Программирование на языке C++
Лекция 7

Множественное наследование

Александр Смаль

Множественное наследование

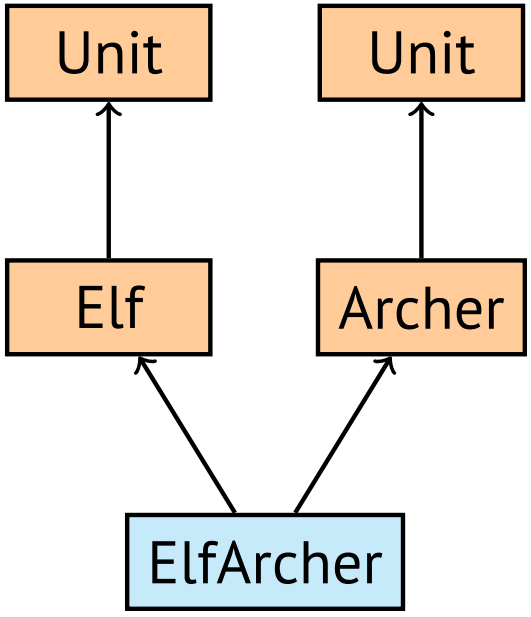
Множественное наследование (multiple inheritance) — возможность наследовать сразу несколько классов.

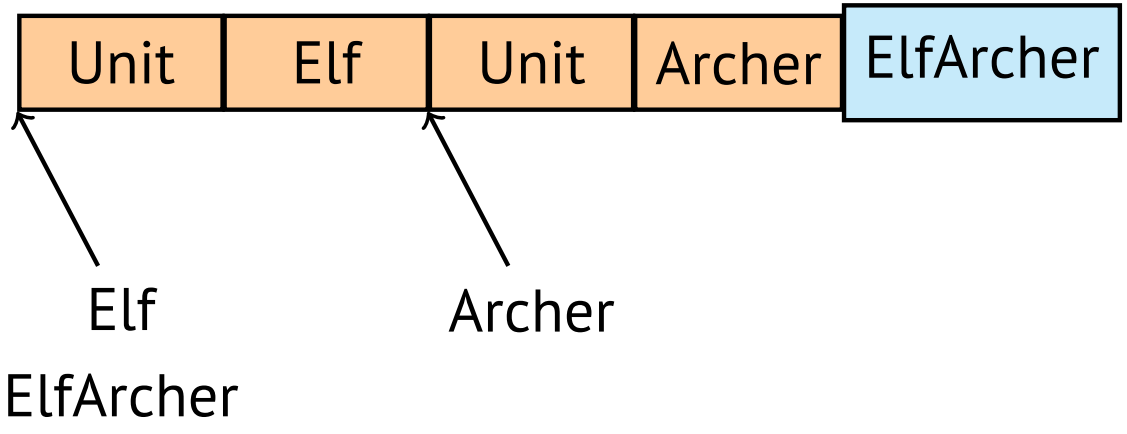
```
struct Unit {
    Unit(unitid id, int hp): id_(id), hp_(hp) {}
    virtual unitid id() const { return id_; }
    virtual int    hp() const { return hp_; }
private:
    unitid id_;
    int    hp_;
};

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

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

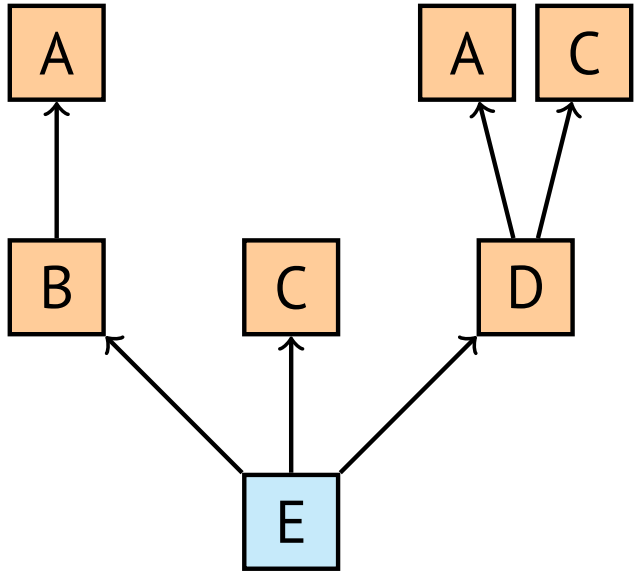
Представление в памяти





Важно: указатели при приведении могут смещаться.

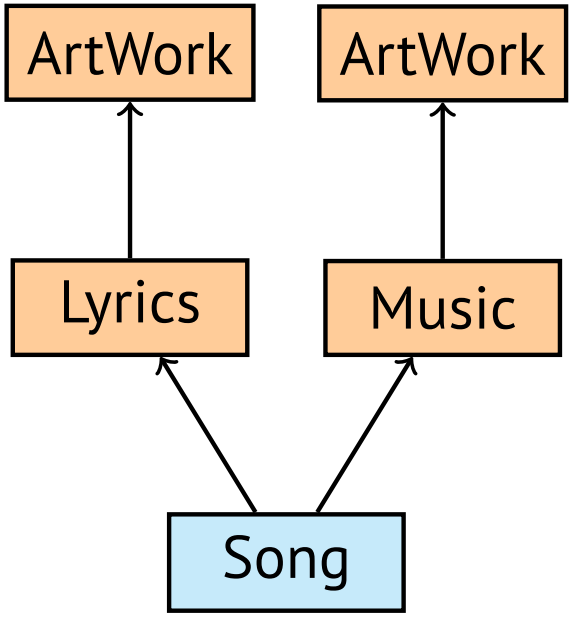
Создание и удаление объекта

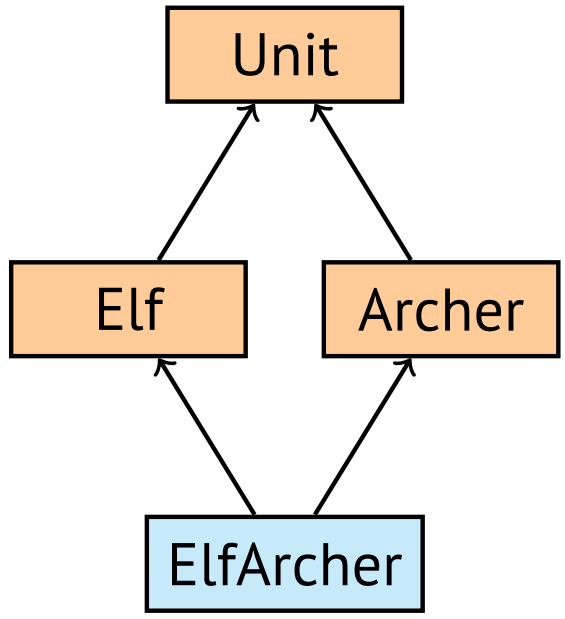


Порядок вызова конструкторов: `A, B, C, A, C, D, E`.
Деструкторы вызываются в обратном порядке.
Проблемы:

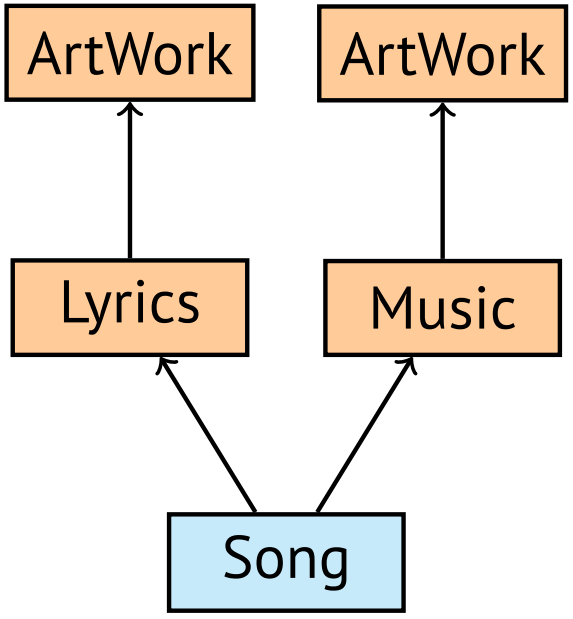
1. Дублирование A и C.
2. Недоступность первого C.

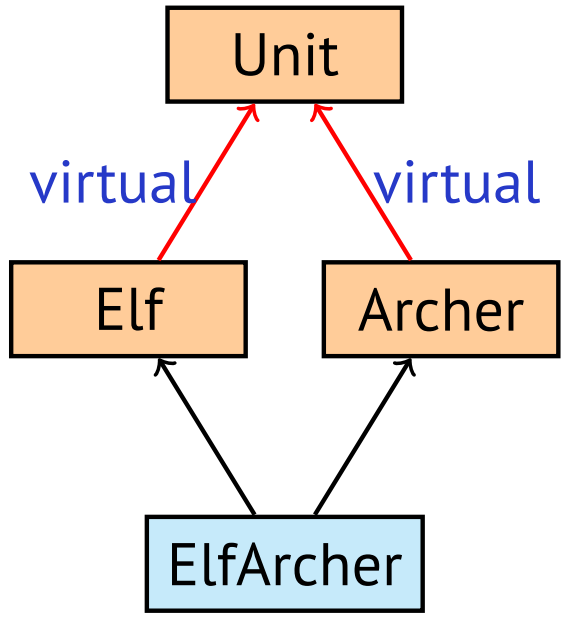
Виртуальное наследование





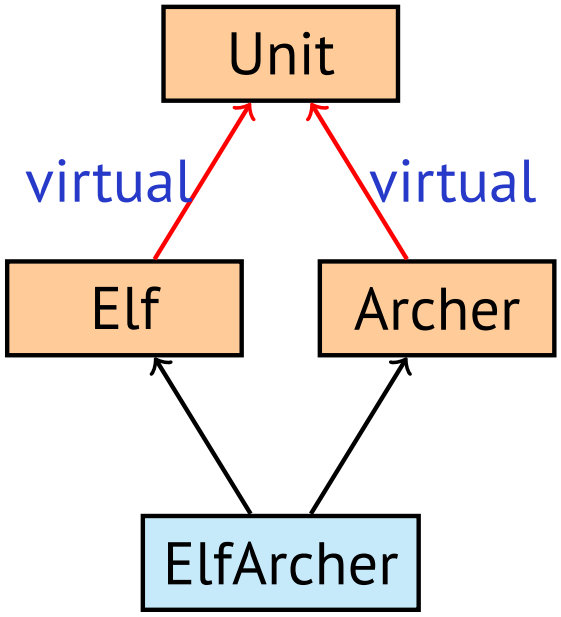
Виртуальное наследование

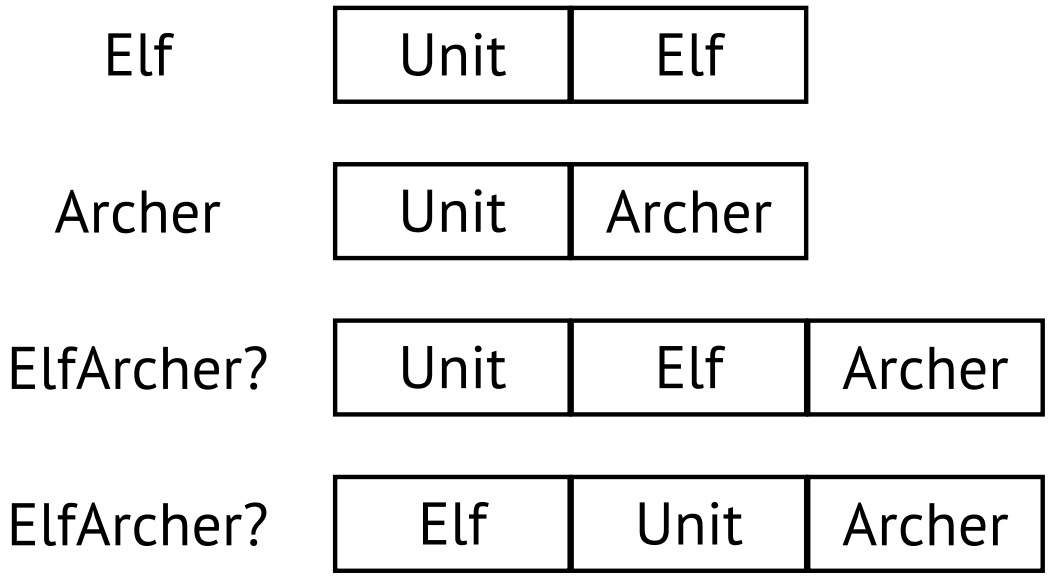




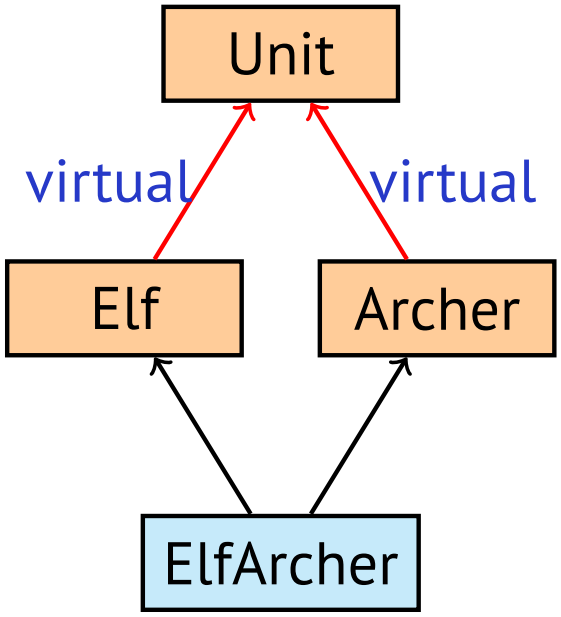

