Множественное наследование и операторы приведения

Александр Смаль

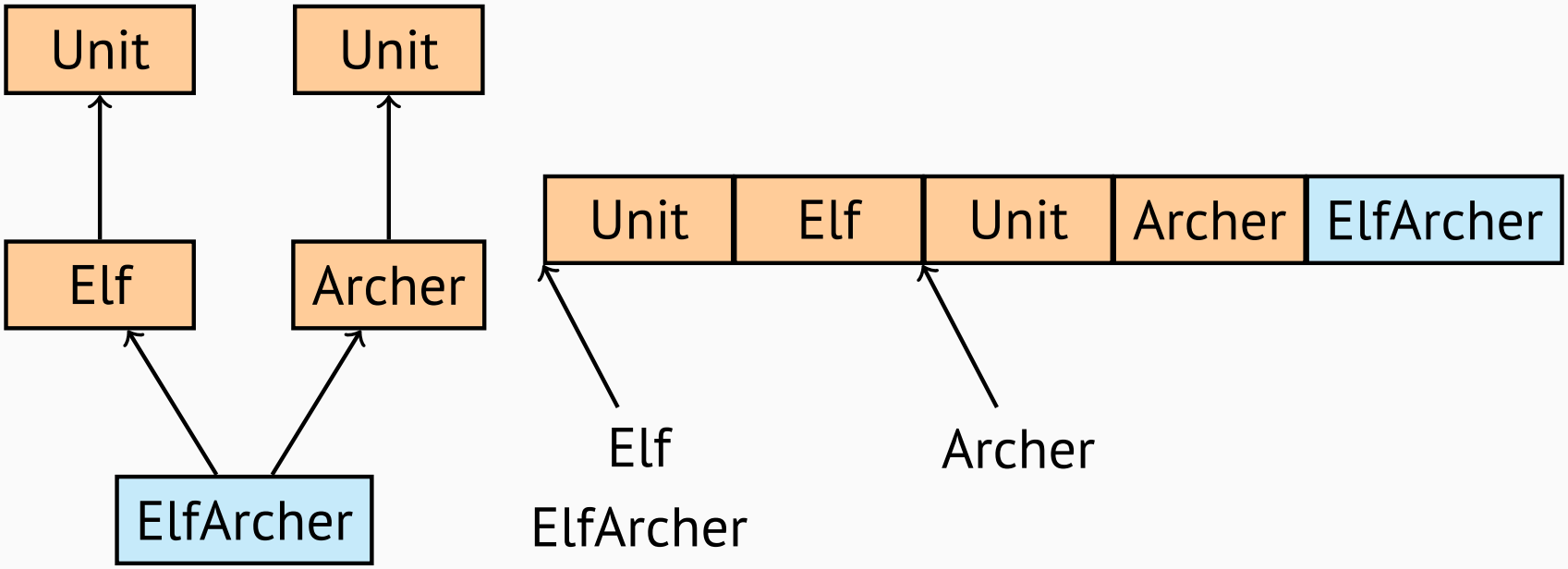
CS центр
13 февраля 2018
Санкт-Петербург

Множественное наследование

Множественное наследование (multiple inheritance) — возможность наследовать сразу несколько классов.

