Продвинутые возможности C++

Спасибо CSCenter

Множественное наследование

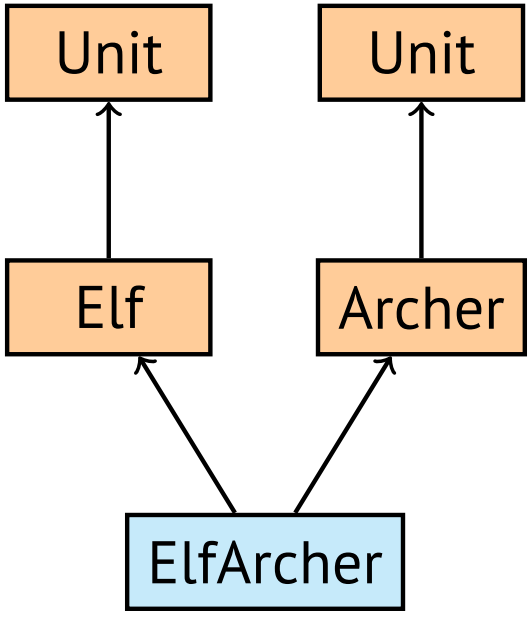
Множественное наследование (multiple inheritance) — возможность наследовать сразу несколько классов.

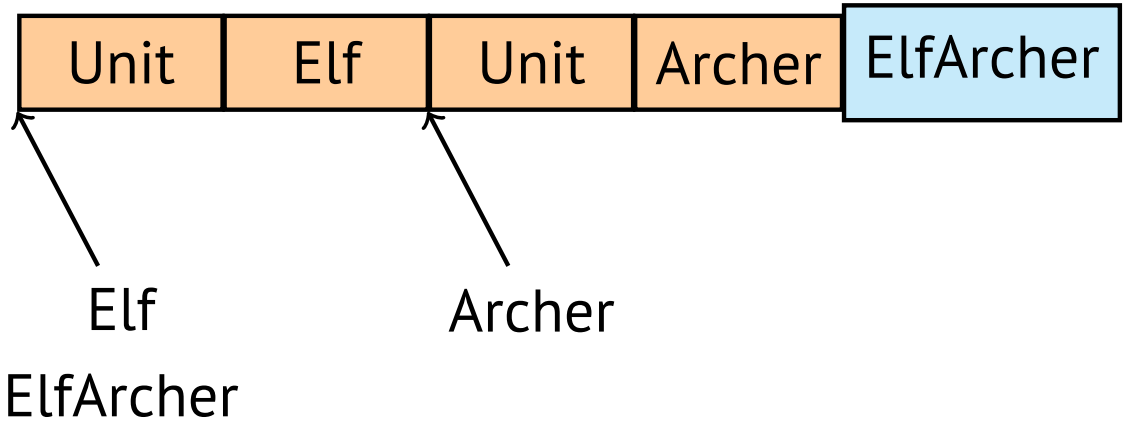
```
struct Unit {
    Unit(unitid id, int hp): id_(id), hp_(hp) {}
    virtual unitid id() const { return id_; }
    virtual int    hp() const { return hp_; }