```
struct Unit {};
struct Elf:  virtual Unit {};
struct Archer: virtual Unit {};
struct ElfArcher: Elf, Archer {};
```

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; // (*)
```

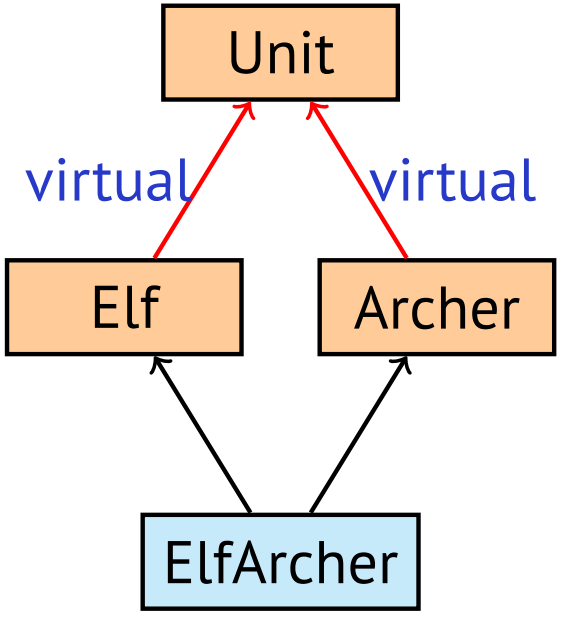
Строка (*) будет преобразована в строку

```
    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.

Программирование на языке C++
Лекция 7

Преобразование в стиле C++

Александр Смаль

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`.

Программирование на языке C++
Лекция 7

Информации о типах времени выполнения

Александр Смаль

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);
}
```

Программирование на языке C++
Лекция 7

Указатели на функции

Александр Смаль

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);
```

Программирование на языке C++
Лекция 7

Указатели на методы и поля класса

Александр Смаль

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`!$=$).

Программирование на языке C++
Лекция 7

Пространства имён

Александр Смаль

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;
```

Программирование на языке C++
Лекция 8

Стандарты C++11/C++14

Александр Смаль

Стандартизация C++

Стандартизация C++

1983 Появление C++.

