` класс. Вы можете определить литерал для километра, а другой — для миль, и порекомендовать пользователю явно указать единицы измерения, написав: `auto d = 42.0_km` или `auto d = 42.0_mi`. Нет преимуществ в производительности или недостатков для пользовательских литералов; они предназначены в первую очередь для удобства или для выведения типа во время компиляции. Стандартная библиотека содержит пользовательские литералы для `std::string`, для `std::complex` и для единиц в операциях времени и длительности в `<chrono>` заголовке:

```
Distance d = 36.0_mi + 42.0_km;           // Custom UDL (see below)
std::string str = "hello"s + "World"s;    // Standard Library <string> UDL
complex<double> num =
    (2.0 + 3.01i) * (5.0 + 4.3i);       // Standard Library <complex> UDL
auto duration = 15ms + 42h;              // Standard Library <chrono> UDLs
```

Сигнатуры определяемых пользователем литеральных операторов

Определяемый пользователем литерал реализуется путем определения **оператора ""** в области пространства имен с помощью одной из следующих форм:

```
ReturnType operator ""_a(unsigned long long int);   // Literal operator for user-defined INTEGRAL literal
ReturnType operator ""_b(long double);               // Literal operator for user-defined FLOATING literal
ReturnType operator ""_c(char);                      // Literal operator for user-defined CHARACTER literal
ReturnType operator ""_d(wchar_t);                   // Literal operator for user-defined CHARACTER literal
ReturnType operator ""_e(char16_t);                  // Literal operator for user-defined CHARACTER literal
ReturnType operator ""_f(char32_t);                  // Literal operator for user-defined CHARACTER literal
ReturnType operator ""_g(const char*, size_t);       // Literal operator for user-defined STRING literal
ReturnType operator ""_h(const wchar_t*, size_t);     // Literal operator for user-defined STRING literal
ReturnType operator ""_i(const char16_t*, size_t);   // Literal operator for user-defined STRING literal
ReturnType operator ""_g(const char32_t*, size_t);   // Literal operator for user-defined STRING literal
ReturnType operator ""_r(const char*);                // Raw literal operator
template<char...> ReturnType operator ""_t();        // Literal operator template
```

Имена операторов, использованные в предыдущем примере, можно заменить на любые другие. Обязательным является лишь символ подчеркивания в начале. (Только стандартная библиотека может определять литералы без подчеркивания.) Тип возвращаемого значения — это место, где вы настраиваете преобразование или другие операции, выполняемые литералом. Кроме того, любой из этих операторов можно определить как `constexpr`.

Литералы с обработкой

В исходном коде любой литерал, определяемый пользователем или нет, по сути представляет собой последовательность буквенно-цифровых символов, например `101`, или `54.7` `"hello"` `true`.

Компилятор интерпретирует последовательность как целое число, float, константа `char * String` и т. д.

Определяемый пользователем литерал, принимающий в качестве входных данных любой тип,

присвоенный литеральному значению, формально называется *литералом обработанные*. Все операторы, описанные выше, за исключением `_r` и `_t`, являются литералами с обработкой. Например, литерал `42.0_km` будет привязан к оператору с именем `_km`, имеющему сигнатуру похожую на `_b`, а литерал `42_km` будет привязан к оператору с сигнатурой похожей на `_a`.

В следующем примере показано, как определяемые пользователем литералы могут потребовать от пользователей явно указывать входные данные. Чтобы создать `Distance`, пользователь должен явно указать километры или мили с помощью соответствующего пользовательского литерала. Такой же результат можно получить другими способами, но определенные пользователем литералы менее детализированы, чем альтернативные варианты.

```

// UDL_Distance.cpp

#include <iostream>
#include <string>

struct Distance
{
private:
    explicit Distance(long double val) : kilometers(val)
    {}

    friend Distance operator"" _km(long double val);
    friend Distance operator"" _mi(long double val);

    long double kilometers{ 0 };
public:
    const static long double km_per_mile;
    long double get_kilometers() { return kilometers; }

    Distance operator+(Distance other)
    {
        return Distance(get_kilometers() + other.get_kilometers());
    }
};

const long double Distance::km_per_mile = 1.609344L;

Distance operator"" _km(long double val)
{
    return Distance(val);
}

Distance operator"" _mi(long double val)
{
    return Distance(val * Distance::km_per_mile);
}

int main()
{
    // Must have a decimal point to bind to the operator we defined!
    Distance d{ 402.0_km }; // construct using kilometers
    std::cout << "Kilometers in d: " << d.get_kilometers() << std::endl; // 402

    Distance d2{ 402.0_mi }; // construct using miles
    std::cout << "Kilometers in d2: " << d2.get_kilometers() << std::endl; // 646.956

    // add distances constructed with different units
    Distance d3 = 36.0_mi + 42.0_km;
    std::cout << "d3 value = " << d3.get_kilometers() << std::endl; // 99.9364

    // Distance d4(90.0); // error constructor not accessible

    std::string s;
    std::getline(std::cin, s);
    return 0;
}

```

Литеральное число должно иметь десятичное значение. В противном случае число будет интерпретировано как целое число, а тип не будет совместим с оператором. Для ввода с плавающей запятой тип должен иметь значение `long double`, а для целочисленных типов — `long long`.

Необработанные литералы

В необработанном определяемом пользователем литерале оператор, который вы определяете,

принимает литерал как последовательность значений Char. Вы можете интерпретировать эту последовательность как число или строку или другой тип. В списке операторов, приведенном ранее на этой странице, есть операторы `_r` и `_t`, которые можно использовать для определения необработанных литералов:

```
ReturnType operator "" _r(const char*);           // Raw literal operator
template<char...> ReturnType operator "" _t();      // Literal operator template
```

Необработанные литералы можно использовать для предоставления пользовательской интерпретации входной последовательности, которая отличается от нормальной работы компилятора. Например, вы можете определить литерал, преобразующий последовательность `4.75987` в пользовательский тип десятичного числа вместо типа с плавающей запятой по стандарту IEEE 754. Необработанные литералы, такие как литералы обработанные, можно также использовать для проверки входных последовательностей во время компиляции.

Пример: ограничения необработанных литералов

Оператор необработанного литерала и шаблон оператора литерала работают только с пользовательскими литералами, имеющими целочисленный тип или тип числа с плавающей запятой, как показано в следующем примере.

```
#include <cstddef>
#include <cstdio>

// Literal operator for user-defined INTEGRAL literal
void operator "" _dump(unsigned long long int lit)
{
    printf("operator \"\" _dump(unsigned long long int) : ===>%llu<===\n", lit);
};

// Literal operator for user-defined FLOATING literal
void operator "" _dump(long double lit)
{
    printf("operator \"\" _dump(long double) : ===>%Lf<===\n", lit);
};

// Literal operator for user-defined CHARACTER literal
void operator "" _dump(char lit)
{
    printf("operator \"\" _dump(char) : ===>%c<===\n", lit);
};

void operator "" _dump(wchar_t lit)
{
    printf("operator \"\" _dump(wchar_t) : ===>%d<===\n", lit);
};

void operator "" _dump(char16_t lit)
{
    printf("operator \"\" _dump(char16_t) : ===>%d<===\n", lit);
};

void operator "" _dump(char32_t lit)
{
    printf("operator \"\" _dump(char32_t) : ===>%d<===\n", lit);
};

// Literal operator for user-defined STRING literal
void operator "" _dump(const char* lit, size_t)
{
    printf("operator \"\" _dump(const char*, size_t): ===>%s<===\n", lit);
};
```

```

void operator "" _dump(const wchar_t* lit, size_t)
{
    printf("operator \"\" _dump(const wchar_t*, size_t): ===>%ls<===%n", lit);
}

void operator "" _dump(const char16_t* lit, size_t)
{
    printf("operator \"\" _dump(const char16_t*, size_t):\n" );
}

void operator "" _dump(const char32_t* lit, size_t)
{
    printf("operator \"\" _dump(const char32_t*, size_t):\n" );
}

// Raw literal operator
void operator "" _dump_raw(const char* lit)
{
    printf("operator \"\" _dump_raw(const char*) : ===>%s<===%n", lit);
}

template<char...> void operator "" _dump_template(); // Literal operator template

int main(int argc, const char* argv[])
{
    42_dump;
    3.1415926_dump;
    3.14e+25_dump;
    'A'_dump;
    L'B'_dump;
    u'C'_dump;
    U'D'_dump;
    "Hello World"_dump;
    L"Wide String"_dump;
    u8"UTF-8 String"_dump;
    u"UTF-16 String"_dump;
    U"UTF-32 String"_dump;
    42_dump_raw;
    3.1415926_dump_raw;
    3.14e+25_dump_raw;

    // There is no raw literal operator or literal operator template support on these types:
    // 'A'_dump_raw;
    // L'B'_dump_raw;
    // u'C'_dump_raw;
    // U'D'_dump_raw;
    // "Hello World"_dump_raw;
    // L"Wide String"_dump_raw;
    // u8"UTF-8 String"_dump_raw;
    // u"UTF-16 String"_dump_raw;
    // U"UTF-32 String"_dump_raw;
}

```

```
operator "" _dump(unsigned long long int) : ====>42<===
operator "" _dump(long double) : ====>3.141593<===
operator "" _dump(long double) : ====>3139999999999998506827776.000000<===
operator "" _dump(char) : ====>A<===
operator "" _dump(wchar_t) : ====>66<===
operator "" _dump(char16_t) : ====>67<===
operator "" _dump(char32_t) : ====>68<===
operator "" _dump(const char*, size_t): ====>Hello World<===
operator "" _dump(const wchar_t*, size_t): ====>Wide String<===
operator "" _dump(const char*, size_t): ====>UTF-8 String<===
operator "" _dump(const char16_t*, size_t):
operator "" _dump(const char32_t*, size_t):
operator "" _dump_raw(const char*) : ====>42<===
operator "" _dump_raw(const char*) : ====>3.1415926<===
operator "" _dump_raw(const char*) : ====>3.14e+25<===
```

Основные понятия (C++)

12.11.2021 • 2 minutes to read

В этом разделе рассматриваются понятия, которые абсолютно необходимы для понимания C++. Программистам, работающим с языком С, большинство из них уже знакомы, однако ряд незначительных отличий в них могут привести к тому, что программы будут порождать непрогнозируемый результат. В этой статье содержатся следующие разделы:

- [Система типов C++](#)
- [Область](#)
- [Единицы трансляции и компоновка](#)
- [Функция main и аргументы командной строки](#)
- [Завершение программы](#)
- [Значения Lvalue и Rvalue](#)
- [Временные объекты](#)
- [Выравнивание](#)
- [Тривиальный, Стандартный-макет и типы POD](#)

См. также

[Справочник по языку C++](#)

Система типов C++

12.11.2021 • 11 minutes to read

Концепция *типа* очень важна в C++. Каждая переменная, аргумент функции и возвращаемое значение функции должны иметь тип, чтобы их можно было скомпилировать. Кроме того, перед вычислением каждого выражения (включая литеральные значения) компилятор неявно назначает ему тип. Некоторые примеры типов включают `int` в себя хранение целочисленных значений `double` для хранения значений с плавающей запятой (также известных как *скалярные типы данных*) или класс стандартной библиотеки `std::basic_string` для хранения текста. Вы можете создать собственный тип, определив `class` или `struct`. Тип определяет объем памяти, выделяемой для переменной (или результата выражения), типы значений, которые могут храниться в этой переменной, способ интерпретации значений (например, битовые шаблоны), и операции, допустимые с переменной. Эта статья содержит неформальный обзор основных особенностей системы типов C++.

Терминология

Переменная: символическое имя количества данных, чтобы имя можно было использовать для доступа к данным, на которые он ссылается в области кода, где он определен. В C++ *переменная* обычно используется для ссылки на экземпляры скалярных типов данных, тогда как экземпляры других типов обычно называются *объектами*.

Объект. для простоты и согласованности в этой статье используется *объект* term для ссылки на любой экземпляр класса или структуры, и когда он используется в общем смысле, включает все типы, даже скалярные переменные.

Тип POD (обычные старые данные): Эта неофициальная Категория типов данных в C++ относится к скалярным типам (см. раздел фундаментальные типы) или к *классам Pod*. Класс POD не содержит статических данных-членов, которые не являются типами POD, а также не содержит пользовательских конструкторов, пользовательских деструкторов или пользовательских операторов присваивания. Кроме того, класс POD не имеет виртуальных функций, базового класса и ни закрытых, ни защищенных нестатических данных-членов. Типы POD часто используются для внешнего обмена данными, например с модулем, написанным на языке С (в котором имеются только типы POD).

Указание типов переменных и функций

C++ — это *строго типизированный* язык, который также является *статически типизированным*. Каждый объект имеет тип, и этот тип никогда не изменяется (не следует путать с статическими объектами данных). При объявлении переменной в коде необходимо либо явно указать ее тип, либо использовать `auto` ключевое слово, чтобы указать компилятору вывести тип из инициализатора. При объявлении функции в коде необходимо указать тип каждого аргумента и его возвращаемое значение или `void` значение, если функция не возвращает никакого значения. Исключением является использование шаблонов функции, которые допускают аргументы произвольных типов.

После объявления переменной изменить ее тип впоследствии уже невозможно. Однако можно скопировать значения переменной или возвращаемое значение функции в другую переменную другого типа. Такие операции называются *преобразованиями типов*, которые иногда являются обязательными, но также являются потенциальными источниками потери или неправильности данных.

При объявлении переменной типа POD настоятельно рекомендуется инициализировать ее, т. е. указать начальное значение. Пока переменная не инициализирована, она имеет "мусорное" значение,

определенное значениями битов, которые ранее были установлены в этом месте памяти. Необходимо учитывать эту особенность языка C++, особенно при переходе с другого языка, который обрабатывает инициализацию автоматически. При объявлении переменной типа, не являющейся классом POD, инициализация обрабатывается конструктором.

В следующем примере показано несколько простых объявлений переменных с небольшим описанием для каждого объявления. В примере также показано, как компилятор использует сведения о типе, чтобы разрешить или запретить некоторые последующие операции с переменной.

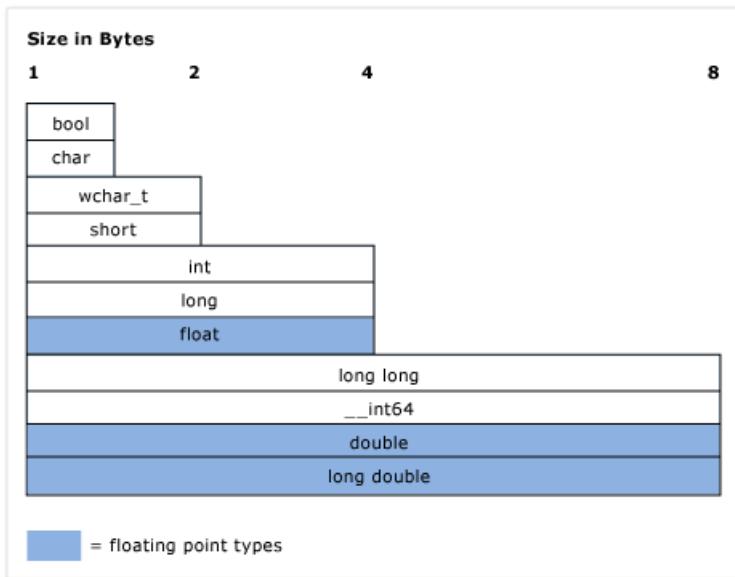
```
int result = 0;           // Declare and initialize an integer.
double coefficient = 10.8; // Declare and initialize a floating
                           // point value.
auto name = "Lady G.";    // Declare a variable and let compiler
                           // deduce the type.
auto address;             // error. Compiler cannot deduce a type
                           // without an initializing value.
age = 12;                 // error. Variable declaration must
                           // specify a type or use auto!
result = "Kenny G.";      // error. Can't assign text to an int.
string result = "zero";    // error. Can't redefine a variable with
                           // new type.
int maxValue;              // Not recommended! maxValue contains
                           // garbage bits until it is initialized.
```

Базовые (встроенные) типы

В отличие от некоторых других языков, в C++ нет универсального базового типа, от которого наследуются все остальные типы. Язык включает множество *фундаментальных типов*, известных также как *встроенные типы*. К ним относятся такие числовые типы `int`, как `double`, `long`, `bool`, а также `char`, `wchar_t` типы и для символов ASCII и Юникода соответственно. Большинство интегральных фундаментальных типов (за исключением `bool`, `double`, `wchar_t` типов, и связанных) имеют `unsigned` версии, которые изменяют диапазон значений, которые может хранить переменная. Например, объект `int`, хранящий 32-разрядное целое число со знаком, может представлять значение от -2 147 483 648 до 2 147 483 647. Объект `unsigned int`, который также хранится как 32-бит, может хранить значение от 0 до 4 294 967 295. Общее количество возможных значений в каждом случае одинаково, отличается только диапазон.

Базовые типы распознаются компилятором, в котором предусмотрены встроенные правила, управляющие операциями, выполняемыми с такими типами, а также преобразованием в другие базовые типы. Полный список встроенных типов, а также их размер и числовые ограничения см. в разделе [Встроенные типы](#).

На следующем рисунке показаны относительные размеры встроенных типов в реализации Microsoft C++:



В следующей таблице перечислены наиболее часто используемые фундаментальные типы и их размеры в реализации Microsoft C++:

ТИП	РАЗМЕР	КОММЕНТИРОВАТЬ
<code>int</code>	4 байта	Выбор по умолчанию для целочисленных значений.
<code>double</code>	8 байт	Выбор по умолчанию для значений с плавающей запятой.
<code>bool</code>	1 байт	Представляет значения, которые могут быть или <code>true</code> , или <code>false</code> .
<code>char</code>	1 байт	Используйте для символов ASCII в старых строках в стиле C или в объектах <code>std::string</code> , которые никогда не будут преобразовываться в Юникод.
<code>wchar_t</code>	2 байта	Представляет "расширенные" символы, которые могут быть представлены в формате Юникод (UTF-16 в Windows, в других операционных системах возможно другое представление). Это символьный тип, используемый в строках типа <code>std::wstring</code> .
<code>unsigned char</code>	1 байт	C++ не имеет встроенного типа Byte. Используется <code>unsigned char</code> для представления байтового значения.
<code>unsigned int</code>	4 байта	Вариант по умолчанию для битовых флагов.
<code>long long</code>	8 байт	Представляет очень большие целочисленные значения.

Другие реализации C++ могут использовать разные размеры для определенных числовых типов. Дополнительные сведения о размерах и отношениях размеров, необходимых стандарту C++, см. в разделе [Встроенные типы](#).

Тип void

`void` Тип является специальным типом; нельзя объявить переменную типа `void`, но можно объявить переменную типа `void *` (указатель на `void`), которая иногда необходима при выделении необработанной памяти (нетипизированной). Однако указатели на `void` не являются строго типизированными и обычно их использование не рекомендуется в современных C++. В объявлении функции `void` возвращаемое значение означает, что функция не возвращает значение. Это общее и допустимое использование `void`. Хотя в языке С обязательны функции, которые имеют нулевые параметры для объявления `void` в списке параметров, `fou(void)` такой подход не рекомендуется в современных C++ и должен быть объявлен `fou()`. Дополнительные сведения см. в разделе [преобразования типов и типизация](#).

Квалификатор типа const

Любой встроенный или пользовательский тип может квалифицироваться ключевым словом `const`. Кроме того, функции-члены могут быть `const` полными и даже `const` перегруженными. Значение `const` типа не может быть изменено после инициализации.

```
const double PI = 3.1415;
PI = .75 //Error. Cannot modify const variable.
```

`const` Квалификатор широко используется в объявлениях функций и переменных, и "правильность констант" является важной концепцией в C++; по сути это означает `const`, что во время компиляции эти значения не изменяются случайным образом. Дополнительные сведения см. на веб-сайте [const](#).

`const` Тип отличается от неконстантной версии, например, имеет тип, отличный `const int` от `int`. Оператор C++ можно использовать `const_cast` в редких случаях, когда необходимо удалить *const-rvalue характеристики* из переменной. Дополнительные сведения см. в разделе [преобразования типов и типизация](#).

Строковые типы

Строго говоря, язык C++ не имеет встроенного строкового типа; `char` и `wchar_t` хранения одиночных символов. необходимо объявить массив этих типов для приблизительной строки, добавив завершающее значение null (например, ASCII `'\0'`) к элементу массива, который находится за последним допустимым символом (также называется *строкой в стиле C*). Строки в стиле C требовали написания гораздо большего объема кода или использования внешних библиотек служебных функций. Но в современных C++ у нас есть стандартные библиотеки типов `std::string` (для символьных строк 8-разрядных `char` типов) или `std::wstring` (для строк символов 16-разрядного `wchar_t` типа). Эти контейнеры стандартной библиотеки C++ можно рассматривать как собственные строковые типы, так как они являются частью стандартных библиотек, включенных в совместимую среду сборки C++. Просто используйте директиву `#include <string>`, чтобы эти типы были доступны в программе. (Если используется MFC или ATL, `CString` класс также доступен, но не является частью стандарта C++.) Использование массивов символов, заканчивающихся нулем (приведенных выше строк в стиле C), не рекомендуется в современных C++.

Определяемые пользователем типы

При определении,, `class` `struct` `union` ИЛИ `enum`, эта конструкция используется в остальной части кода так, как если бы это был фундаментальный тип. Он имеет известный размер в памяти, и в его отношении действуют определенные правила проверки во время компиляции и во время выполнения в течение срока использования программы. Основные различия между базовыми встроенными типами и пользовательскими типами указаны ниже:

- Компилятор не имеет встроенных сведений о пользовательском типе. Он узнает о типе при первом обнаружении определения во время процесса компиляции.
- Пользователь определяет, какие операции можно выполнять с типом и как его можно преобразовать в другие типы, задавая (через перегрузку) соответствующие операторы, либо в виде членов класса, либо в виде функций, не являющихся членами. Дополнительные сведения см. в разделе [перегрузка функций](#).

ТИПЫ УКАЗАТЕЛЕЙ

Датировано назад к самой ранней версии языка С, С++ позволяет объявить переменную типа указателя с помощью специального декларатора `*` (звездочка). Тип указателя хранит адрес расположения в памяти, в котором хранится фактическое значение данных. В современных С++ они называются *необработанными указателями* и доступны в коде с помощью специальных операторов `*` (звездочки) или `->` (тире с символом «больше»). Это называется *разыменованием*, и какой из используемых объектов зависит от того, выполняется ли разыменование указателя на скаляр или указатель на член в объекте. Работа с типами указателя долгое время была одним из наиболее трудных и непонятных аспектов разработки программ на языках С и С++. В этом разделе приводятся некоторые факты и рекомендации по использованию необработанных указателей, если вы хотите, но в современной версии С++ больше не требуется (или рекомендуется) использовать необработанные указатели для владения объектами, так как при развитии [интеллектуального указателя](#) (см. Дополнительные сведения в конце этого раздела). Все еще полезно и безопасно использовать необработанные указатели для отслеживания объектов, но если требуется использовать их для владения объектом, необходимо делать это с осторожностью и после тщательного анализа процедуры создания и уничтожения объектов, которые им принадлежат.

Первое, что необходимо знать, — это то, что при объявлении переменной необработанного указателя выделяется только память, необходимая для хранения адреса расположения памяти, на который будет ссылаться указатель при разыменовывании. Выделение памяти для самого значения данных (также называемое *резервным хранилищем*) еще не выделено. Другими словами, объявив переменную необработанного указателя, вы создаете переменную адреса памяти, а не фактическую переменную данных. Разыменовывание переменной указателя до проверки того, что она содержит действительный адрес в резервном хранилище, приведет к неопределенному поведению (обычно неустранимой ошибке) программы. В следующем примере демонстрируется подобная ошибка:

```
int* pNumber;           // Declare a pointer-to-int variable.  
*pNumber = 10;          // error. Although this may compile, it is  
                      // a serious error. We are dereferencing an  
                      // uninitialized pointer variable with no  
                      // allocated memory to point to.
```

Пример разыменовывает тип указателя без выделения памяти для хранения фактических целочисленных данных или без выделенного допустимого адреса памяти. В следующем коде исправлены эти ошибки:

```

int number = 10;           // Declare and initialize a local integer
                           // variable for data backing store.
int* pNumber = &number;    // Declare and initialize a local integer
                           // pointer variable to a valid memory
                           // address to that backing store.
...
*pNumber = 41;            // Dereference and store a new value in
                           // the memory pointed to by
                           // pNumber, the integer variable called
                           // "number". Note "number" was changed, not
                           // "pNumber".

```

В исправленном примере кода используется локальной память стека для создания резервного хранилища, на который указывает указатель `pNumber`. Базовый тип используется для простоты. На практике резервное хранилище для указателей — это наиболее часто определяемые пользователем типы, динамически выделяемые в области памяти, называемой *кучей* (или *свободным хранилищем*) с помощью `new` ключевого выражения (в программировании в стиле с `malloc()` использовалась старая функция библиотеки времени выполнения `c`). После выделения эти переменные обычно называются объектами, особенно если они основаны на определении класса. Память, выделенная с помощью `new` должна быть удалена соответствующим `delete` оператором (или, если вы использовали `malloc()` функцию для ее выделения, функция времени выполнения `C free()`).

Однако можно легко забыть удалить динамически выделенный объект, особенно в сложном коде, который вызывает ошибку ресурса, называемую *утечкой памяти*. По этой причине в современном C++ настоятельно не рекомендуется использовать необработанные указатели. Почти всегда лучше обернуть необработанный указатель в [Интеллектуальный указатель](#), который автоматически освобождает память при вызове его деструктора (когда код выходит за пределы области для смарт-указателя); с помощью смарт-указателей вы практически устраняете целый класс ошибок в программах на C++. В следующем примере предположим, что `MyClass` — это пользовательский тип, который имеет открытый метод

```
DoSomeWork();
```

```

void someFunction() {
    unique_ptr<MyClass> pMc(new MyClass);
    pMc->DoSomeWork();
}
// No memory leak. Out-of-scope automatically calls the destructor
// for the unique_ptr, freeing the resource.

```

Дополнительные сведения о смарт-указателях см. в разделе [интеллектуальные указатели](#).

Дополнительные сведения о преобразовании указателей см. в разделе [преобразования типов и типизация](#).

Дополнительные сведения об указателях в целом см. в разделе [указатели](#).

Типы данных Windows

в классическом программировании Win32 для C и C++ большинство функций используют определяемые Windows `typedef` и `#define` макросы (определенные в `windef.h`) для указания типов параметров и возвращаемых значений. эти Windows типы данных в основном представляют собой специальные имена (псевдонимы), предоставленные встроенным типам C/C++. полный список определений типов и определения препроцессора см. в разделе [Windows Data types](#). Некоторые из этих определений типов, такие как `HRESULT` и `LCID`, являются полезными и описательными. Другие, например `INT`, не имеют специального смысла и являются просто псевдонимами для основных типов C++. Прочие типы данных Windows имеют имена, которые сохранились с эпохи программирования на языке C для 16-разрядных

процессоров, и не имеют смысла или значения на современном оборудовании или в современных операционных системах. существуют также специальные типы данных, связанные с библиотекой среда выполнения Windows, которые перечислены как [среда выполнения Windows базовых типов данных](#). В современном языке C++ в целом рекомендуется использовать базовые типы C++, за исключением случаев, когда тип Windows сообщает некоторые дополнительные сведения о том, как следует интерпретировать значение.

Дополнительные сведения

Дополнительные сведения о системе типов C++ см. в следующих разделах.

[Типы значений](#)

Описывает [типы значений](#), а также проблемы, связанные с их использованием.

[Преобразования типов и безопасность типов](#)

Описание типовых проблем преобразования типов и способов их избежать.

См. также

[Возвращение к C++](#)

[Справочник по языку C++](#)

[Стандартная библиотека C++](#)

Область (C++)

12.11.2021 • 3 minutes to read

При объявлении программного элемента, такого как класс, функция или переменная, его имя может быть "видимым" и использоваться в определенных частях программы. Контекст, в котором отображается имя, называется его *областью действия*. Например, если объявить переменную `x` внутри функции, `x` она будет видна только в теле этой функции. Он имеет *локальную область*. У вас могут быть другие переменные с одним и тем же именем в программе; пока они находятся в разных областях, они не нарушают правило одного определения и не вызывают ошибку.

Для автоматических нестатических переменных область также определяет, когда они создаются и уничтожаются в памяти программ.

Существует шесть видов областей:

- **Глобальная область** Глобальное имя — это объявление, объявленное вне любого класса, функции или пространства имен. Однако в C++ даже эти имена существуют с неявным глобальным пространством имен. Область глобальных имен расширяется с точки объявления до конца файла, в котором они объявляются. Для глобальных имен видимость также регулируется правилами [компоновки](#), которые определяют, является ли имя видимым в других файлах программы.
- **Область пространства имен** Имя, объявленное в [пространстве имен](#) вне любого класса или определения перечисления или блока функции, видимо от его точки объявления до конца пространства имен. Пространство имен может быть определено в нескольких блоках для разных файлов.
- **Локальная область** Имя, объявленное внутри функции или лямбда-выражения, включая имена параметров, имеет локальную область. Они часто называются "локальными". Они видны только от их точки объявления до конца функции или тела лямбда-выражения. Локальная область — это разновидность области блока, которая обсуждается далее в этой статье.
- **Область класса** Имена членов класса имеют область класса, которая расширяется по всему определению класса независимо от точки объявления. Доступность членов класса дополнительно контролируется с помощью `public`, `private`, `protected` ключевых слов, и. Доступ к открытым или защищенным членам возможен только с помощью операторов выбора членов (`.` или `->`) или операторы указателя на член (`.*` или `->*`).
- **Область инструкции** Имена, объявленные `for` в `if`, `while` операторе, или, `switch` видимы до конца блока инструкции.
- **Область действия функции** [Метка](#) имеет область видимости функции, что означает, что она видна в теле функции, даже до ее точки объявления. Область функции позволяет писать инструкции `goto cleanup`, например перед `cleanup` объявлением метки.

Скрытие имен

Можно скрыть имя, объявив его в закрытом блоке. На следующем рисунке `i` повторно объявляется во внутреннем блоке, таким образом скрывая переменную, связанную с `i` во внешней области видимости блока.

```

Sample::Func(char *szWhat)
{
    int i = 0;
    cout << "i = " << i << "\n";
    {
        int i = 7, j = 9;
        cout << "i = " << i << "\n"
            << "j = " << j << "\n";
    }
    cout << "i = " << i << "\n";
}

```

Внешний блок содержит объект локальной области *i* и параметр формата *szWhat*.

Внутренний блок содержит объекты локальной области *i* и *j*.

Блокировка области и скрытия имен

Выходные данные программы, представленной на рисунке, выглядят следующим образом.

```

i = 0
i = 7
j = 9
i = 0

```

NOTE

Считается, что аргумент *szWhat*, находится в области видимости функции. Поэтому он обрабатывается так, как если бы он был объявлен в крайнем блоке функции.

Скрытие имен классов

Имена классов можно скрыть, объявив функцию, объект, переменную или перечислитель в той же области. Однако при префикссе ключевого слова доступ к имени класса по-прежнему возможен `class`.

```

// hiding_class_names.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

// Declare class Account at global scope.
class Account
{
public:
    Account( double InitialBalance )
    { balance = InitialBalance; }
    double GetBalance()
    { return balance; }
private:
    double balance;
};

double Account = 15.37;           // Hides class name Account

int main()
{
    class Account Checking( Account ); // Qualifies Account as
                                         // class name

    cout << "Opening account with a balance of: "
        << Checking.GetBalance() << "\n";
}
//Output: Opening account with a balance of: 15.37

```

NOTE

В любом месте `Account`, для которого вызывается имя класса (), класс ключевых слов необходимо использовать, чтобы отличать его от учетной записи с глобальной областью действия. Это правило не применяется, если имя класса находится слева от оператора разрешения области действия (::). Имена слева от оператора разрешения области действия всегда считаются именами класса.

В следующем примере показано, как объявить указатель на объект типа `Account` с помощью `class` ключевого слова:

```
class Account *Checking = new class Account( Account );
```

В `Account` инициализаторе (в круглых скобках) в предыдущем операторе есть глобальная область видимости; это тип `double`.

NOTE

Повторное использование имен идентификаторов считается плохим стилем программирования, как показано в следующем примере.

Сведения об объявлении и инициализации объектов класса см. в разделе [классы, структуры и объединения](#). Дополнительные сведения об использовании `new` операторов "и" `delete` Free-Store "см. в разделе [операторы new и DELETE](#).

Скрытие имен с глобальной областью видимости

Можно скрыть имена с глобальной областью, явно объявляя одно и то же имя в области видимости блока. Однако доступ к именам глобальных областей можно получить с помощью оператора разрешения области (::).

```
#include <iostream>

int i = 7;    // i has global scope, outside all blocks
using namespace std;

int main( int argc, char *argv[] ) {
    int i = 5;    // i has block scope, hides i at global scope
    cout << "Block-scoped i has the value: " << i << "\n";
    cout << "Global-scoped i has the value: " << ::i << "\n";
}
```

```
Block-scoped i has the value: 5
Global-scoped i has the value: 7
```

См. также раздел

[Основные понятия](#)

Файлы заголовков (C++)

12.11.2021 • 4 minutes to read

Имена программных элементов, таких как переменные, функции, классы и т. д., должны быть объявлены, прежде чем их можно будет использовать. Например, нельзя просто писать `x = 42` без предварительного объявления "x".

```
int x; // declaration
x = 42; // use x
```

Объявление сообщает компилятору, является ли элемент `int`, а `double`, **функцией** `class` или другим элементом. Кроме того, каждое имя должно быть объявлено (прямо или косвенно) в каждом cpp-файле, в котором он используется. При компиляции программы каждый CPP-файл компилируется независимо в единицу компиляции. Компилятор не имеет сведений о том, какие имена объявляются в других единицах компиляции. Это означает, что если вы определите класс или функцию или глобальную переменную, необходимо предоставить объявление этого объекта в каждом дополнительном cpp-файле, который его использует. Каждое объявление этого элемента должно быть точно одинаковым во всех файлах. Небольшая несогласованность вызовет ошибки или непреднамеренное поведение, когда компоновщик пытается объединить все единицы компиляции в одну программу.

Чтобы максимально сокращать возможности ошибок, в C++ принято соглашение об использовании *файлов заголовков* для хранения объявлений. Объявления можно сделать в файле заголовка, а затем использовать директиву `#include` в каждом cpp-файле или другом файле заголовка, для которого требуется это объявление. Директива `#include` вставляет копию файла заголовка непосредственно в CPP-файл перед компиляцией.

NOTE

В Visual Studio 2019 функция *модулей* C++ 20 появилась в качестве улучшения и в конечном итоге заменяет файлы заголовков. Дополнительные сведения см. в разделе Общие сведения о [модулях в C++](#).

Пример

В следующем примере показан общий способ объявления класса и его использования в другом исходном файле. Начнем с файла заголовка, `my_class.h`. Он содержит определение класса, но обратите внимание, что определение не завершено; Функция-член `do_something` не определена:

```
// my_class.h
namespace N
{
    class my_class
    {
        public:
            void do_something();
    };
}
```

Затем создайте файл реализации (обычно с расширением CPP или аналогичным модулем). Мы вызываем файл `my_class.cpp` и предоставим определение для объявления члена. Мы добавляем `#include`

директиву для файла "my_class.h", чтобы объявление my_class, вставленное в этот момент в cpp, было включено в `<iostream>` объявление для `std::cout`. Обратите внимание, что кавычки используются для файлов заголовков в том же каталоге, что и исходный файл, а угловые скобки используются для заголовков стандартной библиотеки. Кроме того, многие заголовки стандартной библиотеки не имеют h или любого другого расширения файла.

В файле реализации при необходимости можно использовать `using` оператор, чтобы не указывать каждое упоминание "my_class" или "cout" с "N::" или "std::". Не помещайте `using` операторы в файлы заголовков!

```
// my_class.cpp
#include "my_class.h" // header in local directory
#include <iostream> // header in standard library

using namespace N;
using namespace std;

void my_class::do_something()
{
    cout << "Doing something!" << endl;
}
```

Теперь можно использовать `my_class` в другом cpp файле. Мы `#include` заголовочный файл, чтобы компилятор запрашивал объявление. Все компиляторы должны иметь представление о том, что `my_class` — это класс, имеющий открытую функцию-член с именем `do_something()`.

```
// my_program.cpp
#include "my_class.h"

using namespace N;

int main()
{
    my_class mc;
    mc.do_something();
    return 0;
}
```

После того как компилятор завершит компиляцию каждого CPP-файла в OBJ-файлы, он передает OBJ-файлы компоновщику. Когда компоновщик объединяет объектные файлы, обнаруживается только одно определение для `my_class`; Он находится в OBJ-файле, созданном для `my_class.cpp`, и сборка выполняется.

ВКЛЮЧИТЬ УСЛОВИЯ

Как правило, файлы заголовков содержат директиву `include` или, `#pragma once` чтобы убедиться, что они не вставляются несколько раз в один CPP-файл.

```
// my_class.h
#ifndef MY_CLASS_H // include guard
#define MY_CLASS_H

namespace N
{
    class my_class
    {
        public:
            void do_something();
    };
}

#endif /* MY_CLASS_H */
```

Что следует разместить в файле заголовка

Поскольку файл заголовка потенциально может включаться в несколько файлов, он не может содержать определения, которые могут формировать несколько определений с одним и тем же именем. Следующие действия не разрешены или считаются очень неправильными.

- встроенные определения типов в пространстве имен или глобальной области
- невстроенные определения функций
- определения неконстантных переменных
- агрегатные определения
- безымянные пространства имен
- Директивы `using`

Использование `using` директивы не обязательно приведет к ошибке, но может вызвать проблему, так как она переводит пространство имен в область каждого CPP-файла, который напрямую или косвенно включает этот заголовок.

Пример файла заголовка

В следующем примере показаны различные виды объявлений и определений, допустимых в файле заголовка.

```

// sample.h
#pragma once
#include <vector> // #include directive
#include <string>

namespace N // namespace declaration
{
    inline namespace P
    {
        //...
    }

    enum class colors : short { red, blue, purple, azure };

    const double PI = 3.14; // const and constexpr definitions
    constexpr int MeaningOfLife{ 42 };
    constexpr int get_meaning()
    {
        static_assert(MeaningOfLife == 42, "unexpected!"); // static_assert
        return MeaningOfLife;
    }
    using vstr = std::vector<int>; // type alias
    extern double d; // extern variable

#define LOG // macro definition

#ifndef LOG // conditional compilation directive
    void print_to_log();
#endif

    class my_class // regular class definition,
    { // but no non-inline function definitions

        friend class other_class;
    public:
        void do_something(); // definition in my_class.cpp
        inline void put_value(int i) { vals.push_back(i); } // inline OK

    private:
        vstr vals;
        int i;
    };

    struct RGB
    {
        short r{ 0 }; // member initialization
        short g{ 0 };
        short b{ 0 };
    };

    template <typename T> // template definition
    class value_store
    {
    public:
        value_store<T>() = default;
        void write_value(T val)
        {
            //... function definition OK in template
        }
    private:
        std::vector<T> vals;
    };

    template <typename T> // template declaration
    class value_widget;
}

```

Единицы трансляции и компоновка

12.11.2021 • 2 minutes to read

В программе C++ *символ*, например имя переменной или функции, может быть объявлен как любое количество раз в пределах области, но он может быть определен только один раз. Это правило является "одним правилом определения" (С операционными данными). *Объявление* вводит (или повторно вводит) имя в программу. *Определение* вводит имя. Если имя представляет переменную, определение явно инициализирует его. *Определение функции* состоит из сигнатуры и тела функции. Определение класса состоит из имени класса, за которым следует блок, в котором перечислены все члены класса. (При необходимости можно отдельно определить тела функций-членов в другом файле.)

В следующем примере показаны некоторые объявления:

```
int i;
int f(int x);
class C;
```

В следующем примере показаны некоторые определения:

```
int i{42};
int f(int x){ return x * i; }
class C {
public:
    void DoSomething();
};
```

Программа состоит из одного или нескольких *единиц трансляции*. Блок преобразования состоит из файла реализации и всех заголовков, которые он включает прямо или косвенно. Файлы реализации обычно имеют расширение *cpp* или *CXX*. Файлы заголовков обычно имеют расширение *h* или *HPP*. Каждая единица преобразования компилируется независимо компилятором. После завершения компиляции компоновщик объединяет скомпилированные блоки преобразования в одну *программу*. Нарушения правила С операционными данными обычно отображаются как ошибки компоновщика. Ошибки компоновщика возникают, когда одно и то же имя имеет два разных определения в разных единицах преобразования.

Как правило, лучшим способом сделать переменную видимой в нескольких файлах является помещение ее в файл заголовка. Затем добавьте директиву `#include` в каждый файл *cpp*, для которого требуется объявление. Добавляя к содержимому заголовка, вы гарантируете, что объявляемые имена определяются только один раз.

В C++ 20 [модули](#) представлены в качестве улучшенной альтернативы файлам заголовков.

В некоторых случаях может потребоваться объявить глобальную переменную или класс в *cpp*-файле. В таких случаях необходим способ сообщить компилятору и компоновщику, какой тип *компоновки* имеет имя. Тип компоновки определяет, применяется ли имя объекта только к одному файлу или ко всем файлам. Понятие компоновки применяется только к глобальным именам. Понятие компоновки не применяется к именам, объявленным в области. Область определяется набором заключенных фигурных скобок, например в определениях функций или классов.

Внешняя и внутренняя компоновка

Бесплатная функция — это функция, определенная в глобальной области видимости или область пространства имен. Неконстантные глобальные переменные и свободные функции по умолчанию имеют *внешнюю компоновку*; они видимы из любой записи преобразования в программе. Поэтому ни один из других глобальных объектов не может иметь это имя. Символ с *внутренней компоновкой* или *без компоновки* отображается только в пределах блока преобразования, в котором он объявлен. Если имя имеет внутреннюю компоновку, то такое же имя может существовать в другой записи преобразования. Переменные, объявленные в определениях классов или теле функций, не имеют компоновки.

Можно принудительно включить внутреннюю компоновку в глобальное имя, явно объявив ее как `static`. Это ограничивает видимость той же блока преобразования, в котором он объявлен. В этом контексте `static` означает нечто отличное от применения к локальным переменным.

По умолчанию для следующих объектов используется внутренняя компоновка.

- константные объекты
- объекты `constexpr`
- определения типов
- статические объекты в области видимости пространства имен

Чтобы присвоить объекту `Const` объект `External` компоновки, объяйте его как `extern` и присвойте ему значение:

```
extern const int value = 42;
```

Дополнительные сведения см. в разделе [extern](#).

См. также раздел

[Основные понятия](#)

main

аргументы функции и командной строки

12.11.2021 • 7 minutes to read

Все программы на C++ должны иметь `main` функцию. При попытке компиляции программы C++ без `main` функции компилятор вызывает ошибку. (Библиотеки и библиотеки динамической компоновки `static` не имеют `main` функции.) `main` Функция заключается в том, где исходный код начинает выполнение, но до того, как программа введет `main` функцию, всем `static` членам класса без явных инициализаторов присваивается нулевое значение. В Microsoft C++ глобальные `static` объекты также инициализируются перед записью в `main`. К `main` функции, которая не применяется к другим функциям C++, применяются некоторые ограничения. Функция `main`:

- Не может быть перегружен (см. [перегрузку функции](#)).
- Не может быть объявлен как `inline`.
- Не может быть объявлен как `static`.
- Адрес не может быть создан.
- Невозможно вызвать из программы.

main Сигнатура функции

`main` Функция не имеет объявления, так как она встроена в язык. Если это так, синтаксис объявления для `main` будет выглядеть следующим образом:

```
int main();  
int main(int argc, char *argv[]);
```

Если возвращаемое значение не указано в `main`, компилятор предоставляет возвращаемое значение, равное нулю.

Стандартные аргументы командной строки

Аргументы для `main` обеспечения удобного анализа аргументов в командной строке. Типы для параметров `argc` и `argv` определяются языком. Имена `argc` и `argv` являются традиционными, но их можно называть по своему усмотрению.

Используются следующие определения аргументов.

`argc`

Целое число, содержащее число аргументов, следующих за `argv`. `argc` Параметр всегда больше или равен 1.

`argv`

Массив завершающихся `null` строк, представляющих введенные пользователем программы аргументы командной строки. По соглашению `argv[0]` — это команда, с помощью которой вызывается программа.

`argv[1]` Первый аргумент командной строки. Последним аргументом из командной строки является

`argv[argc - 1]`, и `argv[argc]` всегда имеет значение `null`.

Сведения о подавлении обработки в командной строке см. в разделе [Настройка обработки командной строки C++](#).

NOTE

По соглашению `argv[0]` — это имя файла программы. Однако на Windows можно порождать процесс с помощью `CreateProcess`. Если вы используете первый, и второй аргументы (`LpApplicationName` и `LpCommandLine`), `argv[0]` не может быть именем исполняемого файла. С помощью можно `GetModuleFileName` получить имя исполняемого файла и его полный путь.

Расширения, относящиеся к Microsoft

В следующих разделах описывается поведение, характерное для Майкрософт.

wmain ФУНКЦИЯ И _tmain Макрос

Если вы разрабатываете исходный код для использования Юникода Wide char актерс, можно использовать `wmain` точку входа, относящуюся к Microsoft, которая является char актер версией `main`. Ниже приведен эффективный синтаксис объявления для `wmain`:

```
int wmain();
int wmain(int argc, wchar_t *argv[]);
```

Кроме того, можно использовать Microsoft-Специальный `_tmain` макрос, определенный в `tchar.h`. `_tmain` разрешается в `main` Если `_UNICODE` не определен. В противном случае функция `_tmain` разрешается в функцию `wmain`. `_tmain` Макрос и другие макросы, начинающиеся с `_t` полезны для кода, который должен создавать отдельные версии для узких и широких char наборов актер. Дополнительные сведения см. в разделе [Использование универсальных текстовых сопоставлений](#).

Возврат void из main

Как расширение Майкрософт, `main` `wmain` функции и могут быть объявлены как возвращаемые `void` (без возвращаемого значения). Это расширение также доступно в некоторых других компиляторах, но его использование не рекомендуется. Он доступен для симметрии, если `main` не возвращает значение.

Если объявляется `main` или `wmain` возвращается `void`, то нельзя вернуть exit код в родительский процесс или операционную систему с помощью `return` инструкции. Чтобы вернуть exit код, если `main` или `wmain` объявлен как `void`, необходимо использовать `exit` функцию.

envp Аргумент командной строки

`main` `wmain` Сигнатуры или позволяют дополнительному расширению для доступа к переменным среды, относящимся к Microsoft. Это расширение также распространено в других компиляторах для Windows и UNIX систем. Имя `envp` является традиционным, но вы можете присвоить параметру среды любое имя. Ниже приведены эффективные объявления для списков аргументов, включающих параметр среды:

```
int main(int argc, char* argv[], char* envp[]);
int wmain(int argc, wchar_t* argv[], wchar_t* envp[]);
```

envp

Необязательный `envp` параметр представляет собой массив строк, представляющих переменные, заданные в среде пользователя. Этот массив завершается записью NULL. Он может быть объявлен как массив указателей на `char` (`char *envp[]`) или как указатель на указатели на `char` (`char **envp`). Если программа использует `wmain` вместо `main`, используйте `wchar_t` тип данных вместо `char`.

Блок среды, переданный в `main` И `wmain`, является замороженной копией текущей среды. Если впоследствии среда будет изменена путем вызова `putenv` ИЛИ `_wputenv`, то текущая среда (как возвращаемая `getenv` переменной или, `_wgetenv` а также `_environ` переменная или) изменится `_wenviron`, но блок, на который указывает, `envp` не изменится. Дополнительные сведения о подавлении обработки среды см. в разделе [Настройка обработки командной строки C++](#). `envp` Аргумент совместим с стандартом C89, но не с стандартами C++.

Примеры аргументов для `main`

В следующем примере показано, как использовать `argc` аргументы, `argv` И `envp` в следующих `main` случаях:

```
// argument_definitions.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>

using namespace std;
int main( int argc, char *argv[], char *envp[] )
{
    bool numberLines = false;      // Default is no line numbers.

    // If /n is passed to the .exe, display numbered listing
    // of environment variables.
    if ( (argc == 2) && _stricmp( argv[1], "/n" ) == 0 )
        numberLines = true;

    // Walk through list of strings until a NULL is encountered.
    for ( int i = 0; envp[i] != NULL; ++i )
    {
        if ( numberLines )
            cout << i << ": "; // Prefix with numbers if /n specified
        cout << envp[i] << "\n";
    }
}
```

Анализ аргументов командной строки C++

Правила синтаксического анализа командной строки, используемые кодом Microsoft C/C++, специфичны для Microsoft. Код запуска среды выполнения использует эти правила при интерпретации аргументов, заданных в командной строке операционной системы:

- Аргументы разделяются пробелами (пробел или табуляция).
- Первый аргумент (`argv[0]`) обрабатывается особым образом. Он представляет имя программы. Это должен быть допустимый путь, поэтому разрешены части, заключенные в двойные кавычки (`" "`). Эти знаки двойных кавычек не включаются в выходные данные `argv[0]`. Части, заключенные в двойные кавычки, не позволяют интерпретировать пробел или знак табуляции `char` актер в качестве конца аргумента. Последующие правила в этом списке не применяются.
- Стока, заключенная в двойные кавычки, интерпретируется как один аргумент, который может содержать пробелы в `char` актерс. Строку в кавычках можно встроить в аргумент. Курсор (`^`) не распознается как escape- `char` актер или разделитель. Внутри заключенной в кавычки строки пара двойных кавычек интерпретируется как одна экранированная двойная кавычка. Если командная строка заканчивается до тех пор, пока не будет найдена закрывающаяся двойная кавычка, то все `char` прочитанные актерс будут выводиться в качестве последнего аргумента.
- Символ двойной кавычки после обратной косой черты (`\"`) интерпретируется как литеральный символ двойной кавычки (`" "`).

- Символы обратной косой черты считаются литералами, если сразу за ними не стоит двойная кавычка.
- Если двойная кавычка стоит после четного числа символов обратной косой черты, в массив `argv` помещается по одному символу обратной косой черты (`\`) для каждой пары символов обратной косой черты (`\\"`), а сама двойная кавычка (`"`) интерпретируется как разделитель строк.
- Если двойная кавычка стоит после нечетного числа символов обратной косой черты, в массив `argv` помещается по одному символу обратной косой черты (`\`) для каждой пары символов обратной косой черты (`\\"`). Двойная кавычка интерпретируется как escape-последовательность путем main обратной косой черты, что приводит к тому, что литеральная двойная кавычка (`"`) будет помещена в `argv`.

Пример синтаксического анализа аргументов командной строки

В следующем примере программы показана передача аргументов командной строки:

```
// command_line_arguments.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
int main( int argc,      // Number of strings in array argv
          char *argv[],    // Array of command-line argument strings
          char *envp[] )   // Array of environment variable strings
{
    int count;

    // Display each command-line argument.
    cout << "\nCommand-line arguments:\n";
    for( count = 0; count < argc; count++ )
        cout << "  argv[" << count << "]  "
              << argv[count] << "\n";
}
```

Результаты синтаксического анализа командных строк

В следующей таблице показаны примеры входных данных и ожидаемые выходные данные, иллюстрирующие применение правил из приведенного выше списка.

ВХОДНЫЕ ДАННЫЕ КОМАНДНОЙ СТРОКИ	ARGV[1]	ARGV[2]	ARGV3-5
<code>"abc" d e</code>	<code>abc</code>	<code>d</code>	<code>e</code>
<code>a\\b d"e f"g h</code>	<code>a\\b</code>	<code>de fg</code>	<code>h</code>
<code>a\\\"b c d</code>	<code>a\"b</code>	<code>c</code>	<code>d</code>
<code>a\\\\\"b c" d e</code>	<code>a\\b c</code>	<code>d</code>	<code>e</code>
<code>a""c d</code>	<code>ab" c d</code>		

Развертывание подстановочных знаков

Кроме того, компилятор Майкрософт позволяет использовать подстановочный знак `char` актерс, вопросительный знак (`?`) и звездочку (`*`), чтобы указать аргументы filename и Path в командной строке.

Аргументы командной строки обрабатываются внутренней подпрограммой в коде запуска среды выполнения, который по умолчанию не расширяет подстановочные знаки в отдельные строки в массиве строк. Можно включить расширение подстановочных знаков, включив `setargv.obj` файл (`wsetargv.obj` файл для `wmain`) в `/link` параметрах компилятора или в `LINK` командной строке.

Дополнительные сведения о параметрах компоновщика для запуска среды выполнения см. в статье [Параметры ссылок](#).

Настройка обработки командной строки C++

Если программа не принимает аргументы командной строки, можно сохранить небольшой объем пространства, подавив подпрограмму обработки командной строки. Для этого включите файл `noarg.obj` (для `main` и `wmain`) в параметры компилятора `/link` или командную строку `LINK`.

Аналогичным образом, если вы никогда не использовали таблицу среды для доступа к аргументу `envr`, можно подавить внутреннюю подпрограмму обработки среды. Для этого включите файл `noenv.obj` (для `main` и `wmain`) в параметры компилятора `/link` или командную строку `LINK`.

Программа может вызывать семейство подпрограмм `spawn` или `exec` в библиотеке среды выполнения C. В этом случае не следует подавлять подпрограмму обработки среды, так как она используется для передачи данных о среде из родительского процесса в дочерний.

См. также раздел

[Основные понятия](#)

Завершение программы C++

12.11.2021 • 2 minutes to read

В C++ можно выйти из программы следующими способами:

- Вызовите `exit` функцию.
- Вызовите `abort` функцию.
- Выполните `return` инструкцию из `main`.

ФУНКЦИЯ `exit`

`exit` Функция, объявленная в `<stdlib.h>`, завершает программу на C++. Значение, передаваемое в качестве аргумента, `exit` возвращается операционной системе в качестве кода возврата программы или кода выхода. Принято, чтобы нулевым кодом возврата обозначалось, что программа завершена успешно. Константы `EXIT_FAILURE` и `EXIT_SUCCESS`, также определенные в, можно использовать `<stdlib.h>` для обозначения успеха или неудачи программы.

Выдача `return` инструкции из `main` функции эквивалентна вызову `exit` функции с возвращаемым значением в качестве аргумента.

ФУНКЦИЯ `abort`

`abort` Функция, также объявленная в стандартном включаемом файле `<stdlib.h>`, завершает программу на C++. Разница между `exit` и заключается в том `abort`, что `exit` позволяет выполнять обработку завершения при завершении выполнения C++ (вызываются деструкторы глобальных объектов), но `abort` немедленно завершает программу. `abort` Функция обходит обычный процесс уничтожения инициализированных глобальных статических объектов. Он также обходит любую специальную обработку, указанную с помощью `atexit` функции.

ФУНКЦИЯ `atexit`

Используйте `atexit` функцию, чтобы указать действия, которые выполняются до завершения программы. Глобальные статические объекты, инициализированные до уничтожения вызова `atexit`, не будут уничтожены до выполнения функции выхода.

`return` В `main`

Выдача `return` инструкции из `main` функционально эквивалентна вызову `exit` функции. Рассмотрим следующий пример.

```
// return_statement.cpp
#include <stdlib.h>
int main()
{
    exit( 3 );
    return 3;
}
```

`exit` Операторы и `return` в предыдущем примере функционально идентичны. Как правило, для C++

требуются функции, которые возвращают типы, отличные от `void` возвращаемых значений. `main` Функция является исключением; она может завершаться без `return` оператора. В этом случае он возвращает зависящее от реализации значение для вызывающего процесса. `return` Оператор позволяет указать возвращаемое значение из `main`.

Уничтожение статических объектов

При вызове `exit` или выполнении `return` инструкции из `main` статические объекты уничтожаются в обратном порядке их инициализации (после вызова метода, `atexit` если он существует). В следующем примере показано выполнение такого процесса инициализации и удаления.

Пример

В следующем примере статические объекты `sd1` и `sd2` создаются и инициализируются перед записью в `main`. После завершения выполнения этой программы с помощью `return` инструкции сначала `sd2` уничтожается, а затем `sd1`. Деструктор класса `ShowData` закрывает файлы, связанные с этими статическими объектами.

```
// using_exit_or_return1.cpp
#include <stdio.h>
class ShowData {
public:
    // Constructor opens a file.
    ShowData( const char *szDev ) {
        errno_t err;
        err = fopen_s(&OutputDev, szDev, "w" );
    }

    // Destructor closes the file.
    ~ShowData() { fclose( OutputDev ); }

    // Disp function shows a string on the output device.
    void Disp( char *szData ) {
        fputs( szData, OutputDev );
    }
private:
    FILE *OutputDev;
};

// Define a static object of type ShowData. The output device
// selected is "CON" -- the standard output device.
ShowData sd1 = "CON";

// Define another static object of type ShowData. The output
// is directed to a file called "HELLO.DAT"
ShowData sd2 = "hello.dat";

int main() {
    sd1.Disp( "hello to default device\n" );
    sd2.Disp( "hello to file hello.dat\n" );
}
```

Другой способ записать этот код — объявить объекты `ShowData` с областью видимости блока, в результате чего они будут удаляться при выходе из области.

```
int main() {
    ShowData sd1( "CON" ), sd2( "hello.dat" );

    sd1.Disp( "hello to default device\n" );
    sd2.Disp( "hello to file hello.dat\n" );
}
```

См. также

[main](#) аргументы функции и командной строки

Значения Lvalue и Rvalue (C++)

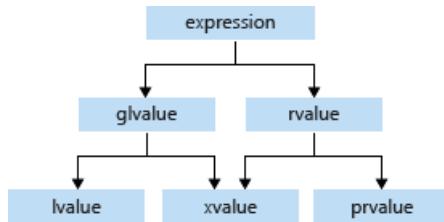
12.11.2021 • 2 minutes to read

Каждое выражение C++ имеет тип и принадлежит к *категории значений*. Категории значений являются основанием для правил, которым должны следовать компиляторы при создании, копировании и перемещении временных объектов во время оценки выражений.

Стандарт C++ 17 определяет категории значений выражений следующим образом:

- *Главалуе* — это выражение, вычисление которого определяет идентификатор объекта, битового поля или функции.
- *Prvalue* — это выражение, вычисление которого Инициализирует объект или битовое поле или выполняет вычисление значения операнда оператора, как указано в контексте, в котором он отображается.
- *XValue* — это главалуе, который обозначает объект или битовое поле, ресурсы которого можно использовать повторно (обычно потому, что он находится ближе к концу своего времени существования). Пример. определенные типы выражений, включающие ссылки rvalue (8.3.2), выдают xvalues, например, вызов функции, тип возвращаемого значения которого является ссылкой rvalue или приведен к ссылочному типу rvalue.
- *Lvalue* — это главалуе, который не является xValue.
- *Rvalue* — это prvalue или xValue.

На следующей схеме показаны связи между категориями.



Lvalue имеет адрес, к которому программа может получить доступ. Примерами выражений lvalue являются имена переменных, включая `const` переменные, элементы массива, вызовы функций, возвращающие ссылки lvalue, битовые поля, объединения и члены класса.

Выражение prvalue не имеет адреса, доступного вашей программой. Примерами выражений prvalue являются литералы, вызовы функций, возвращающие тип, не являющийся ссылочным, и временные объекты, которые создаются во время вычисления выражения, но доступны только компилятору.

Выражение xValue имеет адрес, который больше не доступен для вашей программы, но может использоваться для инициализации ссылки rvalue, которая предоставляет доступ к выражению. К примерам относятся вызовы функций, возвращающие ссылку rvalue, а также индексы массива, члена и указателя на элементы, где массив или объект является ссылкой rvalue.

Пример

В следующем примере показано несколько правильных и неправильных способов использования значений lvalue и rvalues.

```
// lvalues_and_rvalues2.cpp
int main()
{
    int i, j, *p;

    // Correct usage: the variable i is an lvalue and the literal 7 is a prvalue.
    i = 7;

    // Incorrect usage: The left operand must be an lvalue (C2106). `j * 4` is a prvalue.
    7 = i; // C2106
    j * 4 = 7; // C2106

    // Correct usage: the dereferenced pointer is an lvalue.
    *p = i;

    // Correct usage: the conditional operator returns an lvalue.
    ((i < 3) ? i : j) = 7;

    // Incorrect usage: the constant ci is a non-modifiable lvalue (C3892).
    const int ci = 7;
    ci = 9; // C3892
}
```

NOTE

В примерах этого раздела показано правильное и неправильное использование при неперегруженных операторах. Перегрузив операторы, можно преобразовать выражение, такое как `j * 4`, в значение lvalue.

Термины *lvalue* и *rvalue* часто используются при ссылке на ссылки на объекты. Дополнительные сведения о ссылках см. в разделе [декларатор ссылки lvalue: &](#) и [декларатор ссылки rvalue: &&](#).

См. также

[Основные понятия](#)

[Декларатор ссылки Lvalue: &](#)

[Декларатор ссылки rvalue: &&](#)

Временные объекты

12.11.2021 • 2 minutes to read

В некоторых случаях компилятору необходимо создавать временные объекты. Такие объекты могут создаваться по следующим причинам.

- Для инициализации `const` ссылки с инициализатором типа, отличного от базового типа для инициализированной ссылки.
- Для сохранения возвращаемого значения функции, которая возвращает пользовательский тип. Эти временные объекты создаются только в том случае, если программа не копирует возвращаемое значение в объект. Пример:

```
UDT Func1();    // Declare a function that returns a user-defined
                // type.

...

Func1();        // Call Func1, but discard return value.
                // A temporary object is created to store the return
                // value.
```

Поскольку возвращаемое значение не копируется в другой объект, создается временный объект. Более распространенный случай создания временных объектов — во время вычисления выражения, где требуется вызов перегруженных функций оператора. Эти перегруженные функции возвращают пользовательский тип, который часто не копируется в другой объект.

Рассмотрим выражение `ComplexResult = Complex1 + Complex2 + Complex3`. Сначала вычисляется выражение `Complex1 + Complex2`, и результат сохраняется во временном объекте. Далее вычисляется *временное выражение* `+ Complex3`, и результат копируется в `ComplexResult` (предполагая, что оператор присваивания не перегружен).

- Для сохранения результата приведения к пользовательскому типу. Когда объект заданного типа явно преобразуется в пользовательский тип, этот новый объект создается как временный.

Временные объекты имеют время жизни, определяемое точками их создания и удаления. Любое выражение, которое создает несколько временных объектов, в конечном счете удаляет их в порядке, обратном созданию. Точки, в которых происходит удаление, указаны в следующей таблице.

Точки удаления временных объектов

ПРИЧИНА СОЗДАНИЯ ВРЕМЕННОГО ОБЪЕКТА	ТОЧКА УДАЛЕНИЯ
Результат вычисления выражения	Все долгим сроком, созданные в результате вычисления выражения, уничтожаются в конце оператора выражения (то есть с точкой с запятой) или в конце управляемых выражений для <code>for</code> инструкций,, <code>if</code> <code>while</code> <code>do</code> И <code>switch</code> .

ПРИЧИНА СОЗДАНИЯ ВРЕМЕННОГО ОБЪЕКТА	ТОЧКА УДАЛЕНИЯ
Инициализация <code>const</code> ссылок	Если инициализатором не является L-значение того же типа, что и инициализируемая ссылка, создается временный объект базового типа объекта, инициализируемый выражением инициализации. Этот временный объект удаляется сразу после удаления объекта ссылки, с которым он связан.

Выравнивание

12.11.2021 • 3 minutes to read

Одной из низкоуровневых особенностей C++ является возможность указать точное выравнивание объектов в памяти, чтобы максимально использовать конкретную аппаратную архитектуру. По умолчанию компилятор выстраивает члены класса и структуры по значению размера: `bool` и `char` по 1 байтам, `short` по 2 байтам,, и к 4-байтным границам, и `int`, и `long` `float` `long long` `double` `long double` по 8-байтным границам.

В большинстве случаев не нужно беспокоиться о выравнивании, так как выравнивание по умолчанию уже оптимально. Однако в некоторых случаях можно добиться значительного улучшения производительности или экономии памяти, указав пользовательское выравнивание для структур данных. до Visual Studio 2015 можно использовать ключевые слова майкрософт `_alignof` И `_declspec(align)` задать выравнивание, превышающее значение по умолчанию. начиная с Visual Studio 2015 следует использовать стандартные ключевые слова `c++11` `alignof` И `alignas` максимальную переносимость кода. Новые ключевые слова ведут себя так же, как и расширения, специфичные для Майкрософт. Документация для этих расширений также применима к новым ключевым словам. Дополнительные сведения см. в разделе `alignof operator` and `aligned`. Стандарт C++ не задает поведение упаковки для выравнивания на границах меньше, чем значение по умолчанию компилятора для целевой платформы, поэтому в этом случае необходимо использовать корпорацию Майкрософт `#pragma pack`.

Используйте `класс aligned_storage` для выделения памяти структур данных с пользовательскими выравниваниями. `Класс aligned_union` предназначен для указания выравнивания для объединений с нетривиальными конструкторами или деструкторами.

Выравнивание и адреса памяти

Выравнивание представляет собой свойство адреса памяти, выражаемое как числовой адрес по модулю степени 2. Например, адрес 0x0001103F остаток от деления 4 равен 3. Этот адрес считается согласованным с $4n + 3$, где 4 обозначает выбранную степень 2. Выравнивание адреса зависит от выбранной степени 2. Тот же адрес по модулю 8 равен 7. Говорят, что адрес выровнен по X, если его выравнивание — $Xn+0$.

Процессоры выполняют инструкции, которые работают с данными, хранящимися в памяти. Данные идентифицируются по их адресам в памяти. У одной из них также есть размер. Мы вызываем объект «база» *естественным образом*, если его адрес выровнен по размеру. В противном случае он называется *неправильно*. Например, 8-байтная база с плавающей запятой имеет естественное выравнивание, если адрес, используемый для его распознавания, имеет 8-байтовое выравнивание.

Обработка выравнивания данных компилятором

Компиляторы пытаются сделать выделение данных способом, который предотвращает неправильное выравнивание данных.

Для простых типов данных компилятор назначает адреса, которые кратны размеру в байтах для типа данных. Например, компилятор назначает адреса переменным типа `long`, кратным 4, устанавливая младшие 2 бита адреса равными нулю.

Компилятор также дополняет структуры тем способом, который естественным образом соответствует каждому элементу структуры. Рассмотрим структуру `struct x_` в следующем примере кода:

```
struct x_
{
    char a;      // 1 byte
    int b;       // 4 bytes
    short c;    // 2 bytes
    char d;      // 1 byte
} bar[3];
```

Компилятор дополняет эту структуру для принудительного выравнивания естественным образом.

В следующем примере кода показано, как компилятор помещает заполненную структуру в память:

```
// Shows the actual memory layout
struct x_
{
    char a;          // 1 byte
    char _pad0[3];   // padding to put 'b' on 4-byte boundary
    int b;           // 4 bytes
    short c;         // 2 bytes
    char d;          // 1 byte
    char _pad1[1];   // padding to make sizeof(x_) multiple of 4
} bar[3];
```

Оба объявления возвращают `sizeof(struct x_)` 12 байт.

Второе объявление включает два дополнительных элемента:

1. `char _pad0[3]` для выровняйте `int b` элемент на 4-байтовой границе.
2. `char _pad1[1]` для выровняйте элементы массива структуры `struct _x bar[3];` по 4-байтовой границе.

Заполнение выровнено между элементами таким `bar[3]` образом, что обеспечивает естественный доступ.

В следующем примере кода показан `bar[3]` Макет массива:

adr	offset	element
0x0000		-----
0x0000	char a;	// bar[0]
0x0001	char pad0[3];	
0x0004	int b;	
0x0008	short c;	
0x000a	char d;	
0x000b	char _pad1[1];	
0x000c		-----
0x000c	char a;	// bar[1]
0x000d	char _pad0[3];	
0x0010	int b;	
0x0014	short c;	
0x0016	char d;	
0x0017	char _pad1[1];	
0x0018		-----
0x0018	char a;	// bar[2]
0x0019	char _pad0[3];	
0x001c	int b;	
0x0020	short c;	
0x0022	char d;	
0x0023	char _pad1[1];	

alignof И alignas

`alignas` Спецификатор типа является переносимым стандартным способом C++ для указания пользовательского выравнивания переменных и определяемых пользователем типов. Этот `alignof` оператор аналогичен стандартному переносимому способу получения выравнивания указанного типа или переменной.

Пример

Можно использовать `alignas` для класса, структуры или объединения или для отдельных элементов. При `alignas` обнаружении нескольких описателей компилятор выберет наиболее однозначное значение (то есть с наибольшим значением).

```
// alignas_alignof.cpp
// compile with: cl /EHsc alignas_alignof.cpp
#include <iostream>

struct alignas(16) Bar
{
    int i;           // 4 bytes
    int n;           // 4 bytes
    alignas(4) char arr[3];
    short s;         // 2 bytes
};

int main()
{
    std::cout << alignof(Bar) << std::endl; // output: 16
}
```

См. также

[Выравнивание структуры данных](#)

Тривиальные типы, типы стандартной структуры, POD и типы литералов

12.11.2021 • 5 minutes to read

Термин *структура* относится к организации членов объекта класса, структуры или типа объединения в памяти. В некоторых случаях структура четко определена спецификациями языка. Однако если класс или структура содержит определенные возможности языка C++, такие как виртуальные базовые классы, виртуальные функции, члены с разным уровнем управления доступом, компилятор может выбрать структуру самостоятельно. Эта структура может сильно отличаться в зависимости от того, какие оптимизации выполняются, и во многих случаях объект может даже не занимать непрерывную область памяти. Например, если класс имеет виртуальные функции, все экземпляры этого класса могут иметь одну общую таблицу виртуальных функций. Такие типы очень удобны, однако им присущи определенные ограничения. Поскольку структура не определена, их нельзя передавать программам, написанным на других языках, таких как C, а из-за того что они могут быть ненепрерывными, для них не поддерживается надежное копирование с помощью быстрых низкоуровневых функций, таких как `memcpy`, или сериализация по сети.

Чтобы дать возможность компиляторам, а также программам и метапрограммам C++ определять пригодность того или иного типа для операций, зависящих от конкретной структуры памяти, в C++14 представлены три категории простых классов и структур: *тривиальные*, *стандартной структуры* и *POD* (простые старые данные). Стандартная библиотека содержит шаблоны функций `is_trivial<T>`, `is_standard_layout<T>` и `is_pod<T>`, которые определяют, принадлежит ли данный тип данной категории.

Тривиальные типы

Если класс или структура в C++ включает предоставляемые компилятором или явно задаваемые по умолчанию специальные функции-члены — это тривиальный тип. Он занимает непрерывную область памяти. Он может иметь члены с разными спецификаторами доступа. В C++ компилятор может самостоятельно выбирать способ упорядочивания членов в этой ситуации. Таким образом вы можете копировать такие объекты с помощью `memcpy`, однако надежное использование их из программы на языке C невозможно. Тривиальный тип T можно скопировать в массив значений `char` или `unsigned char` и безопасно скопировать обратно в переменную T. Обратите внимание, что из-за требований к выравниванию, между членами типа могут существовать байты заполнения.

Тривиальные типы имеют тривиальный конструктор по умолчанию, тривиальный конструктор копирования, тривиальный оператор назначения копирования и тривиальный деструктор. В любом случае *тривиальный* означает, что конструктор, оператор или деструктор не предоставляется пользователем и принадлежит к классу, у которого

- нет виртуальных функций или виртуальных базовых классов;
- нет базовых классов с соответствующим нетривиальным конструктором, оператором или деструктором;
- нет членов данных типа класса с соответствующим нетривиальным конструктором, оператором или деструктором.

Ниже приведены примеры тривиальных типов. В `Trivial2` наличие конструктора `Trivial2(int a, int b)` требует указания конструктора по умолчанию. Чтобы тип мог считаться тривиальным, необходимо явно задать конструктор по умолчанию.

```

struct Trivial
{
    int i;
private:
    int j;
};

struct Trivial2
{
    int i;
    Trivial2(int a, int b) : i(a), j(b) {}
    Trivial2() = default;
private:
    int j; // Different access control
};

```

Типы стандартной структуры

Если класс или структура не содержит определенные возможности языка C++, такие как виртуальные функции, которых нет в языке С, и все элементы имеют один и тот же уровень управления доступом — это тип стандартной структуры. Его можно скопировать с помощью функции `memcpy`, а структура достаточно определена, чтобы его могли использовать программы на языке С. Типы стандартной структуры могут иметь определенные пользователем специальные функции-члены. Кроме того, типы стандартной структуры имеют следующие характеристики:

- нет виртуальных функций или виртуальных базовых классов;
- все нестатические члены данных имеют один уровень управления доступом;
- все нестатические члены типа класса относятся к стандартному макету;
- все базовые классы относятся к стандартной структуре;
- нет базовых классов того же типа, что и первый нестатический член данных;
- соответствуют одному из следующих условий:
 - нет нестатических членов данных в наиболее производном классе и не более одного базового класса с нестатическими членами данных или
 - нет базовых классов с нестатическими членами данных.

Ниже показан пример кода типа стандартной структуры:

```

struct SL
{
    // All members have same access:
    int i;
    int j;
    SL(int a, int b) : i(a), j(b) {} // User-defined constructor OK
};

```

Последние два требования, возможно, проще проиллюстрировать на примере кода. В следующем примере, хотя `Base` относится к типу стандартной структуры, `Derived` не является стандартным макетом, так как он (наиболее производный класс) и `Base` имеют нестатические члены данных:

```
struct Base
{
    int i;
    int j;
};

// std::is_standard_layout<<Derived> == false!
struct Derived : public Base
{
    int x;
    int y;
};
```

В этом примере `Derived` — тип стандартной структуры, поскольку у `Base` нет нестатических членов данных:

```
struct Base
{
    void Foo() {}
};

// std::is_standard_layout<<Derived> == true
struct Derived : public Base
{
    int x;
    int y;
};
```

Производный класс также будет относиться к стандартной структуре, если у `Base` есть члены данных, а у `Derived` — только функции-члены.

Типы POD

Если класс или структура является тривиальным типом и типом стандартной структуры — это тип POD (обычные старые данные). Таким образом, распределение памяти для типов POD является непрерывным, и адрес каждого члена выше, чем адрес члена, объявленного до него, что дает возможность выполнять побайтовое копирование и двоичный ввод-вывод для этих типов. Скалярные типы, такие как `int`, также являются типами POD. Типы POD, которые являются классами, могут содержать только типы POD в качестве нестатических членов данных.

Пример

В следующем примере показаны различия между тривиальными типами, типами стандартной структуры и POD:

```

#include <type_traits>
#include <iostream>

using namespace std;

struct B
{
protected:
    virtual void Foo() {}
};

// Neither trivial nor standard-layout
struct A : B
{
    int a;
    int b;
    void Foo() override {} // Virtual function
};

// Trivial but not standard-layout
struct C
{
    int a;
private:
    int b; // Different access control
};

// Standard-layout but not trivial
struct D
{
    int a;
    int b;
    D() {} //User-defined constructor
};

struct POD
{
    int a;
    int b;
};

int main()
{
    cout << boolalpha;
    cout << "A is trivial is " << is_trivial<A>() << endl; // false
    cout << "A is standard-layout is " << is_standard_layout<A>() << endl; // false

    cout << "C is trivial is " << is_trivial<C>() << endl; // true
    cout << "C is standard-layout is " << is_standard_layout<C>() << endl; // false

    cout << "D is trivial is " << is_trivial<D>() << endl; // false
    cout << "D is standard-layout is " << is_standard_layout<D>() << endl; // true

    cout << "POD is trivial is " << is_trivial<POD>() << endl; // true
    cout << "POD is standard-layout is " << is_standard_layout<POD>() << endl; // true

    return 0;
}

```

Типы литералов

Тип литерала — это такой тип, макет которого может быть определен во время компиляции. Ниже указаны типы литералов.

- void

- скалярные типы
- Ссылки
- Массивы void, скалярных типов или ссылок
- Класс, имеющий тривиальный деструктор, а также один или несколько конструкторов constexpr, которые не являются конструкторами перемещений или копий. Кроме того, все его нестатические данные-члены и базовые классы должны быть типами литералов и не должны изменяться.

См. также

[Основные понятия](#)

Классы C++ в качестве типов значений

12.11.2021 • 3 minutes to read

Классы C++ по умолчанию имеют типы значений. Они могут быть указаны в виде ссылочных типов, что позволяет использовать полиморфизмы для поддержки объектно-ориентированного программирования. Типы значений иногда просматриваются с точки зрения памяти и элемента управления макета, тогда как ссылочные типы — это базовые классы и виртуальные функции для polybase. По умолчанию типы значений являются копируемыми, что означает, что всегда существует конструктор копии и оператор присваивания копии. Для ссылочных типов можно сделать класс недоступным для копирования (отключить конструктор копирования и оператор присваивания копирования) и использовать виртуальный деструктор, который поддерживает предполагаемый полиморфизм. Типы значений также относятся к содержимому, которое при копировании всегда дает два независимых значения, которые могут быть изменены отдельно. Ссылочные типы относятся к идентификатору — какой тип объекта он представляет? По этой причине «ссылочные типы» также называются «полиморфизмами».

Если вам действительно нужен ссылочный тип (базовый класс, виртуальные функции), необходимо явно отключить копирование, как показано в `MyRefType` классе в следующем коде.

```
// cl /EHsc /nologo /W4

class MyRefType {
private:
    MyRefType & operator=(const MyRefType &);
    MyRefType(const MyRefType &);

public:
    MyRefType () {}
};

int main()
{
    MyRefType Data1, Data2;
    // ...
    Data1 = Data2;
}
```

Компиляция приведенного выше кода приведет к следующей ошибке:

```
test.cpp(15) : error C2248: 'MyRefType::operator =' : cannot access private member declared in class
'MyRefType'
        meow.cpp(5) : see declaration of 'MyRefType::operator ='
        meow.cpp(3) : see declaration of 'MyRefType'
```

Типы значений и эффективность перемещения

В связи с новыми оптимизациями копирования не рекомендуется выполнять копирование ресурсов. Например, при вставке строки в середину вектора строк нет дополнительных затрат на повторное выделение копий, а только перемещение, даже если оно приводит к росту самого вектора. Это также относится к другим операциям, например к экземпляру, выполняющему операцию добавления двух очень больших объектов. Как вы включаете эти оптимизации операций со значениями? В некоторых компиляторах C++ компилятор сделает это неявно, точно так же, как конструкторы копий могут быть автоматически созданы компилятором. Однако в C++ вашему классу потребуется "согласие", чтобы

переместить назначение и конструкторы, объявляя его в определении класса. Для этого используется ссылка на правостороннее амперсанд (`&&`) в соответствующих объявлении функций-членов, определяющая конструкторы перемещения и методы присваивания перемещения. Кроме того, необходимо вставить правильный код, чтобы "украсть отступив" из исходного объекта.

Как решить, требуется ли переход? Если вы уже уверены, что построение копии включено, вы, вероятно, хотите включить перемещение, если это возможно дешевле, чем глубокая копия. Однако если вам известно, что требуется поддержка перемещения, это не обязательно означает, что вы хотите включить копирование. Последний случай называется "типов только для перемещения". Примером, уже представленным в стандартной библиотеке, является `unique_ptr`. В качестве побочной заметки старая `auto_ptr` функция устарела и была заменена `unique_ptr` точно из-за отсутствия поддержки семантики перемещения в предыдущей версии C++.

С помощью семантики перемещения можно вернуть значение по значению или вставить в середину. Move — это оптимизация копирования. В качестве обходного пути требуется выделить кучу. Рассмотрим следующий псевдокод:

```
#include <set>
#include <vector>
#include <string>
using namespace std;

//...
set<widget> LoadHugeData() {
    set<widget> ret;
    // ... load data from disk and populate ret
    return ret;
}
//...
widgets = LoadHugeData(); // efficient, no deep copy

vector<string> v = IfIHadAMillionStrings();
v.insert( begin(v)+v.size()/2, "scott" ); // efficient, no deep copy-shuffle
v.insert( begin(v)+v.size()/2, "Andrei" ); // (just 1M ptr/len assignments)
//...
HugeMatrix operator+(const HugeMatrix&, const HugeMatrix& );
HugeMatrix operator+(const HugeMatrix&, HugeMatrix&& );
HugeMatrix operator+( HugeMatrix&&, const HugeMatrix& );
HugeMatrix operator+( HugeMatrix&&, HugeMatrix&& );
//...
hm5 = hm1+hm2+hm3+hm4+hm5; // efficient, no extra copies
```

Включение функции Move для соответствующих типов значений

Для класса, похожего на значение, в котором перемещение может быть дешевле, чем при глубоком копировании, включите построение перемещения и назначение перемещения для повышения эффективности. Рассмотрим следующий псевдокод:

```
#include <memory>
#include <stdexcept>
using namespace std;
// ...
class my_class {
    unique_ptr<BigHugeData> data;
public:
    my_class( my_class&& other ) // move construction
        : data( move( other.data ) ) { }
    my_class& operator=( my_class&& other ) // move assignment
    { data = move( other.data ); return *this; }
    // ...
    void method() { // check (if appropriate)
        if( !data )
            throw std::runtime_error("RUNTIME ERROR: Insufficient resources!");
    }
};
```

Если включить конструкцию копирования или назначение, также включите создание и назначение Move, если это может быть дешевле, чем глубокая копия.

Некоторые типы, *не являющиеся значениями*, являются только перемещением, например, если не удается клонировать ресурс, перенесите только владение. Например, `unique_ptr`.

См. также

[Система типов C++](#)

[Возвращение к C++](#)

[Справочник по языку C++](#)

[Стандартная библиотека C++](#)

Преобразования типов и безопасность типов

12.11.2021 • 8 minutes to read

В этом документе описаны распространенные проблемы преобразования типов и описывается, как избежать их использования в коде C++.

При написании программы на языке C++ важно убедиться, что она является строго типизированной. Это означает, что каждая переменная, аргумент функции и возвращаемое значение функции сохраняют допустимый тип данных и операции, затрагивающие значения различных типов, и не вызывают потерю данных, неверную интерпретацию битовых шаблонов или повреждение памяти. Неявное или неявное преобразование значений одного типа в другой является типобезопасным по определению. Однако иногда требуются преобразования типов, даже ненадежные преобразования. Например, может возникнуть необходимость сохранить результат операции с плавающей запятой в переменной типа `int` или передать значение в `unsigned int` функцию, которая принимает объект `signed int`. В обоих примерах показаны ненадежные преобразования, так как они могут привести к потере данных или повторной интерпретации значения.

Когда компилятор обнаруживает ненадежное преобразование, он выдает ошибку или предупреждение. Произошла ошибка при остановке компиляции. Предупреждение позволяет продолжить компиляцию, но указывает на возможную ошибку в коде. Однако даже если программа компилируется без предупреждений, она по-прежнему может содержать код, который вызывает неявные преобразования типов, приводящие к неправильным результатам. Ошибки типов также могут вводиться явными преобразованиями или приведениями в коде.

Неявные преобразования типов

Если выражение содержит операнды различных встроенных типов и явные приведения отсутствуют, компилятор использует встроенные *стандартные преобразования* для преобразования одного из операндов, чтобы типы совпадали. Компилятор пытается выполнить преобразования в четко определенной последовательности, пока она не завершится успешно. Если выбранное преобразование является повышением, компилятор не выдает предупреждение. Если преобразование является узким, компилятор выдает предупреждение о возможной утрате данных. Происходит ли фактическая потеря данных, зависит от фактических значений, но рекомендуется считать это предупреждение как ошибку. Если включен определяемый пользователем тип, компилятор пытается использовать преобразования, указанные в определении класса. Если не удается найти допустимое преобразование, компилятор выдает ошибку и не компилирует программу. Дополнительные сведения о правилах, регулирующих стандартные преобразования, см. в разделе [стандартные преобразования](#). Дополнительные сведения о пользовательских преобразованиях см. в разделе [пользовательские преобразования \(C++/CLI\)](#).

Расширяющие преобразования (продвижение)

В расширяющем преобразовании значение меньшей переменной присваивается более крупной переменной без потери данных. Поскольку расширяющие преобразования всегда являются надежными, компилятор выполняет их автоматически и не выдает предупреждения. Следующие преобразования являются расширяющими преобразованиями.

Исходный тип	Кому
Любой <code>signed</code> или <code>unsigned</code> целочисленный тип, кроме <code>long long</code> или <code>_int64</code>	<code>double</code>

Исходный тип	Кому
<code>bool</code> или <code>char</code>	Любой другой встроенный тип
<code>short</code> или <code>wchar_t</code>	<code>int</code> , <code>long</code> , <code>long long</code>
<code>int</code> , <code>long</code>	<code>long long</code>
<code>float</code>	<code>double</code>

Сужающие преобразования (приведение)

Компилятор выполняет сужающие преобразования неявным образом, но предупреждает о возможной потере данных. Выведите эти предупреждения очень серьезно. Если вы уверены, что не произойдет потери данных, так как значения в переменной большего размера всегда помещаются в меньшую переменную, добавьте явное приведение, чтобы компилятор больше не выдавал предупреждение. Если вы не уверены, что преобразование является надежным, добавьте в код какую-либо проверку среди выполнения для обработки возможной потери данных, чтобы она не вызывала неправильные результаты.

Преобразование из типа с плавающей запятой в целочисленный тип является узким преобразованием, так как дробная часть значения с плавающей запятой отбрасывается и теряется.

В следующем примере кода показаны некоторые неявные сужающие преобразования и предупреждения, которые возникают компилятором.

```
int i = INT_MAX + 1; //warning C4307:'+' : integral constant overflow
wchar_t wch = 'A'; //OK
char c = wch; // warning C4244:'initializing':conversion from 'wchar_t'
               // to 'char', possible loss of data
unsigned char c2 = 0xffff; //warning C4305:'initializing': truncation from
                         // 'int' to 'unsigned char'
int j = 1.9f; // warning C4244:'initializing':conversion from 'float' to
              // 'int', possible loss of data
int k = 7.7; // warning C4244:'initializing':conversion from 'double' to
             // 'int', possible loss of data
```

Преобразования со знаком — без знака

Целочисленный тип со знаком и его неподписанный аналог всегда имеют одинаковый размер, но они отличаются тем, как битовый шаблон интерпретируется для преобразования значения. В следующем примере кода показано, что происходит, когда один и тот же битовый шаблон интерпретируется как значение со знаком и как значение без знака. Битовый шаблон, хранящийся как `num`, и `num2` никогда не изменяется из того, что показано на предыдущем рисунке.

```
using namespace std;
unsigned short num = numeric_limits<unsigned short>::max(); // #include <limits>
short num2 = num;
cout << "unsigned val = " << num << " signed val = " << num2 << endl;
// Prints: "unsigned val = 65535 signed val = -1"

// Go the other way.
num2 = -1;
num = num2;
cout << "unsigned val = " << num << " signed val = " << num2 << endl;
// Prints: "unsigned val = 65535 signed val = -1"
```

Обратите внимание, что значения переинтерпретируются в обоих направлениях. Если программа

создает нечетные результаты, при которых знак значения кажется инвертированным от того, что вы ожидаете, найдите неявные преобразования между целыми типами со знаком и без знака. В следующем примере результат выражения (0-1) неявно преобразуется в `int`, `unsigned int` когда он сохраняется в `num`. Это приводит к переинтерпретации битового шаблона.

```
unsigned int u3 = 0 - 1;
cout << u3 << endl; // prints 4294967295
```

Компилятор не предупреждает о неявных преобразованиях между целыми типами со знаком и без знака. Поэтому рекомендуется полностью избегать беззнаковых преобразований. Если вы не можете избежать их, добавьте проверку среди выполнения, чтобы определить, является ли преобразуемое значение большим или равным нулю и меньше или равно максимальному значению типа со знаком. Значения в этом диапазоне будут передаваться из входных файлов в неподписанный или из неподписанных в подписывание без переинтерпретации.

Преобразования указателей

Во многих выражениях массив в стиле С неявно преобразуется в указатель на первый элемент в массиве, а преобразования констант могут выполняться автоматически. Хотя это и удобно, это, вероятно, подвержено ошибкам. Например, следующий плохо спроектированный пример кода кажется бессмысленное, но он компилирует и выдает результат "р". Во-первых, строковый константный литерал "Help" преобразуется в объект `char*`, указывающий на первый элемент массива; этот указатель затем увеличивается на три элемента, чтобы теперь указывал на последний элемент "р".

```
char* s = "Help" + 3;
```

Явные преобразования (приведения)

С помощью операции приведения можно указать компилятору преобразовать значение одного типа в другой тип. В некоторых случаях компилятор вызовет ошибку, если эти два типа полностью не связаны, но в других случаях не вызывает ошибку, даже если операция не является строго типизированной. Используйте приведение с осторожностью, так как любое преобразование из одного типа в другой является потенциальным источником ошибок программы. Однако иногда требуется выполнить приведения, а не все приведения являются опасными. Одно эффективное использование приведения заключается в том, что в коде выполняются понижающие преобразования и известно, что преобразование не приводит к созданию неверных результатов в программе. Фактически, это говорит компилятору о том, что вы делаете, а также о том, что вы выполняете предупреждения. Другой способ заключается в приведении из класса указателя на класс, производный от указатель на базовый. Другой способ — приведение к переменной постоянной, чтобы передать ее в функцию, для которой требуется аргумент, не являющийся константой. Большинство этих операций приведения к некоторым рискам требует определенного риска.

В программировании в стиле С для всех типов приведений используется один и тот же оператор приведения в стиле C.

```
(int) x; // old-style cast, old-style syntax
int(x); // old-style cast, functional syntax
```

Оператор приведения в стиле C идентичен оператору `call ()` и, следовательно, инконспикуаус в коде и легко пропускаться. Оба являются некорректными, так как они трудно распознать на взгляде или найти, и они достаточно разнородны для вызова любой комбинации `static`, `const` и `reinterpret_cast`. Понять, что такое приведение старого стиля, действительно может быть трудно и подвержено ошибкам. По всем

этим причинам, если требуется приведение, рекомендуется использовать один из следующих операторов приведения в C++, который в некоторых случаях значительно более строго типизирован, и что явно упрощает намерение программирования:

- `static_cast`, для приведений, которые проверяются только во время компиляции. `static_cast` Возвращает ошибку, если компилятор обнаруживает, что вы пытаетесь выполнить приведение типов, которые полностью несовместимы. Его также можно использовать для приведения между указателями на базовые и производные указатели, но компилятор не всегда может определить, будут ли такие преобразования небезопасны во время выполнения.

```
double d = 1.58947;
int i = d; // warning C4244 possible loss of data
int j = static_cast<int>(d); // No warning.
string s = static_cast<string>(d); // Error C2440:cannot convert from
// double to std::string

// No error but not necessarily safe.
Base* b = new Base();
Derived* d2 = static_cast<Derived*>(b);
```

Дополнительные сведения см. на веб-сайте [static_cast](#).

- `dynamic_cast`, для безопасного выполнения приведения указателя на `Base` к типу, который проверяется средой. Объект `dynamic_cast` является более безопасным `static_cast`, чем для типов, но проверка среды выполнения требует некоторых дополнительных издержек.

```
Base* b = new Base();

// Run-time check to determine whether b is actually a Derived*
Derived* d3 = dynamic_cast<Derived*>(b);

// If b was originally a Derived*, then d3 is a valid pointer.
if(d3)
{
    // Safe to call Derived method.
    cout << d3->DoSomethingMore() << endl;
}
else
{
    // Run-time check failed.
    cout << "d3 is null" << endl;
}

//Output: d3 is null;
```

Дополнительные сведения см. на веб-сайте [dynamic_cast](#).

- `const_cast`, для приведения параметра `const`-rvalue характеристики переменной или `const` для преобразования непеременной в значение `const`. Приведение `const` незавершенного использования — rvalue характеристики с помощью этого оператора, как и при использовании приведения в стиле C, за исключением того, что у `const_cast` вас меньше вероятность того, что приведение выполняется случайно. Иногда необходимо выполнить приведение `const` rvalue характеристики переменной, например, чтобы передать `const` переменную в функцию, которая принимает не `const` параметр. В приведенном ниже примере показано, как это сделать.

```
void Func(double& d) { ... }
void ConstCast()
{
    const double pi = 3.14;
    Func(const_cast<double&>(pi)); //No error.
}
```

Дополнительные сведения см. на веб-сайте [const_cast](#).

- `reinterpret_cast`, для приведения между несвязанными типами, такими как тип указателя и `int`.

NOTE

Этот оператор приведения не используется так часто, как другие, и не гарантирует перенос в другие компиляторы.

В следующем примере показано `reinterpret_cast` отличие от `static_cast`.

```
const char* str = "hello";
int i = static_cast<int>(str); //error C2440: 'static_cast' : cannot
                                // convert from 'const char *' to 'int'
int j = (int)str; // C-style cast. Did the programmer really intend
                  // to do this?
int k = reinterpret_cast<int>(str); // Programming intent is clear.
                                    // However, it is not 64-bit safe.
```

Дополнительные сведения см. в разделе [reinterpret_cast](#) оператор.

См. также

[Система типов C++](#)

[Возвращение к C++](#)

[Справочник по языку C++](#)

[Стандартная библиотека C++](#)

Стандартные преобразования

12.11.2021 • 11 minutes to read

В языке C++ определены преобразования между его основными типами. Также определяются преобразования для указателей, ссылочных типов и типов указателей на члены. Эти преобразования называются *стандартными преобразованиями*.

В этом разделе рассматриваются следующие стандартные преобразования:

- Восходящие приведения целочисленных типов
- Преобразования целочисленных типов
- Преобразования типов с плавающей запятой
- Преобразования типов с плавающей запятой и целочисленных типов
- Арифметические преобразования
- Преобразования указателей
- Преобразования ссылок
- Преобразования указателей на члены

NOTE

Пользовательские типы могут определять собственные преобразования. Преобразование определяемых пользователем типов рассматривается в [конструкторах](#) и [преобразованиях](#).

Следующий код вызывает преобразования (в данном примере это восходящее приведение целочисленных типов).

```
long long_num1, long_num2;
int int_num;

// int_num promoted to type long prior to assignment.
long_num1 = int_num;

// int_num promoted to type long prior to multiplication.
long_num2 = int_num * long_num2;
```

Результат преобразования является L-значением только в том случае, если получается ссылочный тип. Например, определяемое пользователем преобразование, объявленное как, `operator int&()` возвращает ссылку и является l-значением. Однако преобразование, объявленное как, `operator int()` возвращает объект и не является l-значением.

Восходящие приведения целочисленных типов

Объекты целочисленного типа можно преобразовать в другой более широкий целочисленный тип, то есть тип, который может представлять больший набор значений. Этот расширяющий тип преобразования называется *целочисленным повышением*. С помощью целочисленного повышения можно использовать следующие типы в выражении, где можно использовать другой целочисленный тип:

- Объекты, литералы и константы типа `char` И `short int`
- Типы перечисления.
- `int` битовые поля
- Enumerators

Специальные акции C++ — это «сохранение значения», так как значение после продвижения гарантированно совпадает со значением до повышения уровня. В случае с сохранением данных об акциях объекты более коротких целочисленных типов (например, битовые поля или объекты типа `char`) переносятся в тип, `int`. Если `int` могут представлять полный диапазон исходного типа. Если `int` не может представлять полный диапазон значений, объект повышается до типа `unsigned int`. Хотя эта стратегия аналогична той, которая используется в стандартном языке C, преобразования с сохранением значений не сохраняют "подпись" объекта.

Обычно при повышениях с сохранением значения и повышениях с сохранением наличия знака выдаются одинаковые результаты. Однако они могут дать разные результаты, если объект повышенного уровня выглядит следующим образом:

- Операнд `/`, `%`, `/=`, `%=`, `<`, `<=`, `>` ИЛИ `>=`

Эти операторы зависят от знака для определения результата. При применении к этим операндам предложения с сохранением и обслуживанием, сохраняя при этом подписи, получают разные результаты.

- Левый операнд `>>` или `>>=`

Эти операторы обрабатывают количества со знаком и без знака по-разному в операции сдвига. Для количества со знаком операция сдвига вправо распространяет бит знака на освобожденные позиции битов, а освобожденные разряды заполняются нулем в неподписанных количествах.

- Аргумент для перегруженной функции или operand перегруженного оператора, который зависит от подписи типа operand для соответствующего аргумента. Дополнительные сведения об определении перегруженных операторов см. в разделе [перегруженные операторы](#).

Преобразования целочисленных типов

Целочисленные преобразования — это преобразования между целочисленными типами. Целочисленные типы: `char`, `short` (или `short int`), `int`, `long` И `long long`. Эти типы могут уточняться с помощью `signed` ИЛИ `unsigned`, а также `unsigned` могут использоваться в качестве краткости для `unsigned int`.

Преобразование чисел со знаком в числа без знака

Объекты целочисленных типов со знаком можно преобразовывать в соответствующие типы без знака. При возникновении этих преобразований фактический битовый шаблон не изменяется. Однако интерпретация изменений данных. Рассмотрим этот код:

```
#include <iostream>

using namespace std;
int main()
{
    short i = -3;
    unsigned short u;

    cout << (u = i) << "\n";
}
// Output: 65533
```

В предыдущем примере `signed short i` определено и инициализировано отрицательное число. Выражение `(u = i)` приводит к `i` преобразованию в `unsigned short` перед присваиванием `u`.

Преобразование чисел без знака в числа со знаком

Объекты целочисленных типов без знака можно преобразовывать в соответствующие типы со знаком. Однако если значение без знака находится за пределами представимого диапазона типа со знаком, результат не будет иметь правильное значение, как показано в следующем примере:

```
#include <iostream>

using namespace std;
int main()
{
short i;
unsigned short u = 65533;

cout << (i = u) << "\n";
}
//Output: -3
```

В предыдущем примере `u` — это `unsigned short` целочисленный объект, который необходимо преобразовать в число со знаком для вычисления выражения `(i = u)`. Поскольку его значение не может быть правильно представлено в `signed short`, данные будут неправильно интерпретированы, как показано ниже.

Преобразование чисел с плавающей запятой

Объект типа с плавающей запятой можно безопасно преобразовать в более точный тип с плавающей запятой, то есть без потери значимости. Например, преобразования из `float` в `double` `double` в `long double` являются безнадежными, а значение не изменяется.

Объект плавающего типа также можно преобразовать в менее точный тип, если он находится в диапазоне, представленном этим типом. (См. раздел [плавающие ограничения](#) для диапазонов плавающих типов.) Если исходное значение не может быть представлено точно, его можно преобразовать в следующее большее или следующее более низкое значение. Если такого значения не существует, результат будет неопределенным. Рассмотрим следующий пример.

```
cout << (float)1E300 << endl;
```

Максимальное значение, которое может быть представлено типом, `float` — это 3.402823466 E38 — намного меньшее число, чем 1E300. Таким образом, число преобразуется в бесконечное значение, а результатом является "INF".

Преобразования между целочисленным типом и типом с плавающей запятой

Определенные выражения могут вызывать преобразование объектов плавающего типа в целочисленные типы, и наоборот. Если объект целочисленного типа преобразуется в тип с плавающей запятой, исходное значение не может быть представлено точно, результатом является либо следующее выше, либо следующее меньшее представимое значение.

При преобразовании объекта с плавающего типа в целочисленный тип дробная часть *усекается* или округляется в сторону нуля. Число, например 1,3, преобразуется в 1, а -1,3 преобразуется в -1. Если усеченное значение выше наибольшего допустимого значения или меньше наименьшего представимого

значения, результат будет неопределенным.

Арифметические преобразования

Многие бинарные операторы (обсуждаемые в [выражениях с бинарными операторами](#)) приводят к преобразованию operandов и выдают результаты одинаковым образом. Преобразования эти операторы вызываются *обычными арифметическими преобразованиями*. Арифметические преобразования operandов, которые имеют различные собственные типы, выполняются, как показано в следующей таблице. Типы `typedef` ведут себя в соответствии со своими базовыми собственными типами.

Условия для преобразования типов

ВЫПОЛНЕННЫЕ УСЛОВИЯ	ПРЕОБРАЗОВАНИЕ
Любой operand имеет тип <code>long double</code> .	Другой operand преобразуется в тип <code>long double</code> .
Предыдущее условие не выполнено, и любой из operandов имеет тип <code>double</code> .	Другой operand преобразуется в тип <code>double</code> .
Предыдущие условия не выполнены, и любой из operandов имеет тип <code>float</code> .	Другой operand преобразуется в тип <code>float</code> .
Предыдущие условия не выполнены (ни один из operandов не является operandом с плавающей запятой).	<p>Операнды получают целочисленные продвижения следующим образом:</p> <ul style="list-style-type: none">— Если любой из operandов имеет тип <code>unsigned long</code>, то другой operand преобразуется в тип <code>unsigned long</code>.— Если предыдущее условие не выполнено, и если любой из operandов имеет тип <code>long</code>, а другой тип <code>unsigned int</code>, оба operandы преобразуются в тип <code>unsigned long</code>.— Если предыдущие два условия не выполняются, и если любой из operandов имеет тип <code>long</code>, то другой operand преобразуется в тип <code>long</code>.— Если предыдущие три условия не выполняются, и если любой из operandов имеет тип <code>unsigned int</code>, то другой operand преобразуется в тип <code>unsigned int</code>.— Если ни одно из вышеперечисленных условий не выполняется, оба operandы преобразуются в тип <code>int</code>.

В следующем коде демонстрируются правила преобразования, описанные в таблице.

```
double dVal;
float fVal;
int iVal;
unsigned long ulVal;

int main() {
    // iVal converted to unsigned long
    // result of multiplication converted to double
    dVal = iVal * ulVal;

    // ulVal converted to float
    // result of addition converted to double
    dVal = ulVal + fVal;
}
```

Первый оператор в приведенном выше примере представляет умножение двух целочисленных типов,

`iVal` и `u1Val`. Условие выполнено, так как операнд не имеет плавающего типа, а один операнд имеет тип `unsigned int`. Таким образом, другой операнд, `iVal` преобразуется в тип `unsigned int`. Затем результат присваивается `dVal`. Условие здесь соответствует тому, что один операнд имеет тип `double`, поэтому `unsigned int` результат умножения преобразуется в тип `double`.

Второй оператор в предыдущем примере показывает добавление `float` и целочисленный тип: `fVal` и `u1Val`. `u1Val` Переменная преобразуется в тип `float` (третье условие в таблице). Результат сложения преобразуется в тип `double` (второе условие в таблице) и присваивается `dVal`.

Преобразования указателей

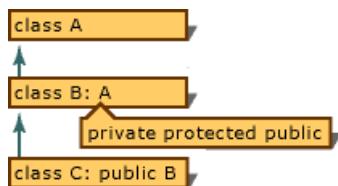
Указатели можно преобразовывать в ходе присваивания, инициализации, сравнения и выполнения других выражений.

Указатель на классы

Указатель на класс можно преобразовать в указатель на базовый класс в двух случаях.

Во-первых, когда указанный базовый класс доступен и преобразование однозначно. Дополнительные сведения об неоднозначных ссылках на базовые классы см. в разделе [несколько базовых классов](#).

Доступность базового класса зависит от используемого типа наследования. Рассмотрим пример наследования на следующем рисунке.



В следующей таблице показана доступность базового класса для ситуации, представленной на рисунке.

ТИП ФУНКЦИИ	НАСЛЕДОВАНИЕ	ДОПУСТИМО ЛИ ПРЕОБРАЗОВАНИЕ ИЗ Б * К * ЮРИДИЧЕСКИМ?
Внешняя функция (без области видимости класса)	Личные	Нет
	Защищенный	Нет
	Общедоступные	Да
Функция-член B (в области B)	Личные	Да
	Защищенный	Да
	Общедоступные	Да
Функция-член C (в области C)	Личные	Нет
	Защищенный	Да
	Общедоступные	Да

Во-вторых, указатель на класс можно преобразовать в указатель на базовый класс при использовании явного преобразования типов. Дополнительные сведения о явных преобразованиях типов см. в разделе [оператор явного преобразования типа](#).

Результатом такого преобразования является указатель на *подобъект*, часть объекта, полностью описываемого базовым классом.

В следующем примере кода определяются два класса: `A` и `B`, где `B` является производным от класса `A`. (Дополнительные сведения о наследовании см. в разделе [производные классы](#).) Затем он определяет `bObject`, объект типа `B` и два указателя (`pA` и `pB`), которые указывают на объект.

```
// C2039 expected
class A
{
public:
    int AComponent;
    int AMemberFunc();
};

class B : public A
{
public:
    int BComponent;
    int BMemberFunc();
};

int main()
{
    B bObject;
    A *pA = &bObject;
    B *pB = &bObject;

    pA->AMemberFunc(); // OK in class A
    pB->AMemberFunc(); // OK: inherited from class A
    pA->BMemberFunc(); // Error: not in class A
}
```

Указатель `pA` принадлежит типу `A *`, и это можно интерпретировать следующим образом: "указатель на объект типа `A`". Члены `bObject` (такие как `BComponent` и `BMemberFunc`) являются уникальными для типа `B` и поэтому недоступны через `pA`. Указатель `pA` предоставляет доступ только к тем характеристикам (функциям-членам и данным) объекта, которые определены в классе `A`.

Указатель на функцию

Указатель на функцию можно преобразовать в тип `void *`, если тип `void *` достаточно большой для хранения указателя.

Указатель на void

Указатели на тип `void` можно преобразовать в указатели на любой другой тип, но только с явно приведенным типом (в отличие от С). Указатель на любой тип можно преобразовать неявно в указатель на тип `void`. Указатель на неполный объект типа можно преобразовать в указатель на `void` (неявно) и обратно (явно). Результат такого преобразования равен значению исходного указателя. Объект считается неполным, если он объявлен, но недостаточно информации для определения его размера или базового класса.

Указатель на любой объект, который не `const` или `volatile` может быть неявно преобразован в указатель типа `void *`.

Указатели с ключевыми словами `const` и `volatile`

C++ не передает стандартное преобразование из `const` `volatile` типа или в тип, который не является

`const` или `volatile`. Однако можно указать любое преобразование с помощью явного приведения типов (включая небезопасные преобразования).

NOTE

Указатели C++ на члены, за исключением указателей на статические члены, отличаются от обычных указателей и не имеют одинаковых стандартных преобразований. Указатели на статические члены являются обычными, и для них имеются такие же преобразования, как и для обычных указателей.

Преобразование пустых (null) указателей

Целочисленное константное выражение, результатом которого является ноль, или такое выражение, приведенное к типу указателя, преобразуется в указатель, называемый *пустым указателем*. Этот указатель всегда сравнивает неравный с указателем на любой допустимый объект или функцию. Исключение — это указатели на основанные объекты, которые могут иметь одинаковое смещение и по-прежнему указывать на разные объекты.

В C++ 11 тип `nullptr` должен быть предпочтительным для пустого указателя в стиле C.

Преобразование выражений указателей

Любое выражение с типом массива можно преобразовать в указатель того же типа. Результатом преобразования будет указатель на первый элемент массива. В следующем примере показано такое преобразование.

```
char szPath[_MAX_PATH]; // Array of type char.  
char *pszPath = szPath; // Equals &szPath[0].
```

Выражение, которое приводит к функции, возвращающей определенный тип, преобразуется в указатель на функцию, возвращающую этот тип, за исключением случая, когда:

- Выражение используется в качестве операнда оператора взятия адреса (`&`).
- выражение используется в качестве операнда для оператора вызова функции.

Преобразования ссылок

В таких случаях ссылка на класс может быть преобразована в ссылку на базовый класс:

- Указанный базовый класс доступен.
- Преобразование является однозначным. (Дополнительные сведения об неоднозначных ссылках на базовый класс см. в разделе [несколько базовых классов](#).)

Результат преобразования — это указатель на вложенный объект, представляющий базовый класс.

Указатель на член

Указатели на члены класса можно преобразовать в ходе присвоения, инициализации, сравнения и выполнения других выражений. В этом разделе описаны следующие преобразования указателей в члены.

Указатель на член базового класса

Указатель на член базового класса можно преобразовать в указатель на член производного от него класса при выполнении следующих условий:

- доступно обратное преобразование из указателя на производный класс в указатель базового класса;

- производный класс не наследуется виртуально от базового класса.

Если левый операнд является указателем на член, правый операнд должен иметь тип указателя на член или являться константным выражением со значением 0. Такое присваивание допустимо только в следующих случаях:

- правый operand является указателем на член того же класса, что и левый operand;
- левый operand является указателем на член класса, открытого и однозначно производного от класса правого operand'a.

преобразование указателя NULL на члены

Целочисленное константное выражение, результатом вычисления которого является ноль, преобразуется в указатель null. Этот указатель всегда сравнивается неравным с указателем на любой допустимый объект или функцию. Исключение — это указатели на основанные объекты, которые могут иметь одинаковое смещение и по-прежнему указывать на разные объекты.

В следующем примере кода демонстрируется определение указателя на член `i` в классе `A`. Указатель `pa` инициализируется со значением 0 и становится пустым указателем.

```
class A
{
public:
    int i;
};

int A::*pa = 0;

int main()
{}
```

См. также

[Справочник по языку C++](#)

Встроенные типы (C++)

12.11.2021 • 4 minutes to read

Встроенные типы (также называемые *фундаментальными типами*) задаются стандартом языка C++ и встроены в компилятор. Встроенные типы не определены в файле заголовка. Встроенные типы делятся на три основные категории: *целые, с плавающей запятой и void*. Целочисленные типы представляют целые числа. Типы с плавающей запятой могут указывать значения, которые могут содержать дробные части. Большинство встроенных типов рассматриваются компилятором как отдельные типы. Однако некоторые типы являются *синонимами* или обрабатываются компилятором как эквивалентные типы.

Тип void

`void` Тип описывает пустой набор значений. Невозможно указать переменную типа `void`. `void` Тип используется в основном для объявления функций, которые не возвращают значения, или для объявления универсальных указателей на нетипизированные или произвольные типизированные данные. Любое выражение может быть явно преобразовано или приведено к типу `void`. Однако такие выражения можно использовать только в следующих операторах и операндах:

- в операторе выражения (Дополнительные сведения см. в разделе [выражения](#).)
- в левом операнде оператора запятой (Дополнительные сведения см. в разделе [оператор-запятая](#).)
- во втором и третьем operandах условного оператора (`? :`). (Дополнительные сведения см. в разделе [выражения с условным оператором](#).)

std:: nullptr_t

Ключевое слово `nullptr` является константой указателя NULL типа `std::nullptr_t`, которая преобразуется в любой Необработанный тип указателя. Дополнительные сведения см. на веб-сайте [nullptr](#).

Тип Boolean

`bool` Тип может иметь значения `true` и `false`. Размер `bool` типа зависит от конкретной реализации. См. раздел [размеры встроенных типов](#) для деталей реализации, связанных с Майкрософт.

Символьные типы

`char` Тип является типом символьного представления, который эффективно кодирует члены базовой кодировки выполнения. Компилятор C++ обрабатывает переменные типа `char`, и, `signed char` `unsigned char` в отличие от разных типов.

Зависящие от Майкрософт: переменные типа `char` помещаются в `int` тип по `signed char` умолчанию, если не `/J` используется параметр компиляции. В этом случае они рассматриваются как тип `unsigned char` и переносятся в `int` без расширения знака.

Переменная типа `wchar_t` является расширенным символом или типом многобайтового символа. Используйте `L` префикс перед символьным или строковым литералом, чтобы указать тип расширенных символов.

Для конкретного Майкрософт: по умолчанию `wchar_t` является собственным типом, но можно

использовать `/Zc:wchar_t-` для создания определения типа `wchar_t` для `unsigned short`. `__wchar_t` Тип является синонимом для собственного типа, характерным для Microsoft `wchar_t`.

`char8_t` Тип используется для символьного представления UTF-8. Он имеет то же представление `unsigned char`, что и, но обрабатывается компилятором как отдельный тип. `char8_t` Тип является новым в C++ 20. Для **Майкрософт**: для использования `char8_t` требуется `/std:c++latest` параметр компилятора.

`char16_t` Тип используется для символьного представления UTF-16. Он должен быть достаточно большим, чтобы представлять любой блок кода UTF-16. Компилятор обрабатывает его как отдельный тип.

`char32_t` Тип используется для символьного представления UTF-32. Он должен быть достаточно большим, чтобы представлять любую единицу кода UTF-32. Компилятор обрабатывает его как отдельный тип.

Типы с плавающей запятой

Типы с плавающей запятой используют представление IEEE-754, чтобы обеспечить приближение дробных значений к широкому диапазону величин. В следующей таблице перечислены типы с плавающей запятой в C++ и сравнительные ограничения размеров типов с плавающей запятой. Эти ограничения задаются стандартом C++ и не зависят от реализации Майкрософт. Абсолютный размер встроенных типов с плавающей запятой не указан в стандарте.

ТИП	СОДЕРЖИМОЕ
<code>float</code>	Тип <code>float</code> — это наименьший тип с плавающей запятой в C++.
<code>double</code>	Тип <code>double</code> — это тип с плавающей запятой, который больше или равен типу <code>float</code> , но меньше или равен размеру типа <code>long double</code> .
<code>long double</code>	Тип <code>long double</code> — это тип с плавающей запятой, который больше или равен типу <code>double</code> .

Конечно для Майкрософт: представление `long double` и `double` идентично. Однако `long double` компилятор обрабатывает как отдельные типы. Компилятор Microsoft C++ использует 4-и 8-байтовые представления с плавающей запятой в формате IEEE-754. Дополнительные сведения см. в разделе [IEEE с плавающей точкой](#).

Целочисленные типы

`int` Тип является базовым целочисленным типом по умолчанию. Он может представлять все целые числа в диапазоне, зависящем от реализации.

Представление целого числа *с знаком* — это одно из значений, которое может содержать положительные и отрицательные значения. Он используется по умолчанию или при `signed` наличии ключевого слова модификатора. `unsigned` Ключевое слово модификатор задает *Неподписанное* представление, которое может содержать только неотрицательные значения.

Модификатор размера задает ширину в битах используемого представления целых чисел. Язык поддерживает `short` `long` модификаторы, и `long long`. `short` Тип должен быть не менее 16 бит в ширину. `long` Тип должен быть не менее 32 бит в ширину. `long long` Тип должен быть не менее 64 бит в ширину. Стандартный задает отношение размера между целыми типами:

```
1 == sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long) <= sizeof(long long)
```

Реализация должна поддерживать как минимальные требования к размеру, так и отношение размера для каждого типа. Однако фактические размеры могут и зависеть от реализации. См. раздел [размеры встроенных типов](#) для деталей реализации, связанных с Microsoft.

`int` Ключевое слово можно опустить, если `signed` `unsigned` заданы модификаторы, или. Модификаторы и `int` тип, если они есть, могут использоваться в любом порядке. Например, `short unsigned` и `unsigned int short` следует ссылаться на один и тот же тип.

Синонимы целочисленного типа

Компилятор считает синонимами следующие группы типов:

- `short`, `short int`, `signed short`, `signed short int`
- `unsigned short`, `unsigned short int`
- `int`, `signed`, `signed int`
- `unsigned`, `unsigned int`
- `long`, `long int`, `signed long`, `signed long int`
- `unsigned long`, `unsigned long int`
- `long long`, `long long int`, `signed long long`, `signed long long int`
- `unsigned long long`, `unsigned long long int`

Целочисленные типы, определяемые **корпорацией Microsoft**, включают в себя конкретные `_int8` типы, `_int16` `_int32` и `_int64`. Эти типы могут использовать `signed` `unsigned` модификаторы и. Тип данных `_int8` аналогичен типу `char`, `_int16` — типу `short`, `_int32` — типу `int`, а `_int64` — типу `long long`.

Размеры встроенных типов

Большинство встроенных типов имеют размеры, определенные реализацией. В следующей таблице перечислены объемы хранилища, необходимые для встроенных типов в Microsoft C++. В частности, `long` имеет 4 байта даже в 64-разрядных операционных системах.

ТИП	РАЗМЕР
<code>bool</code> , <code>char</code> , <code>char8_t</code> , <code>unsigned char</code> , <code>signed char</code> , <code>_int8</code>	1 байт
<code>char16_t</code> , <code>_int16</code> , <code>short</code> , <code>unsigned short</code> , <code>wchar_t</code> , <code>_wchar_t</code>	2 байта
<code>char32_t</code> , <code>float</code> , <code>_int32</code> , <code>int</code> , <code>unsigned int</code> , <code>long</code> , <code>unsigned long</code>	4 байта
<code>double</code> , <code>_int64</code> , <code>long double</code> , <code>long long</code> , <code>unsigned long long</code>	8 байт

Сведения о диапазоне значений каждого типа см. в разделе [диапазоны типов данных](#).

Дополнительные сведения о преобразовании типов см. в разделе [стандартные преобразования](#).

См. также

[Диапазоны типов данных](#)

Диапазоны типов данных

12.11.2021 • 2 minutes to read

Компиляторы Microsoft C++ 32-bit и 64-bit распознают типы в таблице далее в этой статье.

- `int` (`unsigned int`)
- `_int8` (`unsigned _int8`)
- `_int16` (`unsigned _int16`)
- `_int32` (`unsigned _int32`)
- `_int64` (`unsigned _int64`)
- `short` (`unsigned short`)
- `long` (`unsigned long`)
- `long long` (`unsigned long long`)

Если имя начинается с двух символов подчеркивания (`_`), тип данных является нестандартным.

Диапазоны, представленные в следующей таблице, включают указанные значения.

ИМЯ ТИПА	БАЙТЫ	ДРУГИЕ ИМЕНА	ДИАПАЗОН ЗНАЧЕНИЙ
<code>int</code>	4	<code>signed</code>	От -2 147 483 648 до 2 147 483 647
<code>unsigned int</code>	4	<code>unsigned</code>	От 0 до 4 294 967 295
<code>_int8</code>	1	<code>char</code>	От -128 до 127
<code>unsigned _int8</code>	1	<code>unsigned char</code>	От 0 до 255
<code>_int16</code>	2	<code>short</code> , <code>short int</code> , <code>signed short int</code>	От -32 768 до 32 767
<code>unsigned _int16</code>	2	<code>unsigned short</code> , <code>unsigned short int</code>	От 0 до 65 535
<code>_int32</code>	4	<code>signed</code> , <code>signed int</code> , <code>int</code>	От -2 147 483 648 до 2 147 483 647
<code>unsigned _int32</code>	4	<code>unsigned</code> , <code>unsigned int</code>	От 0 до 4 294 967 295
<code>_int64</code>	8	<code>long long</code> , <code>signed long long</code>	От -9 223 372 036 854 775 807 до 9 223 372 036 854 775 807

ИМЯ ТИПА	БАЙТЫ	ДРУГИЕ ИМЕНА	ДИАПАЗОН ЗНАЧЕНИЙ
<code>unsigned __int64</code>	8	<code>unsigned long long</code>	От 0 до 18 446 744 073 709 551 615
<code>bool</code>	1	нет	<code>false</code> ИЛИ <code>true</code>
<code>char</code>	1	нет	от -128 до 127 по умолчанию от 0 до 255 при компиляции с помощью /J
<code>signed char</code>	1	нет	От -128 до 127
<code>unsigned char</code>	1	нет	От 0 до 255
<code>short</code>	2	<code>short int</code> , <code>signed short int</code>	От -32 768 до 32 767
<code>unsigned short</code>	2	<code>unsigned short int</code>	От 0 до 65 535
<code>long</code>	4	<code>long int</code> , <code>signed long int</code>	От -2 147 483 648 до 2 147 483 647
<code>unsigned long</code>	4	<code>unsigned long int</code>	От 0 до 4 294 967 295
<code>long long</code>	8	нет (но эквивалентно <code>__int64</code>)	От -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
<code>unsigned long long</code>	8	нет (но эквивалентно <code>unsigned __int64</code>)	От 0 до 18 446 744 073 709 551 615
<code>enum</code>	непостоянно	нет	
<code>float</code>	4	нет	3,4E +/- 38 (7 знаков)
<code>double</code>	8	нет	1,7E +/- 308 (15 знаков)
<code>long double</code>	то же, что <code>double</code>	нет	То же, что <code>double</code>
<code>wchar_t</code>	2	<code>__wchar_t</code>	От 0 до 65 535

В зависимости от того, как он используется, переменная `__wchar_t` определяет либо тип расширенного символа, либо многобайтовый символ. Чтобы указать константу расширенного символьного типа, перед символьной или строковой константой следует использовать префикс `L`.

`signed` И `unsigned` — это модификаторы, которые можно использовать с любым целочисленным типом, за исключением `bool`. Обратите внимание, что `char`, `signed char` И `unsigned char` — это три различных типа для таких механизмов, как перегрузка и шаблоны.

`int` Типы и `unsigned int` имеют размер четыре байта. Однако переносимый код не должен зависеть от размера, `int` так как языковой стандарт позволяет реализовать его в зависимости от реализации.

С и C++ в Visual Studio также поддерживают целочисленные типы с указанием размера. Дополнительные сведения см. в разделе [__int8, __int16, __int32, __int64](#) и [ограничения целых чисел](#).

Дополнительные сведения об ограничениях размеров каждого типа см. в разделе [Встроенные типы](#).

Диапазон перечисляемых типов зависит от контекста языка и указанных флагков компилятора.

Дополнительные сведения см. в статьях [Объявления перечислений С](#) и [Объявления перечислений С++](#).

См. также

[Ключевые слова](#)

[Встроенные типы](#)

nullptr

12.11.2021 • 2 minutes to read

`nullptr` Ключевое слово указывает константу указателя NULL типа `std::nullptr_t`, которая преобразуется в любой Необработанный тип указателя. Хотя можно использовать ключевое слово `nullptr` без включения заголовков, если код использует тип `std::nullptr_t`, его необходимо определить, включив заголовок `<cstddef>`.

NOTE

`nullptr` Ключевое слово также определено в C++/CLI для приложений управляемого кода и не является взаимозаменяемым с ключевым словом C++ Standard в стандарте ISO. Если код может быть скомпилирован с помощью `/clr` параметра компилятора, который предназначен для управляемого кода, используйте `__nullptr` в любой строке кода, где необходимо гарантировать, что компилятор использует собственную интерпретацию C++. Дополнительные сведения см. в разделе `nullptr` (C++/CLI и C++/CX).

Remarks

Избегайте использования `NULL` или нулевого значения (`0`) в качестве константы указателя NULL; `nullptr` менее уязвимо к неправильному использованию и лучше всего работает в большинстве случаев. Например, если для функции `func(std::pair<const char *, double>)` произвести вызов `func(std::make_pair(NULL, 3.14))`, возникнет ошибка компилятора. Макрос `NULL` разворачивается в `0`, поэтому вызов `std::make_pair(0, 3.14)` возвращает `std::pair<int, double>`, который не может быть преобразован в `std::pair<const char *, double>` тип параметра в `func`. Вызов `func(std::make_pair(nullptr, 3.14))` успешно компилируется, поскольку `std::make_pair(nullptr, 3.14)` возвращает значение `std::pair<std::nullptr_t, double>`, которое допускает преобразование в тип `std::pair<const char *, double>`.

См. также

Ключевые слова

`nullptr` (C++/CLI и C++/CX)

void (C++)

12.11.2021 • 2 minutes to read

При использовании в качестве возвращаемого типа функции `void` ключевое слово указывает, что функция не возвращает значение. При использовании для списка параметров функции `void` указывает, что функция не принимает параметров. При использовании в объявлении указателя `void` указывает, что указатель является универсальным.

Если тип указателя — `void * _`, **указатель может указывать на любую переменную, которая не объявлена с `const`** ключевым словом `_` или `volatile`. Указатель `void * _` **не может быть разыменован, если он не приведен к другому типу**. Указатель `_ void *` можно преобразовать в любой другой тип указателя данных.

`void` Указатель может указывать на функцию, но не на член класса в C++.

Нельзя объявить переменную типа `void`.

Пример

```
// void.cpp
void vobject;    // C2182
void *pv;        // okay
int *pint; int i;
int main() {
    pv = &i;
    // Cast optional in C required in C++
    pint = (int *)pv;
}
```

См. также раздел

[Ключевые слова](#)

[Встроенные типы](#)

bool (C++)

12.11.2021 • 2 minutes to read

Это ключевое слово является встроенным типом. Переменная этого типа может иметь значения `true` и `false`. Условные выражения имеют тип `bool` и поэтому имеют значения типа `bool`. Например, `i != 0` теперь имеет `true` значение или в `false` зависимости от значения `i`.

Visual Studio 2017 версии 15,3 и более поздних версий (доступно в [/std: c++ 17](#)): операнд постфиксного или инкрементного оператора или оператор декремента не может иметь тип `bool`. Иными словами, при наличии переменной `b` типа `bool` эти выражения больше не разрешаются:

```
b++;  
++b;  
b--;  
--b;
```

Значения `true` и `false` имеют следующую связь:

```
!false == true  
!true == false
```

В следующем операторе

```
if (condexpr1) statement1;
```

Если `condexpr1` имеет значение `true`, `statement1` всегда выполняется; если `condexpr1` имеет значение, то никогда не `false`, `statement1` выполняется.

Если к переменной типа применяется постфиксный или префиксный `++` оператор `bool`, переменной присваивается значение `true`.

Visual Studio 2017 версии 15,3 и более поздних версий: `operator++` для `bool` была удалена из языка и больше не поддерживается.

Постфиксный или префиксный `--` оператор не может применяться к переменной этого типа.

`bool` Тип участвует в стандартных повышениях. R-значение типа `bool` может быть преобразовано в значение r-value типа, которое `int` `false` становится равным нулю и `true` становится одним. В качестве отдельного типа `bool` участвует в разрешении перегрузки.

См. также

[Ключевые слова](#)

[Встроенные типы](#)

false (C++)

12.11.2021 • 2 minutes to read

Ключевое слово является одним из двух значений для переменной типа `bool` или условного выражения (условное выражение теперь является `true` логическим выражением). Например, если `i` является переменной типа `bool`, `i = false;` оператор присваивает значение `false` `i`.

Пример

```
// bool_false.cpp
#include <stdio.h>

int main()
{
    bool bb = true;
    printf_s("%d\n", bb);
    bb = false;
    printf_s("%d\n", bb);
}
```

```
1
0
```

См. также

[Ключевые слова](#)

true (C++)

12.11.2021 • 2 minutes to read

Синтаксис

```
bool-identifier = true ;
bool-expression logical-operator true ;
```

Remarks

Это ключевое слово является одним из двух значений для переменной типа `bool` или условного выражения (условное выражение теперь является истинным логическим выражением). Если `i` имеет тип `bool`, инструкция присваивает оператору значение `i = true; true i`.

Пример

```
// bool_true.cpp
#include <stdio.h>
int main()
{
    bool bb = true;
    printf_s("%d\n", bb);
    bb = false;
    printf_s("%d\n", bb);
}
```

```
1
0
```

См. также

[Ключевые слова](#)

char, wchar_t, char8_t, char16_t, char32_t

12.11.2021 • 2 minutes to read

Типы `char`, `wchar_t`, `char8_t`, `char16_t` и `char32_t` являются встроенным типами, представляющими буквенно-цифровые символы, неалфавитные глифы и непечатаемые символы.

Синтаксис

```
char     ch1{ 'a' }; // or { u8'a' }
wchar_t  ch2{ L'a' };
char16_t ch3{ u'a' };
char32_t ch4{ U'a' };
```

Remarks

`char` Тип был исходным типом символа в C и C++. Этот `char` тип можно использовать для хранения символов из кодировки ASCII или любой из КОДИРОВОК ISO-8859, а также для отдельных байтов многобайтовых символов, таких как Shift-JIS или кодировка UTF-8 для набора символов Юникода. В компиляторе Microsoft `char` — это 8-разрядный тип. Это отдельный тип от `signed char` и `unsigned char`. По умолчанию переменные типа `char` получают значение `int` как если бы тип, если `signed char` только `/J` не используется параметр компилятора. В разделе `/J` они рассматриваются как тип `unsigned char` и получают повышенный уровень `int` без расширения знака.

Тип `unsigned char` часто используется для представления *байта*, который не является встроенным типом в C++.

`wchar_t` Тип является определяемым реализацией типом расширенных символов. в компиляторе майкрософт он представляет 16-разрядный символ, используемый для хранения юникода в кодировке UTF-16le, собственный тип символов в операционных системах Windows. версии расширенных символов функций библиотеки универсальной среды выполнения C (UCRT) используют `wchar_t` и его указатели и типы массивов в качестве параметров и возвращаемых значений, как и версии расширенных символов собственного API Windows.

`char8_t` Типы, `char16_t` и `char32_t` представляют 8-разрядные, 16-разрядные и 32-разрядные символы, соответственно. (`char8_t` является новым в c++ 20 и требует `/std:c++latest` параметра компилятора.) Юникод в кодировке UTF-8 может храниться в `char8_t` типе. Строки `char8_t` и `char` типа называются *узкими строками*, даже если они используются для кодирования Юникода или многобайтовых символов. Юникод в кодировке UTF-16 может храниться в `char16_t` типе, а Юникод в кодировке UTF-32 может храниться в `char32_t` типе. Строки этих типов и `wchar_t` все они называются *расширенными строками*, хотя термин часто относится к строкам `wchar_t` типа.

В стандартной библиотеке C++ `basic_string` тип является специализированным для узких и широких строк. Используйте, если символы имеют тип, если символы имеют тип, если символы имеют тип, `std::string`, `char` а также `std::u8string`, `char8_t`, `std::u16string`, `char16_t`, `std::u32string`, `char32_t` `std::wstring` когда символы `wchar_t` имеют тип. Другие типы, представляющие текст, включая `std::stringstream` и, `std::cout` имеют специализации для узких и расширенных строк.

`_int8, _int16, _int32, _int64`

12.11.2021 • 2 minutes to read

Только для систем Майкрософт

В Microsoft C/C++ поддерживаются целочисленные типы с указанием размера. Можно объявить 8-, 16-, 32-или 64-разрядные целочисленные переменные с помощью `_intN` спецификатора типа, где `N` — 8, 16, 32 или 64.

В следующем примере объявляется по одной переменной каждого из этих целочисленных типов с указанием размера:

```
_int8 nSmall;      // Declares 8-bit integer
_int16 nMedium;    // Declares 16-bit integer
_int32 nLarge;     // Declares 32-bit integer
_int64 nHuge;      // Declares 64-bit integer
```

Типы `_int8`, `_int16` и `_int32` являются синонимами для типов ANSI, имеющих одинаковый размер, и полезны для написания переносимого кода, который ведет себя одинаково на нескольких платформах.

`_int8` Тип данных является синонимом типа `char`, `_int16` является синонимом типа `short` и `_int32` является синонимом типа `int`. `_int64` Тип является синонимом типа `long long`.

Для совместимости с предыдущими версиями, `_int8`, `_int16`, `_int32` и `_int64` являются синонимами для `_int8`, `_int16`, `_int32` и, `_int64` если не указан параметр компилятора, не [/za](#) (отключающий расширения языка).

Пример

В следующем примере показано, что `_intN` параметр будет выдвинут на `int`:

```
// sized_int_types.cpp

#include <stdio.h>

void func(int i) {
    printf_s("%s\n", __FUNCTION__);
}

int main()
{
    _int8 i8 = 100;
    func(i8);    // no void func(_int8 i8) function
                 // _int8 will be promoted to int
}
```

```
func
```

См. также раздел

[Ключевые слова](#)

[Встроенные типы](#)

Диапазоны типов данных

__m64

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

__m64 Тип данных предназначен для использования с технологией MMX и 3DNow! встроенные функции и определяются в <xmmmintrin.h> .

```
// data_types__m64.cpp
#include <xmmmintrin.h>
int main()
{
    __m64 x;
```

Remarks

Нет доступа к **__m64** полям напрямую. Однако можно просматривать эти типы в отладчике. Переменная типа **__m64** сопоставляется с регистрами mm [0-7].

Переменные типа **_m64** автоматически согласовываются с границами в 8 байт.

__m64 Тип данных не поддерживается на процессорах x64. Приложения, использующие тип **_m64** в составе встроенных инструкций MMX, необходимо переписать с использованием эквивалентных встроенных инструкций SSE и SSE2.

Завершение блока, относящегося только к системам Microsoft

См. также

[Ключевые слова](#)

[Встроенные типы](#)

[Диапазоны типов данных](#)

`_m128`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Тип **данных** `_m128`, предназначенный для использования с внутренними расширениями Streaming SIMD и Streaming SIMD Extensions 2, определяется в `<xmmmintrin.h>`.

```
// data_types_m128.cpp
#include <xmmmintrin.h>
int main() {
    _m128 x;
}
```

Remarks

Нет доступа к `_m128` полям напрямую. Однако можно просматривать эти типы в отладчике.

Переменная типа `_m128` сопоставляется с регистрами XMM [0 – 7].

Переменные типа `_m128` автоматически согласовываются с 16-байтовыми границами.

`_m128` Тип данных не поддерживается в процессорах ARM.

Завершение блока, относящегося только к системам Microsoft

См. также

[Ключевые слова](#)

[Встроенные типы](#)

[Диапазоны типов данных](#)

`_m128d`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

`_m128d` Тип данных, используемый с внутренними инструкциями Streaming SIMD Extensions 2, определяется в `<emmintrin.h>`.

```
// data_types_m128d.cpp
#include <emmintrin.h>
int main() {
    _m128d x;
}
```

Remarks

Нет доступа к `_m128d` полям напрямую. Однако можно просматривать эти типы в отладчике.

Переменная типа `_m128` сопоставляется с регистрами XMM [0 – 7].

Переменные типа `_m128d` автоматически выводятся по 16-байтным границам.

`_m128d` Тип данных не поддерживается в процессорах ARM.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Ключевые слова](#)

[Встроенные типы](#)

[Диапазоны типов данных](#)

`_m128i`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

`_m128i` Тип данных, используемый с внутренними инструкциями Streaming SIMD Extensions 2 (SSE2), определяется в `<emmintrin.h>`.

```
// data_types_m128i.cpp
#include <emmintrin.h>
int main() {
    _m128i x;
}
```

Remarks

Нет доступа к `_m128i` полям напрямую. Однако можно просматривать эти типы в отладчике.

Переменная типа `_m128i` сопоставляется с регистрами XMM [0 – 7].

Переменные типа `_m128i` автоматически согласовываются с 16-байтовыми границами.

NOTE

Использование переменных типа `_m128i` приведет к тому, что компилятор создаст `movdqa` инструкцию SSE2. Эта инструкция не приводит к сбою в работе процессоров Pentium III, но приводит к неудачному сбою, что может привести к появлению неисправностей, вызванных любыми инструкциями, которые `movdqa` преобразуются в процессоры Pentium III.

`_m128i` Тип данных не поддерживается в процессорах ARM.

Завершение блока, относящегося только к системам Microsoft

См. также

[Ключевые слова](#)

[Встроенные типы](#)

[Диапазоны типов данных](#)

`_ptr32, _ptr64`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

`_ptr32` представляет собственный указатель в 32-разрядной системе, а `_ptr64` представляет собственный указатель в 64-разрядной системе.

В следующем примере показано, как объявить каждый из этих типов указателей.

```
int * __ptr32 p32;
int * __ptr64 p64;
```

В 32-разрядной системе указатель, объявленный с параметром, `_ptr64` усекается до 32-разрядного указателя. В 64-разрядной системе указатель, объявленный с параметром, `_ptr32` приводится к 64-разрядному указателю.

NOTE

Нельзя использовать `_ptr32` или `_ptr64` при компиляции с **параметром/clr: pure**. В противном случае будет сформирована ошибка компилятора C2472. параметры компилятора /clr: pure и /clr: **сейф** являются устаревшими в Visual Studio 2015 и не поддерживаются в Visual Studio 2017.

Для совместимости с предыдущими версиями `_ptr32` и `_ptr64` являются синонимами для `_ptr32` и `_ptr64`, если только параметр компилятора [/ЗА НЕ \(отключил расширения языка\)](#) указан.

Пример

В следующем примере показано, как объявить и выделить указатели с помощью `_ptr32` `_ptr64` ключевых слов и.

```
#include <cstdlib>
#include <iostream>

int main()
{
    using namespace std;

    int * __ptr32 p32;
    int * __ptr64 p64;

    p32 = (int * __ptr32)malloc(4);
    *p32 = 32;
    cout << *p32 << endl;

    p64 = (int * __ptr64)malloc(4);
    *p64 = 64;
    cout << *p64 << endl;
}
```

Завершение блока, относящегося только к системам Microsoft

См. также

[Встроенные типы](#)

Числовые пределы (C++)

12.11.2021 • 2 minutes to read

Два стандартных включаемых файла, `<limits.h>` и `<float.h>`, определяют числовые ограничения или минимальное и максимальное значения, которые могут храниться в переменной данного типа. Эти минимальные и максимальные значения гарантированно переносимы на любой компилятор C++, использующий то же представление данных, что и ANSI C. `<limits.h>` Включаемый файл определяет [Числовые ограничения для целочисленных типов](#) `<float.h>` определяет [Числовые ограничения для плавающих типов](#).

См. также

[Основные понятия](#)

Пределы целых чисел

12.11.2021 • 2 minutes to read

Только для систем Майкрософт

Ограничения для целочисленных типов представлены в следующей таблице. Макросы препроцессора для этих ограничений также определяются при включении стандартного файла заголовка <climits> .

Ограничения для целочисленных констант

КОНСТАНТА	ЗНАЧЕНИЕ	ЗНАЧЕНИЕ
<code>CHAR_BIT</code>	Количество битов в наименьшей переменной, которая не является битовым полем.	8
<code>SCHAR_MIN</code>	Минимальное значение для переменной типа <code>signed char</code> .	-128
<code>SCHAR_MAX</code>	Максимальное значение для переменной типа <code>signed char</code> .	127
<code>UCHAR_MAX</code>	Максимальное значение для переменной типа <code>unsigned char</code> .	255 (0xff)
<code>CHAR_MIN</code>	Минимальное значение для переменной типа <code>char</code> .	-128; 0, если <code>/J</code> используется параметр
<code>CHAR_MAX</code>	Максимальное значение для переменной типа <code>char</code> .	127; 255, если <code>/J</code> используется параметр
<code>MB_LEN_MAX</code>	Максимальное количество байтов в многосимвольной константе.	5
<code>SHRT_MIN</code>	Минимальное значение для переменной типа <code>short</code> .	-32768
<code>SHRT_MAX</code>	Максимальное значение для переменной типа <code>short</code> .	32767
<code>USHRT_MAX</code>	Максимальное значение для переменной типа <code>unsigned short</code> .	65 535 (0xffff)
<code>INT_MIN</code>	Минимальное значение для переменной типа <code>int</code> .	-2147483648
<code>INT_MAX</code>	Максимальное значение для переменной типа <code>int</code> .	2147483647
<code>UINT_MAX</code>	Максимальное значение для переменной типа <code>unsigned int</code> .	4 294 967 295 (0xffffffff)

КОНСТАНТА	ЗНАЧЕНИЕ	ЗНАЧЕНИЕ
LONG_MIN	Минимальное значение для переменной типа <code>long</code> .	-2147483648
LONG_MAX	Максимальное значение для переменной типа <code>long</code> .	2147483647
ULONG_MAX	Максимальное значение для переменной типа <code>unsigned long</code> .	4 294 967 295 (0xffffffff)
LLONG_MIN	Минимальное значение для переменной типа <code>long long</code>	-9223372036854775808
LLONG_MAX	Максимальное значение для переменной типа <code>long long</code>	9223372036854775807
ULLONG_MAX	Максимальное значение для переменной типа <code>unsigned long long</code>	18446744073709551615 (0xfffffffffffffffff)

Если значение превышает максимально возможное представление целочисленного типа, компилятор Microsoft выдает ошибку.

См. также

[Плавающие ограничения](#)

Пределы констант с плавающей запятой

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

В следующей таблице представлены ограничения на значения констант с плавающей запятой. Эти ограничения также определяются в стандартном файле заголовка `<float.h>`.

Ограничения на константы с плавающей запятой

КОНСТАНТА	ЗНАЧЕНИЕ	ЗНАЧЕНИЕ
<code>FLT_DIG</code> <code>DBL_DIG</code> <code>LDBL_DIG</code>	Количество цифр q , при котором число с плавающей запятой с q десятичными цифрами можно округлить в представление с плавающей запятой и обратно без потери точности.	6 15 15
<code>FLT_EPSILON</code> <code>DBL_EPSILON</code> <code>LDBL_EPSILON</code>	Наименьшее положительное число x , при котором $x + 1,0$ не равно $1,0$.	1,192092896e-07F 2,2204460492503131e-016 2,2204460492503131e-016
<code>FLT_GUARD</code>		0
<code>FLT_MANT_DIG</code> <code>DBL_MANT_DIG</code> <code>LDBL_MANT_DIG</code>	Количество цифр в системе счисления, заданное <code>FLT_RADIX</code> в параметре значащим с плавающей запятой. Основание системы счисления — 2; Поэтому эти значения задают биты.	24 53 53
<code>FLT_MAX</code> <code>DBL_MAX</code> <code>LDBL_MAX</code>	Максимальное представимое число с плавающей запятой.	3,402823466e+38F 1,7976931348623158e+308 1,7976931348623158e+308
<code>FLT_MAX_10_EXP</code> <code>DBL_MAX_10_EXP</code> <code>LDBL_MAX_10_EXP</code>	Максимальное целое число, равное 10, возведенное в это число, является представимым числом с плавающей запятой.	38 308 308
<code>FLT_MAX_EXP</code> <code>DBL_MAX_EXP</code> <code>LDBL_MAX_EXP</code>	Максимальное целое число, <code>FLT_RADIX</code> возведенное в это число, является представимым числом с плавающей запятой.	128 1024 1024
<code>FLT_MIN</code> <code>DBL_MIN</code> <code>LDBL_MIN</code>	Минимальное положительное значение.	1,175494351e-38F 2,2250738585072014e-308 2,2250738585072014e-308

КОНСТАНТА	ЗНАЧЕНИЕ	ЗНАЧЕНИЕ
<code>FLT_MIN_10_EXP</code> <code>DBL_MIN_10_EXP</code> <code>LDBL_MIN_10_EXP</code>	Минимальное отрицательное целое число, которое указывает, что 10, возведенное в это число, является представимым числом с плавающей запятой.	-37 -307 -307
<code>FLT_MIN_EXP</code> <code>DBL_MIN_EXP</code> <code>LDBL_MIN_EXP</code>	Минимальное отрицательное целое число, <code>FLT_RADIX</code> возведенное в это число, является представимым числом с плавающей запятой.	-125 -1021 -1021
<code>FLT_NORMALIZE</code>		0
<code>FLT_RADIX</code> <code>_DBL_RADIX</code> <code>_LDBL_RADIX</code>	Основание экспоненциальной формы представления.	2 2 2
<code>FLT_ROUNDS</code> <code>_DBL_ROUNDS</code> <code>_LDBL_ROUNDS</code>	Режим округления для сложения с плавающей запятой.	1 (приблизительно) 1 (приблизительно) 1 (приблизительно)

NOTE

В будущих версиях продукта информация из приведенной выше таблицы может отличаться.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Целочисленные ограничения](#)

Объявления и определения (C++)

12.11.2021 • 4 minutes to read

Программа на C++ состоит из различных сущностей, таких как переменные, функции, типы и пространства имен. Каждая из этих сущностей должна быть *объявлена*, прежде чем их можно будет использовать. В объявлении указывается уникальное имя сущности, а также сведения о ее типе и других характеристиках. В C++ точка, в которой объявлено имя, является точкой, в которой он становится видимым для компилятора. Нельзя ссылаться на функцию или класс, объявленные в более поздней точке в единице компиляции. Переменные должны быть объявлены как можно ближе до точки, в которой они используются.

В следующем примере показаны некоторые объявления:

```
#include <string>

int f(int i); // forward declaration

int main()
{
    const double pi = 3.14; //OK
    int i = f(2); //OK. f is forward-declared
    std::string str; // OK std::string is declared in <string> header
    C obj; // error! C not yet declared.
    j = 0; // error! No type specified.
    auto k = 0; // OK. type inferred as int by compiler.
}

int f(int i)
{
    return i + 42;
}

namespace N {
    class C{/*...*/};
}
```

В строке 5 `main` объявлена функция. В строке 7 `const pi` объявлена и инициализирована переменная с именем. В строке 8 целое число `i` объявляется и инициализируется значением, созданным функцией `f`. Имя `f` является видимым для компилятора из-за *прямого объявления* в строке 3.

В строке 9 `obj` объявлена переменная типа с именем `c`. Однако это объявление вызывает ошибку, так как `c` не объявлено до последующих появлений в программе и не объявлена как прямая. Чтобы устранить эту ошибку, можно либо переместить все *Определение c* ранее, `main` либо добавить в нее прямую декларацию. Это поведение отличается от других языков, таких как C#, в которых функции и классы можно использовать до их точки объявления в исходном файле.

В строке 10 `str` объявлена переменная типа с именем `std::string`. Имя `std::string` является видимым, так как оно представлено в `string` *файле заголовка*, который объединяется с исходным файлом в строке 1. `std` пространство имен, в котором `string` объявлен класс.

В строке 11 возникает ошибка, так как имя `j` не было объявлено. Объявление должно предоставлять тип, в отличие от других языков, таких как JavaScript. В строке 12 `auto` используется ключевое слово, которое указывает компилятору вывести тип `k` на основе значения, с которым он инициализируется. Компилятор в этом случае выбирает `int` тип.

Область видимости объявления

Имя, введенное в объявлении, допустимо в пределах *области*, в которой происходит объявление. В предыдущем примере переменные, объявленные внутри `main` функции, являются *локальными переменными*. Можно объявить другую переменную `i`, именованную за пределами `Main`, в *глобальной области*, и это будет отдельная сущность. Однако такое дублирование имен может привести к путанице и ошибкам программиста, и их следует избегать. В строке 21 класс `c` объявляется в области видимости пространства имен `n`. Использование пространств имен помогает избежать *конфликтов имен*. Большинство имен стандартных библиотек C++ объявляются в `std` пространстве имен. Дополнительные сведения о взаимодействии правил определения области с объявлениями см. в разделе [Scope](#).

Определения

Некоторые сущности, включая функции, классы, перечисления и константные переменные, должны быть определены и объявлены. *Определение* предоставляет компилятору все сведения, необходимые для создания машинного кода, когда сущность используется позже в программе. В предыдущем примере строка 3 содержит объявление для функции, `f` но *Определение* функции предоставляется в строках с 15 по 18. В строке 21 класс `c` объявлен и определен (хотя в соответствии с определением класс не выполняет никаких действий). Константная переменная должна быть определена, иными словами, которой было присвоено значение, в той же инструкции, в которой она объявлена. Объявление встроенного типа, например `int`, автоматически определяется определением, так как компилятор знает, сколько пространства нужно выделить.

В следующем примере показаны объявления, которые также являются определениями:

```
// Declare and define int variables i and j.  
int i;  
int j = 10;  
  
// Declare enumeration suits.  
enum suits { Spades = 1, Clubs, Hearts, Diamonds };  
  
// Declare class CheckBox.  
class CheckBox : public Control  
{  
public:  
    Boolean IsChecked();  
    virtual int     ChangeState() = 0;  
};
```

Ниже приведены некоторые объявления, которые не являются определениями.

```
extern int i;  
char *strchr( const char *Str, const char Target );
```

Определения типов и операторы `using`

В более старых версиях C++ `typedef` ключевое слово используется для объявления нового имени, которое является *псевдонимом* для другого имени. Например, типом `std::string` является другое имя для `std::basic_string<char>`. Должно быть очевидно, почему программисты используют имя `typedef`, а не фактическое имя. В современных C++ `using` ключевое слово предпочтительнее `typedef`, но идея одинакова: новое имя объявляется для сущности, которая уже объявлена и определена.

Члены статических классов

Поскольку члены статических данных класса являются дискретными переменными, общими для всех объектов класса, они должны быть определены и инициализированы вне определения класса.
(Дополнительные сведения см. в разделе [классы](#).)

Объявления `extern`

Программа на C++ может содержать более одной [единицы компиляции](#). Чтобы объявить сущность, определенную в отдельной единице компиляции, используйте `extern` ключевое слово. Сведения в объявлении достаточны для компилятора, но если определение сущности не удается найти на шаге компоновки, то компоновщик вызовет ошибку.

Содержимое раздела

[Классы хранения в C](#)

`const`

`constexpr`

`extern`

[Инициализаторы](#)

[Псевдонимы и определения типов](#)

`using` [повторно](#)

`volatile`

`decltype`

[Атрибуты в C++](#)

См. также

[Основные понятия](#)

Классы хранения

12.11.2021 • 7 minutes to read

Класс хранения в контексте объявлений переменных C++ — это описатель типа, который управляет временем существования, компоновкой и расположением памяти объектов. Каждый объект может иметь только один класс хранения. Переменные, определенные в блоке, имеют автоматическое хранение, если не указано иное с помощью `extern` `static` `thread_local` описателей, или. Автоматически создаваемые объекты и переменные не имеют компоновки. Они не доступны для кода за пределами блока. Память выделяется для них автоматически, когда выполнение входит в блок и освобождается при выходе из блока.

Примечания

1. Ключевое слово `mutable` может рассматриваться как описатель класса хранения. Однако он доступен только в списке членов в определении класса.
2. **Visual Studio 2010 и более поздних версий:** `auto` Ключевое слово больше не является описателем класса хранения C++, а `register` ключевое слово является устаревшим. **Visual Studio 2017 версии 15,7 и более поздних версий:** (доступно с `/std:c++17`): `register` Ключевое слово удаляется из языка C++.

```
register int val; // warning C5033: 'register' is no longer a supported storage class
```

static

`static` Ключевое слово можно использовать для объявления переменных и функций в глобальной области видимости, области пространства имен и области класса. Статические переменные также могут быть объявлены в локальной области видимости.

Статическая длительность означает, что объект или переменная выделяется при запуске программы и освобождается при ее завершении. Внешняя компоновка означает, что имя переменной видно за пределами файла, в котором эта переменная объявлена. Внутренняя компоновка означает, что имя не видно за пределами файла, в котором объявлена переменная. По умолчанию объект или переменная, определенные в глобальном пространстве имен, имеют статическую длительность и внешнюю компоновку. `static` Ключевое слово можно использовать в следующих ситуациях.

1. При объявлении переменной или функции в области видимости файла (глобальной и/или области пространства имен) `static` ключевое слово указывает, что переменная или функция имеет внутреннюю компоновку. При объявлении переменной она имеет статическую длительность и компилятор инициализирует ее со значением 0, если не указано другое значение.
2. При объявлении переменной в функции `static` ключевое слово указывает, что переменная удерживает свое состояние между вызовами этой функции.
3. При объявлении члена данных в объявлении класса `static` ключевое слово указывает, что одна копия элемента совместно используется всеми экземплярами класса. Статические данные-член должны быть определены в области видимости файла. Целочисленный член данных, объявляемый как, `const static` может иметь инициализатор.
4. При объявлении функции-члена в объявлении класса `static` ключевое слово указывает, что

функция совместно используется всеми экземплярами класса. Статическая функция-член не может получить доступ к члену экземпляра, так как функция не имеет неявного `this` указателя. Для доступа к члену экземпляра следует объявить функцию с параметром, являющимся указателем или ссылкой на экземпляр.

5. Объявление членов объединения как статических невозможно. Однако глобально объявленное анонимное объединение должно быть явно объявлено `static`.

В этом примере показано, как переменная `static`, объявленная в функции, удерживает свое состояние между вызовами этой функции.

```
// static1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void showstat( int curr ) {
    static int nStatic;      // Value of nStatic is retained
                            // between each function call
    nStatic += curr;
    cout << "nStatic is " << nStatic << endl;
}

int main() {
    for ( int i = 0; i < 5; i++ )
        showstat( i );
}
```

```
nStatic is 0
nStatic is 1
nStatic is 3
nStatic is 6
nStatic is 10
```

В этом примере показано использование `static` в классе.

```
// static2.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class CMyClass {
public:
    static int m_i;
};

int CMyClass::m_i = 0;
CMyClass myObject1;
CMyClass myObject2;

int main() {
    cout << myObject1.m_i << endl;
    cout << myObject2.m_i << endl;

    myObject1.m_i = 1;
    cout << myObject1.m_i << endl;
    cout << myObject2.m_i << endl;

    myObject2.m_i = 2;
    cout << myObject1.m_i << endl;
    cout << myObject2.m_i << endl;

    CMyClass::m_i = 3;
    cout << myObject1.m_i << endl;
    cout << myObject2.m_i << endl;
}
```

```
0
0
1
1
2
2
3
3
```

В этом примере показана локальная переменная, объявленная `static` в функции-члене. `static` Переменная доступна всей программе; все экземпляры типа совместно используют одну и ту же копию `static` переменной.

```
// static3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
struct C {
    void Test(int value) {
        static int var = 0;
        if (var == value)
            cout << "var == value" << endl;
        else
            cout << "var != value" << endl;

        var = value;
    }
};

int main() {
    C c1;
    C c2;
    c1.Test(100);
    c2.Test(100);
}
```

```
var != value
var == value
```

Начиная с C++ 11, `static` Инициализация локальной переменной гарантирует потокобезопасность. Эта функция иногда называется *магической статичностью*. Однако в многопоточном приложении все последующие назначения должны быть синхронизированы. Потокобезопасную функцию статической инициализации можно отключить с помощью `/Zc:threadSafeInit-` флага, чтобы избежать зависимостей от CRT.

extern

Объекты и переменные, объявленные как `extern` объявляют объект, который определен в другой записи преобразования или во внешней области видимости как внешняя компоновка. Дополнительные сведения см. в разделе [extern](#) и [Преобразование единиц и компоновки](#).

thread_local (C++ 11)

Переменная, объявленная с `thread_local` описателем, доступна только в том потоке, в котором он создан. Переменная создается при создании потока и уничтожается при его уничтожении. Каждый поток имеет свою собственную копию переменной. в Windows `thread_local` функционально эквивалентен атрибуту, характерному для Microsoft `__declspec(thread)` .

```

thread_local float f = 42.0; // Global namespace. Not implicitly static.

struct S // cannot be applied to type definition
{
    thread_local int i; // Illegal. The member must be static.
    thread_local static char buf[10]; // OK
};

void DoSomething()
{
    // Apply thread_local to a local variable.
    // Implicitly "thread_local static S my_struct".
    thread_local S my_struct;
}

```

Обратите внимание на `thread_local` описатель:

- Динамически инициализированные локальные переменные потока в библиотеках DLL могут быть неправильно инициализированы во всех вызывающих потоках. Дополнительные сведения см. на веб-сайте [thread](#).
- `thread_local` Спецификатор можно сочетать с `static` или `extern`.
- Можно применять `thread_local` только к объявлениям и определениям данных; `thread_local` не может использоваться в объявлениях или определениях функций.
- Можно указать `thread_local` только для элементов данных со статической длительностью хранения. Сюда входят глобальные объекты данных (`static` и `extern`), локальные статические объекты и статические члены данных классов. Объявленная локальная переменная `thread_local` неявно является статической, если не предоставлен другой класс хранения; иными словами, в области видимости блока `thread_local` эквивалентна `thread_local static`.
- Необходимо указать `thread_local` как для объявления, так и для определения локального объекта потока, будь то объявление и определение происходят в одном и том же файле или в отдельных файлах.

в Windows `thread_local` функционально эквивалентен `__declspec(thread)` за исключением того, что `*__declspec(thread) *` может применяться к определению типа и является допустимым в коде на языке C. Везде, где это возможно, используйте `thread_local` так как он является частью стандарта C++ и, таким образом, более переносим.

зарегистрировать

Visual Studio 2017 версии 15,3 и более поздних версий (доступно в `/std:c++17`): `register` ключевое слово больше не является поддерживаемым классом хранения. Ключевое слово по-прежнему зарезервировано в стандарте для будущего использования.

```
register int val; // warning C5033: 'register' is no longer a supported storage class
```

Пример: автоматическая и статическая инициализация

Локальные автоматически создаваемые объекты или переменные инициализируются каждый раз, когда поток элемента управления достигает их определения. Локальные статические объекты или переменные инициализируются, когда поток элемента управления достигает их определения в первый раз.

Рассмотрим следующий пример, в котором определяется класс, который регистрирует инициализацию и

удаление объектов, а затем определяет три объекта: **I1**, **I2** и **I3**.

```
// initialization_of_objects.cpp
// compile with: /EHsc
#include <iostream>
#include <string.h>
using namespace std;

// Define a class that logs initializations and destructions.
class InitDemo {
public:
    InitDemo( const char *szWhat );
    ~InitDemo();

private:
    char *szObjName;
    size_t sizeofObjName;
};

// Constructor for class InitDemo
InitDemo::InitDemo( const char *szWhat ) :
    szObjName(NULL), sizeofObjName(0) {
    if ( szWhat != 0 && strlen( szWhat ) > 0 ) {
        // Allocate storage for szObjName, then copy
        // initializer szWhat into szObjName, using
        // secured CRT functions.
        sizeofObjName = strlen( szWhat ) + 1;

        szObjName = new char[ sizeofObjName ];
        strcpy_s( szObjName, sizeofObjName, szWhat );

        cout << "Initializing: " << szObjName << "\n";
    }
    else {
        szObjName = 0;
    }
}

// Destructor for InitDemo
InitDemo::~InitDemo() {
    if( szObjName != 0 ) {
        cout << "Destroying: " << szObjName << "\n";
        delete szObjName;
    }
}

// Enter main function
int main() {
    InitDemo I1( "Auto I1" );
    cout << "In block.\n";
    InitDemo I2( "Auto I2" );
    static InitDemo I3( "Static I3" );
}
cout << "Exited block.\n";
}
```

```
Initializing: Auto I1
In block.
Initializing: Auto I2
Initializing: Static I3
Destroying: Auto I2
Exited block.
Destroying: Auto I1
Destroying: Static I3
```

В этом примере показано, как и когда `i1` `i2` инициализируются объекты, и, а также `i3` когда они уничтожаются.

Существует несколько моментов, которые необходимо учитывать в программе:

- Во-первых, `i1` и `i2` автоматически удаляются, когда поток элемента управления выходит за пределы блока, в котором они определены.
- Во-вторых, в C++ не обязательно объявлять объекты или переменные в начале блока. Более того, эти объекты инициализируются, только если поток элемента управления достигает их определения. (`i2` и `i3` являются примерами таких определений.) Выходные данные показываются точно при инициализации.
- Наконец, статические локальные переменные, например `i3`, сохраняют свои значения длительности программы, но удаляются при завершении работы программы.

См. также

[Объявления и определения](#)

auto (C++)

12.11.2021 • 6 minutes to read

Выводит тип объявленной переменной из выражения инициализации.

NOTE

Стандарт C++ определяет исходное и измененное значение для этого ключевого слова. до Visual Studio 2010, `auto` ключевое слово объявляет переменную в *автоматическом* классе хранения, то есть переменную с локальным временем существования. начиная с Visual Studio 2010, `auto` ключевое слово объявляет переменную, тип которой выведен из выражения инициализации в его объявлении. Параметр компилятора `/zc:auto [-]` определяет значение `auto` ключевого слова.

Синтаксис

`auto инициализатор декларатора ;`

`[](auto param1 , auto Param2) {};`

Remarks

`auto` Ключевое слово направляет компилятору использование выражения инициализации объявленной переменной или параметра лямбда-выражения, чтобы вывести его тип.

Рекомендуется использовать `auto` ключевое слово для большинства ситуаций, если только не требуется преобразование, так как оно предоставляет следующие преимущества:

- **Надежность:** Если тип выражения изменился — это включает в себя изменение типа возвращаемого значения функции — это просто работает.
- **Производительность:** Вы гарантируете, что преобразование не будет выполнено.
- **Удобство использования:** Не нужно беспокоиться о неправильном написании имени типа и опечатке.
- **Эффективность:** Написание кода может быть более эффективным.

Варианты преобразования, в которых может не потребоваться использовать `auto` :

- Если необходимо получить конкретный тип.
- Вспомогательные типы шаблона выражения, например `(valarray+valarray)` .

Чтобы использовать `auto` ключевое слово, используйте его вместо типа для объявления переменной и укажите выражение инициализации. Кроме того, можно изменить `auto` ключевое слово с помощью описателей и деклараторов, таких как `const` , `volatile` , указателя (`*`), ссылки (`&`) и ссылки `rvalue` (`&&`). Компилятор вычисляет выражение инициализации, а затем использует эти сведения, чтобы вывести тип переменной.

`auto` Выражение инициализации может принимать несколько форм:

- Синтаксис универсальной инициализации, например `auto a { 42 };` .

- Синтаксис назначения, например `auto b = 0;`.
- Синтаксис универсального назначения, который объединяет две предыдущие формы, такие как `auto c = { 3.14156 };`.
- Прямая инициализация или синтаксис в стиле конструктора, например `auto d(1.41421f);`.

Дополнительные сведения см. в разделе [инициализаторы](#) и примеры кода далее в этом документе.

Если `auto` используется для объявления параметра Loop в операторе на основе диапазона `for`, используется другой синтаксис инициализации, например `for (auto& i : iterable) do_action(i);`. Дополнительные сведения см. в разделе [инструкция на основе диапазона](#) `for` (C++) .

`auto` Ключевое слово является заполнителем для типа, но не является типом. Поэтому `auto` ключевое слово не может использоваться в приведениях или операторах, таких как `sizeof` и (для C++/CLI) `typeid`

Удобство

`auto` Ключевое слово — это простой способ объявления переменной, имеющей сложный тип. Например, можно использовать `auto` для объявления переменной, в которой выражение инициализации включает шаблоны, указатели на функции или указатели на члены.

Можно также использовать `auto` для объявления и инициализации переменной в лямбда-выражении. Вы не сможете самостоятельно объявить тип переменной, поскольку тип лямбда-выражения известен только компилятору. Дополнительные сведения см. в разделе [Примеры лямбда-выражений](#).

Отслеживание возвращаемых типов

`auto decltype` Для создания библиотек шаблонов можно использовать вместе с описателем типа. Используйте `auto` и `decltype` для объявления функции шаблона, тип возвращаемого значения которого зависит от типов своих аргументов шаблона. Или используйте `auto` и `decltype` для объявления функции-шаблона, которая создает оболочку для вызова другой функции, а затем возвращает все, что является типом возвращаемого значения этой функции. Дополнительные сведения см. на веб-сайте [decltype](#).

Ссылки и cv-квалификаторы

Обратите внимание, что использование удаления `auto` ссылок, `const` квалификаторов и `volatile` квалификаторов. Рассмотрим следующий пример.

```
// cl.exe /analyze /EHsc /W4
#include <iostream>

using namespace std;

int main( )
{
    int count = 10;
    int& countRef = count;
    auto myAuto = countRef;

    countRef = 11;
    cout << count << " ";

    myAuto = 12;
    cout << count << endl;
}
```

В предыдущем примере Мяуто — это, а `int` не `int` ссылка, поэтому выходные данные не так, [11 11](#) [11 12](#) как будто квалификатор ссылки не был удален с помощью `auto`.

Выведение типа с инициализаторами в фигурных скобках (C++ 14)

В следующем примере кода показано, как инициализировать `auto` переменную с помощью фигурных скобок. Обратите внимание на разницу между В и С и между А и Е.

```
#include <initializer_list>

int main()
{
    // std::initializer_list<int>
    auto A = { 1, 2 };

    // std::initializer_list<int>
    auto B = { 3 };

    // int
    auto C{ 4 };

    // C3535: cannot deduce type for 'auto' from initializer list'
    auto D = { 5, 6.7 };

    // C3518 in a direct-list-initialization context the type for 'auto'
    // can only be deduced from a single initializer expression
    auto E{ 8, 9 };

    return 0;
}
```

Ограничения и сообщения об ошибках

В следующей таблице перечислены ограничения на использование `auto` ключевого слова и соответствующее сообщение об ошибке диагностики, выдаваемое компилятором.

Номер ошибки	Описание
C3530	<code>auto</code> Ключевое слово не может использоваться вместе с любым другим описателем типа.
C3531	Символ, объявленный с <code>auto</code> ключевым словом, должен иметь инициализатор.
C3532	Вы неправильно использовали <code>auto</code> ключевое слово для объявления типа. Например, был объявлен тип возвращаемого значения метода или массив.
C3533, C3539	Параметр или аргумент шаблона не может быть объявлен с <code>auto</code> ключевым словом.
C3535	Невозможно объявить метод или параметр шаблона с <code>auto</code> ключевым словом.

НОМЕР ОШИБКИ	ОПИСАНИЕ
C3536	Символ не может быть использован до инициализации. Практически это означает, что переменную нельзя использовать для инициализации самой себя.
C3537	Нельзя привести к типу, объявленному с <code>auto</code> ключевым словом.
C3538	Все символы в списке деклараторов, объявлена с <code>auto</code> ключевым словом, должны разрешаться в один и тот же тип. Дополнительные сведения см. в разделе объявления и определения .
C3540, C3541	Операторы <code>sizeof</code> и <code>typeid</code> не могут применяться к символам, объявленным с <code>auto</code> ключевым словом.

Примеры

Эти фрагменты кода иллюстрируют некоторые способы `auto` использования ключевого слова.

Следующие объявления эквивалентны. В первом операторе переменная `j` объявлена как тип `int`. Во второй инструкции переменная `k` выведена как тип, `int` поскольку выражение инициализации (0) является целым числом.

```
int j = 0; // Variable j is explicitly type int.
auto k = 0; // Variable k is implicitly type int because 0 is an integer.
```

Следующие объявления эквивалентны, но второе объявление проще первого. Одной из наиболее интересных причин использования `auto` ключевого слова является простота.

```
map<int,list<string>>::iterator i = m.begin();
auto i = m.begin();
```

В следующем фрагменте кода объявляется тип переменных `iter` И `elem` при `for` `for` запуске циклов и диапазонов.

```

// cl /EHsc /nologo /W4
#include <deque>
using namespace std;

int main()
{
    deque<double> dqDoubleData(10, 0.1);

    for (auto iter = dqDoubleData.begin(); iter != dqDoubleData.end(); ++iter)
    { /* ... */ }

    // prefer range-for loops with the following information in mind
    // (this applies to any range-for with auto, not just deque)

    for (auto elem : dqDoubleData) // COPIES elements, not much better than the previous examples
    { /* ... */ }

    for (auto& elem : dqDoubleData) // observes and/or modifies elements IN-PLACE
    { /* ... */ }

    for (const auto& elem : dqDoubleData) // observes elements IN-PLACE
    { /* ... */ }
}

```

В следующем фрагменте кода используется `new` объявление оператора и указателя для объявления указателей.

```

double x = 12.34;
auto *y = new auto(x), **z = new auto(&x);

```

В следующем примере кода объявлено несколько символов в каждом операторе объявления. Обратите внимание, что все символы во всех операторах разрешаются к одному и тому же типу.

```

auto x = 1, *y = &x, **z = &y; // Resolves to int.
auto a(2.01), *b (&a);        // Resolves to double.
auto c = 'a', *d(&c);         // Resolves to char.
auto m = 1, &n = m;           // Resolves to int.

```

В этом примере кода используется условный оператор (`? :`). Переменная `x` здесь объявляется как целочисленная переменная со значением 200.

```

int v1 = 100, v2 = 200;
auto x = v1 > v2 ? v1 : v2;

```

Следующий фрагмент кода инициализирует переменную `x` для типа `int`, переменную `y` на ссылку на тип `const int`, а переменная — `fp` на указатель на функцию, которая возвращает тип `int`.

```

int f(int x) { return x; }
int main()
{
    auto x = f(0);
    const auto& y = f(1);
    int (*p)(int x);
    p = f;
    auto fp = p;
    //...
}

```

См. также

Ключевые слова

`/Zc:auto` (Вывод типа переменной)

`sizeof` Оператор

`typeid`

`operator new`

Объявления и определения

Примеры лямбда-выражений

Инициализаторы

`decltype`

const (C++)

12.11.2021 • 3 minutes to read

При изменении объявления данных `const` ключевое слово указывает, что объект или переменная не являются изменяемыми.