private:
    unitid id_;
    int    hp_;
};

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

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

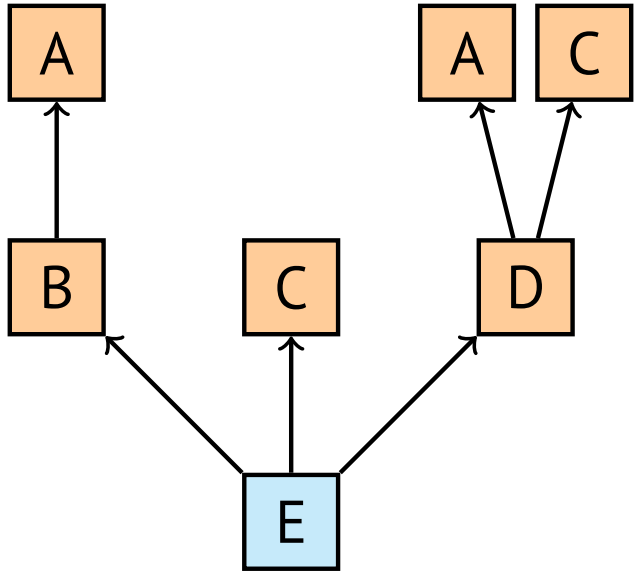
Представление в памяти





Важно: указатели при приведении могут смещаться.

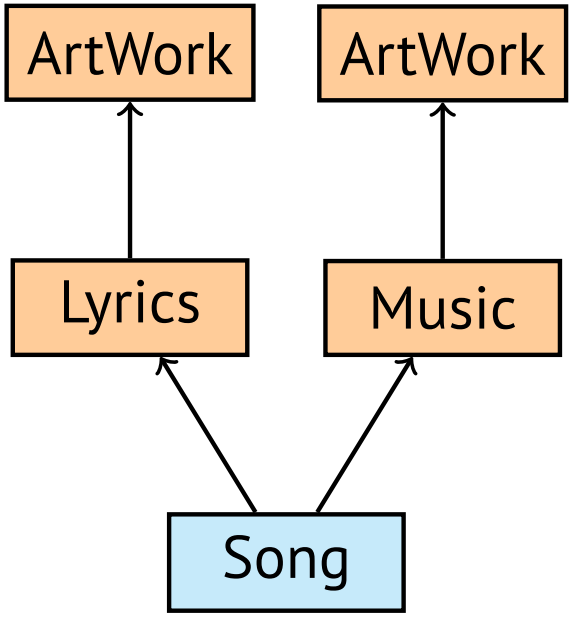
Создание и удаление объекта

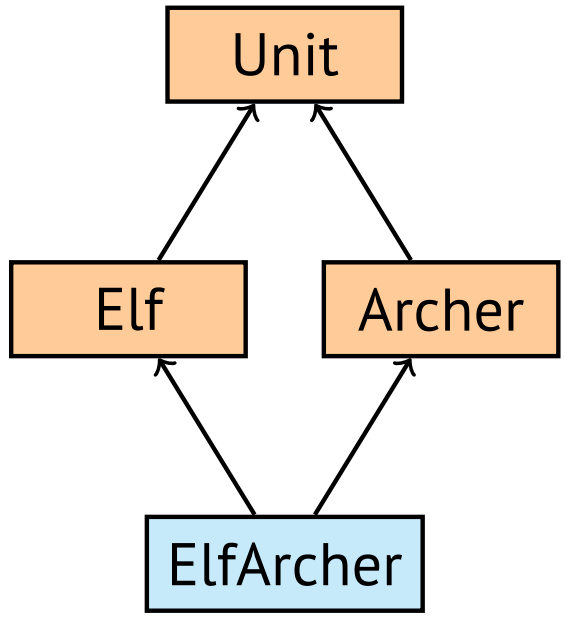