Стандартизация C++

1983 Появление C++.

1998 Первый стандарт ISO/IEC 14882:1998.

Стандартизация C++

1983 Появление C++.

1998 Первый стандарт ISO/IEC 14882:1998.

2003 Стандарт ISO/IEC 14882:2003, исправляющий недостатки стандарта C++98.

Стандартизация C++

1983 Появление C++.

1998 Первый стандарт ISO/IEC 14882:1998.

2003 Стандарт ISO/IEC 14882:2003, исправляющий недостатки стандарта C++98.

2011 Стандарт ISO/IEC 14882:2011.

Стандартизация C++

1983 Появление C++.

1998 Первый стандарт ISO/IEC 14882:1998.

2003 Стандарт ISO/IEC 14882:2003, исправляющий недостатки стандарта C++98.

2011 Стандарт ISO/IEC 14882:2011.

2014 Стандарт ISO/IEC 14882:2014, исправляющий недостатки стандарта C++11.

Стандартизация C++

1983 Появление C++.

1998 Первый стандарт ISO/IEC 14882:1998.

2003 Стандарт ISO/IEC 14882:2003, исправляющий недостатки стандарта C++98.

2011 Стандарт ISO/IEC 14882:2011.

2014 Стандарт ISO/IEC 14882:2014, исправляющий недостатки стандарта C++11.

2017 К концу года планируется выход нового стандарта.

Основные принципы разработки стандарта

Основные принципы разработки стандарта

- поддержка совместимости с предыдущими стандартами;

Основные принципы разработки стандарта

- поддержка совместимости с предыдущими стандартами;
- улучшение техники программирования;

Основные принципы разработки стандарта

- поддержка совместимости с предыдущими стандартами;
- улучшение техники программирования;
- улучшение C++ с точки зрения дизайна;

Основные принципы разработки стандарта

- поддержка совместимости с предыдущими стандартами;
- улучшение техники программирования;
- улучшение C++ с точки зрения дизайна;
- увеличение типобезопасности для обеспечения безопасной альтернативы для существующих опасных подходов;

Основные принципы разработки стандарта

- поддержка совместимости с предыдущими стандартами;
- улучшение техники программирования;
- улучшение C++ с точки зрения дизайна;
- увеличение типобезопасности для обеспечения безопасной альтернативы для существующих опасных подходов;
- увеличение производительности;

Основные принципы разработки стандарта

- поддержка совместимости с предыдущими стандартами;
- улучшение техники программирования;
- улучшение C++ с точки зрения дизайна;
- увеличение типобезопасности для обеспечения безопасной альтернативы для существующих опасных подходов;
- увеличение производительности;
- «не платить за то, что не используешь»;

Основные принципы разработки стандарта

- поддержка совместимости с предыдущими стандартами;
- улучшение техники программирования;
- улучшение C++ с точки зрения дизайна;
- увеличение типобезопасности для обеспечения безопасной альтернативы для существующих опасных подходов;
- увеличение производительности;
- «не платить за то, что не используешь»;
- введение новых возможностей через стандартную библиотеку, а не через ядро языка;

Основные принципы разработки стандарта

- поддержка совместимости с предыдущими стандартами;
- улучшение техники программирования;
- улучшение C++ с точки зрения дизайна;
- увеличение типобезопасности для обеспечения безопасной альтернативы для существующих опасных подходов;
- увеличение производительности;
- «не платить за то, что не используешь»;
- введение новых возможностей через стандартную библиотеку, а не через ядро языка;
- сделать C++ проще для изучения (сохраняя возможности, используемые программистами-экспертами).

Мелкие улучшения

Мелкие улучшения

1. Исправлена проблема с угловыми скобками: `T<U<int>>`.

Мелкие улучшения

1. Исправлена проблема с угловыми скобками: `T<U<int>>`.
2. Определены понятия “тривиальный класс” и “класс со стандартным размещением”.

Мелкие улучшения

1. Исправлена проблема с угловыми скобками: `T<U<int>>`.
2. Определены понятия “тривиальный класс” и “класс со стандартным размещением”.
3. Ключевое слово `explicit` для оператора приведения типа.

```
explicit operator bool () { ... }
```

Мелкие улучшения

1. Исправлена проблема с угловыми скобками: `T<U<int>>`.
2. Определены понятия “тривиальный класс” и “класс со стандартным размещением”.
3. Ключевое слово `explicit` для оператора приведения типа.

```
explicit operator bool () { ... }
```

4. Шаблонный `typedef`

```
template<class A, class B, int N>
class SomeType;

template<typename B>
using TypedefName = SomeType<double, B, 5>;
```

Мелкие улучшения

1. Исправлена проблема с угловыми скобками: `T<U<int>>`.
2. Определены понятия “тривиальный класс” и “класс со стандартным размещением”.
3. Ключевое слово `explicit` для оператора приведения типа.

```
explicit operator bool () { ... }
```

4. Шаблонный `typedef`

```
template<class A, class B, int N>
class SomeType;

template<typename B>
using TypedefName = SomeType<double, B, 5>;
```

```
typedef void (*OtherType)(double);
using OtherType = void (*)(double);
```

Мелкие улучшения (продолжение)

Мелкие улучшения (продолжение)

5. Добавлен тип `long long int`.

Мелкие улучшения (продолжение)

5. Добавлен тип `long long int`.
6. Добавлена библиотека поддержки типов: по типу на этапе компиляции можно узнавать его свойства (см. заголовочный файл `<type_traits>`).

Мелкие улучшения (продолжение)

5. Добавлен тип `long long int`.
6. Добавлена библиотека поддержки типов: по типу на этапе компиляции можно узнавать его свойства (см. заголовочный файл <type_traits>).
7. Добавлены операторы `alignof` и `alignas`.

```
alignas(float) unsigned char c[sizeof(float)];
```

Мелкие улучшения (продолжение)

5. Добавлен тип `long long int`.
6. Добавлена библиотека поддержки типов: по типу на этапе компиляции можно узнавать его свойства (см. заголовочный файл <type_traits>).
7. Добавлены операторы `alignof` и `alignas`.

```
alignas(float) unsigned char c[sizeof(float)];
```

8. Добавлен `static_assert`

```
template <class T>
void run(T * data, size_t n) {
   static_assert(std::is_signed<T>::value,
                 "T is not signed.");
}
```

nullptr

В язык добавлены тип `std::nullptr_t` и литерал `nullptr`.

```
void foo(int a)     { ... }

void foo(int * p)   { ... }

void bar()
{
    foo(0); // вызов foo(int a)
    foo((int *) 0); // C++98
    foo(nullptr);   // C++11
}
```

Тип `std::nullptr_t` имеет единственное значение `nullptr`, которое неявно приводится к нулевому указателю на любой тип.