Синтаксис

```
const declaration ;
member-function const ;
```

Значения-константы

`const` Ключевое слово указывает, что значение переменной является константой и сообщает компилятору о том, что программист не сможет его изменить.

```
// constant_values1.cpp
int main() {
    const int i = 5;
    i = 10;    // C3892
    i++;      // C2105
}
```

В C++ `const` вместо директивы препроцессора `#define` можно использовать ключевое слово, чтобы определить постоянные значения. Значения, определенные с помощью `const` подчиняются проверке типа и могут использоваться вместо константных выражений. В C++ можно указать размер массива с помощью `const` переменной следующим образом:

```
// constant_values2.cpp
// compile with: /c
const int maxarray = 255;
char store_char[maxarray]; // allowed in C++; not allowed in C
```

В языке С константные значения по умолчанию имеют внешнюю компоновку, поэтому они могут использоваться только в файлах исходного кода. В языке C++ константные значения по умолчанию имеют внутреннюю компоновку, которая позволяет использовать их в файлах заголовков.

`const` Ключевое слово также можно использовать в объявлениях указателей.

```
// constant_values3.cpp
int main() {
    char *mybuf = 0, *yourbuf;
    char *const aptr = mybuf;
    *aptr = 'a';    // OK
    aptr = yourbuf; // C3892
}
```

Указатель на переменную, объявленную как `const`, может быть назначен только указателю, который также объявлен как `const`.

```
// constant_values4.cpp
#include <stdio.h>
int main() {
    const char *mybuf = "test";
    char *yourbuf = "test2";
    printf_s("%s\n", mybuf);

    const char *bptr = mybuf; // Pointer to constant data
    printf_s("%s\n", bptr);

    // *bptr = 'a'; // Error
}
```

Указатели на данные-константы можно использовать в качестве параметров функций, чтобы функция не могла изменять параметр, переданный посредством указателя.

Для объектов, объявленных как `const`, можно вызывать только константные функции членов. Это гарантирует, что константный объект не будет изменен.

```
birthday.getMonth(); // Okay
birthday.setMonth( 4 ); // Error
```

Для объектов, не объявленных как константы, можно вызывать как константные, так и неконстантные функции-члены. Можно также перегружать функцию-член с помощью `const` ключевого слова; это позволяет вызывать другую версию функции для постоянных и неконстантных объектов.

Нельзя объявлять конструкторы или деструкторы с `const` ключевым словом.

ФУНКЦИИ-ЧЛЕНЫ-КОНСТАНТЫ

Объявление функции-члена с помощью `const` ключевого слова указывает, что функция является функцией "только для чтения", которая не изменяет объект, для которого она вызывается. Функция-член константы не может изменять какие-либо нестатические элементы данных или вызывать функции-члены, не являющиеся константами. Чтобы объявить функцию-член константы, поместите `const` ключевое слово после закрывающей скобки списка аргументов. `const` Ключевое слово требуется как в объявлении, так и в определении.

```

// constant_member_function.cpp
class Date
{
public:
    Date( int mn, int dy, int yr );
    int getMonth() const;      // A read-only function
    void setMonth( int mn );   // A write function; can't be const
private:
    int month;
};

int Date::getMonth() const
{
    return month;           // Doesn't modify anything
}
void Date::setMonth( int mn )
{
    month = mn;            // Modifies data member
}
int main()
{
    Date MyDate( 7, 4, 1998 );
    const Date BirthDate( 1, 18, 1953 );
    MyDate.setMonth( 4 );    // Okay
    BirthDate.getMonth();   // Okay
    BirthDate.setMonth( 4 ); // C2662 Error
}

```

Различия констант С и С++

При объявлении переменной, как `const` в файле исходного кода на языке С, это можно сделать следующим образом:

```
const int i = 2;
```

Затем эту переменную можно использовать в другом модуле следующим образом:

```
extern const int i;
```

Но чтобы получить такое же поведение в С++, необходимо объявить `const` переменную как:

```
extern const int i = 2;
```

Если вы хотите объявить `extern` переменную в файле исходного кода С++ для использования в файле исходного кода на языке С, используйте:

```
extern "C" const int x=10;
```

для предотвращения изменения имени компилятором С++.

Remarks

При использовании списка параметров функции-члена `const` ключевое слово указывает, что функция не изменяет объект, для которого она вызывается.

Дополнительные сведения о `const` см. в следующих разделах:

- [константные и переменные указатели](#)
- [Квалификаторы типов \(Справочник по языку C\)](#)
- [volatile](#)
- [#define](#)

См. также

[Ключевые слова](#)

constexpr (C++)

12.11.2021 • 5 minutes to read

Ключевое слово `constexpr` было введено в C++ 11 и улучшено в C++ 14. Это означает *const Ant выражение*. Например `const`, его можно применять к переменным: Ошибка компилятора возникает, когда любой код пытается пересчитать if значение у. В отличие от `const`, `constexpr` можно также применять к функциям и классу `const` рукторс. `constexpr` Указывает, что значение (или возвращаемое значение) является `const` Ant и, где возможно, вычислено во время компиляции.

`constexpr` Целочисленное значение можно использовать везде, где `const` требуется целое число, например в аргументах шаблона и в объявлениях массивов. Если значение вычислено во время компиляции, а не во время выполнения, оно помогает программе работать быстрее и использует меньше памяти.

Для ограничения сложности `const` вычислений Ant времени компиляции и их возможного влияния на время компиляции стандарт C++ 14 требует, чтобы типы в `const` выражениях Ant были [литеральными типами](#).

Синтаксис

```
** constexpr ** тип литерала Ident if Иер = const Ant-Expression;
** constexpr ** тип литерала Ident if Иер{ const Ant-Expression } ;
** constexpr ** тип литерала Ident if Иер( params );
** constexpr ** ctor( params );
```

Параметры

`params`

Один или несколько параметров, каждый из которых должен быть типом литерала и сам должен быть `const` выражением Ant.

Возвращаемое значение

`constexpr` Переменная или функция должна возвращать [литеральный тип](#).

Переменные `constexpr`

Основной вывод между `const` `constexpr` переменными и заключается в том, что инициализация `const` переменной может быть отложена до времени выполнения. `constexpr` Переменная должна быть инициализирована во время компиляции. Все `constexpr` переменные имеют значения `const`.

- Переменная может быть объявлена с `constexpr`, если она имеет литеральный тип и инициализирована. Если инициализация задается для MED с помощью `const` руктор, `const` руктор должен быть объявлен как `constexpr`.
- Ссылка может быть объявлена как `constexpr` при выполнении обоих этих условий: объект, на который указывает ссылка, инициализируется `const` выражением Ant, а любые неявные преобразования, вызванные во время инициализации, также `const` Ant выражениями.
- Все объявления `constexpr` переменной или функции должны иметь `constexpr` спецификацию if

Иер.

```
constexpr float x = 42.0;
constexpr float y{108};
constexpr float z = exp(5, 3);
constexpr int i; // Error! Not initialized
int j = 0;
constexpr int k = j + 1; //Error! j not a constant expression
```

ФУНКЦИИ constexpr

`constexpr` Функция — это одна из функций, возвращаемое значение которой вычисляемым во время компиляции, если это требуется для использования кода. Для использования кода требуется возвращаемое значение во время компиляции для инициализации `constexpr` переменной или для предоставления аргумента шаблона, не являющегося типом. Если его аргументы являются `constexpr` значениями, `constexpr` функция создает Ant времени компиляции `const`. При вызове с `constexpr` аргументами, отличными от аргументов, или когда его значение не требуется во время компиляции, оно создает значение во время выполнения, например обычную функцию. (Это двойное поведение избавляет от необходимости писать `constexpr` и не использовать `constexpr` версии одной и той же функции.)

`constexpr` Функция или `const` руктор являются неявными `inline`.

К функциям применяются следующие правила `constexpr`.

- `constexpr` Функция должна принимать и возвращать только [типы литералов](#).
- `constexpr` Функция может быть рекурсивной.
- Он не может быть [виртуальным](#). constРуктор не может быть определен, как `constexpr`. Если включающий класс имеет какие-либо виртуальные базовые классы.
- Тело может быть определено как `= default` ИЛИ `= delete`.
- Текст не может содержать `goto` операторы или `try` блоки.
- Явная специализация не `constexpr` шаблона может быть объявлена следующим образом `constexpr`:
- Явная специализация `constexpr` шаблона также не должна быть такой `constexpr`:

к `constexpr` функциям в Visual Studio 2017 и более поздних версий применяются следующие правила.

- Он может содержать `if` `switch` инструкции и, а также все операторы цикла, включая `for`, основанные на диапазонах `for`, `while` и `Do- while`.
- Он может содержать объявления локальных переменных, но переменная должна быть инициализирована. Он должен быть типом литерала и не может быть `static` или локальным потоком. Локально объявленная переменная не обязательно должна быть `const` и может изменяться.
- Функция, не относящаяся `constexpr static` к члену, не обязательно должна быть неявно `const`.

```
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
}
```

TIP

в отладчике Visual Studio при отладке неоптимизированной отладочной сборки можно определить, `constexpr` вычисляется ли функция во время компиляции, поместив в нее точку останова. Попадание в точку останова означает, что функция была вызвана во время выполнения. Если попадания в точку останова не происходит, это означает, что функция была вызвана во время компиляции.

Название `constexpr`

Параметр компилятора `/Zc: externConstexpr` заставляет компилятор применить **внешнюю компоновку** к переменным, объявленным с **помощью `constexpr` модификатора `extern`**. в более ранних версиях Visual Studio либо по умолчанию, либо при использовании параметра `/zc: externConstexpr- spec if` иед, Visual Studio применяет внутреннюю компоновку к `constexpr` переменным, даже если `extern` используется ключевое слово. параметр `/zc: externConstexpr` доступен начиная с обновления Visual Studio 2017 15,6 и по умолчанию отключен. Параметр `/permissive-` не включает `/Zc: externConstexpr`.

Пример

В следующем примере показаны `constexpr` переменные, функции и определяемый пользователем тип. В последней инструкции в `main()` `constexpr` функция `GetValue()` -член является вызовом во время выполнения, поскольку значение не обязательно должно быть известно во время компиляции.

```

// constexpr.cpp
// Compile with: cl /EHsc /W4 constexpr.cpp
#include <iostream>

using namespace std;

// Pass by value
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
}

// Pass by reference
constexpr float exp2(const float& x, const int& n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp2(x * x, n / 2) :
        exp2(x * x, (n - 1) / 2) * x;
}

// Compile-time computation of array length
template<typename T, int N>
constexpr int length(const T(&)[N])
{
    return N;
}

// Recursive constexpr function
constexpr int fac(int n)
{
    return n == 1 ? 1 : n * fac(n - 1);
}

// User-defined type
class Foo
{
public:
    constexpr explicit Foo(int i) : _i(i) {}
    constexpr int GetValue() const
    {
        return _i;
    }
private:
    int _i;
};

int main()
{
    // foo is const:
    constexpr Foo foo(5);
    // foo = Foo(6); //Error!

    // Compile time:
    constexpr float x = exp(5, 3);
    constexpr float y { exp(2, 5) };
    constexpr int val = foo.GetValue();
    constexpr int f5 = fac(5);
    const int nums[] { 1, 2, 3, 4 };
    const int nums2[length(nums) * 2] { 1, 2, 3, 4, 5, 6, 7, 8 };

    // Run time:
    cout << "The value of foo is " << foo.GetValue() << endl;
}

```

Требования

Visual Studio 2015 или более поздней версии.

См. также

[Объявления и определения](#)

[const](#)

extern (C++)

12.11.2021 • 3 minutes to read

`extern` Ключевое слово может быть применено к глобальной переменной, функции или объявлению шаблона. Он указывает, что символ имеет *extern связь A!* Дополнительные сведения о компоновке и о том, почему не рекомендуется использовать глобальные переменные, см. в разделе [Преобразование единиц и компоновки](#).

`extern` Ключевое слово имеет четыре значения в зависимости от контекста:

- В `const` объявлении неглобальной переменной `extern` указывает, что переменная или функция определена в другой записи преобразования. `extern` Должен применяться ко всем файлам, за исключением того, где определена переменная.
- В `const` объявлении переменной указывается, что переменная имеет *extern связь A!*. `extern` Должен применяться ко всем объявлениям во всех файлах. (`const` По умолчанию глобальные переменные имеют внутреннюю компоновку.)
- `extern "C"` указывает, что функция определена в других местах и использует соглашение о вызовах на языке C. `extern` Модификатор "C" может также применяться к нескольким объявлениям функций в блоке.
- В объявлении шаблона `extern` указывает, что шаблон уже создан в других местах. `extern` сообщает компилятору, что он может повторно использовать другое создание экземпляра, вместо того чтобы создавать новый экземпляр в текущем расположении. Дополнительные сведения об этом использовании см `extern` . в разделе [явное создание экземпляра](#).

extern компоновка для `const` неглобальных элементов

Если компоновщик видит `extern` перед объявлением глобальной переменной, он ищет определение в другой записи преобразования. Объявления `const` непеременных в глобальной области `extern` по умолчанию имеют значение *A!*. Применяется только `extern` к объявлениям, не предоставляемым определение.

```
//fileA.cpp
int i = 42; // declaration and definition

//fileB.cpp
extern int i; // declaration only. same as i in FileA

//fileC.cpp
extern int i; // declaration only. same as i in FileA

//fileD.cpp
int i = 43; // LNK2005! 'i' already has a definition.
extern int i = 43; // same error (extern is ignored on definitions)
```

extern компоновка для `const` глобальных элементов

`const` По умолчанию глобальная переменная имеет внутреннюю компоновку. Если требуется, чтобы переменная соимела *extern связь A!*, примените `extern` к определению ключевое слово и ко всем остальным объявлениям в других файлах:

```
//fileA.cpp
extern const int i = 42; // extern const definition

//fileB.cpp
extern const int i; // declaration only. same as i in FileA
```

extern Компоновка constexpr

в Visual Studio 2017 версии 15,3 и более ранних версиях компилятор всегда присвоил `constexpr` переменную внутреннюю компоновку, даже если переменная была помечена `extern`. в Visual Studio 2017 версии 15,5 и более поздних параметр компилятора `/zc: extern Constexpr` обеспечивает правильное поведение соответствия стандартам. В конечном итоге параметр будет иметь значение по умолчанию. `/permissive`-Параметр не включает `/Zc: extern constexpr`.

```
extern constexpr int x = 10; //error LNK2005: "int const x" already defined
```

Если файл заголовка содержит объявленную переменную `extern constexpr`, она должна быть помечена `__declspec(selectany)` для правильного объединения его повторяющихся объявлений:

```
extern constexpr __declspec(selectany) int x = 10;
```

extern Объявления функций С и extern C++

В C++ при использовании со строкой `extern` указывает, что для деклараторов используются соглашения компоновки другого языка. Доступ к функциям и данным С можно получить только в том случае, если они были ранее объявлены с компоновкой С. Однако они должны быть определены в блоке трансляции, который компилируется отдельно.

Microsoft C++ поддерживает строки "C" и "C++" в поле "строковый литерал". Все стандартные включаемые файлы используют синтаксис `extern "C"`, что позволяет использовать функции библиотеки времени выполнения в программах на C++.

Пример

В следующем примере показано, как объявить имена с компоновкой С:

```

// Declare printf with C linkage.
extern "C" int printf(const char *fmt, ...);

// Cause everything in the specified
// header files to have C linkage.
extern "C" {
    // add your #include statements here
#include <stdio.h>
}

// Declare the two functions ShowChar
// and GetChar with C linkage.
extern "C" {
    char ShowChar(char ch);
    char GetChar(void);
}

// Define the two functions
// ShowChar and GetChar with C linkage.
extern "C" char ShowChar(char ch) {
    putchar(ch);
    return ch;
}

extern "C" char GetChar(void) {
    char ch;
    ch = getchar();
    return ch;
}

// Declare a global variable, errno, with C linkage.
extern "C" int errno;

```

Если у функции несколько спецификаций компоновки, они должны быть согласованы. Объявление функций как с помощью компоновки C, так и C++ является ошибкой. Кроме того, если в программе имеются два объявления функции (одно со спецификацией компоновки, а другое без такой спецификации), объявление с указанием компоновки должен быть первым. Ко всем повторным объявлениям функций, уже имеющих спецификацию компоновки, применяется компоновка из первого объявления. Пример:

```

extern "C" int CFunc1();
...
int CFunc1();           // Redefinition is benign; C linkage is
                       // retained.

int CFunc2();
...
extern "C" int CFunc2(); // Error: not the first declaration of
                       // CFunc2;  cannot contain linkage
                       // specifier.

```

См. также раздел

[Словами](#)

[Единицы и компоновка преобразований](#)

[extern Описатель Storage-Class в C](#)

[Поведение идентификаторов в C](#)

[Компоновка в C](#)

Инициализаторы

12.11.2021 • 11 minutes to read

Инициализатор определяет начальное значение переменной. Можно инициализировать переменные в этих контекстах:

- В определении переменной:

```
int i = 3;  
Point p1{ 1, 2 };
```

- В качестве одного из параметров функции:

```
set_point(Point{ 5, 6 });
```

- В качестве возвращаемого типа функции:

```
Point get_new_point(int x, int y) { return { x, y }; }  
Point get_new_point(int x, int y) { return Point{ x, y }; }
```

Инициализаторы могут принимать эти формы:

- Выражение (или разделенный запятыми список выражений) в скобках:

```
Point p1(1, 2);
```

- Знак равенства с последующим выражением:

```
string s = "hello";
```

- Список инициализации в фигурных скобках. Список может быть пустым или может состоять из набора списков как в приведенном ниже примере.

```
struct Point{  
    int x;  
    int y;  
};  
class PointConsumer{  
public:  
    void set_point(Point p){};  
    void set_points(initializer_list<Point> my_list){};  
};  
int main() {  
    PointConsumer pc{};  
    pc.set_point({});  
    pc.set_point({ 3, 4 });  
    pc.set_points({ { 3, 4 }, { 5, 6 } });  
}
```

Типы инициализации

Существует несколько типов инициализации, которые могут встречаться на различных этапах выполнения программы. Различные типы инициализации не являются взаимоисключающими, например, инициализация списка может активировать инициализацию значений, а в других условиях она может активировать агрегатную инициализацию.

Нулевая инициализация

Нулевая инициализация — задание для переменной нулевого значения, неявно преобразованного в тип:

- Числовые переменные инициализируются значением 0 (или 0,0; 0,0000000000 и т.п.).
- Переменные `char` инициализируются значением `'\0'`.
- Указатели инициализируются значением `nullptr`.
- Члены массивов, классов `Pod`, структур и объединений имеют нулевое значение.

Нулевая инициализация выполняется в разное время:

- При запуске программы — для всех именованных переменных, имеющих статическую длительность. Далее эти переменные могут быть инициализированы повторно.
- Во время инициализации значений — для скалярных типов и типов класса POD, которые инициализируются с помощью пустых фигурных скобок.
- Для массивов, у которых инициализировано только подмножество членов.

Ниже приведены некоторые примеры нулевой инициализации:

```
struct my_struct{
    int i;
    char c;
};

int i0;           // zero-initialized to 0
int main() {
    static float f1; // zero-initialized to 0.00000000
    double d{};     // zero-initialized to 0.0000000000000000
    int* ptr{};     // initialized to nullptr
    char s_array[3]{'a', 'b'}; // the third char is initialized to '\0'
    int int_array[5] = { 8, 9, 10 }; // the fourth and fifth ints are initialized to 0
    my_struct a_struct{}; // i = 0, c = '\0'
}
```

Инициализация по умолчанию

Инициализация по умолчанию для классов, структур и объединений — это инициализация с помощью конструктора по умолчанию. Конструктор по умолчанию можно вызвать без выражения инициализации или с помощью `new` ключевого слова:

```
MyClass mc1;
MyClass* mc3 = new MyClass;
```

Если класс, структура или объединение не имеет конструктора по умолчанию, компилятор выдает ошибку.

Скалярные переменные инициализируются по умолчанию, если при их определении не указываются выражения инициализации. Они имеют неопределенные значения.

```
int i1;
float f;
char c;
```

Массивы инициализируются по умолчанию, если при их определении не указываются выражения инициализации. Если массив инициализируется по умолчанию, его члены инициализируются по умолчанию и приобретают неопределенные значения как в приведенном ниже примере.

```
int int_arr[3];
```

Если члены массива не имеют конструктор по умолчанию, компилятор выдает ошибку.

Инициализация по умолчанию константных переменных

Константные переменные необходимо объявлять вместе с инициализатором. Если они относятся к скалярным типам, они вызывают ошибку компилятора, а если они относятся к типам классов, имеющим конструктор по умолчанию, они вызывают предупреждение:

```
class MyClass{};
int main() {
    //const int i2;    // compiler error C2734: const object must be initialized if not extern
    //const char c2;  // same error
    const MyClass mc1; // compiler error C4269: 'const automatic data initialized with compiler generated
default constructor produces unreliable results
}
```

Инициализация по умолчанию статических переменных

Статические переменные, объявленные без инициализатора, инициализируются значением 0 (с неявным преобразованием к соответствующему типу).

```
class MyClass {
private:
    int m_int;
    char m_char;
};

int main() {
    static int int1;      // 0
    static char char1;   // '\0'
    static bool bool1;   // false
    static MyClass mc1;  // {0, '\0'}
}
```

Дополнительные сведения об инициализации глобальных статических объектов см. в разделе [функция Main и аргументы командной строки](#).

Инициализация значения

Инициализация значения происходит в следующих случаях:

- Именованное значение инициализируется с использованием пустых фигурных скобок.
- Анонимный временный объект инициализируется с помощью пустых круглых или фигурных скобок.
- Объект инициализируется с помощью `new` ключевого слова, а также пустых круглых скобок или фигурных скобок

При инициализации значения выполняются следующие действия:

- Для классов, имеющих хотя бы один открытый конструктор, вызывается конструктор по умолчанию.
- В случае классов, не относящихся к объединениям, у которых нет объявленных конструкторов, объект инициализируется нулевым значением, и вызывается конструктор по умолчанию.
- В случае массивов каждый элемент инициализируется значением.
- Во всех остальных случаях переменная инициализируется нулевым значением.

```
class BaseClass {  
private:  
    int m_int;  
};  
  
int main() {  
    BaseClass bc{};      // class is initialized  
    BaseClass* bc2 = new BaseClass(); // class is initialized, m_int value is 0  
    int int_arr[3]{};   // value of all members is 0  
    int a{};           // value of a is 0  
    double b{};         // value of b is 0.0000000000000000  
}
```

Инициализация копированием

Инициализация копированием — это инициализация одного объекта с использованием другого объекта. Она выполняется в следующих случаях:

- Переменная инициализируется с помощью знака равенства.
- Аргумент передается в функцию.
- Объект возвращается функцией.
- Возникает или перехватывается исключение.
- Нестатический элемент данных инициализируется с помощью знака равенства.
- Класс, структура и члены объединения инициализируются с применением инициализации путем копирования во время агрегатной инициализации. Примеры см. в разделе [агрегатная инициализация](#).

Следующий код демонстрирует несколько примеров инициализации копированием.

```

#include <iostream>
using namespace std;

class MyClass{
public:
    MyClass(int myInt) {}
    void set_int(int myInt) { m_int = myInt; }
    int get_int() const { return m_int; }
private:
    int m_int = 7; // copy initialization of m_int
};

class MyException : public exception{};

int main() {
    int i = 5;           // copy initialization of i
    MyClass mc1{ i };
    MyClass mc2 = mc1;   // copy initialization of mc2 from mc1
    MyClass mc1.set_int(i); // copy initialization of parameter from i
    int i2 = mc2.get_int(); // copy initialization of i2 from return value of get_int()

    try{
        throw MyException();
    }
    catch (MyException ex){ // copy initialization of ex
        cout << ex.what();
    }
}

```

Инициализация копированием не может вызывать явные конструкторы.

```

vector<int> v = 10; // the constructor is explicit; compiler error C2440: cannot convert from 'int' to
'std::vector<int, std::allocator<_Ty>>'
regex r = "a.*b"; // the constructor is explicit; same error
shared_ptr<int> sp = new int(1729); // the constructor is explicit; same error

```

В некоторых случаях, если конструктор копии класса удален или недоступен, копируемая инициализация вызывает ошибку компилятора.

Прямая инициализация

Прямая инициализация — это инициализация с использованием (непустых) круглых или фигурных скобок. В отличие от копируемой инициализации она может вызывать явные конструкторы. Она выполняется в следующих случаях:

- Переменная инициализируется с помощью непустых круглых или фигурных скобок.
- переменная инициализируется с помощью `new` ключевого слова, а также непустых фигурных скобок или круглых скобок
- переменная инициализируется с помощью `static_cast`
- В конструкторе базовые классы и нестатические члены инициализируются с помощью списка инициализации.
- В копии захваченной переменной в лямбда-выражении.

Приведенный ниже код демонстрирует несколько примеров прямой инициализации.

```
class BaseClass{
public:
    BaseClass(int n) :m_int(n){} // m_int is direct initialized
private:
    int m_int;
};

class DerivedClass : public BaseClass{
public:
    // BaseClass and m_char are direct initialized
    DerivedClass(int n, char c) : BaseClass(n), m_char(c) {}
private:
    char m_char;
};

int main(){
    BaseClass bc1(5);
    DerivedClass dc1{ 1, 'c' };
    BaseClass* bc2 = new BaseClass(7);
    BaseClass bc3 = static_cast<BaseClass>(dc1);

    int a = 1;
    function<int()> func = [a](){ return a + 1; }; // a is direct initialized
    int n = func();
}
```

Инициализация списком

Инициализация списком выполняется, когда переменная инициализируется с помощью списка инициализации в фигурных скобках. Списки инициализации в фигурных скобках можно использовать в следующих случаях:

- Инициализируется переменная.
- класс инициализируется с помощью `new` ключевого слова
- Объект возвращается функцией.
- Аргумент передается функции.
- Один из аргументов при прямой инициализации.
- В инициализаторе нестатических элементов данных.
- В списке инициализации конструктора.

Приведенный ниже код демонстрирует несколько примеров инициализации списком.

```

class MyClass {
public:
    MyClass(int myInt, char myChar) {}
private:
    int m_int[]{ 3 };
    char m_char;
};

class MyClassConsumer{
public:
    void set_class(MyClass c) {}
    MyClass get_class() { return MyClass{ 0, '\0' }; }
};

struct MyStruct{
    int my_int;
    char my_char;
    MyClass my_class;
};

int main() {
    MyClass mc1{ 1, 'a' };
    MyClass* mc2 = new MyClass{ 2, 'b' };
    MyClass mc3 = { 3, 'c' };

    MyClassConsumer mcc;
    mcc.set_class(MyClass{ 3, 'c' });
    mcc.set_class({ 4, 'd' });

    MyStruct ms1{ 1, 'a', { 2, 'b' } };
}

```

Агрегатная инициализация

Агрегатная инициализация — форма инициализации списка для массивов и типов классов (часто структур и объединений), со следующими характеристиками:

- Отсутствие закрытых или защищенных членов.
- Отсутствие заданных пользователем конструкторов кроме явно заданных по умолчанию или удаленных конструкторов.
- Отсутствие базовых классов.
- Отсутствие виртуальных функций-членов.

NOTE

в Visual Studio 2015 и более ранних версиях статистическое выражение не может содержать инициализаторы с фигурными или равными скобками для нестатических членов. это ограничение было удалено в стандарте C++ 14 и реализовано в Visual Studio 2017.

Агрегатные инициализаторы состоят из списка инициализации в фигурных скобках со знаком равенства или без него как в приведенном ниже примере:

```

#include <iostream>
using namespace std;

struct MyAggregate{
    int myInt;
    char myChar;
};

struct MyAggregate2{
    int myInt;
    char myChar = 'Z'; // member-initializer OK in C++14
};

int main() {
    MyAggregate agg1{ 1, 'c' };
    MyAggregate2 agg2{2};
    cout << "agg1: " << agg1.myChar << ":" << agg1.myInt << endl;
    cout << "agg2: " << agg2.myChar << ":" << agg2.myInt << endl;

    int myArr1[]{ 1, 2, 3, 4 };
    int myArr2[3] = { 5, 6, 7 };
    int myArr3[5] = { 8, 9, 10 };

    cout << "myArr1: ";
    for (int i : myArr1){
        cout << i << " ";
    }
    cout << endl;

    cout << "myArr3: ";
    for (auto const &i : myArr3) {
        cout << i << " ";
    }
    cout << endl;
}

```

Вы должны увидеть следующий результат.

```

agg1: c: 1
agg2: Z: 2
myArr1: 1 2 3 4
myArr3: 8 9 10 0 0

```

IMPORTANT

Элементы массива, объявляемые, но не инициализированные во время инициализации агрегата, инициализируются нулем, как показано `myArr3` выше.

Инициализация объединений и структур

Если объединение не имеет конструктора, его можно инициализировать одним значением (или другим экземпляром объединения). Значение используется для инициализации первого нестатического поля. Это отличается от инициализации структур, где первое значение в инициализаторе используется для инициализации первого поля, второе — для инициализации второго поля и т. д. Сравните инициализацию объединений и структур в следующем примере:

```

struct MyStruct {
    int myInt;
    char myChar;
};

union MyUnion {
    int my_int;
    char my_char;
    bool my_bool;
    MyStruct my_struct;
};

int main() {
    MyUnion mu1{ 'a' }; // my_int = 97, my_char = 'a', my_bool = true, {myInt = 97, myChar = '\0'}
    MyUnion mu2{ 1 }; // my_int = 1, my_char = 'x1', my_bool = true, {myInt = 1, myChar = '\0'}
    MyUnion mu3{}; // my_int = 0, my_char = '\0', my_bool = false, {myInt = 0, myChar = '\0'}
    MyUnion mu4 = mu3; // my_int = 0, my_char = '\0', my_bool = false, {myInt = 0, myChar = '\0'}
    //MyUnion mu5{ 1, 'a', true }; // compiler error: C2078: too many initializers
    //MyUnion mu6 = 'a'; // compiler error: C2440: cannot convert from 'char' to 'MyUnion'
    //MyUnion mu7 = 1; // compiler error: C2440: cannot convert from 'int' to 'MyUnion'

    MyStruct ms1{ 'a' }; // myInt = 97, myChar = '\0'
    MyStruct ms2{ 1 }; // myInt = 1, myChar = '\0'
    MyStruct ms3{}; // myInt = 0, myChar = '\0'
    MyStruct ms4{1, 'a'}; // myInt = 1, myChar = 'a'
    MyStruct ms5 = { 2, 'b' }; // myInt = 2, myChar = 'b'
}

```

Инициализация статистических выражений, содержащих статистические выражения

Агрегатные типы могут содержать другие агрегатные типы, например массивы массивов, массивы структур и т. п. Эти типы инициализируются с помощью вложенных наборов фигурных скобок, как показано в следующем примере:

```

struct MyStruct {
    int myInt;
    char myChar;
};

int main() {
    int intArr1[2][2]{{ 1, 2 }, { 3, 4 }};
    int intArr3[2][2] = {1, 2, 3, 4};
    MyStruct structArr[]{{ 1, 'a' }, { 2, 'b' }, {3, 'c'} };
}

```

Инициализация ссылок

Переменные ссылочного типа должны инициализироваться объектом типа, на котором основан ссылочный тип, или объектом типа, который можно преобразовать в такой тип. Пример:

```

// initializing_references.cpp
int iVar;
long lVar;
int main()
{
    long& LongRef1 = lVar; // No conversion required.
    long& LongRef2 = iVar; // Error C2440
    const long& LongRef3 = iVar; // OK
    LongRef1 = 23L; // Change lVar through a reference.
    LongRef2 = 11L; // Change iVar through a reference.
    LongRef3 = 11L; // Error C3892
}

```

Единственный способ инициализировать ссылку с помощью временного объекта является инициализация постоянного временного объекта. После инициализации переменная ссылочного типа

всегда указывает на один и тот же объект; ее невозможно изменить, чтобы она указывала на другой объект.

Хотя синтаксис может быть одинаковым, инициализация переменных ссылочного типа и присваивание значений переменным ссылочного типа семантически различаются. В предыдущем примере присваивания, которые изменяют значения переменных `iVar` И `lVar`, выглядят аналогично инициализации, но имеют другой эффект. Инициализация определяет объект, на который указывает переменная ссылочного типа; при присваивании через ссылку производится присваивание значения объекту, на который указывает ссылка.

Поскольку передача аргумента ссылочного типа в функцию и возврат значения ссылочного типа из функции являются инициализацией, формальные аргументы функции, а также возвращаемые ссылки инициализируются правильно.

Переменные ссылочного типа можно объявлять без инициализаторов только в указанных ниже случаях.

- Объявления функций (прототипы). Пример:

```
int func( int& );
```

- Объявления типов значений, возвращаемых функцией. Пример:

```
int& func( int& );
```

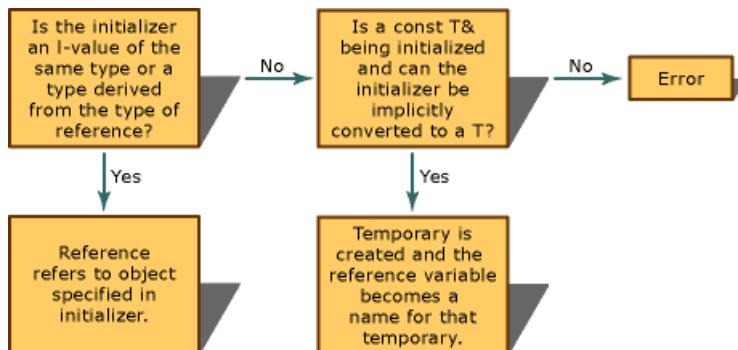
- Объявления члена класса ссылочного типа. Пример:

```
class C {public:    int& i;};
```

- Объявление переменной, явно заданной как `extern`. Пример:

```
extern int& iVal;
```

При инициализации переменной ссылочного типа компилятор с помощью графа принятия решений, показанного на следующем рисунке, выбирает между созданием ссылки на объект и созданием временного объекта, на который указывает ссылка.



Ссылки на `volatile` типы (объявленные как `volatile & идентификатор TypeName`) могут быть инициализированы `volatile` объектами того же типа или с объектами, которые не были объявлены как `volatile`. Однако они не могут быть инициализированы `const` объектами этого типа. Аналогичным образом, ссылки на `const` типы (объявленные как `const & идентификатор TypeName`) можно инициализировать с помощью `const` объектов одного типа (или любого объекта, который имеет преобразование в этот тип или с объектами, которые не были объявлены как `const`). Однако они не

могут быть инициализированы `volatile` объектами этого типа.

Ссылки, которые не являются полными с `const` помощью `volatile` ключевого слова или, могут быть инициализированы только объектами, объявленными как, так `const` и `volatile`.

Инициализация внешних переменных

Объявления автоматических, статических и внешних переменных могут содержать инициализаторы. Однако объявления внешних переменных могут содержать инициализаторы только в том случае, если эти переменные не объявлены как `extern`.

Псевдонимы и определения типов (C++)

12.11.2021 • 6 minutes to read

Объявление псевдонима можно использовать для объявления имени, которое будет использоваться в качестве синонима для ранее объявленного типа. (Этот механизм также называется *псевдонимом типа*). Этот механизм также можно использовать для создания *шаблона псевдонима*, который может быть особенно полезен для пользовательских распределителей.

Синтаксис

```
using identifier = type;
```

Remarks

identifier

Имя псевдонима.

type

Идентификатор типа, для которого создается псевдоним.

Псевдоним не вводит в программу новый тип и не может менять значение существующего имени типа.

Простейшая форма псевдонима эквивалентна `typedef` механизму из C++ 03:

```
// C++11
using counter = long;

// C++03 equivalent:
// typedef long counter;
```

Оба этих механизма позволяют создавать переменные типа "счетчик". Псевдоним типа для

`std::ios_base::fmtflags`, приведенный в следующем примере, может быть более полезен.

```
// C++11
using fmtfl = std::ios_base::fmtflags;

// C++03 equivalent:
// typedef std::ios_base::fmtflags fmtfl;

fmtfl fl_orig = std::cout.flags();
fmtfl fl_hex = (fl_orig & ~std::cout.basefield) | std::cout.showbase | std::cout.hex;
// ...
std::cout.flags(fl_hex);
```

Псевдонимы также работают с указателями на функции, но гораздо удобнее для чтения, чем эквивалентное определение типа:

```
// C++11
using func = void(*)(int);

// C++03 equivalent:
// typedef void (*func)(int);

// func can be assigned to a function pointer value
void actual_function(int arg) { /* some code */ }
func fptr = &actual_function;
```

Ограничением `typedef` механизма является то, что оно не работает с шаблонами. Напротив, синтаксис псевдонима типа в C ++ 11 позволяет создавать шаблоны псевдонимов:

```
template<typename T> using ptr = T*;

// the name 'ptr<T>' is now an alias for pointer to T
ptr<int> ptr_int;
```

Пример

В следующем примере показано, как использовать шаблон псевдонима с пользовательским механизмом выделения памяти — в данном случае целочисленным векторным типом. Вы можете заменить любой тип на `int`, чтобы создать удобный псевдоним для скрытия сложных списков параметров в основном функциональном коде. Применяя по всему коду пользовательские механизмы выделения памяти, можно повысить удобочитаемость и уменьшить риск возникновения ошибок, связанных с опечатками.

```

#include <stdlib.h>
#include <new>

template <typename T> struct MyAlloc {
    typedef T value_type;

    MyAlloc() { }
    template <typename U> MyAlloc(const MyAlloc<U>&) { }

    bool operator==(const MyAlloc&) const { return true; }
    bool operator!=(const MyAlloc&) const { return false; }

    T * allocate(const size_t n) const {
        if (n == 0) {
            return nullptr;
        }

        if (n > static_cast<size_t>(-1) / sizeof(T)) {
            throw std::bad_array_new_length();
        }

        void * const pv = malloc(n * sizeof(T));

        if (!pv) {
            throw std::bad_alloc();
        }

        return static_cast<T *>(pv);
    }

    void deallocate(T * const p, size_t) const {
        free(p);
    }
};

#include <vector>
using MyIntVector = std::vector<int, MyAlloc<int>>;

#include <iostream>

int main ()
{
    MyIntVector foov = { 1701, 1764, 1664 };

    for (auto a: foov) std::cout << a << " ";
    std::cout << "\n";

    return 0;
}

```

1701 1764 1664

Определения типов

typedef Объявление вводит имя, которое в пределах его области видимости преобразуется в синоним для типа, заданного в объявлении *типа*.

Объявления **typedef** можно использовать для создания более коротких или более понятных имен для типов, уже определенных в языке или объявленных пользователем. Имена **typedef** позволяют инкапсулировать детали реализации, которые могут измениться.

В отличие от **class** объявлений, **struct**, **union** и **enum**, **typedef** объявления не предоставляют новые

типы — они представляют новые имена для существующих типов.

Имена, объявленные с помощью `typedef`, занимают то же пространство имен, что и другие идентификаторы (кроме меток операторов). Таким образом, в них не может использоваться тот же идентификатор, что и в объявлении ранее имени, за исключением случаев, когда они находятся в объявлении типа класса. Рассмотрим следующий пример.

```
// typedef_names1.cpp
// C2377 expected
typedef unsigned long UL;    // Declare a typedef name, UL.
int UL;                      // C2377: redefined.
```

Правила скрытия имен, относящиеся к другим идентификаторам, также управляют видимостью имен, объявленных с помощью `typedef`. Поэтому следующий код допустим в C++:

```
// typedef_names2.cpp
typedef unsigned long UL;    // Declare a typedef name, UL
int main()
{
    unsigned int UL;    // Redefinition hides typedef name
}

// typedef UL back in scope
```

```
// typedef_specifier1.cpp
typedef char FlagType;

int main()
{
}

void myproc( int )
{
    int FlagType;
}
```

При объявлении в локальной области идентификатора с тем же именем, что и имя `typedef`, или при объявлении члена структуры либо объединения в той же области или во внутренней области обязательно должен указываться спецификатор типа. Пример:

```
typedef char FlagType;
const FlagType x;
```

Чтобы повторно использовать имя `FlagType` для идентификатора, члена структуры или члена объединения, необходимо указать тип:

```
const int FlagType; // Type specifier required
```

Недостаточно написать

```
const FlagType; // Incomplete specification
```

поскольку `FlagType` воспринимается как часть типа, а не как заново объявляемый идентификатор. Это объявление недопустимо, как и

```
int; // Illegal declaration
```

С помощью `typedef` можно объявить любой тип, включая типы указателей, функций и массивов. Имя `typedef` для типа указателя на структуру или объединение можно объявить до определения типа структуры или объединения, если только определение находится в той же области видимости, что и объявление.

Примеры

Использование `typedef` объявлений состоит в том, чтобы сделать объявления более однородными и компактными. Пример:

```
typedef char CHAR;           // Character type.
typedef CHAR * PSTR;         // Pointer to a string (char *).
PSTR strchr( PSTR source, CHAR target );
typedef unsigned long ulong;
ulong ul;      // Equivalent to "unsigned long ul;"
```

Чтобы использовать `typedef` для указания фундаментальных и производных типов в одном объявлении, можно разделить деклараторы запятыми. Пример:

```
typedef char CHAR, *PSTR;
```

В следующем примере задан тип `DRAWF` для функции, не возвращающей никакого значения и принимающей два аргумента `int`.

```
typedef void DRAWF( int, int );
```

После оператора выше `typedef` объявление

```
DRAWF box;
```

будет эквивалентно следующему:

```
void box( int, int );
```

`typedef` часто объединяется с `struct` для объявления и именования определяемых пользователем типов:

```
// typedefSpecifier2.cpp
#include <stdio.h>

typedef struct mystructtag
{
    int i;
    double f;
} mystruct;

int main()
{
    mystruct ms;
    ms.i = 10;
    ms.f = 0.99;
    printf_s("%d %f\n", ms.i, ms.f);
}
```

```
10 0.990000
```

Повторное объявление определений типов

`typedef` Объявление можно использовать для повторного объявления того же имени для ссылки на один и тот же тип. Пример:

```
// FILE1.H
typedef char CHAR;

// FILE2.H
typedef char CHAR;

// PROG.CPP
#include "file1.h"
#include "file2.h" // OK
```

Программа *Prog.CPP* включает два файла заголовка, оба из которых содержат `typedef` объявления для имени `CHAR`. Если в обоих объявлениях указывается один и тот же тип, такое повторное объявление допустимо.

`typedef` Невозможно переопределить имя, которое ранее было объявлено как другой тип. Таким образом, если *file2.H* содержит

```
// FILE2.H
typedef int CHAR; // Error
```

компилятор выдает ошибку из-за попытки повторного объявления имени `CHAR` как имени другого типа. Это правило распространяется также на конструкции, подобные следующим:

```
typedef char CHAR;
typedef CHAR CHAR; // OK: redeclared as same type

typedef union REGS // OK: name REGS redeclared
{
    // by typedef name with the
    struct wordregs x; // same meaning.
    struct byteregs h;
} REGS;
```

Использование `typedef` спецификатора с типами классов в основном поддерживается из-за объявления неименованных структур в объявлениях в ANSI C `typedef`. Например, многие программисты C используют следующий код.

```
// typedef_with_class_types1.cpp
// compile with: /c
typedef struct { // Declare an unnamed structure and give it the
    // typedef name POINT.
    unsigned x;
    unsigned y;
} POINT;
```

Преимущество такого объявления заключает в том, что можно выполнять объявления

```
POINT ptOrigin;
```

вместо

```
struct point_t ptOrigin;
```

В C++ разница между `typedef` именами и реальными типами (объявленными с `class`, `struct`, `union`, `enum` ключевыми словами), и) более Разна. Хотя методика С объявления структуры без имени в `typedef` инструкции по-прежнему работает, она не предоставляет преимуществ для нотаций, как это делается в C.

```
// typedef_with_class_types2.cpp
// compile with: /c /W1
typedef struct {
    int POINT();
    unsigned x;
    unsigned y;
} POINT;
```

В предыдущем примере объявляется класс с именем `POINT` с использованием синтаксиса неименованного класса `typedef`. `POINT` считается именем класса, однако к именам, предоставленным таким образом, применяются следующие ограничения.

- Имя (сионим) не может использоваться после `class`, `struct`, `union` префикса, или.
- Имя не может использоваться в качестве имени конструктора или деструктора в объявлении класса.

Таким образом, этот синтаксис не предоставляет механизм наследования, создания или удаления.

using - объявление

12.11.2021 • 5 minutes to read

`using` Объявление вводит имя в декларативную область, в которой отображается объявление `using`.

Синтаксис

```
using [typename] nested-name-specifier unqualified-id ;  
using declarator-list ;
```

Параметры

спецификатор вложенного имени Последовательность пространств имен, классов или перечислений, а также операторы разрешения области (::), заканчивающиеся оператором разрешения области. Чтобы ввести имя из глобального пространства имен, можно использовать один оператор разрешения области. Ключевое слово `typename` является необязательным и может использоваться для разрешения зависимых имен при их внедрении в шаблон класса из базового класса.

неполный идентификатор Неполное выражение идентификатора, которое может представлять собой идентификатор, имя перегруженного оператора, определяемый пользователем литеральный оператор или имя функции преобразования, Имя деструктора класса или имя шаблона и список аргументов.

декларатор-список Разделенный запятыми список `typename` описателей с квалифиликаторами неполного имени ([])) с разделителем-запятой, за которым следует, по желанию, многоточие.

Remarks

Объявление `using` вводит неполное имя в качестве синонима сущности, объявленной в другой части. Он позволяет использовать одно имя из определенного пространства имен без явного уточнения в области объявления, в которой оно отображается. Это отличается от [директивы using](#), которая позволяет использовать *все* имена в пространстве имен без уточнения. Ключевое слово также используется для [псевдонимов типов](#).

Пример: `using` объявление в поле класса

Объявление `using` может использоваться в определении класса.

```

// using_declaration1.cpp
#include <stdio.h>
class B {
public:
    void f(char) {
        printf_s("In B::f()\n");
    }

    void g(char) {
        printf_s("In B::g()\n");
    }
};

class D : B {
public:
    using B::f;      // B::f(char) is now visible as D::f(char)
    using B::g;      // B::g(char) is now visible as D::g(char)
    void f(int) {
        printf_s("In D::f()\n");
        f('c');      // Invokes B::f(char) instead of recursing
    }

    void g(int) {
        printf_s("In D::g()\n");
        g('c');      // Invokes B::g(char) instead of recursing
    }
};

int main() {
    D myD;
    myD.f(1);
    myD.g('a');
}

```

```

In D::f()
In B::f()
In B::g()

```

Пример: `using` объявление объявления элемента

Если объявление `using` объявляет член, оно должно ссылаться на член базового класса.

```
// using_declaration2.cpp
#include <stdio.h>

class B {
public:
    void f(char) {
        printf_s("In B::f()\n");
    }

    void g(char) {
        printf_s("In B::g()\n");
    }
};

class C {
public:
    int g();
};

class D2 : public B {
public:
    using B::f; // ok: B is a base of D2
    // using C::g; // error: C isn't a base of D2
};

int main() {
    D2 MyD2;
    MyD2.f('a');
}
```

```
In B::f()
```

Пример: `using` объявление с явной квалификацией

На члены, объявленные с помощью объявления `using`, можно ссылаться с помощью явной квалификации. Префикс `::` указывает на глобальное пространство имен.

```

// using_declaration3.cpp
#include <stdio.h>

void f() {
    printf_s("In f\n");
}

namespace A {
    void g() {
        printf_s("In A::g\n");
    }
}

namespace X {
    using ::f;    // global f is also visible as X::f
    using A::g;   // A's g is now visible as X::g
}

void h() {
    printf_s("In h\n");
    X::f();      // calls ::f
    X::g();      // calls A::g
}

int main() {
    h();
}

```

```

In h
In f
In A::g

```

Пример: `using` объявления синонимы и псевдонимы

Если используется объявление `using`, то созданный им синоним указывает только на определения, которые являются действительными в точке его создания. Определения, добавляемые в пространство имен после объявления `using`, не являются допустимыми синонимами.

Имя, определенное `using` объявлением, является псевдонимом для его исходного имени. Оно не влияет на тип, компоновку и другие атрибуты исходного объявления.

```

// post_declaration_namespace_additions.cpp
// compile with: /c
namespace A {
    void f(int) {}
}

using A::f;    // f is a synonym for A::f(int) only

namespace A {
    void f(char) {}
}

void f() {
    f('a');    // refers to A::f(int), even though A::f(char) exists
}

void b() {
    using A::f;    // refers to A::f(int) AND A::f(char)
    f('a');    // calls A::f(char);
}

```

Пример. локальные объявления И `using` объявления

Говоря о функциях в пространствах имен, если в области объявления задан набор локальных объявлений и объявлений `using` для одного имени, то все они должны указывать на одну и ту же сущность, либо же все они должны указывать на функции.

```

// functions_in_namespaces1.cpp
// C2874 expected
namespace B {
    int i;
    void f(int);
    void f(double);
}

void g() {
    int i;
    using B::i;    // error: i declared twice
    void f(char);
    using B::f;    // ok: each f is a function
}

```

В приведенном выше примере второй оператор `using B::i` объявляет вторую переменную `int i` в функции `g()`. Оператор `using B::f` не конфликтует с функцией `f(char)`, поскольку имена функций, вводимых в код при помощи `B::f`, имеют разные типы параметров.

Пример: объявления локальных функций и `using` объявления

Локальное объявление функции не может иметь такое же имя и тип, что и функция, введенная в код объявлением `using`. Пример:

```
// functions_in_namespaces2.cpp
// C2668 expected
namespace B {
    void f(int);
    void f(double);
}

namespace C {
    void f(int);
    void f(double);
    void f(char);
}

void h() {
    using B::f;           // introduces B::f(int) and B::f(double)
    using C::f;           // C::f(int), C::f(double), and C::f(char)
    f('h');              // calls C::f(char)
    f(1);                // C2668 ambiguous: B::f(int) or C::f(int)?
    void f(int);          // C2883 conflicts with B::f(int) and C::f(int)
}
```

Пример: `using` объявление и наследование

Говоря о наследовании, когда объявление `using` вводит имя из базового класса в область видимости производного класса, то функции-члены из производного класса переопределяют виртуальные функции-члены из базового класса, имеющие такое же имя и типы аргументов.

```

// using_declaration_inheritance1.cpp
#include <stdio.h>
struct B {
    virtual void f(int) {
        printf_s("In B::f(int)\n");
    }

    virtual void f(char) {
        printf_s("In B::f(char)\n");
    }

    void g(int) {
        printf_s("In B::g\n");
    }

    void h(int);
};

struct D : B {
    using B::f;
    void f(int) { // ok: D::f(int) overrides B::f(int)
        printf_s("In D::f(int)\n");
    }

    using B::g;
    void g(char) { // ok: there is no B::g(char)
        printf_s("In D::g(char)\n");
    }

    using B::h;
    void h(int) {} // Note: D::h(int) hides non-virtual B::h(int)
};

void f(D* pd) {
    pd->f(1); // calls D::f(int)
    pd->f('a'); // calls B::f(char)
    pd->g(1); // calls B::g(int)
    pd->g('a'); // calls D::g(char)
}

int main() {
    D * myd = new D();
    f(myd);
}

```

```

In D::f(int)
In B::f(char)
In B::g
In D::g(char)

```

Пример: `using` Специальные возможности объявления

Все экземпляры имени, упоминаемого в объявлении `using`, должны быть доступны. В частности, если объявление `using` используется в производном классе для доступа к базовому классу, то имя члена должно быть доступно. Если имя относится к перегруженной функции-члену, то все функции с этим именем должны быть доступны.

Дополнительные сведения о специальных возможностях членов см. в разделе [Управление доступом к членам](#).

```
// using_declaration_inheritance2.cpp
// C2876 expected
class A {
private:
    void f(char);
public:
    void f(int);
protected:
    void g();
};

class B : public A {
    using A::f;    // C2876: A::f(char) is inaccessible
public:
    using A::g;    // B::g is a public synonym for A::g
};
```

См. также раздел

[Пространства имен](#)

[Ключевые слова](#)

volatile (C++)

12.11.2021 • 3 minutes to read

Квалификатор типа, который используется для объявления о том, что объект может быть изменен в программе аппаратным обеспечением.

Синтаксис

```
volatile declarator ;
```

Remarks

Чтобы изменить способ, которым компилятор интерпретирует это ключевое слово, можно использовать параметр компилятора [/volatile](#).

Visual Studio интерпретирует `volatile` ключевое слово по-разному в зависимости от целевой архитектуры. Для ARM, если не указан параметр компилятора `/volatile`, компилятор выполняет команду, как если бы был указан `/volatile: ISO`. Для архитектур, отличных от ARM, если не указан параметр компилятора `/volatile`, компилятор выполняет команду, как если бы был указан `/volatile: MS`. Таким образом, для архитектур, отличных от ARM, настоятельно рекомендуется указывать `/volatile: ISO` и использовать явные примитивы синхронизации и встроенные функции компилятора при работе с памятью, которая совместно используется в потоках.

Квалификатор можно использовать `volatile` для предоставления доступа к областям памяти, используемым асинхронными процессами, такими как обработчики прерываний.

Если `volatile` используется для переменной, которая также имеет ключевое слово `_restrict`, `volatile` имеет приоритет.

Если `struct` член помечен как `volatile`, то `volatile` он передается всей структуре. Если структура не имеет длины, которую можно скопировать в текущей архитектуре с помощью одной инструкции, `volatile` может быть полностью утеряна в этой структуре.

`volatile` Ключевое слово может не влиять на поле, если выполняется одно из следующих условий.

- Длина поля с ключевым словом `volatile` превышает максимальный размер, который в текущей архитектуре может быть скопирован с помощью одной инструкции.
- Длина самого внешнего объекта `struct`, или если он является членом потенциально вложенного объекта, `struct` превышает максимальный размер, который можно скопировать в текущую архитектуру с помощью одной инструкции.

Несмотря на то, что процессор не переупорядочивает доступ к некэшированной памяти, переменные без возможности кэширования должны быть помечены как, `volatile` чтобы гарантировать, что компилятор не переупорядочивает доступ к памяти.

Объекты, объявленные как `volatile`, не используются в определенных оптимизациях, так как их значения могут изменяться в любое время. При запросе объекта с ключевым словом `volatile` система всегда считывает его текущее значение, даже если оно запрашивалось в предшествовавшей инструкции. Кроме того, значение объекта записывается непосредственно при присваивании.

Блок, относящийся к стандарту ISO

если вы знакомы с ключевым словом "volatile" в C# или знакомы с поведением `volatile` в более ранних версиях компилятора Microsoft C++ (MSVC), имейте в виду, что ключевое слово C++ 11 iso Standard отличается `volatile` и поддерживается в MSVC при указании параметра компилятора `/volatile: ISO`. (Для архитектуры ARM он установлен по умолчанию). `volatile` Ключевое слово в стандартном коде ISO 11 языка C++ используется только для доступа к оборудованию; не используйте его для обмена данными между потоками. Для обмена данными между потоками используйте такие механизмы, как `std::<T> Atomic` из [стандартной библиотеки C++](#).

Конец блока, относящегося к стандарту ISO

Блок, относящийся только к системам Microsoft

Когда используется параметр компилятора `/volatile: MS` — по умолчанию, если нацелены архитектуры, отличные от ARM, компилятор создает дополнительный код для поддержания упорядочения между ссылками на переменные объекты, а также для поддержания порядка ссылок на другие глобальные объекты. В частности:

- Запись в объект с ключевым словом `volatile` (т. н. "запись в изменяемый объект") имеет семантику освобождения. Это означает, что ссылка на глобальный или статический объект, которая находится в последовательности инструкций перед записью в объект с ключевым словом `volatile`, в скомпилированном двоичном файле будет находиться до записи в изменяемый объект.
- Считывание из объекта с ключевым словом `volatile` (т. н. "считывание из изменяемого объекта") имеет семантику получения. Это означает, что ссылка на глобальный или статический объект, которая находится в последовательности инструкций после считывания из объекта с ключевым словом `volatile`, в скомпилированном двоичном файле будет находиться после считывания из изменяемого объекта.

Благодаря этому объекты с ключевым словом `volatile` могут использоваться для блокировки и освобождения памяти в многопоточных приложениях.

NOTE

Если используется расширенная гарантия, предоставляемая при использовании параметра компилятора `/volatile: MS`, код не является переносимым.

Завершение блока, относящегося только к системам Microsoft

См. также

[Ключевые слова](#)

`const`

[константные и переменные указатели](#)

decltype (C++)

12.11.2021 • 5 minutes to read

`decltype` Описатель типа возвращает тип указанного выражения. `decltype` Спецификатор типа (вместе с `auto` [ключевым словом](#)) полезен в основном для разработчиков, создающих библиотеки шаблонов. Используйте `auto` и `decltype` для объявления функции шаблона, тип возвращаемого значения которого зависит от типов своих аргументов шаблона. Или используйте `auto` и `decltype` для объявления функции шаблона, которая заключает в оболочку вызов другой функции, а затем возвращает тип возвращаемого значения функции с оболочкой.

Синтаксис

```
decltype( выражение )
```

Параметры

выражение

Выражение. Дополнительные сведения см. в разделе [выражения](#).

Возвращаемое значение

Тип параметра *выражения*.

Remarks

`decltype` спецификатор типа поддерживается в Visual Studio 2010 или более поздних версиях и может использоваться с машинным или управляемым кодом. `decltype(auto)` (C++ 14) поддерживается в Visual Studio 2015 и более поздних версиях.

Для определения типа параметра *выражения* компилятор использует следующие правила.

- Если параметр *выражения* является идентификатором или [доступом к члену класса](#), `decltype(expression)` то является типом сущности с именем *Expression*. Если такая сущность или параметр *выражения* не называет набор перегруженных функций, компилятор выдает сообщение об ошибке.
- Если параметр *выражения* является вызовом функции или перегруженной функции оператора, то аргумент `decltype(expression)` является типом возвращаемого значения функции. Скобки вокруг перегруженного оператора игнорируются.
- Если параметр *Expression* является `rvalue`, `decltype(expression)` то является типом *выражения*. Если параметр *Expression* является [левосторонним значением](#), `decltype(expression)` то является [ссылкой lvalue](#) на тип *выражения*.

В следующем примере кода демонстрируется использование `decltype` спецификатора типа. Допустим, во-первых, что были закодированы следующие операторы.

```
int var;
const int&& fx();
struct A { double x; }
const A* a = new A();
```

Затем проверьте типы, возвращаемые четырьмя `decltype` инструкциями, приведенными в следующей таблице.

	ТИП	ПРИМЕЧАНИЯ
<code>decltype(fx());</code>	<code>const int&&</code>	Ссылка <code>rvalue</code> на <code>const int</code> .
<code>decltype(var);</code>	<code>int</code>	Тип переменной <code>var</code> .
<code>decltype(a->x);</code>	<code>double</code>	Тип членского доступа.
<code>decltype((a->x));</code>	<code>const double&</code>	Внутренние скобки вызывают оценку оператора в качестве выражения, а не членского доступа. И, поскольку <code>a</code> объявляется как <code>const</code> указатель, тип является ссылкой на <code>const double</code> .

Decltype и Auto

В C++ 14 можно использовать `decltype(auto)` без завершающего возвращаемого типа для объявления функции-шаблона, тип возвращаемого значения которого зависит от типов своих аргументов шаблона.

В C++ 11 можно использовать `decltype` спецификатор типа для завершающего возвращаемого типа вместе с `auto` ключевым словом, чтобы объявить функцию шаблона, тип возвращаемого значения которого зависит от типов своих аргументов шаблона. Например, рассмотрим следующий пример кода, в котором тип возвращаемого значения функции шаблона зависит от типов аргументов шаблона. В примере кода *неизвестный* заполнитель указывает, что тип возвращаемого значения не может быть указан.

```
template<typename T, typename U>
UNKNOWN func(T&& t, U&& u){ return t + u; };
```

Введение `decltype` описателя типа позволяет разработчику получить тип выражения, возвращаемого функцией-шаблоном. Используйте *альтернативный синтаксис объявления функции*, который показан далее, `auto` ключевое слово и `decltype` описатель типа для объявления *позднего указанного* возвращаемого типа. Поздно заданный возвращаемый тип определяется, когда происходит компиляция, а не кодирование объявления.

Следующий прототип иллюстрирует синтаксис альтернативного объявления функции. Обратите внимание, что `const` `volatile` квалифиликаторы и, а также `throw` спецификация исключения являются необязательными. Заполнитель *function_body* представляет составной оператор, указывающий, что делает функция. В качестве лучшего написания кода заполнитель *выражения* в `decltype` операторе должен соответствовать выражению, заданному `return` оператором, если таковое имеется, в *function_body*.

```
auto function_name () Параметры не следует заменять ) const volatile -> decltype(  
выражение opt ) noexcept { function_body };
```

В следующем примере кода поздно заданный возвращаемый тип функции шаблона `myFunc` определяется типами аргументов шаблона `t` и `u`. В качестве лучшего программирования в примере кода также используются ссылки `rvalue` и `forward` шаблон функции, поддерживающие *идеальную пересылку*. Дополнительные сведения см. в статье [Декларатор ссылки Rvalue: &&](#).

```
//C++11
template<typename T, typename U>
auto myFunc(T&& t, U&& u) -> decltype (forward<T>(t) + forward<U>(u))
    { return forward<T>(t) + forward<U>(u); }

//C++14
template<typename T, typename U>
decltype(auto) myFunc(T&& t, U&& u)
    { return forward<T>(t) + forward<U>(u); }
```

Decltype и функции пересылки (C++ 11)

Функции пересылки создают программы-оболочки для вызовов других функций. Рассмотрим шаблон функции, который пересыпает свои аргументы (или результаты выражения с этими аргументами) другой функции. Кроме того, функция пересылки возвращает результат вызова другой функции. В этом сценарии тип возвращаемого значения функции пересылки должен совпадать с типом возвращаемого значения функции в программе-оболочке.

В этом сценарии нельзя писать соответствующее выражение типа без `decltype` спецификатора типа.

`decltype` Спецификатор типа позволяет использовать универсальные функции перенаправления, так как не теряет необходимых сведений о том, возвращает ли функция ссылочный тип. Пример кода функции пересылки см. в предыдущем примере шаблонной функции `myFunc`.

Примеры

В следующем примере кода объявляется поздно заданный возвращаемый тип шаблонной функции `Plus()`. `Plus` Функция обрабатывает два операнда с помощью `operator+` перегрузки. Следовательно, интерпретация оператора плюс (`+`) и возвращаемого типа `Plus` функции зависит от типов аргументов функции.

```

// decltype_1.cpp
// compile with: cl /EHsc decltype_1.cpp

#include <iostream>
#include <string>
#include <utility>
#include <iomanip>

using namespace std;

template<typename T1, typename T2>
auto Plus(T1&& t1, T2&& t2) ->
    decltype(forward<T1>(t1) + forward<T2>(t2))
{
    return forward<T1>(t1) + forward<T2>(t2);
}

class X
{
    friend X operator+(const X& x1, const X& x2)
    {
        return X(x1.m_data + x2.m_data);
    }

public:
    X(int data) : m_data(data) {}
    int Dump() const { return m_data; }
private:
    int m_data;
};

int main()
{
    // Integer
    int i = 4;
    cout <<
        "Plus(i, 9) = " <<
        Plus(i, 9) << endl;

    // Floating point
    float dx = 4.0;
    float dy = 9.5;
    cout <<
        setprecision(3) <<
        "Plus(dx, dy) = " <<
        Plus(dx, dy) << endl;

    // String
    string hello = "Hello, ";
    string world = "world!";
    cout << Plus(hello, world) << endl;

    // Custom type
    X x1(20);
    X x2(22);
    X x3 = Plus(x1, x2);
    cout <<
        "x3.Dump() = " <<
        x3.Dump() << endl;
}

```

```

Plus(i, 9) = 13
Plus(dx, dy) = 13.5
Hello, world!
x3.Dump() = 42

```

Visual Studio 2017 и более поздних версий: Компилятор анализирует `decltype` аргументы при объявлении шаблонов, а не на экземпляре. Следовательно, если в аргументе найдена независимая специализация `decltype`, она не будет откладываться на время создания экземпляра и будет обработана немедленно и все возникшие ошибки будут диагностироваться в это время.

В следующем примере показана такая ошибка компилятора, возникающая во время объявления.

```
#include <utility>
template <class T, class ReturnT, class... ArgsT> class IsCallable
{
public:
    struct BadType {};
    template <class U>
    static decltype(std::declval<T>()(std::declval<ArgsT>(...))) Test(int); //C2064. Should be decltype<U>
    template <class U>
    static BadType Test(...);
    static constexpr bool value = std::is_convertible<decltype(Test<T>(0)), ReturnT>::value;
};

constexpr bool test1 = IsCallable<int(), int>::value;
static_assert(test1, "PASS1");
constexpr bool test2 = !IsCallable<int*, int>::value;
static_assert(test2, "PASS2");
```

Требования

Visual Studio 2010 или более поздних версий.

`decltype(auto)` требуется Visual Studio 2015 или более поздней версии.

Атрибуты в C++

12.11.2021 • 3 minutes to read

Стандарт C++ определяет общий набор атрибутов. Он также позволяет поставщикам компиляторов определять собственные атрибуты в пространстве имен, характерном для конкретного поставщика. Однако компиляторы необходимы только для распознавания атрибутов, определенных в стандарте.

В некоторых случаях стандартные атрибуты перекрываются с параметрами, зависящими от компилятора `__declspec`. В Microsoft C++ `[[deprecated]]` вместо использования можно использовать атрибут `__declspec(deprecated)`. Атрибут распознается любым согласованным компилятором. Для всех остальных `__declspec` параметров, таких как `dllimport` и, пока нет `dllexport` эквивалента атрибута, поэтому необходимо продолжать использовать `__declspec` синтаксис. Атрибуты не влияют на систему типов и не изменяют значение программы. Компиляторы не учитывают значения атрибутов, которые они не распознают.

Visual Studio 2017 версии 15,3 и более поздних версий (доступно в `/std:c++17`): в области списка атрибутов можно указать пространство имен для всех имен одним знаком `using`:

```
void g() {
    [[using rpr: kernel, target(cpu,gpu)]] // equivalent to [[ rpr::kernel, rpr::target(cpu,gpu) ]]
    do task();
}
```

Стандартные атрибуты C++

В C++ 11 атрибуты предоставляют стандартизованный способ комментирования конструкций C++ (включая классы, функции, переменные и блоки) с дополнительными сведениями. Атрибуты могут быть или не зависят от поставщика. Компилятор может использовать эти сведения для создания информационных сообщений или для применения специальной логики при компиляции кода с атрибутом. Компилятор игнорирует все нераспознаваемые атрибуты, что означает, что вы не можете определить собственные настраиваемые атрибуты с помощью этого синтаксиса. Атрибуты заключены в двойные квадратные скобки:

```
[[deprecated]]
void Foo(int);
```

Атрибуты представляют стандартизированную альтернативу модулям, зависящим от поставщика `#pragma`, например директивам, `__declspec()` (Visual C++) или `__attribute__` (GNU). Однако для большинства целей вам по-прежнему потребуется использовать конструкции для конкретного поставщика. В настоящее время стандартный указывает следующие атрибуты, которые должен распознать компилятор:

- `[[noreturn]]` Указывает, что функция никогда не возвращает значение; Иными словами, всегда возникает исключение. Компилятор может настроить правила компиляции для `[[noreturn]]` сущностей.
- `[[carries_dependency]]` Указывает, что функция распространяет упорядочение зависимостей данных для синхронизации потоков. Атрибут можно применить к одному или нескольким параметрам, чтобы указать, что переданный аргумент несет зависимость в тело функции. Атрибут может применяться к самой функции, чтобы указать, что возвращаемое значение несет

зависимость от функции. Компилятор может использовать эти сведения для создания более эффективного кода.

- **[[deprecated]] Visual Studio 2015 и более поздних версий:** Указывает, что функция не предназначена для использования. Кроме того, он может не существовать в будущих версиях интерфейса библиотеки. Компилятор может использовать этот атрибут для создания информационного сообщения, когда клиентский код пытается вызвать функцию. **[[deprecated]]** может применяться к объявлению класса, `typedef`-Name, переменной, нестатическому элементу данных, функции, пространству имен, перечислению, перечислителю или специализации шаблона.
- **[[fallthrough]] Visual Studio 2017 и более поздних версий:** (доступно с `/std:c++17`) **[[fallthrough]]** атрибут может использоваться в контексте `switch` инструкций в качестве подсказки для компилятора (или любого,читывающего код), который предназначен для `fallthrough` поведения. Компилятор Microsoft C++ в настоящее время не предупреждает о поведении `fallthrough`, поэтому этот атрибут не влияет на поведение компилятора.
- **[[nodiscard]] Visual Studio 2017 версии 15,3 и более поздних версий:** (доступно с `/std:c++17`) указывает, что возвращаемое значение функции не предназначено для удаления. Вызывает предупреждение [порог предупреждения c4834](#), как показано в следующем примере:

```
[[nodiscard]]
int foo(int i) { return i * i; }

int main()
{
    foo(42); //warning C4834: discarding return value of function with 'nodiscard' attribute
    return 0;
}
```

- **[[maybe_unused]] Visual Studio 2017 версии 15,3 и более поздних версий:** (доступно с `/std:c++17`) указывает, что переменные, функции, классы, `typedef`, нестатические элементы данных, перечисления или специализацию шаблона могут быть намеренно не использованы. Компилятор не выдает предупреждение, если помеченная сущностью **[[maybe_unused]]** не используется. Сущность, объявленная без атрибута, может быть впоследствии повторно объявлена с атрибутом и наоборот. Сущность считается *помеченной* после первого объявления, помеченного как "**[[maybe_unused]]** проанализировано", и для оставшейся части текущего блока преобразования.

Атрибуты, относящиеся к Microsoft

- **[[gsl::suppress(rules)]]** Этот атрибут, предназначенный для Microsoft, используется для подавления предупреждений от тех, которые применяют правила [библиотеки \(GSL\)](#) в коде. Например, рассмотрим следующий фрагмент кода:

```
int main()
{
    int arr[10]; // GSL warning C26494 will be fired
    int* p = arr; // GSL warning C26485 will be fired
    [[gsl::suppress(bounds.1)]] // This attribute suppresses Bounds rule #1
    {
        int* q = p + 1; // GSL warning C26481 suppressed
        p = q--; // GSL warning C26481 suppressed
    }
}
```

В этом примере возникают следующие предупреждения:

- 26494 (тип правило 5: всегда инициализировать объект.)
- 26485 (ограничивающее правило 3: нет массива для Decay указателя.)
- 26481 (правило границы 1: не используйте арифметические действия с указателями. Вместо этого используйте Span.)

Первые два предупреждения срабатывают при компиляции этого кода с установленным и активируемым средством анализа кода CppCoreCheck. Но третье предупреждение не срабатывает из-за атрибута. Вы можете подавить весь профиль границ, записав его `[[gs1::suppress(bounds)]]` без включения определенного номера правила. C++ Core Guidelines предназначены для помощи в написании лучшего и безопасного кода. Атрибут подавлять упрощает отключение предупреждений, когда они не нужны.

Встроенные операторы, приоритет и ассоциативность C++

12.11.2021 • 2 minutes to read

Язык C++ включает все операторы С и еще несколько новых операторов. Операторы определяют, какое вычисление следует выполнить с одним или несколькими operandами.

Очередность и ассоциативность

Приоритет операторов задает порядок операций в выражениях, содержащих более одного оператора.

Ассоциативность операторов указывает, будет ли operand в выражении, содержащем несколько операторов с одинаковым приоритетом, группироваться по левому краю или по правому.

Альтернативные слова

C++ указывает альтернативные слова для некоторых операторов. В языке С альтернативное написание предоставляется в виде макросов в `<iso646.h>` заголовке. В C++ эти альтернативы являются ключевыми словами, использование `<iso646.h>` или эквивалент C++ `<ciso646>` не рекомендуется. В Microsoft C++ `/permissive-` `/Za` параметр компилятора или необходим для включения альтернативного написания.

Приоритет операторов C++ и таблица ассоциативных данных

В следующей таблице показан приоритет и ассоциативность операторов C++ (в порядке убывания приоритета). Операторы с тем же номером приоритета имеют равный приоритет, если другие связи не заданы явно с помощью круглых скобок.

ОПИСАНИЕ ОПЕРАТОРА	ОПЕРАТОР	АЛЬТЕРНАТИВА
Приоритет группы 1, без ассоциативности		
Разрешение области	<code>::</code>	
Приоритет группы 2, ассоциативность слева направо		
Выбор члена для указателей (объект или указатель)	<code>. ни -></code>	
Индекс массива	<code>[]</code>	
Вызов функции	<code>()</code>	
Постфиксный инкремент	<code>++</code>	
Постфиксный декремент	<code>--</code>	
Имя типа	<code>typeid</code>	

ОПИСАНИЕ ОПЕРАТОРА	ОПЕРАТОР	АЛЬТЕРНАТИВА
Постоянное преобразование типа	<code>const_cast</code>	
Динамическое преобразование типа	<code>dynamic_cast</code>	
Повторно интерпретируемое преобразование типа	<code>reinterpret_cast</code>	
Статическое преобразование типа	<code>static_cast</code>	
Приоритет группы 3, ассоциативность справа налево		
Размер объекта или типа	<code>sizeof</code>	
Префиксный инкремент	<code>++</code>	
Префиксный декремент	<code>--</code>	
Дополнение одного	<code>~</code>	<code>compl</code>
Логическое не	<code>!</code>	<code>not</code>
Унарное отрицание	<code>-</code>	
Унарный плюс	<code>+</code>	
Взятие адреса	<code>&</code>	
Косвенное обращение	<code>*</code>	
Создать объект	<code>new</code>	
Уничтожение объекта	<code>delete</code>	
Приведение	<code>()</code>	
Приоритет группы 4, ассоциативность слева направо		
Указатель на член (объекты или указатели)	<code>.*</code> НИ <code>->*</code>	
Приоритет группы 5, ассоциативность слева направо		
Умножения	<code>*</code>	
Отдел	<code>/</code>	
Modulus	<code>%</code>	

ОПИСАНИЕ ОПЕРАТОРА	ОПЕРАТОР	АЛЬТЕРНАТИВА
Приоритет группы 6, ассоциативность слева направо		
Полняют	<input type="button" value="+"/>	
Вычитания	<input type="button" value="-"/>	
Приоритет группы 7, ассоциативность слева направо		
Сдвиг влево	<input type="button" value="<<"/>	
Сдвиг вправо	<input type="button" value"=""/> >>	
Приоритет группы 8, ассоциативность слева направо		
Меньше	<input type="button" value"=""/> <	
Больше чем	<input type="button" value"=""/> >	
Меньше или равно	<input type="button" value"=""/> <=	
Больше или равно	<input type="button" value"=""/> >=	
Приоритет группы 9, ассоциативность слева направо		
Равенство	<input type="button" value"=""/> ==	
Неравенство	<input type="button" value"=""/> !=	not_eq
Группа 10 приоритета с ассоциативностью слева направо		
Побитовое И	<input type="button" value"=""/> &	bitand
Приоритет группы 11, ассоциативность слева направо		
Побитовое исключающее ИЛИ	<input type="button" value"=""/> ^	xor
Приоритет группы 12, ассоциативность слева направо		
Побитовое ИЛИ	<input type="button" value"=""/>	bitor
Приоритет группы 13, ассоциативность слева направо		

ОПИСАНИЕ ОПЕРАТОРА	ОПЕРАТОР	АЛЬТЕРНАТИВА
Логическое И	&&	and
Приоритет группы 14, ассоциативность слева направо		
Логическое ИЛИ		or
Группирование с приоритетом 15, с ассоциативностью справа налево		
Условие	? :	
Назначение	=	
Присваивание умножения	*=	
Присваивание деления	/=	
Назначение модуля	%=	
Присваивание сложения	+ =	
Присваивание вычитания	- =	
Присваивание сдвига влево	<< =	
Присваивание сдвига вправо	>> =	
Назначение побитового И	& =	and_eq
Назначение побитового включающего ИЛИ	=	or_eq
Назначение побитового исключающего ИЛИ	^ =	xor_eq
Выражение Throw	throw	
Группирование с приоритетом 16, с ассоциативностью слева направо		
Запятая	,	

См. также

[Перегрузка операторов](#)

Оператор alignof

12.11.2021 • 2 minutes to read

`alignof` Оператор возвращает выравнивание в байтах указанного типа в виде значения типа `size_t`.

Синтаксис

```
alignof( type )
```

Remarks

Пример:

EXPRESSION	ЗНАЧЕНИЕ
<code>alignof(char)</code>	1
<code>alignof(short)</code>	2
<code>alignof(int)</code>	4
<code>alignof(long long)</code>	8
<code>alignof(float)</code>	4
<code>alignof(double)</code>	8

Значение совпадает со `alignof` значением для `sizeof` базовых типов. Однако рассмотрим следующий пример.

```
typedef struct { int a; double b; } S;
// alignof(S) == 8
```

В этом случае `alignof` значение является требованием выравнивания для самого крупного элемента в структуре.

Аналогичным образом, в

```
typedef __declspec(align(32)) struct { int a; } S;
```

`alignof(S)` равно 32.

Одним из них будет использование в `alignof` качестве параметра для одной из собственных подпрограмм выделения памяти. Например, в следующей определенной структуре `S` можно вызвать подпрограмму выделения памяти с именем `aligned_malloc` для выделения памяти на определенной границе выравнивания.

```
typedef __declspec(align(32)) struct { int a; double b; } S;
int n = 50; // array size
S* p = (S*)aligned_malloc(n * sizeof(S), alignof(S));
```

Дополнительные сведения об изменении выравнивания см. в следующих разделах.

- [pack](#)
- [align](#)
- [_unaligned](#)
- [/Zp \(Выравнивание члена структуры\)](#)
- [Примеры выравнивания структуры](#) (только для x64)

Дополнительные сведения о различиях в выравнивании в коде для 32- (x86) и 64-разрядных (x64) сред см. в статье

- [Конфликтует с компилятором x86](#)

Специально для систем Майкрософт

`alignof` И `_alignof` являются синонимами в компиляторе Майкрософт. До того, как она стала частью стандарта в C++ 11, оператор, предоставляемый корпорацией Майкрософт, `_alignof` предоставил эти функциональные возможности. Для максимальной переносимости следует использовать `alignof` оператор вместо оператора, относящегося к Microsoft `_alignof`.

Для совместимости с предыдущими версиями аргумент `_alignof` является синонимом, `_alignof` если только параметр компилятора не [/za](#) (отключает расширения языка).

См. также

[Выражения с унарными операторами](#)

[Ключевые слова](#)

Оператор `_uuidof`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Извлекает идентификатор GUID, присоединенный к выражению.

Синтаксис

```
_uuidof ( выражение )
```

Remarks

Выражение может быть именем типа, указателем, ссылкой или массивом этого типа, шаблоном, специализированным для этих типов, или переменной этих типов. Этот аргумент является допустимым, если компилятор может использовать его для поиска прикрепленного GUID.

Особым случаем этого встроенного параметра является то, что в качестве аргумента указывается значение 0 или null. В этом случае вернет `_uuidof` идентификатор GUID, состоящие из нулей.

Это ключевое слово позволяет извлечь GUID, прикрепленный к следующим объектам:

- Объект по `uuid` расширенному атрибуту.
- Блок библиотеки, созданный с помощью `module` атрибута.

NOTE

В отладочной сборке `_uuidof` всегда инициализирует объект динамически (во время выполнения). В сборке выпуска `_uuidof` можно статически (во время компиляции) инициализировать объект.

Для совместимости с предыдущими версиями аргумент `_uuidof` является синонимом, `_uuidof` если только параметр компилятора не `/za` ([отключает расширения языка](#)).

Пример

Следующий код (скомпилированный с библиотекой ole32.lib) будет выводить идентификатор uuid для блока библиотеки, созданного с атрибутом module.

```
// expre_uuidof.cpp
// compile with: ole32.lib
#include <stdio.h>
#include <windows.h>

[emitidl];
[module(name="MyLib")];
[export]
struct stuff {
    int i;
};

int main() {
    LPOLESTR lpolestr;
    StringFromCLSID(__uuidof(MyLib), &lpolestr);
    wprintf_s(L"%s", lpolestr);
    CoTaskMemFree(lpolestr);
}
```

Комментарии

В случаях, когда имя библиотеки больше не находится в области, можно использовать `__LIBID_` вместо `__uuidof`. Пример:

```
StringFromCLSID(__LIBID_, &lpolestr);
```

Завершение блока, относящегося только к системам Майкрософт

См. также

[Выражения с унарными операторами](#)

[Ключевые слова](#)

Аддитивные операторы: + и -

12.11.2021 • 2 minutes to read

Синтаксис

```
expression + expression  
expression - expression
```

Remarks

Ниже перечислены аддитивные операторы.

- Сложение (+)
- Вычитание (-)

Эти бинарные операторы имеют ассоциативность слева направо.

Аддитивные операторы принимают операнды арифметических типов или типов указателей. Результатом оператора сложения (+) является сумма operandов. Результатом оператора вычитания (-) является разница между operandами. Если один или оба operandы являются указателями, они должны быть указателями на объекты, а не на функции. Когда оба operandы являются указателями, результаты имеют смысл только в том случае, если оба operandы указывают на объекты в одном массиве.

Аддитивные операторы принимают operandы *арифметических, целочисленных и скалярных* типов. Они описаны в следующей таблице.

Типы, используемые с аддитивными операторами

ТИП	ЗНАЧЕНИЕ
<i>Арифметические</i>	Целочисленные типы и типы с плавающей запятой собирательно называются "арифметическими" типами.
<i>целочисленный</i>	Типы <i>char</i> и <i>int</i> всех размеров (<i>long, short</i>), а также перечисления называются "целочисленными типами".
<i>функцией</i>	Скалярные operandы — это operandы арифметического типа или типа указателя.

Допускаются следующие сочетания этих операторов:

арифметические операции + арифметические операции

Скалярная величина + целая

целая + Скалярная величина

арифметические операции - арифметические операции

Скалярная величина - Скалярная величина

Обратите внимание, что сложение и вычитание не являются эквивалентными операциями.

Если оба операнда имеют арифметический тип, то преобразования, охваченные [стандартными преобразованиями](#), применяются к operandам, а результат относится к преобразованному типу.

Пример

```
// expre_Additive_Operators.cpp
// compile with: /EHsc
#include <iostream>
#define SIZE 5
using namespace std;
int main() {
    int i = 5, j = 10;
    int n[SIZE] = { 0, 1, 2, 3, 4 };
    cout << "5 + 10 = " << i + j << endl
        << "5 - 10 = " << i - j << endl;

    // use pointer arithmetic on array

    cout << "n[3] = " << *( n + 3 ) << endl;
}
```

Добавление указателей

Если один из operandов в операции сложения является указателем на массив объектов, другой должен иметь целочисленный тип. Результатом является указатель, имеющий тот же тип, что и исходный указатель, и указывающий на другой элемент массива. Эта концепция проиллюстрирована в следующем фрагменте кода.

```
short IntArray[10]; // Objects of type short occupy 2 bytes
short *pIntArray = IntArray;

for( int i = 0; i < 10; ++i )
{
    *pIntArray = i;
    cout << *pIntArray << "\n";
    pIntArray = pIntArray + 1;
}
```

Несмотря на то что целочисленное значение 1 добавляется в `pIntArray`, это не означает "добавить 1 к адресу"; скорее, это означает "корректировать указатель так, чтобы он указывал на следующий объект в массиве", то есть через 2 байта (или `sizeof(int)`).

NOTE

Код формы `pIntArray = pIntArray + 1` редко можно найти в программах на C++; чтобы выполнить пошаговое увеличение, предпочтительно использовать следующие формы: `pIntArray++` или `pIntArray += 1`.

Вычитание указателей

Если оба операнда являются указателями, то результатом вычитания будет число элементов массива, находящихся между operandами. Выражение вычитания возвращает целочисленный результат со знаком типа `ptrdiff_t` (определенный в стандартном включаемом файле `<stddef.h>`).

Второй operand может иметь целочисленное значение. Результат вычитания имеет тот же тип, что и исходный указатель. Значение вычитания является указателем на элемент массива ($n - i$), где n — элемент, на который указывает исходный указатель, а i — это целочисленное значение второго

операнда.

См. также

[Выражения с бинарными операторами](#)

[Операторы C++, приоритет и ассоциативность](#)

[Аддитивные операторы в C](#)

Оператор взятия адреса: &

12.11.2021 • 2 minutes to read

Синтаксис

& *cast-expression*

Remarks

Унарный оператор взятия адреса (`&`) принимает адрес своего операнда. Операнд оператора взятия адреса может быть либо указателем функции, либо l-значением, обозначающим объект, который не является битовым полем.

Оператор взятия адреса можно применять только к переменным типов фундаментальных, структурных, классов или объединений, объявленных на уровне области файла, или на ссылки на массивы в индексах. В этих выражениях константное выражение, которое не включает оператор взятия адреса, можно добавлять в выражение address-of или вычитать из него.

Если этот оператор применяется к функциям или l-значениям, то результатом является тип указателя (r-значение), производный от типа операнда. Например, если operand имеет тип `char`, результат выражения будет иметь тип `pointer`, равный `char`. Оператор взятия адреса, примененный к `const volatile` объектам или, принимает значение `const type *` или `volatile type *`, где `type` — Тип исходного объекта.

Адрес перегруженной функции может быть получен только в том случае, если ясно, какая версия функции упоминается. Сведения о получении адреса определенной перегруженной функции см. в разделе [Перегрузка функции](#).

Если оператор взятия адреса применяется к полному имени, результат зависит от того, указывает ли *полное имя* на статический член. Если да, то результатом является указатель на тип, заданный в определении этого члена. Для элемента, который не является статическим, результатом является указатель на имя члена класса, обозначенного *полным именем класса*. Дополнительные сведения о *квалифицированном имени класса* см. в разделе [основные выражения](#).

Пример: адрес статического члена

В следующем фрагменте кода показано, каким образом результат оператора взятия адреса отличается в зависимости от того, является ли член класса статическим:

```
// expe_Address_Of_Operator.cpp
// C2440 expected
class PTM {
public:
    int iValue;
    static float fValue;
};

int main() {
    int *piValue = &PTM::iValue; // OK: non-static
    float *pfValue = &PTM::fValue; // C2440 error: static
    float *spfValue = &PTM::fValue; // OK
}
```

В этом примере выражение `&PTM::fValue` имеет результатом тип `float *`, а не `float PTM::*`, поскольку `fValue` является статическим членом.

Пример: адрес ссылочного типа

Применение оператора взятия адреса к типу ссылки дает тот же результат, что и применение к объекту, к которому привязана ссылка. Пример:

```
// expr_Address_Of_Operator2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
    double d;           // Define an object of type double.
    double& rd = d;   // Define a reference to the object.

    // Obtain and compare their addresses
    if( &d == &rd )
        cout << "&d equals &rd" << endl;
}
```

```
&d equals &rd
```

Пример. адрес функции в качестве параметра

В следующем примере оператор взятия адреса используется для того, чтобы передать указатель в качестве аргумента функции:

```
// expr_Address_Of_Operator3.cpp
// compile with: /EHsc
// Demonstrate address-of operator &

#include <iostream>
using namespace std;

// Function argument is pointer to type int
int square( int *n ) {
    return (*n) * (*n);
}

int main() {
    int mynum = 5;
    cout << square( &mynum ) << endl;    // pass address of int
}
```

```
25
```

См. также

[Выражения с унарными операторами](#)

[Встроенные операторы, приоритет и ассоциативность C++](#)

[Декларатор ссылки Lvalue: &](#)

[Операторы косвенного обращения и адреса операнда](#)

Операторы присваивания

12.11.2021 • 4 minutes to read

Синтаксис

выражение *оператора присваивания* выражения

assignment-operator: один из следующих операторов:

= *= /= %= += -= <<= >>= &= ^= |=

Remarks

Операторы присваивания хранят значение в объекте, указанном левым операндом. Существует два вида операций присваивания:

- *простое присваивание*, в котором значение второго операнда хранится в объекте, указанном первым операндом.
- *составное присваивание*, в котором выполняется арифметическая операция сдвига или Побитовая перед сохранением результата.

Все операторы присваивания в следующей таблице, за исключением = оператора, являются составными операторами присваивания.

Таблица операторов присваивания

ОПЕРАТОР	ЗНАЧЕНИЕ
=	Сохранение значения второго операнда в объект, указанный первым операндом (простое присваивание).
*=	Умножение значения первого операнда на значение второго операнда; сохранение результата в объект, указанный первым операндом.
/=	Деление значения первого операнда на значение второго операнда; сохранение результата в объект, указанный первым операндом.
%=	деление по модулю первого операнда на значение второго операнда; сохранение результата в объект, указанный первым операндом.
+=	Сложение значения первого операнда со значением второго операнда; сохранение результата в объект, указанный первым операндом.
-=	Вычитание значения второго операнда из значения первого операнда; сохранение результата в объект, указанный первым операндом.

ОПЕРАТОР	ЗНАЧЕНИЕ
<code><<=</code>	Сдвиг значения первого операнда влево на количество битов, заданное значением второго операнда; сохранение результата в объект, указанный первым операндом.
<code>>>=</code>	Сдвиг значения первого операнда вправо на количество битов, заданное значением второго операнда; сохранение результата в объект, указанный первым операндом.
<code>&=</code>	Выполнение операции побитового И для значений первого и второго operandов; сохранение результата в объект, указанный первым операндом.
<code>^=</code>	Выполнение операции побитового исключающего ИЛИ для значений первого и второго operandов; сохранение результата в объект, указанный первым операндом.
<code> =</code>	Выполнение операции побитового включающего ИЛИ для значений первого и второго operandов; сохранение результата в объект, указанный первым операндом.

Ключевые слова операторов

Три составных оператора присваивания имеют эквиваленты ключевого слова. К ним относятся:

ОПЕРАТОР	ЭКВИВАЛЕНТНЫЙ
<code>&=</code>	<code>and_eq</code>
<code> =</code>	<code>or_eq</code>
<code>^=</code>	<code>xor_eq</code>

C++ задает эти ключевые слова операторов в качестве альтернативного написания для составных операторов присваивания. В языке С альтернативное написание предоставляется в виде макросов в `<iso646.h>` заголовке. В C++ альтернативные слова являются ключевыми словами; использование `<iso646.h>` или эквивалент C++ `<ciso646>` не рекомендуется. В Microsoft C++ `/permissive-` `/Za` параметр компилятора или необходим для включения альтернативного написания.

Пример

```
// expre_Assignment_Operators.cpp
// compile with: /EHsc
// Demonstrate assignment operators
#include <iostream>
using namespace std;
int main() {
    int a = 3, b = 6, c = 10, d = 0xAAAA, e = 0x5555;

    a += b;      // a is 9
    b %= a;      // b is 6
    c >>= 1;     // c is 5
    d |= e;      // Bitwise--d is 0xFFFF

    cout << "a = 3, b = 6, c = 10, d = 0xAAAA, e = 0x5555" << endl
        << "a += b yields " << a << endl
        << "b %= a yields " << b << endl
        << "c >>= 1 yields " << c << endl
        << "d |= e yields " << hex << d << endl;
}
```

Простое присваивание

Оператор простого присваивания (`=`) вызывает сохранение значения второго операнда в объекте, указанном первым операндом. Если оба объекта имеют арифметические типы, правый operand преобразуется в тип слева перед сохранением значения.

Объекты `const` и `volatile` типов могут быть назначены l-значениям только типов `volatile`, а не `const` или `volatile`.

Присваивание объектам типа класса (`struct` типы, `union` и `class`) выполняется функцией с именем `operator=`. По умолчанию эта функция-оператор производит побитовое копирование; однако такое поведение можно изменить с помощью перегруженных операторов. Для получения дополнительной информации см. раздел [Перегрузка операторов](#). Типы классов также могут иметь операторы [присваивания и перемещения](#) копирования. Дополнительные сведения см. в разделе [конструкторы копирования и операторы присваивания копирования](#) и [конструкторы перемещения и операторы присваивания перемещения](#).

Объект любого класса, однозначно производного от некоторого базового класса, можно присвоить объекту этого базового класса. Обратный переход не выполняется, поскольку существует неявное преобразование из производного класса в базовый класс, но не из базового класса в производный класс.
Пример:

```

// expre_SimpleAssignment.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
class ABase
{
public:
    ABase() { cout << "constructing ABase\n"; }
};

class ADerived : public ABase
{
public:
    ADerived() { cout << "constructing ADerived\n"; }
};

int main()
{
    ABase aBase;
    ADerived aDerived;

    aBase = aDerived; // OK
    aDerived = aBase; // C2679
}

```

Присваивание ссылочным типам выполняется так, как если бы выполнялось присваивание объекту, на который указывает ссылка.

Для объектов типа класса присваивание отличается от инициализации. Для иллюстрации того, насколько сильно присваивание может отличаться от инициализации, рассмотрим код

```

UserType1 A;
UserType2 B = A;

```

В предыдущем коде показан инициализатор; он вызывает конструктор для типа `UserType2`, который принимает аргумент типа `UserType1`. В коде

```

UserType1 A;
UserType2 B;

B = A;

```

оператор присваивания

```
B = A;
```

может вызывать одно из указанных ниже действий.

- Вызовите функцию `operator=` для `UserType2`, предоставленную `operator=` с `UserType1` аргументом.
- Вызывать функцию явного преобразования `UserType1::operator UserType2`, если такая функция существует.
- Вызывать конструктор `UserType2::UserType2`, если он существует, принимает аргумент `UserType1` и копирует результат.

Составное присваивание

Составные операторы присваивания показаны в [таблице операторы назначения](#). Эти операторы имеют форму $E1 \text{ Op} = E2$, где $E1$ — это `const` неизменяемое l-значение, а $E2$ —:

- арифметический тип
- указатель, если Op имеет значение `+` или `-`

Форма $E1 \text{ Op} = E2$ ведет себя как $E1 = E1 \text{ Op} E2$, но $E1$ вычисляется только один раз.

Составное присваивание перечисляемому типу создает сообщение об ошибке. Если левый operand имеет тип указателя, правый operand должен иметь тип указателя или константное выражение, результатом которого является 0. Если левый operand имеет целочисленный тип, правый operand не должен иметь тип указателя.

Результат операторов присваивания

Операторы присваивания возвращают значение объекта, указанного левым operandом после присваивания. Результирующий тип — это тип левого operandана. Результатом выражения присваивания всегда является l-значение. Эти операторы имеют ассоциативность справа налево. Левый operand должен быть изменяемым l-значением.

В ANSI C результат выражения присваивания не является l-значением. Это означает, что юридическое выражение $C++ (a += b) += c$ не допускается в C.

См. также

[Выражения с бинарными операторами](#)

[Встроенные операторы, приоритет и ассоциативность C++](#)

[Операторы присваивания в C](#)

Оператор побитового и: &

12.11.2021 • 2 minutes to read

Синтаксис

выражение & выражение

Remarks

Оператор побитового и (`&`) сравнивает каждый бит первого операнда с соответствующим битом второго операнда. Если оба бита равны 1, соответствующий бит результата устанавливается равным единице. в противном случае — нулю.

Оба операнда для оператора побитового и должны иметь целочисленные типы. Обычные арифметические преобразования, охваченные [стандартными](#) преобразованиями, применяются к операндам.

Ключевое слово оператора для &

C++ указывает на `bitand` альтернативное написание для `&`. В языке C в качестве макроса в заголовке указывается альтернативное написание `<iso646.h>`. В C++ альтернативным написанием является ключевое слово; использование `<iso646.h>` или эквивалент C++ `<ciso646>` не рекомендуется. В Microsoft C++ `/permissive-` `/Za` параметр компилятора или необходим для включения альтернативного написания.

Пример

```
// expre_Bitwise_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise AND
#include <iostream>
using namespace std;
int main() {
    unsigned short a = 0xFFFF;      // pattern 1111 ...
    unsigned short b = 0xAAAA;      // pattern 1010 ...

    cout << hex << ( a & b ) << endl;   // prints "aaaa", pattern 1010 ...
}
```

См. также

[Встроенные операторы, приоритет и ассоциативность C++](#)

[Побитовые операторы в C](#)

Оператор побитового исключающего ИЛИ: ^

12.11.2021 • 2 minutes to read

Синтаксис

выражение \wedge выражение

Remarks

Оператор побитового исключающего или (\wedge) сравнивает каждый бит своего первого операнда с соответствующим битом второго операнда. Если бит одного из operandов равен 0, а бит второго операнда равен 1, соответствующий бит результата устанавливается в значение 1. в противном случае — нулю.

Оба операнда оператора должны иметь целочисленные типы. Обычные арифметические преобразования, охваченные [стандартными](#) преобразованиями, применяются к operandам.

Дополнительные сведения об альтернативном использовании \wedge символа в c++/CLI и c++/CX см. в разделе [оператор Handle to Object \(^\) \(c++/CLI и c++/CX\)](#).

Ключевое слово оператора для ^

C++ указывает на `xor` альтернативное написание для \wedge . В языке C в качестве макроса в заголовке указывается альтернативное написание `<iso646.h>`. В C++ альтернативным написанием является ключевое слово; использование `<iso646.h>` или эквивалент C++ `<ciso646>` не рекомендуется. В Microsoft C++ `/permissive-` `/Za` параметр компилятора или необходим для включения альтернативного написания.

Пример

```
// expr_Bitwise_Exclusive_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise exclusive OR
#include <iostream>
using namespace std;
int main() {
    unsigned short a = 0x5555;      // pattern 0101 ...
    unsigned short b = 0xFFFF;      // pattern 1111 ...

    cout << hex << ( a ^ b ) << endl;   // prints "aaaa" pattern 1010 ...
}
```

См. также раздел

[Встроенные операторы, приоритет и ассоциативность C++](#)

Оператор побитового инклюзивного или: |

12.11.2021 • 2 minutes to read

Синтаксис

`expression1 | expression2`

Remarks

Оператор побитового включающего или (|) сравнивает каждый бит своего первого операнда с соответствующим битом второго операнда. Если любой из битов равен единице, соответствующий бит результата устанавливается равным единице, а в противном случае — нулю.

Оба операнда оператора должны иметь целочисленные типы. Обычные арифметические преобразования, охваченные [стандартными](#) преобразованиями, применяются к operandам.

Ключевое слово оператора для |

C++ указывает на `bitor` альтернативное написание для | . В языке C в качестве макроса в заголовке указывается альтернативное написание `<iso646.h>` . В C++ альтернативным написанием является ключевое слово; использование `<iso646.h>` или эквивалент C++ `<ciso646>` не рекомендуется. В Microsoft C++ `/permissive-` `/Za` параметр компилятора или необходим для включения альтернативного написания.

Пример

```
// expre_Bitwise_Inclusive_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate bitwise inclusive OR
#include <iostream>
using namespace std;

int main() {
    unsigned short a = 0x5555;      // pattern 0101 ...
    unsigned short b = 0xAAAA;      // pattern 1010 ...

    cout << hex << ( a | b ) << endl;   // prints "ffff" pattern 1111 ...
}
```

См. также раздел

[Встроенные операторы, приоритет и ассоциативность C++](#)

[Побитовые операторы в C](#)

Оператор Cast: ()

12.11.2021 • 2 minutes to read

Приведение типа позволяет выполнить явное преобразование типа объекта в конкретной ситуации.

Синтаксис

```
unary-expression ( type-name ) cast-expression
```

Remarks

Любое унарное выражение считается выражением приведения.

После выполнения приведения типа компилятор считает, что *cast-expression* имеет тип *type-name*.

Приведение типов можно использовать для преобразования объектов любого скалярного типа в любой другой скалярный тип и из любого другого скалярного типа. Явное приведение типов ограничено теми же правилами, которые определяют эффекты неявных преобразований. Дополнительные ограничения на операции приведения могут быть связаны с фактическими размерами или представлением конкретных типов.

Примеры

```
// expre_CastOperator.cpp
// compile with: /EHsc
// Demonstrate cast operator
#include <iostream>

using namespace std;

int main()
{
    double x = 3.1;
    int i;
    cout << "x = " << x << endl;
    i = (int)x;    // assign i the integer part of x
    cout << "i = " << i << endl;
}
```

```

// expre_CastOperator2.cpp
// The following sample shows how to define and use a cast operator.
#include <string.h>
#include <stdio.h>

class CountedAnsiString
{
public:
    // Assume source is not null terminated
    CountedAnsiString(const char *pStr, size_t nSize) :
        m_nSize(nSize)
    {
        m_pStr = new char[sizeOfBuffer];

        strncpy_s(m_pStr, sizeOfBuffer, pStr, m_nSize);
        memset(&m_pStr[m_nSize], '!', 9); // for demonstration purposes.
    }

    // Various string-like methods...

    const char *GetRawBytes() const
    {
        return(m_pStr);
    }

    //
    // operator to cast to a const char *
    //
    operator const char *()
    {
        m_pStr[m_nSize] = '\0';
        return(m_pStr);
    }

    enum
    {
        sizeOfBuffer = 20
    } size;

private:
    char *m_pStr;
    const size_t m_nSize;
};

int main()
{
    const char *kStr = "Excitinggg";
    CountedAnsiString myStr(kStr, 8);

    const char *pRaw = myStr.GetRawBytes();
    printf_s("RawBytes truncated to 10 chars:  %.10s\n", pRaw);

    const char *pCast = (const char *)myStr;
    printf_s("Casted Bytes:  %s\n", pCast);

    puts("Note that the cast changed the raw internal string");
    printf_s("Raw Bytes after cast:  %s\n", pRaw);
}

```

```

RawBytes truncated to 10 chars:  Exciting!!
Casted Bytes:  Exciting
Note that the cast changed the raw internal string
Raw Bytes after cast:  Exciting

```

См. также

[Выражения с унарными операторами](#)

[Операторы C++, приоритет и ассоциативность](#)

[Оператор явного преобразования типа: \(\)](#)

[Операторы приведения](#)

[Операторы приведения](#)

Оператор "запятая": ,

12.11.2021 • 2 minutes to read

Позволяет группировать два оператора, когда ожидается один.

Синтаксис

```
expression , expression
```

Remarks

Оператор-запятая имеет ассоциативность слева направо. Два выражения, разделенные запятой, вычисляются в направлении слева направо. Левый operand вычисляется всегда и перед вычислением правого операнда учитываются все побочные эффекты.

В некоторых контекстах, например в списках аргументов функций, в качестве разделителей можно использовать запятые. Не следует путать использование запятой в качестве разделителя с ее использованием в качестве оператора; эти два варианта использования совершенно различны.

Рассмотрим выражение `e1, e2`. Тип и значение выражения являются типом и значением `E2`; результат вычисления `E1` отбрасывается. Если правый operand — L-значение, результатом будет L-значение.

В тех случаях, когда в качестве разделителя обычно используется запятая (например, в фактических аргументах для функций или составных инициализаторов), оператор-запятую и его operandы следует заключать в скобки. Пример:

```
func_one( x, y + 2, z );
func_two( (x--, y + 2), z );
```

В приведенном выше вызове функции `func_one` передаются три аргумента, разделенные запятыми: `x`, `y + 2` и `z`. В вызове функции `func_two` скобки заставляют компилятор интерпретировать первую запятую в качестве оператора последовательного вычисления. В этом вызове функции в функцию `func_two` передаются два аргумента. Первым аргументом является результат операции последовательного вычисления `(x--, y + 2)`, который имеет значение и тип выражения `y + 2`; второй аргумент — `z`.

Пример

```
// cpp_comma_operator.cpp
#include <stdio.h>
int main () {
    int i = 10, b = 20, c= 30;
    i = b, c;
    printf("%i\n", i);

    i = (b, c);
    printf("%i\n", i);
}
```

См. также

[Выражения с бинарными операторами](#)

[Операторы C++, приоритет и ассоциативность](#)

[Оператор последовательного вычисления](#)

Условный оператор: ?: :

12.11.2021 • 2 minutes to read

Синтаксис

```
expression ? expression : expression
```

Remarks

Условный оператор (?:) является оператором ternary (он принимает три операнда). Условный оператор работает следующим образом.

- Первый operand неявно преобразуется в `bool`. Он вычисляется, и все побочные эффекты завершаются перед продолжением.
- Если первый operand принимает значение `true` (1), вычисляется второй operand.
- Если первый operand принимает значение `false` (0), вычисляется третий operand.

Результатом условного оператора является оцененного операнда — второго или третьего. В условном выражении оценивается только один из последних двух operandов.

Условные выражения имеют ассоциативность справа налево. Первый operand должен иметь целочисленный тип или тип указателя. Следующие правила применяются ко второму и третьему operandам.

- Если оба operandы имеют один и тот же тип, результат имеет тот же тип.
- Если оба operandы имеют арифметические или перечисляемые типы, то для их преобразования в общий тип выполняются обычные арифметические преобразования (охваченные [стандартными преобразованиями](#)).
- Если оба operandы имеют тип указателя или один operand относится к типу указателя, а другой является выражением константы со значением 0, преобразования указателя выполняются с целью преобразования их к общему типу.
- Если оба operandы имеют ссылочные типы, для преобразования их в общий тип используются ссылочные преобразования.
- Если оба operandы имеют тип `void`, общий тип также имеет тип `void`.
- Если оба operandы относятся к одному определяемому пользователем типу, общий тип также относится к этому типу.
- Если operandы относятся к разным типам и по крайней мере один из operandов относится к определяемому пользователем типу, для определения общего типа используются правила языка (см. предупреждение ниже).

Какие-либо сочетания второго и третьего operandов, отсутствующие в предыдущем списке, недопустимы. Тип результата — это общий тип и l-значение, если и второй, и третий operandы имеют один и тот же тип и представляют собой l-значения.

WARNING

Если типы второго и третьего operandов не идентичны, вызываются правила преобразования сложных типов в соответствии со стандартом C++. Эти преобразования могут привести к непредвиденному поведению, включая создание и удаление временных объектов. По этой причине мы настоятельно рекомендуем вам (1) либо избегать использования определяемых пользователем типов в качестве operandов в условных операторах, (2) либо, если определяемые пользователем типы все же используются, явно приводить каждый operand к общему типу.

Пример

```
// expe_Expressions_with_the_Conditional_Operator.cpp
// compile with: /EHsc
// Demonstrate conditional operator
#include <iostream>
using namespace std;
int main() {
    int i = 1, j = 2;
    cout << ( i > j ? i : j ) << " is greater." << endl;
}
```

См. также

[Операторы C++, приоритет и ассоциативность](#)

[Оператор Conditional-Expression](#)

Оператор delete (C++)

12.11.2021 • 2 minutes to read

Отменяет выделение блока памяти.

Синтаксис

```
[ :: ] delete приведение выражения  
[ :: ] delete [] приведение выражения
```

Remarks

Аргумент *Cast-Expression* должен быть указателем на блок памяти, выделенный ранее для объекта, созданного с помощью [оператора New](#). `delete` Оператор имеет результат типа `void` и поэтому не возвращает значение. Пример:

```
CDialog* MyDialog = new CDialog;  
// use MyDialog  
delete MyDialog;
```

Использование `delete` в указателе на объект, не выделенный с помощью `new`, дает непредсказуемые результаты. Однако можно использовать `delete` для указателя со значением 0. Такая инициализация означает, что если `new` в случае сбоя возвращает значение 0, Удаление результата неудачной `new` операции является безвредным. Дополнительные сведения см. в [разделе операторы new и DELETE](#).

`new` Операторы и `delete` можно также использовать для встроенных типов, включая массивы. Если `pointer` ссылается на массив, поместите пустые скобки (`[]`) перед `pointer`:

```
int* set = new int[100];  
//use set[]  
delete [] set;
```

Использование `delete` оператора для объекта освобождает его память. Программа, которая разыменовывает указатель после удаления объекта, может создать непрогнозируемый результат или вызвать сбой.

Если `delete` используется для освобождения памяти для объекта класса C++, деструктор объекта вызывается до освобождения памяти объекта (если у объекта есть деструктор).

Если operand `delete` оператора является изменяемым l-значением, его значение не определено после удаления объекта.

Если указан параметр компилятора [/SDL](#) (включить дополнительные проверки безопасности), операнду оператора присваивается `delete` недопустимое значение после удаления объекта.

Использование оператора delete

Существует два синтаксических варианта для [оператора delete](#): один для единичных объектов, а другой для массивов объектов. В следующем фрагменте кода показано, как они отличаются:

```
// expre_Using_delete.cpp
struct UDType
{
};

int main()
{
    // Allocate a user-defined object, UDObject, and an object
    // of type double on the free store using the
    // new operator.
    UDType *UDObject = new UDType;
    double *dObject = new double;
    // Delete the two objects.
    delete UDObject;
    delete dObject;
    // Allocate an array of user-defined objects on the
    // free store using the new operator.
    UDType (*UDArr)[7] = new UDType[5][7];
    // Use the array syntax to delete the array of objects.
    delete [] UDArr;
}
```

Следующие два варианта приводят к неопределенным результатам: использование формы массива `delete` (`delete []`) для объекта и использование немассивного класса `DELETE` в массиве.

Пример

Примеры использования см `delete` . в разделе [оператор New](#).

Принцип работы `delete`

Оператор `delete` вызывает **оператор функции Delete**.

Для объектов, не имеющих тип класса ([класс, Структура или объединение](#)), вызывается оператор глобального удаления. Для объектов типа класса имя функции освобождения разрешается в глобальной области, если выражение `delete` начинается с оператора унарного разрешения области (`::`). В противном случае перед освобождением памяти оператор удаления вызывает деструктор объекта (если указатель не имеет значения `null`). Оператор удаления можно определять отдельно для каждого класса; если для некоторого класса такое определение отсутствует, вызывается глобальный оператор удаления. Если выражение удаления используется для освобождения объекта класса, статический тип которого имеет виртуальный деструктор, функция освобождение разрешается через виртуальный деструктор динамического типа объекта.

См. также

[Выражения с унарными операторами](#)

[Словами](#)

[Операторы создания и удаления](#)

Операторы равенства: == и !=

12.11.2021 • 2 minutes to read

Синтаксис

```
выражение == выражение
выражение != выражение
```

Remarks

Бинарные операторы равенства сравнивают operandы для строгого равенства или неравенства.

Операторы равенства, равные (==) и не равные (!=), имеют более низкий приоритет, чем операторы отношения, но они ведут себя одинаково. Тип результата для этих операторов — bool .

Оператор Equal-to (==) возвращает true , если оба операнда имеют одинаковое значение; в противном случае возвращается false . Оператор "не равно" (!=) возвращает, true . Если operandы имеют одинаковое значение; в противном случае возвращается false .

Ключевое слово оператора для !=

C++ указывает на not_eq альтернативное написание для != . (Альтернативное написание для не предусмотрено == .) В языке C в качестве макроса в заголовке указывается альтернативное написание <iso646.h> . В C++ альтернативным написанием является ключевое слово; использование <iso646.h> или эквивалент C++ <ciso646> не рекомендуется. В Microsoft C++ /permissive- /Za параметр компилятора или необходим для включения альтернативного написания.

Пример

```
// expre_Equality_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main() {
    cout << boolalpha
        << "The true expression 3 != 2 yields: "
        << (3 != 2) << endl
        << "The false expression 20 == 10 yields: "
        << (20 == 10) << endl;
}
```

Операторы равенства могут сравнивать указатели на члены одного типа. В таких сравнениях выполняются преобразования указателей на члены. Указатели на члены также можно сравнить с константным выражением, результатом которого является значение 0.

См. также раздел

[Выражения с бинарными операторами](#)

[Встроенные операторы C++, приоритет; и ассоциативность](#)

Операторы отношения и равенства C

Оператор явного преобразования типа: ()

12.11.2021 • 2 minutes to read

В C++ разрешаются явные преобразования типов с использованием синтаксиса, аналогичного синтаксису вызова функции.

Синтаксис

```
simple-type-name ( expression-list )
```

Remarks

Простое-Type-Name, за которым следует *список выражений*, заключенный в круглые скобки, конструирует объект указанного типа с помощью указанных выражений. В следующем примере показано явное преобразование типа в тип int.

```
int i = int( d );
```

В следующем примере показан [Point](#) класс.

Пример

```

// expre_Explicit_Type_Conversion_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Point
{
public:
    // Define default constructor.
    Point() { _x = _y = 0; }
    // Define another constructor.
    Point( int X, int Y ) { _x = X; _y = Y; }

    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
    unsigned& y() { return _y; }
    void Show() { cout << "x = " << _x << ", "
                  << "y = " << _y << "\n"; }

private:
    unsigned _x;
    unsigned _y;
};

int main()
{
    Point Point1, Point2;

    // Assign Point1 the explicit conversion
    // of ( 10, 10 ).
    Point1 = Point( 10, 10 );

    // Use x() as an l-value by assigning an explicit
    // conversion of 20 to type unsigned.
    Point1.x() = unsigned( 20 );
    Point1.Show();

    // Assign Point2 the default Point object.
    Point2 = Point();
    Point2.Show();
}

```

Выходные данные

```

x = 20, y = 10
x = 0, y = 0

```

Хотя в предыдущем примере показано явное преобразование типов с помощью констант, тот же метод применим и для выполнения таких преобразований для объектов. Это демонстрируется в следующем фрагменте кода.

```

int i = 7;
float d;

d = float( i );

```

Явные преобразования типов также можно задавать с помощью синтаксиса приведения. Предыдущий пример, перезаписанный с использованием синтаксиса приведения, выглядит следующим образом:

```
d = (float)i;
```

Преобразования отдельных значений в стиле приведения и в стиле функции дают одинаковые результаты. Однако в синтаксисе стиля функции можно определить несколько аргументов для преобразования. Это различие имеет важное значение для пользовательских типов. Рассмотрим класс `Point` и его преобразования.

```
struct Point
{
    Point( short x, short y ) { _x = x; _y = y; }
    ...
    short _x, _y;
};

...
Point pt = Point( 3, 10 );
```

В предыдущем примере, в котором используется преобразование в стиле функции, показано, как преобразовать два значения (один для *x* и один для *y*) в определяемый пользователем тип `Point`.

Caution

Явные преобразования типов следует использовать с осторожностью, поскольку они переопределяют встроенную проверку типов компилятора C++.

Нотация [приведения](#) должна использоваться для преобразований в типы, для которых не заданы простые типы с [именами](#) (например, указатели или ссылки). Преобразование в типы, которые могут быть выражены [простым-типом-именем](#), может быть записано в любой форме.

Определение типа в приведениях недопустимо.

См. также

[Постфиксные выражения](#)

[Операторы C++, приоритет и ассоциативность](#)

Оператор вызова функции: ()

12.11.2021 • 3 minutes to read

Вызов функции — это разновидность `postfix-expression`, сформированная выражением, результатом которого является функция или вызываемый объект, а затем оператор вызова функции `()`. Объект может объявить `operator ()` функцию, которая предоставляет семантику вызова функции для объекта.

Синтаксис

```
postfix-expression :  
    postfix-expression ( argument-expression-list ) opt )
```

Remarks

Аргументы оператора вызова функции берутся из `argument-expression-list`, разделенного запятыми списка выражений. Значения этих выражений передаются в функцию в качестве аргументов. `Argument-expression-list` может быть пустым. До C++ 17 порядок вычисления выражения функции и выражений аргументов не определен и может возникать в любом порядке. В C++ 17 и более поздних версиях выражение функции вычисляется перед любыми выражениями аргументов или аргументами по умолчанию. Выражения аргументов вычисляются в неопределенной последовательности.

`postfix-expression` Вычисляет функцию для вызова. Он может принимать любое из нескольких форм:

- Идентификатор функции, видимый в текущей области или в области любого предоставленного аргумента функции;
- выражение, результатом которого является функция, указатель функции, вызываемый объект или ссылка на один,
- метод доступа к функции-члену, явный или подразумеваемый
- Указатель на разыменование функции-члена.

`postfix-expression` Может быть перегруженный идентификатор функции или метод доступа перегруженной функции члена. Правила для разрешения перегрузки определяют фактическую функцию для вызова. Если функция-член является виртуальной, вызываемая функция определяется во время выполнения.

Некоторые примеры объявлений:

- Функция, возвращающая тип `T`. Пример объявления:

```
T func( int i );
```

- Указатель на функцию, возвращающую тип `T`. Пример объявления:

```
T (*func)( int i );
```

- Ссылка на функцию, возвращающую тип `T`. Пример объявления:

```
T (&func)(int i);
```

- Разыменование функции указателя на член, возвращающее тип `t`. Примеры вызовов функции:

```
(pObject->*pmf)();  
(Object.*pmf)();
```

Пример

В следующем примере вызывается функция стандартной библиотеки `strcat_s` с тремя аргументами:

```
// expr_Expression_Call_Operator.cpp  
// compile with: /EHsc  
  
#include <iostream>  
#include <string>  
  
// C++ Standard Library name space  
using namespace std;  
  
int main()  
{  
    enum  
    {  
        sizeOfBuffer = 20  
    };  
  
    char s1[ sizeOfBuffer ] = "Welcome to ";  
    char s2[ ] = "C++";  
  
    strcat_s( s1, sizeOfBuffer, s2 );  
  
    cout << s1 << endl;  
}
```

```
Welcome to C++
```

Результаты вызова функции

Вызов функции вычисляет значение `rvalue`, если только функция не объявлена как ссылочный тип. Функции с возвращаемыми ссылочными типами оцениваются как значения `lvalue`. Эти функции можно использовать в левой части оператора присваивания, как показано ниже:

```
// expre_Function_Call_Results.cpp
// compile with: /EHsc
#include <iostream>
class Point
{
public:
    // Define "accessor" functions as
    // reference types.
    unsigned& x() { return _x; }
    unsigned& y() { return _y; }
private:
    unsigned _x;
    unsigned _y;
};

using namespace std;
int main()
{
    Point ThePoint;

    ThePoint.x() = 7;           // Use x() as an l-value.
    unsigned y = ThePoint.y(); // Use y() as an r-value.

    // Use x() and y() as r-values.
    cout << "x = " << ThePoint.x() << "\n"
        << "y = " << ThePoint.y() << "\n";
}
```

Приведенный выше код определяет класс с именем `Point`, который содержит закрытые объекты данных, представляющие координаты *x* и *y*. Эти объекты данных необходимо изменить, а значения — извлечь. Программа — это лишь одно из нескольких решений для такого класса. Также можно использовать функции `GetX` и `SetX` или `GetY` и `SetY`.

Функции, возвращающие типы классов, указатели на типы классов или ссылки на типы классов можно использовать как левый operand в операторах выбора члена. Следующий код является допустимым:

```
// expre_Function_Results2.cpp
class A {
public:
    A() {}
    A(int i) {}
    int SetA( int i ) {
        return (I = i);
    }

    int GetA() {
        return I;
    }

private:
    int I;
};

A func1() {
    A a = 0;
    return a;
}

A* func2() {
    A *a = new A();
    return a;
}

A& func3() {
    A *a = new A();
    A &b = *a;
    return b;
}

int main() {
    int iResult = func1().GetA();
    func2()->SetA( 3 );
    func3().SetA( 7 );
}
```

Функции можно вызывать рекурсивно. Дополнительные сведения об объявлениях функций см. в разделе [функции](#). Связанные материалы относятся к единицам трансляции и компоновке.

См. также раздел

[Постфиксные выражения](#)

[Встроенные операторы, приоритет и ассоциативность C++](#)

[Вызов функции](#)

Оператор косвенного обращения: *

12.11.2021 • 2 minutes to read

Синтаксис

```
* cast-expression
```

Remarks

Унарный оператор косвенного обращения (*) отменяет ссылку на указатель, то есть преобразует значение указателя в l-значение. Операнд косвенного оператора должен быть указателем на тип. Результат косвенного выражения — тип, производным которого является тип указателя. Использование * оператора в этом контексте отличается от его значения бинарным оператором, который является умножением.

Если operand указывает на функцию, результатом является указатель функции. Если он указывает на место хранения, результатом является l-значение, указывающее на место хранения.

Косвенный оператор может использоваться кумулятивно для разыменования указателей на указатели.
Пример:

```
// expre_Indirection_Operator.cpp
// compile with: /EHsc
// Demonstrate indirection operator
#include <iostream>
using namespace std;
int main() {
    int n = 5;
    int *pn = &n;
    int **ppn = &pn;

    cout << "Value of n:\n"
        << "direct value: " << n << endl
        << "indirect value: " << *pn << endl
        << "doubly indirect value: " << **ppn << endl
        << "address of n: " << pn << endl
        << "address of n via indirection: " << *ppn << endl;
}
```

Если значение указателя недопустимо, результат становится неопределенным. Ниже приведены наиболее распространенные условия, при которых значение указателя становится недопустимым.

- Указатель является пустым указателем.
- Указатель определяет адрес локального элемента, не видимого во время обращения.
- Указатель определяет адрес, выравнивание которого не подходит для типа указываемого объекта.
- Указатель определяет адрес, который не используется выполняемой программой.

См. также

[Выражения с унарными операторами](#)

[Операторы C++, приоритет и ассоциативность](#)

Оператор address-of: &

Операторы косвенного обращения и адреса операнда

Операторы сдвига влево и вправо (`>>` и `<<`)

12.11.2021 • 5 minutes to read

Операторы побитового сдвига являются оператором сдвига вправо (`>>`), который перемещает биты *сдвига* вправо и оператор сдвига влево (`<<`), который перемещает биты *SHIFT-Expression* влево.¹

Синтаксис

```
SHIFT-выражение << аддитивное выражение  
shift-expression >> additive-expression
```

Remarks

IMPORTANT

следующие описания и примеры допустимы в Windows для архитектур x86 и x64. реализация операторов сдвига влево и сдвига вправо существенно отличается в Windows для устройств ARM. Дополнительные сведения см. в разделе "операторы сдвига" в записи блога [Hello ARM](#).

Сдвиги влево

Оператор сдвига влево приводит к тому, что биты в *выражении сдвига* сдвигаются влево на число позиций, заданное *аддитивным выражением*. Позиции битов, освобожденные при операции сдвига, заполняются нулями. Сдвиг влево является логическим сдвигом (биты, сдвигаемые с конца отбрасываются, включая бит знака). Дополнительные сведения о типах побитовых сдвигов см. в разделе [побитовые сдвиги](#).

В следующем примере показаны операции сдвига влево с использованием чисел без знака. В этом примере показано, что происходит с битами при представлении значения как `bitset`. Дополнительные сведения см. в разделе [класс битовом массиве](#).

```
#include <iostream>  
#include <bitset>  
  
using namespace std;  
  
int main() {  
    unsigned short short1 = 4;  
    bitset<16> bitset1{short1}; // the bitset representation of 4  
    cout << bitset1 << endl; // 0b00000000'00001000  
  
    unsigned short short2 = short1 << 1; // 4 left-shifted by 1 = 8  
    bitset<16> bitset2{short2};  
    cout << bitset2 << endl; // 0b00000000'00001000  
  
    unsigned short short3 = short1 << 2; // 4 left-shifted by 2 = 16  
    bitset<16> bitset3{short3};  
    cout << bitset3 << endl; // 0b00000000'00010000  
}
```

Если выполняется сдвиг влево числа со знаком и при этом затрагивается бит знака, результат не определен. В следующем примере показано, что происходит при сдвиге 1-й бита на разряд знака.

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short short1 = 16384;
    bitset<16> bitset1(short1);
    cout << bitset1 << endl; // 0b01000000'00000000

    short short3 = short1 << 1;
    bitset<16> bitset3(short3); // 16384 left-shifted by 1 = -32768
    cout << bitset3 << endl; // 0b10000000'00000000

    short short4 = short1 << 14;
    bitset<16> bitset4(short4); // 4 left-shifted by 14 = 0
    cout << bitset4 << endl; // 0b00000000'00000000
}

```

Сдвиги вправо

Оператор сдвига вправо вызывает сдвиг битового шаблона в выражении *сдвига вправо* на число позиций, заданное *аддитивным выражением*. Для чисел без знака позиции битов, освобожденные при операции сдвига, заполняются нулями. Для чисел со знаком бит знака используется для заполнения освобожденных позиций битов. Другими словами, если число является положительным, используется 0, если число является отрицательным, используется 1.

IMPORTANT

Результат сдвига вправо отрицательного числа со знаком зависит от реализации. Несмотря на то, что компилятор Microsoft C++ использует бит знака для заполнения освобожденных битовых позиций, нет никакой гарантии, что это также делается для других реализаций.

В следующем примере показаны операции сдвига вправо с использованием чисел без знака.

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned short short11 = 1024;
    bitset<16> bitset11{short11};
    cout << bitset11 << endl; // 0b00000100'00000000

    unsigned short short12 = short11 >> 1; // 512
    bitset<16> bitset12{short12};
    cout << bitset12 << endl; // 0b00000010'00000000

    unsigned short short13 = short11 >> 10; // 1
    bitset<16> bitset13{short13};
    cout << bitset13 << endl; // 0b00000000'00000001

    unsigned short short14 = short11 >> 11; // 0
    bitset<16> bitset14{short14};
    cout << bitset14 << endl; // 0b00000000'00000000
}

```

В следующем примере показаны операции сдвига вправо с использованием положительных чисел со

знаком.

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short short1 = 1024;
    bitset<16> bitset1(short1);
    cout << bitset1 << endl;      // 0b00000100'00000000

    short short2 = short1 >> 1;   // 512
    bitset<16> bitset2(short2);
    cout << bitset2 << endl;      // 0b00000010'00000000

    short short3 = short1 >> 11;  // 0
    bitset<16> bitset3(short3);
    cout << bitset3 << endl;      // 0b00000000'00000000
}
```

В следующем примере показаны операции сдвига вправо с использованием отрицательных целых чисел со знаком.

```
#include <iostream>
#include <bitset>

using namespace std;

int main() {
    short neg1 = -16;
    bitset<16> bn1(neg1);
    cout << bn1 << endl;      // 0b11111111'11110000

    short neg2 = neg1 >> 1; // -8
    bitset<16> bn2(neg2);
    cout << bn2 << endl;      // 0b11111111'11111000

    short neg3 = neg1 >> 2; // -4
    bitset<16> bn3(neg3);
    cout << bn3 << endl;      // 0b11111111'11111100

    short neg4 = neg1 >> 4; // -1
    bitset<16> bn4(neg4);
    cout << bn4 << endl;      // 0b11111111'11111111

    short neg5 = neg1 >> 5; // -1
    bitset<16> bn5(neg5);
    cout << bn5 << endl;      // 0b11111111'11111111
}
```

Сдвиги и перемещения

Выражения с обеих сторон оператора сдвига должны быть целочисленными типами. Целочисленные акции выполняются в соответствии с правилами, описанными в разделе [стандартные преобразования](#). Тип результата совпадает с типом повышенного выражения сдвига.

В следующем примере переменная типа `char` повышается до `int`.

```

#include <iostream>
#include <typeinfo>

using namespace std;

int main() {
    char char1 = 'a';

    auto promoted1 = char1 << 1; // 194
    cout << typeid(promoted1).name() << endl; // int

    auto promoted2 = char1 << 10; // 99328
    cout << typeid(promoted2).name() << endl; // int
}

```

Дополнительные сведения

Результат операции сдвига не определен, если *аддитивное выражение* является отрицательным или если *аддитивное выражение* больше или равно количеству битов в выражении сдвига (повышенной). Если *аддитивное выражение* имеет значение 0, то операция сдвига не выполняется.

```

#include <iostream>
#include <bitset>

using namespace std;

int main() {
    unsigned int int1 = 4;
    bitset<32> b1{int1};
    cout << b1 << endl; // 0b00000000'00000000'00000000'00000100

    unsigned int int2 = int1 << -3; // C4293: '<<' : shift count negative or too big, undefined behavior
    unsigned int int3 = int1 >> -3; // C4293: '>>' : shift count negative or too big, undefined behavior
    unsigned int int4 = int1 << 32; // C4293: '<<' : shift count negative or too big, undefined behavior
    unsigned int int5 = int1 >> 32; // C4293: '>>' : shift count negative or too big, undefined behavior
    unsigned int int6 = int1 << 0;
    bitset<32> b6{int6};
    cout << b6 << endl; // 0b00000000'00000000'00000000'00000100 (no change)
}

```

Примечания

¹ ниже приведено описание операторов сдвига в спецификации C++ 11 ISO (ИНЦИТС/ISO/IEC 14882-2011 [2012]), разделы 5.8.2 и 5.8.3.

Значение $E1 \ll E2$ представляет собой $E1$ со сдвигом влево на $E2$ позиций битов; освобожденные биты заполняются нулями. Если $E1$ имеет тип без знака, то результатом будет значение $E1 \times 2^{E2}$, но по меньшей мере остаток от деления превышает максимальное значение, которое может быть представлено в типе результата. В противном случае, если $E1$ имеет тип со знаком и неотрицательное значение, а $E1 \times 2^{E2}$ может быть представлено в соответствующем неподписанном типе результата, то это значение, преобразованное в тип результата, является результатирующим значением; в противном случае поведение не определено.

Значение $E1 \gg E2$ представляет собой $E1$ со сдвигом вправо на $E2$ позиций битов. Если $E1$ имеет тип без знака или если $E1$ имеет тип со знаком и неотрицательное значение, значение результата является неотъемлемой частью частного числа $E1/2^{E2}$. Если $E1$ имеет тип со знаком и отрицательное значение, значение результата определяется реализацией.

См. также

[Выражения с бинарными операторами](#)

[Операторы C++, приоритет и ассоциативность](#)

Логический оператор AND: &&

12.11.2021 • 2 minutes to read

Синтаксис

выражение && выражение

Remarks

Логический оператор AND (`&&`) возвращает `true` значение, если оба операнда являются `true` и возвращают `false` в противном случае. Перед вычислением operandы неявно преобразуются в тип `bool`, а результат имеет тип `bool`. Логическое И имеет ассоциативность в направлении слева направо.

Операнды логического оператора AND не должны иметь одинаковый тип, но они должны иметь тип Boolean, интеграл или указатель. В качестве operandов часто используются реляционные выражения и выражения равенства.

Первый operand полностью вычисляется и все побочные эффекты выполняются до того, как будет выполнено вычисление логического выражения и.

Второй operand вычисляется только в том случае, если первый operand принимает значение `true` (отличное от нуля). Эта оценка устраняет необходимость недостаточной оценки второго operand, если логическое выражение и имеет значение `false`. Такое сокращенное вычисление можно использовать для предотвращения разыменования пустого указателя, как показано в следующем примере.

```
char *pch = 0;  
// ...  
(pch) && (*pch = 'a');
```

Если `pch` имеет значение NULL (0), правая часть выражения никогда не вычисляется. Эта сокращенная оценка делает назначение через пустой указатель невозможным.

Ключевое слово оператора для &&

C++ указывает на `and` альтернативное написание для `&&`. В языке C в качестве макроса в заголовке указывается альтернативное написание `<iso646.h>`. В C++ альтернативным написанием является ключевое слово; использование `<iso646.h>` или эквивалент C++ `<ciso646>` не рекомендуется. В Microsoft C++ `/permissive-` `/Za` параметр компилятора или необходим для включения альтернативного написания.

Пример

```
// expre_Logical_AND_Operator.cpp
// compile with: /EHsc
// Demonstrate logical AND
#include <iostream>

using namespace std;

int main() {
    int a = 5, b = 10, c = 15;
    cout << boolalpha
        << "The true expression "
        << "a < b && b < c yields "
        << (a < b && b < c) << endl
        << "The false expression "
        << "a > b && b < c yields "
        << (a > b && b < c) << endl;
}
```

См. также раздел

[Встроенные операторы, приоритет и ассоциативность C++](#)

[Логические операторы в C](#)

Оператор логического отрицания: !

12.11.2021 • 2 minutes to read

Синтаксис

** ! ** приведение выражения

Remarks

Оператор логического отрицания (!) меняет значение операнда на противоположное. Операнд должен иметь арифметический тип или тип указателя (либо выражение, результатом которого является арифметический тип или тип указателя). Операнд неявно преобразуется в тип `bool`. Результат имеет значение, `true`. Если преобразованный операнд имеет значение `false`; результат равен, `false`. Если преобразованный операнд имеет значение `true`. Результат имеет тип `bool`.

Для выражения `e` унарное выражение `!e` эквивалентно выражению `(e == 0)`, за исключением случаев, когда задействованы перегруженные операторы.

Ключевое слово оператора для!

C++ указывает на `not` альтернативное написание для ! . В языке C в качестве макроса в заголовке указывается альтернативное написание `<iso646.h>` . В C++ альтернативным написанием является ключевое слово; использование `<iso646.h>` или эквивалент C++ `<ciso646>` не рекомендуется. В Microsoft C++ `/permissive-` `/za` параметр компилятора или необходим для включения альтернативного написания.

Пример

```
// expre_Logical_NOT_Operator.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main() {
    int i = 0;
    if (!i)
        cout << "i is zero" << endl;
}
```

См. также

[Выражения с унарными операторами](#)

[Встроенные операторы, приоритет и ассоциативность C++](#)

[Унарные арифметические операторы](#)

Оператор логического или: ||

12.11.2021 • 2 minutes to read

Синтаксис

логическое выражение или || логическое выражение

Remarks

Логический оператор или (||) возвращает логическое значение, true. Если один или оба операнда являются true и возвращают false в противном случае. Перед вычислением операнды неявно преобразуются в тип bool, а результат имеет тип bool. Логическое ИЛИ имеет ассоциативность в направлении слева направо.

Операнды логического оператора или не должны иметь одинаковый тип, но они должны иметь тип Boolean, интеграл или указатель. В качестве operandов часто используются реляционные выражения и выражения равенства.

В выражении логического ИЛИ сначала полностью вычисляется первый operand и учитываются все побочные эффекты, и лишь после этого вычисление продолжается.

Второй operand вычисляется только в том случае, если первый operand имеет значение false, так как вычисление не требуется, если логическое выражение или true. Она известна как *Сокращенная оценка*.

```
printf( "%d" , (x == w || x == y || x == z) );
```

В приведенном выше примере, если x равен w, у или z, второй аргумент printf функции возвращает true значение, которое затем передается в целое число, и печатается 1. В противном случае он вычисляется false и значение 0 выводится на печать. Как только одно из условий примет значение true, вычисление остановится.

Ключевое слово оператора для ||

C++ указывает на or альтернативное написание для ||. В языке C в качестве макроса в заголовке указывается альтернативное написание <iso646.h>. В C++ альтернативным написанием является ключевое слово; использование <iso646.h> или эквивалент C++ <ciso646> не рекомендуется. В Microsoft C++ /permissive- /Za параметр компилятора или необходим для включения альтернативного написания.

Пример

```
// expre_Logical_OR_Operator.cpp
// compile with: /EHsc
// Demonstrate logical OR
#include <iostream>
using namespace std;
int main() {
    int a = 5, b = 10, c = 15;
    cout << boolalpha
        << "The true expression "
        << "a < b || b > c yields "
        << (a < b || b > c) << endl
        << "The false expression "
        << "a > b || b > c yields "
        << (a > b || b > c) << endl;
}
```

См. также раздел

[Встроенные операторы, приоритет и ассоциативность C++](#)

[Логические операторы в C](#)

Операторы доступа к членам: перетаскивани>

12.11.2021 • 2 minutes to read

Синтаксис

```
postfix-expression . name  
postfix-expression -> name
```

Remarks

Операторы доступа к членам . и -> используются для ссылки на члены структур, объединений и классов. Выражения доступа к членам имеют значение и тип выбранного члена.

Предусмотрено две формы выражения доступа к члену:

1. В первой форме *постфиксное выражение* представляет значение структуры, класса или типа объединения, а *имя Name* является членом указанной структуры, объединения или класса. Значением операции является *имя* и является l-значением, если *постфикс-выражение* является l-значением.
2. Во второй форме *постфиксное выражение* представляет указатель на структуру, объединение или класс, а *имя Name* является членом указанной структуры, объединения или класса. Значение равно *имени* и является l-значением. Оператор отменяет -> ссылку на указатель. Таким образом, выражения `e->member` И `(*e).member` (где *e* представляет указатель) выдают идентичные результаты (за исключением случаев, когда операторы -> или * перегружены).

Пример

В следующем примере показаны обе формы оператора доступа к членам.

```
// expe_Selection_Operator.cpp  
// compile with: /EHsc  
#include <iostream>  
using namespace std;  
  
struct Date {  
    Date(int i, int j, int k) : day(i), month(j), year(k){}  
    int month;  
    int day;  
    int year;  
};  
  
int main() {  
    Date mydate(1,1,1900);  
    mydate.month = 2;  
    cout << mydate.month << "/" << mydate.day  
       << "/" << mydate.year << endl;  
  
    Date *mydate2 = new Date(1,1,2000);  
    mydate2->month = 2;  
    cout << mydate2->month << "/" << mydate2->day  
       << "/" << mydate2->year << endl;  
    delete mydate2;  
}
```

2/1/1900

2/1/2000

См. также

[Постфиксные выражения](#)

[Операторы C++, приоритет и ассоциативность](#)

[Классы и структуры](#)

[Члены структур и объединений](#)

Операторы умножения и оператор модуля

12.11.2021 • 2 minutes to read

Синтаксис

```
expression * expression
expression / expression
expression % expression
```

Remarks

Ниже перечислены мультипликативные операторы.

- Умножение (*)
- Деление (/)
- Модуль (остаток от деления) (%)

Эти бинарные операторы имеют ассоциативность слева направо.

Мультипликативные операторы принимают операнды арифметических типов. Оператор модуля (%) имеет более строгое требование, что его операнды должны иметь целочисленный тип. (Чтобы получить остаток от деления с плавающей точкой, используйте функцию времени выполнения [FMOD](#).) Преобразования, охваченные [стандартными преобразованиями](#), применяются к operandам, а результат относится к преобразованному типу.

Оператор умножения возвращает результат умножения первого операнда на второй.

Оператор деления возвращает результат деления первого операнда на второй.

Оператор модуля выдает остаток, заданный следующим выражением, где $E1$ является первым operandом, а $E2$ — вторым: $E1 - (E1 / E2) * E2$, где оба operandы имеют целочисленные типы.

Деление на 0 в выражении деления или модуля не определено и вызывает ошибку времени выполнения. Поэтому следующие выражения создают неопределенные ошибочные результаты.

```
i % 0
f / 0.0
```

Если оба operandы в выражении умножения, деления или модуля имеют одинаковые знаки, результат будет положительным. В противном случае результат будет отрицательным. Знак результата операции модуля определяется реализацией.

NOTE

Поскольку преобразования, выполняемые мультипликативными операторами, не обеспечивают условия переполнения и потери значимости, данные могут быть потеряны, если результат мультипликативной операции невозможно представить в типе operandов после преобразования.

В Microsoft C++ знак результата выражения модуля всегда совпадает со знаком первого операнда.

Завершение блока, относящегося только к системам Microsoft

Если значение, полученное при делении двух целых чисел, неточное и только один операнд является отрицательным, результатом будет наибольшее целое число (по величине без учета знака), которое меньше точного значения, которое было бы получено при операции деления. Например, вычисленное значение -11/3 равно -3,666666666. Результат этого целого деления равен -3.

Связь между операторами мультипликативные задается удостоверением $(E1 / E2) * E2 + E1 \% E2 == E1$.

Пример

В следующей программе показаны мультипликативные операторы. Обратите внимание, что любой из операндов `10 / 3` должен быть явно приведен к типу, `float` чтобы избежать усечения, чтобы оба операнда были типа `float` до деления.

```
// expr_Multiplicative_Operators.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main() {
    int x = 3, y = 6, z = 10;
    cout << "3 * 6 is " << x * y << endl
        << "6 / 3 is " << y / x << endl
        << "10 % 3 is " << z % x << endl
        << "10 / 3 is " << (float) z / x << endl;
}
```

См. также:

[Выражения с бинарными операторами](#)

[Операторы C++, приоритет и ассоциативность](#)

[Операторы умножения в C](#)

Оператор new (C++)

12.11.2021 • 7 minutes to read

Выделяет память для объекта или массива объектов *типа-Name* из свободного хранилища и возвращает подходящим образом типизированный ненулевой указатель на объект.

NOTE

Расширения компонентов Microsoft C++ обеспечивают поддержку `new` ключевого слова для добавления записей слота vtable. Дополнительные сведения см. в разделе [создать \(новый слот в таблице vtable\)](#).

Синтаксис

```
[::] new [placement] new-type-name [new-initializer]  
[::] new [placement] ( type-name ) [new-initializer]
```

Remarks

В случае неудачи `new` возвращает ноль или создает исключение; дополнительные сведения см. в разделе [операторы new и DELETE](#). Это поведение по умолчанию можно изменить, написав настраиваемую подпрограмму обработки исключений и вызывая функцию `_set_new_handler` библиотеки времени выполнения с именем функции в качестве аргумента.

Сведения о создании объекта в управляемой куче см. в разделе [gcnew](#).

Если `new` используется для выделения памяти для объекта класса C++, конструктор объекта вызывается после выделения памяти.

Используйте оператор `Delete`, чтобы освободить память, выделенную с помощью `new` оператора.

В следующем примере выделяется и затем освобождается двумерный массив символов размером `dim` на 10. При выделении многомерного массива все измерения, кроме первого, должны быть константными выражениями, которые возвращают положительные значения; самое левое измерение массива может являться любым выражением, результатом которого является положительное значение. При выделении массива с помощью `new` оператора первое измерение может равняться нулю — `new` оператор возвращает уникальный указатель.

```
char (*pchar)[10] = new char[dim][10];  
delete [] pchar;
```

Имя типа не может содержать имена `const`, `volatile`, объявления классов или объявления перечислений. Таким образом, следующее выражение является недопустимым:

```
volatile char *vch = new volatile char[20];
```

`new` Оператор не выделяет ссылочные типы, так как они не являются объектами.

`new` Оператор не может использоваться для выделения функции, но может использоваться для выделения указателей на функции. В следующем примере выделяется и затем освобождается массив из

семи указателей на функции, которые возвращают целые числа.

```
int (**p) () = new (int (*[7]) ());
delete *p;
```

Если вы используете оператор `new` без дополнительных аргументов и компилируете с параметром `/GX`, `/EHs` или `/EHa`, компилятор создаст код для вызова оператора, `delete`. Если конструктор выдаст исключение.

В следующем списке описаны элементы грамматики `new`:

размещаем

Представляет способ передачи дополнительных аргументов при перегрузке `new`.

имя типа

Определяет тип для распределения; может быть встроенным или пользовательским типом. Если спецификация типа является сложной, она может быть окружена круглыми скобками, чтобы принудительно реализовать порядок привязки.

initializer

Представляет значение для инициализированного объекта. Инициализаторы невозможно задать для массивов. `new` Оператор создает массивы объектов, только если у класса есть конструктор по умолчанию.

Пример: выделение и освобождение массива символов

В следующем примере кода выделяется и освобождается массив символов и объект класса `CName`.

```

// expre_new_Operator.cpp
// compile with: /EHsc
#include <string.h>

class CName {
public:
    enum {
        sizeOfBuffer = 256
    };

    char m_szFirst[sizeOfBuffer];
    char m_szLast[sizeOfBuffer];

public:
    void SetName(char* pszFirst, char* pszLast) {
        strcpy_s(m_szFirst, sizeOfBuffer, pszFirst);
        strcpy_s(m_szLast, sizeOfBuffer, pszLast);
    }
};

int main() {
    // Allocate memory for the array
    char* pCharArray = new char[CName::sizeOfBuffer];
    strcpy_s(pCharArray, CName::sizeOfBuffer, "Array of characters");

    // Deallocate memory for the array
    delete [] pCharArray;
    pCharArray = NULL;

    // Allocate memory for the object
    CName* pName = new CName;
    pName->SetName("Firstname", "Lastname");

    // Deallocate memory for the object
    delete pName;
    pName = NULL;
}

```

Пример: `new` оператор

Если используется новая форма `new` оператора размещения, форма с аргументами в дополнение к размеру выделения, компилятор не поддерживает форму размещения `delete` оператора, если конструктор создает исключение. Пример:

```

// expre_new_Operator2.cpp
// C2660 expected
class A {
public:
    A(int) { throw "Fail!"; }
};

void F(void) {
    try {
        // heap memory pointed to by pa1 will be deallocated
        // by calling ::operator delete(void*).
        A* pa1 = new A(10);
    } catch (...) {
    }
    try {
        // This will call ::operator new(size_t, char*, int).
        // When A::A(int) does a throw, we should call
        // ::operator delete(void*, char*, int) to deallocate
        // the memory pointed to by pa2. Since
        // ::operator delete(void*, char*, int) has not been implemented,
        // memory will be leaked when the deallocation cannot occur.

        A* pa2 = new(__FILE__, __LINE__) A(20);
    } catch (...) {
    }
}

int main() {
    A a;
}

```

Инициализация объектов, выделенных с помощью оператора new

Необязательное поле *инициализатора* включено в грамматику для `new` оператора. Это позволяет инициализировать новые объекты с помощью пользовательских конструкторов. Дополнительные сведения о том, как выполняется инициализация, см. в разделе [инициализаторы](#). В следующем примере показано, как использовать выражение инициализации с `new` оператором:

```

// expre_Initializing_Objects_Allocated_with_new.cpp
class Acct
{
public:
    // Define default constructor and a constructor that accepts
    // an initial balance.
    Acct() { balance = 0.0; }
    Acct( double init_balance ) { balance = init_balance; }

private:
    double balance;
};

int main()
{
    Acct *CheckingAcct = new Acct;
    Acct *SavingsAcct = new Acct ( 34.98 );
    double *HowMuch = new double ( 43.0 );
    // ...
}

```

В этом примере объект `CheckingAcct` выделяется с помощью `new` оператора, но инициализация по умолчанию не задана. Поэтому вызывается конструктор по умолчанию для класса `Acct()`. Затем аналогичным образом выделяется объект `SavingsAcct` с единственным отличием: он явно инициализируется значением 34,98. Поскольку 34,98 имеет тип `double`, для работы с инициализацией

вызывается конструктор, принимающий аргумент этого типа. Наконец, неклассовый тип `HowMuch` инициализируется значением 43,0.

Если объект имеет тип класса и этот класс имеет конструкторы (как в предыдущем примере), объект может инициализироваться `new` оператором только в том случае, если выполняется одно из следующих условий:

- Аргументы, предоставленные в инициализаторе, согласуются с аргументами конструктора.
- Класс имеет конструктор по умолчанию (конструктор, который можно вызвать без аргументов).

При выделении массивов с помощью оператора явная инициализация каждого элемента невозможна `new`; вызывается только конструктор по умолчанию, если он имеется. Дополнительные сведения см. в разделе [аргументы по умолчанию](#).

Если выделение памяти завершается ошибкой (**оператор New** возвращает значение 0), инициализация не выполняется. Это обеспечивает защиту от попыток инициализировать данные, которые не существуют.

Как и в случае вызова функций, порядок вычисления выражений инициализации не определен. Кроме того, не следует полагаться на то, что эти выражения полностью вычислены до выделения памяти. Если выделение памяти завершается ошибкой и `new` оператор возвращает ноль, некоторые выражения в инициализаторе не могут быть полностью оценены.

Время жизни объектов, выделенных с помощью оператора new

Объекты, выделенные с помощью `new` оператора, не уничтожаются, когда закрывается область, в которой они определены. Поскольку `new` оператор возвращает указатель на объекты, которые он выделяет, программа должна определить указатель с подходящей областью для доступа к этим объектам. Пример:

```
// expe_Lifetime_of_Objects_Allocated_with_new.cpp
// C2541 expected
int main()
{
    // Use new operator to allocate an array of 20 characters.
    char *AnArray = new char[20];

    for( int i = 0; i < 20; ++i )
    {
        // On the first iteration of the loop, allocate
        // another array of 20 characters.
        if( i == 0 )
        {
            char *AnotherArray = new char[20];
        }
    }

    delete [] AnotherArray; // Error: pointer out of scope.
    delete [] AnArray;     // OK: pointer still in scope.
}
```

После того как указатель `AnotherArray` в этом примере вышел за пределы области видимости, объект невозможно удалить.

Принцип работы оператора new

Выражение распределения — выражение, содержащее `new` оператор, выполняет три вещи:

- Находит и резервирует хранилище для объекта или объектов, которым нужно выделить память. После завершения этого этапа выделен требуемый объем памяти, но это еще не объект.
- Инициализирует объекты. После завершения инициализации имеется достаточно информации, чтобы выделенная память являлась объектом.
- Возвращает указатель на объекты типа указателя, производного от `new-Type-Name` или `Type-Name`. Программа использует этот указатель для доступа к новому объекту, которому выделена память.

`new` Оператор вызывает **оператор функции New**. Для массивов любого типа и для объектов, которые не относятся `class` к типу, `struct` или `union`, для выделения хранилища вызывается глобальная функция `::operator new`. Объекты класса-типов могут определять **собственные статические функции-члены** для отдельных классов.

Когда компилятор обнаруживает `new` оператор для выделения объекта типа `Type`, он выдает вызов `type :: operator new (sizeof (type))` или, если не определен определенный пользователем **оператор New**, `:: оператор New (sizeof (type))`. Таким образом, `new` оператор может выделить правильный объем памяти для объекта.

NOTE

Аргумент для **оператора New** имеет тип `size_t`. Этот тип определен в `<direct.h>`, `<malloc.h>`, `<memory.h>`, `<search.h>`, `<stddef.h>`, `<stdio.h>`, `<stdlib.h>`, `<string.h>` и `<time.h>`.

Параметр в грамматике разрешает указание *расположения* (см. статью грамматика для **оператора New**).

Параметр *размещения* может использоваться только для определяемых пользователем реализаций **оператора New**; Он позволяет передавать дополнительные сведения **оператору New**. Выражение с полем *размещения*, например, `T *TObject = new (0x0040) T;` преобразуется в

`T *TObject = T::operator new(sizeof(T), 0x0040);`, если класс `T` имеет оператор-член `New`, в противном случае — значение `T *TObject = ::operator new(sizeof(T), 0x0040);`.

Исходная цель поля *размещения* — разрешить выделение аппаратно зависимых объектов по указанным пользователем адресам.

NOTE

Хотя в предыдущем примере показан только один аргумент в поле *размещения*, нет ограничений на количество дополнительных аргументов, которые можно передать **оператору** следующим образом.

Даже если для типа класса определен **оператор New**, глобальный оператор можно использовать в следующем примере:

```
T *TObject = ::new TObject;
```

Оператор разрешения области действия (`::`) принудительно использует глобальный `new` оператор.

См. также

[Выражения с унарными операторами](#)

[Ключевые слова](#)

[Операторы new и delete](#)

Оператор дополнения до единицы: ~

12.11.2021 • 2 minutes to read

Синтаксис

```
~ cast-expression
```

Remarks

Оператор дополнения до единицы (`~`), иногда называемый оператором *побитового дополнения*, возвращает побитовое дополнение его операнда. То есть каждый бит, равный в операнде 1, в результате становится равным 0, а каждый бит, равный в операнде 0, в результате становится равным 1. Операнд оператора дополнения до единицы должен быть целочисленного типа.

Ключевое слово оператора для ~

C++ указывает на `compl` альтернативное написание для `~`. В языке C в качестве макроса в заголовке указывается альтернативное написание `<iso646.h>`. В C++ альтернативным написанием является ключевое слово; использование `<iso646.h>` или эквивалент C++ `<ciso646>` не рекомендуется. В Microsoft C++ `/permissive-` `/Za` параметр компилятора или необходим для включения альтернативного написания.

Пример

```
// expre_One_Complement_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main () {
    unsigned short y = 0xFFFF;
    cout << hex << y << endl;
    y = ~y;      // Take one's complement
    cout << hex << y << endl;
}
```

В этом примере новое значение, присвоенное переменной `y`, является дополнением до единицы значения 0xFFFF без знака, т. е. значением 0x0000.

Над целочисленными operandами выполняется восходящее приведение целого типа. Тип операнда, который повышается до, является результирующим типом. Дополнительные сведения о целочисленном повышении см. в разделе [стандартные преобразования](#).

См. также

[Выражения с унарными операторами](#)

[Встроенные операторы, приоритет и ассоциативность C++](#)

[Унарные арифметические операторы](#)

Операторы указателей на члены: * и->*

12.11.2021 • 2 minutes to read

Синтаксис

```
expression .* expression
expression ->* expression
```

Remarks

Операторы указателя на член, `*` и `->*`, возвращают значение определенного члена класса для объекта, указанного в левой части выражения. Правая часть должна определять член класса. В следующем примере показано использование этих операторов.

```
// expre_Expressions_with_Pointer_Member_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

class Testpm {
public:
    void m_func1() { cout << "m_func1\n"; }
    int m_num;
};

// Define derived types pmfn and pmd.
// These types are pointers to members m_func1() and
// m_num, respectively.
void (Testpm::*pmfn)() = &Testpm::m_func1;
int Testpm::*pmd = &Testpm::m_num;

int main() {
    Testpm ATestpm;
    Testpm *pTestpm = new Testpm;

    // Access the member function
    (ATestpm.*pmfn)();
    (pTestpm->*pmfn)(); // Parentheses required since * binds
                          // less tightly than the function call.

    // Access the member data
    ATestpm.*pmd = 1;
    pTestpm->*pmd = 2;

    cout << ATestpm.*pmd << endl
        << pTestpm->*pmd << endl;
    delete pTestpm;
}
```

Выходные данные

```
m_func1
m_func1
1
2
```

В приведенном выше примере указатель на член, `pmfn`, используется для вызова функции-члена `m_func1`. Другой указатель на член, `pmd`, используется для обращения к члену `m_num`.

Бинарный оператор `.*` объединяет свой первый operand, который должен быть объектом типа класса, со своим вторым operandом, который должен иметь тип указателя на член.

Бинарный оператор `-> *` объединяет свой первый operand, который должен быть указателем на объект типа класса, со вторым operandом, который должен быть типом указателя на член.

В выражении с оператором `.*` первый operand должен относиться к тому же типу класса, что и указатель на член, заданный во втором операнде (и быть доступным для него), или иметь доступный тип, однозначно производный от этого класса и доступный для него.

В выражении, содержащем оператор `-> *`, первый operand должен иметь тип "указатель на тип класса" типа, указанного во втором операнде, или тип должен быть однозначно производным от этого класса.

Пример

Рассмотрим следующие классы и фрагмент программы:

```
// expr_expressions_with_Pointer_Member_Operators2.cpp
// C2440 expected
class BaseClass {
public:
    BaseClass(); // Base class constructor.
    void Func1();
};

// Declare a pointer to member function Func1.
void (BaseClass::*pmfnFunc1)() = &BaseClass::Func1;

class Derived : public BaseClass {
public:
    Derived(); // Derived class constructor.
    void Func2();
};

// Declare a pointer to member function Func2.
void (Derived::*pmfnFunc2)() = &Derived::Func2;

int main() {
    BaseClass ABase;
    Derived ADerived;

    (ABase.*pmfnFunc1)(); // OK: defined for BaseClass.
    (ABase.*pmfnFunc2)(); // Error: cannot use base class to
                          // access pointers to members of
                          // derived classes.

    (ADerived.*pmfnFunc1)(); // OK: Derived is unambiguously
                           // derived from BaseClass.
    (ADerived.*pmfnFunc2)(); // OK: defined for Derived.
}
```

Результатом операторов указателя на член `*` или `->*` является объект или функция типа, указанного в объявлении указателя на член. Таким образом, в предыдущем примере результатом выражения

`ADerived.*pmfnFunc1()` является указатель на функцию, которая не возвращает значения (void). Если второй операнд имеет l-значение, то и результатом будет l-значение.

NOTE

Если результатом одного из операторов указателя на член является функция, то результат может использоваться только как operand к оператору вызова функции.

См. также

[Операторы C++, приоритет и ассоциативность](#)

Постфиксные операторы увеличения и уменьшения ++ и --

12.11.2021 • 2 minutes to read

Синтаксис

```
postfix-expression ++
postfix-expression --
```

Remarks

В C++ доступны префиксные и постфиксные операции инкремента и декремента. В этом разделе описываются только их постфиксные формы. (Дополнительные сведения см. в разделе [Операторы инкремента и декремента префикса](#).) Разница между ними заключается в том, что в постфиксной нотации оператор появляется после *постфиксного выражения*, тогда как в нотации префикса оператор появляется перед *выражением*. В следующем примере представлен постфиксный оператор инкремента:

```
i++;
```

Результатом применения оператора постфиксного инкремента (`++`) является то, что значение операнда увеличивается на одну единицу соответствующего типа. Аналогичным образом, результат применения постфиксного оператора декремента (`--`) заключается в том, что значение операнда уменьшается на одну единицу соответствующего типа.

Важно отметить, что постфиксное выражение инкремента или декремента вычисляет значение выражения *до* применения соответствующего оператора. Операция инкремента или декремента выполняется *после* вычисления операнда. Эта особенность возникает, только когда постфиксная операция инкремента или декремента выполняется в контексте выражения.

Если же постфиксный оператор применяется к аргументу функции, то инкремент или декrement значения аргумента необязательно будет выполнен до его передачи в функцию. Дополнительные сведения см. в разделе 1.9.17 стандарта C++.

Применение постфиксного оператора инкремента к указателю на массив объектов типа `long` фактически добавляет четыре к внутреннему представлению указателя. Это поведение приводит к тому, что указатель, который ранее ссылался на *n*-й элемент массива, ссылается на элемент (*n*+1) ТН.

Операнды для постфиксного инкремента и постфиксных операторов декремента должны быть изменяемыми (а не `const`) I-значениями арифметического типа. Тип результата такой же, как у *постфиксного выражения*, но больше не является I-значением.

Visual Studio 2017 версии 15,3 и более поздних версий (доступно в [/std: c++ 17](#)): операнд постфиксного оператора инкремента или декремента не может иметь тип `bool`.

Ниже показан пример постфиксного оператора инкремента.

```
// expre_Postfix_Increment_and_Decrement_Operators.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

int main() {
    int i = 10;
    cout << i++ << endl;
    cout << i << endl;
}
```

Постфиксные операции инкремента или декремента для типов перечисления не поддерживаются:

```
enum Compass { North, South, East, West };
Compass myCompass;
for( myCompass = North; myCompass != West; myCompass++ ) // Error
```

См. также раздел

[Постфиксные выражения](#)

[Операторы C++, приоритет и ассоциативность](#)

[Постфиксные операторы увеличения и уменьшения в C](#)

Префиксные операторы увеличения и уменьшения ++ и --

12.11.2021 • 2 minutes to read

Синтаксис

```
++ unary-expression  
-- unary-expression
```

Remarks

Оператор префикса инкремента (`++`) добавляет единицу к его операнду; это увеличенное значение является результатом выражения. Операнд должен быть L-значением, не имеющим тип `const`.

Результат — L-значение того же типа, как у операнда.

Оператор префикса декремента (`--`) аналогичен оператору префикса инкремента, за исключением того, что операнд уменьшается на единицу, а результатом является это уменьшенное значение.

Visual Studio 2017 версии 15,3 и более поздних версий (доступно в [/std: c++ 17](#)): операнд оператора инкремента или декремента не может иметь тип `bool`.

Операторы префиксных и постфиксных инкремента и декремента влияют на свои operandы. Они различаются между собой порядком выполнения инкремента или декремента при вычислении выражения (Дополнительные сведения см. в разделе [Постфиксные операторы инкремента и декремента](#).) В форме префикса приращение или уменьшение происходит до того, как значение используется при вычислении выражения, поэтому значение выражения отличается от значения операнда. В постфиксной форме инкремент или декремент выполняется после использования значения при вычислении выражения, поэтому значение выражения совпадает со значением операнда. Например, в следующей программе выполняется вывод на печать "`++i = 6`".

```
// exre_Increment_and_Decrement_Operators.cpp  
// compile with: /EHsc  
#include <iostream>  
  
using namespace std;  
  
int main() {  
    int i = 5;  
    cout << "++i = " << ++i << endl;  
}
```

Операнд целочисленного типа или типа с плавающей запятой инкрементируется или декрементируется на целое значение 1. Тип результата совпадает с типом операнда. Операнд типа указателя инкрементируется или декрементируется на значение размера объекта, к которому он относится. Инкрементированный указатель указывает на следующий объект, а декрементированный — на предыдущий.

Поскольку операторы инкремента и декремента имеют побочные эффекты, использование выражений с операторами инкремента или декремента в [макросе препроцессора](#) может иметь нежелательные результаты. Рассмотрим следующий пример.

```
// expre_Increment_and_Decrement_Operators2.cpp
#define max(a,b) ((a)<(b))?(b):(a)

int main()
{
    int i = 0, j = 0, k;
    k = max( ++i, j );
}
```

Макрос разворачивается до следующего выражения:

```
k = (((++i)<(j))?(j):(++i);
```

Если значение i больше или равно j или меньше j на 1, оно будет инкрементировано дважды.

NOTE

Встраиваемые функции C++ предпочтительнее макросов во многих случаях, поскольку исключают побочные эффекты, подобные описанным здесь, и позволяют языку выполнять более полную проверку типов.

См. также

[Выражения с унарными операторами](#)

[Операторы C++, приоритет и ассоциативность](#)

[Префиксные операторы увеличения и уменьшения](#)

Операторы отношения: < , > , < = И >=

12.11.2021 • 2 minutes to read

Синтаксис

```
expression < expression
expression > expression
expression <= expression
expression >= expression
```

Remarks

Бинарные операторы отношения определяют следующие отношения:

- Меньше (<)
- Больше (>)
- Меньше или равно (< =)
- Больше или равно (> =)

Операторы отношения обладают ассоциативностью слева направо. Оба операнда операторов отношения должны быть арифметического типа или типа указателя. Они получают значения типа `bool`. Возвращаемое значение равно `false` (0), если связь в выражении имеет значение `false`; в противном случае возвращается значение `true` (1).

Пример

```
// exre_Relational_Operators.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;

int main() {
    cout << "The true expression 3 > 2 yields: "
        << (3 > 2) << endl
        << "The false expression 20 < 10 yields: "
        << (20 < 10) << endl;
}
```

Выражения в предыдущем примере должны быть заключены в круглые скобки, так как оператор вставки потока (`<<`) имеет более высокий приоритет, чем операторы отношения. Поэтому первое выражение без скобок вычислялось бы следующим образом:

```
(cout << "The true expression 3 > 2 yields: " << 3) < (2 << "\n");
```

Обычные арифметические преобразования, охваченные [стандартными преобразованиями](#), применяются к operandам арифметических типов.

Сравнение указателей

При сравнении двух указателей на объекты одного типа результат определяется расположением объектов, указываемых в адресном пространстве программы. Указатели также можно сравнивать с константным выражением, результатом которого является значение 0 или указатель типа `void *`. Если сравнение указателей выполняется с указателем типа `void *`, другой указатель неявно преобразуется в тип `void *`. Затем выполняется сравнение.

Два указателя разных типов можно сравнивать, только если выполняются следующие условия.

- Один тип является типом класса, производным от другого типа.
- По крайней мере один из указателей явно преобразован (CAST) в тип `void *`. (Другой указатель неявно преобразуется в тип `void *` для преобразования.)

Гарантируется, что два указателя одного типа, указывающие на один и тот же объект, равны. Если сравниваются два указателя на нестатические члены объекта, применяются следующие правила.

- Если тип класса не является `union`, и если два члена не разделены *спецификатором доступа*, например `public`, `protected` ИЛИ `private`, то указатель на член, объявленный последним, будет сравниваться больше, чем указатель на член, объявленный ранее.
- Если два члена разделены *спецификатором доступа*, результаты будут неопределенными.
- Если класс имеет тип, то `union` указатели на различные элементы данных в этом `union` сравнении равны.

Если два указателя указывают на элементы одного массива или на элемент, находящийся за пределами массива, большим является указатель на объект с более высоким нижним индексом. Сравнение указателей гарантированно является допустимым, только если указатели ссылаются на объекты в одном массиве или на расположение объекта после конца массива.

См. также

[Выражения с бинарными операторами](#)

[Операторы C++, приоритет и ассоциативность](#)

[Операторы отношения и равенства C](#)

Оператор разрешения области:

12.11.2021 • 2 minutes to read

Оператор разрешения области `::` используется для идентификации и устранения неоднозначности идентификаторов, используемых в разных областях. Дополнительные сведения об области действия см. в разделе [Scope](#).

Синтаксис

```
qualified-id :  
    nested-name-specifier template необ. unqualified-id
```

```
nested-name-specifier :  
    ::  
    type-name ::  
    namespace-name ::  
    decltype-specifier ::  
    nested-name-specifier identifier ::  
    nested-name-specifier template неявное согласие simple-template-id ::
```

```
unqualified-id :  
    identifier  
    operator-function-id  
    conversion-function-id  
    literal-operator-id  
    ~ type-name  
    ~ decltype-specifier  
    template-id
```

Remarks

`identifier` может быть переменной, функцией или значением перечисления.

Использование `::` для классов и пространств имен

В следующем примере показано, каким образом оператор разрешения области используется с пространствами имен и классами:

```

namespace NamespaceA{
    int x;
    class ClassA {
        public:
            int x;
    };
}

int main() {

    // A namespace name used to disambiguate
    NamespaceA::x = 1;

    // A class name used to disambiguate
    NamespaceA::ClassA a1;
    a1.x = 2;
}

```

Оператор разрешения области без квалификатора области применяется к глобальному пространству имен.

```

namespace NamespaceA{
    int x;
}

int x;

int main() {
    int x;

    // the x in main()
    x = 0;
    // The x in the global namespace
    ::x = 1;

    // The x in the A namespace
    NamespaceA::x = 2;
}

```

Оператор разрешения области можно использовать для определения члена `namespace` ИЛИ для определения пространства имен, которое определяет пространство имен члена в `using` директиве. В приведенном ниже примере можно использовать `NamespaceC` для уточнения `ClassB`, даже если `ClassB` он был объявлен в пространстве имен `NamespaceB`, так как `NamespaceB` был назначен в `NamespaceC` `using` директиве.

```

namespace NamespaceB {
    class ClassB {
        public:
            int x;
    };
}

namespace NamespaceC{
    using namespace NamespaceB;
}

int main() {
    NamespaceB::ClassB b_b;
    NamespaceC::ClassB c_b;

    b_b.x = 3;
    c_b.x = 4;
}

```

Можно использовать цепочки операторов разрешения области. В следующем примере

`NamespaceD::NamespaceD1` определяет вложенное пространство имен `NamespaceD1`, а
`NamespaceE::ClassE::ClassE1` определяет вложенный класс `ClassE1`.

```

namespace NamespaceD{
    namespace NamespaceD1{
        int x;
    }
}

namespace NamespaceE{
    class ClassE{
        public:
            class ClassE1{
                public:
                    int x;
            };
    };
}

int main() {
    NamespaceD:: NamespaceD1::x = 6;
    NamespaceE::ClassE::ClassE1 e1;
    e1.x = 7 ;
}

```

Использовать `::` для статических членов

Оператор разрешения области необходимо использовать для вызова статических членов класса.