Порядок вызова конструкторов: `A, B, C, A, C, D, E`.
Деструкторы вызываются в обратном порядке.
Проблемы:

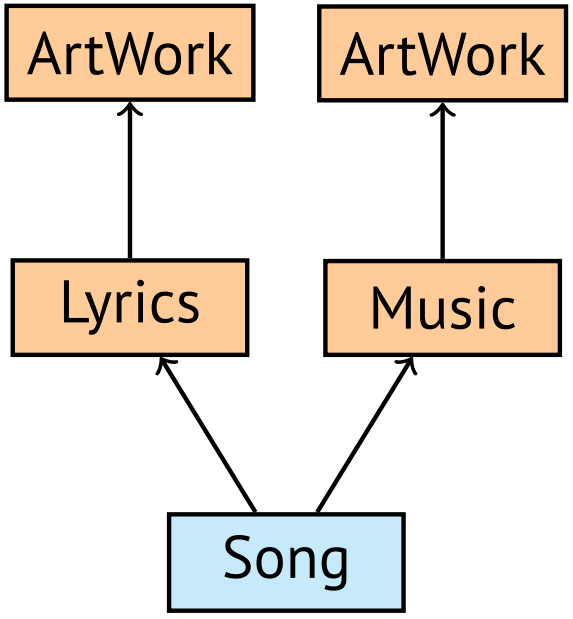
1. Дублирование A и C.
2. Недоступность первого C.

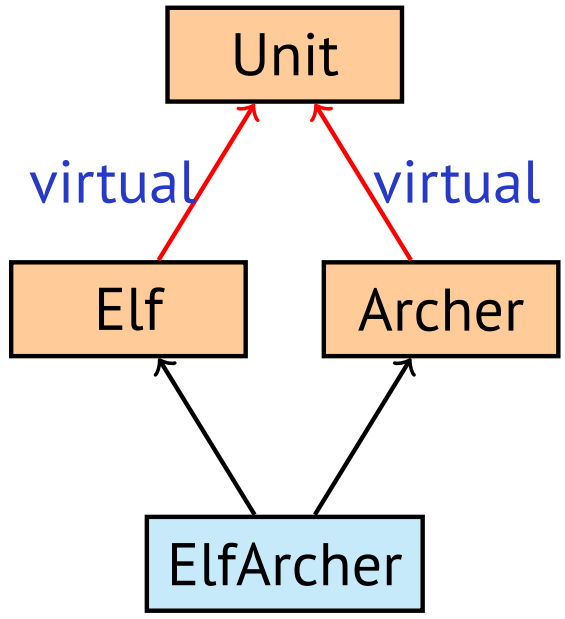
Виртуальное наследование





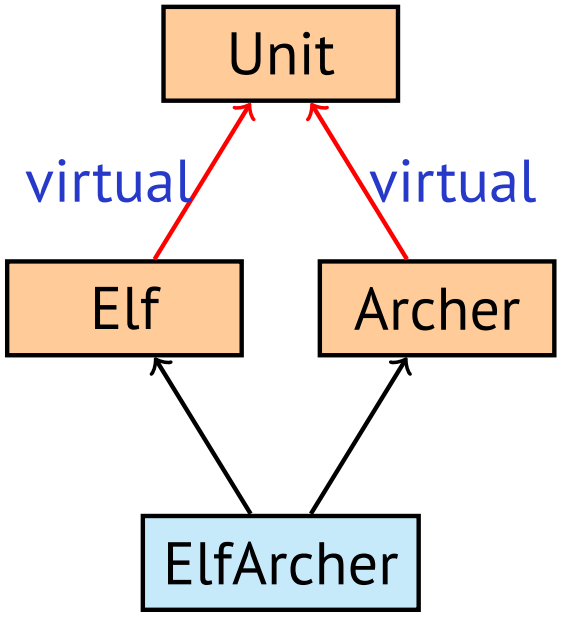
Виртуальное наследование

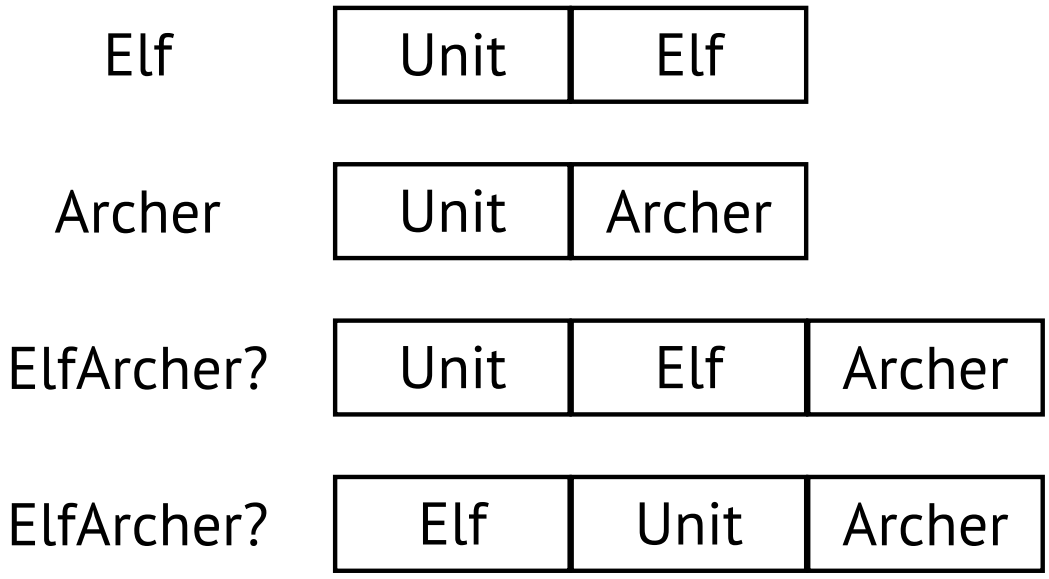




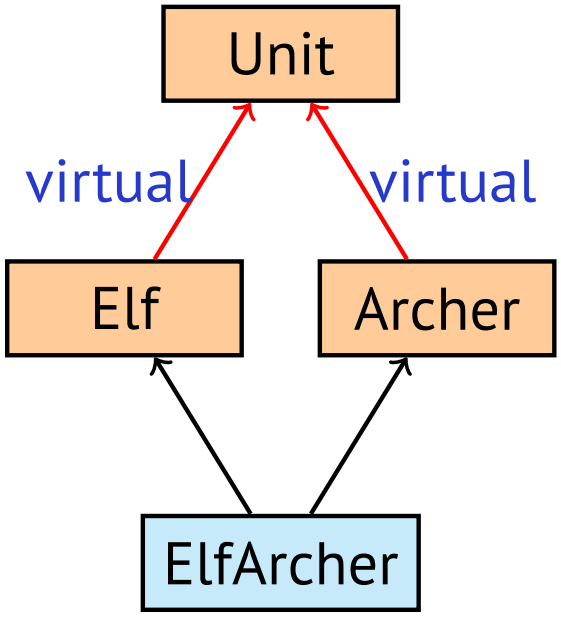