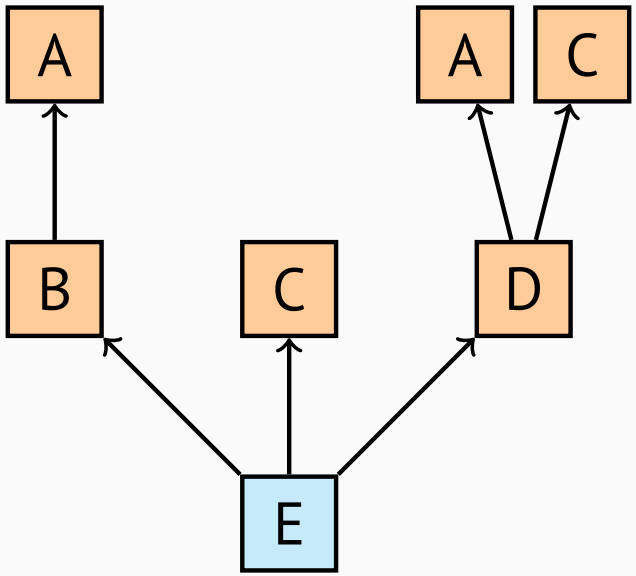
```
struct Unit {
    Unit(unitid id, int hp): id_(id), hp_(hp) {}
    virtual unitid id() const { return id_; }
    virtual int    hp() const { return hp_; }
private:
    unitid id_;
    int    hp_;
};

struct Elf:    Unit { ... };
struct Archer: Unit { ... };

struct ElfArcher: Elf, Archer {
    unitid id() const { return Elf::id(); }
    int    hp() const { return Elf::hp(); }
};
```

Представление в памяти



Важно: указатели при приведении могут смещаться.

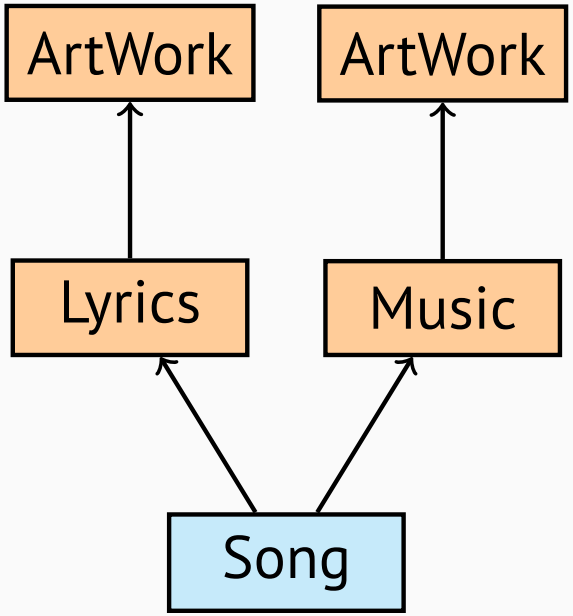
Создание и удаление объекта

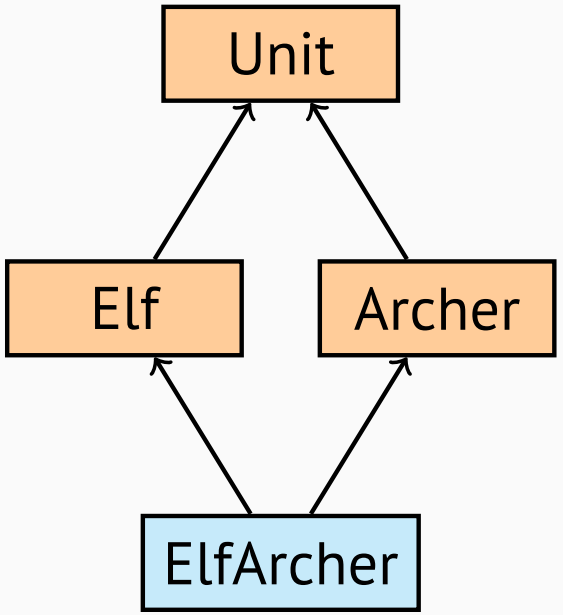


Порядок вызова конструкторов: `A`, `B`, `C`, `A`, `C`, `D`, `E`.
Деструкторы вызываются в обратном порядке.
Проблемы:

