Приведение reinterpret_cast — cppreference.com

Производит приведение одного типа к другому, меняя интерпретацию подлежащего набора битов.

[править] Синтаксис

`reinterpret_cast <` новый_тип `> (` выражение `)`

Возвращает значение типа новый_тип.

[править] Объяснение

В отличие от static_cast, но подобно const_cast, выражение reinterpret_cast не компилируется в какие-либо процессорные инструкции (кроме преобразований целого числа в указатель и наоборот или на малопонятных архитектурах, где представление указателя зависит от его типа) и является лишь директивой времени компиляции, которая предписывает компилятору трактовать выражение, как имеющее тип новый_тип.

При помощи reinterpret_cast могут быть произведены только следующие преобразования (кроме случаев, когда такие преобразования отменили бы ограничения, налагаемые модификаторами *const* или *volatile*):

1) Выражение целочисленного типа, перечисление, указатель, или указатель-на-член могут быть преобразованы в свой собственный тип. Значение результата `выражение` при этом не изменяется. (начиная с C++11)

2) Указатель может быть преобразован к любому целочисленному типу достаточного размера, вмещающего все его значения, например к std::uintptr_t.

3) Значение любого целочисленного типа или перечисления может быть преобразовано в указательный тип. Указатель, преобразованный в целое число достаточного размера, после обратного преобразования к типу того же указателя гарантированно будет иметь исходное значение, в противном случае результирующий указатель не может быть безопасно разыменован (возможность преобразований туда и обратно в противоположном направлении не гарантирована; один и тот же указатель может иметь множество представлений в виде целого числа). Не гарантируется, что нулевая указательная константа NULL или целочисленный ноль, дадут значение нулевого указателя целевого типа; для этой цели должен быть использовано приведение static_cast или неявное приведение.

4) Любое значение типа std::nullptr_t, включая nullptr, может быть преобразовано к любому целочисленному типу, как если бы оно было (void*)0, но никакое значение, в том числе и nullptr, не может быть преобразовано к std::nullptr_t - для этого должно быть использовано приведение static_cast. (начиная с C++11)

5) Любой указатель на объектный тип `T1` может быть преобразован к указателю на другой объектный тип *cv* `T2`. Такое приведение эквивалентно static_cast<cv T2*>(static_cast<cv void*>(выражение)) (подразумевается, что, если требование по выравниванию для `T2` не строже, чем для `T1`, то значение указателя не изменится и обратное приведение указателя к исходному типу даст исходное значение). В любом случае, результат может быть безопасно разыменован, только если это разрешено правилами на *псевдоним типа* (см. ниже)

6) lvalue-выражение типа `T1` может быть преобразовано к ссылке на другой тип `T2`. Результатом будет lvalue или xvalue, ссылающееся на тот же объект исходного lvalue, но другого типа. При этом, ни копии исходного объекта, ни временного объекта не создается, а также не вызываются конструкторы и операторы преобразования. Доступ к объекту по результирующей ссылке может быть осуществлен безопасно, только если это разрешено правилами на *псевдоним типа* (см. ниже).

7) Любой указатель на функцию может быть преобразован к указателю на другой функциональный тип. Вызов функции через указатель на другой функциональный тип приводит к неопределенному поведению, но приведение такого указателя обратно к указателю на исходный функциональный тип даст указатель на исходную функцию.

8) В некоторых реализациях (в частности, на любой POSIX-совместимой системе, как требует функция dlsym), указатель на функцию может быть преобразован к void* или к любому другому указателю на объект и наоборот. Если реализация поддерживает преобразование в обоих направлениях, то преобразование к исходному типу даст исходное значение, иначе результирующий указатель не может быть быть разыменован или вызван безопасно.

9) Нулевой указатель любого типа может быть преобразован к любому другому указательному типу, давая нулевой указатель этого типа. Заметьте, что константа nullptr или любое другое

значение типа std::nullptr_t не может быть преобразовано к указателю при помощи reinterpret_cast - для этого должно быть использовано неявное преобразование или приведение static_cast.

10) Указатель на функцию-член может быть преобразован к указателю на другую функцию-член другого типа. Обратное преобразование к исходному типу даст исходное значение, в противном случае результирующий указатель не может быть безопасно использован.

11) Указатель на объект-член класса `T1` может быть преобразован к указателю на другой объект-член другого класса `T2`. Если выравнивание `T2` не строже, чем выравнивание `T1`, то преобразование к исходному типу даст исходное значение, в противном случае результирующий указатель не может быть безопасно использован.

Как и во всех литых выражения, результат

Оригинал:

As with all cast expressions, the result is:

Текст был переведён автоматически используя Переводчик Google.
Вы можете проверить и исправить перевод. Для инструкций щёлкните сюда.