```
class ClassG {  
public:  
    static int get_x() { return x;}  
    static int x;  
};  
  
int ClassG::x = 6;  
  
int main() {  
  
    int gx1 = ClassG::x;  
    int gx2 = ClassG::get_x();  
}
```

Используется `::` для перечислений с заданной областью

Оператор разрешения с заданной областью также используется со значениями [объявлений перечисления](#) в области перечисления, как показано в следующем примере:

```
enum class EnumA{  
    First,  
    Second,  
    Third  
};  
  
int main() {  
    EnumA enum_value = EnumA::First;  
}
```

См. также:

[Встроенные операторы, приоритет и ассоциативность C++](#)
[Пространства имен](#)

Оператор sizeof

12.11.2021 • 2 minutes to read

Возвращает размер операнда по отношению к размеру типа `char`.

NOTE

Дополнительные сведения об `sizeof ...` операторе см. в разделе [шаблоны Variadic и многоточия](#).

Синтаксис

```
sizeof unary-expression  
sizeof ( type-name )
```

Remarks

Результат `sizeof` оператора имеет тип `size_t`, целочисленный тип, определенный в включаемом файле `<stddef.h>`. Этот оператор позволяет избежать задания зависимых от компьютера размера данных в программах.

Операнд `sizeof` может быть одним из следующих:

- Имя типа. Чтобы использовать `sizeof` с именем типа, имя должно быть заключено в круглые скобки.
- Выражение. При использовании с выражением `sizeof` можно указать с круглыми скобками или без них. Значение выражения не вычисляется.

Когда `sizeof` оператор применяется к объекту типа `char`, он возвращает 1. Когда `sizeof` оператор применяется к массиву, он возвращает общее число байтов в этом массиве, а не размер указателя, представленного идентификатором массива. Чтобы получить размер указателя, представленного идентификатором массива, передайте его в качестве параметра в функцию, которая использует `sizeof`. Пример:

Пример

```

#include <iostream>
using namespace std;

size_t getPtrSize( char *ptr )
{
    return sizeof( ptr );
}

int main()
{
    char szHello[] = "Hello, world!";

    cout << "The size of a char is: "
        << sizeof( char )
        << "\nThe length of " << szHello << " is: "
        << sizeof szHello
        << "\nThe size of the pointer is "
        << getPtrSize( szHello ) << endl;
}

```

Пример выходных данных

```

The size of a char is: 1
The length of Hello, world! is: 14
The size of the pointer is 4

```

Если `sizeof` оператор применяется к `class`, `struct` типу, или `union`, результатом является число байтов в объекте этого типа, а также любое дополнение, добавленное для выровняйте члены по границам слов. Результат не обязательно должен соответствовать размеру, вычисляемому путем добавления требований к хранению отдельных членов. Параметр компилятора [/Zp](#) и директива `pragma Pack` влияют на границы выравнивания для элементов.

`sizeof` Оператор никогда не возвращает значение 0 даже для пустого класса.

`sizeof` Оператор не может использоваться со следующими операндами:

- функции. (Однако `sizeof` может применяться к указателям на функции.)
- Битовые поля.
- Неопределенные классы.
- Тип `void`.
- Динамически создаваемые массивы.
- Внешние массивы.
- Неполные типы.
- Заключенные в скобки имена неполных типов.

Если `sizeof` оператор применяется к ссылке, результат будет таким же, как если `sizeof` бы он был применен к самому объекту.

Если безразмерный массив является последним элементом структуры, оператор `sizeof` возвращает размер структуры без массива.

`sizeof` Оператор часто используется для вычисления количества элементов в массиве с помощью выражения в форме:

```
sizeof array / sizeof array[0]
```

См. также

[Выражения с унарными операторами](#)

[Ключевые слова](#)

Индексный оператор []

12.11.2021 • 3 minutes to read

Синтаксис

```
postfix-expression [ expression ]
```

Remarks

Постфиксное выражение (которое также может быть первичным выражением), за которым следует оператор подстрочного индекса [], указывает индексирование массива.

Сведения об управляемых массивах в C++/CLI см. в разделе [массивы](#).

Как правило, значение, представленное *постфиксным выражением*, является значением указателя, например идентификатором массива, а *выражение* — целочисленным значением (включая перечислимые типы). Однако все, что необходимо синтаксически, — это чтобы одно из выражений имело тип указателя, а другие — целочисленный тип. Поэтому целочисленное значение может находиться в позиции *постфиксного выражения*, а значение указателя может быть в квадратных скобках в позиции *выражения* или подстрочного индекса. Рассмотрим следующий фрагмент кода:

```
int nArray[5] = { 0, 1, 2, 3, 4 };
cout << nArray[2] << endl;           // prints "2"
cout << 2[nArray] << endl;         // prints "2"
```

В предыдущем примере выражение `nArray[2]` совпадает с `2[nArray]`. Причина в том, что результат выражения индекса определяется `e1[e2]` следующим образом:

```
*((e2) + (e1))
```

Адрес, полученный в результате выражения, не может быть равен *E2* байтам из адреса *E1*. Вместо этого адрес масштабируется, чтобы дать следующий объект в массиве *E2*. Пример:

```
double aDb1[2];
```

Адреса `aDb[0]` и `aDb[1]` равны 8 байтам — размер объекта типа `double`. Это масштабирование в соответствии с типом объекта выполняется автоматически языком C++ и определяется в [аддитивных операторах](#), где обсуждается сложение и вычитание операндов типа указателя.

Индексное выражение также может иметь несколько индексов, как показано ниже:

```
expression1[ выражение2 ] [ expression3 ] ...
```

Индексные выражения связываются в направлении слева направо. Сначала вычисляется левое индексное выражение `expression1[expression2]`. Адрес, получающийся в результате сложения `expression1` и `expression2`, формирует выражение указателя. Затем к этому выражению указателя добавляется выражение `expression3`, чтобы образовать новое выражение указателя. Эти операции повторяются до тех пор, пока не будет добавлено последнее индексное выражение. Оператор косвенного обращения (`*`) применяется после вычисления последнего индексного выражения, если только значение конечного указателя не соответствует типу массива.

Выражения с несколькими индексами ссылаются на элементы многомерных массивов. Многомерный массив — это массив, элементы которого сами являются массивами. Например, первый элемент трехмерного массива является двумерным массивом. В следующем примере объявляется и инициализируется простой двухмерный массив символов.

```
// exprre_Subscript_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
#define MAX_ROWS 2
#define MAX_COLS 2

int main() {
    char c[ MAX_ROWS ][ MAX_COLS ] = { { 'a', 'b' }, { 'c', 'd' } };
    for ( int i = 0; i < MAX_ROWS; i++ )
        for ( int j = 0; j < MAX_COLS; j++ )
            cout << c[ i ][ j ] << endl;
}
```

Положительные и отрицательные индексы

Первым элементом массива является элемент с номером 0. Диапазон массива C++ состоит из *массива[0]* в *массив[size - 1]*. Однако C++ поддерживает положительные и отрицательные индексы. Отрицательные индексы не должны выходить за границы массива; в противном случае результаты непредсказуемы. В следующем примере кода показаны положительные и отрицательные индексы массива:

```
#include <iostream>
using namespace std;

int main() {
    int intArray[1024];
    for ( int i = 0, j = 0; i < 1024; i++ )
    {
        intArray[i] = j++;
    }

    cout << intArray[512] << endl; // 512
    cout << 257[intArray] << endl; // 257

    int *midArray = &intArray[512]; // pointer to the middle of the array

    cout << midArray[-256] << endl; // 256
    cout << intArray[-256] << endl; // unpredictable, may crash
}
```

Отрицательный индекс в последней строке может привести к ошибке во время выполнения, так как он указывает на позицию адреса 256 `int` ниже, чем у источника массива. Указатель `midArray` инициализируется в середину `intArray`; таким образом можно (но опасно) использовать для него как положительные, так и отрицательные индексы массива. Ошибки индексов массивов не создают ошибки времени компиляции, но дают непредсказуемые результаты.

Оператор индекса коммутативен. Поэтому *массив выражений [index]* и индекс *[Array]* гарантированно эквивалентны при условии, что оператор *индекса* не перегружен (см. раздел [перегруженные операторы](#)). Программисты чаще всего используют первую форму, но вторая форма также правильна.

См. также

[Постфиксные выражения](#)

[Операторы C++, приоритет и ассоциативность](#)

[Массивы](#)

[Одномерные массивы](#)

[Многомерные массивы](#)

Оператор typeid

12.11.2021 • 2 minutes to read

Синтаксис

```
typeid(type-id)
typeid(expression)
```

Remarks

`typeid` Оператор позволяет определить тип объекта во время выполнения.

Результатом `typeid` является `const type_info&`. Значение является ссылкой на `type_info` объект, представляющий либо *тип-ID*, либо тип *выражения* в зависимости от `typeid` используемой формы. Дополнительные сведения см. в разделе [класс type_info](#).

`typeid` Оператор не работает с управляемыми типами (абстрактными деклараторами или экземплярами). Сведения о получении *Type* указанного типа см. в разделе [typeid](#).

`typeid` Оператор выполняет проверку во время выполнения, когда применяется к l-значению типа полиморфизма, где истинный тип объекта не может быть определен с помощью предоставленной статической информации. К таким случаям относятся следующие.

- Ссылка на класс
- Указатель, разыменование `*`
- Указатель в индексе (`[]`). (Нельзя использовать индекс с указателем на полиморфизм типа.)

Если *выражение* указывает на тип базового класса, но объект фактически является типом, производным от базового класса, то `type_info` ссылка на производный класс является результатом. *Выражение* должно указывать на полиморфизм типа (класс с виртуальными функциями). В противном случае результатом является `type_info` статический класс, на который ссылается *выражение*. Кроме того, указатель должен быть разыменован таким образом, чтобы использовался объект, на который он указывает. Без разыменования указателя результат будет `type_info` для указателя, а не того, на который он указывает. Пример:

```

// expre_typeid_Operator.cpp
// compile with: /GR /EHsc
#include <iostream>
#include <typeinfo>

class Base {
public:
    virtual void vfunc() {}
};

class Derived : public Base {};

using namespace std;
int main() {
    Derived* pd = new Derived;
    Base* pb = pd;
    cout << typeid( pb ).name() << endl; //prints "class Base *"
    cout << typeid( *pb ).name() << endl; //prints "class Derived"
    cout << typeid( pd ).name() << endl; //prints "class Derived *"
    cout << typeid( *pd ).name() << endl; //prints "class Derived"
    delete pd;
}

```

Если *выражение* ссылается на указатель и значение этого указателя равно нулю, `typeid` вызывает исключение `bad_typeid`. Если указатель не указывает на допустимый объект, `__non_rtti_object` возникает исключение. Указывает, что попытка анализа RTTI, вызвавшего ошибку, вызвана тем, что объект является каким-то недопустимым. (Например, это неверный указатель, или код не был скомпилирован с помощью `/GR`).

Если *выражение* не является указателем, а не ссылкой на базовый класс объекта, результатом является `type_info` ссылка, представляющая статический тип *выражения*. Статический тип выражения ссылается на тип выражения, так как он известен во время компиляции. Семантика исполнения игнорируется при оценке статического типа выражения. Кроме того, ссылки по возможности игнорируются при определении статического типа выражения:

```

// expre_typeid_Operator_2.cpp
#include <typeinfo>

int main()
{
    typeid(int) == typeid(int&); // evaluates to true
}

```

`typeid` также может использоваться в шаблонах для определения типа параметра шаблона:

```

// expre_typeid_Operator_3.cpp
// compile with: /c
#include <typeinfo>
template < typename T >
T max( T arg1, T arg2 ) {
    cout << typeid( T ).name() << "s compared." << endl;
    return ( arg1 > arg2 ? arg1 : arg2 );
}

```

См. также

[Сведения о типах времени выполнения](#)

[Ключевые слова](#)

Унарные операторы «плюс» и «отрицание»: + и -

12.11.2021 • 2 minutes to read

Синтаксис

- + cast-expression
- cast-expression

+ - оператор

Результатом оператора унарного сложения (+) является значение его операнда. Операнд оператора унарного оператора сложения должен иметь арифметический тип.

Над целочисленными operandами выполняется восходящее приведение целого типа. Результирующим типом является тип, до которого повышается уровень операнда. Поэтому выражение `+ch`, где `ch` имеет тип `char`, приводит к типу `int` значение не изменяется. Дополнительные сведения о том, как выполняется продвижение, см. в разделе [стандартные преобразования](#).

- - оператор

Унарный оператор отрицания (-) создает отрицательное значение операнда. Операнд оператора унарного отрицания должен быть арифметическим типом.

Над целочисленными operandами выполняется восходящее приведение целого типа, и результирующим типом является тип, до которого повышается уровень операнда. Дополнительные сведения о том, как выполняется продвижение, см. в разделе [стандартные преобразования](#).

Блок, относящийся только к системам Microsoft

Унарное отрицание величин без знака выполняется путем вычитания значения операнда из числа 2^n , где n — количество битов в объекте заданного типа без знака.

Завершение блока, относящегося только к системам Microsoft

См. также

[Выражения с унарными операторами](#)

[Операторы C++, приоритет и ассоциативность](#)

Выражения (C++)

12.11.2021 • 2 minutes to read

В этом разделе описываются выражения C++. Выражения — это последовательности операторов и операндов, используемые в следующих целях.

- Вычисление значения из operandов.
- Назначение объектов или функций.
- Создание побочных эффектов. (Побочные эффекты — это любые действия, отличные от вычисления выражения, например изменение значения объекта.)

В C++ операторы могут перегружаться, и их значения могут определяться пользователем. Однако их приоритет и число принимаемых operandов изменить невозможно. В этом разделе описывается синтаксис и семантика не перегруженных операторов в том состоянии, в котором они предоставляются языком. Помимо [типов выражений](#) и [семантики выражений](#), рассматриваются следующие темы:

- [Первичные выражения](#)
- [Оператор разрешения области](#)
- [Постфиксные выражения](#)
- [Выражения с унарными операторами](#)
- [Выражения с бинарными операторами](#)
- [Условный оператор](#)
- [Константные выражения](#)
- [Операторы приведения](#)
- [Сведения о типах времени выполнения](#)

Разделы об операторах в других разделах:

- [Операторы C++, приоритет и ассоциативность](#)
- [Перегруженные операторы](#)
- [typeid \(C++/CLI\)](#)

NOTE

Операторы для встроенных типов не могут быть перегружены; их поведение предопределено.

См. также

[Справочник по языку C++](#)

Типы выражений

12.11.2021 • 2 minutes to read

Выражения C++ делятся на несколько категорий:

- [Первичные выражения](#). Это исходные компоненты, из которых состоят все остальные выражения.
- [Постфиксные выражения](#). Эти основные выражения, за которыми следует оператор (например, индекс массива или постфиксный оператор инкремента).
- [Выражения, сформированные с помощью унарных операторов](#). Унарные операторы действуют только на один operand в выражении.
- [Выражения, сформированные с помощью бинарных операторов](#). Бинарные операторы действуют на два операнда в выражении.
- [Выражения с условным оператором](#). Условным является троичный оператор — единственный такой оператор в языке C++. Он принимает три операнда.
- [Константные выражения](#). Константные выражения целиком состоят из константных данных.
- [Выражения с явными преобразованиями типов](#). Явные преобразования (приведения) типов могут использоваться в выражениях.
- [Выражения с операторами указателей на члены](#).
- [Приведение](#). Типобезопасное приведение можно использовать в выражениях.
- [Сведения о типах времени выполнения](#). Задает тип объекта во время выполнения программы.

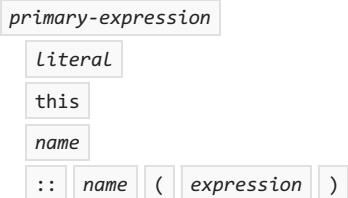
См. также

[Выражения](#)

Основные выражения

12.11.2021 • 2 minutes to read

Основные выражения являются стандартными блоками более сложных выражений. Они могут быть литералами, именами и именами, дополненными оператором разрешения области (`::`). Основное выражение может иметь любую из следующих форм.



`Literal` Является константным первичным выражением. Его тип зависит от формы его спецификации. Полные сведения об указании литералов см. в разделе [литералы](#).

`this` Ключевое слово является указателем на объект класса. Он доступен в нестатических функциях члена. Он указывает на экземпляр класса, для которого была вызвана функция. `this` Ключевое слово не может использоваться за пределами тела функции-члена класса.

Тип `this` указателя — `type * const` (где `type` — имя класса) в функциях, которые не изменяют `this` указатель. В следующем примере показаны объявления функций членов и типы `this`:

```
// expe_Primary_Expressions.cpp
// compile with: /LD
class Example
{
public:
    void Func();           // * const this
    void Func() const;     // const * const this
    void Func() volatile; // volatile * const this
};
```

Дополнительные сведения об изменении типа `this` указателя см. в разделе [this указатель](#).

Оператор разрешения области действия (`::`), за которым следует имя, является первичным выражением. Такие имена должны быть именами в глобальной области, а не именами членов. Тип выражения определяется объявлением имени. Это l-значение (т. е. оно может присутствовать в левой части выражения назначения), если объявляемое имя является l-значением. Оператор разрешения области действия позволяет ссылаться на глобальное имя, даже если это имя скрыто в текущей области. Пример использования оператора разрешения области действия см. в разделе [Scope](#).

Выражение, заключенное в круглые скобки, является первичным выражением. Его тип и значение идентичны типу и значению выражения, заключенного в скобки. Это l-значение, если выражение, заключенное в скобки, является l-значением.

Ниже приводятся примеры основных выражений.

```
100 // literal
'c' // literal
this // in a member function, a pointer to the class instance
::func // a global function
::operator + // a global operator function
::A::B // a global qualified name
( i + 1 ) // a parenthesized expression
```

Все эти примеры считаются *именами*, а также в качестве первичных выражений в различных формах:

```
MyClass // an identifier
MyClass::f // a qualified name
operator = // an operator function name
operator char* // a conversion operator function name
~MyClass // a destructor name
A::B // a qualified name
A<int> // a template id
```

См. также

[Типы выражений](#)

Шаблоны с многоточием и Variadic

12.11.2021 • 3 minutes to read

В этой статье показано, как использовать многоточие (...) в шаблонах C++ с переменным числом аргументов. Многоточие активно использовалось в C и C++. Они вводят переменные списки аргументов для функций. Одним из наиболее известных примеров является функция `printf()` из библиотеки времени выполнения C.

Шаблон *Variadic* — это класс или шаблон функции, поддерживающий произвольное число аргументов. Этот механизм особенно удобен для разработчиков библиотек C++, поскольку его можно применить к как к шаблонам классов, так и к шаблонам функций. Таким образом, он предоставляет широкий спектр широкий спектр типобезопасных и нетривиальных функций и гибких возможностей.

Синтаксис

В шаблонах шаблонами с переменным числом аргументов многоточие используется двумя способами. Слева от имени параметра обозначается *пакет параметров*, а справа от имени параметра — пакеты параметров разворачиваются в отдельные имена.

Ниже приведен простой пример синтаксиса определения *класса шаблона Variadic*:

```
template<typename... Arguments> class classname;
```

В обоих случаях (как при введении пакета параметров, так и при его развертывании) вокруг многоточия можно оставить пробельные символы, как показано в этом примере:

```
template<typename ...Arguments> class classname;
```

ИЛИ

```
template<typename ... Arguments> class classname;
```

Обратите внимание, что в этой статье используется соглашение, показанное в первом примере (многоточие присоединяется к `typename`).

В предыдущих примерах *аргументы* являются пакетом параметров. Класс `classname` может принимать переменное число аргументов, как показано в следующих примерах:

```
template<typename... Arguments> class vtclass;  
  
vtclass< > vtinstance1;  
vtclass<int> vtinstance2;  
vtclass<float, bool> vtinstance3;  
vtclass<long, std::vector<int>, std::string> vtinstance4;
```

Кроме того, определение шаблонного класса с переменным числом аргументов может устанавливать требование о том, что должен быть передан по меньшей мере один параметр:

```
template <typename First, typename... Rest> class classname;
```

Ниже приведен простой пример синтаксиса функции шаблона *Variadic*:

```
template <typename... Arguments> returntype functionname(Arguments... args);
```

Затем пакет параметров *arguments* разворачивается для использования, как показано в следующем разделе **Общие сведения о шаблонах Variadic**.

Возможны и другие формы синтаксиса шаблонной функции с переменным количеством аргументов возможны. Некоторые примеры приведены ниже.

```
template <typename... Arguments> returntype functionname(Arguments&... args);
template <typename... Arguments> returntype functionname(Arguments&&... args);
template <typename... Arguments> returntype functionname(Arguments*... args);
```

`const` Также разрешены спецификаторы:

```
template <typename... Arguments> returntype functionname(const Arguments&... args);
```

Шаблонные функции с переменным числом аргументов (как и аналогичные шаблонные классы) также могут устанавливать требование о том, что должен быть передан по меньшей мере один параметр.

```
template <typename First, typename... Rest> returntype functionname(const First& first, const Rest&... args);
```

В шаблонах с переменным числом аргументов используется оператор `sizeof...()` (он не имеет отношения к старому оператору `sizeof()`).

```
template<typename... Arguments>
void tfunc(const Arguments&... args)
{
    constexpr auto numargs{ sizeof...(Arguments) };

    X xobj[numargs]; // array of some previously defined type X

    helper_func(xobj, args...);
}
```

Дополнительные сведения о положении многоточия

Выше в этой статье говорилось, что если многоточие определяет пакеты параметров и их развертывание, то "Слева от имени параметра оно означает пакет параметров, а справа от имени параметра оно служит для развертывания пакетов параметров в отдельные имена". Технически это верно, но может порождать неоднозначности при трансляции в код. При этом нужно учитывать указанные ниже особенности.

- В шаблоне-parameter-list (`template <parameter-list>`) `typename...` представляет пакет параметров шаблона.
- В предложении "параметр-объявление" (`func(parameter-list)`) многоточие "верхнего уровня" представляет пакет параметров функции, а расположение многоточия имеет важное значение:

```
// v1 is NOT a function parameter pack:  
template <typename... Types> void func1(std::vector<Types...> v1);  
  
// v2 IS a function parameter pack:  
template <typename... Types> void func2(std::vector<Types>... v2);
```

- Там, где многоточие стоит непосредственно за именем параметра, оно используется для развертывания пакета параметров.

Пример

Механизм действия шаблонных функций с переменным числом аргументов можно проиллюстрировать на показательном примере — переписать с ее использованием какую-либо функциональность `printf`:

```
#include <iostream>  
  
using namespace std;  
  
void print() {  
    cout << endl;  
}  
  
template <typename T> void print(const T& t) {  
    cout << t << endl;  
}  
  
template <typename First, typename... Rest> void print(const First& first, const Rest&... rest) {  
    cout << first << ", ";  
    print(rest...); // recursive call using pack expansion syntax  
}  
  
int main()  
{  
    print(); // calls first overload, outputting only a newline  
    print(1); // calls second overload  
  
    // these call the third overload, the variadic template,  
    // which uses recursion as needed.  
    print(10, 20);  
    print(100, 200, 300);  
    print("first", 2, "third", 3.14159);  
}
```

Выходные данные

```
1  
10, 20  
100, 200, 300  
first, 2, third, 3.14159
```

NOTE

Большинство реализаций, включающих функции шаблонов Variadic, используют рекурсию некоторой формы, но немного отличается от традиционной рекурсии. Традиционная рекурсия включает функцию, вызывающую саму себя, используя ту же сигнатуру. (Она может быть перегружена или включена в шаблон, но одна и та же подпись выбирается каждый раз.) Рекурсия Variadic заключается в вызове шаблона функции Variadic с использованием отличающихся (почти всегда уменьшающихся) чисел аргументов и, таким образом, каждый раз отмечать разные сигнатуры. "Базовый случай" по-прежнему является обязательным, но природа рекурсии отличается.

Постфиксные выражения

12.11.2021 • 5 minutes to read

Постфиксные выражения состоят из основных выражений или выражений, в которых постфиксные операторы следуют за основным выражением. Постфиксные операторы перечислены в следующей таблице.

Постфиксные операторы

ИМЯ ОПЕРАТОРА	НОТАЦИЯ ОПЕРАТОРА
Оператор индекса	[]
Оператор вызова функции	()
Оператор явного преобразования типа	Type-Name()
Оператор доступа к элементу	. или ->,
Постфиксный оператор приращения	++
Постфиксный оператор уменьшения	--

Следующий синтаксис описывает возможные постфиксные выражения:

```
primary-expression
postfix-expression[expression]postfix-expression(expression-list)simple-type-name(expression-list)postfix-
expression.namepostfix-expression->namepostfix-expression++postfix-expression--cast-keyword < typename >
(expression )typeid ( typename )
```

Постфиксное выражение выше может быть [основным выражением](#) или другим постфиксным выражением. Постфиксные выражения группируются слева направо, делая возможным следующее связывание выражений.

```
func(1)->GetValue()++
```

В приведенном выше выражении является первичным выражением, является постфиксным выражением `func` `func(1)` функции, `func(1)->GetValue` является постфиксным выражением, определяющим член класса, `func(1)->GetValue()` является выражением постфикса функции, а все выражением является постфиксное выражение, увеличивающее возвращаемое значение `GetValue`. Значение выражения в целом имеет следующий смысл: "вызовите функцию, передающую 1 в качестве аргумента, и получите указатель на класс в качестве возвращаемого значения. Затем вызовите `GetValue()` для этого класса, а затем увеличьте возвращаемое значение.

Перечисленные выше выражения — это выражения присваивания, что означает, что результат этих выражений должен представлять собой r-значение.

Форма постфиксного выражения

```
simple-type-name ( expression-list )
```

показывает вызов конструктора. Если simple-type-name — это фундаментальный тип, список выражений должен представлять собой отдельное выражение, и это выражение обозначает приведение значения выражения к фундаментальному типу. Данный тип выражения приведения копирует конструктор. Поскольку эта форма позволяет создавать фундаментальные типы и классы с использованием одного и того же синтаксиса, эта форма особенно полезна при определении шаблонных классов.

Ключевое слово CAST — одно из `dynamic_cast`, `static_cast` или `reinterpret_cast`. Дополнительные сведения можно найти в `dynamic_cast`, `static_cast` и `reinterpret_cast`.

`typeid` Оператор считается постфиксным выражением. См. раздел **оператор typeid**.

Формальные и фактические аргументы

При вызове программ сведения передаются в вызываемую функцию в фактических аргументах. Вызываемые функции получают доступ к сведениям с помощью соответствующих формальных аргументов.

При вызове функции выполняются следующие задачи.

- Вычисляются все фактические аргументы (представляемые вызывающим объектом). Эти аргументы вычисляются в произвольном порядке, но все аргументы вычисляются и все побочные эффекты завершаются перед переходом в функцию.
- Каждый формальный аргумент инициализируется с соответствующим фактическим аргументом в списке выражений. (Формальный аргумент — это аргумент, объявленный в заголовке функции и используемый в теле функции.) Преобразования выполняются как при инициализации — как стандартные, так и пользовательские преобразования выполняются при преобразовании фактического аргумента в правильный тип. В общем виде выполненная инициализация показана в следующем коде.

```
void Func( int i ); // Function prototype
...
Func( 7 );           // Execute function call
```

Ниже представлены концептуальные инициализации до вызова.

```
int Temp_i = 7;
Func( Temp_i );
```

Обратите внимание, что инициализация выполняется таким образом, как если бы использовался синтаксис со знаком равенства, а не синтаксис с круглыми скобками. Копия `i` создается до передачи значения в функцию. (Дополнительные сведения см. в разделе [инициализаторы и преобразования](#)).

Таким образом, если прототип функции (объявление) вызывает аргумент типа `long`, и если вызывающая программа предоставляет фактический аргумент типа `int`, фактический аргумент повышается с помощью преобразования стандартного типа в тип `long` (см. раздел [стандартные преобразования](#)).

Предоставление фактического аргумента при отсутствии стандартного или пользовательского преобразования в тип формального аргумента будет ошибкой.

В случае фактических аргументов типа класса формальный аргумент инициализируется путем вызова конструктора класса. (Дополнительные сведения об этих специальных функциях-членах класса см. в разделе [конструкторы](#).)

- Выполняется вызов функции.

В следующем фрагменте программного кода показан вызов функции.

```
// expe_Formal_and_Actual_Arguments.cpp
void func( long param1, double param2 );

int main()
{
    long i = 1;
    double j = 2;

    // Call func with actual arguments i and j.
    func( i, j );
}

// Define func with formal parameters param1 and param2.
void func( long param1, double param2 )
{
```

При `func` вызове из Main формальный параметр `param1` инициализируется значением `i` (`i` преобразуется в тип в соответствии с `long` правильным типом, используя стандартное преобразование), а формальный параметр `param2` инициализируется значением `j` (`j` преобразуется в тип `double` с помощью стандартного преобразования).

Работа с типами аргументов

Формальные аргументы, объявленные как `const` типы, нельзя изменить в теле функции. Функции могут изменять любой аргумент, не являющийся типом `const`. Однако изменение является локальным для функции и не влияет на фактическое значение аргумента, если фактический аргумент не был ссылкой на объект, не являющийся типом `const`.

Некоторые из этих понятий показаны на примере следующих функций.

```
// expe_Treatment_of_Argument_Types.cpp
int func1( const int i, int j, char *c ) {
    i = 7;      // C3892 i is const.
    j = i;      // value of j is lost at return
    *c = 'a' + j; // changes value of c in calling function
    return i;
}

double& func2( double& d, const char *c ) {
    d = 14.387; // changes value of d in calling function.
    *c = 'a';   // C3892 c is a pointer to a const object.
    return d;
}
```

Многоточие и аргументы по умолчанию

Чтобы функции принимали меньше аргументов, чем указано в определении функции, их можно определять, используя многоточие (`...`) или аргументы по умолчанию.

Многоточие означает, что аргументы могут быть обязательными, но число и типы не указаны в

объявлении. Это обычно плохой стиль программирования на языке C++, поскольку он сводит на нет одно из преимуществ этого языка: безопасность типа. Различные преобразования применяются к функциям, объявленным с многоточием, а не к функциям, для которых известны формальный и фактический типы аргументов:

- Если фактический аргумент имеет тип `float`, `double` перед вызовом функции он повышается до типа.
- Любой `signed char`, `unsigned char`, `signed short` перечисляемый тип или, или, или `unsigned short` битовое поле преобразуются в `signed int` или `unsigned int` с помощью целочисленного повышения.
- Любой аргумент типа класса передается по значению в виде структуры данных; копия создается путем двоичного копирования, а не путем вызова конструктора копии класса (если он имеется).

Многоточие, если используется, должен быть объявлен последним в списке аргументов.

Дополнительные сведения о передаче переменного числа аргументов см. в обсуждении [va_arg](#), [va_start](#) и [va_list](#) в *справочнике по библиотеке времени выполнения*.

Дополнительные сведения о аргументах по умолчанию в программировании CLR см. в разделе [списки аргументов переменных \(...\)](#) (C++/CLI).

Аргументы по умолчанию позволяют задать значение, которое должен принять аргумент, если в вызове функции значение не передано. В следующем фрагменте кода показано, как работают аргументы по умолчанию. Дополнительные сведения об ограничениях на указание аргументов по умолчанию см. в разделе [аргументы по умолчанию](#).

```
// expe_Ellipsis_and_Default_Arguments.cpp
// compile with: /EHsc
#include <iostream>

// Declare the function print that prints a string,
// then a terminator.
void print( const char *string,
            const char *terminator = "\n" );

int main()
{
    print( "hello," );
    print( "world!" );

    print( "good morning", ", " );
    print( "sunshine." );
}

using namespace std;
// Define print.
void print( const char *string, const char *terminator )
{
    if( string != NULL )
        cout << string;

    if( terminator != NULL )
        cout << terminator;
}
```

Представленная выше программа объявляет функцию `print`, принимающую два аргумента. Однако второй аргумент, *признак конца*, имеет значение по умолчанию, `"\n"`. В `main` первые два вызова `print` позволяют использовать второй аргумент по умолчанию для предоставления новой строки для завершения выводимой строки. В третьем вызове указывается явное значение второго аргумента. После

выполнения этой программы выводится следующий результат:

```
hello,  
world!  
good morning, sunshine.
```

См. также раздел

[Типы выражений](#)

Выражения с унарными операторами

12.11.2021 • 2 minutes to read

Унарные операторы действуют только на один operand в выражении. Ниже приводится список унарных операторов:

- Оператор косвенного обращения (*)
- Оператор взятия адреса (&)
- Оператор унарного плюса (+)
- Унарный оператор отрицания (-)
- Оператор логического отрицания (!)
- Оператор дополнения до единицы (~)
- Префиксный оператор инкремента (++)
- Оператор префикса декремента (--)
- Оператор CAST ()
- Оператор `sizeof`
- Оператор `_uuidof`
- Оператор `alignof`
- Оператор `new`
- Оператор `delete`

Эти операторы имеют ассоциативность справа налево. Обычно синтаксис унарных выражений предшествует синтаксису постфиксных или основных выражений.

Ниже перечислены возможные формы унарных выражений.

- *postfix-expression*
- `++ unary-expression`
- `-- unary-expression`
- приведение унарных операторов -выражение
- `** sizeof ** унарное выражение`
- `sizeof(Имя типа)`
- `decltype(Выражение)`
- `Выражение выделения`
- `unallocation — выражение`

Любое постфиксное выражение считается *унарным выражением*, а так как любое первичное выражение считается *постфиксным выражением*, любые первичные выражения считаются также *унарным*

выражением. Дополнительные сведения см. в разделе [постфиксные выражения](#) и [Первичные выражения](#).

Унарный оператор состоит из одного или нескольких следующих символов: `* & + - ! ~`

Выражение CAST-Expression является унарным выражением с необязательным приведением для изменения типа. Дополнительные сведения см. в разделе [оператор CAST: \(\)](#).

Выражение может быть любым выражением. Дополнительные сведения см. в разделе [выражения](#).

Выражение выделения ссылается на `new` оператор. *Выражение освобождения* — это `delete` оператор. Дополнительные сведения см. по ссылкам, приведенным выше.

См. также

[Типы выражений](#)

Выражения с бинарными операторами

12.11.2021 • 2 minutes to read

Бинарные операторы действуют на два операнда в выражении. Используются следующие бинарные операторы.

- [Мультипликативные операторы](#)

- Умножение (*)
- Деление (/)
- Остаток (%)

- [Аддитивные операторы](#)

- Сложение (+)
- Вычитание (-)

- [Операторы сдвига](#)

- Сдвиг вправо (>>)
- Сдвиг влево (<<)

- [Операторы отношения и равенства](#)

- Знак "меньше" (<)
- Знак «больше» >)
- Меньше или равно (< =)
- Больше или равно (> =)
- Равно (==)
- Не равно (!=)

- битовые операторы;

- Побитовое и (&)
- Побитовое исключающее или (^)
- Побитовое инклюзивное или (|)

- Логические операторы

- Логическое и (&&)
- Логическое или (||)

- [Операторы присваивания](#)

- Присваивание (=)
- Присваивание сложения (+=)
- Присваивание вычитания (-=)

- Присваивание умножения ($*=$)
 - Присваивание деления ($/=$)
 - Присваивание остатка ($\% =$)
 - Присваивание сдвига влево ($<< =$)
 - Присваивание сдвига вправо ($>> =$)
 - Присваивание побитового И ($\&=$)
 - Присваивание побитового исключающего ИЛИ ($\^{}=$)
 - Побитовое инклюзивное или присваивание ($\|=$)
- Оператор "запятая" (,)

См. также раздел

[Типы выражений](#)

Постоянные выражения C++

12.11.2021 • 2 minutes to read

Константа — это значение, которое не изменяется. В C++ имеются два ключевых слова, позволяющих указать, что объект не должен изменяться, и для осуществления этого намерения.

В C++ требуются константные выражения, то есть выражения, результатом вычисления которых является константа, для объявления следующих элементов.

- Границы массива.
- Селекторы в операторах case.
- Спецификация длины битового поля.
- Инициализаторы перечисления.

Ниже перечислены единственные допустимые операнды в константных выражениях.

- Литералы
- Константы перечисления.
- Значения, объявленные как константы, которые инициализированы с константными выражениями.
- `sizeof` выражения

Нецелочисленные константы должны преобразовываться (явно или неявно) в целочисленные типы, чтобы быть допустимыми в константном выражении. Поэтому следующий код допустим.

```
const double Size = 11.0;
char chArray[(int)Size];
```

Явные преобразования в целочисленные типы допустимы в константных выражениях; все остальные типы и производные типы являются недопустимыми, за исключением случаев, когда оператор используется в качестве operandов `sizeof`.

Запятую-оператор и операторы присваивания невозможно использовать в константных выражениях.

См. также

[Типы выражений](#)

Семантика выражений

12.11.2021 • 3 minutes to read

Выражения оцениваются по приоритетности и группировке операторов. ([Приоритет операторов и ассоциативность в лексических соглашениях](#) показывает связи, которые операторы C++ накладывают на выражения.)

Порядок вычислений

Рассмотрим следующий пример.

```
// Order_of_Evaluation.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
int main()
{
    int a = 2, b = 4, c = 9;

    cout << a + b * c << "\n";
    cout << a + (b * c) << "\n";
    cout << (a + b) * c << "\n";
}
```

38
38
54

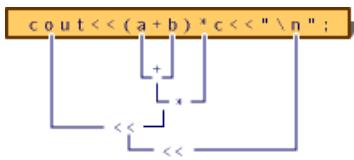


Порядок вычисления выражений

Порядок, в котором оценивается выражение, показанное на рисунке выше, определяется приоритетностью и ассоциативностью операторов.

- Умножение (*) имеет наибольший приоритет в этом выражении; следовательно, выражение $b * c$ оценивается первым.
- Сложение (+) имеет следующий наибольший приоритет, поэтому a добавляется к произведению b и c .
- Сдвиг влево (<<) имеет наименьший приоритет в выражении, но есть два вхождения. Поскольку оператор левого переноса группирует выражение слева направо, левое вложенное выражение оценивается первым, а затем — правое.

Если для группировки вложенных выражений используются скобки, они меняют приоритет и порядок оценки выражения, как показано на следующем рисунке.



Порядок вычисления выражений с круглыми скобками

Выражения, такие как на рисунке выше, оцениваются исключительно для побочного эффекта, в данном случае — для переноса информации на стандартное устройство вывода.

Нотация в выражениях

В языке C++ определен ряд параметров совместимости, действующих при указании операндов. В следующей таблице показаны типы операндов, допустимых для операторов, для которых требуются операнды типа *Type*.

Типы операндов, допустимые для операторов

ТРЕБУЕТСЯ ТИП	ДОПУСКАЮТСЯ ТИПЫ
<i>type</i>	$\boxed{\text{const}} \ \boxed{\text{volatile}} \ \boxed{\text{тип}}$ $\boxed{\text{const}} \ \boxed{\text{volatile}} \ \boxed{\text{тип\&}}$ $\boxed{\text{type\&}}$ $\boxed{\text{const}} \ \boxed{\text{тип\&}}$ $\boxed{\text{volatile}} \ \boxed{\text{тип\&}}$ $\boxed{\text{volatile const}} \ \boxed{\text{тип}}$ $\boxed{\text{volatile const}} \ \boxed{\text{тип\&}}$
<i>тип*</i>	$\boxed{\text{тип}}$ $\boxed{\text{const}} \ \boxed{\text{тип}}$ $\boxed{\text{volatile}} \ \boxed{\text{тип}}$ $\boxed{\text{volatile const}} \ \boxed{\text{тип}}$
$\boxed{\text{const}} \ \boxed{\text{тип}}$	$\boxed{\text{type}}$ $\boxed{\text{const}} \ \boxed{\text{тип}}$ $\boxed{\text{const}} \ \boxed{\text{тип\&}}$
$\boxed{\text{volatile}} \ \boxed{\text{тип}}$	$\boxed{\text{type}}$ $\boxed{\text{volatile}} \ \boxed{\text{тип}}$ $\boxed{\text{volatile}} \ \boxed{\text{тип\&}}$

Поскольку перечисленные выше правила всегда могут сочетаться друг с другом, то в тех операторах, в которых требуется указатель, можно задавать указатель с модификатором *const* на объект с модификатором *volatile*.

Неоднозначные выражения

Значение некоторых выражений неоднозначно. Такие выражения чаще всего встречаются, если значение объекта несколько раз изменяется в одном выражении. Подобные выражения зависят от конкретного порядка вычисления, когда этот порядок не определяется языком. Рассмотрим следующий пример.

```

int i = 7;

func( i, ++i );
  
```

Язык C++ не гарантирует порядок, в котором вычисляются аргументы при вызове функции. Поэтому в предыдущем примере в функцию `func` могут быть переданы значения параметров 7 и 8 или 8 и 8, в зависимости от порядка вычисления параметров — слева направо или справа налево.

Точки последовательности C++ (только для Microsoft)

Выражение может изменить значение объекта только один раз между следующими друг за другом точками следования.

В языке C++ в настоящее время точки следования не определены. В Microsoft C++ для любого выражения, содержащего операторы C и не содержащего перегруженные операторы, используются те же точки следования, что и в ANSI C. Когда операторы перегружены, семантика изменяется с последовательности операторов на последовательность вызовов функций. В Microsoft C++ используются следующие точки следования.

- Левый operand логического оператора AND (`&&`). Перед продолжением полностью вычисляется левый operand оператора логического И и учитываются все побочные эффекты. Гарантии вычисления правого operandна оператора логического И нет.
- Левый operand логического оператора или (`||`). Перед продолжением полностью вычисляется левый operand оператора логического ИЛИ и учитываются все побочные эффекты. Гарантии вычисления правого operandна оператора логического ИЛИ нет.
- Левый operand оператора запятой. Перед продолжением полностью вычисляется левый operand оператора запятой и учитываются все побочные эффекты. Оба operandна оператора запятой вычисляются всегда.
- Оператор вызова функции. До входа в функцию вычисляются выражение вызова функции и все ее аргументы, включая аргументы по умолчанию, а также учитываются все побочные эффекты. Порядок вычисления аргументов и выражения вызова функции не определен.
- Первый operand условного оператора. Перед продолжением полностью вычисляется первый operand условного оператора и учитываются все побочные эффекты.
- Конец полного выражения инициализации, например конец инициализации в операторе объявления.
- Выражение в операторе выражения. Операторы выражения состоят из необязательного выражения с последующей точкой с запятой (`;`). Выражение полностью вычисляется для учета его побочных эффектов.
- Управляющее выражение в операторе выбора (`if` или `switch`). До выполнения кода, зависящего от сделанного выбора, полностью вычисляется выражение и учитываются все побочные эффекты.
- Управляющее выражение оператора `while` или `do`. До выполнения любых операторов в следующей итерации цикла `while` или `do` полностью вычисляется выражение и учитываются все побочные эффекты.
- Каждое из трех выражений оператора `for`. До перехода к следующему выражению полностью вычисляется каждое выражение и учитываются все побочные эффекты.
- Выражение в операторе `return`. До возврата управления в вызывающую функцию полностью вычисляется выражение и учитываются все побочные эффекты.

См. также

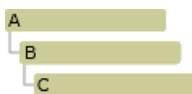
[Выражения](#)

Приведение

12.11.2021 • 2 minutes to read

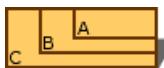
Если класс является производным от базового класса, содержащего виртуальные функции, то в языке C++ указатель на такой тип базового класса можно использовать для вызова реализаций виртуальных функций, находящихся в объекте производного класса. Класс, содержащий виртуальные функции, иногда называется "полиморфным".

Поскольку производный класс содержит определения всех базовых классов, от которых он является производным, можно безопасно привести указатель с повышением по иерархии классов к любому из этих базовых классов. Для указателя на базовый класс можно безопасно выполнить приведение с понижением по иерархии. Эта операция безопасна, если объект, на который осуществляется указание, фактически имеет тип, производный от базового класса. В этом случае говорят, что фактический объект является "полным". Говорят, что указатель на базовый класс указывает на "подчиненный объект" полного объекта. Например, рассмотрим иерархию классов, показанную на следующем рисунке.



Иерархия классов

Объект типа `C` может быть визуализирован, как показано на следующем рисунке.



Класс C с подобъектами B и A

Для экземпляра класса `C` имеются подчиненный объект `B` и подчиненный объект `A`. Экземпляр `C`, содержащий подчиненные объекты `A` и `B`, является "полным объектом".

Используя информацию о типах во время выполнения, можно проверить, указывает ли указатель на полный объект и можно ли безопасно выполнить приведение этого указателя, чтобы он указывал на другой объект в своей иерархии. Для выполнения этих типов приведений можно использовать оператор `dynamic_cast`. Данный оператор также выполняет проверку во время выполнения, чтобы сделать эту операцию безопасной.

Для преобразования типов нонполиморфик можно использовать оператор `static_cast` (в этом разделе объясняется различие между преобразованиями статического и динамического приведения и, когда они подходят для использования).

В этом разделе описываются следующие темы:

- [Операторы приведения](#)
- [Сведения о типах времени выполнения](#)

См. также

[Выражения](#)

Операторы приведения

12.11.2021 • 2 minutes to read

Некоторые операторы приведения типа используются только в языке C++. Эти операторы позволяют устранить неоднозначность и возможности допустить ошибку, которые характерны для приведения типов в стиле языка С. Эти операторы перечислены ниже.

- [dynamic_cast](#) Используется для преобразования полиморфизма типов.
- [static_cast](#) Используется для преобразования типов нонполиморфик.
- [const_cast](#) Используется для удаления `const` атрибутов, `volatile` и `__unaligned`.
- [reinterpret_cast](#) Используется для простой повторной интерпретации битов.
- [safe_cast](#) Используется в C++/CLI для создания проверяемого MSIL.

Используйте `const_cast` и `reinterpret_cast` в качестве последнего средства, так как эти операторы представляют те же опасности, что и старые преобразования в стиле. Однако они необходимы, чтобы полностью заменить приведения старого стиля.

См. также раздел

[Приведение](#)

Оператор dynamic_cast

12.11.2021 • 6 minutes to read

Преобразует operand `expression` в объект типа `type-id`.

Синтаксис

```
dynamic_cast < type-id > ( expression )
```

Remarks

Параметр `type-id` должен быть указателем или ссылкой на ранее определенный тип класса или "указателем на void". Тип операнда `expression` должен быть указателем, если `type-id` является указателем, или l-значением, если `type-id` является ссылкой.

Описание различий между преобразованиями статического и динамического приведения см. в разделе [static_cast](#).

Существует два критических изменения в работе `dynamic_cast` управляемого кода:

- `dynamic_cast` Указатель на базовый тип упакованного перечисления завершится ошибкой во время выполнения, возвращая 0 вместо преобразованного указателя.
- `dynamic_cast` больше не создает исключение `type-id`, когда является внутренним указателем на тип значения, при этом операция приведения завершается сбоем во время выполнения.
Приведение теперь возвращает значение указателя 0, а исключение не создается.

Если `type-id` является указателем на однозначно доступный прямой или косвенный базовый класс операнда `expression`, то результатом будет указатель на уникальный подобъект типа `type-id`. Пример:

```
// dynamic_cast_1.cpp
// compile with: /c
class B { };
class C : public B { };
class D : public C { };

void f(D* pd) {
    C* pc = dynamic_cast<C*>(pd);    // ok: C is a direct base class
                                         // pc points to C subobject of pd
    B* pb = dynamic_cast<B*>(pd);    // ok: B is an indirect base class
                                         // pb points to B subobject of pd
}
```

Этот тип преобразования называется "восходящим приведением типа", поскольку при нем указатель перемещается вверх по иерархии классов: от производного класса к классу, от которого он является производным. Восходящее приведение типа является неявным преобразованием.

Если `type-id` является указателем `void*`, выполняется проверка во время выполнения, чтобы определить фактический тип операнда `expression`. Результатом является указатель на полный объект, на который указывает operand `expression`. Пример:

```

// dynamic_cast_2.cpp
// compile with: /c /GR
class A {virtual void f();};
class B {virtual void f();};

void f() {
    A* pa = new A;
    B* pb = new B;
    void* pv = dynamic_cast<void*>(pa);
    // pv now points to an object of type A

    pv = dynamic_cast<void*>(pb);
    // pv now points to an object of type B
}

```

Если `type-id` не является указателем `void*`, то выполняется проверка во время выполнения, чтобы определить, может ли объект, на который указывает операнд `expression`, быть преобразован в тип, указанный параметром `type-id`.

Если тип операнда `expression` является базовым классом типа, заданного параметром `type-id`, то выполняется проверка во время выполнения, чтобы определить, указывает ли операнд `expression` на полный объект типа, заданного параметром `type-id`. Если это так, то результатом является указатель на полный объект типа, заданного параметром `type-id`. Пример:

```

// dynamic_cast_3.cpp
// compile with: /c /GR
class B {virtual void f();};
class D : public B {virtual void f();};

void f() {
    B* pb = new D;    // unclear but ok
    B* pb2 = new B;

    D* pd = dynamic_cast<D*>(pb);    // ok: pb actually points to a D
    D* pd2 = dynamic_cast<D*>(pb2);    // pb2 points to a B not a D
}

```

Этот тип преобразования называется "нисходящим приведением типа", поскольку при нем указатель перемещается вниз по иерархии классов: от заданного класса к производному от него классу.

В случаях множественного наследования возникают возможности для неоднозначности. Рассмотрим для примера иерархию классов, показанную на следующем рисунке.

Для типов CLR `dynamic_cast` результатом является отсутствие операции, если преобразование может быть выполнено неявно, или `isinst` инструкция MSIL, выполняющая динамическую проверку и возвращающая `nullptr` в случае сбоя преобразования.

В следующем примере используется `dynamic_cast`, чтобы определить, является ли класс экземпляром определенного типа:

```

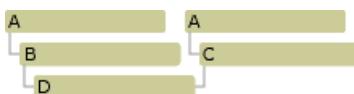
// dynamic_cast_clr.cpp
// compile with: /clr
using namespace System;

void PrintObjectType( Object^o ) {
    if( dynamic_cast<String>(o) )
        Console::WriteLine("Object is a String");
    else if( dynamic_cast<int>(o) )
        Console::WriteLine("Object is an int");
}

int main() {
    Object^o1 = "hello";
    Object^o2 = 10;

    PrintObjectType(o1);
    PrintObjectType(o2);
}

```



Иерархия классов, показывающая множественное наследование

Указатель на объект типа `D` можно безопасно привести к `B` или `C`. Однако если в результате приведения `D` указывает на объект `A`, какой экземпляр объекта `A` будет являться результатом? Это может привести к ошибке неоднозначного приведения. Чтобы обойти эту проблему, можно выполнить два однозначных приведения. Пример:

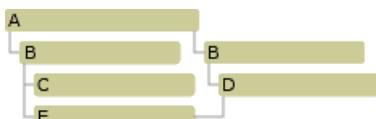
```

// dynamic_cast_4.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A {virtual void f();};
class D : public B, public C {virtual void f();};

void f() {
    D* pd = new D;
    A* pa = dynamic_cast<A*>(pd); // C4540, ambiguous cast fails at runtime
    B* pb = dynamic_cast<B*>(pd); // first cast to B
    A* pa2 = dynamic_cast<A*>(pb); // ok: unambiguous
}

```

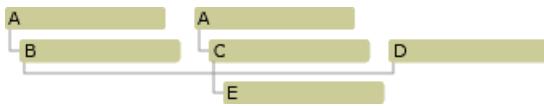
При использовании виртуальных базовых классов могут возникать дополнительные неоднозначности. Рассмотрим для примера иерархию классов, показанную на следующем рисунке.



Иерархия классов, показывающая виртуальные базовые классы

В этой иерархии объект `A` является виртуальным базовым классом. При указании экземпляра класса `E` и указателя на этот `A` подобъект, в `dynamic_cast` указатель на `B` ошибку произойдет сбой из-за неоднозначности. Необходимо сначала выполнить обратное приведение к полному объекту `E`, затем однозначным образом вернуться вверх по иерархии, чтобы дойти до нужного объекта `B`.

Рассмотрим для примера иерархию классов, показанную на следующем рисунке.



Иерархия классов, показывающая повторяющиеся базовые классы

При использовании заданного объекта типа `E` и указателя на подобъект `D` можно выполнить три преобразования, чтобы перейти от подобъекта `D` к крайнему слева подобъекту `A`. Можно выполнить `dynamic_cast` Преобразование из `D` указателя на `E` указатель, затем преобразование (`dynamic_cast` или неявное преобразование) из `E` в `B` и, наконец, неявное преобразование из `B` в `A`. Пример:

```

// dynamic_cast_5.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A {};
class D {virtual void f();};
class E : public B, public C, public D {virtual void f();};

void f(D* pd) {
    E* pe = dynamic_cast<E*>(pd);
    B* pb = pe;    // upcast, implicit conversion
    A* pa = pb;    // upcast, implicit conversion
}

```

`dynamic_cast` Оператор также можно использовать для выполнения перекрестного приведения. В иерархии классов из предыдущего примера можно выполнить приведение указателя, например, из подобъекта `B` в подобъект `D`, если полный объект имеет тип `E`.

С учетом перекрестного приведения можно выполнить преобразование из указателя на объект `D` в крайний левый подобъект `A` всего за два шага. Можно перекрестное приведение из `D` в `B`, а затем неявное преобразование из `B` в `A`. Пример:

```

// dynamic_cast_6.cpp
// compile with: /c /GR
class A {virtual void f();};
class B : public A {virtual void f();};
class C : public A {};
class D {virtual void f();};
class E : public B, public C, public D {virtual void f();};

void f(D* pd) {
    B* pb = dynamic_cast<B*>(pd);    // cross cast
    A* pa = pb;    // upcast, implicit conversion
}

```

Значение указателя `NULL` преобразуется в значение указателя `null` целевого типа `dynamic_cast`.

Если при использовании оператора `dynamic_cast < type-id > (expression)` невозможно точно преобразовать operand `expression` в тип `type-id`, то проверка во время выполнения приводит к сбою приведения. Пример:

```
// dynamic_cast_7.cpp
// compile with: /c /GR
class A {virtual void f();};
class B {virtual void f();};

void f() {
    A* pa = new A;
    B* pb = dynamic_cast<B*>(pa);    // fails at runtime, not safe;
    // B not derived from A
}
```

Значением приведения к типу указателя, которое привело к сбою, является пустой указатель. Неудачное приведение к ссылочному типу вызывает [исключение bad_cast](#). Если не `expression` указывает на допустимый объект или ссылается на него, `__non_rtti_object` возникает исключение.

Описание исключения см. в разделе `typeid __non_rtti_object`.

Пример

В следующем примере создается указатель базового класса (структура A) на объект (структуре C). Это (а также тот факт, что они являются виртуальными функциями) порождает полиморфизм времени выполнения.

В этом примере также вызывается невиртуальная функция в иерархии.

```

// dynamic_cast_8.cpp
// compile with: /GR /EHsc
#include <stdio.h>
#include <iostream>

struct A {
    virtual void test() {
        printf_s("in A\n");
    }
};

struct B : A {
    virtual void test() {
        printf_s("in B\n");
    }

    void test2() {
        printf_s("test2 in B\n");
    }
};

struct C : B {
    virtual void test() {
        printf_s("in C\n");
    }

    void test2() {
        printf_s("test2 in C\n");
    }
};

void Globaltest(A& a) {
    try {
        C &c = dynamic_cast<C&>(a);
        printf_s("in GlobalTest\n");
    }
    catch(std::bad_cast) {
        printf_s("Can't cast to C\n");
    }
}

int main() {
    A *pa = new C;
    A *pa2 = new B;

    pa->test();

    B * pb = dynamic_cast<B *>(pa);
    if (pb)
        pb->test2();

    C * pc = dynamic_cast<C *>(pa2);
    if (pc)
        pc->test2();

    C ConStack;
    Globaltest(ConStack);

    // will fail because B knows nothing about C
    B BonStack;
    Globaltest(BonStack);
}

```

```
in C
test2 in B
in GlobalTest
Can't cast to C
```

См. также

[Операторы приведения](#)

[Ключевые слова](#)

Исключение bad_cast

12.11.2021 • 2 minutes to read

Bad_cast исключение создается `dynamic_cast` оператором в результате неудачного приведения к ссылочному типу.

Синтаксис

```
catch (bad_cast)
    statement
```

Remarks

Интерфейс для `bad_cast`:

```
class bad_cast : public exception
```

Следующий код содержит пример сбоя `dynamic_cast`, который вызывает исключение `bad_cast`.

```
// expre_bad_cast_Exception.cpp
// compile with: /EHsc /GR
#include <typeinfo>
#include <iostream>

class Shape {
public:
    virtual void virtualfunc() const {}
};

class Circle: public Shape {
public:
    virtual void virtualfunc() const {}
};

using namespace std;
int main() {
    Shape shape_instance;
    Shape& ref_shape = shape_instance;
    try {
        Circle& ref_circle = dynamic_cast<Circle&>(ref_shape);
    }
    catch (bad_cast b) {
        cout << "Caught: " << b.what();
    }
}
```

Исключение возникает, так как объект, который поддается (`Shape`), не является производным от указанного типа приведения (`Circle`). Чтобы исключение не создавалось, добавьте в функцию `main` следующие объявления:

```
Circle circle_instance;
Circle& ref_circle = circle_instance;
```

Затем следует обратить смысл приведения в `try` блоке следующим образом:

```
Shape& ref_shape = dynamic_cast<Shape&>(ref_circle);
```

Члены

Конструкторы

КОНСТРУКТОР	ОПИСАНИЕ
<code>bad_cast</code>	Конструктор для объектов типа <code>bad_cast</code> .

Функции

КОМПОНЕНТ	ОПИСАНИЕ
<code>What</code>	TBD

Операторы

ОПЕРАТОР	ОПИСАНИЕ
<code>Оператор =</code>	Оператор присваивания, который присваивает один <code>bad_cast</code> объект другому.

bad_cast

Конструктор для объектов типа `bad_cast`.

```
bad_cast(const char * _Message = "bad cast");
bad_cast(const bad_cast &);
```

Оператор =

Оператор присваивания, который присваивает один `bad_cast` объект другому.

```
bad_cast& operator=(const bad_cast&) noexcept;
```

What

```
const char* what() const noexcept override;
```

См. также

[Оператор `dynamic_cast`](#)

[Словами](#)

[Современные рекомендации по C++ для исключений и обработки ошибок](#)

Оператор static_cast

12.11.2021 • 3 minutes to read

Преобразует выражение в тип *типа-ID*, основанный только на типах, имеющихся в выражении.

Синтаксис

```
static_cast <type-id> ( expression )
```

Remarks

В стандартном языке C++, проверка типа во время выполнения не выполняется, что обеспечивает безопасность преобразования. В C++/CX выполняются проверки во время компиляции и во время выполнения. Дополнительные сведения см. в разделе [Приведение](#).

`static_cast` Оператор можно использовать для таких операций, как преобразование указателя на базовый класс в указатель на производный класс. Такие преобразования не всегда являются безопасными.

В целом, `static_cast` Если требуется преобразовать числовые типы данных, такие как enums, в ints или ints в float, а также вы уверены, какие типы данных участвуют в преобразовании. `static_cast` преобразования не так надежны `dynamic_cast`, как преобразования, поскольку не `static_cast` выполняет проверку типов во время выполнения `dynamic_cast`. В случае `dynamic_cast` неоднозначного указателя произойдет сбой, а `static_cast` возвращается, как если бы ничего не возникало. Это может быть опасно. Хотя `dynamic_cast` преобразования являются более безопасными, `dynamic_cast` работают только с указателями или ссылками, а проверка типов во время выполнения является дополнительной нагрузкой. Дополнительные сведения см. в разделе [оператор dynamic_cast](#).

В следующем примере строка `D* pd2 = static_cast<D*>(pb);` небезопасна, поскольку `D` может иметь поля и методы, не входящие в `B`. Однако строка `B* pb2 = static_cast<B*>(pd);` является безопасным преобразованием, поскольку `D` всегда содержит все `B`.

```
// static_cast_Operator.cpp
// compile with: /LD
class B {};

class D : public B {};

void f(B* pb, D* pd) {
    D* pd2 = static_cast<D*>(pb);    // Not safe, D can have fields
                                         // and methods that are not in B.

    B* pb2 = static_cast<B*>(pd);    // Safe conversion, D always
                                         // contains all of B.
}
```

В отличие от `dynamic_cast`, при преобразовании не выполняется проверка во время выполнения `static_cast` `pb`. Объект, на который указывает `pb`, может не быть объектом типа `D`, и в этом случае использование `*pd2` может привести к ужасным последствиям. Например, вызов функции, являющейся членом класса `D`, но не класса `B`, может привести к нарушению прав доступа.

`dynamic_cast` Операторы И `static_cast` перемещают указатель на всю иерархию классов. Однако `static_cast` полагается исключительно на информацию, предоставленную в инструкции CAST, и поэтому может быть ненадежной. Пример:

```
// static_cast_Operator_2.cpp
// compile with: /LD /GR
class B {
public:
    virtual void Test(){}
};

class D : public B {};

void f(B* pb) {
    D* pd1 = dynamic_cast<D*>(pb);
    D* pd2 = static_cast<D*>(pb);
}
```

Если `pb` действительно указывает на объект типа `D`, `pd1` и `pd2` получат одно и то же значение. Также они получат одно и то же значение, если `pb == 0`.

Если `pb` указывает на объект типа `B` а не на полный `D` класс, то `dynamic_cast` будет достаточно, чтобы вернуть ноль. Однако `static_cast` полагается на утверждение программиста, которое `pb` указывает на объект типа `D` и просто возвращает указатель на этот предполагаемый `D` объект.

Следовательно, `static_cast` может выполнить обратное преобразование неявных преобразований, в этом случае результаты будут неопределенными. Программисту остается убедиться, что результаты `static_cast` преобразования являются надежными.

Это поведение также применяется к типам, отличным от типов класса. Например, `static_cast` можно использовать для преобразования из типа `int` в `char`. Однако в результате `char` может быть недостаточно битов для хранения всего `int` значения. Опять же, программисту остается убедиться, что результаты `static_cast` преобразования являются надежными.

`static_cast` Оператор также можно использовать для выполнения любого неявного преобразования, включая стандартные преобразования и пользовательские преобразования. Пример:

```
// static_cast_Operator_3.cpp
// compile with: /LD /GR
typedef unsigned char BYTE;

void f() {
    char ch;
    int i = 65;
    float f = 2.5;
    double dbl;

    ch = static_cast<char>(i); // int to char
    dbl = static_cast<double>(f); // float to double
    i = static_cast<BYTE>(ch);
}
```

`static_cast` Оператор может явно преобразовать целочисленное значение в тип перечисления. Если значение типа целого не оказывается в диапазоне значений перечисления, получаемое значение перечисления не определено.

`static_cast` Оператор преобразует нулевое значение указателя в значение указателя null целевого типа.

Любое выражение может быть явно преобразовано в тип `void` `static_cast` оператором. Тип `void` назначения может дополнительно включать `const` `volatile` атрибут, или `__unaligned`.

`static_cast` Оператор не может привести к отделам `const` `volatile` атрибуты, или `__unaligned`.

Сведения об удалении этих атрибутов см. в разделе [оператор `const_cast`](#).

C++/CLI: Из-за опасности возникновения непроверенных приведений на вершине повторного обнаружения сборщика мусора использование класса `static_cast` должно быть только в критическом для производительности коде, только если вы уверены, что он будет работать правильно. Если необходимо использовать `static_cast` в режиме выпуска, замените его `safe_cast` в отладочных сборках, чтобы убедиться в успешном выполнении.

См. также раздел

[Операторы приведения](#)

[Ключевые слова](#)

Оператор `const_cast`

12.11.2021 • 2 minutes to read

Удаляет `const` атрибуты, `volatile` и `__unaligned` из класса.

Синтаксис

```
const_cast <type-id> (expression)
```

Remarks

Указатель на любой тип объекта или указатель на член данных можно явно преобразовать в тип, идентичный, за исключением `const`, `volatile`, `__unaligned` квалификаторов, и. Для указателей и ссылок результат будет указывать на исходный объект. Для указателей на данные-члены результат будет указывать на тот же член, что и исходный указатель (`uncast`) на данные-член. В зависимости от типа объекта, на который осуществляется ссылка, операция записи с помощью результирующего указателя, ссылки или указателя на данные-член может привести к неопределенному поведению.

Нельзя использовать `const_cast` оператор для непосредственного переопределения состояния константы константной переменной.

`const_cast` Оператор преобразует нулевое значение указателя в значение указателя `null` целевого типа.

Пример

```
// expre_const_cast_Operator.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class CCTest {
public:
    void setNumber( int );
    void printNumber() const;
private:
    int number;
};

void CCTest::setNumber( int num ) { number = num; }

void CCTest::printNumber() const {
    cout << "\nBefore: " << number;
    const_cast< CCTest * >( this )->number--;
    cout << "\nAfter: " << number;
}

int main() {
    CCTest X;
    X.setNumber( 8 );
    X.printNumber();
}
```

В строке `const_cast`, содержащей тип данных `this` указателя — `const CCTest *`. `const_cast` Оператор

изменяет тип данных `this` указателя на `cctest *`, позволяя `number` изменять элемент. Приведение выполняется только для оставшейся части оператора, в котором оно указано.

См. также

[Операторы приведения](#)

[Ключевые слова](#)

Оператор reinterpret_cast

12.11.2021 • 2 minutes to read

Позволяет преобразовывать любой указатель в указатель любого другого типа. Также позволяет преобразовывать любой целочисленный тип в любой тип указателя и наоборот.

Синтаксис

```
reinterpret_cast < type-id > ( expression )
```

Remarks

Неправильное использование `reinterpret_cast` оператора может быть ненадежным. Если требуемое преобразование не является низкоуровневым по самой своей природе, следует использовать один из других операторов приведения типов.

`reinterpret_cast` Оператор можно использовать для преобразований `char*`, например, в `int*`, или `One_class*` в `Unrelated_class*`, которые по своей природе являются небезопасными.

Результат `reinterpret_cast` нельзя безопасно использовать ни для чего, кроме приведения обратно к исходному типу. Другие применения в лучшем случае будут непереносимыми.

`reinterpret_cast` Оператор не может привести к отделам `const` `volatile` атрибуты, или `__unaligned`. Сведения об удалении этих атрибутов см. в разделе [оператор const_cast](#).

`reinterpret_cast` Оператор преобразует нулевое значение указателя в значение указателя null целевого типа.

Одним из практических `reinterpret_cast` способов использования функции является функция хэширования, которая сопоставляет значение с индексом таким образом, чтобы два различных значения редко применялись к одному и тому же индексу.

```
#include <iostream>
using namespace std;

// Returns a hash code based on an address
unsigned short Hash( void *p ) {
    unsigned int val = reinterpret_cast<unsigned int>( p );
    return ( unsigned short )( val ^ (val >> 16));
}

using namespace std;
int main() {
    int a[20];
    for ( int i = 0; i < 20; i++ )
        cout << Hash( a + i ) << endl;
}

Output:
64641
64645
64889
64893
64881
64885
64873
64877
64865
64869
64857
64861
64849
64853
64841
64845
64833
64837
64825
64829
```

`reinterpret_cast` Позволяет обрабатывать указатель как целочисленный тип. Затем для получения уникального индекса (уникального с высокой степенью вероятности) к результату применяется побитовый сдвиг и операция XOR с самим собой. Далее индекс усекается с использованием стандартного для языка C приведения к типу возвращаемого значения функции.

См. также:

[Операторы приведения](#)

[Ключевые слова](#)

Сведения о типах времени выполнения

12.11.2021 • 2 minutes to read

Информация о типах времени выполнения (RTTI) — это механизм, позволяющий определить тип объекта во время выполнения программы. Функция RTTI была добавлена в язык C++, поскольку многие поставщики библиотек классов реализовывали ее самостоятельно. Это приводило к проблемам совместимости между библиотеками. Таким образом, стало очевидно, что необходима поддержка информации о типах времени выполнения на уровне языка.

Для ясности этот раздел о RTTI почти полностью посвящен указателям. Однако рассматриваемые понятия также применяются ко ссылкам.

Существует три основных элемента языка C++ для информации о типах времени выполнения.

- Оператор `dynamic_cast`.

Используется для преобразования полиморфных типов.

- Оператор `typeid`.

Используется для указания точного типа объекта.

- Класс `type_info`.

Используется для хранения сведений о типе, возвращаемых `typeid` оператором.

См. также

[Приведение](#)

Исключение bad_typeid

12.11.2021 • 2 minutes to read

Bad_typeid исключение вызывается [оператором typeid](#), если operand для typeid является пустым указателем.

Синтаксис

```
catch (bad_typeid)
    statement
```

Remarks

Интерфейс для bad_typeid :

```
class bad_typeid : public exception
{
public:
    bad_typeid();
    bad_typeid(const char * _Message = "bad typeid");
    bad_typeid(const bad_typeid & );
    virtual ~bad_typeid();

    bad_typeid& operator=(const bad_typeid& );
    const char* what() const;
};
```

В следующем примере показано, typeid как оператор создает исключение bad_typeid .

```
// expe_bad_typeid.cpp
// compile with: /EHsc /GR
#include <typeinfo>
#include <iostream>

class A{
public:
    // object for class needs vtable
    // for RTTI
    virtual ~A();
};

using namespace std;
int main() {
A* a = NULL;

try {
    cout << typeid(*a).name() << endl; // Error condition
}
catch (bad_typeid){
    cout << "Object is NULL" << endl;
}
}
```

Выход

Object is NULL

См. также

[Сведения о типах времени выполнения](#)

[Ключевые слова](#)

Класс type_info

12.11.2021 • 2 minutes to read

Класс `type_info` описывает сведения о типах, созданные в программе компилятором. Объекты этого класса эффективно сохраняют указатель на имя для типа. Класс `type_info` также хранит закодированное значение, пригодное для сравнения двух типов на равенство или порядок сортировки. Правила кодирования и последовательность размещения типов не указаны и могут различаться в разных программах.

`<typeinfo>` Для использования класса `type_info` должен быть добавлен файл заголовка. Интерфейс для класса `type_info` :

```
class type_info {
public:
    type_info(const type_info& rhs) = delete; // cannot be copied
    virtual ~type_info();
    size_t hash_code() const;
    _CRTIMP_PURE bool operator==(const type_info& rhs) const;
    type_info& operator=(const type_info& rhs) = delete; // cannot be copied
    _CRTIMP_PURE bool operator!=(const type_info& rhs) const;
    _CRTIMP_PURE int before(const type_info& rhs) const;
    size_t hash_code() const noexcept;
    _CRTIMP_PURE const char* name() const;
    _CRTIMP_PURE const char* raw_name() const;
};
```

Нельзя напрямую создавать экземпляры объектов класса `type_info`, так как класс имеет только закрытый конструктор копии. Единственный способ создать (временный) объект `type_info` — использовать оператор `typeid`. Поскольку оператор присваивания также является закрытым, нельзя копировать или назначать объекты класса `type_info`.

`type_info::hash_code` определяет хэш-функцию, подходящую для сопоставления значений типа `typeInfo` с распределением значений индекса.

Операторы `==` и `!=` могут использоваться для сравнения на равенство и неравенство с другими `type_info` объектами соответственно.

Нет связи между порядком размещения типов и отношениями наследования. Используйте `type_info::before` функцию члена для определения порядка следования типов. Нет никакой гарантии, что `type_info::before` будет давать тот же результат в различных программах или даже в разных запусках одной программы. Таким образом, `type_info::before` аналогичен `(&)` оператору взятия адреса.

`type_info::name` Функция-член возвращает `const char*` в строку, завершающуюся нулем, представляющую понятное имя типа. Указываемая память кэшируется. Ее никогда не следует освобождать напрямую.

`type_info::raw_name` Функция-член возвращает `const char*` в строку, завершающуюся нулем, представляющую декорированное имя типа объекта. Имя фактически хранится в оформленном виде с целью экономии пространства. Следовательно, эта функция работает быстрее, чем `type_info::name` потому, что ей не нужно разменять имя. Стока, возвращаемая `type_info::raw_name` функцией, полезна в операциях сравнения, но недоступна для чтения. Если требуется доступная для человека строка, используйте `type_info::name` вместо нее функцию.

Сведения о типе для полиморфизма создаются только в том случае, если указан параметр компилятора

/GR (Enable Run-Time Type Information) .

См. также

[Сведения о типах среды выполнения](#)

Операторы (C++)

12.11.2021 • 2 minutes to read

Операторы C++ представляют собой элементы программы, которые контролируют способ и порядок обработки объектов. Этот раздел состоит из следующих частей.

- [Обзор](#)
- [Операторы с метками](#)
- Категории операторов
 - [Операторы выражений](#). Эти операторы вычисляют выражение для определения побочных эффектов и возвращаемого значения.
 - [Операторы NULL](#). Эти операторы могут указываться в тех случаях, когда синтаксис C++ требует использования оператора, однако никакие действия не требуются.
 - [Составные инструкции](#). Эти операторы представляют собой группы операторов, окруженные фигурными скобками ({ }). Их можно во всех случаях, где может использоваться отдельный оператор.
 - [Операторы выбора](#). Эти операторы выполняют проверку, а затем, если проверка дает результат true (ненулевое значение), выполняют один из участков кода. Если проверка даст результат false, может выполняться другой участок кода.
 - [Операторы итерации](#). Эти операторы вызывают повторное выполнение блока кода до тех пор, пока не будет выполнен заданный критерий завершения.
 - [Операторы перехода](#). Эти операторы либо передают управление непосредственно в другое место функции, либо возвращают управление из функции.
 - [Операторы объявления](#). Объявления вводят имя в программу.

Сведения об инструкциях по обработке исключений см. в разделе [обработка исключений](#).

См. также

[Справочник по языку C++](#)

Общие сведения об операторах в C++

12.11.2021 • 2 minutes to read

Операторы C++ выполняются последовательно, кроме случаев, когда эта последовательность специально изменяется с помощью оператора выражения, оператора выбора, оператора итерации или оператора перехода.

Операторы могут быть следующих типов:

```
labeled-statement  
expression-statement  
compound-statement  
selection-statement  
iteration-statement  
jump-statement  
declaration-statement  
try-throw-catch
```

В большинстве случаев синтаксис инструкции C++ идентичен тому, что относится к стандарту ANSI C89. Основное различие между ними заключается в том, что в C89 объявления допускаются только в начале блока. C++ добавляет объект `declaration-statement`, который эффективно удаляет это ограничение. Это позволяет вводить переменные в том месте программы, в котором можно вычислить заранее рассчитанное значение инициализации.

Объявление переменных внутри блоков также обеспечивает точный контроль над областью видимости и временем существования этих переменных.

В статьях, посвященных инструкциям, описываются следующие ключевые слова C++:

```
break  
case  
catch  
continue  
default  
do  
  
else  
__except  
__finally  
for  
goto  
  
if  
__if_exists  
__if_not_exists  
__leave  
return  
  
switch  
throw
```

`__try`

`try`

`while`

См. также

[Операторы](#)

Операторы с метками

12.11.2021 • 2 minutes to read

Метки используются для передачи управления программой непосредственно указанному оператору.

```
identifier : statement
case constant-expression : statement
default : statement
```

Областью метки является вся функция, в которой она объявлена.

Remarks

Существует три типа операторов с метками. Во всех для отделения метки от оператора используется двоеточие. Метки case и default предназначены для операторов case.

```
#include <iostream>
using namespace std;

void test_label(int x) {

    if (x == 1){
        goto label1;
    }
    goto label2;

label1:
    cout << "in label1" << endl;
    return;

label2:
    cout << "in label2" << endl;
    return;
}

int main() {
    test_label(1); // in label1
    test_label(2); // in label2
}
```

Оператор goto

Внешний вид метки *идентификатора* в исходной программе объявляет метку. Только оператор [goto](#) может передавать управление метке *идентификатора*. В следующем фрагменте кода показано использование [goto](#) оператора и метки *идентификатора*.

Метка не может отображаться самостоятельно, она всегда прикреплена к оператору. Если необходимо использовать метку самостоятельно, поместите оператор null после метки.

Метка имеет область функции и не может быть повторно объявлена в пределах функции. Однако одно и то же имя может использоваться как метка в разных функциях.

```

// labels_with_goto.cpp
// compile with: /EHsc
#include <iostream>
int main() {
    using namespace std;
    goto Test2;

    cout << "testing" << endl;

    Test2:
    cerr << "At Test2 label." << endl;
}

//Output: At Test2 label.

```

Оператор case

Метки, которые появляются после `case` ключевого слова, также не могут находиться за пределами `switch` оператора. (Это ограничение также применяется к `default` ключевому слову.) В следующем фрагменте кода показано правильное использование `case` МЕТОК.

```

// Sample Microsoft Windows message processing loop.
switch( msg )
{
    case WM_TIMER:      // Process timer event.
        SetClassWord( hWnd, GCW_HICON, ahIcon[nIcon++] );
        ShowWindow( hWnd, SW_SHOWNA );
        nIcon %= 14;
        Yield();
        break;

    case WM_PAINT:
        memset( &ps, 0x00, sizeof(PAINTSTRUCT) );
        hDC = BeginPaint( hWnd, &ps );
        EndPaint( hWnd, &ps );
        break;

    default:
        // This choice is taken for all messages not specifically
        // covered by a case statement.

        return DefWindowProc( hWnd, Message, wParam, lParam );
    break;
}

```

Метки в операторе case

Метки, которые появляются после `case` ключевого слова, также не могут находиться за пределами `switch` оператора. (Это ограничение также применяется к `default` ключевому слову.) В следующем фрагменте кода показано правильное использование `case` МЕТОК.

```
// Sample Microsoft Windows message processing loop.
switch( msg )
{
    case WM_TIMER:      // Process timer event.
        SetClassWord( hWnd, GCW_HICON, ahIcon[nIcon++] );
        ShowWindow( hWnd, SW_SHOWNA );
        nIcon %= 14;
        Yield();
        break;

    case WM_PAINT:
        // Obtain a handle to the device context.
        // BeginPaint will send WM_ERASEBKGND if appropriate.

        memset( &ps, 0x00, sizeof(PAINTSTRUCT) );
        hDC = BeginPaint( hWnd, &ps );

        // Inform Windows that painting is complete.

        EndPaint( hWnd, &ps );
        break;

    case WM_CLOSE:
        // Close this window and all child windows.

        KillTimer( hWnd, TIMER1 );
        DestroyWindow( hWnd );
        if ( hWnd == hWndMain )
            PostQuitMessage( 0 ); // Quit the application.
        break;

    default:
        // This choice is taken for all messages not specifically
        // covered by a case statement.

        return DefWindowProc( hWnd, Message, wParam, lParam );
    break;
}
```

Метки в операторе goto

Внешний вид метки *идентификатора* в исходной программе объявляет метку. Только оператор [goto](#) может передавать управление метке *идентификатора*. В следующем фрагменте кода показано использование `goto` оператора и метки *идентификатора*.

Метка не может отображаться самостоятельно, она всегда прикреплена к оператору. Если необходимо использовать метку самостоятельно, поместите оператор `null` после метки.

Метка имеет область функции и не может быть повторно объявлена в пределах функции. Однако одно и то же имя может использоваться как метка в разных функциях.

```
// labels_with_goto.cpp
// compile with: /EHsc
#include <iostream>
int main() {
    using namespace std;
    goto Test2;

    cout << "testing" << endl;

    Test2:
    cerr << "At Test2 label." << endl;
    // At Test2 label.
}
```

См. также раздел

[Общие сведения о инструкциях C++](#)

[Оператор Switch \(C++\)](#)

Оператор выражений

12.11.2021 • 2 minutes to read

Операторы выражений обеспечивают вычисление выражений. В результате выполнения оператора выражения ни передачи управления, ни итерации не происходит.

Синтаксис оператора выражения простой:

Синтаксис

```
[expression] ;
```

Remarks

Вычисление всех выражений в операторе выражения и учет всех побочных эффектов осуществляются до выполнения следующего оператора. Самые распространенные операторы выражений — присваивания и вызовы функций. Поскольку выражение является необязательным, только точка с запятой считается пустым оператором выражения, называемой оператором `null`.

См. также

[Общие сведения о инструкциях C++](#)

Оператор NULL

12.11.2021 • 2 minutes to read

Оператор null является оператором выражения с отсутствующим *выражением*. Она полезна, если синтаксис языка требует инструкции, но не оценки выражения. Она состоит из точки с запятой.

Инструкции null часто используются как местозаполнители в инструкциях итерации или как инструкции, в которых нужно разместить метки в конце сложных инструкций или функций.

В следующем фрагменте кода показано, как копировать одну строку в другую. Кроме того, код содержит инструкцию null.

```
// null_statement.cpp
char *myStrCpy( char *Dest, const char *Source )
{
    char *DestStart = Dest;

    // Assign value pointed to by Source to
    // Dest until the end-of-string 0 is
    // encountered.
    while( *Dest++ = *Source++ )
        ;    // Null statement.

    return DestStart;
}

int main()
{
}
```

См. также

[Оператор выражения](#)

Составные операторы (блоки)

12.11.2021 • 2 minutes to read

Составной оператор состоит из нуля или более операторов, заключенных в фигурные скобки ({}).

Составной оператор может использоваться везде, где ожидается оператор. Составные инструкции часто называются "блоками".

Синтаксис

```
{ [ statement-list ] }
```

Remarks

В следующем примере составной оператор *используется в качестве части оператора if* инструкции (Дополнительные сведения о синтаксисе см. в [операторе if](#)):

```
if( Amount > 100 )
{
    cout << "Amount was too large to handle\n";
    Alert();
}
else
{
    Balance -= Amount;
}
```

NOTE

Поскольку объявление является оператором, объявление может быть одной из инструкций в *списке инструкций*. Таким образом, имена, которые были объявлены в составном операторе, но не были явно объявлены как статичные, имеют локальную область видимости, а объекты — еще и локальное время существования. Дополнительные сведения об обработке имен с локальной областью см. в разделе [область](#).

См. также

[Общие сведения о инструкциях C++](#)

Операторы выбора (C++)

12.11.2021 • 2 minutes to read

Инструкции выбора C++, [If](#) и [If](#), предоставляют средства для условного выполнения разделов кода.

Операторы [__if_exists](#) и [__if_not_exists](#) позволяют условно включать код в зависимости от существования символа.

Синтаксис этих операторов см. в соответствующих разделах.

См. также:

[Общие сведения о инструкциях C++](#)

оператор if-else (C++)

12.11.2021 • 2 minutes to read

Оператор if-else управляет условным ветвлением. Операторы в `if-branch` выполняются, только если `condition` результатом вычисления является ненулевое значение (или `true`). Если значение `condition` не равно нулю, выполняется следующая инструкция, а инструкция, следующая за необязательным, `else` пропускается. В противном случае пропускается Следующая инструкция, и, если имеется `else` оператор после оператора, `else` выполняется инструкция.

`condition` выражения, принимающие ненулевые значения:

- `true`
- указатель, отличный от `NULL`,
- любое ненулевое арифметическое значение или
- тип класса, определяющий однозначное преобразование в арифметический, логический или тип указателя. (Дополнительные сведения о преобразованиях см. в разделе [стандартные преобразования](#).)

Синтаксис

```
init-statement :  
    expression-statement  
    simple-declaration  
  
condition :  
    expression  
    attribute-specifier-seq неявное согласие decl-specifier-seq declarator ** brace-or-equal-initializer  
  
statement :  
    expression-statement  
    compound-statement  
  
expression-statement :  
    expression неявное согласие ;  
  
compound-statement :  
    { statement-seq неявное согласие }  
  
statement-seq :  
    statement  
    statement-seq statement  
  
if-branch :  
    statement  
  
else-branch :  
    statement  
  
selection-statement :  
    if constexpr отказOPT17 ( init-statement OPT17 condition ) if-branch  
    if constexpr отказOPT17 ( init-statement OPT17 condition ) if-branch else else-branch
```

¹⁷ этот необязательный элемент доступен начиная с c++ 17.

операторы if-else

Для всех форм `if` инструкции, `condition` которая может иметь любое значение, кроме структуры, вычисляется, включая все побочные эффекты. Управление передается из `if` оператора в следующий оператор в программе, если только не выполняется `if-branch` или не `else-branch` содержит `break`, `continue` или `goto`.

`else` Предложение `if...else` оператора связано с ближайшим предыдущим `if` оператором в той же области, у которой нет соответствующей `else` инструкции.

Пример

В этом примере кода показано `if`. Использование нескольких использованных операторов как с, так и без него `else`:

```
// if_else_statement.cpp
#include <iostream>

using namespace std;

class C
{
public:
    void do_something(){}
};

void init(C){}
bool is_true() { return true; }
int x = 10;

int main()
{
    if (is_true())
    {
        cout << "b is true!\n"; // executed
    }
    else
    {
        cout << "b is false!\n";
    }

    // no else statement
    if (x == 10)
    {
        x = 0;
    }

    C* c;
    init(c);
    if (c)
    {
        c->do_something();
    }
    else
    {
        cout << "c is null!\n";
    }
}
```

Оператор If с инициализатором

Начиная с C++ 17, `if` оператор может также содержать `init-statement` выражение, которое объявляет и инициализирует именованную переменную. Используйте эту форму оператора if, если переменная необходима только в области действия оператора if. Для Microsoft эта форма доступна начиная с Visual Studio 2017 версии 15,3, и для нее требуется по крайней мере `/std:c++17` параметр компилятора.

Пример

```
#include <iostream>
#include <mutex>
#include <map>
#include <string>
#include <algorithm>

using namespace std;

map<int, string> m;
mutex mx;
bool shared_flag; // guarded by mx
void unsafe_operation() {}

int main()
{
    if (auto it = m.find(10); it != m.end())
    {
        cout << it->second;
        return 0;
    }

    if (char buf[10]; fgets(buf, 10, stdin))
    {
        m[0] += buf;
    }

    if (lock_guard<mutex> lock(mx); shared_flag)
    {
        unsafe_operation();
        shared_flag = false;
    }

    string s{ "if" };
    if (auto keywords = { "if", "for", "while" }; any_of(keywords.begin(), keywords.end(), [&s](const char* kw) { return s == kw; } ))
    {
        cout << "Error! Token must not be a keyword\n";
    }
}
```

If constexpr операторы

Начиная с C++ 17, можно использовать `if constexpr` инструкцию в шаблонах функций, чтобы принимать решения о ветвлении во время компиляции без необходимости прибегать к нескольким перегрузкам функций. Для Microsoft эта форма доступна начиная с Visual Studio 2017 версии 15,3, и для нее требуется по крайней мере `/std:c++17` параметр компилятора.

Пример

В этом примере показано, как можно написать одну функцию, которая обрабатывает распаковку параметров. Никаких перегрузок с нулевым параметром не требуется:

```
template <class T, class... Rest>
void f(T&& t, Rest&&... r)
{
    // handle t
    do_something(t);

    // handle r conditionally
    if constexpr (sizeof...(r))
    {
        f(r...);
    }
    else
    {
        g(r...);
    }
}
```

См. также раздел

[Операторы выбора](#)

[Словами](#)

[Оператор `switch` \(C++\)](#)

Оператор __if_exists

12.11.2021 • 2 minutes to read

`__if_exists` Оператор проверяет, существует ли указанный идентификатор. Если идентификатор существует, выполняется определенный блок операторов.

Синтаксис

```
__if_exists ( identifier ) {  
    statements  
};
```

Параметры

identifier

Идентификатор, наличие которого требуется проверить.

инструкции

Одна или несколько инструкций для выполнения, если *идентификатор* существует.

Remarks

Caution

Для достижения наиболее достоверных результатов используйте `__if_exists` инструкцию под следующими ограничениями.

- Примените `__if_exists` инструкцию только к простым типам, а не к шаблонам.
- Примените `__if_exists` инструкцию к идентификаторам как внутри, так и вне класса. Не применайте `__if_exists` инструкцию к локальным переменным.
- Используйте `__if_exists` оператор только в теле функции. За пределами тела функции `__if_exists` инструкция может проверять только полностью определенные типы.
- При проверке перегруженных функций невозможно выполнить проверку определенной формы перегрузки.

Дополнение к `__if_exists` оператору является оператором `__if_not_exists`.

Пример

Обратите внимание, что в этом примере используются шаблоны, что не рекомендуется.

```

// the_if_exists_statement.cpp
// compile with: /EHsc
#include <iostream>

template<typename T>
class X : public T {
public:
    void Dump() {
        std::cout << "In X<T>::Dump()" << std::endl;

        __if_exists(T::Dump) {
            T::Dump();
        }

        __if_not_exists(T::Dump) {
            std::cout << "T::Dump does not exist" << std::endl;
        }
    }
};

class A {
public:
    void Dump() {
        std::cout << "In A::Dump()" << std::endl;
    }
};

class B {};

bool g_bFlag = true;

class C {
public:
    void f(int);
    void f(double);
};

int main() {
    X<A> x1;
    X<B> x2;

    x1.Dump();
    x2.Dump();

    __if_exists(::g_bFlag) {
        std::cout << "g_bFlag = " << g_bFlag << std::endl;
    }

    __if_exists(C::f) {
        std::cout << "C::f exists" << std::endl;
    }

    return 0;
}

```

Вывод

```

In X<T>::Dump()
In A::Dump()
In X<T>::Dump()
T::Dump does not exist
g_bFlag = 1
C::f exists

```

См. также

[Инструкции выбора](#)

[Ключевые слова](#)

[__if_not_exists, инструкция](#)

Оператор __if_not_exists

12.11.2021 • 2 minutes to read

`__if_not_exists` Оператор проверяет, существует ли указанный идентификатор. Если идентификатор не существует, выполняется определенный блок операторов.

Синтаксис

```
__if_not_exists ( identifier ) {  
    statements  
};
```

Параметры

identifier

Идентификатор, наличие которого требуется проверить.

инструкции

Одна или несколько инструкций для выполнения, если *идентификатор* не существует.

Remarks

Caution

Для достижения наиболее достоверных результатов используйте `__if_not_exists` инструкцию под следующими ограничениями.

- Примените `__if_not_exists` инструкцию только к простым типам, а не к шаблонам.
- Примените `__if_not_exists` инструкцию к идентификаторам как внутри, так и вне класса. Не применайте `__if_not_exists` инструкцию к локальным переменным.
- Используйте `__if_not_exists` оператор только в теле функции. За пределами тела функции `__if_not_exists` инструкция может проверять только полностью определенные типы.
- При проверке перегруженных функций невозможно выполнить проверку определенной формы перегрузки.

Дополнение к `__if_not_exists` оператору является оператором `__if_exists`.

Пример

Пример использования см `__if_not_exists` . в разделе [оператор __if_exists](#).

См. также

[Инструкции выбора](#)

[Ключевые слова](#)

[__if_exists, инструкция](#)

Оператор `switch` (C++)

12.11.2021 • 4 minutes to read

Позволяет выбирать между несколькими разделами кода в зависимости от значения целочисленного выражения.

Синтаксис

```
selection-statement :  
    switch ( init-statement opt C++ 17 condition ) statement
```

```
init-statement :  
    expression-statement  
    simple-declaration
```

```
condition :  
    expression  
    attribute-specifier-seq opt decl-specifier-seq declarator brace-or-equal-initializer
```

```
Labeled-statement :  
    case constant-expression : statement  
    default : statement
```

Примечания

Оператор `switch` передает управление одному из `Labeled-statement` в своем теле в зависимости от значения `condition`.

Объект `condition` должен иметь целочисленный тип или быть типом класса с однозначным преобразованием в целочисленный тип. Целочисленное повышение выполняется, как описано в разделе [стандартные преобразования](#).

`switch` Текст инструкции состоит из ряда `case` меток и опт `default` метки ионал. Является `Labeled-statement` одной из этих меток и инструкциями, приведенными ниже. Помеченные операторы не являются синтаксическими требованиями, но `switch` оператор не имеет смысла без них. Ни одно `constant-expression` из двух значений в `case` операторах не может иметь одно и то же значение. `default` Метка может отображаться только один раз. `default` Оператор часто помещается в конец, но может находиться в любом месте `switch` тела инструкции. Метка `case` ИЛИ `default` может располагаться только внутри оператора `switch`.

`constant-expression` В каждой `case` метке преобразуется в постоянное значение, которое имеет тот же тип, что и `condition`. Затем он сравнивается с `condition` для равенства. Управление передается первой инструкции после `case` `constant-expression` значения, совпадающего со значением `condition`. Поведение, полученное в результате, показано в следующей таблице.

`switch` **поведение инструкции**

УСЛОВИЕ	ДЕЙСТВИЕ
Преобразованное значение соответствует значению выражения управления с повышенным уровнем.	Управление передается оператору, следующему за этой меткой.
Ни одна из констант не соответствует константам в <code>case</code> метках; <code>default</code> имеется метка.	Элемент управления передается в <code>default</code> метку.
Ни одна из констант не соответствует константам в <code>case</code> метках; <code>default</code> Метка отсутствует.	Элемент управления передается оператору после <code>switch</code> оператора.

Если найдено совпадающее выражение, выполнение можно продолжить через более поздние `case` или `default` метки. `break` Оператор используется для того, чтобы прерывать выполнение и передавать управление оператору после `switch` инструкции. Без `break` оператора выполняется выполнение всех инструкций из соответствующей `case` метки в конец `switch`, включая `default`. Пример:

```
// switch_statement1.cpp
#include <stdio.h>

int main() {
    const char *buffer = "Any character stream";
    int uppercase_A, lowercase_a, other;
    char c;
    uppercase_A = lowercase_a = other = 0;

    while ( c = *buffer++ ) // Walks buffer until NULL
    {
        switch ( c )
        {
            case 'A':
                uppercase_A++;
                break;
            case 'a':
                lowercase_a++;
                break;
            default:
                other++;
        }
    }
    printf_s( "\nUppercase A: %d\nLowercase a: %d\nTotal: %d\n",
              uppercase_A, lowercase_a, (uppercase_A + lowercase_a + other) );
}
```

В приведенном выше примере `uppercase_A` увеличивается, если `c` является верхним `case 'A'`. `break` Инструкция после `uppercase_A++` завершает выполнение `switch` тела оператора и управление передается в `while` ЦИКЛ. Без `break` оператора выполнение перейдет к следующему оператору с меткой, так что `lowercase_a` И `other` будет также увеличен. Аналогичное назначение обрабатывается `break` оператором для `case 'a'`. Если значение меньше `c` `case 'a'`, `lowercase_a` то увеличивается и `break` оператор завершает `switch` тело оператора. Если `c` не является `'a'` или `'A'`, `default` выполняется инструкция.

Visual Studio 2017 и более поздних версий: (доступно с `/std:c++17`) `[[fallthrough]]` атрибут указан в стандарте `c++17`. Его можно использовать в `switch` операторе. Это подсказка для компилятора или любой, кто читает код, это пошаговое поведение является намеренным. Компилятор Microsoft C++ в настоящее время не предупреждает о поведении `fallthrough`, поэтому этот атрибут не влияет на поведение компилятора. В этом примере атрибут применяется к пустой инструкции в незавершенном операторе с меткой. Иными словами, необходимо поставить точку с запятой.

```
int main()
{
    int n = 5;
    switch (n)
    {

        case 1:
            a();
            break;
        case 2:
            b();
            d();
            [[fallthrough]]; // I meant to do this!
        case 3:
            c();
            break;
        default:
            d();
            break;
    }

    return 0;
}
```

Visual Studio 2017 **версии 15,3 и более поздних** версий (доступно в [/std: c++ 17](#)). `switch` Оператор может содержать `init-statement` предложение, которое заканчивается точкой с запятой. Он вводит и инициализирует переменную, область которой ограничена блоком `switch` оператора:

```
switch (Gadget gadget(args); auto s = gadget.get_status())
{
    case status::good:
        gadget.zip();
        break;
    case status::bad:
        throw BadGadget();
}
```

Внутренний блок `switch` инструкции может содержать определения с инициализаторами, если они **достижимы**, то есть не обходятся всеми возможными путями выполнения. Имена, добавленные с помощью этих объявлений, имеют локальную область видимости. Пример:

```
// switch_statement2.cpp
// C2360 expected
#include <iostream>
using namespace std;
int main(int argc, char *argv[])
{
    switch( tolower( *argv[1] ) )
    {
        // Error. Unreachable declaration.
        char szChEntered[] = "Character entered was: ";

        case 'a' :
        {
            // Declaration of szChEntered OK. Local scope.
            char szChEntered[] = "Character entered was: ";
            cout << szChEntered << "a\n";
        }
        break;

        case 'b' :
        // Value of szChEntered undefined.
        cout << szChEntered << "b\n";
        break;

        default:
        // Value of szChEntered undefined.
        cout << szChEntered << "neither a nor b\n";
        break;
    }
}
```

`switch` Оператор может быть вложенным. При вложении `case` метки или `default` связываются с ближайшим `switch` оператором, в котором они заключены.

Поведение в системах Microsoft

Microsoft C++ не ограничивает количество `case` значений в `switch` операторе. Это число ограничивается только объемом доступной памяти.

См. также раздел

[Инструкции выбора](#)

[Ключевые слова](#)

Операторы перебора (C++)

12.11.2021 • 2 minutes to read

Операторы итерации приводят к тому, что операторы (или составные операторы) выполняются ноль или более раз в соответствии с некоторыми критериями завершения цикла. Если эти инструкции являются составными операторами, они выполняются по порядку, за исключением случаев, когда встречается либо оператор `break`, либо оператор `Continue`.

C++ предоставляет четыре оператора итерации — [хотя](#), [Do](#), [для](#) и [основан на диапазоне для](#). Каждый из этих итераций перебирается до тех пор, пока выражение завершения не примет значение 0 (false) или пока завершение цикла не будет принудительно применено к `break` оператору. В следующей таблице приведены сводные сведения об этих операторах и их действии. Дополнительные сведения о каждом из них см. в последующих разделах.

Операторы итерации

	ВРЕМЯ ВЫЧИСЛЕНИЯ	ИНИЦИАЛИЗАЦИЯ	ПРИРАЩЕНИЕ
<code>while</code>	Начало цикла	Нет	Нет
<code>do</code>	Конец цикла	Нет	Нет
<code>for</code>	Начало цикла	Да	Да
<code>range-based for</code>	Начало цикла	Да	Да

Часть оператора итерации не может быть объявлением. Однако она может быть составным оператором, содержащим объявление.

См. также

[Общие сведения о инструкциях C++](#)

Оператор while (C++)

12.11.2021 • 2 minutes to read

Выполняет *инструкцию* повторно, пока *выражение* не примет значение 0.

Синтаксис

```
while ( expression )
    statement
```

Remarks

Перед каждым выполнением цикла выполняется проверка *выражения*. Таким образом, `while` цикл выполняется ноль или более раз. *выражение* должно иметь целочисленный тип, тип указателя или тип класса с однозначным преобразованием в целочисленный или тип указателя.

`while` Цикл также может завершиться, когда выполняется `break`, `goto` или `return` в теле оператора.

Используйте `Continue`, чтобы завершить текущую итерацию без выхода из `while` цикла. `continue` передает управление следующей итерации `while` цикла.

В следующем коде используется `while` цикл для удаления конечных подчеркивания из строки.

```
// while_statement.cpp

#include <string.h>
#include <stdio.h>
char *trim( char *szSource )
{
    char *pszEOS = 0;

    // Set pointer to character before terminating NULL
    pszEOS = szSource + strlen( szSource ) - 1;

    // iterate backwards until non '_' is found
    while( (pszEOS >= szSource) && (*pszEOS == '_') )
        *pszEOS-- = '\0';

    return szSource;
}
int main()
{
    char szbuf[] = "12345_____";

    printf_s("\nBefore trim: %s", szbuf);
    printf_s("\nAfter trim: %s\n", trim(szbuf));
}
```

Условие завершения вычисляется в начале цикла. Если символов подчеркивания в конце строки нет, цикл никогда не выполняется.

См. также

[Операторы итерации](#)

[Ключевые слова](#)

[Оператор do-while \(C\)](#)

[Оператор for \(C++\)](#)

[Основанное на диапазоне выражение for \(C++\)](#)

Выражение do-while (C++)

12.11.2021 • 2 minutes to read

Выполняет *инструкцию* повторно, пока указанное условие завершения (*выражение*) не примет значение 0.

Синтаксис

```
do
    statement
  while ( expression ) ;
```

Remarks

Проверка условия завершения выполняется после каждого выполнения цикла. Таким образом, цикл *do-while* выполняется один или несколько раз в зависимости от значения выражения завершения.

Выполнение оператора *do-while* также может прерваться, если в теле оператора выполняется оператор *break*, *goto* или *return*.

Выражение *expression* должно иметь арифметический тип или тип указателя. Выполнение происходит следующим образом:

1. Выполняется тело оператора.
2. Затем вычисляется значение *expression*. Если выражение *expression* имеет значение *false*, выполнение оператора *do-while* завершается и управление передается следующему оператору программы. Если *expression* имеет значение *true* (то есть не равно нулю), процесс повторяется с шага 1.

Пример

В следующем примере показан оператор *do-while*:

```
// do_while_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        printf_s("\n%d", i++);
    } while (i < 3);
}
```

См. также

[Операторы итерации](#)

[Ключевые слова](#)

[Оператор while \(C++\)](#)

[Оператор for \(C++\)](#)

[Основанное на диапазоне выражение for \(C++\)](#)

Оператор `for` (C++)

12.11.2021 • 3 minutes to read

Выполняет оператор повторно до тех пор, пока условное значение не станет `false`. Сведения об операторе на основе диапазона `for` см. в разделе [оператор на основе диапазона `for` \(C++\)](#).

Синтаксис

```
for ( init-expression ; cond-expression ; Loop-expression )  
    statement
```

Remarks

Используйте `for` инструкцию для создания циклов, которые должны выполняться указанное число раз.

`for` Инструкция состоит из трех дополнительных частей, как показано в следующей таблице.

элементы цикла `for`

ИМЯ СИНТАКСИСА	ПРИ ВЫПОЛНЕНИИ	ОПИСАНИЕ
<code>init-expression</code>	Перед любым другим элементом инструкции выполняется <code>for</code> <code>init-expression</code> только один раз. Затем управление передается в <code>cond-expression</code> .	Часто используется для инициализации индексов цикла. Может содержать выражения или объявления.
<code>cond-expression</code>	Перед выполнением каждой итерации <code>statement</code> , включая первую итерацию. <code>statement</code> выполняется, только если <code>cond-expression</code> имеет значение <code>true</code> (отличное от нуля).	Выражение, значение которого относится к целочисленному типу или типу класса, для которого имеется однозначное преобразование к целочисленному типу. Обычно используется для проверки критериев завершения цикла <code>for</code> .
<code>Loop-expression</code>	В конце каждой итерации <code>statement</code> . После <code>Loop-expression</code> выполнения <code>cond-expression</code> вычисляется.	Обычно используется для приращения индексов цикла.

В следующих примерах показаны различные способы использования `for` инструкции.

```

#include <iostream>
using namespace std;

int main() {
    // The counter variable can be declared in the init-expression.
    for (int i = 0; i < 2; i++ ){
        cout << i;
    }
    // Output: 01
    // The counter variable can be declared outside the for loop.
    int i;
    for (i = 0; i < 2; i++){
        cout << i;
    }
    // Output: 01
    // These for loops are the equivalent of a while loop.
    i = 0;
    while (i < 2){
        cout << i++;
    }
    // Output: 01
}

```

`init-expression` И `Loop-expression` могут содержать несколько инструкций, разделенных запятыми.

Пример:

```

#include <iostream>
using namespace std;

int main(){
    int i, j;
    for ( i = 5, j = 10 ; i + j < 20; i++, j++ ) {
        cout << "i + j = " << (i + j) << '\n';
    }
}
// Output:
i + j = 15
i + j = 17
i + j = 19

```

`Loop-expression` можно увеличить или уменьшить или изменить другими способами.

```

#include <iostream>
using namespace std;

int main(){
for (int i = 10; i > 0; i--) {
    cout << i << ' ';
}
// Output: 10 9 8 7 6 5 4 3 2 1
for (int i = 10; i < 20; i = i+2) {
    cout << i << ' ';
}
// Output: 10 12 14 16 18

```

`for` Цикл завершается `break`, когда выполняется, возвращаю или `goto` (оператор с меткой вне `for` цикла) внутри `statement`. `continue` Оператор в `for` цикле завершает только текущую итерацию.

Если `cond-expression` аргумент опущен, то считается, что `true` `for` цикл не завершается без оператора `break`, `return` или `goto` В `statement`.

Хотя три поля `for` инструкции обычно используются для инициализации, тестирования завершения и увеличения, они не ограничиваются этими применениеми. Например, следующий код выводит числа от 0 до 4. В этом случае `statement` является оператором NULL:

```
#include <iostream>
using namespace std;

int main()
{
    int i;
    for( i = 0; i < 5; cout << i << '\n', i++){
        ;
    }
}
```

for ЦИКЛЫ И СТАНДАРТ C++

Стандарт C++ говорит о том, что переменная, объявленная в `for` цикле, должна выйти из области действия после `for` завершения цикла. Пример:

```
for (int i = 0 ; i < 5 ; i++) {
    // do something
}
// i is now out of scope under /Za or /Zc:forScope
```

По умолчанию в параметре `/Za` переменная, объявленная в `for` цикле, остается в области видимости до `for` конца охватывающей области цикла.

`/Zc:forScope` обеспечивает стандартное поведение переменных, объявленных в цикле `for`, без необходимости указывать `/za`.

Можно также использовать различия в области `for` действия цикла для повторного объявления переменных в `/ze` следующим образом:

```
// for_statement5.cpp
int main(){
    int i = 0;    // hidden by var with same name declared in for loop
    for ( int i = 0 ; i < 3; i++ ) {}

    for ( int i = 0 ; i < 3; i++ ) {}
}
```

Такое поведение более точно имитирует стандартное поведение переменной, объявленной в `for` цикле, что требует, чтобы переменные, объявленные в `for` цикле, выходят за пределы области действия после завершения цикла. При объявлении переменной в `for` цикле компилятор внутренне переводит его в локальную переменную в `for` области видимости цикла. Она повышается, даже если уже существует локальная переменная с таким же именем.

См. также

[Инструкции итерации](#)

[Ключевые слова](#)

[оператор while \(C++\)](#)

[Оператор do-while \(C++\)](#)

[Основанный на диапазоне оператор for \(C++\)](#)

Основанное на диапазоне выражение for (C++)

12.11.2021 • 2 minutes to read

Циклически и последовательно выполняет оператор (`statement`) каждого элемента в выражении (`expression`).

Синтаксис

```
for ( объявление для- диапазона : выражение** ) **  
Оператор
```

Remarks

Используйте оператор на основе диапазона `for` для создания циклов, которые должны выполняться с помощью *диапазона*, который определяется как то, что можно перебирать, например, `std::vector` или любую другую последовательность стандартной библиотеки C++, диапазон которой определяется `begin()` и `end()`. Имя, объявленное в части, `for-range-declaration` является локальным для `for` инструкции и не может быть повторно объявлено в `expression` ИЛИ `statement`. Обратите внимание, что `auto` ключевое слово является предпочтительным в `for-range-declaration` части инструкции.

новые Visual Studio 2017: Для циклов на основе диапазонов `for` больше не требуется, чтобы `begin()` и `end()` возвращать объекты одного и того же типа. Это позволяет `end()` возвращать объект Sentinel, например, используемый диапазонами, как определено в предложении Ranges-v3. Дополнительные сведения см. в разделе [generalize For цикл Range-Based](#) и [библиотеку range-V3 на GitHub](#).

В этом коде показано, как использовать циклы на основе диапазона `for` для прохода по массиву и вектору:

```

// range-based-for.cpp
// compile by using: cl /EHsc /nologo /W4
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // Basic 10-element integer array.
    int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

    // Range-based for loop to iterate through the array.
    for( int y : x ) { // Access by value using a copy declared as a specific type.
        // Not preferred.
        cout << y << " ";
    }
    cout << endl;

    // The auto keyword causes type inference to be used. Preferred.

    for( auto y : x ) { // Copy of 'x', almost always undesirable
        cout << y << " ";
    }
    cout << endl;

    for( auto &y : x ) { // Type inference by reference.
        // Observes and/or modifies in-place. Preferred when modify is needed.
        cout << y << " ";
    }
    cout << endl;

    for( const auto &y : x ) { // Type inference by const reference.
        // Observes in-place. Preferred when no modify is needed.
        cout << y << " ";
    }
    cout << endl;
    cout << "end of integer array test" << endl;
    cout << endl;

    // Create a vector object that contains 10 elements.
    vector<double> v;
    for (int i = 0; i < 10; ++i) {
        v.push_back(i + 0.14159);
    }

    // Range-based for loop to iterate through the vector, observing in-place.
    for( const auto &j : v ) {
        cout << j << " ";
    }
    cout << endl;
    cout << "end of vector test" << endl;
}

```

Результат выглядит так:

```

1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
end of integer array test

0.14159 1.14159 2.14159 3.14159 4.14159 5.14159 6.14159 7.14159 8.14159 9.14159
end of vector test

```

Цикл на основе диапазона `for` завершается, когда выполняется один из этих элементов `statement`: а `break`, `return` или `goto` в помеченную инструкцию за пределами цикла, основанного на диапазоне `for`.
• `continue` Оператор в цикле на основе диапазона `for` завершает только текущую итерацию.

Помните о таких фактах, как основано на диапазоне `for`:

- Такие циклы автоматически распознают массивы.
- Такие циклы автоматически распознают контейнеры с методами `.begin()` и `.end()`.
- Для всех остальных итераторов в них используются поиск, зависящий от аргументов (`.begin()` и `.end()`).

См. также

`auto`

[Операторы итерации](#)

[Ключевые слова](#)

[Оператор `while` \(C++\)](#)

[Оператор `do-while` \(C++\)](#)

[Оператор `for` \(C++\)](#)

Операторы перехода (C++)

12.11.2021 • 2 minutes to read

Оператор перехода C++ выполняет немедленную локальную передачу контроля.

Синтаксис

```
break;  
continue;  
return [expression];  
goto identifier;
```

Remarks

См. описание операторов перехода C++ в следующих разделах.

- [Оператор break](#)
- [Оператор continue](#)
- [Оператор return](#)
- [Оператор GoTo](#)

См. также

[Общие сведения о инструкциях C++](#)

Оператор break (C++)

12.11.2021 • 2 minutes to read

`break` Оператор завершает выполнение ближайшего включающего цикла или условного оператора, в котором он отображается. Управление передается оператору, который расположен после оператора, при его наличии.

Синтаксис

```
break;
```

Remarks

`break` Оператор используется с оператором условного [переключения](#) и с операторами `Do`, `for` и `while`.

В `switch` операторе `break` инструкция заставляет программу выполнить следующую инструкцию за пределами `switch` оператора. Без `break` инструкции выполняется каждая инструкция из сопоставленной `case` метки в конец `switch` оператора, включая `default` предложение.

В циклах `break` оператор завершает выполнение ближайшего включающего `do`, `for` оператора, или `while`. Управление передается оператору, который расположен после завершенного оператора, при его наличии.

Внутри вложенных операторов `break` оператор завершает только `do`, `for` оператор, `switch` или `while`, который непосредственно заключает его в блок. Оператор или можно использовать `return` `goto` для перемещения управления из более глубоко вложенных структур.

Пример

В следующем коде показано, как использовать `break` инструкцию в `for` цикле.

```

#include <iostream>
using namespace std;

int main()
{
    // An example of a standard for loop
    for (int i = 1; i < 10; i++)
    {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
    }

    // An example of a range-based for loop
    int nums []{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    for (int i : nums) {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
    }
}

```

In each case:

1
2
3

В следующем коде показано, как использовать `break` в `while` цикле и в `do` цикле.

```

#include <iostream>
using namespace std;

int main() {
    int i = 0;

    while (i < 10) {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
        i++;
    }

    i = 0;
    do {
        if (i == 4) {
            break;
        }
        cout << i << '\n';
        i++;
    } while (i < 10);
}

```

In each case:

0123

В следующем коде показано, как использовать `break` в операторе `switch`. Необходимо использовать `break` в каждом случае, если нужно обменять каждый вариант отдельно. Если не использовать `break`,

выполнение кода будет проходить до следующего варианта.

```
#include <iostream>
using namespace std;

enum Suit{ Diamonds, Hearts, Clubs, Spades };

int main() {

    Suit hand;
    . . .
    // Assume that some enum value is set for hand
    // In this example, each case is handled separately
    switch (hand)
    {
        case Diamonds:
            cout << "got Diamonds \n";
            break;
        case Hearts:
            cout << "got Hearts \n";
            break;
        case Clubs:
            cout << "got Clubs \n";
            break;
        case Spades:
            cout << "got Spades \n";
            break;
        default:
            cout << "didn't get card \n";
    }
    // In this example, Diamonds and Hearts are handled one way, and
    // Clubs, Spades, and the default value are handled another way
    switch (hand)
    {
        case Diamonds:
        case Hearts:
            cout << "got a red card \n";
            break;
        case Clubs:
        case Spades:
        default:
            cout << "didn't get a red card \n";
    }
}
```

См. также раздел

[Операторы перехода](#)

[Ключевые слова](#)

[Оператор continue](#)

Оператор continue (C++)

12.11.2021 • 2 minutes to read

Принудительно передает управление управляющему выражению наименьшего включающего цикла [Do](#), [for](#) или [while](#).

Синтаксис

```
continue;
```

Remarks

Все остальные операторы текущей итерации не выполняются. Следующая итерация цикла определяется следующим образом.

- В `do while` цикле или Следующая итерация начинается с переоценки управляющего выражения `do while` оператора или.
- В `for` цикле (с использованием синтаксиса `for(<init-expr> ; <cond-expr> ; <loop-expr>)`) `<loop-expr>` выполняется предложение. Затем повторно выполняется предложение `<cond-expr>` и, в зависимости от результата, цикл завершается или начинается другая итерация.

В следующем примере показано, как `continue` можно использовать инструкцию для обхода частей кода и начать следующую итерацию цикла.

Пример

```
// continue_statement.cpp
#include <stdio.h>
int main()
{
    int i = 0;
    do
    {
        i++;
        printf_s("before the continue\n");
        continue;
        printf("after the continue, should never print\n");
    } while (i < 3);

    printf_s("after the do loop\n");
}
```

```
before the continue
before the continue
before the continue
after the do loop
```

См. также раздел

[Операторы перехода](#)

Ключевые слова

Оператор return (C++)

12.11.2021 • 2 minutes to read

Завершает выполнение функции и возвращает элемент управления в вызывающую функцию (или в операционную систему при передаче управления из функции `main`). Выполнение возобновляется в вызывающей функции в точке сразу после вызова.

Синтаксис

```
return [expression];
```

Remarks

Предложение `expression`, при его наличии, преобразуется в тип, указанный в объявлении функции, как если бы выполнялась инициализация. Преобразование из типа выражения в `return` тип функции может создавать временные объекты. Дополнительные сведения о том, как и когда создаются долгим сроком, см. в разделе [временные объекты](#).

Значение предложения `expression` возвращается в вызывающую функцию. Если выражение пропущено, то возвращаемое значение функции не определено. Конструкторы, деструкторы и функции типа `void` не могут указывать выражение в `return` инструкции. Функции всех других типов должны указывать выражение в `return` операторе.

Когда поток управления выходит из блока, содержащего определение функции, результат будет таким же, как при `return` выполнении инструкции без выражения. Это недопустимо для функций, объявленных как возвращающие значение.

У функции может быть любое количество `return` инструкций.

В следующем примере выражение с `return` оператором используется для получения самого крупного из двух целых чисел.

Пример

```
// return_statement2.cpp
#include <stdio.h>

int max ( int a, int b )
{
    return ( a > b ? a : b );
}

int main()
{
    int nOne = 5;
    int nTwo = 7;

    printf_s("\n%d is bigger\n", max( nOne, nTwo ) );
}
```

См. также

[Операторы перехода](#)

[Ключевые слова](#)

Оператор goto (C++)

12.11.2021 • 2 minutes to read

`goto` Оператор безусловно передает управление оператору, помеченному указанным идентификатором.

Синтаксис

```
goto identifier;
```

Remarks

Оператор, метка которого задана в параметре `identifier`, должен находиться в текущей функции. Все имена, заданные в параметре `identifier`, являются членами внутреннего пространства имен и, следовательно, не пересекаются с другими идентификаторами.

Метка оператора имеет смысл только для `goto` инструкции; в противном случае метки операторов игнорируются. Повторное объявление меток невозможно.

`goto` Оператору не разрешено передавать управление в расположение, которое пропускает инициализацию любой переменной, которая находится в области этого расположения. В следующем примере создается C2362:

```
int goto_fn(bool b)
{
    if (!b)
    {
        goto exit; // C2362
    }
    else
    { /*...*/ }

    int error_code = 42;

exit:
    return error_code;
}
```

Рекомендуется использовать `break` `continue` операторы, и `return` вместо `goto` инструкции, когда это возможно. Однако, поскольку `break` инструкция завершается только из одного уровня цикла, может потребоваться использовать `goto` оператор для выхода из глубоко вложенного цикла.

Дополнительные сведения о метках и `goto` инструкции см. в разделе [Операторы с метками](#).

Пример

В этом примере `goto` оператор передает управление в точку, помеченную `stop` когда `i` равно 3.

```
// goto_statement.cpp
#include <stdio.h>
int main()
{
    int i, j;

    for ( i = 0; i < 10; i++ )
    {
        printf_s( "Outer loop executing. i = %d\n", i );
        for ( j = 0; j < 2; j++ )
        {
            printf_s( " Inner loop executing. j = %d\n", j );
            if ( i == 3 )
                goto stop;
        }
    }

    // This message does not print:
    printf_s( "Loop exited. i = %d\n", i );

stop:
    printf_s( "Jumped to stop. i = %d\n", i );
}
```

```
Outer loop executing. i = 0
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 1
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 2
Inner loop executing. j = 0
Inner loop executing. j = 1
Outer loop executing. i = 3
Inner loop executing. j = 0
Jumped to stop. i = 3
```

См. также

[Операторы перехода](#)

[Ключевые слова](#)

Передача управления

12.11.2021 • 2 minutes to read

Можно использовать `goto` оператор или `case` метку в `switch` операторе, чтобы указать программе, которая выполняет ветвление после инициализатора. Такой код не допускается, если объявление, содержащее инициализатор, не будет находиться в блоке, заключенном в блоке, в котором выполняется оператор `jmp`.

В следующем примере показан цикл, в котором выполняется объявление и инициализация объектов `total`, `ch` и `i`. Существует также ошибочный `goto` оператор, который передает управление после инициализатора.

```
// transfers_of_control.cpp
// compile with: /W1
// Read input until a nonnumeric character is entered.
int main()
{
    char MyArray[5] = {'2','2','a','c'};
    int i = 0;
    while( 1 )
    {
        int total = 0;

        char ch = MyArray[i++];

        if ( ch >= '0' && ch <= '9' )
        {
            goto Label1;

            int i = ch - '0';
        Label1:
            total += i;    // C4700: transfers past initialization of i.
        } // i would be destroyed here if goto error were not present
    else
        // Break statement transfers control out of loop,
        // destroying total and ch.
        break;
    }
}
```

В предыдущем примере `goto` Инструкция пытается переслать управление после инициализации `i`. Однако если бы объект `i` был объявлен, но не инициализирован, передача была бы допустима.

Объекты `total` и `ch`, объявленные в блоке, который выступает в качестве *инструкции* `while` инструкции, уничтожаются при выходе из этого блока с помощью `break` инструкции.

Пространства имен (C++)

12.11.2021 • 6 minutes to read

Пространство имен — это декларативная область, в рамках которой определяются различные идентификаторы (имена типов, функций, переменных, и т. д.). Пространства имен используются для организации кода в виде логических групп и с целью избежания конфликтов имен, которые могут возникнуть, особенно в таких случаях, когда база кода включает несколько библиотек. Все идентификаторы в пределах пространства имен доступны друг другу без уточнения. Идентификаторы за пределами пространства имен могут обращаться к членам с помощью полного имени для каждого идентификатора, например `std::vector<std::string> vec;`, или else с помощью объявления `using` для одного идентификатора (`using std::string`) или директивы `using` для всех идентификаторов в пространстве имен (`using namespace std;`). Код в файлах заголовков всегда должен содержать полное имя в пространстве имен.

В следующем примере показано объявление пространства имен и продемонстрированы три способа доступа к членам пространства имен из кода за его пределами.

```
namespace ContosoData
{
    class ObjectManager
    {
    public:
        void DoSomething() {}
    };
    void Func(ObjectManager) {}
}
```

Использование полного имени:

```
ContosoData::ObjectManager mgr;
mgr.DoSomething();
ContosoData::Func(mgr);
```

Чтобы добавить в область видимости один идентификатор, используйте объявление `using`:

```
using ContosoData::ObjectManager;
ObjectManager mgr;
mgr.DoSomething();
```

Чтобы добавить в область видимости все идентификаторы пространства имен, используйте директиву `using`:

```
using namespace ContosoData;

ObjectManager mgr;
mgr.DoSomething();
Func(mgr);
```

директивы `using`

`using` Директива позволяет использовать все имена в `namespace` для использования без *имени*

пространства имен в качестве явного квалификатора. Использование директивы `using` в файле реализации (т. е. `*.cpp`) при использовании нескольких различных идентификаторов в пространстве имен; Если вы используете только один или два идентификатора, рассмотрите использование объявления `using`, чтобы привести эти идентификаторы в область, а не все идентификаторы в пространстве имен. Если локальная переменная имеет такое же имя, как и переменная пространства имен, то переменная пространства имен будет скрытой. Создавать переменную пространства имен с тем же именем, что и у глобальной переменной, является ошибкой.

NOTE

Директиву `using` можно поместить в верхнюю часть CPP-файла (в область видимости файла) или внутрь определения класса или функции.

Без особой необходимости не размещайте директивы `using` в файлах заголовков (`*.h`), так как любой файл, содержащий этот заголовок, добавит все идентификаторы пространства имен в область видимости, что может вызвать скрытие или конфликты имен, которые очень трудно отлаживать. В файлах заголовков всегда используйте полные имена. Если эти имена получаются слишком длинными, используйте псевдоним пространства имен для их сокращения. (См. ниже.)

Объявление пространств имен и их членов

Как правило, пространство имен объявляется в файле заголовка. Если реализации функций находятся в отдельном файле, определяйте имена функций полностью, как показано в следующем примере.

```
//contosoData.h
#pragma once
namespace ContosoDataServer
{
    void Foo();
    int Bar();
}
```

Реализации функций в контосодата. cpp должны использовать полное имя, даже если поместить `using` директиву в начало файла:

```
#include "contosodata.h"
using namespace ContosoDataServer;

void ContosoDataServer::Foo() // use fully-qualified name here
{
    // no qualification needed for Bar()
    Bar();
}

int ContosoDataServer::Bar(){return 0;}
```

Пространство имен может быть объявлено в нескольких блоках в одном файле и в нескольких файлах. Компилятор соединит вместе все части во время предварительной обработки и полученное в результате пространство имен будет содержать все члены, объявленные во всех частях. Примером этого является пространство имен `std`, которое объявляется в каждом из файлов заголовка в стандартной библиотеке.

Члены именованного пространства имен могут определяться за его границами, если они объявлены путем явной квалификации определяемого пространства имен. Однако определение должно располагаться после точки объявления в пространстве имен, окружающем то пространство имен, где находится объявление. Пример:

```
// defining_namespace_members.cpp
// C2039 expected
namespace V {
    void f();
}

void V::f() { }           // ok
void V::g() { }           // C2039, g() is not yet a member of V

namespace V {
    void g();
}
```

Эта ошибка может возникнуть, когда члены пространства имен объявляются в нескольких файлах заголовка и эти заголовки не включены в правильном порядке.

Глобальное пространство имен

Если идентификатор не объявлен явно в пространстве имен, он неявно считается входящим в глобальное пространство имен. В общем случае старайтесь не делать объявления в глобальной области, если это возможно, за исключением [функции Main](#) точки входа, которая должна находиться в глобальном пространстве имен. Чтобы явно указать глобальный идентификатор, используйте оператор разрешения области видимости без имени, как сделано в `::SomeFunction(x);`. Это позволит отличать данный идентификатор от любого другого элемента с таким же именем, находящегося в другом пространстве имен. Кроме того, это облегчит понимание кода.

Пространство имен std

Все типы и функции стандартной библиотеки C++ объявляются в `std` пространстве имен или пространствах имен, вложенных в `std`.

Вложенные пространства имен

Пространства имен могут быть вложенными. Обычное вложенное пространство имен имеет неполный доступ к членам родительского элемента, но родительские элементы не имеют неполного доступа к вложенному пространству имен (если только оно не объявлено как встроенное), как показано в следующем примере:

```
namespace ContosoDataServer
{
    void Foo();

    namespace Details
    {
        int CountImpl;
        void Bar() { return Foo(); }
    }

    int Baz(int i) { return Details::CountImpl; }
}
```

Обычные вложенные пространства имен можно использовать для инкапсуляции данных о внутренней реализации, которые не являются частью открытого интерфейса родительского пространства имен.

Встроенные пространства имен (C++ 11)

В отличие от обычных вложенных пространств имен члены встроенного пространства имен обрабатываются как члены родительского пространства имен. Эта особенность позволяет выполнять поиск перегруженных функций с зависимостью от аргументов среди функций, которые имеют перегрузки в родительском и вложенном встроенном пространстве имен. Это также позволяет объявлять специализации в родительском пространстве имен для шаблонов, объявленных во встроенном пространстве имен. В следующем примере показано, как внешний код привязывается к встроенному пространству имен по умолчанию.

```
//Header.h
#include <string>

namespace Test
{
    namespace old_ns
    {
        std::string Func() { return std::string("Hello from old"); }
    }

    inline namespace new_ns
    {
        std::string Func() { return std::string("Hello from new"); }
    }
}

#include "header.h"
#include <string>
#include <iostream>

int main()
{
    using namespace Test;
    using namespace std;

    string s = Func();
    std::cout << s << std::endl; // "Hello from new"
    return 0;
}
```

В следующем примере показано, как можно объявить специализацию в родительском пространстве имен шаблона, объявленного во встроенном пространстве имен.

```
namespace Parent
{
    inline namespace new_ns
    {
        template <typename T>
        struct C
        {
            T member;
        };
    }
    template<>
    class C<int> {};
}
```

Встроенные пространства имен можно использовать как механизм управления версиями для управления изменениями в открытом интерфейсе библиотеки. Например, можно создать одно родительское пространство имен и инкапсулировать каждую версию интерфейса в своем собственном пространстве имен, вложенном в родительское. Пространство имен, которое содержит самую последнюю или основную версию, квалифицируется как встроенное и поэтому представляется так, будто оно является непосредственным членом родительского пространства имен. Клиентский код, вызывающий Parent::Class,

автоматически привязывается к новому коду. Клиенты, которые предпочитают использовать старую версию, могут по-прежнему получить доступ к ней, используя полный путь к вложенному пространству имен, содержащему данный код.

Ключевое слово `inline` должно применяться к первому объявлению пространства имен в единице компиляции.

В следующем примере показано две версии интерфейса: каждое — во вложенном пространстве имен. Пространство имен `v_20` содержит некоторые изменения из интерфейса `v_10` и помечается как встроенное. Клиентский код, который использует новую библиотеку и вызывает `Contoso::Funcs::Add`, вызовет версию `v_20`. Код, который пытается вызвать `Contoso::Funcs::Divide`, теперь будет вызывать ошибку времени компиляции. Если действительно требуется эта функция, доступ к версии `v_10` можно получить путем явного вызова `Contoso::v_10::Funcs::Divide`.

```
namespace Contoso
{
    namespace v_10
    {
        template <typename T>
        class Funcs
        {
        public:
            Funcs(void);
            T Add(T a, T b);
            T Subtract(T a, T b);
            T Multiply(T a, T b);
            T Divide(T a, T b);
        };
    }

    inline namespace v_20
    {
        template <typename T>
        class Funcs
        {
        public:
            Funcs(void);
            T Add(T a, T b);
            T Subtract(T a, T b);
            T Multiply(T a, T b);
            std::vector<double> Log(double);
            T Accumulate(std::vector<T> nums);
        };
    }
}
```

Псевдонимы пространств имен

Имена пространств имен должны быть уникальными, из-за чего зачастую они получаются не слишком короткими. Если длина имени затрудняет чтение кода или утомительно вводить файл заголовка, где нельзя использовать директивы `using`, можно создать псевдоним пространства имен, который служит аббревиатурой для фактического имени. Пример:

```
namespace a_very_long_namespace_name { class Foo {}; }
namespace AVLNN = a_very_long_namespace_name;
void Bar(AVLNN::Foo foo){ }
```

анонимные или безымянные пространства имен

Вы можете создать явное пространство имен, но не присвоить ему имя.

```
namespace
{
    int MyFunc(){}
}
```

Это называется безымянным или анонимным пространством имен, и его можно использовать, если нужно сделать объявления переменных невидимыми для кода в других файлах (т. е. обеспечить их внутреннюю компоновку) без создания именованного пространства имен. Весь код, находящийся в том же файле, может видеть идентификаторы в безымянном пространстве имен, но эти идентификаторы, а также само пространство имен, будут невидимым за пределами этого файла или, точнее, вне блока перевода.

См. также

[Объявления и определения](#)

Перечисления (C++)

12.11.2021 • 4 minutes to read

Перечисление — это пользовательский тип, состоящий из набора целочисленных констант, называемых перечислителями.

NOTE

В этой статье рассматривается тип языка C++ Standard в формате ISO `enum` и тип **класса** `Enum` с областью действия (или строго типизированный), который появился в C++ 11. Сведения об **открытых классах** `enum` или **закрытых типах классов** `enum` в c++/CLI и c++/CX см. в разделе [Класс Enum](#).

Синтаксис

```
// unscoped enum:  
enum [identifier] [: type]  
{enum-list};  
  
// scoped enum:  
enum [class|struct]  
[identifier] [: type]  
{enum-list};
```

```
// Forward declaration of enumerations (C++11):  
enum A : int; // non-scoped enum must have type specified  
enum class B; // scoped enum defaults to int but ...  
enum class C : short; // ... may have any integral underlying type
```

Параметры

identifier

Имя типа, присваиваемое перечислению.

type

Базовый тип перечислителей; все перечислители имеют один базовый тип. Может быть любым целочисленным типом.

Перечисление-список

Разделенный запятыми список перечислителей в перечислении. Каждый перечислитель или имя переменной в области должны быть уникальными. Однако значения могут повторяться. В неограниченном перечислении область является окружающей областью. в перечислении с заданной областью областью является сам *список перечисления*. В перечислении с заданной областью список может быть пустым, что фактически определяет новый целочисленный тип.

class

С помощью этого ключевого слова в объявлении указывается перечисление с областью видимости, и необходимо указать *идентификатор*. Можно также использовать `struct` ключевое слово вместо `class`, так как они семантически эквивалентны в данном контексте.

Область перечислителя

Перечисление предоставляет контекст для описания диапазона значений, которые представлены в виде именованных констант и также называются перечислителями. В первоначальных типах перечислений C и C++ перечислители с неполным именем являются видимыми внутри области видимости, в которой объявлено перечисление. В ограниченных перечислениях имя перечислителя должно уточняться именем типа перечисления. В следующем примере демонстрируется основное различие между двумя видами перечислений.

```
namespace CardGame_Scoped
{
    enum class Suit { Diamonds, Hearts, Clubs, Spades };

    void PlayCard(Suit suit)
    {
        if (suit == Suit::Clubs) // Enumerator must be qualified by enum type
        { /*...*/}
    }
}

namespace CardGame_NonScoped
{
    enum Suit { Diamonds, Hearts, Clubs, Spades };

    void PlayCard(Suit suit)
    {
        if (suit == Clubs) // Enumerator is visible without qualification
        { /*...*/}
    }
}
```

Каждому имени в перечислении присваивается целочисленное значение, которое соответствует определенному месту в порядке значений в перечислении. По умолчанию первому значению присваивается 0, следующему — 1 и т. д., но можно задать значение перечислителя явным образом, как показано ниже:

```
enum Suit { Diamonds = 1, Hearts, Clubs, Spades };
```

Перечислителю `Diamonds` присваивается значение `1`. Если последующим перечислителям не присваиваются явные значения, они получают значение предыдущего перечислителя плюс один. В предыдущем примере `Hearts` имел бы значение 2, `Clubs` — значение 3 и т.д.

Каждый перечислитель рассматривается как константа и должен иметь уникальное имя в пределах области, где `enum` определено (для перечислений с неограниченным диапазоном) или внутри `enum` самого себя (для перечислений с областью видимости). Значения, задаваемые имена, могут быть неуникальными. Например, для следующего объявления неограниченного перечисления `suit`:

```
enum Suit { Diamonds = 5, Hearts, Clubs = 4, Spades };
```

значения `Diamonds`, `Hearts`, `Clubs` и `Spades` равны 5, 6, 4 и 5 соответственно. Обратите внимание, что значение 5 используется несколько раз; это допускается, независимо от намерений разработчика. Такие же правила распространяются на ограниченные перечисления.

Приведение правил

Неограниченные константы перечисления могут быть неявно преобразованы в, но никогда не может быть неявно преобразовано `int` в значение `enum`. В следующем примере показано, что пройдет при попытке присвоить переменной `hand` значение, не относящееся к типу `Suit`:

```
int account_num = 135692;
Suit hand;
hand = account_num; // error C2440: '=' : cannot convert from 'int' to 'Suit'
```

Для преобразования `int` в перечислитель с областью или вне области действия требуется приведение. Однако неограниченный перечислитель можно преобразовать в целочисленное значение без приведения.

```
int account_num = Hearts; //OK if Hearts is in a unscoped enum
```

Использование подобных неявных преобразований может приводить к непредвиденным побочным эффектам. Чтобы избежать ошибок программирования, связанных с неограниченными перечислениями, значения ограниченных перечислений являются строго типизированными. Ограниченные перечислители должны уточняться именем типа перечисления (идентификатором); они не могут быть неявно преобразованы, как показано в следующем примере:

```
namespace ScopedEnumConversions
{
    enum class Suit { Diamonds, Hearts, Clubs, Spades };

    void AttemptConversions()
    {
        Suit hand;
        hand = Clubs; // error C2065: 'Clubs' : undeclared identifier
        hand = Suit::Clubs; //Correct.
        int account_num = 135692;
        hand = account_num; // error C2440: '=' : cannot convert from 'int' to 'Suit'
        hand = static_cast<Suit>(account_num); // OK, but probably a bug!!!

        account_num = Suit::Hearts; // error C2440: '=' : cannot convert from 'Suit' to 'int'
        account_num = static_cast<int>(Suit::Hearts); // OK
    }
}
```

Обратите внимание, что в строке `hand = account_num;` по-прежнему содержится ошибка, которая происходит при использовании неограниченных перечислений, как показано выше. Эта ошибка устраняется с помощью явного приведения. Однако при использовании ограниченных перечислений попытка преобразования в следующем операторе — `account_num = Suit::Hearts;` — больше не будет разрешена без явного приведения.

Перечисления без перечислителей

Visual Studio 2017 **версии 15,3 и более поздних** версий (доступно в [/std: c++ 17](#)): путем определения перечисления (regular или с ограниченной областью) с явным базовым типом и без перечислителей, вы можете использовать новый целочисленный тип, не имеющий неявного преобразования в любой другой тип. Используя этот тип вместо встроенного базового типа, можно исключить потенциальные ошибки для незначительных ошибок, вызванных случайными неявными преобразованиями.

```
enum class byte : unsigned char { };
```

Новый тип является точной копией базового типа и, таким образом, имеет то же соглашение о вызовах, что означает, что его можно использовать в ABI без снижения производительности. Если переменные типа инициализируются с помощью инициализации прямого списка, приведение не требуется. В следующем примере показано, как инициализировать перечисления без перечислителей в различных контекстах:

```
enum class byte : unsigned char { };

enum class E : int { };
E e1{ 0 };
E e2 = E{ 0 };

struct X
{
    E e{ 0 };
    X() : e{ 0 } { }
};

E* p = new E{ 0 };

void f(E e) {}

int main()
{
    f(E{ 0 });
    byte i{ 42 };
    byte j = byte{ 42 };

    // unsigned char c = j; // C2440: 'initializing': cannot convert from 'byte' to 'unsigned char'
    return 0;
}
```

См. также

[Объявления перечислений C](#)

[Ключевые слова](#)

NOTE

В C++ 17 и более поздних версиях `std::variant` class — это строго типизированная альтернатива для union .

`union` — Это определяемый пользователем тип, в котором все члены совместно используют одно и то же расположение в памяти. Это определение означает, что в любой момент времени объект union может содержать не более одного объекта из списка членов. Кроме того, это означает, что независимо от количества элементов union у пользователя всегда используется достаточно памяти для хранения самого большого элемента.

union Может быть полезен для экономии памяти при наличии большого количества объектов и ограниченной памяти. Тем не менее, union для правильной работы требуется дополнительные меры. Вы несете ответственность за обеспечение доступа к тому же назначенному Вами члену. Если какие-либо типы членов имеют нетривиальный Construct или, то необходимо написать дополнительный код для явного struct и уничтожения этого члена. Прежде чем использовать union , определите, может ли проблема, которую вы пытаетесь решить, лучше выражаться с помощью базового class и производного class типов.

Синтаксис

```
union tag неявное согласие { member-List };
```

Параметры

`tag`

Имя типа, присвоенное union .

`member-List`

Элементы, которые union может содержать.

Объявите элемент union

Начните объявление объекта с union помостью `union` ключевого слова и заключите список членов в фигурные скобки:

```

// declaring_a_union.cpp
union RecordType    // Declare a simple union type
{
    char    ch;
    int     i;
    long    l;
    float   f;
    double  d;
    int *int_ptr;
};

int main()
{
    RecordType t;
    t.i = 5; // t holds an int
    t.f = 7.25; // t now holds a float
}

```

Используйте union

В предыдущем примере любой код, обращающийся к потребностям, должен union понять, какой элемент содержит данные. Наиболее распространенное решение этой проблемы называется *размеченные union*. Он заключает union в struct и включает enum член, указывающий тип элемента, который в настоящее время хранится в union . В следующем примере демонстрируется использование основного подхода:

```

#include <queue>

using namespace std;

enum class WeatherDataType
{
    Temperature, Wind
};

struct TempData
{
    int StationId;
    time_t time;
    double current;
    double max;
    double min;
};

struct WindData
{
    int StationId;
    time_t time;
    int speed;
    short direction;
};

struct Input
{
    WeatherDataType type;
    union
    {
        TempData temp;
        WindData wind;
    };
};

// Functions that are specific to data types
void Process_Temp(TempData t) {}
void Process_Wind(WindData w) {}

```

```

void Initialize(std::queue<Input>& inputs)
{
    Input first;
    first.type = WeatherDataType::Temperature;
    first.temp = { 101, 1418855664, 91.8, 108.5, 67.2 };
    inputs.push(first);

    Input second;
    second.type = WeatherDataType::Wind;
    second.wind = { 204, 1418859354, 14, 27 };
    inputs.push(second);
}

int main(int argc, char* argv[])
{
    // Container for all the data records
    queue<Input> inputs;
    Initialize(inputs);
    while (!inputs.empty())
    {
        Input const i = inputs.front();
        switch (i.type)
        {
        case WeatherDataType::Temperature:
            Process_Temp(i.temp);
            break;
        case WeatherDataType::Wind:
            Process_Wind(i.wind);
            break;
        default:
            break;
        }
        inputs.pop();
    }
    return 0;
}

```

В предыдущем примере элемент в не union `Input` struct имеет имени, поэтому он называется *анонимным union*. Доступ к его членам можно получить напрямую, как если бы они были членами класса struct. Дополнительные сведения об использовании анонимных служб union см. в разделе [anonymous union](#).

В предыдущем примере показана проблема, которую можно также решить с помощью class типов, производных от общего базового класса class. Код можно разветвление на основе типа среди выполнения каждого объекта в контейнере. Код может быть проще в обслуживании и понимании, но он также может быть медленнее, чем при использовании union. Кроме того, с union можно хранить несвязанные типы. unionПозволяет динамически изменять тип хранимого значения, не изменяя тип union самой переменной. Например, можно создать разнородный массив `MyUnionType`, элементы которого хранят различные значения различных типов.

В примере несложно неверно `Input` struct. Пользователь должен правильно использовать дискриминатор для доступа к члену, содержащему данные. Вы можете защититься от неправильного использования, сделав union `private` и предоставляя специальные функции доступа, как показано в следующем примере.

Без ограничений union (c++ 11)

В C++ 03 и более ранних версий объект union может содержать не являющиеся static данными элементы, имеющие class тип, если тип не имеет пользователя Con struct OR, de struct or или операторы присваивания. В C++11 эти ограничения отсутствуют. Если включить такой элемент в union, компилятор автоматически помечает все специальные функции-члены, которые не предоставляются

пользователем как `deleted`. Если объект union является анонимным union внутри class или struct , то любые специальные функции-члены объекта class или struct , не указанные пользователем, помечаются как `deleted` . В следующем примере показано, как справиться с этим вариантом. Один из членов union имеет член, который требует этой особой обработки:

```
// for MyVariant
#include <crtdbg.h>
#include <new>
#include <utility>

// for sample objects and output
#include <string>
#include <vector>
#include <iostream>

using namespace std;

struct A
{
    A() = default;
    A(int i, const string& str) : num(i), name(str) {}

    int num;
    string name;
    //...
};

struct B
{
    B() = default;
    B(int i, const string& str) : num(i), name(str) {}

    int num;
    string name;
    vector<int> vec;
    // ...
};

enum class Kind { None, A, B, Integer };

#pragma warning (push)
#pragma warning(disable:4624)
class MyVariant
{
public:
    MyVariant()
        : kind_(Kind::None)
    {
    }

    MyVariant(Kind kind)
        : kind_(kind)
    {
        switch (kind_)
        {
        case Kind::None:
            break;
        case Kind::A:
            new (&a_) A();
            break;
        case Kind::B:
            new (&b_) B();
            break;
        case Kind::Integer:
            i_ = 0;
            break;
        default:
            break;
        }
    }
};
```

```

        _ASSERT(false);
        break;
    }
}

~MyVariant()
{
    switch (kind_)
    {
    case Kind::None:
        break;
    case Kind::A:
        a_.~A();
        break;
    case Kind::B:
        b_.~B();
        break;
    case Kind::Integer:
        break;
    default:
        _ASSERT(false);
        break;
    }
    kind_ = Kind::None;
}

MyVariant(const MyVariant& other)
: kind_(other.kind_)
{
    switch (kind_)
    {
    case Kind::None:
        break;
    case Kind::A:
        new (&a_) A(other.a_);
        break;
    case Kind::B:
        new (&b_) B(other.b_);
        break;
    case Kind::Integer:
        i_ = other.i_;
        break;
    default:
        _ASSERT(false);
        break;
    }
}

MyVariant(MyVariant&& other)
: kind_(other.kind_)
{
    switch (kind_)
    {
    case Kind::None:
        break;
    case Kind::A:
        new (&a_) A(move(other.a_));
        break;
    case Kind::B:
        new (&b_) B(move(other.b_));
        break;
    case Kind::Integer:
        i_ = other.i_;
        break;
    default:
        _ASSERT(false);
        break;
    }
    other.kind_ = Kind::None;
}

```

```

        _ASSERT(this->kind_ == Kind::None,
    }

MyVariant& operator=(const MyVariant& other)
{
    if (&other != this)
    {
        switch (other.kind_)
        {
        case Kind::None:
            this->~MyVariant();
            break;
        case Kind::A:
            *this = other.a_;
            break;
        case Kind::B:
            *this = other.b_;
            break;
        case Kind::Integer:
            *this = other.i_;
            break;
        default:
            _ASSERT(false);
            break;
        }
    }
    return *this;
}

MyVariant& operator=(MyVariant&& other)
{
    _ASSERT(this != &other);
    switch (other.kind_)
    {
    case Kind::None:
        this->~MyVariant();
        break;
    case Kind::A:
        *this = move(other.a_);
        break;
    case Kind::B:
        *this = move(other.b_);
        break;
    case Kind::Integer:
        *this = other.i_;
        break;
    default:
        _ASSERT(false);
        break;
    }
    other.kind_ = Kind::None;
    return *this;
}

MyVariant(const A& a)
    : kind_(Kind::A), a_(a)
{
}

MyVariant(A&& a)
    : kind_(Kind::A), a_(move(a))
{
}

MyVariant& operator=(const A& a)
{
    if (kind_ != Kind::A)
    {
        this->~MyVariant();
        new (&this->kind_) MyVariant(Kind::A);
        this->a_.operator=(a);
    }
    return *this;
}

```

```

        new (this) MyVariant(d);
    }
    else
    {
        a_ = a;
    }
    return *this;
}

MyVariant& operator=(A&& a)
{
    if (kind_ != Kind::A)
    {
        this->~MyVariant();
        new (this) MyVariant(move(a));
    }
    else
    {
        a_ = move(a);
    }
    return *this;
}

MyVariant(const B& b)
: kind_(Kind::B), b_(b)
{
}

MyVariant(B&& b)
: kind_(Kind::B), b_(move(b))
{
}

MyVariant& operator=(const B& b)
{
    if (kind_ != Kind::B)
    {
        this->~MyVariant();
        new (this) MyVariant(b);
    }
    else
    {
        b_ = b;
    }
    return *this;
}

MyVariant& operator=(B&& b)
{
    if (kind_ != Kind::B)
    {
        this->~MyVariant();
        new (this) MyVariant(move(b));
    }
    else
    {
        b_ = move(b);
    }
    return *this;
}

MyVariant(int i)
: kind_(Kind::Integer), i_(i)
{
}

MyVariant& operator=(int i)
{
    if (kind_ != Kind::Integer)
    {

```

```

    {
        this->~MyVariant();
        new (this) MyVariant(i);
    }
    else
    {
        i_ = i;
    }
    return *this;
}

Kind GetKind() const
{
    return kind_;
}

A& GetA()
{
    _ASSERT(kind_ == Kind::A);
    return a_;
}

const A& GetA() const
{
    _ASSERT(kind_ == Kind::A);
    return a_;
}

B& GetB()
{
    _ASSERT(kind_ == Kind::B);
    return b_;
}

const B& GetB() const
{
    _ASSERT(kind_ == Kind::B);
    return b_;
}

int& GetInteger()
{
    _ASSERT(kind_ == Kind::Integer);
    return i_;
}

const int& GetInteger() const
{
    _ASSERT(kind_ == Kind::Integer);
    return i_;
}

private:
    Kind kind_;
    union
    {
        A a_;
        B b_;
        int i_;
    };
};

#pragma warning (pop)

int main()
{
    A a(1, "Hello from A");
    B b(2, "Hello from B");

    MyVariant mv_1 = a;
}

```

```

cout << "mv_1 = a: " << mv_1.GetA().name << endl;
mv_1 = b;
cout << "mv_1 = b: " << mv_1.GetB().name << endl;
mv_1 = A(3, "hello again from A");
cout << R"aaa(mv_1 = A(3, "hello again from A"): )aaa" << mv_1.GetA().name << endl;
mv_1 = 42;
cout << "mv_1 = 42: " << mv_1.GetInteger() << endl;

b.vec = { 10,20,30,40,50 };

mv_1 = move(b);
cout << "After move, mv_1 = b: vec.size = " << mv_1.GetB().vec.size() << endl;

cout << endl << "Press a letter" << endl;
char c;
cin >> c;
}

```

Объект union не может хранить ссылку. Объект union также не поддерживает наследование. Это означает, что нельзя использовать в union качестве основы class или наследовать от другого class или иметь виртуальные функции.

Инициализация union

Можно объявить и инициализировать union в той же инструкции, назначив выражение, заключенное в фигурные скобки. Выражение вычисляется и присваивается первому полю union .

```

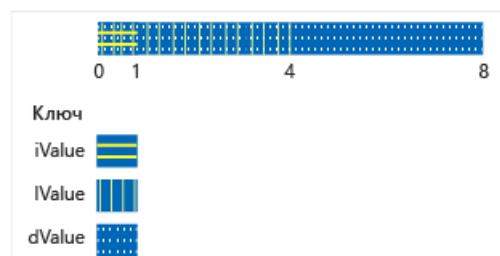
#include <iostream>
using namespace std;

union NumericType
{
    short      iValue;
    long       lValue;
    double     dValue;
};

int main()
{
    union NumericType Values = { 10 }; // iValue = 10
    cout << Values.iValue << endl;
    Values.dValue = 3.1416;
    cout << Values.dValue << endl;
}
/* Output:
10
3.141600
*/

```

Объект `NumericType` union упорядочивается в память (по принципу концептуально), как показано на следующем рисунке.



служба хранилища данных в `NumericType` union

Подключившихся union

Анонимный объект union объявлен без `class-name` или `declarator-list`.

```
union { member-List }
```

Имена, объявленные в анонимном режиме `union`, используются непосредственно, как и переменные, не являющиеся членами. Это означает, что имена, объявленные в анонимном режиме, `union` должны быть уникальными в окружающей области.

На анонимность `union` распространяются следующие дополнительные ограничения:

- Если он объявлен в области видимости файла или пространства имен, он также должен быть объявлен как `static`.
- Он может иметь только `public` члены; Having `private` И `protected` члены анонимно `union` создают ошибки.
- Он не может содержать функции элементов.

См. также

[Классы и структуры](#)

[Ключевые слова](#)

`class`

`struct`

Функции (C++)

12.11.2021 • 11 minutes to read

Функции — это блоки кода, выполняющие определенные операции. Если требуется, функция может определять входные параметры, позволяющие вызывающим объектам передавать ей аргументы. При необходимости функция также может возвращать значение как выходное. Функции полезны для инкапсуляции основных операций в едином блоке, который может многократно использоваться. В идеальном случае имя этого блока должно четко описывать назначение функции. Следующая функция принимает два целых числа от вызывающего объекта и возвращает их сумму; *a* и *b* — это *Параметры* типа `int`.

```
int sum(int a, int b)
{
    return a + b;
}
```

Функцию можно вызывать или *вызывать* из любого числа мест в программе. Значения, передаваемые в функцию, являются *аргументами*, типы которых должны быть совместимы с типами параметров в определении функции.

```
int main()
{
    int i = sum(10, 32);
    int j = sum(i, 66);
    cout << "The value of j is" << j << endl; // 108
}
```

Длина функции практически не ограничена, однако для максимальной эффективности кода целесообразно использовать функции, каждая из которых выполняет одиночную, четко определенную задачу. Сложные алгоритмы лучше разбивать на более короткие и простые для понимания функции, если это возможно.

Функции, определенные в области видимости класса, называются функциями-членами. В C++, в отличие от других языков, функции можно также определять в области видимости пространства имен (включая неявное глобальное пространство имен). Такие функции называются *бесплатными функциями* или *функциями, не являющимися членами*. Они широко используются в стандартной библиотеке.

Функции могут быть *перегружены*, а это значит, что разные версии функции могут совместно использовать одно и то же имя, если они отличаются числом и (или) типом формальных параметров. Дополнительные сведения см. в разделе [перегрузка функций](#).

Части объявления функции

Минимальное *объявление* функции состоит из возвращаемого типа, имени функции и списка параметров (который может быть пустым) вместе с дополнительными ключевыми словами, которые предоставляют компилятору дополнительные инструкции. В следующем примере показано объявление функции:

```
int sum(int a, int b);
```

Определение функции состоит из *объявления*, а также *тела*, который является кодом между фигурными

скобками:

```
int sum(int a, int b)
{
    return a + b;
}
```

Объявление функции, за которым следует точка с запятой, может многократно встречаться в разных местах кода программы. Оно необходимо перед любыми вызовами этой функции в каждой записи преобразования. По правилу одного определения, определение функции должно фигурировать в коде программы лишь один раз.

При объявлении функции необходимо указать:

1. Возвращаемый тип, указывающий тип значения, возвращаемого функцией, или `void` значение, если значения не возвращаются. В C++ 11 `auto` является допустимым возвращаемым типом, который указывает компилятору вывести тип из оператора `return`. В C++ 14 `decltype(auto)` также разрешено. Дополнительные сведения см. в подразделе "Выведение возвращаемых типов" ниже.
2. Имя функции, которое должно начинаться с буквы или символа подчеркивания и не должно содержать пробелов. В стандартной библиотеке со знака подчеркивания обычно начинаются имена закрытых функций-членов или функций, не являющихся членами и не предназначенных для использования в вашем коде.
3. Список параметров, заключенный в скобки. В этом списке через запятую указывается нужное (возможно, нулевое) число параметров, задающих тип и, при необходимости, локальное имя, по которому к значениям можно получить доступ в теле функции.

Необязательные элементы объявления функции:

1. `constexpr`, который указывает, что возвращаемое значение функции является константой, которое может быть вычислено во время компиляции.

```
constexpr float exp(float x, int n)
{
    return n == 0 ? 1 :
        n % 2 == 0 ? exp(x * x, n / 2) :
        exp(x * x, (n - 1) / 2) * x;
};
```

2. Спецификация компоновки `extern` или `static`.

```
//Declare printf with C linkage.
extern "C" int printf( const char *fmt, ... );
```

Дополнительные сведения см. в разделе [Преобразование единиц и компоновки](#).

3. `inline`, который указывает компилятору заменить каждый вызов функции на код самой функции. Подстановка может улучшить эффективность кода в сценариях, где функция выполняется быстро и многократно вызывается во фрагментах, являющихся критическими для производительности программы.

```
inline double Account::GetBalance()
{
    return balance;
}
```

Дополнительные сведения см. в разделе [встроенные функции](#).

4. `noexcept` Выражение, указывающее, может ли функция вызывать исключение. В следующем примере функция не создает исключение, если `is_pod` выражение принимает значение `true`.

```
#include <type_traits>

template <typename T>
T copy_object(T& obj) noexcept(std::is_pod<T>) {...}
```

Дополнительные сведения см. на веб-сайте [noexcept](#).

5. (Только функции-члены) Квалификаторы ОПС, которые указывают, является ли функция `const` или `volatile`.
6. (Только функции-члены) `virtual`, `override` или `final`. `virtual` Указывает, что функцию можно переопределить в производном классе. `override` означает, что функция в производном классе переопределяет виртуальную функцию. `final` означает, что функция не может быть переопределена в любом последующем производном классе. Дополнительные сведения см. в разделе [виртуальные функции](#).
7. (только функции-члены) `static` применение к функции-члену означает, что функция не связана ни с одним экземпляром объекта класса.
8. (Только функции-члены, не являющиеся статическими) Квалификатор `ref`, указывающий компилятору, какую перегрузку функции следует выбрать, когда неявный параметр объекта (`*this`) является ссылкой `rvalue` или ссылкой `lvalue`. Дополнительные сведения см. в разделе [перегрузка функций](#).

На следующем рисунке показаны компоненты определения функции. Затененная область является телом функции.



Части определения функции

Определения функций

Определение функции состоит из объявления и тела функции, заключенной в фигурные скобки, которые содержат объявления переменных, инструкции и выражения. В следующем примере показано полное определение функции:

```
int foo(int i, std::string s)
{
    int value {i};
    MyClass mc;
    if(strcmp(s, "default") != 0)
    {
        value = mc.do_something(i);
    }
    return value;
}
```

Переменные, объявленные в теле функции, называются локальными. Они исчезают из области видимости при выходе из функции, поэтому функция никогда не должна возвращать ссылку на локальную переменную.

```
MyClass& boom(int i, std::string s)
{
    int value {i};
    MyClass mc;
    mc.Initialize(i,s);
    return mc;
}
```

функции `const` и `constexpr`

Функцию-член можно объявить как `const`, чтобы указать, что функция не может изменять значения каких-либо элементов данных в классе. Объявляя функцию-член как `const`, вы помогаете компилятору обеспечить *правильность константы*. Если кто-то по ошибке пытается изменить объект с помощью функции, объявленной как `const`, возникает ошибка компилятора. Дополнительные сведения см. в разделе [const](#).

Объявите функцию, как `constexpr`, если бы получаемое значение могло быть определено во время компиляции. Функция `constexpr` обычно выполняется быстрее, чем обычная функция. Дополнительные сведения см. на веб-сайте [constexpr](#).

Шаблоны функций

Шаблоны функций подобны шаблонам классов. Их задача заключается в создании конкретных функций на основе аргументов шаблонов. Во многих случаях шаблоны могут определять типы аргументов, поэтому их не требуется явно указывать.

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
    return lhs + rhs;
}

auto a = Add2(3.13, 2.895); // a is a double
auto b = Add2(string{ "Hello" }, string{ " World" }); // b is a std::string
```

Дополнительные сведения см. в статье [шаблоны функций](#).

Параметры и аргументы функций

У функции имеется список параметров, в котором через запятую перечислено необходимое (возможно, нулевое) число типов. Каждому параметру присваивается имя, по которому к нему можно получить

доступ в теле функции. В шаблоне функции могут указываться дополнительные типы или значения параметров. Вызывающий объект передает аргументы, представляющие собой конкретные значения, типы которых совместимы со списком параметров.

По умолчанию аргументы передаются функции по значению, то есть функция получает копию передаваемого объекта. Копирование крупных объектов может быть ресурсозатратным и неоправданным. Чтобы аргументы передавались по ссылке (в частности в ссылке lvalue), добавьте в параметр квантификатор ссылки.

```
void DoSomething(std::string& input){...}
```

Если функция изменяет аргумент, передаваемый по ссылке, изменяется исходный объект, а не его локальная копия. Чтобы предотвратить изменение такого аргумента функцией, укажите для параметра значение `const&`:

```
void DoSomething(const std::string& input){...}
```

C++ 11: Чтобы явно отреагировать на аргументы, передаваемые по ссылке rvalue или lvalue-Reference, используйте двойной амперсанд для параметра, чтобы указать универсальную ссылку:

```
void DoSomething(const std::string&& input){...}
```

Функция, объявленная с ключевым словом `Single void` в списке объявлений параметров, не принимает аргументов, если ключевое слово `void` является первым и единственным членом списка объявлений аргумента. Аргументы типа `void` в любом расположении в списке выдают ошибки. Пример:

```
// OK same as GetTickCount()
long GetTickCount( void );
```

Обратите внимание, что, хотя недопустимо указывать `void` аргумент, за исключением описанного здесь, типы, производные от типа `void` (например, указатели на `void` и массивы `void`), могут отображаться в любом месте списка объявлений аргумента.

Аргументы по умолчанию

Последним параметрам в сигнатуре функции можно назначить аргумент по умолчанию, т. е. вызывающий объект сможет опустить аргумент при вызове функции, если не требуется указать какое-либо другое значение.

```

int DoSomething(int num,
    string str,
    Allocator& alloc = defaultAllocator)
{ ... }

// OK both parameters are at end
int DoSomethingElse(int num,
    string str = string{ "Working" },
    Allocator& alloc = defaultAllocator)
{ ... }

// C2548: 'DoMore': missing default parameter for parameter 2
int DoMore(int num = 5, // Not a trailing parameter!
    string str,
    Allocator& = defaultAllocator)
{...}

```

Дополнительные сведения см. в разделе [аргументы по умолчанию](#).

ТИПОВ ВОЗВРАЩАЕМЫХ ФУНКЦИЯМИ ЗНАЧЕНИЙ;

Функция не может возвращать другую функцию или встроенный массив. Однако он может возвращать указатели на эти типы или **лямбда-выражение**, которое создает объект функции. За исключением этих случаев, функция может возвращать значение любого типа в области или не может возвращать значение, в этом случае возвращаемым типом является `void`.

Завершающие возвращаемые типы

"Обычные" возвращаемые типы расположены слева от сигнатуры функции. **Завершающий возвращаемый тип** расположен в правой части сигнатуры и предшествует `->` оператору. Завершающие возвращаемые типы особенно полезны в шаблонах функций, когда тип возвращаемого значения зависит от параметров шаблона.

```

template<typename Lhs, typename Rhs>
auto Add(const Lhs& lhs, const Rhs& rhs) -> decltype(lhs + rhs)
{
    return lhs + rhs;
}

```

Если `auto` используется в сочетании с завершающим возвращаемым типом, он просто выступает в качестве заполнителя для любого результата выражения `decltype` и не выполняет выведение типов.

Локальные переменные функции

Переменная, объявленная внутри тела функции, называется **локальной переменной** или просто **локальным**. Нестатические локальные переменные видны только в теле функции. Если локальные переменные объявляются в стеке, они исчезают из области видимости при выходе из функции. Когда вы конструируете локальную переменную и возвращаете ее по значению, компилятор обычно может выполнить **оптимизацию именованного возвращаемого значения**, чтобы избежать ненужных операций копирования. Если локальная переменная возвращается по ссылке, компилятор выдаст предупреждение, поскольку любые попытки вызывающего объекта использовать эту ссылку произойдут после уничтожения локальной переменной.

В C++ локальные переменные можно объявлять как статические. Переменная является видимой только в теле функции, однако для всех экземпляров функции существует только одна копия переменной. Локальные статические объекты удаляются во время завершения, определенного директивой `atexit`. Если статический объект не был создан из-за того, что поток кода программы обошел соответствующее

объявление, попытка уничтожения этого объект не предпринимается.

Вычисление типов в возвращаемых типах (C++ 14)

В C++ 14 можно использовать `auto` чтобы дать компилятору инструкцию вывести возвращаемый тип из тела функции без необходимости предоставления завершающего возвращаемого типа. Обратите внимание, что всегда выводится `auto` на возврат по значению. Используйте `auto&&`, чтобы дать компилятору команду вывода ссылки.

В этом примере `auto` будет выведена как неконстантная копия значения суммы LHS и RHS.

```
template<typename Lhs, typename Rhs>
auto Add2(const Lhs& lhs, const Rhs& rhs)
{
    return lhs + rhs; //returns a non-const object by value
}
```

Обратите внимание, что не `auto` сохраняет константу-rvalue характеристики типа, который он выводит. Для функций перенаправления, возвращаемое значение которых должно сохранять аргументы `const-rvalue` характеристики или `ref-rvalue` характеристики из своих аргументов, можно использовать `decltype(auto)` ключевое слово, которое использует `decltype` правила вывода типа и сохраняет все сведения о типе. `decltype(auto)` может использоваться как обычное возвращаемое значение в левой части или как завершающее возвращаемое значение.

В следующем примере (на основе кода из [N3493](#)) показано, как `decltype(auto)` использовать, чтобы обеспечить точную пересылку аргументов функции в возвращаемый тип, который не известен до создания экземпляра шаблона.

```
template<typename F, typename Tuple = tuple<T...>, int... I>
decltype(auto) apply_(F&& f, Tuple&& args, index_sequence<I...>)
{
    return std::forward<F>(f)(std::get<I>(std::forward<Tuple>(args))...);
}

template<typename F, typename Tuple = tuple<T...>,
         typename Indices = make_index_sequence<tuple_size<Tuple>::value >>
decltype( auto )
apply(F&& f, Tuple&& args)
{
    return apply_(std::forward<F>(f), std::forward<Tuple>(args), Indices());
}
```

Возврат нескольких значений из функции

Существует несколько способов вернуть более одного значения из функции:

- Инкапсулирует значения в именованном классе или объекте структуры. Требует, чтобы определение класса или структуры было видимым для вызывающего объекта:

```

#include <string>
#include <iostream>

using namespace std;

struct S
{
    string name;
    int num;
};

S g()
{
    string t{ "hello" };
    int u{ 42 };
    return { t, u };
}

int main()
{
    S s = g();
    cout << s.name << " " << s.num << endl;
    return 0;
}

```

2. Возвращает объект "канал" std:: Tuple или std::p Air:

```

#include <tuple>
#include <string>
#include <iostream>

using namespace std;

tuple<int, string, double> f()
{
    int i{ 108 };
    string s{ "Some text" };
    double d{ .01 };
    return { i,s,d };
}

int main()
{
    auto t = f();
    cout << get<0>(t) << " " << get<1>(t) << " " << get<2>(t) << endl;

    // --or--

    int myval;
    string myname;
    double mydecimal;
    tie(myval, myname, mydecimal) = f();
    cout << myval << " " << myname << " " << mydecimal << endl;

    return 0;
}

```

3. Visual Studio 2017 **версии 15,3 и более поздних** версий (доступно в [/std:c+17](#)):

используйте структурированные привязки. Преимущество структурированных привязок заключается в том, что переменные, хранящие возвращаемые значения, инициализируются одновременно с объявлением, что в некоторых случаях может быть значительно более эффективным. В операторе `auto[x, y, z] = f();` скобки представляют и инициализируют имена, которые находятся в области действия для всего блока Function.

```

#include <tuple>
#include <string>
#include <iostream>

using namespace std;

tuple<int, string, double> f()
{
    int i{ 108 };
    string s{ "Some text" };
    double d{ .01 };
    return { i,s,d };
}

struct S
{
    string name;
    int num;
};

S g()
{
    string t{ "hello" };
    int u{ 42 };
    return { t, u };
}

int main()
{
    auto[x, y, z] = f(); // init from tuple
    cout << x << " " << y << " " << z << endl;

    auto[a, b] = g(); // init from POD struct
    cout << a << " " << b << endl;
    return 0;
}

```

4. Помимо использования возвращаемого значения, можно "возвращать" значения, определив любое количество параметров для использования передачи по ссылке, чтобы функция могла изменять или инициализировать значения объектов, предоставляемых вызывающим объектом.
- Дополнительные сведения см. в разделе [аргументы функции ссылочного типа](#).

Указатели функций

Как и в C, в C++ поддерживаются указатели на функции. Однако более типобезопасной альтернативой обычно служит использование объекта-функции.

Рекомендуется `typedef` использовать для объявления псевдонима для типа указателя функции при объявлении функции, возвращающей тип указателя функции. Например.

```

typedef int (*fp)(int);
fp myFunction(char* s); // function returning function pointer

```

Если оно не используется, то правильный синтаксис объявления функции можно вывести из синтаксиса декларатора для указателя на функцию, заменив идентификатор (в приведенном выше примере — `fp`) на имя функции и список аргументов, как показано выше:

```

int (*myFunction(char* s))(int);

```

Предыдущее объявление эквивалентно объявлению, `typedef` приведенному выше.

См. также

[Перегрузка функций](#)

[Функции с переменными списками аргументов](#)

[Явно заданные по умолчанию и удаленные функции](#)

[Подстановка с зависящим от аргументом именем \(Поиск Koenig\) для функций](#)

[Аргументы по умолчанию](#)

[Встраиваемые функции](#)

ФУНКЦИИ СО СПИСКАМИ АРГУМЕНТОВ ПЕРЕМЕННЫХ (C++)

12.11.2021 • 3 minutes to read

Функции, в объявлениях которых в качестве последнего члена указано многоточие (...), могут принимать переменное число аргументов. В таких случаях C++ обеспечивает проверку типа только для явно объявленных аргументов. Переменные списки аргументов можно использовать, если функция должна быть настолько универсальной, что могут изменяться даже количество и типы аргументов. Семейство функций — это пример функций, использующих списки аргументов переменных. `printf` *список-объявление аргументов*

ФУНКЦИИ С ПЕРЕМЕННЫМИ АРГУМЕНТАМИ

Чтобы получить доступ к аргументам после их объявления, используйте макросы, содержащиеся в стандартном файле `<stdarg.h>` как описано ниже.

БЛОК, ОТНОСЯЩИЙСЯ ТОЛЬКО К СИСТЕМАМ Microsoft

Microsoft C++ допускает указывать многоточие в качестве аргумента, если это последний аргумент и перед многоточием стоит запятая. Поэтому объявление `int Func(int i, ...);` допускается, а объявление `int Func(int i ...);` — нет.

Завершение блока, относящегося только к системам Microsoft

В объявлении функции, которая принимает переменное число аргументов, требуется по крайней мере один аргумент-местозаполнитель, даже если он не используется. Если этот аргумент-местозаполнитель не указан, доступ к остальным аргументам невозможен.

Если аргументы типа `char` передаются как переменные аргументы, они преобразуются в тип `int`. Аналогично, если аргументы типа `float` передаются как переменные аргументы, они преобразуются в тип `double`. Для аргументов остальных типов могут выполняться обычные восходящие приведения целочисленных типов и типов с плавающей запятой. Дополнительные сведения см. в разделе [стандартные преобразования](#).