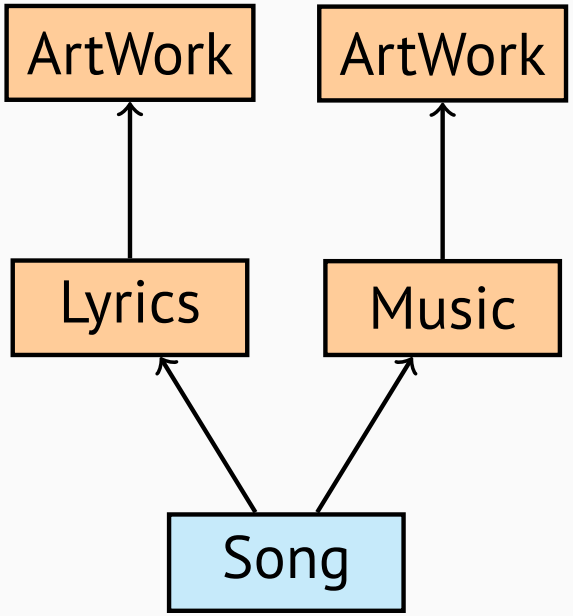
1. Дублирование A и C.
2. Недоступность первого C.

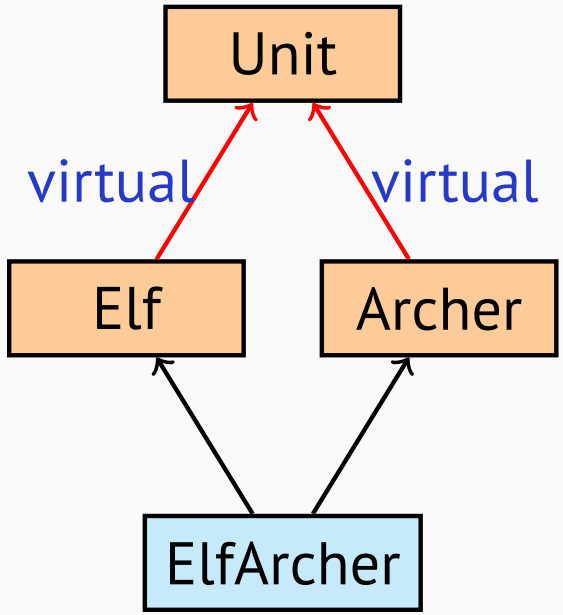
Виртуальное наследование





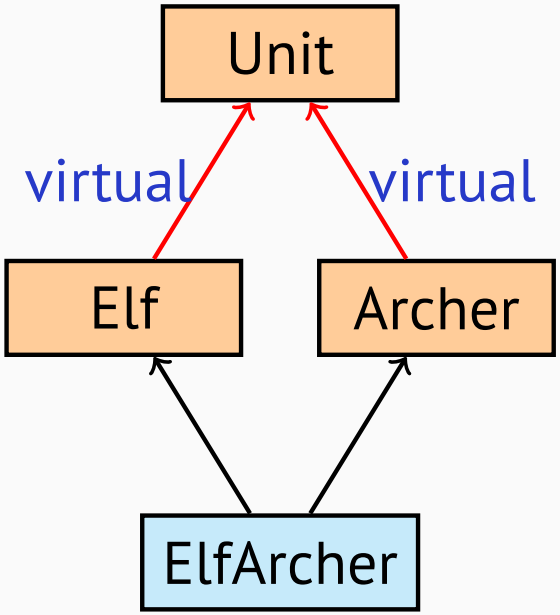
Виртуальное наследование

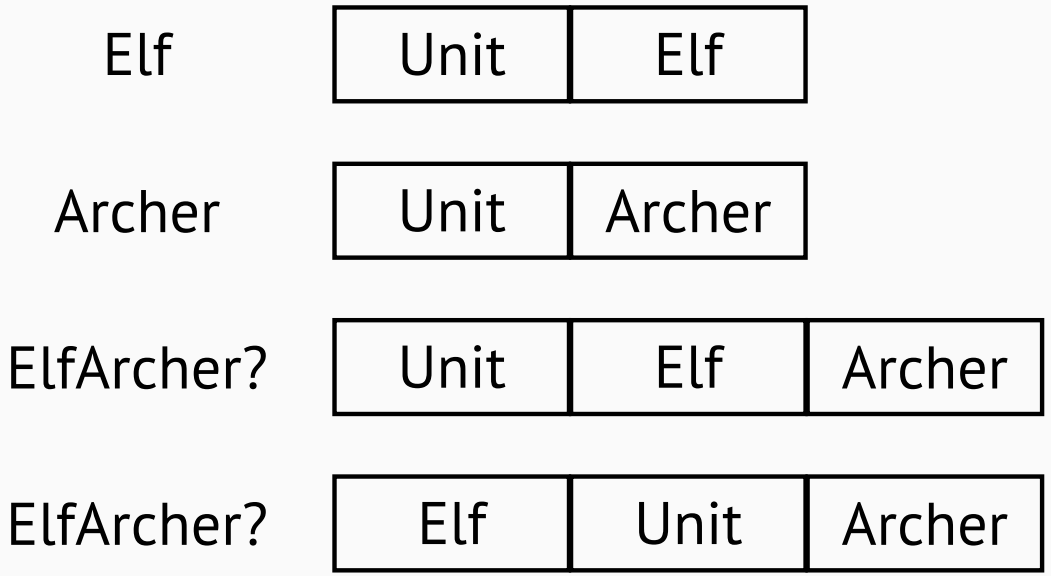




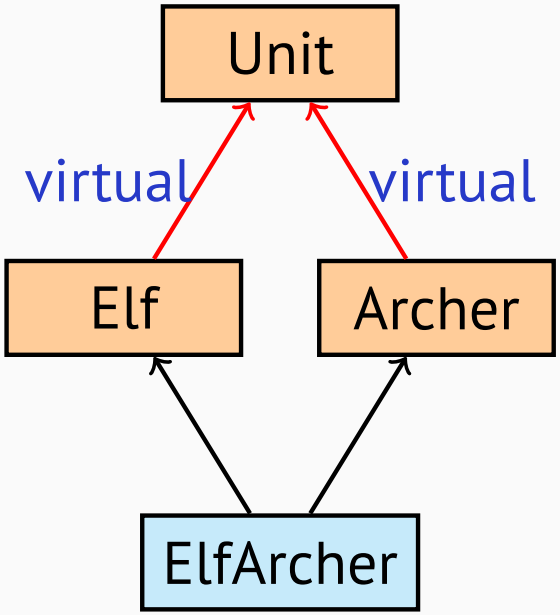

```
struct Unit {};
struct Elf:  virtual Unit {};
struct Archer: virtual Unit {};
struct ElfArcher: Elf, Archer {};
```

Как устроено расположение в памяти?



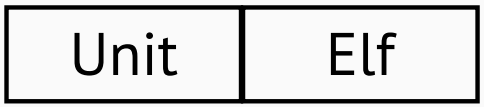


Как устроено расположение в памяти?



На самом деле.

Elf

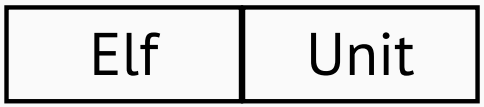


Archer

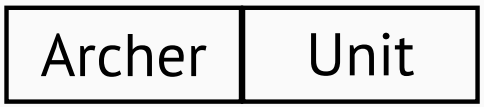
ElfArcher?

ElfArcher?

Elf



Archer



ElfArcher

Доступ через таблицу виртуальных методов

```
struct Unit {
    unitid id;
};
struct Elf  : virtual Unit { };
struct Archer : virtual Unit { };
struct ElfArcher : Elf, Archer { };
```

Доступ через таблицу виртуальных методов

```
struct Unit {
    unitid id;
};
struct Elf  : virtual Unit { };
struct Archer : virtual Unit { };
struct ElfArcher : Elf, Archer { };
```

Рассмотрим такой код:

```
    Elf * e = (rand() % 2)? new Elf() : new ElfArcher();
    unitid id = e->id; // (*)
```

Доступ через таблицу виртуальных методов

```
struct Unit {
    unitid id;
};
struct Elf  : virtual Unit { };
struct Archer : virtual Unit { };
struct ElfArcher : Elf, Archer { };
```

Рассмотрим такой код:

```
    Elf * e = (rand() % 2)? new Elf() : new ElfArcher();
    unitid id = e->id; // (*)
```