```
struct Unit {};
struct Elf:  virtual Unit {};
struct Archer: virtual Unit {};
struct ElfArcher: Elf, Archer {};
```

Как устроено расположение в памяти?





Как устроено расположение в памяти?



На самом деле.

Elf	Unit	Elf	
Archer	Unit	Archer	
ElfArcher?	Unit	Elf	Archer
ElfArcher?	Elf	Unit	Archer
Elf	Elf	Unit	
Archer	Archer	Unit	
ElfArcher	Elf	Archer	Unit

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; // (*)
```

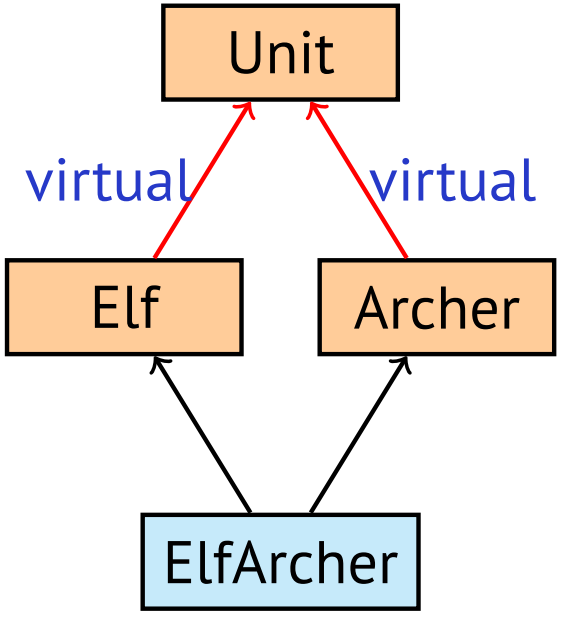
Строка (*) будет преобразована в строку

```
    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);
}
```

Указатели на функции

Кроме указателей на значения в C++ присутствуют три особенных типа указателей:

1. указатели на функции (унаследовано из C),
2. указатели на методы,
3. указатели на поля классов.