Функции, которым необходимы списки с переменным количеством аргументов, объявляются с многоточием (...) в списках аргументов. Используйте типы и макросы, описанные в `<stdarg.h>` включаемом файле, для доступа к аргументам, передаваемым из списка переменных. Дополнительные сведения об этих макросах см. в разделе [va_arg, va_copy, va_end va_start](#) в документации по библиотеке времени выполнения C.

В следующем примере показано, как макросы работают вместе с типом (объявленным в `<stdarg.h>`):

```
// variable_argument_lists.cpp
#include <stdio.h>
#include <stdarg.h>

// Declaration, but not definition, of ShowVar.
void ShowVar( char *szTypes, ... );
int main() {
    ShowVar( "fcsl", 32.4f, 'a', "Test string", 4 );
}

// ShowVar takes a format string of the form
//   szFormatString -> szTypes
```

```

// "itcs", where each character specifies the
// type of the argument in that position.
//
// i = int
// f = float
// c = char
// s = string (char *)
//
// Following the format specification is a variable
// list of arguments. Each argument corresponds to
// a format character in the format string to which
// the szTypes parameter points
void ShowVar( char *szTypes, ... ) {
    va_list vl;
    int i;

    // szTypes is the last argument specified; you must access
    // all others using the variable-argument macros.
    va_start( vl, szTypes );

    // Step through the list.
    for( i = 0; szTypes[i] != '\0'; ++i ) {
        union Printable_t {
            int      i;
            float    f;
            char     c;
            char    *s;
        } Printable;

        switch( szTypes[i] ) {    // Type to expect.
            case 'i':
                Printable.i = va_arg( vl, int );
                printf_s( "%i\n", Printable.i );
                break;

            case 'f':
                Printable.f = va_arg( vl, double );
                printf_s( "%f\n", Printable.f );
                break;

            case 'c':
                Printable.c = va_arg( vl, char );
                printf_s( "%c\n", Printable.c );
                break;

            case 's':
                Printable.s = va_arg( vl, char * );
                printf_s( "%s\n", Printable.s );
                break;

            default:
                break;
        }
    }
    va_end( vl );
}

//Output:
// 32.400002
// a
// Test string

```

Приведенный выше пример иллюстрирует следующие важные правила:

- Перед тем как обращаться к аргументам из списка переменной длины, необходимо установить его маркер в качестве переменной типа `va_list`. В приведенном выше примере этот маркер имеет имя `vl`.

2. К отдельным аргументам можно обращаться при помощи макроса `va_arg`. Макросу `va_arg` необходимо указать тип получаемых аргументов, чтобы он мог перенести из стека нужное количество байтов. Если вы ошибетесь и укажете другой тип, а его размер не будет совпадать с типом, который вызывающая программа передает макросу `va_arg`, это может привести к непредсказуемым результатам.
3. Результат, полученный при помощи макроса `va_arg`, необходимо явным образом преобразовывать в нужный вам тип.

Для завершения обработки списка с переменным количеством аргументов необходимо вызвать макрос.

`va_end`

Перегрузка функций

12.11.2021 • 16 minutes to read

С++ позволяет определять несколько функций с одинаковым именем в одной области. Эти функции называются *перегруженными* функциями. Перегруженные функции позволяют указать другую семантику для функции в зависимости от типов и числа аргументов.

Например, `print` функция, которая принимает аргумент, `std::string` может выполнять задачи, отличные от тех, которые принимают аргумент типа `double`. Перегрузка позволяет избежать использования таких имен, как `print_string` ИЛИ `print_double`. Во время компиляции компилятор выбирает, какую перегрузку следует использовать в зависимости от типа аргументов, передаваемых вызывающим объектом. При вызове `print(42.0)` `void print(double d)` функции будет вызвана функция. При вызове метода `print("hello world")` `void print(std::string)` будет вызвана перегрузка.

Можно перегружать как функции-члены, так и функции, не являющиеся членами. В следующей таблице указаны компоненты объявления функций, используемые языком С++ для различения групп функций с одинаковым именем в одной области.

Заметки по перегрузке

ЭЛЕМЕНТ ОБЪЯВЛЕНИЯ ФУНКЦИИ	ИСПОЛЬЗОВАНИЕ ДЛЯ ПЕРЕГРУЗКИ
Тип возвращаемого функцией значения	Нет
Число аргументов	Да
Тип аргументов	Да
Наличие или отсутствие многоточия	Да
Использование <code>typedef</code> имен	Нет
Незаданные границы массива	Нет
<code>const</code> или <code>volatile</code>	Да, при применении ко всей функции
Квалификаторы <code>ref</code>	Да

Пример

В следующем примере показано использование перегрузки.

```
// function_overloading.cpp
// compile with: /EHsc
#include <iostream>
#include <math.h>
#include <string>

// Prototype three print functions.
int print(std::string s);           // Print a string.
int print(double dvalue);          // Print a double.
int print(double dvalue, int prec); // Print a double with a
```

```

        // given precision.

using namespace std;
int main(int argc, char *argv[])
{
    const double d = 893094.2987;
    if (argc < 2)
    {
        // These calls to print invoke print( char *s ).
        print("This program requires one argument.");
        print("The argument specifies the number of");
        print("digits precision for the second number");
        print("printed.");
        exit(0);
    }

    // Invoke print( double dvalue ).
    print(d);

    // Invoke print( double dvalue, int prec ).
    print(d, atoi(argv[1]));
}

// Print a string.
int print(string s)
{
    cout << s << endl;
    return cout.good();
}

// Print a double in default precision.
int print(double dvalue)
{
    cout << dvalue << endl;
    return cout.good();
}

// Print a double in specified precision.
// Positive numbers for precision indicate how many digits
// precision after the decimal point to show. Negative
// numbers for precision indicate where to round the number
// to the left of the decimal point.
int print(double dvalue, int prec)
{
    // Use table-lookup for rounding/truncation.
    static const double rgPow10[] = {
        10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1,
        10E0, 10E1, 10E2, 10E3, 10E4, 10E5, 10E6 };
    const int iPowZero = 6;

    // If precision out of range, just print the number.
    if (prec < -6 || prec > 7)
    {
        return print(dvalue);
    }
    // Scale, truncate, then rescale.
    dvalue = floor(dvalue / rgPow10[iPowZero - prec]) *
        rgPow10[iPowZero - prec];
    cout << dvalue << endl;
    return cout.good();
}

```

В приведенном выше коде отображается перегрузка функции `print` в области видимости файла.

Аргумент по умолчанию не считается частью типа функции. Поэтому он не используется при выборе перегруженных функций. Две функции, которые различаются только в своих аргументах, считаются множественными определениями, а не перегруженными функциями.

Аргументы по умолчанию не могут быть указаны для перегруженных операторов.

Сопоставление аргументов

Перегруженные функции выбираются для оптимального соответствия объявлений функций в текущей области аргументам, предоставленным в вызове функции. Если подходящая функция найдена, эта функция вызывается. Подходящее значение в этом контексте означает:

- Точное соответствие найдено.
- Тривиальное преобразование выполнено.
- Восходящее приведение целого типа выполнено.
- Стандартное преобразование в требуемый тип аргумента существует.
- Пользовательское преобразование (оператор преобразования или конструктор) в требуемый тип аргумента существует.
- Аргументы, представленные многоточием, найдены.

Компилятор создает набор функций-кандидатов для каждого аргумента. Функции-кандидаты — это функции, в которых фактический аргумент в данной позиции можно преобразовать в тип формального аргумента.

Для каждого аргумента создается набор наиболее подходящих функций, и выбранная функция представляет собой пересечение всех наборов. Если на пересечении находится несколько функций, перегрузка является неоднозначной и выдает ошибку. Функция, которая выбирается в конечном итоге, всегда является самой подходящей по сравнению с остальными функциями в группе по крайней мере для одного аргумента. Если нет ничего ясного, вызов функции приведет к ошибке.

Рассмотрим следующие объявления (функции отмечены как `Variant 1`, `Variant 2` и `Variant 3` для ссылки в последующем обсуждении).

```
Fraction &Add( Fraction &f, long l );      // Variant 1
Fraction &Add( long l, Fraction &f );      // Variant 2
Fraction &Add( Fraction &f, Fraction &f ); // Variant 3

Fraction F1, F2;
```

Рассмотрим следующий оператор.

```
F1 = Add( F2, 23 );
```

Представленный выше оператор создает два набора.

НАБОР 1. ФУНКЦИИ-КАНДИДАТЫ, ИМЕЮЩИЕ ПЕРВЫЙ АРГУМЕНТ ДРОБНОГО ТИПА

Variant 1

Variant 3

SET 2: ФУНКЦИИ-КАНДИДАТЫ, ВТОРОЙ АРГУМЕНТ КОТОРОГО МОЖНО ПРЕОБРАЗОВАТЬ В ТИП `INT`

Вариант 1 (`int` можно преобразовать в `long` использование стандартного преобразования)

Функции в наборе 2 — это функции, для которых существуют неявные преобразования фактического типа параметра в формальный тип параметра, а среди таких функций есть функция, для которой «стоимость» преобразования фактического типа параметра в формальный тип параметра является

наименьшей.

Пересечением этих двух наборов является функция Variant 1. Ниже представлен пример неоднозначного вызова функции.

```
F1 = Add( 3, 6 );
```

В предыдущем вызове функции создаются следующие наборы.

SET 1: ПОТЕНЦИАЛЬНЫЕ ФУНКЦИИ, ИМЕЮЩИЕ ПЕРВЫЙ АРГУМЕНТ ТИПА <code>INT</code>	SET 2: ПОТЕНЦИАЛЬНЫЕ ФУНКЦИИ С ВТОРЫМ АРГУМЕНТОМ ТИПА <code>INT</code>
Вариант 2 (<code>int</code> можно преобразовать в <code>long</code> использование стандартного преобразования)	Вариант 1 (<code>int</code> можно преобразовать в <code>long</code> использование стандартного преобразования)

Поскольку пересечение этих двух наборов пусто, компилятор выдает сообщение об ошибке.

Для сопоставления аргументов функция с n аргументами по умолчанию обрабатывается как $n+1$ отдельных функций, каждая из которых имеет разное число аргументов.

Многоточие (...) выступает в качестве подстановочного знака; оно соответствует любому фактическому аргументу. Это может привести к созданию множества неоднозначных наборов, если вы не разрабатываете перегруженные наборы функций с крайней осторожностью.

NOTE

Неоднозначность перегруженных функций не может быть определена до тех пор, пока не будет обнаружен вызов функции. На этом этапе наборы создаются для каждого аргумента в вызове функции, и можно определить, существует ли неоднозначная перегрузка. Это означает, что неоднозначности могут оставаться в коде до тех пор, пока они не будут вызваны конкретным вызовом функции.

Различия типов аргументов

Перегруженные функции различают типы аргументов, имеющие разные инициализаторы.

Следовательно, аргумент заданного типа и ссылка на этот тип считаются одинаковыми для перегрузки, поскольку имеют одни и те же инициализаторы. Например, `max(double, double)` — то же самое, что и `max(double &, double &)`. Объявление двух таких функций приводит к ошибке.

По той же причине аргументы функции типа, измененные или, `const` `volatile` не обрабатываются иначе, чем базовый тип для перегрузки.

Однако механизм перегрузки функций может различать ссылки, уточняющие их на базовый тип и. Он делает код следующим:

```

// argument_type_differences.cpp
// compile with: /EHsc /W3
// C4521 expected
#include <iostream>

using namespace std;
class Over {
public:
    Over() { cout << "Over default constructor\n"; }
    Over( Over &o ) { cout << "Over&\n"; }
    Over( const Over &co ) { cout << "const Over&\n"; }
    Over( volatile Over &vo ) { cout << "volatile Over&\n"; }
};

int main() {
    Over o1;           // Calls default constructor.
    Over o2( o1 );    // Calls Over( Over& ). 
    const Over o3;    // Calls default constructor.
    Over o4( o3 );    // Calls Over( const Over& ). 
    volatile Over o5; // Calls default constructor.
    Over o6( o5 );    // Calls Over( volatile Over& ). 
}

```

Выходные данные

```

Over default constructor
Over&
Over default constructor
const Over&
Over default constructor
volatile Over&

```

Указатели `const` на `volatile` объекты и также считаются отличными от указателей на базовый тип в целях перегрузки.

Сопоставление аргументов и преобразования

Когда компилятор пытается сопоставить фактические аргументы с аргументами в объявлениях функций и точное соответствие найти не удается, для получения правильного типа он может выполнять стандартные или пользовательские преобразования. Для преобразований действуют следующие правила:

- последовательности преобразований, содержащие несколько пользовательских преобразований, не учитываются;
- последовательности преобразований, которые могут быть сокращены путем удаления промежуточных преобразований, не учитываются.

Получающаяся последовательность преобразований (если такие имеются), называется наилучшей последовательностью сопоставления. Существует несколько способов преобразования объекта типа `int` в тип `unsigned long` с помощью стандартных преобразований (см. описание в разделе [стандартные преобразования](#)).

- Выполните преобразование из `int` в `long`, а затем из `long` в `unsigned long`.
- Преобразование из `int` в `unsigned long`.

Первая последовательность, хотя она достигает требуемой цели, не является наилучшей совпадающей последовательностью — существует более короткая последовательность.

В представленной ниже таблице показана группа преобразований, называемых тривиальными. Они оказывают ограниченное влияние на определение наилучшей последовательности сопоставления. В списке, приведенном после таблицы, рассматриваются экземпляры, в которых тривиальные преобразования влияют на выбор последовательности.

Тривиальные преобразования

ТИП, ИЗ КОТОРОГО ВЫПОЛНЯЕТСЯ ПРЕОБРАЗОВАНИЕ	ТИП, В КОТОРЫЙ ВЫПОЛНЯЕТСЯ ПРЕОБРАЗОВАНИЕ
Имя типа	Имя типа ^{**&**}
Имя типа ^{**&**}	Имя типа
Type-Name []	Имя типа*
Type-Name (Argument-List)	(* Type-Name) (Argument-List)
Имя типа	^{** const} Имя типа
Имя типа	^{** volatile} Имя типа
Имя типа*	^{** const} Имя типа*
Имя типа*	^{** volatile} Имя типа*

Ниже приведена последовательность, в которой делаются попытки выполнения преобразований.

1. Точное соответствие. Точное соответствие между типами, с которыми функция вызывается, и типами, объявленными в прототипе функции, всегда является наилучшим соответствием. Последовательности тривиальных преобразований классифицируются как точные соответствия. Однако последовательности, которые не делают ни одно из этих преобразований, рассматриваются лучше, чем последовательности, которые преобразуют:
 - От указателя к указателю на `const (type * to const type *).`
 - От указателя к указателю на `volatile (type * to volatile type *).`
 - Ссылка на ссылку на `const (type & to const type &).`
 - Ссылка на ссылку на `volatile (type & to volatile type &).`
2. Сопоставление с использованием повышений. Любая последовательность, не классифицированная как точное соответствие, которая содержит только целочисленные акции, преобразования из `float` в `double`, и тривиальные преобразования классифицируется как соответствие с помощью специальных предложений. Хотя сопоставление с использованием повышений не такое хорошее, как точное, оно лучше сопоставления с использованием стандартных преобразований.
3. Сопоставление с использованием стандартных преобразований. Любая последовательность, не классифицированная как точное соответствие или сопоставление с использованием повышений и содержащая только стандартные и тривиальные преобразования, классифицируется как сопоставление с использованием стандартных преобразований. В этой категории применяются следующие правила:
 - Преобразование указателя на производный класс в указатель на прямой или косвенный базовый класс является предпочтительным для преобразования в `void *` ИЛИ `const void *`.

- преобразование из указателя на производный класс в указатель на базовый класс создает тем более хорошее соответствие, чем ближе базовый класс к прямому базовому классу.
- Предположим, что иерархия классов имеет вид, показанный на следующем рисунке.



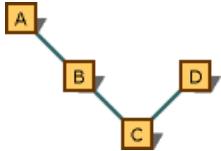
Преобразование из типа D^* в тип C^* предпочтительнее преобразования из типа D^* в тип B^* .

Аналогично, преобразование из типа D^* в тип B^* предпочтительнее преобразования из типа D^* в тип A^* .

Это же правило применяется для преобразований ссылок. Преобразование из типа $D&$ в тип $C&$ предпочтительнее преобразования из типа $D&$ в тип $B&$ и т. д.

Это же правило применяется для преобразований указателей на член. Преобразование из типа $T D::*$ в тип $T C::*$ предпочтительнее преобразования из типа $T D::*$ в тип $T B::*$ и т. д. (T — тип члена.)

Предыдущее правило применяется только в определенном пути наследования. Рассмотрим график, показанный на следующем рисунке.



Преобразование из типа C^* в тип B^* предпочтительнее преобразования из типа C^* в тип A^* .

Причина заключается в том, что эти преобразования находятся на одном пути и узел B^* ближе. Однако преобразование из типа C^* в тип D^* не является предпочтительным для преобразования в тип A^* ; нет предпочтений, так как преобразования следуют разным путям.

1. Сопоставление с пользовательскими преобразованиями. Эта последовательность не может классифицироваться как точное совпадение, сопоставление с помощью рекламных акций или соответствие с помощью стандартных преобразований. Чтобы последовательность можно было классифицировать как сопоставление с пользовательскими преобразованиями, она должна содержать только пользовательские, стандартные или тривиальные преобразования.
Сопоставление с пользовательскими преобразованиями лучше сопоставления с многоточием, но хуже сопоставления со стандартными преобразованиями.
2. Сопоставление с многоточием. Любая последовательность, соответствующая многоточию в объявлении, классифицируется как сопоставление с многоточием. Это считается самым слабым совпадением.

Пользовательские преобразования применяются при отсутствии встроенного повышения или преобразования. Эти преобразования выбираются на основе типа сопоставляемого аргумента. Рассмотрим следующий код.

```
// argument_matching1.cpp
class UDC
{
public:
    operator int()
    {
        return 0;
    }
    operator long();
};

void Print( int i )
{
};

UDC udc;

int main()
{
    Print( udc );
}
```

Доступные пользовательские преобразования для класса `UDC` относятся к типу `int` и типу `long`.

Поэтому компилятор проверяет преобразования для типа сопоставляемого объекта: `UDC`.

Преобразование в `int` существует и его выбор.

В процессе сопоставления аргументов стандартные преобразования можно применять как к аргументу, так и к результату пользовательского преобразования. Поэтому следующий код работает.

```
void LogToFile( long l );
...
UDC udc;
LogToFile( udc );
```

В приведенном выше примере для преобразования в тип вызывается определенное пользователем преобразование, **оператор Long** `udc long`. Если определяемое пользователем преобразование в тип не было `long` определено, преобразование было бы выполнено следующим образом: тип `UDC` будет преобразован в тип `int` с помощью пользовательского преобразования. Затем стандартное преобразование из типа `int` в тип было `long` бы применено для соответствия аргументу в объявлении.

Если какие-либо определенные пользователем преобразования должны соответствовать аргументу, стандартные преобразования не используются при вычислении наилучшего соответствия. Даже если для нескольких потенциальных функций требуется определенное пользователем преобразование, эти функции считаются равными. Пример:

```

// argument_matching2.cpp
// C2668 expected
class UDC1
{
public:
    UDC1( int ); // User-defined conversion from int.
};

class UDC2
{
public:
    UDC2( long ); // User-defined conversion from long.
};

void Func( UDC1 );
void Func( UDC2 );

int main()
{
    Func( 1 );
}

```

Обеим версиям `Func` требуется определенное пользователем преобразование для преобразования типа `int` в аргумент типа класса. Возможные преобразования:

- Преобразование из типа `int` в тип `UDC1` (определенное пользователем преобразование).
- Преобразование из типа `int` в тип `long`; затем преобразование в тип `UDC2` (преобразование из двух шагов).

Несмотря на то, что второй требуется как стандартное преобразование, так и определяемое пользователем преобразование, два преобразования по-прежнему считаются равными.

NOTE

Пользовательские преобразования считаются преобразованиями посредством создания или инициализации (функции преобразования). При рассмотрении наилучшего соответствия оба метода считаются одинаковыми.

Сопоставление аргументов и указатель `this`

Функции-члены класса обрабатываются по-разному, в зависимости от того, объявлены ли они как `static`. Поскольку нестатические функции имеют неявный аргумент, который предоставляет `this` указатель, нестатические функции считаются одним аргументом, чем статические функции. в противном случае они объявляются одинаково.

Для этих нестатических функций-членов требуется, чтобы подразумеваемый `this` указатель совпадал с типом объекта, через который вызывается функция, или для перегруженных операторов требуется, чтобы первый аргумент соответствовал объекту, к которому применяется оператор. (Дополнительные сведения о перегруженных операторах см. в разделе [перегруженные операторы](#).)

В отличие от других аргументов перегруженных функций, временные объекты не вводятся, и при попытке сопоставления аргумента указателя не выполняется никаких преобразований `this`.

Если `->` оператор выбора члена используется для доступа к функции-члену класса `class_name`, `this` аргумент указателя имеет тип `class_name * const`. Если члены объявляются как `const` или `volatile`, типы `const class_name * const` и `volatile class_name * const` соответственно.

Оператор выбора члена `.` работает точно так же, за исключением того, что в качестве префикса к

имени объекта подставляется неявный оператор взятия адреса `&`. В следующем примере показано, как это делается:

```
// Expression encountered in code
obj.name

// How the compiler treats it
(&obj)->name
```

С точки зрения сопоставления аргументов, левый operand операторов `->*` и `.*` (указатель на член) обрабатывается так же, как и для операторов `.` и `->` (выбор члена).

Квалификаторы `ref` для функций-членов

Квалификаторы `ref` позволяют перегружать функцию-член на основе того, является ли объект, на который указывает, значением `this rvalue` или `lvalue`. Эту функцию можно использовать, чтобы избежать ненужных операций копирования в сценариях, где вы решили не предоставлять доступ к данным с помощью указателя. Например, предположим, что класс `C` инициализирует некоторые данные в своем конструкторе и возвращает копию этих данных в функции-члене `get_data()`. Если объект типа `C` является `rvalue`, который будет уничтожен, компилятор выберет `get_data() &&` перегрузку, которая перемещает данные, а не копирует их.

```
#include <iostream>
#include <vector>

using namespace std;

class C
{
public:
    C() /*expensive initialization*/
    vector<unsigned> get_data() &
    {
        cout << "lvalue\n";
        return _data;
    }
    vector<unsigned> get_data() &&
    {
        cout << "rvalue\n";
        return std::move(_data);
    }

private:
    vector<unsigned> _data;
};

int main()
{
    C c;
    auto v = c.get_data(); // get a copy. prints "lvalue".
    auto v2 = C().get_data(); // get the original. prints "rvalue"
    return 0;
}
```

Ограничения на перегрузку

К допустимому набору перегруженных функций применяется несколько ограничений.

- Любые две функции в наборе перегруженных функций должны иметь разные списки аргументов.

- Перегрузка функций со списками аргументов одного типа лишь на основании возвращаемого типа недопустима.

Блок, относящийся только к системам Microsoft

Оператор New можно перегружать только на основе возвращаемого типа, а именно на основе указанного модификатора модели памяти.

Завершение блока, относящегося только к системам Microsoft

- Функции элементов не могут быть перегружены только на основе одного статического, а другого нестатического.
- `typedef` объявления не определяют новые типы; они представляют синонимы для существующих типов. Они не влияют на механизм перегрузки. Рассмотрим следующий код.

```
typedef char * PSTR;  
  
void Print( char *szToPrint );  
void Print( PSTR szToPrint );
```

Две указанные выше функции имеют идентичные списки аргументов. `PSTR` является синонимом для типа `char *`. В области члена этот код возвращает ошибку.

- Перечисляемые типы являются отдельными типами и могут использоваться для различия перегруженных функций.
- Типы "массив" и "указатель на" считаются идентичными в целях различия перегруженных функций, но только для одноэлементных измерений массивов. Вот почему эти перегруженные функции конфликтуют и создают сообщение об ошибке:

```
void Print( char *szToPrint );  
void Print( char szToPrint[] );
```

В случае многомерных массивов второе и все последующие измерения являются частью типа. Поэтому они используются для различия перегруженных функций.

```
void Print( char szToPrint[] );  
void Print( char szToPrint[][7] );  
void Print( char szToPrint[][9][42] );
```

Перегрузка, переопределение и скрытие

Любые два объявления функции с одинаковым именем в одной области видимости могут ссылаться на одну функцию или на две разные перегруженные функции. Если списки аргументов в объявлениях содержат аргументы эквивалентных типов (как описано в предыдущем разделе), эти объявления относятся к одной и той же функции. В противном случае они ссылаются на две различные функции, которые выбираются с использованием перегрузки.

Область класса строго наблюдалась; Поэтому функция, объявленная в базовом классе, находится не в той же области, что и функция, объявленная в производном классе. Если функция в производном классе объявлена с тем же именем, что и виртуальная функция в базовом классе, функция производного класса **переопределяет** функцию базового класса. Дополнительные сведения см. в разделе [виртуальные функции](#).

Если функция базового класса не объявлена как `Virtual`, то говорят, что функция производного класса

скрывает ее. Переопределение и скрытие отличаются от перегрузки.

Область видимости блока строго наблюдалась; Поэтому функция, объявленная в области файла, не находится в той же области, что и функция, объявлена локально. Если локально объявлена функция имеет то же имя, что и функция, объявлена в области файла, локально объявлена функция скрывает функцию области файла, не вызывая перегрузки. Пример:

```
// declaration_matching1.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
void func( int i )
{
    cout << "Called file-scoped func : " << i << endl;
}

void func( char *sz )
{
    cout << "Called locally declared func : " << sz << endl;
}

int main()
{
    // Declare func local to main.
    extern void func( char *sz );

    func( 3 );    // C2664 Error. func( int ) is hidden.
    func( "s" );
}
```

В предыдущем коде показаны два определения функции `func`. Определение, принимающее аргумент типа `char *`, является локальным для `main` из-за `extern` оператора. Поэтому определение, принимающее аргумент типа `int`, скрыто, а первый вызов метода `func` имеет значение Error.

В случае перегруженных функций-членов различным версиям функции могут предоставляться разные права доступа. Они по-прежнему считаются находящимися в области видимости включающего класса и, таким образом, являются перегруженными функциями. Рассмотрим следующий код, в котором функция-член `Deposit` перегружена; одна версия является открытой, вторая — закрытой.

Целью кода в примере является предоставление класса `Account`, в котором для внесения средств требуется правильный пароль. Для этого используется перегрузка.

Вызов `Deposit` в `Account::Deposit` вызывает закрытую функцию члена. Этот вызов является правильным, так как `Account::Deposit` является функцией-членом и имеет доступ к закрытым членам класса.

```
// declaration_matching2.cpp
class Account
{
public:
    Account()
    {
    }
    double Deposit( double dAmount, char *szPassword );

private:
    double Deposit( double dAmount )
    {
        return 0.0;
    }
    int Validate( char *szPassword )
    {
        return 0;
    }

};

int main()
{
    // Allocate a new object of type Account.
    Account *pAcct = new Account;

    // Deposit $57.22. Error: calls a private function.
    // pAcct->Deposit( 57.22 );

    // Deposit $57.22 and supply a password. OK: calls a
    // public function.
    pAcct->Deposit( 52.77, "pswd" );
}

double Account::Deposit( double dAmount, char *szPassword )
{
    if ( Validate( szPassword ) )
        return Deposit( dAmount );
    else
        return 0.0;
}
```

См. также

[Функции \(C++\)](#)

Явно используемые по умолчанию и удаленные функции

12.11.2021 • 5 minutes to read

В C++11 установленные по умолчанию и удаленные функции позволяют явным образом указывать, будут ли специальные функции-члены создаваться автоматически. Кроме того, удаленные функции определяют простой язык, который помогает предотвращать проблемы с повышением типов в аргументах любых функций — специальных функций-членов, а также обычных функций членов и функций, не являющихся членами. Такие проблемы могут приводить к ошибкам в вызовах функций.

Преимущества явным образом установленных по умолчанию и удаленных функций

Если в типе не определен конструктор по умолчанию, конструктор копии, оператор присваивания копии и деструктор для типа, то компилятор C++ создает их автоматически. Эти функции называются *специальными функциями-членами*, и они делают простые определяемые пользователем типы в C++ такими же, как структуры, в C. То есть вы можете создавать, копировать и уничтожать их без каких-либо дополнительных усилий по программированию. В C++11 в язык привносится семантика перемещения: для этого в список специальных функций-членов, которые компилятор может создавать автоматически, добавлены конструктор перемещения и оператор перемещения и присваивания.

Это удобно при работе с простыми типами, однако в сложных типах часто определяются собственные специальные функции-члены, которые могут препятствовать автоматическому созданию других специальных функций-членов. Вот как это выглядит на практике.

- Если конструктор был объявлен явным образом, то автоматическое создание конструктора не выполняется.
- Если виртуальный деструктор был объявлен явным образом, то автоматическое создание деструктора не выполняется.
- Если конструктор перемещения или оператор перемещения и присваивания был объявлен явным образом, то:
 - автоматическое создание конструктора копии не выполняется;
 - автоматическое создание оператора копирования и присваивания не выполняется.
- Если конструктор копии, оператор копирования и присваивания, конструктор перемещения, оператор перемещения и присваивания или деструктор был объявлен явным образом, то:
 - автоматическое создание конструктора перемещения не выполняется;
 - автоматическое создание оператора перемещения и присваивания не выполняется.

NOTE

Кроме того, в стандарте C++11 определены следующие дополнительные правила.

- Если конструктор копии или деструктор был объявлен явным образом, то автоматическое создание оператора копирования и присваивания не рекомендуется.
- Если оператор копирования и присваивания или деструктор был объявлен явным образом, то автоматическое создание конструктора копии не рекомендуется.

В обоих случаях Visual Studio продолжит неявное автоматическое создание необходимых функций; предупреждение не создается.

Возможна утечка этих правил в иерархии объектов. Например, если по какой-либо причине базовому классу не удается получить конструктор по умолчанию, вызываемый из производного класса, то `public` есть `protected` конструктора или, который не принимает параметров, то класс, производный от него, не может автоматически создать собственный конструктор по умолчанию.

Эти правила могут усложнить реализацию пользовательских типов и стандартных идиом C++, которые должны быть простыми и понятными. К примеру, если конструктор копии и оператор копирования и присваивания в пользовательском типе определены, но не объявлены, то такой тип будет некопируемым.

```
struct noncopyable
{
    noncopyable() {};

private:
    noncopyable(const noncopyable&);
    noncopyable& operator=(const noncopyable&);
};
```

В версиях до C++11 такой фрагмент кода был идиоматичной формой некопируемых типов. Однако здесь возникает ряд проблем.

- Конструктор копии должен быть объявлен закрытым, чтобы скрыть его, но поскольку он объявлен вообще, автоматическое создание конструктора по умолчанию запрещено. Если конструктор по умолчанию необходим, он должен быть определен явным образом, даже если он не выполняет никаких действий.
- Даже если явно определенный конструктор по умолчанию не выполняет никаких действий, компилятор считает его нетривиальным. Это не так эффективно, как автоматическое создание конструктора по умолчанию, поскольку в этом случае тип `noncopyable` не может являться истинным типом POD.
- Хотя конструктор копии и оператор копирования и присваивания скрыты от внешнего кода, однако функции-члены и дружественные функции для типа `noncopyable` все равно могут их видеть и вызывать. Если они объявлены, но не определены, при их вызове возникает ошибка компоновщика.
- Хотя это и общепринятая идиома, однако если у вас нет четкого понимания всех правил автоматического создания специальных функций-членов, намерение будет неясным.

В C++11 идиому некопируемости можно реализовать более простым способом.

```
struct noncopyable
{
    noncopyable() =default;
    noncopyable(const noncopyable&) =delete;
    noncopyable& operator=(const noncopyable&) =delete;
};
```

Обратите внимание, как разрешаются проблемы с такими идиомами в версиях до C++11.

- Создание конструктора по умолчанию по-прежнему можно предотвратить путем объявления конструктора копии, однако его можно восстановить, явным установив его по умолчанию.
- Явно установленные по умолчанию специальные функции-члены по-прежнему считаются тривиальными, поэтому производительность не снижается, а тип `noncopyable` может быть истинным типом POD.
- Конструктор копии и оператор копирования и присваивания являются открытыми, но удаленным. Определение или вызов удаленной функции представляет собой ошибку времени компиляции.
- Намерение ясно всем, кто разобрался в сути ключевых слов `=default` и `=delete`. Понимать правила автоматического создания специальных функций-членов не требуется.

Похожие идиомы существуют для создания пользовательских типов: которые не могут перемещаться, которые могут получать только динамическое выделение памяти и которые не могут получать динамическое выделение памяти. Для каждой из этих идиом имеются реализации в версиях до C++11, которым свойственны аналогичные проблемы и которые в C++11 разрешаются схожим образом — путем реализации их в виде установленных по умолчанию и удаленных специальных функций-членов.

Явно установленные по умолчанию функции

Любую специальную функцию-член можно установить по умолчанию — чтобы явным образом определить, что она использует реализацию по умолчанию, чтобы определить специальную функцию-член с любым квалификатором доступа, кроме `public`, или чтобы возобновить специальную функцию-член, автоматическое создание которой было исключено в силу других обстоятельств.

Специальная функция-член устанавливается по умолчанию путем ее объявления, как показано в следующем примере.

```
struct widget
{
    widget()=default;

    inline widget& operator=(const widget&);

    inline widget& widget::operator=(const widget&) =default;
```

Обратите внимание, что можно по умолчанию выделить специальную функцию-член за пределами тела класса, если он инициабле.

Из-за преимуществ, которые тривиальные специальные функции-члены обеспечивают в плане производительности, когда требуется поведение по умолчанию, рекомендуется использовать вместо пустых тел функций автоматически создаваемые специальные функции-члены. Это можно сделать, либо явным образом задав специальную функцию-член по умолчанию, либо не объявляя ее (и также не объявляя другие специальные функции-члены, которые будут мешать ее автоматическому созданию).

Удаленные функции

Для того чтобы специальные функции-члены, а также обычные функции-члены и функции, не являющиеся членами, не могли определяться или вызываться, их можно удалить. Удаление специальных функций-членов дает возможность избежать создания ненужных специальных функций-членов компилятором. Функция должна удаляться сразу после объявления, а не позже (в отличие от объявления по умолчанию, которое можно выполнять позже объявления).

```
struct widget
{
    // deleted operator new prevents widget from being dynamically allocated.
    void* operator new(std::size_t) = delete;
};
```

Если удалить обычные функции-члены или функции, не являющиеся членами, то проблемы с повышением типов не смогут приводить к ошибкам при вызове функции. Это работает, поскольку удаленные функции по-прежнему участвуют в разрешении перегрузок и обеспечивают лучшее соответствие, чем функция, которая может быть вызвана после повышения уровня типов. Вызов функции разрешается к более конкретным, но удаленным функциям и приводит к ошибке компиляции.

```
// deleted overload prevents call through type promotion of float to double from succeeding.
void call_with_true_double_only(float) =delete;
void call_with_true_double_only(double param) { return; }
```

Обратите внимание, что в приведенном выше примере вызов с `call_with_true_double_only` ПОМОЩЬЮ `float` аргумента приведет к ошибке компилятора, но вызов с `call_with_true_double_only` ПОМОЩЬЮ `int` аргумента не будет. В `int` случае аргумент будет выдвинут с `int` на `double` и успешно вызвать `double` версию функции, даже если это может быть не так. Чтобы гарантировать, что любой вызов этой функции с помощью аргумента, отличного от `Double`, вызовет ошибку компилятора, можно объявить версию шаблона удаляемой функции.

```
template < typename T >
void call_with_true_double_only(T) =delete; //prevent call through type promotion of any T to double from
succeding.

void call_with_true_double_only(double param) { return; } // also define for const double, double&, etc. as
needed.
```

Поиск имен функций с зависимостью от аргументов (поиск Koenig)

12.11.2021 • 2 minutes to read

Компилятор может использовать поиск имени с зависимостью от аргументов для поиска определения неопределенного вызова функции. Поиск имени с зависимостью от аргументов также называется поиском Koenig. Тип каждого аргумента в вызове функции определяется в иерархии пространств имен, классов, структур, объединений или шаблонов. При указании неуточненного [постфиксного](#) вызова функции компилятор ищет определение функции в иерархии, связанной с каждым типом аргумента.

Пример

В образце компилятор замечает, что функция `f()` принимает аргумент `x`. Аргумент `x` принадлежит типу `A::x`, определенному в пространстве имен `A`. Компилятор выполняет поиск пространства имен `A` и обнаруживает определение функции `f()`, принимающей аргумент типа `A::x`.

```
// argument_dependent_name_koenig_lookup_on_functions.cpp
namespace A
{
    struct X
    {
    };
    void f(const X&)
    {
    }
}
int main()
{
    // The compiler finds A::f() in namespace A, which is where
    // the type of argument x is defined. The type of x is A::X.
    A::X x;
    f(x);
}
```

Аргументы по умолчанию

12.11.2021 • 2 minutes to read

Во многих случаях функции имеют аргументы, которые используются настолько редко, что достаточно значения по умолчанию. В таких случаях возможность задания аргументов по умолчанию позволяет указывать только те аргументы функции, которые важны в конкретном вызове. Чтобы проиллюстрировать эту концепцию, рассмотрим пример, представленный в [перегрузке функции](#).

```
// Prototype three print functions.  
int print( char *s ); // Print a string.  
int print( double dvalue ); // Print a double.  
int print( double dvalue, int prec ); // Print a double with a  
// given precision.
```

Во многих приложениях для аргумента `prec` можно задать разумное значение по умолчанию, что исключает необходимость наличия двух функций:

```
// Prototype two print functions.  
int print( char *s ); // Print a string.  
int print( double dvalue, int prec=2 ); // Print a double with a  
// given precision.
```

Реализация `print` функции немного изменилась, чтобы отразить тот факт, что для типа существует только одна такая функция `double`:

```
// default_arguments.cpp  
// compile with: /EHsc /c  
  
// Print a double in specified precision.  
// Positive numbers for precision indicate how many digits  
// precision after the decimal point to show. Negative  
// numbers for precision indicate where to round the number  
// to the left of the decimal point.  
  
#include <iostream>  
#include <math.h>  
using namespace std;  
  
int print( double dvalue, int prec ) {  
    // Use table-lookup for rounding/truncation.  
    static const double rgPow10[] = {  
        10E-7, 10E-6, 10E-5, 10E-4, 10E-3, 10E-2, 10E-1, 10E0,  
        10E1, 10E2, 10E3, 10E4, 10E5, 10E6  
    };  
    const int iPowZero = 6;  
    // If precision out of range, just print the number.  
    if( prec >= -6 && prec <= 7 )  
        // Scale, truncate, then rescale.  
        dvalue = floor( dvalue / rgPow10[iPowZero - prec] ) *  
            rgPow10[iPowZero - prec];  
    cout << dvalue << endl;  
    return cout.good();  
}
```

Чтобы вызвать новую функцию `print`, используйте следующий код:

```
print( d );      // Precision of 2 supplied by default argument.  
print( d, 0 ); // Override default argument to achieve other  
// results.
```

При использовании аргументов по умолчанию обратите внимание на следующие моменты:

- Аргументы по умолчанию используются только в вызовах функции, в которых опущены заключительные аргументы — они должны быть последними аргументами. Поэтому следующий код недопустим:

```
int print( double dvalue = 0.0, int prec );
```

- Переопределение аргумента по умолчанию в последующих объявлениях не допускается, даже если переопределение совпадает с оригиналом. Поэтому приведенный ниже код вызывает ошибку:

```
// Prototype for print function.  
int print( double dvalue, int prec = 2 );  
  
...  
  
// Definition for print function.  
int print( double dvalue, int prec = 2 )  
{  
    ...  
}
```

Проблема с этим кодом заключается в том, что объявление функции в определении переопределяет аргумент по умолчанию для аргумента `prec`.

- В последующих определениях допускается добавлять дополнительные аргументы по умолчанию.
- Аргументы по умолчанию могут указываться для указателей на функции. Пример:

```
int (*pShowIntVal)( int i = 0 );
```

Встраиваемые функции (C++)

12.11.2021 • 6 minutes to read

Функция, определенная в теле объявления класса, является встроенной.

Пример

В показанном ниже объявлении класса конструктор `Account` является встраиваемой функцией. Функции члена `GetBalance`, `Deposit` и `Withdraw` не указаны как `inline` но могут быть реализованы как встроенные функции.

```
// Inline_Member_Functions.cpp
class Account
{
public:
    Account(double initial_balance) { balance = initial_balance; }
    double GetBalance();
    double Deposit( double Amount );
    double Withdraw( double Amount );
private:
    double balance;
};

inline double Account::GetBalance()
{
    return balance;
}

inline double Account::Deposit( double Amount )
{
    return ( balance += Amount );
}

inline double Account::Withdraw( double Amount )
{
    return ( balance -= Amount );
}
int main()
{
```

NOTE

В объявлении класса функции были объявлены без `inline` ключевого слова. `inline` Ключевое слово может быть указано в объявлении класса; результат будет таким же.

Заданная встроенная функция-член должна быть объявлена одинаково в каждом блоке компиляции. Из-за этого ограничения встраиваемые функции работают так же, как если бы они были созданными экземплярами функций. Кроме того, должно существовать только одно определение встраиваемой функции.

Функция-член класса по умолчанию имеет внешнюю компоновку, если только определение этой функции не содержит `inline` спецификатор. В предыдущем примере показано, что вам не нужно явно объявлять эти функции с помощью `inline` описателя. Использование `inline` в определении функции приводит к тому, что оно будет встроенной функцией. Однако не допускается повторное объявление

функции `inline` после вызова этой функции.

`inline`, `_inline` И `_forceinline`.

`inline` Описатели и `_inline` указывают компилятору вставить копию тела функции в каждое место вызова функции.

Вставка, называемая *встроенным развертыванием* или *встраиванием*, выполняется только в том случае, если анализ стоимости компилятора показывает, что это целесообразно. Встроенное расширение минимизирует затраты на вызов функций за счет потенциальных затрат на больший размер кода.

`_forceinline` Ключевое слово переопределяет анализ затратных преимуществ и полагается на ненужное программиста. Соблюдайте осторожность при использовании `_forceinline`. Использование беспорядочного использования `_forceinline` может привести к увеличению объема кода с незначительным выигрышем в производительности или, в некоторых случаях, даже к снижению производительности (из-за увеличения объема подкачки для большого исполняемого файла).

Применение встраиваемых функций может ускорить выполнение программ, поскольку устраняет нагрузку на вызов функций. Для подставляемых функций выполняются оптимизации кода, которые недоступны для обычных функций.

Параметры и ключевые слова подстановки компилятор обрабатывает как рекомендации. Нет никакой гарантии, что функции будут встроены. Нельзя заставить компилятор подставлять определенную функцию даже с помощью `_forceinline` ключевого слова. При компиляции с `/c1r` компилятор не будет встроен в функцию, если к функции применены атрибуты безопасности.

`inline` Ключевое слово доступно только в C++. `_inline` `_forceinline` Ключевые слова и доступны как в C, так и в C++. Для совместимости с предыдущими версиями `_inline` И `_forceinline` являются синонимами для `inline`, и `_forceinline` если не указан параметр компилятора, не следует [/za](#) (отключать расширения языка).

`inline` Ключевое слово указывает компилятору, что предпочтительнее встроенное расширение. Однако компилятор может создать отдельный экземпляр функции и вместо подстановки кода создать стандартную компоновку вызова. В двух случаях может возникнуть такая ситуация:

- Рекурсивные функции.
- Функции, на которые создаются ссылки посредством указателя в любом месте блока трансляции.

Эти причины могут повлиять на встраивание, *как и другие*, на усмотрение компилятора. не следует зависеть от `inline` спецификатора, чтобы сделать функцию встроенной.

Как и в случае с обычными функциями, в встроенной функции не определен порядок вычисления аргументов. На самом деле он может отличаться от порядка вычисления аргументов при передаче с использованием обычного протокола вызова функций.

`/Ob` Параметр оптимизации компилятора позволяет определить, выполняется ли фактическое расширение функции.

`/LTCG` выполняет Межмодульное встраивание независимо от того, запрошены ли они в исходном коде.

Пример 1

```
// inline_keyword1.cpp
// compile with: /c
inline int max( int a , int b ) {
    if( a > b )
        return a;
    return b;
}
```

Функции-члены класса могут быть объявлены встроенными либо с помощью `inline` ключевого слова, либо путем помещения определения функции в определение класса.

Пример 2

```
// inline_keyword2.cpp
// compile with: /EHsc /c
#include <iostream>
using namespace std;

class MyClass {
public:
    void print() { cout << i << ' ' ; } // Implicitly inline
private:
    int i;
};
```

Только для систем Майкрософт

`_inline` Ключевое слово эквивалентно значению `inline`.

Даже при использовании `_forceinline` компилятор не может встроить код во всех обстоятельствах. Компилятор не может выполнить встраивание функции, если:

- Функция или ее вызывающий объект компилируются вместе с `/Ob0` (параметром по умолчанию для отладочных сборок).
- В функции и вызывающем объекте используются разные типы (в одном — обработка исключений C++, а в другом — структурированная).
- Функция имеет переменное число аргументов.
- Функция использует встроенную сборку, если она не скомпилирована с помощью `/Ox`, `/O1` или `/O2`.
- Функция является рекурсивной и не имеет `#pragma inline_recursion(on)` набора. С помощью этой директивы выполняется подстановка рекурсивных функций с глубиной по умолчанию, 16 вызовов. Чтобы уменьшить глубину встраивания, используйте `inline_depth` директиву pragma.
- Функция является виртуальной, и для нее используется виртуальный вызов. Прямые вызовы виртуальных функций могут подставляться.
- Программа принимает адрес функции, и вызов совершается через указатель на функцию. Прямые вызовы функций, чей адрес был принят, могут подставляться.
- Функция также помечается `naked` `_declspec` модификатором.

Если компилятор не может выполнить встраивание функции, объявленной с помощью `_forceinline`, создается предупреждение уровня 1, за исключением случаев, когда:

- Функция компилируется с помощью /OD или /Ob0. В таких случаях встраивание не ожидается.

- Функция определена извне, в включенной библиотеке или другой записи преобразования либо является виртуальным целевым объектом вызова или адресатом косвенного вызова. Компилятор не может определить невстроенный код, который не удается найти в текущей записи преобразования.

Рекурсивные функции могут быть заменены встроенным кодом на глубину, заданную [inline_depth](#) директивой `pragma`, до максимум 16 вызовов. Начиная с этой глубины рекурсивные функции обрабатываются как вызовы на экземпляр функции. Глубина, для которой рекурсивные функции проверяются встроенным эвристическим методом, не может превышать 16. [inline_recursion](#) Директива `pragma` управляет встроенным расширением функции, находящегося в данный момент в расширении. Дополнительные сведения см. в описании параметра компилятора для [расширения встроенной функции](#) (/OB).

Завершение блока, относящегося только к системам Microsoft

Дополнительные сведения об использовании `inline` спецификатора см. в следующих статьях:

- [Подставляемые функции-члены класса](#)
- [Определение встроенных функций C++ с помощью `dllexport` и `dllimport`](#)

Когда использовать встраиваемые функции

Встроенные функции лучше использовать для небольших функций, например функций доступа к закрытым элементам данных. Основной целью таких функций метода доступа является возврат сведений о состоянии объектов. Короткие функции чувствительны к издержкам вызовов функций. Более длинные функции тратят пропорционально меньше времени в вызове и возвращают последовательность и выделяют меньше от встраивания.

`Point` Класс можно определить следующим образом:

```
// when_to_use_inline_functions.cpp
class Point
{
public:
    // Define "accessor" functions as
    // reference types.
    unsigned& x();
    unsigned& y();
private:
    unsigned _x;
    unsigned _y;
};

inline unsigned& Point::x()
{
    return _x;
}
inline unsigned& Point::y()
{
    return _y;
}
int main()
{
}
```

Предполагается, что обработка координат является относительно распространенной операцией в клиенте такого класса, указывая две функции доступа (`x` и `y` в предыдущем примере), как `inline` правило, позволяет экономить издержки:

- вызовы функций (включая передачу параметров и размещение адреса объекта в стеке);
- сохранение кадра стека вызывающего объекта;
- Настройка нового кадра стека
- передачу возвращаемого значения;
- Восстановление старого кадра стека
- Возвращает

Встроенные функции и макросы

Встроенные функции похожи на макросы, так как код функции разворачивается в точке вызова во время компиляции. Однако встроенные функции анализируются компилятором, а макросы развертываются препроцессором. В результате, имеется несколько важных различий.

- Встраиваемые функции выполняют все протоколы безопасности типов, обязательные для обычных функций.
- Встроенные функции указываются с использованием того же синтаксиса, что и любая другая функция, за исключением того, что они содержат `inline` ключевое слово в объявлении функции.
- Выражения, передаваемые во встраиваемые функции в качестве аргументов, вычисляются один раз. В некоторых случаях выражения, передаваемые в макросы в качестве аргументов, можно вычислить несколько раз.

В следующем примере показан макрос, преобразующий строчные буквы в прописные.

```
// inline_functions_macro.c
#include <stdio.h>
#include <conio.h>

#define toupper(a) ((a) >= 'a' && ((a) <= 'z') ? ((a)-('a'-'A')):(a))

int main() {
    char ch;
    printf_s("Enter a character: ");
    ch = toupper( getc(stdin) );
    printf_s( "%c", ch );
}

// Sample Input: xyz
// Sample Output: Z
```

Цель выражения заключается в том `toupper(getc(stdin))`, что символ должен быть считан с консольного устройства (`stdin`) и, при необходимости, преобразован в верхний регистр.

Из-за реализации макроса `getc` выполняется один раз для определения того, что символ больше или равен "a", и один раз, чтобы определить, меньше ли он или равен z. Если символ находится в этом диапазоне, `getc` выполняется еще раз для преобразования символа в прописную букву. Это означает, что программа ожидает два или три символа, когда, в идеале, будет ожидать только один.

Подставляемые функции позволяют устраниить описанную выше проблему.

```
// inline_functions_inline.cpp
#include <stdio.h>
#include <conio.h>

inline char toupper( char a ) {
    return ((a >= 'a' && a <= 'z') ? a-('a'-'A') : a );
}

int main() {
    printf_s("Enter a character: ");
    char ch = toupper( getc(stdin) );
    printf_s( "%c", ch );
}
```

Sample Input: a
Sample Output: A

См. также раздел

[noinline](#)

[auto_inline](#)

Перегрузка операторов

12.11.2021 • 2 minutes to read

operator Ключевое слово объявляет функцию, указывающую, какой *оператор-Symbol* означает при применении к экземплярам класса. Это дает оператору более одного значения — "перегружает" его. Компилятор различает разные значения оператора, проверяя типы его operandов.

Синтаксис

тип operator operator-символ(parameter-list)

Remarks

Функцию большинства встроенных операторов можно переопределить глобально или для отдельных классов. Перегруженные операторы реализуются в виде функции.

Имя перегруженного оператора — `operator x`, где `x` — это оператор, как показано в следующей таблице. Например, для перегрузки оператора сложения необходимо определить функцию с именем `operator +`. Аналогично, чтобы перегрузить оператор сложения и присваивания, `+ =` Определите функцию с именем `operator +=`.

Переопределяемые операторы

ОПЕРАТОР	ИМЯ	ТИП
,	Запятая	Двоичные данные
!	Логическое НЕ	Унарный
!=	Неравенство	Двоичные данные
%	Modulus	Двоичные данные
%=	Назначение модуля	Двоичные данные
&	Побитовое И	Двоичные данные
&	Взятие адреса	Унарный
&&	Логическое И	Двоичные данные
&=	Назначение побитового И	Двоичные данные
()	Вызов функции	—
()	Оператор приведения	Унарный
*	Умножение	Двоичные данные

ОПЕРАТОР	ИМЯ	ТИП
*	Разыменование указателя	Унарный
* =	Присваивание умножения	Двоичные данные
+	Сложение	Двоичные данные
+	Унарный плюс	Унарный
++	Шаг ¹	Унарный
+ =	Присваивание сложения	Двоичные данные
-	Вычитание	Двоичные данные
-	Унарное отрицание	Унарный
--	Уменьшить ¹	Унарный
- =	Присваивание вычитания	Двоичные данные
- >	Выбор члена	Двоичные данные
— > *	Выбор указателя на член	Двоичные данные
/	Отдел	Двоичные данные
/ =	Присваивание деления	Двоичные данные
<	Меньше чем	Двоичные данные
<<	Сдвиг влево	Двоичные данные
<< =	Сдвиг влево и присваивание	Двоичные данные
<=	Меньше или равно	Двоичные данные
=	Назначение	Двоичные данные
==	Равенство	Двоичные данные
>	Больше	Двоичные данные
> =	Больше или равно	Двоичные данные
>>	Сдвиг вправо	Двоичные данные
>> =	Сдвиг вправо и присваивание	Двоичные данные
[]	Индекс массива	—

ОПЕРАТОР	ИМЯ	ТИП
<code>^</code>	Исключающее ИЛИ	Двоичные данные
<code>^=</code>	Исключающее ИЛИ/присваивание	Двоичные данные
<code> </code>	Побитовое ИЛИ	Двоичные данные
<code> =</code>	Назначение побитового включающего ИЛИ	Двоичные данные
<code> </code>	Логическое ИЛИ	Двоичные данные
<code>~</code>	Дополнение до единицы	Унарный
<code>delete</code>	Удаление	—
<code>new</code>	Создать	—
операторы преобразования	операторы преобразования	Унарный

Существует ¹ две версии унарных операторов инкремента и декремента: добавочное и инкрементное.

Дополнительные сведения см. в разделе [Общие правила перегрузки операторов](#). Ограничения для разных категорий перегруженных операторов описываются в следующих разделах.

- [Унарные операторы](#)
- [Бинарные операторы](#)
- [Назначение](#)
- [Вызов функции](#)
- [Индексация ;](#)
- [Обращение к членам класса](#)
- [Инкремент и декремента.](#)
- [Преобразования определяемого пользователем типа](#)

Операторы, перечисленные в следующей таблице, не могут быть перегружены. Таблица содержит символы препроцессора # и ## .

Непереопределяемые операторы

ОПЕРАТОР	ИМЯ
<code>.</code>	Выбор члена
<code>. *</code>	Выбор указателя на член
<code>::</code>	Разрешение области
<code>? :</code>	Условная логика

ОПЕРАТОР	ИМЯ
#	Препроцессор: преобразование в строку
##	Препроцессор: конкатенация

Хотя перегруженные операторы обычно называются компилятором неявным образом при их появлении в коде, их можно вызывать и явным образом — точно так же, как и любую функцию-член или функцию, не являющуюся членом.

```
Point pt;
pt.operator+( 3 ); // Call addition operator to add 3 to pt.
```

Пример

В следующем примере оператор перегружается + для добавления двух комплексных чисел и возвращает результат.

```
// operator_overloading.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct Complex {
    Complex( double r, double i ) : re(r), im(i) {}
    Complex operator+( Complex &other );
    void Display( ) { cout << re << ", " << im << endl; }
private:
    double re, im;
};

// Operator overloaded using a member function
Complex Complex::operator+( Complex &other ) {
    return Complex( re + other.re, im + other.im );
}

int main() {
    Complex a = Complex( 1.2, 3.4 );
    Complex b = Complex( 5.6, 7.8 );
    Complex c = Complex( 0.0, 0.0 );

    c = a + b;
    c.Display();
}
```

6.8, 11.2

Содержимое раздела

- [Общие правила для перегрузки операторов](#)
- [Перегрузка унарных операторов](#)
- [Бинарные операторы](#)
- [Назначение](#)

- Вызов функции
- Индексация ;
- Доступ к членам

См. также раздел

[Операторы C++](#), приоритет и ассоциативность

[Ключевые слова](#)

Общие правила перегрузки операторов

12.11.2021 • 2 minutes to read

Приведенные ниже правила накладывают ограничения на реализацию перегруженных операторов. Однако они не применяются к операторам `New` и `Delete`, которые рассматриваются отдельно.

- Нельзя определить новые операторы, например .
- Не допускается переопределение операторов применительно ко встроенным типам данных.
- Перегруженные операторы должны быть нестатической функцией-членом класса или глобальной функцией. Глобальная функция, которой требуется доступ к частным или защищенным членам класса, должна быть объявлена в качестве дружественной функции этого класса. Глобальная функция должна принимать хотя бы один аргумент, имеющий тип класса или перечисляемый тип либо являющийся ссылкой на тип класса или перечисляемый тип. Пример:

```
// rules_for_operator_overloading.cpp
class Point
{
public:
    Point operator<( Point & ); // Declare a member operator
                                // overload.
    // Declare addition operators.
    friend Point operator+( Point&, int );
    friend Point operator+( int, Point& );
};

int main()
{}
```

В предыдущем примере кода оператор "меньше чем" объявляется как функция-член; однако операторы сложения объявляются как глобальные функции, имеющие дружественный доступ. Обратите внимание, что для каждого оператора можно предоставить несколько реализаций. Выше для оператора сложения предоставлены две реализации, обеспечивающие его коммутативность. Это так же вероятно, что могут быть реализованы операторы, добавляющие в, в `Point` `Point` `int` `Point` и т. д.

- Операторы подчиняются правилам приоритета, группирования и числа operandов, определяемым их типичным использованием со встроенными типами. Таким образом, невозможно выразить понятие Add 2 и 3 в объект типа `Point`, «ожидается добавление 2 к координате x и 3 для добавления к координате y ».
- Унарные операторы, объявленные как функции-члены, не принимают аргументов; при объявлении как глобальные функции они принимают один аргумент.
- Бинарные операторы, объявленные как функции-члены, принимают один аргумент; при объявлении как глобальные функции они принимают два аргумента.
- Если оператор можно использовать в качестве унарного или бинарного оператора (`&` , `*` , `+` и `-`), можно перегружать каждый вариант использования отдельно.
- Перегруженные операторы не могут иметь аргументов по умолчанию.
- Все перегруженные операторы, за исключением присваивания (`operator =`), наследуются

производными классами.

- Первым аргументом операторов, перегруженных в виде функций-членов, всегда является тип класса объекта, для которого вызывается этот оператор (класса, в котором объявлен оператор, или класса, производного от этого класса). Для первого аргумента никакие преобразования не предоставляются.

Обратите внимание, что значение любого оператора можно полностью изменить., Который включает значение адреса (&), присваивания (=) и операторов вызова функций. Кроме того, эквивалентность, на которую можно полагаться при использовании встроенных типов, может быть изменена при перегрузке операторов. Например, после полного вычисления следующие четыре оператора обычно эквивалентны:

```
var = var + 1;  
var += 1;  
var++;  
++var;
```

Для типов классов с перегруженными операторами на такую эквивалентность полагаться невозможно. Более того, некоторые из неявных требований, существующих при использовании этих операторов для базовых типов, для перегруженных операторов ослабляются. Например, оператор сложения/присваивания требует, += чтобы левый операнд был l-значением при применении к базовым типам. такое требование не предусмотрено, если оператор перегружен.

NOTE

Для согласованности при определении перегруженных операторов рекомендуется следовать модели для встроенных типов. Если семантика перегруженного оператора существенно отличается от его значения в других контекстах, это может скорее запутывать ситуацию, чем приносить пользу.

См. также

[Перегрузка операторов](#)

Перегрузка унарных операторов

12.11.2021 • 2 minutes to read

Перегрузке могут быть подвергнуты следующие унарные операторы.

1. `!` (логическое не)
2. `&` (адрес)
3. `~` (дополнение одного)
4. `*` (разыменование указателя)
5. `+` (унарный плюс)
6. `-` (унарное отрицание)
7. `++` (приращение)
8. `--` (декремента)
9. операторы преобразования

Постфиксные операторы инкремента и декремента (`++` и `--`) обрабатываются отдельно при [инкременте и декремента](#).

Операторы преобразования также обсуждаются в отдельном разделе. см. раздел [преобразование определяемого пользователем типа](#).

Следующие правила распространяются на все остальные унарные операторы. Чтобы объявить функцию унарного оператора как нестатический член, необходимо объявить ее в форме

```
RET-тип operator Op()
```

где *RET-Type* — это тип возвращаемого значения, а *Op* — один из операторов, перечисленных в предыдущей таблице.

Чтобы объявить функцию унарного оператора как глобальную функцию, необходимо объявить ее в форме

```
RET-тип operator Op( arg)
```

где *RET-Type* и *Op* описаны для функций *оператора-члена*, а *аргумент* является аргументом типа класса, с которым будет работать.

NOTE

Не существует ограничения на типы возвращаемого значения унарных операторов. Например, логическому НЕ (`!`) имеет смысл возвращать целочисленное значение, однако это не реализовано принудительно.

См. также раздел

[Перегрузка операторов](#)

Перегрузка операторов увеличения и уменьшения (C++)

12.11.2021 • 2 minutes to read

Операторы инкремента и декремента относятся к особой категории, поскольку имеется два варианта каждого из них:

- преинкрементный и постинкрементный операторы;
- предекрементный и постдекрементный операторы.

При написании функций перегруженных операторов полезно реализовать отдельные версии для префиксной и постфиксной форм этих операторов. Для различия этих двух правил наблюдается следующее правило: префиксная форма оператора объявляется точно так же, как любой другой унарный оператор. постфиксная форма принимает дополнительный аргумент типа `int`.

NOTE

При указании перегруженного оператора для постфиксной формы оператора инкремента или декремента дополнительный аргумент должен иметь тип `int`; при указании любого другого типа возникает ошибка.

В следующем примере показано определение операторов префиксных и постфиксных инкремента и декремента для класса `Point`.

```

// increment_and_decrement1.cpp
class Point
{
public:
    // Declare prefix and postfix increment operators.
    Point& operator++();           // Prefix increment operator.
    Point operator++(int);         // Postfix increment operator.

    // Declare prefix and postfix decrement operators.
    Point& operator--();           // Prefix decrement operator.
    Point operator--(int);         // Postfix decrement operator.

    // Define default constructor.
    Point() { _x = _y = 0; }

    // Define accessor functions.
    int x() { return _x; }
    int y() { return _y; }
private:
    int _x, _y;
};

// Define prefix increment operator.
Point& Point::operator++()
{
    _x++;
    _y++;
    return *this;
}

// Define postfix increment operator.
Point Point::operator++(int)
{
    Point temp = *this;
    ++*this;
    return temp;
}

// Define prefix decrement operator.
Point& Point::operator--()
{
    _x--;
    _y--;
    return *this;
}

// Define postfix decrement operator.
Point Point::operator--(int)
{
    Point temp = *this;
    --*this;
    return temp;
}
int main()
{
}

```

Те же операторы можно определить в области видимости файла (глобально) с помощью следующих заголовков функций:

```

friend Point& operator++( Point& )      // Prefix increment
friend Point& operator++( Point&, int ) // Postfix increment
friend Point& operator--( Point& )      // Prefix decrement
friend Point& operator--( Point&, int ) // Postfix decrement

```

Аргумент типа `int`, который обозначает постфиксную форму оператора инкремента или декремента, обычно не используется для передачи аргументов. Он обычно содержит значение 0. Однако его можно использовать следующим образом:

```
// increment_and_decrement2.cpp
class Int
{
public:
    Int &operator++( int n );
private:
    int _i;
};

Int& Int::operator++( int n )
{
    if( n != 0 )    // Handle case where an argument is passed.
        _i += n;
    else
        _i++;        // Handle case where no argument is passed.
    return *this;
}
int main()
{
    Int i;
    i.operator++( 25 ); // Increment by 25.
}
```

Для передачи этих значений с помощью операторов инкремента и декремента не существует иного синтаксиса, кроме явного вызова, как показано в предыдущем примере. Более простой способ реализации этой функции — перегрузить оператор сложения и присваивания (`+=`).

См. также

[Перегрузка операторов](#)

Бинарные операторы

12.11.2021 • 2 minutes to read

В следующей таблице приведен список операторов, которые можно перегрузить.

Переопределяемые бинарные операторы

ОПЕРАТОР	ИМЯ
,	Запятая
!=	Неравенство
%	Modulus
%=	Модуль/назначение
&	Побитовое И
&&	Логическое И
&=	Побитовое И/назначение
*	Умножение
*=	Умножение/назначение
+	Сложение
+ =	Сложение/назначение
-	Вычитание
- =	Вычитание/назначение
- >	Выбор члена
— > *	Выбор указателя на член
/	Отдел
/=	Деление/назначение
<	Меньше чем
< <	Сдвиг влево
< < =	Сдвиг влево/назначение

ОПЕРАТОР	ИМЯ
<code><=</code>	Меньше или равно
<code>=</code>	Назначение
<code>==</code>	Равенство
<code>></code>	Больше
<code>>=</code>	Больше или равно
<code>>></code>	Сдвиг вправо
<code>>>=</code>	Сдвиг вправо/назначение
<code>^</code>	Исключающее ИЛИ
<code>^=</code>	Исключающее ИЛИ/назначение
<code> </code>	Побитовое ИЛИ
<code> =</code>	Побитовое включающее ИЛИ/назначение
<code> </code>	Логическое ИЛИ

Чтобы объявить функцию бинарного оператора как нестатический член, необходимо объявить ее в виде

```
RET-тип operator Op( arg)
```

где *RET-Type* — это тип возвращаемого значения, *Op* является одним из операторов, перечисленных в предыдущей таблице, а аргумент *arg* является аргументом любого типа.

Чтобы объявить функцию бинарного оператора как глобальную функцию, необходимо объявить ее в виде

```
RET-тип operator Op( arg1, arg2)
```

где *RET-Type* и *Op* описаны для функций оператора-члена, а *arg1* и *arg2* — это аргументы. Хотя бы один из аргументов должен принадлежать типу класса.

NOTE

Ограничений на типы возвращаемого значения бинарных операторов нет, однако большинство пользовательских бинарных операторов возвращает тип класса или ссылку на тип класса.

См. также

[Перегрузка операторов](#)

Назначение

12.11.2021 • 2 minutes to read

Оператор присваивания (=), строго говоря, является бинарным оператором. Его объявление идентично объявлению любого другого бинарного оператора, со следующими исключениями:

- Он должен быть нестатической функцией-членом. **Оператор** = не может быть объявлен как функция, не являющийся членом.
- Он не наследуется производными классами.
- Компилятор может создать функцию operator = по умолчанию для типов классов, если она не существует.

В следующем примере показано, как объявить оператор присваивания:

```
class Point
{
public:
    int _x, _y;

    // Right side of copy assignment is the argument.
    Point& operator=(const Point&);

};

// Define copy assignment operator.
Point& Point::operator=(const Point& otherPoint)
{
    _x = otherPoint._x;
    _y = otherPoint._y;

    // Assignment operator returns left side of assignment.
    return *this;
}

int main()
{
    Point pt1, pt2;
    pt1 = pt2;
}
```

Переданный аргумент является правой стороной выражения. Оператор возвращает объект для сохранения поведения оператора присваивания, который после завершения присваивания возвращает значение левой части. Это позволяет создавать цепочки назначений, например:

```
pt1 = pt2 = pt3;
```

Оператор присваивания копии не следует путать с конструктором копии. Второй метод вызывается во время создания нового объекта из существующего.

```
// Copy constructor is called--not overloaded copy assignment operator!
Point pt3 = pt1;

// The previous initialization is similar to the following:
Point pt4(pt1); // Copy constructor call.
```

NOTE

Рекомендуется следовать [правилам трех](#), чтобы класс, определяющий оператор присваивания копии, также явно определял конструктор копии, деструктор и, начиная с C++ 11, конструктор перемещения и оператор присваивания перемещения.

См. также

- [Перегрузка операторов](#)
- [Конструкторы копий и операторы присваивания копий \(C++\)](#)

Вызов функций (C++)

12.11.2021 • 2 minutes to read

Оператор вызова функции, записываемый при помощи скобок, является бинарным оператором.

Синтаксис

```
primary-expression ( expression-list )
```

Remarks

В этом контексте `primary-expression` является первым operandом, а `expression-list` (список аргументов, который может быть пустым) — вторым. Оператор вызова функции используется для операций, для которых необходимо определенное количество параметров. Такой подход действует, потому что `expression-list` представляет собой не отдельный operand, а список. Оператор вызова функции должен представлять собой нестатическую функцию-член.

Перезагрузка оператора вызова функции не меняет способ вызова функций; она лишь изменяет способ интерпретации оператора при применении к объектам заданного типа класса. К примеру, приведенный ниже код в обычных условиях не имеет смысла.

```
Point pt;
pt( 3, 2 );
```

Однако, если оператор вызова функции был соответствующим образом перегружен, то этот синтаксис может использоваться для сдвига координаты `x` на три единицы, а координаты `y` — на две. Такое определение приводится в следующем примере.

```
// function_call.cpp
class Point
{
public:
    Point() { _x = _y = 0; }
    Point &operator()( int dx, int dy )
        { _x += dx; _y += dy; return *this; }
private:
    int _x, _y;
};

int main()
{
    Point pt;
    pt( 3, 2 );
}
```

Обратите внимание, что оператор вызова функции применяется к имени объекта, а не к имени функции.

Перегрузку оператора вызова функции также можно выполнить, используя указатель на функцию (вместо самой функции).

```
typedef void(*ptf)();
void func()
{
}
struct S
{
    operator ptf()
    {
        return func;
    }
};

int main()
{
    S s;
    s(); //operates as s.operator ptf()()
}
```

См. также

[Перегрузка операторов](#)

Индексация ;

12.11.2021 • 2 minutes to read

Оператор подстрочного индекса ([]), как и оператор вызова функции, считается бинарным оператором.

Оператор индекса должен быть нестатической функцией-членом, которая принимает один аргумент.

Этот аргумент может быть любого типа и определяет требуемый индекс массива.

Пример

В следующем примере показано, как создать вектор типа `int`, который реализует проверку границ:

```
// subscripting.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class IntVector {
public:
    IntVector( int cElements );
    ~IntVector() { delete [] _iElements; }
    int& operator[](int nSubscript);
private:
    int *_iElements;
    int _iUpperBound;
};

// Construct an IntVector.
IntVector::IntVector( int cElements ) {
    _iElements = new int[cElements];
    _iUpperBound = cElements;
}

// Subscript operator for IntVector.
int& IntVector::operator[](int nSubscript) {
    static int iErr = -1;

    if( nSubscript >= 0 && nSubscript < _iUpperBound )
        return _iElements[nSubscript];
    else {
        clog << "Array bounds violation." << endl;
        return iErr;
    }
}

// Test the IntVector class.
int main() {
    IntVector v( 10 );
    int i;

    for( i = 0; i <= 10; ++i )
        v[i] = i;

    v[3] = v[9];

    for ( i = 0; i <= 10; ++i )
        cout << "Element: [" << i << "] = " << v[i] << endl;
}
```

```
Array bounds violation.  
Element: [0] = 0  
Element: [1] = 1  
Element: [2] = 2  
Element: [3] = 9  
Element: [4] = 4  
Element: [5] = 5  
Element: [6] = 6  
Element: [7] = 7  
Element: [8] = 8  
Element: [9] = 9  
Array bounds violation.  
Element: [10] = 10
```

Комментарии

При `i` достижении 10 в предыдущей программе **оператор** `[]` обнаруживает, что индекс вне границ используется и выдает сообщение об ошибке.

Обратите внимание, что **оператор Function** `[]` возвращает ссылочный тип. В результате она является значением l-value, что позволяет использовать выражения с индексами с любой стороны операторов присваивания.

См. также

[Перегрузка операторов](#)

Доступ к членам

12.11.2021 • 2 minutes to read

Доступ к членам класса можно контролировать путем перегрузки оператора доступа к членам (->). В данном случае этот оператор считается унарным оператором, и функция перегруженного оператора должна быть функцией-членом класса. Поэтому объявление такой функции выглядит следующим образом.

Синтаксис

```
class-type *operator->()
```

Remarks

где *Class-Type* — это имя класса, к которому принадлежит этот оператор. Функция оператора доступа к члену должна быть нестатической функцией-членом.

Этот оператор используется (часто вместе с оператором разыменования указателя) для реализации интеллектуальных указателей, которые проверяют указатели до разыменования или подсчета.

Языковой элемент . оператор доступа к членам не может быть перегружен.

См. также

[Перегрузка операторов](#)

Классы и структуры (C++)

12.11.2021 • 2 minutes to read

В этом разделе приводится информация о классах и структурах C++. В C++ эти конструкции идентичны, за исключением того факта, что структуры по умолчанию открыты для доступа, а классы — закрыты.

Классы и структуры являются конструкциями, в которых пользователь определяет собственные типы. Классы и структуры могут включать данные-члены и функции-члены, позволяющие описывать состояние и поведение данного типа.

В этой статье содержатся следующие разделы:

- [class](#)
- [struct](#)
- [Общие сведения о членах класса](#)
- [Управление доступом к членам](#)
- [Наследование](#)
- [Статические члены](#)
- [Преобразования определяемого пользователем типа](#)
- [Изменяемые члены данных \(изменяемый спецификатор\)](#)
- [Объявления вложенных классов](#)
- [Типы анонимных классов](#)
- [Указатели на члены](#)
- [Этот указатель](#)
- [Битовые поля C++](#)

Существует три типа классов: структура, класс и объединение. Они объявляются с помощью ключевых слов `struct`, `Class` и `Union`. В следующей таблице показаны различия между этими тремя типами классов.

Дополнительные сведения о объединениях см. в разделе [объединения](#). Сведения о классах и структурах в C++/CLI и C++/CX см. в разделе [классы и структуры](#).

Управление доступом и ограничения для структур, классов и объединений

СТРУКТУРЫ	КЛАССЫ	ОБЪЕДИНЕНИЯ
ключ класса <code>struct</code>	ключ класса <code>class</code>	ключ класса <code>union</code>
Доступ по умолчанию: <code>public</code> (открытый).	Доступ по умолчанию: <code>private</code> (закрытый).	Доступ по умолчанию: <code>public</code> (открытый).
Нет ограничений на использование	Нет ограничений на использование	Используется только один член за один раз

См. также

[Справочник по языку C++](#)

class (C++)

12.11.2021 • 2 minutes to read

`class` Ключевое слово объявляет тип класса или определяет объект типа класса.

Синтаксис

```
[template-spec]
class [ms-decl-spec] [tag [: base-list ]]
{
    member-list
} [declarators];
[ class ] tag declarators;
```

Параметры

Спецификация шаблона

Необязательные спецификации шаблона. Дополнительные сведения см. в разделе [шаблоны](#).

`class`

`class` Ключевое слово.

MS-decl-Spec

Необязательная спецификация класса хранения. Дополнительные сведения см. в разделе [ключевое слово __declspec](#).

tag

Имя типа, присваиваемое классу. Этот параметр `tag` становится зарезервированным ключевым словом в области класса. Тег является необязательным. Если он опущен, определяется анонимный класс.

Дополнительные сведения см. в разделе [типы анонимных классов](#).

base-list

Необязательный список классов или структур, от которых этот класс будет наследовать члены.

Дополнительные сведения см. в разделе [базовые классы](#). Каждому базовому классу или имени структуры может предшествовать спецификатор доступа ([открытый](#), [частный](#), [защищенный](#)) и ключевое слово [Virtual](#). Дополнительные сведения см. в таблице доступ к элементам в разделе [Управление доступом к членам класса](#).

Список участников

Список членов класса. Дополнительные сведения см. в разделе [Обзор членов класса](#).

declarators

Список деклараторов, в котором указываются имена одного или нескольких экземпляров типа класса.

Деклараторы могут включать списки инициализаторов, если все элементы данных класса имеют значение `public`. Это более распространено в структурах, члены данных которых `public` по умолчанию имеют значение, а не в классах. Дополнительные сведения см. в разделе [Общие сведения об деклараторах](#).

Remarks

Дополнительные сведения о классах в целом см. в следующих разделах:

- [struct](#)

- [union](#)
- [_multiple_inheritance](#)
- [_single_inheritance](#)
- [_virtual_inheritance](#)

Сведения об управляемых классах и структурах в C++/CLI и C++/CX см. в разделе [классы и структуры](#).

Пример

```
// class.cpp
// compile with: /EHsc
// Example of the class keyword
// Exhibits polymorphism/virtual functions.

#include <iostream>
#include <string>
using namespace std;

class dog
{
public:
    dog()
    {
        _legs = 4;
        _bark = true;
    }

    void setDogSize(string dogSize)
    {
        _dogSize = dogSize;
    }
    virtual void setEars(string type)      // virtual function
    {
        _earType = type;
    }

private:
    string _dogSize, _earType;
    int _legs;
    bool _bark;

};

class breed : public dog
{
public:
    breed( string color, string size)
    {
        _color = color;
        setDogSize(size);
    }

    string getColor()
    {
        return _color;
    }

    // virtual function redefined
    void setEars(string length, string type)
    {
        _earLength = length;
        _earType = type;
    }
}
```

```
protected:  
    string _color, _earLength, _earType;  
};  
  
int main()  
{  
    dog mongrel;  
    breed labrador("yellow", "large");  
    mongrel.setEars("pointy");  
    labrador.setEars("long", "floppy");  
    cout << "Cody is a " << labrador.getColor() << " labrador" << endl;  
}
```

См. также

[Ключевые слова](#)

[Классы и структуры](#)

struct (C++)

12.11.2021 • 2 minutes to read

`struct` Ключевое слово определяет тип структуры и (или) переменную типа структуры.

Синтаксис

```
[template-spec] struct [ms-decl-spec] [tag [: base-list ]]
{
    member-list
} [declarators];
[struct] tag declarators;
```

Параметры

Спецификация шаблона

Необязательные спецификации шаблона. Дополнительные сведения см. в разделе [спецификации шаблонов](#).

`struct`

`struct` Ключевое слово.

MS-decl-Spec

Необязательная спецификация класса хранения. Дополнительные сведения см. в разделе [ключевое слово __declspec](#).

тег

Имя типа, присваиваемое структуре. Тег становится зарезервированным ключевым словом в области структуры. Тег является необязательным. Если он опущен, определяется анонимная структура.

Дополнительные сведения см. в разделе [типы анонимных классов](#).

base-list

Необязательный список классов или структур, из которых эта структура будет наследовать члены.

Дополнительные сведения см. в разделе [базовые классы](#). Каждому базовому классу или имени структуры может предшествовать спецификатор доступа ([открытый](#), [частный](#), [защищенный](#)) и ключевое слово

[Virtual](#). Дополнительные сведения см. в таблице доступ к элементам в разделе [Управление доступом к членам класса](#).

Список участников

Список членов структуры. Дополнительные сведения см. в разделе [Обзор членов класса](#). Единственное отличие заключается в том, что `struct` используется вместо `class`.

declarators

Список деклараторов, указывающий имена структуры. В списках деклараторов объявляются один или несколько экземпляров типа структуры. Деклараторы могут включать списки инициализаторов, если все элементы данных структуры имеют значение `public`. Списки инициализаторов являются общими в структурах, так как элементы данных `public` по умолчанию имеют значение. Дополнительные сведения см. в разделе [Общие сведения об деклараторах](#).

Remarks

Тип структуры — это пользовательский составной тип. Он состоит из полей или членов, которые могут

иметь разные типы.

В C++ структура является такой же, как и класс, за исключением того, что ее члены по `public` умолчанию имеют значение.

Сведения об управляемых классах и структурах в C++/CLI см. в разделе [классы и структуры](#).

Использование структуры

В языке C `struct` для объявления структуры необходимо явно использовать ключевое слово. В C++ не нужно использовать `struct` ключевое слово после определения типа.

Если тип структуры определен путем размещения одной или нескольких разделенных запятыми имен переменных между закрывающей фигурной скобкой и точкой с запятой, имеется возможность объявления переменных.

Переменные структуры можно инициализировать. Инициализация каждой переменной должна быть заключена в скобки.

Связанные сведения см. в разделе [класс, объединение и перечисление](#).

Пример

```
#include <iostream>
using namespace std;

struct PERSON {    // Declare PERSON struct type
    int age;      // Declare member types
    long ss;
    float weight;
    char name[25];
} family_member;    // Define object of type PERSON

struct CELL {    // Declare CELL bit field
    unsigned short character : 8;    // 00000000 ???????
    unsigned short foreground : 3;    // 00000??? 00000000
    unsigned short intensity : 1;    // 0000?000 00000000
    unsigned short background : 3;    // 0???0000 00000000
    unsigned short blink : 1;        // ?0000000 00000000
} screen[25][80];    // Array of bit fields

int main() {
    struct PERSON sister;    // C style structure declaration
    PERSON brother;    // C++ style structure declaration
    sister.age = 13;    // assign values to members
    brother.age = 7;
    cout << "sister.age = " << sister.age << '\n';
    cout << "brother.age = " << brother.age << '\n';

    CELL my_cell;
    my_cell.character = 1;
    cout << "my_cell.character = " << my_cell.character;
}
// Output:
// sister.age = 13
// brother.age = 7
// my_cell.character = 1
```

Обзор членов класса

12.11.2021 • 3 minutes to read

Класс или структура состоит из членов. Действия, выполняемые классом, выполняются с помощью его функций-членов. Состояние, в котором он находится, хранится в его элементах данных. Инициализация членов выполняется конструкторами, а очистка, например освобождение памяти и освобождение ресурсов, выполняется деструкторами. В C++ 11 и более поздних версиях элементы данных можно (и обычно следует) инициализировать при объявлении.

Виды членов класса

Ниже приведен полный список категорий членов.

- [Специальные функции элементов](#).
- [Общие сведения о функциях элементов](#).
- [Элементы данных](#), включая встроенные типы и другие определяемые пользователем типы.
- Операторы
- [Вложенные объявления классов и.\)](#)
- [Объединения](#)
- [Перечисления](#).
- [Битовые поля](#).
- [Друзья](#).
- [Псевдонимы и определения типов](#).

NOTE

Дружественные объекты включены в этот список, поскольку они содержатся в объявлении класса. Однако они не являются истинными членами класса, поскольку они не находятся в области класса.

Пример объявления класса

В следующем примере показано объявление простого класса.

```

// TestRun.h

class TestRun
{
    // Start member list.

    //The class interface accessible to all callers.
public:
    // Use compiler-generated default constructor:
    TestRun() = default;
    // Don't generate a copy constructor:
    TestRun(const TestRun&) = delete;
    TestRun(std::string name);
    void DoSomething();
    int Calculate(int a, double d);
    virtual ~TestRun();
    enum class State { Active, Suspended };

    // Accessible to this class and derived classes only.
protected:
    virtual void Initialize();
    virtual void Suspend();
    State GetState();

    // Accessible to this class only.
private:
    // Default brace-initialization of instance members:
    State _state{ State::Suspended };
    std::string _testName{ "" };
    int _index{ 0 };

    // Non-const static member:
    static int _instances;
    // End member list.
};

// Define and initialize static member.
int TestRun::_instances{ 0 };

```

Доступность членов

Члены класса объявляются в списке членов. Список членов класса можно разделить на любое количество `private`, `protected` и `public` разделов и с помощью ключевых слов, известных как описатели доступа. Двоеточие : должно следовать за спецификатором доступа. Эти разделы не обязательно должны быть непрерывными, то есть любое из этих ключевых слов может отображаться в списке элементов несколько раз. Ключевое слово назначает доступ всех членов до следующего описателя доступа или закрывающей фигурной скобки. Дополнительные сведения см. в разделе [Управление доступом к членам \(C++\)](#).

Статические члены

Элемент данных может быть объявлен как статический, что означает, что все объекты класса имеют доступ к одной и той же его копии. Функция-член может быть объявлена как статическая, в этом случае она может обращаться только к статическим членам данных класса (и не имеет этого указателя). Дополнительные сведения см. в разделе [статические элементы данных](#).

Специальные функции-члены

Специальные функции-члены — это функции, которые автоматически создаются компилятором, если не указаны в исходном коде.

1. Конструктор по умолчанию
2. Конструктор копии
3. (C++ 11) Конструктор перемещения
4. Оператор присваивания копированием
5. (C++ 11) Оператор присваивания перемещения
6. Деструктор

Дополнительные сведения см. в разделе [специальные функции элементов](#).

Поэлементная инициализация

В C++ 11 и более поздних версиях деклараторы нестатических членов могут содержать инициализаторы.

```
class CanInit
{
public:
    long num {7};           // OK in C++11
    int k = 9;              // OK in C++11
    static int i = 9; // Error: must be defined and initialized
                      // outside of class declaration.

    // initializes num to 7 and k to 9
    CanInit(){}

    // overwrites original initialized value of num:
    CanInit(int val) : num(val) {}

};

int main()
{}
```

Если в конструкторе члену присваивается значение, оно заменяет то значение, каким он был инициализирован при объявлении.

Существует только один общий экземпляр статических данных-членов для всех объектов заданного типа классов. Статические данные-члены необходимо определить и инициализировать в области файлов. (Дополнительные сведения о статических элементах данных см. в разделе [статические элементы данных](#).) В следующем примере показано, как выполнить эти инициализации:

```
// class_members2.cpp
class CanInit2
{
public:
    CanInit2() {} // Initializes num to 7 when new objects of type
                  // CanInit are created.
    long      num {7};
    static int i;
    static int j;
};

// At file scope:

// i is defined at file scope and initialized to 15.
// The initializer is evaluated in the scope of CanInit.
int CanInit2::i = 15;

// The right side of the initializer is in the scope
// of the object being initialized
int CanInit2::j = i;
```

NOTE

Имя класса, `CanInit2`, должно предшествовать `i`, чтобы указать, что `i` определяется как член класса `CanInit2`.

См. также

[Классы и структуры](#)

Управление доступом к членам (C++)

12.11.2021 • 5 minutes to read

Элементы управления доступом позволяют разделять [открытый](#) интерфейс класса от [закрытых](#) деталей реализации и [защищенных](#) членов, которые используются только производными классами. Спецификатор доступа действует для всех членов, объявленных после него, пока не будет объявлен следующий спецификатор доступа.

```
class Point
{
public:
    Point( int, int ) // Declare public constructor.;
    Point(); // Declare public default constructor.
    int &x( int ); // Declare public accessor.
    int &y( int ); // Declare public accessor.

private:           // Declare private state variables.
    int _x;
    int _y;

protected:        // Declare protected function for derived classes only.
    Point ToWindowCoords();
};
```

По умолчанию доступ осуществляется `private` в классе, `public` в структуре или объединении. Спецификаторы доступа класса могут использоваться любое количество раз и в любом порядке. Выделение хранилища для объектов типов классов зависит от реализации, но членам гарантировано присваиваются старшие адреса памяти, расположенные подряд между описателями доступа.

Управление доступом к членам

ТИП ДОСТУПА	ЗНАЧЕНИЕ
<code>private</code>	Члены класса, объявленные как, <code>private</code> могут использоваться только функциями-членами и друзьями (классами или функциями) класса.
<code>protected</code>	Члены класса, объявленные как, <code>protected</code> могут использоваться функциями-членами и друзьями (классами или функциями) класса. Кроме того, они могут использоваться производными классами данного класса.
<code>public</code>	Члены класса, объявленные как, <code>public</code> могут использоваться любой функцией.

Управление доступом помогает предотвратить использование объектов в неправомерных целях. Такая защита теряется при выполнении явных преобразований типов (приведении типов).

NOTE

Управление доступом одинаково применимо ко всем именам: функциям-членам, данным членам, вложенным классам и перечислителям.

Управление доступом в производном классе

Доступность в производном классе членов базового класса и унаследованных членов определяется двумя следующими факторами.

- Объявляет ли производный класс базовый класс с помощью `public` описателя доступа.
- Какой доступ к членам предоставляется в базовом классе.

В следующей таблице показана взаимосвязь между этими факторами и определен доступ к членам в базовом классе.

Доступ к членам в базовом классе

PRIVATE	PROTECTED	ОБЩЕДОСТУПНЫЕ
Всегда отсутствует независимо от доступа при наследовании	Закрытый в производном классе при использовании закрытого наследования	Закрытый в производном классе при использовании закрытого наследования
	Защищенный в производном классе при использовании защищенного наследования	Защищенный в производном классе при использовании защищенного наследования
	Защищенный в производном классе при использовании открытого наследования	Открытый в производном классе при использовании открытого наследования

Проиллюстрируем это на примере.

```

// access_specifiers_for_base_classes.cpp
class BaseClass
{
public:
    int PublicFunc(); // Declare a public member.
protected:
    int ProtectedFunc(); // Declare a protected member.
private:
    int PrivateFunc(); // Declare a private member.
};

// Declare two classes derived from BaseClass.
class DerivedClass1 : public BaseClass
{
    void foo()
    {
        PublicFunc();
        ProtectedFunc();
        PrivateFunc(); // function is inaccessible
    }
};

class DerivedClass2 : private BaseClass
{
    void foo()
    {
        PublicFunc();
        ProtectedFunc();
        PrivateFunc(); // function is inaccessible
    }
};

int main()
{
    DerivedClass1 derived_class1;
    DerivedClass2 derived_class2;
    derived_class1.PublicFunc();
    derived_class2.PublicFunc(); // function is inaccessible
}

```

В классе `DerivedClass1` функция-член `PublicFunc` является открытым членом, а функция `ProtectedFunc` — защищенным членом, поскольку `BaseClass` — открытый базовый класс. Функция `PrivateFunc` закрыта для класса `BaseClass` и не доступна всем производным классам.

В классе `DerivedClass2` функции `PublicFunc` и `ProtectedFunc` считаются закрытыми членами, поскольку `BaseClass` — закрытый базовый класс. Функция `PrivateFunc` снова закрыта для класса `BaseClass` и не доступна всем производным классам.

Производный класс можно объявить без спецификатора доступа базового класса. В этом случае наследование считается закрытым, если в объявлении производного класса используется `class` ключевое слово. Наследование считается открытым, если в объявлении производного класса используется `struct` ключевое слово. Например, приведенный ниже код

```

class Derived : Base
...

```

эквивалентно выражению

```

class Derived : private Base
...

```

Аналогично код

```
struct Derived : Base  
{  
    ...
```

эквивалентно выражению

```
struct Derived : public Base  
{  
    ...
```

Обратите внимание, что члены, объявленные как имеющие закрытый доступ, недоступны для функций или производных классов, если эти функции или классы не объявлены с помощью `friend` объявления в базовом классе.

`union` Тип не может иметь базовый класс.

NOTE

При указании закрытого базового класса рекомендуется явно использовать `private` ключевое слово, чтобы пользователи производного класса могли понимать доступ к членам.

Управление доступом и статические члены

Если базовый класс указан как `private`, он влияет только на нестатические члены. Открытые статические члены по-прежнему доступны в производных классах. Однако доступ к членам базового класса с помощью указателей, ссылок и объектов может потребовать преобразования, а во время выполнения управление доступом применяется снова. Рассмотрим следующий пример.

```
// access_control.cpp  
class Base  
{  
public:  
    int Print();           // Nonstatic member.  
    static int CountOf(); // Static member.  
};  
  
// Derived1 declares Base as a private base class.  
class Derived1 : private Base  
{  
};  
// Derived2 declares Derived1 as a public base class.  
class Derived2 : public Derived1  
{  
    int ShowCount();      // Nonstatic member.  
};  
// Define ShowCount function for Derived2.  
int Derived2::ShowCount()  
{  
    // Call static member function CountOf explicitly.  
    int cCount = Base::CountOf(); // OK.  
  
    // Call static member function CountOf using pointer.  
    cCount = this->CountOf(); // C2247. Conversion of  
                            // Derived2 * to Base * not  
                            // permitted.  
    return cCount;  
}
```

В предыдущем примере кода управление доступом запрещает преобразование указателя на `Derived2` к указателю на `Base . this`. Указатель неявно имеет тип `Derived2 *`. Чтобы выбрать `Countof` функцию, `this` необходимо преобразовать ее в тип `Base *`. Такое преобразование не поддерживается, поскольку `Base` — это закрытый косвенный базовый класс в `Derived2`. Преобразование к типу закрытого базового класса применимо только для указателей на непосредственные производные классы. Поэтому указатели типа `Derived1 *` можно преобразовать к типу `Base *`.

Обратите внимание, что при явном вызове функции `Countof` без использования указателя, ссылки или объекта для выделения этой функции, преобразование не выполняется. Поэтому вызов разрешен.

Члены и дружественные функции производного класса, `T`, могут преобразовать указатель на `T` в указатель на частный прямой базовый класс `T`.

Доступ к виртуальным функциям

Управление доступом к [виртуальным](#) функциям определяется типом, используемым для выполнения вызова функции. Переопределение объявлений функции не влияет на управление доступом для данного типа. Пример:

```
// access_to_virtual_functions.cpp
class VFuncBase
{
public:
    virtual int GetState() { return _state; }
protected:
    int _state;
};

class VFuncDerived : public VFuncBase
{
private:
    int GetState() { return _state; }
};

int main()
{
    VFuncDerived vfd;           // Object of derived type.
    VFuncBase *pvfb = &vfd;     // Pointer to base type.
    VFuncDerived *pvfd = &vfd;   // Pointer to derived type.

    int State;

    State = pvfb->GetState();  // GetState is public.
    State = pvfd->GetState();  // C2248 error expected; GetState is private;
}
```

В приведенном выше примере вызов виртуальной функции `GetState` с помощью указателя на тип `VFuncBase` приводит к вызову функции `VFuncDerived::GetState`, и функция `GetState` обрабатывается как открытая. Однако вызов `GetState` с помощью указателя на тип `VFuncDerived` нарушает правила управления доступом, поскольку функция `GetState` в классе `VFuncDerived` объявляется закрытой.

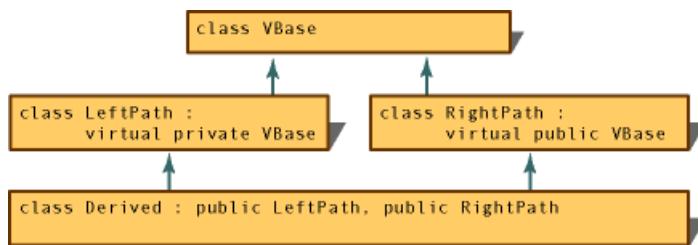
Caution

Виртуальную функцию `GetState` можно вызвать с помощью указателя на базовый класс `VFuncBase`. Это не означает, что вызываемая функция является версией базового класса данной функции.

Управление доступом с множественным наследованием

В решетках множественного наследования, в которых используются виртуальные базовые классы, к конкретным именам можно обращаться несколькими путями. Поскольку в разных путях могут применяться разные средства управления доступом, компилятор выбирает тот путь, который позволяет

получить наиболее широкий доступ. См. следующий рисунок.



Доступ вдоль линий графа наследования

На этом рисунке обращение к имени, которое было объявлено в классе `VBase`, всегда будет выполняться через класс `RightPath`. Путь справа дает более широкий доступ, поскольку в `RightPath` класс `VBase` объявлен как общедоступный базовый, а в `LeftPath` класс `VBase` объявлен как закрытый.

См. также

[Справочник по языку C++](#)

friend (C++)

12.11.2021 • 5 minutes to read

В некоторых случаях удобнее предоставить доступ на уровне членов к функциям, которые не являются членами класса или всем членам в отдельном классе. Только реализатор класса может объявить, что является для него дружественным элементом. Функция или класс не могут объявить себя дружественным элементом для любого класса. В определении класса используйте `friend` ключевое слово и имя функции, не являющейся членом, или другой класс, чтобы предоставить ему доступ к закрытым и защищенным членам класса. В определении шаблона параметр типа может быть объявлен дружественным.

Синтаксис

```
class friend F  
friend F;
```

Объявления дружественных элементов

При объявлении дружественной функции, которая не была объявлена ранее, эта функция экспортируется во включающую область вне класса.

Функции, объявленные в дружественном объявлении, обрабатываются так, как если бы они были объявлены с помощью `extern` ключевого слова. Дополнительные сведения см. в разделе [extern](#).

Хотя функции с глобальной областью действия могут быть объявлены как дружественные до объявления своих прототипов, функции-члены не могут быть объявлены как дружественные функции до полного объявления их класса. В следующем коде показано, почему при этом возникает ошибка.

```
class ForwardDeclared; // Class name is known.  
class HasFriends  
{  
    friend int ForwardDeclared::IsAFriend(); // C2039 error expected  
};
```

В предыдущем примере в области действия вводится имя класса `ForwardDeclared`, но полное объявление — в частности, часть, в которой объявляется функция `IsAFriend`, — отсутствует. Поэтому `friend` объявление в классе `HasFriends` создает ошибку.

Начиная с C++ 11, существует две формы дружественных объявлений для класса:

```
friend class F;  
friend F;
```

Первая форма представляет новый класс `F`, если в самом внутреннем пространстве имен не найден существующий класс с таким именем. C++ 11. во второй форме не представлен новый класс. его можно использовать, если класс уже объявлен и должен использоваться при объявлении параметра типа шаблона или `typedef` как дружественного.

Используется `friend class F` , если тип, на который указывает ссылка, еще не объявлен:

```
namespace NS
{
    class M
    {
        friend class F; // Introduces F but doesn't define it
    };
}
```

```
namespace NS
{
    class M
    {
        friend F; // error C2433: 'NS::F': 'friend' not permitted on data declarations
    };
}
```

В следующем примере `friend F` ссылается на `F` класс, объявленный вне области видимости NS.

```
class F {};
namespace NS
{
    class M
    {
        friend F; // OK
    };
}
```

Используйте `friend F` для объявления параметра шаблона в качестве дружественного:

```
template <typename T>
class my_class
{
    friend T;
    //...
};
```

Используйте `friend F`, чтобы объявить `typedef` как `Friend`:

```
class Foo {};
typedef Foo F;

class G
{
    friend F; // OK
    friend class F // Error C2371 -- redefinition
};
```

Чтобы объявить два класса как дружественные друг другу, весь второй класс должен быть указан как дружественный для первого класса. Причина такого ограничения заключается в том, что компилятор получает достаточные сведения для объявления отдельных дружественных функций только в момент объявления второго класса.

NOTE

Хотя весь второй класс должен быть дружественным для первого класса, можно выбрать, какие функции первого класса будут дружественными для второго класса.

дружественные функции

`friend` Функция — это функция, которая не является членом класса, но имеет доступ к закрытым и защищенным членам класса. Дружественные функции не считаются членами класса; это обычные внешние функции с особыми правами доступа. Друзья не находятся в области класса и не вызываются с помощью операторов выбора членов (. и ->), если они не являются членами другого класса. `friend` Функция объявляется классом, который предоставляет доступ. `friend` Объявление можно поместить в любое место объявления класса. На него не влияют ключевые слова управления доступом.

В следующем примере показан класс `Point` и дружественная функция `ChangePrivate`. `friend` Функция имеет доступ к закрытым членам данных объекта, который `Point` он получает в качестве параметра.

```
// friend_functions.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }

private:
    int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
    Point sPoint;
    sPoint.PrintPrivate();
    ChangePrivate(sPoint);
    sPoint.PrintPrivate();
// Output: 0
    1
}
```

Члены класса как дружественные элементы

Функции-члены класса могут быть объявлены в других классах как дружественные. Рассмотрим следующий пример.

```
// classes_as_friends1.cpp
// compile with: /c
class B;

class A {
public:
    int Func1( B& b );

private:
    int Func2( B& b );
};

class B {
private:
    int _b;

    // A::Func1 is a friend function to class B
    // so A::Func1 has access to all members of B
    friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }    // OK
int A::Func2( B& b ) { return b._b; }    // C2248
```

В предыдущем примере дружественный доступ к классу `A::Func1(B&)` предоставляется только функции `B`. Таким образом, доступ к закрытому члену `_b` будет правильным в `Func1` классе, `A` но не в `Func2`.

`friend` Класс — это класс, все функции-члены которого являются дружественными функциями класса, то есть у функций-членов есть доступ к закрытым и защищенным членам другого класса. Предположим, что `friend` объявление в классе `B` было:

```
friend class A;
```

В этом случае все функции-члены из класса `A` имели бы дружественный доступ к классу `B`. В следующем коде приведен пример дружественного класса.

```

// classes_as_friends2.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class YourClass {
friend class YourOtherClass; // Declare a friend class
public:
    YourClass() : topSecret(0){}
    void printMember() { cout << topSecret << endl; }
private:
    int topSecret;
};

class YourOtherClass {
public:
    void change( YourClass& yc, int x ){yc.topSecret = x;}
};

int main() {
    YourClass yc1;
    YourOtherClass yoc1;
    yc1.printMember();
    yoc1.change( yc1, 5 );
    yc1.printMember();
}

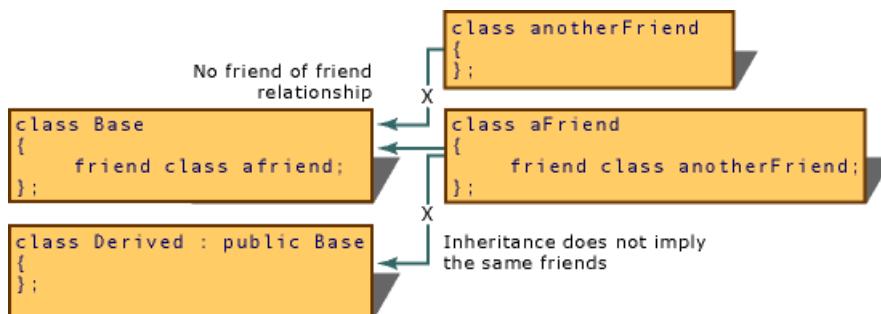
```

Дружественные отношения не являются взаимными, если это не указано явным образом. В предыдущем примере функции-члены класса `YourClass` не имеют доступа к закрытым членам класса `YourOtherClass`.

Управляемый тип (в C++/CLI) не может иметь дружественных функций, дружественных классов или дружественных интерфейсов.

Дружественные отношения не наследуются; это означает, что классы, производные от `YourOtherClass`, не могут обращаться к закрытым членам класса `YourClass`. Дружественные отношения не являются переходящими, поэтому классы, дружественные классу `YourOtherClass`, не могут обращаться к закрытым членам класса `YourClass`.

На следующем рисунке показаны объявления 4 классов: `Base`, `Derived`, `aFriend` и `anotherFriend`. Только класс `aFriend` имеет прямой доступ к закрытым членам класса `Base` (и к любым возможным унаследованным членам класса `Base`).



Последствия дружественных отношений

Встроенные определения дружественных элементов

Дружественные функции могут быть определены (при наличии тела функции) внутри объявлений класса. Эти функции являются встраиваемыми, и как встраиваемые функции членов они ведут себя так, как если бы они были определены сразу после просмотра всех членов класса, но до закрытия области класса (конец объявления класса). Дружественные функции, определенные внутри объявлений класса, находятся в области включающего класса.

См. также

[Ключевые слова](#)

private (C++)

12.11.2021 • 2 minutes to read

Синтаксис

```
private:  
    [member-list]  
private base-class
```

Remarks

Когда список членов класса предшествует, `private` ключевое слово указывает, что эти элементы доступны только в функциях-членах и друзьях класса. Это применяется ко всем членам, объявленным до следующего описателя доступа или до конца класса.

При предшествующем имени базового класса `private` ключевое слово указывает, что открытые и защищенные члены базового класса являются закрытыми членами производного класса.

Доступ членов в классе по умолчанию является закрытым. Доступ членов в структуре или объединении по умолчанию является открытым.

Доступ базового класса по умолчанию является закрытым для классов и открытым для структур. Объединения не могут иметь базовые классы.

Дополнительные сведения см. в разделе [дружественная, открытая, защищенная](#) и таблица доступа к членам в разделе [Управление доступом к членам класса](#).

Специально для /clr

В типах CLR ключевые слова спецификатора доступа C++ (`public`, `private` и `protected`) могут влиять на видимость типов и методов относительно сборок. Дополнительные сведения см. в разделе [Управление доступом к членам](#).

NOTE

Это поведение не влияет на файлы, скомпилированные с параметром `/LN`. В этом случае все управляемые классы (открытые или закрытые) будут видны.

КОНЕЦ специально для /clr

Пример

```
// keyword_private.cpp
class BaseClass {
public:
    // privMem accessible from member function
    int pubFunc() { return privMem; }
private:
    void privMem;
};

class DerivedClass : public BaseClass {
public:
    void usePrivate( int i )
        { privMem = i; } // C2248: privMem not accessible
                          // from derived class
};

class DerivedClass2 : private BaseClass {
public:
    // pubFunc() accessible from derived class
    int usePublic() { return pubFunc(); }
};

int main() {
    BaseClass aBase;
    DerivedClass aDerived;
    DerivedClass2 aDerived2;
    aBase.privMem = 1; // C2248: privMem not accessible
    aDerived.privMem = 1; // C2248: privMem not accessible
                         // in derived class
    aDerived2.pubFunc(); // C2247: pubFunc() is private in
                        // derived class
}
```

См. также раздел

[Управление доступом к членам классов](#)

[Ключевые слова](#)

protected (C++)

12.11.2021 • 2 minutes to read

Синтаксис

```
protected:  
    [member-list]  
protected base-class
```

Remarks

`protected` Ключевое слово задает доступ к членам класса в *списке Members* до следующего спецификатора доступа (`public` ИЛИ `private`) или в конце определения класса. Члены класса, объявленные как, `protected` МОГУТ использоваться только следующим образом:

- Функции-члены класса, который изначально объявил эти элементы.
- Дружественные элементы класса, который изначально объявил эти элементы.
- Классы, производные с использованием открытого или защищенного доступа от класса, который изначально объявил эти элементы.
- Прямые производные с использованием закрытого доступа классы, которые также имеют закрытый доступ к защищенным элементам.

При предшествующем имени базового класса `protected` ключевое слово указывает, что открытые и защищенные члены базового класса являются защищенными членами его производных классов.

Защищенные члены не являются частными как `private` члены, которые доступны только членам класса, в котором они объявляются, но не являются общими как `public` члены, доступные в любой функции.

Защищенные члены, которые также объявлены как `static`, доступны для любого дружественного или функции-члена производного класса. Защищенные члены, которые не объявлены как `static`, доступны друзьям и функциям-членам в производном классе только через указатель на, ссылку на или объект производного класса.

Дополнительные сведения см. в статьях [дружественный](#), [общедоступный](#), [частный](#) и таблица доступа к членам в разделе [Управление доступом к членам класса](#).

Специально для /clr

В типах CLR ключевые слова спецификатора доступа C++ (`public` , `private` И `protected`) могут влиять на видимость типов и методов относительно сборок. Дополнительные сведения см. в разделе [Управление доступом к членам](#).

NOTE

Это поведение не влияет на файлы, скомпилированные с параметром `/LN`. В этом случае все управляемые классы (открытые или закрытые) будут видны.

КОНЕЦ специально для /clr

Пример

```
// keyword_protected.cpp
// compile with: /EHsc
#include <iostream>

using namespace std;
class X {
public:
    void setProtMemb( int i ) { m_protMemb = i; }
    void Display() { cout << m_protMemb << endl; }
protected:
    int m_protMemb;
    void Protfunc() { cout << "\nAccess allowed\n"; }
} x;

class Y : public X {
public:
    void useProtfunc() { Protfunc(); }
} y;

int main() {
    // x.m_protMemb;           error, m_protMemb is protected
    x.setProtMemb( 0 );      // OK, uses public access function
    x.Display();
    y.setProtMemb( 5 );      // OK, uses public access function
    y.Display();
    // x.Protfunc();          error, Protfunc() is protected
    y.useProtfunc();         // OK, uses public access function
    // in derived class
}
```

См. также раздел

[Управление доступом к членам классов](#)

[Ключевые слова](#)

public (C++)

12.11.2021 • 2 minutes to read

Синтаксис

```
public:  
    [member-list]  
public base-class
```

Remarks

Когда список членов класса предшествует, `public` ключевое слово указывает, что эти элементы доступны из любой функции. Это применяется ко всем членам, объявленным до следующего описателя доступа или до конца класса.

При предшествующем имени базового класса `public` ключевое слово указывает, что открытые и защищенные члены базового класса являются открытыми и защищенными членами, соответственно производным классом.

Доступ членов в классе по умолчанию является закрытым. Доступ членов в структуре или объединении по умолчанию является открытым.

Доступ базового класса по умолчанию является закрытым для классов и открытым для структур. Объединения не могут иметь базовые классы.

Дополнительные сведения см. в статьях [закрытые](#), [защищенные](#), [дружественные](#) и таблица доступа к членам в разделе [Управление доступом к членам класса](#).

Специально для /cIg

В типах CLR ключевые слова спецификатора доступа C++ (`public`, `private` и `protected`) могут влиять на видимость типов и методов относительно сборок. Дополнительные сведения см. в разделе [Управление доступом к членам](#).

NOTE

Это поведение не влияет на файлы, скомпилированные с параметром `/LN`. В этом случае все управляемые классы (открытые или закрытые) будут видны.

КОНЕЦ специально для /cIg

Пример

```
// keyword_public.cpp
class BaseClass {
public:
    int pubFunc() { return 0; }
};

class DerivedClass : public BaseClass {};

int main() {
    BaseClass aBase;
    DerivedClass aDerived;
    aBase.pubFunc();           // pubFunc() is accessible
                             // from any function
    aDerived.pubFunc();        // pubFunc() is still public in
                             // derived class
}
```

См. также раздел

[Управление доступом к членам классов](#)

[Ключевые слова](#)

Инициализация фигурных скобок

12.11.2021 • 3 minutes to read

Не всегда обязательно определять конструктор для класса (особенно для относительно простых классов). Пользователи могут использовать для объектов класса или структур унифицированную инициализацию, как показано в следующем примере:

```
// no_constructor.cpp
// Compile with: cl /EHsc no_constructor.cpp
#include <time.h>

// No constructor
struct TempData
{
    int StationId;
    time_t timeSet;
    double current;
    double maxTemp;
    double minTemp;
};

// Has a constructor
struct TempData2
{
    TempData2(double minimum, double maximum, double cur, int id, time_t t) :
        stationId{id}, timeSet{t}, current{cur}, maxTemp{maximum}, minTemp{minimum} {}
    int stationId;
    time_t timeSet;
    double current;
    double maxTemp;
    double minTemp;
};

int main()
{
    time_t time_to_set;

    // Member initialization (in order of declaration):
    TempData td{ 45978, time(&time_to_set), 28.9, 37.0, 16.7 };

    // Default initialization = {0,0,0,0,0}
    TempData td_default{};

    // Uninitialized = if used, emits warning C4700 uninitialized local variable
    TempData td_noInit;

    // Member declaration (in order of ctor parameters)
    TempData2 td2{ 16.7, 37.0, 28.9, 45978, time(&time_to_set) };

    return 0;
}
```

Обратите внимание, что если класс или структура не имеет конструктора, вы предоставляете элементы списка в том порядке, в котором элементы объявляются в классе. Если у класса есть конструктор, укажите элементы в порядке параметров. Если у типа есть конструктор по умолчанию, явно или неявно объявленный, можно использовать инициализацию скобок по умолчанию (с пустыми фигурными скобками). Например, следующий класс можно инициализировать с помощью инициализации по умолчанию и стандартной фигурной скобки:

```

#include <string>
using namespace std;

class class_a {
public:
    class_a() {}
    class_a(string str) : m_string{ str } {}
    class_a(string str, double dbl) : m_string{ str }, m_double{ dbl } {}
    double m_double;
    string m_string;
};

int main()
{
    class_a c1{};
    class_a c1_1;

    class_a c2{ "ww" };
    class_a c2_1("xx");

    // order of parameters is the same as the constructor
    class_a c3{ "yy", 4.4 };
    class_a c3_1("zz", 5.5);
}

```

Если класс имеет конструкторы, отличные от конструктора по умолчанию, то порядок, в котором члены класса отображаются в инициализаторе фигурных скобок, является порядком, в котором соответствующие параметры отображаются в конструкторе, а не в том порядке, в котором объявляются элементы (как `class_a` в предыдущем примере). В противном случае, если тип не имеет объявленного конструктора, порядок, в котором элементы отображаются в инициализаторе фигурных скобок, совпадает с порядком, в котором они объявляются. в этом случае можно инициализировать столько открытых членов, сколько нужно, но нельзя пропустить ни одного элемента. В следующем примере показан порядок, который используется в инициализации фигурных скобок при отсутствии объявленного конструктора:

```

class class_d {
public:
    float m_float;
    string m_string;
    wchar_t m_char;
};

int main()
{
    class_d d1{};
    class_d d1{ 4.5 };
    class_d d2{ 4.5, "string" };
    class_d d3{ 4.5, "string", 'c' };

    class_d d4{ "string", 'c' }; // compiler error
    class_d d5{ "string", 'c', 2.0 }; // compiler error
}

```

Если конструктор по умолчанию объявлен явно, но помечен как удаленный, инициализацию по умолчанию нельзя использовать:

```
class class_f {
public:
    class_f() = delete;
    class_f(string x): m_string { x } {}
    string m_string;
};

int main()
{
    class_f cf{ "hello" };
    class_f cf1{}; // compiler error C2280: attempting to reference a deleted function
}
```

Можно использовать инициализацию фигурных скобок, где обычно выполняется инициализация, например в качестве параметра функции или возвращаемого значения или с помощью `new` ключевого слова:

```
class_d* cf = new class_d{4.5};
kr->add_d({ 4.5 });
return { 4.5 };
```

B/std: **режим c++ 17** правила инициализации пустой скобки немного более ограниченны. См. раздел [производные конструкторы и расширенная агрегатная инициализация](#).

конструкторы `initializer_list`

Класс `initializer_list` представляет список объектов указанного типа, которые могут использоваться в конструкторе и в других контекстах. Вы можете создать `initializer_list` с помощью инициализации фигурных скобок:

```
initializer_list<int> int_list{5, 6, 7};
```

IMPORTANT

Чтобы использовать этот класс, необходимо включить `<initializer_list>` заголовок.

`initializer_list` Можно скопировать. В этом случае члены нового списка являются ссылками на элементы исходного списка:

```
initializer_list<int> ilist1{ 5, 6, 7 };
initializer_list<int> ilist2( ilist1 );
if (ilist1.begin() == ilist2.begin())
    cout << "yes" << endl; // expect "yes"
```

Классы контейнеров стандартной библиотеки, а также, `string` `wstring` И, `regex` имеют `initializer_list` конструкторы. В следующих примерах показано, как выполнить инициализацию фигурных скобок с помощью этих конструкторов:

```
vector<int> v1{ 9, 10, 11 };
map<int, string> m1{ {1, "a"}, {2, "b"} };
string s{ 'a', 'b', 'c' };
regex rgx{ 'x', 'y', 'z' };
```

См. также

[Классы и структуры](#)

[Конструкторы](#)

Управление временем жизни и ресурсами объекта (RAII)

12.11.2021 • 2 minutes to read

В отличие от управляемых языков C++ не имеет автоматической *сборки мусора*. Это внутренний процесс, который освобождает память кучи и другие ресурсы при выполнении программы. Программа на языке C++ отвечает за возврат всех приобретенных ресурсов операционной системе. Сбой освобождения неиспользуемого ресурса называется *утечкой*. Потерянные ресурсы недоступны для других программ до завершения процесса. Утечки памяти в частности являются распространенной причиной ошибок в программировании в стиле C.

Современные C++ не позволяют использовать память кучи максимально точно, объявляя объекты в стеке. Если ресурс слишком велик для стека, он должен *принадлежать* объекту. При инициализации объекта он получает ресурс, который он владеет. Затем объект отвечает за освобождение ресурса в его деструкторе. Сам объект-владелец объявляется в стеке. Принцип, в котором *объекты владеют ресурсами*, также называется "получением ресурсов является инициализацией" или RAII.

Если объект стека, владеющий ресурсами, выходит за пределы области видимости, его деструктор вызывается автоматически. Таким образом, сборка мусора в C++ тесно связана с временем существования объекта и является детерминированной. Ресурс всегда освобождается в известной точке программы, которой можно управлять. Только детерминированные деструкторы, такие как, в C++, могут одновременно работать с памятью и не ресурсами памяти.

В следующем примере показан простой объект `w`. Он объявлен в стеке в области видимости функции и уничтожается в конце блока функции. Объект `w` не владеет *ресурсами* (например, памятью, выделенной в куче). `g` Сам по себе член сам по себе объявлен в стеке и просто выходит за пределы области действия вместе с `w`. В деструкторе не требуется специальный код `widget`.

```
class widget {
private:
    gadget g;    // lifetime automatically tied to enclosing object
public:
    void draw();
};

void functionUsingWidget () {
    widget w;    // lifetime automatically tied to enclosing scope
                 // constructs w, including the w.g gadget member
    // ...
    w.draw();
    // ...
} // automatic destruction and deallocation for w and w.g
// automatic exception safety,
// as if "finally { w.dispose(); w.g.dispose(); }"
```

В следующем примере `w` владельцем ресурса является память, и поэтому в его деструкторе должен быть код для удаления памяти.

```

class widget
{
private:
    int* data;
public:
    widget(const int size) { data = new int[size]; } // acquire
    ~widget() { delete[] data; } // release
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000);    // lifetime automatically tied to enclosing scope
                          // constructs w, including the w.data member
    w.do_something();

} // automatic destruction and deallocation for w and w.data

```

Начиная с C++ 11, существует лучший способ написать предыдущий пример: с помощью смарт-указателя из стандартной библиотеки. Интеллектуальный указатель обрабатывает выделение и удаление памяти, которую она владеет. Использование смарт-указателя устраняет необходимость в явном деструкторе в `widget` классе.

```

#include <memory>
class widget
{
private:
    std::unique_ptr<int> data;
public:
    widget(const int size) { data = std::make_unique<int>(size); }
    void do_something() {}
};

void functionUsingWidget() {
    widget w(1000000);    // lifetime automatically tied to enclosing scope
                          // constructs w, including the w.data gadget member
    // ...
    w.do_something();
    // ...
} // automatic destruction and deallocation for w and w.data

```

При использовании смарт-указателей для выделения памяти можно устраниить потенциальные утечки памяти. Эта модель подходит для других ресурсов, таких как дескрипторы файлов или сокеты. Вы можете управлять своими ресурсами аналогичным образом в своих классах. Дополнительные сведения см. в разделе [интеллектуальные указатели](#).

Конструкция C++ гарантирует, что объекты уничтожаются, когда выходят за пределы области. То есть они уничтожаются как блоки, завершаются в обратную последовательность конструкции. При уничтожении объекта его базовые объекты и члены уничтожаются в определенном порядке. Объекты, объявленные вне любого блока в глобальной области, могут привести к проблемам. Отладка может быть сложной, если конструктор глобального объекта вызывает исключение.

См. также

[Возвращение к C++](#)

[Справочник по языку C++](#)

[Стандартная библиотека C++](#)

Pimpl для инкапсуляции времени компиляции (современный C++)

12.11.2021 • 2 minutes to read

Идиома Пимпл является современной методикой C++ для скрытия реализации, для сворачивания связывания и разделения интерфейсов. Пимпл является коротким для "указателя на реализацию". Возможно, вы уже знакомы с концепцией, но знаете ее по другим именам, таким как Чешир Cat или идиома брандмауэра компилятора.

Зачем использовать Пимпл?

Вот как пимплная идиома может улучшить жизненный цикл разработки программного обеспечения:

- Минимизация зависимостей компиляции.
- Разделение интерфейса и реализации.
- Мобильность.

Заголовок Пимпл

```
// my_class.h
class my_class {
    // ... all public and protected stuff goes here ...
private:
    class impl; unique_ptr<impl> pimpl; // opaque type here
};
```

Идиом Пимпл позволяет избежать перестроения каскадных и нестабильных макетов объектов. Он хорошо подходит для (транзитивно) популярных типов.

Реализация Пимпл

Определите `impl` класс в cpp – файле.

```
// my_class.cpp
class my_class::impl { // defined privately here
    // ... all private data and functions: all of these
    //      can now change without recompiling callers ...
};

my_class::my_class(): pimpl( new impl )
{
    // ... set impl values ...
}
```

Рекомендации

Рассмотрите возможность добавления поддержки для специализации, не создающей исключение.

См. также раздел

[Возвращение к C++](#)

[Справочник по языку C++](#)

[Стандартная библиотека C++](#)

Переносимость на границах ABI

12.11.2021 • 2 minutes to read

Используйте достаточно переносимые типы и соглашения на границах двоичного интерфейса. "Portable Type" — это встроенный тип С или структура, содержащая только встроенные типы С. Типы классов можно использовать, только когда вызывающий и вызываемый принимает на себя макет, соглашение о вызовах и т. д. Это возможно только в том случае, если оба скомпилированы с одинаковыми параметрами компилятора и компилятора.

Как свести класс к переносимости на С

Если вызывающие объекты могут быть скомпилированы с другим компилятором или языком, то "сведение" к внешнему API "С" с определенным соглашением о вызовах:

```
// class widget {  
//     widget();  
//     ~widget();  
//     double method( int, gadget& );  
// };  
extern "C" {      // functions using explicit "this"  
    struct widget; // opaque type (forward declaration only)  
    widget* STDCALL widget_create();      // constructor creates new "this"  
    void STDCALL widget_destroy(widget*); // destructor consumes "this"  
    double STDCALL widget_method(widget*, int, gadget*); // method uses "this"  
}
```

См. также

[Возвращение к C++](#)

[Справочник по языку C++](#)

[Стандартная библиотека C++](#)

Конструкторы (C++)

12.11.2021 • 16 minutes to read

Чтобы настроить способ инициализации элементов класса или вызвать функции при создании объекта класса, определите *конструктор*. Конструкторы имеют имена, совпадающие с именами классов, и не имеют возвращаемых значений. Можно определить столько перегруженных конструкторов, сколько необходимо для настройки инициализации различными способами. Как правило, конструкторы имеют открытый доступ, поэтому код за пределами определения класса или иерархии наследования может создавать объекты класса. Но можно также объявить конструктор как `protected` ИЛИ `private`.

При необходимости конструкторы могут получить список инициализации элемента. Это более эффективный способ инициализации членов класса, чем назначение значений в теле конструктора. В следующем примере показан класс `Box` с тремя перегруженными конструкторами. Последние два списка инициализации элементов `use`:

```
class Box {
public:
    // Default constructor
    Box() {}

    // Initialize a Box with equal dimensions (i.e. a cube)
    explicit Box(int i) : m_width(i), m_length(i), m_height(i) // member init list
    {}

    // Initialize a Box with custom dimensions
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height)
    {}

    int Volume() { return m_width * m_length * m_height; }

private:
    // Will have value of 0 when default constructor is called.
    // If we didn't zero-init here, default constructor would
    // leave them uninitialized with garbage values.
    int m_width{ 0 };
    int m_length{ 0 };
    int m_height{ 0 };
};
```

При объявлении экземпляра класса компилятор выбирает конструктор для вызова на основе правил разрешения перегрузки:

```
int main()
{
    Box b; // Calls Box()

    // Using uniform initialization (preferred):
    Box b2 {5}; // Calls Box(int)
    Box b3 {5, 8, 12}; // Calls Box(int, int, int)

    // Using function-style notation:
    Box b4(2, 4, 6); // Calls Box(int, int, int)
}
```

- Конструкторы могут быть объявлены как `inline`, **явные** `friend` ИЛИ `constexpr`.

- Конструктор может инициализировать объект, объявленный как `const`, `volatile` ИЛИ `const volatile`. Объект станет `const` после завершения конструктора.
- Чтобы определить конструктор в файле реализации, присвойте ему полное имя с любой другой функцией-членом: `Box::Box(){...}`.

Списки инициализаторов членов

Конструктор может дополнительно иметь список инициализаторов членов, который инициализирует члены класса перед выполнением тела конструктора. (Обратите внимание, что список инициализаторов членов не то же самое, что и *Список инициализаторов* типа `std <T> :: initializer_list`.)

Использование списка инициализаторов членов предпочтительнее, чем назначение значений в теле конструктора, так как он непосредственно Инициализирует элемент. В следующем примере показан список инициализаторов членов, состоящий из всех выражений **идентификаторов (аргументов)** после двоеточия:

```
Box(int width, int length, int height)
    : m_width(width), m_length(length), m_height(height)
{}
```

Идентификатор должен ссылаться на член класса; он инициализируется значением аргумента. Аргумент может быть одним из параметров конструктора, вызовом функции или `std:: initializer_list <T>`.

`const` члены и члены ссылочного типа должны быть инициализированы в списке инициализаторов членов.

В списке инициализаторов должны быть сделаны вызовы для параметризованных конструкторов базового класса, чтобы гарантировать, что базовый класс полностью инициализирован до выполнения производного конструктора.

Конструкторы по умолчанию

Конструкторы по умолчанию обычно не имеют параметров, но могут иметь параметры со значениями по умолчанию.

```
class Box {
public:
    Box() { /*perform any required default initialization steps*/}

    // All params have default values
    Box (int w = 1, int l = 1, int h = 1): m_width(w), m_height(h), m_length(l){}
    ...
}
```

Конструкторы по умолчанию являются одной из [специальных функций элементов](#). Если в классе не объявлен ни один конструктор, компилятор предоставляет неявный `inline` конструктор по умолчанию.

```

#include <iostream>
using namespace std;

class Box {
public:
    int Volume() {return m_width * m_height * m_length;}
private:
    int m_width { 0 };
    int m_height { 0 };
    int m_length { 0 };
};

int main() {
    Box box1; // Invoke compiler-generated constructor
    cout << "box1.Volume: " << box1.Volume() << endl; // Outputs 0
}

```

Если вы полагаетесь на неявный конструктор по умолчанию, обязательно инициализируйте элементы в определении класса, как показано в предыдущем примере. Без этих инициализаторов члены будут неинициализированы, а вызов Volume () создаст значение мусора. Как правило, рекомендуется инициализировать элементы таким образом, даже если не полагается на неявный конструктор по умолчанию.

Можно запретить компилятору создавать неявный конструктор по умолчанию, определив его как [Удаленный](#):

```

// Default constructor
Box() = delete;

```

Созданный компилятором конструктор по умолчанию будет определен как удаленный, если какие-либо члены класса не являются конструируемыми по умолчанию. Например, все члены типа класса и их члены типа класса должны иметь доступ к конструктору по умолчанию и деструкторам, которые доступны. Все элементы данных ссылочного типа, а также `const` члены класса должны иметь инициализатор членов по умолчанию.

При вызове конструктора, созданного компилятором по умолчанию, и попытке использовать круглые скобки выдается предупреждение:

```

class myclass{};
int main(){
    myclass mc();      // warning C4930: prototyped function not called (was a variable definition intended?)
}

```

Это пример проблемы Most Vexing Parse (наиболее неоднозначного анализа). Поскольку выражение примера можно интерпретировать как объявление функции или как вызов конструктора по умолчанию и в связи с тем, что средства синтаксического анализа C++ отдают предпочтение объявлению перед другими действиями, данное выражение обрабатывается как объявление функции. Дополнительные сведения см. в разделе [досадной Parse](#).

В случае явного объявления конструкторов компилятор не предоставляет конструктор по умолчанию:

```
class Box {  
public:  
    Box(int width, int length, int height)  
        : m_width(width), m_length(length), m_height(height){}  
private:  
    int m_width;  
    int m_length;  
    int m_height;  
  
};  
  
int main(){  
  
    Box box1(1, 2, 3);  
    Box box2{ 2, 3, 4 };  
    Box box3; // C2512: no appropriate default constructor available  
}
```

Если у класса нет конструктора по умолчанию, массив объектов этого класса не может быть создан только с помощью синтаксиса двух квадратных скобок. Например, в представленном выше блоке кода массив Boxes не может быть объявлен следующим образом:

```
Box boxes[3]; // C2512: no appropriate default constructor available
```

Однако можно использовать набор списков инициализаторов для инициализации массива объектов Box:

```
Box boxes[3]{ { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

Дополнительные сведения см. в разделе [инициализаторы](#).

Конструкторы копий

Конструктор копии Инициализирует объект, копируя значения элементов из объекта того же типа. Если все члены класса являются простыми типами, такими как скалярные значения, конструктор копий, созданный компилятором, достаточно, и вам не нужно определять собственный. Если для класса требуется более сложная инициализация, необходимо реализовать пользовательский конструктор копии. Например, если член класса является указателем, необходимо определить конструктор копии, чтобы выделить новую память и скопировать значения из объекта, указывающего на другой объект. Созданный компилятором конструктор копий просто копирует указатель, так что новый указатель по-прежнему указывает на расположение в памяти другого.

Конструктор копии может иметь одну из следующих сигнатур:

```
Box(Box& other); // Avoid if possible--allows modification of other.  
Box(const Box& other);  
Box(volatile Box& other);  
Box(volatile const Box& other);  
  
// Additional parameters OK if they have default values  
Box(Box& other, int i = 42, string label = "Box");
```

При определении конструктора копии необходимо также определить оператор присваивания копирования (=). Дополнительные сведения см. в разделе конструкторы [присваивания](#) и [копирования](#) и [операторы присваивания копирования](#).

Вы можете запретить копирование объекта, определив конструктор копии как удаленный:

```
Box (const Box& other) = delete;
```

Попытка копирования объекта приводит к ошибке *C2280: попытка ссылки на удаленную функцию*.

Конструкторы перемещения

Конструктор перемещения — это специальная функция-член, которая перемещает владение данными существующего объекта в новую переменную без копирования исходных данных. Он принимает в качестве первого параметра ссылку `rvalue`, а все дополнительные параметры должны иметь значения по умолчанию. Конструкторы перемещения могут значительно повысить эффективность программы при передаче больших объектов.

```
Box(Box&& other);
```

Компилятор выбирает конструктор перемещения в определенных ситуациях, где объект инициализируется другим объектом того же типа, который будет уничтожен и больше не нуждается в его ресурсах. В следующем примере показан один случай, когда конструктор перемещения выбирается с помощью разрешения перегрузки. В конструкторе, который вызывает `get_Box()`, возвращаемое значение является `xValue` (значение срока действия). Она не назначается какой-либо переменной, поэтому она собирается выйти из области. Чтобы создать мотивацию для этого примера, пусть `Box` имеет большой вектор строк, который представляет его содержимое. Вместо копирования вектора и его строк конструктор Move "украсть" из поля "Box Expires", чтобы вектор теперь принадлежал новому объекту. Вызов `std::move` является необходимым, поскольку оба `vector` `string` класса и реализуют собственные конструкторы перемещения.

```

#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

class Box {
public:
    Box() { std::cout << "default" << std::endl; }
    Box(int width, int height, int length)
        : m_width(width), m_height(height), m_length(length)
    {
        std::cout << "int,int,int" << std::endl;
    }
    Box(Box& other)
        : m_width(other.m_width), m_height(other.m_height), m_length(other.m_length)
    {
        std::cout << "copy" << std::endl;
    }
    Box(Box&& other) : m_width(other.m_width), m_height(other.m_height), m_length(other.m_length)
    {
        m_contents = std::move(other.m_contents);
        std::cout << "move" << std::endl;
    }
    int Volume() { return m_width * m_height * m_length; }
    void Add_Item(string item) { m_contents.push_back(item); }
    void Print_Contents()
    {
        for (const auto& item : m_contents)
        {
            cout << item << " ";
        }
    }
private:
    int m_width{ 0 };
    int m_height{ 0 };
    int m_length{ 0 };
    vector<string> m_contents;
};

Box get_Box()
{
    Box b(5, 10, 18); // "int,int,int"
    b.Add_Item("Toupee");
    b.Add_Item("Megaphone");
    b.Add_Item("Suit");

    return b;
}

int main()
{
    Box b; // "default"
    Box b1(b); // "copy"
    Box b2(get_Box()); // "move"
    cout << "b2 contents: ";
    b2.Print_Contents(); // Prove that we have all the values

    char ch;
    cin >> ch; // keep window open
    return 0;
}

```

Если класс не определяет конструктор перемещения, компилятор создает неявный экземпляр, если не существует объявленного пользователем конструктора копии, оператора присваивания копирования, оператора присваивания перемещения или деструктора. Если явный или неявный конструктор

перемещения не определен, операции, в которых в противном случае использовался конструктор перемещения, используют вместо этого конструктор копий. Если класс объявляет конструктор перемещения или оператор присваивания перемещения, неявно объявленный конструктор копии определяется как удаленный.

Неявно объявленный конструктор перемещения определяется как удаленный, если какие-либо члены, являющиеся типами классов, не имеют деструктора или компилятор не может определить, какой конструктор использовать для операции перемещения.

Дополнительные сведения о написании нетривиальных конструкторов перемещения см. в разделе [конструкторы перемещения и операторы присваивания перемещения \(C++\)](#).

Явно заданные по умолчанию и удаленные конструкторы

Можно явно заключать конструкторы копирования, конструкторы *по умолчанию*, конструкторы перемещения, операторы присваивания копирования, операторы присваивания перемещения и деструкторы. Вы можете явно *Удалить* все специальные функции элементов.

```
class Box
{
public:
    Box2() = delete;
    Box2(const Box2& other) = default;
    Box2& operator=(const Box2& other) = default;
    Box2(Box2&& other) = default;
    Box2& operator=(Box2&& other) = default;
    //...
};
```

Дополнительные сведения см. в разделе [явно заданные по умолчанию и удаленные функции](#).

Конструкторы `constexpr`

Конструктор может быть объявлен как `constexpr`, если

- Он либо объявляется по умолчанию, либо, в противном случае, удовлетворяет всем условиям для функций `constexpr` в целом;
- класс не имеет виртуальных базовых классов;
- Каждый из параметров имеет [тип литерала](#);
- тело не является блоком функции `try`;
- все нестатические элементы данных и подобъекты базового класса инициализируются;
- Если класс имеет (a) объединение с вариантными членами или (b) имеет анонимные объединения, то инициализируется только один из членов Union.
- Каждый нестатический элемент данных типа класса, а все подобъекты базового класса имеют конструктор `constexpr`

Конструкторы списка инициализаторов

Если конструктор принимает в качестве параметра значение `std::<T> : initializer_list`, а все другие параметры имеют аргументы по умолчанию, этот конструктор будет выбран в разрешении перегрузки при создании экземпляра класса с помощью прямой инициализации. Можно использовать `initializer_list` для инициализации любого участника, который может принять его. Например, предположим, что класс `Box` (показан ранее) имеет `std::vector<string>` элемент `m_contents`. Вы можете предоставить конструктор следующим образом:

```
Box(initializer_list<string> list, int w = 0, int h = 0, int l = 0)
    : m_contents(list), m_width(w), m_height(h), m_length(l)
{}
```

Затем создайте объекты Box следующим образом:

```
Box b{ "apples", "oranges", "pears" }; // or ...
Box b2(initializer_list<string> { "bread", "cheese", "wine" }, 2, 4, 6);
```

Явные конструкторы

Если у класса имеется конструктор с одним параметром, или у всех параметров, кроме одного, имеются значения по умолчанию, тип параметра можно неявно преобразовать в тип класса. Например, если у класса `Box` имеется конструктор, подобный следующему:

```
Box(int size): m_width(size), m_length(size), m_height(size){}
```

то возможно инициализировать объект Box следующим образом:

```
Box b = 42;
```

Или передать целое значение функции, принимающей объект Box:

```
class ShippingOrder
{
public:
    ShippingOrder(Box b, double postage) : m_box(b), m_postage(postage) {}

private:
    Box m_box;
    double m_postage;
}
//elsewhere...
ShippingOrder so(42, 10.8);
```

В некоторых случаях подобные преобразования могут быть полезны, однако чаще всего они могут привести к незаметным, но серьезным ошибкам в вашем коде. В качестве общего правила следует использовать `explicit` ключевое слово в конструкторе (и определяемых пользователем операторах) для предотвращения такого рода неявного преобразования типов:

```
explicit Box(int size): m_width(size), m_length(size), m_height(size){}
```

Когда конструктор является явным, эта строка вызывает ошибку компилятора:

`ShippingOrder so(42, 10.8);`. Дополнительные сведения см. в разделе [преобразования определяемого пользователем типа](#).

Порядок создания

Конструктор выполняет свою работу в следующем порядке.

1. Вызывает конструкторы базовых классов и членов в порядке объявления.
2. Если класс является производным от виртуальных базовых классов, конструктор инициализирует

указатели виртуальных базовых классов объекта.

3. Если класс имеет или наследует виртуальные функции, конструктор инициализирует указатели виртуальных функций объекта. Указатели виртуальных функций указывают на таблицу виртуальных функций класса, чтобы обеспечить правильную привязку вызовов виртуальных функций к коду.
4. Выполняет весь код в теле функции.

В следующем примере показан порядок, в котором конструкторы базовых классов и членов вызываются в конструкторе для производного класса. Сначала вызывается конструктор базового класса, затем инициализируются члены базового класса в порядке их появления в объявлении класса. После этого вызывается конструктор производного класса.

```
#include <iostream>

using namespace std;

class Contained1 {
public:
    Contained1() { cout << "Contained1 ctor\n"; }
};

class Contained2 {
public:
    Contained2() { cout << "Contained2 ctor\n"; }
};

class Contained3 {
public:
    Contained3() { cout << "Contained3 ctor\n"; }
};

class BaseContainer {
public:
    BaseContainer() { cout << "BaseContainer ctor\n"; }
private:
    Contained1 c1;
    Contained2 c2;
};

class DerivedContainer : public BaseContainer {
public:
    DerivedContainer() : BaseContainer() { cout << "DerivedContainer ctor\n"; }
private:
    Contained3 c3;
};

int main() {
    DerivedContainer dc;
}
```

Выходные данные будут выглядеть следующим образом.

```
Contained1 ctor
Contained2 ctor
BaseContainer ctor
Contained3 ctor
DerivedContainer ctor
```

Конструктор производного класса всегда вызывает конструктор базового класса, чтобы перед выполнением любых дополнительных операций иметь в своем распоряжении полностью созданные

базовые классы. Конструкторы базовых классов вызываются в порядке наследования — например, если `ClassA` является производным от класса `ClassB`, производного от, то `ClassC ClassA` сначала вызывается конструктор, затем `ClassB` конструктор, а затем `ClassA` конструктор.

Если базовый класс не имеет конструктора по умолчанию, в конструкторе производного класса необходимо указать параметры конструктора базового класса.

```
class Box {  
public:  
    Box(int width, int length, int height){  
        m_width = width;  
        m_length = length;  
        m_height = height;  
    }  
  
private:  
    int m_width;  
    int m_length;  
    int m_height;  
};  
  
class StorageBox : public Box {  
public:  
    StorageBox(int width, int length, int height, const string label& ) : Box(width, length, height){  
        m_label = label;  
    }  
private:  
    string m_label;  
};  
  
int main(){  
  
    const string aLabel = "aLabel";  
    StorageBox sb(1, 2, 3, aLabel);  
}
```

Если конструктор создает исключение, то удаление выполняется в порядке обратном созданию.

1. Отменяется код в теле функции конструктора.
2. Объекты базовых классов и объекты-члены удаляются в порядке, обратном объявлению.
3. Если конструктор не является делегирующим, удаляются все полностью созданные объекты базовых классов и объекты-члены. Однако поскольку сам объект создан не полностью, деструктор не выполняется.

Производные конструкторы и расширенная агрегатная инициализация

если конструктор базового класса не является открытым, но доступен для производного класса, то в режиме `/std: c++ 17` в Visual Studio 2017 и более поздних версиях нельзя использовать пустые фигурные скобки для инициализации объекта производного типа.

В следующем примере показана соответствующая реакция на событие в C++ 14:

```

struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {}
};

struct Derived : Base {};

Derived d1; // OK. No aggregate init involved.
Derived d2 {}; // OK in C++14: Calls Derived::Derived()
               // which can call Base ctor.

```

В C++17 `Derived` теперь считается агрегатным типом. Это означает, что инициализация `Base` через закрытый конструктор по умолчанию происходит непосредственно как часть расширенного правила агрегатной инициализации. Закрытый конструктор `Base` ранее вызывался через конструктор `Derived`, и это работало успешно благодаря объявлению дружественных отношений.

в следующем примере показано поведение c++ 17 в Visual Studio 2017 и более поздних версиях в `/std:режим c++ 17`:

```

struct Derived;

struct Base {
    friend struct Derived;
private:
    Base() {}
};

struct Derived : Base {
    Derived() {} // add user-defined constructor
                 // to call with {} initialization
};

Derived d1; // OK. No aggregate init involved.

Derived d2 {}; // error C2248: 'Base::Base': cannot access
               // private member declared in class 'Base'

```

Конструкторы для классов с несколькими наследованиями

Если класс является производным от нескольких базовых классов, конструкторы базовых классов вызываются в том порядке, в котором они перечислены в объявлении производного класса.

```

#include <iostream>
using namespace std;

class BaseClass1 {
public:
    BaseClass1() { cout << "BaseClass1 ctor\n"; }
};

class BaseClass2 {
public:
    BaseClass2() { cout << "BaseClass2 ctor\n"; }
};

class BaseClass3 {
public:
    BaseClass3() { cout << "BaseClass3 ctor\n"; }
};

class DerivedClass : public BaseClass1,
                     public BaseClass2,
                     public BaseClass3
{
public:
    DerivedClass() { cout << "DerivedClass ctor\n"; }
};

int main() {
    DerivedClass dc;
}

```

Должны выводиться следующие выходные данные:

```

BaseClass1 ctor
BaseClass2 ctor
BaseClass3 ctor
DerivedClass ctor

```

Делегирование конструкторов

Делегирующий конструктор вызывает другой конструктор в том же классе для выполнения некоторой работы по инициализации. Это полезно, если у вас есть несколько конструкторов, которые должны выполнять одинаковую работу. Можно написать основную логику в одном конструкторе и вызвать ее из других. В следующем тривиальном примере Box (int) делегирует свою работу Box (int, int, int):

```

class Box {
public:
    // Default constructor
    Box() {}

    // Initialize a Box with equal dimensions (i.e. a cube)
    Box(int i) : Box(i, i, i); // delegating constructor
    {}

    // Initialize a Box with custom dimensions
    Box(int width, int length, int height)
        : m_width(width), m_length(length), m_height(height)
    {}
    //... rest of class as before
};

```

Объект, созданный конструкторами, полностью инициализируется сразу после выполнения любого конструктора. Дополнительные сведения см. в разделе [Делегирование конструкторов](#).

Наследование конструкторов (C++ 11)

Производный класс может наследовать конструкторы от прямого базового класса с помощью объявления `using` как показано в следующем примере:

```
#include <iostream>
using namespace std;

class Base
{
public:
    Base() { cout << "Base()" << endl; }
    Base(const Base& other) { cout << "Base(Base&)" << endl; }
    explicit Base(int i) : num(i) { cout << "Base(int)" << endl; }
    explicit Base(char c) : letter(c) { cout << "Base(char)" << endl; }

private:
    int num;
    char letter;
};

class Derived : Base
{
public:
    // Inherit all constructors from Base
    using Base::Base;

private:
    // Can't initialize newMember from Base constructors.
    int newMember{ 0 };
};

int main()
{
    cout << "Derived d1(5) calls: ";
    Derived d1(5);
    cout << "Derived d1('c') calls: ";
    Derived d2('c');
    cout << "Derived d3 = d2 calls: " ;
    Derived d3 = d2;
    cout << "Derived d4 calls: ";
    Derived d4;
}

/* Output:
Derived d1(5) calls: Base(int)
Derived d1('c') calls: Base(char)
Derived d3 = d2 calls: Base(Base&)
Derived d4 calls: Base()*/
```

Visual Studio 2017 и более поздних версий: `using` инструкция в режиме `/std: c++ 17`

предоставляет все конструкторы из базового класса, за исключением тех, которые имеют идентичную сигнатуру для конструкторов в производном классе. Обычно, если в производном классе не объявляются новые данные-члены или конструкторы, оптимальным решением будет использовать наследуемые конструкторы.

Шаблон класса может наследовать все конструкторы от аргумента типа, если этот тип определяет базовый класс:

```
template< typename T >
class Derived : T {
    using T::T; // declare the constructors from T
    // ...
};
```

Производный класс не может наследовать от нескольких базовых классов, если у этих базовых классов есть конструкторы с идентичными сигнатурами.

Конструкторы и составные классы

Классы, содержащие члены типа класса, называются *составными классами*. При создании члена типа класса составного класса конструктор вызывается перед собственным конструктором класса. Если у содержащегося класса нет конструктора по умолчанию, необходимо использовать список инициализации в конструкторе составного класса. В предыдущем примере `StorageBox` при присвоении типу переменной-члена `m_label` нового класса `Label` необходимо вызвать конструктор базового класса и инициализировать переменную `m_label` в конструкторе `StorageBox`:

```
class Label {
public:
    Label(const string& name, const string& address) { m_name = name; m_address = address; }
    string m_name;
    string m_address;
};

class StorageBox : public Box {
public:
    StorageBox(int width, int length, int height, Label label)
        : Box(width, length, height), m_label(label){}
private:
    Label m_label;
};

int main(){
// passing a named Label
    Label label1{ "some_name", "some_address" };
    StorageBox sb1(1, 2, 3, label1);

    // passing a temporary label
    StorageBox sb2(3, 4, 5, Label{ "another name", "another address" });

    // passing a temporary label as an initializer list
    StorageBox sb3(1, 2, 3, {"myname", "myaddress"});
}
```

Содержимое раздела

- [Конструкторы копий и операторы присваивания копий](#)
- [Конструкторы перемещения и операторы присваивания перемещением](#)
- [Делегирующие конструкторы](#)

См. также

[Классы и структуры](#)

Конструкторы копий и операторы присваивания копий (C++)

12.11.2021 • 2 minutes to read

NOTE

Начиная с C++ 11, в языке поддерживаются два вида присваивания: *копирование назначения* и *Перемещение*. В этой статье "присваивание" означает "присваивание копированием", если явно не указано другое. Сведения о назначении Move см. в разделе [конструкторы Move и операторы присваивания перемещения \(C++\)](#).

Как при операции назначения, так и при операции инициализации выполняется копирование объектов.

- **Назначение:** когда значение одного объекта присваивается другому объекту, первый объект копируется во второй объект. Поэтому

```
Point a, b;  
...  
a = b;
```

приводит к тому, что значение `b` копируется в значение `a`.

- **Инициализация:** инициализация происходит при объявлении нового объекта, когда аргументы передаются в функции по значению или когда значения возвращаются из функций по значению.

Можно определить семантику копии объектов типа класса. Рассмотрим для примера такой код:

```
TextFile a, b;  
a.Open( "FILE1.DAT" );  
b.Open( "FILE2.DAT" );  
b = a;
```

Предыдущий код может означать "копировать содержимое FILE1.DAT в FILE2.DAT" или "игнорировать FILE2.DAT и сделать `b` вторым дескриптором FILE1.DAT". Необходимо вложить соответствующую семантику копирования в каждый класс следующим образом.

- С помощью оператора присваивания `operator =` вместе со ссылкой на тип класса в качестве возвращаемого типа и параметра, передаваемого по `const` ссылке, например
`ClassName& operator=(const ClassName& x);`.
- С помощью конструктора копии.

Если конструктор копии не объявлен, компилятор создает конструктор копии для каждого члена. Если оператор присваивания копированием не объявлен, компилятор создает оператор присваивания копированием для каждого члена. Объявление конструктора копии не подавляет созданный компилятором оператор присваивания копий, и наоборот. При реализации любого из этих способов рекомендуется также реализовать другой способ, чтобы значение кода было четким.

Конструктор копий принимает аргумент типа `Class-Name &`, где `Class-Name` — это имя класса, для которого определен конструктор. Пример:

```
// spec1_copying_class_objects.cpp
class Window
{
public:
    Window( const Window& ); // Declare copy constructor.
    // ...
};

int main()
{
}
```

NOTE

По возможности сделайте тип аргумента `const` -имя класса для конструктора копирования &. Это поможет избежать случайного изменения копируемого объекта конструктором копии. Он также позволяет копировать из `const` объектов.

Конструкторы копии, создаваемые компилятором

Созданные компилятором конструкторы копий, такие как пользовательские конструкторы копий, имеют один аргумент типа "ссылка на имя-класса". Исключением является то, что все базовые классы и классы элементов имеют конструкторы копий, объявленные как принимающие один аргумент типа `const Class-Name&`. В этом случае аргументом конструктора копии, созданным компилятором, также является `const`.

Если тип аргумента для конструктора копирования отсутствует `const`, инициализация путем копирования `const` объекта приводит к ошибке. Обратная неверно: Если аргумент имеет значение `const`, инициализацию можно выполнить путем копирования объекта, который не является `const`.

Созданные компилятором операторы присваивания следуют тому же шаблону, что и для `const`. Они принимают один аргумент типа `Class-Name`, &. Если операторы присваивания во всех базовых классах и классов элементов не принимают аргументы типа `const Class-Name&`. В этом случае созданный классом оператор присваивания принимает `const` аргумент.

NOTE

Если виртуальные базовые классы инициализируются конструкторами копии, создаются компиляторами или определяются пользователем, они инициализируются только один раз, во время создания.

Последствия аналогичны тем, которые имеет конструктор копии. Если тип аргумента не равен `const`, присваивание из `const` объекта приводит к ошибке. Обратная неверно: Если `const` значение присвоено значению, которое не является `const`, то назначение выполняется.

Дополнительные сведения о перегруженных операторах присваивания см. в разделе [назначение](#).

Конструкторы move и операторы присваивания move (C++)

12.11.2021 • 5 minutes to read

В этом разделе описывается написание *конструктора перемещения* и оператора присваивания перемещения для класса C++. Конструктор перемещения позволяет перемещать ресурсы, принадлежащие объекту rvalue, в lvalue без копирования. Дополнительные сведения о семантике перемещения см. в разделе [декларатор ссылок rvalue: &&](#).

Этот раздел построен на основе приведенного ниже класса C++ `MemoryBlock`, который управляет буфером памяти.

```
// MemoryBlock.h
#pragma once
#include <iostream>
#include <algorithm>

class MemoryBlock
{
public:

    // Simple constructor that initializes the resource.
    explicit MemoryBlock(size_t length)
        : _length(length)
        , _data(new int[length])
    {
        std::cout << "In MemoryBlock(size_t). length = "
              << _length << "." << std::endl;
    }

    // Destructor.
    ~MemoryBlock()
    {
        std::cout << "In ~MemoryBlock(). length = "
              << _length << ".";

        if (_data != nullptr)
        {
            std::cout << " Deleting resource.";
            // Delete the resource.
            delete[] _data;
        }

        std::cout << std::endl;
    }

    // Copy constructor.
    MemoryBlock(const MemoryBlock& other)
        : _length(other._length)
        , _data(new int[other._length])
    {
        std::cout << "In MemoryBlock(const MemoryBlock&). length = "
              << other._length << ". Copying resource." << std::endl;

        std::copy(other._data, other._data + _length, _data);
    }

    // Copy assignment operator.
    MemoryBlock& operator=(const MemoryBlock& other)
    {
```

```

    {
        std::cout << "In operator=(const MemoryBlock&). length = "
            << other._length << ". Copying resource." << std::endl;

        if (this != &other)
        {
            // Free the existing resource.
            delete[] _data;

            _length = other._length;
            _data = new int[_length];
            std::copy(other._data, other._data + _length, _data);
        }
        return *this;
    }

    // Retrieves the length of the data resource.
    size_t Length() const
    {
        return _length;
    }

private:
    size_t _length; // The length of the resource.
    int* _data; // The resource.
};


```

В следующих процедурах описывается создание конструктора перемещения и оператора присваивания перемещения для этого примера класса C++.

Создание конструктора перемещения для класса C++

1. Определите пустой метод конструктора, принимающий в качестве параметра ссылку rvalue на тип класса, как показано в следующем примере:

```

MemoryBlock(MemoryBlock&& other)
: _data(nullptr)
, _length(0)
{
}

```

2. В конструкторе перемещения присвойте создаваемому объекту данные-члены класса из исходного объекта:

```

_data = other._data;
_length = other._length;

```

3. Присвойте данным-членам исходного объекта значения по умолчанию. Это не позволяет деструктору многократно освобождать ресурсы (например, память):

```

other._data = nullptr;
other._length = 0;

```

Создание оператора присваивания перемещения для класса C++

1. Определите пустой оператор присваивания, принимающий в качестве параметра ссылку rvalue на тип класса и возвращающий ссылку на тип класса, как показано в следующем примере:

```
MemoryBlock& operator=(MemoryBlock&& other)
{
}
```

2. В операторе присваивания перемещения добавьте условный оператор, который не выполняет никакой операции при попытке присвоить объект самому себе.

```
if (this != &other)
{
}
```

3. В условном операторе освободите все ресурсы (такие как память) из объекта, которому производится присваивание.

В следующем примере освобождается член `_data` из объекта, которому производится присваивание:

```
// Free the existing resource.
delete[] _data;
```

Выполните шаги 2 и 3 из первой процедуры, чтобы переместить данные-члены из исходного объекта в создаваемый объект:

```
// Copy the data pointer and its length from the
// source object.
_data = other._data;
_length = other._length;

// Release the data pointer from the source object so that
// the destructor does not free the memory multiple times.
other._data = nullptr;
other._length = 0;
```

4. Верните ссылку на текущий объект, как показано в следующем примере:

```
return *this;
```

Пример. Завершение конструктора перемещения и оператора присваивания

В следующем примере показаны полные конструктор перемещения и оператор назначения перемещения для класса `MemoryBlock`:

```

// Move constructor.
MemoryBlock(MemoryBlock&& other) noexcept
    : _data(nullptr)
    , _length(0)
{
    std::cout << "In MemoryBlock(MemoryBlock&&). length = "
        << other._length << ". Moving resource." << std::endl;

    // Copy the data pointer and its length from the
    // source object.
    _data = other._data;
    _length = other._length;

    // Release the data pointer from the source object so that
    // the destructor does not free the memory multiple times.
    other._data = nullptr;
    other._length = 0;
}

// Move assignment operator.
MemoryBlock& operator=(MemoryBlock&& other) noexcept
{
    std::cout << "In operator=(MemoryBlock&&). length = "
        << other._length << "." << std::endl;

    if (this != &other)
    {
        // Free the existing resource.
        delete[] _data;

        // Copy the data pointer and its length from the
        // source object.
        _data = other._data;
        _length = other._length;

        // Release the data pointer from the source object so that
        // the destructor does not free the memory multiple times.
        other._data = nullptr;
        other._length = 0;
    }
    return *this;
}

```

Пример использования семантики перемещения для повышения производительности

В следующем примере показано, как семантика перемещения может повысить производительность приложений. В примере добавляются два элемента в объект-вектор, а затем вставляется новый элемент между двумя существующими элементами. `vector` Класс использует семантику перемещения для эффективного выполнения операции вставки, перемещая элементы вектора вместо копирования.

```

// rvalue-references-move-semantics.cpp
// compile with: /EHsc
#include "MemoryBlock.h"
#include <vector>

using namespace std;

int main()
{
    // Create a vector object and add a few elements to it.
    vector<MemoryBlock> v;
    v.push_back(MemoryBlock(25));
    v.push_back(MemoryBlock(75));

    // Insert a new element into the second position of the vector.
    v.insert(v.begin() + 1, MemoryBlock(50));
}

```

В этом примере выводятся следующие данные:

```

In MemoryBlock(size_t). length = 25.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In ~MemoryBlock(). length = 0.
In MemoryBlock(size_t). length = 75.
In MemoryBlock(MemoryBlock&&). length = 75. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In MemoryBlock(size_t). length = 50.
In MemoryBlock(MemoryBlock&&). length = 50. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 25. Moving resource.
In MemoryBlock(MemoryBlock&&). length = 75. Moving resource.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 0.
In ~MemoryBlock(). length = 25. Deleting resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 75. Deleting resource.

```

до Visual Studio 2010 в этом примере создаются следующие выходные данные:

```

In MemoryBlock(size_t). length = 25.
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.
In ~MemoryBlock(). length = 25. Deleting resource.
In MemoryBlock(size_t). length = 75.
In MemoryBlock(const MemoryBlock&). length = 25. Copying resource.
In ~MemoryBlock(). length = 25. Deleting resource.
In MemoryBlock(const MemoryBlock&). length = 75. Copying resource.
In ~MemoryBlock(). length = 75. Deleting resource.
In MemoryBlock(size_t). length = 50.
In MemoryBlock(const MemoryBlock&). length = 50. Copying resource.
In MemoryBlock(const MemoryBlock&). length = 50. Copying resource.
In operator=(const MemoryBlock&). length = 75. Copying resource.
In operator=(const MemoryBlock&). length = 50. Copying resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 25. Deleting resource.
In ~MemoryBlock(). length = 50. Deleting resource.
In ~MemoryBlock(). length = 75. Deleting resource.

```

Версия этого примера, в которой используется семантика перемещения, более эффективна, чем версия, в которой эта семантика не используется, поскольку в ней выполняется меньше операций копирования,

выделения памяти и освобождения памяти.

Отказоустойчивость

Во избежание утечки ресурсов (таких как память, дескрипторы файлов и сокеты) обязательно освобождайте их в операторе присваивания перемещения.

Чтобы предотвратить невосстановимое уничтожение ресурсов, в операторе присваивания перемещения необходимо правильно обрабатывать присваивания самому себе.

Если для класса определены как конструктор перемещения, так и оператор присваивания перемещения, можно исключить избыточный код, написав конструктор перемещения так, чтобы он вызывал оператор присваивания перемещения. В следующем примере показана измененная версия конструктора перемещения,зывающая оператор присваивания перемещения:

```
// Move constructor.  
MemoryBlock(MemoryBlock&& other) noexcept  
    : _data(nullptr)  
    , _length(0)  
{  
    *this = std::move(other);  
}
```

Функция `std::Move` преобразует lvalue `other` в rvalue.

См. также

[Декларатор ссылки rvalue: &&](#)
[std::Move](#)

Делегирующие конструкторы

12.11.2021 • 2 minutes to read

Многие классы имеют несколько конструкторов, которые выполняют аналогичные действия, например, проверяют параметры:

```
class class_c {
public:
    int max;
    int min;
    int middle;

    class_c() {}
    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) {
        max = my_max > 0 ? my_max : 10;
        min = my_min > 0 && my_min < max ? my_min : 1;
        middle = my_middle < max && my_middle > min ? my_middle : 5;
    }
};
```

Можно сократить повторяющийся код, добавив функцию, которая выполняет всю проверку, но код для будет `class_c` проще понять и поддерживать, если один конструктор может делегировать некоторую работу другому. Чтобы добавить делегирование конструкторов, используйте

`constructor (. . .) : constructor (. . .)` СИНТАКСИС:

```
class class_c {
public:
    int max;
    int min;
    int middle;

    class_c(int my_max) {
        max = my_max > 0 ? my_max : 10;
    }
    class_c(int my_max, int my_min) : class_c(my_max) {
        min = my_min > 0 && my_min < max ? my_min : 1;
    }
    class_c(int my_max, int my_min, int my_middle) : class_c (my_max, my_min){
        middle = my_middle < max && my_middle > min ? my_middle : 5;
    }
};
int main() {
    class_c c1{ 1, 3, 2 };
}
```

При пошаговом выполнении предыдущего примера обратите внимание, что конструктор

`class_c(int, int, int)` сначала вызывает конструктор `class_c(int, int)`, который в свою очередь вызывает `class_c(int)`. Каждый из конструкторов выполняет только работу, которая не выполняется другими конструкторами.

Первый конструктор, который вызывается, инициализирует объект, чтобы все его члены были инициализированы в этой точке. Невозможно выполнить инициализацию члена в конструкторе, который делегирует другому конструктору, как показано ниже:

```
class class_a {  
public:  
    class_a() {}  
    // member initialization here, no delegate  
    class_a(string str) : m_string{ str } {}  
  
    //can't do member initialization here  
    // error C3511: a call to a delegating constructor shall be the only member-initializer  
    class_a(string str, double dbl) : class_a(str) , m_double{ dbl } {}  
  
    // only member assignment  
    class_a(string str, double dbl) : class_a(str) { m_double = dbl; }  
    double m_double{ 1.0 };  
    string m_string;  
};
```

В следующем примере показано использование нестатических инициализаторов элементов данных.

Обратите внимание, что если конструктор также инициализирует данный элемент данных, инициализатор члена переопределяется:

```
class class_a {  
public:  
    class_a() {}  
    class_a(string str) : m_string{ str } {}  
    class_a(string str, double dbl) : class_a(str) { m_double = dbl; }  
    double m_double{ 1.0 };  
    string m_string{ m_double < 10.0 ? "alpha" : "beta" };  
};  
  
int main() {  
    class_a a{ "hello", 2.0 }; //expect a.m_double == 2.0, a.m_string == "hello"  
    int y = 4;  
}
```

Синтаксис делегирования конструктора не предотвращает случайное создание рекурсии конструктора — Constructor1 вызывает Constructor2, который вызывает Constructor1, и ошибки не создаются, пока не произойдет переполнение стека. Вы обязаны избегать циклов.

```
class class_f{  
public:  
    int max;  
    int min;  
  
    // don't do this  
    class_f() : class_f(6, 3){ }  
    class_f(int my_max, int my_min) : class_f() { }  
};
```

Деструкторы (C++)

12.11.2021 • 6 minutes to read

Деструктор — это функция-член, которая вызывается автоматически, когда объект выходит из области действия или явно уничтожается вызовом метода `delete`. Деструктор имеет то же имя, что и класс, перед которым стоит тильда (`~`). Например, деструктор для класса `String` объявляется следующим образом: `~String()`.

Если деструктор не определен, компилятор будет предоставлять его по умолчанию. Для многих классов это достаточно. Необходимо определить пользовательский деструктор, если класс хранит дескрипторы для системных ресурсов, которые необходимо освободить, или указатели, владеющие памятью, на которую они указывают.

Рассмотрим следующее объявление класса `String`:

```
// spec1_destructors.cpp
#include <string>

class String {
public:
    String( char *ch ); // Declare constructor
    ~String();          // and destructor.
private:
    char    *_text;
    size_t  sizeOfText;
};

// Define the constructor.
String::String( char *ch ) {
    sizeOfText = strlen( ch ) + 1;

    // Dynamically allocate the correct amount of memory.
    _text = new char[ sizeOfText ];

    // If the allocation succeeds, copy the initialization string.
    if( _text )
        strcpy_s( _text, sizeOfText, ch );
}

// Define the destructor.
String::~String() {
    // Deallocate the memory that was previously reserved
    // for this string.
    delete[] _text;
}

int main() {
    String str("The piper in the glen...");
}
```

В предыдущем примере деструктор `String::~String` использует `delete` оператор для освобождения пространства, динамически выделяемого для хранения текста.

Объявление деструкторов

Деструкторы — это функции с тем же именем, что и класс, но с добавленным в начало знаком тильды (`~`).

При объявлении деструкторов действуют несколько правил. Деструкторы:

- Не могут иметь аргументов.
- Не возвращают значение (или `void`).
- Не может быть объявлен как `const`, `volatile` ИЛИ `static`. Однако они МОГУТ вызываться для уничтожения объектов, объявленных как `const`, `volatile` ИЛИ `static`.
- Может быть объявлен как `virtual`. Используя виртуальные деструкторы, можно уничтожать объекты, не зная их тип — правильный деструктор для объекта вызывается с помощью механизма виртуальных функций. Обратите внимание, что для абстрактных классов деструкторы также могут объявляться как чисто виртуальные функции.

Использование деструкторов

Деструкторы вызываются, когда происходит одно из следующих событий:

- Локальный (автоматический) объект с областью видимости блока выходит за пределы области видимости.
- Объект, выделенный с помощью `new` оператора, явным образом освобождается с помощью `delete`.
- Время существования временного объекта заканчивается.
- Программа заканчивается, глобальные или статические объекты продолжают существовать.
- Деструктор явно вызывается с использованием полного имени функции деструктора.

Деструкторы могут свободно вызывать функции-члена класса и осуществлять доступ к данным членов класса.

Существуют два ограничения на использование деструкторов.

- Вы не можете получить его адрес.
- Производные классы не наследуют деструктор своего базового класса.

Порядок уничтожения

Когда объект выходит за пределы области или удаляется, последовательность событий при его полном уничтожении выглядит следующим образом:

1. Вызывается деструктор класса, и выполняется тело функции деструктора.
2. Деструкторы для объектов нестатических членов вызываются в порядке, обратном порядку их появления в объявлении класса. Необязательный список инициализации элементов, используемый при создании этих элементов, не влияет на порядок создания или уничтожения.
3. Деструкторы для невиртуальных базовых классов вызываются в обратную последовательность объявления.
4. Деструкторы для виртуальных базовых классов вызываются в порядке, обратном порядку их объявления.

```

// order_of_destruction.cpp
#include <cstdio>

struct A1      { virtual ~A1() { printf("A1 dtor\n"); } };
struct A2 : A1 { virtual ~A2() { printf("A2 dtor\n"); } };
struct A3 : A2 { virtual ~A3() { printf("A3 dtor\n"); } };

struct B1      { ~B1() { printf("B1 dtor\n"); } };
struct B2 : B1 { ~B2() { printf("B2 dtor\n"); } };
struct B3 : B2 { ~B3() { printf("B3 dtor\n"); } };

int main() {
    A1 * a = new A3;
    delete a;
    printf("\n");

    B1 * b = new B3;
    delete b;
    printf("\n");

    B3 * b2 = new B3;
    delete b2;
}

Output: A3 dtor
A2 dtor
A1 dtor

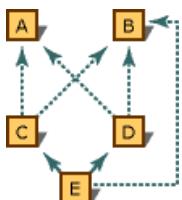
B1 dtor

B3 dtor
B2 dtor
B1 dtor

```

Виртуальные базовые классы

Деструкторы для виртуальных базовых классов вызываются в порядке, обратном их указанию в направленном ациклическом графе (в глубину, слева направо, обход в обратном порядке). На следующем рисунке представлен график наследования.



Граф наследования, показывающий виртуальные базовые классы

Ниже перечислены заголовки классов, представленных на рисунке.

```

class A
class B
class C : virtual public A, virtual public B
class D : virtual public A, virtual public B
class E : public C, public D, virtual public B

```

Чтобы определить порядок удаления виртуальных базовых классов объекта типа `E`, компилятор выполняет сборку списка, применяя следующий алгоритм.

- Просмотрите левую часть графа, начиная с самой глубокой точки графа (в данном случае `E`).
- Просматривайте граф справа налево, пока не будут пройдены все узлы. Запомните имя текущего узла.

3. Пересмотрите предыдущий узел (вниз и вправо), чтобы определить, является ли рассматриваемый узел виртуальным базовым классом.
4. Если рассматриваемый узел является виртуальным базовым классом, просмотрите список, чтобы проверить, был ли он введен ранее. Если он не является виртуальным базовым классом, игнорируйте его.
5. Если рассматриваемого узла еще нет в списке, добавьте его вниз списка.
6. Просмотрите граф вверх и вдоль следующего пути вправо.
7. Перейдите к шагу 2.
8. Если путь последний путь вверх исчерпан, запомните имя текущего узла.
9. Перейдите к шагу 3.
10. Выполняйте этот процесс, пока нижний узел снова не станет текущим узлом.

Таким образом, для класса `E` порядок удаления будет следующим.

1. Невиртуальный базовый класс `E`.
2. Невиртуальный базовый класс `D`.
3. Невиртуальный базовый класс `C`.
4. Виртуальный базовый класс `B`.
5. Виртуальный базовый класс `A`.

В ходе этого процесса создается упорядоченный список уникальных записей. Имя класса никогда не отображается дважды. После создания список просматривается в обратном порядке, и вызывается деструктор для каждого класса в списке от последнего к первому.

Порядок построения или удаления очень важен, когда конструкторы и деструкторы в одном классе полагаются на другой компонент, который создается первым или сохраняется дольше, например если деструктор `A` (на рисунке выше) полагается на то, что `B` будет по-прежнему присутствовать после выполнения кода, или наоборот.

Такие взаимозависимости между классами в графе наследования опасны, поскольку классы, наследуемые впоследствии, могут изменить крайний левый путь, тем самым изменив порядок построения и удаления.

Не являющиеся виртуальными базовыми классами

Деструкторы для невиртуальных базовых классов вызываются в порядке, в котором объявляются имена базовых классов. Рассмотрим следующее объявление класса.