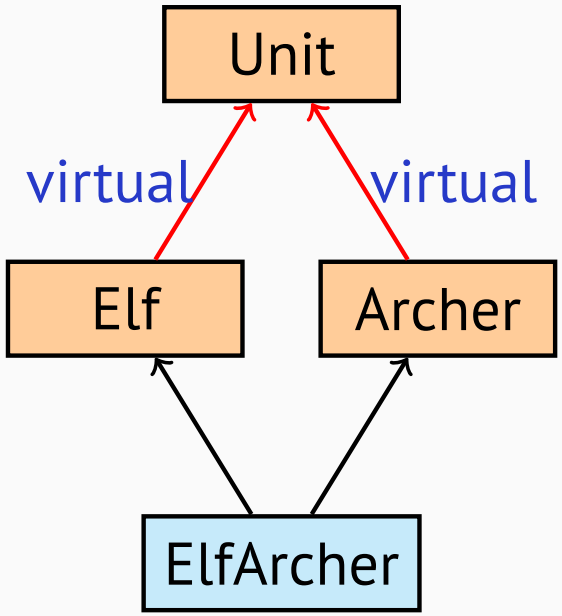
Строка (*) будет преобразована в строку

```
    unitid id = e->__getUnitPtr__()->id;
```

где `__getUnitPtr__()` – это служебный виртуальный метод.

Кто вызывает конструктор базового класса?

```
struct Unit {
    Unit(unitid id, int health_points);
};
struct Elf: virtual Unit {
    explicit Elf(unitid id)
        : Unit(id, 100) {}
};
struct Archer: virtual Unit {
    explicit Archer(unitid id)
        : Unit(id, 120) {}
};
struct ElfArcher: Elf, Archer {
    explicit ElfArcher(unitid id)
        : Unit(id, 150)
        , Elf(id)
        , Archer(id) {}
};
```



Заключение

- Не используйте множественное наследование для наследования реализации.

Заключение

- Не используйте множественное наследование для наследования реализации.
- Используйте концепцию интерфейсов (классы без реализаций и членов данных).

Заключение

- Не используйте множественное наследование для наследования реализации.
- Используйте концепцию интерфейсов (классы без реализаций и членов данных).
- Помните о неприятностях, связанных с множественным наследованием.

Заключение

- Не используйте множественное наследование для наследования реализации.
- Используйте концепцию интерфейсов (классы без реализаций и членов данных).
- Помните о неприятностях, связанных с множественным наследованием.
- Хорошо подумайте перед тем, как использовать виртуальное наследование.

Заключение

- Не используйте множественное наследование для наследования реализации.
- Используйте концепцию интерфейсов (классы без реализаций и членов данных).
- Помните о неприятностях, связанных с множественным наследованием.
- Хорошо подумайте перед тем, как использовать виртуальное наследование.
- Помните о неприятностях, связанных с виртуальным наследованием.

Преобразование в стиле C

В C этот оператор преобразует встроенные типы и указатели.

```
int a = 2;
int b = 3;

// int → double
double size = ((double)a) / b * 100;

// double → int
void * data = malloc(sizeof(double) * int(size));

// void * → double *
double * array = (double *)data;

// double * → char *
char * bytes = (char *)array;
```

Преобразования в C++: `static_cast`

Служит для преобразований связанных типов:

Преобразования в C++: `static_cast`

Служит для преобразований связанных типов:

- Стандартные преобразования.
  - Преобразования числовых типов.

```
double s = static_cast<double>(2) / 3 * 100;
s = static_cast<int>(d);
```

  - Указатель/ссылка на производный класс в указатель/ссылку на базовый класс.
  - T* в `void`*.