- именующее если new_type является тип именующее ссылку или RValue ссылка на функцию типа

  Оригинал:

  an lvalue if new_type is an lvalue reference type or an rvalue reference to function type;

  Текст был переведён автоматически используя Переводчик Google.
  Вы можете проверить и исправить перевод. Для инструкций щёлкните сюда.

- xvalue если new_type является RValue ссылку на объект типа

  Оригинал:

  an xvalue if new_type is an rvalue reference to object type;

  Текст был переведён автоматически используя Переводчик Google.
  Вы можете проверить и исправить перевод. Для инструкций щёлкните сюда.

- prvalue иначе.

  Оригинал:

  a prvalue otherwise.

  Текст был переведён автоматически используя Переводчик Google.
  Вы можете проверить и исправить перевод. Для инструкций щёлкните сюда.

[править] Ключевые слова

`reinterpret_cast`

[править] Псевдоним типа

Любое чтение или изменение хранимого значения объектного типа `DynamicType` через glvalue типа `AliasedType` является неопределенным поведением, за исключением случаев:

- `AliasedType` и `DynamicType` *подобны*.
- `AliasedType` (возможно cv-квалифицированный) является signed- или unsigned-вариантом типа `DynamicType`.

- `AliasedType` является `std::byte`, (начиная с C++17)char или unsigned char: это позволяет рассматривать объектное представление любого объекта, как массив байт.

Неформально два типа являются *подобными*, если, отбросив cv-квалификаторы верхнего уровня:

- они совпадают
- или они оба указатели и указываемые типы подобны
- или они оба указатели на член одного класса и типы указываемых членов подобны
- или они оба массивы одинакового размера или оба массивы, чей размер не задан, и типы их элементов подобны.

Например:

- const int * volatile * и int * * const подобны;
- const int (* volatile S::* const)[20] и int (* const S::* volatile)[20] подобны;
- int (* const *)(int *) и int (* volatile *)(int *) подобны;
- int (S::*)() const и int (S::*)() *не* подобны;
- int (*)(int *) и int (*)(const int *) *не* подобны;
- const int (*)(int *) и int (*)(int *) *не* подобны;
- int (*)(int * const) и int (*)(int *) подобны (типы совпадают);
- std::pair<int, int> и std::pair<const int, int> *не* подобны.

Это правило позволяет производить анализ псевдонима на основе типа, при котором компилятор подразумевает, что значение, прочитанное посредством glvalue одного типа, не изменится при записи в glvalue другого типа (имеются исключения, указанные выше).

Заметьте, что многие компиляторы C++ ослабляют это правило, применяя нестандартные расширения языка для доступа с другим типом к неактивному члену объединения (в C такой доступ не является неопределенным поведением).

[править] Замечания

Параграф стандарта, определяющий правило перекрытия объектов в памяти, содержит два дополнительных пункта, частично перешедших из стандарта C:

- `AliasedType` является агрегатным типом или типом объединения, который содержит один из ранее упомянутых типов в качестве элемента или нестатического члена (включая, рекурсивно, элементы подагрегатов и нестатические данные-члены содержащихся объединений).
- `AliasedType` является (возможно cv-квалифицированным) базовый класс с типом `DynamicType`.

Эти параграфы описывают ситуации, которые не могут возникнуть в C++ и, таким образом, не рассмотрены выше. В C, доступ к объекту-агрегату при копировании и присваивании осуществляется как к единому целому. Однако в C++ эти действия всегда выполняются через вызов функции-члена, которая имеет дело с индивидуальными подобъектами, а не с целым объектом (или, в случае объединений, копирует представление объекта через unsigned char). См. core issue 2051.

Предполагая, что требования по выравниванию соблюдены, `reinterpret_cast` не изменяет значение указателя, за исключением некоторого числа ограниченных случаев, имеющих дело с объектами *указательно-приводимыми-друг-в-друга*:

```
struct S1 { int a; } s1;
struct S2 { int a; private: int b; } s2; // нестандартная компоновка
union U { int a; double b; } u = {0};
int arr[2];

int* p1 = reinterpret_cast<int*>(&s1); // значение p1 является "указателем на s1.a", т.к. s1.a
                                       // и s1 - указательно-приводимы друг в друга

int* p2 = reinterpret_cast<int*>(&s2); // значение p2 не изменяется при помощи reinterpret_cast и
                                       // является "указателем на s2".

int* p3 = reinterpret_cast<int*>(&u);  // значение p3 является "указателем на u.a": u.a и u -
                                       // указательно-приводимы друг в друга
```

```
double* p4 = reinterpret_cast<double*>(p3); // значение p4 является "указателем на u.b": u.a и u.b -
                                            // указательно-приводимы друг в друга, т.к. оба -
                                            // указательно-приводимы друг в друга через u

int* p5 = reinterpret_cast<int*>(&arr); // значение p5 не изменяется при помощи reinterpret_cast и
                                        // является "указателем на arr"
```