```
class MultInherit : public Base1, public Base2
...
```

В предыдущем примере деструктор `Base2` вызывается перед деструктором `Base1`.

Явные вызовы деструктора

Редко возникает необходимость в явном вызове деструктора. Однако может быть полезно выполнить удаление объектов, размещенных по абсолютным адресам. Обычно эти объекты выделяются с помощью определяемого пользователем `new` оператора, принимающего аргумент размещения. `delete` Оператор не может освободить эту память, так как она не выделена из бесплатного хранилища (Дополнительные сведения см. в разделе [операторы new и DELETE](#)). Вызов деструктора, однако, может выполнить

соответствующую очистку. Для явного вызова деструктора для объекта (`s`) класса `String` воспользуйтесь одним из следующих операторов.

```
s.String::~String();      // non-virtual call  
ps->String::~String();  // non-virtual call  
  
s.~String();            // Virtual call  
ps->~String();         // Virtual call
```

Нотация для явных вызовов деструкторов, показанная в предыдущем примере, может использоваться независимо от того, определяет ли тип деструктор. Это позволяет выполнять такие явные вызовы, не зная, определен ли деструктор для типа. Явный вызов деструктора, если ни один из них не определен, не имеет никакого эффекта.

Отказоустойчивость

Классу требуется деструктор, если он получает ресурс, и для безопасного управления ресурсом, вероятно, потребуется реализовать конструктор копии и назначение копирования.

Если эти специальные функции не определены пользователем, они неявно определяются компилятором. Неявно созданные конструкторы и операторы присваивания выполняют поверхностную почлененную копию, которая почти наверняка неверно, если объект управляет ресурсом.

В следующем примере неявно созданный конструктор копии сделает указатели `str1.text` и `str2.text` ссылался на одну и ту же память, и, когда мы вернемся из `copy_strings()`, эта память будет удалена дважды, что является неопределенным поведением:

```
void copy_strings()  
{  
    String str1("I have a sense of impending disaster...");  
    String str2 = str1; // str1.text and str2.text now refer to the same object  
} // delete[] _text; deallocates the same memory twice  
// undefined behavior
```

Явное определение деструктора, конструктора копирования или оператора присваивания копирования предотвращает неявное определение конструктора перемещения и оператора присваивания перемещения. В этом случае не удастся предоставить операции перемещения, если копирование занимает много ресурсов, но пропущенная возможность оптимизации.

См. также

[Конструкторы копий и операторы присваивания копирования](#)

[Конструкторы перемещения и операторы присваивания перемещения](#)

Общие сведения о функциях-членах

12.11.2021 • 2 minutes to read

Функции-члены являются либо статическими, либо нестатическими. Поведение статических функций-членов отличается от других функций-членов, поскольку статические функции-члены не имеют неявных `this` аргументов. Нестатические функции-члены имеют `this` указатель. Функции-члены, статические или нестатические, можно определить в объявлении класса или вне его.

Если функция-член определяется в объявлении класса, она обрабатывается как встраиваемая функция, и нет необходимости уточнять имя функции именем класса. Хотя функции, определенные внутри объявлений класса, уже обрабатываются как встроенные функции, можно использовать `inline` ключевое слово для документирования кода.

Ниже приводится пример объявления функции в объявлении класса.

```
// overview_of_member_functions1.cpp
class Account
{
public:
    // Declare the member function Deposit within the declaration
    // of class Account.
    double Deposit( double HowMuch )
    {
        balance += HowMuch;
        return balance;
    }
private:
    double balance;
};

int main()
{
}
```

Если определение функции-члена находится вне объявления класса, оно рассматривается как встроенная функция только в том случае, если она явно объявлена как `inline`. Кроме того, имя функции в определении должно уточняться именем класса с помощью оператора разрешения области действия (`::`).

Следующий пример идентичен предыдущему объявлению класса `Account` за исключением того, что функция `Deposit` определяется вне объявления класса.

```
// overview_of_member_functions2.cpp
class Account
{
public:
    // Declare the member function Deposit but do not define it.
    double Deposit( double HowMuch );
private:
    double balance;
};

inline double Account::Deposit( double HowMuch )
{
    balance += HowMuch;
    return balance;
}

int main()
{
}
```

NOTE

Хотя функции-члены можно определить внутри объявления класса или вне его, функции-члены нельзя добавить в класс после определения класса.

Классы, содержащие функции-члены, могут иметь несколько объявлений, но сами функции-члены должны иметь только одно определение в программе. При наличии нескольких определений выдается сообщение об ошибке во время компоновки. Если класс содержит определения встраиваемых функций, определения функций должны быть идентичными для соблюдения данного правила одного определения.

Спецификатор `virtual`

12.11.2021 • 2 minutes to read

Ключевое слово [Virtual](#) может применяться только к нестатическим функциям члена класса. Это означает, что привязка вызовов функции откладывается до времени выполнения. Дополнительные сведения см. в разделе [виртуальные функции](#).

Спецификатор override

12.11.2021 • 2 minutes to read

Ключевое слово **override** можно использовать для обозначения функций-членов, которые переопределяют виртуальную функцию в базовом классе.

Синтаксис

```
function-declaration override;
```

Remarks

Переопределение является контекстно-зависимым и имеет специальное значение только в том случае, если оно используется после объявления функции-члена. В противном случае это не зарезервированное ключевое слово.

Пример

Для предотвращения случайного наследования в коде используйте функцию **переопределения**. В следующем примере показано, где, без использования **переопределения**, поведение функции члена производного класса может не быть предполагалось. Компилятор не выдает ошибки при использовании этого кода.

```
class BaseClass
{
    virtual void funcA();
    virtual void funcB() const;
    virtual void funcC(int = 0);
    void funcD();
};

class DerivedClass: public BaseClass
{
    virtual void funcA(); // ok, works as intended

    virtual void funcB(); // DerivedClass::funcB() is non-const, so it does not
                         // override BaseClass::funcB() const and it is a new member function

    virtual void funcC(double = 0.0); // DerivedClass::funcC(double) has a different
                                    // parameter type than BaseClass::funcC(int), so
                                    // DerivedClass::funcC(double) is a new member function
};
```

При использовании **переопределения** компилятор создает ошибки вместо автоматического создания новых функций элементов.

```
class BaseClass
{
    virtual void funcA();
    virtual void funcB() const;
    virtual void funcC(int = 0);
    void funcD();
};

class DerivedClass: public BaseClass
{
    virtual void funcA() override; // ok

    virtual void funcB() override; // compiler error: DerivedClass::funcB() does not
                                  // override BaseClass::funcB() const

    virtual void funcC( double = 0.0 ) override; // compiler error:
                                                // DerivedClass::funcC(double) does not
                                                // override BaseClass::funcC(int)

    void funcD() override; // compiler error: DerivedClass::funcD() does not
                          // override the non-virtual BaseClass::funcD()
};
```

Чтобы указать, что функции не могут быть переопределены и что классы не могут быть унаследованы, используйте ключевое слово **final**.

См. также раздел

[Окончательный описатель](#)

[Ключевые слова](#)

Спецификатор final

12.11.2021 • 2 minutes to read

Ключевое слово **final** можно использовать для обозначения виртуальных функций, которые не могут быть переопределены в производном классе. Можно также использовать это ключевое слово для назначения классов, которые невозможно наследовать.

Синтаксис

```
function-declaration final;  
class class-name final base-classes
```

Remarks

final является контекстно-зависимым и имеет специальное значение только в том случае, если оно используется после объявления функции или имени класса; в противном случае это не зарезервированное ключевое слово.

Если аргумент **final** используется в объявлениях классов, `base-classes` является необязательной частью объявления.

Пример

В следующем примере ключевое слово **final** используется для указания того, что виртуальную функцию нельзя переопределить.

```
class BaseClass  
{  
    virtual void func() final;  
};  
  
class DerivedClass: public BaseClass  
{  
    virtual void func(); // compiler error: attempting to  
                        // override a final function  
};
```

Сведения о том, как указать, что функции элементов можно переопределить, см. в разделе [спецификатор переопределения](#).

В следующем примере ключевое слово **final** используется для указания того, что класс не может быть унаследован.

```
class BaseClass final  
{  
};  
  
class DerivedClass: public BaseClass // compiler error: BaseClass is  
                                    // marked as non-inheritable  
{  
};
```

См. также раздел

[Ключевые слова](#)

[Спецификатор переопределения](#)

Наследование (C++)

12.11.2021 • 2 minutes to read

В этом разделе рассматривается использование производных классов для создания расширяемых программ.

Обзор

Новые классы могут быть производными от существующих классов с помощью механизма наследования (см. сведения, начиная с [одиночного наследования](#)). Классы, используемые для наследования, называются "базовыми классами" определенного производного класса. Производный класс объявляется с помощью следующего синтаксиса:

```
class Derived : [virtual] [access-specifier] Base
{
    // member list
};

class Derived : [virtual] [access-specifier] Base1,
    [virtual] [access-specifier] Base2, . . .
{
    // member list
};
```

После тега (имени) класса следует двоеточие и список базовых спецификаций. Названные таким образом базовые классы, вероятно, были объявлены ранее. Базовые спецификации могут содержать описатель доступа, который является одним из ключевых слов `public` `protected` ИЛИ `private`. Эти описатели доступа отображаются перед именем базового класса и применяются только к базовому классу. Эти описатели контролируют разрешение производного класса на использование членов базового класса. Сведения о доступе к членам базового класса см. в разделе [Управление доступом](#) к членам. Если описатель доступа не указан, будет учитываться доступ к этой базе `private`. Базовые спецификации могут содержать ключевое слово `virtual` для указания виртуального наследования. Это ключевое слово может отображаться до или после описателя доступа, если таковые имеются. Если используется виртуальное наследование, базовый класс называется виртуальным базовым классом.

Можно определить несколько базовых классов, разделив их запятыми. Если указан один базовый класс, то модель наследования является [одиночным наследованием](#). Если указано более одного базового класса, то модель наследования называется [множественным наследованием](#).

В этой статье содержатся следующие разделы:

- [Одиночное наследование](#)
- [Несколько базовых классов](#)
- [Виртуальные функции](#)
- [Явное переопределение](#)
- [Абстрактные классы](#)
- [Общие сведения о правилах области](#)

В этом разделе описаны ключевые слова `_super` и `_interface`.

См. также

[Справочник по языку C++](#)

Виртуальные функции

12.11.2021 • 3 minutes to read

Виртуальная функция — это функция-член, которую предполагается переопределить в производных классах. При ссылке на объект производного класса с помощью указателя или ссылки на базовый класс можно вызвать виртуальную функцию для этого объекта и выполнить версию функции производного класса.

Виртуальные функции обеспечивают вызов соответствующей функции для объекта независимо от выражения, используемого для вызова функции.

Предположим, что базовый класс содержит функцию, объявленную как [Виртуальная](#), а производный класс определяет ту же функцию. Функция производного класса вызывается для объектов производного класса, даже если она вызывается с помощью указателя или ссылки на базовый класс. В следующем примере показан базовый класс, который предоставляет реализацию функции `PrintBalance` и два производных класса.

```
// deriv_VirtualFunctions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Account {
public:
    Account( double d ) { _balance = d; }
    virtual ~Account() {}
    virtual double GetBalance() { return _balance; }
    virtual void PrintBalance() { cerr << "Error. Balance not available for base type." << endl; }
private:
    double _balance;
};

class CheckingAccount : public Account {
public:
    CheckingAccount(double d) : Account(d) {}
    void PrintBalance() { cout << "Checking account balance: " << GetBalance() << endl; }
};

class SavingsAccount : public Account {
public:
    SavingsAccount(double d) : Account(d) {}
    void PrintBalance() { cout << "Savings account balance: " << GetBalance(); }
};

int main() {
    // Create objects of type CheckingAccount and SavingsAccount.
    CheckingAccount checking( 100.00 );
    SavingsAccount savings( 1000.00 );

    // Call PrintBalance using a pointer to Account.
    Account *pAccount = &checking;
    pAccount->PrintBalance();

    // Call PrintBalance using a pointer to Account.
    pAccount = &savings;
    pAccount->PrintBalance();
}
```

В предыдущем примере вызовы `PrintBalance` идентичны за исключением тех, на которые указывает объект `pAccount`. Поскольку функция `PrintBalance` виртуальная, вызывается версия функции, определенная для каждого объекта. Функция `PrintBalance` в производных классах `CheckingAccount` и `SavingsAccount` переопределяет функцию в базовом классе `Account`.

Если объявлен класс, который не предоставляет переопределяющую реализацию функции `PrintBalance`, используется реализация по умолчанию из базового класса `Account`.

Функции в производных классах переопределяют виртуальные функции в базовых классах, только если их тип совпадает. Функция в производном классе не может отличаться от виртуальной функции в базовом классе только возвращаемым типом; список аргументов также должен отличаться.

При вызове функции с помощью указателей или ссылок применяются следующие правила.

- Вызов виртуальной функции разрешается в соответствии с базовым типом объекта, для которого она вызывается.
- Вызов невиртуальной функции разрешается в соответствии с типом указателя или ссылки.

В следующем примере показано поведение виртуальной и невиртуальной функций при вызове с помощью указателей.

```

// deriv_VirtualFunctions2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Base {
public:
    virtual void NameOf(); // Virtual function.
    void InvokingClass(); // Nonvirtual function.
};

// Implement the two functions.
void Base::NameOf() {
    cout << "Base::NameOf\n";
}

void Base::InvokingClass() {
    cout << "Invoked by Base\n";
}

class Derived : public Base {
public:
    void NameOf(); // Virtual function.
    void InvokingClass(); // Nonvirtual function.
};

// Implement the two functions.
void Derived::NameOf() {
    cout << "Derived::NameOf\n";
}

void Derived::InvokingClass() {
    cout << "Invoked by Derived\n";
}

int main() {
    // Declare an object of type Derived.
    Derived aDerived;

    // Declare two pointers, one of type Derived * and the other
    // of type Base *, and initialize them to point to aDerived.
    Derived *pDerived = &aDerived;
    Base   *pBase   = &aDerived;

    // Call the functions.
    pBase->NameOf();           // Call virtual function.
    pBase->InvokingClass();    // Call nonvirtual function.
    pDerived->NameOf();         // Call virtual function.
    pDerived->InvokingClass(); // Call nonvirtual function.
}

```

```

Derived::NameOf
Invoked by Base
Derived::NameOf
Invoked by Derived

```

Обратите внимание, что независимо от того, вызывается ли функция `NameOf` с помощью указателя на `Base` или указателя на `Derived`, вызывается функция для `Derived`. Вызывается функция для `Derived`, поскольку `NameOf` является виртуальной функцией, и `pBase` и `pDerived` указывают на объект типа `Derived`.

Поскольку виртуальные функции вызываются только для объектов типов классов, глобальные или статические функции нельзя объявлять как `virtual`.

`virtual` Ключевое слово можно использовать при объявлении переопределяемых функций в производном классе, но это не обязательно; переопределения виртуальных функций всегда являются виртуальными.

Виртуальные функции в базовом классе должны быть определены, если они не объявлены с помощью *чистого описателя*. (Дополнительные сведения о чистых виртуальных функциях см. в разделе [абстрактные классы](#).)

Механизм вызова виртуальных функций можно подавить, явно указав имя функции с помощью оператора разрешения области действия (`::`). Рассмотрим предыдущий пример с классом `Account`. Для вызова `PrintBalance` в базовом классе используйте следующий код.

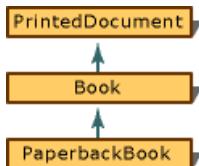
```
CheckingAccount *pChecking = new CheckingAccount( 100.00 );  
  
pChecking->Account::PrintBalance(); // Explicit qualification.  
  
Account *pAccount = pChecking; // Call Account::PrintBalance  
  
pAccount->Account::PrintBalance(); // Explicit qualification.
```

Оба вызова `PrintBalance` в предыдущем примере подавляют механизм вызова виртуальных функций.

Одиночное наследование

12.11.2021 • 3 minutes to read

При единичном наследовании, самой распространенной форме наследования, каждый класс имеет только один базовый класс. Рассмотрим пример взаимоотношений на следующем рисунке.



Обратите внимание на переход от общего к конкретному на этом рисунке. В структуре большинства иерархий классов заметна еще одна общая особенность: каждый производный класс является разновидностью базового класса. На этом рисунке видно, что класс Book является разновидностью класса PrintedDocument, а PaperbackBook — разновидностью класса book.

На этом рисунке необходимо обратить внимание еще на одну особенность: Book является не только производным (от класса PrintedDocument), но и базовым классом (PaperbackBook является производным от Book). В следующем примере показана структура объявления такой иерархии классов:

```
// deriv_SingleInheritance.cpp
// compile with: /LD
class PrintedDocument {};

// Book is derived from PrintedDocument.
class Book : public PrintedDocument {};

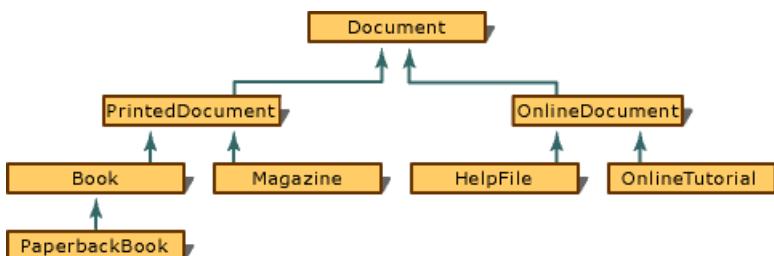
// PaperbackBook is derived from Book.
class PaperbackBook : public Book {};
```

Для PrintedDocument класс Book считается прямым базовым классом, а для PaperbackBook — косвенным. Различие заключается в том, что первый из них выводится в списке базовых классов в объявлении класса, а последний — нет.

Базовый класс, производными от которого являются другие классы, объявляется раньше производных классов. Для базового класса недостаточно предоставить объявление с опережающей ссылкой; объявление должно быть полным.

В предыдущем примере используется спецификатор доступа `public`. Значение открытого, защищенного и закрытого наследования описывается в разделе [Управление доступом к членам](#).

Один класс может быть базовым для нескольких более специализированных классов, как показано в следующем примере.



На приведенной выше схеме ориентированного ациклического графа (DAG) некоторые классы являются базовыми для нескольких производных классов. Но обратное не выполняется: для каждого производного класса имеется только один прямой базовый класс. На этом рисунке представлена структура с единичным наследованием.

NOTE

Ориентированные ациклические графы могут описывать структуры не только с единственным, но и с множественным наследованием.

При наследовании производный класс содержит члены базового класса, а также те члены, которые вы в него добавили. В результате производный класс может обращаться к членам базового класса (если в производном классе такие члены были переопределены). Если члены прямых или косвенных базовых классов были переопределены в производном классе, то обращаться к ним можно при помощи оператора разрешения области видимости (`::`). Рассмотрим следующий пример.

```
// deriv_SingleInheritance2.cpp
// compile with: /EHsc /c
#include <iostream>
using namespace std;
class Document {
public:
    char *Name; // Document name.
    void PrintNameOf(); // Print name.
};

// Implementation of PrintNameOf function from class Document.
void Document::PrintNameOf() {
    cout << Name << endl;
}

class Book : public Document {
public:
    Book( char *name, long pagecount );
private:
    long PageCount;
};

// Constructor from class Book.
Book::Book( char *name, long pagecount ) {
    Name = new char[ strlen( name ) + 1 ];
    strcpy_s( Name, strlen(Name), name );
    PageCount = pagecount;
}
```

Обратите внимание, что конструктор класса `Book` (`Book::Book`) может обращаться к элементу данных `Name`. В программе объект типа `Book` можно создать и использовать следующим образом:

```
// Create a new object of type Book. This invokes the
// constructor Book::Book.
Book LibraryBook( "Programming Windows, 2nd Ed", 944 );

...
// Use PrintNameOf function inherited from class Document.
LibraryBook.PrintNameOf();
```

Как показано в предыдущем примере, член класса и наследуемые данные и функции используются одинаково. Если реализации класса `Book` требует повторно реализовать функцию `PrintNameOf`, то та

функция, которая принадлежит классу `Document`, может вызываться только при помощи оператора разрешения области видимости (`::`):

```
// deriv_SingleInheritance3.cpp
// compile with: /EHsc /LD
#include <iostream>
using namespace std;

class Document {
public:
    char *Name;           // Document name.
    void PrintNameOf() {} // Print name.
};

class Book : public Document {
    Book( char *name, long pagecount );
    void PrintNameOf();
    long PageCount;
};

void Book::PrintNameOf() {
    cout << "Name of book: ";
    Document::PrintNameOf();
}
```

Указатели и ссылки на производные классы могут быть неявно преобразованы в указатели и ссылки на их базовые классы (если имеется доступный и однозначный базовый класс). Эта концепция показана в следующем фрагменте кода на примере указателей (тот же принцип действует и для ссылок):

```
// deriv_SingleInheritance4.cpp
// compile with: /W3
struct Document {
    char *Name;
    void PrintNameOf() {}
};

class PaperbackBook : public Document {};

int main() {
    Document * DocLib[10]; // Library of ten documents.
    for (int i = 0 ; i < 5 ; i++)
        DocLib[i] = new Document;
    for (int i = 5 ; i < 10 ; i++)
        DocLib[i] = new PaperbackBook;
}
```

В приведенном выше примере создаются разные типы. Однако поскольку все эти типы являются производными от класса `Document`, то выполняется неявное преобразование в `Document *`. В результате этого массив `DocLib` содержит объекты разных типов и является "разнородным списком" (то есть не все его объекты относятся к одному и тому же типу).

Поскольку класс `Document` имеет функцию `PrintNameOf`, он позволяет напечатать имя каждой книги в библиотеке, хотя при этом может опускаться часть информации, которая характерна только для данного типа документов (количество страниц для класса `Book`, размер в байтах для класса `HelpFile` и т. д.).

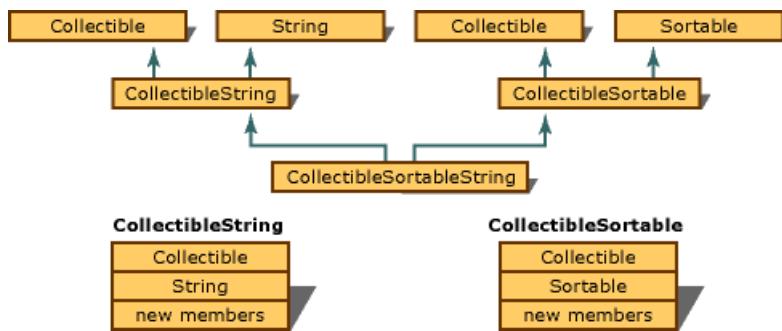
NOTE

Такие функции, как `PrintNameOf`, не рекомендуется реализовывать в базовых классах. [Виртуальные функции](#) предлагают другие альтернативные варианты проектирования.

Базовые классы

12.11.2021 • 2 minutes to read

Процесс наследования создает новый производный класс, который состоит из членов базового класса или классов и всех новых элементов, добавленных производным классом. В множественном наследовании можно создать граф наследования, где один и тот же базовый класс является частью нескольких производных классов. На следующем рисунке показан такой график.



Несколько экземпляров одного базового класса

На рисунке представлены наглядные представления компонентов `CollectibleString` И `CollectibleSortable`. Однако базовый класс (`Collectible`) находится в `CollectibleSortableString` На протяжении путей `CollectibleString` И `CollectibleSortable`. Для устранения этой избыточности такие классы при наследовании можно объявлять как виртуальные базовые классы.

Несколько базовых классов

12.11.2021 • 7 minutes to read

Класс может быть производным от более чем одного базового класса. В модели с множественным наследованием (где классы являются производными от более чем одного базового класса) базовые классы задаются с помощью элемента грамматики *базового списка*. Например, объявление класса для `CollectionOfBook`, производного от `Collection` и `Book`, можно указать следующим образом.

```
// deriv_MultipleBaseClasses.cpp
// compile with: /LD
class Collection {
};
class Book {};
class CollectionOfBook : public Book, public Collection {
    // New members
};
```

Порядок, в котором указываются базовые классы, не имеет значения, кроме некоторых случаев, когда вызываются конструкторы и деструкторы. В таких случаях порядок, в котором указываются базовые классы, влияет на следующее.

- Порядок, в котором конструктор выполняет инициализацию. Если код основан на том, что инициализация части `Book``CollectionOfBook` должна выполняться перед частью `Collection`, порядок указания важен. Инициализация выполняется в порядке указания классов в *базовом списке*.
- Порядок, в котором вызываются деструкторы для очистки. Опять же, если определенная часть класса должна присутствовать, а другая часть должна быть удалена, порядок имеет значение. Деструкторы вызываются в противоположном порядке классов, указанных в *базовом списке*.

NOTE

Порядок указания базовых классов может повлиять на структуру памяти класса. Не принимайте никаких программных решений на основе порядка базовых членов в памяти.

При указании *базового списка* нельзя указывать одно и то же имя класса более одного раза. Однако класс может стать косвенным базовым классом производного класса несколько раз.

Виртуальные базовые классы

Поскольку класс может несколько раз выступать как косвенный базовый класс к производному классу, в C++ имеется способ оптимизировать функционирование таких базовых классов. Виртуальные базовые классы позволяют экономить пространство и исключать неоднозначности в иерархиях классов, в которых используется множественное наследование.

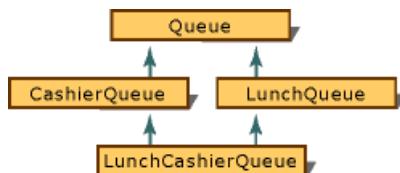
Каждый невиртуальный объект содержит копию элементов данных, определенных в базовом классе. Такой повтор данных приводит к ненужному увеличению их объема. Кроме того, при каждой попытке обращения к элементам базового класса приходится указывать, какая именно их копия требуется.

Если базовый класс определен как виртуальный базовый класс, то он может несколько раз выступать как косвенный базовый класс без дублирования элементов данных. Единственная копия его элементов

данных совместно используется всеми базовыми классами, которые используют его как виртуальный базовый класс.

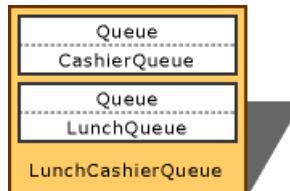
При объявлении виртуального базового класса `virtual` ключевое слово появляется в базовых списках производных классов.

Рассмотрим иерархию классов, представленную на следующем рисунке, на котором показана имитация графа Lunch-Line.



Смоделированный график обеда

Как видно на рисунке, класс `Queue` является базовым для двух других классов: `CashierQueue` И `LunchQueue`. Однако когда эти два класса объединяются и образуют класс `LunchCashierQueue`, возникает следующая проблема: новый класс содержит два подчиненных объекта типа `Queue` — один из `CashierQueue`, а другой из `LunchQueue`. На следующем рисунке показана концептуальная структура памяти (фактическая структура памяти может быть оптимизирована).

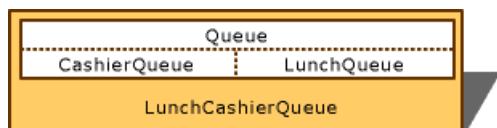


Имитируемый объект очереди

Обратите внимание, что в объекте `Queue` имеется два подчиненных объекта `LunchCashierQueue`. В следующем коде содержится объявление `Queue` как виртуального базового класса:

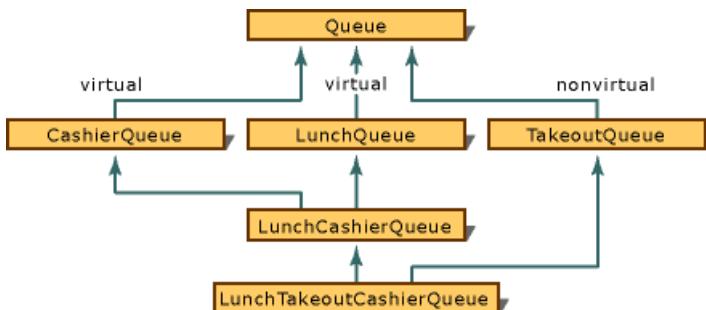
```
// deriv_VirtualBaseClasses.cpp
// compile with: /LD
class Queue {};
class CashierQueue : virtual public Queue {};
class LunchQueue : virtual public Queue {};
class LunchCashierQueue : public LunchQueue, public CashierQueue {};
```

`virtual` Ключевое слово гарантирует, что включается только одна копия `Queue` вложенного объекта (см. следующий рисунок).



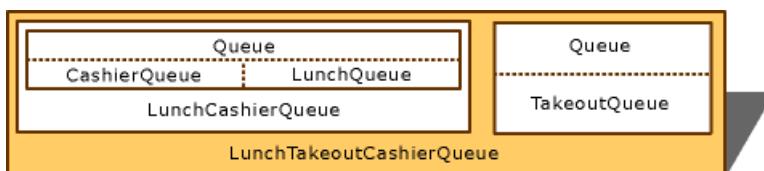
Имитация объекта обеда с виртуальными базовыми классами

Класс может иметь как виртуальный, так и невиртуальный компонент заданного типа. Это происходит при условиях, которые иллюстрирует следующий рисунок.



Виртуальные и невиртуальные компоненты одного и того же класса

На этом рисунке показано, что классы `CashierQueue` И `LunchQueue` используют `Queue` как виртуальный базовый класс. Однако `TakeoutQueue` определяется `Queue` в качестве базового класса, а не виртуального базового класса. Поэтому в `LunchTakeoutCashierQueue` имеется два подчиненных объекта типа `Queue`: один из пути наследования, включающего `LunchCashierQueue`, а второй из пути, включающего `TakeoutQueue`. Это показано на следующем рисунке.



Макет объекта с виртуальным и невиртуальным наследованием

NOTE

Наследование от виртуальных базовых классов позволяет существенно сократить объем данных по сравнению с наследованием от невиртуальных классов. Однако оно может породить дополнительные затраты на обработку.

Если производный класс переопределяет виртуальную функцию, которую он наследует от виртуального базового класса, и если конструктор или деструктор производного базового класса вызывает эту функцию при помощи указателя на виртуальный базовый класс, то компилятор может вставить дополнительные скрытые поля `vtordisp` в классы с виртуальными базовыми классами. `/vd0` Параметр компилятора подавляет Добавление скрытого элемента смещения конструктора или деструктора `vtordisp`. `/vd1` Параметр компилятора, используемый по умолчанию, включает их там, где это необходимо. Добавление полей `vtordisps` следует отключать только в том случае, если точно известно, что все конструкторы и деструкторы классов вызывают виртуальные функции виртуально.

`/vd` Параметр компилятора влияет на весь модуль компиляции. Используйте `vtordisp` директиву `pragma` для подавления и повторного включения `vtordisp` полей в зависимости от класса:

```
#pragma vtordisp( off )
class GetReal : virtual public { ... };
#pragma vtordisp( on )
```

Неоднозначность имен

Множественное наследование предоставляет возможность наследования имен по нескольким путям. Имена членов класса в этих путях не обязательно должны быть уникальными. Эти конфликты имен называются неоднозначностями.

Любое выражение, которое ссылается на член класса, должно иметь однозначную ссылку. В следующем примере показано, как появляются неоднозначности.

```

// deriv_NameAmbiguities.cpp
// compile with: /LD
// Declare two base classes, A and B.
class A {
public:
    unsigned a;
    unsigned b();
};

class B {
public:
    unsigned a(); // Note that class A also has a member "a"
    int b();      // and a member "b".
    char c;
};

// Define class C as derived from A and B.
class C : public A, public B {};

```

При наличии указанных выше объявлений класса код, такой как указано ниже, является неоднозначным, поскольку не ясно, ссылается ли `a` на `b` в `A` или в `B`.

```

C *pc = new C;

pc->b();

```

Рассмотрим предыдущий пример. Поскольку имя `a` является членом обоих классов `A` и `B`, компилятор не может определить, какая переменная `a` обозначает функцию, которую необходимо вызвать. Доступ к члену неоднозначен, если он может ссылаться на несколько функций, объектов, типов или перечислителей.

Компилятор определяет неоднозначности, выполняя тесты в указанном порядке.

- Если доступ к имени неоднозначен (как описано выше), создается сообщение об ошибке.
- Если перегруженные функции однозначны, они разрешаются.
- Если доступ к имени нарушает разрешение доступа к членам, создается сообщение об ошибке.
(Дополнительные сведения см. в разделе [Управление доступом к членам](#).)

Если выражение приводит к неоднозначности в результате наследования, его можно разрешить вручную, указав вместо данного имени имя класса. Чтобы выполнить компиляцию в предыдущем примере без неоднозначностей, можно использовать следующий код.

```

C *pc = new C;

pc->B::a();

```

NOTE

Если объявлен `C`, могут возникнуть ошибки, если сослаться на `b` в области `C`. Однако ошибка не выдается, если не внести неквалифицированную ссылку на `b` в области `C`.

Доминирование

Через граф наследования можно достичь несколько имен (функции, объекта или перечислителя). С невиртуальными базовыми классами такие случаи неоднозначны. Они также неоднозначны с виртуальными базовыми классами, если одно из имен не доминирует над другими.

То или иное имя доминирует над другим, если оно определено в обоих классах и один класс является производным от другого. Доминирующее имя — это имя в производном классе; оно используется тогда, когда в противном случае возникла бы неоднозначность, как показано в следующем примере.

```
// deriv_Dominance.cpp
// compile with: /LD
class A {
public:
    int a;
};

class B : public virtual A {
public:
    int a();
};

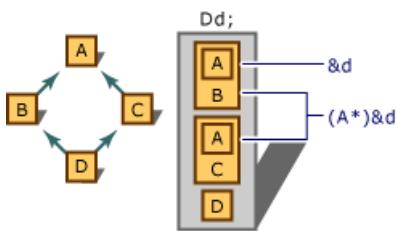
class C : public virtual A {};

class D : public B, public C {
public:
    D() { a(); } // Not ambiguous. B::a() dominates A::a.
};
```

Неоднозначные преобразования

Явные и неявные преобразования указателей и ссылок в типы класса могут приводить к неоднозначности. На следующем рисунке "Неоднозначное преобразование указателей в базовые классы" показано следующее:

- Объявление объекта типа `D`.
- Результат применения оператора взятия адреса (`&`) к этому объекту. Обратите внимание, что оператор взятия адреса всегда возвращает базовый адрес объекта.
- Результат явного преобразования указателя, полученного с помощью оператора взятия адреса, в тип базового класса `A`. Обратите внимание, что приведение адреса объекта к типу `A*` не всегда предоставляет компилятору достаточно информации о том, какой из типов подобъектов типа `A` следует выбрать; в данном случае существуют два подобъекта.



Неоднозначное преобразование указателей в базовые классы

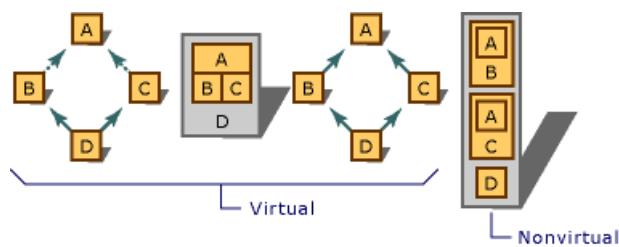
Преобразование в тип `A*` (указатель на `A`) является неоднозначным, поскольку нет способа определить, какой подобъект типа `A` является правильным. Обратите внимание, что неоднозначности можно избежать, явно указав используемый подобъект, как показано ниже:

```
(A *)(B *)&d      // Use B subobject.
(A *)(C *)&d      // Use C subobject.
```

Неоднозначности и виртуальные базовые классы

Если используются виртуальные базовые классы, доступ к функциям, объектам, типам и перечислениям можно получить по путям множественного наследования. Поскольку существует только один экземпляр базового класса, неоднозначность при доступе к этим именам отсутствует.

На следующем рисунке показано составление объектов с использованием виртуального и невиртуального наследования.



Виртуальное и невиртуальное наследование

На этом рисунке доступ к любому члену класса `A` через невиртуальную базовую классы вызывает неоднозначность; у компилятора нет сведений, поясняющих, нужно ли использовать вложенный объект, связанный с `B`, или вложенный объект, связанный с `C`. Однако если `A` задано как виртуальный базовый класс, вопросов о том, к какому из вложенных объектов осуществляется доступ, не возникает.

См. также раздел

[Наследование](#)

Явные переопределения (C++)

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Если одна и та же виртуальная функция объявлена в двух или более [интерфейсах](#), а класс является производным от этих интерфейсов, можно явно переопределить каждую виртуальную функцию.

Сведения о явных переопределениях в управляемом коде с использованием C++/CLI см. в разделе [явные переопределения](#).

Завершение блока, относящегося только к системам Microsoft

Пример

В следующем примере кода показано использование явного переопределения:

```
// deriv_ExplicitOverrides.cpp
// compile with: /GR
extern "C" int printf_s(const char *, ...);

__interface IMyInt1 {
    void mf1();
    void mf1(int);
    void mf2();
    void mf2(int);
};

__interface IMyInt2 {
    void mf1();
    void mf1(int);
    void mf2();
    void mf2(int);
};

class CMyClass : public IMyInt1, public IMyInt2 {
public:
    void IMyInt1::mf1() {
        printf_s("In CMyClass::IMyInt1::mf1()\n");
    }

    void IMyInt1::mf1(int) {
        printf_s("In CMyClass::IMyInt1::mf1(int)\n");
    }

    void IMyInt1::mf2();
    void IMyInt1::mf2(int);

    void IMyInt2::mf1() {
        printf_s("In CMyClass::IMyInt2::mf1()\n");
    }

    void IMyInt2::mf1(int) {
        printf_s("In CMyClass::IMyInt2::mf1(int)\n");
    }

    void IMyInt2::mf2();
    void IMyInt2::mf2(int);
};
```

```

void CMyClass::IMyInt1::mf2() {
    printf_s("In CMyClass::IMyInt1::mf2()\n");
}

void CMyClass::IMyInt1::mf2(int) {
    printf_s("In CMyClass::IMyInt1::mf2(int)\n");
}

void CMyClass::IMyInt2::mf2() {
    printf_s("In CMyClass::IMyInt2::mf2()\n");
}

void CMyClass::IMyInt2::mf2(int) {
    printf_s("In CMyClass::IMyInt2::mf2(int)\n");
}

int main() {
    IMyInt1 *pIMyInt1 = new CMyClass();
    IMyInt2 *pIMyInt2 = dynamic_cast<IMyInt2 *>(pIMyInt1);

    pIMyInt1->mf1();
    pIMyInt1->mf1(1);
    pIMyInt1->mf2();
    pIMyInt1->mf2(2);
    pIMyInt2->mf1();
    pIMyInt2->mf1(3);
    pIMyInt2->mf2();
    pIMyInt2->mf2(4);

    // Cast to a CMyClass pointer so that the destructor gets called
    CMyClass *p = dynamic_cast<CMyClass *>(pIMyInt1);
    delete p;
}

```

```

In CMyClass::IMyInt1::mf1()
In CMyClass::IMyInt1::mf1(int)
In CMyClass::IMyInt1::mf2()
In CMyClass::IMyInt1::mf2(int)
In CMyClass::IMyInt2::mf1()
In CMyClass::IMyInt2::mf1(int)
In CMyClass::IMyInt2::mf2()
In CMyClass::IMyInt2::mf2(int)

```

См. также раздел

[Наследование](#)

Абстрактные классы (C++)

12.11.2021 • 2 minutes to read

Абстрактные классы используются в качестве обобщенных концепций, на основе которых можно создавать более конкретные производные классы. Нельзя создать объект типа абстрактного класса. Однако можно использовать указатели и ссылки на абстрактные типы классов.

Абстрактный класс создается путем объявления по крайней мере одной чистой виртуальной функции члена. Это виртуальная функция, объявленная с помощью синтаксиса *чистого описателя* (`= 0`). Классы, производные от абстрактного класса, должны реализовывать чисто виртуальную функцию; в противном случае они также будут абстрактными.

Рассмотрим пример, представленный в [виртуальных функциях](#). Класс `Account` создан для того, чтобы предоставлять общие функции, но объекты типа `Account` имеют слишком общий характер для практического применения. Это означает `Account` хороший кандидат для абстрактного класса:

```
// deriv_AbstractClasses.cpp
// compile with: /LD
class Account {
public:
    Account( double d ); // Constructor.
    virtual double GetBalance(); // Obtain balance.
    virtual void PrintBalance() = 0; // Pure virtual function.
private:
    double _balance;
};
```

Единственное различие между этим и предыдущим объявлениемми состоит в том, что функция `PrintBalance` объявлена со спецификатором чисто виртуальной функции *pure* (`= 0`).

Ограничения на использование абстрактных классов

Абстрактные классы нельзя использовать для:

- переменных и данных членов;
- типов аргументов;
- типов возвращаемых функциями значений;
- типов явных преобразований.

Если конструктор абстрактного класса вызывает чисто виртуальную функцию, прямо или косвенно, результат не определен. Однако конструкторы и деструкторы абстрактных классов могут вызывать другие функции-члены.

Определенные чистые виртуальные функции

Чистые виртуальные функции в абстрактных классах могут быть *определенны* или иметь реализацию. Вызывать эти функции можно только с помощью полного синтаксиса:

abstract — имя класса:Function-Name()

Определенные чистые виртуальные функции полезны при проектировании иерархий классов, базовые

классы которых содержат чистые виртуальные деструкторы. Это обусловлено тем, что деструкторы базового класса всегда вызываются во время уничтожения объекта. Рассмотрим следующий пример.

```
// deriv_RestrictionsOnUsingAbstractClasses.cpp
// Declare an abstract base class with a pure virtual destructor.
// It's the simplest possible abstract class.
class base
{
public:
    base() {}
    // To define the virtual destructor outside the class:
    virtual ~base() = 0;
    // Microsoft-specific extension to define it inline:
//    virtual ~base() = 0 {};
};

base::~base() {} // required if not using Microsoft extension

class derived : public base
{
public:
    derived() {}
    ~derived() {}
};

int main()
{
    derived aDerived; // destructor called when it goes out of scope
}
```

В примере показано, как расширение компилятора Майкрософ트 позволяет добавить встроенное определение в чистый виртуальный `~base()`. Его также можно определить за пределами класса с помощью `base::~base() {}`.

Когда объект `aDerived` выходит из области действия, `derived` вызывается деструктор класса.

Компилятор создает код для неявного вызова деструктора класса `base` после `derived` деструктора.

Пустая реализация чисто виртуальной функции `~base` гарантирует, что для функции существует хотя бы определенная реализация. Без него компоновщик создает неразрешенную ошибку внешнего символа для неявного вызова.

NOTE

В предыдущем примере чистая виртуальная функция `base::~base` вызывается неявно из `derived::~derived`.

Кроме того, можно явно вызывать чистые виртуальные функции, используя полное имя функции-члена. Такие функции должны иметь реализацию, или вызов приводит к ошибке во время компоновки.

См. также раздел

[Наследование](#)

Общие сведения о правилах области

12.11.2021 • 2 minutes to read

Использование имени должно быть однозначно в пределах его области видимости (до той точки, в которой задана перезагрузка). Если имя обозначает функцию, то она должна быть однозначной с точки зрения количества и типа параметров. Если имя остается неоднозначным, применяются правила [доступа к членам](#).

Инициализаторы конструктора

[Инициализаторы конструктора](#) оцениваются в области внешнего блока конструктора, для которого они указаны. Следовательно, они могут использовать имена параметров конструктора.

Глобальные имена

Имя объекта, функции или перечислителя является глобальным, если оно объявлено вне любых функций или классов либо в качестве префикса указан глобальный унарный оператор области действия (`::`) и это имя не используется в сочетании с любым из следующих бинарных операторов:

- Разрешение области (`::`)
- Выбор членов для объектов и ссылок (`.`)
- Выбор членов для указателей (`->`)

Полные имена

Имена, используемые с бинарным оператором разрешения области действия (`::`), называются полными. Имя, указанное после такого оператора, должно быть членом класса, заданного слева от оператора, или членом его базового класса (классов).

Имена, указанные после оператора выбора члена (`.` или `->`) должны быть членами типа класса объекта, указанного слева от оператора или членов его базового класса (ES). Имена, указанные справа от оператора выбора члена (`->`), также могут быть объектами другого типа класса, при условии, что левая часть `->` является объектом класса и что класс определяет перегруженный оператор выбора члена (`->`), результатом которого является указатель на какой-либо другой тип класса. (Эта процедура более подробно обсуждается в разделе [доступ к членам класса](#).)

Компилятор осуществляет поиск имен в указанном ниже порядке. При обнаружении искомого имени поиск останавливается.

1. Поиск в текущей области видимости блока, если имя используется внутри функции; в противном случае поиск выполняется в глобальной области.
2. Поиск за пределами каждой включающей области видимости блока, в том числе во внешней области видимости функции (которая содержит параметры функции).
3. Если имя используется в функции-члене, оно ищется в области видимости класса.
4. Поиск имени выполняется в базовых классах класса.
5. Поиск во включающей области видимости вложенного класса (если она имеется) и ее базах. Он продолжается до тех пор, пока не будет просмотрена внешняя включающая область видимости

класса.

6. Поиск в глобальной области.

В этот порядок поиска можно внести изменения следующим образом.

1. Если именам предшествует оператор `::`, поиск начинается в глобальной области.
2. Имена, которым предшествуют `class` `struct` Ключевые слова, и, `union` заставляют компилятор искать только `class` имена, `struct` ИЛИ `union`.
3. Имена в левой части оператора разрешения области действия (`::`) могут быть только `class` `struct` именами,, `namespace` ИЛИ `union`.

Если имя относится к нестатическому члену, но используется в статической функции-члене, выдается сообщение об ошибке. Аналогично, если имя относится к любому нестатическому члену в включающем классе, создается сообщение об ошибке, поскольку вложенные классы не имеют `this` указателей классов.

Имена параметров функции

Имена параметров функций в определениях функций считаются принадлежащими области видимости самого внешнего блока функции. Следовательно, это локальные имена, которые исчезают из области видимости при выходе из функции.

Имена параметров функций в объявлениях функций (прототипах) находятся в локальной области видимости объявления и выходят из области видимости в конце объявления.

Параметры, имеющие значения по умолчанию, находятся в области видимости параметра, для которого они являются параметрами по умолчанию, как описано в предыдущих двух параграфах. Однако они не могут получить доступ к локальным переменным или нестатическим членам класса. Параметры, имеющие значения по умолчанию, вычисляются в точке вызова функции, но в исходной области видимости объявления функции. Поэтому параметры, имеющие значения по умолчанию, для функций-членов всегда вычисляются в области видимости класса.

См. также раздел

[Наследование](#)

Ключевые слова наследования

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

```
class class-name  
class __single_inheritance class-name  
class __multiple_inheritance class-name  
class __virtual_inheritance class-name
```

где:

```
class-name
```

Имя объявляемого класса.

C++ позволяет объявить указатель на член класса перед определением класса. Пример:

```
class S;  
int S::*p;
```

В приведенном выше коде `p` объявляется как указатель на целочисленный член класса S. Однако `class s` еще не определена в этом коде; он объявлен только как. Когда компилятор обнаруживает такой указатель, он должен создать обобщенное представление указателя. Размер представления зависит от указанной модели наследования. Существует три способа указания модели наследования для компилятора:

- В командной строке с `/vmp` параметром
- Использование `pointers_to_members` директивы pragma
- С помощью ключевых слов наследования `__single_inheritance`, `__multiple_inheritance` и `__virtual_inheritance`. При использовании этого метода управление моделью наследования осуществляется на уровне класса.

NOTE

Если указатель на член класса всегда объявляется после определения класса, нет необходимости использовать какой-либо из этих параметров.

Если вы объявили указатель на член класса до определения класса, он может негативно повлиять на размер и скорость получаемого исполняемого файла. Чем сложнее наследование, используемое классом, тем больше число байтов, необходимое для представления указателя на член класса. И, чем больше код, необходимый для интерпретации указателя. Одиночное (или нет) наследование не менее сложное, а виртуальное наследование является наиболее сложным. Указатели на члены, объявляемые до определения класса, всегда используют наибольшее, более сложное представление.

Если приведенный выше пример изменить на

```
class __single_inheritance S;  
int S::*p;
```

в то же время, независимо от указанных параметров командной строки или директив pragma, указатели на члены `class s` будут использовать наименьшее возможное представление.

NOTE

То же опережающее объявление представления указателя на член должно быть включено в каждую запись преобразования, которая объявляет указатели на члены этого класса, и объявление должно выполняться до объявления указателей на члены.

Для совместимости с предыдущими версиями, `__single_inheritance`, `__multiple_inheritance` и `__virtual_inheritance` являются синонимами для `__single_inheritance`, `__multiple_inheritance` и `__virtual_inheritance` если не задан параметр компилятора, если не `/za` (Отключить расширения языка)

Завершение блока, относящегося только к системам Майкрософт

См. также

[Ключевые слова](#)

virtual (C++)

12.11.2021 • 2 minutes to read

`virtual` Ключевое слово объявляет виртуальную функцию или виртуальный базовый класс.

Синтаксис

```
virtual [type-specifiers] member-function-declarator  
virtual [access-specifier] base-class-name
```

Параметры

описатели типа

Указывает тип возвращаемого значения виртуальной функции-члена.

декларатор с функцией-членом

Объявляет функцию-член.

описатель доступа

Определяет уровень доступа к базовому классу, `public` `protected` ИЛИ `private`. Может использоваться до или после `virtual` ключевого слова.

имя базового класса

Определяет ранее объявленный тип класса.

Remarks

Дополнительные сведения см. в разделе [виртуальные функции](#).

Также см. следующие ключевые слова: [класс](#), [частный](#), [открытый](#) и [защищенный](#).

См. также

[Ключевые слова](#)

Блок, относящийся только к системам Microsoft

Позволяет явно указать, что для переопределяемой функции вызывается реализация из базового класса.

Синтаксис

```
_super::member_function();
```

Remarks

На этапе разрешения перегрузки учитываются все доступные методы базового класса, и вызывается функция, которая обеспечивает наилучшее совпадение.

`_super` может использоваться только в теле функции-члена.

`_super` не может использоваться с объявлением `using`. Дополнительные сведения см. [в разделе Использование объявления](#).

С появлением [атрибутов](#), которые вставляют код, код может содержать один или несколько базовых классов, имена которых могут быть незнакомы, но содержат методы, которые вы хотите вызвать.

Пример

```
// deriv_super.cpp
// compile with: /c
struct B1 {
    void mf(int) {}
};

struct B2 {
    void mf(short) {}

    void mf(char) {}
};

struct D : B1, B2 {
    void mf(short) {
        _super::mf(1);    // Calls B1::mf(int)
        _super::mf('s');  // Calls B2::mf(char)
    }
};
```

Завершение блока, относящегося только к системам Майкрософт

См. также

[Ключевые слова](#)

`_interface`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Интерфейс Microsoft C++ можно определить следующим образом:

- Может наследовать от произвольного (включая 0) числа базовых интерфейсов.
- Не может наследовать от базового класса.
- Может содержать только открытые, чисто виртуальные методы.
- Не может содержать конструкторы, деструкторы или операторы.
- Не может содержать статические методы.
- Не может содержать члены данных; свойства допускаются.

Синтаксис

```
modifier __interface interface-name {interface-definition};
```

Remarks

Класс или [структуру](#) C++ можно реализовать с помощью этих правил, но `__interface` применять их.

Например, ниже приведен пример определения интерфейса:

```
__interface IMyInterface {
    HRESULT CommitX();
    HRESULT get_X(BSTR* pbstrName);
};
```

Дополнительные сведения об управляемых интерфейсах см. в разделе [класс интерфейса](#).

Обратите внимание — нет необходимости явно указывать, что функции `CommitX` и `get_X` являются чистой виртуальными. Эквивалентное объявление для первой функции могло бы быть следующим:

```
virtual HRESULT CommitX() = 0;
```

`__interface` подразумевает модификатор `vtable` `__declspec`.

Пример

В следующем примере показано, как использовать свойства, объявленные в интерфейсе.

```
// deriv_interface.cpp
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <string.h>
#include <comdef.h>
```

```
#include <stdio.h>

[module(name="test")];

[ object, uuid("00000000-0000-0000-0000-000000000001"), library_block ]
__interface IFace {
    [ id(0) ] int int_data;
    [ id(5) ] BSTR bstr_data;
};

[ coclass, uuid("00000000-0000-0000-0000-000000000002") ]
class MyClass : public IFace {
private:
    int m_i;
    BSTR m_bstr;

public:
    MyClass()
    {
        m_i = 0;
        m_bstr = 0;
    }

    ~MyClass()
    {
        if (m_bstr)
            ::SysFreeString(m_bstr);
    }

    int get_int_data()
    {
        return m_i;
    }

    void put_int_data(int _i)
    {
        m_i = _i;
    }

    BSTR get_bstr_data()
    {
        BSTR bstr = ::SysAllocString(m_bstr);
        return bstr;
    }

    void put_bstr_data(BSTR bstr)
    {
        if (m_bstr)
            ::SysFreeString(m_bstr);
        m_bstr = ::SysAllocString(bstr);
    }
};

int main()
{
    _bstr_t bstr("Testing");
    CoInitialize(NULL);
    CComObject<MyClass>* p;
    CComObject<MyClass>::CreateInstance(&p);
    p->int_data = 100;
    printf_s("p->int_data = %d\n", p->int_data);
    p->bstr_data = bstr;
    printf_s("bstr_data = %S\n", p->bstr_data);
}
```

```
p->int_data = 100  
bstr_data = Testing
```

Завершение блока, относящегося только к системам Майкрософт

См. также

[Ключевые слова](#)

[Атрибуты интерфейса](#)

Специальные функции-члены

12.11.2021 • 2 minutes to read

К *особым функциям-членам* относятся члены класса (или структуры), которые в некоторых случаях автоматически генерируются компилятором. Эти функции являются [конструктором по умолчанию](#), [деструктором](#), [конструктором копирования](#) и [оператором присваивания копирования](#), а также [конструктором перемещения](#) и [оператором присваивания перемещения](#). Если в классе не определена одна или несколько специальных функций члена, компилятор может неявно объявить и определить используемые функции. Реализации, создаваемые компилятором, называются специальными функциями-членами *по умолчанию*. Компилятор не создает функции, если они не требуются.

Вы можете явно объявить специальную функцию-член по умолчанию с помощью ключевого слова = `Default`. Это приводит к тому, что компилятор определит функцию только при необходимости, так же, как если бы функция не была объявлена вообще.

В некоторых случаях компилятор может создавать *Удаленные специальные функции* элементов, которые не определены и поэтому не вызываемы. Это может произойти в тех случаях, когда вызов определенной специальной функции-члена в классе не имеет смысла, учитывая другие свойства класса. Чтобы явно запретить автоматическое создание специальной функции-члена, можно объявить ее как удаленную с помощью ключевого слова = `Delete`.

Компилятор создает *конструктор по умолчанию*— конструктор, который не принимает аргументы, только если не объявлен какой-либо другой конструктор. Если вы объявили только конструктор, который принимает параметры, то код, который пытается вызвать конструктор по умолчанию, заставляет компилятор выдавать сообщение об ошибке. Созданный компилятором конструктор по умолчанию выполняет простую [инициализацию объекта по умолчанию](#) на уровне элементов. При инициализации по умолчанию все переменные элемента остаются в неопределенном состоянии.

Деструктор по умолчанию выполняет уничтожение объекта на уровне элементов. Он является виртуальным только в том случае, если деструктор базового класса является виртуальным.

Операции по созданию и перемещению по умолчанию для операций копирования и перемещения выполняют копирование или перемещение нестатических элементов данных с помощью побитового шаблона. Операции перемещения создаются только в том случае, если не объявлен ни один деструктор или операции перемещения или копирования. Конструктор копии по умолчанию создается только в том случае, если конструктор копии не объявлен. Он неявно удаляется при объявлении операции перемещения. Оператор присваивания копии по умолчанию создается только в том случае, если оператор присваивания копии не объявлен явным образом. Он неявно удаляется при объявлении операции перемещения.

См. также

[Справочник по языку C++](#)

Статические члены (C++)

12.11.2021 • 2 minutes to read

Классы могут содержать статические данные-члены и функции-члены. При объявлении элемента данных как `static`, только одна копия данных сохраняется для всех объектов класса.

Статические данные-члены не входят в состав объектов указанного типа класса. В результате объявление статических данных-члена не является определением. Данные-член объявляются в области видимости класса, однако определение выполняется в области видимости файла. Такие статические члены имеют внешнюю компоновку; Проиллюстрируем это на примере.

```
// static_data_members.cpp
class BufferedOutput
{
public:
    // Return number of bytes written by any object of this class.
    short BytesWritten()
    {
        return bytecount;
    }

    // Reset the counter.
    static void ResetCount()
    {
        bytecount = 0;
    }

    // Static member declaration.
    static long bytecount;
};

// Define bytecount in file scope.
long BufferedOutput::bytecount;

int main()
{
}
```

В предыдущем примере член `bytecount` объявляется в классе `BufferedOutput`, однако его определение должно находиться за пределами объявления класса.

На статические данные-члены можно ссылаться без указания ссылок на объект типа класса. Число байтов, записываемых с помощью объектов `BufferedOutput`, можно получить следующим образом.

```
long nBytes = BufferedOutput::bytecount;
```

Для того чтобы существовал статический член, не обязательно, чтобы существовали любые объекты типа класса. Доступ к статическим членам также можно получить с помощью выбора члена (`. ->` операторы и). Пример:

```
BufferedOutput Console;

long nBytes = Console.bytecount;
```

В предыдущем случае не ссылка на объект (`Console`) не вычисляется; возвращается значение статического объекта `bytecount`.

Статические данные-члены подчиняются правилам доступа члена класса, поэтому закрытый доступ к ним допускается только для функций и дружественных объектов класса-члена. Эти правила описаны в разделе [Управление доступом к членам](#). Имеется исключение: статические данные-члены должны быть определены в области видимости файла, вне зависимости от их ограничений доступа. Если данные-член необходимо инициализировать явным образом, инициализатор должен быть предоставлен в определении.

Тип статического члена не квалифицируется его именем класса. Таким образом, типом `BufferedOutput::bytecount` является `long`.

См. также

[Классы и структуры](#)

Заданные пользователем преобразования типов (C++)

12.11.2021 • 8 minutes to read

Преобразование создает новое значение некоторого типа из значения другого типа. *Стандартные преобразования* встроены в язык C++ и поддерживают встроенные типы. Кроме того, можно создавать *пользовательские преобразования* для выполнения преобразований в типы, из или между определяемыми пользователем типами.

Стандартные преобразования выполняют преобразование между встроенными типами, между указателями или ссылками на типы, связанные наследованием, в и из указателей void и в пустой указатель. Дополнительные сведения см. в разделе [стандартные преобразования](#). Пользовательские преобразования выполняют преобразование между пользовательскими типами или между пользовательскими и встроенными типами. Их можно реализовать как [конструкторы преобразования](#) или как [функции преобразования](#).

Преобразования могут быть явными, когда программист вызывает преобразование одного типа в другой (как в приведении или прямой инициализации) или неявными, когда язык или программа вызывают типы, которые отличаются от заданных программистом.

Попытка неявного преобразования выполняется, когда

- тип аргумента, предоставленного для функции, не совпадает с соответствующим параметром;
- тип значения, возвращаемого функцией, не совпадает с типом возвращаемого значения функции;
- тип выражения инициализатора не совпадает с типом инициализируемого объекта;
- тип результата выражения, которое управляет условным оператором, циклической конструкцией или параметром, не совпадает с тем, который требуется для управления;
- тип операнда, предоставленного для оператора, не совпадает с соответствующим параметром операнда. Для встроенных операторов тип обоих операндов должен совпадать; он преобразуется в общий тип, который может представлять оба операнда. Дополнительные сведения см. в разделе [стандартные преобразования](#). Для пользовательских операторов тип каждого операнда должен совпадать с соответствующим параметром операнда.

Если не удается выполнить неявное преобразование с помощью стандартного преобразования, компилятор может использовать пользовательское преобразование, за которым (при необходимости) будет следовать дополнительное стандартное преобразование.

Если на сайте преобразования есть два и более пользовательских преобразования, выполняющих одно преобразование, преобразование называется неоднозначным. Неоднозначность подразумевает ошибку, так как компилятор не может определить, какое из доступных преобразований выбрать. Тем не менее, не будет ошибкой определить несколько способов выполнения одного преобразования, так как набор доступных преобразований может отличаться в разных участках исходного кода, например в зависимости от того, какие файлы заголовков входят в исходный файл. Пока на сайте преобразования доступно только одно преобразование, о неоднозначности речь не идет. Существует несколько путей возникновения неоднозначных преобразований, однако самые распространенные перечислены ниже.

- Множественное наследование. Преобразование определено в нескольких базовых классах.
- Вызов неоднозначной функции. Преобразование определено как конструктор преобразования

типа целевого объекта и как функция преобразования типа источника. Дополнительные сведения см. в разделе [функции преобразования](#).

Неоднозначность, как правило, можно устраниТЬ, просто более полно указав имя соответствующего типа или выполнив явное приведение для пояснения намерения.

Конструкторы преобразования и функции преобразования подчиняются правилам управления доступом членов, однако доступность преобразований учитывается, только если можно определить неоднозначное преобразование. Это означает, что преобразование может быть неоднозначным, даже если уровень доступа конкурирующего преобразования будет блокировать его использование. Дополнительные сведения о специальных возможностях членов см. в разделе [Управление доступом к членам](#).

Ключевое слово `explicit` и проблемы с неявным преобразованием

По умолчанию при создании пользовательского преобразования компилятор может использовать его для выполнения неявных преобразований. Иногда это совпадает с вашими намерениями, но в других случаях простые правила, которые определяют выполнение неявных преобразований компилятором, могут привести к тому, что он примет нежелательный код.

Один известный пример неявного преобразования, который может вызвать проблемы, — преобразование в `bool`. Существует множество причин, по которым может потребоваться создать тип класса, который может использоваться в логическом контексте, например, чтобы его можно было использовать для управления `if` оператором или циклом, но когда компилятор выполняет определенное пользователем преобразование в встроенный тип, компилятор может применить дополнительное стандартное преобразование впоследствии. Назначение этого дополнительного стандартного преобразования заключается в том, чтобы сделать возможным продвижение от `short` до `int`, но оно также открывает дверцу для менее очевидных преобразований, например из `bool` в `int`, что позволяет использовать тип класса в целочисленных контекстах, которые никогда не предполагались. Эта конкретная проблема известна как *проблема Сейф Bool*. Проблема такого рода заключается в том, что `explicit` ключевое слово может помочь.

`explicit` Ключевое слово сообщает компилятору, что указанное преобразование нельзя использовать для выполнения неявных преобразований. Если вы хотите использовать синтаксис неявных преобразований до того `explicit`, как было введено ключевое слово, пришлось либо принять непредвиденные последствия неявного преобразования, либо использовать в качестве обходного пути менее удобные функции именованного преобразования. Теперь с помощью `explicit` ключевого слова можно создавать удобные преобразования, которые можно использовать только для выполнения явных приведений или прямой инициализации, что не приводит к возникновению проблем, содержащихся с проблемой Сейф Bool.

`explicit` Ключевое слово можно применять к конструкторам преобразования с C++ 98, а также к функциям преобразования, начиная с C++ 11. В следующих разделах содержатся дополнительные сведения об использовании `explicit` ключевого слова.

Конструкторы преобразования

Конструкторы преобразования определяют преобразование из пользовательских или встроенных типов в пользовательские типы. В следующем примере демонстрируется конструктор преобразования, который преобразует из встроенного типа `double` в определяемый пользователем тип `Money`.

```

#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance.amount << std::endl;
}

int main(int argc, char* argv[])
{
    Money payable{ 79.99 };

    display_balance(payable);
    display_balance(49.95);
    display_balance(9.99f);

    return 0;
}

```

Обратите внимание, что первый вызов функции `display_balance`, которая принимает аргументы типа `Money`, не требует преобразования, так как аргумент принадлежит кциальному типу. Однако при втором вызове требуется `display_balance`. Преобразование, так как тип аргумента, `double` со значением `49.95`, не является функцией, которую требует функция. Функция не может использовать это значение напрямую, но из-за преобразования из типа аргумента — `double` в тип соответствующего параметра — `Money` — временное значение типа `Money` формируется из аргумента и используется для завершения вызова функции. При третьем вызове `display_balance` функции Обратите внимание, что аргумент не является `double`, но вместо него имеет `float` значение `9.99` – и, однако, функция может быть завершена, так как компилятор может выполнить стандартное преобразование — в данном случае —, `float` `double` а затем выполнить пользовательское преобразование из `double` В для `Money` завершения необходимого преобразования.

Объявление конструкторов преобразования

Следующие правила применяются к объявлению конструктора преобразования.

- Целевым типом преобразования является сконструированный пользовательский тип.
- Конструкторы преобразований, как правило, принимают только один аргумент типа источника. Однако конструктор преобразования может указывать дополнительные параметры, если у каждого из них есть значение по умолчанию. Тип источника остается типом первого параметра.
- Конструкторы преобразований, как и все конструкторы, не указывают тип возвращаемого значения. Указание типа возвращаемого значения в объявлении является ошибкой.
- Конструкторы преобразования могут быть явными.

Явные конструкторы преобразования

Объявляя конструктор преобразования как `explicit`, его можно использовать только для выполнения непосредственной инициализации объекта или для выполнения явного приведения. Это не дает функциям, которые принимают аргумент типа класса, также неявно принимать аргументы типа источника конструктора преобразования, а также блокирует инициализацию копирования типа класса из значения типа источника. В следующем примере демонстрируется определение явного конструктора

преобразования и влияние на правильный синтаксис кода.

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    explicit Money(double _amount) : amount{ _amount } {};

    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance.amount << std::endl;
}

int main(int argc, char* argv[])
{
    Money payable{ 79.99 };           // Legal: direct initialization is explicit.

    display_balance(payable);         // Legal: no conversion required
    display_balance(49.95);           // Error: no suitable conversion exists to convert from double to Money.
    display_balance((Money)9.99f);     // Legal: explicit cast to Money

    return 0;
}
```

В этом примере обратите внимание, что явный конструктор преобразования можно использовать для выполнения прямой инициализации типа `payable`. Если же вы попытаетесь выполнить инициализацию копирования `Money payable = 79.99;`, это приведет к ошибке. Первый вызов `display_balance` не включает преобразование, так как указан аргумент правильного типа. Второй вызов `display_balance` является ошибкой, так как конструктор преобразования нельзя использовать для выполнения неявного преобразования. Третий вызов `display_balance` является допустимым из-за явного приведения к `Money`, но обратите внимание, что компилятор по-прежнему помог выполнить приведение путем вставки неявного приведения `float` в `double`.

Несмотря на то, что использование неявных преобразований кажется удобным, в результате могут возникать трудновыявляемые ошибки. Как показывает опыт, лучше всего объявлять все конструкторы преобразований явными за исключением тех случаев, когда необходимо, чтобы определенное преобразование выполнялось неявно.

ФУНКЦИИ ПРЕОБРАЗОВАНИЯ

Функции преобразования определяют преобразования из пользовательского в другие типы. Эти функции иногда называют "операторами приведения", так как они, наряду с конструкторами преобразования, вызываются, когда значение приводится к другому типу. В следующем примере демонстрируется функция преобразования, преобразующая из определяемого пользователем типа, `Money` в встроенный тип `double`:

```

#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    operator double() const { return amount; }
private:
    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << balance << std::endl;
}

```

Обратите внимание, что переменная `amount` -член сделана закрытой, а открытая функция преобразования в тип `double` введена только для возврата значения `amount`. В функции `display_balance` неявное преобразование возникает, когда значение `balance` направляется в стандартный вывод с помощью оператора вставки в поток `<<`. Так как для определяемого пользователем типа не определен оператор вставки потока, `Money` но имеется один для встроенного типа `double`, компилятор может использовать функцию преобразования из `Money` в `double` для удовлетворения оператора вставки потока.

Функции преобразования наследуются производными классами. Функции преобразования в производном классе переопределяют наследуемую функцию преобразования, только когда выполняют преобразование в точно такой же тип. Например, определяемая пользователем функция преобразования оператора производного класса `int` не переопределяет (или даже не влияет) на определяемую пользователем функцию преобразования **оператора** базового класса `Short`, даже если стандартные преобразования определяют отношение преобразования между `int` и `short`.

Объявление функций преобразования

Следующие правила применяются к объявлению функции преобразования.

- Целевой тип преобразования должен быть объявлен до объявления функции преобразования. Классы, структуры, перечисления и определения типа нельзя объявлять в объявлении функции преобразования.

```
operator struct String { char string_storage; }() // illegal
```

- Функции преобразования не принимают аргументов. Указание любых параметров в объявлении является ошибкой.
- Функции преобразования имеют тип возвращаемого значения, задаваемый именем функции преобразования, которое также является именем типа целевого объекта преобразования. Указание типа возвращаемого значения в объявлении является ошибкой.
- Функции преобразования могут быть виртуальными.
- Функции преобразования могут быть явными.

Явные функции преобразования

Если функция преобразования объявлена как явная, ее можно использовать только для выполнения явного приведения. Это не дает функциям, которые принимают аргумент типа целевого объекта функции преобразования, также неявно принимать аргументы типа класса, а также блокирует

инициализацию копирования экземпляров типа целевого объекта из значения типа класса. В следующем примере демонстрируется определение явной функции преобразования и влияние на правильный синтаксис кода.

```
#include <iostream>

class Money
{
public:
    Money() : amount{ 0.0 } {};
    Money(double _amount) : amount{ _amount } {};

    explicit operator double() const { return amount; }

private:
    double amount;
};

void display_balance(const Money balance)
{
    std::cout << "The balance is: " << (double)balance << std::endl;
}
```

Здесь оператор функции преобразования `Double` был явно объявлен, а явное приведение к типу `double` было представлено в функции `display_balance` для выполнения преобразования. Если пропустить это преобразование, компилятор не сможет найти подходящий оператор вставки в поток `<<` для типа `Money` и может возникнуть ошибка.

Изменяемые члены данных (C++)

12.11.2021 • 2 minutes to read

Это ключевое слово может применяться только к данным-членам класса, которые не являются статическими или константами. Если объявлен элемент данных `mutable`, ему допустимо присвоить значение этому элементу данных из `const` функции-члена.

Синтаксис

```
mutable member-variable-declaration;
```

Remarks

Например, следующий код будет компилироваться без ошибок, поскольку `m_accessCount` объявлен как `mutable`, и поэтому может быть изменен, `GetFlag` даже если `GetFlag` является константной функцией-членом.

```
// mutable.cpp
class X
{
public:
    bool GetFlag() const
    {
        m_accessCount++;
        return m_flag;
    }
private:
    bool m_flag;
    mutable int m_accessCount;
};

int main()
{}
```

См. также

[Ключевые слова](#)

Объявления вложенных классов

12.11.2021 • 3 minutes to read

Класс можно объявить в области другого класса. Такой класс называется вложенным классом. Считается, что вложенные классы находятся в области включающего класса и доступны для использования внутри этой области. Для обращения ко вложенному классу из области, отличной от непосредственно включающей его области, следует использовать полное имя.

В следующем примере показано, как объявить вложенные классы.

```
// nested_class_declarations.cpp
class BufferedIO
{
public:
    enum IOError { None, Access, General };

    // Declare nested class BufferedInput.
    class BufferedInput
    {
    public:
        int read();
        int good()
        {
            return _inputerror == None;
        }
    private:
        IOError _inputerror;
    };
};

// Declare nested class BufferedOutput.
class BufferedOutput
{
    // Member list
};
};

int main()
{}
```

`BufferedIO::BufferedInput` И `BufferedIO::BufferedOutput` объявляются в `BufferedIO`. Эти имена классов не видимы за пределами области класса `BufferedIO`. Однако объект типа `BufferedIO` не содержит объекты типа `BufferedInput` ИЛИ `BufferedOutput`.

Вложенные классы могут непосредственно использовать имена, имена типов, имена статических членов и перечислители только из включающего класса. Для использования имен других членов класса необходимо использовать указатели, ссылки или имена объектов.

В предыдущем примере `BufferedIO` к перечислению `IOError` можно получить доступ непосредственно с помощью функций-членов во вложенных классах `BufferedIO::BufferedInput` ИЛИ `BufferedIO::BufferedOutput`, как показано в функции `good`.

NOTE

Вложенные классы объявляют только типы в пределах области класса. Они не создают объекты по вложенному классу. В предыдущем примере объявляется два вложенных класса, но не создаются объекты этих типов классов.

Исключением из видимости области объявления вложенного класса является объявление имени типа вместе с опережающим объявлением. В этом случае имя класса, объявленное с помощью опережающего объявления, видимо за пределами включающего класса, при этом область определена как наименьшая включающая область вне класса. Пример:

```
// nested_class_declarations_2.cpp
class C
{
public:
    typedef class U u_t; // class U visible outside class C scope
    typedef class V {} v_t; // class V not visible outside class C
};

int main()
{
    // okay, forward declaration used above so file scope is used
    U* pu;

    // error, type name only exists in class C scope
    u_t* pu2; // C2065

    // error, class defined above so class C scope
    V* pv; // C2065

    // okay, fully qualified name
    C::V* pv2;
}
```

Права доступа во вложенных классах

Вложение класса в другой класс не предоставляет особых прав доступа к функциям-членам вложенного класса. Аналогичным образом, функции-члены включающего класса не имеют особых прав доступа к членам вложенного класса.

Функции-члены во вложенных классах

Функции-члены, объявленные во вложенных классах, могут быть определены в области файла. Предыдущий пример можно было бы записать следующим образом:

```

// member_functions_in_nested_classes.cpp
class BufferedIO
{
public:
    enum IOError { None, Access, General };
    class BufferedInput
    {
    public:
        int read(); // Declare but do not define member
        int good(); // functions read and good.
    private:
        IOError _inputerror;
    };

    class BufferedOutput
    {
        // Member list.
    };
};

// Define member functions read and good in
// file scope.
int BufferedIO::BufferedInput::read()
{
    return(1);
}

int BufferedIO::BufferedInput::good()
{
    return _inputerror == None;
}

int main()
{
}

```

В предыдущем примере синтаксис *квалифицированного типа* используется для объявления имени функции. Объявление:

```
BufferedIO::BufferedInput::read()
```

означает "функция `read`", которая является членом класса `BufferedInput`, который находится в области класса `BufferedIO`". Поскольку в этом объявлении используется синтаксис *квалифицированного типа*, возможны конструкции следующей формы:

```

typedef BufferedIO::BufferedInput BIO_INPUT;

int BIO_INPUT::read()

```

Предыдущее объявление эквивалентно предыдущему, но вместо `typedef` имен классов используется имя.

Дружественные функции во вложенных классах

Считается, что дружественные функции, объявленные во вложенном классе, находятся в области вложенного, а не включающего класса. Поэтому дружественные функции не получают особых прав доступа к членам или функциям-членам включающего класса. Если требуется использовать имя, объявленное во вложенном классе, в дружественной функции, и дружественная функция определена в области видимости файла, используйте полные имена типов, как показано ниже.

```

// friend_functions_and_nested_classes.cpp

#include <string.h>

enum
{
    sizeOfMessage = 255
};

char *rgszMessage[sizeOfMessage];

class BufferedIO
{
public:
    class BufferedInput
    {
public:
        friend int GetExtendedErrorStatus();
        static char *message;
        static int messageSize;
        int iMsgNo;
    };
};

char *BufferedIO::BufferedInput::message;
int BufferedIO::BufferedInput::messageSize;

int GetExtendedErrorStatus()
{
    int iMsgNo = 1; // assign arbitrary value as message number

    strcpy_s( BufferedIO::BufferedInput::message,
              BufferedIO::BufferedInput::messageSize,
              rgszMessage[ iMsgNo ] );

    return iMsgNo;
}

int main()
{
}

```

В следующем коде показана функция `GetExtendedErrorStatus`, объявленная в качестве дружественной функции. В функции, определенной в области видимости файла, сообщение копируется из статического массива в член класса. Обратите внимание, что для оптимальной реализации функции `GetExtendedErrorStatus` рекомендуется объявить ее следующим образом.

```
int GetExtendedErrorStatus( char *message )
```

В предыдущем интерфейсе несколько классов могут использовать службы этой функции, передав адрес памяти, в которую требуется скопировать сообщение об ошибке.

См. также

[Классы и структуры](#)

Типы анонимных классов

12.11.2021 • 2 minutes to read

Классы могут быть анонимными, то есть их можно объявлять без *идентификатора*. Это полезно при замене имени класса `typedef` именем, как показано ниже:

```
typedef struct
{
    unsigned x;
    unsigned y;
} POINT;
```

NOTE

Использование анонимных классов, показанное в предыдущем примере, полезно для поддержки совместимости с существующими кодом C. В некоторых языках кода на языке C `typedef` распространено использование совместно с анонимными структурами.

Анонимные классы также полезны, если требуется, чтобы ссылка на член класса отображалась так, как если бы она не содержалась в отдельном классе, как показано в следующем примере.

```
struct PTValue
{
    POINT ptLoc;
    union
    {
        int iValue;
        long lValue;
    };
};

PTValue ptv;
```

В приведенном выше коде `iValue` можно получить доступ с помощью оператора выбора члена объекта `(.)` следующим образом:

```
int i = ptv.iValue;
```

Анонимные классы имеют некоторые ограничения. (Дополнительные сведения о анонимных объединениях см. в разделе [объединения](#).) Анонимные классы:

- Не могут иметь конструктор или деструктор.
- Не может передаваться в качестве аргументов функциям (если только проверка типа не будет отменяться с помощью многоточия).
- Не могут быть возвращены в качестве возвращаемых значений из функций.

Анонимные структуры

Блок, относящийся только к системам Microsoft

Расширение Microsoft C позволяет объявить переменную структуры в другой структуре без указания ее имени. Эти вложенные структуры называются анонимными структурами. В C++ анонимные структуры не допускаются.

Можно получить доступ к членам анонимной структуры, как если бы они были членами содержащей их структуры.

```
// anonymous_structures.c
#include <stdio.h>

struct phone
{
    int areacode;
    long number;
};

struct person
{
    char name[30];
    char gender;
    int age;
    int weight;
    struct phone; // Anonymous structure; no name needed
} Jim;

int main()
{
    Jim.number = 1234567;
    printf_s("%d\n", Jim.number);
}
//Output: 1234567
```

Завершение блока, относящегося только к системам Майкрософт

Указатели на члены

12.11.2021 • 3 minutes to read

Объявления указателей на члены — это особый случай объявлений указателей. Они объявляются с помощью следующей последовательности:

спецификаторы класса хранения_{необязательно} — Квалификаторы не заменяют спецификаторы типа "описатель" квалификатора MS-модификаторов опт :: ;*

1. Спецификатор объявления:

- Необязательный спецификатор класса хранения.
- Необязательные `const` `volatile` спецификаторы и.
- Спецификатор типа: имя типа. Это тип элемента, на который указывает, а не класс.

2. Декларатор:

- Необязательный модификатор, используемый в системах Microsoft. Дополнительные сведения см. в разделе [модификаторы, зависящие от Майкрософта](#).
- Полное имя класса, содержащего члены, на которые должен указывать указатель.
- `::` Оператор.
- `*` Оператор.
- Необязательные `const` `volatile` спецификаторы и.
- Идентификатор, задающий имя указателя на член.

3. Необязательный инициализатор указателя на член:

- `=` Оператор.
- `&` Оператор.
- Полное имя класса.
- `::` Оператор.
- Имя не являющегося статическим члена класса соответствующего типа.

Как обычно, в одном объявлении допускается несколько деклараторов (и любые связанные инициализаторы). Указатель на член может не указывать на статический член класса, член ссылочного типа или `void`.

Указатель на член класса отличается от обычного указателя: он имеет как информацию о типе для типа элемента, так и для класса, которому принадлежит элемент. Обычный указатель идентифицирует только один объект в памяти (содержит адрес этого объекта). Указатель на член класса идентифицирует этот член в любом экземпляре класса. В следующем примере объявляется класс `Window` и несколько указателей на данные-член.

```

// pointers_to_members1.cpp
class Window
{
public:
    Window();                                // Default constructor.
    Window( int x1, int y1,                  // Constructor specifying
            int x2, int y2 );                // Window size.
    bool SetCaption( const char *szTitle ); // Set window caption.
    const char *GetCaption();               // Get window caption.
    char *szWinCaption;                    // Window caption.
};

// Declare a pointer to the data member szWinCaption.
char * Window::* pwCaption = &Window::szWinCaption;
int main()
{
}

```

В предыдущем примере `pwCaption` — это указатель на любой член класса `Window`, который имеет тип `char*`. `pwCaption` имеет тип `char * Window::*`. В следующем фрагменте кода объявляются указатели на функции-члены `SetCaption` и `GetCaption`.

```

const char * (Window::* pfnwGC)() = &Window::GetCaption;
bool (Window::* pfnwSC)( const char * ) = &Window::SetCaption;

```

Указатели `pfnwGC` и `pfnwSC` указывают на функции `GetCaption` и `SetCaption` класса `Window` соответственно. Код копирует информацию непосредственно в заголовок окна с помощью указателя на член `pwCaption`:

```

Window wMainWindow;
Window *pwChildWindow = new Window;
char *szUntitled = "Untitled - ";
int cUntitledLen = strlen( szUntitled );

strcpy_s( wMainWindow.*pwCaption, cUntitledLen, szUntitled );
(wMainWindow.*pwCaption)[cUntitledLen - 1] = '1'; // same as
// wMainWindow.SzWinCaption [cUntitledLen - 1] = '1';
strcpy_s( pwChildWindow->*pwCaption, cUntitledLen, szUntitled );
(pwChildWindow->*pwCaption)[cUntitledLen - 1] = '2'; // same as
// pwChildWindow->szWinCaption[cUntitledLen - 1] = '2';

```

Разница между `.*` и `->*` операторами (операторами указателей на члены) заключается в том, что `.*` оператор выбирает члены по ссылке на объект или объект, а `->*` оператор выбирает члены с помощью указателя. Дополнительные сведения об этих операторах см. [в разделе выражения с операторами указателей на члены](#).

Результатом операторов указателя на член является тип элемента. В этом случае он выглядит так:

`char *`.

В следующем фрагменте кода функции-члены `GetCaption` и `SetCaption` вызываются с использованием указателей на члены.

```
// Allocate a buffer.
enum {
    sizeOfBuffer = 100
};
char szCaptionBase[sizeOfBuffer];

// Copy the main window caption into the buffer
// and append " [View 1]".
strcpy_s( szCaptionBase, sizeOfBuffer, (wMainWindow.*pfnwGC)() );
strcat_s( szCaptionBase, sizeOfBuffer, " [View 1]" );
// Set the child window's caption.
(pwChildWindow->*pfnwSC)( szCaptionBase );
```

Ограничения указателей на члены

Адрес статического элемента не является указателем на элемент. Это обычный указатель на один экземпляр статического элемента. Для всех объектов данного класса существует только один экземпляр статического элемента. Это означает, что можно использовать обычные операторы address-of (&) и разыменование (*).

Указатели на члены и виртуальные функции

Вызов виртуальной функции с помощью функции указателя на член работает так же, как если бы функция вызывалась напрямую. Правильная функция ищется в таблице v-table и вызывается.

Ключ для виртуальных функций, работающих как обычно, вызывает их через указатель на базовый класс. (Дополнительные сведения о виртуальных функциях см. в разделе [виртуальные функции](#).)

В следующем коде показан вызов виртуальной функции через функцию указателя на член.

```
// virtual_functions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class Base
{
public:
    virtual void Print();
};

void (Base::* bfnPrint)() = &Base::Print;
void Base::Print()
{
    cout << "Print function for class Base" << endl;
}

class Derived : public Base
{
public:
    void Print(); // Print is still a virtual function.
};

void Derived::Print()
{
    cout << "Print function for class Derived" << endl;
}

int main()
{
    Base *bPtr;
    Base bObject;
    Derived dObject;
    bPtr = &bObject; // Set pointer to address of bObject.
    (bPtr->*bfmPrint)();
    bPtr = &dObject; // Set pointer to address of dObject.
    (bPtr->*bfmPrint)();
}

// Output:
// Print function for class Base
// Print function for class Derived
```

Указатель this

12.11.2021 • 3 minutes to read

`this` Указатель является указателем, доступным только в нестатических функциях-членах `class` `struct` типа, или `union`. Он указывает на объект, для которого вызывается функция-член. Статические функции-члены не имеют `this` указателя.

Синтаксис

```
this  
this->member-identifier
```

Remarks

`this` Указатель объекта не является частью самого объекта. Он не отражается в результате выполнения `sizeof` инструкции для объекта. При вызове нестатической функции-члена для объекта компилятор передает адрес объекта в функцию в виде скрытого аргумента. Например, при вызове следующей функции

```
myDate.setMonth( 3 );
```

может интерпретироваться как:

```
setMonth( &myDate, 3 );
```

Адрес объекта доступен в функции-члене в качестве `this` указателя. Большинство `this` используемых указателей являются неявными. Он является юридическим, хотя и ненужным, для использования явной `this` ссылки на члены `class`. Пример:

```
void Date::setMonth( int mn )  
{  
    month = mn;           // These three statements  
    this->month = mn;     // are equivalent  
    (*this).month = mn;  
}
```

Выражение `*this` обычно используется для возврата текущего объекта из функции-члена:

```
return *this;
```

`this` Указатель также используется для защиты от самостоятельной ссылки:

```
if (&Object != this) {  
// do not execute in cases of self-reference
```

NOTE

Поскольку `this` указатель не является изменяемым, присваивания `this` указателю не допускаются. Более ранние реализации C++ позволяли назначать `this`.

Иногда `this` указатель используется напрямую, например для управления самостоятельными ссылочными данными struct Урес, где требуется адрес текущего объекта.

Пример

```

// this_pointer.cpp
// compile with: /EHsc

#include <iostream>
#include <string.h>

using namespace std;

class Buf
{
public:
    Buf( char* szBuffer, size_t sizeOfBuffer );
    Buf& operator=( const Buf & );
    void Display() { cout << buffer << endl; }

private:
    char*   buffer;
    size_t   sizeOfBuffer;
};

Buf::Buf( char* szBuffer, size_t sizeOfBuffer )
{
    sizeOfBuffer++; // account for a NULL terminator

    buffer = new char[ sizeOfBuffer ];
    if (buffer)
    {
        strcpy_s( buffer, sizeOfBuffer, szBuffer );
        sizeOfBuffer = sizeOfBuffer;
    }
}

Buf& Buf::operator=( const Buf &otherbuf )
{
    if( &otherbuf != this )
    {
        if (buffer)
            delete [] buffer;

        sizeOfBuffer = strlen( otherbuf.buffer ) + 1;
        buffer = new char[sizeOfBuffer];
        strcpy_s( buffer, sizeOfBuffer, otherbuf.buffer );
    }
    return *this;
}

int main()
{
    Buf myBuf( "my buffer", 10 );
    Buf yourBuf( "your buffer", 12 );

    // Display 'my buffer'
    myBuf.Display();

    // assignment operator
    myBuf = yourBuf;

    // Display 'your buffer'
    myBuf.Display();
}

```

```

my buffer
your buffer

```

Тип this указателя

`this` Тип указателя можно изменить в объявлении функции с помощью `const` `volatile` ключевых слов и. Чтобы объявить функцию, имеющую один из этих атрибутов, добавьте ключевые слова после списка аргументов функции.

Рассмотрим пример.

```
// type_of_this_pointer1.cpp
class Point
{
    unsigned X() const;
};
int main()
{
}
```

В приведенном выше коде объявляется функция-член, `x` в которой `this` указатель обрабатывается как `const` указатель на `const` объект. Можно использовать сочетания параметров *ОПС-mod-list*, но они всегда изменяют объект, на который указывает `this` указатель, а не сам указатель. В следующем объявлении объявляется функция `x`, в которой `this` указатель является `const` указателем на `const` объект:

```
// type_of_this_pointer2.cpp
class Point
{
    unsigned X() const;
};
int main()
{
}
```

Тип `this` в функции-члене описан в следующем синтаксисе. Список «*ОПС-квалификатор*» определяется на основе декларатора функции-члена. Это может быть `const` или `volatile` (или). *class -Type* — это имя *class*:

[*ОПС-квалификатор-List*] *class -тип** `const` `this`

Иными словами, `this` указатель всегда является `const` указателем. Его нельзя назначить. `const` `volatile` Квалификаторы или, используемые в объявлении функции члена, применяются к *class* экземпляру, на `this` который указывает указатель в области этой функции.

В следующей таблице приведены дополнительные сведения о том, как работают эти модификаторы.

Семантика this модификаторов

МОДИФИКАТОР	ЗНАЧЕНИЕ
<code>const</code>	Не удается изменить данные элемента; не удается вызвать функции члена, которые не являются <code>const</code> .
<code>volatile</code>	Данные элементов загружаются из памяти при каждом доступе к ним; отключает некоторые оптимизации.

Передача `const` объекта в функцию-член, которая не имеет значение, является ошибкой `const`.

Аналогично, также возникает ошибка передачи `volatile` объекта в функцию-член, которая не является

`volatile` .

Функции-члены, объявленные как `const` не могут изменять данные элементов — в таких функциях `this` указатель является указателем на `const` объект.

NOTE

Con struct OR и de struct or нельзя объявлять как `const` или `volatile` . Однако они могут вызываться для `const` `volatile` объектов или.

См. также

[Ключевые слова](#)

Битовые поля в C++

12.11.2021 • 2 minutes to read

Классы и структуры могут содержать члены, которые занимают меньше пространства в памяти, чем целочисленный тип. Эти члены определяются как битовые поля. Синтаксис для спецификации *объявления члена* в битовом поле приведен ниже:

Синтаксис

декларатор: константное выражение

Remarks

Декларатор (необязательный) — это имя, по которому осуществляется доступ к элементу в программе. Он должен иметь один из целочисленных типов (включая перечисляемые типы). *Константа-выражение* указывает количество битов, занимаемых элементом в структуре. Анонимные битовые поля, (т. е. битовые поля без идентификатора) можно использовать для заполнения.

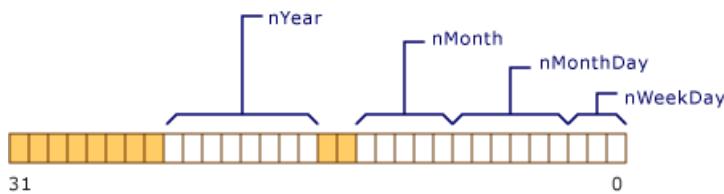
NOTE

Неименованное битовое поле Width 0 приводит к выравниванию следующего битового поля до границы следующего **типа**, где Type — это тип элемента.

В следующем примере объявляется структура, которая содержит битовые поля:

```
// bit_fields1.cpp
// compile with: /LD
struct Date {
    unsigned short nYear      : 3;      // 0..7   (3 bits)
    unsigned short nMonthDay : 6;      // 0..31  (6 bits)
    unsigned short nMonth    : 5;      // 0..12  (5 bits)
    unsigned short nYear      : 8;      // 0..100 (8 bits)
};
```

На следующем рисунке показана концептуальная структура памяти для объекта типа `Date`.



Структура памяти объекта типа Date

Обратите внимание, что `nYear` имеет длину 8 бит и приведет к переполнению границы слова объявленного типа `unsigned short`. Поэтому он начинается в начале нового `unsigned short`. Совсем не обязательно, чтобы все битовые поля помещались в один объект базового типа; в зависимости от количества бит, запрошенных в объявлении, выделяются новые единицы хранения.

Блок, относящийся только к системам Microsoft

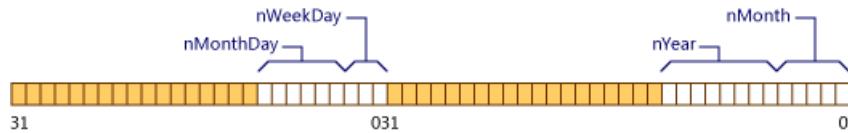
Данные, объявленные в качестве битовых полей, упорядочиваются от младшего бита к старшему, как показано на рисунке выше.

Завершение блока, относящегося только к системам Майкрософт

Объявление структуры может содержать неименованное поле длиной 0, как показано в следующем примере.

```
// bit_fields2.cpp
// compile with: /LD
struct Date {
    unsigned nWeekDay : 3;      // 0..7  (3 bits)
    unsigned nMonthDay : 6;     // 0..31 (6 bits)
    unsigned : 0;               // Force alignment to next boundary.
    unsigned nMonth : 5;        // 0..12 (5 bits)
    unsigned nYear : 8;         // 0..100 (8 bits)
};
```

затем макет памяти показан на следующем рисунке:



Структура объекта типа Date с битовым полем нулевой длины

Базовый тип битового поля должен быть целочисленным типом, как описано в разделе [встроенные типы](#).

Если инициализатор для ссылки типа `const T&` является левосторонним значением, указывающим на битовое поле типа `T`, ссылка не привязывается к битовому полю напрямую. Вместо этого ссылка привязывается к временной инициализированной переменной для хранения значения битового поля.

Ограничения для битовых полей

В следующем списке указаны ошибочные операции с битовыми полями:

- Получение адреса битового поля.
- Инициализация не `const` ссылки с битовым полем.

См. также

[Классы и структуры](#)

Лямбда-выражения в C++

12.11.2021 • 10 minutes to read

В C++ 11 и более поздних версиях лямбда-выражение, часто называемое *лямбда* — это удобный способ определения объекта анонимной функции (замыкания) непосредственно в расположении, где оно вызывается или передается в качестве аргумента функции. Обычно лямбда-выражения используются для инкапсуляции нескольких строк кода, передаваемых алгоритмам или асинхронным функциям. В этой статье определяются лямбда-выражения и их сравнение с другими методами программирования. Он описывает их преимущества и предоставляет некоторые основные примеры.

Похожие статьи

- [Лямбда-выражения и объекты функций](#)
- [Работа с лямбда-выражениями](#)
- [Лямбда-выражения constexpr](#)

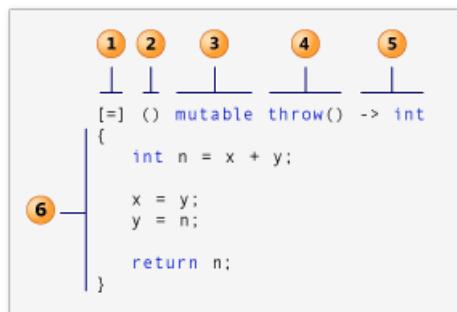
Части лямбда-выражения

В стандарте ISO C++ демонстрируется простое лямбда-выражение, передаваемое функции `std::sort()` в качестве третьего аргумента:

```
#include <algorithm>
#include <cmath>

void abssort(float* x, unsigned n) {
    std::sort(x, x + n,
              // Lambda expression begins
              [] (float a, float b) {
                  return (std::abs(a) < std::abs(b));
              } // end of lambda expression
    );
}
```

На следующем рисунке показана структура лямбда-выражения:



1. *предложение Capture* (также известное как оператор *лямбда-выражения* в спецификации C++).
2. *список параметров Используемых*. (Также называется *лямбда-объявлением*)
3. *изменяемая спецификация Используемых*.
4. *Спецификация Exception Используемых*.
5. *замыкающий-возвращаемый тип Используемых*.

6. тело лямбда-выражения.

Предложение Capture

Лямбда-выражение может добавлять новые переменные в тексте (в C++ 14), а также получать доступ к переменным из окружающей области или записывать их. Лямбда-выражение начинается с предложения Capture. Он указывает, какие переменные фиксируются, а также указывает, является ли запись по значению или по ссылке. Доступ к переменным с префиксом амперсанда (`&`) осуществляется по ссылке и к переменным, к которым нет доступа по значению.

Пустое предложение фиксации (`[]`) показывает, что тело лямбда-выражения не осуществляет доступ к переменным во внешней области видимости.

Можно использовать режим захвата по умолчанию, чтобы указать, как фиксировать все внешние переменные, упоминаемые в теле лямбда-выражения: означает, что `[&]` все переменные, на которые вы ссылаетесь, захватываются по ссылке, а `[=]` значит, они записываются по значению. Можно сначала использовать режим фиксации по умолчанию, а затем применить для определенных переменных другой режим. Например, если тело лямбда-выражения осуществляет доступ к внешней переменной `total` по ссылке, а к внешней переменной `factor` по значению, следующие предложения фиксации эквивалентны:

```
[&total, factor]
[factor, &total]
[&, factor]
[factor, &]
[=, &total]
[&total, =]
```

При использовании записи по умолчанию фиксируются только те переменные, которые упоминаются в теле лямбда-выражения.

Если предложение Capture включает запись-Default, то `&` ни один идентификатор в записи этого предложения записи не может иметь форму `&identifier`. Аналогично, если предложение Capture включает запись по умолчанию `=`, то ни один из этих предложений не может иметь форму `=identifier`. Идентификатор или `this` не может использоваться в предложении Capture более одного раза. В следующем фрагменте кода показаны некоторые примеры.

```
struct S { void f(int i); };

void S::f(int i) {
    [&, i]{};           // OK
    [&, &i]{};          // ERROR: i preceded by & when & is the default
    [=, this]{};         // ERROR: this when = is the default
    [=, *this]{};        // OK: captures this by value. See below.
    [i, i]{};           // ERROR: i repeated
}
```

Захват, за которым следует многоточие, — это расширение пакета, как показано в следующем примере шаблона [Variadic](#):

```
template<class... Args>
void f(Args... args) {
    auto x = [args...] { return g(args...); };
    x();
}
```

Чтобы использовать лямбда-выражения в теле функции члена класса, передайте `this` указатель в предложение Capture, чтобы предоставить доступ к функциям и членам данных включающего класса.

Visual Studio 2017 **версии 15,3 и более поздних** версий (доступно в [/std: c++ 17](#)): `this` указатель может быть захвачен значением путем указания `*this` в предложении capture. Захват по значению копирует весь замыкание на каждый узел вызова, где вызывается лямбда-выражение. (Замыканием является объект анонимной функции, инкапсулирующий лямбда-выражение.) Захват по значению полезен, когда лямбда выполняется в параллельных или асинхронных операциях. Это особенно полезно на некоторых аппаратных архитектурах, таких как NUMA.

Пример, демонстрирующий использование лямбда-выражений с функциями членов класса, см. в разделе "пример: использование лямбда-выражения в методе" в [примерах лямбда-выражений](#).

При использовании предложения Capture рекомендуется учитывать такие моменты, особенно при использовании лямбда-выражений с многопоточностью:

- Захваты ссылок можно использовать для изменения переменных вне, но захваты значений не могут. (`mutable` позволяет изменять копии, но не оригиналы.)
- Захват ссылок отражает обновления переменных вне, но не фиксирует значения.
- Фиксация ссылки вводит зависимость от времени существования, тогда как фиксация значения не обладает зависимостями от времени существования. Это особенно важно при асинхронном выполнении лямбда-выражения. Если вы захватываете локальную по ссылке в асинхронном лямбда-выражении, это локально может быть легко пропала в момент выполнения лямбда-выражения. Код может вызвать нарушение прав доступа во время выполнения.

Обобщенная фиксация (C++14)

В C++14 вы можете объявлять и инициализировать новые переменные в предложении фиксации. Для этого не требуется, чтобы эти переменные существовали во внешней области видимости лямбда-функции. Инициализация может быть выражена в качестве любого произвольного выражения. Тип новой переменной определяется типом, который создается выражением. Эта функция позволяет собирать переменные только для перемещения (например, `std::unique_ptr`) из окружающей области и использовать их в лямбда-выражении.

```
pNums = make_unique<vector<int>>(nums);
//...
auto a = [ptr = move(pNums)]()
{
    // use ptr
};
```

Список параметров

Лямбда-выражения могут записывать переменные и принимать входные параметры. Список параметров (лямбда-декларатор в стандартном синтаксисе) является необязательным и в большинстве аспектов напоминает список параметров для функции.

```
auto y = [] (int first, int second)
{
    return first + second;
};
```

В C++ 14, если тип параметра является универсальным, можно использовать `auto` ключевое слово в качестве спецификатора типа. Это ключевое слово указывает компилятору создать оператор вызова функции в качестве шаблона. Каждый экземпляр `auto` в списке параметров эквивалентен отдельному параметру типа.

```
auto y = [] (auto first, auto second)
{
    return first + second;
};
```

Лямбда-выражение может принимать другое лямбда-выражение в качестве своего аргумента.

Дополнительные сведения см. в разделе «Лямбда-выражения более высокого порядка» в статье [Примеры лямбда-выражений](#).

Поскольку список параметров является необязательным, можно опустить пустые скобки, если аргументы не передаются в лямбда-выражение и его лямбда-декларатор не содержит спецификацию *Exception*, завершающего-*Return-Type* или `mutable`.

Изменяемая спецификация

Как правило, оператор вызова функции лямбда-выражения является константой по значению, но использование `mutable` ключевого слова отменяет это. Он не создает изменяемых элементов данных. `mutable` Спецификация позволяет тексту лямбда-выражения изменять переменные, захваченные по значению. В некоторых примерах, приведенных далее в этой статье, показано, как использовать `mutable`.

Спецификация исключений

Можно использовать `noexcept` спецификацию исключения, чтобы указать, что лямбда-выражение не создает никаких исключений. Как и в случае с обычными функциями, компилятор Microsoft C++ создает предупреждение [C4297](#), если лямбда-выражение объявляет `noexcept` спецификацию исключения, а тело лямбда-выражения создает исключение, как показано ниже:

```
// throw_lambda_expression.cpp
// compile with: /W4 /EHsc
int main() // C4297 expected
{
    []() noexcept { throw 5; }();
}
```

Дополнительные сведения см. в разделе [спецификации исключений \(throw\)](#).

Возвращаемый тип

Возвращаемый тип лямбда-выражения выводится автоматически. Не обязательно использовать `auto` ключевое слово, если не указан *завершающий возвращаемый тип*. *Замыкающий возвращаемый тип* напоминает часть функции, возвращающей возвращаемый тип, и функцию-член. Однако возвращаемый тип должен следовать за списком параметров `->` перед возвращаемым типом необходимо включить ключевое слово *замыкающего возвращаемого типа*.

Можно опустить часть возвращаемого типа лямбда-выражения, если тело лямбды содержит только один оператор `return`. Или, если выражение не возвращает значение. Если тело лямбда-выражения содержит один оператор `return`, компилятор выводит тип возвращаемого значения из типа возвращаемого выражения. В противном случае компилятор выводит тип возвращаемого значения как `void`.

Рассмотрим следующие примеры фрагментов кода, иллюстрирующих этот принцип:

```
auto x1 = [](int i){ return i; }; // OK: return type is int
auto x2 = []{ return{ 1, 2 }; }; // ERROR: return type is void, deducing
                                // return type from braced-init-list isn't valid
```

Лямбда-выражение может создавать другое лямбда-выражение в качестве своего возвращаемого значения. Дополнительные сведения см. в разделе "лямбда-выражения более высокого порядка" в

примерах лямбда-выражений.

Тело лямбды

Тело лямбда-выражения является составным оператором. Он может содержать все, что разрешено в теле обычной функции или функции-члена. Тело обычной функции и лямбда-выражения может осуществлять доступ к следующим типам переменных:

- Фиксированные переменные из внешней области видимости (см. выше).
- Параметры.
- Локально объявленные переменные.
- Члены данных класса, объявленные внутри класса и `this` захваченные.
- Любая переменная, имеющая статическую длительность хранения, например глобальные переменные.

В следующем примере содержится лямбда-выражение, которое явно фиксирует переменную `n` по значению и неявно фиксирует переменную `m` по ссылке.

```
// captures_lambda_expression.cpp
// compile with: /W4 /EHsc
#include <iostream>
using namespace std;

int main()
{
    int m = 0;
    int n = 0;
    [&, n] (int a) mutable { m = ++n + a; }(4);
    cout << m << endl << n << endl;
}
```

```
5
0
```

Поскольку переменная `n` фиксируется по значению, ее значение после вызова лямбда-выражения остается равным `0`. Спецификацию можно `mutable` изменить в лямбда-выражении.

Лямбда-выражение может записывать только переменные с автоматическим длительностью хранения. Однако можно использовать переменные со статической длительностью хранения в теле лямбда-выражения. В следующем примере функция `generate` и лямбда-выражение используются для присвоения значения каждому элементу объекта `vector`. Лямбда-выражение изменяет статическую переменную для получения значения следующего элемента.

```
void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this isn't thread-safe and is shown for illustration only
}
```

Дополнительные сведения см. в разделе [Generate](#).

В следующем примере кода используется функция из предыдущего примера и добавляется пример лямбда-выражения, использующего алгоритм стандартной библиотеки C++ `generate_n`. Это лямбда-выражение назначает элемент объекта `vector` сумме предыдущих двух элементов. Ключевое слово `mutable` используется, чтобы тело лямбда-выражения может изменить свои копии внешних переменных `x` и `y`, которое захватывает лямбда-выражение по значению. Поскольку лямбда-выражение захватывает исходные переменные `x` и `y` по значению, их значения остаются равными `1` после выполнения лямбда-выражения.

```
// compile with: /W4 /EHsc
#include <algorithm>
#include <iostream>
#include <vector>
#include <string>

using namespace std;

template <typename C> void print(const string& s, const C& c) {
    cout << s;

    for (const auto& e : c) {
        cout << e << " ";
    }

    cout << endl;
}

void fillVector(vector<int>& v)
{
    // A local static variable.
    static int nextValue = 1;

    // The lambda expression that appears in the following call to
    // the generate function modifies and uses the local static
    // variable nextValue.
    generate(v.begin(), v.end(), [] { return nextValue++; });
    //WARNING: this isn't thread-safe and is shown for illustration only
}

int main()
{
    // The number of elements in the vector.
    const int elementCount = 9;

    // Create a vector object with each element set to 1.
    vector<int> v(elementCount, 1);

    // These variables hold the previous two elements of the vector.
    int x = 1;
    int y = 1;

    // Sets each element in the vector to the sum of the
    // previous two elements.
    generate_n(v.begin() + 2,
               elementCount - 2,
               [=]() mutable throw() -> int { // lambda is the 3rd parameter
                   // Generate current value.
                   int n = x + y;
                   // Update previous two values.
                   x = y;
                   y = n;
                   return n;
               });
    print("vector v after call to generate_n() with lambda: ", v);

    // Print the local variables x and y.
```

```

// The values of x and y hold their initial values because
// they are captured by value.
cout << "x: " << x << " y: " << y << endl;

// Fill the vector with a sequence of numbers
fillVector(v);
print("vector v after 1st call to fillVector(): ", v);
// Fill the vector with the next sequence of numbers
fillVector(v);
print("vector v after 2nd call to fillVector(): ", v);
}

```

```

vector v after call to generate_n() with lambda: 1 1 2 3 5 8 13 21 34
x: 1 y: 1
vector v after 1st call to fillVector(): 1 2 3 4 5 6 7 8 9
vector v after 2nd call to fillVector(): 10 11 12 13 14 15 16 17 18

```

Дополнительные сведения см. в разделе [generate_n](#).

constexpr лямбда-выражения

Visual Studio 2017 версии 15,3 и более поздних версий (доступно в [/std:c++17](#)): лямбда-выражение можно объявить как `constexpr` (или использовать его в константном выражении), если инициализация каждого захваченного или введенного элемента данных разрешена в константном выражении.

```

int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}

```

Лямбда-выражение неявно, `constexpr` если его результат удовлетворяет требованиям `constexpr` функции:

```

auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);

```

Если лямбда-выражение неявно или неявное `constexpr`, то преобразование в указатель функции создает `constexpr` функцию:

```

auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;

```

Специально для систем Майкрософт

Лямбда-выражения не поддерживаются в следующих управляемых сущностях среды CLR: `ref class`, `ref struct`, `value class` ИЛИ `value struct`.

Если вы используете модификатор, зависящий от Майкрософт `_declspec`, например, его можно вставить в лямбда-выражение сразу после `parameter-declaration-clause`. Пример:

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t; };
```

Чтобы определить, поддерживается ли определенный модификатор лямбда-выражениями, см. статью об модификаторе в разделе [модификаторы, относящиеся к Microsoft](#).

Visual Studio поддерживает стандартную лямбда-функцию C++ 11 и *лямбда-выражения без отслеживания состояния*. Лямбда без отслеживания состояния преобразуется в указатель функции, который использует произвольное соглашение о вызовах.

См. также

[Справочник по языку C++](#)

[Объекты функций в стандартной библиотеке C++](#)

[Вызов функции](#)

[for_each](#)

Синтаксис лямбда-выражений

12.11.2021 • 3 minutes to read

В этой статье демонстрируется синтаксис и структурные элементы лямбда-выражений. Описание лямбда-выражений см. в разделе [лямбда-выражения](#).

Объекты функций и лямбда-выражения

При написании кода вы, вероятно, используете указатели функций и объекты функций для решения проблем и выполнения вычислений, особенно при использовании [алгоритмов стандартной библиотеки C++](#). У объектов-функций и указателей на функции есть как преимущества, так и недостатки. Например, указатели на функции отличаются минимальными требованиями к синтаксису, но не сохраняют состояние в области видимости. Объекты-функции, в свою очередь, могут сохранять состояние, но требуют дополнительного синтаксиса в определении класса.

Лямбда-выражение сочетает преимущества указателей на функции и объектов-функций, избегая их недостатков. Как и объект функции, лямбда-выражение является гибким и может поддерживать состояние, но в отличие от объекта функции, его синтаксис Compact не требует явного определения класса. С помощью лямбда-выражений можно написать код, который более простым и менее подверженным появлению ошибок, чем код для соответствующего объекта функции.

В следующих примерах сравнивается использование лямбда-выражения и объекта функции. В первом примере лямбда-выражение используется для вывода на консоль независимо от того, четным или нечетным является каждый элемент в объекте `vector`. Во втором примере для выполнения той же задачи используется объект функции.

Пример 1: использование лямбда-выражения

В этом примере лямбда-выражение передается в функцию `for_each`. Лямбда-выражение выводит сообщение о четности или нечетности каждого из элементов объекта `vector`.

Код

```

// even_lambda.cpp
// compile with: cl /EHsc /nologo /W4 /MTd
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    // Create a vector object that contains 9 elements.
    vector<int> v;
    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }

    // Count the number of even numbers in the vector by
    // using the for_each function and a lambda.
    int evenCount = 0;
    for_each(v.begin(), v.end(), [&evenCount] (int n) {
        cout << n;
        if (n % 2 == 0) {
            cout << " is even " << endl;
            ++evenCount;
        } else {
            cout << " is odd " << endl;
        }
    });

    // Print the count of even numbers to the console.
    cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}

```

```

1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.

```

Комментарии

В примере третий аргумент функции `for_each` является лямбда-выражением. Часть `[&evenCount]` указывает предложение захвата выражения, `(int n)` определяет список параметров, а оставшаяся часть определяет тело выражения.

Пример 2: использование объекта-функции

Иногда лямбда-выражение может быть слишком громоздким для значительного расширения из состояния, показанного в предыдущем примере. В следующем примере вместо лямбда-выражения используется объект функции, а также функция `for_each` для получения тех же результатов, что и в примере 1. Оба примера хранят количество четных чисел в объекте `vector`. Для поддержания состояния операции класс `FunctorClass` хранит переменную `m_evenCount` по ссылке как переменную-член. Для выполнения операции `FunctorClass` реализует оператор вызова функции, **оператор ()**. Компилятор Microsoft C++ создает код, сравнимый по размеру и производительности, с лямбда-кодом в примере 1. Для несложной проблемы, такой как в этом примере, более простая конструкция лямбда-выражения, возможно, лучше, чем конструкция объекта-функции. Однако если вы считаете, что в будущем

потребуется значительно расширить функциональность, используйте конструкцию объекта-функции, чтобы упростить обслуживание кода.

Дополнительные сведения об **операторе ()** см. в разделе [вызов функции](#). Дополнительные сведения о функции **for_each** см. в разделе [for_each](#).

Код

```
// even_functor.cpp
// compile with: /EHsc
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

class FunctorClass
{
public:
    // The required constructor for this example.
    explicit FunctorClass(int& evenCount)
        : m_evenCount(evenCount) { }

    // The function-call operator prints whether the number is
    // even or odd. If the number is even, this method updates
    // the counter.
    void operator()(int n) const {
        cout << n;

        if (n % 2 == 0) {
            cout << " is even " << endl;
            ++m_evenCount;
        } else {
            cout << " is odd " << endl;
        }
    }

private:
    // Default assignment operator to silence warning C4512.
    FunctorClass& operator=(const FunctorClass&);

    int& m_evenCount; // the number of even variables in the vector.
};

int main()
{
    // Create a vector object that contains 9 elements.
    vector<int> v;
    for (int i = 1; i < 10; ++i) {
        v.push_back(i);
    }

    // Count the number of even numbers in the vector by
    // using the for_each function and a function object.
    int evenCount = 0;
    for_each(v.begin(), v.end(), FunctorClass(evenCount));

    // Print the count of even numbers to the console.
    cout << "There are " << evenCount
        << " even numbers in the vector." << endl;
}
```

```
1 is odd
2 is even
3 is odd
4 is even
5 is odd
6 is even
7 is odd
8 is even
9 is odd
There are 4 even numbers in the vector.
```

См. также

[Лямбда-выражения](#)

[Примеры лямбда-выражений](#)

[привести](#)

[generate_n](#)

[for_each](#)

[Спецификации исключений \(throw\)](#)

[Предупреждение компилятора \(уровень 1\) C4297](#)

[Модификаторы, специфичные для Майкрософт](#)

Примеры лямбда-выражений

12.11.2021 • 8 minutes to read

В данной статье приводится описание методов использования лямбда-выражений в программах. Общие сведения о лямбда-выражениях см. в разделе [лямбда-выражения](#). Дополнительные сведения о структуре лямбда-выражения см. в разделе [синтаксис лямбда-выражения](#).

Объявление лямбда-выражений

Пример 1

Поскольку лямбда-выражение типизировано, его можно назначить `auto` переменной или `function` объекту, как показано ниже:

```
// declaring_lambda_expressions1.cpp
// compile with: /EHsc /W4
#include <functional>
#include <iostream>

int main()
{
    using namespace std;

    // Assign the lambda expression that adds two numbers to an auto variable.
    auto f1 = [](int x, int y) { return x + y; };

    cout << f1(2, 3) << endl;

    // Assign the same lambda expression to a function object.
    function<int(int, int)> f2 = [](int x, int y) { return x + y; };

    cout << f2(3, 4) << endl;
}
```

В примере получается следующий результат.

```
5
7
```

Remarks

Дополнительные сведения см. в разделе [auto](#), [function](#) [классы вызов функции](#).

Хотя лямбда-выражения чаще всего объявляются в теле функции, можно объявлять их в любом месте, где можно инициализировать переменные.

Пример 2

Компилятор Microsoft C++ привязывает лямбда-выражение к захваченным переменным при объявлении выражения, а не при вызове выражения. В следующем примере содержится лямбда-выражение, которое фиксирует локальную переменную `i` по значению, а локальную переменную `j` — по ссылке. Поскольку лямбда-выражение захватывает `i` по значению, переопределение `i` далее в программе не влияет на результат выражения. Однако, поскольку лямбда-выражение захватывает `j` по ссылке, переопределение `j` влияет на результат выражения.

```
// declaring_lambda_expressions2.cpp
// compile with: /EHsc /W4
#include <functional>
#include <iostream>

int main()
{
    using namespace std;

    int i = 3;
    int j = 5;

    // The following lambda expression captures i by value and
    // j by reference.
    function<int (void)> f = [i, &j] { return i + j; };

    // Change the values of i and j.
    i = 22;
    j = 44;

    // Call f and print its result.
    cout << f() << endl;
}
```

В примере получается следующий результат.

47

[\[В этой статье\]](#)

Вызов лямбда-выражений

Можно вызывать лямбда-выражение сразу же, как показано в следующем фрагменте кода. Во втором фрагменте показано, как передать лямбда-выражение в качестве аргумента в алгоритмы стандартной библиотеки C++, такие как `find_if`.

Пример 1

В этом примере объявляется лямбда-выражение, которое возвращает сумму двух целых чисел и сразу же вызывает выражение с аргументами `5` и `4`.

```
// calling_lambda_expressions1.cpp
// compile with: /EHsc
#include <iostream>

int main()
{
    using namespace std;
    int n = [] (int x, int y) { return x + y; }(5, 4);
    cout << n << endl;
}
```

В примере получается следующий результат.

9

Пример 2

В этом примере лямбда-выражение передается в качестве аргумента функции `find_if`. Лямбда-

выражение возвращает значение, `true` если его параметр имеет четное число.

```
// calling_lambda_expressions2.cpp
// compile with: /EHsc /W4
#include <list>
#include <algorithm>
#include <iostream>

int main()
{
    using namespace std;

    // Create a list of integers with a few initial elements.
    list<int> numbers;
    numbers.push_back(13);
    numbers.push_back(17);
    numbers.push_back(42);
    numbers.push_back(46);
    numbers.push_back(99);

    // Use the find_if function and a lambda expression to find the
    // first even number in the list.
    const list<int>::const_iterator result =
        find_if(numbers.begin(), numbers.end(), [](int n) { return (n % 2) == 0; });

    // Print the result.
    if (result != numbers.end()) {
        cout << "The first even number in the list is " << *result << "." << endl;
    } else {
        cout << "The list contains no even numbers." << endl;
    }
}
```

В примере получается следующий результат.

```
The first even number in the list is 42.
```

Remarks

Дополнительные сведения о `find_if` функции см. в разделе [find_if](#). Дополнительные сведения о функциях стандартной библиотеки C++, которые выполняют стандартные алгоритмы, см. в разделе [<algorithm>](#).

[\[В этой статье\]](#)

Вложенные лямбда-выражения

Пример

Можно вложить одно лямбда-выражение в другое, как показано в следующем примере. Внутреннее лямбда-выражение умножает его аргумент на 2 и возвращает результат. Внешнее лямбда-выражение вызывает внутреннее лямбда-выражение с использованием его аргумента и добавляет к результату 3.

```
// nesting_lambda_expressions.cpp
// compile with: /EHsc /W4
#include <iostream>

int main()
{
    using namespace std;

    // The following lambda expression contains a nested lambda
    // expression.
    int timestwoplusthree = [](int x) { return [](int y) { return y * 2; }(x) + 3; }(5);

    // Print the result.
    cout << timestwoplusthree << endl;
}
```

В примере получается следующий результат.

13

Remarks

В этом примере значение параметра `[](int y) { return y * 2; }` является вложенным лямбда-выражением.

[\[В этой статье\]](#)

Higher-Order лямбда-функций

Пример

Многие языки программирования поддерживают концепцию *функции более высокого порядка*. Функция высшего порядка представляет собой лямбда-выражение, которое принимает другое лямбда-выражение в качестве аргумента или возвращает лямбда-выражение. Класс можно использовать, `function` чтобы разрешить лямбда-выражение C++ работать как функция высшего порядка. В следующем примере показано лямбда-выражение, которое возвращает объект `function`, и лямбда-выражение, которое принимает объект `function` в качестве аргумента.

```
// higher_order_lambda_expression.cpp
// compile with: /EHsc /W4
#include <iostream>
#include <functional>

int main()
{
    using namespace std;

    // The following code declares a lambda expression that returns
    // another lambda expression that adds two numbers.
    // The returned lambda expression captures parameter x by value.
    auto addtwointegers = [](int x) -> function<int(int)> {
        return [=](int y) { return x + y; };
    };

    // The following code declares a lambda expression that takes another
    // lambda expression as its argument.
    // The lambda expression applies the argument z to the function f
    // and multiplies by 2.
    auto higherorder = [](const function<int(int)>& f, int z) {
        return f(z) * 2;
    };

    // Call the lambda expression that is bound to higherorder.
    auto answer = higherorder(addtwointegers(7), 8);

    // Print the result, which is (7+8)*2.
    cout << answer << endl;
}
```

В примере получается следующий результат.

30

[\[В этой статье\]](#)

Использование лямбда-выражения в функции

Пример

Лямбда-выражения можно использовать в теле функции. Лямбда-выражение может получать доступ к любой функции или данным-членам, которые способна использовать включающая функция. Можно явно или неявно захватывать указатель, `this` чтобы предоставить доступ к функциям и элементам данных включающего класса. **Visual Studio 2017 версии 15,3 и более поздних** версий (доступно с `/std:c++17`): захват `this` по значению (`[*this]`), если лямбда-выражение будет использоваться в асинхронных или параллельных операциях, где код может быть выполнен после того, как исходный объект выходит из области действия.

Указатель можно использовать `this` явно в функции, как показано ниже:

```

// capture "this" by reference
void ApplyScale(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [this](int n) { cout << n * _scale << endl; });
}

// capture "this" by value (Visual Studio 2017 version 15.3 and later)
void ApplyScale2(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [*this](int n) { cout << n * _scale << endl; });
}

```

Кроме того, указатель можно захватывать `this` неявно:

```

void ApplyScale(const vector<int>& v) const
{
    for_each(v.begin(), v.end(),
        [=](int n) { cout << n * _scale << endl; });
}

```

В следующем примере показан класс `Scale`, который инкапсулирует значение масштаба.

```

// function_lambda_expression.cpp
// compile with: /EHsc /W4
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

class Scale
{
public:
    // The constructor.
    explicit Scale(int scale) : _scale(scale) {}

    // Prints the product of each element in a vector object
    // and the scale value to the console.
    void ApplyScale(const vector<int>& v) const
    {
        for_each(v.begin(), v.end(), [=](int n) { cout << n * _scale << endl; });
    }

private:
    int _scale;
};

int main()
{
    vector<int> values;
    values.push_back(1);
    values.push_back(2);
    values.push_back(3);
    values.push_back(4);

    // Create a Scale object that scales elements by 3 and apply
    // it to the vector object. Does not modify the vector.
    Scale s(3);
    s.ApplyScale(values);
}

```

В примере получается следующий результат.

```
3  
6  
9  
12
```

Remarks

Функция `ApplyScale` использует лямбда-выражение для выводения произведения масштаба на каждый элемент объекта `vector`. Лямбда-выражение неявно захватывает, `this` чтобы он мог получить доступ к `_scale` элементу.

[\[В этой статье\]](#)

Использование лямбда-выражений с шаблонами

Пример

Поскольку лямбда-выражения имеют тип, их можно использовать с шаблонами C++. В следующем примере показаны функции `negate_all` и `print_all`. `negate_all` Функция применяет унарный `operator-` к каждому элементу в `vector` объекте. Функция `print_all` печатает каждый элемент в объекте `vector` в консоли.

```
// template_lambda_expression.cpp  
// compile with: /EHsc  
#include <vector>  
#include <algorithm>  
#include <iostream>  
  
using namespace std;  
  
// Negates each element in the vector object. Assumes signed data type.  
template <typename T>  
void negate_all(vector<T>& v)  
{  
    for_each(v.begin(), v.end(), [](T& n) { n = -n; });  
}  
  
// Prints to the console each element in the vector object.  
template <typename T>  
void print_all(const vector<T>& v)  
{  
    for_each(v.begin(), v.end(), [](const T& n) { cout << n << endl; });  
}  
  
int main()  
{  
    // Create a vector of signed integers with a few elements.  
    vector<int> v;  
    v.push_back(34);  
    v.push_back(-43);  
    v.push_back(56);  
  
    print_all(v);  
    negate_all(v);  
    cout << "After negate_all(): " << endl;  
    print_all(v);  
}
```

В примере получается следующий результат.

```
34
-43
56
After negate_all():
-34
43
-56
```

Remarks

Дополнительные сведения о шаблонах C++ см. в разделе [шаблоны](#).

[\[В этой статье\]](#)

Обработка исключений

Пример

Тело лямбда-выражения выполняет правила как для структурированной обработки исключений (SEH), так и для обработки исключений C++. Можно обработать возникшее исключение в теле лямбда-выражения или перенести обработку исключения во включающий фрагмент. В следующем примере используется `for_each` функция и лямбда-выражение для заполнения `vector` объекта значениями другого. Он использует `try / catch` блок для управления недопустимым доступом к первому вектору.

```
// eh_lambda_expression.cpp
// compile with: /EHsc /W4
#include <vector>
#include <algorithm>
#include <iostream>
using namespace std;

int main()
{
    // Create a vector that contains 3 elements.
    vector<int> elements(3);

    // Create another vector that contains index values.
    vector<int> indices(3);
    indices[0] = 0;
    indices[1] = -1; // This is not a valid subscript. It will trigger an exception.
    indices[2] = 2;

    // Use the values from the vector of index values to
    // fill the elements vector. This example uses a
    // try/catch block to handle invalid access to the
    // elements vector.
    try
    {
        for_each(indices.begin(), indices.end(), [&](int index) {
            elements.at(index) = index;
        });
    }
    catch (const out_of_range& e)
    {
        cerr << "Caught '" << e.what() << "'." << endl;
    };
}
```

В примере получается следующий результат.

```
Caught 'invalid vector<T> subscript'.
```

Remarks

Дополнительные сведения об обработке исключений см. в разделе [обработка исключений](#).

[[В этой статье](#)]

Использование лямбда-выражений с управляемыми типами (C++/CLI)

Пример

Предложение захвата лямбда-выражения не может содержать переменную, которая имеет управляемый тип. Однако можно передать аргумент с управляемым типом в список параметров лямбда-выражения. В следующем примере содержится лямбда-выражение, которое захватывает локальную неуправляемую переменную `ch` по значению и принимает объект `System.String` в качестве параметра.

```
// managed_lambda_expression.cpp
// compile with: /clr
using namespace System;

int main()
{
    char ch = '!'; // a local unmanaged variable

    // The following lambda expression captures local variables
    // by value and takes a managed String object as its parameter.
    [=](String ^s) {
        Console::WriteLine(s + Convert::ToChar(ch));
    }("Hello");
}
```

В примере получается следующий результат.

```
Hello!
```

Remarks

Можно также использовать лямбда-выражения с библиотекой STL/CLR. Дополнительные сведения см. в разделе [Справочник по библиотеке STL/CLR](#).

IMPORTANT

Лямбда-выражения не поддерживаются в следующих управляемых сущностях среды CLR: `ref class`, `ref struct`, `value class` и `value struct`.

[[В этой статье](#)]

См. также

[Лямбда-выражения](#)

[Синтаксис лямбда-выражений](#)

`auto`

`function` См

`find_if`

`<algorithm>`

[Вызов функции](#)

[Шаблоны](#)

Обработка исключений

Справочник по библиотеке STL/CLR

Лямбда-выражения constexpr в C++

12.11.2021 • 2 minutes to read

Visual Studio 2017 версии 15.3 и более поздних версий (доступно с `/std:c++17`): лямбда-выражение может быть объявлено как `constexpr` или использоваться в константном выражении, когда инициализация каждого элемента данных, который он захватывает или вводит, разрешена в константном выражении.

```
int y = 32;
auto answer = [y]() constexpr
{
    int x = 10;
    return y + x;
};

constexpr int Increment(int n)
{
    return [n] { return n + 1; }();
}
```

Лямбда-выражение неявно, `constexpr` если его результат удовлетворяет требованиям `constexpr` функции:

```
auto answer = [](int n)
{
    return 32 + n;
};

constexpr int response = answer(10);
```

Если лямбда-выражение является неявно или явным `constexpr` и преобразуется в указатель функции, то результирующая функция также будет иметь `constexpr` следующее:

```
auto Increment = [](int n)
{
    return n + 1;
};

constexpr int(*inc)(int) = Increment;
```

См. также

[Справочник по языку C++](#)
[Объекты функций в стандартной библиотеке C++](#)
[Вызов функции](#)
[for_each](#)

Массивы (C++)

12.11.2021 • 9 minutes to read

Массив — это последовательность объектов того же типа, которые занимают смежную область памяти. Традиционные массивы в стиле C являются источником многих ошибок, но по-прежнему являются общими, особенно в старых базах кода. В современных C++ мы настоятельно рекомендуем использовать `std::Vector` или `std::Array` вместо массивов в стиле C, описанных в этом разделе. Оба этих типа стандартных библиотек хранят свои элементы в виде непрерывного блока памяти. Однако они обеспечивают гораздо большую безопасность типов и итераторы поддержки, которые гарантированно указывают на допустимое расположение в последовательности. Дополнительные сведения см. в разделе [контейнеры](#).

Объявления стека

В объявлении массива C++ размер массива указывается после имени переменной, а не после имени типа, как в некоторых других языках. В следующем примере объявляется массив значений типа `Double` 1000, которые будут выделяться в стеке. Число элементов должно быть указано как целочисленный литерал или `else` в качестве константного выражения. Это обусловлено тем, что компилятору необходимо выяснить, сколько пространства стека следует выделить; оно не может использовать значение, вычисленное во время выполнения. Каждому элементу массива присваивается значение по умолчанию 0. Если не назначить значение по умолчанию, каждый элемент изначально будет содержать случайные значения, находящимся в этой области памяти.

```
constexpr size_t size = 1000;

// Declare an array of doubles to be allocated on the stack
double numbers[size] {0};

// Assign a new value to the first element
numbers[0] = 1;

// Assign a value to each subsequent element
// (numbers[1] is the second element in the array.)
for (size_t i = 1; i < size; i++)
{
    numbers[i] = numbers[i-1] * 1.1;
}

// Access each element
for (size_t i = 0; i < size; i++)
{
    std::cout << numbers[i] << " ";
```

Первый элемент в массиве является элементом начальном. Последним элементом является элемент $(n-1)$, где n — число элементов, которые может содержать массив. Число элементов в объявлении должно иметь целочисленный тип и должно быть больше 0. Вы обязаны убедиться, что программа никогда не передает значение оператору индекса, который больше `(size - 1)`.

Массив нулевого размера допустим только в том случае, если массив является последним полем в `struct` или `union` и если расширения Microsoft включены (`/za` или `/permissive-` не заданы).

Массивы на основе стека быстрее выделяются и получают доступ, чем массивы на основе кучи. Однако пространство стека ограничено. Число элементов массива не может быть настолько большим, что в нем

используется слишком много памяти стека. Насколько сильно зависит от программы. Для определения того, является ли массив слишком большим, можно использовать средства профилирования.

Объявления кучи

Может потребоваться, чтобы массив был слишком большим для выделения в стеке или его размер не известен во время компиляции. Можно выделить этот массив в куче с помощью `new[]` выражения.

Оператор возвращает указатель на первый элемент. Оператор индекса работает с переменной-указателем так же, как и с массивом на основе стека. Также можно использовать [арифметические операции с указателями](#) для перемещения указателя на произвольные элементы в массиве. Вы обязаны убедиться в том, что:

- всегда сохраняется копия адреса исходного указателя, чтобы можно было удалить память, когда массив больше не нужен.
- Вы не увеличиваете или уменьшаете адрес указателя после границ массива.

В следующем примере показано, как определить массив в куче во время выполнения. В нем показано, как получить доступ к элементам массива с помощью оператора индекса и с помощью арифметики указателей:

```

void do_something(size_t size)
{
    // Declare an array of doubles to be allocated on the heap
    double* numbers = new double[size]{ 0 };

    // Assign a new value to the first element
    numbers[0] = 1;

    // Assign a value to each subsequent element
    // (numbers[1] is the second element in the array.)
    for (size_t i = 1; i < size; i++)
    {
        numbers[i] = numbers[i - 1] * 1.1;
    }

    // Access each element with subscript operator
    for (size_t i = 0; i < size; i++)
    {
        std::cout << numbers[i] << " ";
    }

    // Access each element with pointer arithmetic
    // Use a copy of the pointer for iterating
    double* p = numbers;

    for (size_t i = 0; i < size; i++)
    {
        // Dereference the pointer, then increment it
        std::cout << *p++ << " ";
    }

    // Alternate method:
    // Reset p to numbers[0]:
    p = numbers;

    // Use address of pointer to compute bounds.
    // The compiler computes size as the number
    // of elements * (bytes per element).
    while (p < (numbers + size))
    {
        // Dereference the pointer, then increment it
        std::cout << *p++ << " ";
    }

    delete[] numbers; // don't forget to do this!
}

int main()
{
    do_something(108);
}

```

Инициализация массивов

Можно инициализировать массив в цикле, по одному элементу за раз или в одной инструкции.

Содержимое следующих двух массивов идентично:

```
int a[10];
for (int i = 0; i < 10; ++i)
{
    a[i] = i + 1;
}

int b[10]{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Передача массивов в функции

Когда массив передается в функцию, он передается в качестве указателя на первый элемент, независимо от того, является ли он массивом на основе стека или кучи. Указатель не содержит дополнительных сведений о размере или типе. Такое поведение называется *указателем Decay*. При передаче массива в функцию необходимо всегда указывать количество элементов в отдельном параметре. Такое поведение также подразумевает, что элементы массива не копируются, когда массив передается в функцию. Чтобы запретить функции изменять элементы, укажите параметр в качестве указателя на `const` элементы.

В следующем примере показана функция, которая принимает массив и длину. Указатель указывает на исходный массив, а не на копию. Поскольку параметр не `const` имеет значение, функция может изменять элементы массива.

```
void process(double *p, const size_t len)
{
    std::cout << "process:\n";
    for (size_t i = 0; i < len; ++i)
    {
        // do something with p[i]
    }
}
```

Объявите и определите параметр массива `p` так, `const` чтобы он был доступен только для чтения в блоке функции:

```
void process(const double *p, const size_t len);
```

Одна и та же функция может также быть объявлена в таких случаях без изменения поведения. Массив по-прежнему передается в качестве указателя на первый элемент:

```
// Unsized array
void process(const double p[], const size_t len);

// Fixed-size array. Length must still be specified explicitly.
void process(const double p[1000], const size_t len);
```

Многомерные массивы

Массивы, созданные из других массивов, являются многомерными. Такие многомерные массивы определяются путем последовательного размещения нескольких константных выражений, заключенных в квадратные скобки. Рассмотрим, например, следующее объявление:

```
int i2[5][7];
```

Он задает массив типа, по `int` сути упорядоченный в двумерной матрице из пяти строк и семи столбцов,

как показано на следующем рисунке.

0, 0	0, 1	0, 2	0, 3	0, 4	0, 5	0, 6
1, 0	1, 1	1, 2	1, 3	1, 4	1, 5	1, 6
2, 0	2, 1	2, 2	2, 3	2, 4	2, 5	2, 6
3, 0	3, 1	3, 2	3, 3	3, 4	3, 5	3, 6
4, 0	4, 1	4, 2	4, 3	4, 4	4, 5	4, 6

Концептуальная структура многомерного массива

Можно объявить многомерные массивы, имеющие список инициализаторов (как описано в разделе [инициализаторы](#)). В этих объявлениях константное выражение, указывающее границы для первого измерения, может быть опущено. Пример:

```
// arrays2.cpp
// compile with: /c
const int cMarkets = 4;
// Declare a float that represents the transportation costs.
double TransportCosts[][][cMarkets] = {
    { 32.19, 47.29, 31.99, 19.11 },
    { 11.29, 22.49, 33.47, 17.29 },
    { 41.97, 22.09, 9.76, 22.55 }
};
```

В показанном выше объявлении определяется массив, состоящий из трех строк и четырех столбцов. Строки представляют фабрики, а столбцы — рынки, на которые фабрики поставляют свою продукцию. Значения — это стоимости транспортировки с фабрик на рынки. Первое измерение массива опущено, но компилятор заполняет его, проверяя инициализатор.

Использование оператора косвенного обращения (*) в n-мерном массиве приводит к получению n-1 многомерного массива. Если n равно 1, создается скаляр (или элемент массива).

Массивы C++ размещаются в памяти по строкам. Построчный порядок означает, что быстрее всего изменяется последний индекс.

Пример

Можно также опустить спецификацию границ для первого измерения многомерного массива в объявлениях функций, как показано ниже:

```

// multidimensional_arrays.cpp
// compile with: /EHsc
// arguments: 3
#include <limits> // Includes DBL_MAX
#include <iostream>

const int cMkts = 4, cFacts = 2;

// Declare a float that represents the transportation costs
double TransportCosts[][][cMkts] = {
    { 32.19, 47.29, 31.99, 19.11 },
    { 11.29, 22.49, 33.47, 17.29 },
    { 41.97, 22.09, 9.76, 22.55 }
};

// Calculate size of unspecified dimension
const int cFactories = sizeof TransportCosts /
    sizeof( double[cMkts] );

double FindMinToMkt( int Mkt, double myTransportCosts[][cMkts], int mycFacts);

using namespace std;

int main( int argc, char *argv[] ) {
    double MinCost;

    if (argc[1] == 0) {
        cout << "You must specify the number of markets." << endl;
        exit(0);
    }
    MinCost = FindMinToMkt( *argv[1] - '0', TransportCosts, cFacts);
    cout << "The minimum cost to Market " << argv[1] << " is: "
        << MinCost << "\n";
}

double FindMinToMkt(int Mkt, double myTransportCosts[][cMkts], int mycFacts) {
    double MinCost = DBL_MAX;

    for( size_t i = 0; i < cFacts; ++i )
        MinCost = (MinCost < TransportCosts[i][Mkt]) ?
            MinCost : TransportCosts[i][Mkt];

    return MinCost;
}

```

The minimum cost to Market 3 is: 17.29

Эта функция `FindMinToMkt` написана таким, что добавление новых фабрик не требует каких-либо изменений кода, а только перекомпиляции.

Инициализация массивов

Массивы объектов, имеющих конструктор класса, инициализируются конструктором. Если в списке инициализаторов меньше элементов, чем элементов массива, то для остальных элементов используется конструктор по умолчанию. Если для класса не определен конструктор по умолчанию, список инициализаторов должен быть *завершен*, то есть должен быть один инициализатор для каждого элемента в массиве.

Рассмотрим класс `Point`, определяющий два конструктора:

```

// initializing_arrays1.cpp
class Point
{
public:
    Point() // Default constructor.
    {
    }
    Point( int, int ) // Construct from two ints
    {
    }
};

// An array of Point objects can be declared as follows:
Point aPoint[3] = {
    Point( 3, 3 ) // Use int, int constructor.
};

int main()
{
}

```

Первый элемент `aPoint` создается с помощью конструктора `Point(int, int)`, а оставшиеся два элемента — с помощью конструктора по умолчанию.

Статические массивы членов (`const` вне зависимости от объявления класса) могут быть инициализированы в своих определениях. Пример:

```

// initializing_arrays2.cpp
class WindowColors
{
public:
    static const char *rgszWindowPartList[7];
};

const char *WindowColors::rgszWindowPartList[7] = {
    "Active Title Bar", "Inactive Title Bar", "Title Bar Text",
    "Menu Bar", "Menu Bar Text", "Window Background", "Frame"  };
int main()
{
}

```

Доступ к элементам массива

К отдельным элементам массива можно обращаться при помощи оператора индекса массива (`[]`). При использовании имени одномерного массива без индекса он вычисляется как указатель на первый элемент массива.

```

// using_arrays.cpp
int main() {
    char chArray[10];
    char *pch = chArray; // Evaluates to a pointer to the first element.
    char ch = chArray[0]; // Evaluates to the value of the first element.
    ch = chArray[3]; // Evaluates to the value of the fourth element.
}

```

Если используются многомерные массивы, в выражениях можно использовать различные сочетания.

```

// using_arrays_2.cpp
// compile with: /EHsc /W1
#include <iostream>
using namespace std;
int main() {
    double multi[4][4][3]; // Declare the array.
    double (*p2multi)[3];
    double (*p1multi);

    cout << multi[3][2][2] << "\n"; // C4700 Use three subscripts.
    p2multi = multi[3]; // Make p2multi point to
                        // fourth "plane" of multi.
    p1multi = multi[3][2]; // Make p1multi point to
                        // fourth plane, third row
                        // of multi.
}

```

В приведенном выше коде `multi` является трехмерным массивом типа `double`. Указатель `p2multi` указывает на массив типа `double`, размер которого равен `double` трем. В этом примере массив используется с одним, двумя и тремя индексами. Хотя чаще всего указывается все индексы, как в `cout` инструкции, иногда бывает полезно выбрать конкретное подмножество элементов массива, как показано в следующих инструкциях `cout`.

Перегрузка оператора индекса

Как и другие операторы, оператор индекса (`[]`) может быть переопределен пользователем. Поведение оператора индекса по умолчанию, если он не перегружен, — совмещать имя массива и индекс с помощью следующего метода.

```
*((array_name) + (subscript))
```

Как и во всех дополнениях, включающих типы указателей, масштабирование выполняется автоматически для корректировки размера типа. Результатирующее значение не *n* байт из источника `array_name`; вместо этого это *n*-й элемент массива. Дополнительные сведения об этом преобразовании см. в разделе [аддитивные операторы](#).

Аналогично, для многомерных массивов адрес извлекается с использованием следующего метода.

```
((array_name) + (subscript1 * max2 * max3 * ... * maxn) + (subscript2 * max3 * ... * maxn) + ... + subscriptn))
```

Массивы в выражениях

Если идентификатор типа массива встречается в выражении, отличном от `sizeof`, адрес (`&`) или инициализации ссылки, он преобразуется в указатель на первый элемент массива. Пример:

```

char szError1[] = "Error: Disk drive not ready.";
char *psz = szError1;

```

Указатель `psz` указывает на первый элемент массива `szError1`. Массивы, в отличие от указателей, не являются изменяемыми л-значениями. Вот почему следующее назначение недопустимо:

```
szError1 = psz;
```

См. также раздел

std:: Array

Ссылки (C++)

12.11.2021 • 2 minutes to read

В ссылке, как и в указателе, хранится адрес объекта, расположенного в другой области памяти. В отличие от указателя, после инициализации ссылку нельзя перенаправить на другой объект или присвоить ей нулевое значение. Существует два вида ссылок: ссылки lvalue, которые ссылаются на именованную переменную и ссылки rvalue, которые ссылаются на [временный объект](#). Оператор & указывает на ссылку lvalue, а оператор && указывает либо на ссылку rvalue, либо на универсальную ссылку (rvalue или lvalue) в зависимости от контекста.

Ссылки могут объявляться с помощью следующего синтаксиса.

[спецификаторы класса хранения] [ОПС-квалификаторы] выражения -описатели типа [MS-Modifiers]
выражение декларатора [=];

Можно использовать любой допустимый декларатор, задающий ссылку. Следующий упрощенный синтаксис применяется всегда, кроме случаев, когда ссылка является ссылкой на функцию или тип массива.

[спецификаторы класса хранения] [ОПС-квалификаторы] описатели типа [& или &&] [ОПС-квалификаторы] [= выражение идентификатора];

Ссылки объявляются с использованием следующей последовательности.

1. Спецификаторы объявления:

- Необязательный спецификатор класса хранения.
- Необязательные `const` и/или `volatile` квалификаторы.
- Спецификатор типа: имя типа.

2. Декларатор:

- Необязательный модификатор, используемый в системах Microsoft. Дополнительные сведения см. в разделе [модификаторы, зависящие от Microsoft](#).
- & Оператор или && оператор.
- Необязательный `const` и/или `volatile` квалифиликаторов.
- Идентификатор.

3. Необязательный инициализатор.

Более сложные формы декларатора для указателей на массивы и функции также применяются к ссылкам на массивы и функции. Дополнительные сведения см. в разделе [указатели](#).

Несколько деклараторов и инициализаторов могут отображаться в разделенном запятыми списке после отдельного спецификатора объявления. Пример:

```
int &i;  
int &i, &j;
```

Ссылки, указатели и объекты могут быть объявлены вместе.

```
int &ref, *ptr, k;
```

Ссылка содержит адрес объекта, однако с синтаксической точки зрения ведет себя как объект.

В следующей программе обратите внимание, что имя объекта, `s` и ссылка на объект, `SRef`, могут использоваться идентично в следующих программах.

Пример

```
// references.cpp
#include <stdio.h>
struct S {
    short i;
};

int main() {
    S s;      // Declare the object.
    S& SRef = s;    // Declare the reference.
    s.i = 3;

    printf_s("%d\n", s.i);
    printf_s("%d\n", SRef.i);

    SRef.i = 4;
    printf_s("%d\n", s.i);
    printf_s("%d\n", SRef.i);
}
```

```
3
3
4
4
```

См. также раздел

[Аргументы функции ссылочного типа](#)

[Функции ссылочного типа возвращают](#)

[Ссылки на указатели](#)

Декларатор ссылки lvalue: &

12.11.2021 • 2 minutes to read

Содержит адрес объекта, но синтаксически ведет себя подобно объекту.

Синтаксис

```
type-id & cast-expression
```

Remarks

Ссылку lvalue можно считать другим именем для объекта. Объявление ссылки lvalue состоит из необязательного списка спецификаторов, за которым следует декларатор ссылки. Ссылка должна быть инициализирована и не может быть изменена.

Любой объект, адрес которого можно преобразовать в некоторый тип указателя, можно также преобразовать в аналогичный ссылочный тип. Например, любой объект, адрес которого можно преобразовать в тип `char *`, можно также преобразовать в тип `char &`.

Не путайте объявления ссылок с использованием [оператора взятия адреса](#). Если `&` *идентификатору* предшествует тип, например `int` или `char`, *идентификатор* объявляется как ссылка на тип. Когда `&` *идентификатору* не предшествует тип, использование является оператором взятия адреса.

Пример

В следующем примере демонстрируется декларатор ссылки путем объявления объекта `Person` и ссылки на этот объект. Поскольку `rFriend` является ссылкой на `myFriend`, при обновлении любой из этих переменных изменяется один и тот же объект.

```
// reference_declarator.cpp
// compile with: /EHsc
// Demonstrates the reference declarator.
#include <iostream>
using namespace std;

struct Person
{
    char* Name;
    short Age;
};

int main()
{
    // Declare a Person object.
    Person myFriend;

    // Declare a reference to the Person object.
    Person& rFriend = myFriend;

    // Set the fields of the Person object.
    // Updating either variable changes the same object.
    myFriend.Name = "Bill";
    rFriend.Age = 40;

    // Print the fields of the Person object to the console.
    cout << rFriend.Name << " is " << myFriend.Age << endl;
}
```

```
Bill is 40
```

См. также

[Справочные материалы](#)

[Аргументы функции ссылочного типа](#)

[Функции ссылочного типа возвращают](#)

[Ссылки на указатели](#)

Декларатор ссылки rvalue: &&

12.11.2021 • 10 minutes to read

Содержит ссылку на выражение rvalue.

Синтаксис

```
type-id && cast-expression
```

Remarks

Ссылки rvalue позволяют различать значения lvalue и rvalue. Ссылки lvalue и rvalue синтаксически и семантически аналогичны, однако они подчиняются несколько различающимся правилам.

Дополнительные сведения о значениях lvalue и rvalue см. в разделе [значения lvalue и rvalues](#).

Дополнительные сведения о ссылках lvalue см. в разделе [декларатор ссылки lvalue: &](#).

В следующих разделах описывается, как ссылки rvalue поддерживают реализацию *семантики перемещения и идеальной пересылки*.

Семантика перемещения

Ссылки rvalue поддерживают реализацию *семантики перемещения*, что может значительно повысить производительность приложений. Семантика перемещения позволяет создавать код, который переносит ресурсы (например, динамически выделяемую память) из одного объекта в другой. Семантика перемещения работает, поскольку она позволяет переносить ресурсы из временных объектов, на которые невозможно ссылаться из других мест в программе.

Для реализации семантики перемещения, как правило, предоставляется *конструктор перемещения* и при необходимости оператор присваивания перемещения (*operator =*) к вашему классу. Операции копирования и присваивания, источниками которых являются значения rvalue, затем автоматически используют семантику перемещения. В отличие от конструктора копирования по умолчанию, компилятор не предоставляет конструктор перемещения по умолчанию. Дополнительные сведения о написании конструктора перемещения и его использовании в приложении см. в разделе [конструкторы Move и операторы присваивания перемещения \(C++\)](#).

Можно также перегрузить обычные функции и операторы, чтобы воспользоваться преимуществами семантики перемещения. В Visual Studio 2010 введена семантика перемещения в стандартную библиотеку C++. Например, класс `string` реализует операции, использующие семантику перемещения. Рассмотрим следующий пример, в котором объединяются несколько строк и выводится результат:

```
// string_concatenation.cpp
// compile with: /EHsc
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string s = string("h") + "e" + "ll" + "o";
    cout << s << endl;
}
```

до Visual Studio 2010 каждый вызов **operator +** выделяет и возвращает новый временный `string` объект (`rvalue`). **оператор +** не может добавить одну строку в другую, так как не знает, являются ли исходные строки значения `lvalue` или `rvalue`. Если обе исходные строки являются значениями `lvalue`, на них могут указывать ссылки из какого-либо другого места программы, поэтому их не следует изменять. При использовании ссылок `rvalue` **оператор +** может быть изменен для получения `rvalue`, на который нельзя ссылаться в других местах программы. Таким образом, **оператор +** теперь может добавлять одну строку к другой. Это может значительно снизить количество операций динамического выделения памяти, которые должен выполнять класс `string`. Дополнительные сведения о `string` классе см. в разделе [класс `basic_string`](#).

Семантика перемещения также помогает, когда компилятор не может использовать оптимизацию возвращаемого значения (RVO) или оптимизацию именованного возвращаемого значения (NRVO). В таких случаях компилятор вызывает конструктор перемещения, если он определен в типе.

Для лучшего понимания семантики перемещения рассмотрим пример вставки элемента в объект `vector`. Если ресурсы объекта `vector` превышены, объект `vector` должен заново выделить память для своих элементов, а затем скопировать каждый элемент в другое расположение в памяти, чтобы освободить место для добавленного элемента. Когда операция вставки копирует элемент, она создает новый элемент, вызывает конструктор копирования для копирования данных из предыдущего элемента в новый элемент, а затем уничтожает предыдущий элемент. Семантика перемещения позволяет перемещать объекты напрямую, не выполняя ресурсоемкие операции выделения памяти и копирования.

Чтобы воспользоваться преимуществами семантики перемещения в примере `vector`, можно создать конструктор перемещения для перемещения данных из одного объекта в другой.

дополнительные сведения о семантике перемещения в стандартную библиотеку `c++` в Visual Studio 2010 см. в разделе [стандартная библиотека `c++`](#).

Точная пересылка

Точная пересылка уменьшает необходимость в использовании перегруженных функций и позволяет избежать проблем пересылки. *Проблема пересылки* может возникнуть при написании универсальной функции, которая принимает ссылки в качестве параметров и передает (или *пересыпает*) эти параметры другой функции. Например, если универсальная функция принимает параметр типа `const T&`, вызываемая функция не может изменять значение этого параметра. Если универсальная функция принимает параметр типа `T&`, эта функция не может вызываться с использованием значения `rvalue` (такого как временный объект или целочисленный литерал).

Обычно для решения этой проблемы необходимо предоставить перегруженные версии универсальной функции, принимающие для каждого из своих параметров значения `T&` и `const T&`. В результате число перегруженных функций экспоненциально возрастает по мере увеличения числа параметров. Ссылки `rvalue` позволяют создать одну версию функции, принимающую произвольные аргументы и пересылающую их в другую функцию, как если бы эта другая функция вызывалась напрямую.

Рассмотрим следующий пример, в котором определяются четыре типа: `W`, `X`, `Y` и `Z`. Конструктор для каждого типа принимает в качестве параметров разные сочетания `const` и ссылки, отличные от `const` левостороннего.

```
struct W
{
    W(int&, int&) {}
};

struct X
{
    X(const int&, int&) {}
};

struct Y
{
    Y(int&, const int&) {}
};

struct Z
{
    Z(const int&, const int&) {}
};
```

Предположим, требуется написать универсальную функцию, которая создает объекты. В следующем примере показан один из возможных способов написания этой функции:

```
template <typename T, typename A1, typename A2>
T* factory(A1& a1, A2& a2)
{
    return new T(a1, a2);
}
```

В следующем примере показан допустимый вызов функции `factory`:

```
int a = 4, b = 5;
W* pw = factory<W>(a, b);
```

Однако следующий пример содержит недопустимый вызов функции `factory`, поскольку функция `factory` принимает в качестве параметров ссылки `lvalue`, допускающие изменения, но вызывается с использованием значений `rvalue`:

```
Z* pz = factory<Z>(2, 2);
```

Обычно для решения этой проблемы необходимо создать перегруженные версии функции `factory` для каждого сочетания параметров `A&` и `const A&`. Ссылки `rvalue` позволяют создать одну версию функции `factory`, как показано в следующем примере:

```
template <typename T, typename A1, typename A2>
T* factory(A1&& a1, A2&& a2)
{
    return new T(std::forward<A1>(a1), std::forward<A2>(a2));
}
```

В этом примере в качестве параметров функции `factory` используются ссылки `rvalue`. Функция `std::Forward` пересыпает параметры функции `Factory` конструктору класса шаблона.

В следующем примере показана функция `main`, использующая измененную функцию `factory` для создания экземпляров классов `W`, `X`, `Y` и `Z`. Измененная функция `factory` пересыпает свои параметры (значения `lvalue` или `rvalue`) в конструктор соответствующего класса.

```
int main()
{
    int a = 4, b = 5;
    W* pw = factory<W>(a, b);
    X* px = factory<X>(2, b);
    Y* py = factory<Y>(a, 2);
    Z* pz = factory<Z>(2, 2);

    delete pw;
    delete px;
    delete py;
    delete pz;
}
```

Дополнительные свойства ссылок rvalue

Можно перегрузить функцию, чтобы она принимала ссылки lvalue и rvalue.

Перегружая функцию для получения `const` ссылки `lvalue` или ссылки `rvalue`, можно написать код, который различает неизменяемые объекты (значения `lvalue`) и изменяемые временные значения (`rvalue`). Вы можете передать объект в функцию, которая принимает ссылку `rvalue`, если только объект не помечен как `const`. В следующем примере показана перегруженная функция `f`, принимающая ссылки `lvalue` и `rvalue`. Функция `main` вызывает функцию `f` как со значениями `lvalue`, так и со значениями `rvalue`.

```
// reference-overload.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
    // TODO: Add resources for the class here.
};

void f(const MemoryBlock&)
{
    cout << "In f(const MemoryBlock&). This version cannot modify the parameter." << endl;
}

void f(MemoryBlock&&)
{
    cout << "In f(MemoryBlock&&). This version can modify the parameter." << endl;
}

int main()
{
    MemoryBlock block;
    f(block);
    f(MemoryBlock());
}
```

В этом примере выводятся следующие данные:

```
In f(const MemoryBlock&). This version cannot modify the parameter.  
In f(MemoryBlock&&). This version can modify the parameter.
```

В этом примере при первом вызове функции `f` в качестве аргумента передается локальная переменная (значение `lvalue`). При втором вызове функции `f` в качестве аргумента передается временный объект. Поскольку на временный объект невозможно ссылаться из какого-либо другого места в программе, вызов привязывается к перегруженной версии функции `f`, которая принимает ссылку `rvalue` и может свободно изменять этот объект.

Компилятор обрабатывает именованную ссылку rvalue как значение lvalue, а безымянную ссылку rvalue как значение rvalue.

При создании функции, которая принимает в качестве параметра ссылку `rvalue`, в теле функции этот параметр обрабатывается как значение `lvalue`. Компилятор обрабатывает именованную ссылку `rvalue` как значение `lvalue`, поскольку на именованный объект возможны ссылки из нескольких частей программы; было бы опасно разрешить нескольким частям программы изменять или удалять ресурсы из этого объекта. Например, если несколько частей программы попытаются переместить ресурсы из одного и того же объекта, только первая часть сможет успешно переместить ресурс.

В следующем примере показана перегруженная функция `g`, принимающая ссылки `lvalue` и `rvalue`. Функция `f` принимает ссылку `rvalue` в качестве своего параметра (именованная ссылка `rvalue`) и возвращает ссылку `rvalue` (безымянная ссылка `rvalue`). При вызове функции `g` из функции `f` механизм разрешения перегрузок выбирает версию `g`, которая принимает ссылку `lvalue`, так как в теле функции `f` ее параметр рассматривается как значение `lvalue`. При вызове функции `g` из функции `main` механизм разрешения перегрузок выбирает версию `g`, которая принимает ссылку `rvalue`, так как функция `f` возвращает ссылку `rvalue`.

```
// named-reference.cpp  
// Compile with: /EHsc  
#include <iostream>  
using namespace std;  
  
// A class that contains a memory resource.  
class MemoryBlock  
{  
    // TODO: Add resources for the class here.  
};  
  
void g(const MemoryBlock&)  
{  
    cout << "In g(const MemoryBlock&)." << endl;  
}  
  
void g(MemoryBlock&&)  
{  
    cout << "In g(MemoryBlock&&)." << endl;  
}  
  
MemoryBlock&& f(MemoryBlock&& block)  
{  
    g(block);  
    return move(block);  
}  
  
int main()  
{  
    g(f(MemoryBlock()));  
}
```

В этом примере выводятся следующие данные:

```
In g(const MemoryBlock&).
In g(MemoryBlock&&).
```

В этом примере функция `main` передает значение `rvalue` в функцию `f`. В теле функции `f` ее именованный параметр рассматривается как значение `lvalue`. При вызове функции `f` из функции `g` параметр связывается со ссылкой `lvalue` (первая перегруженная версия функции `g`).

- **Можно привести значение lvalue к ссылке rvalue.**

Функция `std::Move` стандартной библиотеки C++ позволяет преобразовать объект в ссылку `rvalue` на этот объект. Кроме того, можно использовать `static_cast` ключевое слово для приведения левостороннего значения к ссылке `rvalue`, как показано в следующем примере:

```
// cast-reference.cpp
// Compile with: /EHsc
#include <iostream>
using namespace std;

// A class that contains a memory resource.
class MemoryBlock
{
    // TODO: Add resources for the class here.
};

void g(const MemoryBlock&)
{
    cout << "In g(const MemoryBlock&)." << endl;
}

void g(MemoryBlock&&)
{
    cout << "In g(MemoryBlock&&)." << endl;
}

int main()
{
    MemoryBlock block;
    g(block);
    g(static_cast<MemoryBlock&&>(block));
}
```

В этом примере выводятся следующие данные:

```
In g(const MemoryBlock&).
In g(MemoryBlock&&).
```

В шаблонах функций выводятся свои типы аргументов шаблона, а затем используются правила сворачивания ссылок.

Часто создается шаблон функции, который передает (или *пересыпает*) свои параметры другой функции. Важно понимать, как работает вывод типа шаблона для шаблонов функций, принимающих ссылки `rvalue`.

Если аргумент функции является значением `rvalue`, компилятор считает, что аргумент является ссылкой `rvalue`. Например, при передаче ссылки `rvalue` на объект типа `x` в шаблон функции, который принимает в качестве параметра тип `t&&`, логика выведения аргумента шаблона определит, что `t` — это `x`.

Поэтому параметр имеет тип `x&&`. Если аргумент функции является левосторонним `const` значением или `lvalue`, компилятор выводит его тип как ссылку `lvalue` или `const` ссылку `lvalue` этого типа.

В следующем примере объявляется один шаблон структуры, который затем специализируется для различных ссылочных типов. Функция `print_type_and_value` принимает в качестве параметра ссылку `rvalue` и пересыпает ее в соответствующую специализированную версию метода `S::print`. Функция `main` показывает различные способы вызова метода `S::print`.

```
// template-type-deduction.cpp
// Compile with: /EHsc
#include <iostream>
#include <string>
using namespace std;

template<typename T> struct S;

// The following structures specialize S by
// lvalue reference (T&), const lvalue reference (const T&),
// rvalue reference (T&&), and const rvalue reference (const T&&).
// Each structure provides a print method that prints the type of
// the structure and its parameter.

template<typename T> struct S<T&> {
    static void print(T& t)
    {
        cout << "print<T&>: " << t << endl;
    }
};

template<typename T> struct S<const T&> {
    static void print(const T& t)
    {
        cout << "print<const T&>: " << t << endl;
    }
};

template<typename T> struct S<T&&> {
    static void print(T&& t)
    {
        cout << "print<T&&>: " << t << endl;
    }
};

template<typename T> struct S<const T&&> {
    static void print(const T&& t)
    {
        cout << "print<const T&&>: " << t << endl;
    }
};

// This function forwards its parameter to a specialized
// version of the S type.
template <typename T> void print_type_and_value(T&& t)
{
    S<T&&>::print(std::forward<T>(t));
}

// This function returns the constant string "fourth".
const string fourth() { return string("fourth"); }

int main()
{
    // The following call resolves to:
    // print_type_and_value<string&&>(string& && t)
    // Which collapses to:
    // print_type_and_value<string&&>(string& t)
    string s1("first");
    print_type_and_value(s1);

    // The following call resolves to:
}
```

```

// The following call resolves to:
// print_type_and_value<const string&>(const string& && t)
// Which collapses to:
// print_type_and_value<const string&>(const string& t)
const string s2("second");
print_type_and_value(s2);

// The following call resolves to:
// print_type_and_value<string&&>(string&& t)
print_type_and_value(string("third"));

// The following call resolves to:
// print_type_and_value<const string&&>(const string&& t)
print_type_and_value(fourth());
}

```

В этом примере выводятся следующие данные:

```

print<T&>; first
print<const T&>; second
print<T&&>; third
print<const T&&>; fourth

```

Чтобы разрешить каждый вызов функции `print_type_and_value`, компилятор сначала выполняет выведение аргументов шаблона. Затем при подстановке аргументов шаблона, выведенных для типов параметра, компилятор применяет правила сворачивания ссылок. Например, при передаче локальной переменной `s1` в функцию `print_type_and_value` компилятор создает следующую сигнатуру функции:

```
print_type_and_value<string&>(string& && t)
```

Используя правила сворачивания ссылок, компилятор уменьшает сигнатуру до следующей:

```
print_type_and_value<string&>(string& t)
```

Затем эта версия функции `print_type_and_value` пересыпает свой параметр в требуемую специализированную версию метода `S::print`.

В следующей таблице приведены правила сворачивания ссылок для выведения типа аргументов шаблонов:

РАЗВЕРНУТЫЙ ТИП	СВЕРНУТЫЙ ТИП
<code>T& &</code>	<code>T&</code>
<code>T& &&</code>	<code>T&</code>
<code>T&& &</code>	<code>T&</code>
<code>T&& &&</code>	<code>T&&</code>

Вывод аргументов шаблонов — это важный элемент реализации точной пересылки. Более подробно точная пересылка рассматривается выше в подразделе "Точная пересылка" этого раздела.

Итоги

Ссылки `rvalue` различают значения `lvalue` и `rvalue`. Они помогают повысить производительность

приложений, устранивая необходимость в ненужных операциях выделения памяти и копирования. Они также позволяют создавать одну версию функции, принимающую произвольные аргументы и пересылающую их в другую функцию, как если бы эта другая функция вызывалась напрямую.

См. также

[Выражения с унарными операторами](#)

[Декларатор ссылки Lvalue: &](#)

[Значения lvalue и rvalue](#)

[Конструкторы перемещения и операторы присваивания перемещением \(C++\)](#)

[Стандартная библиотека C++](#)

Аргументы функции ссылочного типа

12.11.2021 • 2 minutes to read

Вместо крупных объектов эффективнее бывает передавать функциям ссылки. Это позволяет компилятору передавать адрес объекта, сохраняя при этом синтаксис, который использовался бы для обращения к этому объекту. Рассмотрим следующий пример, в котором используется структура `Date`:

```
// reference_type_function_arguments.cpp
#include <iostream>

struct Date
{
    short Month;
    short Day;
    short Year;
};

// Create a date of the form DDDYYYY (day of year, year)
// from a Date.
long DateOfYear( Date& date )
{
    static int cDaysInMonth[] = {
        31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
    };
    long dateOfYear = 0;

    // Add in days for months already elapsed.
    for ( int i = 0; i < date.Month - 1; ++i )
        dateOfYear += cDaysInMonth[i];

    // Add in days for this month.
    dateOfYear += date.Day;

    // Check for leap year.
    if ( date.Month > 2 &&
        (( date.Year % 100 != 0 || date.Year % 400 == 0 ) &&
        date.Year % 4 == 0 ))
        dateOfYear++;

    // Add in year.
    dateOfYear *= 10000;
    dateOfYear += date.Year;

    return dateOfYear;
}

int main()
{
    Date date{ 8, 27, 2018 };
    long dateOfYear = DateOfYear(date);
    std::cout << dateOfYear << std::endl;
}
```

Приведенный выше код показывает, что доступ к членам структуры, передаваемым по ссылке, осуществляется с помощью оператора выбора члена `(.)`, а не оператора выделения члена указателя `(->)`.

Хотя аргументы, передаваемые как ссылочные типы, определяют синтаксис типов, не являющихся указателями, они поддерживают одну важную характеристику типов указателей: они являются изменяемыми, если не объявлены как `const`. Поскольку задача приведенного выше кода заключается не

в том, чтобы изменить объект `date`, более подходящим будет следующий прототип функции:

```
long DateOfYear( const Date& date );
```

Этот прототип гарантирует, что функция `DateOfYear` не изменит его аргумент.

Любая функция, заданная как принимающая ссылочный тип, может принять объект того же типа в своем месте, так как существует стандартное преобразование из `TypeName` в `TypeName &`.

См. также раздел

[Справочные материалы](#)

Возвращаемые значения функции ссылочного типа

12.11.2021 • 2 minutes to read

Функции можно объявить, чтобы они возвращали ссылочный тип. Существует две причины создания такого объявления.

- Возвращаемая информация представляет собой настолько крупный объект, что возврат ссылки является более эффективным, чем возврат копии.
- Тип функции должен представлять собой I-значение.
- Объект, на который указывает ссылка, не выйдет из области видимости при возврате управления функцией.

Так же, как можно повысить эффективность передачи больших объектов *в* функции по ссылке, также может быть более эффективным возврат больших объектов *из* функций по ссылке. Протокол возврата ссылок устраниет необходимость копировать объект во временное хранилище перед возвратом.

Типы возврата ссылок также могут оказаться полезными, если результатом функции должно быть I-значение. Большинство перегруженных операторов относятся к этой категории, в частности оператор присваивания. Перегруженные операторы рассматриваются в [перегруженных операторах](#).

Пример

Рассмотрим пример `Point`.

```

// refType_function_returns.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;

class Point
{
public:
// Define "accessor" functions as
// reference types.
unsigned& x();
unsigned& y();

private:
// Note that these are declared at class scope:
unsigned obj_x;
unsigned obj_y;
};

unsigned& Point :: x()
{
return obj_x;
}

unsigned& Point :: y()
{
return obj_y;
}

int main()
{
Point ThePoint;
// Use x() and y() as l-values.
ThePoint.x() = 7;
ThePoint.y() = 9;

// Use x() and y() as r-values.
cout << "x = " << ThePoint.x() << "\n"
<< "y = " << ThePoint.y() << "\n";
}

```

Выходные данные

```

x = 7
y = 9

```

Обратите внимание, что функции `x` и `y` объявляются как типы возвращаемых ссылок. Эти функции можно использовать на любой стороне оператора присваивания.

Также обратите внимание, что в функции `main` объект `ThePoint` остается в области видимости, поэтому его ссылочные элементы продолжают действовать, и к ним можно получить безопасный доступ.

Объявления ссылочных типов должны содержать инициализаторы. Исключение составляют следующие случаи.

- Явное `extern` объявление
- Объявление члена класса
- Объявление в классе
- Объявление аргумента в адрес функции или типа возвращаемого значения для функции

Предупреждение при возвращении адреса локальной переменной

Если объект объявляется в локальной области видимости, он будет уничтожен, когда функция вернет ссылку. Возвращенная этому объекту ссылка может нарушить доступ в среде выполнения, поскольку вызывающий объект попытается использовать пустую ссылку.

```
// C4172 means Don't do this!!!
Foo& GetFoo()
{
    Foo f;
    ...
    return f;
} // f is destroyed here
```

Компилятор выдает предупреждение в этом случае:

`warning C4172: returning address of local variable or temporary`. В простых программах доступ может случайно сохраниться, если ссылка будет использована вызывающим объектом до перезаписи соответствующей области памяти. Однако это чистая случайность. Обратите внимание на предупреждение.