Преобразования в C++: `static_cast`

Служит для преобразований связанных типов:

- Стандартные преобразования.
  - Преобразования числовых типов.

```
double s = static_cast<double>(2) / 3 * 100;
s = static_cast<int>(d);
```

  - Указатель/ссылка на производный класс в указатель/ссылку на базовый класс.
  - `T*` в `void*`.
- Явное (пользовательское) приведение типа:

```
Person p = static_cast<Person>("Ivan");
```

Преобразования в C++: `static_cast`

Служит для преобразований связанных типов:

- Стандартные преобразования.
  - Преобразования числовых типов.

```
double s = static_cast<double>(2) / 3 * 100;
s = static_cast<int>(d);
```

  - Указатель/ссылка на производный класс в указатель/ссылку на базовый класс.
  - T* в `void`*.
- Явное (пользовательское) приведение типа:

```
Person p = static_cast<Person>("Ivan");
```

- Обратные варианты стандартных преобразований:
  - Указатель/ссылка на базовый класс в указатель/ссылку на производный класс (преобразование вниз, downcast),
  - `void`* в любой T*.

Преобразования в C++: `static_cast`

Служит для преобразований связанных типов:

- Стандартные преобразования.
  - Преобразования числовых типов.

```
double s = static_cast<double>(2) / 3 * 100;
s = static_cast<int>(d);
```

  - Указатель/ссылка на производный класс в указатель/ссылку на базовый класс.
  - T* в `void`*.
- Явное (пользовательское) приведение типа:

```
Person p = static_cast<Person>("Ivan");
```

- Обратные варианты стандартных преобразований:
  - Указатель/ссылка на базовый класс в указатель/ссылку на производный класс (преобразование вниз, downcast),
  - `void`* в любой T*.
- Преобразование к `void`.

Преобразования в C++: `const_cast`

Служит для снятия/добавления константности.

Преобразования в C++: const_cast

Служит для снятия/добавления константности.

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

Использование const_cast — признак плохого дизайна.

Преобразования в C++: const_cast

Служит для снятия/добавления константности.

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

Использование const_cast — признак плохого дизайна.

Кроме редких 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 указателей и ссылок на несвязанные типы.

Преобразования в C++: `reinterpret_cast`

Служит для преобразований указателей и ссылок на несвязанные типы.

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

Преобразования в C++: reinterpret_cast

Служит для преобразований указателей и ссылок на несвязанные типы.

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

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

Преобразования в C++: reinterpret_cast

Служит для преобразований указателей и ссылок на несвязанные типы.

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

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

```
size_t length = 0;
double * m = reinterpret_cast<double*>(receive(&length));
length = length / sizeof(double);
```

Преобразования в C++: reinterpret_cast

Служит для преобразований указателей и ссылок на несвязанные типы.

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

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

```
size_t length = 0;
double * m = reinterpret_cast<double*>(receive(&length));
length = length / sizeof(double);
```

Поможет преобразовать указатель в число.

```
size_t ms = reinterpret_cast<size_t>(m);
```

Границы применимости преобразования в стиле C

Границы применимости преобразования в стиле C

- Преобразования в стиле C может заменить любое из рассмотренных преобразований:
  - `static_cast`,
  - `reinterpret_cast`,
  - `const_cast`.

Границы применимости преобразования в стиле C

- Преобразования в стиле C может заменить любое из рассмотренных преобразований:
  - `static_cast`,
  - `reinterpret_cast`,
  - `const_cast`.
- Преобразования в стиле C можно использовать для
  - преобразование встроенных типов,
  - преобразование указателей на явные типы.

Границы применимости преобразования в стиле C

- Преобразования в стиле C может заменить любое из рассмотренных преобразований:
  - `static_cast`,
  - `reinterpret_cast`,
  - `const_cast`.
- Преобразования в стиле C можно использовать для
  - преобразование встроенных типов,
  - преобразование указателей на явные типы.
- Преобразования в стиле C не стоит использовать:
  - с пользовательскими типами и указателями на них,
  - в шаблонах.