Указатели на функции

Кроме указателей на значения в C++ присутствуют три особенных типа указателей:

1. указатели на функции (унаследовано из C),
2. указатели на методы,
3. указатели на поля классов.

Указатели на функции (и методы) используются для

1. параметризации алгоритмов,
2. обратных вызовов (callback),
3. подписки на события (шаблон Listener),
4. создания очередей событий/заданий.

Указатели на функции: параметризация алгоритмов

```
void qsort (void* base, size_t num, size_t size,
          int (*compar)(void const*,void const*));
```

```
int doublecmp(void const * a, void const * b)
{
    double da = *static_cast<double const*>(a);
    double db = *static_cast<double const*>(b);
    if (da < db) return -1;
    if (da > db) return 1;
    return 0;
}

void sort(double * p, double * q)
{
    qsort(p, q - p, sizeof(double), &doublecmp);
}
```

Указатели на функции: параметризация алгоритмов

Упростим предыдущий пример и сделаем его типобезопасным:

```
void sort(int * p, int * q, bool (*cmp)(int, int))
{
    for (int * m = q; m != p; --m)
        for (int * r = p; r + 1 < m; ++r)
         // if ( *(r + 1) < *r )
            if ( cmp(*(r + 1), *r) )
                swap(*r, *(r + 1));
}

bool less   (int a, int b) { return a < b; }

bool greater(int a, int b) { return a > b; }

void sort(int * p, int * q, bool asc = true)
{
    sort(p, q, asc ? &less : &greater);
}
```

О полезности `typedef`

Что здесь объявлено?

```
char * (*func(int, int))(int, int, int *, float);
```

О полезности `typedef`

Что здесь объявлено?

```
char * (*func(int, int))(int, int, int *, float);
```

Функция двух целочисленных параметров, возвращающая указатель на функцию, которая возвращает указатель на `char` и имеет собственный список формальных параметров вида:
`(int, int, int *, float)`

О полезности `typedef`

Что здесь объявлено?

```
char * (*func(int, int))(int, int, int *, float);
```

Функция двух целочисленных параметров, возвращающая указатель на функцию, которая возвращает указатель на `char` и имеет собственный список формальных параметров вида:
`(int, int, int *, float)`

Как стоило это написать:

```
typedef char* (*MyFunction)(int,int,int*,float);

MyFunction func(int, int);
```

Указатели на методы: параметризация алгоритмов

Для вызова метода по указателю нужен объект.

```
struct Unit
{
    virtual unsigned id()    const;
    virtual unsigned hp()    const;
};

typedef unsigned (Unit::*UnitMethod)() const;

void sort(Unit* p, Unit* q, UnitMethod mtd)
{
    for (Unit * m = q; m != p; --m)
        for (Unit * r = p; r + 1 < m; ++r)
            if ( (r->*mtd)() > ((r+1)->*mtd)() )
                swap(*r, *(r+1));
}
```

```
    sort(p, q, &Unit::hp);
```

Указатели на поля: параметризация алгоритмов

Для обращения к полю по указателю нужен объект.

```
struct Unit
{
        unsigned id;
        unsigned hp;
};

typedef unsigned Unit::*UnitField;

void sort(Unit* p, Unit* q, UnitField f)
{
    for (Unit * m = q; m != p; --m)
        for (Unit * r = p; r + 1 < m; ++r)
            if ( (r->*f) > ((r+1)->*f) )
                swap(*r, *(r+1));
}
```

```
    sort(p, q, &Unit::id);
```

Резюме по синтаксису

Указатели на методы и поля класса.

```
struct Unit
{
    unsigned id() const;
    unsigned hp;
};
```

```
unsigned (Unit::*mtd)() const = &Unit::id;
unsigned  Unit::*fld          = &Unit::hp;

Unit u;
Unit * p = &u;

(u.*mtd)() == (p->*mtd)();
(u.*fld)   == (p->*fld);
```

Как такие указатели устроены?