Вывод типов

```
Array<Unit *> units;

for(size_t i = 0; i != units.size(); ++i) {
    // Unit *
    auto u = units[i];

    // Array<Item> const &
    decltype(u->items()) items = u->items();
    ...
```

Вывод типов

```
Array<Unit *> units;

for(size_t i = 0; i != units.size(); ++i) {
    // Unit *
    auto u = units[i];

    // Array<Item> const &
    decltype(u->items()) items = u->items();
    ...
```

```
    auto a = items[0];          // a - Item
    decltype(items[0]) b = a; // b - Item const &

    decltype(a)    c = a;      // c - Item
    decltype((a))  d = a;      // d - Item &

    decltype(b)    e = b;      // e - Item const &
    decltype((b))  f = b;      // f - Item const &
```

Альтернативный синтаксис для функций

```
// RETURN_TYPE = ?
template <typename A, typename B>
RETURN_TYPE Plus(A a, B b) { return a + b; }
```

```
// некорректно, a и b определены позже
template <typename A, typename B>
decltype(a + b) Plus(A a, B b) { return a + b; }
```

```
// C++11
template <typename A, typename B>
auto Plus(A a, B b) -> decltype(a + b) {
    return a + b;
}
```

```
// C++14
template <typename A, typename B>
auto Plus(A a, B b) {
    return a + b;
}
```

Шаблоны с переменным числом аргументов

```
void printf(char const *s) {
    while (*s) {
        if (*s == '%' && *(++s) != '%')
            // обработать ошибку
        std::cout << *s++;
    }
}
template<typename T, typename... Args>
void printf(char const *s, T value, Args... args) {
    while (*s) {
        if (*s == '%' && *(++s) != '%') {
            std::cout << value;
            printf(++s, args...);
            return;
        }
        std::cout << *s++;
    }
    // обработать ошибку
}
```

Ключевые слова default и delete

```
struct SomeType {
    SomeType() = default; // Конструктор по умолчанию.
    SomeType(OtherType value);
};

struct NonCopyable {
    NonCopyable() = default;
    NonCopyable(const NonCopyable&) = delete;
    NonCopyable & operator=(const NonCopyable&) = delete;
};
```

Ключевые слова default и delete

```
struct SomeType {
    SomeType() = default; // Конструктор по умолчанию.
    SomeType(OtherType value);
};

struct NonCopyable {
    NonCopyable() = default;
    NonCopyable(const NonCopyable&) = delete;
    NonCopyable & operator=(const NonCopyable&) = delete;
};
```

Удалять можно и обычные функции.

```
template<class T>
void foo(T const * p) { ... }

void foo(char const *) = delete;
```

Делегация конструкторов

```
struct SomeType  {
    SomeType(int newNumber): number(newNumber) {}
    SomeType() : SomeType(42) {}
private:
    int number;
};
struct SomeClass {
    SomeClass() {}
    explicit SomeClass(int newValue): value(newValue) {}
private:
    int value = 5;
};
struct BaseClass {
    BaseClass(int value);
};
struct DerivedClass : public BaseClass {
    using BaseClass::BaseClass;
};
```

Явное переопределение и финальность

```
struct Base {
    virtual void update();
    virtual void foo(int);
    virtual void bar() const;
};
struct Derived : Base {
    void updata() override;                  // error
    void foo(int) override;                  // OK
    virtual void foo(long) override;         // error
    virtual void foo(int) const override; // error
    virtual int  foo(int) override;          // error
    virtual void bar(long);                  // OK
    virtual void bar() const final;          // OK
};
struct Derived2 final : Derived {
    virtual void bar() const;          // error
};
struct Derived3 : Derived2 {};         // error
```

Программирование на языке C++
Лекция 8

Семантика перемещения

Александр Смаль

Излишнее копирование

```
struct String {
    String() = default;
    String(String const & s);
    String & operator=(String const & s);
    //...
private:
    char * data_ = nullptr;
    size_t size_ = 0;
};
```

```
String getCurrentDateString() {
    String date;
    // date заполняется "21 октября 2015 года"
    return date;
}
```

```
String date = getCurrentDateString();
```

Перемещающий конструктор и перемещающий оператор присваивания

```
struct String
{
    String (String && s) // && - rvalue reference
        : data_(s.data_)
        , size_(s.size_) {
        s.data_ = nullptr;
        s.size_ = 0;
    }
    String & operator = (String && s) {
        delete [] data_;
        data_ = s.data_;
        size_ = s.size_;
        s.data_ = nullptr;
        s.size_ = 0;
        return *this;
    }
};
```

Перемещающие методы при помощи swap

```
#include<utility>

struct String
{
    void swap(String & s) {
        std::swap(data_, s.data_);
        std::swap(size_, s.size_);
    }

    String (String && s) {
        swap(s);
    }

    String & operator = (String && s) {
        swap(s);
        return *this;
    }
};
```

Использование перемещения

```
struct String {
    String() = default;
    String(String const & s);  // lvalue-reference
    String & operator=(String const & s);
    String(String && s);       // rvalue-reference
    String & operator=(String && s);
private:
    char * data_ = nullptr;
    size_t size_ = 0;
};
```

```
String getCurrentDateString() {
    String date;
    // date заполняется "21 октября 2015 года"
    return std::move(date);
}
```

```
String date = getCurrentDateString();
```

Перегрузка с lvalue/rvalue ссылками

При перегрузке перемещающий метод вызывается для временных объектов и для явно перемещённых с помощью `std::move`.

```
String a(String("Hello"));  // перемещение

String b(a);                // копирование

String c(std::move(b));     // перемещение

a = b;                      // копирование

b = std::move(c);           // перемещение

c = String("world");        // перемещение
```

Это касается и обычных методов и функций,
которые принимают lvalue/rvalue-ссылки.

Перемещающие особые методы

- Особые методы класса:
  - конструктор по умолчанию,
  - конструктор копирования,
  - оператор присваивания,
  - деструктор,
  - перемещающий конструктор,
  - перемещающий оператор присваивания.

Перемещающие особые методы

- Особые методы класса:
  - конструктор по умолчанию,
  - конструктор копирования,
  - оператор присваивания,
  - деструктор,
  - перемещающий конструктор,
  - перемещающий оператор присваивания.
- Перемещающие методы генерируются только, если в классе отсутствуют пользовательские копирующие операции, перемещающие операции и деструктор.

Перемещающие особые методы

- Особые методы класса:
  - конструктор по умолчанию,
  - конструктор копирования,
  - оператор присваивания,
  - деструктор,
  - перемещающий конструктор,
  - перемещающий оператор присваивания.
- Перемещающие методы генерируются только, если в классе отсутствуют пользовательские копирующие операции, перемещающие операции и деструктор.
- Генерация копирующих методов для классов с пользовательским конструктором признана устаревшей.

Пример: unique_ptr

```
#include <memory>
#include "units.hpp"

void foo(std::unique_ptr<Unit> p);

std::unique_ptr<Unit> bar();

int main() {
    // p1 владеет указателем
    std::unique_ptr<Unit> p1(new Elf());

    // теперь p2 владеет указателем
    std::unique_ptr<Unit> p2(std::move(p1));

    p1 = std::move(p2);  // владение передаётся p1

    foo(std::move(p1));  // p1 передаётся в foo

    p2 = bar();  // std::move не нужен
}
```

Программирование на языке C++
Лекция 8

Ещё о нововведениях C++11/C++14

Александр Смаль

Кортежи

```
std::tuple<std::string, int, int> getUnitInfo(int id) {
    if (id == 0) return std::make_tuple("Elf",   60, 9);
    if (id == 1) return std::make_tuple("Dwarf", 80, 6);
    if (id == 2) return std::make_tuple("Orc",   90, 3);
    //...
}
int main() {
    auto ui0 = getUnitInfo(0);
    std::cout << "race: "<< std::get<0>(ui0) << ", "
              << "hp: "  << std::get<1>(ui0) << ", "
              << "iq: "  << std::get<2>(ui0) << "\n";

    std::string race1; int hp1; int iq1;
    std::tie(race1, hp1, iq1) = getUnitInfo(1);
    std::cout << "race: " << race1 << ", "
              << "hp:   " << hp1   << ", "
              << "iq:   " << iq1   << "\n";
}
```

Константные выражения

Для констант и функций времени компиляции.

```
constexpr double accOfGravity = 9.8;
constexpr double moonGravity = accOfGravity / 6;

constexpr int pow(int x, int k)
{ return k == 0 ? 1 : x * pow(x, k - 1); }

int data[pow(3, 5)] = {};
```

```
struct Point {
   double x, y;
   constexpr Point(double x = 0, double y = 0)
       : x(x), y(y) {}
   constexpr double getX() const { return x; }
   constexpr double getY() const { return y; }
};
constexpr Point p(moonGravity, accOfGravity);
constexpr auto x = p.getX();
```

Range-based for

Синтаксическая конструкция для работы с последовательностями.

```
int array[] = {1, 4, 9, 16, 25, 36, 49};

int sum = 0;
// по значению
for (int x : array) {
    sum += x;
}

// по ссылке
for (int & x : array) {
    x *= 2;
}
```

Применим к встроенным массивам, спискам инициализации, контейнерам из стандартной библиотеки и любым другим типам, для которых определены функции `begin()` и `end()`, возвращающие итераторы (об этом будет рассказано дальше).