См. также раздел

[Справочные материалы](#)

Ссылки на указатели

12.11.2021 • 2 minutes to read

Ссылки на указатели можно объявлять так же, как ссылки на объекты. Ссылка на указатель является изменяемым значением, которое используется как стандартный указатель.

Пример

В этом примере кода показано различие между использованием указателя на указатель и ссылки на указатель.

Функции `Add1` и `Add2` функционально эквивалентны, хотя они не называются аналогичным образом. Разница заключается в том, что `Add1` использует двойное косвенное обращение, но `Add2` использует удобство ссылки на указатель.

```
// references_to_pointers.cpp
// compile with: /EHsc

#include <iostream>
#include <string>

// C++ Standard Library namespace
using namespace std;

enum {
    sizeOfBuffer = 132
};

// Define a binary tree structure.
struct BTTree {
    char *szText;
    BTTree *Left;
    BTTree *Right;
};

// Define a pointer to the root of the tree.
BTTree *btRoot = 0;

int Add1( BTTree **Root, char *szToAdd );
int Add2( BTTree*& Root, char *szToAdd );
void PrintTree( BTTree* btRoot );

int main( int argc, char *argv[] ) {
    // Usage message
    if( argc < 2 ) {
        cerr << "Usage: " << argv[0] << " [1 | 2]" << "\n";
        cerr << "\nwhere:\n";
        cerr << "1 uses double indirection\n";
        cerr << "2 uses a reference to a pointer.\n";
        cerr << "\nInput is from stdin. Use ^Z to terminate input.\n";
        return 1;
    }

    char *szBuf = new char[sizeOfBuffer];
    if (szBuf == NULL) {
        cerr << "Out of memory!\n";
        return -1;
    }

    // Read a text file from the standard input device and
```

```

// build a binary tree.
while( !cin.eof() )
{
    cin.get( szBuf, sizeOfBuffer, '\n' );
    cin.get();

    if ( strlen( szBuf ) ) {
        switch ( *argv[1] ) {
            // Method 1: Use double indirection.
            case '1':
                Add1( &btRoot, szBuf );
                break;
            // Method 2: Use reference to a pointer.
            case '2':
                Add2( btRoot, szBuf );
                break;
            default:
                cerr << "Illegal value '"
                    << *argv[1]
                    << "' supplied for add method.\n"
                    << "Choose 1 or 2.\n";
                return -1;
        }
    }
}

// Display the sorted list.
PrintTree( btRoot );
}

// PrintTree: Display the binary tree in order.
void PrintTree( BTREE* MybtRoot ) {
    // Traverse the left branch of the tree recursively.
    if ( MybtRoot->Left )
        PrintTree( MybtRoot->Left );

    // Print the current node.
    cout << MybtRoot->szText << "\n";

    // Traverse the right branch of the tree recursively.
    if ( MybtRoot->Right )
        PrintTree( MybtRoot->Right );
}

// Add1: Add a node to the binary tree.
//      Uses double indirection.
int Add1( BTREE **Root, char *szToAdd ) {
    if ( (*Root) == 0 ) {
        (*Root) = new BTREE;
        (*Root)->Left = 0;
        (*Root)->Right = 0;
        (*Root)->szText = new char[strlen( szToAdd ) + 1];
        strcpy_s((*Root)->szText, (strlen( szToAdd ) + 1), szToAdd );
        return 1;
    }
    else {
        if ( strcmp( (*Root)->szText, szToAdd ) > 0 )
            return Add1( &(*Root)->Left, szToAdd );
        else
            return Add1( &(*Root)->Right, szToAdd );
    }
}

// Add2: Add a node to the binary tree.
//      Uses reference to pointer
int Add2( BTREE*& Root, char *szToAdd ) {
    if ( Root == 0 ) {
        Root = new BTREE;
        Root->Left = 0;
        Root->Right = 0;
    }
}

```

```
Root->szText = 0;
Root->szText = new char[strlen( szToAdd ) + 1];
strcpy_s( Root->szText, (strlen( szToAdd ) + 1), szToAdd );
return 1;
}
else {
    if ( strcmp( Root->szText, szToAdd ) > 0 )
        return Add2( Root->Left, szToAdd );
    else
        return Add2( Root->Right, szToAdd );
}
}
```

Usage: references_to_pointers.exe [1 | 2]

where:

1 uses double indirection
2 uses a reference to a pointer.

Input is from stdin. Use ^Z to terminate input.

См. также раздел

[Справочные материалы](#)

Указатели (C++)

12.11.2021 • 2 minutes to read

Указатель — это переменная, в которой хранится адрес памяти объекта. Указатели широко используются в C и C++ в трех основных целях:

- чтобы выделить новые объекты в куче,
- передача функций в другие функции
- для итерации элементов в массивах или других структурах данных.

В программировании в стиле C для всех этих сценариев используются *необработанные указатели*.

Однако необработанные указатели являются источником многих серьезных ошибок программирования. Таким образом, их использование настоятельно не рекомендуется, за исключением случаев, когда они обеспечивают значительное преимущество в производительности, и неоднозначность, с которой указатель является *указателем-владельцем*, отвечающим за удаление объекта. Современный C++ предоставляет *интеллектуальные указатели* для выделения объектов, *итераторов* для обхода структур данных и *лямбда-выражений* для передачи функций. Используя эти средства языка и библиотеки вместо необработанных указателей, вы сделаете программу более безопасной, более простой в отладке и более удобной для понимания и сопровождения. Дополнительные сведения см. в разделе [интеллектуальные указатели, итераторы и лямбда-выражения](#).

Содержимое раздела

- [Необработанные указатели](#)
- [Константные и переменные указатели](#)
- [Операторы new и delete](#)
- [Смарт-указатели](#)
- [Практическое руководство. Создание и использование экземпляров unique_ptr](#)
- [Практическое руководство. Создание и использование экземпляров shared_ptr](#)
- [Практическое руководство. Создание и использование экземпляров weak_ptr](#)
- [Практическое руководство. Создание и использование экземпляров CComPtr и CComQIPtr](#)

См. также раздел

[Итераторы](#)

[Лямбда-выражения](#)

Необработанные указатели (C++)

12.11.2021 • 8 minutes to read

Указатель — это тип переменной. Он сохраняет адрес объекта в памяти и используется для доступа к этому объекту. *Необработанный указатель* — это указатель, время существования которого не управляется инкапсулированным объектом, например [интеллектуальным указателем](#). Необработанному указателю может быть присвоен адрес другой переменной, не являющейся указателем, или ему может быть присвоено значение `nullptr`. Указатель, которому не присвоено значение, содержит случайные данные.

Указатель может также быть *разыменован* для получения значения объекта, на который он указывает. *Оператор доступа к членам* предоставляет доступ к членам объекта.

```
int* p = nullptr; // declare pointer and initialize it
                  // so that it doesn't store a random address
int i = 5;
p = &i; // assign pointer to address of object
int j = *p; // dereference p to retrieve the value at its address
```

Указатель может указывать на типизированный объект или на `void`. Когда программа выделяет объект в [куче](#) в памяти, он получает адрес этого объекта в виде указателя. Такие указатели называются *указателями-владельцами*. Для явного освобождения выделенного в куче объекта, когда он больше не нужен, необходимо использовать указатель-владелец (или его копию). Сбой освобождения памяти приводит к *утечке памяти* и отображению этого расположения памяти, недоступного для любой другой программы на компьютере. Память, выделенная с помощью `new`, должна быть освобождена с помощью `delete` (`delete []` или). Дополнительные сведения см. в разделе [new delete операторы и](#).

```
MyClass* mc = new MyClass(); // allocate object on the heap
mc->print(); // access class member
delete mc; // delete object (please don't forget!)
```

Указатель (если он не объявлен как `const`) можно увеличить или уменьшить до точки в другом месте в памяти. Эта операция называется *арифметикой указателя*. Он используется в программировании в стиле С для итерации элементов в массивах или других структурах данных. `const` Невозможно создать указатель для указания на другое место в памяти, и в этом смысле похоже на [ссылку](#). Дополнительные сведения см. в разделе [const и временные указатели](#).

```
// declare a C-style string. Compiler adds terminating '\0'.
const char* str = "Hello world";

const int c = 1;
const int* pconst = &c; // declare a non-const pointer to const int
const int c2 = 2;
pconst = &c2; // OK pconst itself isn't const
const int* const pconst2 = &c;
// pconst2 = &c2; // Error! pconst2 is const.
```

В 64-разрядных операционных системах указатель имеет размер 64 бит. Размер указателя системы определяет, сколько памяти она может иметь. Все копии точки указателя на одно и то же место в памяти. Указатели (вместе со ссылками) широко используются в C++ для передачи больших объектов в функции и из них. Это связано с тем, что часто более эффективно копировать адрес объекта, чем

копировать весь объект. При определении функции укажите параметры-указатели, как `const` если бы вы не предполагали, что функция не изменит объект. Как правило, `const` ссылки являются предпочтительным способом передачи объектов в функции, если только значение объекта не может быть `nullptr`.

[Указатели на функции](#) позволяют передавать функции другим функциям и использоваться для обратных вызовов в программировании в стиле C. В современных C++ для этой цели используются [лямбда-выражения](#).

Доступ к инициализации и членам

В следующем примере показано, как объявить, инициализировать и использовать необработанный указатель. Он инициализируется с помощью `new`, чтобы указать объект, выделенный в куче, который необходимо явным образом `delete`. В примере также показано несколько опасностей, связанных с необработанными указателями. (Помните, что этот пример — программирование в стиле C, а не современные C++)

```
#include <iostream>
#include <string>

class MyClass
{
public:
    int num;
    std::string name;
    void print() { std::cout << name << ":" << num << std::endl; }
};

// Accepts a MyClass pointer
void func_A(MyClass* mc)
{
    // Modify the object that mc points to.
    // All copies of the pointer will point to
    // the same modified object.
    mc->num = 3;
}

// Accepts a MyClass object
void func_B(MyClass mc)
{
    // mc here is a regular object, not a pointer.
    // Use the "." operator to access members.
    // This statement modifies only the local copy of mc.
    mc.num = 21;
    std::cout << "Local copy of mc:";
    mc.print(); // "Erika, 21"
}

int main()
{
    // Use the * operator to declare a pointer type
    // Use new to allocate and initialize memory
    MyClass* pmc = new MyClass{ 108, "Nick" };

    // Prints the memory address. Usually not what you want.
    std::cout << pmc << std::endl;

    // Copy the pointed-to object by dereferencing the pointer
    // to access the contents of the memory location.
    // mc is a separate object, allocated here on the stack
    MyClass mc = *pmc;

    // Declare a pointer that points to mc using the addressof operator
```

```

// This code is provided just for showing the basic concept.
MyClass* pcopy = &mc;

// Use the -> operator to access the object's public members
pmc->print(); // "Nick, 108"

// Copy the pointer. Now pmc and pmc2 point to same object!
MyClass* pmc2 = pmc;

// Use copied pointer to modify the original object
pmc2->name = "Erika";
pmc->print(); // "Erika, 108"
pmc2->print(); // "Erika, 108"

// Pass the pointer to a function.
func_A(pmc);
pmc->print(); // "Erika, 3"
pmc2->print(); // "Erika, 3"

// Dereference the pointer and pass a copy
// of the pointed-to object to a function
func_B(*pmc);
pmc->print(); // "Erika, 3" (original not modified by function)

delete(pmc); // don't forget to give memory back to operating system!
// delete(pmc2); //crash! memory location was already deleted
}

```

Арифметические и массивные указатели

Указатели и массивы тесно связаны друг с друга. Когда массив передается в функцию по значению, он передается в качестве указателя на первый элемент. В следующем примере показаны следующие важные свойства указателей и массивов.

- `sizeof` оператор возвращает общий размер массива в байтах
- чтобы определить количество элементов, разделите общее число байтов на размер одного элемента.
- когда массив передается в функцию, он *декайлс* к типу указателя
- `sizeof` оператор, применяемый к указателю, возвращает размер указателя, 4 байта на x86 или 8 байт в x64

```

#include <iostream>

void func(int arr[], int length)
{
    // returns pointer size. not useful here.
    size_t test = sizeof(arr);

    for(int i = 0; i < length; ++i)
    {
        std::cout << arr[i] << " ";
    }
}

int main()
{
    int i[5]{ 1,2,3,4,5 };
    // sizeof(i) = total bytes
    int j = sizeof(i) / sizeof(i[0]);
    func(i,j);
}

```

Некоторые арифметические операции можно использовать для `const` неуказателей, чтобы они указывали

на другое место в памяти. Указатели увеличиваются и уменьшаются с помощью `++` и `+=` операторов, `-` и `--`. Этот метод можно использовать в массивах и особенно полезен в буферах нетипизированных данных. Значение `void * _ увеличивается на размер_` `char` (1 байта). Типизированный указатель увеличивается по размеру типа, на который он указывает.

В следующем примере показано, как можно использовать арифметические действия с указателями для доступа к отдельным пикселям точечного рисунка на Windows. Обратите внимание на `new` использование `delete` операторов и разыменование оператора.

```

#include <Windows.h>
#include <fstream>

using namespace std;

int main()
{
    BITMAPINFOHEADER header;
    header.biHeight = 100; // Multiple of 4 for simplicity.
    header.biWidth = 100;
    header.biBitCount = 24;
    header.biPlanes = 1;
    header.biCompression = BI_RGB;
    header.biSize = sizeof(BITMAPINFOHEADER);

    constexpr int bufferSize = 30000;
    unsigned char* buffer = new unsigned char[bufferSize];

    BITMAPFILEHEADER bf;
    bf.bfType = 0x4D42;
    bf.bfSize = header.biSize + 14 + bufferSize;
    bf.bfReserved1 = 0;
    bf.bfReserved2 = 0;
    bf.bfOffBits = sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFOHEADER); //54

    // Create a gray square with a 2-pixel wide outline.
    unsigned char* begin = &buffer[0];
    unsigned char* end = &buffer[0] + bufferSize;
    unsigned char* p = begin;
    constexpr int pixelWidth = 3;
    constexpr int borderWidth = 2;

    while (p < end)
    {
        // Is top or bottom edge?
        if ((p < begin + header.biWidth * pixelWidth * borderWidth)
            || (p > end - header.biWidth * pixelWidth * borderWidth))
            // Is left or right edge?
            || (p - begin) % (header.biWidth * pixelWidth) < (borderWidth * pixelWidth)
            || (p - begin) % (header.biWidth * pixelWidth) > ((header.biWidth - borderWidth) * pixelWidth))
        {
            *p = 0x0; // Black
        }
        else
        {
            *p = 0xC3; // Gray
        }
        p++; // Increment one byte sizeof(unsigned char).
    }

    ofstream wf(R"(box.bmp)", ios::out | ios::binary);

    wf.write(reinterpret_cast<char*>(&bf), sizeof(bf));
    wf.write(reinterpret_cast<char*>(&header), sizeof(header));
    wf.write(reinterpret_cast<char*>(begin), bufferSize);

    delete[] buffer; // Return memory to the OS.
    wf.close();
}

```

void* указатели

Указатель на `void` просто указывает на расположение необработанной памяти. Иногда необходимо использовать `void*` указатели, например при передаче между кодом C++ и функциями C.

Если типизированный указатель приводится к void указателю, содержимое области памяти не изменяется. Однако сведения о типе теряются, поэтому нельзя выполнять операции увеличения или уменьшения. Место в памяти может быть приведено к типу, например, от MyClass* до void* и обратно в MyClass*. Такие операции по сути подвержены ошибкам и нуждаются в отличных от них ошибках void. Современный C++ не рекомендует использовать void указатели практически во всех обстоятельствах.

```
//func.c
void func(void* data, int length)
{
    char* c = (char*)(data);

    // fill in the buffer with data
    for (int i = 0; i < length; ++i)
    {
        *c = 0x41;
        ++c;
    }
}

// main.cpp
#include <iostream>

extern "C"
{
    void func(void* data, int length);
}

class MyClass
{
public:
    int num;
    std::string name;
    void print() { std::cout << name << ":" << num << std::endl; }
};

int main()
{
    MyClass* mc = new MyClass{10, "Marian"};
    void* p = static_cast<void*>(mc);
    MyClass* mc2 = static_cast<MyClass*>(p);
    std::cout << mc2->name << std::endl; // "Marian"

    // use operator new to allocate untyped memory block
    void* pvoid = operator new(1000);
    char* pchar = static_cast<char*>(pvoid);
    for(char* c = pchar; c < pchar + 1000; ++c)
    {
        *c = 0x00;
    }
    func(pvoid, 1000);
    char ch = static_cast<char*>(pvoid)[0];
    std::cout << ch << std::endl; // 'A'
    operator delete(p);
}
```

Указатели на функции

В программировании в стиле С указатели функций используются в основном для передачи функций другим функциям. Этот метод позволяет вызывающему объекту настраивать поведение функции, не изменяя ее. В современных C++ [лямбда-выражения](#) обеспечивают те же возможности, что и более строгая типизация, и другие преимущества.

Объявление указателя на функцию указывает сигнатуру, которую должна иметь функция, на которую должен быть указан объект.

```
// Declare pointer to any function that...

// ...accepts a string and returns a string
string (*g)(string a);

// has no return value and no parameters
void (*x)();

// ...returns an int and takes three parameters
// of the specified types
int (*i)(int i, string s, double d);
```

В следующем примере показана функция `combine`, которая принимает в качестве параметра любую функцию, которая принимает `std::string` и возвращает `std::string`. В зависимости от функции, которая передается `combine`, она либо добавляет строку в начале, либо добавляет ее.

```
#include <iostream>
#include <string>

using namespace std;

string base {"hello world"};

string append(string s)
{
    return base.append(" ").append(s);
}

string prepend(string s)
{
    return s.append(" ").append(base);
}

string combine(string s, string(*g)(string a))
{
    return (*g)(s);
}

int main()
{
    cout << combine("from MSVC", append) << "\n";
    cout << combine("Good morning and", prepend) << "\n";
}
```

См. также

[Интеллектуальные указатели Оператор косвенного обращения: *](#)

[Оператор address-of: &](#)

[Возвращение к C++](#)

Указатели с ключевыми словами `const` и `volatile`

12.11.2021 • 3 minutes to read

Ключевые слова `const` и `volatile` изменяют, как обрабатываются указатели. `const` Ключевое слово указывает, что указатель не может быть изменен после инициализации; после этого указатель защищен от изменения.

`volatile` Ключевое слово указывает, что значение, связанное с указанным ниже именем, может быть изменено действиями, отличными от тех, которые указаны в пользовательском приложении. Таким образом, `volatile` ключевое слово полезно для объявления объектов в общей памяти, к которым могут обращаться несколько процессов или глобальных областей данных, используемых для связи с процедурами службы прерываний.

Если имя объявлено как `volatile`, компилятор повторно загружает значение из памяти при каждом доступе программы. Это значительно сокращает возможности оптимизации. Однако если состояние объекта может неожиданно изменяться, то это единственный способ гарантировать предсказуемую производительность программы.

Чтобы объявить объект, на который указывает указатель, как `const` или `volatile`, используйте объявление формы:

```
const char *cpch;  
volatile char *vpch;
```

Чтобы объявить значение указателя, то есть фактический адрес, хранящийся в указателе, — как `const` или `volatile`, используйте объявление формы:

```
char * const pchc;  
char * volatile pchv;
```

Язык C++ не допускает назначения, которые позволяют изменять объект или указатель, объявленный как `const`. Такие присваивания могут удалить информацию, с которой был объявлен объект или указатель, и тем самым подменить смысл исходного объявления. Рассмотрим следующее объявление:

```
const char cch = 'A';  
char ch = 'B';
```

Учитывая приведенные выше объявления двух объектов (`cch`, типа `const char` и `ch` типа `char`), допустимы следующие объявления и инициализации:

```
const char *pch1 = &cch;  
const char *const pch4 = &cch;  
const char *pch5 = &ch;  
char *pch6 = &ch;  
char *const pch7 = &ch;  
const char *const pch8 = &ch;
```

Следующие объявления и инициализации вызывают ошибки.

```
char *pch2 = &cch; // Error
char *const pch3 = &cch; // Error
```

В объявлении `pch2` задается указатель, при помощи которого может быть изменен постоянный объект, поэтому это объявление запрещено. Объявление `pch3` указывает, что указатель является константой, а не объектом; объявление не допускается по той же причине, что `pch2` объявление запрещено.

В следующих восьми примерах демонстрируется присваивание через указатель и изменение значения указателя для приведенных выше объявлений. Здесь мы предполагаем, что инициализация указателей `pch1` – `pch8` была выполнена без ошибок.

```
*pch1 = 'A'; // Error: object declared const
pch1 = &ch; // OK: pointer not declared const
*pch2 = 'A'; // OK: normal pointer
pch2 = &ch; // OK: normal pointer
*pch3 = 'A'; // OK: object not declared const
pch3 = &ch; // Error: pointer declared const
*pch4 = 'A'; // Error: object declared const
pch4 = &ch; // Error: pointer declared const
```

Указатели, объявленные как `volatile`, или как сочетание `const` и `volatile`, подчиняются тем же правилам.

Указатели на `const` объекты часто используются в объявлениях функций следующим образом:

```
errno_t strcpy_s( char *strDestination, size_t numberOfElements, const char *strSource );
```

В предыдущем операторе объявляется функция, `strcpy_s`, где два из трех аргументов имеют тип `pointer char`. Поскольку аргументы передаются по ссылке, а не по значению, функция может быть бесплатной, чтобы изменить обе функции `strDestination` и `strSource`. Если `strSource` они не были объявлены как `const`. Объявление `strSource` как `const` гарантирует вызывающий объект, который `strSource` не может быть изменен вызываемой функцией.

NOTE

Поскольку существует стандартное преобразование из `TypeName*` в `const TypeName*`, допускается передача аргумента типа `char *` в `strcpy_s`. Однако обратная не имеет значения true; не существует неявного преобразования для удаления `const` атрибута из объекта или указателя.

`const` Указатель данного типа может быть назначен указателю того же типа. Однако указатель, который не `const` может быть назначен `const` указателю. В следующем коде показано одно верное и одно неверное присваивание.

```
// const_pointer.cpp
int *const cpObject = 0;
int *pObject;

int main() {
    pObject = cpObject;
    cpObject = pObject; // C3892
}
```

В следующем примере показано, как объявить объект как `const`, когда имеется указатель на объект.

```
// const_pointer2.cpp
struct X {
    X(int i) : m_i(i) { }
    int m_i;
};

int main() {
    // correct
    const X cx(10);
    const X * pcx = &cx;
    const X ** ppcx = &pcx;

    // also correct
    X const cx2(20);
    X const * pcx2 = &cx2;
    X const ** ppcx2 = &pcx2;
}
```

См. также раздел

[Указатели Необработанные указатели](#)

Операторы `new` и `delete`

12.11.2021 • 5 minutes to read

C++ поддерживает динамическое выделение и освобождение объектов с помощью `new` операторов и `delete`. Эти операторы выделяют память для объектов из пула, называемого свободным хранилищем. `new` Оператор вызывает специальную функцию `operator new`, и `delete` оператор вызывает специальную функцию `operator delete`.

`new` Функция в стандартной библиотеке C++ поддерживает поведение, заданное в стандарте C++, что вызывает `std::bad_alloc` исключение в случае сбоя выделения памяти. Если вы по-прежнему хотите использовать не вызывающую версию `new`, свяжите программу с `nothrownew.obj`. Однако при компоновке с параметром `nothrownew.obj` по умолчанию `operator new` в стандартной библиотеке C++ больше не работает.

Список файлов библиотеки в библиотеке времени выполнения C и стандартной библиотеке C++ см. в разделе [функции библиотеки CRT](#).

`new` Оператор

Компилятор преобразует инструкцию, такую как `this`, в вызов функции `operator new`:

```
char *pch = new char[BUFFER_SIZE];
```

Если запрос имеет нулевые байты хранилища, функция `operator new` возвращает указатель на отдельный объект. То есть повторные вызовы `operator new` возвращают разные указатели. Если недостаточно памяти для запроса на выделение, `operator new` создает `std::bad_alloc` исключение. Или возвращает, `nullptr`. Если вы связались с поддержкой без вызова `operator new`.

Можно написать подпрограммы, которая пытается освободить память и повторить выделение памяти. Дополнительные сведения см. на веб-сайте [_set_new_handler](#). Дополнительные сведения о схеме восстановления см. в разделе [Обработка нехватки памяти](#).

`operator new` В следующей таблице описаны две области для функций.

Область действия для `operator new` функций

ОПЕРАТОР	ОБЛАСТЬ
<code>::operator new</code>	Глобальный
<code>имя класса** ::operator new **</code>	Класс

Первый аргумент для `operator new` должен иметь тип `size_t`, определенный в `<stddef.h>`, а тип возвращаемого значения — Always `void*`.

Глобальная `operator new` функция вызывается, когда `new` оператор используется для выделения объектов встроенных типов, объектов типа класса, которые не содержат определяемых пользователем `operator new` функций, и массивов любого типа. Если `new` оператор используется для выделения объектов типа класса, в котором `operator new` определен объект, `operator new` вызывается этот класс.

`operator new` Функция, определенная для класса, является статической функцией-членом (которая не может быть виртуальной), которая скрывает глобальную `operator new` функцию для объектов этого типа класса. Рассмотрим случай, когда `new` используется для выделения и установки памяти для заданного значения:

```
#include <malloc.h>
#include <memory.h>

class Blanks
{
public:
    Blanks(){}
    void *operator new( size_t stAllocateBlock, char chInit );
};

void *Blanks::operator new( size_t stAllocateBlock, char chInit )
{
    void *pvTemp = malloc( stAllocateBlock );
    if( pvTemp != 0 )
        memset( pvTemp, chInit, stAllocateBlock );
    return pvTemp;
}

// For discrete objects of type Blanks, the global operator new function
// is hidden. Therefore, the following code allocates an object of type
// Blanks and initializes it to 0xa5
int main()
{
    Blanks *a5 = new(0xa5) Blanks;
    return a5 != 0;
}
```

Аргумент, заданный в круглых скобках, `new` передается в `Blanks::operator new` качестве `chInit` аргумента. Однако глобальная `operator new` функция скрыта, что приводит к формированию ошибки следующим кодом:

```
Blanks *SomeBlanks = new Blanks;
```

Компилятор поддерживает массив `new` и операторы-члены `delete` в объявлении класса. Пример:

```
class MyClass
{
public:
    void * operator new[] (size_t)
    {
        return 0;
    }
    void operator delete[] (void*)
    {
    }
};

int main()
{
    MyClass *pMyClass = new MyClass[5];
    delete [] pMyClass;
}
```

Обработка нехватки памяти

Тестирование на неудачное выделение памяти можно выполнить, как показано ниже.

```
#include <iostream>
using namespace std;
#define BIG_NUMBER 100000000
int main() {
    int *pI = new int[BIG_NUMBER];
    if( pI == 0x0 ) {
        cout << "Insufficient memory" << endl;
        return -1;
    }
}
```

Существует другой способ обработки запросов на выделение памяти, завершившихся сбоем. Напишите пользовательскую подсистему восстановления для решения такой ошибки, а затем зарегистрируйте функцию, вызывав `_set_new_handler` функцию времени выполнения.

delete Оператор

Память, выделенная динамически с помощью `new` оператора, может быть освобождена с помощью `delete` оператора. Оператор `delete` вызывает `operator delete` функцию, которая освобождает память в доступном пуле. Использование `delete` оператора также приводит к вызову деструктора класса (если он существует).

Существует глобальная функция и функции уровня класса `operator delete`. `operator delete` Для данного класса может быть определена только одна функция. Если она определена, то она скрывает глобальную `operator delete` функцию. Глобальная `operator delete` функция всегда вызывается для массивов любого типа.

Глобальная `operator delete` функция. Для глобальных `operator delete` функций и членов класса существуют две формы `operator delete` :

```
void operator delete( void * );
void operator delete( void *, size_t );
```

Для данного класса может присутствовать только одна из двух предыдущих форм. Первая форма принимает один аргумент типа `void *`, который содержит указатель на объект, который необходимо освободить. Вторая форма, размер освобождения, принимает два аргумента: первый — указатель на блок памяти для освобождения, а второй — число байтов для освобождения. Тип возвращаемого значения обеих форм — `void` (`operator delete` не может возвращать значение).

Цель второй формы — ускорить поиск нужной категории размера объекта для удаления. Эти сведения часто не хранятся рядом с самим выделением и, скорее всего, не кэшируются. Вторая форма полезна, когда `operator delete` функция из базового класса используется для удаления объекта производного класса.

`operator delete` Функция является статической, поэтому она не может быть виртуальной.

`operator delete` Функция подчиняется контролю доступа, как описано в разделе [Управление доступом к членам](#).

В следующем примере показаны определяемые пользователем `operator new` функции и, `operator delete` предназначенные для записи в журнал выделений и освобождений памяти:

```

#include <iostream>
using namespace std;

int fLogMemory = 0;      // Perform logging (0=no; nonzero=yes)?
int cBlocksAllocated = 0; // Count of blocks allocated.

// User-defined operator new.
void *operator new( size_t stAllocateBlock ) {
    static int fInOpNew = 0; // Guard flag.

    if ( fLogMemory && !fInOpNew ) {
        fInOpNew = 1;
        clog << "Memory block " << ++cBlocksAllocated
            << " allocated for " << stAllocateBlock
            << " bytes\n";
        fInOpNew = 0;
    }
    return malloc( stAllocateBlock );
}

// User-defined operator delete.
void operator delete( void *pvMem ) {
    static int fInOpDelete = 0; // Guard flag.
    if ( fLogMemory && !fInOpDelete ) {
        fInOpDelete = 1;
        clog << "Memory block " << cBlocksAllocated--
            << " deallocated\n";
        fInOpDelete = 0;
    }

    free( pvMem );
}

int main( int argc, char *argv[] ) {
    fLogMemory = 1; // Turn logging on
    if( argc > 1 )
        for( int i = 0; i < atoi( argv[1] ); ++i ) {
            char *pMem = new char[10];
            delete[] pMem;
        }
    fLogMemory = 0; // Turn logging off.
    return cBlocksAllocated;
}

```

Приведенный выше код можно использовать для обнаружения "утечки памяти", то есть памяти, выделенной в бесплатном хранилище, но не освобожденной. Чтобы обнаружить утечки, глобальные `new` операторы и `delete` переопределяются для подсчета выделения и освобождения памяти.

Компилятор поддерживает массив `new` и операторы-члены `delete` в объявлении класса. Пример:

```

// spec1_the_operator_delete_function2.cpp
// compile with: /c
class X {
public:
    void * operator new[] (size_t) {
        return 0;
    }
    void operator delete[] (void*) {}
};

void f() {
    X *pX = new X[5];
    delete [] pX;
}

```

Интеллектуальные указатели (современный C++)

12.11.2021 • 6 minutes to read

В современном программировании на C++ Стандартная библиотека содержит *смарт-указатели*, которые позволяют гарантировать, что программы свободны от памяти, утечки ресурсов и являются надежными.

Использование интеллектуальных указателей

Смарт-указатели определяются в `std` пространстве имен в `<memory>` файле заголовка. Они крайне важны для `RAll` или *получения ресурса* — это идиома программирования инициализации. Главная задача этой идиомы — обеспечить, чтобы одновременно с получением ресурса производилась инициализация объекта, чтобы все ресурсы для объекта создавались и подготавливались в одной строке кода. На практике основным принципом `RAll` является предоставление владения любым ресурсом в куче (например, динамически выделенной памятью или дескрипторами системных объектов) объекту, выделенному стеком, деструктор которого содержит код для удаления или освобождения ресурса, а также весь связанный код очистки.

В большинстве случаев при инициализации необработанного указателя или дескриптора ресурса для указания на фактический ресурс следует сразу же передать указатель в интеллектуальный указатель. В современном C++ необработанные указатели используются только в небольших блоках кода с ограниченной областью, циклах или вспомогательных функциях, когда важна производительность и вероятность проблем с владением низкая.

В следующем примере сравниваются объявления необработанного и интеллектуального указателей.

```
void UseRawPointer()
{
    // Using a raw pointer -- not recommended.
    Song* pSong = new Song(L"Nothing on You", L"Bruno Mars");

    // Use pSong...

    // Don't forget to delete!
    delete pSong;
}

void UseSmartPointer()
{
    // Declare a smart pointer on stack and pass it the raw pointer.
    unique_ptr<Song> song2(new Song(L"Nothing on You", L"Bruno Mars"));

    // Use song2...
    wstring s = song2->duration_;
    //...

} // song2 is deleted automatically here.
```

Как показано в примере, интеллектуальный указатель — это шаблон класса, который объявляется в стеке и инициализируется с помощью необработанного указателя, указывающего на размещенный в куче объект. После инициализации интеллектуальный указатель становится владельцем необработанного указателя. Это означает, что интеллектуальный указатель отвечает за удаление памяти, заданной необработанным указателем. Деструктор интеллектуального указателя содержит вызов для удаления, и поскольку интеллектуальный указатель объявлен в стеке, его деструктор вызывается, как только

интеллектуальный указатель оказывается вне области, даже если исключение создается где-либо в другой части стека.

Доступ к инкапсулированному указателю осуществляется с помощью знакомых операторов указателя `->` и `*`, которые класс интеллектуального указателя перегружает для возврата инкапсулированного необработанного указателя.

Этот интеллектуальный указатель C++ напоминает создание объектов в таких языках, как C#: вы создаете объект, а система удаляет его в правильный момент. Отличие заключается в том, что отсутствует отдельный сборщик мусора, работающий в фоновом режиме; память управляет через стандартные правила области C++, чтобы среда выполнения функционировала быстрее и эффективнее.

IMPORTANT

Всегда создавайте интеллектуальные указатели в отдельной строке кода; ни в коем случае не делайте это в списке параметров, чтобы не произошла небольшая утечка ресурсов, связанная с определенными правилами выделения памяти спискам параметров.

В следующем примере показано, как `unique_ptr` тип интеллектуального указателя из стандартной библиотеки C++ можно использовать для инкапсуляции указателя на большой объект.

```
class LargeObject
{
public:
    void DoSomething(){}
};

void ProcessLargeObject(const LargeObject& lo){}
void SmartPointerDemo()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    //Call a method on the object
    pLarge->DoSomething();

    // Pass a reference to a method.
    ProcessLargeObject(*pLarge);

} //pLarge is deleted automatically when function block goes out of scope.
```

В этом примере показаны следующие важные шаги, необходимые для использования интеллектуальных указателей.

1. Объявите интеллектуальный указатель как автоматическую (локальную) переменную. (Не используйте `new` `malloc` выражение или для самого интеллектуального указателя.)
2. В параметре типа укажите тип, на который указывает инкапсулированный указатель.
3. Передайте необработанный указатель на `new` объект-ED в конструкторе интеллектуального указателя. (Некоторые служебные функции или конструкторы интеллектуальных указателей делают это автоматически.)
4. Используйте перегруженные операторы `->` и `*` для доступа к объекту.
5. Интеллектуальный указатель удаляет объект автоматически.

Интеллектуальные указатели разработаны для обеспечения максимальной эффективности в отношении

памяти и производительности. Например, единственный элемент данных в `unique_ptr` — это инкапсулированный указатель. Это означает, что размер `unique_ptr` точно такой же, как и у указателя — 4 или 8 байтов. Доступ к инкапсулированному указателю с помощью перегруженных операторов смарт-указателя `*` и `->` не значительно медленнее, чем доступ к необработанным указателям напрямую.

Интеллектуальные указатели имеют собственные функции-члены, доступ к которым осуществляется с помощью нотации "точка". Например, некоторые интеллектуальные указатели стандартной библиотеки C++ имеют функцию-член `reset`, которая освобождает владение указателем. Это полезно, когда нужно освободить память, принадлежащую интеллектуальному указателю, не дожидаясь, пока интеллектуальный указатель окажется вне области, как показано в следующем примере.

```
void SmartPointerDemo2()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    // Call a method on the object
    pLarge->DoSomething();

    // Free the memory before we exit function block.
    pLarge.reset();

    // Do some other work...
}
```

Интеллектуальные указатели обычно предоставляют способ прямого доступа к необработанному указателю. Интеллектуальные указатели стандартной библиотеки C++ имеют `get` функцию-член для этой цели и `comPtr` имеют открытый `p` член класса. Предоставляя прямой доступ к базовому указателю, можно использовать интеллектуальный указатель для управления памятью в своем коде и по-прежнему передавать необработанный указатель коду, который не поддерживает интеллектуальные указатели.

```
void SmartPointerDemo4()
{
    // Create the object and pass it to a smart pointer
    std::unique_ptr<LargeObject> pLarge(new LargeObject());

    // Call a method on the object
    pLarge->DoSomething();

    // Pass raw pointer to a legacy API
    LegacyLargeObjectFunction(pLarge.get());
}
```

Виды смарт-указателей

В следующем разделе приведены различные виды интеллектуальных указателей, доступные в среде программирования Windows, и приводится описание их использования.

Интеллектуальные указатели стандартной библиотеки C++

Используйте эти интеллектуальные указатели как основной вариант для инкапсуляции указателей на простые старые объекты C++ (POCO).

- `unique_ptr`

Обеспечивает, чтобы у базового указателя был только один владелец. Используйте как вариант по умолчанию для POCO, кроме случая, когда вы точно знаете, что требуется `shared_ptr`. Может

быть передан новому владельцу, но не может быть скопирован или сделан общим. Заменяет `auto_ptr`, использовать который не рекомендуется. Сравните с `boost::scoped_ptr`. `unique_ptr` является небольшим и эффективным; Размер — один указатель, который поддерживает ссылки `rvalue` для быстрой вставки и извлечения из коллекций стандартных библиотек C++. Файл заголовка: `<memory>`. Дополнительные сведения см. [в разделе инструкции. Создание и использование экземпляров `unique_ptr` и `unique_ptr` класса](#).

- `shared_ptr`

Интеллектуальный указатель с подсчитанными ссылками. Используйте, когда необходимо присвоить один необработанный указатель нескольким владельцам, например, когда копия указателя возвращается из контейнера, но требуется сохранить оригинал. Необработанный указатель не будет удален до тех пор, пока все владельцы `shared_ptr` не выйдут из области или не откажутся от владения. Размер — 2 указателя; один — для объекта и второй — для блока общего элемента управления, который содержит счетчик ссылок. Файл заголовка: `<memory>`.

Дополнительные сведения см. [в разделе инструкции. Создание и использование экземпляров `shared_ptr` и `shared_ptr` класса](#).

- `weak_ptr`

Интеллектуальный указатель для особых случаев использования с `shared_ptr`. `weak_ptr` предоставляет доступ к объекту, который принадлежит одному или нескольким экземплярам `shared_ptr`, но не участвует в подсчете ссылок. Используйте, когда требуется отслеживать объект, но не требуется, чтобы он оставался в активном состоянии. Требуется в некоторых случаях для разрыва циклических ссылок между экземплярами `shared_ptr`. Файл заголовка: `<memory>`.

Дополнительные сведения см. [в разделе инструкции. Создание и использование экземпляров `weak_ptr` и `weak_ptr` класса](#).

интеллектуальные указатели для COM-объектов (классическое программирование Windows)

При работе с COM-объектами создайте оболочку для указателей интерфейса в соответствующем типе интеллектуальных указателей. Библиотека шаблонных классов (ATL) определяет несколько интеллектуальных указателей для различных целей. Можно также использовать тип интеллектуального указателя `_com_ptr_t`, который компилятор использует при создании классов оболочки из файлов с расширением TLB. Это лучший вариант, если вы не хотите включать файлы заголовков ATL.

Класс `CComPtr`

Используйте, если невозможно использовать ATL. Выполняет подсчет ссылок с помощью методов `AddRef` и `Release`. Дополнительные сведения см. [в разделе инструкции. Создание и использование экземпляров `CComPtr` и `CComQIPtr`](#).

Класс `CComQIPtr`

Похож на `CComPtr`, но также предоставляет упрощенный синтаксис для вызова `QueryInterface` COM-объекта. Дополнительные сведения см. [в разделе инструкции. Создание и использование экземпляров `CComPtr` и `CComQIPtr`](#).

Класс `Kcomheapptr`

Интеллектуальный указатель на объекты, которые используют `CoTaskMemFree` для освобождения памяти.

Класс `Kcomgitptr`

Интеллектуальный указатель для интерфейсов, получаемых из глобальной таблицы интерфейсов (GIT).

Класс `_com_ptr_t`

По функциональности аналогичен `CComQIPtr`, но не зависит от заголовков ATL.

Интеллектуальные указатели ATL для объектов POCO

Помимо смарт-указателей для COM-объектов, ATL также определяет смарт-указатели и коллекции смарт-указателей для простых старых объектов C++ (POCO). в классическом Windows

в программировании эти типы являются полезными альтернативами для коллекций стандартной библиотеки C++, особенно если переносимость кода не требуется или если не требуется смешивать модели программирования стандартной библиотеки C++ и ATL.

[Класс Каутоптр](#)

Интеллектуальный указатель, принудительно реализующий уникальное владение путем переноса владения на копию. Сравним с нерекомендуемым классом `std::auto_ptr`.

[Класс Чеапптр](#)

Интеллектуальный указатель для объектов, которые выделены с помощью функции `malloc` C.

[Класс Каутовекторптр](#)

Интеллектуальный указатель для массивов, память для которых выделяется с помощью `new[]`.

[Класс Каутоптрапррай](#)

Класс, инкапсулирующий массив элементов `CAutoPtr`.

[Класс Каутоптраплист](#)

Класс, инкапсулирующий методы для управления списком узлов `CAutoPtr`.

См. также:

[Указатели](#)

[Справочник по языку C++](#)

[Стандартная библиотека C++](#)

Практическое руководство. Создание и использование экземпляров unique_ptr

12.11.2021 • 2 minutes to read

Unique_ptr не имеет общего доступа к указателю. Его нельзя скопировать в другую unique_ptr, передать по значению в функцию или использовать в любом алгоритме стандартной библиотеки C++, для которого требуется выполнить копирование. unique_ptr можно только переместить. Это означает, что владение ресурсов памяти переносится в другое unique_ptr и оригинал unique_ptr больше им не владеет. Рекомендуется ограничить объект одним владельцем, поскольку множественное владение усложняет логику программы. Поэтому, если требуется интеллектуальный указатель для простого объекта C++, используйте unique_ptr, а при создании unique_ptr используйте вспомогательную функцию make_unique.

Следующая схема иллюстрирует передачу прав собственности между двумя экземплярами unique_ptr.

```
auto ptrA = make_unique<Song>(L"Diana Krall", L"The Look of Love");
```

```
auto ptrB = std::move(ptrA);
```

ptrA

Song object

ptrB

Song object

unique_ptr определяется в <memory> заголовке в стандартной библиотеке C++. Она точно так же эффективна, что и необработанный указатель, и может использоваться в контейнерах стандартной библиотеки C++. Добавление unique_ptr экземпляров в контейнеры стандартной библиотеки C++ является эффективным, так как конструктор перемещения объекта unique_ptr исключает необходимость в операции копирования.

Пример 1

В следующем примере описывается порядок создания экземпляров unique_ptr и передачи их между функциями.

```

unique_ptr<Song> SongFactory(const std::wstring& artist, const std::wstring& title)
{
    // Implicit move operation into the variable that stores the result.
    return make_unique<Song>(artist, title);
}

void MakeSongs()
{
    // Create a new unique_ptr with a new object.
    auto song = make_unique<Song>(L"Mr. Children", L"Namonaki Uta");

    // Use the unique_ptr.
    vector<wstring> titles = { song->title };

    // Move raw pointer from one unique_ptr to another.
    unique_ptr<Song> song2 = std::move(song);

    // Obtain unique_ptr from function that returns by value.
    auto song3 = SongFactory(L"Michael Jackson", L"Beat It");
}

```

В этих примерах демонстрируется эта базовая характеристика `unique_ptr`: ее можно изменить, но не для копирования. "Перемещение" перемещает владельца в новый `unique_ptr` и сбрасывает старый `unique_ptr`.

Пример 2

В следующем примере описывается порядок создания экземпляров `unique_ptr` и их использования в векторе.

```

void SongVector()
{
    vector<unique_ptr<Song>> songs;

    // Create a few new unique_ptr<Song> instances
    // and add them to vector using implicit move semantics.
    songs.push_back(make_unique<Song>(L"B'z", L"Juice"));
    songs.push_back(make_unique<Song>(L"Namie Amuro", L"Funky Town"));
    songs.push_back(make_unique<Song>(L"Kome Kome Club", L"Kimi ga Iru Dake de"));
    songs.push_back(make_unique<Song>(L"Ayumi Hamasaki", L"Poker Face"));

    // Pass by const reference when possible to avoid copying.
    for (const auto& song : songs)
    {
        wcout << L"Artist: " << song->artist << L" Title: " << song->title << endl;
    }
}

```

Обратите внимание, что в диапазоне для цикла `unique_ptr` передается по ссылке. При попытке передачи по значению компилятор выдаст ошибку, поскольку конструктор копирования `unique_ptr` удален.

Пример 3

В следующем примере показана инициализация `unique_ptr`, являющегося членом класса.

```
class MyClass
{
private:
    // MyClass owns the unique_ptr.
    unique_ptr<ClassFactory> factory;
public:

    // Initialize by using make_unique with ClassFactory default constructor.
    MyClass() : factory ( make_unique<ClassFactory>())
    {}

    void MakeClass()
    {
        factory->DoSomething();
    }
};
```

Пример 4

[Make_unique](#) можно использовать для создания `unique_ptr` массива, но нельзя использовать `make_unique` для инициализации элементов массива.

```
// Create a unique_ptr to an array of 5 integers.
auto p = make_unique<int[]>(5);

// Initialize the array.
for (int i = 0; i < 5; ++i)
{
    p[i] = i;
    wcout << p[i] << endl;
}
```

Дополнительные примеры см. в разделе [make_unique](#).

См. также раздел

[Смарт-указатели \(современный C++\)](#)

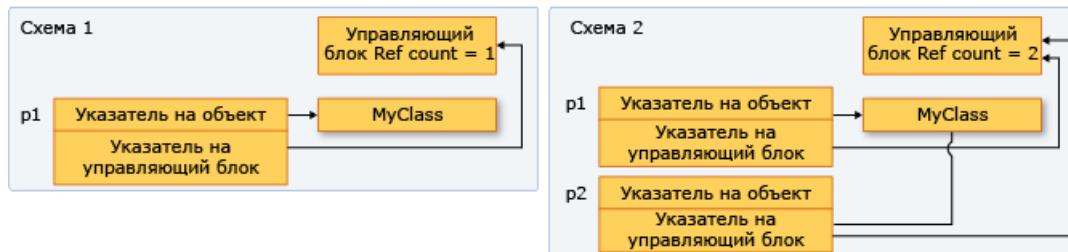
[make_unique](#)

Как создавать и использовать экземпляры shared_ptr

12.11.2021 • 6 minutes to read

Тип `shared_ptr` — это смарт-указатель в стандартной библиотеке C++, который предназначен для ситуаций, когда управлять временем существования объекта в памяти требуется нескольким владельцам. После инициализации указателя `shared_ptr` его можно копировать, передавать по значению в аргументах функций и присваивать другим экземплярам `shared_ptr`. Все экземпляры указывают на один и тот же объект и имеют общий доступ к одному "блоку управления", который увеличивает и уменьшает счетчик ссылок, когда указатель `shared_ptr` добавляется, выходит из области действия или сбрасывается. Когда счетчик ссылок достигает нуля, блок управления удаляет ресурс в памяти и самого себя.

На схеме ниже показано несколько экземпляров `shared_ptr`, указывающих на одно расположение в памяти.



Пример конфигурации

Во всех примерах далее предполагается, что вы включили необходимые заголовки и объявили необходимые типы следующим образом:

```

// shared_ptr-examples.cpp
// The following examples assume these declarations:
#include <algorithm>
#include <iostream>
#include <memory>
#include <string>
#include <vector>

struct MediaAsset
{
    virtual ~MediaAsset() = default; // make it polymorphic
};

struct Song : public MediaAsset
{
    std::wstring artist;
    std::wstring title;
    Song(const std::wstring& artist_, const std::wstring& title_) :
        artist{ artist_ }, title{ title_ } {}
};

struct Photo : public MediaAsset
{
    std::wstring date;
    std::wstring location;
    std::wstring subject;
    Photo(
        const std::wstring& date_,
        const std::wstring& location_,
        const std::wstring& subject_) :
        date{ date_ }, location{ location_ }, subject{ subject_ } {}
};

using namespace std;

int main()
{
    // The examples go here, in order:
    // Example 1
    // Example 2
    // Example 3
    // Example 4
    // Example 6
}

```

Пример 1

Когда ресурс в памяти создается впервые, по возможности используйте функцию `make_shared` для создания указателя `shared_ptr`. Функция `make_shared` безопасна в отношении исключений. Для выделения памяти под блок управления и ресурс используется один вызов, что снижает накладные расходы. Если вы не используете `make_shared`, то необходимо использовать явное `new` выражение для создания объекта перед его передачей в `shared_ptr` конструктор. В приведенном ниже примере представлены различные способы объявления и инициализации указателя `shared_ptr` вместе с новым объектом.

```
// Use make_shared function when possible.  
auto sp1 = make_shared<Song>(L"The Beatles", L"Im Happy Just to Dance With You");  
  
// Ok, but slightly less efficient.  
// Note: Using new expression as constructor argument  
// creates no named variable for other code to access.  
shared_ptr<Song> sp2(new Song(LLady Gaga", L"Just Dance"));  
  
// When initialization must be separate from declaration, e.g. class members,  
// initialize with nullptr to make your programming intent explicit.  
shared_ptr<Song> sp5(nullptr);  
//Equivalent to: shared_ptr<Song> sp5;  
//...  
sp5 = make_shared<Song>(L"Elton John", L"I'm Still Standing");
```

Пример 2

В приведенном ниже примере показано, как объявить и инициализировать экземпляры `shared_ptr`, которые будут совместно владеть объектом, память для которого уже выделена с помощью другого экземпляра `shared_ptr`. Предполагается, что `sp2` — это инициализированный указатель `shared_ptr`.

```
//Initialize with copy constructor. Increments ref count.  
auto sp3(sp2);  
  
//Initialize via assignment. Increments ref count.  
auto sp4 = sp2;  
  
//Initialize with nullptr. sp7 is empty.  
shared_ptr<Song> sp7(nullptr);  
  
// Initialize with another shared_ptr. sp1 and sp2  
// swap pointers as well as ref counts.  
sp1.swap(sp2);
```

Пример 3

Указатели `shared_ptr` также полезны при использовании алгоритмов копирования элементов в контейнеры стандартной библиотеки C++. Элемент можно заключить в указатель `shared_ptr`, а затем копировать его в другие контейнеры, учитывая при этом, что выделенная область памяти доступна только до тех пор, пока она требуется. В приведенном ниже примере показано, как использовать алгоритм `remove_copy_if` применительно к экземплярам `shared_ptr` в векторе.

```

vector<shared_ptr<Song>> v;

v.push_back(make_shared<Song>(L"Bob Dylan", L"The Times They Are A Changing"));
v.push_back(make_shared<Song>(L"Aretha Franklin", L"Bridge Over Troubled Water"));
v.push_back(make_shared<Song>(L"Thalida", L"Entre El Mar y Una Estrella"));

vector<shared_ptr<Song>> v2;
remove_copy_if(v.begin(), v.end(), back_inserter(v2), [] (shared_ptr<Song> s)
{
    return s->artist.compare(L"Bob Dylan") == 0;
});

for (const auto& s : v2)
{
    wcout << s->artist << L":" << s->title << endl;
}

```

Пример 4

Для приведения указателя `shared_ptr` можно использовать функции `dynamic_pointer_cast`, `static_pointer_cast` и `const_pointer_cast`. Эти функции похожи на `dynamic_cast` операторы, `static_cast` и `const_cast`. В приведенном ниже примере показано, как протестировать производный тип каждого элемента в векторе, содержащем указатель `shared_ptr` базовых классов, а затем скопировать элементы и отобразить сведения о них.

```

vector<shared_ptr<MediaAsset>> assets;

assets.push_back(shared_ptr<Song>(new Song(L"Himesh Reshammiya", L"Tera Surroor")));
assets.push_back(shared_ptr<Song>(new Song(L"Penaz Masani", L"Tu Dil De De")));
assets.push_back(shared_ptr<Photo>(new Photo(L"2011-04-06", L"Redmond, WA", L"Soccer field at Microsoft.")));

vector<shared_ptr<MediaAsset>> photos;

copy_if(assets.begin(), assets.end(), back_inserter(photos), [] (shared_ptr<MediaAsset> p) -> bool
{
    // Use dynamic_pointer_cast to test whether
    // element is a shared_ptr<Photo>.
    shared_ptr<Photo> temp = dynamic_pointer_cast<Photo>(p);
    return temp.get() != nullptr;
});

for (const auto& p : photos)
{
    // We know that the photos vector contains only
    // shared_ptr<Photo> objects, so use static_cast.
    wcout << "Photo location: " << (static_pointer_cast<Photo>(p))->location_ << endl;
}

```

Пример 5

Указатель `shared_ptr` можно передать в другую функцию описанными ниже способами.

- Передача `shared_ptr` по значению. При этом вызывается конструктор копий, увеличивается счетчик ссылок, и вызываемый объект становится владельцем. С этой операцией связаны некоторые накладные расходы, которые зависят от количества передаваемых объектов `shared_ptr`. Используйте этот вариант, если в соответствии с неявным или явным контрактом кода междузывающим и вызываемым объектами последний должен быть владельцем.

- Передача `shared_ptr` по ссылке или константной ссылке. В этом случае счетчик ссылок не увеличивается, а указатель доступен вызываемому объекту, пока вызывающий объект остается в области действия. Вызываемый объект может также создать указатель `shared_ptr` на основе ссылки и стать общим владельцем. Используйте этот вариант, когда вызываемый объект неизвестен вызывающему или когда необходимо передать указатель `shared_ptr`, избегая операции копирования из соображений производительности.
- Передача базового указателя или ссылки на базовый объект. Это позволяет вызываемому объекту использовать объект, но не становиться его общим владельцем и не продлять его время существования. Если вызываемая сторона создает указатель `shared_ptr` на основе необработанного указателя, новый указатель `shared_ptr` не зависит от исходного и не управляет базовым ресурсом. Используйте этот вариант, если в контракте между вызывающей и вызываемой сторонами явно указано, что вызывающая сторона сохраняет контроль над временем существования `shared_ptr`.
- При выборе способа передачи `shared_ptr`, установите, должна ли вызываемая сторона также быть владельцем базового ресурса. Владелец — это объект или функция, которые могут поддерживать существование базового ресурса до тех пор, пока он нужен. Если вызывающая сторона должна гарантировать, что вызываемая сторона может продлить существование указателя после того, как она перестанет существовать, используйте первый вариант. Если возможность продления времени существования вызываемой стороной не важна, передайте указатель по ссылке, чтобы вызываемая сторона скопировала его или не скопировала.
- Если нужно предоставить доступ к базовому указателю вспомогательной функции, которая использует его и возвращает управление до возврата управления вызывающей функцией, этой функции не требуется быть общим владельцем базового указателя. Ей просто необходим доступ к указателю в течение времени существования `shared_ptr` вызывающей стороны. В этом случае можно спокойно передать `shared_ptr` по ссылке либо передать необработанный указатель или ссылку на базовый объект. Это дает небольшой выигрыш в производительности и порой позволяет более ясно выразить назначение кода.
- Иногда, например в `std::vector<shared_ptr<T>>`, может быть необходимо передать каждый указатель `shared_ptr` в тело лямбда-выражения или в именованный объект функции. Если в лямбда-выражении или функции указатель не сохраняется, передайте `shared_ptr` по ссылке, чтобы не вызывать конструктор копий для каждого элемента.

Пример 6

В приведенном ниже примере показано, как `shared_ptr` перегружает различные операторы сравнения, чтобы обеспечить сравнение указателей в памяти, принадлежащей экземплярам `shared_ptr`.

```
// Initialize two separate raw pointers.  
// Note that they contain the same values.  
auto song1 = new Song(L"Village People", L"YMCA");  
auto song2 = new Song(L"Village People", L"YMCA");  
  
// Create two unrelated shared_ptrs.  
shared_ptr<Song> p1(song1);  
shared_ptr<Song> p2(song2);  
  
// Unrelated shared_ptrs are never equal.  
wcout << "p1 < p2 = " << std::boolalpha << (p1 < p2) << endl;  
wcout << "p1 == p2 = " << std::boolalpha << (p1 == p2) << endl;  
  
// Related shared_ptr instances are always equal.  
shared_ptr<Song> p3(p2);  
wcout << "p3 == p2 = " << std::boolalpha << (p3 == p2) << endl;
```

См. также

[Смарт-указатели \(современный C++\)](#)

Практическое руководство. Создание и использование экземпляров `weak_ptr`

12.11.2021 • 3 minutes to read

Иногда объект должен хранить способ доступа к базовому объекту `shared_ptr`, не вызывая увеличения счетчика ссылок. Как правило, такая ситуация возникает при наличии циклических ссылок между `shared_ptr` экземплярами.

Оптимальное проектирование заключается в том, чтобы не допустить совместного владения указателями, когда это возможно. Однако, если требуется совместное владение `shared_ptr` экземплярами, Избегайте циклических ссылок между ними. Если циклические ссылки нецелесообразны или еще предпочтительнее по какой-либо причине, используйте `weak_ptr`, чтобы предоставить одному или нескольким владельцам слабую ссылку на другую `shared_ptr`. С помощью `weak_ptr` можно создать объект `shared_ptr`, который объединяется с существующим набором связанных экземпляров, но только в том случае, если ресурс базовой памяти по-прежнему является допустимым. `weak_ptr` Сам по себе не участвует в подсчете ссылок и, следовательно, не может препятствовать переходу на нулевое значение счетчика ссылок. Однако можно использовать, `weak_ptr` чтобы попытаться получить новую копию объекта, `shared_ptr` с которым он был инициализирован. Если память уже была удалена, `weak_ptr` оператор `bool` возвращает значение `false`. Если память по-прежнему является допустимой, новый общий указатель увеличивает счетчик ссылок и гарантирует, что память будет действительной, пока `shared_ptr` переменная остается в области.

Пример

В следующем примере кода показан случай, когда `weak_ptr` используется для обеспечения правильного удаления объектов, имеющих циклические зависимости. При изучении примера предположим, что он был создан только после рассмотрения альтернативных решений. `Controller` Объекты представляют некоторые аспекты процесса компьютера и работают независимо друг от друга. Каждый контроллер должен иметь возможность запрашивать состояние других контроллеров в любое время, и каждый из них содержит частный `vector<weak_ptr<Controller>>` для этой цели. Каждый вектор содержит циклическую ссылку, и поэтому `weak_ptr` вместо используется экземпляр `shared_ptr`.

```
#include <iostream>
#include <memory>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

class Controller
{
public:
    int Num;
    wstring Status;
    vector<weak_ptr<Controller>> others;
    explicit Controller(int i) : Num(i), Status(L"On")
    {
        wcout << L"Creating Controller" << Num << endl;
    }

    ~Controller()
    {
```

```

        wcout << L"Destroying Controller" << Num << endl;
    }

    // Demonstrates how to test whether the
    // pointed-to memory still exists or not.
    void CheckStatuses() const
    {
        for_each(others.begin(), others.end(), [](weak_ptr<Controller> wp) {
            auto p = wp.lock();
            if (p)
            {
                wcout << L"Status of " << p->Num << " = " << p->Status << endl;
            }
            else
            {
                wcout << L"Null object" << endl;
            }
        });
    }
};

void RunTest()
{
    vector<shared_ptr<Controller>> v{
        make_shared<Controller>(0),
        make_shared<Controller>(1),
        make_shared<Controller>(2),
        make_shared<Controller>(3),
        make_shared<Controller>(4),
    };

    // Each controller depends on all others not being deleted.
    // Give each controller a pointer to all the others.
    for (int i = 0; i < v.size(); ++i)
    {
        for_each(v.begin(), v.end(), [&v, i](shared_ptr<Controller> p) {
            if (p->Num != i)
            {
                v[i]->others.push_back(weak_ptr<Controller>(p));
                wcout << L"push_back to v[" << i << "]: " << p->Num << endl;
            }
        });
    }

    for_each(v.begin(), v.end(), [](shared_ptr<Controller> &p) {
        wcout << L"use_count = " << p.use_count() << endl;
        p->CheckStatuses();
    });
}

int main()
{
    RunTest();
    wcout << L"Press any key" << endl;
    char ch;
    cin.getline(&ch, 1);
}

```

```
Creating Controller0
Creating Controller1
Creating Controller2
Creating Controller3
Creating Controller4
push_back to v[0]: 1
push_back to v[0]: 2
push_back to v[0]: 3
push_back to v[0]: 4
push_back to v[1]: 0
push_back to v[1]: 2
push_back to v[1]: 3
push_back to v[1]: 4
push_back to v[2]: 0
push_back to v[2]: 1
push_back to v[2]: 3
push_back to v[2]: 4
push_back to v[3]: 0
push_back to v[3]: 1
push_back to v[3]: 2
push_back to v[3]: 4
push_back to v[4]: 0
push_back to v[4]: 1
push_back to v[4]: 2
push_back to v[4]: 3
use_count = 1
Status of 1 = On
Status of 2 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 2 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 3 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 2 = On
Status of 4 = On
use_count = 1
Status of 0 = On
Status of 1 = On
Status of 2 = On
Status of 3 = On
Destroying Controller0
Destroying Controller1
Destroying Controller2
Destroying Controller3
Destroying Controller4
Press any key
```

В качестве эксперимента измените вектор `others` на `vector<shared_ptr<Controller>>`, а затем в выходных данных обратите внимание, что при возврате не вызываются деструкторы `RunTest`.

См. также раздел

[Смарт-указатели \(современный C++\)](#)

Практическое руководство. Создание и использование экземпляров CComPtr и CComQIPtr

12.11.2021 • 3 minutes to read

В классическом программировании Windows библиотеки часто реализуются как СОМ-объекты (или, более точно, как СОМ-серверы). Многие компоненты операционной системы Windows реализованы в виде СОМ-серверов, и большинство разработчиков предоставляют библиотеки в этой форме. Сведения об основах СОМ см. в разделе [Component Object Model \(COM\)](#).

При создании СОМ-объекта сохраните указатель на интерфейс в интеллектуальном указателе СОМ, который подсчитывает ссылки с помощью вызовов `AddRef` и `Release` в деструкторе. Если вы работаете с библиотекой ATL или библиотекой MFC, используйте интеллектуальный указатель `CComPtr`. В противном случае используйте `_com_ptr_t`. Поскольку эквивалент СОМ для `std::unique_ptr` отсутствует, применяйте эти интеллектуальные указатели в сценариях с одним и несколькими владельцами. `CComPtr` и `ComQIPtr` поддерживают операции перемещения, имеющие ссылки `rvalue`.

Пример: CComPtr

В следующем примере показано, как использовать `CComPtr` для создания СОМ-объекта и получения указателей на его интерфейсы. Обратите внимание, что для создания объекта используется функция-член `CComPtr::CoCreateInstance`, а не функция Win32 с таким же именем.

```

void CComPtrDemo()
{
    HRESULT hr = CoInitialize(NULL);

    // Declare the smart pointer.
    CComPtr pGraph;

    // Use its member function CoCreateInstance to
    // create the COM object and obtain the IGraphBuilder pointer.
    hr = pGraph.CoCreateInstance(CLSID_FilterGraph);
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Use the overloaded -> operator to call the interface methods.
    hr = pGraph->RenderFile(L"C:\\\\Users\\\\Public\\\\Music\\\\Sample Music\\\\Sleep Away.mp3", NULL);
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Declare a second smart pointer and use it to
    // obtain another interface from the object.
    CComPtr pControl;
    hr = pGraph->QueryInterface(IID_PPV_ARGS(&pControl));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Obtain a third interface.
    CComPtr pEvent;
    hr = pGraph->QueryInterface(IID_PPV_ARGS(&pEvent));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Use the second interface.
    hr = pControl->Run();
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Use the third interface.
    long evCode = 0;
    pEvent->WaitForCompletion(INFINITE, &evCode);

    CoUninitialize();

    // Let the smart pointers do all reference counting.
}

```

`CComPtr` и родственники являются частью библиотеки ATL и определяются в `<atlcomcli.h>`. `_com_ptr_t` объявлен в `<comip.h>`. Компилятор создает специализации `_com_ptr_t` при создании классов-оболочек для библиотек типов.

Пример: Ккомкипт

Кроме того, библиотека ATL предоставляет указатель `CComQIPtr` с более простым синтаксисом запросов к COM-объекту для получения новых интерфейсов. Однако рекомендуется использовать `CComPtr`, так как он поддерживает все возможности `CComQIPtr` и семантически более совместим с необработанными указателями на интерфейс COM. Если запрос на интерфейс выполняется с помощью `CComPtr`, указатель на новый интерфейс помещается в выходной параметр. Если вызов завершается ошибкой, возвращается значение `HRESULT`, которое является стандартным шаблоном COM. В случае с `CComQIPtr` возвращаемым значением является сам указатель, и при сбое вызова внутреннее возвращаемое значение `HRESULT` будет недоступно. В следующих двух строках показано различие механизмов обработки ошибок `CComPtr` и `CComQIPtr`.

```

// CComPtr with error handling:
CComPtr<IMediaControl> pControl;
hr = pGraph->QueryInterface(IID_PPV_ARGS(&pControl));
if(FAILED(hr)){ /*... handle hr error*/ }

// CComQIPtr with error handling
CComQIPtr<IMediaEvent> pEvent = pControl;
if(!pEvent){ /*... handle NULL pointer error*/ }

// Use the second interface.
hr = pControl->Run();
if(FAILED(hr)){ /*... handle hr error*/ }

```

Пример: IDispatch

`CComPtr` предоставляет специализацию для `IDispatch`, который позволяет сохранить указатели на компоненты автоматизации COM и вызывать методы в интерфейсе с помощью позднего связывания. `CComDispatchDriver` является определением типа `CComQIPtr<IDispatch, &IID_IDispatch>`, который неявно преобразуется в `CComPtr<IDispatch>`. Таким образом, когда любое из этих трех имен появляется в коде, оно считается эквивалентным `CComPtr<IDispatch>`. В примере ниже демонстрируется получение указателя на объектную модель Microsoft Word с помощью `CComPtr<IDispatch>`.

```

void COMAutomationSmartPointerDemo()
{
    CComPtr<IDispatch> pWord;
    CComQIPtr<IDispatch, &IID_IDispatch> pqi = pWord;
    CComDispatchDriver pDriver = pqi;

    HRESULT hr;
    _variant_t pOutVal;

    CoInitialize(NULL);
    hr = pWord.CoCreateInstance(L"Word.Application", NULL, CLSCTX_LOCAL_SERVER);
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Make Word visible.
    hr = pWord.PutPropertyByName(_bstr_t("Visible"), &_variant_t(1));
    if(FAILED(hr)){ /*... handle hr error*/ }

    // Get the Documents collection and store it in new CComPtr
    hr = pWord.GetPropertyByName(_bstr_t("Documents"), &pOutVal);
    if(FAILED(hr)){ /*... handle hr error*/ }

    CComPtr<IDispatch> pDocuments = pOutVal.pdispVal;

    // Use Documents to open a document
    hr = pDocuments.Invoke1 (_bstr_t("Open"),
    &_variant_t("c:\\users\\public\\documents\\sometext.txt"),&pOutVal);
    if(FAILED(hr)){ /*... handle hr error*/ }

    CoUninitialize();
}

```

См. также

[Смарт-указатели \(современный C++\)](#)

Обработка исключений в MSVC

12.11.2021 • 2 minutes to read

Исключение — это условие ошибки, возможно вне элемента управления программы, которое не позволяет продолжать выполнение программы по обычному пути выполнения. Некоторые операции, включая создание объектов, ввод-вывод файлов и вызовы функций из других модулей, являются потенциальными источниками исключений, даже если программа работает правильно. В надежном коде можно предвидеть и обработать исключения. Для обнаружения логических ошибок используйте утверждения, а не исключения (см. раздел [использование утверждений](#)).

Виды исключений

компилятор Microsoft C++ (MSVC) поддерживает три типа обработки исключений:

- [Обработка исключений C++](#)

Для большинства программ на C++ следует использовать обработку исключений C++. Он является типобезопасным и обеспечивает вызов деструкторов объектов во время очистки стека.

- [Структурированная обработка исключений](#)

Windows предоставляет собственный механизм обработки исключений, называемый структурной обработкой исключений (SEH). Не рекомендуется использовать для программирования C++ или MFC. Используйте SEH только в программах, написанных на языке С без использования MFC.

- [Исключения MFC](#)

Начиная с версии 3,0, в MFC использовались исключения C++. Он по-прежнему поддерживает старые макросы обработки исключений, которые похожи на исключения C++ в форме.

Рекомендации по смешению макросов MFC и исключений C++ см. в разделе [исключения: использование макросов MFC и исключений C++](#).

Используйте параметр компилятора [/EH](#), чтобы указать модель обработки исключений для использования в проекте C++. Стандартная обработка исключений C++ ([/EHsc](#)) используется по умолчанию в новых проектах C++ в Visual Studio.

Мы не рекомендуем смешивать механизмы обработки исключений. Например, не используйте исключения C++ с структурированной обработкой исключений. Использование обработки исключений C++ исключительно делает ваш код более переносимым и позволяет обрабатывать исключения любого типа. Дополнительные сведения о недостатках структурированной обработки исключений см. в разделе [структурная обработка исключений](#).

Содержимое раздела

- [Современные рекомендации по C++ для исключений и обработки ошибок](#)
- [Практическое руководство. Разработка с учетом безопасности исключений](#)
- [Практическое руководство. Интерфейс между кодом с исключениями и без исключений](#)
- [Операторы try, catch и throw](#)
- [Как оцениваются блоки catch](#)

- Исключения и очистка стека
- Спецификации исключений
- noexcept
- Необработанные исключения C++
- Смешивание исключений C (структурированная) и C++
- Обработка структурированных исключений (C/C++)

См. также

[Справочник по языку C++](#)

[Обработка исключений в 64-разрядных системах](#)

[Обработка исключений \(C++/CLI и C++/CX\)](#)

Современные рекомендации по C++ для исключений и обработки ошибок

12.11.2021 • 5 minutes to read

В современных C++ в большинстве случаев предпочтительным способом сообщить и обрабатывать ошибки как логические ошибки, так и ошибки времени выполнения — использовать исключения. Особенно это касается того, что стек может содержать несколько вызовов функций между функцией, которая обнаруживает ошибку, и функцией, которая имеет контекст для ее устранения. Исключения предоставляют формальный, четко определенный способ для кода, который обнаруживает ошибки для передачи информации вверх по стеку вызовов.

Использовать исключения для кода исключительного пользования

Ошибки программы часто делятся на две категории: логические ошибки, вызванные ошибками программирования, например, ошибкой «индекс вне диапазона». И ошибки времени выполнения, которые выходят за рамки управления программистом, например "ошибка Сетевая служба недоступна". В программировании в стиле C и в COM Управление отчетами об ошибках осуществляется либо путем возвращения значения, представляющего код ошибки, либо кода состояния для конкретной функции, либо путем установки глобальной переменной, которую вызывающий может дополнительно получить после каждого вызова функции, чтобы проверить, были ли обнаружены ошибки. Например, при программировании COM для передачи ошибок вызывающему объекту используется возвращаемое значение HRESULT. API-интерфейс Win32 содержит `GetLastError` функцию для получения последней ошибки, о которой сообщил стек вызовов. В обоих случаях для распознавания кода и реагирования на него требуется вызывающая сторона. Если вызывающий объект не обрабатывает код ошибки явным образом, программа может аварийно завершить работу без предупреждения. Или можно продолжить выполнение с использованием неверных данных и получить неверные результаты.

Исключения являются предпочтительными в современных C++ по следующим причинам:

- Исключение приводит к тому, что вызывающий код распознает состояние ошибки и обрабатывает его. Необработанные исключения останавливают выполнение программы.
- Исключение переходит к точке в стеке вызовов, которая может справиться с ошибкой. Промежуточные функции могут позволить распространить исключение. Они не должны координироваться с другими уровнями.
- Механизм обратной записи исключений уничтожает все объекты в области действия после возникновения исключения в соответствии с четко определенными правилами.
- Исключение позволяет четко отделить код, который определяет ошибку, и код, обрабатывающий ошибку.

В следующем упрощенном примере показан синтаксис, необходимый для генерации и перехвата исключений в C++.

```

#include <stdexcept>
#include <limits>
#include <iostream>

using namespace std;

void MyFunc(int c)
{
    if (c > numeric_limits< char> ::max())
        throw invalid_argument("MyFunc argument too large.");
    //...
}

int main()
{
    try
    {
        MyFunc(256); //cause an exception to throw
    }

    catch (invalid_argument& e)
    {
        cerr << e.what() << endl;
        return -1;
    }
    //...
    return 0;
}

```

Исключения в C++ похожи на такие языки, как C# и Java. В `try` блоке при *возникновении* исключения оно будет *перехвачено* первым связанным `catch` блоком, тип которого совпадает с типом исключения. Иными словами, выполнение переходит от `throw` оператора к `catch` оператору. Если подходящий блок `catch` не найден, `std::terminate` вызывается метод и программа завершает работу. В C++ может быть вызван любой тип; Однако рекомендуется создавать тип, прямо или косвенно производный от `std::exception`. В предыдущем примере тип исключения `invalid_argument` определен в стандартной библиотеке в `<stdexcept>` файле заголовка. C++ не предоставляет или не требует `finally` блока, чтобы гарантировать освобождение всех ресурсов при возникновении исключения. Идиома получения ресурсов — инициализация (RAII), которая использует интеллектуальные указатели, предоставляет необходимые функции для очистки ресурсов. Дополнительные сведения см. в [разделе руководство](#).

проектирование безопасности исключений. Дополнительные сведения о механизме развертывания стека C++ см. в разделе [исключения и очистка стека](#).

Основные рекомендации

Надежная обработка ошибок является сложной задачей в любом языке программирования. Хотя исключения предоставляют несколько функций, которые поддерживают хорошую обработку ошибок, они не могут выполнить всю работу. Чтобы реализовать преимущества механизма исключения, помните об исключениях при проектировании кода.

- Используйте утверждения, чтобы проверить наличие ошибок, которые не должны возникать. Используйте исключения для проверки ошибок, которые могут возникать, например, ошибок при проверке входных данных для параметров открытых функций. Дополнительные сведения см. в разделе [исключения и утверждения](#).
- Используйте исключения, если код, обрабатывающий ошибку, отделен от кода, который обнаруживает ошибку одним или несколькими промежуточными вызовами функций. Рассмотрите возможность использования кодов ошибок в циклах, критических для производительности, когда код, обрабатывающий ошибку, тесно связан с кодом, который ее обнаруживает.

- Для каждой функции, которая может выдавать или распространять исключение, следует предоставить одно из трех гарантий исключений: строгая гарантия, Базовая гарантия или "Throw" (Except). Дополнительные сведения см. в разделе [руководство. проектирование безопасности исключений](#).
- Вызывайте исключения по значению, перехватите их по ссылке. Не перехватывайте объекты, которые не могут быть обработаны.
- Не используйте спецификации исключений, которые являются устаревшими в C++ 11. Дополнительные сведения см. в разделе [спецификации исключений и noexcept](#) раздел.
- Используйте типы исключений стандартной библиотеки при их применении. Наследовать пользовательские типы исключений от иерархии [exception классов](#).
- Не разрешать исключения для экранирования из деструкторов или функций освобождения памяти.

Исключения и производительность

Механизм исключения имеет минимальные затраты на производительность, если исключение не создается. При возникновении исключения стоимость прохода стека и его очистки приблизительно сравнима с затратами на вызов функции. Дополнительные структуры данных необходимы для контроля стека вызовов после того, как был выполнен блок, и при возникновении исключения требуются дополнительные инструкции для очистки стека. Однако в большинстве случаев затраты на производительность и объем памяти не являются существенными. Негативное воздействие исключений на производительность может быть значительным только для систем с ограниченным объемом памяти. Кроме того, в циклах, критических с точки зрения производительности, часто возникает ошибка, и существует тесная связь между кодом и его обработкой. В любом случае невозможно понять фактическую стоимость исключений без профилирования и измерения. Даже в редких случаях, когда стоимость существенна, можно взвесить ее на более высокую правильность, упростить обслуживание и другие преимущества, предоставляемые хорошо спроектированной политикой исключений.

Исключения и утверждения

Исключения и утверждения — это два отдельных механизма для обнаружения ошибок во время выполнения в программе. Используйте `assert` инструкции для проверки условий во время разработки, которые никогда не должны быть истинными, если весь код правильный. Нет никакой точки в обработке такой ошибки с помощью исключения, так как эта ошибка указывает на то, что что-то в коде должно быть исправлено. Он не представляет условие, которое программа может восстанавливать из среды выполнения. `assert` Останавливает выполнение на инструкции, чтобы можно было проверить состояние программы в отладчике. Исключение продолжит выполнение из первого соответствующего обработчика `catch`. Используйте исключения для проверки ошибок, которые могут возникнуть во время выполнения, даже если код правильный, например "файл не найден" или "недостаточно памяти". Исключения могут обработаны, даже если восстановление просто выводит сообщение в журнал и завершает программу. Всегда проверяйте аргументы для открытых функций с помощью исключений. Даже если функция бесплатна, вы можете не иметь полного контроля над аргументами, которые пользователь может передать ей.

ИСКЛЮЧЕНИЯ C++ И Windows ИСКЛЮЧЕНИЯ SEH

программы С и C++ могут использовать механизм структурированной обработки исключений (SEH) в Windows операционной системе. Понятия SEH похожи на объекты в исключениях C++, за исключением того, что SEH `try` использует `_except` конструкции, и, а `_finally` не `try` и `catch`. В компиляторе Microsoft C++ (MSVC) исключения C++ реализуются для SEH. Однако при написании кода C++

используйте синтаксис исключения C++.

Дополнительные сведения о SEH см. в разделе [структурированная обработка исключений \(C/C++\)](#).

Спецификации исключений и `noexcept`

Спецификации исключений были введены в C++ как способ указания исключений, которые может вызывать функция. Однако спецификации исключений выдают проблемы на практике и являются устаревшими в стандарте "черновик C++ 11". Мы рекомендуем не использовать `throw` спецификации исключений `throw()`, кроме, что указывает, что функция не допускает исключений для экранирования. если необходимо использовать спецификации исключений устаревшей формы `throw(type-name)`, MSVC поддержка ограничена. Дополнительные сведения см. в разделе [спецификации исключений \(throw\)](#).

`noexcept` Описатель вводится в C++ 11 в качестве предпочтительного варианта `throw()`.

См. также раздел

[Как взаимодействовать с исключительным и неисключительным кодом](#)

[Справочник по языку C++](#)

[Стандартная библиотека C++](#)

Руководство. проектирование безопасности исключений

12.11.2021 • 5 minutes to read

Одним из преимуществ механизма исключения является то, что выполнение вместе с данными об исключении переходит непосредственно из оператора, который создает исключение, в первую инструкцию `Catch`, которая его обрабатывает. Обработчик может иметь любое количество уровней вверх в стеке вызовов. Функции, которые вызываются между оператором `try` и оператором `throw`, не обязательно должны знать какие-либо сведения о возникшем исключении. Однако они должны быть спроектированы таким образом, чтобы они могли выйти из области "непредвиденно" в любой момент, где исключение может быть распространено ниже, и сделать это без выхода из частично созданных объектов, утечки памяти или структур данных, которые находятся в непригодных для использования состояниях.

Основные методы

Надежная политика обработки исключений требует тщательного мышления и должна быть частью процесса разработки. Как правило, большинство исключений обнаруживаются и создаются на нижних уровнях программного модуля, но обычно эти уровни не имеют достаточного контекста для устранения ошибки или предоставления сообщения конечным пользователям. В средних слоях функции могут перехватывать и выдавать исключения, когда необходимо проверить объект исключения, или у них есть дополнительная полезная информация для верхнего уровня, который в конечном итоге перехватывает исключение. Функция должна перехватывать и "проглотить" исключение, только если оно может полностью восстановиться из нее. Во многих случаях правильное поведение на средних уровнях заключается в том, чтобы разрешить распространение исключения вверх по стеку вызовов. Даже на самом верхнем уровне может быть целесообразно позволить необработанному исключению завершить программу, если исключение покидает программу в состоянии, в котором ее правильность не гарантируется.

Независимо от того, как функция обрабатывает исключение, чтобы гарантировать, что оно является "типобезопасным", оно должно быть спроектировано согласно следующим основным правилам.

Простота сохранения классов ресурсов

При инкапсулировании управления ресурсами вручную в классах используйте класс, который не выполняет никаких действий, кроме управления одним ресурсом. Благодаря простоте использования класса вы снижаете риск возникновения утечек ресурсов. Используйте [смарт-указатели](#), если это возможно, как показано в следующем примере. Этот пример намеренно искусственно и упрощен, чтобы выделить различия при `shared_ptr` использовании.

```

// old-style new/delete version
class NDResourceClass {
private:
    int*    m_p;
    float*  m_q;
public:
    NDResourceClass() : m_p(0), m_q(0) {
        m_p = new int;
        m_q = new float;
    }

    ~NDResourceClass() {
        delete m_p;
        delete m_q;
    }
    // Potential leak! When a constructor emits an exception,
    // the destructor will not be invoked.
};

// shared_ptr version
#include <memory>

using namespace std;

class SPResourceClass {
private:
    shared_ptr<int> m_p;
    shared_ptr<float> m_q;
public:
    SPResourceClass() : m_p(new int), m_q(new float) { }
    // Implicitly defined dtor is OK for these members,
    // shared_ptr will clean up and avoid leaks regardless.
};

// A more powerful case for shared_ptr

class Shape {
    // ...
};

class Circle : public Shape {
    // ...
};

class Triangle : public Shape {
    // ...
};

class SPSHapeResourceClass {
private:
    shared_ptr<Shape> m_p;
    shared_ptr<Shape> m_q;
public:
    SPSHapeResourceClass() : m_p(new Circle), m_q(new Triangle) { }
};

```

Использование идиомы RAII для управления ресурсами

Чтобы обеспечить безопасность исключений, функция должна гарантировать, что объекты, выделенные с помощью или, `malloc` `new` уничтожаются, и все ресурсы, такие как дескрипторы файлов, закрываются или освобождаются, даже если возникает исключение. Функция *получения ресурсов — инициализация* (RAII) идиома связывает такие ресурсы с сроком службы автоматических переменных. Если функция выходит за пределы области действия, либо путем возвращения обычно или из-за исключения, вызываются деструкторы для всех полностью сформированных автоматических переменных. Объект-оболочка RAII, такой как интеллектуальный указатель, вызывает соответствующую функцию `DELETE` или

Close в своем деструкторе. В коде, защищенном с исключением, очень важно немедленно передавать владение каждым ресурсом объекту RAII. Обратите внимание, что `vector` классы, `string` `make_shared`, `fstream` и схожие по себе обрабатывали ресурсы для получения. Однако `unique_ptr` традиционные `shared_ptr` конструкции являются специальными, так как получение ресурса выполняется пользователем, а не объектом. Следовательно, они считаются *освобождением ресурсов*, но их можно считать RAII.

Три гарантии исключений

Как правило, безопасность исключений обсуждается в трех исключениях, которые гарантируют, что функция может предоставить следующие возможности: *гарантия отсутствия ошибок*, *строгая гарантия* и *Базовая гарантия*.

Гарантия № — сбой

Гарантия «без ошибок» (или «без выдачи») является самой высокой гарантией, которую может предоставить функция. Он указывает, что функция не будет вызывать исключение или разрешить распространение. Однако нельзя гарантировать такую гарантию, если (а) известно, что все функции, которые вызывает эта функция, также не являются ошибками, или (б) известно, что все возникшие исключения перехватываются до того, как они достигли этой функции, или (с) вы узнали, как перехватить и правильно обрабатывать все исключения, которые могут достичь этой функции.

Как строгая гарантия, так и Базовая гарантия основываются на предположении, что деструкторы являются неуспешными. Все контейнеры и типы в стандартной библиотеке гарантируют, что их деструкторы не вызывают исключение. Существует также и обратное требование: Стандартная библиотека требует, чтобы определяемые пользователем типы, например, в качестве аргументов шаблона, не создавали деструкторы без вызова.

Строгая гарантия

Строгая гарантия указывает, что если функция выходит за пределы области действия из-за исключения, она не будет приводить к утечке памяти, а состояние программы не будет изменено. Функция, обеспечивающая строгую гарантию, по сути, является транзакцией, которая имеет семантику фиксации или отката: либо полностью завершается успешно, либо не оказывает никакого воздействия.

Базовая гарантия

Основная гарантия является слабой из трех. Тем не менее, это может быть лучшим выбором, когда строгая гарантия является слишком дорогостоящей в потреблении памяти или производительности. Основная гарантия указывает, что при возникновении исключения память не утечка, и объект остается в рабочем состоянии, даже если данные могли быть изменены.

Классы, защищенные с исключением

Класс помогает обеспечить собственную безопасность исключений, даже если она используется небезопасными функциями, предотвращая ее частичное построение или частичное уничтожение. Если конструктор класса завершает работу до завершения, объект никогда не создается и его деструктор никогда не вызывается. Хотя автоматические переменные, инициализированные до исключения, будут вызывать свои деструкторы, динамически выделенная память или ресурсы, не управляемые смарт-указателем или аналогичной автоматической переменной, будут потеряны.

Встроенные типы являются неудачными, а стандартные типы библиотек поддерживают базовую гарантию как минимум. Следуйте этим рекомендациям для любого определяемого пользователем типа, который должен быть защищен от исключений.

- Для управления всеми ресурсами используйте смарт-указатели или другие оболочки типа RAII. Избегайте функций управления ресурсами в деструкторе класса, так как деструктор не будет

вызываться, если конструктор выдаст исключение. Однако если класс является выделенным диспетчером ресурсов, который управляет только одним ресурсом, то для управления ресурсами допустимо использовать деструктор.

- Понимать, что исключение, созданное в конструкторе базового класса, не может быть проглатываются в конструкторе производного класса. Если необходимо преобразовать и повторно создать исключение базового класса в производном конструкторе, используйте блок `try` функции.
- Определите, следует ли сохранять все состояния класса в члене данных, заключенном в интеллектуальный указатель, особенно если класс имеет концепцию "инициализация, которая может быть неудачной". Хотя C++ допускает неинициализированные элементы данных, он не поддерживает неинициализированные или частично инициализированные экземпляры класса. Конструктор должен быть успешно выполнен или завершился ошибкой. Если конструктор не выполняется до завершения, объект не создается.
- Не разрешать исключения в escape-последовательности из деструктора. Базовый аксиома C++ заключается в том, что деструкторы никогда не должны разрешать исключение для распространения стека вызовов. Если деструктор должен выполнить операцию, вызывающую исключение, он должен быть выполнен в блоке `try catch` и проглотить исключение. Стандартная библиотека обеспечивает эту гарантию для всех деструкторов, которые он определяет.

См. также

[Современные рекомендации по C++ для исключений и обработки ошибок](#)
[Как взаимодействовать с исключительным и неисключительным кодом](#)

Как взаимодействовать с исключительным и неисключительным кодом

12.11.2021 • 5 minutes to read

В этой статье описывается, как реализовать последовательную обработку исключений в модуле C++, а также как преобразовать эти исключения в коды ошибок и из них на границах исключений.

Иногда модуль C++ может взаимодействовать с кодом, который не использует исключения (код не является исключительным). Такой интерфейс называется *границей исключения*. Например, может потребоваться вызвать функцию Win32 `CreateFile` в программе C++. `CreateFile` не создает исключений; вместо этого он устанавливает коды ошибок, которые могут быть получены `GetLastError` функцией. Если ваша программа на C++ не является тривиальной, то, скорее всего, предпочтительно иметь единообразную политику обработки ошибок на основе исключений. И вы, вероятно, не захотите отменять исключения просто потому, что вы будете взаимодействовать с неисключительным кодом, и ни на что не хотите смешивать политики ошибок на основе исключений и исключений в модуле C++.

Вызов неисключительных функций из C++

При вызове неисключительной функции из C++ идея заключается в переносе этой функции в функцию C++, которая обнаруживает ошибки, а затем, возможно, вызывает исключение. При проектировании такой функции-оболочки сначала необходимо решить, какой тип гарантии исключения следует предоставить: без-Throw, строгий или базовый. Во-вторых, разработайте функцию таким образом, чтобы все ресурсы, например дескрипторы файлов, были правильно освобождены при возникновении исключения. Как правило, это означает, что для владельца ресурсов используются смарт-указатели или аналогичные диспетчеры ресурсов. Дополнительные сведения о вопросах проектирования см. [в разделе руководства. проектирование безопасности исключений](#).

Пример

В следующем примере показаны функции C++, которые используют Win32 `CreateFile` и `ReadFile` функции для внутреннего открытия и чтения двух файлов. Класс — это программа-`File` оболочка для получения ресурсов (RAII) для дескрипторов файлов. Его конструктор обнаруживает условие "файл не найден" и создает исключение для распространения ошибки вверх по стеку вызовов модуля C++ (в данном примере это `main()` функция). Если после создания объекта создается исключение `File`, то деструктор автоматически вызывает метод `CloseHandle` для освобождения файла. (Если вы предпочитаете для этой цели можно использовать класс библиотеки активных шаблонов (ATL) `CHandle` или `unique_ptr` вместе с пользовательским.) Функции, вызывающие API Win32 и CRT, обнаруживают ошибки, а затем вызывают исключения C++ с помощью локально определенной `ThrowLastErrorIf` функции, которая, в свою очередь, использует `Win32Exception` класс, производный от `runtime_error` класса. Все функции в этом примере обеспечивают строгую гарантию исключений. Если исключение создается в любой точке этих функций, ресурсы не теряются и состояние программы не изменяется.

```
// compile with: /EHsc
#include <Windows.h>
#include <stdlib.h>
#include <vector>
#include <iostream>
#include <string>
#include <limits>
#include <stdexcept>
```

```

using namespace std;

string FormatErrorMessage(DWORD error, const string& msg)
{
    static const int BUFFERLENGTH = 1024;
    vector<char> buf(BUFFERLENGTH);
    FormatMessageA(FORMAT_MESSAGE_FROM_SYSTEM, 0, error, 0, buf.data(),
        BUFFERLENGTH - 1, 0);
    return string(buf.data()) + " (" + msg + ")";
}

class Win32Exception : public runtime_error
{
private:
    DWORD m_error;
public:
    Win32Exception(DWORD error, const string& msg)
        : runtime_error(FormatErrorMessage(error, msg)), m_error(error) { }

    DWORD GetErrorCode() const { return m_error; }
};

void ThrowLastErrorHandlerIf(bool expression, const string& msg)
{
    if (expression) {
        throw Win32Exception(GetLastError(), msg);
    }
}

class File
{
private:
    HANDLE m_handle;

    // Declared but not defined, to avoid double closing.
    File& operator=(const File&);
    File(const File&);

public:
    explicit File(const string& filename)
    {
        m_handle = CreateFileA(filename.c_str(), GENERIC_READ, FILE_SHARE_READ,
            nullptr, OPEN_EXISTING, FILE_ATTRIBUTE_READONLY, nullptr);
        ThrowLastErrorHandlerIf(m_handle == INVALID_HANDLE_VALUE,
            "CreateFile call failed on file named " + filename);
    }

    ~File() { CloseHandle(m_handle); }

    HANDLE GetHandle() { return m_handle; }
};

size_t GetFileSizeSafe(const string& filename)
{
    File fobj(filename);
    LARGE_INTEGER filesize;

    BOOL result = GetFileSizeEx(fobj.GetHandle(), &filesize);
    ThrowLastErrorHandlerIf(result == FALSE, "GetFileSizeEx failed: " + filename);

    if (filesize.QuadPart < (numeric_limits<size_t>::max)()) {
        return filesize.QuadPart;
    } else {
        throw;
    }
}

vector<char> ReadFileVector(const string& filename)
{
    File fobj(filename);

```

```

size_t filesize = GetFileSizeSafe(filename);
DWORD bytesRead = 0;

vector<char> readbuffer(filesize);

BOOL result = ReadFile(fobj.GetHandle(), readbuffer.data(), readbuffer.size(),
    &bytesRead, nullptr);
ThrowLastErrorIf(result == FALSE, "ReadFile failed: " + filename);

cout << filename << " file size: " << filesize << ", bytesRead: "
    << bytesRead << endl;

return readbuffer;
}

bool IsFileDiff(const string& filename1, const string& filename2)
{
    return ReadFileVector(filename1) != ReadFileVector(filename2);
}

#include <iomanip>

int main ( int argc, char* argv[] )
{
    string filename1("file1.txt");
    string filename2("file2.txt");

    try
    {
        if(argc > 2) {
            filename1 = argv[1];
            filename2 = argv[2];
        }

        cout << "Using file names " << filename1 << " and " << filename2 << endl;

        if (IsFileDiff(filename1, filename2)) {
            cout << "++ Files are different." << endl;
        } else {
            cout << "== Files match." << endl;
        }
    }
    catch(const Win32Exception& e)
    {
        ios state(nullptr);
        state.copyfmt(cout);
        cout << e.what() << endl;
        cout << "Error code: 0x" << hex << uppercase << setw(8) << setfill('0')
            << e.GetErrorCode() << endl;
        cout.copyfmt(state); // restore previous formatting
    }
}

```

Вызов исключительного кода из неисключительного кода

Функции C++, объявленные как "extern C", могут вызываться программами на языке С. СОМ-серверы C++ можно использовать в коде, написанном на любом из нескольких различных языков. При реализации открытых функций с поддержкой исключений в C++ для вызова неисключительным кодом функция C++ не должна разрешать передачу исключений вызывающему объекту. Поэтому функция C++ должна явным образом перехватывать каждое исключение, которое ему известно, как выполнять обработку и, при необходимости, преобразовывать исключение в код ошибки, распознаваемый вызывающим объектом. Если известны не все потенциальные исключения, функция C++ должна иметь блок в `catch(...)` качестве последнего обработчика. В этом случае лучше сообщить вызывающей стороне о неустранимой ошибке, так как ваша программа может находиться в неизвестном состоянии.

В следующем примере показана функция, которая предполагает, что любое исключение, которое может быть создано, является либо Win32Exception, либо типом исключения, производным от `std::exception`. Функция перехватывает любое исключение этих типов и распространяет сведения об ошибке как код ошибки Win32 вызывающему объекту.

```
BOOL DiffFiles2(const string& file1, const string& file2)
{
    try
    {
        File f1(file1);
        File f2(file2);
        if (IsTextFileDiff(f1, f2))
        {
            SetLastError(MY_APPLICATION_ERROR_FILE_MISMATCH);
            return FALSE;
        }
        return TRUE;
    }
    catch(Win32Exception& e)
    {
        SetLastError(e.GetErrorCode());
    }

    catch(std::exception& e)
    {
        SetLastError(MY_APPLICATION_GENERAL_ERROR);
    }
    return FALSE;
}
```

При преобразовании из исключений в коды ошибок часто возникают проблемы, возникающие в том, что коды ошибок зачастую не содержат богатую информацию, которую может хранить исключение. Чтобы решить эту ошибку, можно предоставить `catch` блок для каждого конкретного типа исключения, который может быть создан, и выполнить ведение журнала для записи сведений об исключении перед его преобразованием в код ошибки. Такой подход может создать множество повторов кода, если в нескольких функциях используется один набор `catch` блоков. Хорошим способом избежать повторения кода является рефакторинг этих блоков в одну закрытую служебную функцию, которая реализует `try` `catch` блоки и принимает объект функции, который вызывается в `try` блоке. В каждой открытой функции передайте код в служебную функцию в виде лямбда-выражения.

```
template<typename Func>
bool Win32ExceptionBoundary(Func&& f)
{
    try
    {
        return f();
    }
    catch(Win32Exception& e)
    {
        SetLastError(e.GetErrorCode());
    }
    catch(const std::exception& e)
    {
        SetLastError(MY_APPLICATION_GENERAL_ERROR);
    }
    return false;
}
```

В следующем примере показано, как написать лямбда-выражение, определяющее функтор. Если функтор определен как "встроенный" с помощью лямбда-выражения, часто бывает проще прочитать, чем если бы

он был записан как объект именованной функции.

```
bool DiffFiles3(const string& file1, const string& file2)
{
    return Win32ExceptionBoundary([&]() -> bool
    {
        File f1(file1);
        File f2(file2);
        if (IsTextFileDiff(f1, f2))
        {
            SetLastError(MY_APPLICATION_ERROR_FILE_MISMATCH);
            return false;
        }
        return true;
    });
}
```

Дополнительные сведения о лямбда-выражениях см. в разделе [лямбда-выражения](#).

См. также

[Современные рекомендации по C++ для исключений и обработки ошибок](#)

[How to: Design for Exception Safety](#) (Практическое руководство. Разработка с учетом безопасности исключений)

Операторы try, throw и catch (C++)

12.11.2021 • 2 minutes to read

Для реализации обработки исключений в C++ используются `try`, `throw` выражения, и `catch`.

Во-первых, используйте `try` блок, чтобы заключить одну или несколько инструкций, которые могут вызвать исключение.

`throw` Выражение сигнализирует о том, что в блоке встречается исключительная ситуация — часто возникает ошибка `try`. В качестве операнда выражения можно использовать объект любого типа `throw`. Обычно этот объект используется для передачи информации об ошибке. В большинстве случаев рекомендуется использовать класс `std::Exception` или один из производных классов, определенных в стандартной библиотеке. Если один из них не подходит, рекомендуется создать собственный производный класс исключений из `std::exception`.

Чтобы обрабатывать исключения, которые могут быть вызваны, реализуйте один или несколько `catch` блоков сразу после `try` блока. Каждый `catch` блок указывает тип исключения, которое может быть обработано.

В этом примере показан `try` блок и его обработчики. Предположим, `GetNetworkResource()` получает данные через сетевое подключение, а 2 типа исключений являются определенными пользователем классами, производными от `std::exception`. Обратите внимание, что исключения перехватываются по `const` ссылке в `catch` инструкции. Рекомендуется создавать исключения по значению и захватывать их ссылкой константы.

Пример

```

MyData md;
try {
    // Code that could throw an exception
    md = GetNetworkResource();
}
catch (const networkIOException& e) {
    // Code that executes when an exception of type
    // networkIOException is thrown in the try block
    // ...
    // Log error message in the exception object
    cerr << e.what();
}
catch (const myDataFormatException& e) {
    // Code that handles another exception type
    // ...
    cerr << e.what();
}

// The following syntax shows a throw expression
MyData GetNetworkResource()
{
    // ...
    if (IOSuccess == false)
        throw networkIOException("Unable to connect");
    // ...
    if (readError)
        throw myDataFormatException("Format error");
    // ...
}

```

Remarks

Код после `try` предложения является защищенным разделом кода. `throw` Выражение вызывает , то есть вызывает исключение. Блок кода после `catch` предложения является обработчиком исключений. Это обработчик, который *перехватывает* исключение, возникающее, если типы в `throw` `catch` выражениях и совместимы. Список правил, регулирующих сопоставление типов в `catch` блоках, см. в разделе [Вычисление блоков catch](#). Если `catch` оператор задает многоточие (...) вместо типа, `catch` блок обрабатывает каждый тип исключения. При компиляции с параметром /EHа они могут включать структурированные исключения C и созданные системой или асинхронные исключения, такие как защита памяти, деление на ноль и нарушения операций с плавающей запятой. Поскольку `catch` блоки обрабатываются в порядке программ для поиска соответствующего типа, обработчик многоточия должен быть последним обработчиком для связанного `try` блока. Используйте `catch(...)` осторожно, не позволяйте программе продолжать выполнение, если блоку `catch` не известно, как обработать конкретное перехваченное исключение. Как правило, блок `catch(...)` используется для ведения журнала ошибок и выполнения специальной очистки перед остановкой выполнения программы.

`throw` Выражение, не имеющее операнда, повторно создает исключение, которое сейчас обрабатывается. Рекомендуется использовать эту форму при повторном создании исключения, поскольку это позволяет сохранить исходные сведения полиморфного типа исключения. Такое выражение должно использоваться только в `catch` обработчике или в функции, которая вызывается из `catch` обработчика. Вновь созданный объект исключения представляет собой исходный объект исключения, а не его копию.

```
try {
    throw CSomeOtherException();
}
catch(...) {
    // Catch all exceptions - dangerous!!!
    // Respond (perhaps only partially) to the exception, then
    // re-throw to pass the exception to some other handler
    // ...
    throw;
}
```

См. также

[Современные рекомендации по C++ для исключений и обработки ошибок](#)

[Ключевые слова](#)

[Необработанные исключения C++](#)

[__uncaught_exception](#)

Проверка блоков Catch (C++)

12.11.2021 • 2 minutes to read

C++ позволяет создавать исключения любого типа, хотя обычно рекомендуется создавать типы, производные от `std::exception`. Исключение C++ может быть перехвачено `catch` обработчиком, который указывает тот же тип, что и выданное исключение, или обработчиком, который может перехватить любой тип исключения.

Если созданное исключение имеет тип класса, у которого имеется один или несколько базовых классов, то его могут перехватывать обработчики, которые принимают базовые классы (и ссылки на базовые классы) этого типа исключения. Обратите внимание, что если исключение перехватывается по ссылке, то оно привязывается к самому объекту исключения; в противном случае обрабатывается его копия (как и в случае с аргументами функции).

При возникновении исключения оно может быть перехвачено следующими типами `catch` обработчиков:

- Обработчик, который может принимать любой тип данных (синтаксис с многоточием).
- Обработчик, принимающий тот же тип, что и объект исключения; так как это копия, `const` `volatile` модификаторы не учитываются.
- Обработчик, который принимает ссылку на тот же тип, что и у объекта исключения.
- Обработчик, принимающий ссылку на `const` форму или того `volatile` же типа, что и объект исключения.
- Обработчик, принимающий базовый класс того же типа, что и объект исключения; так как это копия, `const` `volatile` модификаторы не учитываются. `catch` Обработчик для базового класса не должен предшествовать `catch` обработчику для производного класса.
- Обработчик, который принимает ссылку на базовый класс того же типа, что и у объекта исключения.
- Обработчик, принимающий ссылку на `const` `volatile` форму или базового класса того же типа, что и объект исключения.
- Обработчик, который принимает указатель, в который можно преобразовать созданный объект указателя при помощи стандартных правил преобразования указателей.

Порядок, в котором `catch` отображаются обработчики, важен, так как обработчики для данного `try` блока анализируются в порядке их внешнего вида. Например, ошибкой будет поместить обработчик для базового класса перед обработчиком для производного класса. После обнаружения соответствующего `catch` обработчика последующие обработчики не проверяются. В результате обработчик многоточия `catch` должен быть последним обработчиком для своего `try` блока. Пример:

```
// ...
try
{
    // ...
}
catch( ... )
{
    // Handle exception here.
}
// Error: the next two handlers are never examined.
catch( const char * str )
{
    cout << "Caught exception: " << str << endl;
}
catch( CExcptClass E )
{
    // Handle CExcptClass exception here.
}
```

В этом примере обработчик многоточия `catch` является единственным проверенным обработчиком.

См. также

[Современные рекомендации по C++ для исключений и обработки ошибок](#)

Исключения и освобождение стека в C++

12.11.2021 • 3 minutes to read

В механизме исключений C++ элемент управления перемещается из оператора `throw` в первый оператор `catch`, который может обработать выданный тип. При достижении оператора `Catch` все автоматические переменные, которые находятся в области между операторами `Throw` и `catch`, уничтожаются в процессе, известном как *Очистка стека*. При очистке стека выполнение продолжается следующим образом.

1. Управление достигает `try` оператора с помощью обычного последовательного выполнения. В `try` блоке выполняется защищенный раздел.
2. Если во время выполнения защищенного раздела исключение не возникает, то `catch` предложения, следующие за `try` блоком, не выполняются. Выполнение продолжится в операторе после последнего `catch` предложения, следующего за соответствующим `try` блоком.
3. Если исключение возникает во время выполнения защищенного раздела или в любой подпрограмме, что защищенный раздел вызывает напрямую или косвенно, объект исключения создается из объекта, созданного `throw` операндом. (Это означает, что может быть вовлечен конструктор копии.) На этом этапе компилятор ищет `catch` предложение в более высоком контексте выполнения, который может управлять исключением вызываемого типа или `catch` обработчиком, который может обработать любые типы исключений. `catch` Обработчики анализируются в порядке их отображения после `try` блока. Если соответствующий обработчик не найден, проверяется следующий динамически охватывающий `try` блок. Этот процесс будет продолжен до тех пор, пока `try` не будет проверен внешний внешний блок.
4. Если соответствующий обработчик по-прежнему не найден или исключение возникает во время процесса очистки до получения элемента управления обработчиком, вызывается предопределенная функция времени выполнения `terminate`. Если исключение возникает после создания исключения, но до начала процесса очистки, вызывается функция `terminate`.
5. Если найден соответствующий `catch` обработчик, который перехватывается по значению, его формальный параметр инициализируется путем копирования объекта исключения. Если обработчик выполняет перехват по ссылке, параметр инициализируется для ссылки на объект исключения. После инициализации формального параметра начинается процесс очистки стека. Это подразумевает уничтожение всех автоматически созданных объектов, которые были полностью созданы, но еще не были удалены, между началом `try` блока, который связан с `catch` обработчиком, и исключением из-за исключения. Удаление происходит в порядке, обратном созданию. `catch` Обработчик выполняется, и программа возобновляет выполнение после последнего обработчика, то есть в первой инструкции или конструкции, которая не является `catch` обработчиком. Элемент управления может вводить `catch` обработчик только через выданное исключение, никогда через `goto` оператор или `case` метку в `switch` инструкции.

Пример раскрутки стека

В следующем примере показано, как очистить стек при создании исключения. Выполнение потока переходит от оператора `throw` в `c` к оператору `catch` в `main`, и при этом удаляются все функции. Обратите внимание, что порядок создания и удаления объектов `Dummy` соответствует порядку их выхода из области видимости. Также обратите внимание, что завершается выполнение только функции `main`, содержащей оператор `catch`. Функция `a` никогда не возвращается после вызова `b()`, и `b` никогда не возвращается после вызова `c()`. Обратите внимание, что если раскомментировать определение

указателя `Dummy` и соответствующую инструкцию `DELETE`, а затем запустить программу, указатель не удаляется. Это показывает, что может произойти, если функции не предоставляют гарантию исключения. Дополнительные сведения см. в разделе "Практическое руководство. Разработка исключений". Если закомментировать оператор `catch`, можно наблюдать за тем, что происходит при завершении выполнения программы в результате необработанного исключения.

```
#include <string>
#include <iostream>
using namespace std;

class MyException{};
class Dummy
{
public:
    Dummy(string s) : MyName(s) { PrintMsg("Created Dummy:"); }
    Dummy(const Dummy& other) : MyName(other.MyName){ PrintMsg("Copy created Dummy:"); }
    ~Dummy(){ PrintMsg("Destroyed Dummy:"); }
    void PrintMsg(string s) { cout << s << MyName << endl; }
    string MyName;
    int level;
};

void C(Dummy d, int i)
{
    cout << "Entering FunctionC" << endl;
    d.MyName = " C";
    throw MyException();

    cout << "Exiting FunctionC" << endl;
}

void B(Dummy d, int i)
{
    cout << "Entering FunctionB" << endl;
    d.MyName = "B";
    C(d, i + 1);
    cout << "Exiting FunctionB" << endl;
}

void A(Dummy d, int i)
{
    cout << "Entering FunctionA" << endl;
    d.MyName = " A" ;
    // Dummy* pd = new Dummy("new Dummy"); //Not exception safe!!!
    B(d, i + 1);
    // delete pd;
    cout << "Exiting FunctionA" << endl;
}

int main()
{
    cout << "Entering main" << endl;
    try
    {
        Dummy d(" M");
        A(d,1);
    }
    catch (MyException& e)
    {
        cout << "Caught an exception of type: " << typeid(e).name() << endl;
    }

    cout << "Exiting main." << endl;
    char c;
    cin >> c;
}
```

```
/* Output:  
Entering main  
Created Dummy: M  
Copy created Dummy: M  
Entering FunctionA  
Copy created Dummy: A  
Entering FunctionB  
Copy created Dummy: B  
Entering FunctionC  
Destroyed Dummy: C  
Destroyed Dummy: B  
Destroyed Dummy: A  
Destroyed Dummy: M  
Caught an exception of type: class MyException  
Exiting main.
```

```
*/
```

Спецификации исключений (throw, не Except) (C++)

12.11.2021 • 3 minutes to read

Спецификации исключений — это функция языка C++, которая указывает намерение программиста о типах исключений, которые могут распространяться функцией. Можно указать, что функция может или не может выйти за исключением, используя *спецификацию исключения*. Компилятор может использовать эти сведения для оптимизации вызовов функции, а также для завершения программы, если непредвиденное исключение приводит к экранированию функции.

До C++ 17 существовали два типа спецификаций исключений. В C++ 11 была введена *только спецификация*. Он указывает, является ли набор возможных исключений, которые могут вызывать функцию, пустым. Спецификация или спецификация *динамического исключения*

`throw(optional_type_list)` устарели в C++ 11 и удалены в C++ 17, за исключением `throw()`, который является псевдонимом для `noexcept(true)`. Эта спецификация исключений была разработана для предоставления сводных сведений о том, какие исключения могут быть вызваны из функции, но на практике обнаружены проблемы. Одна из динамических исключений, которая была бы довольно полезной, была неусловной `throw()` спецификацией. Например, объявление функции:

```
void MyFunction(int i) throw();
```

сообщает компилятору, что функция не создает исключений. Однако в /std: **режим C++ 14** это может привести к неопределенному поведению, если функция создает исключение. Поэтому рекомендуется использовать оператор " `except` " вместо того, который приведен выше.

```
void MyFunction(int i) noexcept;
```

В следующей таблице перечислены реализации спецификации исключений в Microsoft C++.

СПЕЦИФИКАЦИЯ ИСКЛЮЧЕНИЙ	ЗНАЧЕНИЕ
<code>noexcept</code> <code>noexcept(true)</code> <code>throw()</code>	Функция не вызывает исключений. В /std: режим C++ 14 (по умолчанию) <code>noexcept</code> и <code>noexcept(true)</code> эквивалентны. При возникновении исключения из функции, которая объявлена <code>noexcept</code> или <code>noexcept(true)</code> вызывается <code>std::Terminate</code> . При возникновении исключения из функции, объявленной как <code>throw()</code> в режиме /std: C++ 14 , результат является неопределенным поведением. Никакая конкретная функция не вызывается. Это расхождение по стандарту C++ 14, которое требует, чтобы компилятор вызывал <code>std::непредвиденный</code> . Visual Studio 2017 версий 15,5 и более поздних версий: в /std: режим C++ 17 , <code>noexcept</code> , <code>noexcept(true)</code> , и <code>throw()</code> все эквивалентны. В /std: режим C++ 17 <code>throw()</code> является псевдонимом для <code>noexcept(true)</code> . В режиме /std: C++ 17 при возникновении исключения из функции, объявленной с любой из этих спецификаций, <code>std::Terminate</code> вызывается согласно требованиям стандарта C++ 17.

СПЕЦИФИКАЦИЯ ИСКЛЮЧЕНИЙ	ЗНАЧЕНИЕ
<pre>noexcept(false) throw(...)</pre> <p>Нет спецификации</p>	Функция может вызывать исключение любого типа.
<pre>throw(type)</pre>	(C++ 14 и более ранние версии) Функция может вызывать исключение типа <code>type</code> . Компилятор принимает синтаксис, но интерпретирует его как <code>noexcept(false)</code> . В /std: режиме C++ 17 компилятор выдает предупреждение C5040.

Если в приложении используется обработка исключений, в стеке вызовов должна быть функция, которая обрабатывает созданные исключения до выхода из внешней области функции, помеченной как `noexcept`, `noexcept(true)` или `throw()`. Если какие-либо функции, вызываемые между исключением и обработчиком исключения, задаются как `noexcept`, `noexcept(true)` (или `throw()`) в режиме /std: C++ 17, программа завершается, когда исключение распространяется с помощью функции Except.

Поведение функции зависит от следующих факторов:

- Установлен режим компиляции стандартного языка .
- В какой среде выполняется компиляция функции — в С или C++.
- Используемый параметр компилятора /EH .
- Задана ли явно спецификация исключений.

Явные спецификации исключений не разрешено использовать для функций С. Предполагается, что функция С не создает исключения в /EHsc и может вызывать структурированные исключения в порядке /EHs, /EHa или /exac.

В следующей таблице приведены сведения о том, может ли функция C++ вызываться в различных параметрах обработки исключений компилятора.

ФУНКЦИЯ	/EHSC	/EHs	/EHa	/EHAc
Функция C++ без спецификации исключений	Да	Да	Да	Да
Функция C++ со <code>noexcept</code> , <code>noexcept(true)</code> , <code>throw()</code> спецификацией исключения, или	Нет	Нет	Да	Да
Функция C++ со <code>noexcept(false)</code> , <code>throw(...)</code> , <code>throw(type)</code> спецификацией исключения, или	Да	Да	Да	Да

Пример

```

// exception_specification.cpp
// compile with: /EHs
#include <stdio.h>

void handler() {
    printf_s("in handler\n");
}

void f1(void) throw(int) {
    printf_s("About to throw 1\n");
    if (1)
        throw 1;
}

void f5(void) throw() {
    try {
        f1();
    }
    catch(...) {
        handler();
    }
}

// invalid, doesn't handle the int exception thrown from f1()
// void f3(void) throw() {
//     f1();
// }

void __declspec(nothrow) f2(void) {
    try {
        f1();
    }
    catch(int) {
        handler();
    }
}

// only valid if compiled without /EHc
// /EHc means assume extern "C" functions don't throw exceptions
extern "C" void f4(void);
void f4(void) {
    f1();
}

int main() {
    f2();

    try {
        f4();
    }
    catch(...) {
        printf_s("Caught exception from f4\n");
    }
    f5();
}

```

```

About to throw 1
in handler
About to throw 1
Caught exception from f4
About to throw 1
in handler

```

См. также раздел

[Операторы try, throw и catch \(C++\)](#)

[Современные рекомендации по C++ для исключений и обработки ошибок](#)

noexcept (C++)

12.11.2021 • 2 minutes to read

C++ 11: Указывает ли функция создавать исключения.

Синтаксис

выражение except:

noexcept

noexcept (константное выражение)

Параметры

Константное выражение

Константное выражение типа `bool`, которое представляет, является ли набор возможных типов исключений пустым. Неусловная версия эквивалентна `noexcept(true)`.

Remarks

Выражение "без исключения" является разновидностью спецификации исключений, суффиксом для объявления функции, представляющей набор типов, которые могут сопоставляться обработчиком исключений для любого исключения, которое выходит из функции. Унарный условный оператор `noexcept(constant_expression,)` где `constant_expression` выдаст `true` и его безусловный синоним `noexcept`, указывает, что набор возможных типов исключений, которые могут выходить из функции, пуст. Это значит, что функция никогда не создает исключение и никогда не позволяет распространять исключение за пределы области действия. Оператор `noexcept(constant_expression,)` где выдается `constant_expression false`, или отсутствие спецификации исключения (кроме деструктора или функции освобождения) указывает, что набор возможных исключений, которые могут выйти из функции, — это набор всех типов.

Пометьте функцию как `noexcept`, только если все функции, которые она вызывает, либо прямо, либо косвенно, являются также `noexcept` или `const`. Компилятор не обязательно проверяет каждый путь кода на наличие исключений, которые могут быть всплывающими до `noexcept` функции. Если исключение выходит из внешней области видимости функции, помеченной как `noexcept`, функция `std::Terminate` вызывается немедленно, и нет гарантии, что будут вызываться деструкторы любых объектов в области. Используйте `noexcept` вместо динамического описателя исключений `throw()`, который теперь является устаревшим в стандарте. Мы рекомендуем применить `noexcept` к любой функции, которая никогда не позволяет использовать исключение для распространения стека вызовов. При объявлении функции `noexcept` она позволяет компилятору создавать более эффективный код в нескольких разных контекстах. Дополнительные сведения см. в разделе спецификации исключений.

Пример

Функция шаблона, которая копирует свой аргумент, может быть объявлена `noexcept` в условии, что копируемый объект является обычным старым типом данных (Pod). Такая функция может быть объявлена следующим образом:

```
#include <type_traits>

template <typename T>
T copy_object(const T& obj) noexcept(std::is_pod<T>)
{
    // ...
}
```

См. также раздел

[Современные рекомендации по C++ для исключений и обработки ошибок](#)
[Спецификации исключений \(throw, не Except\)](#)

Необработанные исключения C++

12.11.2021 • 2 minutes to read

Если `catch` для текущего исключения не удается найти соответствующий обработчик (или обработчик многоточия), вызывается предопределенная `terminate` функция времени выполнения. (Вы также можете явно вызвать `terminate` в любом обработчике.) Действие по умолчанию `terminate` — вызвать `abort`. Если вам необходимо, чтобы перед выходом из приложения функция `terminate` в вашей программе вызывала какую-то другую функцию, вызовите функцию `set_terminate`, указав в качестве ее единственного аргумента ту функцию, которую нужно вызывать. Функцию `set_terminate` можно вызвать из любого места программы. `terminate` Подпрограммы всегда вызывают последнюю функцию, заданную в качестве аргумента для `set_terminate`.

Пример

В следующем примере создается исключение типа `char *`, однако в нем нет обработчика, предназначенного для перехвата исключений `char *`. Вызов `set_terminate` содержит инструкцию, согласно которой `terminate` вызывает функцию `term_func`.

```
// exceptions_Unhandled_Exceptions.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;
void term_func() {
    cout << "term_func was called by terminate." << endl;
    exit( -1 );
}
int main() {
    try
    {
        set_terminate( term_func );
        throw "Out of memory!" // No catch handler for this exception
    }
    catch( int )
    {
        cout << "Integer exception raised." << endl;
    }
    return 0;
}
```

Выходные данные

```
term_func was called by terminate.
```

Функция `term_func` должна завершать программу или текущий поток (желательно путем вызова функции `exit`). Если вместо этого она возвращает управление вызвавшему объекту, то вызывается функция `abort`.

См. также:

[Современные рекомендации по C++ для исключений и обработки ошибок](#)

Смешивание исключений C (структурированная) и C++

12.11.2021 • 2 minutes to read

Если вы хотите написать переносимый код, использование структурированной обработки исключений (SEH) в программе C++ не рекомендуется. Однако иногда может потребоваться компиляция с использованием [/EHs](#) и смешение структурированных исключений и исходного кода C++, а также необходимость в некоторых средствах для обработки обоих видов исключений. Поскольку структурированный обработчик исключений не имеет концепции объектов или типизированных исключений, он не может управлять исключениями, созданными кодом C++. Однако `catch` обработчики C++ могут управлять структуризованными исключениями. Синтаксис обработки исключений C++ (`try`, `throw`, `catch`) не принимается компилятором C, но структурированный синтаксис обработки исключений (`_try`, `_except`, `_finally`) поддерживается компилятором C++.

[`_set_se_translator`](#) Дополнительные сведения об обработке структурированных исключений в виде исключений C++ см. в разделе.

При смешении структурированных и C++ исключений следует учитывать следующие возможные проблемы:

- Исключения C++ и структурированные исключения не могут смешиваться в одной и той же функции.
- Обработчики завершения (`_finally` блоки) всегда выполняются даже во время очистки после возникновения исключения.
- Обработка исключений C++ может перехватывать и сохранять семантику очистки во всех модулях, скомпилированных с помощью [/EH](#) параметров компилятора, которые обеспечивают семантику очистки.
- Могут возникнуть ситуации, в которых функции деструктора не вызываются для всех объектов. Например, структурированное исключение может возникнуть при попытке выполнить вызов функции через неинициализированный указатель функции. Если параметры функции являются объектами, созданными до вызова, деструкторы этих объектов не вызываются во время очистки стека.

Дальнейшие действия

- Использование `setjmp` ИЛИ `longjmp` в программах на C++

См. Дополнительные сведения об использовании `setjmp` И `longjmp` В программах на C++.

- Обработка структурированных исключений в C++

См. примеры способов использования C++ для управления структуризованными исключениями.

См. также

[Современные рекомендации по C++ для исключений и обработки ошибок](#)

Использование setjmp и longjmp

12.11.2021 • 2 minutes to read

При совместном использовании `setjmp` и `longjmp` они предоставляют способ выполнения нелокальной `goto`. Они обычно используются в коде C для передачи управления выполнения в код обработки ошибок или восстановления в ранее вызванной подпрограмме без использования стандартных соглашений о вызове или возврате.

Caution

Поскольку `setjmp` и `longjmp` не поддерживают правильное уничтожение объектов кадра стека между компиляторами C++, и, поскольку они могут ухудшить производительность, предотвращая оптимизацию локальных переменных, мы не рекомендуем использовать их в программах на C++. Вместо этого рекомендуется использовать `try` `catch` конструкции и.

Если вы решили использовать `setjmp` и `longjmp` в программе на C++, также включите `<setjmp.h>` или `<setjmpex.h>`, чтобы обеспечить правильное взаимодействие между функциями и структурной обработкой исключений (SEH) или C++.

Блок, относящийся только к системам Microsoft

Если для компиляции кода C++ используется параметр `/EH`, деструкторы локальных объектов вызываются во время очистки стека. Однако при использовании параметра `/EHsc` или `/EHscs` для компиляции и одной из функций, использующих вызовы, [за исключением](#) вызовов, то `longjmp` деструктор для этой функции может не произойти, в зависимости от состояния оптимизатора.

В переносимом коде при `longjmp` выполнении вызова правильное уничтожение объектов на основе кадров явно не гарантируется стандартом и может не поддерживаться другими компиляторами. Чтобы сообщить, что на уровне предупреждений 4 вызывается `setjmp` предупреждение C4611: взаимодействие между "_setjmp" и уничтожением объектов C++ не является переносимым.

Завершение блока, относящегося только к системам Microsoft

См. также

[Смешивание исключений C \(структурированная\) и C++](#)

Обработка структурированных исключений в C++

12.11.2021 • 3 minutes to read

Основное различие между обработкой исключений С (SEH) и C++ заключается в том, что модель обработки исключений C++ работает в типах, а модель обработки структурированных исключений С работает с исключениями одного типа. в частности, `unsigned int`. То есть, исключения языка С идентифицируются целым числом без знака, тогда как исключения языка C++ идентифицируются типом данных. При возникновении структурированного исключения в С каждый возможный обработчик выполняет фильтр, который проверяет контекст исключения С и определяет, следует ли принимать исключение, передавать его другому обработчику или игнорировать. При возникновении исключения в языке C++ оно может быть любого типа.

Второе отличие заключается в том, что модель обработки структурированных исключений С называется *асинхронной*, поскольку исключения происходят во вторичном потоке управления. Механизм обработки исключений C++ полностью *синхронен*. Это означает, что исключения происходят только при возникновении.

При использовании параметра компилятора [/EHs](#) или [/EHsc](#) обработчики исключений C++ не обрабатывали структурированные исключения. Эти исключения обрабатываются только `__except` структуризованными обработчиками исключений или `__finally` структуризованными обработчиками завершения. Дополнительные сведения см. в разделе [структурированная обработка исключений \(C/C++\)](#).

В случае с параметром компилятора [/EHa](#), если в программе C++ возникает исключение С, оно может быть обработано структуризованным обработчиком исключений со связанным фильтром или `catch` обработчиком C++, в зависимости от того, какой из них динамически находится ближе к контексту исключения. Например, этот пример программы C++ вызывает исключение С внутри `try` контекста C++:

Пример. перехват исключения С в блоке catch C++

```
// exceptions_Exception_Handling_Differences.cpp
// compile with: /EHs
#include <iostream>

using namespace std;
void SEHFunc( void );

int main() {
    try {
        SEHFunc();
    }
    catch( ... ) {
        cout << "Caught a C exception." << endl;
    }
}

void SEHFunc() {
    __try {
        int x, y = 0;
        x = 5 / y;
    }
    __finally {
        cout << "In finally." << endl;
    }
}
```

```
In finally.
Caught a C exception.
```

Классы-оболочки исключений C

В простом примере, как показано выше, исключение C может быть перехвачено только многоточием (...) `catch` обработке. Обработчику не передается никакая информация о типе или характере исключения. Хотя этот метод работает, в некоторых случаях может потребоваться определить преобразование между двумя моделями обработки исключений, чтобы каждое исключение C было связано с конкретным классом. Для преобразования можно определить класс-оболочку исключения C, который может использоваться или быть производным от, чтобы атрибут определенного типа класса использовался в исключении C. Таким образом, каждое исключение C может обрабатываться отдельно от конкретного `catch` обработчика C++, а не из одного обработчика.

Класс-оболочка может иметь интерфейс, состоящий из некоторых функций-членов, которые определяют значение исключения; этот интерфейс может получать расширенные сведения о контексте исключения, предоставляемые моделью исключений языка C. Также может потребоваться определить конструктор по умолчанию и конструктор, принимающий `unsigned int` аргумент (для обеспечения базового представления исключения C) и побитовый конструктор копирования. Ниже приведена возможная реализация класса-оболочки исключения C.

```
// exceptions_Exception_Handling_Differences2.cpp
// compile with: /c
class SE_Exception {
private:
    SE_Exception() {}
    SE_Exception( SE_Exception& ) {}
    unsigned int nSE;
public:
    SE_Exception( unsigned int n ) : nSE( n ) {}
    ~SE_Exception() {}
    unsigned int getSeNumber() {
        return nSE;
    }
};
```

Чтобы использовать этот класс, установите пользовательскую функцию преобразования исключений C, которая вызывается внутренним механизмом обработки исключений каждый раз при возникновении исключения C. В функции перевода можно создать любое типизированное исключение (возможно `SE_Exception`, тип или класс, производный от `SE_Exception`), который может быть перехвачен соответствующим `catch` обработчиком C++. Вместо этого функция перевода может возвращать значение, указывающее, что она не обрабатывала исключение. Если функция перевода вызывает исключение C, вызывается метод `Terminate`.

Чтобы указать пользовательскую функцию перевода, вызовите функцию `_set_se_translator` с именем функции перевода в качестве одного аргумента. Функция перевода, которую вы пишете, вызывается один раз для каждого вызова функции в стеке, который имеет `try` блоки. Функция перевода по умолчанию отсутствует; Если не указать его путем вызова `_set_se_translator`, исключение C может быть перехвачено только `catch` обработчиком многоточия.

Пример. Использование пользовательской функции перевода

Например, следующий код устанавливает пользовательскую функцию преобразования, а затем создает исключение языка C, которое находится в оболочке класса `SE_Exception`:

```

// exceptions_Exception_Handling_Differences3.cpp
// compile with: /EHs
#include <stdio.h>
#include <eh.h>
#include <windows.h>

class SE_Exception {
private:
    SE_Exception() {}
    unsigned int nSE;
public:
    SE_Exception( SE_Exception& e ) : nSE(e.nSE) {}
    SE_Exception(unsigned int n) : nSE(n) {}
    ~SE_Exception() {}
    unsigned int getSeNumber() { return nSE; }
};

void SEFunc() {
    __try {
        int x, y = 0;
        x = 5 / y;
    }
    __finally {
        printf_s( "In finally\n" );
    }
}

void trans_func( unsigned int u, _EXCEPTION_POINTERS* pExp ) {
    printf_s( "In trans_func.\n" );
    throw SE_Exception( u );
}

int main() {
    _set_se_translator( trans_func );
    try {
        SEFunc();
    }
    catch( SE_Exception e ) {
        printf_s( "Caught a __try exception with SE_Exception.\n" );
        printf_s( "nSE = 0x%x\n", e.getSeNumber() );
    }
}

```

```

In trans_func.
In finally
Caught a __try exception with SE_Exception.
nSE = 0xc0000094

```

См. также

[Сочетание исключений С \(структурированные\) и C++](#)

Structured Exception Handling (C/C++)

12.11.2021 • 3 minutes to read

Структурированная обработка исключений (SEH) — это расширение Майкрософт для языка С для обработки определенных исключительных ситуаций кода, таких как ошибки оборудования, корректно. хотя Windows и Microsoft C++ поддерживают SEH, рекомендуется использовать обработку исключений C++ в стандарте ISO. Это делает код более переносимым и гибким. Однако для поддержки существующего кода или определенных типов программ все равно может потребоваться использовать SEH.

Зависит от корпорации Майкрософт:

Грамматика

```
try-except-statement :  
    __try compound-statement __except ( expression ) compound-statement  
  
try-finally-statement :  
    __try compound-statement __finally compound-statement
```

Remarks

С помощью SEH можно обеспечить правильное освобождение ресурсов, например блоков памяти и файлов, в случае непредвиденного завершения выполнения. Можно также решить определенные проблемы (например, недостаток памяти), используя краткий структурированный код, который не полагается на `goto` инструкции или тщательное тестирование кодов возврата.

`try-except` Инструкции и, `try-finally` упоминаемые в этой статье, являются расширениями Майкрософт для языка С. Они поддерживают SEH, позволяя приложениям получать контроль над программой после событий, которые в иных ситуациях привели бы к завершению выполнения. Хотя обработка ошибок SEH работает с исходными файлами C++, она не была создана специально для этого языка. При использовании SEH в программе C++, компилируемой с помощью параметра `/EHa` или `/EHsc`, вызываются деструкторы для локальных объектов, но другое поведение выполнения может отличаться от ожидаемого. Иллюстрации см. в примере далее в этой статье. В большинстве случаев вместо SEH рекомендуется использовать [обработку исключений C++](#) в стандарте ISO, которую также поддерживает компилятор Microsoft C++. С помощью обработки исключений C++ можно повысить переносимость кода и обрабатывать исключения любого типа.

При наличии кода C, использующего SEH, можно смешивать его с кодом C++, использующим обработку исключений C++. Дополнительные сведения см. в разделе [Handle Structured Exceptions in C++](#).

Существует два механизма SEH.

- [Обработчики исключений](#) или `__except` блоки, которые могут реагировать на исключение или закрыть его.
- [Обработчики завершения](#) или `__finally` блоки, которые вызываются всегда, если исключение вызывает завершение или нет.

Эти два типа обработчиков различаются, но тесно связаны с процессом, называемым очисткой *стека*. при возникновении структурированного исключения Windows ищет недавно установленный обработчик

исключений, который сейчас является активным. Обработчик может выполнить одно из трех действий:

- не распознать исключение и передать управление другим обработчикам;
- распознать исключение, но отбросить его;
- распознать исключение и обработать его.

Обработчик исключений, распознавший исключение, может находиться за пределами функции, которая выполнялась, когда возникло исключение. В стеке он может находиться в функции намного выше.

Выполняемая в настоящее время функция и все прочие функции в кадре стека завершаются. Во время этого процесса стек будет *развернут*. То есть локальные нестатические переменные завершенных функций удаляются из стека.

По мере развертывания стека операционная система вызывает все обработчики завершения, которые были написаны для каждой функции. Используя обработчик завершения, вы очищаете ресурсы, которые в противном случае останутся открытыми из-за аварийного завершения. Если вы ввели критическую секцию, вы можете выйти из него в обработчике завершения. После завершения работы программы можно выполнять другие задачи по обслуживанию, такие как закрытие и удаление временных файлов.

Дальнейшие действия

- [Написание обработчика исключений](#)
- [Написание обработчика завершения](#)
- [Обработка структурированных исключений в C++](#)

Пример

Как упоминалось ранее, деструкторы для локальных объектов вызываются, если вы используете SEH в программе C++ и компилируете его с помощью `/Eha` `/EHsc` параметра или. Однако поведение во время выполнения может отличаться от ожидаемого, если вы также используете исключения C++. В этом примере демонстрируются эти различия в поведении.

```

#include <stdio.h>
#include <Windows.h>
#include <exception>

class TestClass
{
public:
    ~TestClass()
    {
        printf("Destroying TestClass!\r\n");
    }
};

__declspec(noinline) void TestCPPEX()
{
#ifdef CPPEX
    printf("Throwing C++ exception\r\n");
    throw std::exception("");
#else
    printf("Triggering SEH exception\r\n");
    volatile int *pInt = 0x00000000;
    *pInt = 20;
#endif
}

__declspec(noinline) void TestExceptions()
{
    TestClass d;
    TestCPPEX();
}

int main()
{
    __try
    {
        TestExceptions();
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
        printf("Executing SEH __except block\r\n");
    }

    return 0;
}

```

Если вы используете `/EHsc` для компиляции этого кода, но локальный макрос элемента управления теста `CPPEX` не определен, `TestClass` деструктор не выполняется. Выходные данные выглядят следующим образом:

```

Triggering SEH exception
Executing SEH __except block

```

Если используется `/EHsc` для компиляции кода и `CPPEX` определяется с помощью `/DCPPEX` (чтобы выдавалось исключение C++), `TestClass` выполняется деструктор, а выходные данные выглядят следующим образом:

```

Throwing C++ exception
Destroying TestClass!
Executing SEH __except block

```

Если используется `/EHa` для компиляции кода, деструктор выполняет исключение, вызванное вызовом

`TestClass` исключения с помощью или с `std::throw` помощью SEH. То есть определяет, `CPPEX` определен ли тип или нет. Выходные данные выглядят следующим образом:

```
Throwing C++ exception
Destroying TestClass!
Executing SEH __except block
```

Дополнительные сведения см. в разделе [/EH](#) (модель обработки исключений).

КОНЕЦ Только для систем Microsoft

См. также раздел

[Обработка исключений](#)

[Ключевые слова](#)

`<exception>`

[Ошибки и обработка исключений](#)

[Структурированная обработка исключений \(Windows\)](#)

Написание обработчика исключений

12.11.2021 • 2 minutes to read

Обработчики исключений обычно используются для ответа на конкретные ошибки. Можно использовать синтаксис обработки исключений, чтобы фильтровать все исключения, отличные от тех, которые вы знаете, как обработать. Прочие исключения должны передаваться другим обработчикам (возможно, в библиотеке среды выполнения или ОС), созданным для поиска этих конкретных исключений.

Обработчики исключений используют оператор `try-except`.

Дополнительные сведения

- [Оператор `try-except`](#)
- [Написание фильтра исключений](#)
- [Создание исключений программного обеспечения](#)
- [Исключения оборудования](#)
- [Ограничения обработчиков исключений](#)

См. также

[Structured Exception Handling \(C/C++\)](#)

ИНСТРУКЦИЯ `try-except`

12.11.2021 • 4 minutes to read

`try-except` Оператор — это расширение для Microsoft , которое поддерживает структурированную обработку исключений в языках C и C++.

```
// . . .
__try {
    // guarded code
}
__except ( /* filter expression */ ) {
    // termination code
}
// . . .
```

Грамматика

```
try-except-statement :
    __try compound-statement __except ( expression ) compound-statement
```

Remarks

`try-except` Оператор является расширением Microsoft для языков C и C++. Она позволяет целевым приложениям получать контроль над возникновением событий, которые обычно завершают выполнение программы. Такие события называются *структурированными исключениями* или *исключениями* для коротких. Механизм, который обрабатывает эти исключения, называется *структурной обработкой исключений* (SEH).

Связанные сведения см. в описании [оператора try-finally](#).

Исключения могут быть основаны на оборудовании или программном обеспечении. Структурированная обработка исключений полезна даже в том случае, когда приложения не могут полностью восстанавливаться после исключений оборудования или программного обеспечения. SEH позволяет отображать сведения об ошибках и захватывать внутреннее состояние приложения, чтобы помочь в диагностике проблемы. Это особенно полезно для периодических проблем, которые не очень просты в воспроизведении.

NOTE

Структурированная обработка исключений поддерживается в Win32 для исходных файлов как на C, так и на C++. Однако он не предназначен специально для C++. Для того чтобы ваш код лучше переносился, лучше использовать механизм обработки исключений языка C++. Кроме того, этот механизм отличается большей гибкостью, поскольку может обрабатывать исключения любого типа. Для программ на C++ рекомендуется использовать собственные операторы обработки исключений C++: [try](#), [catch](#) и [Throw](#) .

Составной оператор после `__try` предложения — *тело* или *зашитенный* раздел. `__except` Выражение также называется критерием *фильтра*. Его значение определяет, как обрабатываются исключения. Составной оператор после предложения `__except` является обработчиком исключения. Обработчик задает действия, выполняемые при возникновении исключения во время выполнения раздела *body*. Выполнение происходит следующим образом:

- Сначала выполняется защищенный раздел.
- Если исключение при этом не возникает, выполнение переходит в инструкцию, стоящую после предложения `_except`.
- Если во время выполнения защищенного раздела возникает исключение или в любой подпрограмме вызывается защищенный раздел, `_except` выражение вычисляется. Возможны три значения.
 - `EXCEPTION_CONTINUE_EXECUTION` (-1) Исключение закрыто. Выполнение продолжается в точке, в которой возникло исключение.
 - `EXCEPTION_CONTINUE_SEARCH` (0) исключение не распознано. Программа переходит к поиску обработчика в стеке (сначала находятся выражения с оператором `try-except`, а затем обработчики со следующим наивысшим приоритетом).
 - `EXCEPTION_EXECUTE_HANDLER` (1) распознано исключение. Передайте управление обработчику исключений, выполнив `_except` составной оператор, а затем продолжайте выполнение после `_except` блока.

`_except` Выражение вычисляется как выражение С. Оно ограничено одним значением, оператором условного выражения или оператором-запятой. Если требуется более сложная обработка, выражение может вызывать процедуру, которая возвращает одно из этих трех значений.

Каждое приложение может иметь свой собственный обработчик исключений.

Не допускается переход к `_try` оператору, но он допустим для перехода из одной инструкции. Обработчик исключений не вызывается, если процесс завершается в процессе выполнения `try-except` инструкции.

Для совместимости с предыдущими версиями `_try`, `_except` и `_leave` являются синонимами для `_try`, `_except` и, если не `_leave` задан параметр компилятора [/Za \(Отключить расширения языка\)](#).

`_leave` Ключевое слово

`_leave` Ключевое слово допустимо только в защищенном разделе `try-except` оператора, и его результат — переход к концу защищенного раздела. Выполнение продолжается с первого оператора, следующего за обработчиком исключений.

`goto` Оператор также может перейти из защищенного раздела и не снизит производительность, как это делается в операторе `try-finally`. Это связано с тем, что очистка стека не происходит. Однако рекомендуется использовать `_leave` ключевое слово, а не `goto` оператор. Причина заключается в том, что вы менее вероятно намерены создать ошибку программирования, если защищенный раздел большой или сложный.

Встроенные функции обработки структурированных исключений

Структурированная обработка исключений предоставляет две встроенные функции, которые можно использовать с `try-except` инструкцией: `GetExceptionCode` и `GetExceptionInformation`.

`GetExceptionCode` Возвращает код (32-разрядное целое число) исключения.

Встроенная функция `GetExceptionInformation` возвращает указатель на структуру `EXCEPTION_POINTERS`, содержащую дополнительные сведения об исключении. Через этот указатель можно обращаться к состоянию компьютера, которое существовало в момент возникновения аппаратного исключения. Его структура выглядит следующим образом.

```
typedef struct _EXCEPTION_POINTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Типы указателей `PEXCEPTION_RECORD` и определяются `PCONTEXT` в включаемом файле `<winnt.h>` и `_EXCEPTION_RECORD` `_CONTEXT` определяются в включаемом файле. `<excpt.h>`

Можно использовать `GetExceptionCode` в обработчике исключений. Однако можно использовать `GetExceptionInformation` только в пределах выражения фильтра исключений. Информация, на которую он указывает, обычно находится в стеке и больше недоступна при передаче управления в обработчик исключений.

Встроенная функция `абнормалтерминатион` доступна в обработчике завершения. Он возвращает 0, если тело оператора `try-finally` завершается последовательно. В остальных случаях функция возвращает 1.

`<excpt.h>` определяет некоторые альтернативные имена для этих встроенных функций:

`GetExceptionCode` — это эквивалент `_exception_code`

`GetExceptionInformation` — это эквивалент `_exception_info`

`AbnormalTermination` — это эквивалент `_abnormal_termination`

Пример

```

// exceptions_try_except_Statement.cpp
// Example of try-except and try-finally statements
#include <stdio.h>
#include <windows.h> // for EXCEPTION_ACCESS_VIOLATION
#include <excpt.h>

int filter(unsigned int code, struct _EXCEPTION_POINTERS *ep)
{
    puts("in filter.");
    if (code == EXCEPTION_ACCESS_VIOLATION)
    {
        puts("caught AV as expected.");
        return EXCEPTION_EXECUTE_HANDLER;
    }
    else
    {
        puts("didn't catch AV, unexpected.");
        return EXCEPTION_CONTINUE_SEARCH;
    }
}

int main()
{
    int* p = 0x00000000;    // pointer to NULL
    puts("hello");
    __try
    {
        puts("in try");
        __try
        {
            puts("in try");
            *p = 13;      // causes an access violation exception;
        }
        __finally
        {
            puts("in finally. termination: ");
            puts(AbnormalTermination() ? "\tabnormal" : "\tnormal");
        }
    }
    __except(filter(GetExceptionCode(), GetExceptionInformation()))
    {
        puts("in except");
    }
    puts("world");
}

```

Вывод

```

hello
in try
in try
in filter.
caught AV as expected.
in finally. termination:
    abnormal
in except
world

```

См. также

[Написание обработчика исключений](#)

[Structured Exception Handling \(C/C++\)](#)

[Ключевые слова](#)

Написание фильтра исключений

12.11.2021 • 2 minutes to read

Исключение можно обработать посредством перехода на уровень обработчика исключений или путем продолжения выполнения. Вместо того чтобы использовать код обработчика исключений для обработки исключения и перебора, можно использовать критерий *фильтра*, чтобы устраниТЬ проблему. Затем, возвращая `EXCEPTION_CONTINUE_EXECUTION` (-1), вы можете возобновить нормальную последовательность, не очистив стек.

NOTE

После некоторых исключений возобновление невозможно. Если для такого исключения *Фильтр* принимает значение -1, система создает новое исключение. При вызове `RaiseException` определяется, будет ли продолжаться исключение.

Например, следующий код использует вызов функции в критерии *фильтра*: Эта функция обрабатывает проблему, а затем возвращает значение -1, чтобы возобновить нормальный поток управления:

```
// exceptions_Writing_an_Exception_Filter.cpp
#include <windows.h>
int main() {
    int Eval_Exception( int );

    __try {}

    __except ( Eval_Exception( GetExceptionCode( )) ) {
        ;
    }

}
void ResetVars( int ) {}
int Eval_Exception ( int n_except ) {
    if ( n_except != STATUS_INTEGER_OVERFLOW &&
        n_except != STATUS_FLOAT_OVERFLOW ) // Pass on most exceptions
    return EXCEPTION_CONTINUE_SEARCH;

    // Execute some code to clean up problem
    ResetVars( 0 ); // initializes data to 0
    return EXCEPTION_CONTINUE_EXECUTION;
}
```

Рекомендуется использовать вызов функции в выражении *фильтра*, когда *Фильтр* должен выполнить все сложнее. Вычисление выражения приводит к выполнению функции, в данном случае — `Eval_Exception`.

Обратите внимание на использование `GetExceptionCode` для определения исключения. Эта функция должна вызываться внутри выражения фильтра `__except` инструкции. `Eval_Exception` не удается вызвать `GetExceptionCode`, но ему должен быть передан код исключения.

Если исключение не вызвано переполнением при операции с целыми числами или числами с плавающей запятой, этот обработчик передает управление другому обработчику. В этом случае обработчик вызывает функцию (`ResetVars` — это только пример, а не функция API), чтобы сбросить некоторые глобальные переменные. `__except` Блок операторов, который в этом примере пуст, не может быть выполнен, так как `Eval_Exception` никогда не возвращает `EXCEPTION_EXECUTE_HANDLER` (1).

Вызов функции — хороший способ работы со сложными выражениями фильтров. Удобны также две другие возможности языка C:

- условный оператор;
- оператор "запятая".

Условный оператор часто полезен здесь. Его можно использовать для проверки конкретного кода возврата и возврата одного из двух различных значений. Например, фильтр в следующем коде распознает исключение только в том случае, если исключение `STATUS_INTEGER_OVERFLOW` :

```
__except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ? 1 : 0 ) {
```

Цель условного оператора в этом случае — в основном, обеспечить ясность, так как следующий код дает такие же результаты.

```
__except( GetExceptionCode() == STATUS_INTEGER_OVERFLOW ) {
```

Условный оператор полезнее в ситуациях, когда может потребоваться, чтобы фильтр выдает значение-1, `EXCEPTION_CONTINUE_EXECUTION` .

Оператор "запятая" позволяет выполнять несколько выражений последовательно. Затем он возвращает значение последнего выражения. Например, в следующем коде код исключения сохраняется в переменной и затем проверяется.

```
__except( nCode = GetExceptionCode(), nCode == STATUS_INTEGER_OVERFLOW )
```

См. также:

[Написание обработчика исключений](#)
[Structured Exception Handling \(C/C++\)](#)

Создание исключений программного обеспечения

12.11.2021 • 2 minutes to read

Некоторые из самых распространенных источников ошибок программы не отмечены системой как исключения. Например, при попытке выделить блок памяти при недостаточной памяти среда выполнения или функция API не создает исключение, но возвращает код ошибки.

Тем не менее можно рассматривать любое условие как исключение, выявляя это условие в коде, а затем сообщая о нем, вызвав функцию `RaiseException`. Отмечая ошибки таким образом, можно использовать преимущества структурированной обработки исключений в любом типе ошибки времени выполнения.

Чтобы использовать структурированную обработку исключений с ошибками, выполните следующие действия.

- Определите собственный код исключения для события.
- Вызов `RaiseException` при обнаружении проблемы.
- Используйте фильтры обработки исключений для проверки определенного кода исключения.

В `<winerror.h>` файле отображается формат кодов исключений. Чтобы проверить, что определяемый код не будет конфликтовать с существующим кодом исключения, задайте для третьего наиболее значимого бита значение 1. Следует установить четыре наиболее значимых бита, как показано в следующей таблице.

BITS	РЕКОМЕНДУЕМЫЙ ДВОИЧНЫЙ ПАРАМЕТР	ОПИСАНИЕ
31-30	11	Эти два бита описываются основное состояние кода: 11 = ошибка, 00 = успех, 01 = информация, 10 = предупреждение.
29	1	Бит клиента. Установите значение 1 для пользовательских кодов.
28	0	Зарезервированный бит. (Оставьте значение 0.)

При желании для первых двух битов можно задать параметр, отличный от двоичного параметра 11, хотя параметр "ошибка" подходит для большинства исключений. Помните, что биты 29 и 28 следует установить так, как показано в предыдущей таблице.

В результате код ошибки должен иметь старшие четыре бита, равные шестнадцатеричному E. например, следующие определения определяют коды исключений, которые не конфликтуют с кодами исключений Windows. (Хотя может потребоваться проверить, какие коды используются сторонними библиотеками DLL.)

```
#define STATUS_INSUFFICIENT_MEM      0xE0000001  
#define STATUS_FILE_BAD_FORMAT       0xE0000002
```

После определения кода исключения его можно использовать для создания исключения. Например, следующий код вызывает `STATUS_INSUFFICIENT_MEM` исключение в ответ на проблему выделения памяти:

```
lpstr = _malloc( nBufferSize );
if (lpstr == NULL)
    RaiseException( STATUS_INSUFFICIENT_MEM, 0, 0, 0 );
```

Если требуется просто создать исключение, можно установить для трех последних параметров значение 0. Три последних параметра полезны при передаче дополнительной информации и установке флага, который запрещает обработчикам продолжать выполнение. дополнительные сведения см. в описании функции [RaiseException](#) в Windows SDK.

Затем можно проверить определенные коды в фильтрах обработки исключений. Пример:

```
__try {
    ...
}
__except (GetExceptionCode() == STATUS_INSUFFICIENT_MEM ||
          GetExceptionCode() == STATUS_FILE_BAD_FORMAT )
```

См. также

[Написание обработчика исключений](#)

[Структурированная обработка исключений \(C/C++\)](#)

Исключения оборудования

12.11.2021 • 2 minutes to read

Большинство стандартных исключений, распознаваемых операционной системой, являются аппаратными исключениями. Windows распознает несколько низкоуровневых программных исключений, однако обычно операционная система обрабатывает их лучше всего.

Windows сопоставляет аппаратные ошибки различных процессоров с кодами исключений, которые описаны в этом разделе. В некоторых случаях процессор может создавать только часть из этих исключений. Windows предварительно обрабатывает сведения об исключении и передает соответствующий код исключения.

В следующей таблице приводятся сведения об аппаратных исключениях, распознаваемых Windows.

КОД ИСКЛЮЧЕНИЯ	ПРИЧИНА ИСКЛЮЧЕНИЯ
STATUS_ACCESS_VIOLATION	Чтение или запись в недоступный адрес в памяти.
STATUS_BREAKPOINT	Обнаружение аппаратной точки останова; используется только в отладчиках.
STATUS_DATATYPE_MISALIGNMENT	Чтение и запись в данные по адресу с неверным выравниванием. Например, 16-разрядные записи должны выравниваться относительно машинных слов с границей 2 байта. (Неприменимо к процессорам Intel 80x86.)
STATUS_FLOAT_DIVIDE_BY_ZERO	Деление значения с плавающей запятой на 0,0.
STATUS_FLOAT_OVERFLOW	Превышение максимальной положительной экспоненты с плавающей запятой.
STATUS_FLOAT_UNDERFLOW	Превышение модуля наименьшей отрицательной экспоненты с плавающей запятой.
STATUS_FLOATING_RESEVERED_OPERAND	Использование зарезервированного формата с плавающей запятой (недопустимое использование формата).
STATUS_ILLEGAL_INSTRUCTION	Попытка выполнить код инструкции, не определенной процессором.
STATUS_PRIVILEGED_INSTRUCTION	Выполнение инструкции, которая не разрешена в текущем режиме машины.
STATUS_INTEGER_DIVIDE_BY_ZERO	Деление значения целочисленного типа на 0.
STATUS_INTEGER_OVERFLOW	Попытка выполнения операции, которая превышает диапазон целочисленного типа.
STATUS_SINGLE_STEP	Выполнение одной инструкции в пошаговом режиме; используется только в отладчиках.

Большинство исключений, перечисленных в приведенной выше таблице, должны обрабатываться отладчиками, операционной системой или другим низкоуровневым кодом. Ваш код не должен обрабатывать эти ошибки, за исключением ошибок целочисленных значений и значений с плавающей запятой. Поэтому в большинстве случаев вам потребуется использовать фильтр обработки исключений, чтобы игнорировать исключения (значение 0). В противном случае низкоуровневые механизмы не смогут правильно реагировать на них. Однако можно предпринять соответствующие меры предосторожности в отношении потенциального воздействия этих низкоуровневых ошибок, [написав обработчики завершения](#).

См. также

[Написание обработчика исключений](#)

[Structured Exception Handling \(C/C++\)](#)

Ограничения обработчиков исключений

12.11.2021 • 2 minutes to read

Основным ограничением использования обработчиков исключений в коде является то, что нельзя использовать `goto` оператор для перехода в `__try` блок операторов. Входить в этот блок необходимо только через обычный поток управления. Можно выйти из `__try` блока операторов, и вы можете вкладывать обработчики исключений по своему выбору.

См. также

[Написание обработчика исключений](#)

[Structured Exception Handling \(C/C++\)](#)

Написание обработчика завершения

12.11.2021 • 2 minutes to read

В отличие от обработчика исключений, обработчик завершения выполняется всегда независимо от того, завершен ли в обычном режиме защищенный блок кода. Единственным назначением обработчика завершения должна быть проверка правильности закрытия таких ресурсов, как память, дескрипторы и файлы, независимо от того, как завершается выполнение фрагмента кода.

Обработчики завершения используют оператор `try-finally`.

Дополнительные сведения

- [Оператор try-finally](#)
- [Освобождение ресурсов](#)
- [Время действий в обработке исключений](#)
- [Ограничения обработчиков завершения](#)

См. также:

[Structured Exception Handling \(C/C++\)](#)

ИНСТРУКЦИЯ `try-finally`

12.11.2021 • 3 minutes to read

`try-finally` Оператор — это расширение для Microsoft , которое поддерживает структурированную обработку исключений в языках C и C++.

Синтаксис

Следующий синтаксис описывает `try-finally` инструкцию:

```
// . . .
__try {
    // guarded code
}
__finally {
    // termination code
}
// . . .
```

Грамматика

```
try-finally-statement :
    __try compound-statement __finally compound-statement
```

`try-finally` Инструкция является расширением Microsoft для языков C и C++, которые позволяют целевым приложениям гарантировать выполнение кода очистки при прерывании выполнения блока кода. Очистка включает такие задачи, как отмена распределения памяти, закрытие файлов и освобождение их дескрипторов. Оператор `try-finally` особенно полезен для подпрограмм, в которых в нескольких местах выполняется проверка на наличие ошибок, способных вызвать преждевременное возвращение из подпрограммы.

Связанные сведения и пример кода см. в разделе [try-except](#) инструкция. Дополнительные сведения об структурированной обработке исключений в целом см. в разделе [структурированная обработка исключений](#). Дополнительные сведения об обработке исключений в управляемых приложениях с помощью C++/CLI см. в разделе [Обработка /clr исключений](#).

NOTE

Структурированная обработка исключений поддерживается в Win32 для исходных файлов как на C, так и на C++. Однако она не предназначена специально для C++. Для того чтобы ваш код лучше переносился, лучше использовать механизм обработки исключений языка C++. Кроме того, этот механизм отличается большей гибкостью, поскольку может обрабатывать исключения любого типа. Для программ на C++ рекомендуется использовать механизм обработки исключений C++ (операторы `try`, `catch` и `throw`).

Составной оператор после предложения `__try` — это защищенный раздел. Составной оператор после предложения `__finally` является обработчиком завершения. Обработчик задает набор действий, которые выполняются при выходе из защищенного раздела, вне зависимости от того, выходит ли он из защищенного раздела на исключение (аномальное завершение) или по стандартному переходу (нормальное завершение).

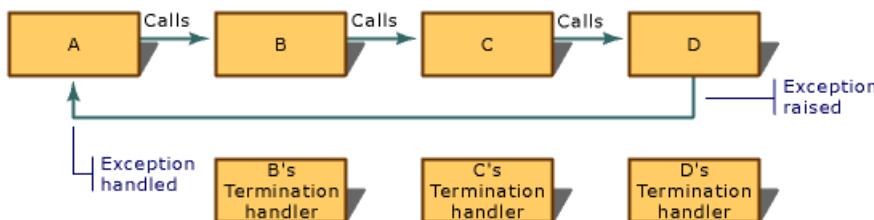
Управление переходит к оператору `__try` в процессе обычного последовательного выполнения (передачи управления дальше). Когда элемент управления вводит `__try`, связанный с ним обработчик становится активным. Если поток элементов управления достигает конца блока `try`, выполнение продолжается следующим образом.

1. Вызывается обработчик завершения.
2. По окончании работы обработчика завершения выполнение продолжается после оператора `__finally`. Однако защищенный раздел заканчивается (например, с помощью из `goto` защищенного тела или `return` оператора) обработчик завершения выполняется *до того, как* поток управления перемещается из защищенного раздела.

`__finally` Инструкция не блокирует поиск соответствующего обработчика исключений.

Если в блоке возникает исключение `__try`, операционная система должна найти обработчик для исключения, иначе программа завершится ошибкой. Если обработчик найден, `__finally` выполняются все блоки и выполнение возобновляется в обработчике.

Например, предположим, ряд вызовов функций связывает функцию A с функцией D, как показано на следующем рисунке. Каждая функция имеет один обработчик завершения. Если исключение создается в функции D и обрабатывается в функции A, обработчики завершения вызываются в том порядке, в котором система освобождает стек: D, C и B.



Порядок выполнения обработчиков завершения

NOTE

Поведение `try-finally` отличается от других языков, поддерживающих использование `finally`, например C#. Один `__try` может иметь либо, но не оба, `__finally` и `__except`. Если оба следует использовать одновременно, оператор `try-except` должен включать внутренней оператор `try-finally`. Правила, задающие время выполнения каждого блока, также различаются.

Для совместимости с предыдущими версиями, `__try` `__finally` и `__leave` являются синонимами для `__try`, `__finally` и `__leave` если не задан параметр компилятора [/Za](#) (отключить расширения языка).

Ключевое слово `__leave`

`__leave` Ключевое слово допустимо только в защищенном разделе `try-finally` оператора, и его результат — переход к концу защищенного раздела. Выполнение продолжается с первого оператора в обработчике завершения.

`goto` Инструкция также может выйти из защищенного раздела, но снижает производительность, так как она вызывает раскрутку стека. Эта `__leave` инструкция более эффективна, так как она не приводит к очистке стека.

Аварийное завершение

Выход из `try-finally` инструкции с помощью функции времени выполнения `longjmp` считается аномальным завершением. Переход к оператору `__try` недопустим, но выход из него допускается.

`__finally` Должны быть выполнены все инструкции, активные между точкой отправления (нормальное завершение `__try` блока) и назначением (`__except` блок, обрабатывающий исключение). Это называется *локальной раскруткой*.

Если `__try` блок преждевременно завершается по какой-либо причине, включая переход за пределы блока, система выполняет связанный `__finally` блок как часть процесса очистки стека. В таких случаях `AbnormalTermination` функция возвращает значение `true`. Если вызывается из `__finally` блока, в противном случае возвращается значение `false`.

Обработчик завершения не вызывается, если процесс завершается в процессе выполнения `try-finally` инструкции.

КОНЕЦ Только для систем Майкрософт

См. также раздел

[Написание обработчика завершения](#)

[Structured Exception Handling \(C/C++\)](#)

[Ключевые слова](#)

[Синтаксис обработчика завершения](#)

Освобождение ресурсов

12.11.2021 • 2 minutes to read

Во время выполнения обработчика завершения может быть неизвестно, какие ресурсы были получены до вызова обработчика завершения. Возможно, `__try` блок инструкций был прерван до получения всех ресурсов, так что не все ресурсы были открыты.

Чтобы быть в безопасности, перед продолжением очистки для обработки завершения следует проверить, какие ресурсы открыты. Ниже описана рекомендованная процедура.

1. Инициализируйте дескрипторы со значением `null`.
2. В `__try` блоке инструкций получите ресурсы. Для дескрипторов задаются положительные значения при получении ресурса.
3. В `__finally` блоке инструкций выпустите каждый ресурс, соответствующий маркер или переменная флага которого не равны нулю или не имеют значение `null`.

Пример

Например, в следующем коде обработчик завершения используется для закрытия трех файлов и освобождения блока памяти. Эти ресурсы были получены в `__try` блоке инструкций. Перед очисткой ресурса код сначала проверяет, был ли получен ресурс.

```
// exceptions_Cleaning_up_Resources.cpp
#include <stdlib.h>
#include <malloc.h>
#include <stdio.h>
#include <windows.h>

void fileOps() {
    FILE *fp1 = NULL,
        *fp2 = NULL,
        *fp3 = NULL;
    LPVOID lpvoid = NULL;
    errno_t err;

    __try {
        lpvoid = malloc( BUFSIZ );

        err = fopen_s(&fp1, "ADDRESS.DAT", "w+" );
        err = fopen_s(&fp2, "NAMES.DAT", "w+" );
        err = fopen_s(&fp3, "CARS.DAT", "wt" );
    }
    __finally {
        if ( fp1 )
            fclose( fp1 );
        if ( fp2 )
            fclose( fp2 );
        if ( fp3 )
            fclose( fp3 );
        if ( lpvoid )
            free( lpvoid );
    }
}

int main() {
    fileOps();
}
```

См. также:

[Написание обработчика завершения](#)
[Structured Exception Handling \(C/C++\)](#)

Время обработки исключений. Общие сведения

12.11.2021 • 2 minutes to read

Обработчик завершения выполняется независимо от того, как `__try` завершается блок инструкции.

Причины включают в себя выход из `__try` блока, `longjmp` инструкцию, которая передает управление из блока и очищает стек из-за обработки исключений.

NOTE

Компилятор Microsoft C++ поддерживает две формы `setjmp` `longjmp` операторов и. Быстрая версия обходит обработку завершения, однако является более эффективной. Чтобы использовать эту версию, включите файл `<setjmp.h>`. Другая версия поддерживает обработку завершения, как описано в предыдущем абзаце. Чтобы использовать эту версию, включите файл `<setjmpex.h>`. Увеличение производительности быстрой версии зависит от конфигурации оборудования.

Операционная система выполняет все обработчики завершения в нужном порядке, прежде чем сможет быть выполнен какой-либо другой код, включая тело обработчика исключений.

Если причина прерывания — исключение, система должна сначала выполнить фильтровую часть одного или нескольких обработчиков исключений, а потом определять объект завершения. Используется следующий порядок событий:

1. Выдается исключение.
2. Система просматривает иерархию активных обработчиков исключений и выполняет фильтр обработчика с наивысшим приоритетом. Это обработчик исключений, который недавно был установлен и наиболее глубоко вложен в блоки и вызовы функций.
3. Если этот фильтр передает управление (Возвращает значение 0), процесс продолжится до тех пор, пока не будет найден фильтр, который не передает управление.
4. Если этот фильтр возвращает значение -1, выполнение продолжается там, где было вызвано исключение, и не происходит завершения.
5. Если фильтр возвращает 1, возникают следующие события:
 - Система очищает стек: удаляет все кадры стека между местом возникновения исключения и кадром стека, который содержит обработчик исключений.
 - По мере освобождения стека выполняется каждый обработчик завершения в стеке.
 - Выполняется сам обработчик исключений.
 - Контроль переходит к строке кода после окончания этого обработчика исключений.

См. также раздел

[Написание обработчика завершения](#)

[Structured Exception Handling \(C/C++\)](#)

Ограничения обработчиков завершения

12.11.2021 • 2 minutes to read

Нельзя использовать `goto` инструкцию для перехода в `__try` блок операторов или `__finally` блок операторов. Входить в этот блок необходимо только через обычный поток управления. (Однако можно перейти к `__try` блоку операторов.) Кроме того, нельзя вложить обработчик исключений или обработчик завершения внутри `__finally` блока.

Некоторые виды кода, разрешенные в обработчике завершения, позволяют получить сомнительные результаты, поэтому следует использовать их с осторожностью, если вообще. Один из них — `goto` оператор, который выполняет переход из `__finally` блока операторов. Если блок выполняется как часть нормального завершения работы, ничего необычного не происходит. Но если система перематывает стек, то эта очистка останавливается. Затем текущая функция получает управление как при отсутствии аномального завершения.

`return` Оператор в `__finally` блоке операторов представляет примерно такую же ситуацию. Управление возвращается непосредственно вызывающему объекту функции, содержащей обработчик завершения. Если система перематывает стек, этот процесс прерывается. Затем программа будет работать так, как будто исключения не возникали.

См. также

[Написание обработчика завершения](#)

[Structured Exception Handling \(C/C++\)](#)

Перенос исключений между потоками

12.11.2021 • 9 minutes to read

компилятор Microsoft C++ (MSVC) поддерживает *транспортное исключение* из одного потока в другой. Передача исключений позволяет перехватывать исключение в одном потоке и затем обеспечить видимость того, что исключение возникло в другом потоке. Например, эту возможность можно использовать для создания многопоточного приложения, где первостепенный поток обрабатывает все исключения, создаваемые его второстепенными потоками. Передача исключений в основном полезна для разработчиков, создающих системы или библиотеки параллельного программирования, чтобы реализовать перенос исключений, MSVC предоставляет `exception_ptr` тип и функции `current_exception`, `rethrow_exception` и `make_exception_ptr`.

Синтаксис

```
namespace std
{
    typedef unspecified exception_ptr;
    exception_ptr current_exception();
    void rethrow_exception(exception_ptr p);
    template<class E>
        exception_ptr make_exception_ptr(E e) noexcept;
}
```

Параметры

Unspecified

Неуказанный внутренний класс, используемый для реализации типа `exception_ptr`.

ш

Объект `exception_ptr`, который ссылается на исключение.

&

Класс, представляющий исключение.

&

Экземпляр класса `E` параметров.

Возвращаемое значение

Функция `current_exception` возвращает объект `exception_ptr`, который ссылается на исключение, выполняемое в данный момент. Если никакое исключение не выполняется, функция возвращает объект `exception_ptr`, не связанный ни с одним исключением.

`make_exception_ptr` Функция возвращает `exception_ptr` объект, который ссылается на исключение, заданное параметром `e`.

Remarks

Сценарий

Предположим, что требуется создать приложение, которое может масштабироваться для выполнения переменного объема работы. Для достижения этой цели создается многопоточное приложение, где исходный основной поток создает столько второстепенных потоков, сколько необходимо для

выполнения задачи. Второстепенные потоки помогают первичному потоку распоряжаться ресурсами, распределять нагрузку и повышать пропускную способность. Благодаря распределению работы многопоточное приложение работает быстрее, чем однопоточное.

Однако если второстепенный поток вызывает исключение, необходимо обеспечить его обработку основным потоком. Это обусловлено тем, что требуется обеспечить обработку исключений приложением единообразным, унифицированным способом независимо от количества второстепенных потоков.

Решение

Для выполнения указанного выше сценария стандарт C++ поддерживает передачу исключений между потоками. Если дополнительный поток создает исключение, это исключение станет *текущим исключением*. По аналогии с реальным миром, текущее исключение говорится *в полете*. Текущее исключение находится в полете со времени создания до возвращения результата обработчиком исключений, перехватывающим его.

Вторичный поток может перехватить текущее исключение в `catch` блоке, а затем вызвать `current_exception` функцию для хранения исключения в `exception_ptr` объекте. Объект `exception_ptr` должен быть доступным для второстепенного и первичного потоков. Например, объект `exception_ptr` может быть глобальной переменной, доступ к которой регулируется мьютексом. Термин *транспорт — исключение* означает, что исключение в одном потоке можно преобразовать в форму, к которой может получить доступ другой поток.

Далее основной поток вызывает функцию `rethrow_exception`, которая извлекает исключение из объекта `exception_ptr` и затем вызывает его. При вызове исключения оно становится текущим исключением в основном потоке. Это означает, что создается видимость того, что исключение возникает в основном потоке.

Наконец, основной поток может перехватить текущее исключение в `catch` блоке, а затем обработать его или создать обработчик исключений более высокого уровня. Также основной поток может проигнорировать исключение и позволить процессу завершиться.

Большинству приложений не требуется передавать исключения между потоками. Однако эта возможность полезна в системе параллельных вычислений, поскольку система может распределить работу между второстепенными потоками, процессорами или ядрами. В среде параллельных вычислений один выделенный поток может обрабатывать все исключения из второстепенных потоков и может предоставлять единообразную модель обработки исключений любому приложению.

Дополнительные сведения о предложении Комитета по стандартам C++ можно получить, найдя в Интернете документ с номером N2179 «Поддержка передачи исключений между потоками в языке».

Модели обработки исключений и параметры компилятора

Модель обработки исключений приложения определяет, возможен ли в нем перехват и передача исключения. Visual C++ поддерживает 3 модели, которые могут обрабатывать исключения C++, исключения структурированной обработки (SEH) и исключения среды CLR. Используйте параметры компилятора `/EH` и `/CLR`, чтобы указать модель обработки исключений приложения.

Только следующее сочетание параметров компилятора и операторов программирования может передавать исключение. Другие сочетания либо не могут перехватывать исключения, либо могут перехватывать, но не могут передавать исключения.

- Параметр компилятора `/EHa` и `catch` оператор может передавать исключения SEH и C++.
- Параметры компилятора `/EHa`, `/EHs` и `/EHsc`, а также `catch` инструкции могут передавать исключения C++.

- Параметр компилятора /CLR и `catch` инструкция могут передавать исключения C++. Параметр компилятора /CLR подразумевает спецификацию параметра /EHsc. Обратите внимание, что компилятор не поддерживает передачу управляемых исключений. Это обусловлено тем, что управляемые исключения, которые являются производными от [класса System.Exception](#), уже являются объектами, которые можно перемещать между потоками с помощью средств общей среды выполнения лангуаже.

IMPORTANT

Рекомендуется указывать параметр компилятора /EHsc и перехватывать только исключения C++. Вы предоставляете себе угрозу безопасности, если используете параметр компилятора /EHsc или /CLR и `catch` оператор с многоточием `Exception (catch(...)`). Возможно, вы планируете использовать `catch` инструкцию для записи нескольких конкретных исключений. Однако оператор `catch(...)` перехватывает все исключения C++ и SEH, включая непредвиденные, которые должны приводить к сбою. Если непредвиденное исключение игнорируется или обрабатывается неверно, вредоносный код может использовать эту возможность, чтобы подорвать безопасность программы.

Использование

В следующих разделах описывается передача исключений с помощью `exception_ptr` типа, а также `current_exception` `rethrow_exception` функций, и `make_exception_ptr`.

Тип exception_ptr

Используйте объект `exception_ptr` для ссылки на текущее исключение или экземпляр указанного пользователем исключения. В реализации Microsoft исключение представлено структурой `EXCEPTION_RECORD`. Каждый объект `exception_ptr` содержит поле ссылки на исключение, указывающее на копию структуры `EXCEPTION_RECORD`, представляющую исключение.

При объявлении переменной `exception_ptr` эта переменная не связана ни с одним исключением. То есть в поле ссылки на исключение находится значение NULL. Такой объект `exception_ptr` называется `exception_ptr null`.

Используйте функцию `current_exception` ИЛИ `make_exception_ptr` для назначения исключения объекту `exception_ptr`. При назначении исключения переменной `exception_ptr` поле ссылки на исключение переменной указывает на копию исключения. При нехватке памяти для копирования исключения поле ссылки на исключение указывает на копию исключения `std::bad_alloc`. Если `current_exception` функция или `make_exception_ptr` не может скопировать исключение по какой бы то ни было другим причинам, функция вызывает функцию `Terminate` для выхода из текущего процесса.

Несмотря на свое имя, объект `exception_ptr` не является указателем. Он не подчиняется семантике указателя и не может использоваться с операторами доступа к членам указателя (`->`) или косвенного обращения (`*`). Объект `exception_ptr` не имеет открытых данных-членов и функций-членов.

Сравнение

Можно использовать операторы равенства (`==`) и неравенства (`!=`) для сравнения двух объектов `exception_ptr`. Эти операторы не сравнивают бинарное значение (битовый шаблон) структур `EXCEPTION_RECORD`, которые представляют исключения. Вместо этого операторы сравнивают адреса в поле ссылки на исключение объектов `exception_ptr`. Поэтому `exception_ptr` со значением `null` и значение `NULL` при сравнении считаются равными.

Функция current_exception

Вызовите `current_exception` функцию в `catch` блоке. Если исключение находится в полете и `catch` блок может перехватить исключение, `current_exception` функция возвращает `exception_ptr` объект, который ссылается на исключение. В противном случае функция возвращает объект `exception_ptr` со значением `null`.

Сведения

`current_exception` Функция захватывает исключение, которое находится в полете, независимо от того, `catch` указывает ли инструкция [объявление исключения](#).

Деструктор текущего исключения вызывается в конце `catch` блока, если исключение не будет создано заново. Но даже при вызове функции `current_exception` в деструкторе эта функция возвращает объект `exception_ptr`, который ссылается на текущее исключение.

Последующие вызовы функции `current_exception` возвращают объекты `exception_ptr`, которые ссылаются на различные копии текущего исключения. Соответственно, при сравнении объекты не признаются равными, поскольку они ссылаются на различные копии, даже если эти копии имеют одинаковые бинарные значения.

Исключения SEH

При использовании параметра компилятора /EHа можно перехватывать исключение SEH в `catch` блоке C++. Функция `current_exception` возвращает объект `exception_ptr`, который ссылается на исключение SEH. И `rethrow_exception` функция создает исключение SEH, если вы вызываете его с помощью `exception_ptr` объекта сетранспортера в качестве аргумента.

`current_exception` Функция возвращает значение NULL `exception_ptr` при вызове в `__finally` обработчике завершения SEH, `__except` обработчике исключений или `__except` критерии фильтра.

Переданное исключение не поддерживает вложенные исключения. Вложенное исключение возникает, если во время обработки одного исключения возникает другое исключение. Если производится перехват вложенного исключения, данные-член `EXCEPTION_RECORD.ExceptionRecord` указывает на цепочку структур `EXCEPTION_RECORD`, описывающих соответствующие исключения. Функция `current_exception` не поддерживает вложенные исключения, поскольку она возвращает объект `exception_ptr`, данные-член `ExceptionRecord` которого обнулен.

Если производится перехват исключения SEH, необходимо обеспечить распределение памяти, на которую ссылается любой указатель в массиве данных-членов `EXCEPTION_RECORD.ExceptionInformation`. Необходимо гарантировать, что эта память действительна в течение времени существования соответствующего объекта `exception_ptr` и что эта память освобождается, когда объект `exception_ptr` удаляется.

Можно использовать возможности преобразователя структурированного исключения (SE) вместе с возможностью передачи исключений. Если исключение SEH преобразуется в исключение C++, функция `current_exception` возвращает `exception_ptr`, который ссылается на преобразованное исключение вместо исходного исключения SEH. Затем функция `rethrow_exception` вызывает преобразованное, а не исходное исключение. Дополнительные сведения о SE функциях переводчиков см. в разделе [_set_se_translator](#).

Функция `rethrow_exception`

После сохранения перехваченного исключения в объект `exception_ptr` основной поток может обработать этот объект. В основном потоке вызовите функцию `rethrow_exception`, указав объект `exception_ptr` в качестве аргумента. Функция `rethrow_exception` извлекает исключение из объекта `exception_ptr` и затем вызывает это исключение в контексте основного потока. Если параметр `p` `rethrow_exception` функции имеет значение NULL `exception_ptr`, функция создает исключение `std::bad_exception`.

Извлеченное исключение теперь является текущим исключением в основном потоке, и его можно обработать как любое другое исключение. Если вы перехватите исключение, его можно немедленно обработать или использовать `throw` оператор, чтобы отправить его в обработчик исключений более высокого уровня. В противном случае можно ничего не делать и позволить системному обработчику исключений по умолчанию завершить процесс.

Функция `make_exception_ptr`

Функция `make_exception_ptr` принимает экземпляр класса в качестве аргумента и затем возвращает `exception_ptr`, который ссылается на этот экземпляр. Обычно объект [класс исключений](#) указывается в качестве аргумента функции `make_exception_ptr`, однако аргументом может быть любой объект класса.

Вызов `make_exception_ptr` функции эквивалентен созданию исключения C++, его перехвату в `catch` блоке и последующему вызову `current_exception` функции для возврата `exception_ptr` объекта, который ссылается на исключение. Реализация Microsoft для функции `make_exception_ptr` является более эффективной, чем создание и последующий перехват исключения.

Приложение обычно не требует функции `make_exception_ptr`, и мы не рекомендуем использовать ее.

Пример

В следующем примере стандартное исключение C++ и пользовательское исключение C++ передаются из одного потока в другой.