Как такие указатели устроены?

Что хранится в указателе на функцию?

Как такие указатели устроены?

Что хранится в указателе на функцию?

Хранится адрес функции.

Как такие указатели устроены?

Что хранится в указателе на функцию?

Хранится адрес функции.

Что хранится в указателе на поле класса?

Как такие указатели устроены?

Что хранится в указателе на функцию?

Хранится адрес функции.

Что хранится в указателе на поле класса?

Хранится смещение поля от начала объекта.

Как такие указатели устроены?

Что хранится в указателе на функцию?

Хранится адрес функции.

Что хранится в указателе на поле класса?

Хранится смещение поля от начала объекта.

Что хранится в указателе на метод?

Как такие указатели устроены?

Что хранится в указателе на функцию?

Хранится адрес функции.

Что хранится в указателе на поле класса?

Хранится смещение поля от начала объекта.

Что хранится в указателе на метод?

Там хранятся:

Как такие указатели устроены?

Что хранится в указателе на функцию?

Хранится адрес функции.

Что хранится в указателе на поле класса?

Хранится смещение поля от начала объекта.

Что хранится в указателе на метод?

Там хранятся:

1. адрес метода,

Как такие указатели устроены?

Что хранится в указателе на функцию?

Хранится адрес функции.

Что хранится в указателе на поле класса?

Хранится смещение поля от начала объекта.

Что хранится в указателе на метод?

Там хранятся:

1. адрес метода,
2. номер в таблице виртуальных методов,

Как такие указатели устроены?

Что хранится в указателе на функцию?

Хранится адрес функции.

Что хранится в указателе на поле класса?

Хранится смещение поля от начала объекта.

Что хранится в указателе на метод?

Там хранятся:

1. адрес метода,
2. номер в таблице виртуальных методов,
3. смещение.

Зачем нужно смещение?

```
struct Elf {
    string secretName;
};

struct Archer {
    unsigned arrows() { return arrows_; }
    unsigned arrows_;
};

struct ElfArcher : Elf, Archer {};

void foo() {
    ElfArcher ea;
    unsigned (ElfArcher::*m)() = &Archer::arrows;
    (ea.*m)();
}
```

Важные моменты

Важные моменты

- Использование неинициализированных указателей на функции и методы влечёт неопределённое поведение.

Важные моменты

- Использование неинициализированных указателей на функции и методы влечёт неопределённое поведение.
- Для использования указателей на методы и поля классов нужны экземпляры этих классов.

Важные моменты

- Использование неинициализированных указателей на функции и методы влечёт неопределённое поведение.
- Для использования указателей на методы и поля классов нужны экземпляры этих классов.
- Указатели на методы и поля класса ни к чему не приводятся.

Важные моменты

- Использование неинициализированных указателей на функции и методы влечёт неопределённое поведение.
- Для использования указателей на методы и поля классов нужны экземпляры этих классов.
- Указатели на методы и поля класса ни к чему не приводятся.
- Указатель на статический метод — это указатель на функцию, а указатель на статическое поле — это обычный указатель.

Важные моменты

- Использование неинициализированных указателей на функции и методы влечёт неопределённое поведение.
- Для использования указателей на методы и поля классов нужны экземпляры этих классов.
- Указатели на методы и поля класса ни к чему не приводятся.
- Указатель на статический метод — это указатель на функцию, а указатель на статическое поле — это обычный указатель.
- В шаблонном коде указатель на функцию ведёт себя так же, как и объект класса с оператором `()`. Это позволяет использовать указатели на функции в качестве функторов.