Списки инициализации

Возможность передать в функцию список значений.

```
// в конструкторах массивов и других контейнеров
template<typename T>
struct Array {
    Array(std::initializer_list<T> list);
};

Array<int> primes = {2, 3, 5, 7, 11, 13, 17};
```

```
// в обычных функциях
int sum(std::initializer_list<int> list) {
    int result = 0;
    for (int x : list)
        result += x;
    return result;
}

int s = sum({1, 1, 2, 3, 5, 8, 13, 21});
```

Универсальная инициализация

```
struct CStyleStruct {
    int x;
    double y;
};
struct CPPStyleStruct {
    CPPStyleStruct(int x, double y): x(x), y(y) {}
    int x;
    double y;
};
```

```
// C++03
CStyleStruct    s1 = {19, 72.0};// инициализация
CPPStyleStruct  s2(19, 83.0);   // вызов конструктора
```

```
// C++11
CStyleStruct    s1{19, 72.0}; // инициализация
CPPStyleStruct  s2{19, 83.0}; // вызов конструктора
```

```
// тип не обязателен
CStyleStruct getValue() { return {6, 4.2}; }
```

std::function

Универсальный класс для хранения указателей на функции, указателей на методы и функциональных объектов.

```
int mult (int x, int y) { return x * y; }

struct IntDiv {
    int operator()(int x, int y) const {
        return x / y;
    }
};
```

```
std::function<int (int, int)> op;
if ( OP == '*' )
    op = &mult;
else if ( OP == '/')
    op = IntDiv();
int result = op(7,8);
```

Позволяет работать и с указателями на методы.

Лямбда-выражения

```
std::function<int (int, int)> op =
    [](int x, int y) { return x / y; } // IntDiv

// то же, но с указанием типа возвращаемого значения
op = [](int x, int y) -> int { return x / y; }

// C++14
op = [](auto x, auto y) { return x / y; }
```

Можно захватывать *локальные* переменные.

```
// захват по ссылке
int total = 0;
auto addToTotal = [&total](int x) { total += x; };

// захват по значению
auto subTotal = [total](int & x) { x -= total ; };

// Можно захватывать this
auto callUpdate = [this](){ this->update(); };
```

Различные виды захвата

Могут быть разные типы захвата, в т.ч. смешанные:
`[]`, `[x, &y]`, `[&]`, `[=]`, `[&, x]`, `[=, &z]`

Перемещающий захват `[x = std::move(y)]` (только в C++14).

Не стоит использовать захват по умолчанию `[&]` или `[=]`.

```
std::function<bool(int)> getFilter(Checker const& c) {
    auto d  = c.getModulo();
    // захватывает ссылку на локальную переменную
    return [&] (int i) { return i % d == 0; }
}
```

```
struct Checker {
    std::function<bool(int)> getFilter() const {
        // захватывает this, а не d
        return [=] (int x) { return x % d == 0; }
    }
    int d;
};
```

Новые строковые литералы

```
u8"I'm a UTF-8 string."           // char[]
u"This is a UTF-16 string."       // char_16_t[]
U"This is a UTF-32 string."       // char_32_t[]
L"This is a wide-char string."    // wchar_t[]

u8"This is a Unicode Character: \u2018."
u"This is a bigger Unicode Character: \u2018."
U"This is a Unicode Character: \U00002018."

R"(The String Data \ Stuff " )"
R"delimiter(The String Data \ Stuff " )delimiter"

LR"(Raw wide string literal \t (without a tab))"
u8R"XXX(I'm a "raw UTF-8" string.)XXX"
uR"*(This is a "raw UTF-16" string.)*"
UR"(This is a "raw UTF-32" string.)"
```

Программирование на языке C++
Лекция 8

Как работают rvalue-ссылки

Александр Смаль

Преобразование ссылок в шаблонах

“Склейка” ссылок:

- `T& &` $\rightarrow$ `T&`
- `T& &&` $\rightarrow$ `T&`
- `T&& &` $\rightarrow$ `T&`
- `T&& &&` $\rightarrow$ `T&&`

Универсальная ссылка

```
template<typename T>
void foo(T && t) {}
```

- Если вызвать `foo` от lvalue типа A, то `T` = A&.
- Если вызвать `foo` от rvalue типа A, то `T` = A.

Как работает std::move?

Определение std::move:

```
template<class T>
typename remove_reference<T>::type&&
   move(T&& a)
{
   typedef typename remove_reference<T>::type&& RvalRef;
   return static_cast<RvalRef>(a);
}
```

Замечание

std::move не выполняет никаких действий времени выполнения.

std::move для lvalue

Вызываем std::move для lvalue объекта.

```
X x;
x = std::move(x);
```

Тип T выводится как X&.

```
typename remove_reference<X&>::type&&
    move(X& && a)
{
    typedef typename remove_reference<X&>::type&& RvalRef;
    return static_cast<RvalRef>(a);
}
```

После склейки ссылок получаем:

```
X&& move(X& a)
{
    return static_cast<X&&>(a);
}
```

std::move для rvalue

Вызываем std::move для временного объекта.

```
X x = std::move(X());
```

Тип T выводится как X.

```
typename remove_reference<X>::type&&
   move(X&& a)
{
   typedef typename remove_reference<X>::type&& RvalRef;
   return static_cast<RvalRef>(a);
}
```

После склейки ссылок получаем:

```
X&& move(X&& a)
{
   return static_cast<X&&>(a);
}
```

Perfect forwarding

```
// для lvalue
template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg & arg) {
    return unique_ptr<T>(new T(arg));
}

// для rvalue
template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg && arg) {
    return unique_ptr<T>(new T(std::move(arg)));
}
```

`std::forward` позволяет записать это одной функцией.

```
template<typename T, typename Arg>
unique_ptr<T> make_unique(Arg&& arg) {
    return unique_ptr<T>(
        new T(std::forward<Arg>(arg)));
}
```

Как работает `std::forward`?