```
// transport_exception.cpp
// compile with: /EHsc /MD
#include <windows.h>
#include <stdio.h>
#include <exception>
#include <stdexcept>

using namespace std;

// Define thread-specific information.
#define THREADCOUNT 2
exception_ptr aException[THREADCOUNT];
int           aArg[THREADCOUNT];

DWORD WINAPI ThrowExceptions( LPVOID );
// Specify a user-defined, custom exception.
// As a best practice, derive your exception
// directly or indirectly from std::exception.
class myException : public std::exception {
};

int main()
{
    HANDLE aThread[THREADCOUNT];
    DWORD ThreadID;

    // Create secondary threads.
    for( int i=0; i < THREADCOUNT; i++ )
    {
        aArg[i] = i;
        aThread[i] = CreateThread(
            NULL,          // Default security attributes.
            0,             // Default stack size.
            (LPTHREAD_START_ROUTINE) ThrowExceptions,
            (LPVOID) &aArg[i], // Thread function argument.
            0,             // Default creation flags.
            &ThreadID); // Receives thread identifier.
        if( aThread[i] == NULL )

```

```

    {
        printf("CreateThread error: %d\n", GetLastError());
        return -1;
    }
}

// Wait for all threads to terminate.
WaitForMultipleObjects(THREADCOUNT, aThread, TRUE, INFINITE);
// Close thread handles.
for( int i=0; i < THREADCOUNT; i++ ) {
    CloseHandle(aThread[i]);
}

// Rethrow and catch the transported exceptions.
for ( int i = 0; i < THREADCOUNT; i++ ) {
    try {
        if (aException[i] == NULL) {
            printf("exception_ptr %d: No exception was transported.\n", i);
        }
        else {
            rethrow_exception( aException[i] );
        }
    }
    catch( const invalid_argument & ) {
        printf("exception_ptr %d: Caught an invalid_argument exception.\n", i);
    }
    catch( const myException & ) {
        printf("exception_ptr %d: Caught a myException exception.\n", i);
    }
}
}

// Each thread throws an exception depending on its thread
// function argument, and then ends.
DWORD WINAPI ThrowExceptions( LPVOID lpParam )
{
    int x = *((int*)lpParam);
    if (x == 0) {
        try {
            // Standard C++ exception.
            // This example explicitly throws invalid_argument exception.
            // In practice, your application performs an operation that
            // implicitly throws an exception.
            throw invalid_argument("A C++ exception.");
        }
        catch ( const invalid_argument & ) {
            aException[x] = current_exception();
        }
    }
    else {
        // User-defined exception.
        aException[x] = make_exception_ptr( myException() );
    }
    return TRUE;
}

```

```

exception_ptr 0: Caught an invalid_argument exception.
exception_ptr 1: Caught a myException exception.

```

Требования

Заголовок:<exception>

См. также

Обработка исключений

/EH (модель обработки исключений)

/clr (компиляция среды выполнения)

Утверждение и сообщения об ошибках, предоставленные пользователем (C++)

12.11.2021 • 2 minutes to read

Язык C++ поддерживает три механизма обработки ошибок, которые помогут вам выполнить отладку приложения: [директива #error](#), ключевое слово [static_assert](#) и [макрос assert, _ASSERT, _wassert](#). Все три механизма создают сообщения об ошибках, а два из них также проверяют утверждения программного обеспечения. Программное утверждение определяет условие, которое должно выполняться на определенном этапе работы программы. Если утверждение времени компиляции ложно, компилятор создает диагностическое сообщение и ошибку компиляции. Если утверждение времени выполнения ложно, операционная система выводит диагностическое сообщение и закрывает приложение.

Remarks

Жизненный цикл приложения состоит из этапа предварительной обработки, этапа компиляции и этапа времени выполнения. Каждый механизм обработки ошибок обращается к отладочной информации, доступной в ходе одного из этих этапов. Для эффективной отладки выберите механизм, обеспечивающий требуемую информацию об этом этапе.

- [Директива #error](#) действует во время предварительной обработки. Она безусловно выводит определенное пользователем сообщение и вызывает сбой компиляции с ошибкой. Сообщение может содержать текст, управляемый директивами препроцессора, но никакие результирующие выражения не вычисляются.
- Объявление [static_assert](#) действует во время компиляции. Оно проверяет утверждения программного обеспечения, которые определяются заданным пользователем целочисленным выражением, допускающим преобразование в логическое значение. Если выражение равно нулю (ложно), компилятор выдает определенное пользователем сообщение и компиляция завершается сбоем с ошибкой.

`static_assert` Объявление особенно полезно для отладки шаблонов, так как аргументы шаблона могут включаться в указанное пользователем выражение.

- [Макрос assert, _ASSERT, _wassert](#) макрос действует во время выполнения. Он вычисляет определенное пользователем выражение, и если результат равен нулю, система выводит диагностическое сообщение и закрывает приложение. Многие другие макросы, такие как [_ASSERT](#) и [_ASERTE](#), похожи на этот макрос, но выдают различные системные и определяемые пользователем диагностические сообщения.

См. также раздел

[Директива #error \(C/C++\)](#)

[Макрос assert, _assert, _wassert](#)

[Макросы _ASSERT, _ASERTE и _ASSERT_EXPR](#)

[static_assert](#)

[Макрос _STATIC_ASSERT](#)

[Шаблоны](#)

static_assert

12.11.2021 • 2 minutes to read

Проверяет программное утверждение во время компиляции. Если указанное константное выражение имеет значение `false`, компилятор отображает указанное сообщение, если оно предоставлено, и компиляция завершается ошибкой C2338; в противном случае объявление не оказывает никакого влияния.

Синтаксис

```
static_assert( constant-expression, string-literal );  
static_assert( constant-expression ); // C++17 (Visual Studio 2017 and later)
```

Параметры

Константное выражение

Целочисленное константное выражение, которое можно преобразовать в логическое значение. Если вычисленное выражение равно нулю (`false`), то отображается параметр *строкового литерала* и компиляция завершается ошибкой. Если выражение не равно нулю (`true`), `static_assert` объявление не оказывает никакого влияния.

Строковый литерал

Сообщение, которое отображается, если параметр *константного выражения* равен нулю. Сообщение представляет собой строку символов в [базовой кодировке](#) компилятора; то есть не [многобайтовые или расширенные символы](#).

Remarks

Параметр *константного выражения* `static_assert` объявления представляет *программное утверждение*. Программное утверждение определяет условие, которое должно выполняться на определенном этапе работы программы. Если условие имеет значение `true`, `static_assert` объявление не оказывает никакого влияния. Если условие имеет значение `false`, то утверждение не выполняется, компилятор отображает сообщение в параметре *строки-литерала*, и компиляция завершается ошибкой. В Visual Studio 2017 и более поздних версиях параметр строкового литерала является необязательным.

В `static_assert` объявлении проверяется программное утверждение во время компиляции. В отличие от этого, [макросы Assert и функции _ASSERT и _wassert](#) проверяют программное утверждение во время выполнения и приводят к затратам времени выполнения в пространстве или времени. `static_assert` Объявление особенно полезно для отладки шаблонов, так как аргументы шаблона могут быть добавлены в параметр *константного выражения*.

Компилятор проверяет `static_assert` объявление на наличие синтаксических ошибок при обнаружении объявления. Компилятор вычисляет параметр *константного выражения* немедленно, если он не зависит от параметра шаблона. В противном случае компилятор вычисляет параметр *константного выражения* при создании экземпляра шаблона. Таким образом, компилятор может вывести одно диагностическое сообщение, когда встретит объявление, а второе — когда будет создавать экземпляр шаблона.

`static_assert` Ключевое слово можно использовать в пространстве имен, класса или области видимости блока. (`static_assert` Ключевое слово является техническим объявлением, хотя оно не вводит новое имя в программу, так как оно может использоваться в области видимости пространства имен.)

Описание `static_assert` области пространства имен

В следующем примере `static_assert` объявление имеет область пространства имен. Поскольку компилятору известен размер типа `void *`, выражение вычисляется немедленно.

Пример: `static_assert` с областью пространства имен

```
static_assert(sizeof(void *) == 4, "64-bit code generation is not supported.");
```

Описание `static_assert` с областью видимости класса

В следующем примере `static_assert` объявление имеет область класса. `static_assert` Проверяет, является ли параметр шаблона *обычным старым типом данных* (Pod). Компилятор проверяет `static_assert` объявление при объявлении, но не вычисляет параметр *константного выражения* до тех пор, пока не `basic_string` будет создан экземпляр шаблона класса в `main()`.

Пример: `static_assert` с областью класса

```
#include <type_traits>
#include <iostream>
namespace std {
template <class CharT, class Traits = std::char_traits<CharT> >
class basic_string {
    static_assert(std::is_pod<CharT>::value,
                 "Template argument CharT must be a POD type in class template basic_string");
    // ...
};
}

struct NonPOD {
    NonPOD(const NonPOD &);
    virtual ~NonPOD() {}
};

int main()
{
    std::basic_string<char> bs;
```

Описание `static_assert` с областью видимости блока

В следующем примере `static_assert` объявление имеет область видимости блока. `static_assert` Проверяет, равен ли размер структуры вмпаже виртуальной памяти PageSize системы.

Пример: `static_assert` в области видимости блока

```
#include <sys/param.h> // defines PAGESIZE
class VMMClient {
public:
    struct VMPage { // ...
    };
    int check_pagesize() {
        static_assert(sizeof(VMPage) == PAGESIZE,
            "Struct VMPage must be the same size as a system virtual memory page.");
        // ...
    }
    // ...
};
```

См. также

[Утверждение и сообщения об ошибках, предоставленные пользователем \(C++\)](#)

[Директива #error \(C/C++\)](#)

[Макрос assert, _assert, _wassert](#)

[Шаблоны](#)

[Набор символов ASCII](#)

[Объявления и определения](#)

Обзор модулей в C++

12.11.2021 • 6 minutes to read

В C++ 20 представлены *модули*, современное решение для компонентов библиотек и программ C++. Модуль — это набор файлов исходного кода, которые компилируются независимо от [единиц трансляции](#), которые их импортируют. Модули устраниют или значительно снижают многие проблемы, связанные с использованием файлов заголовков, а также могут сократить время компиляции. Макросы, директивы препроцессора и неэкспортированные имена, объявленные в модуле, не видны и поэтому не влияют на компиляцию записи преобразования, которая импортирует модуль. Модули можно импортировать в любом порядке, не заботясь о переопределениях макросов. Объявления в импортируемой записи не участвуют в разрешении перегрузки или поиске имен в импортированном модуле. После компиляции модуля результаты сохраняются в двоичном файле, который описывает все экспортированные типы, функции и шаблоны. Этот файл может обрабатываться гораздо быстрее, чем файл заголовка, и может использоваться компилятором каждый раз, когда модуль импортируется в проект.

Модули можно использовать параллельно с файлами заголовков. Исходный файл C++ может импортировать модули, а также `#include` файлы заголовков. В некоторых случаях файл заголовка можно импортировать как модуль, а не в текстовом `#included` препроцессором. Рекомендуется, чтобы новые проекты использовали модули, а не файлы заголовков, насколько это возможно. Для больших существующих проектов в рамках активной разработки мы рекомендуем поэкспериментировать с преобразованием устаревших заголовков в модули, чтобы определить, будет ли получено осмысленное сокращение времени компиляции.

Включение модулей в компиляторе Microsoft C++

начиная с Visual Studio 2019 версии 16,2, модули не полностью реализованы в компиляторе Microsoft C++. С помощью функции "модули" можно создавать модули с одной секцией и импортировать модули стандартной библиотеки, предоставляемые корпорацией Майкрософт. Чтобы включить поддержку модулей, Скомпилируйте с помощью [/экспериментал: module](#) и `/std: c++ + Latest`. в проекте Visual Studio щелкните правой кнопкой мыши узел проекта в **обозреватель решений** и выберите пункт **свойства**. Задайте в раскрывающемся списке **Конфигурация** значение **все конфигурации**, а затем выберите **Свойства конфигурации > язык C/C++ > включить модули C++ (экспериментальные)**.

Модуль и код, который его использует, должны быть скомпилированы с теми же параметрами компилятора.

Использование стандартной библиотеки C++ в качестве модулей

Корпорация Майкрософт позволяет импортировать стандартную библиотеку C++ как модули, хотя это не указано в стандарте C++ 20. Импортировав стандартную библиотеку C++ как модули, а не `#including` их через заголовочные файлы, вы можете ускорить компиляцию в зависимости от размера проекта. Библиотека является компонентом следующих модулей:

- STD. Regex предоставляет содержимое заголовка `<regex>`
- STD. FileSystem предоставляет содержимое заголовка `<filesystem>`
- STD. Memory предоставляет содержимое заголовка `<memory>`
- STD. Threading предоставляет содержимое заголовков `<atomic>`, `<condition_variable>`, `<future>`, `<mutex>`, `<shared_mutex>` и `<thread>`
- STD. Core предоставляет все остальное в стандартной библиотеке C++

Чтобы использовать эти модули, просто добавьте объявление импорта в начало файла исходного кода.

Пример:

```
import std.core;
import std.regex;
```

Чтобы использовать модуль стандартной библиотеки Майкрософт, скомпилируйте программу с параметрами [/EHsc](#) и [/MD](#).

Простой пример

В следующем примере показано простое определение модуля в исходном файле с именем `foo`. **ИКСКС**. Расширение `.икскс` требуется для файлов интерфейса модуля в Visual Studio. В этом примере файл интерфейса содержит определение функции, а также объявление. Однако определения можно также поместить в один или несколько отдельных файлов (как показано в следующем примере). Инструкция `Export Module` `foo` указывает, что этот файл является основным интерфейсом для модуля с именем `Foo`. Модификатор в `f()` указывает, что эта функция будет видима при `Foo` импорте другой программой или модулем. Обратите внимание, что модуль ссылается на пространство имен `Bar`.

```
export module Foo;

#define ANSWER 42

namespace Bar
{
    int f_internal() {
        return ANSWER;
    }

    export int f() {
        return f_internal();
    }
}
```

Файл `MyProgram.cpp` использует объявление **импорта** для доступа к имени, экспортируемому `Foo`. Обратите внимание, что имя отображается `Bar` здесь, но не все его члены. Также обратите внимание, что макрос `ANSWER` не отображается.

```
import Foo;
import std.core;

using namespace std;

int main()
{
    cout << "The result of f() is " << Bar::f() << endl; // 42
    // int i = Bar::f_internal(); // C2039
    // int j = ANSWER; //C2065
}
```

Объявление импорта может использоваться только в глобальной области видимости.

Реализация модулей

Можно создать модуль с одним файлом интерфейса (`.икскс`), который экспортирует имена и включает

реализации всех функций и типов. Можно также разместить реализации в одном или нескольких отдельных файлах реализации, аналогично использованию `h` и `CPP Files`. `export` Ключевое слово используется только в файле интерфейса. Файл реализации может **импортировать** другой модуль, но не может иметь `export` никаких имен. Файлы реализации могут называться любым расширением. Файл интерфейса и набор файлов реализации, которые его поддерживают, обрабатываются как особый тип записи преобразования, называемой **единицей модуля**. Имя, объявленное в любом файле реализации, автоматически отображается во всех остальных файлах в той же единице модуля.

Для больших модулей модуль можно разделить на несколько единиц модуля, называемых *секциями*. Каждая секция состоит из файла интерфейса, поддерживаемого одним или несколькими файлами реализации. (начиная с Visual Studio 2019 версии 16.2, секции еще не полностью реализованы.)

Модули, пространства имен и поиск с зависимостью от аргументов

Правила для пространств имен в модулях такие же, как и в любом другом коде. Если экспортируется объявление в пространстве имен, то включающее пространство имен (за исключением неэкспортированных элементов) также будет неявно экспортано. Если пространство имен экспортируется явным образом, экспортируются все объявления в этом определении пространства имен.

При выполнении уточняющего запроса, зависящего от аргумента, для разрешения перегрузки в импортируемой записи преобразования компилятор учитывает функции, объявленные в одной и той же записи преобразования (включая интерфейсы модулей), как и при определении типа аргументов функции.

Секции модулей

NOTE

Этот раздел предназначен для полноты. Секции еще не реализованы в компиляторе Microsoft C++.

Модуль может быть разбит на *Секции*, каждый из которых состоит из файла интерфейса, а также от нуля или нескольких файлов реализации. Раздел модуля аналогичен модулю, за исключением того, что он разделяет владение всеми объявлениями во всем модуле. Все имена, экспортанные с помощью файлов интерфейса секционирования, импортируются и экспортано повторно с помощью первичного файла интерфейса. Имя секции должно начинаться с имени модуля, за которым следует двоеточие. Объявления в любой из секций видны во всем модуле. Специальные меры предосторожности не требуются, чтобы избежать ошибок с одним определением (С операционными данными). Можно объявить имя (функцию, класс и т. д.) в одной секции и определить их в другой. Файл реализации секции начинается следующим образом:

```
module Foo:part1
```

и файл интерфейса секционирования начинается следующим образом:

```
export module Foo:part1
```

Чтобы получить доступ к объявлениям в другой секции, необходимо импортировать ее секцию, но она может использовать только имя секции, а не имя модуля:

```
module Foo:part2;
import :part1;
```

Основная единица интерфейса должна импортировать и повторно экспортировать все файлы разделов интерфейса модуля следующим образом:

```
export import :part1
export import :part2
...
```

Основная единица интерфейса может импортировать файлы реализации секций, но не может экспортировать их, так как эти файлы не могут экспортировать какие-либо имена. Это позволяет модулю удержать внутренние сведения о реализации модуля.

Модули и файлы заголовков

Файлы заголовков можно включить в исходный файл модуля, поместив `#include` директиву перед объявлением модуля. Эти файлы считаются частью *глобального фрагмента модуля*. Модуль может видеть только имена в *фрагменте глобального модуля*, который содержит явно заголовков. Фрагмент глобального модуля содержит только символы, которые фактически используются.

```
// MyModuleA.cpp

#include "customlib.h"
#include "anotherlib.h"

import std.core;
import MyModuleB;

//... rest of file
```

Для управления импортируемыми модулями можно использовать традиционный заголовочный файл.

```
// MyProgram.h
import std.core;
#ifndef DEBUG_LOGGING
import std.filesystem;
#endif
```

Импортированные файлы заголовков

NOTE

Этот раздел не содержит практических действий. Устаревшие импорты еще не реализованы в компиляторе Microsoft C++.

Некоторые заголовки достаточно самодостаточны, что их можно использовать с помощью ключевого слова `Import`. Основное различие между импортированным заголовком и импортированным модулем заключается в том, что все определения препроцессора в заголовке видимы в программе импорта сразу после оператора `import`. (Определения препроцессора в файлах, содержащихся в этом заголовке, *не видны*.)

```
import <vector>
import "myheader.h"
```

См. также

модуль, импорт, экспорт

МОДУЛЬ, ИМПОРТ, ЭКСПОРТ

12.11.2021 • 2 minutes to read

Модуль, Импорт и `export` **объявления доступны в C++ 20**, и требуется параметр компилятора `/экспериментал: module` вместе с `/std: C++ Latest`. Дополнительные сведения см. в разделе Общие сведения о [модулях в C++](#).

module

Поместите объявление **модуля** в начало файла реализации модуля, чтобы указать, что содержимое файла принадлежит к именованному модулю.

```
module ModuleA;
```

Экспорт

Используйте объявление **модуля экспорта** для основного файла интерфейса модуля, который должен иметь расширение **.икскс**:

```
export module ModuleA;
```

В файле интерфейса используйте `export` Модификатор для имен, которые должны быть частью общего интерфейса:

```
// ModuleA.ixx

export module ModuleA;

namespace Bar
{
    export int f();
    export double d();
    double internal_f(); // not exported
}
```

Неэкспортированные имена невидимы для кода, который импортирует модуль:

```
//MyProgram.cpp

import module ModuleA;

int main() {
    Bar::f(); // OK
    Bar::d(); // OK
    Bar::internal_f(); // Ill-formed: error C2065: 'internal_f': undeclared identifier
}
```

`export` Ключевое слово не может присутствовать в файле реализации модуля. Когда `export` применяется к имени пространства имен, экспортируются все имена в пространстве имен.

ИМПОРТ

Используйте объявление **импорта**, чтобы сделать имена модуля видимыми в программе. Объявление импорта должно располагаться после объявления модуля и после любой директивы #include, но перед любыми объявлениями в файле.

```
module ModuleA;

#include "custom-lib.h"
import std.core;
import std.regex;
import ModuleB;

// begin declarations here:
template <class T>
class Baz
{...};
```

Remarks

Импорт и **модуль** рассматриваются как ключевые слова, только если они появляются в начале логической строки:

```
// OK:
module ;
module module-name
import :
import <
import "
import module-name
export module ;
export module module-name
export import :
export import <
export import "
export import module-name

// Error:
int i; module ;
```

Блок, относящийся только к системам Microsoft

В Microsoft C++ **Импорт** и **модуль** маркеров всегда являются идентификаторами и никогда не являются ключевыми словами, когда они используются в качестве аргументов макроса.

Пример

```
#define foo(...) __VA_ARGS__
foo(
import // Always an identifier, never a keyword
)
```

Завершение Microsoft для конкретных

См. также:

[Обзор модулей в C++](#)

Шаблоны (C++)

12.11.2021 • 6 minutes to read

Шаблоны служат основанием для универсального программирования на C++. В качестве строго типизированного языка C++ требует, чтобы все переменные имели конкретный тип, либо явно объявленный программистом, либо выведенный компилятором. Однако многие структуры данных и алгоритмы выглядят одинаково независимо от типа, на котором они работают. Шаблоны позволяют определить операции класса или функции и предоставить пользователю указание конкретных типов, с которыми должны работать эти операции.

Определение и использование шаблонов

Шаблон — это конструкция, которая создает обычный тип или функцию во время компиляции на основе аргументов, предоставленных пользователем для параметров шаблона. Например, можно определить шаблон функции следующим образом:

```
template <typename T>
T minimum(const T& lhs, const T& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

Приведенный выше код описывает шаблон для универсальной функции с одним параметром типа *T*, возвращаемое значение и параметры вызова (LHS и RHS) всех этих типов. Вы можете присвоить параметру типа любое имя, но, по правилам, наиболее часто используются буквы в одном верхнем регистре. *T* является параметром шаблона; `typename` ключевое слово говорит о том, что этот параметр является заполнителем для типа. При вызове функции компилятор заменит каждый экземпляр `T` с конкретным аргументом типа, который либо задается пользователем, либо выведенным компилятором. Процесс, в котором компилятор создает класс или функцию из шаблона, называется *созданием экземпляра шаблона*. `minimum<int>` — это экземпляр шаблона `minimum<T>`.

В других случаях пользователь может объявить экземпляр шаблона, специализированный для типа `int`. Предположим, что `get_a()` и `get_b()` — это функции, возвращающие `int`:

```
int a = get_a();
int b = get_b();
int i = minimum<int>(a, b);
```

Однако, поскольку это шаблон функции, и компилятор может вывести тип `T` из аргументов *a* и *b*, можно вызвать его так же, как обычная функция:

```
int i = minimum(a, b);
```

Когда компилятор встречает последнюю инструкцию, он создает новую функцию, в которой каждое вхождение *T* в шаблоне заменяется на `int`:

```
int minimum(const int& lhs, const int& rhs)
{
    return lhs < rhs ? lhs : rhs;
}
```

Правила, определяющие, как компилятор выполняет выведение типов в шаблонах функций, основаны на правилах для обычных функций. Дополнительные сведения см. в разделе [разрешение перегрузки вызовов шаблона функции](#).

Параметры типа

В `minimum` приведенном выше шаблоне Обратите внимание на то, что параметр типа `T` не определен каким-либо образом, пока он не будет использован в параметрах вызова функции, где добавляются квалификаторы `Const` и `Reference`.

Практически не существует ограничения на количество параметров типа. Несколько параметров разделяются запятыми:

```
template <typename T, typename U, typename V> class Foo{};
```

Ключевое слово `class` эквивалентно `typename` в данном контексте. Предыдущий пример можно выразить следующим образом:

```
template <class T, class U, class V> class Foo{};
```

Оператор с многоточием (...) можно использовать для определения шаблона, принимающего произвольное число из нуля или более параметров типа:

```
template<typename... Arguments> class vtclass;

vtclass< > vtinstance1;
vtclass<int> vtinstance2;
vtclass<float, bool> vtinstance3;
```

В качестве аргумента типа можно использовать любой встроенный или определяемый пользователем тип. Например, можно использовать `std::Vector` в стандартной библиотеке для хранения переменных типа `int`, `double`, `std::String`, `MyClass`, `const MyClass *`, `MyClass&` и т. д. Основным ограничением при использовании шаблонов является то, что аргумент типа должен поддерживать любые операции, применяемые к параметрам типа. Например, если мы вызываем метод `minimum` using `MyClass`, как в следующем примере:

```
class MyClass
{
public:
    int num;
    std::wstring description;
};

int main()
{
    MyClass mc1 {1, L"hello"};
    MyClass mc2 {2, L"goodbye"};
    auto result = minimum(mc1, mc2); // Error! C2678
}
```

Будет создана ошибка компилятора, так как не `MyClass` предоставляет перегрузку для < оператора.

Нет необходимости в том, что аргументы типа для любого конкретного шаблона принадлежат к одной и той же иерархии объектов, хотя можно определить шаблон, обеспечивающий такое ограничение. Объектно-ориентированные методики можно сочетать с шаблонами. Например, можно сохранить производный * в векторе `<Base*>`. Обратите внимание, что аргументы должны быть указателями

```
vector<MyClass*> vec;
    MyDerived d(3, L"back again", time(0));
    vec.push_back(&d);

// or more realistically:
vector<shared_ptr<MyClass>> vec2;
vec2.push_back(make_shared<MyDerived>());
```

Основные требования, которые `std::vector` и другие контейнеры стандартной библиотеки, накладываются на элементы `T`, являются `T` копируемыми и конструируемыми.

Параметры, не являющиеся типами

В отличие от универсальных типов на других языках, таких как C# и Java, шаблоны C++ поддерживают *Параметры, не являющиеся типами*, также называемые параметрами значений. Например, можно предоставить постоянное целочисленное значение, чтобы указать длину массива, как в этом примере, подобно классу `std:: Array` в стандартной библиотеке:

```
template<typename T, size_t L>
class MyArray
{
    T arr[L];
public:
    MyArray() { ... }
};
```

Обратите внимание на синтаксис в объявлении шаблона. `size_t` Значение передается в качестве аргумента шаблона во время компиляции и должно быть `const` или `constexpr` выражением. Используйте его следующим образом:

```
MyArray<MyClass*, 10> arr;
```

Другие виды значений, включая указатели и ссылки, могут передаваться как параметры, не являющиеся типами. Например, можно передать указатель на функцию или объект функции для настройки некоторой операции внутри кода шаблона.

Выведение типа для параметров шаблона, не являющихся типами

в Visual Studio 2017 и более поздних версиях, в режиме `/std: c++ 17`, компилятор выводит тип аргумента шаблона, не являющегося типом, который объявлен с помощью `auto`:

```
template <auto x> constexpr auto constant = x;

auto v1 = constant<5>;      // v1 == 5, decltype(v1) is int
auto v2 = constant<true>;    // v2 == true, decltype(v2) is bool
auto v3 = constant<'a'>;     // v3 == 'a', decltype(v3) is char
```

Шаблоны в качестве параметров шаблона

Шаблон может быть параметром шаблона. В этом примере `MyClass2` имеет два параметра шаблона: параметр `typeName T` и параметр шаблона `arr`:

```
template<typename T, template<typename U, int I> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
    U u; //Error. U not in scope
};
```

Поскольку сам параметр `arr` не имеет тела, его имена параметров не требуются. На самом деле, является ошибкой ссылаться на имена `TypeName` или параметров класса `arr` в теле `MyClass2`. По этой причине имена параметров типа `arr` можно опустить, как показано в следующем примере:

```
template<typename T, template<typename, int> class Arr>
class MyClass2
{
    T t; //OK
    Arr<T, 10> a;
};
```

Аргументы шаблона по умолчанию

Шаблоны классов и функций могут иметь аргументы по умолчанию. Если шаблон имеет аргумент по умолчанию, его можно оставить неопределенным при его использовании. Например, шаблон `std:: Vector` имеет аргумент по умолчанию для распределителя:

```
template <class T, class Allocator = allocator<T>> class vector;
```

В большинстве случаев класс по умолчанию `std:: Vector` распределитель приемлем, поэтому вы используете такой же вектор:

```
vector<int> myInts;
```

Но при необходимости можно указать пользовательский распределитель следующим образом:

```
vector<int, MyAllocator> ints;
```

При наличии нескольких аргументов шаблона все аргументы после первого аргумента по умолчанию должны иметь аргументы по умолчанию.

При использовании шаблона, параметры которого заданы по умолчанию, используйте пустые угловые скобки:

```
template<typename A = int, typename B = double>
class Bar
{
    //...
};

...
int main()
{
    Bar<> bar; // use all default type arguments
}
```

Специализация шаблонов

В некоторых случаях невозможно или нежелательно, чтобы шаблон определял точно такой же код для любого типа. Например, может потребоваться определить путь кода, который будет выполняться, только если аргумент типа является указателем или std::wstring или типом, производным от конкретного базового класса. В таких случаях можно определить *специализацию* шаблона для этого конкретного типа. Когда пользователь создает экземпляр шаблона с этим типом, компилятор использует специализацию для создания класса, а для всех остальных типов компилятор выбирает более общий шаблон. Специализации, в которых все параметры являются специализированными, являются *полными специализациями*. Если только некоторые из параметров являются специализированными, это называется *частичной специализацией*.

```
template <typename K, typename V>
class MyMap{/*...*/};

// partial specialization for string keys
template<typename V>
class MyMap<string, V> {/*...*/};
...

MyMap<int, MyClass> classes; // uses original template
MyMap<string, MyClass> classes2; // uses the partial specialization
```

Шаблон может иметь любое количество специализаций, если каждый специализированный параметр типа уникален. Только шаблоны классов могут быть частично специализированными. Все полные и частичные специализации шаблона должны быть объявлены в том же пространстве имен, что и исходный шаблон.

Дополнительные сведения см. в разделе [специализация шаблонов](#).

typename

12.11.2021 • 2 minutes to read

В определениях шаблонов предоставляет компилятору указание о том, что неизвестный идентификатор является типом. В списках параметров шаблона используется для указания параметра типа.

Синтаксис

```
typename identifier;
```

Remarks

Это ключевое слово следует использовать, если имя в определении шаблона является полным именем, которое зависит от аргумента шаблона. Это необязательно, если полное имя не зависит от.

Дополнительные сведения см. в разделе [шаблоны и разрешение имен](#).

`typename` может использоваться любым типом в любом месте в объявлении или определении шаблона. В списке базовых классов оно не допускается, если не применяется в качестве аргумента шаблона для базового класса шаблона.

```
template <class T>
class C1 : typename T::InnerType // Error - typename not allowed.
{};
template <class T>
class C2 : A<typename T::InnerType> // typename OK.
{};


```

`typename` Ключевое слово также можно использовать вместо `class` в списках параметров шаблона.

Например, следующие операторы семантически эквивалентны:

```
template<class T1, class T2>...
template<typename T1, typename T2>...
```

Пример

```
// typename.cpp
template<class T> class X
{
    typename T::Y m_y; // treat Y as a type
};

int main()
{
}
```

См. также

[Шаблоны](#)

[Ключевые слова](#)

Шаблоны классов

12.11.2021 • 6 minutes to read

В этом разделе описываются правила, относящиеся к шаблонам классов C++.

ФУНКЦИИ ЭЛЕМЕНТОВ В ШАБЛОНАХ КЛАССОВ

Функции-члены могут быть определены как внутри шаблона класса, так и за его пределами. В последнем случае они определяются как шаблоны функций.

```
// member_function_templates1.cpp
template<class T, int i> class MyStack
{
    T* pStack;
    T StackBuffer[i];
    static const int cItems = i * sizeof(T);
public:
    MyStack( void );
    void push( const T item );
    T& pop( void );
};

template< class T, int i > MyStack< T, i >::MyStack( void )
{
};

template< class T, int i > void MyStack< T, i >::push( const T item )
{
};

template< class T, int i > T& MyStack< T, i >::pop( void )
{
};

int main()
{
}
```

Обратите внимание, что как и в функциях-членах класса шаблона, определение функции-члена для конструктора класса подразумевает, что список аргументов шаблона приводится дважды.

Функции-члены сами могут быть шаблонами функций, в которых указываются дополнительные параметры, как показано в следующем примере.

```

// member_templates.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u);
};

template<typename T> template <typename U>
void X<T>::mf(const U &u)
{
}

int main()
{
}

```

Шаблоны вложенных классов

Шаблоны можно определить в классах или шаблонах классов (в этом случае они называются шаблонами членов). Шаблоны членов, которые являются классами, называются шаблонами вложенных классов.

Шаблоны элементов, которые являются функциями, обсуждаются в [шаблонах функций элементов](#).

Шаблоны вложенных классов объявляются как шаблоны классов внутри области внешнего класса. Их можно определить во включающем классе или вне его.

В следующем примере кода демонстрируется шаблон вложенного класса внутри обычного класса.

```

// nested_class_template1.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

class X
{
    template <class T>
    struct Y
    {
        T m_t;
        Y(T t): m_t(t) { }
    };

    Y<int> yInt;
    Y<char> yChar;

public:
    X(int i, char c) : yInt(i), yChar(c) { }
    void print()
    {
        cout << yInt.m_t << " " << yChar.m_t << endl;
    }
};

int main()
{
    X x(1, 'a');
    x.print();
}

```

```
// nested_class_template2.cpp
```

```

// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class X
{
    template <class U> class Y
    {
        U* u;
    public:
        Y();
        U& Value();
        void print();
        ~Y();
    };
    Y<int> y;
public:
    X(T t) { y.Value() = t; }
    void print() { y.print(); }
};

template <class T>
template <class U>
X<T>::Y<U>::Y()
{
    cout << "X<T>::Y<U>::Y()" << endl;
    u = new U();
}

template <class T>
template <class U>
U& X<T>::Y<U>::Value()
{
    return *u;
}

template <class T>
template <class U>
void X<T>::Y<U>::print()
{
    cout << this->Value() << endl;
}

template <class T>
template <class U>
X<T>::Y<U>::~Y()
{
    cout << "X<T>::Y<U>::~Y()" << endl;
    delete u;
}

int main()
{
    X<int>* xi = new X<int>(10);
    X<char>* xc = new X<char>('c');
    xi->print();
    xc->print();
    delete xi;
    delete xc;
}

//Output:
X<T>::Y<U>::Y()
X<T>::Y<U>::Y()
10
99
X<T>::Y<U>::~Y()

```

Локальные классы не могут иметь шаблоны элементов.

Друзья в шаблоне

Шаблоны классов могут иметь [друзей](#). Дружественными объектами класса-шаблона могут быть классы или шаблоны классов, функции или шаблоны функций. Ими также могут быть специализации (кроме частичных) шаблонов классов или шаблонов функций.

В следующем примере дружественная функция определена как шаблон функции в шаблоне класса. Этот код создает по одной версии дружественной функции для каждого экземпляра шаблона. Такую конструкцию можно использовать в тех ситуациях, когда дружественная функция зависит от тех же параметров шаблона, что и класс.

```
// template_friend1.cpp
// compile with: /EHsc

#include <iostream>
using namespace std;

template <class T> class Array {
    T* array;
    int size;

public:
    Array(int sz): size(sz) {
        array = new T[size];
        memset(array, 0, size * sizeof(T));
    }

    Array(const Array& a) {
        size = a.size;
        array = new T[size];
        memcpy_s(array, a.array, sizeof(T));
    }

    T& operator[](int i) {
        return *(array + i);
    }

    int Length() { return size; }

    void print() {
        for (int i = 0; i < size; i++)
            cout << *(array + i) << " ";
        cout << endl;
    }
};

template<class T>
friend Array<T>* combine(Array<T>& a1, Array<T>& a2);
};

template<class T>
Array<T>* combine(Array<T>& a1, Array<T>& a2) {
    Array<T>* a = new Array<T>(a1.size + a2.size);
    for (int i = 0; i < a1.size; i++)
        (*a)[i] = *(a1.array + i);

    for (int i = 0; i < a2.size; i++)
        (*a)[i + a1.size] = *(a2.array + i);

    return a;
}
```

```

}

int main() {
    Array<char> alpha1(26);
    for (int i = 0 ; i < alpha1.Length() ; i++)
        alpha1[i] = 'A' + i;

    alpha1.print();

    Array<char> alpha2(26);
    for (int i = 0 ; i < alpha2.Length() ; i++)
        alpha2[i] = 'a' + i;

    alpha2.print();
    Array<char*>*alpha3 = combine(alpha1, alpha2);
    alpha3->print();
    delete alpha3;
}

//Output:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z a b c d e f g h i j k l m n o p q r s t u v w x y z

```

В следующем примере создается дружественная функция, которая имеет специализацию шаблона. Если исходный шаблон функции является дружественным объектом, то и его специализация автоматически становится дружественной.

Кроме того, как отмечается в комментарии перед объявлением дружественной функции в следующем коде, вы можете взять только специализированную версию шаблона и объявить ее в качестве дружественной. В этом случае определение дружественной специализации шаблона необходимо поместить за пределами класса шаблона.

```

// template_friend2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class Array;

template <class T>
void f(Array<T>& a);

template <class T> class Array
{
    T* array;
    int size;

public:
    Array(int sz): size(sz)
    {
        array = new T[size];
        memset(array, 0, size * sizeof(T));
    }
    Array(const Array& a)
    {
        size = a.size;
        array = new T[size];
        memcpy_s(array, a.array, sizeof(T));
    }
    T& operator[](int i)
    {
        return *(array + i);
    }
    int Length()
    {

```

```

        return size;
    }
    void print()
    {
        for (int i = 0; i < size; i++)
        {
            cout << *(array + i) << " ";
        }
        cout << endl;
    }
    // If you replace the friend declaration with the int-specific
    // version, only the int specialization will be a friend.
    // The code in the generic f will fail
    // with C2248: 'Array<T>::size' :
    // cannot access private member declared in class 'Array<T>'.
    //friend void f<int>(Array<int>& a);

    friend void f<>(Array<T>& a);
};

// f function template, friend of Array<T>
template <class T>
void f(Array<T>& a)
{
    cout << a.size << " generic" << endl;
}

// Specialization of f for int arrays
// will be a friend because the template f is a friend.
template<> void f(Array<int>& a)
{
    cout << a.size << " int" << endl;
}

int main()
{
    Array<char> ac(10);
    f(ac);

    Array<int> a(10);
    f(a);
}
//Output:
10 generic
10 int

```

В следующем примере показан дружественный шаблон класса, объявленный в пределах шаблона класса. Затем этот шаблон класса используется в качестве аргумента шаблона для дружественного класса. Дружественные шаблоны классов должны определяться за пределами шаблона класса, в котором они объявлены. Все специализации или частичные специализации дружественного шаблона также являются дружественными объектами исходного шаблона класса.

```

// template_friend3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T>
class X
{
private:
    T* data;
    void InitData(int seed) { data = new T(seed); }
public:
    void print() { cout << *data << endl; }
    template <class U> friend class Factory;
};

template <class U>
class Factory
{
public:
    U* GetNewObject(int seed)
    {
        U* pu = new U;
        pu->InitData(seed);
        return pu;
    }
};

int main()
{
    Factory< X<int> > XintFactory;
    X<int>* x1 = XintFactory.GetNewObject(65);
    X<int>* x2 = XintFactory.GetNewObject(97);

    Factory< X<char> > XcharFactory;
    X<char>* x3 = XcharFactory.GetNewObject(65);
    X<char>* x4 = XcharFactory.GetNewObject(97);
    x1->print();
    x2->print();
    x3->print();
    x4->print();
}
//Output:
65
97
A
a

```

Повторное использование параметров шаблона

Параметры шаблона могут повторно использоваться в списке параметров шаблона. Например, приведенный ниже код допустим:

```
// template_specifications2.cpp

class Y
{
};

template<class T, T* pT> class X1
{
};

template<class T1, class T2 = T1> class X2
{
};

Y aY;

X1<Y, &aY> x1;
X2<int> x2;

int main()
{
}
```

См. также раздел

[Шаблоны](#)

Шаблоны функций

12.11.2021 • 2 minutes to read

Шаблоны классов определяют семейство связанных классов, основанных на типе аргументов, переданных классу при создании его экземпляра. Шаблоны функций похожи на шаблоны классов, но определяют семейство функций. С помощью шаблонов функций можно задавать наборы функций, основанных на одном коде, но действующих в разных типах или классах. Следующий шаблон функции меняет местами два элемента.

```
// function_templates1.cpp
template< class T > void MySwap( T& a, T& b ) {
    T c(a);
    a = b;
    b = c;
}
int main() { }
```

Этот код определяет семейство функций, которые меняют местами значения аргументов. Из этого шаблона можно создавать функции, которые будут подключаться `int` и `long` типами, а также определяемыми пользователем типами. Функция `MySwap` даже меняет местами классы, если правильно определены конструктор копии и оператор присваивания.

Кроме того, шаблон функции будет препятствовать обмену объектами разных типов, так как компилятор знает типы параметров `a` и `b` во время компиляции.

Хотя эта функция может выполняться нешаблонной функцией с использованием указателей `void`, версия шаблона типобезопасна. Рассмотрим следующие вызовы:

```
int j = 10;
int k = 18;
CString Hello = "Hello, Windows!";
MySwap( j, k );           //OK
MySwap( j, Hello );       //error
```

Второй вызов `MySwap` приводит к ошибке времени компиляции, поскольку компилятору не удается создать функцию `MySwap` с параметрами разных типов. Если бы использовались указатели `void`, оба вызова функции скомпилировались бы правильно, но функция не работала бы должным образом во время выполнения.

Для шаблона функции можно явно задавать аргументы. Пример:

```
// function_templates2.cpp
template<class T> void f(T) {}
int main(int j) {
    f<char>(j);   // Generate the specialization f(char).
    // If not explicitly specified, f(int) would be deduced.
}
```

Если аргумент шаблона задан явно, выполняются обычные неявные преобразования для преобразования аргумента функции в тип соответствующих параметров шаблона функции. В приведенном выше примере компилятор преобразуется `j` в тип `char`.

См. также

[Шаблоны](#)

[Создание экземпляра шаблона функции](#)

[Явное создание экземпляра](#)

[Явная специализация шаблонов функций](#)

Создание экземпляра шаблона функции

12.11.2021 • 2 minutes to read

При первом вызове шаблона функции для каждого типа компилятор создает экземпляр. Каждый экземпляр представляет собой шаблонную функцию, специализированную для данного типа. Этот экземпляр будет вызываться каждый раз, когда функция используется для данного типа. Если имеется несколько одинаковых экземпляров, даже в различных модулях, в исполняемый файл будет помещен только один экземпляр.

В шаблонах функций преобразование аргументов функции разрешается для любых пар аргументов и параметров, в которых параметр не зависит от аргумента шаблона.

Можно явно создавать экземпляры шаблонов функций, объявляя шаблон с определенным типом в качестве аргумента. Например, приведенный ниже код допустим:

```
// function_template_instantiation.cpp
template<class T> void f(T) { }

// Instantiate f with the explicitly specified template.
// argument 'int'
//
template void f<int> (int);

// Instantiate f with the deduced template argument 'char'.
template void f(char);
int main()
{
}
```

См. также

[Шаблоны функций](#)

Явное создание экземпляра

12.11.2021 • 2 minutes to read

Явное создание экземпляров можно использовать для создания шаблонного класса или функции, чтобы обойтись без их использования в коде. Поскольку это полезно для создания файлов библиотек (.lib), в которых шаблоны используются для распространения, то определения шаблонов без создания экземпляров не включаются в объектные файлы (.obj).

Этот код явно создает экземпляры `MyStack` для `int` переменных и шести элементов:

```
template class MyStack<int, 6>;
```

Этот оператор создает экземпляр класса `MyStack`, но не резервирует пространство в памяти для полученного объекта. Код создается для всех членов.

В следующей строке явное создание экземпляра выполняется только для функции-члена конструктора.

```
template MyStack<int, 6>::MyStack( void );
```

Можно явно создать экземпляры шаблонов функций, используя конкретный аргумент типа, чтобы повторно объявить их, как показано в примере в [создании экземпляра шаблона функции](#).

`extern` Ключевое слово можно использовать для предотвращения автоматического создания экземпляров членов. Пример:

```
extern template class MyStack<int, 6>;
```

Аналогично, определенные элементы можно пометить как внешние, для которых не был создан экземпляр.

```
extern template MyStack<int, 6>::MyStack( void );
```

Можно использовать `extern` ключевое слово, чтобы компилятор создавал один и тот же код создания в нескольких модулях объектов. Если функция вызывается, то экземпляр шаблонной функции должен быть создан с использованием заданных явных параметров шаблона по меньшей мере в одном связанном модуле; в противном случае при сборке программы возникнет ошибка компоновщика.

NOTE

`extern` Ключевое слово в специализации применяется только к функциям элементов, определенным за пределами тела класса. Функции, определенные внутри объявления класса, рассматриваются как встраиваемые; для них всегда создаются экземпляры.

См. также раздел

[Шаблоны функций](#)

Явная специализация шаблонов функций

12.11.2021 • 2 minutes to read

Используя шаблон функции, можно указать особое поведение для определенного типа, предоставив явную специализацию (переопределение) шаблона функции для этого типа. Пример:

```
template<> void MySwap(double a, double b);
```

Это объявление позволяет определить другую функцию для `double` переменных. Как и в случае с функциями, не являющимися шаблонами, применяются преобразования стандартных типов (например, повышение переменной типа `float` до `double`).

Пример

```
// explicit_specialization.cpp
template<class T> void f(T t)
{
};

// Explicit specialization of f with 'char' with the
// template argument explicitly specified:
//
template<> void f<char>(char c)
{
};

// Explicit specialization of f with 'double' with the
// template argument deduced:
//
template<> void f(double d)
{
};
int main()
{}
```

См. также раздел

[Шаблоны функций](#)

Частичное упорядочение шаблонов функций (C++)

12.11.2021 • 2 minutes to read

Может быть доступно несколько шаблонов функций, соответствующих списку аргументов в вызове функции. В C++ определено частичное упорядочение шаблонов функции, что позволяет указать, какую функцию необходимо вызвать. Упорядочение выполняется частично, поскольку может быть несколько шаблонов, которые считаются в равной мере специализированными.

Компилятор выбирает функцию наиболее специализированного шаблона среди всех возможных совпадений. Например, если шаблон функции принимает тип `t` и существует другой шаблон функции, который принимает `t*` доступ, то `t*` версия считается более специализированной. Он предпочтительнее для универсальной `t` версии, когда аргумент является типом указателя, хотя оба могут быть допустимыми.

Чтобы определить, является ли один кандидатов — шаблонов функций — более специализированным, используйте следующую процедуру:

1. Рассмотрим два шаблона функции, `T1` и `T2`.
2. Замените параметры в шаблоне `T1` на гипотетический уникальный тип `X`.
3. Пользуясь списком параметров из шаблона `T1`, проверьте, является ли `T2` допустимым шаблоном для этого списка параметров. Любые неявные преобразования пропускайте.
4. Повторите ту же самую процедуру, поменяв шаблоны `T1` и `T2` местами.
5. Если один шаблон является допустимым списком аргументов шаблона для другого шаблона, но наоборот не имеет значения, этот шаблон считается менее специализированным, чем другой шаблон. Если с помощью предыдущего шага оба шаблона формируют допустимые аргументы для друг друга, они считаются одинаково специализированными, и при попытке их использования происходит неоднозначный вызов.
6. При проведении процедуры руководствуйтесь следующими правилами:
 - a. Шаблон для определенного типа является более специализированным, чем шаблон, принимающий аргумент универсального типа.
 - b. Шаблон только `t*` более специализирован, чем один `t`, поскольку гипотетический тип `x*` является допустимым аргументом для `t` аргумента шаблона, но не `x` является допустимым аргументом для `t*` аргумента шаблона.
 - c. `const T` является более специализированным `t`, чем, поскольку `const X` является допустимым аргументом для `t` аргумента шаблона, но не `x` является допустимым аргументом для `const T` аргумента шаблона.
 - d. `const T*` является более специализированным `t*`, чем, поскольку `const X*` является допустимым аргументом для `t*` аргумента шаблона, но не `x*` является допустимым аргументом для `const T*` аргумента шаблона.

Пример

Следующий пример работает, как указано в стандарте:

```
// partial_ordering_of_function_templates.cpp
// compile with: /EHsc
#include <iostream>

template <class T> void f(T) {
    printf_s("Less specialized function called\n");
}

template <class T> void f(T*) {
    printf_s("More specialized function called\n");
}

template <class T> void f(const T*) {
    printf_s("Even more specialized function for const T*\n");
}

int main() {
    int i = 0;
    const int j = 0;
    int *pi = &i;
    const int *cpi = &j;

    f(i); // Calls less specialized function.
    f(pi); // Calls more specialized function.
    f(cpi); // Calls even more specialized function.
    // Without partial ordering, these calls would be ambiguous.
}
```

Выход

```
Less specialized function called
More specialized function called
Even more specialized function for const T*
```

См. также

[Шаблоны функций](#)

Шаблоны функций-членов

12.11.2021 • 2 minutes to read

Шаблон элементов терминов относится как к шаблонам функций-членов, так и к шаблонам вложенных классов. Шаблоны функций-членов — это функции-шаблоны, являющиеся членами класса или шаблона класса.

Функции-члены могут являться функциями-шаблонами в нескольких контекстах. Все функции шаблонов классов являются общими, однако они не являются шаблонами элементов или шаблонами функций-членов. Если такие функции-члены принимают собственные аргументы шаблона, они считаются шаблонами функций-членов.

Пример. объявление шаблонов функций элементов

Шаблоны функции-члена нешаблонных или шаблонных классов объявляются в виде шаблонов функций с собственными шаблонными параметрами.

```
// member_function_templates.cpp
struct X
{
    template <class T> void mf(T* t) {}
};

int main()
{
    int i;
    X* x = new X();
    x->mf(&i);
}
```

Пример: шаблон функции члена класса шаблона

В следующем примере показан шаблон функции-члена шаблонного класса.

```
// member_function_templates2.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u)
    {
    }
};

int main()
```

Пример: Определение шаблонов элементов за пределами класса

```
// defining_member_templates_outside_class.cpp
template<typename T>
class X
{
public:
    template<typename U>
    void mf(const U &u);
};

template<typename T> template <typename U>
void X<T>::mf(const U &u)
{
}

int main()
{
```

Пример. шаблонное определяемое пользователем преобразование

Локальные классы не могут иметь шаблоны элементов.

Функции шаблонов-элементов не могут быть виртуальными функциями или переопределять виртуальные функции из базового класса, если они объявлены с тем же именем, что и виртуальная функция базового класса.

В следующем примере показано определяемое пользователем преобразование с помощью шаблона:

```
// templated_user_defined_conversions.cpp
template <class T>
struct S
{
    template <class U> operator S<U>()
    {
        return S<U>();
    }
};

int main()
{
    S<int> s1;
    S<long> s2 = s1; // Convert s1 using UDC and copy constructs S<long>.
}
```

См. также

[Шаблоны функций](#)

Специализация шаблона (C++)

12.11.2021 • 5 minutes to read

Шаблоны класса можно частично специализировать, при этом получившийся класс по-прежнему будет шаблоном. Частичная специализация позволяет частично настроить код шаблона для определенных типов, например:

- Шаблон имеет несколько типов, и только некоторые из них требуют специализации. Результат для остальных типов параметризован шаблоном.
- Шаблон имеет только один тип, но специализация необходима для типов указателя, ссылки, указателя на член или указателя на функцию. Специализация сама по себе по-прежнему является шаблоном с типом, на который задан указатель или ссылка.

Пример: частичная специализация шаблонов классов

```
// partial_specialization_of_class_templates.cpp
#include <stdio.h>

template <class T> struct PTS {
    enum {
        IsPointer = 0,
        IsPointerToDataMember = 0
    };
};

template <class T> struct PTS<T*> {
    enum {
        IsPointer = 1,
        IsPointerToDataMember = 0
    };
};

template <class T, class U> struct PTS<T U::*> {
    enum {
        IsPointer = 0,
        IsPointerToDataMember = 1
    };
};

struct S{};

int main() {
    printf_s("PTS<S>::IsPointer == %d \nPTS<S>::IsPointerToDataMember == %d\n",
            PTS<S>::IsPointer, PTS<S>:: IsPointerToDataMember);
    printf_s("PTS<S*>::IsPointer == %d \nPTS<S*>::IsPointerToDataMember == %d\n",
            , PTS<S*>::IsPointer, PTS<S*>:: IsPointerToDataMember);
    printf_s("PTS<int S::*>::IsPointer == %d \nPTS"
            "<int S::*>::IsPointerToDataMember == %d\n",
            PTS<int S::*>::IsPointer, PTS<int S::*>::
            IsPointerToDataMember);
}
```

```

PTS<S>::IsPointer == 0
PTS<S>::IsPointerToDataMember == 0
PTS<S*>::IsPointer == 1
PTS<S*>::IsPointerToDataMember == 0
PTS<int S::*>::IsPointer == 0
PTS<int S::*>::IsPointerToDataMember == 1

```

Пример: частичная специализация для типов указателей

Если имеется класс коллекции шаблонов, принимающий любой тип `T`, можно создать частичную специализацию, которая принимает любой тип указателя `T*`. В следующем коде показан шаблон класса коллекции `Bag` и частичная специализация для типов указателей, в которой коллекция разыменовывает типы указателей перед их копированием в массив. Затем коллекция сохраняет значения, на которые указывают указатели. В исходном шаблоне в коллекции сохранялись бы только сами указатели, а данные оставались бы уязвимы в отношении удаления или изменения. В этой версии коллекции, специально предназначеннной для указателей, добавлен код для проверки наличия пустых указателей `null` в методе `add`.

```

// partial_specialization_of_class_templates2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

// Original template collection class.
template <class T> class Bag {
    T* elem;
    int size;
    int max_size;

public:
    Bag() : elem(0), size(0), max_size(1) {}
    void add(T t) {
        T* tmp;
        if (size + 1 >= max_size) {
            max_size *= 2;
            tmp = new T [max_size];
            for (int i = 0; i < size; i++)
                tmp[i] = elem[i];
            tmp[size++] = t;
            delete[] elem;
            elem = tmp;
        }
        else
            elem[size++] = t;
    }

    void print() {
        for (int i = 0; i < size; i++)
            cout << elem[i] << " ";
        cout << endl;
    }
};

// Template partial specialization for pointer types.
// The collection has been modified to check for NULL
// and store types pointed to.
template <class T> class Bag<T*> {
    T* elem;
    int size;
    int max_size;

public:
    Bag() : elem(0), size(0), max_size(1) {}

```

```

bag() : elem(0), size(0), max_size(1) {}

void add(T* t) {
    T* tmp;
    if (t == NULL) { // Check for NULL
        cout << "Null pointer!" << endl;
        return;
    }

    if (size + 1 >= max_size) {
        max_size *= 2;
        tmp = new T [max_size];
        for (int i = 0; i < size; i++)
            tmp[i] = elem[i];
        tmp[size++] = *t; // Dereference
        delete[] elem;
        elem = tmp;
    }
    else
        elem[size++] = *t; // Dereference
}

void print() {
    for (int i = 0; i < size; i++)
        cout << elem[i] << " ";
    cout << endl;
}

};

int main() {
    Bag<int> xi;
    Bag<char> xc;
    Bag<int*> xp; // Uses partial specialization for pointer types.

    xi.add(10);
    xi.add(9);
    xi.add(8);
    xi.print();

    xc.add('a');
    xc.add('b');
    xc.add('c');
    xc.print();

    int i = 3, j = 87, *p = new int[2];
    *p = 8;
    *(p + 1) = 100;
    xp.add(&i);
    xp.add(&j);
    xp.add(p);
    xp.add(p + 1);
    delete[] p;
    p = NULL;
    xp.add(p);
    xp.print();
}

```

```

10 9 8
a b c
Null pointer!
3 87 8 100

```

Пример. Определение частичной специализации для одного типа
int

В следующем примере определяется класс шаблона, который принимает пары из двух типов, а затем определяет частичную специализацию этого класса шаблона, чтобы один из типов был `int`. Специализация определяет дополнительный метод сортировки, который реализует простую сортировку пузырьковым методом на основе целого числа.

```
// partial_specialization_of_class_templates3.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class Key, class Value> class Dictionary {
    Key* keys;
    Value* values;
    int size;
    int max_size;
public:
    Dictionary(int initial_size) : size(0) {
        max_size = 1;
        while (initial_size >= max_size)
            max_size *= 2;
        keys = new Key[max_size];
        values = new Value[max_size];
    }
    void add(Key key, Value value) {
        Key* tmpKey;
        Value* tmpVal;
        if (size + 1 >= max_size) {
            max_size *= 2;
            tmpKey = new Key [max_size];
            tmpVal = new Value [max_size];
            for (int i = 0; i < size; i++) {
                tmpKey[i] = keys[i];
                tmpVal[i] = values[i];
            }
            tmpKey[size] = key;
            tmpVal[size] = value;
            delete[] keys;
            delete[] values;
            keys = tmpKey;
            values = tmpVal;
        }
        else {
            keys[size] = key;
            values[size] = value;
        }
        size++;
    }
    void print() {
        for (int i = 0; i < size; i++)
            cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
    }
};

// Template partial specialization: Key is specified to be int.
template <class Value> class Dictionary<int, Value> {
    int* keys;
    Value* values;
    int size;
    int max_size;
public:
    Dictionary(int initial_size) : size(0) {
        max_size = 1;
        while (initial_size >= max_size)
            max_size *= 2;
        keys = new int[max_size];
        values = new Value[max_size];
    }
};
```

```

    }

    void add(int key, Value value) {
        int* tmpKey;
        Value* tmpVal;
        if (size + 1 >= max_size) {
            max_size *= 2;
            tmpKey = new int [max_size];
            tmpVal = new Value [max_size];
            for (int i = 0; i < size; i++) {
                tmpKey[i] = keys[i];
                tmpVal[i] = values[i];
            }
            tmpKey[size] = key;
            tmpVal[size] = value;
            delete[] keys;
            delete[] values;
            keys = tmpKey;
            values = tmpVal;
        }
        else {
            keys[size] = key;
            values[size] = value;
        }
        size++;
    }

    void sort() {
        // Sort method is defined.
        int smallest = 0;
        for (int i = 0; i < size - 1; i++) {
            for (int j = i; j < size; j++) {
                if (keys[j] < keys[smallest])
                    smallest = j;
            }
            swap(keys[i], keys[smallest]);
            swap(values[i], values[smallest]);
        }
    }

    void print() {
        for (int i = 0; i < size; i++)
            cout << "{" << keys[i] << ", " << values[i] << "}" << endl;
    }
};

int main() {
    Dictionary<const char*, const char*> dict(10);
    dict.print();
    dict.add("apple", "fruit");
    dict.add("banana", "fruit");
    dict.add("dog", "animal");
    dict.print();

    Dictionary<int, const char*> dict_specialized(10);
    dict_specialized.print();
    dict_specialized.add(100, "apple");
    dict_specialized.add(101, "banana");
    dict_specialized.add(103, "dog");
    dict_specialized.add(89, "cat");
    dict_specialized.print();
    dict_specialized.sort();
    cout << endl << "Sorted list:" << endl;
    dict_specialized.print();
}

```

```
{apple, fruit}  
{banana, fruit}  
{dog, animal}  
{100, apple}  
{101, banana}  
{103, dog}  
{89, cat}
```

Sorted list:

```
{89, cat}  
{100, apple}  
{101, banana}  
{103, dog}
```

Шаблоны и разрешение имен

12.11.2021 • 2 minutes to read

В определениях шаблонов существует три типа имен:

- локально объявленные имена, включая имя самого шаблона, и все имена, объявленные в рамках определения шаблона;
- имена из внешней области видимости вне определения шаблона;
- имена, зависящие каким-либо образом от аргументов шаблона (называемые зависимыми именами).

Хотя первые два типа имен также относятся к областям видимости класса и функции, в определениях шаблонов необходимо иметь особые правила разрешения имен для устранения проблем, связанных с дополнительной сложностью зависимых имен. Причина этого в том, что до создания экземпляра шаблона компилятор мало знает об этих именах, поскольку они могут быть совершенно разных типов в зависимости от используемых аргументов шаблона. Поиск независимых имен осуществляется в соответствии с обычными правилами и в точке определения шаблона. Эти имена, не зависящие от аргументов шаблона, ищутся один раз для всех специализаций шаблона. Поиск зависимых имен не выполняется, пока не будет создан экземпляр шаблона, и такие имена ищутся отдельно для каждой специализации.

Тип является зависимым, если он зависит от аргументов шаблона. В частности, он зависим, если представляет собой:

- аргумент шаблона:

```
T
```

- полное имя с квалификацией, включающей зависимый тип:

```
T::myType
```

- полное имя, если неопределенная часть идентифицирует зависимый тип:

```
N::T
```

- тип const или volatile, для которого базовый тип является зависимым:

```
const T
```

- тип указателя, ссылки, массива или указателя функции в зависимости от зависимого типа:

```
T *, T &, T [10], T (*)()
```

- массив, размер которого зависит от параметра шаблона:

```
template <int arg> class X {  
    int x[arg] ; // dependent type  
}
```

- тип шаблона, созданного из параметра шаблона:

```
T<int>, MyTemplate<T>
```

Зависимость от типа и зависимость от значения

Имена и выражения, зависящие от параметра шаблона, классифицируются как зависящие от типа или значения в зависимости от того, является ли параметр шаблона параметром типа или параметром значения. Кроме того, все идентификаторы, объявленные в шаблоне с типом, зависящим от аргумента шаблона, считаются зависимыми от значения, так как имеют целочисленный тип или тип перечисления, инициализированный выражением, зависящим от значения.

Выражения, зависящие от типа, и выражения, зависящие от значения, — это выражения, содержащие переменные, зависящие от типа или значения. Эти выражения могут иметь семантику, зависящую от параметров, используемых для шаблона.

См. также раздел

[Шаблоны](#)

Разрешение имен зависимых типов

12.11.2021 • 2 minutes to read

Используйте `typename` для уточненных имен в определениях шаблонов, чтобы сообщить компилятору, что заданное полное имя определяет тип. Дополнительные сведения см. в разделе [TypeName](#).

```
// template_name_resolution1.cpp
#include <stdio.h>
template <class T> class X
{
public:
    void f(typename T::myType* mt) {}
};

class Yarg
{
public:
    struct myType { };
};

int main()
{
    X<Yarg> x;
    x.f(new Yarg::myType());
    printf("Name resolved by using typename keyword.");
}
```

```
Name resolved by using typename keyword.
```

Поиск имен для зависимых имен проверяет имена из контекста определения шаблона — в следующем примере этот контекст будет найден `myFunction(char)` — и контекст создания экземпляра шаблона. В следующем примере создается экземпляр шаблона в Main. Таким образом, объект отображается `MyNamespace::myFunction` с точки зрения создания экземпляра и выбирается в качестве лучшего соответствия. В случае переименования функции `MyNamespace::myFunction` вместо нее будет вызываться функция `myFunction(char)`.

Разрешение имен выполняется так, как если бы они были зависимыми. Однако при наличии любой возможности конфликта рекомендуется использовать полные имена.

```

// template_name_resolution2.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void myFunction(char)
{
    cout << "Char myFunction" << endl;
}

template <class T> class Class1
{
public:
    Class1(T i)
    {
        // If replaced with myFunction(1), myFunction(char)
        // will be called
        myFunction(i);
    }
};

namespace MyNamespace
{
    void myFunction(int)
    {
        cout << "Int MyNamespace::myFunction" << endl;
    }
};

using namespace MyNamespace;

int main()
{
    Class1<int>* c1 = new Class1<int>(100);
}

```

Выходные данные

```
Int MyNamespace::myFunction
```

Устранение неоднозначности шаблона

Visual Studio 2012 применяет стандартные правила C++ 98/03/11 для устранения неоднозначности с помощью ключевого слова "template". В следующем примере Visual Studio 2010 принимает и несоответствующие строки, и соответствующие строки. Visual Studio 2012 принимает только строки соответствия.

```
#include <iostream>
#include <ostream>
#include <typeinfo>
using namespace std;

template <typename T> struct Allocator {
    template <typename U> struct Rebind {
        typedef Allocator<U> Other;
    };
};

template <typename X, typename AY> struct Container {
    #if defined(NONCONFORMANT)
        typedef typename AY::Rebind<X>::Other AX; // nonconformant
    #elif defined(CONFORMANT)
        typedef typename AY::template Rebind<X>::Other AX; // conformant
    #else
        #error Define NONCONFORMANT or CONFORMANT.
    #endif
};

int main() {
    cout << typeid(Container<int, Allocator<float>>::AX).name() << endl;
}
```

Соответствие правилам устранения неоднозначности необходимо соблюдать, поскольку по умолчанию C++ предполагает, что `AY::Rebind` не является шаблоном, в результате чего компилятор интерпретирует знак "`<`" как "меньше чем". Чтобы компилятор мог правильно интерпретировать знак "`Rebind`" как угловую скобку, он должен знать, что `<` — это шаблон.

См. также раздел

[Разрешение имен](#)

Разрешение локально объявленных имен

12.11.2021 • 3 minutes to read

Можно создать ссылку на само имя шаблона с аргументами шаблона или без них. В области шаблона класса само имя относится к шаблону. В области специализации шаблона или частичной специализации само имя относится к специализации или частичной специализации. Можно также создать ссылки на другие специализации или частичные специализации шаблона с использованием соответствующих аргументов шаблона.

Пример: специализация и частичная специализация

Следующий фрагмент кода показывает, что имя A шаблона классов интерпретируется по-разному в области видимости специализации или частичной специализации.

```
// template_name_resolution3.cpp
// compile with: /c
template <class T> class A {
    A* a1;    // A refers to A<T>
    A<int>* a2; // A<int> refers to a specialization of A.
    A<T*>* a3; // A<T*> refers to the partial specialization A<T*>.
};

template <class T> class A<T*> {
    A* a4; // A refers to A<T*>.
};

template<> class A<int> {
    A* a5; // A refers to A<int>.
};
```

Пример: конфликт имен между параметром шаблона и объектом

В случае конфликта имен между параметром шаблона и другим объектом параметр шаблона может быть скрыт, но это не обязательно. Следующие правила помогут определить приоритет.

Параметр шаблона находится в области видимости из точки, где он впервые появляется, и до конца класса или шаблона функции. Если имя снова отображается в списке аргументов шаблона или в списке базовых классов, оно относится к тому же типу. В стандартном языке C++ невозможно объявить в той же области видимости никакое другое имя, идентичное параметру шаблона. Расширение Microsoft позволяет переопределять параметр шаблона в области видимости шаблона. В следующем примере показано использование параметра шаблона в базовой спецификации шаблона класса.

```
// template_name_resolution4.cpp
// compile with: /EHsc
template <class T>
class Base1 {};

template <class T>
class Derived1 : Base1<T> {};

int main() {
    // Derived1<int> d;
}
```

Пример: определение функции элемента вне шаблона класса

При определении функций-членов шаблона за пределами шаблона классов можно использовать другое имя параметра шаблонов. Если в определении функции-члена шаблона используется другое имя параметра шаблона, нежели в декларации, а имя, используемое в определении, конфликтует с другим элементом объявления, приоритет получает элемент в объявлении шаблона.

```
// template_name_resolution5.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

template <class T> class C {
public:
    struct Z {
        Z() { cout << "Z::Z()" << endl; }
    };
    void f();
};

template <class Z>
void C<Z>::f() {
    // Z refers to the struct Z, not to the template arg;
    // Therefore, the constructor for struct Z will be called.
    Z z;
}

int main() {
    C<int> c;
    c.f();
}
```

```
Z::Z()
```

Пример: определение шаблона или функции элемента вне пространства имен

При определении функции шаблона или функции-члена за пределами пространства имен, в котором объявлен шаблон, аргумент шаблона имеет приоритет над именами других элементов пространства имен.

```

// template_name_resolution6.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

namespace NS {
    void g() { cout << "NS::g" << endl; }

    template <class T> struct C {
        void f();
        void g() { cout << "C<T>::g" << endl; }
    };
}

template <class T>
void NS::C<T>::f() {
    g(); // C<T>::g, not NS::g
}

int main() {
    NS::C<int> c;
    c.f();
}

```

C<T>::g

Пример: имя базового класса или члена скрывает аргумент шаблона

Если класс шаблона в определениях вне объявления класса шаблонов имеет базовый класс, который не зависит от аргумента шаблона и если базовый класс или один из его элементов имеют то же имя, что и аргумент шаблона, имя базового класса или элемента скрывает аргумент шаблона.

```

// template_name_resolution7.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

struct B {
    int i;
    void print() { cout << "Base" << endl; }
};

template <class T, int i> struct C : public B {
    void f();
};

template <class B, int i>
void C<B, i>::f() {
    B b;    // Base class b, not template argument.
    b.print();
    i = 1; // Set base class's i to 1.
}

int main() {
    C<int, 1> c;
    c.f();
    cout << c.i << endl;
}

```

Base

1

См. также:

[Разрешение имен](#)

Разрешение перегрузки вызовов шаблонов функций

12.11.2021 • 2 minutes to read

Шаблон функции может перегрузить нешаблонные функции с одинаковым именем. В этом сценарии вызовы функций разрешаются с помощью вычета аргумента шаблона для создания экземпляра шаблона функции с уникальной специализацией. Если вычет аргумента шаблона завершается с ошибкой, считается, что другие перезагрузки функций разрешат вызов. Эти другие перезагрузки, также известные как набор кандидатов, включают нешаблонные функции и другие экземпляры шаблонов функций. Если вычет аргумента шаблона завершается успешно, то созданная функция сравнивается с другими функциями для определения оптимального совпадения с учетом правил разрешения перегрузок.

Дополнительные сведения см. в разделе [перегрузка функций](#).

Пример. Выбор нешаблонной функции

Если нешаблонная функция хорошо соответствует функции шаблона, используется нешаблонная функция (если аргументы шаблона не заданы явно), как в вызове `f(1, 1)` в следующем примере.

```
// template_name_resolution9.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void f(int, int) { cout << "f(int, int)" << endl; }
void f(char, char) { cout << "f(char, char)" << endl; }

template <class T1, class T2>
void f(T1, T2)
{
    cout << "void f(T1, T2)" << endl;
};

int main()
{
    f(1, 1);    // Equally good match; choose the nontemplate function.
    f('a', 1); // Chooses the template function.
    f<int, int>(2, 2); // Template arguments explicitly specified.
}
```

```
f(int, int)
void f(T1, T2)
void f(T1, T2)
```

Пример: предпочтительная функция шаблона точного соответствия

В следующем примере показано, что точное соответствие функции шаблона предпочтительнее, если требуется преобразование нешаблонной функции.

```
// template_name_resolution10.cpp
// compile with: /EHsc
#include <iostream>
using namespace std;

void f(int, int) { cout << "f(int, int)" << endl; }

template <class T1, class T2>
void f(T1, T2)
{
    cout << "void f(T1, T2)" << endl;
}

int main()
{
    long l = 0;
    int i = 0;
    // Call the template function f(long, int) because f(int, int)
    // would require a conversion from long to int.
    f(l, i);
}
```

```
void f(T1, T2)
```

См. также раздел

[Разрешение имен
имени](#)

Организация исходного кода (шаблоны C++)

12.11.2021 • 3 minutes to read

При определении шаблона класса необходимо организовать исходный код таким образом, чтобы определения элементов были видны компилятору, когда они ему нужны. Вы можете выбрать *модель включения* или модель *явного создания экземпляра*. В модели включения определения элементов включаются в каждый файл, использующий шаблон. Это самый простой подход. Он обеспечивает максимальную гибкость с точки зрения того, какие конкретные типы могут использоваться в шаблоне. Его недостаток в том, что он может увеличивать время компиляции. Воздействие может быть значительным, если проект или сами включенные файлы имеют большой размер. В рамках подхода явного создания экземпляров сам шаблон создает экземпляры конкретных классов или элементов класса для определенных типов. Этот подход может уменьшить время компиляции, но он ограничивает использование только теми классами, которые разработчик шаблона включил заранее. Как правило, модель включения рекомендуется использовать, если время компиляции не является проблемой.

История

Шаблоны не похожи на обычные классы в том смысле, что компилятор не создает объектный код для шаблона или любого из его элементов. Ничего не создается, пока экземпляр шаблона не будет создан с конкретными типами. Когда компилятор обнаруживает создание экземпляра шаблона, такого как `MyClass<int> mc;`, и класс с такой сигнатурой еще не существует, он создает такой класс. Он также пытается создать код для любых используемых функций-элементов. Если эти определения находятся в файле, который прямо или косвенно не включен (`#include`) в CPP-файл, который компилируется, компилятор не сможет их увидеть. С точки зрения компилятора, это не обязательно ошибка, потому что функции могут быть определены в другой записи преобразования, и в этом случае компоновщик найдет их. Если компоновщик не находит этот код, он создает *неразрешенную внешнюю ошибку*.

Модель включения

Самый простой и наиболее распространенный способ сделать определения шаблонов видимыми во всей записи преобразования — это поместить определения в сам файл заголовка. Любой CPP-файл, который использует шаблон, должен включать в себя (`#include`) заголовок. Этот подход используется в стандартной библиотеке.

```

#ifndef MYARRAY
#define MYARRAY
#include <iostream>

template<typename T, size_t N>
class MyArray
{
    T arr[N];
public:
    // Full definitions:
    MyArray(){}
    void Print()
    {
        for (const auto v : arr)
        {
            std::cout << v << " , ";
        }
    }

    T& operator[](int i)
    {
        return arr[i];
    }
};

#endif

```

При таком подходе компилятор имеет доступ к полному определению шаблона и может создавать экземпляры шаблонов по запросу для любого типа. Он прост и достаточно легкий в плане обслуживания. Тем не менее модель включения затратнее с точки зрения времени компиляции. Эти затраты могут возрасти в крупных программах, особенно если сам заголовок шаблона содержит (#include) другие заголовки. Каждый `.cpp` файл, #includes заголовок, получает собственную копию шаблонов функций и всех определений. Обычно компоновщик сможет разобраться в ситуации, чтобы у вас не было нескольких определений для функции, но для этого потребуется время. В случае небольших программ дополнительное время компиляции будет существенно меньше.

Модель явного создания экземпляра

Если модель включения не подходит для проекта и вы точно понимаете набор типов, которые будут использоваться для создания экземпляра шаблона, можно разделить код шаблона на `.h` файл и `.cpp`. В `.cpp` файле явным образом создать экземпляры шаблонов. Это приведет к созданию объектного кода, который компилятор распознает при обработке пользовательских экземпляров.

Чтобы явно создать экземпляр, используйте шаблон ключевого слова, за которым следует сигнатура сущности, которую вы хотите создать. Это может быть тип или элемент. Если вы явно создаете экземпляр типа, будут созданы все элементы.

```
template MyArray<double, 5>;
```

```

//MyArray.h
#ifndef MYARRAY
#define MYARRAY

template<typename T, size_t N>
class MyArray
{
    T arr[N];
public:
    MyArray();
    void Print();
    T& operator[](int i);
};

#endif

//MyArray.cpp
#include <iostream>
#include "MyArray.h"

using namespace std;

template<typename T, size_t N>
MyArray<T,N>::MyArray(){}

```

```

template<typename T, size_t N>
void MyArray<T,N>::Print()
{
    for (const auto v : arr)
    {
        cout << v << " ";
    }
    cout << endl;
}

```

```
template MyArray<double, 5>;template MyArray<string, 5>;
```

В предыдущем примере явные экземпляры находятся в нижней части СРР-файла. `MyArray` Может использоваться только для `double` `String` типов или.

NOTE

В C++ 11 `export` ключевое слово было объявлено устаревшим в контексте определений шаблонов. На практике это мало что дает, потому что большинство компиляторов никогда его не поддерживали.

Обработка событий

12.11.2021 • 2 minutes to read

Обработка событий в основном поддерживается для классов COM (классы C++ , которые реализуют COM-объекты, как правило, с помощью классов ATL или атрибута [coclass](#)). Дополнительные сведения см. в разделе [Обработка событий в com](#).

Обработка событий также поддерживается для собственных классов C++ (классы C++ , которые не реализуют COM-объекты). Поддержка обработки событий в машинном коде C++ устарела и будет удалена в следующем выпуске. Дополнительные сведения см. в разделе [Обработка событий в машинном коде C++](#).

NOTE

Атрибуты событий в машинном коде C++ несовместимы со стандартным C++. Они не компилируются при указании `/permissive-` режима соответствия.

Обработка событий поддерживает однопотоковый и однопотковое использование. Он защищает данные от одновременного многопоточного доступа. Подклассы можно наследовать из классов источников или получателей событий. Эти подклассы поддерживают источники и получение расширенных событий.

Компилятор Microsoft C++ включает атрибуты и ключевые слова для объявления событий и обработчиков событий. Атрибуты событий и ключевые слова можно использовать в программах CLR и собственных программах C++ .

СТАТЬЯ	ОПИСАНИЕ
<code>event_source</code>	Создает источник событий.
<code>event_receiver</code>	Создает приемник событий (получатель).
<code>_event</code>	Объявление события.
<code>_raise</code>	Выделяет место вызова события.
<code>_hook</code>	Связывает метод обработчика с событием.
<code>_unhook</code>	Отменяет связь метода обработчика с событием.

См. также раздел

[Справочник по языку C++](#)

[Ключевые слова](#)

`_event` This

12.11.2021 • 3 minutes to read

Объявление события.

NOTE

Атрибуты событий в машинном коде C++ несовместимы со стандартным C++. Они не компилируются при указании `/permissive-` режима соответствия.

Синтаксис

```
_event member-function-declarator ;
_event __interface interface-specifier ;
_event data-member-declarator ;
```

Remarks

Ключевое слово Microsoft `_event` может быть применено к объявлению функции-члена, объявлению интерфейса или объявлению члена данных. Однако нельзя использовать `_event` ключевое слово для уточнения члена вложенного класса.

В зависимости от того, являются ли источник и получатель события неуправляемыми объектами C++, объектами COM или управляемыми объектами (в .NET Framework), в качестве событий можно использовать следующие конструкции:

МАШИННЫЙ C++	COM	УПРАВЛЯЕМЫЙ (.NET FRAMEWORK)
функция-член	-	method
-	interface	-
-	-	элемент данных

Используйте `_hook` в приемнике событий, чтобы связать функцию-член обработчика с функцией-членом события. После создания события с `_event` ключевым словом все обработчики событий, подключенные к этому событию, будут вызываться при вызове события.

`_event` Объявление функции-члена не может иметь определения; определение неявно создается, поэтому функцию-член события можно вызвать так, как если бы она была обычной функцией-членом.

NOTE

Шаблонный класс или структура не может содержать события.

Собственные события

Собственные события — это функции элементов. Тип возвращаемого значения обычно `HRESULT` или

`void`, но может быть любым целочисленным типом, включая `enum`. Если событие использует целочисленный тип возвращаемого значения, условие ошибки определяется, когда обработчик событий возвращает ненулевое значение. В этом случае событие, которое вызывается, вызывает другие делегаты.

```
// Examples of native C++ events:  
__event void OnDoubleClick();  
__event HRESULT OnClick(int* b, char* s);
```

Пример кода см. в разделе [Обработка событий в машинном коде C++](#).

COM-события

События COM являются интерфейсами. Параметры функции-члена в интерфейсе источника события должны быть *в* параметрах, но не строго применены. Это обусловлено тем, что параметр *out* не полезен при многоадресной рассылке. Если используется параметр *out*, выдается предупреждение уровня 1.

Тип возвращаемого значения обычно `HRESULT` или `void`, но может быть любым целочисленным типом, включая `enum`. Если событие использует целочисленный возвращаемый тип, а обработчик событий возвращает ненулевое значение, это может быть условие ошибки. Вызванное событие прерывает вызовы других делегатов. Компилятор автоматически помечает интерфейс источника события как `source` в созданном IDL.

`__interface` Ключевое слово всегда требуется после `__event` для источника события com.

```
// Example of a COM event:  
__event __interface IEvent1;
```

Пример кода см. в разделе [Обработка событий в com](#).

Управляемые события

Дополнительные сведения о создании кода событий в новом синтаксисе см. в разделе [event](#).

Управляемые события — это элементы данных или функции элементов. При использовании с событием тип возвращаемого значения делегата должен соответствовать [спецификации CLS](#). Возвращаемый тип обработчика события должен соответствовать возвращаемому типу делегата. Дополнительные сведения о делегатах см. в разделе [делегаты и события](#). Если управляемое событие является данными-членом, его тип должен быть указателем на делегат.

В платформе .NET Framework данные-член можно рассматривать, как если бы это был сам метод (т. е., метод `Invoke` соответствующего делегата). Для этого необходимо определить тип делегата для объявления члена данных управляемого события. В отличие от этого, метод управляемого события неявно определяет соответствующий управляемый делегат, если он еще не определен. Например, значение события, такое как `OnClick`, можно объявить как событие следующим образом:

```
// Examples of managed events:  
__event ClickEventHandler* OnClick; // data member as event  
__event void OnClick(String* s); // method as event
```

При неявном объявлении управляемого события можно указать `add` и `remove` методы доступа и, которые вызываются при добавлении или удалении обработчиков событий. Можно также определить функцию-член, которая вызывает (вызывает) событие извне класса.

Пример: собственные события

```
// EventHandling_Native_Event.cpp
// compile with: /c
[event_source(native)]
class CSource {
public:
    __event void MyEvent(int nValue);
};
```

Пример: события COM

```
// EventHandling_COM_Event.cpp
// compile with: /c
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>

[ module(dll, name="EventSource", uuid="6E46B59E-89C3-4c15-A6D8-B8A1CEC98830") ];

[ dual, uuid("00000000-0000-0000-0000-000000000002") ]
__interface IEventSource {
    [id(1)] HRESULT MyEvent();
};

[ coclass, uuid("00000000-0000-0000-0000-000000000003"), event_source(com) ]
class CSource : public IEventSource {
public:
    __event __interface IEventSource;
    HRESULT FireEvent() {
        __raise MyEvent();
        return S_OK;
    }
};
```

См. также

[Словами](#)

[Обработка событий](#)

[event_source](#)
[event_receiver](#)
[__hook](#)
[__unhook](#)
[__raise](#)



12.11.2021 • 2 minutes to read

Связывает метод обработчика с событием.

NOTE

Атрибуты событий в машинном коде C++ несовместимы со стандартным C++. Они не компилируются при указании `/permissive-` режима соответствия.

Синтаксис

```
long __hook(
    &SourceClass::EventMethod,
    source,
    &ReceiverClass::HandlerMethod
    [, receiver = this]
);

long __hook(
    interface,
    source
);
```

Параметры

`&SourceClass::EventMethod`

Указатель на метод события, к которому присоединяется метод обработчика событий.

- Собственные события C++: `SourceClass` является классом источника событий и `EventMethod` является событием.
- СОМ-события — `SourceClass` это интерфейс источника событий, который `EventMethod` является одним из методов.
- Управляемые события: `SourceClass` — это класс источника событий, а `EventMethod` — событие.

`interface`

Имя интерфейса, к которому выполняется подключение `receiver`, только для приемников событий СОМ, в которых `Layout_dependent` параметр `event_receiver` атрибута имеет значение `true`.

`source`

Указатель на экземпляр источника события. В зависимости от кода `type`, указанного в `event_receiver`, `source` может иметь один из следующих типов:

- Собственный указатель на объект источника события.
- `IUnknown` Указатель на основе объекта (сом-источник).
- Указатель на управляемый объект (для управляемых событий).

`&ReceiverClass::HandlerMethod`

Указатель на метод обработчика событий, который необходимо присоединить к событию. Обработчик

указывается как метод класса или ссылки на тот же объект. Если не указать имя класса, `__hook` предполагается, что класс является тем, из которого он вызывается.

- Собственные события C++: `ReceiverClass` — это класс приемника событий и `HandlerMethod` обработчик.
- COM-события — `ReceiverClass` это интерфейс приемника событий, который `HandlerMethod` является одним из его обработчиков.
- Управляемые события: `ReceiverClass` — это класс приемника событий, а `HandlerMethod` — обработчик.

`receiver`

Используемых Указатель на экземпляр класса приемника событий. Если получатель не указан, по умолчанию используется класс получателя или структура, в которой `__hook` вызывается.

Использование

Можно использовать в любой области видимости функции, включая функцию `main`, вне класса приемника событий.

Remarks

Используйте встроенную функцию `__hook` в приемнике событий, чтобы связать метод обработчика или привязать его к методу события. Затем, когда этот источник вызывает указанное событие, вызывается указанный обработчик. Можно подключить несколько обработчиков к одному событию или несколько событий к одному обработчику.

Существует две формы `__hook`. В большинстве случаев можно использовать первую форму (с четырьмя аргументами), в частности, для приемников событий COM, в которых параметром `layout_dependent event_receiver` атрибута является `false`.

В таких случаях не нужно подключать все методы в интерфейсе перед срабатыванием события в одном из методов. Необходимо только подключить метод, обрабатывающий событие. Вторую форму (с двумя аргументами) можно использовать `__hook` только для приемника событий COM, в котором `Layout_dependent = true`.

`__hook` Возвращает значение типа `long`. Ненулевое возвращаемое значение указывает на наличие ошибки (управляемые события создают исключение).

Компилятор проверяет наличие события и соответствие сигнатур события и делегата.

Можно вызывать `__hook` и `__unhook` за пределами приемника событий, за исключением событий COM.

Альтернативой использованию `__hook` является использование оператора `+ =`.

Дополнительные сведения о кодировании управляемых событий в новом синтаксисе см `event`. В разделе.

NOTE

Класс-шаблон или структура не могут содержать события.

Пример

Примеры см. в разделе [Обработка событий в машинном коде C++](#) и [Обработка событий в com](#).

См. также

[Словами](#)

[Обработка событий](#)

[event_source](#)

[event_receiver](#)

[__event](#)

[__unhook](#)

[__raise](#)

`_raise` This

12.11.2021 • 2 minutes to read

Выделяет место вызова события.

NOTE

Атрибуты событий в машинном коде C++ несовместимы со стандартным C++. Они не компилируются при указании `/permissive-` режима соответствия.

Синтаксис

```
_raise method-declarator ;
```

Remarks

Из управляемого кода событие может быть вызвано только из класса, в котором он определен.

Дополнительные сведения см. на веб-сайте [event](#).

Ключевое слово `_raise` вызывает ошибку при вызове события, не являющегося событием.

NOTE

Класс-шаблон или структура не могут содержать события.

Пример

```
// EventHandlingRef_raise.cpp
struct E {
    __event void func1();
    void func1(int) {}

    void func2() {}

    void b() {
        __raise func1();
        __raise func1(1); // C3745: 'int Event::bar(int)':           //
                           // only an event can be 'raised'
        __raise func2(); // C3745
    }
};

int main() {
    E e;
    __raise e.func1();
    __raise e.func1(1); // C3745
    __raise e.func2(); // C3745
}
```

См. также

Словами

Обработка событий

`_event`

`_hook`

`_unhook`

Расширения компонентов для .NET и UWP

Отменяет связь метода обработчика с событием.

NOTE

Атрибуты событий в машинном коде C++ несовместимы со стандартным C++. Они не компилируются при указании `/permissive-` режима соответствия.

Синтаксис

```
long __unhook(
    &SourceClass::EventMethod,
    source,
    &ReceiverClass::HandlerMethod
    [, receiver = this]
);

long __unhook(
    interface,
    source
);

long __unhook(
    source
);
```

Параметры

`&SourceClass::EventMethod`

Указатель на метод события, с помощью которого отсоединяется метод обработчика событий.

- Собственные события C++: `SourceClass` является классом источника событий и `EventMethod` является событием.
- СОМ-события — `SourceClass` это интерфейс источника событий, который `EventMethod` является одним из методов.
- Управляемые события: `SourceClass` — это класс источника событий, а `EventMethod` — событие.

`interface`

Имя интерфейса, которое отсоединяется от получателя, только для приемников событий СОМ, в которых параметром `layout_dependent event_receiver` атрибута является `true`.

`source`

Указатель на экземпляр источника события. В зависимости от кода `type`, указанного в `event_receiver`, источником может быть один из следующих типов:

- Собственный указатель на объект источника события.
- `IUnknown` Указатель на основе объекта (сом-источник).
- Указатель на управляемый объект (для управляемых событий).

`&ReceiverClass::HandlerMethod` Указатель на метод обработчика событий, который необходимо отсоединить от события. Обработчик указывается как метод класса или ссылки на один и тот же объект. Если не указать имя класса, `_unhook` предполагается, что класс будет называться.

- Собственные события C++: `ReceiverClass` — это класс приемника событий и `HandlerMethod` обработчик.
- COM-события — `ReceiverClass` это интерфейс приемника событий, который `HandlerMethod` является одним из его обработчиков.
- Управляемые события: `ReceiverClass` — это класс приемника событий, а `HandlerMethod` — обработчик.

`receiver` используемых Указатель на экземпляр класса приемника событий. Если получатель не указан, по умолчанию используется класс получателя или структура, в которой `_unhook` вызывается.

Использование

Может использоваться в любой области видимости функции, включая `main`, за пределами класса приемника событий.

Remarks

Используйте встроенную функцию `_unhook` в приемнике событий, чтобы разорвать связь или отсоединить метод обработчика от метода события.

Существует три формы `_unhook`. В большинстве случаев можно использовать первую форму (четыре аргумента). Вторую форму (с двумя аргументами) можно использовать `_unhook` только для приемника событий COM; она отсоединяет весь интерфейс событий. Третья форма (один аргумент) используется для отсоединения всех делегатов от указанного источника.

Ненулевое возвращаемое значение указывает на наличие ошибки (управляемые события создадут исключение).

При вызове `_unhook` для события и обработчика событий, которые еще не подключены, они не будут действовать.

Во время компиляции компилятор проверяет, что событие существует, и выполняет проверку типа параметра с указанным обработчиком.

Можно вызывать `_hook` и `_unhook` за пределами приемника событий, за исключением событий COM.

Альтернативой использованию `_unhook` является использование оператора `=`.

Дополнительные сведения о кодировании управляемых событий в новом синтаксисе см. в разделе [event](#).

NOTE

Класс-шаблон или структура не могут содержать события.

Пример

Примеры см. в разделе [Обработка событий в машинном коде C++](#) и [Обработка событий в com](#).

См. также:

Словами

event_source
event_receiver
__event
__hook
__raise

Обработка событий в неуправляемом C++

12.11.2021 • 2 minutes to read

При обработке событий в собственном C++ вы настраиваете источник события и приемник событий, используя атрибуты `event_source` и `event_receiver` соответственно, указывая `type = native`. Эти атрибуты позволяют классам, к которым они применяются, вызывать события и управлять событиями в собственном контексте, не являющемся контекстом COM.

NOTE

Атрибуты событий в машинном коде C++ несовместимы со стандартным C++. Они не компилируются при указании `/permissive-` режима соответствия.

Обявление событий

В классе источника событий используйте `_event` ключевое слово в объявлении метода, чтобы объявить метод как событие. Обязательно объягите метод, но не определяйте его. В этом случае возникает ошибка компилятора, так как компилятор определяет метод неявно, когда он преобразуется в событие. Собственные события могут быть методами с нулевым или большим количеством параметров. Возвращаемым типом может быть `void` или любой целочисленный тип.

Определение обработчиков событий

В классе приемника событий определяются обработчики событий. Обработчики событий — это методы с сигнатурами (возвращаемыми типами, соглашениями о вызовах и аргументами), которые соответствуют событию, которое они будут обработаны.

Подключение обработчиков событий к событиям

Кроме того, в классе приемника событий используется встроенная функция `_hook` для связывания событий с обработчиками событий и для разрыва `_unhook` связи между событиями из обработчиков событий. Можно прикрепить несколько событий к обработчику событий либо несколько обработчиков событий к одному событию.

События, вызывающие срабатывание

Чтобы запустить событие, вызовите метод, объявленный как событие в классе источника событий. Если обработчики прикреплены к событию, они будут вызваны.

Код события машинного кода C++

В следующем примере демонстрируется порождение события в собственном коде C++. Для компиляции и выполнения примера см. комментарии в коде. Чтобы создать код в Visual Studio интегрированной среде разработки, убедитесь, что `/permissive-` параметр отключен.

Пример

Код

```

// evh_native.cpp
// compile by using: cl /EHsc /W3 evh_native.cpp
#include <stdio.h>

[event_source(native)]
class CSource {
public:
    __event void MyEvent(int nValue);
};

[event_receiver(native)]
class CReceiver {
public:
    void MyHandler1(int nValue) {
        printf_s("MyHandler1 was called with value %d.\n", nValue);
    }

    void MyHandler2(int nValue) {
        printf_s("MyHandler2 was called with value %d.\n", nValue);
    }

    void hookEvent(CSource* pSource) {
        __hook(&CSource::MyEvent, pSource, &CReceiver::MyHandler1);
        __hook(&CSource::MyEvent, pSource, &CReceiver::MyHandler2);
    }

    void unhookEvent(CSource* pSource) {
        __unhook(&CSource::MyEvent, pSource, &CReceiver::MyHandler1);
        __unhook(&CSource::MyEvent, pSource, &CReceiver::MyHandler2);
    }
};

int main() {
    CSource source;
    CReceiver receiver;

    receiver.hookEvent(&source);
    __raise source.MyEvent(123);
    receiver.unhookEvent(&source);
}

```

Выход

```

MyHandler2 was called with value 123.
MyHandler1 was called with value 123.

```

См. также

[Обработка событий](#)

Обработка событий в СОМ

12.11.2021 • 3 minutes to read

При обработке СОМ-событий вы настраиваете источник события и приемник событий с помощью `event_source` `event_receiver` атрибутов и соответственно, указывая `type = com`. Эти атрибуты вставляют соответствующий код для пользовательских, диспетчеризации и сдвоенных интерфейсов. Внедренный код позволяет классам с атрибутами вызывать события и управлять событиями через точки подключения СОМ.

NOTE

Атрибуты событий в машинном коде C++ несовместимы со стандартным C++. Они не компилируются при указании `/permissive-` режима соответствия.

Обявление событий

В классе источника событий используйте `_event` ключевое слово в объявлении интерфейса, чтобы объявить методы интерфейса как события. События этого интерфейса запускаются при вызове их в качестве методов интерфейса. Методы в интерфейсах событий могут иметь ноль или более параметров (которые должны быть *в параметрах*). Тип возвращаемого значения может быть пустым или любым целочисленным типом.

Определение обработчиков событий

Обработчики событий определяются в классе приемника событий. Обработчики событий — это методы с сигнатурами (возвращаемыми типами, соглашениями о вызовах и аргументами), которые соответствуют событию, которое они будут обработаны. Для СОМ-событий соглашения о вызовах не должны совпадать. Дополнительные сведения см. в разделе [зависимые от макета события com](#) ниже.

Подключение обработчиков событий к событиям

Кроме того, в классе приемника событий используется встроенная функция `_hook` для связывания событий с обработчиками событий и для разрыва `_unhook` связи между событиями из обработчиков событий. Можно прикрепить несколько событий к обработчику событий либо несколько обработчиков событий к одному событию.

NOTE

Как правило, существует два метода предоставления приемнику событий СОМ доступа к определениям интерфейса исходного кода событий. Во-первых, как показано ниже, можно предоставить общий доступ к одному файлу заголовка. Второй — использовать `#import` с `embedded_idl` квалификатором импорта, чтобы библиотека типов источника событий записывалась в TLH-файл с сохранением сформированного атрибутом кода.

События, вызывающие срабатывание

Чтобы запустить событие, вызовите метод в интерфейсе, объявленном с помощью `_event` ключевого слова в классе источника событий. Если обработчики прикреплены к событию, они будут вызваны.

Код события COM

В следующем примере демонстрируется запуск события в COM-классе. Для компиляции и выполнения примера см. комментарии в коде.

```
// evh_server.h
#pragma once

[ dual, uuid("00000000-0000-0000-0000-000000000001") ]
__interface IEvents {
    [id(1)] HRESULT MyEvent([in] int value);
};

[ dual, uuid("00000000-0000-0000-0000-000000000002") ]
__interface IEventSource {
    [id(1)] HRESULT FireEvent();
};

class DECLSPEC_UUID("530DF3AD-6936-3214-A83B-27B63C7997C4") CSource;
```

Затем сервер.

```
// evh_server.cpp
// compile with: /LD
// post-build command: Regsvr32.exe /s evh_server.dll
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include "evh_server.h"

[ module(dll, name="EventSource", uuid="6E46B59E-89C3-4c15-A6D8-B8A1CEC98830") ];

[coclass, event_source(com), uuid("530DF3AD-6936-3214-A83B-27B63C7997C4")]
class CSource : public IEventSource {
public:
    __event __interface IEvents;

    HRESULT FireEvent() {
        __raise MyEvent(123);
        return S_OK;
    }
};
```

Затем клиент.

```

// evh_client.cpp
// compile with: /link /OPT:NOREF
#define _ATL_ATTRIBUTES 1
#include <atlbase.h>
#include <atlcom.h>
#include <stdio.h>
#include "evh_server.h"

[ module(name="EventReceiver") ];

[ event_receiver(com) ]
class CReceiver {
public:
    HRESULT MyHandler1(int nValue) {
        printf_s("MyHandler1 was called with value %d.\n", nValue);
        return S_OK;
    }

    HRESULT MyHandler2(int nValue) {
        printf_s("MyHandler2 was called with value %d.\n", nValue);
        return S_OK;
    }

    void HookEvent(IEventSource* pSource) {
        __hook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler1);
        __hook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler2);
    }

    void UnhookEvent(IEventSource* pSource) {
        __unhook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler1);
        __unhook(&IEvents::MyEvent, pSource, &CReceiver::MyHandler2);
    }
};

int main() {
    // Create COM object
    CoInitialize(NULL);
    {
        IEventSource* pSource = 0;
        HRESULT hr = CoCreateInstance(__uuidof(CSource), NULL, CLSCTX_ALL, __uuidof(IEventSource),
(void **) &pSource);
        if (FAILED(hr)) {
            return -1;
        }

        // Create receiver and fire event
        CReceiver receiver;
        receiver.HookEvent(pSource);
        pSource->FireEvent();
        receiver.UnhookEvent(pSource);
    }
    CoUninitialize();
    return 0;
}

```

Выходные данные

```

MyHandler1 was called with value 123.
MyHandler2 was called with value 123.

```

События COM, зависимые от макета

Зависимость от макета становится проблемой только в программировании COM. При обработке

собственных и управляемых событий сигнатуры (возвращаемый тип, соглашение о вызовах и аргументы) должны соответствовать своим событиям, но имена обработчиков не должны совпадать с их событиями.

Однако при обработке событий COM, если параметру присвоить значение `Layout_dependent` `event_receiver` `true`, применяется имя и сопоставление подписи. Имена и подписи обработчиков в приемнике событий и в обработчиках событий должны точно совпадать.

Если параметр `Layout_dependent` имеет значение `false`, соглашение о вызовах и класс хранения (виртуальный, статический и т. д.) могут быть смешанными и сопоставлены между методом события обработки и методами обработчика (его делегатами). Это немного более эффективно `Layout_dependent` = `true`.

Например, предположим, что `IEventSource` определен как объект, имеющий следующие методы.

```
[id(1)] HRESULT MyEvent1([in] int value);
[id(2)] HRESULT MyEvent2([in] int value);
```

Предположим, источник событий имеет следующую форму:

```
[coclass, event_source(com)]
class CSource : public IEventSource {
public:
    __event __interface IEvents;

    HRESULT FireEvent() {
        MyEvent1(123);
        MyEvent2(123);
        return S_OK;
    }
};
```

Затем в приемнике событий любой обработчик, который прикрепляется к методу в `IEventSource`, должен соответствовать имени и подписи следующим образом.

```
[coclass, event_receiver(com, true)]
class CReceiver {
public:
    HRESULT MyEvent1(int nValue) { // name and signature matches MyEvent1
        ...
    }
    HRESULT MyEvent2(E c, char* pc) { // signature doesn't match MyEvent2
        ...
    }
    HRESULT MyHandler1(int nValue) { // name doesn't match MyEvent1 (or 2)
        ...
    }
    void HookEvent(IEventSource* pSource) {
        __hook(IFace, pSource); // Hooks up all name-matched events
                               // under layout_dependent = true
        __hook(&IFace::MyEvent1, pSource, &CReceive::MyEvent1); // valid
        __hook(&IFace::MyEvent2, pSource, &CSink::MyEvent2); // not valid
        __hook(&IFace::MyEvent1, pSource, &CSink:: MyHandler1); // not valid
    }
};
```

См. также раздел

[Обработка событий](#)

Модификаторы, используемые в системах Microsoft

12.11.2021 • 2 minutes to read

В этом разделе описываются специальные расширения Microsoft для C++ в следующих аспектах языка:

- [На основе адресации](#), практика использования указателя в качестве основания, на основе которого могут быть смещены другие указатели
- [Соглашение о вызовах функций](#)
- Расширенные атрибуты класса хранения, объявленные с помощью ключевого слова `_declspec`
- Ключевое слово `_w64`

специальные ключевые слова Microsoft

Многие ключевые слова для систем Microsoft позволяют изменять деклараторы, чтобы создавать производные типы. Дополнительные сведения об деклараторах см. в разделе [деклараторы](#).

КЛЮЧЕВОЕ СЛОВО	ЗНАЧЕНИЕ	ПОЗВОЛЯЮТ СОЗДАВАТЬ ПРОИЗВОДНЫЕ ТИПЫ?
<code>_based</code>	Указанное после него имя объявляет 32-разрядное смещение относительно 32-разрядной базы, содержащейся в объявлении.	Да
<code>_cdecl</code>	В указанном после него имени используются соглашения об именовании и вызовах языка C.	Да
<code>_declspec</code>	Указанное после него имя задает атрибут класса хранения для систем Microsoft.	Нет
<code>_fastcall</code>	Указанное после него имя объявляет функцию, в которой аргументы передаются не через стек, а через регистры (если они доступны).	Да
<code>_restrict</code>	Аналогично <code>_declspec</code> (<code>restrict</code>), но для использования в переменных.	Нет
<code>_stdcall</code>	Указанное после него имя задает функцию, в которой соблюдается стандартное соглашение об именовании.	Да
<code>_w64</code>	Указывает, что в 64-разрядном компиляторе тип данных имеет больший размер.	Нет

КЛЮЧЕВОЕ СЛОВО	ЗНАЧЕНИЕ	ПОЗВОЛЯЮТ СОЗДАВАТЬ ПРОИЗВОДНЫЕ ТИПЫ?
<code>_unaligned</code>	Указывает, что указатель на тип или другие данные не выровнен.	Нет
<code>_vectorcall</code>	Указанное после него имя объявляет функцию, в которой аргументы передаются не через стек, а через регистры (если они доступны), включая регистры SSE.	Да

См. также

[Справочник по языку C++](#)

Базовые адреса

12.11.2021 • 2 minutes to read

Этот раздел содержит следующие подразделы:

- [Грамматика __based](#)
- [Указатели на основе](#)

См. также

[Модификаторы, специфичные для Майкрософт](#)

Грамматика выражения __based

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Базовая адресация полезна, когда требуется точный контроль над сегментом, в котором выделяются объекты (статические и динамические данные).

Единственная форма одноадресной адресации в 32-разрядных и 64-разрядных компиляциях — это "основано на указателе", который определяет тип, который 32 содержит откомпилированное или 64-разрядное смещение к-разрядному или 32-битному основанию или на основе `void`.

Грамматика

Модификатор на основе диапазона. __based (базовое выражение)

базовый-Expression: на основе вариаблебасед-abstract-деклараторсегмент-намесегмент-CAST

переменная на основе. идентификатор

декларатор на основе абстрактного объявления. абстрактное объявление

базовый тип. имя типа

Завершение блока, относящегося только к системам Microsoft

См. также

[Указатели на основе](#)

Основанные указатели (C++)

12.11.2021 • 2 minutes to read

`__based` Ключевое слово позволяет объявлять указатели на основе указателей (указателей, которые смещены от существующих указателей). `__based` Ключевое слово относится только к Microsoft.

Синтаксис

```
type __based( base ) declarator
```

Remarks

Указатели, основанные на адресах указателей, являются единственной формой `__based` ключевого слова, допустимого в 32-разрядных или 64-разрядных компиляциях. В 32-разрядном компиляторе Microsoft C/C++ относительный указатель имеет 32-разрядное смещение от 32-разрядной базы указателя. То же ограничение действует и для 64-разрядных сред, где относительный указатель имеет 64-разрядное смещение от 64-разрядной базы указателя.

Указатели на основе указателей, в частности, используются для постоянных идентификаторов, которые содержат указатели. Связанный список, состоящий из указателей на основе указателей, можно сохранить на диск, а затем перезагрузить в другое место в памяти. При этом все указатели останутся действительными. Пример:

```
// based_pointers1.cpp
// compile with: /c
void *vpBuffer;
struct llist_t {
    void __based( vpBuffer ) *vpData;
    struct llist_t __based( vpBuffer ) *llNext;
};
```

Указателю `vpBuffer` назначается адрес в памяти, который выделяется на более позднем этапе программы. Связанный список перемещается относительно значения `vpBuffer`.

NOTE

Сохранение идентификаторов, содержащих указатели, также можно выполнить с помощью [отображенных в память файлов](#).

Когда выполняется разыменование относительных указателей, база должна быть либо явно указана, либо неявно известна из объявления.

Для совместимости с предыдущими версиями `_based` является синонимом, `__based` если только параметр компилятора [/Za](#) не ([отключил расширения языка](#)) указан.

Пример

В следующем примере демонстрируется изменение относительного указателя путем изменения его базы.

```
// based_pointers2.cpp
// compile with: /EHsc
#include <iostream>

int a1[] = { 1,2,3 };
int a2[] = { 10,11,12 };
int *pBased;

typedef int __based(pBased) * pBasedPtr;

using namespace std;
int main() {
    pBased = &a1[0];
    pBasedPtr pb = 0;

    cout << *pb << endl;
    cout << *(pb+1) << endl;

    pBased = &a2[0];

    cout << *pb << endl;
    cout << *(pb+1) << endl;
}
```

```
1
2
10
11
```

См. также

[Ключевые слова](#)
`alloc_text`

Соглашения о вызовах

12.11.2021 • 2 minutes to read

В компиляторе Visual C/C++ принято несколько разных соглашений о вызовах внутренних и внешних функций. Зная эти подходы, вам будет проще выполнять отладку программ и привязывать свой код к процедурам на языке ассемблера.

В разделах, посвященных этой теме, говорится о том, чем различаются эти соглашения о вызовах, как передаются аргументы и как функции возвращают значения. Кроме того, в них рассматриваются вызовы возможностей с атрибутом naked — дополнительная возможность, благодаря которой вы сможете создавать собственный код пролога и эпилог.

Сведения о соглашениях о вызовах для процессоров x64 см. в разделе [соглашение о вызовах](#).

Разделы этой статьи

- [Передача аргументов и соглашения об именовании](#) (`__cdecl` , `__stdcall` , `__fastcall` и др.)
- [Пример вызова: прототип функции и вызов](#)
- [Использование вызова функции с атрибутом naked для написания собственного кода пролога и эпилога](#)
- [Соглашения о сопроцессоре и вызовах с плавающей точкой](#)
- [Устаревшие соглашения о вызовах](#)

См. также

[Модификаторы, специфичные для Майкрософт](#)

Передача аргументов и соглашения именования

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Компиляторы Microsoft C++ позволяют задавать соглашения для передачи аргументов и возвращаемых значений между функциями и вызывающими объектами. Некоторые соглашения доступны не на всех поддерживаемых платформах, и в некоторых соглашениях используются реализации, зависящие от платформы. В большинстве случаев ключевые слова или параметры компилятора, которые задают неподдерживаемое на определенной платформе соглашение, игнорируются, и используется соглашение по умолчанию для данной платформы.

На платформах x86 все аргументы расширяются до 32 бит, когда они передаются. Возвращаемые значения также расширяются до 32 бит и возвращаются в регистре EAX, за исключением 8-байтовых структур, которые возвращаются в паре регистров EDX:EAX. Более крупные структуры возвращаются в регистре EAX в виде указателей на скрытые возвращаемые структуры. Параметры помещаются в стек справа налево. Структуры, не являющиеся данными POD (обычные старые данные), не возвращаются в регистрах.

Компилятор создает код пролога и эпилога для сохранения и восстановления регистров ESI, EDI, EBX и EBP, если они используются в функции.

NOTE

Если функция возвращает структуру, объединение или класс по значению, все определения типа должны быть одинаковыми, в противном случае программа завершается сбоем в среде выполнения.

Сведения о том, как определить собственный Пролог и код Эпилога, см. в разделе [вызовы функций с атрибутом naked](#).

Сведения о соглашениях о вызовах по умолчанию в коде, предназначенном для платформ x64, см. в разделе [соглашение о вызовах x64](#). Сведения о проблемах с соглашениями о вызовах в коде, предназначенном для платформ ARM, см. в разделе [общие Visual C++ проблемы миграции ARM](#).

Компилятор Visual C/C++ поддерживает следующие соглашения о вызовах.

КЛЮЧЕВОЕ СЛОВО	ОЧИСТКА СТЕКА	ПЕРЕДАЧА ПАРАМЕТРОВ
_cdecl	Caller	Параметры помещаются в стек в обратном порядке (справа налево)
_clrcall	Недоступно	Параметры загружаются в стек выражений CLR по-порядку (слева направо).
_stdcall	Вызывающая функция	Параметры помещаются в стек в обратном порядке (справа налево)
_fastcall	Вызывающая функция	Хранятся в регистрах, затем помещаются в стек

КЛЮЧЕВОЕ СЛОВО	ОЧИСТКА СТЕКА	ПЕРЕДАЧА ПАРАМЕТРОВ
<code>_thiscall</code>	Вызываемая функция	Помещается в стек; <code>this</code> указатель, хранящийся в ECX
<code>_vectorcall</code>	Вызываемая функция	Хранятся в регистрах, затем помещаются в стек в обратном порядке (справа налево)

Связанные сведения см. в разделе [устаревшие соглашения о вызовах](#).

Завершение блока, относящегося только к системам Майкрософт

См. также

[Соглашения о вызовах](#)

`_cdecl`

12.11.2021 • 2 minutes to read

`_cdecl` является соглашением о вызовах по умолчанию для программ С и С++. Так как стек очищается вызывающим объектом, он может выполнять `vararg` функции. `_cdecl` Соглашение о вызовах создает большие исполняемые объекты, чем `stdcall`, так как требует, чтобы каждый вызов функции включал код очистки стека. В следующем списке показана реализация этого соглашения о вызовах. `_cdecl` Модификатор является специфичным для Microsoft.

ЭЛЕМЕНТ	РЕАЛИЗАЦИЯ
Порядок передачи аргументов	Справа налево.
Обязанность по обслуживанию стека	Вызывающая функция выводит аргументы из стека.
Соглашение об оформлении имен	Имя символа подчеркивания (_) предваряется именами, за исключением случаев, когда _ экспортируются <code>_cdecl</code> функции, использующие компоновку С.
Соглашение о преобразовании регистра	Изменение регистра не выполняется.

NOTE

Связанные сведения см. в разделе [декорированные имена](#).

Поместите `_cdecl` модификатор перед именем переменной или функции. Так как соглашения об именовании и вызове С являются стандартными, единственное время, которое необходимо использовать `_cdecl` в коде x86, — это только в том случае, если вы указали `/Gv` параметр компилятора (векторкалл), `/Gz` (STDCALL) или `/Gr` (fastcall). Параметр компилятора `/GD` вызывает принудительное `_cdecl` соглашение о вызовах.

В процессорах ARM и x64 `_cdecl` принимается, но обычно игнорируется компилятором. По соглашению на ARM и x64 аргументы передаются в регистрах, когда это возможно, а последующие аргументы передаются в стек. В коде x64 используйте `_cdecl` для переопределения параметра компилятора `/GV` и используйте соглашение о вызовах по умолчанию x64.

Если используется внетрочное определение нестатической функции класса, то модификатор соглашения о вызовах не должен быть задан во внетрочном определении. То есть для нестатических методов-членов считается, что соглашение о вызовах, указанное во время объявления, было сделано в точке определения. Рассмотрим следующее определение класса:

```
struct CMyClass {  
    void __cdecl mymethod();  
};
```

В этом случае следующий код:

```
void CMyClass::mymethod() { return; }
```

эквивалентен следующему:

```
void __cdecl CMyClass::mymethod() { return; }
```

Для совместимости с предыдущими версиями `cdecl` и `_cdecl` являются синонимами, `__cdecl` если только параметр компилятора [/Za не \(отключил расширения языка\)](#) указан.

Пример

В следующем примере компилятору дается инструкция использовать для функции `system` соглашения об именовании и вызовах С:

```
// Example of the __cdecl keyword on function
int __cdecl system(const char *);
// Example of the __cdecl keyword on function pointer
typedef BOOL (__cdecl *funcname_ptr)(void * arg1, const char * arg2, DWORD flags, ...);
```

См. также

[Передача аргументов и соглашения об именовании](#)

[Ключевые слова](#)

__clrcall

12.11.2021 • 2 minutes to read

Указывает, что функцию можно вызвать только из управляемого кода. Используйте __clrcall для всех виртуальных функций, которые будут вызываться только из управляемого кода. Однако это соглашение о вызовах невозможно использовать для функций, которые будут вызываться из машинного кода.

Модификатор __clrcall зависит от корпорации Майкрософт.

Используйте __clrcall, чтобы повысить производительность при вызове из управляемой функции в виртуальную управляемую функцию или из управляемой функции в управляемую функцию через указатель.

Точки входа представляют собой отдельные функции, создаваемые компилятором. Если функция имеет машинные и управляемые точки входа, одна из них будет фактической функцией с реализацией функции. Другая функция будет отдельной функцией (преобразователем), которая вызывает фактическую функцию и позволяет среде CLR выполнять PInvoke. При пометке функции как __clrcall необходимо указать, что реализация функции должна быть MSIL и что собственная функция точки входа не будет создана.

При получении адреса собственной функции, если параметр __clrcall не указан, компилятор использует собственную точку входа. __clrcall указывает, что функция является управляемой, и нет необходимости выполнять переход от управляемого кода к машинному. В этом случае компилятор использует управляемую точку входа.

Если `/clr` используется (не `/clr:pure` или `/clr:safe`) и __clrcall не используется, при получении адреса функции всегда возвращается адрес собственной функции точки входа. При использовании __clrcall собственная функция точки входа не создается, поэтому вы получаете адрес управляемой функции, а не функцию преобразователя точки входа. Дополнительные сведения см. в разделе [Двойное преобразователь](#). Параметры компилятора `/clr: pure` и `/clr: Сейф` являются устаревшими в Visual Studio 2015 и не поддерживаются в Visual Studio 2017.

[параметр/CLR \(компиляция общеязыковой среды выполнения\)](#) подразумевает, что все функции и указатели функций __clrcall, и компилятор не позволит пометить функцию внутри компилируемого объекта как не __clrcall. При использовании `/clr: pure` __clrcall можно указывать только в указателях на функции и внешних объявлениях.

Вы можете напрямую вызывать функции __clrcall из существующего кода C++, который был скомпилирован с помощью /CLR, если эта функция имеет реализацию MSIL. функции __clrcall не могут вызываться напрямую из функций, которые имеют встроенный ASM и вызывают интринисикс для конкретного процессора, например, даже если эти функции компилируются с помощью `/clr`.

__clrcall указатели функций предназначены только для использования в домене приложения, в котором они были созданы. Вместо того, чтобы передавать __clrcall указатели функций между доменами приложений, используйте [CrossAppDomainDelegate](#). Дополнительные сведения см. в разделе [домены приложений и Visual C++](#).

Примеры

Обратите внимание, что при объявлении функции с __clrcall при необходимости будет сформирован код. Например, при вызове функции.

```

// clrcall2.cpp
// compile with: /clr
using namespace System;
int __clrcall Func1() {
    Console::WriteLine("in Func1");
    return 0;
}

// Func1 hasn't been used at this point (code has not been generated),
// so runtime returns the address of a stub to the function
int (__clrcall *pf)() = &Func1;

// code calls the function, code generated at difference address
int i = pf(); // comment this line and comparison will pass

int main() {
    if (&Func1 == pf)
        Console::WriteLine("&Func1 == pf, comparison succeeds");
    else
        Console::WriteLine("&Func1 != pf, comparison fails");

    // even though comparison fails, stub and function call are correct
    pf();
    Func1();
}

```

```

in Func1
&Func1 != pf, comparison fails
in Func1
in Func1

```

В следующем примере показано, что можно определить указатель функции, например объявить, что указатель функции будет вызываться только из управляемого кода. Это позволит компилятору непосредственно вызвать управляемую функцию и избежать машинной точки входа (проблема двойного преобразования).

```

// clrcall3.cpp
// compile with: /clr
void Test() {
    System::Console::WriteLine("in Test");
}

int main() {
    void (*pTest)() = &Test;
    (*pTest)();

    void (__clrcall *pTest2)() = &Test;
    (*pTest2)();
}

```

См. также раздел

[Передача аргументов и соглашения об именовании](#)

[Ключевые слова](#)

`_stdcall`

12.11.2021 • 2 minutes to read

`_stdcall` Соглашение о вызовах используется для вызова функций API Win32. Вызываемый очищает стек, поэтому компилятор делает `vararg` функции `_cdecl`. Для функций, использующих данное соглашение о вызовах, требуется прототип. `_stdcall` Модификатор является специфичным для Microsoft.

Синтаксис

Тип возвращаемого значения `_stdcall` имя-функции`[(Argument-List)]`

Remarks

В следующем списке показана реализация этого соглашения о вызовах.

ЭЛЕМЕНТ	РЕАЛИЗАЦИЯ
Порядок передачи аргументов	Справа налево.
Соглашение о передаче аргументов	По значению, кроме случаев передачи указателя или ссылочного типа.
Обязанность по обслуживанию стека	Вызываемая функция извлекает свои аргументы из стека.
Соглашение об оформлении имен	Символ подчеркивания (<code>_</code>) предваряется именем. За именем следует знак at (<code>@</code>), за которым следует число байтов (в десятичной системе) в списке аргументов. Поэтому функция, объявленная как <code>int func(int a, double b)</code> декорируется следующим образом: <code>_func@12</code>
Соглашение о преобразовании регистра	None

Параметр компилятора `/gz` указывает `_stdcall` для всех функций, которые не были явно объявлены с другим соглашением о вызовах.

Для совместимости с предыдущими версиями аргумент `_stdcall` является синонимом, `_stdcall` если только параметр компилятора не `/za` ([отключает расширения языка](#)).

Функции, объявленные с помощью `_stdcall` возвращаемых модификатором значений, аналогичны функциям, объявленным с помощью `_cdecl`.

В процессорах ARM и x64 метод `_stdcall` принимает и игнорирует компилятор; в архитектурах ARM и x64 по соглашению аргументы передаются в регистры, когда это возможно, а последующие аргументы передаются в стек.

Если используется внетрочное определение нестатической функции класса, то модификатор соглашения о вызовах не должен быть задан во внетрочном определении. То есть для нестатических методов-членов считается, что соглашение о вызовах, указанное во время объявления, было сделано в

точке определения. Для приведенного ниже определения класса

```
struct CMyClass {  
    void __stdcall mymethod();  
};
```

this

```
void CMyClass::mymethod() { return; }
```

эквивалентно следующему

```
void __stdcall CMyClass::mymethod() { return; }
```

Пример

В следующем примере использование `__stdcall` результатов во всех `WINAPI` типах функций обрабатывается как стандартный вызов:

```
// Example of the __stdcall keyword  
#define WINAPI __stdcall  
// Example of the __stdcall keyword on function pointer  
typedef BOOL (__stdcall *funcname_ptr)(void * arg1, const char * arg2, DWORD flags, ...);
```

См. также:

[Передача аргументов и соглашения об именовании](#)

[Ключевые слова](#)

`_fastcall`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

`_fastcall` Соглашение о вызовах указывает, что аргументы функции должны передаваться в регистры, если это возможно. Это соглашение о вызовах применяется только к архитектуре x86. В следующем списке показана реализация этого соглашения о вызовах.

ЭЛЕМЕНТ	РЕАЛИЗАЦИЯ
Порядок передачи аргументов	Первые два значения DWORD или меньшие аргументы, найденные в списке аргументов слева направо, передаются в регистрах ECX и EDX; все остальные аргументы передаются в стек справа налево.
Обязанность по обслуживанию стека	Вызываемая функция извлекает аргументы из стека.
Соглашение об оформлении имен	@Префикс in () предваряется имя; знак AT, за которым следует число байтов (в десятичной системе) в списке параметров, является суффиксом для имен.
Соглашение о преобразовании регистра	Изменение регистра не выполняется.

NOTE

В следующих версиях компилятора могут использовать другие регистры для сохранения параметров.

Использование параметра компилятора `/GR` приводит к тому, что каждая функция в модуле будет компилироваться так, как если бы она не `_fastcall` была объявлена с помощью конфликтующего атрибута, или же имя функции имеет значение `main`.

`_fastcall` Ключевое слово принимается и игнорируется компиляторами, предназначенными для архитектуры ARM и x64; в микросхеме x64 по соглашению первые четыре аргумента передаются в регистры по возможности, а дополнительные аргументы передаются в стек. Дополнительные сведения см. в разделе [соглашение о вызовах x64](#). На микросхеме ARM можно передавать в регистрах до четырех целочисленных аргументов и до восьми аргументов с плавающей запятой; дополнительные аргументы передаются в стеке.

Если используется внетрочное определение нестатической функции класса, то модификатор соглашения о вызовах не должен быть задан во внетрочном определении. То есть для нестатических методов-членов считается, что соглашение о вызовах, указанное во время объявления, было сделано в точке определения. Рассмотрим следующее определение класса:

```
struct CMyClass {  
    void __fastcall mymethod();  
};
```

В этом случае следующий код:

```
void CMyClass::mymethod() { return; }
```

эквивалентен следующему:

```
void __fastcall CMyClass::mymethod() { return; }
```

Для совместимости с предыдущими версиями `_fastcall` является синонимом, `__fastcall` если только параметр компилятора [/Za не \(отключил расширения языка\)](#) указан.

Пример

В следующем примере аргументы передаются в функцию `DeleteAggrWrapper` в регистрах.

```
// Example of the __fastcall keyword
#define FASTCALL __fastcall

void FASTCALL DeleteAggrWrapper(void* pWrapper);
// Example of the __fastcall keyword on function pointer
typedef BOOL (_fastcall *funcname_ptr)(void * arg1, const char * arg2, DWORD flags, ...);
```

Завершение блока, относящегося только к системам Майкрософт

См. также

[Передача аргументов и соглашения об именовании](#)

[Ключевые слова](#)

`__thiscall`

12.11.2021 • 2 minutes to read

Соглашение о вызовах, **характерное для Майкрософт**, `__thiscall` используется в функциях-членах классов C++ в архитектуре x86. Это соглашение о вызовах по умолчанию, используемое функциями-членами, которые не используют переменные аргументы (`vararg` функции).

В разделе `__thiscall` вызываемый очищает стек, что невозможно для `vararg` функций. Аргументы помещаются в стек справа налево. `this` Указатель передается через Register ECX, а не в стек.

На компьютерах ARM, ARM64 и x64 `__thiscall` принимается и игнорируется компилятором. Это обусловлено тем, что в них по умолчанию используется соглашение о вызовах на основе регистров.

Одна из причин использования `__thiscall` — в классах, функции-члены которых используют по `__clrcall` умолчанию. В этом случае можно использовать, `__thiscall` чтобы сделать отдельные функции для членов, вызываемые из машинного кода.

При компиляции с параметром `/clr:pure` все функции и указатели функций задаются, `__clrcall` если не указано иное. `/clr:pure` `/clr:safe` параметры компилятора и не рекомендуются в Visual Studio 2015 и не поддерживаются в Visual Studio 2017.

`vararg` функции элементов используют `__cdecl` соглашение о вызовах. Все аргументы функции помещаются в стек, и `this` указатель помещается в стек последним.

Так как это соглашение о вызове применяется только к C++, оно не имеет схемы декорирования имен C.

При определении нестатический функции-члена класса, которая не является статической, укажите модификатор соглашения о вызове только в объявлении. Его не нужно указывать повторно в определении вне строки. Компилятор использует соглашение о вызовах, указанное во время объявления в точке определения.

См. также раздел

[Передача аргументов и соглашения об именовании](#)

`_vectorcall`

12.11.2021 • 11 minutes to read

Блок, относящийся только к системам Microsoft

`_vectorcall` Соглашение о вызовах указывает, что аргументы функции должны передаваться в регистры, если это возможно. `_vectorcall` использует больше регистров для аргументов `_fastcall`, чем или используемое по умолчанию [соглашение о вызовах x64](#). `_vectorcall` Соглашение о вызовах поддерживается только в машинном коде на процессорах x86 и x64, включающих Streaming SIMD Extensions 2 (SSE2) и более поздних версий. Используйте `_vectorcall` для ускорения функций, которые передают несколько аргументов вектора с плавающей запятой или SIMD, и выполняют операции, использующие преимущества аргументов, загруженных в регистры. В следующем списке перечислены функции, общие для реализаций x86 и x64 `_vectorcall`. Различия объясняются ниже в этом разделе.

ЭЛЕМЕНТ	РЕАЛИЗАЦИЯ
Соглашение об оформлении имен C	Имена функций добавляются в суффикс с двумя знаками "at" (@ @), за которыми следует число байтов (в десятичной системе) в списке параметров.
Соглашение о преобразовании регистра	Преобразование регистра не выполняется.

Использование `/Gv` параметра компилятора приводит к тому, что каждая функция в модуле будет компилироваться так, как если бы она `_vectorcall` не была функцией-членом, объявлена с конфликтующим атрибутом соглашения о вызове, использует `vararg` список аргументов переменной или имеет имя `main`.

Можно передать три типа аргументов путем регистрации в `_vectorcall` функциях: значения *целочисленного типа*, значения *векторного типа* и *однородные векторные значения* (Хва).

Целочисленный тип удовлетворяет двум требованиям: он соответствует размеру собственного регистра процессора (например, 4 байта на компьютере с архитектурой x86 или 8 байт на компьютере с архитектурой x64) и его можно преобразовывать в целое число с длиной, соответствующей длине регистра, и обратно без изменения его битового представления. Например, любой тип, который можно повысить до версии `int` x86 (`long long` в x64), например, или или `char` `short`, который можно привести к `int` (`long long` в x64) и вернуть исходный тип без изменений, является целочисленным типом. Целочисленные типы включают указатель, ссылку и `struct` `union` типы 4 байт (8 байт на x64) или меньше. На платформах x64 больший размер `struct` и `union` типы передаются по ссылке на память, выделенную вызывающей стороной; на платформах x86 они передаются по значению в стеке.

Тип Vector является либо типом с плавающей запятой (например,, либо `float` `double`), либо типом ВЕКТОРа SIMD, например, `_m128` или `_m256` .

Тип HVA — это составной тип, в котором может содержаться до 4 элементов данных, имеющих идентичные векторные типы. Тип HVA предъявляет то же требование к выравниванию, что и векторный тип его членов. Это пример `struct` определения Хва, которое содержит три одинаковых векторных типа и имеет 32-байтное выравнивание:

```
typedef struct {
    __m256 x;
    __m256 y;
    __m256 z;
} hva3;      // 3 element HVA type on __m256
```

Объявите функции явным образом с помощью `__vectorcall` ключевого слова в файлах заголовков, чтобы разрешить компоновку независимо скомпилированного кода без ошибок. Функции должны иметь прототипы для использования `__vectorcall` и не могут использовать `vararg` список аргументов переменной длины.

Функцию-член можно объявить с помощью `__vectorcall` описателя. Скрытый `this` Указатель передается функцией Register в качестве первого аргумента целочисленного типа.

На компьютерах ARM `__vectorcall` принимается и игнорируется компилятором.

Если используется внетрочное определение нестатической функции-члена класса, то модификатор соглашения о вызовах не должен быть задан во внетрочном определении. То есть для нестатических членов класса считается, что соглашение о вызовах, указанное во время объявления, было сделано в точке определения. Рассмотрим следующее определение класса:

```
struct MyClass {
    void __vectorcall mymethod();
};
```

В этом случае следующий код:

```
void MyClass::mymethod() { return; }
```

эквивалентен следующему:

```
void __vectorcall MyClass::mymethod() { return; }
```

`__vectorcall` При создании указателя на функцию необходимо указать модификатор соглашения о вызовах `__vectorcall`. В следующем примере создается `typedef` для указателя на `__vectorcall` функцию, которая принимает четыре `double` аргумента и возвращает `__m256` значение.

```
typedef __m256 (__vectorcall * vcfnptr)(double, double, double, double);
```

Для совместимости с предыдущими версиями аргумент `__vectorcall` является синонимом, `__vectorcall` если только параметр компилятора не [/Za](#) (отключает расширения языка) .

Соглашение о вызовах `__vectorcall` для архитектуры x64

`__vectorcall` Соглашение о вызовах в x64 расширяет стандартное соглашение о вызовах x64, чтобы воспользоваться преимуществами дополнительных регистров. Аргументы как целочисленного, так и векторного типа сопоставляются с регистрами исходя из позиции в списке аргументов. Аргументы HVA назначаются неиспользуемым векторным регистрам.

Если любые из первых четырех аргументов в порядке слева направо являются целочисленными, они передаются в регистрах, соответствующих их позициям: RCX, RDX, R8 или R9. Скрытый `this` указатель рассматривается как аргумент первого целочисленного типа. Если аргумент HVA на одной из первых

четырех позиций не может быть передан в доступных регистрах, вместо него в соответствующем регистре для целочисленных значений передается ссылка на память, выделенную вызывающим объектом. Аргументы целочисленного типа, расположенные за четвертой позицией в списке параметров, передаются в стеке.

Если любые из первых шести аргументов в порядке слева направо являются аргументами векторного типа, они передаются по значению в векторных регистрах SSE 0–5, соответствующих их позициям.

Операции с плавающей запятой и `__m128` типы передаются в регистрах XMM, а `__m256` типы передаются в регистры ИММ. В этом состоит отличие от стандартного соглашения о вызовах x64, так как векторные типы передаются по значению, а не по ссылкам; кроме того, используются дополнительные регистры. Пространство теневого стека, выделенное для аргументов векторного типа, фиксировано в 8 байт, а `/homeparams` параметр не применяется. Аргументы векторного типа начиная с седьмой позиции в списке параметров передаются в стеке по ссылке на память, выделенную вызывающим объектом.

После выделения регистров для векторных аргументов элементы данных в аргументах Хва выделяются в порядке возрастания на неиспользуемые векторные регистры XMM0 в XMM5 (или YMM0 в YMM5, для `__m256` типов), если для всего Хва доступно достаточно регистров. Если регистров недостаточно, аргумент HVA передается по ссылке на память, выделенную вызывающим объектом. Пространство теневого стека, выделенное для аргументов типа HVA, имеет фиксированный размер 8 байт с неопределенным содержимым. Аргументы HVA назначаются регистрам в порядке слева направо в списке параметров и могут находиться в любой позиции. Аргументы HVA, находящиеся на одной из первых четырех позиций в списке аргументов и не назначенные векторным регистрам, передаются по ссылке в целочисленном регистре, соответствующем их позициям. Аргументы HVA, переданные по ссылке после четвертой позиции в списке параметров, помещаются в стек.

Результаты `__vectorcall` функций возвращаются по значению в регистрах, если это возможно.

Результаты целочисленных типов, включая struct и union размером 8 байт и менее, возвращаются по значению в RAX. Результаты векторного типа возвращаются по значению в XMM0 или YMM0 в зависимости от их размера. В результатах HVA каждый элемент данных возвращается по значению в регистрах XMM0:XMM3 или YMM0:YMM3 в зависимости от размера элемента. Типы результатов, которые не помещаются в соответствующее регистры, возвращаются по ссылке на память, выделенную вызывающим объектом.

Стек поддерживается вызывающим объектом в реализации x64 `__vectorcall`. Код пролога и эпилога вызывающего объекта выделяет и очищает стек для вызываемой функции. Аргументы помещаются в стек в порядке справа налево, а для аргументов, передаваемых в регистрах, выделяется пространство теневого стека.

Примеры:

```
// crt_vc64.c
// Build for amd64 with: cl /arch:AVX /W3 /FAs crt_vc64.c
// This example creates an annotated assembly listing in
// crt_vc64.asm.

#include <intrin.h>
#include <xmmmintrin.h>

typedef struct {
    __m128 array[2];
} hva2;      // 2 element HVA type on __m128

typedef struct {
    __m256 array[4];
} hva4;      // 4 element HVA type on __m256

// Example 1: All vectors
// Passes a in XMM0, b in XMM1, c in YMM2, d in XMM3, e in YMM4.
// Return value in YMM0
```

```

// Return value in XMM0.
__m128 __vectorcall
example1(__m128 a, __m128 b, __m256 c, __m128 d, __m256 e) {
    return d;
}

// Example 2: Mixed int, float and vector parameters
// Passes a in RCX, b in XMM1, c in R8, d in XMM3, e in YMM4,
// f in XMM5, g pushed on stack.
// Return value in YMM0.
__m256 __vectorcall
example2(int a, __m128 b, int c, __m128 d, __m256 e, float f, int g) {
    return e;
}

// Example 3: Mixed int and HVA parameters
// Passes a in RCX, c in R8, d in R9, and e pushed on stack.
// Passes b by element in [XMM0:XMM1];
// b's stack shadow area is 8-bytes of undefined value.
// Return value in XMM0.
__m128 __vectorcall example3(int a, hva2 b, int c, int d, int e) {
    return b.array[0];
}

// Example 4: Discontiguous HVA
// Passes a in RCX, b in XMM1, d in XMM3, and e is pushed on stack.
// Passes c by element in [YMM0,YMM2,YMM4,YMM5], discontiguous because
// vector arguments b and d were allocated first.
// Shadow area for c is an 8-byte undefined value.
// Return value in XMM0.
float __vectorcall example4(int a, float b, hva4 c, __m128 d, int e) {
    return b;
}

// Example 5: Multiple HVA arguments
// Passes a in RCX, c in R8, e pushed on stack.
// Passes b in [XMM0:XMM1], d in [YMM2:YMM5], each with
// stack shadow areas of an 8-byte undefined value.
// Return value in RAX.
int __vectorcall example5(int a, hva2 b, int c, hva4 d, int e) {
    return c + e;
}

// Example 6: HVA argument passed by reference, returned by register
// Passes a in [XMM0:XMM1], b passed by reference in RDX, c in YMM2,
// d in [XMM3:XMM4].
// Register space was insufficient for b, but not for d.
// Return value in [YMM0:YMM3].
hva4 __vectorcall example6(hva2 a, hva4 b, __m256 c, hva2 d) {
    return b;
}

int __cdecl main( void )
{
    hva4 h4;
    hva2 h2;
    int i;
    float f;
    __m128 a, b, d;
    __m256 c, e;

    a = b = d = _mm_set1_ps(3.0f);
    c = e = _mm256_set1_ps(5.0f);
    h2.array[0] = _mm_set1_ps(6.0f);
    h4.array[0] = _mm256_set1_ps(7.0f);

    b = example1(a, b, c, d, e);
    e = example2(1, b, 3, d, e, 6.0f, 7);
    d = example3(1, h2, 3, 4, 5);
    c = example4(1, f, h4, d, e);
}

```

```
f = example4(1, 2.0f, h4, d, 5);
i = example5(1, h2, 3, h4, 5);
h4 = example6(h2, h4, c, h2);
}
```

Соглашение о вызовах __vectorcall для архитектуры x86

__vectorcall Соглашение о вызовах соответствует __fastcall соглашению для аргументов 32-разрядного целочисленного типа и использует преимущества регистров вектора SSE для векторного типа и аргументов Xva.

Первые два целочисленных аргумента, встреченные в списке параметров в порядке слева направо, помещаются в регистры ECX и EDX соответственно. Скрытый this указатель рассматривается как аргумент первого целочисленного типа и передается в ECX. Первые шесть аргументов векторного типа передаются по значению через векторные регистры SSE 0–5 в регистры XMM или YMM в зависимости от размера аргумента.

Первые шесть аргументов векторного типа в порядке слева направо передаются по значению в векторных регистрах SSE 0–5. Операции с плавающей запятой и __m128 типы передаются в регистрах XMM, а __m256 типы передаются в регистры ИММ. Для аргументов векторного типа, которые передаются через регистр, пространство теневого стека не выделяется. Аргументы векторного типа начиная с седьмой позиции в списке передаются в стек по ссылке на память, выделенную вызывающим объектом. Ограничение на ошибку компилятора C2719 не применяется к этим аргументам.

После выделения регистров для векторных аргументов элементы данных в аргументах Xva выделяются в порядке возрастания на неиспользуемые векторы XMM0 в XMM5 (или YMM0 в YMM5, для __m256 типов), если доступно достаточное количество регистров для всего Xva. Если регистров недостаточно, аргумент HVA передается в стек по ссылке на память, выделенную вызывающим объектом. Пространство теневого стека аргументу HVA не выделяется. Аргументы HVA назначаются регистрам в порядке слева направо в списке параметров и могут находиться в любой позиции.

Результаты __vectorcall функций возвращаются по значению в регистрах, если это возможно. Результаты целочисленных типов, включая structs и union размером 4 байт и менее, возвращаются по значению в EAX. Структуры или объединения целочисленного типа размером 8 байт или менее возвращаются по значению в регистрах EDX:EAX. Результаты векторного типа возвращаются по значению в XMM0 или YMM0 в зависимости от их размера. В результатах HVA каждый элемент данных возвращается по значению в регистрах XMM0:XMM3 или YMM0:YMM3 в зависимости от размера элемента. Результаты других типов возвращаются по ссылке на память, выделенную вызывающим объектом.

Реализация x86 __vectorcall соответствует соглашению о аргументах, передаваемых в стеке справа налево вызывающим объектом, а вызываемая функция очищает стек непосредственно перед возвратом. В стек передаются только те аргументы, которые не помещаются в регистры.

Примеры:

```
// crt_vc86.c
// Build for x86 with: cl /arch:AVX /W3 /FAs crt_vc86.c
// This example creates an annotated assembly listing in
// crt_vc86.asm.

#include <intrin.h>
#include <xmmmintrin.h>

typedef struct {
    __m128 array[2];
} hva2; // 2 element HVA type on __m128
```

```

typedef struct {
    __m256 array[4];
} hva4;      // 4 element HVA type on __m256

// Example 1: All vectors
// Passes a in XMM0, b in XMM1, c in YMM2, d in XMM3, e in YMM4.
// Return value in XMM0.
__m128 __vectorcall
example1(__m128 a, __m128 b, __m256 c, __m128 d, __m256 e) {
    return d;
}

// Example 2: Mixed int, float and vector parameters
// Passes a in ECX, b in XMM0, c in EDX, d in XMM1, e in YMM2,
// f in XMM3, g pushed on stack.
// Return value in YMM0.
__m256 __vectorcall
example2(int a, __m128 b, int c, __m128 d, __m256 e, float f, int g) {
    return e;
}

// Example 3: Mixed int and HVA parameters
// Passes a in ECX, c in EDX, d and e pushed on stack.
// Passes b by element in [XMM0:XMM1].
// Return value in XMM0.
__m128 __vectorcall example3(int a, hva2 b, int c, int d, int e) {
    return b.array[0];
}

// Example 4: HVA assigned after vector types
// Passes a in ECX, b in XMM0, d in XMM1, and e in EDX.
// Passes c by element in [YMM2:YMM5].
// Return value in XMM0.
float __vectorcall example4(int a, float b, hva4 c, __m128 d, int e) {
    return b;
}

// Example 5: Multiple HVA arguments
// Passes a in ECX, c in EDX, e pushed on stack.
// Passes b in [XMM0:XMM1], d in [YMM2:YMM5].
// Return value in EAX.
int __vectorcall example5(int a, hva2 b, int c, hva4 d, int e) {
    return c + e;
}

// Example 6: HVA argument passed by reference, returned by register
// Passes a in [XMM1:XMM2], b passed by reference in ECX, c in YMM0,
// d in [XMM3:XMM4].
// Register space was insufficient for b, but not for d.
// Return value in [YMM0:YMM3].
hva4 __vectorcall example6(hva2 a, hva4 b, __m256 c, hva2 d) {
    return b;
}

int __cdecl main( void )
{
    hva4 h4;
    hva2 h2;
    int i;
    float f;
    __m128 a, b, d;
    __m256 c, e;

    a = b = d = _mm_set1_ps(3.0f);
    c = e = _mm256_set1_ps(5.0f);
    h2.array[0] = _mm_set1_ps(6.0f);
    h4.array[0] = _mm256_set1_ps(7.0f);

    b = example1(a, b, c, d, e);
}

```

```
a = example1(a, b, c, d, e,);
e = example2(1, b, 3, d, e, 6.0f, 7);
d = example3(1, h2, 3, 4, 5);
f = example4(1, 2.0f, h4, d, 5);
i = example5(1, h2, 3, h4, 5);
h4 = example6(h2, h4, c, h2);
}
```

Завершение Microsoft для конкретных

См. также:

[Передача аргументов и соглашения об именовании](#)

[Ключевые слова](#)

Пример вызова. Прототип и вызов функции

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

В следующем примере показаны результаты вызова функции с использованием различных соглашений о вызовах.

Этот пример основан на следующей схеме функции. Замените `calltype` соответствующим соглашением о вызове.

```
void    calltype MyFunc( char c, short s, int i, double f );
.
.
.

void    MyFunc( char c, short s, int i, double f )
{
    .
    .
    .
}

MyFunc ('x', 12, 8192, 2.7183);
```

Дополнительные сведения см. в разделе [результаты вызова примера](#).

Завершение блока, относящегося только к системам Майкрософт

См. также

[Соглашения о вызовах](#)

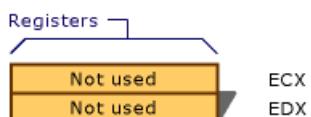
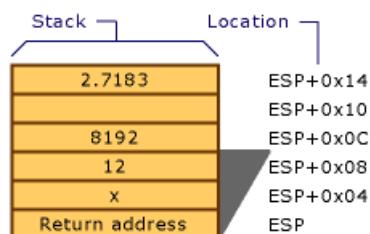
Пример результатов вызова

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

cdecl

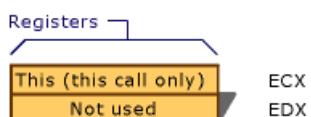
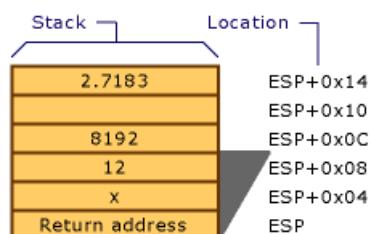
Имя функции с декорированием в C — `_MyFunc`.



`_cdecl` Соглашение о вызовах

stdcall и thiscall

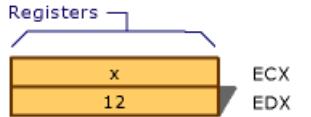
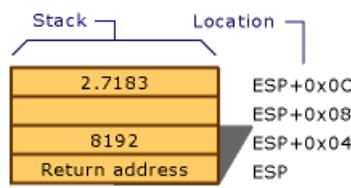
Декорированное имя C (`_stdcall`) имеет значение `_MyFunc@20`. Декорированное имя C++ зависит от конкретной реализации.



Соглашения о вызовах `stdcall` и `thiscall`

fastcall

Декорированное имя C (`_fastcall`) имеет значение `@MyFunc@20`. Декорированное имя C++ зависит от конкретной реализации.



Соглашение о вызовах __fastcall

Завершение блока, относящегося только к системам Майкрософт

См. также

[Пример вызова: прототип функции и вызов](#)

Вызовы функций Naked

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Функции, объявленные с `naked` атрибутом, создаются без кода пролога или эпилога, что позволяет создавать собственные последовательности пролога или эпилога с помощью [встроенного ассемблера](#).

Функции с атрибутом `naked` предоставляются как дополнительные функции. С их помощью можно объявить функцию, которая вызывается из другого контекста (и не C или C++), и тем самым указать другое место расположения параметров, в которых хранятся регистры. В качестве примера можно назвать такие процедуры, как обработчики прерываний. Эта возможность особенно полезна при написании драйверов виртуальных устройств (VxD).

Дополнительные сведения

- [naked](#)
- [Правила и ограничения для функций с атрибутом naked](#)
- [Рекомендации по написанию кода пролога или эпилога](#)

Завершение блока, относящегося только к системам Microsoft

См. также

[Соглашения о вызовах](#)

Правила и ограничения для функций с атрибутом naked

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Для функций с атрибутом naked действуют следующие правила и ограничения:

- `return` Инструкция не разрешена.
- Конструкции структурированной обработки исключений и обработки исключений C++ не допускаются, потому что они должны выполнять очистку в кадре стека.
- По этой же причине запрещена любая форма `setjmp`.
- Использование функции `_allocas` запрещено.
- Чтобы перед последовательностью пролога гарантировано не было никакого кода инициализации локальных переменных, инициализируемые локальные переменные в области функции не допускаются. В частности, в области функции не допускается объявление объектов C++. Однако допускаются инициализируемые данные во вложенной области.
- Оптимизация указателя кадра (параметр компилятора /Oу) не рекомендуется, но для функций с атрибутом naked она автоматически подавляется.
- Не допускается объявление объектов класса C++ в лексической области функции. Однако можно объявлять объекты во вложенном блоке.
- `naked` При компиляции с [параметром/CLR](#) ключевое слово игнорируется.
- Для функций `__fastcall` с атрибутом naked при наличии ссылки в коде C/C++ на один из аргументов регистра код пролога должен хранить значения этой регистрации в стеке для этой переменной.

Пример:

```
// nkdfastcl.cpp
// compile with: /c
// processor: x86
__declspec(naked) int __fastcall power(int i, int j) {
    // calculates i^j, assumes that j >= 0

    // prolog
    __asm {
        push ebp
        mov ebp, esp
        sub esp, __LOCAL_SIZE
        // store ECX and EDX into stack locations allocated for i and j
        mov i, ecx
        mov j, edx
    }

    {
        int k = 1;    // return value
        while (j-- > 0)
            k *= i;
        __asm {
            mov eax, k
        };
    }

    // epilog
    __asm {
        mov esp, ebp
        pop ebp
        ret
    }
}
```

Завершение блока, относящегося только к системам Майкрософт

См. также

[Вызовы функций с атрибутом naked](#)

Особенности написания кода пролога и эпилога

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Перед написанием собственных последовательностей кода пролога и эпилога важно понимать, как располагается кадр стека. Также полезно иметь представление об использовании `__LOCAL_SIZE` символа.

Макет кадра стека

В данном примере показан стандартный код пролога, который может присутствовать в 32-разрядной функции.

```
push    ebp          ; Save ebp
mov     ebp, esp      ; Set stack frame pointer
sub     esp, localbytes ; Allocate space for locals
push    <registers>   ; Save registers
```

Переменная `localbytes` представляет число байтов, которые требуются в стеке для локальных переменных, а переменная `<registers>` — это заполнитель, представляющий список сохраняемых в стеке регистров. После проталкивания регистров можно разместить в стеке все другие необходимые данные. Ниже приведен соответствующий код эпилога.

```
pop    <registers>   ; Restore registers
mov     esp, ebp      ; Restore stack pointer
pop     ebp          ; Restore ebp
ret                 ; Return from function
```

Стек всегда расширяется в направлении вниз (от старших адресов памяти к младшим). Указатель базы (`ebp`) указывает на помещенное в стек значение `ebp`. Область локальных переменных начинается с адреса `ebp-4`. Для доступа к локальным переменным необходимо вычислить смещение от `ebp` путем вычитания соответствующего значения из `ebp`.

`__LOCAL_SIZE`

Компилятор предоставляет символ, `__LOCAL_SIZE`, который используется во встроенном ассемблерном блоке кода пролога функции. Этот символ служит для выделения пространства локальным переменным в кадре стека пользовательского кода пролога.

Компилятор определяет значение `__LOCAL_SIZE`. Это значение представляет общее количество байтов всех определяемых пользователем локальных переменных и временных переменных, создаваемых компилятором. `__LOCAL_SIZE` может использоваться только в качестве немедленного операнда; его нельзя использовать в выражении. Значение этого символа не следует изменять и переопределять.
Пример:

```
mov     eax, __LOCAL_SIZE        ; Immediate operand--Okay
mov     eax, [ebp - __LOCAL_SIZE] ; Error
```

В следующем примере функции с атрибутом `naked`, содержащей пользовательские последовательности пролога и эпилога, используется `__LOCAL_SIZE` символ в последовательности пролога:

```
// the_local_size_symbol.cpp
// processor: x86
__declspec ( naked ) int main() {
    int i;
    int j;

    __asm {      /* prolog */
        push    ebp
        mov     ebp, esp
        sub     esp, __LOCAL_SIZE
    }

    /* Function body */
    __asm {      /* epilog */
        mov     esp, ebp
        pop    ebp
        ret
    }
}
```

Завершение блока, относящегося только к системам Майкрософт

См. также

[Вызовы функций с атрибутом naked](#)

Сопроцессор для вычислений с плавающей запятой и соглашения о вызовах

12.11.2021 • 2 minutes to read

При написании подпрограмм сборки для сопроцессора с плавающей запятой необходимо сохранить управляющее слово с плавающей запятой и очистить стек сопроцессоров, если не возвращать `float` значение или (которое функция должна возвращать в `St(0)`).

См. также

[Соглашения о вызовах](#)

Устаревшие соглашения о вызовах

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Соглашения о вызовах `__pascal`, `__fortran` и `__syscall` больше не поддерживаются. Их функциональные возможности можно эмулировать с помощью одного из поддерживаемых соглашений о вызовах и соответствующих параметров компоновщика.

`<windows.h>` Теперь поддерживает макрос WINAPI, который преобразуется в соответствующее соглашение о вызовах для целевого объекта. Используйте WINAPI, где ранее использовался стиль PASCAL или `__far __pascal`.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Передача аргументов и соглашения об именовании](#)

restrict (C++ AMP)

12.11.2021 • 2 minutes to read

Спецификатор ограничения может применяться к объявлениям функций и лямбда-выражений. Он реализует ограничения на код функции и на ее поведение в приложениях, в которых используется среда выполнения C++ AMP.

NOTE

Дополнительные сведения о `restrict` ключевом слове, которое является частью `__declspec` атрибутов класса хранения, см. в разделе [restrict](#).

`restrict` Предложение принимает следующие формы:

ПРЕДЛОЖЕНИЕ	ОПИСАНИЕ
<code>restrict(cpu)</code>	В функции могут использоваться все возможности языка C++. Вызывать ее могут только другие функции, объявленные с помощью функции <code>restrict(cpu)</code> .
<code>restrict(amp)</code>	В функции может использоваться только то подмножество языка C++, выполнение которого может ускорить C++ AMP.
Последовательность функций <code>restrict(cpu)</code> и <code>restrict(amp)</code> .	Функция должна соблюдать ограничения, устанавливаемые обеими функциями: и <code>restrict(cpu)</code> , и <code>restrict(amp)</code> . Вызывать ее могут функции, объявленные при помощи <code>restrict(cpu)</code> , <code>restrict(amp)</code> , <code>restrict(cpu, amp)</code> или <code>restrict(amp, cpu)</code> . Форма <code>restrict(A) restrict(B)</code> может быть записана в виде <code>restrict(A,B)</code> .

Remarks

`restrict` Ключевое слово является контекстным. Спецификаторы ограничения `cpu` и `amp` не являются зарезервированными словами. Список спецификаторов не может быть расширен. Функция, у которой нет предложения, совпадает с функцией `restrict`, имеющей `restrict(cpu)` предложение.

Для функции с предложением `restrict(amp)` действуют следующие ограничения:

- Функция может вызывать только функции с предложением `restrict(amp)`.
- Функция должна поддерживать подстановку.
- Функция может объявлять только `int` переменные, `unsigned int`, `float` и `double`, а также классы и структуры, содержащие только эти типы. `bool` параметр также разрешен, но при использовании его в составном типе он должен иметь тип, сопоставленный с 4 байтами.
- Лямбда-функции не могут захватывать по ссылке и не могут захватывать указатели.

- Указатели с одним косвенным обращением и ссылки поддерживаются только как локальные переменные, аргументы функций и типы возвращаемых значений.
- Следующие возможности не допускаются:
 - Рекурсия.
 - Переменные, объявленные с ключевым словом `volatile`.
 - Виртуальные функции.
 - Указатели на функции.
 - Указатели на функции-члены.
 - Указатели в структурах.
 - Указатели на указатели.
 - `goto` инструкции.
 - Операторы с метками.
 - `try` **** `catch` операторы, ИЛИ `throw`.
 - Глобальные переменные.
 - Статические переменные. Вместо этого используйте [ключевое слово tile_static](#).
 - `dynamic_cast` приведения.
 - `typeid` Оператор.
 - Объявления `asm`.
 - Использование `vararg`.

Описание ограничений функций см. в разделе [ограничения ограничения \(amp\)](#).

Пример

В следующем примере показано, как использовать `restrict(amp)` предложение.

```
void functionAmp() restrict(amp) {}
void functionNonAmp() {}

void callFunctions() restrict(amp)
{
    // int is allowed.
    int x;
    // long long int is not allowed in an amp-restricted function. This generates a compiler error.
    // long long int y;

    // Calling an amp-restricted function is allowed.
    functionAmp();

    // Calling a non-amp-restricted function is not allowed.
    // functionNonAmp();
}
```

См. также

[C++ AMP \(C++ Accelerated Massive Parallelism\)](#)

Ключевое слово tile_static

12.11.2021 • 3 minutes to read

Ключевое слово `tile_static` используется для объявления переменной, к которой могут обращаться все потоки в плитке потоков. Время существования переменной начинается, когда при выполнении достигается точка объявления, и завершается при возврате из функции ядра. Дополнительные сведения об использовании плиток см. в разделе [использование плиток](#).

Ключевое слово `tile_static` имеет следующие ограничения.

- Его можно использовать только для переменных, находящихся в функции с модификатором `restrict(amp)`.
- Не допускается его использование для переменных, являющихся указателями или ссылочными типами.
- Переменная `tile_static` не может иметь инициализатор. Конструкторы и деструкторы по умолчанию не вызываются автоматически.
- Значение неинициализированной `tile_static` переменной не определено.
- Если переменная `tile_static` объявлена в графе вызовов, которая является корневой для вызова без мозаичного заполнения `parallel_for_each`, то создается предупреждение и поведение переменной не определено.

Пример

В следующем примере показано, как можно использовать переменную `tile_static` для накопления данных по нескольким потокам в плитке.

```
// Sample data:  
int sampledata[] = {  
    2, 2, 9, 7, 1, 4,  
    4, 4, 8, 8, 3, 4,  
    1, 5, 1, 2, 5, 2,  
    6, 8, 3, 2, 7, 2};  
  
// The tiles:  
// 2 2      9 7      1 4  
// 4 4      8 8      3 4  
//  
// 1 5      1 2      5 2  
// 6 8      3 2      7 2  
  
// Averages:  
int averagedata[] = {  
    0, 0, 0, 0, 0,  
    0, 0, 0, 0, 0,  
    0, 0, 0, 0, 0,  
    0, 0, 0, 0, 0,  
};  
  
array_view<int, 2> sample(4, 6, sampledata);  
array_view<int, 2> average(4, 6, averagedata);  
  
parallel_for_each(  
    // Create threads for sample.extent and divide the extent into 2 x 2 tiles.  
    sample.extent.tile<2,2>(),
```

```

[=](tiled_index<2,2> idx) restrict(amp)
{
    // Create a 2 x 2 array to hold the values in this tile.
    tile_static int nums[2][2];
    // Copy the values for the tile into the 2 x 2 array.
    nums[idx.local[1]][idx.local[0]] = sample[idx.global];
    // When all the threads have executed and the 2 x 2 array is complete, find the average.
    idx.barrier.wait();
    int sum = nums[0][0] + nums[0][1] + nums[1][0] + nums[1][1];
    // Copy the average into the array_view.
    average[idx.global] = sum / 4;
}
);

for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 6; j++) {
        std::cout << average(i,j) << " ";
    }
    std::cout << "\n";
}

// Output:
// 3 3 8 8 3 3
// 3 3 8 8 3 3
// 5 5 2 2 4 4
// 5 5 2 2 4 4
// Sample data.
int sampledata[] = {
    2, 2, 9, 7, 1, 4,
    4, 4, 8, 8, 3, 4,
    1, 5, 1, 2, 5, 2,
    6, 8, 3, 2, 7, 2};

// The tiles are:
// 2 2      9 7      1 4
// 4 4      8 8      3 4
//
// 1 5      1 2      5 2
// 6 8      3 2      7 2

// Averages.
int averagedata[] = {
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0,
};

array_view<int, 2> sample(4, 6, sampledata);
array_view<int, 2> average(4, 6, averagedata);

parallel_for_each(
    // Create threads for sample.grid and divide the grid into 2 x 2 tiles.
    sample.extent.tile<2,2>(),
    [=](tiled_index<2,2> idx) restrict(amp)
    {
        // Create a 2 x 2 array to hold the values in this tile.
        tile_static int nums[2][2];
        // Copy the values for the tile into the 2 x 2 array.
        nums[idx.local[1]][idx.local[0]] = sample[idx.global];
        // When all the threads have executed and the 2 x 2 array is complete, find the average.
        idx.barrier.wait();
        int sum = nums[0][0] + nums[0][1] + nums[1][0] + nums[1][1];
        // Copy the average into the array_view.
        average[idx.global] = sum / 4;
    }
);

for (int i = 0; i < 4; i++) {

```

```
    for (int j = 0; j < 6; j++) {
        std::cout << average(i,j) << " ";
    }
    std::cout << "\n";
}

// Output.
// 3 3 8 8 3 3
// 3 3 8 8 3 3
// 5 5 2 2 4 4
// 5 5 2 2 4 4
```

См. также раздел

[Модификаторы, специфичные для Майкрософт](#)

[C++ AMP Средств](#)

[Функция parallel_for_each \(C++ AMP\)](#)

[Пошаговое руководство. умножение матрицы](#)

Блок, относящийся только к системам Microsoft

В синтаксисе расширенных атрибутов для указания сведений о классе хранения используется `__declspec` ключевое слово, которое указывает, что экземпляр заданного типа должен храниться в указанном ниже атрибуте класса хранения, связанном с Microsoft. Примеры других модификаторов класса хранения включают ключевые слова `static` и `extern`. Однако эти ключевые слова — часть спецификации ANSI языков C и C++, и как таковые они не описаны расширенным синтаксисом атрибутов. Расширенный синтаксис атрибутов упрощает и стандартизирует расширения для систем Microsoft в соответствии с правилами языков C и C++.

Грамматика

```
decl-specifier :  
    __declspec ( extended-decl-modifier-seq )  
  
extended-decl-modifier-seq :  
    extended-decl-modifier необ.  
    extended-decl-modifier extended-decl-modifier-seq  
  
extended-decl-modifier :  
    align( ЧИСЛО )  
    allocate(" СЕГНАМЕ ")  
    allocator  
    appdomain  
    code_seg(" СЕГНАМЕ ")  
    deprecated  
    dllimport  
    dllexport  
    jitintrinsic  
    naked  
    noalias  
    noinline  
    noreturn  
    noexcept  
    novtable  
    no_SANITIZE_ADDRESS  
    process  
    property( { get= Get-Func-Name | ,put= Вставить-Func-Name } )  
    restrict  
    safebuffers  
    selectany  
    spectre(nomitigation)  
    thread  
    uuid(" КОМОБЖЕКТГУИД ")
```

Пробел отделяет последовательность модификаторов объявления. Примеры приведены в дальнейших

разделах.

