# Приведения типов в стиле C++

# Приведение типов в стиле C

Чтобы результат выражения был приведен к требуемому типу (это бывает необходимо, например, для вызова требуемой перегрузки функции или шаблон) можно воспользоваться приведением типов, доставшимся нам в наследство из языка C:

- `(<требуемый тип>)<выражение>` - классическое приведение в стиле C
- `<требуемый тип>(<выражение>)` - функциональное приведение (только в C++)

```
void f(int);
void f(double);

template <class T>
void g(T);
// ...
double x = 0.0;
f((int)x);  // void f(int)
g(int(x));  // void g<int>(int)
```

# Приведение типов в стиле C

C-style приведение - самый агрессивный вид приведения, который существует в языке C++. Помимо "безопасных" приведений оно может снимать константность (в том числе со ссылок и элементов под указателями!) и менять интерпретацию битов по определенному месту в памяти.

```
const int cx = 11;
const int* pcx = &cx;
const int& rcx = cx;

int* px = (int*)pcx;  // Ok (!!!)
int& rx = (int&)rcx;  // Ok (!!!)
*px = 25;  // UB
rx = 34;   // UB

std::cout << *((float*)pcx);  // pcx будет разыменован как float!
```

`static_cast`

# `static_cast`

`static_cast` осуществляет допустимые в C++ явные пребразования типов (с учетом константности)

Синтаксис: `static_cast< <тип> >( <выражение> )`

- `<тип>` - целевой тип
- `<выражение>` - значение, которое нужно преобразовать

Корректность преобразований проверяется на этапе компиляции!

# static_cast

```
int f(int);
float f(float);

int x = 0;

// Преобразования между числовыми типами
auto x_float = static_cast<float>(x);       // type(x_float) == float
f(static_cast<int>(x_float));               // int f(int)

// Пользовательские преобразования
auto stack = static_cast<Stack>(5);         // явно используется к-р преобразования

// Преобразования из void* в указатель на тип и наоборот
void* ptr = std::malloc(sizeof(int));
auto int_ptr = static_cast<int*>(ptr);      // int*
std::free(static_cast<void*>(int_ptr));

// Выбор перегрузки функции
auto f_ptr = static_cast<int(*)(int)>(f);   // указатель на int f(int)
static_cast<float(*)(float)>(f)(x);         // float f(float)
```

# `const_cast`

# `const_cast`

Позволяет снять/добавить квалификаторы `const` и `volatile`

Синтаксис: `const_cast< <тип> >( <выражение> )`

- `<выражение>` - возвращает значение, которое можно проинтерпретировать как указатель или ссылку
- `<тип>` - совпадает с типом `<выражение>` с точностью до `const` и `volatile`

Корректность проверяется на этапе компиляции

# const_cast

```
int x = 0;

// Относительно безопасные преобразования
auto& x_cref = const_cast<const int&>(x);  // type(x_cref) == const int&
auto& x_ref = const_cast<int&>(x_cref);    // type(x_ref) == int&
auto x_cptr = const_cast<const int*>(&x);  // type(x_cptr) == const int*
auto x_ptr = const_cast<int*>(x_cptr);     // type(x_ptr) == int*

const int cx = 0;
const char* c_str = "string";

// Опасные преобразования
auto& cx_ref = const_cast<int&>(cx);    // type(cx_ref) == int&
auto cx_ptr = const_cast<int*>(&cx);    // type(cx_ptr) == int*
auto str = const_cast<char*>(c_str);    // type(str) == char*
```

`reinterpret_cast`

# `reinterpret_cast`

Позволяет изменить интерпретацию битов в памяти

Синтаксис: `reinterpret_cast< <тип> >( <выражение> )`

- `<выражение>` - должно иметь целый тип, тип указателя или ссылки
- `<тип>` - целый тип или указатель. Если `<выражение>` - ссылка (не временное значение), то `<тип>` тоже может быть ссылкой

Корректность проверяется на этапе компиляции

# reinterpret_cast

```
uint64_t ux = 11;

std::cout << reinterpret_cast<float*>(ux) << '\n';      // 0xb
std::cout << *reinterpret_cast<double*>(&ux) << '\n';  // 5.43472e-323

int32_t sx = -123;
std::cout << reinterpret_cast<uint32_t&>(sx) << '\n';   // 4294967173
```

# `dynamic_cast`

# `dynamic_cast`

Позволяет преобразовывать указатели/ссылки на полиморфный класс в указатели/ссылки на его предка или потомка.

В отличие от `static_cast` проверяет возможномсть преобразования во время выполнения, а не во время компиляции.

Синтаксис: `dynamic_cast< <тип> >( <выражение> )`

- `<выражение>` - указатель или ссылка на полиморфный класс
- `<тип>` - указатель или ссылка на потомка или предка

Если преобразование некорректно (реальный тип объекта не совпадает с запрошенным) возвращает `nullptr` для указателей или бросает исключение `std::bad_cast` в случае ссылок.

# dynamic_cast

```
struct A {};

struct B : public A {
    virtual void f();
};

struct C : public B {};
struct D : public C {};
```

```
C* c_ptr = new C;
B* b_ptr = dynamic_cast<B*>(c_ptr);    // OK
A* a_ptr = dynamic_cast<A*>(b_ptr);    // CE: A - не полиморфный!

dynamic_cast<C*>(b_ptr);               // == c_ptr
dynamic_cast<D*>(b_ptr);               // == nullptr
dynamic_cast<C&>(*b_ptr);              // == *c_ptr
dynamic_cast<D&>(*b_ptr);              // RE: std::bad_cast
```

# Перегрузка операции преобразования типа

# Перегрузка операции преобразования типа

Преобразование из других типов в свой тип задается конструктором преобразования.

Для преобразования своего типа в другой нужно перегрузить соответствующую операцию преобразования.

```
class Complex {
// ...
    operator double() const {
        return re_;
    }
};

void f(double);

Complex c{1.5, 6.7};
f(c);  // Ok: f(1.5) <- хорошо ли это?
```

# Перегрузка операции преобразования типа

Преобразование из других типов в свой тип задается конструктором преобразования.

Для преобразования своего типа в другой нужно перегрузить соответствующую операцию преобразования.

```
class Complex {
// ...
    operator double() const {
        return re_;
    }
};

void f(double);

Complex c{1.5, 6.7};
f(c);  // Ok: f(1.5) <- хорошо ли это?
```

# Перегрузка операции преобразования типа

```
void f(double);

Complex c{1.5, 6.7};
f(c);  // Ok: f(1.5) <- хорошо ли это?
```

Для запрета подобных преобразований можно использовать `explicit`:

```
explicit Complex::operator double() const;

f(c);  // CE
```

```
error: cannot convert 'Complex' to 'double' for argument '1' to 'void f(double)'
```

Исключение - `explicit operator bool()`:

https://stackoverflow.com/questions/39995573/when-can-i-use-explicit-operator-bool-without-a-cast

# Перегрузка операции преобразования типа

Можно перегрузить `operator auto`

Несмотря на многообещающее название, это просто синтаксический сахар, позволяющий вывести тип, к которому осуществляется преобразование на основе **возвращаемого значения**

```
class Complex {
// ...
    explicit operator auto() const {  // то же, что и operator double() !
        return re_;
    }
};
```

# Резюме

- Пользуйтесь `static_cast` для разрешенных преобразований типов, для неразрешенных преобразований воспользуйтесь отказом от них.