Преобразования типов и RTTI

Александр Смаль

Академический университет
27 марта 2014
Санкт-Петербург

C-style cast

Стандартный способ приведения типов в C.

```
int a = 10;
int b = 3;

double d = (double)(10) / 3 + 3.5;
d = int(d);

double * m = (double *) malloc(sizeof(double) * 100);
m[0] = 10.5;

char * mc = (char *)m;
mc[4] = 23;
```

В C преобразует арифметические типы и указатели.

Что делает в C++?

Преобразования в C++: static_cast

Служит для преобразований связанных типов:

- Стандартные преобразования преобразования.

```
double d = static_cast<double>(10) / 3 + 3.5;
d = static_cast<int>(d);
```

- Неявное приведение типа:

```
T t = static_cast<T>(e); // T t(e);
```

- Обратные варианты стандартных преобразований:
  - целочисленные типы в перечисляемые,
  - Base * в Derived * (downcast),
  - T Base:: * в T Derived:: *,
  - void * в любой T *
- Преобразование к void.

```
static_cast<void>(5);
```

Преобразования в C++: const_cast

Служит для снятия/добавления константности:

```
void f(double const& d) {
    const_cast<double &>(d) = 10;
}
```

Использование const_cast — признак плохого дизайна.

```
T& operator[](size_t i) {
    return data_[i];
}
const T& operator[](size_t i) const {
    return data_[i];
}
```

Преобразования в C++: const_cast

Служит для снятия/добавления константности:

```
void f(double const& d) {
    const_cast<double &>(d) = 10;
}
```

Использование const_cast — признак плохого дизайна.
Кроме некоторых исключений:

```
T & operator[](size_t i) {
    return  const_cast<T &>(
            const_cast<Vector const &>(*this)[i]);
}

T const & operator[](size_t i) const {
    assert(i < size_);
    return data_[i]
}
```

Преобразования в C++: reinterpret_cast

Служит для преобразований несвязанных указателей.

Danger!

```
void send(char const * data, size_t length);
char * recv(size_t * length);

double * m = reinterpret_cast<double *>(
                malloc(sizeof(double) * 100));
...
char * mc = reinterpret_cast<char *>(m);
send(mc, sizeof(double) * 100);

// other side
size_t l = 0;
double * r = reinterpret_cast<double *>(recv(&l));
l /= sizeof(double);
```

Границы применимости C-style cast

C-style cast может вызвать любое из преобразований: `static_cast`, `reinterpret_cast`, `const_cast`.

Можно использовать:

- Преобразование встроенных типов.
- Преобразование указателей на явные типы.

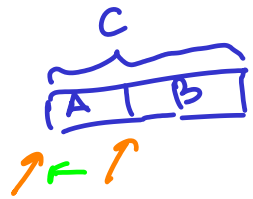
Не стоит использовать:

- В шаблонах.
- Для преобразования пользовательских типов и указателей на них.

```
struct A; struct B; struct C;

C * f(B * b) {
    return (C *)b;      // reinterpret_cast
    // return static_cast<C *>(b); doesn't compile
}

struct A {int a;};
struct B {};
struct C : A, B {};
```



Run-time type information

В C++ есть механизм получения информации о типах времени выполнения. Состоит из двух компонент:

1. type_info и typeid
2. dynamic_cast

type_info

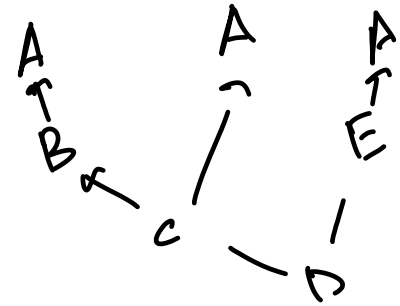
- Класс объявленный в <typeinfo>.
- Методы: ==, !=, name, before (т.е. не копируется).
- Можно получить type_info при помощи оператора typeid.
- typeid от нулевого указателя бросает bad_typeid.

```
struct A { virtual ~A() { } }; struct B : A { };
A* ap = new B();
A& ar = bobj;
cout << typeid(*ap).name() << endl; // B
cout << typeid(ar) .name() << endl; // B
cout << typeid(ap) .name() << endl; // A *
cout << typeid(A *).name() << endl; // A *
```

Преобразования в C++: dynamic_cast

Позволяет делать преобразования с проверкой типа времени выполнения.

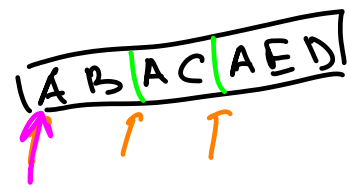
```
A * a = (rand() % 2) ? new B() : new C();
if(B * b = dynamic_cast<B *>(a))
    ...
else if (C * c = dynamic_cast<C *>(a))
    ...
```



Особенности:

- Не заменяется преобразованием в стиле C.
- Требует наличие виртуальных функций (полиморфность типа).
- При приведении к ссылке кидает исключение bad_cast.
- При приведении к указателю может вернуть 0.

Что возвращает dynamic_cast<void *>(a)?



Почему следует избегать RTTI?

Пример: double dispatch

```
struct Triangle; struct Rectangle; struct Circle;
struct Shape {
    virtual ~Shape() {}
    virtual bool intersect( Rectangle * r ) = 0;
    virtual bool intersect( Triangle  * t ) = 0;
    virtual bool intersect( Circle    * c ) = 0;
    virtual bool intersect( Shape     * s ) = 0;
};
struct Triangle : Shape {
    bool intersect( Rectangle * r ) { ... }
    bool intersect( Triangle  * t ) { ... }
    bool intersect( Circle    * c ) { ... }
    bool intersect( Shape     * s ) {
        return s->intersect(this);
    }
};
...

bool intersect( Shape * a, Shape * b ) {
    return a->intersect(b);
}
```

Triangle *