Важные моменты

- Использование неинициализированных указателей на функции и методы влечёт неопределённое поведение.
- Для использования указателей на методы и поля классов нужны экземпляры этих классов.
- Указатели на методы и поля класса ни к чему не приводятся.
- Указатель на статический метод — это указатель на функцию, а указатель на статическое поле — это обычный указатель.
- В шаблонном коде указатель на функцию ведёт себя так же, как и объект класса с оператором `()`. Это позволяет использовать указатели на функции в качестве функторов.
- Используйте `typedef`!$=$).

Пространства имён

Пространства имён (namespaces) — это способ разграничения областей видимости имён в C++.

Пространства имён

Пространства имён (namespaces) — это способ разграничения областей видимости имён в C++.

Имена в C++:

1. имена переменных и констант,
2. имена функций,
3. имена структур и классов,
4. имена шаблонов,
5. синонимы типов (typedef-ы),
6. enum-ы и union-ы,
7. имена пространств имён.

Примеры

В C для избежания конфликта имён используются префиксы.
К примеру, имена в библиотеке Expat начинаются с XML_.

```
struct XML_Parser;
int XML_GetCurrentLineNumber(XML_Parser * parser);
```

Примеры

В C для избежания конфликта имён используются префиксы.
К примеру, имена в библиотеке Expat начинаются с XML_.

```
struct XML_Parser;
int XML_GetCurrentLineNumber(XML_Parser * parser);
```

В C++ это можно было бы записать так:

```
namespace XML {
    struct Parser;
    int GetCurrentLineNumber(Parser * parser);
}
```

Тогда полные имена структуры и функции будут соответственно:
`XML::Parser` и `XML::GetCurrentLineNumber`.

Описание пространства имён

1. Пространства имён могут быть вложенными:

```
namespace items { namespace food {
    struct Fruit {...};
}}
items::food::Fruit apple("Apple");
```

Описание пространства имён

1. Пространства имён могут быть вложенными:

```
namespace items { namespace food {
    struct Fruit {...};
}}
items::food::Fruit apple("Apple");
```

2. Определение пространств имён можно разделять:

```
namespace weapons { struct Bow { ... }; }
namespace items   {
    struct Scroll { ... };
    struct Artefact  { ... };
}
namespace weapons { struct Sword { ... }; }
```

Описание пространства имён

1. Пространства имён могут быть вложенными:

```
namespace items { namespace food {
    struct Fruit {...};
}}
items::food::Fruit apple("Apple");
```

2. Определение пространств имён можно разделять:

```
namespace weapons { struct Bow { ... }; }
namespace items   {
    struct Scroll { ... };
    struct Artefact  { ... };
}
namespace weapons { struct Sword { ... }; }
```

3. Классы и структуры определяют одноимённый namespace.

Доступ к именам

1. Внутри того же `namespace` все имена доступны напрямую.

Доступ к именам

1. Внутри того же `namespace` все имена доступны напрямую.
2. `NS::` позволяет обратиться внутрь пространства имён NS.

```
namespace NS { int foo() { return 0; } }
int i = NS::foo();
```

Доступ к именам

1. Внутри того же `namespace` все имена доступны напрямую.
2. `NS::` позволяет обратиться внутрь пространства имён NS.

```
namespace NS { int foo() { return 0; } }
int i = NS::foo();
```

3. Оператор `::` позволяет обратиться к *глобальному пространству имён*.

```
struct Dictionary {...};

namespace items
{
    struct Dictionary {...};

    ::Dictionary globalDictionary;
}
```

Поиск имён

Поиск имён — это процесс разрешения имени.

1. Если такое имя есть в текущем `namespace`
   - выдать *все* одноимённые сущности в текущем `namespace`.
   - завершить поиск.
2. Если текущий `namespace` — глобальный
   - завершить поиск и выдать ошибку.
3. Текущий `namespace` $\leftarrow$ родительский `namespace`.
4. Перейти на шаг 1.