Определение `std::forward`:

```
template<class S>
S&& forward(typename remove_reference<S>::type& a)
{
    return static_cast<S&&>(a);
}
```

Замечание

`std::forward` не выполняет никаких действий
времени выполнения.

std::forward для lvalue

```
X x;
auto p = make_unique<A>(x);      // Arg = X&
```

```
unique_ptr<A> make_unique(X& && arg) {
  return unique_ptr<A>(new A(std::forward<X&>(arg)));
}

X& && forward(remove_reference<X&>::type& a) {
  return static_cast<X& &&>(a);
}
```

std::forward для lvalue

```
X x;
auto p = make_unique<A>(x);      // Arg = X&
```

```
unique_ptr<A> make_unique(X& && arg) {
  return unique_ptr<A>(new A(std::forward<X&>(arg)));
}

X& && forward(remove_reference<X&>::type& a) {
  return static_cast<X& &&>(a);
}
```

После склейки ссылок:

```
unique_ptr<A> make_unique(X& arg) {
  return unique_ptr<A>(new A(std::forward<X&>(arg)));
}

X& forward(X& a) {
  return static_cast<X&>(a);
}
```

std::forward для rvalue

```
auto p = make_unique<A>(X());    // Arg = X
```

```
unique_ptr<A> make_unique(X&& arg) {
  return unique_ptr<A>(new A(std::forward<X>(arg)));
}

X&& forward(remove_reference<X>::type& a) {
  return static_cast<X&&>(a);
}
```

std::forward для rvalue

```
auto p = make_unique<A>(X());    // Arg = X
```

```
unique_ptr<A> make_unique(X&& arg) {
  return unique_ptr<A>(new A(std::forward<X>(arg)));
}

X&& forward(remove_reference<X>::type& a) {
  return static_cast<X&&>(a);
}
```

После склейки ссылок:

```
unique_ptr<A> make_unique(X&& arg) {
  return unique_ptr<A>(new A(std::forward<X>(arg)));
}

X&& forward(X& a) {
  return static_cast<X&&>(a);
}
```

Variadic templates + perfect forwarding

Можно применить `std::forward` для списка параметров.

```
template<typename T, typename ...Args>
std::unique_ptr<T> make_unique(Args&&... args) {
    return std::unique_ptr<T>(
                new T(std::forward<Args>(args)...));
}
```

Теперь `make_unique` работает для произвольного числа аргументов.

```
auto p = make_unique<Array<string>>(10, string("Hello"));
```

Программирование на языке C++
Лекция 10

Обработка ошибок

Александр Смаль

Логические ошибки и исключительные ситуации

- **Логические ошибки**.

  Ошибки в логике работы программы, которые происходят из-за неправильно написанного кода, т.е. это ошибки программиста:

  - выход за границу массива,
  - попытка деления на ноль,
  - обращение по нулевому указателю,
  - …

Логические ошибки и исключительные ситуации

- **Логические ошибки**.

  Ошибки в логике работы программы, которые происходят из-за неправильно написанного кода, т.е. это ошибки программиста:

  - выход за границу массива,
  - попытка деления на ноль,
  - обращение по нулевому указателю,
  - …

- **Исключительные ситуации**.

  Ситуации, которые требуют особой обработки.

  Возникновение таких ситуаций — это „нормальное“ поведение программы.

  - ошибка записи на диск,
  - недоступность сервера,
  - неправильный формат файла,
  - …

Выявление логических ошибок на этапе разработки

- Оператор `static_assert`.

```
#include<type_traits>



template<class T>
void countdown(T start) {
    static_assert(std::is_integral<T>::value
                  && std::is_signed<T>::value,
                     "Requires signed integral type");

    while (start >= 0) {
        std::cout << start-- << std::endl;
    }
}
```

Выявление логических ошибок на этапе разработки

- Оператор `static_assert`.
- Макрос `assert`.

```
#include<type_traits>
//#define NDEBUG
#include <cassert>

template<class T>
void countdown(T start) {
    static_assert(std::is_integral<T>::value
                  && std::is_signed<T>::value,
                     "Requires signed integral type");
    assert(start >= 0);
    while (start >= 0) {
        std::cout << start-- << std::endl;
    }
}
```

Способы сообщения об ошибке

```
size_t write(string file, string data);
```

Способы сообщения об ошибке

```
size_t write(string file, string data);
```

- Возврат статуса операции:

```
bool write(string file, string data, size_t & bytes);
```

Способы сообщения об ошибке

```
size_t write(string file, string data);
```

- Возврат статуса операции:

```
bool write(string file, string data, size_t & bytes);
```

- Возврат кода ошибки:

```
int const OK = 0, IO_WRITE_FAIL = 1, IO_OPEN_FAIL = 2;
int write(string file, string data, size_t & bytes);
```

Способы сообщения об ошибке

```
size_t write(string file, string data);
```

- Возврат статуса операции:

```
bool write(string file, string data, size_t & bytes);
```

- Возврат кода ошибки:

```
int const OK = 0, IO_WRITE_FAIL = 1, IO_OPEN_FAIL = 2;
int write(string file, string data, size_t & bytes);
```

- Глобальная переменная для кода ошибки:

```
size_t write(string file, string data);
```

```
size_t bytes = write(f, data);
if (errno) {
    cerr << strerror(errno);
    errno = 0;
}
```

Способы сообщения об ошибке

```
size_t write(string file, string data);
```

- Возврат статуса операции:

```
bool write(string file, string data, size_t & bytes);
```

- Возврат кода ошибки:

```
int const OK = 0, IO_WRITE_FAIL = 1, IO_OPEN_FAIL = 2;
int write(string file, string data, size_t & bytes);
```

- Глобальная переменная для кода ошибки:

```
size_t write(string file, string data);
```

```
size_t bytes = write(f, data);
if (errno) {
    cerr << strerror(errno);
    errno = 0;
}
```

- Исключения.

Исключения

```
size_t write(string file, string data) {
    if (!open(file)) throw FileOpenError(file);
    //...
}
double safediv(int x, int y) {
    if (y == 0) throw MathError("Division by zero");
    return double(x) / y;
}
void write_x_div_y(string file, int x, int y) {
    try {
        write(file, to_string(safediv(x, y)));
    } catch (MathError & s) {
        // обработка ошибки в safediv
    } catch (FileError & e) {
        // обработка ошибки в write
    } catch (...) {
        // все остальные ошибки
    }
}
```

Stack unwinding

При возникновении исключения объекты на стеке уничтожаются в естественном (обратном) порядке.

```
void foo() {
    D d;
    E e(d);
    if (!e) throw F();
    G g(e);
}
void bar() {
    A a;
    try {
        B b;
        foo();
        C c;
    } catch (F & f) {
        // обработка и пересылка
        throw f;
    }
}
```

Почему не стоит бросать встроенные типы

```
int foo() {
    if (...) throw -1;
    if (...) throw 3.1415;
}
void bar(int a) {
    if (a == 0) throw string("Not my fault!");
}
int main () {
    try { bar(foo());
    } catch (string & s) {
        // только текст
    } catch (int a) {
        // мало информации
    } catch (double d) {
        // мало информации
    } catch (...) {
        // нет информации
    }
}
```

Стандартные классы исключений

Базовый класс для всех исключений (в <exception>):

```
struct exception {
  virtual ~exception();
  virtual const char* what() const;
};
```

Стандартные классы ошибок (в <stdexcept>):

- logic_error: domain_error, invalid_argument, length_error, out_of_range
- runtime_error: range_error, overflow_error, underflow_error

```
int main() {
    try { ... }
    catch (std::exception const& e) {
       std::cerr << e.what() << '\n';
    }
}
```

Исключения в стандартной библиотеке

- Метод at контейнеров array, vector, deque, basic_string, bitset, map, unordered_map бросает out_of_range.
- Оператор new T бросает bad_alloc.
  Оператор new (std::nothrow) T в возвращает 0.
- Оператор typeid от разыменованного нулевого указателя бросает bad_typeid.
- Потоки ввода-вывода.

```
std::ifstream file;
file.exceptions( std::ifstream::failbit
                 | std::ifstream::badbit );
try {
    file.open ("test.txt");
    cout << file.get() << endl;
    file.close();
}
catch (std::ifstream::failure const& e) {
    cerr << e.what() << endl;
}
```

Как обрабатывать ошибки?

Есть несколько „правил хорошего тона“.

- Разделяйте „ошибки программиста“ и „исключительные ситуации“.
- Используйте `assert` и `static_assert` для выявления ошибок на этапе разработки.
- В пределах одной логической части кода обрабатывайте ошибки централизованно и однообразно.
- Обрабатывайте ошибки там, где их можно обработать.
- Если в данном месте ошибку не обработать, то пересылайте её выше при помощи исключения.
- Бросайте только стандартные классы исключений или производные от них.
- Бросайте исключения по значению, а отлавливайте по ссылке.
- Отлавливайте все исключения в точке входа.