Когда преобразование в стиле C приводит к ошибке

```
// abc.h
struct A { int a; };

struct B {};

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

Когда преобразование в стиле C приводит к ошибке

```
// abc.h
struct A { int a; };

struct B {};

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

```
#include "abc.h"

C * foo(B * b) {
    return (C *)b;
}
```

Когда преобразование в стиле C приводит к ошибке

```
// abc.h
struct A { int a; };

struct B {};

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

```
#include "abc.h"

C * foo(B * b) {
    return (C *)b;
}
```

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

C * foo(B * b) {
    return (C *)b;
}
```

Когда преобразование в стиле C приводит к ошибке

```
// abc.h
struct A { int a; };

struct B {};

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

```
#include "abc.h"

C * foo(B * b) {
    return (C *)b;
}
```

Если в этой точке известны определения классов, то происходит преобразование `static_cast`.

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

C * foo(B * b) {
    return (C *)b;
}
```

Если известны только объявления, то происходит преобразование `reinterpret_cast`.

Run-Time Type Information (RTTI)

В C++ этот механизм состоит из двух компонент:

1. оператор `typeid` и тип `std::type_info`,
2. оператор `dynamic_cast`.

Run-Time Type Information (RTTI)

В C++ этот механизм состоит из двух компонент:

1. оператор `typeid` и тип `std::type_info`,
2. оператор `dynamic_cast`.

Тип `type_info`

- Класс, объявленный в `<typeinfo>`.
- Содержит информацию о типе.
- Методы: `==`, `!=`, `name`, `before`.
- Нет публичных конструкторов и оператора присваивания.
- Можно получить ссылку на `type_info`, соответствующий значению или типу, при помощи оператора `typeid`.

Использование typeid и type_info

```
struct Unit {
    // наличие виртуальных методов необходимо
    virtual ~Unit() { }
};

struct Elf : Unit { };

int main() {
    Elf e;
    Unit & ur = e;
    Unit * up = &e;
    cout << typeid(ur) .name() << endl; // Elf
    cout << typeid(*up).name() << endl; // Elf
    cout << typeid(up) .name() << endl; // Unit *
    cout << typeid(Elf).name() << endl; // Elf
    cout << (typeid(ur) == typeid(Elf)); // 1
}
```

Преобразования в C++: dynamic_cast

Преобразования с проверкой типа времени выполнения.

```
Unit * u = (rand() % 2)? new Elf(): new Dwarf();
...
if (Elf * e = dynamic_cast<Elf *>(u))
    ...
else if (Dwarf * d = dynamic_cast<Dwarf *>(u))
    ...
```

Преобразования в C++: dynamic_cast

Преобразования с проверкой типа времени выполнения.

```
Unit * u = (rand() % 2)? new Elf(): new Dwarf();
...
if (Elf * e = dynamic_cast<Elf *>(u))
    ...
else if (Dwarf * d = dynamic_cast<Dwarf *>(u))
    ...
```

Особенности:

- Не заменяется преобразованием в стиле C.
- Требует наличие виртуальных функций (полиморфность).

Преобразования в C++: dynamic_cast

Преобразования с проверкой типа времени выполнения.

```
Unit * u = (rand() % 2)? new Elf(): new Dwarf();
...
if (Elf * e = dynamic_cast<Elf *>(u))
    ...
else if (Dwarf * d = dynamic_cast<Dwarf *>(u))
    ...
```

Особенности:

- Не заменяется преобразованием в стиле C.
- Требует наличие виртуальных функций (полиморфность).

Вопросы:

- Почему следует избегать RTTI?
- Что возвращает `dynamic_cast<void *>(u)`?

Пример обхода dynamic_cast: double dispatch

```
struct Rectangle; struct Circle;

struct Shape {
    virtual ~Shape() {}
    virtual bool intersect( Rectangle * r ) = 0;
    virtual bool intersect( Circle    * c ) = 0;
    virtual bool intersect( Shape     * s ) = 0;
};

struct Circle : Shape {
    bool intersect( Rectangle * r ) { ... }
    bool intersect( Circle    * c ) { ... }
    bool intersect( Shape     * s ) {
        return s->intersect(this);
    }
};
bool intersect(Shape * a, Shape * b) {
    return a->intersect(b);
}
```