Поиск имён

```
int foo(int i) { return 1; }

namespace ru
{
    int foo(float f) { return 2; }

    int foo(double a, double b) { return 3; }

    namespace spb {
        int global = foo(5);
    }
}
```

Важно: поиск продолжается до первого совпадения.
В перегрузке участвуют только найденные к этому моменту функции.

Ключевое слово using

Существуют два различных использования слова using.

```
namespace ru
{
    namespace msk {
        int foo(int i) { return 1; }
        int bar(int i) { return -1; }
    }

    using namespace msk; // все имена из msk
    using msk::foo;      // только msk::foo

    int foo(float f) { return 2; }
    int foo(double a, double b) { return 3; }

    namespace spb {
        int global = foo(5);
    }
}
```

Поиск Кёнига

```
namespace cg {
    struct Vector2 {...};
    Vector2 operator+(Vector2 a, Vector2 const& b);
}
```

Поиск Кёнига

```
namespace cg {
    struct Vector2 {...};
    Vector2 operator+(Vector2 a, Vector2 const& b);
}
```

```
    cg::Vector2 a(1,2);
    cg::Vector2 b(3,4);
    b = a + b; // эквивалентно: b = operator+(a, b)
    b = cg::operator+(a, b); // OK
```

Поиск Кёнига

```
namespace cg {
    struct Vector2 {...};
    Vector2 operator+(Vector2 a, Vector2 const& b);
}
```

```
    cg::Vector2 a(1,2);
    cg::Vector2 b(3,4);
    b = a + b; // эквивалентно: b = operator+(a, b)
    b = cg::operator+(a, b); // OK
```

Argument-dependent name lookup (ADL, Поиск Кёнига)

При поиске имени функции на первой фазе рассматриваются имена из текущего пространства имён *и пространств имён, к которым принадлежат аргументы функции*.

Безымянный namespace

Пространство имён с гарантированно уникальным именем.

```
namespace { // безымянный namespace
    struct Test { std::string name; };
}
```

Это эквивалентно:

```
namespace $GeneratedName$ {
    struct Test { std::string name; };
}
using namespace $GeneratedName$;
```

Безымянные пространства имён — замена для `static`.

Заключение

Заключение

1. Используйте пространства имён для исключения конфликта имён.

Заключение

1. Используйте пространства имён для исключения конфликта имён.
2. Помните, что поиск имён прекращается после первого совпадения. Используйте `using` и полные имена.

Заключение

1. Используйте пространства имён для исключения конфликта имён.
2. Помните, что поиск имён прекращается после первого совпадения. Используйте `using` и полные имена.
3. Не используйте `using namespace` в заголовочных файлах.

Заключение

1. Используйте пространства имён для исключения конфликта имён.
2. Помните, что поиск имён прекращается после первого совпадения. Используйте `using` и полные имена.
3. Не используйте `using namespace` в заголовочных файлах.
4. Всегда определяйте операторы в том же пространстве имён, что и типы, для которых они определены.

Заключение

1. Используйте пространства имён для исключения конфликта имён.
2. Помните, что поиск имён прекращается после первого совпадения. Используйте `using` и полные имена.
3. Не используйте `using namespace` в заголовочных файлах.
4. Всегда определяйте операторы в том же пространстве имён, что и типы, для которых они определены.
5. Используйте безымянные пространства имён для маленьких локальных классов и как замену слова `static`.

Заключение

1. Используйте пространства имён для исключения конфликта имён.
2. Помните, что поиск имён прекращается после первого совпадения. Используйте `using` и полные имена.
3. Не используйте `using namespace` в заголовочных файлах.
4. Всегда определяйте операторы в том же пространстве имён, что и типы, для которых они определены.
5. Используйте безымянные пространства имён для маленьких локальных классов и как замену слова `static`.
6. Для длинных имён `namespace`-ов используйте синонимы:

```
namespace csccpp17 = ru::spb::csc::cpp17;
```