Программирование на языке C++
Лекция 10

Исключения в деструкторах и конструкторах

Александр Смаль

Исключения в деструкторах

Исключения не должны покидать деструкторы.

- Двойное исключение:

```
void foo() {
    try {
        Bad b; // исключение в деструкторе
        bar(); // исключение
    } catch (std::exception & e) {
        // ...
    }
}
```

- Неопределённое поведение:

```
void bar() {
    Bad * bad = new Bad[100];
    // ислючение в деструкторе №20
    delete [] bad;
}
```

Исключения в конструкторе

Исключения — это единственный способ прервать конструирование объекта и сообщить об ошибке.

```
struct Database {
    explicit Database(string const& uri) {
        if (!connect(uri))
            throw ConnectionError(uri);
    }
    ~Database() { disconnect(); }
    // ...
};
int main() {
    try {
        Database * db = new Database("db.local");
        db->dump("db-local-dump.sql");
        delete db;
    } catch (std::exception const& e) {
        std::cerr << e.what() << '\n';
    }
}
```

Исключения в списке инициализации

Позволяет отловить исключения при создании полей класса.

```
struct System
{
    System(string const& uri, string const& data)
    try : db_(uri), dh_(data)
    {
        // тело конструктора
    }
    catch (std::exception & e) {
        log("System constructor: ", e);
        throw;
    }

    Database    db_;
    DataHolder  dh_;
};
```

Программирование на языке C++
Лекция 10

Спецификация исключений

Александр Смаль

Спецификация исключений

Устаревшая возможность C++, позволяющая указать список исключений, которые могут быть выброшены из функции.

```
void foo() throw(std::logic_error) {
    if (...) throw std::logic_error();
    if (...) throw std::runtime_error();
}
```

Если сработает второй `if`, то программа аварийно завершится.

```
void foo() {
    try {
        if (...) throw std::logic_error();
        if (...) throw std::runtime_error();
    } catch (std::logic_error & e) {
        throw e;
    } catch (...) {
        terminate();
    }
}
```

Ключевое слово `noexcept`

- Используется в двух значениях:
  - Спецификатор функции, которая не бросает исключение.
  - Оператор, проверяющий во время компиляции, что выражение специфицированно как небросающее исключение.
- Если функцию со спецификацией `noexcept` покинет исключение, то стек не обязательно будет свёрнут, перед тем как программа завершится.
  В отличие от аналогичной ситуации с `throw()`.
- Использование спецификации `noexcept` позволяет компилятору лучше оптимизировать код, т.к. не нужно заботиться о сворачивании стека.

Использование noexcept

```
void no_throw() noexcept;
void may_throw();

// копирующий конструктор noexcept
struct NoThrow { int m[100] = {}; };

// копирующий конструктор noexcept(false)
struct MayThrow { std::vector<int> v; };
```

```
MayThrow mt;
NoThrow  nt;

bool a = noexcept(may_throw());  // false
bool b = noexcept(no_throw());   // true

bool c = noexcept(MayThrow(mt)); // false
bool d = noexcept(NoThrow(nt));  // true
```

Условный noexcept

В спецификации noexcept можно использовать условные выражения времени компиляции.

```
template <class T, size_t N>
void swap(T (&a)[N], T (&b)[N])
        noexcept(noexcept(swap(*a, *b)));

template <class T1, class T2>
struct pair {
    void swap(pair & p)
        noexcept(noexcept(swap(first, p.first)) &&
                 noexcept(swap(second, p.second)))
    {
        swap(first, p.first);
        swap(second, p.second);
    }

    T1 first;
    T2 second;
};
```

Зависимость от noexcept

Проверка `noexcept` используется в стандартной библиотеке для обеспечения строгой гарантии безопасности исключений с помощью `std::move_if_noexcept` (например, `vector::push_back`).

```
struct Bad {
    Bad() {}
    Bad(Bad&&);       // может бросить
    Bad(const Bad&); // не важно
};
struct Good {
    Good() {}
    Good(Good&&) noexcept; // не бросает
    Good(const Good&) ;    // не важно
};
```

```
Good g1;
Bad  b1;
Good g2 = std::move_if_noexcept(g1); // move
Bad  b2 = std::move_if_noexcept(b1); // copy
```

Программирование на языке C++
Лекция 10

Гарантии безопасности исключений

Александр Смаль

Гарантии безопасности исключений

Гарантия отсутствия исключений

“Ни при каких обстоятельствах функция не будет генерировать исключения”.

Гарантии безопасности исключений

Гарантия отсутствия исключений

“Ни при каких обстоятельствах функция
не будет генерировать исключения”.

Базовая гарантия

“При возникновении любого исключения
состояние программы останется согласованным”.

Гарантии безопасности исключений

Гарантия отсутствия исключений

“Ни при каких обстоятельствах функция
не будет генерировать исключения”.

Базовая гарантия

“При возникновении любого исключения
состояние программы останется согласованным”.

Строгая гарантия

“Если при выполнении операции возникнет исключение,
то программа останется том же в состоянии,
которое было до начала выполнения операции”.

Строгая гарантия безопасности исключений

Строгая гарантия безопасности исключений

- В каком случае мы не можем обеспечить строгую гарантию безопасности исключений?

Строгая гарантия безопасности исключений

- В каком случае мы не можем обеспечить строгую гарантию безопасности исключений?

  При наличии взаимодействия со внешним окружением.

Строгая гарантия безопасности исключений

- В каком случае мы не можем обеспечить строгую гарантию безопасности исключений?

  При наличии взаимодействия со внешним окружением.

- Как обеспечить строгую гарантию безопасности исключений в остальных случаях?

Строгая гарантия безопасности исключений

- В каком случае мы не можем обеспечить строгую гарантию безопасности исключений?

  При наличии взаимодействия со внешним окружением.

- Как обеспечить строгую гарантию безопасности исключений в остальных случаях?

  Выполнять операцию над копией состояния программы.
  Если операция прошла успешно, заменить состояние на копию.

Строгая гарантия безопасности исключений

- В каком случае мы не можем обеспечить строгую гарантию безопасности исключений?

  При наличии взаимодействия со внешним окружением.

- Как обеспечить строгую гарантию безопасности исключений в остальных случаях?

  Выполнять операцию над копией состояния программы.
  Если операция прошла успешно, заменить состояние на копию.

- Когда можно обеспечить строгую гарантию *эффективно*?

Строгая гарантия безопасности исключений

- В каком случае мы не можем обеспечить строгую гарантию безопасности исключений?

  При наличии взаимодействия со внешним окружением.

- Как обеспечить строгую гарантию безопасности исключений в остальных случаях?

  Выполнять операцию над копией состояния программы.
  Если операция прошла успешно, заменить состояние на копию.

- Когда можно обеспечить строгую гарантию *эффективно*?

  Это вопрос архитектуры приложения.