Грамматика расширенных атрибутов поддерживает следующие атрибуты классов хранения, предназначенные для Майкрософт: `align` `allocate` `allocator` `appdomain` `code_seg` `deprecated` `dllexport` `dllimport` `jitintrinsic` `naked` `noalias` `noinline` `noreturn` `nothrow` `novtable` `no-sanitize_address` `process` `restrict` `safebuffers` `selectany` `spectre` И `thread`. Он также поддерживает следующие атрибуты COM-Object: `property` И `uuid`.

`code_seg` `dllexport` `dllimport` Атрибуты класса хранения `naked` `noalias` `nothrow`, `no-sanitize_address`, `property`, `restrict`, `selectany`, `thread` И `uuid` являются свойствами только объявления объекта или функции, к которым они применяются. `thread` Атрибут влияет только на данные и объекты. `naked` Атрибуты И `spectre` влияют только на функции. `dllimport` Атрибуты и `dllexport` влияют на функции, данные и объекты. `property` Атрибуты, `selectany` И `uuid` влияют на COM-объекты.

Для совместимости с предыдущими версиями аргумент `_declspec` является синонимом, `__declspec` если только параметр компилятора [/Za](#) не (отключил расширения языка).

`__declspec` Ключевые слова должны располагаться в начале простого объявления. Компилятор игнорирует, без предупреждения, все `__declspec` Ключевые слова, помещенные после * или &, а также перед идентификатором переменной в объявлении.

`__declspec` Атрибут, указанный в начале объявления определяемого пользователем типа, применяется к переменной этого типа. Пример:

```
__declspec(dllexport) class X {} varX;
```

В этом случае атрибут применяется к `varX`. `__declspec` Атрибут, помещенный после `class` `struct` ключевого слова или, применяется к определяемому пользователем типу. Пример:

```
class __declspec(dllexport) X {};
```

В этом случае атрибут применяется к `X`.

Ниже приведены общие рекомендации по использованию `__declspec` атрибута для простых объявлений.

описателе-описатель-seq init-декларатор-List,

Описатель `decl-seq` должен содержать, помимо прочего, базовый тип (например, `int` `float` `typedef` или имя класса), класс хранения (например `static`, `extern`) или `__declspec` расширение. Элемент `init-declarator-List` должен содержать, помимо прочего, указательную часть объявлений. Пример:

```
__declspec(selectany) int * pi1 = 0; //Recommended, selectany & int both part of decl-specifier
int __declspec(selectany) * pi2 = 0; //OK, selectany & int both part of decl-specifier
int * __declspec(selectany) pi3 = 0; //ERROR, selectany is not part of a declarator
```

В следующем примере кода объявлена целочисленная локальная переменная потока и инициализирована с использованием следующего значения:

```
// Example of the __declspec keyword
__declspec( thread ) int tls_i = 1;
```

Завершение блока, относящегося только к системам Майкрософт

См. также

[Словами](#)

[Расширенные атрибуты классов хранения в C](#)

align (C++)

12.11.2021 • 7 minutes to read

в Visual Studio 2015 и более поздних версиях используйте стандартный описатель C++ 11 `alignas` для управления выравниванием. Дополнительные сведения см. в разделе [Выравнивание](#).

Блок, относящийся только к системам Microsoft

Используйте `__declspec(align(#))` для точного управления выравниванием пользовательских данных (например, статическими распределениями или автоматическими данными в функции).

Синтаксис

Декларатор `__declspec (выровняйте (#))`

Remarks

Написание приложений, использующих последние инструкции процессора, связано с некоторыми новыми ограничениями и проблемами. Для многих новых инструкций требуются данные, которые выводятся по 16-байтным границам. Кроме того, путем согласования часто используемых данных с размером строки кэша процессора повышается производительность кэша. Например, если определена структура, размер которой меньше 32 байт, то может потребоваться 32 байта, чтобы обеспечить эффективное кэширование объектов этого типа структуры.

значение выравнивания. Допустимые записи — целые степени двух значений от 1 до 8192 (байты), например 2, 4, 8, 16, 32 или 64. `declarator` — Это данные, которые объявляются как согласованные.

Сведения о том, как вернуть значение типа `size_t`, которое является требованием выравнивания для типа, см. в разделе [alignof](#). Сведения об объявлении несогласованных указателей при нацеливании на 64-разрядные процессоры см. в разделе [unaligned](#).

Можно использовать `__declspec(align(#))` при определении `struct`, `union` или `class`, или при объявлении переменной.

Компилятор не гарантирует или не пытается сохранить атрибут выравнивания данных во время операции копирования или преобразования данных. Например, `memcpy` может скопировать структуру, объявленную с `__declspec(align(#))`, в любое расположение. Обычные распределители (например, `malloc` C++ `operator new` и распределителя Win32) обычно возвращают память, которая не полностью соответствует `__declspec(align(#))` структурам или массивам структур. Чтобы гарантировать, что назначение копирования или операции преобразования данных будет правильно согласовано, используйте `_aligned_malloc`. Или напишите собственный распределитель.

Нельзя указать выравнивание для параметров функции. При передаче данных с атрибутом выравнивания по значению в стеке его выравнивание регулируется соглашением о вызовах. Если в вызываемой функции важно выравнивание данных, скопируйте параметр в правильно выровненную память перед использованием.

Без этого `__declspec(align(#))` компилятор обычно выровняет данные по естественным границам, исходя из целевого процессора и размера данных, до 4-байтовых границ на 32-разрядных процессорах и 8-байтовых границ на 64-разрядных процессорах. Данные в классах или структурах выравниваются в классе или структуре на минимальном уровне естественного выравнивания и текущем параметре

упаковки (от `#pragma pack` или `/Zp` параметра компилятора).

В этом примере демонстрируется использование `__declspec(align(#))`:

```
__declspec(align(32)) struct Str1{
    int a, b, c, d, e;
};
```

Теперь этот тип содержит 32-разрядный атрибут выравнивания. Это означает, что все статические и автоматические экземпляры начинаются на границе 32 байта. Дополнительные типы структуры, объявленные с этим типом в качестве члена, сохраняют атрибут выравнивания этого типа, то есть любая структура с `Str1` элементом имеет атрибут выравнивания не менее 32.

Здесь, `sizeof(struct Str1)` равно 32. Это означает, что если создается массив `Str1` объектов, а основание массива составляет 32 байт, то каждый элемент массива также имеет значение, равное 32 байт. Чтобы создать массив, базовый объект которого правильно соответствует динамической памяти, используйте `_aligned_malloc`. Или напишите собственный распределитель.

`sizeof` Значением для любой структуры является смещение последнего элемента, а также его размер, округленный до ближайшего числа, кратного наибольшему значению выравнивания элемента или целому значению выравнивания структуры, в зависимости от того, какое значение больше.

Компилятор использует эти правила для выравнивания структуры:

- Если выравнивание не переопределяется с помощью `__declspec(align(#))`, выравнивание скалярного члена структуры — это его минимальный размер и текущая упаковка.
- Если выравнивание не переопределяется с помощью `__declspec(align(#))`, выравнивание структуры — это максимальное число отдельных выравниваний членов.
- Элемент структуры помещается со смещением от начала его родительской структуры, которая является наименьшей, кратным значению смещения конца предыдущего элемента.
- Размер структуры — это наименьшее число, кратное ее выравниванию, больше или равное смещению в конце его последнего члена.

`__declspec(align(#))` может только увеличить ограничения выравнивания.

Дополнительные сведения см. в разделе:

- [align Примеров](#)
- [Определение новых типов с помощью `__declspec\(align\(#\)\)`](#)
- [Выравнивание данных в локальном хранилище потока](#)
- [Принцип `align` работы с упаковкой данных](#)
- [Примеры выравнивания структуры](#) (только для x64)

Примеры выровняйте

В следующих примерах показано, как `__declspec(align(#))` влияет на размер и выравнивание структур данных. В примерах допускаются следующие определения.

```
#define CACHE_LINE 32
#define CACHE_ALIGN __declspec(align(CACHE_LINE))
```

В этом примере структура `s1` определена с помощью `__declspec(align(32))`. Все случаи использования `s1` для определения переменных или в объявлении других типов выравниваются по 32 байтам. `sizeof(struct s1)` возвращает значение 32, а `s1` имеет 16 байтов заполнения после 16 байтов, необходимых для удержания четырех целых чисел. Каждый `int` элемент требует 4-байтового выравнивания, но выравнивание самой структуры объявляется как 32. Затем общее выравнивание составляет 32.

```
struct CACHE_ALIGN S1 { // cache align all instances of S1
    int a, b, c, d;
};

struct S1 s1; // s1 is 32-byte cache aligned
```

В этом примере `sizeof(struct s2)` возвращает 16. Это сумма размеров членов, поскольку это число является кратным наибольшему требуемому выравниванию (кратное 8).

```
__declspec(align(8)) struct S2 {
    int a, b, c, d;
};
```

В следующем примере `sizeof(struct s3)` возвращает 64.

```
struct S3 {
    struct S1 s1; // S3 inherits cache alignment requirement
                   // from S1 declaration
    int a;         // a is now cache aligned because of s1
                   // 28 bytes of trailing padding
};
```

Обратите внимание, что в этом примере `a` выравнивается по своему естественному типу, то есть, в данном случае, по 4 байтам. Однако значение `s1` должно быть выровнено по 32-байтовой границе. Далее следует 28 байтов заполнения `a`, чтобы оно `s1` начиналось со смещения 32. `s4` затем наследует требование выравнивания `s1`, поскольку оно является самым большим требованием к выравниванию в структуре. `sizeof(struct s4)` возвращает 64.

```
struct S4 {
    int a;
    // 28 bytes padding
    struct S1 s1; // S4 inherits cache alignment requirement of S1
};
```

Следующие три объявления переменных также используют `__declspec(align(#))`. В каждом случае переменная должна быть выровнена по 32 байтам. В массиве базовый адрес массива, а не каждый элемент массива составляет 32 байт. `sizeof` Значение для каждого элемента массива не изменяется при использовании `__declspec(align(#))`.

```
CACHE_ALIGN int i;
CACHE_ALIGN int array[128];
CACHE_ALIGN struct s2 s;
```

Для выравнивания каждого члена массива нужно использовать следующий код:

```
typedef CACHE_ALIGN struct { int a; } S5;
S5 array[10];
```

Обратите внимание, что в этом примере выравнивание самой структуры и первого элемента работают одинаково:

```
CACHE_ALIGN struct S6 {
    int a;
    int b;
};

struct S7 {
    CACHE_ALIGN int a;
    int b;
};
```

S6 и S7 имеют одинаковые характеристики выравнивания, выделения и размера.

В этом примере выравнивание начальных адресов a, b, c и d — соответственно 4, 1, 4, и 1.

```
void fn() {
    int a;
    char b;
    long c;
    char d[10]
}
```

Выравнивание при выделении памяти в куче зависит от того, какая функция выделения вызвана.

Например, если используется `malloc`, результат зависит от размера операнда. Если аргумент `arg >= 8`, возвращается память размером 8 байт. Если аргумент `arg < 8`, то при выравнивании возвращаемой памяти первая степень числа 2 меньше аргумента будет первой. Например, если используется `malloc(7)`, выравнивание составляет 4 байта.

Определение новых типов с помощью `__declspec(align(#))`

Можно определить тип с характеристикой выравнивания.

Например, можно определить `struct` с помощью значения выравнивания следующим образом:

```
struct aType {int a; int b;};
typedef __declspec(align(32)) struct aType bType;
```

Теперь `aType` и `bType` имеют одинаковый размер (8 байт), но переменные типа `bType` — 32 байт.

Выровняйте данные в локальном хранилище потока

Статическое локальное хранилище потока (TLS), созданное с помощью атрибута `__declspec(thread)` и помещенное в раздел TLS образа, обеспечивает выравнивание так же, как стандартные статические данные. Для создания данных TLS операционная система выделяет память в размере раздела TLS и сохраняет атрибут выравнивания раздела TLS.

В этом примере показаны различные способы помещения выровненных данных в локальное хранилище потока.

```

// put an aligned integer in TLS
__declspec(thread) __declspec(align(32)) int a;

// define an aligned structure and put a variable of the struct type
// into TLS
__declspec(thread) __declspec(align(32)) struct F1 { int a; int b; } a;

// create an aligned structure
struct CACHE_ALIGN S9 {
    int a;
    int b;
};

// put a variable of the structure type into TLS
__declspec(thread) struct S9 a;

```

Принцип `align` работы с упаковкой данных

`/zp` Параметр компилятора и `pack` директива pragma имеют воздействие на данные упаковки для членов структуры и объединения. В этом примере показано `/zp`, как и `__declspec(align(#))` совместно работать:

```

struct S {
    char a;
    short b;
    double c;
    CACHE_ALIGN double d;
    char e;
    double f;
};

```

В следующей таблице перечислены смещения каждого элемента в различных `/zp` значениях (или `#pragma pack`), показывающие, как эти два взаимодействуют.

ПЕРЕМЕННАЯ	<code>/ZP1</code>	<code>/ZP2</code>	<code>/ZP4</code>	<code>/ZP8</code>
a	0	0	0	0
b	1	2	2	2
c	3	4	4	8
d	32	32	32	32
й	40	40	40	40
f	41	42	44	48
<code>sizeof(S)</code>	64	64	64	64

Дополнительные сведения см. в разделе [/zp \(выравнивание членов структуры\)](#).

Смещение объекта зависит от смещения предыдущего объекта и от текущего параметра упаковки, если у объекта нет атрибута `__declspec(align(#))`. Если же такой атрибут есть, то выравнивание зависит от смещения предыдущего объекта и от значения `__declspec(align(#))` объекта.

Завершение блока, относящегося только к системам Microsoft

См. также

[__declspec](#)

[Обзор соглашений ABI ARM](#)

[Программные соглашения для 64-разрядных систем](#)

allocate

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

`allocate` Описатель объявления именует сегмент данных, в котором будет выделен элемент данных.

Синтаксис

```
** __declspec(allocate("**сегнаме )) декларатор
```

Remarks

Имя *сегнаме* должно быть объявлено с помощью одной из следующих директив pragma:

- [code_seg](#)
- [const_seg](#)
- [data_seg](#)
- [init_seg](#)
- [раздела](#)

Пример

```
// allocate.cpp
#pragma section("mycode", read)
__declspec(allocate("mycode")) int i = 0;

int main() {
```

Завершение блока, относящегося только к системам Майкрософт

См. также

[__declspec](#)

[Ключевые слова](#)

allocator

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

`allocator` спецификатор объявления можно применить к пользовательским функциям выделения памяти, чтобы сделать выделение видимым с помощью трассировки событий для Windows (ETW).

Синтаксис

```
__declspec(allocator)
```

Remarks

собственный профилировщик памяти в Visual Studio работает путем сбора данных событий ETW выделения, созданных во время выполнения. Распределители в CRT и пакете Windows SDK аннотированы на уровне исходного кода, что позволяет регистрировать их данные выделения. Если вы создаете собственные распределители, то любые функции, возвращающие указатель на только что выделенную память кучи, могут быть дополнены с помощью `__declspec(allocator)`, как показано в следующем примере для myMalloc:

```
__declspec(allocator) void* myMalloc(size_t size)
```

дополнительные сведения см. [в разделе измерение использования памяти в Visual Studio](#) и [пользовательские события кучи ETW для машинного кода](#).

Завершение блока, относящегося только к системам Майкрософт

appdomain

12.11.2021 • 2 minutes to read

Указывает, что каждый домен приложения в вашем управляемого приложения должен иметь собственную копию заданной глобальной переменной или статической переменной-члена.

Дополнительные сведения см. в разделе [домены приложений и Visual C++](#).

В каждом домене приложения имеется собственная копия переменной, создаваемой на уровне домена приложения. Конструктор этой переменной выполняется при загрузке сборки в домен приложения, а деструктор — при выгрузке домена приложения.

Если вам необходимо, чтобы глобальная переменная использовалась всеми доменами приложения в рамках процесса CLR, используйте модификатор `__declspec(process)`. `__declspec(process)` действует по умолчанию в [параметре/CLR](#). параметры компилятора /clr: pure и /clr: **сейф** являются устаревшими в Visual Studio 2015 и не поддерживаются в Visual Studio 2017.

`__declspec(appdomain)` параметр допустим только в том случае, если используется один из параметров компилятора /CLR . Модификатором `__declspec(appdomain)` можно пометить только глобальную переменную, статическую переменную-член или статическую локальную переменную. Применять модификатор `__declspec(appdomain)` к статическим членам управляемых типов будет ошибкой, поскольку они в любом случае имеют соответствующее поведение.

использование `__declspec(appdomain)` аналогично использованию [локальной службы хранилища потока \(TLS\)](#). Как и у доменов приложения, у потоков имеются собственные области памяти. Если используется ключевое слово `__declspec(appdomain)`, то для глобальной переменной гарантированно создается собственная область памяти в каждом домене приложения, созданном для данного приложения.

Существует ряд ограничений на смешение использования для каждого процесса и переменных AppDomain. Дополнительные сведения см. в разделе [процесс](#).

Например, при запуске программы сначала инициализируются все переменные на уровне процесса, а затем все переменные на уровне домена приложения. Таким образом, инициализация переменных на уровне процесса не может зависеть от значений переменных на уровне домена приложения.

Одновременно использовать (присваивать) переменные на уровне процесса и домена приложения не рекомендуется.

Сведения о вызове функции в конкретном домене приложения см. в разделе [Call_in_appdomain Function](#).

Пример

```
// declspec_appdomain.cpp
// compile with: /clr
#include <stdio.h>
using namespace System;

class CGlobal {
public:
    CGlobal(bool bProcess) {
        Counter = 10;
        m_bProcess = bProcess;
        Console::WriteLine("__declspec({0}) CGlobal::CGlobal constructor", m_bProcess ? (String^)"process" :
(String^)"appdomain");
    }
}
```

```

~CrossDomain() {
    Console::WriteLine("__declspec({0}) CGlobal::~CGlobal destructor", m_bProcess ? (String^)"process" :
(String^)"appdomain");
}

int Counter;

private:
    bool m_bProcess;
};

__declspec(process) CGlobal process_global = CGlobal(true);
__declspec(appdomain) CGlobal appdomain_global = CGlobal(false);

value class Functions {
public:
    static void change() {
        ++appdomain_global.Counter;
    }

    static void display() {
        Console::WriteLine("process_global value in appdomain '{0}': {1}",
                           AppDomain::CurrentDomain->FriendlyName,
                           process_global.Counter);

        Console::WriteLine("appdomain_global value in appdomain '{0}': {1}",
                           AppDomain::CurrentDomain->FriendlyName,
                           appdomain_global.Counter);
    }
};

int main() {
    AppDomain^ defaultDomain = AppDomain::CurrentDomain;
    AppDomain^ domain = AppDomain::CreateDomain("Domain 1");
    AppDomain^ domain2 = AppDomain::CreateDomain("Domain 2");
    CrossAppDomainDelegate^ changeDelegate = gcnew CrossAppDomainDelegate(&Functions::change);
    CrossAppDomainDelegate^ displayDelegate = gcnew CrossAppDomainDelegate(&Functions::display);

    // Print the initial values of appdomain_global in all appdomains.
    Console::WriteLine("Initial value");
    defaultDomain->DoCallBack(displayDelegate);
    domain->DoCallBack(displayDelegate);
    domain2->DoCallBack(displayDelegate);

    // Changing the value of appdomain_global in the domain and domain2
    // appdomain_global value in "default" appdomain remain unchanged
    process_global.Counter = 20;
    domain->DoCallBack(changeDelegate);
    domain2->DoCallBack(changeDelegate);
    domain2->DoCallBack(changeDelegate);

    // Print values again
    Console::WriteLine("Changed value");
    defaultDomain->DoCallBack(displayDelegate);
    domain->DoCallBack(displayDelegate);
    domain2->DoCallBack(displayDelegate);

    AppDomain::Unload(domain);
    AppDomain::Unload(domain2);
}

```

```
__declspec(process) CGlobal::CGlobal constructor
__declspec(appdomain) CGlobal::CGlobal constructor
Initial value
process_global value in appdomain 'declspec_appdomain.exe': 10
appdomain_global value in appdomain 'declspec_appdomain.exe': 10
__declspec(appdomain) CGlobal::CGlobal constructor
process_global value in appdomain 'Domain 1': 10
appdomain_global value in appdomain 'Domain 1': 10
__declspec(appdomain) CGlobal::CGlobal constructor
process_global value in appdomain 'Domain 2': 10
appdomain_global value in appdomain 'Domain 2': 10
Changed value
process_global value in appdomain 'declspec_appdomain.exe': 20
appdomain_global value in appdomain 'declspec_appdomain.exe': 10
process_global value in appdomain 'Domain 1': 20
appdomain_global value in appdomain 'Domain 1': 11
process_global value in appdomain 'Domain 2': 20
appdomain_global value in appdomain 'Domain 2': 12
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(appdomain) CGlobal::~CGlobal destructor
__declspec(process) CGlobal::~CGlobal destructor
```

См. также

[__declspec](#)

[Ключевые слова](#)

code_seg (_declspec)

12.11.2021 • 3 minutes to read

Блок, относящийся только к системам Microsoft

Атрибут объявления `code_seg` именует фрагмент текста исполняемого файла в OBJ-файле, в котором будут храниться объектный код для функций-членов функции или класса.

Синтаксис

```
_declspec(code_seg("segname")) declarator
```

Remarks

Атрибут `_declspec(code_seg(...))` позволяет помещать код в различные именованные сегменты, которые можно выгружать или блокировать в памяти по отдельности. Можно использовать этот атрибут для управления размещением шаблонов с созданными экземплярами и кода, созданного компилятором.

Сегмент — это именованный блок данных в OBJ-файле, который загружается в память как единое целое. *Сегмент текста* — это сегмент, содержащий исполняемый код. Термин *раздел* часто используется с сегментом.

Объектный код, создаваемый при определении `declarator`, помещается в сегмент текста, указанный `segname`, а это узкий строковый литерал. Имя `segname` не обязательно должно быть указано в директиве `pragma`, прежде чем его можно будет использовать в объявлении. По умолчанию если значение `code_seg` не указано, объектный код помещается в сегмент с именем `.text`. Атрибут `code_seg` переопределяет любую существующую директиву `#pragma code_seg`. Атрибут `code_seg`, применяемый к функции-члену, переопределяет любой атрибут `code_seg`, примененный к включающему классу.

Если сущность имеет атрибут `code_seg`, все объявления и определения одной и той же сущности должны иметь одинаковые `code_seg` атрибуты. Если базовый класс имеет атрибут `code_seg`, то производные классы должны иметь один и тот же атрибут.

Когда к функции-члену применяется атрибут `code_seg`, объектный код для этой функции помещается в указанный сегмент текста. Если этот атрибут применяется к классу, все функции-члены класса и вложенные классы, включая созданные компилятором специальные функции члены, помещаются в указанный сегмент. Локально определенные классы, например классы, определенные в теле функции-члена, не наследуют атрибут `code_seg` включающей области.

При применении атрибута `code_seg` к классу шаблона или функции шаблона все неявные специализации шаблона помещаются в указанный сегмент. Явные или частичные специализации не наследуют атрибут `code_seg` из основного шаблона. Вы можете указать тот же или другой атрибут `code_seg` в специализации. Атрибут `code_seg` нельзя применить к явному созданию экземпляра шаблона.

По умолчанию созданный компилятором, такой как специальная функция-член, помещается в сегмент `.text`. Директива `#pragma code_seg` не переопределяет это значение по умолчанию. Используйте атрибут `code_seg` в классе, шаблоне класса или шаблоне функции, чтобы управлять размещением кода, сгенерированного компилятором.

Лямбда-выражения наследуют `code_seg` атрибуты из их включающей области. Чтобы указать сегмент

для лямбда-выражения, примените атрибут `code_seg` после предложения объявления параметра и перед любыми переменными или спецификациями исключений, а также любыми конечными спецификациями возвращаемого типа и телом лямбда-выражения. Дополнительные сведения см. в разделе [синтаксис лямбда-выражений](#). В этом примере определяется лямбда-выражение в сегменте с именем `PagedMem`:

```
auto Sqr = [](int t) __declspec(code_seg("PagedMem")) -> int { return t*t; };
```

Будьте внимательны при помещении определенных функций-членов, особенно виртуальных функций-членов, в разные сегменты. При определении виртуальной функции в производном классе, находящемся в выгруженном сегменте, когда метод базового класса находится в невыгруженном в сегменте, другие методы базового класса или пользовательский код могут предположить, что вызов виртуального метода не активирует ошибку выгрузки.

Пример

В этом примере показано, как атрибут `code_seg` управляет размещением сегментов при использовании неявной и явной специализации шаблона:

```
// code_seg.cpp
// Compile: cl /EHsc /W4 code_seg.cpp

// Base template places object code in Segment_1 segment
template<class T>
class __declspec(code_seg("Segment_1")) Example
{
public:
    virtual void VirtualMemberFunction(T /*arg*/) {}

};

// bool specialization places code in default .text segment
template<>
class Example<bool>
{
public:
    virtual void VirtualMemberFunction(bool /*arg*/) {}

};

// int specialization places code in Segment_2 segment
template<>
class __declspec(code_seg("Segment_2")) Example<int>
{
public:
    virtual void VirtualMemberFunction(int /*arg*/) {}

};

// Compiler warns and ignores __declspec(code_seg("Segment_3"))
// in this explicit specialization
__declspec(code_seg("Segment_3")) Example<short>; // C4071

int main()
{
    // implicit double specialization uses base template's
    // __declspec(code_seg("Segment_1")) to place object code
    Example<double> doubleExample{};
    doubleExample.VirtualMemberFunction(3.14L);

    // bool specialization places object code in default .text segment
    Example<bool> boolExample{};
    boolExample.VirtualMemberFunction(true);

    // int specialization uses __declspec(code_seg("Segment_2"))
    // to place object code
    Example<int> intExample{};
    intExample.VirtualMemberFunction(42);
}
```

Завершение блока, относящегося только к системам Microsoft

См. также

[__declspec](#)

[Ключевые слова](#)

deprecated (C++)

12.11.2021 • 2 minutes to read

В этом разделе рассматриваются устаревшие объявления `declspec`, относящиеся к Microsoft. Сведения об атрибуте C++ 14 `[[deprecated]]` и рекомендации по использованию этого атрибута и `declspec` или директивы `pragma` Microsoft см. в разделе [стандартные атрибуты C++](#).

В приведенных ниже исключениях `deprecated` объявление предоставляет те же функциональные возможности, что и [нерекомендуемая](#) директива `pragma`:

- `deprecated` Объявление позволяет указать определенные формы перегрузок функций как устаревшие, в то время как форма `pragma` применяется ко всем перегруженным формам имени функции.
- `deprecated` Объявление позволяет указать сообщение, которое будет отображаться во время компиляции. Текст сообщения может быть взят из макроса.
- Макросы могут быть помечены как нерекомендуемые с помощью `deprecated` директивы `pragma`.

Если компилятор обнаруживает использование устаревшего идентификатора или стандартного `[[deprecated]]` атрибута, выдается предупреждение [C4996](#).

Примеры

В следующем примере показано, как отметить функции как нерекомендуемые и как указать сообщение, которое будет отображаться во время компиляции, если будет использоваться нерекомендуемая функция.

```
// deprecated.cpp
// compile with: /W3
#define MY_TEXT "function is deprecated"
void func1(void) {}
__declspec(deprecated) void func1(int) {}
__declspec(deprecated("** this is a deprecated function **")) void func2(int) {}
__declspec(deprecated(MY_TEXT)) void func3(int) {}

int main() {
    func1();
    func1(1); // C4996
    func2(1); // C4996
    func3(1); // C4996
}
```

В следующем примере показано, как отметить классы как нерекомендуемые и как указать сообщение, которое будет отображаться во время компиляции, если будет использоваться нерекомендуемый класс.

```
// deprecate_class.cpp
// compile with: /W3
struct __declspec(deprecated) X {
    void f(){}
};

struct __declspec(deprecated("** X2 is deprecated **")) X2 {
    void f(){}
};

int main() {
    X x;      // C4996
    X2 x2;    // C4996
}
```

См. также

[__declspec](#)

[Ключевые слова](#)

dllexport, dllimport

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

`dllexport` `dllimport` Атрибуты класса хранения и представляют собой расширения для языков С и С++, специфичные для Майкрософт. Их можно использовать для экспорта функций, данных и объектов в библиотеку DLL или импорта из такой библиотеки.

Синтаксис

```
_declspec( dllimport ) declarator
_declspec( dllexport ) declarator
```

Remarks

Эти атрибуты явно определяют интерфейс DLL для ее клиента, который может быть исполняемым файлом или другой библиотекой DLL. Объявление функций как `dllexport` устраняет необходимость в файле определения модуля (DEF), по крайней мере, по отношению к спецификации экспорттированных функций. `dllexport` Атрибут заменяет ключевое слово `__export`.

Если класс отмечен как `declspec(dllexport)`, все специализации шаблонов класса в иерархии классов неявно отмечаются как `declspec (dllexport)`. Это означает, что шаблоны класса создаются явно и члены класса должны быть определены.

`dllexport` функции предоставляет функцию с декорированным именем. Для функций С++ сюда входит видоизменение имени. Для функций С или функций, объявленных с модификатором `extern "C"`, сюда входит зависящее от платформы декорирование, основанное на соглашении о вызовах. Сведения о декорировании имен в коде С/С++ см. в разделе [декорированные имена](#). К экспорттированным функциям С и функциям С++ не применяется декорирование имен `extern "C"` с использованием `__cdecl` соглашения о вызовах.

Чтобы экспорттировать недекорированное имя, можно установить связь с помощью файла определения модуля (DEF-файла), определяющего недекорированное имя в разделе EXPORTS. Дополнительные сведения см. в разделе [EXPORTS](#). Другим способом экспортита недекорированного имени является использование `#pragma comment(linker, "/export:alias=decorated_name")` директивы в исходном коде.

При объявлении `dllexport` или `dllimport` необходимо использовать [синтаксис расширенных атрибутов](#) и `__declspec` ключевое слово.

Пример

```
// Example of the dllimport and dllexport class attributes
__declspec( dllimport ) int i;
__declspec( dllexport ) void func();
```

Кроме того, чтобы сделать код более понятным, можно использовать макроопределения:

```
#define DllImport __declspec( dllexport )
#define DllExport __declspec( dllimport )

DllExport void func();
DllExport int i = 10;
DllImport int j;
DllExport int n;
```

Дополнительные сведения см. в разделе:

- [Определения и объявления](#)
- [Определение встроенных функций C++ с помощью dllexport и dllimport](#)
- [Общие правила и ограничения](#)
- [Использование dllimport и dllexport в классах C++](#)

Завершение блока, относящегося только к системам Microsoft

См. также

[__declspec](#)

[Ключевые слова](#)

Определения и объявления (C++)

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Интерфейс DLL ссылается на все элементы (функции и данные), которые могут быть экспортированы какой-либо программой в системе; то есть все элементы, объявленные как `dllimport` или `dllexport`. Все объявления, входящие в интерфейс DLL, должны указывать либо `dllimport` атрибут, либо `dllexport`. Однако определение должно указывать только `dllexport` атрибут. Например, следующее определение функции вызовет ошибку компилятора.

```
__declspec( dllimport ) int func() { // Error; dllimport
                                     // prohibited on definition.
    return 1;
}
```

Показанный ниже код также вызовет ошибку.

```
__declspec( dllimport ) int i = 10; // Error; this is a definition.
```

Однако следующий синтаксис правильный.

```
__declspec( dllexport ) int i = 10; // Okay--export definition
```

Использование класса `dllexport` подразумевает определение, а `dllimport` подразумевает объявление. `extern` Для принудительного объявления необходимо использовать ключевое слово `WITH dllexport`. В противном случае подразумевается определение. Таким образом, приведенные ниже примеры правильны.

```
#define DllImport __declspec( dllimport )
#define DllExport __declspec( dllexport )

extern DllExport int k; // These are both correct and imply a
DllImport int j;       // declaration.
```

В следующих примерах поясняются предшествующие.

```
static __declspec( dllexport ) int l; // Error; not declared extern.

void func() {
    static __declspec( dllimport ) int s; // Error; not declared
                                         // extern.
    __declspec( dllimport ) int m;      // Okay; this is a
                                         // declaration.
    __declspec( dllexport ) int n;     // Error; implies external
                                         // definition in local scope.
    extern __declspec( dllimport ) int i; // Okay; this is a
                                         // declaration.
    extern __declspec( dllexport ) int k; // Okay; extern implies
                                         // declaration.
    __declspec( dllexport ) int x = 5;   // Error; implies external
                                         // definition in local scope.
}
```

Завершение блока, относящегося только к системам Майкрософт

См. также

[dllexport, dllimport](#)

Определение встроенных функций C++ с помощью `__declspec(dllexport)` и `__declspec(dllimport)`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Можно определить как встроенную функцию с `__declspec(dllexport)` атрибутом. В этом случае всегда создается экземпляр функции и она экспортируется независимо от того, ссылается ли на нее какой-либо модуль в программе. Предполагается, что функция должна импортироваться другой программой.

В качестве встроенной можно также определить функцию, объявленную с атрибутом `__declspec(dllimport)`. В этом случае функцию можно расширить (согласно спецификациям /Ob), но ее экземпляр никогда не создается. В частности, если принимается адрес встроенной импортированной функции, возвращается адрес функции, находящейся в библиотеке DLL. Это поведение аналогично получению адреса невстроенной импортированной функции.

Эти правила применяются для встраиваемых функций, определения которых отображаются внутри определения класса. Кроме того, локальные статические данные и строки во встроенных функциях поддерживают одинаковые идентификаторы между библиотекой DLL и клиентом так, как это было бы в одной программе (т. е. исполняемый файл без интерфейса DLL).

При предоставлении импортированных встроенных функций соблюдайте осторожность. Например, при обновлении библиотеки DLL не следует предполагать, что клиент использует ее измененную версию. Чтобы убедиться в загрузке правильной версии библиотеки DLL, перестройте также клиент этой библиотеки.

Завершение блока, относящегося только к системам Microsoft

См. также

[__declspec\(dllexport\)](#), [__declspec\(dllimport\)](#)

Общие правила и ограничения

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

- При объявлении функции или объекта без `dllimport` `dllexport` атрибута или функция или объект не считаются частью интерфейса DLL. Поэтому объявление функции или объекта должно находиться в том же самом модуле или в другом модуле той же программы. Чтобы сделать функцию или объект частью интерфейса DLL, необходимо объявить определение функции или объекта в другом модуле как `dllexport`. В противном случае возникает ошибка компоновщика.

Если объявить функцию или объект с `dllexport` атрибутом, его определение должно появиться в некоторых модулях той же программы. В противном случае возникает ошибка компоновщика.

- Если один модуль в программе содержит `dllimport` и объявления, и `dllexport` для одной и той же функции или объекта, `dllexport` атрибут имеет приоритет над `dllimport` атрибутом. Однако компилятор создает предупреждение. Пример:

```
_declspec( dllimport ) int i;
_dedeclspec( dllexport ) int i; // Warning; inconsistent;
                             // dllexport takes precedence.
```

- В C++ можно инициализировать глобально объявленный или статический указатель локальных данных или адрес объекта данных, объявленного с помощью `dllimport` атрибута, что приводит к ошибке в C. Кроме того, можно инициализировать статический указатель локальной функции с адресом функции, объявленной с помощью `dllimport` атрибута. В C такое присваивание задает указатель на адрес преобразователя импорта библиотеки DLL (заглушки кода, передающей контроль функции), а не адрес функции. В C++ оно задает указатель на адрес функции. Пример:

```
_declspec( dllimport ) void func1( void );
_dedeclspec( dllimport ) int i;

int *pi = &i; // Error in C
static void ( *pf )( void ) = &func1; // Address of thunk in C,
                                    // function in C++

void func2()
{
    static int *pi = &i; // Error in C
    static void ( *pf )( void ) = &func1; // Address of thunk in C,
                                         // function in C++
}
```

Однако поскольку программа, включающая `dllexport` атрибут в объявлении объекта, должна предоставлять определение для этого объекта в программе, можно инициализировать глобальный или локальный указатель на статическую функцию с адресом `dllexport` функции. Аналогичным образом можно инициализировать глобальный или локальный указатель на статические данные с адресом `dllexport` объекта данных. Например, следующий код не создает ошибки в C или C++:

```
__declspec( dllexport ) void func1( void );
__declspec( dllexport ) int i;

int *pi = &i;                                // Okay
static void ( *pf )( void ) = &func1;           // Okay

void func2()
{
    static int *pi = &i;                      // Okay
    static void ( *pf )( void ) = &func1; // Okay
}
```

- Если применить `dllexport` к обычному классу с базовым классом, который не помечен как `dllexport`, компилятор создаст C4275.

Компилятор создает то же самое предупреждение, если базовый класс является специализацией шаблона классов. Чтобы обойти это, пометьте базовый класс на `dllexport`. Проблема с специализацией шаблона класса — место для размещения `__declspec(dllexport)`; вы не можете помечать шаблон класса. Вместо этого следует явно создать экземпляр шаблона класса и пометить это явное создание экземпляра с помощью `dllexport`. Пример:

```
template class __declspec(dllexport) B<int>;
class __declspec(dllexport) D : public B<int> {
// ...
```

Этот обходной путь завершается сбоем, если аргумент шаблона — это производный класс. Пример:

```
class __declspec(dllexport) D : public B<D> {
// ...
```

Поскольку это распространенный шаблон с шаблонами, компилятор изменил семантику `dllexport` при применении к классу с одним или несколькими базовыми классами и когда один или несколько базовых классов являются специализацией шаблона класса. В этом случае компилятор неявно применяется `dllexport` к специализациям шаблонов классов. Вы можете выполнять следующие действия и не получать предупреждения:

```
class __declspec(dllexport) D : public B<D> {
// ...
```

Завершение блока, относящегося только к системам Microsoft

См. также

[`dllexport`, `dllimport`](#)

Использование `dllimport` и `dllexport` в классах C++

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Классы C++ можно объявлять с помощью `dllimport` атрибута или `dllexport`. Эти формы подразумевают, что импортирован или экспортирован весь класс. Классы, которые можно экспорттировать таким образом, называются экспортруемыми классами.

В следующем примере определяется экспортруемый класс. Экспортируются все его функции-члены и статические данные.

```
#define DllExport __declspec( dllexport )

class DllExport C {
    int i;
    virtual int func( void ) { return 1; }
};
```

Обратите внимание, что явное использование `dllimport` `dllexport` атрибутов и для членов экспортруемого класса запрещено.

Классы `dllexport`

При объявлении класса `dllexport` экспортруются все его функции-члены и статические элементы данных. Необходимо предоставить определения всех таких членов в одной программе. В противном случае возникает ошибка компоновщика. Единственным исключением из этого правила являются чистые виртуальные функции, для которых не требуется предоставлять явные определения. Однако, поскольку деструктор абстрактного класса всегда вызывается деструктором базового класса, чистые виртуальные деструкторы должны всегда предоставлять определение. Обратите внимание, что те же правила применяются и к неэкспортруемым классам.

При экспорте данных типа класса или функций, которые возвращают классы, не забудьте экспортовать класс.

Классы `dllimport`

При объявлении класса `dllimport` импортруются все его функции-члены и статические элементы данных. В отличие от поведения `dllimport` и `dllexport` в типах, не имеющих класса, статические члены данных не могут указывать определение в той же программе, в которой `dllimport` определен класс.

Наследование и экспортруемые классы

Все базовые классы экспортруемого класса должны быть экспортруемыми. В противном случае создается предупреждение компилятора. Кроме того, все доступные члены, которые также являются классами, должны быть доступными для экспортации. Это правило позволяет `dllexport` классу наследовать от `dllimport` класса, а `dllimport` класс должен наследовать от `dllexport` класса (хотя второй не рекомендуется). Как правило, все, что доступно клиенту библиотеки DLL (в соответствии с правилами доступа C++), должно быть частью экспортруемого интерфейса. Сюда входят закрытые данные-члены,

на которые ссылаются встраиваемые функции.

ИМПОРТ И ЭКСПОРТ СЕЛЕКТИВНОГО ЧЛЕНА

Поскольку функции-члены и статические данные в классе неявно имеют внешнюю компоновку, их можно объявить с `dllimport` помощью `dllexport` атрибута или, если только не экспортируется весь класс. Если импортируется или экспортируется весь класс, явное объявление функций-членов и данных в качестве `dllimport` или `dllexport` запрещено. Если объявить статический элемент данных в определении класса как `dllexport`, определение должно выполняться в той же программе (как и с внешней компоновкой некласса).

Аналогичным образом можно объявить функции элементов с `dllimport` `dllexport` атрибутами или. В этом случае необходимо указать `dllexport` Определение в той же программе.

Обратите внимание на несколько важных аспектов выборочного импорта и экспорта членов.

- Выборочный импорт и экспорт членов рекомендуется использовать для предоставления версии экспортированного интерфейса класса, который является более ограничивающим, то есть для которого можно разработать библиотеку DLL, предоставляющую меньше открытых и закрытых компонентов, чем разрешил бы язык. Кроме того, это может пригодиться для точной настройки экспортируемого интерфейса: если известно, что клиент по определению не может получить доступ к некоторым закрытым данным, не требуется экспортировать целый класс.
- При экспорте одной виртуальной функции в классе необходимо экспортировать все функции или хотя бы предоставить версии, которые клиент может использовать напрямую.
- Если имеется класс, в котором используется выборочный импорт и экспорт членов с виртуальными функциями, функции должны быть расположены в экспортируемом интерфейсе или определены встроенным образом (видимым клиенту).
- Если вы определили член как, `dllexport` но не включили его в определение класса, создается ошибка компилятора. Необходимо определить член в заголовке класса.
- Несмотря на то, что определение членов `dllimport` класса `dllexport` разрешено, переопределение интерфейса, указанного в определении класса, невозможно.
- Если определить функцию-член в месте, отличном от тела определения класса, в котором он был объявлен, то создается предупреждение, если функция определена как `dllexport` или `dllimport` (если это определение отличается от указанного в объявлении класса).

Завершение блока, относящегося только к системам Microsoft

См. также

[dllexport, dllimport](#)

jitintrinsic

12.11.2021 • 2 minutes to read

Помечает функцию как значимую для 64-разрядной среды CLR. Этот модификатор используется с определенными функциями в библиотеках Microsoft.

Синтаксис

```
__declspec(jitintrinsic)
```

Remarks

`jitintrinsic` Добавляет МОДОРТ ([IsJitIntrinsic](#)) в сигнатуру функции.

Пользователям не рекомендуется использовать этот `__declspec` модификатор, так как могут возникать непредвиденные результаты.

См. также

[__declspec](#)

[Ключевые слова](#)

naked (C++)

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Для функций, объявленных с `naked` атрибутом, компилятор создает код без пролога и кода эпилога. Эту возможность можно использовать, чтобы создавать свой собственный код на языке ассемблера и вставлять его в качестве пролога и эпилога. Функции с атрибутом `naked` особенно полезны для написания драйверов виртуальных устройств. Обратите внимание, что `naked` атрибут допустим только на платформах x86 и ARM и недоступен в x64.

Синтаксис

```
_declspec(naked) declarator
```

Remarks

Поскольку `naked` атрибут относится только к определению функции и не является модификатором типа, функции с атрибутом `naked` должны использовать синтаксис расширенных атрибутов и ключевое слово `_declspec`.

Компилятор не может создать встроенную функцию для функции, помеченной атрибутом `naked`, даже если эта функция также помечена ключевым словом `_forceinline`.

Компилятор выдает ошибку, если `naked` атрибут применяется к любому другому, кроме определения метода, не являющемуся членом.

Примеры

Этот код определяет функцию с `naked` атрибутом:

```
_declspec( naked ) int func( formal_parameters ) {}
```

Другой пример.

```
#define Naked __declspec( naked )
Naked int func( formal_parameters ) {}
```

`naked` Атрибут влияет только на характер создания кода компилятора для последовательностей пролога и эпилога функции. Код, который создается для вызова таких функций, не зависит от этого атрибута. Таким `naked` же атрибут не считается частью типа функции, а указатели функций не могут иметь `naked` атрибут. Более того, `naked` атрибут нельзя применить к определению данных. Например, следующий код вызывает ошибку:

```
_declspec( naked ) int i;
// Error--naked attribute not permitted on data declarations.
```

`naked` Атрибут относится только к определению функции и не может быть указан в прототипе функции.

Например, следующее объявление создает ошибку компилятора:

```
__declspec( naked ) int func(); // Error--naked attribute not permitted on function declarations
```

Завершение блока, относящегося только к системам Microsoft

См. также

[__declspec](#)

[Ключевые слова](#)

[Вызовы функций с атрибутом naked](#)

Только для систем Майкрософт

`noalias` означает, что вызов функции не изменяет и не ссылается на видимое глобальное состояние и изменяет только память, на которую указывает *непосредственно* параметры указателя (косвенные обращения).

Если функция помечена как `noalias`, оптимизатор может предположить, что только сами параметры, а также только внутренние косвенные обращения к параметрам указателей указываются или изменяются внутри функции.

`noalias` Аннотация применяется только в теле функции с аннотацией. Пометка функции как `__declspec(noalias)` не влияет на псевдонимы указателей, возвращаемых функцией.

Другое Примечание, которое может повлиять на присвоение псевдонимов, см. в разделе

[`__declspec\(restrict\)`](#).

Пример

В следующем примере демонстрируется использование `__declspec(noalias)`.

Когда функция `multiply`, обращающаяся к памяти, снабжена заметками `__declspec(noalias)`, она сообщает компилятору, что эта функция не изменяет глобальное состояние, за исключением указателей в списке параметров.

```

// declspec_noalias.c
#include <stdio.h>
#include <stdlib.h>

#define M 800
#define N 600
#define P 700

float * mempool, * memptr;

float * ma(int size)
{
    float * retval;
    retval = memptr;
    memptr += size;
    return retval;
}

float * init(int m, int n)
{
    float * a;
    int i, j;
    int k=1;

    a = ma(m * n);
    if (!a) exit(1);
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            a[i*n+j] = 0.1/k++;
    return a;
}

__declspec(noalias) void multiply(float * a, float * b, float * c)
{
    int i, j, k;

    for (j=0; j<P; j++)
        for (i=0; i<M; i++)
            for (k=0; k<N; k++)
                c[i * P + j] =
                    a[i * N + k] *
                    b[k * P + j];
}

int main()
{
    float * a, * b, * c;

    mempool = (float *) malloc(sizeof(float) * (M*N + N*P + M*P));

    if (!mempool)
    {
        puts("ERROR: Malloc returned null");
        exit(1);
    }

    memptr = mempool;
    a = init(M, N);
    b = init(N, P);
    c = init(M, P);

    multiply(a, b, c);
}

```

См. также

[__declspec](#)

Ключевые слова

[__declspec\(restrict\)](#)

noinline

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

`__declspec(noinline)` Указывает компилятору никогда не подставляемую конкретную функцию-член (функцию в классе).

Запрещать использовать функцию как встроенную имеет смысл, если она небольшая и не оказывает критического влияния на производительность кода. То есть, если функция небольшая и вряд ли будет вызываться часто (например, функция, которая обрабатывает условие ошибки).

Помните, что если функция помечена `noinline`, то вызывающая функция будет меньше и, таким же, является кандидатом для встраивания компилятора.

```
class X {  
    __declspec(noinline) int mbrfunc() {  
        return 0;  
    } // will not inline  
};
```

Завершение блока, относящегося только к системам Майкрософт

См. также

[__declspec](#)

[Ключевые слова](#)

[inline, __inline, __forceinline](#)

noreturn

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Этот `__declspec` атрибут сообщает компилятору, что функция не возвращает значение. Как следствие, компилятор знает, что код, следующий за вызовом `__declspec(noreturn)` функции, недоступен.

Если компилятор обнаруживает функцию с путем элемента управления, которая не возвращает значение, он создает предупреждение (C4715) или сообщение об ошибке (C2202). Если путь к элементу управления недоступен из-за функции, которая никогда не возвращает, можно использовать `__declspec(noreturn)` чтобы предотвратить это предупреждение или ошибку.

NOTE

Добавление `__declspec(noreturn)` в функцию, которая должна возвращать значение, может привести к неопределенному поведению.

Пример

В следующем примере `else` предложение не содержит оператор `return`. Объявление `fatal` как `__declspec(noreturn)` не позволяет избежать сообщения об ошибке или предупреждении.

```
// noreturn2.cpp
__declspec(noreturn) extern void fatal () {}

int main() {
    if(1)
        return 1;
    else if(0)
        return 0;
    else
        fatal();
}
```

Завершение блока, относящегося только к системам Майкрософт

См. также

[__declspec](#)

[Ключевые слова](#)

no_sanitize_address

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

`__declspec(no_sanitize_address)` Описатель сообщает компилятору, что следует отключить очистку адреса для функций, локальных переменных или глобальных переменных. Этот описатель используется в сочетании с [адрессанитизер](#).

NOTE

`__declspec(no_sanitize_address)` отключает поведение *компилятора*, а не поведения *среды выполнения*.

Пример

Примеры см. в [справочнике по сборке адрессанитизер](#).

Завершение блока, относящегося только к системам Майкрософт

См. также

[__declspec](#)

[Словами](#)

[AddressSanitizer](#)

nothrow (C++)

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

`__declspec` Расширенный атрибут, который может использоваться в объявлении функций.

Синтаксис

`__declspec` *возвращаемого типа* (`throw`) [*соглашение о вызовах*] *функция-имя* ([*Argument-List*])

Remarks

Рекомендуется, чтобы в новом коде использовался оператор `except`, а не `__declspec(nothrow)`.

Этот атрибут сообщает компилятору, что и объявленная функция, и функции, которые она вызывает, никогда не создают исключений. Однако директива не применяется. Иными словами, он никогда не вызывает `std::terminate`, в отличие от `noexcept`, или в режиме std: c++ 17 (Visual Studio 2017 версии 15,5 и более поздних) `throw()`.

Поскольку по умолчанию теперь используется модель синхронной обработки исключений, то компилятор может исключить механизм, позволяющий отслеживать время существования удаляемых объектов в такой функции, и значительно уменьшить объем кода. При наличии следующей директивы препроцессора три приведенные ниже объявления функций эквивалентны в режиме /std: c++ 14 :

```
#define WINAPI __declspec(nothrow) __stdcall  
  
void WINAPI f1();  
void __declspec(nothrow) __stdcall f2();  
void __stdcall f3() throw();
```

В /std: режим c++ 17 `throw()` не эквивалентен другим, которые используют, `__declspec(nothrow)` Так как вызывается `std::terminate` при возникновении исключения из функции.

В `void __stdcall f3() throw();` объявлении используется синтаксис, определенный стандартом C++. В C++ 17 `throw()` ключевое слово является устаревшим.

Завершение блока, относящегося только к системам Майкрософт

См. также

[__declspec](#)

[noexcept](#)

[Ключевые слова](#)

novtable

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Это `__declspec` Расширенный атрибут.

Такая форма применима `__declspec` к любому объявлению класса, но должна применяться только к чистым классам интерфейса, то есть к классам, экземпляры которых никогда не будут создаваться самостоятельно. `__declspec` Компонент останавливает компилятор на создание кода для инициализации вфptr в конструкторах и деструкторе класса. Во многих случаях это приводит к удалению единственной ссылки на связанную с классом таблицу vtable, в результате чего компоновщик удаляет ее.

Использование такой формы `__declspec` может привести к значительному сокращению размера кода.

Если попытаться создать экземпляр класса, помеченного как, `novtable` а затем получить доступ к члену класса, вы получите нарушение прав доступа (AV).

Пример

```
// novtable.cpp
#include <stdio.h>

struct __declspec(novtable) X {
    virtual void mf();
};

struct Y : public X {
    void mf() {
        printf_s("In Y\n");
    }
};

int main() {
    // X *pX = new X();
    // pX->mf();    // Causes a runtime access violation.

    Y *pY = new Y();
    pY->mf();
}
```

In Y

Завершение блока, относящегося только к системам Майкрософт

См. также

[__declspec](#)

[Ключевые слова](#)

Указывает, что управляемый процесс приложения должен иметь одну копию конкретной глобальной переменной, статической переменной-члена или статической локальной переменной, совместно используемой во всех доменах приложения в этом процессе. В основном это было предназначено для использования при компиляции с параметром `/clr:pure`, который является устаревшим в Visual Studio 2015 и не поддерживается в Visual Studio 2017. При компиляции с `/clr`, глобальные и статические переменные по умолчанию используются для каждого процесса и не требуют использования `_declspec(process)`.

С помощью можно пометить только глобальную переменную, статическую переменную-член или статическую локальную переменную собственного типа `_declspec(process)`.

`process` допускается только при компиляции с `/clr`.

Если требуется, чтобы каждый домен приложения имел собственную копию глобальной переменной, используйте [домен приложения](#).

Дополнительные сведения см. в разделе [домены приложений и Visual C++](#).

См. также раздел

[`_declspec`](#)

[Ключевые слова](#)

property (C++)

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Этот атрибут может применяться к нестатическим "виртуальным данным-членам" в определении класса или структуры. Компилятор обрабатывает эти "виртуальные данные-члены" как данные-член, заменяя ссылки вызовами функций.

Синтаксис

```
__declspec( property( get=get_func_name ) ) declarator  
__declspec( property( put=put_func_name ) ) declarator  
__declspec( property( get=get_func_name, put=put_func_name ) ) declarator
```

Remarks

Когда компилятор видит элемент данных, объявленный с этим атрибутом, справа от оператора выбора члена ("." или "->"), он преобразует операцию в `get` `put` функцию или, в зависимости от того, является ли такое выражение l-значением или r-value. В более сложных контекстах, таких как «`+=`», переписывание выполняется как, так `get` и `put`.

Этот атрибут также может использоваться при объявлении пустого массива в определении класса или структуры. Пример:

```
__declspec(property(get=GetX, put=PutX)) int x[];
```

Приведенный выше оператор указывает, что `x[]` может использоваться с одним или несколькими индексами массива. В этом случае выражение `i=p->x[a][b]` будет преобразовано в `i=p->GetX(a, b)`, а выражение `p->x[a][b] = i` будет преобразовано в `p->PutX(a, b, i);`

Завершение блока, относящегося только к системам Майкрософт

Пример

```
// declspec_property.cpp
struct S {
    int i;
    void putprop(int j) {
        i = j;
    }

    int getprop() {
        return i;
    }

    __declspec(property(get = getprop, put = putprop)) int the_prop;
};

int main() {
    S s;
    s.the_prop = 5;
    return s.the_prop;
}
```

См. также

[__declspec](#)

[Ключевые слова](#)

restrict

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

При применении к объявлению или определению функции, возвращающему тип указателя, `restrict` сообщает компилятору, что функция возвращает объект, который не имеет псевдонима, то есть на него ссылаются любые другие указатели. Это позволяет компилятору выполнять дополнительные оптимизации.

Синтаксис

```
__declspec(restrict) функция pointer_return_type();
```

Remarks

Компилятор распространяет `__declspec(restrict)`. Например, `malloc` функция CRT имеет `__declspec(restrict)` декорирование, поэтому компилятор предполагает, что указатели, инициализированные в области памяти, `malloc` также не имеют псевдонимов ранее существующих указателей.

Компилятор не проверяет, что возвращаемый указатель не является псевдонимом. Ответственность разработчика в том, чтобы программа не применяет псевдоним, помеченный модификатором `restrict __declspec`.

Аналогичные семантики для переменных см. в разделе [__restrict](#).

Другие аннотации, применяемые к псевдонимам в функции, см. в разделе [__declspec \(псевдоним\)](#).

дополнительные сведения о `restrict` ключевом слове, которое является частью C++ AMP, см. в разделе [restrict \(C++ AMP\)](#).

Пример

В следующем примере демонстрируется использование `__declspec(restrict)`.

Если `__declspec(restrict)` применяется к функции, возвращающей указатель, это указывает компилятору, что память, на которую указывает возвращаемое значение, не имеет псевдонима. В этом примере указатели `mempool` и `memptr` являются глобальными, поэтому компилятор не может гарантировать, что память, на которую они ссылаются, не имеет псевдонимов. Однако они используются в `ma` и вызывающем объекте `init` таким образом, что возвращает память, которая не ссылается программой, поэтому для оптимизатора используется `__declspec (restrict)`. Это аналогично тому, как заголовки CRT выделяют функции выделения, например с `malloc` помощью `__declspec(restrict)` чтобы указать, что они всегда возвращают память, которая не может быть псевдонимом существующих указателей.

```

// declspec_restrict.c
// Compile with: cl /W4 declspec_restrict.c
#include <stdio.h>
#include <stdlib.h>

#define M 800
#define N 600
#define P 700

float * mempool, * memptr;

__declspec(restrict) float * ma(int size)
{
    float * retval;
    retval = memptr;
    memptr += size;
    return retval;
}

__declspec(restrict) float * init(int m, int n)
{
    float * a;
    int i, j;
    int k=1;

    a = ma(m * n);
    if (!a) exit(1);
    for (i=0; i<m; i++)
        for (j=0; j<n; j++)
            a[i*n+j] = 0.1f/k++;
    return a;
}

void multiply(float * a, float * b, float * c)
{
    int i, j, k;

    for (j=0; j<P; j++)
        for (i=0; i<M; i++)
            for (k=0; k<N; k++)
                c[i * P + j] =
                    a[i * N + k] *
                    b[k * P + j];
}

int main()
{
    float * a, * b, * c;

    mempool = (float *) malloc(sizeof(float) * (M*N + N*P + M*P));

    if (!mempool)
    {
        puts("ERROR: Malloc returned null");
        exit(1);
    }

    memptr = mempool;
    a = init(M, N);
    b = init(N, P);
    c = init(M, P);

    multiply(a, b, c);
}

```

Завершение блока, относящегося только к системам Майкрософт

См. также

[Ключевые слова](#)

[_declspec](#)

[_declspec \(псевдоним\)](#)

safebuffers

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Указывает компилятору не вставлять проверки безопасности на переполнение буфера для функции.

Синтаксис

```
__declspec( safebuffers )
```

Remarks

Параметр компилятора /GS заставляет компилятор проверить переполнение буфера, вставив проверки безопасности в стек. Типы структур данных, подходящих для проверок безопасности, описаны в параметре [/GS \(проверка безопасности буфера\)](#). Дополнительные сведения об обнаружении переполнения буфера см. в [разделе функции безопасности в MSVC](#).

Экспертная проверка кода вручную или внешний анализ могут показать, что функция безопасна с точки зрения переполнения буфера. В этом случае можно отключить проверку безопасности для функции, применив к `__declspec(safebuffers)` объявлению функции ключевое слово.

Caution

Проверки безопасности буфера — это важное средство обеспечения безопасности, которое практически не оказывается производительности. Поэтому отключать их не рекомендуется, кроме тех редких случаев, когда производительность функции исключительно важна, а сама функция заведомо безопасна.

Встраиваемые функции

Основная функция может использовать ключевое слово [встраивания](#) для вставки копии *вторичной функции*. Если `__declspec(safebuffers)` ключевое слово применяется к функции, Обнаружение переполнения буфера для этой функции подавляется. Однако встраивание влияет на `__declspec(safebuffers)` ключевое слово следующим образом.

Предположим, что для обеих функций указан параметр компилятора /GS , но основная функция указывает `__declspec(safebuffers)` ключевое слово. Структуры данных в дополнительной функции делают ее законным объектом для проверок безопасности, поэтому такие проверки в ней не отключаются. В данном случае:

- Укажите ключевое слово [_forceinline](#) во вторичной функции, чтобы заставить компилятор подставляемую функцию независимо от оптимизации компилятора.
- Так как вторичная функция подходит для проверок безопасности, проверки безопасности также применяются к основной функции, даже если она указывает `__declspec(safebuffers)` ключевое слово.

Пример

В следующем коде показано, как использовать `__declspec(safebuffers)` ключевое слово.

```
// compile with: /c /GS
typedef struct {
    int x[20];
} BUFFER;
static int checkBuffers() {
    BUFFER cb;
    // Use the buffer...
    return 0;
};
static __declspec(safebuffers)
int noCheckBuffers() {
    BUFFER ncb;
    // Use the buffer...
    return 0;
}
int wmain() {
    checkBuffers();
    noCheckBuffers();
    return 0;
}
```

Завершение блока, относящегося только к системам Майкрософт

См. также

[__declspec](#)
[Ключевые слова](#)
[inline, __inline, __forceinline](#)
[strict_gs_check](#)

Блок, относящийся только к системам Microsoft

Сообщает компилятору, что объявленный элемент глобальных данных (переменная или объект) является универсальным COMDAT (упакованной функцией).

Синтаксис

```
** __declspec( selectany ) **декларатор
```

Remarks

Во время компоновки, если отображается несколько определений COMDAT, компоновщик выбирает один из них и игнорирует остальные. Если выбран параметр компоновщика [/OPT:REF](#) (оптимизации), то будет удалено исключение COMDAT, чтобы удалить все элементы данных, которые не ссылаются на выходные данные компоновщика.

Конструкторы и назначение с помощью глобальной функции или статических методов в объявлении не создадут ссылку и не предотвратят исключение [/OPT:REF](#). Побочные эффекты такого кода не должны быть зависимыми, если не существует других ссылок на данные.

Для динамически инициализированных глобальных объектов `selectany` будет также удален код инициализации нессылающегося объекта.

Обычно элемент глобальных данных можно инициализировать только один раз в проекте EXE или DLL. `selectany` может использоваться при инициализации глобальных данных, определенных заголовками, когда один и тот же заголовок отображается в нескольких исходных файлах. `selectany` доступен в компиляторах C и C++.

NOTE

`selectany` может применяться только к фактической инициализации элементов глобальных данных, видимых извне.

Пример: `selectany` атрибут

В этом коде показано, как использовать `selectany` атрибут:

```

//Correct - x1 is initialized and externally visible
__declspec(selectany) int x1=1;

//Incorrect - const is by default static in C++, so
//x2 is not visible externally (This is OK in C, since
//const is not by default static in C)
const __declspec(selectany) int x2 =2;

//Correct - x3 is extern const, so externally visible
extern const __declspec(selectany) int x3=3;

//Correct - x4 is extern const, so it is externally visible
extern const int x4;
const __declspec(selectany) int x4=4;

//Incorrect - __declspec(selectany) is applied to the uninitialized
//declaration of x5
extern __declspec(selectany) int x5;

// OK: dynamic initialization of global object
class X {
public:
X(int i){i++;}
int i;
};

__declspec(selectany) X x(1);

```

Пример: использование `selectany` атрибута для свертывания данных COMDAT

В этом коде показано, как с помощью `selectany` атрибута обеспечить сворачивание данных COMDAT при использовании `/OPT:ICF` параметра компоновщика. Обратите внимание, что данные должны быть помечены `selectany` и помещены в `const` раздел (ReadOnly). Раздел, доступный только для чтения, необходимо указать явно.

```

// selectany2.cpp
// in the following lines, const marks the variables as read only
__declspec(selectany) extern const int ix = 5;
__declspec(selectany) extern const int jx = 5;
int main() {
    int ij;
    ij = ix + jx;
}

```

Завершение блока, относящегося только к системам Microsoft

См. также

[__declspec](#)

[Ключевые слова](#)

spectre

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Сообщает компилятору, что не нужно вставлять инструкции устранением рисков Spectre с непостоянными барьерами выполнения для функции.

Синтаксис

`__declspec (устранением рисков Spectre (смягчение))`

Remarks

Параметр компилятора `/Qspectre` указывает компилятору на необходимость вставки гипотетических инструкций по барьеру выполнения. Они вставляются, когда анализ указывает на наличие уязвимости устранением рисков Spectre с вариантом 1. Определенные инструкции зависят от процессора. Хотя эти инструкции должны оказать минимальное влияние на размер или производительность кода, возможны случаи, когда код не подвержен воздействию уязвимости и требует максимальной производительности.

Экспертный анализ может определить, что функция является надежной из-за дефекта обхода проверки устранением рисков Spectre с вариантом 1. В этом случае можно подавить создание кода устранения проблем в функции, применив `__declspec(spectre(nomitigation))` к объявлению функции.

Caution

Инструкции по снижению производительности `/Qspectre` обеспечивают важную защиту и значительно влияют на производительность. Поэтому отключать их не рекомендуется, кроме тех редких случаев, когда производительность функции исключительно важна, а сама функция заведомо безопасна.

Пример

В приведенном ниже коде показано использование `__declspec(spectre(nomitigation))`.

```
// compile with: /c /Qspectre
static __declspec(spectre(nomitigation))
int noSpectreIssues() {
    // No Spectre variant 1 vulnerability here
    // ...
    return 0;
}

int main() {
    noSpectreIssues();
    return 0;
}
```

Завершение блока, относящегося только к системам Майкрософт

См. также

[__declspec](#)

[Ключевые слова](#)

/Qspectre

thread

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

`thread` Расширенный модификатор класса хранения используется для объявления локальной переменной потока. Для переносимого эквивалента в C++ 11 и более поздних версиях используйте описатель класса хранения `thread_local` для переносимого кода. В Windows `thread_local` реализуется с помощью `__declspec(thread)`.

Синтаксис

`** __declspec(thread) **декларатор`

Remarks

Локальное хранилище потока (TLS) — это механизм, с помощью которого в каждом потоке многопоточного процесса выделяется хранилище для хранения данных определенного потока. В стандартных многопоточных программах данные совместно используются всеми потоками заданного процесса, в то время как локальное хранилище потоков является механизмом предоставления данных для конкретного потока. Полное описание потоков см. в разделе [многопоточность](#).

Объявления локальных переменных потока должны использовать [синтаксис расширенных атрибутов](#) и `__declspec` ключевое слово с `thread` ключевым словом. В следующем примере кода показано, как объявлять целочисленную локальную переменную потока и инициализировать её некоторым значением:

```
__declspec( thread ) int tls_i = 1;
```

При использовании локальных переменных потока в динамически загружаемых библиотеках необходимо иметь в виду факторы, которые могут привести к неправильной инициализации локальной переменной потока:

- Если переменная инициализируется вызовом функции (включая конструкторы), эта функция будет вызываться только для потока, который вызвал загрузку в процесс двоичного файла или библиотеки DLL, а также для потоков, запущенных после загрузки двоичного файла или библиотеки DLL. Функции инициализации не вызываются для других потоков, которые уже выполнялись при загрузке библиотеки DLL. Динамическая инициализация происходит при вызове DllMain для DLL_THREAD_ATTACH, но библиотека DLL никогда не получает это сообщение, если библиотека DLL не находится в процессе запуска потока.
- Локальные переменные потока, инициализированные статически с постоянными значениями, обычно инициализируютсяенным образом на всех потоках. Однако по состоянию на декабрь 2017 существует известная ошибка соответствия в компиляторе Microsoft C++, в результате чего `constexpr` переменные получают динамическую, а не статическую инициализацию.

Примечание. обе эти проблемы должны быть исправлены в будущих обновлениях компилятора.

Кроме того, при объявлении локальных объектов и переменных потока необходимо следовать приведенным ниже рекомендациям.

- Атрибут можно применять `thread` только к объявлениям и определениям классов и данных;

`thread` не может использоваться в объявлениях или определениях функций.

- Атрибут можно указать `thread` только для элементов данных со статической длительностью хранения. Сюда входят глобальные объекты данных (`static` И `extern`), локальные статические объекты и статические члены данных классов. Нельзя объявлять автоматические объекты данных с `thread` атрибутом.
- Необходимо использовать `thread` атрибут для объявления и определение локального объекта потока, будь то объявление и определение происходят в одном и том же файле или в отдельных файлах.
- Нельзя использовать атрибут в `thread` качестве модификатора типа.
- Так как объявление объектов, использующих `thread` атрибут, разрешено, эти два примера семантически эквивалентны:

```
// declspec_thread_2.cpp
// compile with: /LD
__declspec( thread ) class B {
public:
    int data;
} BObject; // BObject declared thread local.

class B2 {
public:
    int data;
};
__declspec( thread ) B2 BObject2; // BObject2 declared thread local.
```

- Стандартный С разрешает инициализацию объекта или переменной с помощью выражения, включающего ссылку на себя, но только для нестатических объектов. Хотя C++ обычно разрешает такую динамическую инициализацию объекта с помощью выражения, включающего ссылку на себя, этот тип инициализации не разрешен с локальными объектами потока. Пример:

```
// declspec_thread_3.cpp
// compile with: /LD
#define Thread __declspec( thread )
int j = j; // Okay in C++; C error
Thread int tls_i = sizeof( tls_i ); // Okay in C and C++
```

`sizeof` Выражение, включающее инициализируемый объект, не является ссылкой на себя и допускается в С и C++.

Завершение блока, относящегося только к системам Майкрософт

См. также

[__declspec](#)
[Ключевые слова](#)
[локальный служба хранилища потока \(TLS\)](#)

uuid (C++)

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Компилятор присоединяет идентификатор GUID к классу или структуре, объявленным или определенным (только для определений полных объектов COM) с `uuid` атрибутом.

Синтаксис

```
__declspec( uuid("ComObjectGUID") ) declarator
```

Remarks

`uuid` Атрибут принимает в качестве аргумента строку. Эта строка присваивает идентификатор GUID в стандартном формате реестра с разделителями {} или без них. Пример:

```
struct __declspec(uuid("00000000-0000-0000-c000-000000000046")) IUnknown;
struct __declspec(uuid("{00020400-0000-0000-c000-000000000046}) IDispatch;
```

Этот атрибут можно применить при повторном объявлении. Это позволяет заголовкам систем предоставлять определения интерфейсов `IUnknown`, например, и повторное объявление в каком-либо другом заголовке (например, <comdef.h>) для предоставления идентификатора GUID.

Ключевое слово `_uuidof` можно применить для получения постоянного GUID, присоединенного к определяемому пользователем типу.

Завершение блока, относящегося только к системам Microsoft

См. также

[__declspec](#)

[Ключевые слова](#)

`_restrict`

12.11.2021 • 2 minutes to read

Как и модификатор `_declspec (restrict)`, `_restrict` ключевое слово (два символа подчеркивания "_") указывает, что символ не является псевдонимом в текущей области. `_restrict` Ключевое слово отличается от `_declspec (restrict)` модификатора следующими способами.

- `_restrict` Ключевое слово допустимо только для переменных и `_declspec (restrict)` допустимо только в объявлениях и определениях функций.
- `_restrict` похоже на `restrict` С, начиная с C99 и доступных в режиме [/std:c11](#) ИЛИ [/std:c17](#), но `_restrict` может использоваться в программах C++ и С.
- Если `_restrict` используется, компилятор не распространяет свойство переменной без псевдонима. Это значит, что при присвоении `_restrict` переменной значения, не являющегося `_restrict` переменной, компилятор по-прежнему будет разрешать псевдониму `ne_restrict` переменной. Это отличается от поведения `restrict` ключевого слова языка С C99.

Как правило, если вы хотите повлиять на поведение целой функции, используйте `_declspec (restrict)` вместо ключевого слова.

Для совместимости с предыдущими версиями аргумент `_restrict` является синонимом, `_restrict` если только параметр компилятора не [/za](#) (отключает расширения языка) .

в Visual Studio 2015 и более поздних версий `_restrict` можно использовать ссылки на C++.

NOTE

При использовании в переменной, которая также имеет `volatile` ключевое слово, имеет `volatile` приоритет.

Пример

```
// _restrict_keyword.c
// compile with: /LD
// In the following function, declare a and b as disjoint arrays
// but do not have same assurance for c and d.
void sum2(int n, int * __restrict a, int * __restrict b,
          int * c, int * d) {
    int i;
    for (i = 0; i < n; i++) {
        a[i] = b[i] + c[i];
        c[i] = b[i] + d[i];
    }
}

// By marking union members as __restrict, tell compiler that
// only z.x or z.y will be accessed in any given scope.
union z {
    int * __restrict x;
    double * __restrict y;
};
```

См. также

[Ключевые слова](#)

`_sptr, _uptr`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Используйте `_sptr` Модификатор или `_uptr` для объявления 32-разрядного указателя, чтобы указать, как компилятор преобразует 32-разрядный указатель в 64-разрядный указатель. 32-разрядный указатель преобразуется, например, при присвоении 64-разрядной переменной или при разыменовывании на 64-разрядной платформе.

В документации корпорации Microsoft по поддержке 64-разрядных платформ иногда старший значащий бит 32-разрядного указателя называется знаковым битом. По умолчанию компилятор использует расширение знака для преобразования 32-разрядного указателя в 64-разрядный. При этом для младших значащих 32 битов 64-разрядного указателя устанавливается значение 32-разрядного указателя, а для старших значащих 32 битов устанавливается значение знакового бита 32-разрядного указателя. Такое преобразование дает правильные результаты, если знаковый бит равен 0, но если он равен 1, то результаты получаются неправильные. Например, 32-разрядный адрес 0xFFFFFFFF преобразуется в эквивалентный 64-разрядный адрес 0x000000007FFFFFFF, а 32-разрядный адрес 0x80000000 ошибочно преобразуется в 0xFFFFFFFF80000000.

`_sptr` Модификатор, подписанный указателем, указывает, что преобразование указателя устанавливает наиболее значимые биты 64-разрядного указателя на бит знака 32-разрядного указателя. `_uptr` Модификатор (или неподписанный указатель) указывает, что преобразование устанавливает наиболее значимые биты в ноль. В следующих объявлениях показаны `_sptr` `_uptr` модификаторы и, используемые с двумя неквалифицированными указателями, два указателя, дополненные типом `_ptr32`, и параметр функции.

```
int * __sptr psp;
int * __uptr pup;
int * __ptr32 __sptr psp32;
int * __ptr32 __uptr pup32;
void MyFunction(char * __uptr __ptr32 myValue);
```

Используйте `_sptr` `_uptr` модификаторы и в объявлениях указателей. Используйте модификаторы в позиции [квалификатора типа указателя](#), что означает, что модификатор должен следовать за звездочкой. Нельзя использовать модификаторы с [указателями на члены](#). Модификаторы не влияют на объявления, не являющиеся объявлениями указателей.

Для совместимости с предыдущими версиями `_sptr` и `_uptr` являются синонимами для `_sptr` и, `_uptr` если только параметр компилятора [/Za не \(отключил расширения языка\)](#) указан.

Пример

В следующем примере объявляются 32-разрядные указатели, которые используют `_sptr` `_uptr` модификаторы и, присваивают каждый 32-разрядный указатель переменной указателя 64-bit, а затем отображает шестнадцатеричное значение 64 каждого из динамических битовых указателей. Пример компилировался с помощью собственного 64-разрядного компилятора и выполнялся на 64-разрядной платформе.

```

// sptr_uptr.cpp
// processor: x64
#include <stdio.h>

int main()
{
    void *      __ptr64 p64;
    void *      __ptr32 p32d; //default signed pointer
    void * __sptr __ptr32 p32s; //explicit signed pointer
    void * __uptr __ptr32 p32u; //explicit unsigned pointer

    // Set the 32-bit pointers to a value whose sign bit is 1.
    p32d = reinterpret_cast<void *>(0x87654321);
    p32s = p32d;
    p32u = p32d;

    // The printf() function automatically displays leading zeroes with each 32-bit pointer. These are unrelated
    // to the __sptr and __uptr modifiers.
    printf("Display each 32-bit pointer (as an unsigned 64-bit pointer):\n");
    printf("p32d:      %p\n", p32d);
    printf("p32s:      %p\n", p32s);
    printf("p32u:      %p\n", p32u);

    printf("\nDisplay the 64-bit pointer created from each 32-bit pointer:\n");
    p64 = p32d;
    printf("p32d: p64 = %p\n", p64);
    p64 = p32s;
    printf("p32s: p64 = %p\n", p64);
    p64 = p32u;
    printf("p32u: p64 = %p\n", p64);
    return 0;
}

```

```

Display each 32-bit pointer (as an unsigned 64-bit pointer):
p32d:      0000000087654321
p32s:      0000000087654321
p32u:      0000000087654321

```

```

Display the 64-bit pointer created from each 32-bit pointer:
p32d: p64 = FFFFFFFF87654321
p32s: p64 = FFFFFFFF87654321
p32u: p64 = 0000000087654321

```

Завершение блока, относящегося только к системам Майкрософт

См. также

[Модификаторы, специфичные для Майкрософт](#)

`_unaligned`

12.11.2021 • 2 minutes to read

Зависит от корпорации Майкрософт. При объявлении указателя с `_unaligned` модификатором компилятор предполагает, что указатель обращается к несогласованным данным. Следовательно, код, соответствующий платформе, создается для управления несогласованными операциями чтения и записи через указатель.

Remarks

Этот модификатор описывает выравнивание данных, адресованных указателю; Предполагается, что указатель должен быть согласован.

Необходимость в `_unaligned` ключевом слове зависит от платформы и среды. Неправильное пометка данных может привести к проблемам, от снижения производительности до сбоев оборудования.

`_unaligned` Модификатор недопустим для платформы x86.

Для совместимости с предыдущими версиями аргумент `_unaligned` является синонимом, `_unaligned` если только параметр компилятора не `/za` ([отключает расширения языка](#)) .

Дополнительные сведения о выравнивании см. в разделах:

- [align](#)
- [alignof Оператор](#)
- [pack](#)
- [/zp](#) (Выравнивание членов структуры)
- [Примеры выравнивания структуры](#)

См. также

[Ключевые слова](#)

_w64

12.11.2021 • 2 minutes to read

Это ключевое слово Майкрософт устарело. в версиях Visual Studio более ранние, чем Visual Studio 2013, это позволяет помечать переменные, чтобы при компиляции с помощью [/Wp64](#) компилятор выдавал предупреждения о предупреждении, которые будут выводиться при компиляции с помощью 64-разрядного компилятора.

Синтаксис

тип `_w64` идентификатор

Параметры

type

Один из трех типов, которые могут вызвать проблемы в коде, перенесенном из 32-бит в 64-разрядный компилятор: `int`, `long` или указатель.

identifier

Идентификатор создаваемой переменной.

Remarks

IMPORTANT

параметр компилятора [/Wp64](#) и `_w64` ключевое слово не рекомендуются в Visual Studio 2010 и Visual Studio 2013 и удаляются начиная с Visual Studio 2013. Если в `/Wp64` командной строке используется параметр компилятора, компилятор выдает предупреждение Command-Line D9002 Warning. `_w64` Ключевое слово игнорируется без уведомления. Вместо использования этого параметра и ключевого слова для обнаружения проблем переносимости в 64-разрядном режиме используйте компилятор Microsoft C++, предназначенный для 64-разрядной платформы. Дополнительные сведения см. в статье [настройка Visual C++ для 64-разрядных целевых платформ x64](#).

Любое определение `typedef`, имеющее `_w64` на нем, должно иметь 32 бит на x86 и 64 бит на x64.

для обнаружения проблем переносимости с использованием версий компилятора Microsoft C++, предшествующих Visual Studio 2010, `_w64` ключевое слово должно быть задано для всех типов `typedef`, которые меняют размер между 32 бит и 64 разрядных платформ. Для любого такого типа `_w64` должен присутствовать только в 32-разрядном определении `typedef`.

Для совместимости с предыдущими версиями `_w64` является синонимом, `_w64` если только параметр компилятора [/Za](#) не (отключил расширения языка) указан.

`_w64` Ключевое слово игнорируется, если компиляция не использует `/Wp64`.

Дополнительные сведения о переносе на 64-разрядные платформы см. в следующих разделах:

- [Параметры компилятора MSVC](#)
- [Перенос 32-разрядного кода в 64-разрядный код](#)
- [Настройка Visual C++ для 64-разрядных целевых объектов с архитектурой x64](#)

Пример

```
// __w64.cpp
// compile with: /W3 /WP64
typedef int Int_32;
#ifndef _WIN64
typedef __int64 Int_Native;
#else
typedef int __w64 Int_Native;
#endif

int main() {
    Int_32 i0 = 5;
    Int_Native i1 = 10;
    i0 = i1;    // C4244 64-bit int assigned to 32-bit int

    // char __w64 c;  error, cannot use __w64 on char
}
```

См. также

[Ключевые слова](#)

func

12.11.2021 • 2 minutes to read

(C++ 11) Предопределенный идентификатор `__Func__` неявно определяется как строка, содержащая неполное и недекорированное имя включающей функции. `__Func__` задается стандартом C++ и не является расширением Microsoft.

Синтаксис

```
__func__
```

Возвращаемое значение

Возвращает константный массив символов, заканчивающийся нулем, содержащий имя функции.

Пример

```
#include <string>
#include <iostream>

namespace Test
{
    struct Foo
    {
        static void DoSomething(int i, std::string s)
        {
            std::cout << __func__ << std::endl; // Output: DoSomething
        }
    };
}

int main()
{
    Test::Foo::DoSomething(42, "Hello");

    return 0;
}
```

Требования

C++11

Поддержка СОМ компилятора

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Компилятор Microsoft C++ может напрямую читать библиотеки типов модели СОМ и переводить содержимое в исходный код C++, который может быть добавлен в компиляцию. Расширения языка доступны для упрощения программирования СОМ на стороне клиента для классических приложений.

С помощью [директивы препроцессора #import](#) компилятор может прочитать библиотеку типов и преобразовать ее в заголовочный файл C++, ОПИСЫВАЮЩИЙ СОМ-интерфейсы в качестве классов. Набор атрибутов `#import` доступен для пользовательского контроля содержимого в полученных файлах заголовка библиотеки типов.

Можно использовать идентификатор [UUID](#) расширенного атрибута `_declspec`, чтобы назначить глобально уникальный идентификатор (GUID) для СОМ-объекта. Ключевое слово `_uuidof` можно использовать для извлечения идентификатора GUID, связанного с СОМ-объектом. `_declspec` Можно использовать другой атрибут [Property](#), чтобы указать `get` `set` методы и для элемента данных СОМ-объекта.

Набор функций СОМ поддерживает глобальные функции и классы, предназначенные для поддержки `VARIANT` типов и `BSTR`, реализации смарт-указателей и инкапсуляции объекта `Error`, вызываемого `_com_raise_error`:

- [Глобальные функции компилятора СОМ](#)
- [_bstr_t](#)
- [_com_error](#)
- [_com_ptr_t](#)
- [_variant_t](#)

Завершение блока, относящегося только к системам Microsoft

См. также

[Классы поддержки СОМ компилятора](#)
[Глобальные функции компилятора СОМ](#)

Глобальные функции СОМ-модели компилятора

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Доступны следующие процедуры.

ФУНКЦИЯ	ОПИСАНИЕ
_com_raise_error	Вызывает исключение <code>_com_error</code> в ответ на сбой.
_set_com_error_handler	Заменяет функцию по умолчанию, используемую для обработки ошибок СОМ.
ConvertBSTRToString	Преобразует значение типа <code>BSTR</code> в значение типа <code>char *</code> .
ConvertStringToBSTR	Преобразует значение типа <code>char *</code> в значение типа <code>BSTR</code> .

Завершение блока, относящегося только к системам Microsoft

См. также

[Классы поддержки СОМ компилятора](#)

[Поддержка СОМ компилятором](#)

_com_raise_error

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Вызывает исключение [_com_error](#) в ответ на сбой.

Синтаксис

```
void __stdcall _com_raise_error(
    HRESULT hr,
    IErrorInfo* perrinfo = 0
);
```

Параметры

hr

Сведения HRESULT.

перринфо

Объект [IErrorInfo](#).

Remarks

`_com_raise_error`, который определен в `<comdef.h>`, может быть заменен на написанную пользователем версию с тем же именем и прототипом. Это можно сделать, если требуется использовать `#import` без обработки исключений C++. В этом случае пользовательская версия `_com_raise_error` может решить `longjmp` или отобразить окно сообщения и остановить. Однако пользовательская версия не должна возвращаться, поскольку код поддержки COM в компиляторе не ожидает ее возврата.

Можно также использовать `_set_com_error_handler` для замены функции обработки ошибок по умолчанию.

По умолчанию `_com_raise_error` определяется следующим образом:

```
void __stdcall _com_raise_error(HRESULT hr, IErrorInfo* perrinfo) {
    throw _com_error(hr, perrinfo);
}
```

Завершение блока, относящегося только к системам Майкрософт

Требования

Заголовок: `<comdef.h>`

Библиотека: Если `wchar_t` является параметром компилятора `native Type`, используется `комсуппв.lib` или `комсуппвд.lib`. Если `wchar_t` имеет собственный тип, то используйте `комсупп.lib`.

Дополнительные сведения см. в разделе [/Zc: wchar_t \(wchar_t является собственным типом\)](#).

См. также

[Глобальные функции компилятора COM](#)

_set_com_error_handler

ConvertStringToBSTR

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Преобразует значение типа `char *` в значение типа `BSTR`.

Синтаксис

```
BSTR __stdcall ConvertStringToBSTR(const char* pSrc)
```

Параметры

pSrc

`char *` Переменная.

Пример

```
// ConvertStringToBSTR.cpp
#include <comutil.h>
#include <stdio.h>

#pragma comment(lib, "comsuppw.lib")
#pragma comment(lib, "kernel32.lib")

int main() {
    char* lpszText = "Test";
    printf_s("char * text: %s\n", lpszText);

    BSTR bstrText = _com_util::ConvertStringToBSTR(lpszText);
    wprintf_s(L"BSTR text: %s\n", bstrText);

    SysFreeString(bstrText);
}
```

```
char * text: Test
BSTR text: Test
```

Завершение блока, относящегося только к системам Майкрософт

Требования

Заголовок: <comutil.h>

Lib: комсуппв. lib или комсуппвд. lib (Дополнительные сведения см. в разделе [/Zc: wchar_t \(wchar_t является собственным типом\)](#))

См. также

[Глобальные функции компилятора COM](#)

ConvertBSTRToString

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Преобразует значение типа `BSTR` в значение типа `char *`.

Синтаксис

```
char* __stdcall ConvertBSTRToString(BSTR pSrc);
```

Параметры

pSrc

Переменная `BSTR`.

Remarks

Конвертбстртостринг выделяет строку, которую необходимо удалить.

Пример

```
// ConvertBSTRToString.cpp
#include <comutil.h>
#include <stdio.h>

#pragma comment(lib, "comsuppw.lib")

int main() {
    BSTR bstrText = ::SysAllocString(L"Test");
    wprintf_s(L"BSTR text: %s\n", bstrText);

    char* lpszText2 = _com_util::ConvertBSTRToString(bstrText);
    printf_s("char * text: %s\n", lpszText2);

    SysFreeString(bstrText);
    delete[] lpszText2;
}
```

```
BSTR text: Test
char * text: Test
```

Завершение блока, относящегося только к системам Майкрософт

Требования

Заголовок: `<comutil.h>`

Lib: комсуппв. lib или комсуппвд. lib (Дополнительные сведения см. в разделе [/Zc: wchar_t \(wchar_t является собственным типом\)](#))

См. также раздел

Глобальные функции компилятора COM

_set_com_error_handler

12.11.2021 • 2 minutes to read

Заменяет функцию по умолчанию, используемую для обработки ошибок COM. `_set_com_error_handler` зависит от корпорации Майкрософт.

Синтаксис

```
void __stdcall _set_com_error_handler(
    void (__stdcall *pHandler)(
        HRESULT hr,
        IErrorInfo* perrinfo
    )
);
```

Параметры

фандлер

Указатель на функцию замены.

hr

Сведения HRESULT.

перринфо

Объект `IErrorInfo`.

Remarks

По умолчанию `_com_raise_error` обрабатывает все ошибки COM. Это поведение можно изменить с помощью `_set_com_error_handler` для вызова собственной функции обработки ошибок.

Функция замены должна иметь сигнатуру, эквивалентную сигнатуре `_com_raise_error`.

Пример

```

// _set_com_error_handler.cpp
// compile with /EHsc
#include <stdio.h>
#include <comdef.h>
#include <comutil.h>

// Importing ado dll to attempt to establish an ado connection.
// Not related to _set_com_error_handler
#import "C:\Program Files\Common Files\System\ado\msado15.dll" no_namespace rename("EOF", "adoEOF")

void __stdcall _My_com_raise_error(HRESULT hr, IErrorInfo* perrinfo)
{
    throw "Unable to establish the connection!";
}

int main()
{
    _set_com_error_handler(_My_com_raise_error);
    _bstr_t bstrEmpty(L"");
    _ConnectionPtr Connection = NULL;
    try
    {
        Connection.CreateInstance(__uuidof(Connection));
        Connection->Open(bstrEmpty, bstrEmpty, bstrEmpty, 0);
    }
    catch(char* errorMessage)
    {
        printf("Exception raised: %s\n", errorMessage);
    }

    return 0;
}

```

Exception raised: Unable to establish the connection!

Требования

Заголовок: <comdef.h>

Библиотека: Если указан параметр компилятора /Zc: wchar_t (по умолчанию), используйте комсуппв.lib или комсуппвд.lib. Если указан параметр /Zc: wchar_t- Compiler, используйте комсупп.lib. Дополнительные сведения, в том числе о том, как задать этот параметр в интегрированной среде разработки, см. в разделе [/Zc: wchar_t \(wchar_t является собственным типом\)](#).

См. также раздел

[Глобальные функции компилятора COM](#)

Классы поддержки компилятора COM

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Стандартные классы используются для поддержки некоторых типов модели COM. Классы определяются в `<comdef.h>` и файлы заголовков, созданные из библиотеки типов.

КЛАСС	ОПИСАНИЕ
<code>_bstr_t</code>	Создает программу оболочку для типа <code>BSTR</code> , предоставляя полезные операторы и методы.
<code>_com_error</code>	Определяет объект <code>Error</code> ,ываемый <code>_com_raise_error</code> в большинстве сбоев.
<code>_com_ptr_t</code>	Инкапсулирует указатели на COM-интерфейсы и автоматизирует необходимые вызовы в <code>AddRef</code> , <code>Release</code> и <code>QueryInterface</code> .
<code>_variant_t</code>	Создает программу оболочку для типа <code>VARIANT</code> , предоставляя полезные операторы и методы.

Завершение блока, относящегося только к системам Microsoft

См. также

[Поддержка COM компилятором](#)

[Глобальные функции компилятора COM](#)

[Справочник по языку C++](#)

Класс `_bstr_t`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

`_bstr_t` Объект инкапсулирует [тип данных BSTR](#). Класс управляет выделением и освобождением ресурсов с помощью вызовов функций `SysAllocString` и `SysFreeString` других `BSTR` API, если это уместно. `_bstr_t` Класс использует подсчет ссылок во избежание чрезмерных издержек.

Элементы

Строительство

КОНСТРУКТОР	ОПИСАНИЕ
<code>_bstr_t</code>	Формирует объект <code>_bstr_t</code> .

Операции

ФУНКЦИЯ	ОПИСАНИЕ
<code>Assign</code>	Копирует строку <code>BSTR</code> в строку <code>BSTR</code> , инкапсулированную объектом <code>_bstr_t</code> .
<code>Attach</code>	Связывает упаковщик <code>_bstr_t</code> со строкой <code>BSTR</code> .
<code>copy</code>	Создает копию инкапсулированного объекта <code>BSTR</code> .
<code>Detach</code>	Возвращает строку <code>BSTR</code> , инкапсулированную объектом <code>_bstr_t</code> , и отсоединяет ее (<code>BSTR</code>) от этого объекта (<code>_bstr_t</code>).
<code>GetAddress</code>	Указывает на строку <code>BSTR</code> , инкапсулированную объектом <code>_bstr_t</code> .
<code>GetBSTR</code>	Указывает на начало строки <code>BSTR</code> , инкапсулированной объектом <code>_bstr_t</code> .
<code>length</code>	Возвращает число символов в объекте <code>_bstr_t</code> .

Операторы

ОПЕРАТОР	ОПИСАНИЕ
<code>operator =</code>	Присваивает новое значение существующему объекту <code>_bstr_t</code> .
<code>operator +=</code>	Добавляет символы в конец объекта <code>_bstr_t</code> .

ОПЕРАТОР	ОПИСАНИЕ
<code>operator +</code>	Объединяет две строки.
<code>operator !</code>	Проверяет, является ли Инкапсулированная <code>BSTR</code> строка пустой.
<code>operator ==</code> <code>operator !=</code> <code>operator <</code> <code>operator ></code> <code>operator <=</code> <code>operator >=</code>	Сравнивает два объекта <code>_bstr_t</code> .
<code>operator wchar_t*</code> <code>operator char*</code>	Извлекает указатели на инкапсулированный объект Юникода или многобайтовый объект <code>BSTR</code> .

Завершение блока, относящегося только к системам Майкрософт

Требования

Заголовок: <comutil.h>

Библиотека: `comsuppw.Lib` ИЛИ `comsuppwd.Lib` (Дополнительные сведения см. в разделе [/Zc:wchar_t](#) (`wchar_t` является собственным типом)).

См. также раздел

[Классы поддержки СОМ компилятора](#)

`_bstr_t::_bstr_t`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Формирует объект `_bstr_t`.

Синтаксис

```
_bstr_t( ) throw( );
_bstr_t(
    const _bstr_t& s1
) throw( );
_bstr_t(
    const char* s2
);
_bstr_t(
    const wchar_t* s3
);
_bstr_t(
    const _variant_t& var
);
_bstr_t(
    BSTR bstr,
    bool fCopy
);
```

Параметры

`s1`

Копируемый объект `_bstr_t`.

`s2`

Многобайтовая строка.

`s3`

Строка Юникода.

`var`

Объект `_variant_t`.

`bstr`

Существующий объект `BSTR`.

`fCopy`

Если `false` значение `bstr` равно, аргумент прикрепляется к новому объекту без создания копии путем вызова метода `SysAllocString`.

Remarks

`_bstr_t` Класс предоставляет несколько конструкторов:

`_bstr_t()`

Конструирует объект по умолчанию `_bstr_t`, инкапсулирующий `BSTR` объект null.

```
_bstr_t( _bstr_t& s1 )
```

Создает объект `_bstr_t` как копию другого объекта. Этот конструктор создает *неполную* копию, которая увеличивает счетчик ссылок инкапсулированного `BSTR` объекта, а не создает новый.

```
_bstr_t( char* s2 )
```

Создает новый объект `_bstr_t`, вызывая функцию `SysAllocString` для создания нового объекта `BSTR`, а затем инкапсулирует его. Этот конструктор конструктор сначала преобразует многобайтовую строку в строку Юникода.

```
_bstr_t( wchar_t* s3 )
```

Создает новый объект `_bstr_t`, вызывая функцию `SysAllocString` для создания нового объекта `BSTR`, а затем инкапсулирует его.

```
_bstr_t( _variant_t& var )
```

Конструирует `_bstr_t` объект из `_variant_t` объекта, сначала получая `BSTR` объект из инкапсулированного `VARIANT` объекта.

```
_bstr_t( BSTR bstr, bool fCopy )
```

Создает объект `_bstr_t` из существующего объекта `BSTR` (а не из строки `wchar_t*`). Если `fCopy` имеет значение `false`, предоставленный `BSTR` объект присоединяется к новому объекту без создания новой копии с помощью `SysAllocString`. Этот конструктор используется функциями-оболочками в заголовках библиотек типов для инкапсуляции и получения владения объектом `BSTR`, возвращаемым методом интерфейса.

Завершение блока, относящегося только к системам Microsoft

См. также

`_bstr_t` CM

Класс `_variant_t`

`_bstr_t::Assign`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Копирует строку `BSTR` в строку `BSTR`, инкапсулированную объектом `_bstr_t`.

Синтаксис

```
void Assign(  
    BSTR s  
) ;
```

Параметры

`s`

Объект `BSTR`, копируемый в объект `BSTR`, инкапсулированный объектом `_bstr_t`.

Remarks

`Assign` Выполняет двоичную копию всей длины `BSTR`, независимо от содержимого.

Пример

```

// _bstr_t_Assign.cpp

#include <comdef.h>
#include <stdio.h>

int main()
{
    // creates a _bstr_t wrapper
    _bstr_t bstrWrapper;

    // creates BSTR and attaches to it
    bstrWrapper = "some text";
    wprintf_s(L"bstrWrapper = %s\n",
             static_cast<wchar_t*>(bstrWrapper));

    // bstrWrapper releases its BSTR
    BSTR bstr = bstrWrapper.Detach();
    wprintf_s(L"bstrWrapper = %s\n",
             static_cast<wchar_t*>(bstrWrapper));
    // "some text"
    wprintf_s(L"bstr = %s\n", bstr);

    bstrWrapper.Attach(SysAllocString(OLESTR("SysAllocatedString")));
    wprintf_s(L"bstrWrapper = %s\n",
             static_cast<wchar_t*>(bstrWrapper));

    // assign a BSTR to our _bstr_t
    bstrWrapper.Assign(bstr);
    wprintf_s(L"bstrWrapper = %s\n",
             static_cast<wchar_t*>(bstrWrapper));

    // done with BSTR, do manual cleanup
    SysFreeString(bstr);

    // resuse bstr
    bstr= SysAllocString(OLESTR("Yet another string"));
    // two wrappers, one BSTR
    _bstr_t bstrWrapper2 = bstrWrapper;

    *bstrWrapper.GetAddress() = bstr;

    // bstrWrapper and bstrWrapper2 do still point to BSTR
    bstr = 0;
    wprintf_s(L"bstrWrapper = %s\n",
             static_cast<wchar_t*>(bstrWrapper));
    wprintf_s(L"bstrWrapper2 = %s\n",
             static_cast<wchar_t*>(bstrWrapper2));

    // new value into BSTR
    _snwprintf_s(bstrWrapper.GetBSTR(), 100, bstrWrapper.length(),
                 L"changing BSTR");
    wprintf_s(L"bstrWrapper = %s\n",
             static_cast<wchar_t*>(bstrWrapper));
    wprintf_s(L"bstrWrapper2 = %s\n",
             static_cast<wchar_t*>(bstrWrapper2));
}

```

```
bstrWrapper = some text
bstrWrapper = (null)
bstr = some text
bstrWrapper = SysAllocedString
bstrWrapper = some text
bstrWrapper = Yet another string
bstrWrapper2 = some text
bstrWrapper = changing BSTR
bstrWrapper2 = some text
```

Завершение блока, относящегося только к системам Майкрософт

См. также

Класс [_bstr_t](#)

`_bstr_t::Attach`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Связывает упаковщик `_bstr_t` со строкой `BSTR`.

Синтаксис

```
void Attach(  
    BSTR s  
) ;
```

Параметры

`s`

Ресурс `BSTR` для связывания с переменной `_bstr_t` или назначения ей.

Remarks

Если переменная `_bstr_t` была ранее присоединена к другому ресурсу `BSTR`, `_bstr_t` очистит ресурсы `BSTR`, если другие переменные `_bstr_t` не используют `BSTR`.

Пример

[`_bstr_t::Assign`](#) Пример использования см. в разделе `Attach`.

Завершение блока, относящегося только к системам Microsoft

См. также

[`_bstr_t`](#) См

`_bstr_t::copy`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Создает копию инкапсулированного объекта `BSTR`.

Синтаксис

```
BSTR copy( bool fCopy = true ) const;
```

Параметры

`fCopy`

Если `true` значение `copy` равно, функция возвращает копию вложенного объекта `BSTR`, в противном случае `copy` возвращает фактическую `BSTR`.

Remarks

Возвращает только что выделенную копию инкапсулированного `BSTR` объекта или самого инкапсулированного объекта в зависимости от параметра.

Пример

```
STDMETHODIMP CAlertMsg::get_ConnectionStr(BSTR *pVal){ // m_bsConStr is _bstr_t
    *pVal = m_bsConStr.copy();
}
```

Завершение блока, относящегося только к системам Майкрософт

См. также

Класс `_bstr_t`

`_bstr_t::Detach`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Возвращает строку `BSTR`, инкапсулированную объектом `_bstr_t`, и отсоединяет ее (`BSTR`) от этого объекта (`_bstr_t`).

Синтаксис

```
BSTR Detach( ) throw;
```

Возвращаемое значение

Возвращает объект, `BSTR` инкапсулированный в `_bstr_t`.

Пример

См [`_bstr_t::Assign`](#). пример, в котором используется `Detach`.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс `_bstr_t`](#)

`_bstr_t::GetAddress`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Освобождает любую существующую строку и возвращает адрес новой строки, которой была выделена память.

Синтаксис

```
BSTR* GetAddress( );
```

Возвращаемое значение

Указатель на строку `BSTR`, инкапсулированную объектом `_bstr_t`.

Remarks

`GetAddress` затрагивает все `_bstr_t` объекты, которые совместно используют `BSTR`. `_bstr_t` С помощью `BSTR` конструктора копирования и может совместно использовать несколько экземпляров `operator=`.

Пример

См [`_bstr_t::Assign`](#). пример, в котором используется `GetAddress`.

Завершение блока, относящегося только к системам Майкрософт

См. также

Класс [`_bstr_t`](#)

`_bstr_t::GetBSTR`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Указывает на начало строки `BSTR`, инкапсулированной объектом `_bstr_t`.

Синтаксис

```
BSTR& GetBSTR( );
```

Возвращаемое значение

Начало строки `BSTR`, инкапсулированной объектом `_bstr_t`.

Remarks

`GetBSTR` затрагивает все `_bstr_t` объекты, которые совместно используют `BSTR`. `_bstr_t` С помощью `BSTR` конструктора копирования и может совместно использовать несколько экземпляров `operator=`.

Пример

См `_bstr_t::Assign`. пример, в котором используется `GetBSTR`.

Завершение блока, относящегося только к системам Майкрософт

См. также

Класс `_bstr_t`

`_bstr_t::length`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Возвращает число символов в объекте `_bstr_t` инкапсулированной строки `BSTR` без учета завершающего нуль-символа.

Синтаксис

```
unsigned int length( ) const throw( );
```

Remarks

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс `_bstr_t`](#)

`_bstr_t::operator =`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Присваивает новое значение существующему объекту `_bstr_t`.

Синтаксис

```
_bstr_t& operator=(const _bstr_t& s1) throw ( );
_bstr_t& operator=(const char* s2);
_bstr_t& operator=(const wchar_t* s3);
_bstr_t& operator=(const _variant_t& var);
```

Параметры

`s1`

Объект `_bstr_t`, который требуется присвоить существующему объекту `_bstr_t`.

`s2`

Многобайтовая строка, которую требуется присвоить существующему объекту `_bstr_t`.

`s3`

Строка Юникода, которую требуется присвоить существующему объекту `_bstr_t`.

`var`

Объект `_variant_t`, который требуется присвоить существующему объекту `_bstr_t`.

Завершение блока, относящегося только к системам Майкрософт

Пример

См. [_bstr_t::Assign](#). пример, в котором используется `operator=`.

См. также:

[Класс `_bstr_t`](#)

`_bstr_t::operator +=`, `_bstr_t::operator +`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Добавляет символы в конец `_bstr_t` объекта или объединяет две строки.

Синтаксис

```
_bstr_t& operator+=( const _bstr_t& s1 );
_bstr_t operator+( const _bstr_t& s1 );
friend _bstr_t operator+( const char* s2, const _bstr_t& s1);
friend _bstr_t operator+( const wchar_t* s3, const _bstr_t& s1);
```

Параметры

`s1`

Объект `_bstr_t`.

`s2`

Многобайтовая строка.

`s3`

Строка Юникода.

Remarks

Эти операторы выполняют объединение строк:

- `operator+=(s1)` Добавляет символы в инкапсулированный в `BSTR` `s1` конец инкапсулированного объекта `BSTR`.
- `operator+(s1)` Возвращает новый `_bstr_t` объект, сформированный путем сцепления этого объекта с `BSTR` и его в `s1`.
- `operator+(s2, s1)` Возвращает новый объект `_bstr_t`, сформированный путем сцепления многобайтовой строки `s2`, преобразованной в Юникод, и `BSTR` инкапсулированного в `s1`.
- `operator+(s3, s1)` Возвращает новый объект `_bstr_t`, сформированный путем сцепления строки в Юникоде `s3` и `BSTR` инкапсулированной в `s1`.

Завершение блока, относящегося только к системам Майкрософт

См. также

Класс `_bstr_t`

`_bstr_t::operator !`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Проверяет, является ли Инкапсулированная `BSTR` строка пустой.

Синтаксис

```
bool operator!( ) const throw( );
```

Возвращаемое значение

Он возвращает `true` значение, если инкапсулированный `BSTR` является строкой null, `false` Если нет.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс `_bstr_t`](#)

`_bstr_t`

Операторы отношения

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Сравнивает два объекта `_bstr_t`.

Синтаксис

```
bool operator==(const _bstr_t& str) const throw( );
bool operator!=(const _bstr_t& str) const throw( );
bool operator<(const _bstr_t& str) const throw( );
bool operator>(const _bstr_t& str) const throw( );
bool operator<=(const _bstr_t& str) const throw( );
bool operator>=(const _bstr_t& str) const throw( );
```

Remarks

Эти операторы производят лексикографическое сравнение двух объектов `_bstr_t`. Операторы возвращают `true`, если сравнения содержат, в противном случае возвращает `false`.

Завершение блока, относящегося только к системам Майкрософт

См. также

Класс `_bstr_t`

`_bstr_t::wchar_t*`, `_bstr_t::char*`

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Возвращает `BSTR` символы в виде узких или расширенных массивов символов.

Синтаксис

```
operator const wchar_t*( ) const throw( );
operator wchar_t*( ) const throw( );
operator const char*( ) const;
operator char*( ) const;
```

Remarks

Эти операторы можно использовать для извлечения символьных данных, инкапсулированных `BSTR` объектом. Присвоение возвращенному указателю нового значения не приводит к изменению исходных `BSTR` данных.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс `_bstr_t`](#)

Класс _com_error

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Объект `_com_error` представляет условие исключения, обнаруженнное функциями-оболочками обработки ошибок в файлах заголовков, созданных из библиотеки типов или одним из классов поддержки COM. Класс `_com_error` инкапсулирует код ошибки HRESULT и любой связанный `IErrorInfo Interface` объект.

Строительство

Имя	Описание
<code>_com_error</code>	Конструирует объект <code>_com_error</code> .

Операторы

Имя	Описание
<code>Оператор =</code>	Присваивает существующий объект <code>_com_error</code> другому объекту.

Функции извлечения

Имя	Описание
<code>Ошибка</code>	Извлекает значение HRESULT, передаваемое конструктору.
<code>ErrorInfo</code>	Получает объект <code>IErrorInfo</code> , переданный конструктору.
<code>WCode</code>	Извлекает 16-разрядный код ошибки, сопоставленный с инкапсульзованным значением HRESULT.

Функции `IErrorInfo`

Имя	Описание
<code>Описание</code>	Вызывает функцию <code>IErrorInfo::GetDescription</code> .
<code>HelpContext</code>	Вызывает функцию <code>IErrorInfo::GetHelpContext</code> .
<code>HelpFile</code>	Вызывает функцию <code>IErrorInfo::GetHelpFile</code> .
<code>Источник</code>	Вызывает функцию <code>IErrorInfo::GetSource</code> .
<code>GUID</code>	Вызывает функцию <code>IErrorInfo::GetGUID</code> .

Извлечение сообщения формата

Имя	Описание
ErrorMessage	Извлекает строковое сообщение для HRESULT, хранящегося в объекте _com_error .

Средства сопоставления ExepInfo.wCode с HRESULT

Имя	Описание
HRESULTToWCode	Карты 32-бит HRESULT в 16-разрядный wCode .
WCodeToHRESULT	Карты wCode от 16 до 32-bit HRESULT.

Завершение блока, относящегося только к системам Майкрософт

Требования

Заголовок: <comdef.h>

Lib: комсуппв. lib или комсуппвд. lib (Дополнительные сведения см. в разделе [/Zc: wchar_t \(wchar_t является собственным типом\)](#))

См. также:

[Классы поддержки СОМ компилятора](#)

[Интерфейс IErrorInfo](#)

Функции-члены _com_error

12.11.2021 • 2 minutes to read

Дополнительные сведения о функциях элементов `_com_error` см. в разделе [класс `_com_error`](#).

См. также:

[Класс `_com_error`](#)

_com_error::_com_error

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Конструирует объект _com_error .

Синтаксис

```
_com_error(
    HRESULT hr,
    IErrorInfo* perrinfo = NULL,
    bool fAddRef=false) throw( );

_com_error( const _com_error& that ) throw( );
```

Параметры

hr

Сведения HRESULT.

перринфо

Объект `IErrorInfo`.

фаддреф

По умолчанию конструктор вызывает AddRef для интерфейса, отличного от NULL `IErrorInfo` . Это обеспечивает правильный подсчет ссылок в общем случае, когда владение интерфейсом передается в объект `_com_error` , например:

```
throw _com_error(hr, perrinfo);
```

Если вы не хотите, чтобы код переносит владение объекту `_com_error` , а `AddRef` требуется смещение в `Release` деструкторе `_com_error` , создайте объект следующим образом:

```
_com_error err(hr, perrinfo, true);
```

что

Существующий объект `_com_error` .

Remarks

Первый конструктор создает новый объект с заданным значением HRESULT и необязательным `IErrorInfo` объектом. Вторая создает копию существующего объекта `_com_error` .

Завершение блока, относящегося только к системам Microsoft

См. также

[Класс _com_error](#)

_com_error::Description

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Вызывает функцию `IErrorInfo::GetDescription`.

Синтаксис

```
_bstr_t Description( ) const;
```

Возвращаемое значение

Возвращает результат `IErrorInfo::GetDescription` для `IErrorInfo` объекта, записанного в `_com_error` объекте. Результирующая функция `BSTR` инкапсулируется в объект `_bstr_t`. Если `IErrorInfo` запись не записана, возвращается пустое значение `_bstr_t`.

Remarks

Вызывает `IErrorInfo::GetDescription` функцию и получает `IErrorInfo` запись в `_com_error` объект. Любой сбой при вызове `IErrorInfo::GetDescription` метода игнорируется.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_error](#)

_com_error::Error

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Извлекает значение HRESULT, передаваемое конструктору.

Синтаксис

```
HRESULT Error( ) const throw( );
```

Возвращаемое значение

Необработанный элемент HRESULT, переданный в конструктор.

Remarks

Извлекает инкапсулированный элемент HRESULT в `_com_error` объекте.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_error](#)

_com_error::ErrorInfo

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Получает объект `IErrorInfo`, переданный конструктору.

Синтаксис

```
IErrorInfo * ErrorInfo( ) const throw( );
```

Возвращаемое значение

Необработанный элемент `IErrorInfo`, переданный в конструктор.

Remarks

Извлекает инкапсулированный `IErrorInfo` элемент в `_com_error` объекте или значение null, если элемент не `IErrorInfo` записан. Вызывающий объект должен вызвать `Release` для возвращенного объекта, когда он завершил использовать его.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_error](#)

_com_error::ErrorMessage

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Получает строковое сообщение для значения HRESULT, хранящегося в объекте `_com_error`.

Синтаксис

```
const TCHAR * ErrorMessage( ) const throw( );
```

Возвращаемое значение

Возвращает строковое сообщение для HRESULT, записанного в `_com_error` объекте. Если HRESULT является сопоставленным 16-битным `wCode`, возвращается универсальное сообщение "`IDispatch error #<wCode>`". Если сообщение не найдено, возвращается универсальное сообщение "`Unknown error #<HRESULT>`". В зависимости от состояния макроса `_UNICODE`, возвращается строка Юникода или многобайтовая строка.

Remarks

Извлекает соответствующий текст системного сообщения для HRESULT, записанного в `_com_error` объекте. Текст системного сообщения получается путем вызова функции "Win32 `FormatMessage`". Возвращаемая строка выделяется API `FormatMessage`; эта строка освобождается при уничтожении объекта `_com_error`.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_error](#)

_com_error::GUID

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Вызывает функцию `IErrorInfo::GetGUID`.

Синтаксис

```
GUID GUID( ) const throw( );
```

Возвращаемое значение

Возвращает результат `IErrorInfo::GetGUID` для `IErrorInfo` объекта, записанного в `_com_error` объекте. Если `IErrorInfo` объект не записан, возвращается значение `GUID_NULL`.

Remarks

Любой сбой при вызове `IErrorInfo::GetGUID` метода игнорируется.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_error](#)

_com_error::HelpContext

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Вызывает функцию `IErrorInfo::GetHelpContext`.

Синтаксис

```
DWORD HelpContext( ) const throw( );
```

Возвращаемое значение

Возвращает результат `IErrorInfo::GetHelpContext` для `IErrorInfo` объекта, записанного в `_com_error` объекте. Если `IErrorInfo` объект не записан, возвращается ноль.

Remarks

Любой сбой при вызове `IErrorInfo::GetHelpContext` метода игнорируется.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_error](#)

_com_error::HelpFile

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Вызывает функцию `IErrorInfo::GetHelpFile`.

Синтаксис

```
_bstr_t HelpFile() const;
```

Возвращаемое значение

Возвращает результат `IErrorInfo::GetHelpFile` для `IErrorInfo` объекта, записанного в `_com_error` объекте. Результирующая строка BSTR инкапсулируется в объект `_bstr_t`. Если `IErrorInfo` запись не записана, возвращается пустое значение `_bstr_t`.

Remarks

Любой сбой при вызове `IErrorInfo::GetHelpFile` метода игнорируется.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_error](#)

_com_error::HRESULTToWCode

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Карты 32-бит HRESULT в 16-разрядный `wCode`.

Синтаксис

```
static WORD HRESULTToWCode(
    HRESULT hr
) throw( );
```

Параметры

hr

32-разрядное значение HRESULT, которое должно быть сопоставлено с 16-битным значением `wCode`.

Возвращаемое значение

16-разрядный `wCode` параметр, сопоставленный с 32-битным значением HRESULT.

Remarks

Дополнительные сведения см. в разделе [_com_error::wCode](#).

Завершение блока, относящегося только к системам Майкрософт

См. также

[_com_error::WCode](#)

[_com_error::WCodeToHRESULT](#)

[Класс _com_error](#)

_com_error::Source

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Вызывает функцию `IErrorInfo::GetSource`.

Синтаксис

```
_bstr_t Source() const;
```

Возвращаемое значение

Возвращает результат `IErrorInfo::GetSource` для `IErrorInfo` объекта, записанного в `_com_error` объекте. Результирующая функция `BSTR` инкапсулируется в объект `_bstr_t`. Если `IErrorInfo` запись не записана, возвращается пустое значение `_bstr_t`.

Remarks

Любой сбой при вызове `IErrorInfo::GetSource` метода игнорируется.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_error](#)

_com_error::WCode

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Извлекает 16-разрядный код ошибки, сопоставленный с инкапсулированным значением HRESULT.

Синтаксис

```
WORD WCode( ) const throw( );
```

Возвращаемое значение

Если значение HRESULT находится в диапазоне от 0x80040200 до 0x8004FFFF, `wCode` метод возвращает значение HRESULT минус 0x80040200. В противном случае возвращается ноль.

Remarks

`WCode` Метод используется для отмены сопоставления, происходящего в коде поддержки COM. Оболочка для `dispinterface` свойства или метода вызывает подпрограмму поддержки, которая упаковывает аргументы и вызывает `IDispatch::Invoke`. При возврате, если возвращается ошибка HRESULT of `DISP_E_EXCEPTION`, сведения об ошибке извлекаются из `EXCEPINFO` структуры, передаваемой в `IDispatch::Invoke`. Код ошибки может быть 16-битным значением, хранящимся в элементе `wCode` `EXCEPINFO` структуры, или полным 32-битным значением в элементе `scode` `EXCEPINFO` структуры. Если возвращается 16-разрядный объект `wCode`, он сначала должен быть сопоставлен с ЗНАЧЕНИЕМ HRESULT 32-разрядного сбоя.

Завершение блока, относящегося только к системам Microsoft

См. также

[_com_error::HRESULTToWCode](#)

[_com_error::WCodeToHRESULT](#)

[Класс _com_error](#)

_com_error::WCodeToHRESULT

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Карты 16-разрядное *wCode* в 32-разрядное значение HRESULT.

Синтаксис

```
static HRESULT WCodeToHRESULT(
    WORD wCode
) throw( );
```

Параметры

wCode

16-разрядный *wCode* сопоставляется с 32-БИТНЫМ значением HRESULT.

Возвращаемое значение

32-разрядный HRESULT, сопоставленный из 16-разрядного *wCode*.

Remarks

См. функцию члена [wCode](#).

Завершение блока, относящегося только к системам Майкрософт

См. также

[_com_error::WCode](#)

[_com_error::HRESULTToWCode](#)

[Класс _com_error](#)

Операторы _com_error

12.11.2021 • 2 minutes to read

Дополнительные сведения об операторах _com_error см. в разделе [класс _com_error](#).

См. также

[Класс _com_error](#)

_com_error::operator =

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Присваивает существующий объект `_com_error` другому объекту.

Синтаксис

```
_com_error& operator = (
    const _com_error& that
) throw ( );
```

Параметры

что

Объект `_com_error`.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_error](#)

Класс _com_ptr_t

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Объект _com_ptr_t инкапсулирует указатель на интерфейс COM и называется "смарт-указателем". Этот класс шаблона управляет выделением и освобождением ресурсов с помощью вызовов функций в функциях- членах: `QueryInterface` , `AddRef` И `Release` .

Ссылка на интеллектуальный указатель обычно осуществляется с помощью определения `typedef`, предоставляемого макросом `_COM_SMARTPTR_TYPEDEF`. Этот макрос принимает имя интерфейса и IID и объявляет специализацию `_com_ptr_t` с именем интерфейса и суффиксом `Ptr` . Пример:

```
_COM_SMARTPTR_TYPEDEF(IMyInterface, __uuidof(IMyInterface));
```

объявляет `_com_ptr_t` специализацию `IMyInterfacePtr` .

Набор [шаблонов функций](#), не входящих в этот класс шаблона, поддерживает сравнение со смарт-указателем в правой части оператора сравнения.

Строительство

Имя	Описание:
<code>_com_ptr_t</code>	Конструирует объект <code>_com_ptr_t</code> .

Низкоуровневые операции

Имя	Описание:
<code>AddRef</code>	Вызывает <code>AddRef</code> функцию члена <code>IUnknown</code> в инкапсулированном указателе интерфейса.
<code>Attach</code>	Инкапсулирует необработанный указатель на интерфейс для типа этого интеллектуального указателя.
<code>CreateInstance</code>	Создает новый экземпляр объекта с заданным <code>CLSID</code> или <code>ProgID</code> .
<code>Отсоединить</code>	Извлекает и возвращает инкапсулированный указатель на интерфейс.
<code>GetActiveObject</code>	Присоединяет к существующему экземпляру объекта, заданному <code>CLSID</code> или <code>ProgID</code> .
<code>GetInterfacePtr</code>	Возвращает инкапсулированный указатель на интерфейс.
<code>QueryInterface</code>	Вызывает <code>QueryInterface</code> функцию члена <code>IUnknown</code> в инкапсулированном указателе интерфейса.

ИМЯ	ОПИСАНИЕ:
Релиз	Вызывает <code>Release</code> функцию члена <code>IUnknown</code> в инкапсулированном указателе интерфейса.

Операторы

ИМЯ	ОПИСАНИЕ:
Оператор =	Присваивает новое значение существующему объекту <code>_com_ptr_t</code> .
операторы = =,! =, <, > , <=, >=	Сравнивают объект интеллектуального указателя с другим интеллектуальным указателем, необработанным указателем на интерфейс или значением NULL.
Средства извлечения	Извлекают инкапсулированный указатель на СОМ-интерфейс.

Завершение блока, относящегося только к системам Майкрософт

Требования

Заголовок: <comip.h>

Lib: комсуппв. lib или комсуппвд. lib (Дополнительные сведения см. в разделе [/Zc: wchar_t \(wchar_t является собственным типом\)](#))

См. также раздел

[Классы поддержки СОМ компилятора](#)

Функции-члены _com_ptr_t

12.11.2021 • 2 minutes to read

Дополнительные сведения о функциях элементов _com_ptr_t см. в разделе [класс _com_ptr_t](#).

См. также

[Класс _com_ptr_t](#)

_com_ptr_t::_com_ptr_t

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Конструирует объект _com_ptr_t .

Синтаксис

```
// Default constructor.  
// Constructs a NULL smart pointer.  
_com_ptr_t() throw();  
  
// Constructs a NULL smart pointer. The NULL argument must be zero.  
_com_ptr_t(  
    int null  
);  
  
// Constructs a smart pointer as a copy of another instance of the  
// same smart pointer. AddRef is called to increment the reference  
// count for the encapsulated interface pointer.  
_com_ptr_t(  
    const _com_ptr_t& cp  
) throw();  
  
// Move constructor (Visual Studio 2015 Update 3 and later)  
_com_ptr_t(_com_ptr_t&& cp) throw();  
  
// Constructs a smart pointer from a raw interface pointer of this  
// smart pointer's type. If fAddRef is true, AddRef is called  
// to increment the reference count for the encapsulated  
// interface pointer. If fAddRef is false, this constructor  
// takes ownership of the raw interface pointer without calling AddRef.  
_com_ptr_t(  
    Interface* pInterface,  
    bool fAddRef  
) throw();  
  
// Construct pointer for a _variant_t object.  
// Constructs a smart pointer from a _variant_t object. The  
// encapsulated VARIANT must be of type VT_DISPATCH or VT_UNKNOWN, or  
// it can be converted into one of these two types. If QueryInterface  
// fails with an E_NOINTERFACE error, a NULL smart pointer is  
// constructed.  
_com_ptr_t(  
    const _variant_t& varSrc  
);  
  
// Constructs a smart pointer given the CLSID of a coclass. This  
// function calls CoCreateInstance, by the member function  
// CreateInstance, to create a new COM object and then queries for  
// this smart pointer's interface type. If QueryInterface fails with  
// an E_NOINTERFACE error, a NULL smart pointer is constructed.  
explicit _com_ptr_t(  
    const CLSID& clsid,  
    IUnknown* pOuter = NULL,  
    DWORD dwClssContext = CLSCTX_ALL  
);  
  
// Calls CoCreateClass with provided CLSID retrieved from string.  
explicit _com_ptr_t(
```

```

LPCWSTR str,
IUnknown* pOuter = NULL,
DWORD dwClsContext = CLSCTX_ALL
);

// Constructs a smart pointer given a multibyte character string that
// holds either a CLSID (starting with "{") or a ProgID. This function
// calls CoCreateInstance, by the member function CreateInstance, to
// create a new COM object and then queries for this smart pointer's
// interface type. If QueryInterface fails with an E_NOINTERFACE error,
// a NULL smart pointer is constructed.
explicit _com_ptr_t(
    LPCSTR str,
    IUnknown* pOuter = NULL,
    DWORD dwClsContext = CLSCTX_ALL
);

// Saves the interface.
template<>
_com_ptr_t(
    Interface* pInterface
) throw();

// Make sure correct ctor is called
template<>
_com_ptr_t(
    LPSTR str
);

// Make sure correct ctor is called
template<>
_com_ptr_t(
    LPWSTR str
);

// Constructs a smart pointer from a different smart pointer type or
// from a different raw interface pointer. QueryInterface is called to
// find an interface pointer of this smart pointer's type. If
// QueryInterface fails with an E_NOINTERFACE error, a NULL smart
// pointer is constructed.
template<typename _OtherIID>
_com_ptr_t(
    const _com_ptr_t<_OtherIID>& p
);

// Constructs a smart-pointer from any IUnknown-based interface pointer.
template<typename _InterfaceType>
_com_ptr_t(
    _InterfaceType* p
);

// Disable conversion using _com_ptr_t* specialization of
// template<typename _InterfaceType> _com_ptr_t(_InterfaceType* p)
template<>
explicit _com_ptr_t(
    _com_ptr_t* p
);

```

Параметры

интерфейс

Необработанный указатель на интерфейс.

фаддреf

Если `true` `AddRef` задано значение, вызывается метод для увеличения числа ссылок в указателе инкапсулированного интерфейса.

cp

Объект `_com_ptr_t`.

p

Необработанный указатель интерфейса, его тип отличается от типа интеллектуального указателя данного объекта `_com_ptr_t`.

varSrc

Объект `_variant_t`.

этому

Объект `CLSID` coclass.

деклеконтекст

Контекст для выполняющегося исполняемого кода.

lpCTSTR

Многобайтовая строка, содержащая либо `CLSID` (начиная с "{"), либо `ProgID`.

паутер

Внешняя неизвестная для [агрегирования](#).

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_ptr_t](#)

_com_ptr_t::AddRef

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Вызывает `AddRef` функцию члена `IUnknown` в инкапсулированном указателе интерфейса.

Синтаксис

```
void AddRef( );
```

Remarks

Вызывает `IUnknown::AddRef` указатель на инкапсулированный интерфейс, вызывая `E_POINTER` ошибку, если указатель имеет значение null.

Завершение блока, относящегося только к системам Microsoft

См. также

[Класс _com_ptr_t](#)

_com_ptr_t::Attach

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Инкапсулирует необработанный указатель на интерфейс для типа этого интеллектуального указателя.

Синтаксис

```
void Attach( Interface* pInterface ) throw( );
void Attach( Interface* pInterface, bool fAddRef ) throw( );
```

Параметры

пинтерфаце

Необработанный указатель на интерфейс.

фаддреф

Если это так `true`, то `AddRef` вызывается. Если это так `false`, `_com_ptr_t` объект принимает владение указателем необработанного интерфейса без вызова `AddRef`.

Remarks

- `Attach (пинтерфаце)` `AddRef` не вызывается. Право на владение интерфейсом передается данному объекту `_com_ptr_t`. `Release` метод вызывается для уменьшения числа ссылок для ранее инкапсулированного указателя.
- `Attach (пинтерфаце, фаддреф)` Если *фаддреф* имеет значение `true`, `AddRef` вызывается метод для увеличения числа ссылок для указателя инкапсулированного интерфейса. Если *фаддреф* имеет значение `false`, этот `_com_ptr_t` объект берет на себя владение указателем необработанного интерфейса без вызова `AddRef`. `Release` метод вызывается для уменьшения числа ссылок для ранее инкапсулированного указателя.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_ptr_t](#)

_com_ptr_t::CreateInstance

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Создает новый экземпляр объекта с заданным `CLSID` или `ProgID`.

Синтаксис

```
HRESULT CreateInstance(
    const CLSID& rclsid,
    IUnknown* pOuter=NULL,
    DWORD dwClsContext = CLSCTX_ALL
) throw( );
HRESULT CreateInstance(
    LPCWSTR clsidString,
    IUnknown* pOuter=NULL,
    DWORD dwClsContext = CLSCTX_ALL
) throw( );
HRESULT CreateInstance(
    LPCSTR clsidStringA,
    IUnknown* pOuter=NULL,
    DWORD dwClsContext = CLSCTX_ALL
) throw( );
```

Параметры

`рклсид`

`CLSID` Объект объекта.

`клсидстринг`

Строка в Юникоде, которая содержит значение `CLSID` (начиная с "{") или `ProgID`.

`клсидстринга`

Многобайтовая строка, использующая кодовую страницу ANSI, которая содержит значение `CLSID` (начиная с "(") или `ProgID`.

`двлсконтекст`

Контекст для выполняющегося исполняемого кода.

`паутер`

Внешняя неизвестная для [агрегирования](#).

Remarks

Эти функции-члены вызывают `CoCreateInstance` для создания нового СОМ-объекта, а затем запрашивают тип интерфейса данного интеллектуального указателя. Результатирующий указатель затем инкапсулируется в этот объект `_com_ptr_t`. `Release` метод вызывается для уменьшения числа ссылок для ранее инкапсулированного указателя. Эта подпрограмма возвращает значение `HRESULT` для обозначения успеха или неудачи.

- `CreateInstance (рклсид , двлсконтекст)` Создает новый выполняющийся экземпляр объекта, используя объект `CLSID`.
- `CreateInstance (клайдстринг , двлсконтекст)` Создает новый запущенный экземпляр объекта с

заданной строкой Юникода, содержащей значение `CLSID` (начиная с "{") или `ProgID` .

- `CreateInstance (клсидстринга, двлсконтекст)` Создает новый запущенный экземпляр объекта с заданной строкой многобайтовых символов, содержащей либо `CLSID` (начиная с "{"), либо `ProgID` . Вызывает `MultiByteToWideChar`, который предполагает, что строка находится в кодовой странице ANSI, а не на кодовой странице OEM.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс_com_ptr_t](#)

_com_ptr_t::Detach

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Извлекает и возвращает инкапсулированный указатель на интерфейс.

Синтаксис

```
Interface* Detach( ) throw( );
```

Remarks

Извлекает и возвращает инкапсулированный указатель на интерфейс, а затем очищает область хранения инкапсулированного указателя, присваивая ему значение NULL. Поэтому указатель на интерфейс удаляется из инкапсуляции. Вы должны вызвать метод `Release` в возвращенном указателе интерфейса.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_ptr_t](#)

_com_ptr_t::GetActiveObject

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Присоединяет к существующему экземпляру объекта, заданному `CLSID` или `ProgID`.

Синтаксис

```
HRESULT GetActiveObject(
    const CLSID& rclsid
) throw( );
HRESULT GetActiveObject(
    LPCWSTR clsidString
) throw( );
HRESULT GetActiveObject(
    LPCSTR clsidStringA
) throw( );
```

Параметры

`rclsid`

`CLSID` Объект объекта.

`clsidString`

Строка в Юникоде, которая содержит значение `CLSID` (начиная с "{") или `ProgID`.

`clsidStringA`

Многобайтовая строка, использующая кодовую страницу ANSI, которая содержит значение `CLSID` (начиная с "{") или `ProgID`.

Remarks

Эти функции-члены вызывают **Жетактивеобъект**, чтобы получить указатель на запущенный объект, зарегистрированный в OLE, а затем запрашивает тип интерфейса этого интеллектуального указателя. Результирующий указатель затем инкапсулируется в этот объект `_com_ptr_t`. `Release` метод вызывается для уменьшения числа ссылок для ранее инкапсулированного указателя. Эта подпрограмма возвращает значение HRESULT для обозначения успеха или неудачи.

- **Жетактивеобъект** (`rclsid`) подключается к существующему экземпляру объекта, заданному `CLSID`.
- **Жетактивеобъект** (`clsidString`) подключается к существующему экземпляру объекта при наличии строки в Юникоде, содержащей значение `CLSID` (начиная с "{") или `ProgID`.
- **Жетактивеобъект** (`clsidStringA`) подключается к существующему экземпляру объекта при наличии строки многобайтовых символов, содержащей значение `CLSID` (начиная с "{") или `ProgID`. Вызывает [MultiByteToWideChar](#), который предполагает, что строка находится в кодовой странице ANSI, а не на кодовой странице OEM.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_ptr_t](#)

_com_ptr_t::GetInterfacePtr

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Возвращает инкапсулированный указатель на интерфейс.

Синтаксис

```
Interface* GetInterfacePtr( ) const throw( );
Interface*& GetInterfacePtr() throw();
```

Remarks

Возвращает инкапсулированный указатель на интерфейс, который может иметь значение NULL.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_ptr_t](#)

_com_ptr_t::QueryInterface

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Вызывает функцию члена QueryInterface `IUnknown` в указателе инкапсулированного интерфейса.

Синтаксис

```
template<typename _InterfaceType> HRESULT QueryInterface (
    const IID& iid,
    _InterfaceType*& p
) throw ( );
template<typename _InterfaceType> HRESULT QueryInterface (
    const IID& iid,
    _InterfaceType** p
) throw( );
```

Параметры

IID

`IUnknown` указателя интерфейса.

p

Необработанный указатель на интерфейс.

Remarks

Вызывает `IUnknown::QueryInterface` указатель инкапсулированного интерфейса с указанным `IID` и возвращает полученный указатель необработанного интерфейса в *p*. Эта подпрограмма возвращает значение HRESULT для обозначения успеха или неудачи.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_ptr_t](#)

_com_ptr_t::Release

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Вызывает функцию члена **выпуска** `IUnknown` в указателе инкапсулированного интерфейса.

Синтаксис

```
void Release( );
```

Remarks

Вызывает `IUnknown::Release` указатель на инкапсулированный интерфейс, вызывая `E_POINTER` ошибку, если этот указатель интерфейса имеет значение null.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_ptr_t](#)

Операторы _com_ptr_t

12.11.2021 • 2 minutes to read

Дополнительные сведения об `_com_ptr_t` операторах см. в разделе [класс _com_ptr_t](#).

См. также

[Класс _com_ptr_t](#)

_com_ptr_t::operator =

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Присваивает новое значение существующему объекту `_com_ptr_t`.

Синтаксис

```
template<typename _OtherIID>
_com_ptr_t& operator=( const _com_ptr_t<_OtherIID>& p );

// Sets a smart pointer to be a different smart pointer of a different
// type or a different raw interface pointer. QueryInterface is called
// to find an interface pointer of this smart pointer's type, and
// Release is called to decrement the reference count for the previously
// encapsulated pointer. If QueryInterface fails with an E_NOINTERFACE,
// a NULL smart pointer results.
template<typename _InterfaceType>
_com_ptr_t& operator=(_InterfaceType* p );

// Encapsulates a raw interface pointer of this smart pointer's type.
// AddRef is called to increment the reference count for the encapsulated
// interface pointer, and Release is called to decrement the reference
// count for the previously encapsulated pointer.
template<> _com_ptr_t&
operator=( Interface* pInterface ) throw();

// Sets a smart pointer to be a copy of another instance of the same
// smart pointer of the same type. AddRef is called to increment the
// reference count for the encapsulated interface pointer, and Release
// is called to decrement the reference count for the previously
// encapsulated pointer.
_com_ptr_t& operator=( const _com_ptr_t& cp ) throw();

// Sets a smart pointer to NULL. The NULL argument must be a zero.
_com_ptr_t& operator=( int null );

// Sets a smart pointer to be a _variant_t object. The encapsulated
// VARIANT must be of type VT_DISPATCH or VT_UNKNOWN, or it can be
// converted to one of these two types. If QueryInterface fails with an
// E_NOINTERFACE error, a NULL smart pointer results.
_com_ptr_t& operator=( const _variant_t& varSrc );
```

Remarks

Назначает этому объекту `_com_ptr_t` указатель на интерфейс.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_ptr_t](#)

Операторы отношения _com_ptr_t

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Сравнивают объект интеллектуального указателя с другим интеллектуальным указателем, необработанным указателем на интерфейс или значением NULL.

Синтаксис

```
template<typename _OtherIID>
bool operator==( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator==( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator==( _InterfaceType* p );

template<>
bool operator==( Interface* p );

template<>
bool operator==( const _com_ptr_t& p ) throw();

template<>
bool operator==( _com_ptr_t& p ) throw();

bool operator==( Int null );

template<typename _OtherIID>
bool operator!=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator!=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator!=( _InterfaceType* p );

bool operator!=( Int null );

template<typename _OtherIID>
bool operator<( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator<( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator<( _InterfaceType* p );

template<typename _OtherIID>
bool operator>( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator>( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator>( _InterfaceType* p );

template<typename _OtherIID>
bool operator<=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator<=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator<=( _InterfaceType* p );

template<typename _OtherIID>
bool operator>=( const _com_ptr_t<_OtherIID>& p );

template<typename _OtherIID>
bool operator>=( _com_ptr_t<_OtherIID>& p );

template<typename _InterfaceType>
bool operator>=( _InterfaceType* p );
```

Remarks

Сравнивает объект интеллектуального указателя с другим интеллектуальным указателем, необработанным указателем на интерфейс или значением NULL. За исключением проверки ПУСТого указателя, эти операторы сначала запрашивают оба указателя для `IUnknown` и сравнивают результаты.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_ptr_t](#)

Только для систем Майкрософт

Извлекают инкапсулированный указатель на СОМ-интерфейс.

Синтаксис

```
operator Interface*( ) const throw( );
operator Interface&( ) const;
Interface& operator*( ) const;
Interface* operator->( ) const;
Interface** operator&( ) throw( );
operator bool( ) const throw( );
```

Remarks

- `operator Interface*` Возвращает указатель инкапсулированного интерфейса, который может иметь значение `NULL`.
- `operator Interface&` Возвращает ссылку на инкапсулированный указатель интерфейса и выдает ошибку, если указатель имеет значение `NULL`.
- `operator*` Позволяет объекту интеллектуального указателя действовать так, будто он был фактически инкапсулированным интерфейсом при разыменовании.
- `operator->` Позволяет объекту интеллектуального указателя действовать так, будто он был фактически инкапсулированным интерфейсом при разыменовании.
- `operator&` Освобождает любой инкапсулированный указатель интерфейса, заменяя его значением `NULL`, и возвращает адрес инкапсулированного указателя. Этот оператор позволяет передать смарт-указатель по адресу в функцию с параметром `out`, через который он возвращает указатель интерфейса.
- `operator bool` Позволяет использовать объект интеллектуального указателя в условном выражении. Этот оператор возвращает `true`, если указатель не имеет значение `null`.

NOTE

Поскольку не `operator bool` объявлен как `explicit`, неявно преобразуется `_com_ptr_t` в, которое преобразуется `bool` в любой скалярный тип. Это может привести к непредвиденным последствиям в коде. Включите [Предупреждение компилятора \(уровень 4\) C4800](#), чтобы предотвратить непреднамеренное использование этого преобразования.

См. также раздел

[_com_ptr_t - класс](#)

Шаблоны реляционных функций

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Синтаксис

```
template<typename _InterfaceType> bool operator==(  
    int NULL,  
    _com_ptr_t<_InterfaceType>& p  
>;  
template<typename _Interface,  
        typename _InterfacePtr> bool operator==(  
    _Interface* i,  
    _com_ptr_t<_InterfacePtr>& p  
>;  
template<typename _Interface> bool operator!=(  
    int NULL,  
    _com_ptr_t<_Interface>& p  
>;  
template<typename _Interface,  
        typename _InterfacePtr> bool operator!=(  
    _Interface* i,  
    _com_ptr_t<_InterfacePtr>& p  
>;  
template<typename _Interface> bool operator<(  
    int NULL,  
    _com_ptr_t<_Interface>& p  
>;  
template<typename _Interface,  
        typename _InterfacePtr> bool operator<(br/>    _Interface* i,  
    _com_ptr_t<_InterfacePtr>& p  
>;  
template<typename _Interface> bool operator>(  
    int NULL,  
    _com_ptr_t<_Interface>& p  
>;  
template<typename _Interface,  
        typename _InterfacePtr> bool operator>(  
    _Interface* i,  
    _com_ptr_t<_InterfacePtr>& p  
>;  
template<typename _Interface> bool operator<=(  
    int NULL,  
    _com_ptr_t<_Interface>& p  
>;  
template<typename _Interface,  
        typename _InterfacePtr> bool operator<=(  
    _Interface* i,  
    _com_ptr_t<_InterfacePtr>& p  
>;  
template<typename _Interface> bool operator>=(  
    int NULL,  
    _com_ptr_t<_Interface>& p  
>;  
template<typename _Interface,  
        typename _InterfacePtr> bool operator>=(  
    _Interface* i,  
    _com_ptr_t<_InterfacePtr>& p  
>;
```

Параметры

i

Необработанный указатель на интерфейс.

p

Интеллектуальный указатель.

Remarks

Эти шаблоны функции позволяют выполнять сравнение с интеллектуальным указателем, расположенным в правой части оператора сравнения. Они не являются функциями-членами объекта `_com_ptr_t`.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _com_ptr_t](#)

Класс _variant_t

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Объект _variant_t инкапсулирует `VARIANT` тип данных. Класс управляет выделением и освобождением ресурсов и делает вызовы функций `VariantInit` и `VariantClear` соответствующим образом.

Строительство

Имя	Описание
<code>_variant_t</code>	Конструирует объект <code>_variant_t</code> .

Операции

Имя	Описание
<code>Attach</code>	Присоединяет <code>VARIANT</code> объект к объекту <code>_variant_t</code> .
<code>Очистить</code>	Очищает инкапсулированный <code>VARIANT</code> объект.
<code>ChangeType</code>	Изменяет тип объекта <code>_variant_t</code> на указанный <code>VARTYPE</code> .
<code>Отсоединить</code>	Отсоединяет инкапсулированный <code>VARIANT</code> объект от этого <code>_variant_t</code> объекта.
<code>SetString</code>	Присваивает строку этому объекту <code>_variant_t</code> .

Операторы

Имя	Описание
<code>Оператор =</code>	Присваивает новое значение существующему объекту <code>_variant_t</code> .
<code>оператор == !=</code>	Сравните два <code>_variant_t</code> объектов на равенство или неравенство.
<code>Средства извлечения</code>	Извлечение данных из инкапсулированного <code>VARIANT</code> объекта.

Завершение блока, относящегося только к системам Майкрософт

Требования

Заголовок: <comutil.h>

Lib: комсуппв. lib или комсуппвд. lib (Дополнительные сведения см. в разделе [/Zc: wchar_t \(wchar_t является собственным типом\)](#))

См. также раздел

[Классы поддержки СОМ компилятора](#)

Функции-члены `_variant_t`

12.11.2021 • 2 minutes to read

Дополнительные сведения о функциях элементов `_variant_t` см. в разделе [класс `_variant_t`](#).

См. также раздел

[Класс `_variant_t`](#)

_variant_t::_variant_t

12.11.2021 • 3 minutes to read

Блок, относящийся только к системам Microsoft

Формирует объект `_variant_t`.

Синтаксис

```
_variant_t( ) throw( );

_variant_t(
    const VARIANT& varSrc
);

_variant_t(
    const VARIANT* pVarSrc
);

_variant_t(
    const _variant_t& var_t_Src
);

_variant_t(
    VARIANT& varSrc,
    bool fCopy
);

_variant_t(
    short sSrc,
    VARTYPE vtSrc = VT_I2
);

_variant_t(
    long lSrc,
    VARTYPE vtSrc = VT_I4
);

_variant_t(
    float fltSrc
) throw( );

_variant_t(
    double dblSrc,
    VARTYPE vtSrc = VT_R8
);

_variant_t(
    const CY& cySrc
) throw( );

_variant_t(
    const _bstr_t& bstrSrc
);

_variant_t(
    const wchar_t *wstrSrc
);

_variant_t(
    const char* strSrc
```

```

);

_variant_t(
    IDispatch* pDispSrc,
    bool fAddRef = true
) throw( );

_variant_t(
    bool bSrc
) throw( );

_variant_t(
    IUnknown* pIUnknownSrc,
    bool fAddRef = true
) throw( );

_variant_t(
    const DECIMAL& decSrc
) throw( );

_variant_t(
    BYTE bSrc
) throw( );

variant_t(
    char cSrc
) throw();

_variant_t(
    unsigned short usSrc
) throw();

_variant_t(
    unsigned long ulSrc
) throw();

_variant_t(
    int iSrc
) throw();

_variant_t(
    unsigned int uiSrc
) throw();

_variant_t(
    __int64 i8Src
) throw();

_variant_t(
    unsigned __int64 ui8Src
) throw();

```

Параметры

varSrc

Объект `VARIANT`, который необходимо скопировать в новый объект `_variant_t`.

pVarSrc

Указатель на `VARIANT` объект, который должен быть скопирован в новый `_variant_t` объект.

var_t_Src

Объект `_variant_t`, который необходимо скопировать в новый объект `_variant_t`.

фкопи

Если `false` задано значение, предоставленный `VARIANT` объект присоединяется к новому `_variant_t` объекту без создания новой копии с помощью `VariantCopy`.

ISrc, sSrc

Целочисленное значение, которое необходимо скопировать в новый объект `_variant_t`.

vtSrc

`VARTYPE` Для нового `_variant_t` объекта.

fltSrc, dblSrc

Числовое значение, которое необходимо скопировать в новый объект `_variant_t`.

ciSrc

Объект `CY`, который необходимо скопировать в новый объект `_variant_t`.

bstrSrc

Объект `_bstr_t`, который необходимо скопировать в новый объект `_variant_t`.

strSrc, wstrSrc

Строка, которую необходимо скопировать в новый объект `_variant_t`.

bcrk

`bool` Значение, которое должно быть скопировано в новый `_variant_t` объект.

piукновнsrc

Указатель COM-интерфейса на объект VT_UNKNOWN, который должен быть инкапсулирован в новый `_variant_t` объект.

pdispSrc

Указатель COM-интерфейса на объект VT_DISPATCH, который должен быть инкапсулирован в новый `_variant_t` объект.

дексрк

Значение `DECIMAL`, которое необходимо скопировать в новый объект `_variant_t`.

bcrk

Значение `BYTE`, которое необходимо скопировать в новый объект `_variant_t`.

ксрк

`char` Значение, которое должно быть скопировано в новый `_variant_t` объект.

usSrc

`unsigned short` Значение, которое должно быть скопировано в новый `_variant_t` объект.

улсрк

`unsigned long` Значение, которое должно быть скопировано в новый `_variant_t` объект.

исрк

`int` Значение, которое необходимо скопировать в новый `_variant_t` объект.

uiSrc

`unsigned int` Значение, которое необходимо скопировать в новый `_variant_t` объект.

i8Src

`_int64` Значение, которое необходимо скопировать в новый `_variant_t` объект.

ui8Src

Значение `_int64` **без знака**, которое необходимо скопировать в новый `_variant_t` объект.

Remarks

- `_variant_t ()` Конструирует пустой `_variant_t` объект `VT_EMPTY`.

- `_variant_t (variant& варсрк**)`** Конструирует `_variant_t` объект из копии `VARIANT` объекта. Тип variant сохранен.
- `_variant_t (Variant * пварсрк**)`** конструирует `_variant_t` объект из копии `VARIANT` объекта. Тип variant сохранен.
- `_variant_t (_variant_t& var_t_Src**)`** Конструирует `_variant_t` объект из другого `_variant_t` объекта. Тип variant сохранен.
- `_variant_t (Variant& варсрк, bool fCopy)` конструирует `_variant_t` объект из существующего `VARIANT` объекта. Если `fCopy` имеет значение `false`, объект `Variant` присоединяется к новому объекту без создания копии.
- `_variant_t (Short CCPK, VarType vtSrc = VT_I2)` конструирует `_variant_t` объект типа `VT_I2` или `VT_BOOL` из `short` целочисленного значения. Другие `VARTYPE` результаты в случае ошибки `E_INVALIDARG`.
- `_variant_t (Long lSrc , VarType vtSrc = VT_I4)` конструирует `_variant_t` объект типа `VT_I4`, `VT_BOOL` или `VT_ERROR` из `long` целочисленного значения. Другие `VARTYPE` результаты в случае ошибки `E_INVALIDARG`.
- `_variant_t (float fltSrc)` конструирует `_variant_t` объект типа `VT_R4` из `float` числового значения.
- `_variant_t (Double dblSrc , VarType vtSrc = VT_R8)` конструирует `_variant_t` объект типа `VT_R8` или `VT_DATE` из `double` числового значения. Другие `VARTYPE` результаты в случае ошибки `E_INVALIDARG`.
- `_variant_t (CY& cySrc)` конструирует `_variant_t` объект типа `VT_CY` из `cy` объекта.
- `_variant_t (_bstr_t& bstrSrc)` конструирует `_variant_t` объект типа `VT_BSTR` из `_bstr_t` объекта. Выделяется новый параметр `BSTR`.
- `_variant_t (wchar_t * встстрсрк)` конструирует `_variant_t` объект типа `VT_BSTR` из строки Юникода. Выделяется новый параметр `BSTR`.
- `_variant_t (char * strSrc)` Конструирует `_variant_t` объект типа `VT_BSTR` из строки. Выделяется новый параметр `BSTR`.
- `_variant_t (bool bSrc)` конструирует `_variant_t` объект типа `VT_BOOL` из `bool` значения.
- `_variant_t (IUnknown * pIUnknownSrc , bool fAddRef = true)` Конструирует `_variant_t` объект типа `VT_UNKNOWN` из указателя COM-интерфейса. Если `fAddRef` имеет значение `true`, то `AddRef` метод вызывается в указанном указателе интерфейса для сопоставления с вызовом `Release`, который будет происходить при `_variant_t` уничтожении объекта. Вам нужно вызвать `Release` по указанному указателю интерфейса. Если `fAddRef` имеет значение `false`, этот конструктор принимает владение переданным указателем интерфейса; не вызывайте `Release` по указанной указатель интерфейса.
- `_variant_t (IDispatch * pDispSrc , bool fAddRef = true)` Конструирует `_variant_t` объект типа `VT_DISPATCH` из указателя COM-интерфейса. Если `fAddRef` имеет значение `true`, то `AddRef` метод вызывается в указанном указателе интерфейса для сопоставления с вызовом `Release`, который будет происходить при `_variant_t` уничтожении объекта. Вам нужно вызвать `Release` по указанному указателю интерфейса. Если `fAddRef` имеет значение `false`, этот конструктор принимает владение переданным указателем интерфейса; не вызывайте `Release` по указанной указатель интерфейса.

- `_variant_t (десятичное& decSrc)` конструирует `_variant_t` объект типа VT_DECIMAL из `DECIMAL` значения.
- `_variant_t (Byte bSrc)` конструирует `_variant_t` объект типа VT_UI1 из BYTE значения.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _variant_t](#)

_variant_t::Attach

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Присоединяет `VARIANT` объект к объекту `_variant_t`.

Синтаксис

```
void Attach(VARIANT& varSrc);
```

Параметры

varSrc

`VARIANT` Объект, присоединяемый к этому `_variant_t` объекту.

Remarks

Принимает владение `VARIANT` объектом, инкапсулирующим его. Эта функция члена освобождает все существующие инкапсулированные `VARIANT`, затем копирует предоставленный объект `VARIANT` и присваивает свойству `VARTYPE` значение `VT_EMPTY`, чтобы убедиться, что его ресурсы можно освободить только с помощью деструктора `_variant_t`.

Завершение блока, относящегося только к системам Microsoft

См. также

[Класс _variant_t](#)

_variant_t::Clear

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Очищает инкапсулированный `VARIANT` объект.

Синтаксис

```
void Clear( );
```

Remarks

Вызывает `VariantClear` Инкапсулированный `VARIANT` объект.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _variant_t](#)

_variant_t::ChangeType

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Изменяет тип `_variant_t` объекта на указанный `VARTYPE`.

Синтаксис

```
void ChangeType(
    VARTYPE vartype,
    const _variant_t* pSrc = NULL
);
```

Параметры

VarType

`VARTYPE` Для этого `_variant_t` объекта.

pSrc

Указатель на объект `_variant_t`, который необходимо преобразовать. Если это значение равно `NULL`, преобразование выполняется на месте.

Remarks

Эта функция-член преобразует `_variant_t` объект в указанный `VARTYPE`. Если *pSrc* имеет значение `null`, преобразование выполняется на месте, в противном случае `_variant_t` объект копируется из *pSrc* и затем преобразуется.

Завершение блока, относящегося только к системам Microsoft

См. также

[Класс _variant_t](#)

_variant_t::Detach

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Отсоединяет инкапсулированный `VARIANT` объект от этого `_variant_t` объекта.

Синтаксис

```
VARIANT Detach( );
```

Возвращаемое значение

Инкапсулированный `VARIANT`.

Remarks

Извлекает и возвращает инкапсулированный `VARIANT`, а затем очищает этот `_variant_t` объект без его уничтожения. Эта функция члена удаляет `VARIANT` из инкапсуляции и задает `VARTYPE` для этого объекта значение `_variant_t` `VT_EMPTY`. Для освобождения возвращенного `VARIANT` метода необходимо вызвать функцию [вариантклеар](#).

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _variant_t](#)

_variant_t::SetString

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Присваивает строку данному объекту `_variant_t`.

Синтаксис

```
void SetString(const char* pSrc);
```

Параметры

pSrc

Указатель на строку знаков.

Remarks

Преобразует символьную строку ANSI в строку `BSTR` Юникода и присваивает ее данному объекту `_variant_t`.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _variant_t](#)

Операторы _variant_t

12.11.2021 • 2 minutes to read

Дополнительные сведения об операторах _variant_t см. в разделе [класс _variant_t](#).

См. также

[Класс _variant_t](#)

_variant_t::operator=

12.11.2021 • 2 minutes to read

Присваивает новое значение `_variant_t` экземпляру.

`_variant_t` Класс и его `operator=` член являются **специфичными для Microsoft**.

Синтаксис

```
_variant_t& operator=( const VARIANT& varSrc );
_variant_t& operator=( const VARIANT* pVarSrc );
_variant_t& operator=( const _variant_t& var_t_Src );
_variant_t& operator=( short sSrc );
_variant_t& operator=( long lSrc );
_variant_t& operator=( float fltSrc );
_variant_t& operator=( double dblSrc );
_variant_t& operator=( const CY& cySrc );
_variant_t& operator=( const _bstr_t& bstrSrc );
_variant_t& operator=( const wchar_t* wstrSrc );
_variant_t& operator=( const char* strSrc );
_variant_t& operator=( IDispatch* pDispSrc );
_variant_t& operator=( bool bSrc );
_variant_t& operator=( IUnknown* pSrc );
_variant_t& operator=( const DECIMAL& decSrc );
_variant_t& operator=( BYTE byteSrc );
_variant_t& operator=( char cSrc );
_variant_t& operator=( unsigned short usSrc );
_variant_t& operator=( unsigned long ulSrc );
_variant_t& operator=( int iSrc );
_variant_t& operator=( unsigned int uiSrc );
_variant_t& operator=( __int64 i8Src );
_variant_t& operator=( unsigned __int64 ui8Src );
```

Параметры

`varSrc`

Ссылка на объект, `VARIANT` из которого копируются содержимое и `VT_*` тип.

`pVarSrc`

Указатель на объект, `VARIANT` из которого копируются содержимое и `VT_*` тип.

`var_t_Src`

Ссылка на объект, `_variant_t` из которого копируются содержимое и `VT_*` тип.

`sSrc`

`short` Целочисленное значение для копирования. Заданный тип, `VT_BOOL` Если `*this` имеет тип `VT_BOOL`. В противном случае задается тип `VT_I2`.

`lSrc`

`long` Целочисленное значение для копирования. Заданный тип, `VT_BOOL` Если `*this` имеет тип `VT_BOOL`. Заданный тип, `VT_ERROR` Если `*this` имеет тип `VT_ERROR`. В противном случае заданный тип `VT_I4`.

`fltSrc`

`float` Числовое значение для копирования. Заданный тип `VT_R4`.

`dblSrc`

`double` Числовое значение для копирования. Заданный тип `VT_DATE`. Если `this` имеет тип `VT_DATE`. В противном случае заданный тип `VT_R8`.

`cySrc`

Объект `CY` для копирования. Заданный тип `VT_CY`.

`bstrSrc`

Объект `BSTR` для копирования. Заданный тип `VT_BSTR`.

`wstrSrc`

Строка в Юникоде для копирования, хранимая как `BSTR` и заданный тип `VT_BSTR`.

`strSrc`

Многобайтовая строка для копирования, хранимая как `BSTR` и заданный тип `VT_BSTR`.

`pDispSrc`

`IDispatch` Указатель, который нужно скопировать с помощью вызова `AddRef`. Заданный тип `VT_DISPATCH`.

`bSrc`

`bool` Копируемое значение. Заданный тип `VT_BOOL`.

`pSrc`

`IUnknown` Указатель, который нужно скопировать с помощью вызова `AddRef`. Заданный тип `VT_UNKNOWN`.

`decSrc`

Объект `DECIMAL` для копирования. Заданный тип `VT_DECIMAL`.

`byteSrc`

`BYTE` Копируемое значение. Заданный тип `VT_UI1`.

`cSrc`

`char` Копируемое значение. Заданный тип `VT_I1`.

`usSrc`

`unsigned short` Копируемое значение. Заданный тип `VT_UI2`.

`ulSrc`

`unsigned long` Копируемое значение. Заданный тип `VT_UI4`.

`iSrc`

`int` Копируемое значение. Заданный тип `VT_INT`.

`uiSrc`

`unsigned int` Копируемое значение. Заданный тип `VT_UINT`.

`i8Src`

`_int64` `long long` Копируемое значение или. Заданный тип `VT_I8`.

`ui8Src`

`unsigned _int64` `unsigned long long` Копируемое значение или. Заданный тип `VT_UI8`.

Remarks

`operator=` Оператор присваивания очищает любое существующее значение, которое удаляет типы объектов или вызывает `Release`, `IDispatch*`, `IUnknown*` типы и. Затем он копирует новое значение в `_variant_t` объект. Он изменяет `_variant_t` тип так, чтобы он соответствовал назначенному значению, за исключением случаев, указанных для `short`, `long` аргументов, и `double`. Типы значений копируются

напрямую. `VARIANT` Указатель или `_variant_t` ссылочный аргумент или копирует содержимое и тип назначенного объекта. Другие аргументы указателя или ссылочного типа создают копию назначенного объекта. Оператор присваивания вызывает `AddRef` `IDispatch*` аргументы и `IUnknown*`.

`operator=` вызывается `_com_raise_error` при возникновении ошибки.

`operator=` Возвращает ссылку на обновленный `_variant_t` объект.

См. также

Класс `_variant_t`

Операторы отношения _variant_t

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Сравнивает два объекта `_variant_t` и определяет, равны ли они.

Синтаксис

```
bool operator==(  
    const VARIANT& varSrc) const;  
bool operator==(  
    const VARIANT* pSrc) const;  
bool operator!=(  
    const VARIANT& varSrc) const;  
bool operator!=(  
    const VARIANT* pSrc) const;
```

Параметры

varSrc

Объект, `VARIANT` сравниваемый с `_variant_t` объектом.

pSrc

Указатель на `VARIANT` объект, сравниваемый с `_variant_t` объектом.

Возвращаемое значение

Возвращает `true`, если сравнение содержит, `false` Если нет.

Remarks

Сравнивает `_variant_t` объект с объектом `VARIANT`, проверяя на равенство или неравенство.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _variant_t](#)

Средства извлечения _variant_t

12.11.2021 • 2 minutes to read

Блок, относящийся только к системам Microsoft

Извлечение данных из инкапсулированного `VARIANT` объекта.

Синтаксис

```
operator short( ) const;
operator long( ) const;
operator float( ) const;
operator double( ) const;
operator CY( ) const;
operator _bstr_t( ) const;
operator IDispatch*( ) const;
operator bool( ) const;
operator IUnknown*( ) const;
operator DECIMAL( ) const;
operator BYTE( ) const;
operator VARIANT() const throw();
operator char() const;
operator unsigned short() const;
operator unsigned long() const;
operator int() const;
operator unsigned int() const;
operator __int64() const;
operator unsigned __int64() const;
```

Remarks

Извлекает необработанные данные из инкапсулированного объекта `VARIANT`. Если `VARIANT` тип не является правильным, `VariantChangeType` используется для попытки преобразования, а при сбое возникает ошибка:

- **оператор Short ()** Извлекает `short` целочисленное значение.
- **оператор Long ()** Извлекает `long` целочисленное значение.
- **оператор float ()** Извлекает `float` числовое значение.
- **оператор Double ()** Извлекает `double` целочисленное значение.
- **оператор CY ()** Извлекает `CY` объект.
- **operator bool ()** Извлекает `bool` значение.
- **оператор Decimal ()** Извлекает `DECIMAL` значение.
- **Byte оператора ()** Извлекает `BYTE` значение.
- **оператор _bstr_t ()** Извлекает строку, инкапсулированную в `_bstr_t` объекте.
- **оператор IDispatch * ()** извлекает указатель disp-интерфейса из инкапсулированного `VARIANT`.
`AddRef` метод вызывается для полученного указателя, поэтому его можно вызвать `Release` для освобождения.

- **оператор IUnknown *** () ИЗВЛЕКАЕТ указатель интерфейса COM из инкапсулированного объекта `VARIANT`. Метод вызывается для полученного указателя, поэтому его можно вызвать `Release` для освобождения.

Завершение блока, относящегося только к системам Майкрософт

См. также

[Класс _variant_t](#)

Расширения Майкрософт

12.11.2021 • 2 minutes to read

`asm-statement`:

`__asm assembly-instruction ;` неявное согласие
`__asm { assembly-instruction-list } ;` неявное согласие

`assembly-instruction-List`:

`assembly-instruction ;` необ.
`assembly-instruction ; assembly-instruction-List ;` неявное согласие

`ms-modifier-List`:

`ms-modifier ms-modifier-List` необ.

`ms-modifier`:

`__cdecl`
`__fastcall`
`__stdcall`
`__syscall` (зарезервировано для будущих реализаций)
`__oldcall` (зарезервировано для будущих реализаций)
`__unaligned` (зарезервировано для будущих реализаций)
`based-modifier`

`based-modifier`:

`__based (based-type)`

`based-type`:

`name`

Нестандартное поведение

12.11.2021 • 2 minutes to read

В следующих разделах перечислены некоторые из мест, где реализация Microsoft C++ не соответствует стандарту C++. Приведенные ниже номера разделов соответствуют номерам разделов в стандарте C++ 11 (ISO/IEC 14882:2011(E)).

Список ограничений компилятора, отличающихся от заданных в стандарте C++, задается в [ограничениях компилятора](#).

Ковариантные возвращаемые типы

Виртуальные базовые классы не поддерживаются в качестве ковариантных возвращаемых типов, если виртуальная функция имеет виртуальное число аргументов. Это не соответствует пункту 7 раздела 10.3 спецификации C++ ISO. Следующий пример не компилируется, выдавая ошибку компилятора [C2688](#)

```
// CovariantReturn.cpp
class A
{
    virtual A* f(int c, ...); // remove ...
};

class B : virtual A
{
    B* f(int c, ...); // C2688 remove ...
};
```

Привязка независимых имен в шаблонах

Компилятор Microsoft C++ в настоящее время не поддерживает привязку независимых имен при первоначальном анализе шаблона. Это не соответствует разделу 14.6.3 спецификации C++ ISO. Это может привести к тому, что перегруженные версии, объявленные после шаблона (но до создания его экземпляра), будут видны.

```
#include <iostream>
using namespace std;

namespace N {
    void f(int) { cout << "f(int)" << endl;}
}

template <class T> void g(T) {
    N::f('a'); // calls f(char), should call f(int)
}

namespace N {
    void f(char) { cout << "f(char)" << endl;}
}

int main() {
    g('c');
}
// Output: f(char)
```

Описатели исключений функций

Описатели исключений функции, отличные от `throw()`, анализируются, но не используются. Это не соответствует разделу 15.4 спецификации C++ ISO. Пример:

```
void f() throw(int); // parsed but not used
void g() throw();    // parsed and used
```

Дополнительные сведения о спецификациях исключений см. в разделе [спецификации исключений](#).

char_traits::eof()

Стандарт C++ указывает, что `char_traits::EOF` не должно соответствовать допустимому `char_type` значению. Компилятор Microsoft C++ применяет это ограничение для типа `char`, но не для типа `wchar_t`. Это не соответствует требованиям в таблице 62 в разделе 12.1.1 спецификации ISO C++. Это демонстрируется в приведенном ниже примере.

```
#include <iostream>

int main()
{
    using namespace std;

    char_traits<char>::int_type int2 = char_traits<char>::eof();
    cout << "The eof marker for char_traits<char> is: " << int2 << endl;

    char_traits<wchar_t>::int_type int3 = char_traits<wchar_t>::eof();
    cout << "The eof marker for char_traits<wchar_t> is: " << int3 << endl;
}
```

Место хранения объектов

Стандарт языка C++ (раздел 1.8, пункт 6) требует, чтобы полные объекты C++ имели уникальные расположения хранения. Однако в Microsoft C++ существуют случаи, когда типы без элементов данных будут совместно использовать место хранения с другими типами в течение времени существования объекта.

Ограничения компилятора

12.11.2021 • 2 minutes to read

Стандарт языка C++ рекомендует ограничения для различных языковых конструкций. Ниже приведен список случаев, когда компилятор Microsoft C++ не реализует рекомендованные ограничения. Первое число — это ограничение, установленное в стандарте ISO C++ 11 (ИНЦИТС/ISO/IEC 14882-2011 [2012], приложение B), а второе число — это ограничение, реализуемое компилятором Microsoft C++:

- Вложенность уровней составных инструкций, структур управления итерацией и структур элементов управления выбором — C++ Standard: 256, компилятор Microsoft C++: зависит от сочетания вложенных операторов, но обычно между 100 и 110.
- Параметры в одном определении макроса — стандарт C++: 256, компилятор Microsoft C++: 127.
- Аргументы в одном вызове макроса — стандарт C++: 256, компилятор Microsoft C++ 127.
- Символы в символьном строковом литерале или расширенном строковом литерале (после объединения) — C++ Standard: 65536, компилятор Microsoft C++: 65535 однобайтовых символов, включая завершающий символ NULL, и 32767 двухбайтовые символы, включая знак завершения NULL.
- Уровни вложенных определений классов, структур или объединений в одном `struct-declaration-list` стандарте C++: 256, компиляторе Microsoft C++: 16.
- Инициализаторы членов в определении конструктора — стандарт C++: 6144, компилятор Microsoft C++: не менее 6144.
- Квалификация области одного идентификатора — стандарт C++: 256, компилятор Microsoft C++: 127.
- Вложенные `extern` спецификации — C++ Standard: 1024, компилятор Microsoft C++: 9 (не подсчитывает неявную `extern` спецификацию в глобальной области видимости, или 10, если вы подсчитываете неявную `extern` спецификацию в глобальной области видимости.)
- Аргументы шаблона в объявлении шаблона — стандарт C++: 1024, компилятор Microsoft C++: 2046.

См. также

[Нестандартное поведение](#)

Справочник по препроцессору в C/C++

12.11.2021 • 2 minutes to read

В *справочнике по препроцессору C/c++* описывается препроцессор, реализованный в Microsoft C/c++.

Препроцессор выполняет предварительные операции с файлами C и C++ перед их передачей компилятору. Препроцессор можно использовать для условной компиляции кода, вставки файлов, задания сообщений для ошибок времени компиляции, а также для применения правил, зависящих от компьютера, к разделам кода.

в Visual Studio 2019 параметр компилятора `/zc:препроцессор` предоставляет полностью согласованный препроцессор C11 и C17. Это значение по умолчанию при использовании флага компилятора `/std:c11` или `/std:c17`.

Содержимое раздела

[Препроцессора](#)

Предоставляет общие сведения о традиционных и новых препроцессорах.

[Директивы препроцессора](#)

Описание директив, обычно используемых, чтобы исходные программы можно было легко изменять и компилировать в разных средах выполнения.

[Операторы препроцессора](#)

Описание четырех относящихся к препроцессору операторов, используемых в контексте директивы `#define`.

[Предопределенные макросы](#)

Описывает стандартные макросы, указанные в стандартах C и C++, а также в Microsoft C++.

[Директивы pragma](#)

Описание директив `#pragma`, которые позволяют каждому компилятору предоставлять возможности, зависящие от компьютера и операционной системы, в то же время сохраняя общую совместимость с языками C и C++.

Связанные разделы

[Справочник по языку C++](#)

Справочные материалы по реализации языка C++ корпорации Microsoft.

[Справочник по языку C](#)

Справочные материалы по реализации языка C корпорации Microsoft.

[Справочник по сборке C/C++](#)

Ссылки на разделы, в которых рассматриваются параметры компилятора и компоновщика.

[проекты Visual Studio — C++](#)

Описание пользовательского интерфейса в Visual Studio, позволяющего определять каталоги, в которых система проектов будет выполнять поиск файлов для проекта C++.

Справочник по стандартной библиотеке C++

12.11.2021 • 2 minutes to read

Программа на языке C++ может вызывать множество функций из этой реализации стандартной библиотеки C++. Эти функции выполняют такие службы, как входные и выходные данные, и предоставляют эффективные реализации часто используемых операций.

Дополнительные сведения о связывании с соответствующим файлом среды выполнения Visual C++ [.lib](#) см. в разделе [среды выполнения C \(CRT\)](#) и [.lib](#) [файлы стандартной библиотеки C++ \(STL\)](#).

Содержимое раздела

[Общие сведения о стандартной библиотеке C++](#)

Обзор реализации стандартной библиотеки C++ корпорации Майкрософт.

[`iostream` Программирование](#)

Содержит общие сведения о `iostream` программировании.

[Справочник по файлам заголовков](#)

Содержит ссылки на справочные разделы о файлах заголовков стандартной библиотеки C++ с примерами кода.