Доступ к члену класса, представляющего собой нестатическое данное-член или вызов нестатической функции-члена у glvalue-объекта, который не на самом деле не представляет собой объект подходящего типа и полученный через `reinterpret_cast` приводит к неопределенному поведению:

```
struct S { int x; };
struct T { int x; int f(); };
struct S1 : S {}; // стандартная компоновка
struct ST : S, T {}; // нестандартная компоновка

S s = {};
auto p = reinterpret_cast<T*>(&s); // p - "указатель на s"
auto i = p->x; // здесь доступ к члену класса приводит к неопределенному поведению, поскольку s не является объектом типа T
p->x = 1; // неопределенное поведение
p->f();   // неопределенное поведение

S1 s1 = {};
auto p1 = reinterpret_cast<S*>(&s1); // p1 является "указателем на подобъект S объекта s1"
auto i = p1->x; // OK
p1->x = 1; // OK

ST st = {};
auto p2 = reinterpret_cast<S*>(&st); // p2 является "указателем на st"
auto i = p2->x; // неопределенное поведение
p2->x = 1; // неопределенное поведение
```

В этих случаях многие компиляторы выдают предупреждение о "перекрытии объектов в памяти", хотя технически такие выражения вступают в конфликт с чем-то иным, чем параграф, известный как "правило перекрытия объектов в памяти".

Назначение правила перекрытия объектов в памяти и сопутствующих правил состоит в том, чтобы разрешить проведение анализа псевдонимов на основе типа. Этого анализа можно было избежать, если бы программа смогла законным образом создать ситуацию, при которой два указателя на несвязанные типы (например int* и float*) существовали одновременно и оба могли бы быть использованы для чтения или записи в ту же область памяти (см. this email on SG12 reflector). Таким образом, любой способ, который на первый взляд может создать такую ситуацию, неминуемо приведёт к неопределенному поведению.

Если необходимо интерпретировать байты объекта как значение другого типа, могут быть использованы std::memcpy или `std::bit_cast` (начиная с C++20):

```
double d = 0.1;
std::int64_t n;
static_assert(sizeof n == sizeof d);
// n = *reinterpret_cast<std::int64_t*>(&d); // неопределенное поведение
std::memcpy(&n, &d, sizeof d); // OK
n = std::bit_cast<std::int64_t>(d); // OK
```

[править] Дефекты стандарта

Следующие изменения поведения были применены с обратной силой к ранее опубликованным стандартам C++:

Номер	Применён	Поведение в стандарте	Корректное поведение
CWG 195	C++98	conversion between function pointers и object pointers not allowed	made conditionally-supported

[править] Пример

Демонстрирует некоторые варианты применения reinterpret_cast:

```
#include <cstdint>
#include <cassert>
#include <iostream>
int f() { return 42; }
int main()
{
    int i = 7;

    // в указатель на целое число и обратно
    std::uintptr_t v1 = reinterpret_cast<std::uintptr_t>(&i); // static_cast is an error
    std::cout << "Значение &i - 0x" << std::hex << v1 << '\n';
    int* p1 = reinterpret_cast<int*>(v1);
    assert(p1 == &i);

    // указатель на функцию в указатель на другую функцию и обратно
    void(*fp1)() = reinterpret_cast<void(*)()>(f);
    // fp1(); неопределенное поведение
    int(*fp2)() = reinterpret_cast<int(*)()>(fp1);
    std::cout << std::dec << fp2() << '\n'; // безопасно

    // псевдоним типа через указатель
    char* p2 = reinterpret_cast<char*>(&i);
    if(p2[0] == '\x7')
        std::cout << "Порядок байт - little-endian\n";
    else
        std::cout << "Порядок байт - big-endian\n";

    // псевдоним типа через ссылку
    reinterpret_cast<unsigned int&>(i) = 42;
    std::cout << i << '\n';

    [[maybe_unused]] const int &const_iref = i;
    //int &iref = reinterpret_cast<int&>(const_iref); //ошибка компилятора - нельзя избавится от const
    //Здесь должен быть применён const_cast:  int &iref = const_cast<int&>(const_iref);
}
```

Возможный вывод:

```
Значение &i - 0x7fff352c3580
42
Порядок байт - little-endian
42
```

[править] См. также