Как добиться строгой гарантии?

```
template<class T>
struct Array
{
    void resize(size_t n)
    {
        T * ndata = new T[n];
        for (size_t i = 0; i != n && i != size_; ++i)
            ndata[i] = data_[i];

        delete [] data_;
        data_ = ndata;
        size_ = n;
    }

    T *     data_;
    size_t  size_;
};
```

Как добиться строгой гарантии: вручную

```
template<class T>
struct Array
{
    void resize(size_t n) {
        T * ndata = new T[n];
        try {
            for (size_t i = 0; i != n && i != size_; ++i)
                ndata[i] = data_[i];
        } catch (...) {
            delete [] ndata;
            throw;
        }
        delete [] data_;
        data_ = ndata;
        size_ = n;
    }

    T *     data_;
    size_t  size_;
};
```

Как добиться строгой гарантии: RAII

```
template<class T>
struct Array
{
    void resize(size_t n) {
        unique_ptr<T[]> ndata(new T[n]);

        for (size_t i = 0; i != n && i != size_; ++i)
            ndata[i] = data_[i];

        data_ = std::move(ndata);
        size_ = n;
    }

    unique_ptr<T[]> data_;
    size_t          size_;
};
```

Как добиться строгой гарантии: swap

```
template<class T>
struct Array
{
    void resize(size_t n) {
        Array t(n);
        for (size_t i = 0; i != n && i != size_; ++i)
            t[i] = data_[i];

        t.swap(*this);
    }

    T       * data_;
    size_t    size_;
};
```

Проектирование с учётом исключений

Рассмотрим традиционный интерфейс стека:

```
template<class T>
struct Stack
{
    void push(T const& t)
    {
        data_.push_back(t);
    }

    T pop()
    {
        T tmp = data_.back();
        data_.pop_back();
        return tmp;
    }

    std::vector<T> data_;
};
```

Проектирование с учётом исключений

Рассмотрим традиционный интерфейс стека:

```
template<class T>
struct Stack
{
    void push(T const& t)
    {
        data_.push_back(t);
    }

    void pop(T & res)
    {
        res = data_.back();
        data_.pop_back();
    }

    std::vector<T> data_;
};
```

Использование unique_ptr

```
template<class T>
struct Stack
{
    void push(T const& t)
    {
        data_.push_back(t);
    }

    unique_ptr<T> pop()
    {
%        unique_ptr<T> tmp(new T(data_.back()));
        data_.pop_back();
        return std::move(tmp);
    }

    std::vector<T> data_;
};
```

Заключение

- Проектируйте архитектуру приложения с учётом ислючений.

Заключение

- Проектируйте архитектуру приложения с учётом ислючений.
- Функции, не бросающие исключения, нужно объявлять как `noexcept`.

Заключение

- Проектируйте архитектуру приложения с учётом ислючений.
- Функции, не бросающие исключения, нужно объявлять как `noexcept`.
- Все использующие исключения функции должны обеспечивать как минимум базовую гарантию безопасности исключений.

Заключение

- Проектируйте архитектуру приложения с учётом ислючений.
- Функции, не бросающие исключения, нужно объявлять как `noexcept`.
- Все использующие исключения функции должны обеспечивать как минимум базовую гарантию безопасности исключений.
- Там, где это возможно, старайтесь обеспечить строгую гарантию безопасности исключений.

Заключение

- Проектируйте архитектуру приложения с учётом ислючений.
- Функции, не бросающие исключения, нужно объявлять как `noexcept`.
- Все использующие исключения функции должны обеспечивать как минимум базовую гарантию безопасности исключений.
- Там, где это возможно, старайтесь обеспечить строгую гарантию безопасности исключений.
- Используйте `swap`, умные указатели и другие RAII объекты для обеспечения строгой безопасности исключений.

Программирование на языке C++
Лекция 11

Многопоточное программирование

Александр Смаль

Асинхронное выполнение

Предположим, что мы хотим вычислить doAsyncWork асинхронно.

```
int doAsyncWork();
```

В C++ есть два способа выполнения задач асинхронно:

- создать поток вручную std::thread,

```
#include <thread>
// создание потока, вычисляющего doAsyncWork()
std::thread t(doAsyncWork);
```

- использование std::async.

```
#include <future>
// использование std::async
std::future<int> fut = std::async(doAsyncWork);
int res = fut.get();
```

std::async может в некоторых случаях (зависит от планировщика) отложить выполнение задачи до вызова get или wait.

std::async

- Имеет две стратегии выполнения: асинхронное выполнение и отложенное (синхронное) выполнение.
  1. `std::launch::async`
  2. `std::launch::deferred`
- По умолчанию имеет стратегию:
  `std::launch::async | std::launch::deferred`

```
// гарантирует асинхронное выполнение
std::future<int> fut =
    std::async(std::launch::async, doAsyncWork);
int res = fut.get();
```

- Отложенная задача может никогда не выполниться, если не будет вызвано `get` или `wait`.
- Возвращает `std::future<T>`, который позволяет получить возвращаемое значение.
- Позволяет обрабатывать исключения.

std::thread

- Сразу же начинает вычислять переданную функцию.
- Игнорирует возвращаемое значение функции.

```
// переменная для возвращаемого значения
int res = 0;
std::thread t([&res](){res = doAsyncWork();});
t.join();
```

- Метод `join()` позволяет заблокировать текущий поток, пока выполнение потока не завершится.
- Метод `detach()` позволяет отключить поток от объекта, т.е. разорвать связь между объектом и потоком.
- При вызове деструктора подключаемого потока программа завершается, т.е. необходимо вызвать `join` или `detach`.
- Исключения не могут покидать пределы потока.
- `native_handle()` возвращает дескриптор потока.

Синхронизация

```
double shared = 0;          // разделяемая переменная
std::mutex mtx;             // мьютекс для shared

void compute(int begin, int end)  {
    for (int i = begin; i != end; ++i) {
        double current = someFunction(i);
        // критическая секция
        std::lock_guard<std::mutex> lck(mtx);
        shared += current;
    }
}

int main () {
  std::thread th1 (compute, 0,   100);
  std::thread th2 (compute, 100, 200);
  th1.join();
  th2.join();

  std::cout << shared << std::endl;
}
```

std::atomic

- Шаблон `std::atomic` позволяет определить переменную, операции с которой будут атомарны.
- Определён только для целочисленных встроенных типов и указателей.

```
template<class T>
struct shared_ptr_data
{
    void addref()
    {
        ++counter; // atomic increment
    }

    T * ptr;
    std::atomic<size_t> counter;
};
```

Общие советы и замечания

Общие советы и замечания

- Предпочитайте `std::async` прямому созданию потоков.

Общие советы и замечания

- Предпочитайте `std::async` прямому созданию потоков.
- Гарантируйте неподключённость потоков на всех путях выполнения (в т.ч. при возникновении исключений).

Общие советы и замечания

- Предпочитайте `std::async` прямому созданию потоков.
- Гарантируйте неподключённость потоков на всех путях выполнения (в т.ч. при возникновении исключений).
- При использовании `std::thread` следите за тем, чтобы исключения не покидали функцию потока.

Общие советы и замечания

- Предпочитайте `std::async` прямому созданию потоков.
- Гарантируйте неподключённость потоков на всех путях выполнения (в т.ч. при возникновении исключений).
- При использовании `std::thread` следите за тем, чтобы исключения не покидали функцию потока.
- Используйте `std::atomic` вместо мьютекса, когда синхронизация нужна только для одной целочисленной переменной.

Общие советы и замечания

- Предпочитайте `std::async` прямому созданию потоков.
- Гарантируйте неподключённость потоков на всех путях выполнения (в т.ч. при возникновении исключений).
- При использовании `std::thread` следите за тем, чтобы исключения не покидали функцию потока.
- Используйте `std::atomic` вместо мьютекса, когда синхронизация нужна только для одной целочисленной переменной.
- Делайте константные методы безопасными в смысле потоков (например, при кешировании).

Общие советы и замечания

- Предпочитайте `std::async` прямому созданию потоков.
- Гарантируйте неподключённость потоков на всех путях выполнения (в т.ч. при возникновении исключений).
- При использовании `std::thread` следите за тем, чтобы исключения не покидали функцию потока.
- Используйте `std::atomic` вместо мьютекса, когда синхронизация нужна только для одной целочисленной переменной.
- Делайте константные методы безопасными в смысле потоков (например, при кешировании).
- `volatile` — это не про многопоточность.

Программирование на языке C++
Лекция 11

Коллекция библиотек Boost

Александр Смаль

Boost

- Это коллекция библиотек, расширяющих функциональность C++.

Boost

- Это коллекция библиотек, расширяющих функциональность C++.
- Свободно распространяются по лицензии Boost Software License вместе с исходным кодом и документацией на `www.boost.org`.

Boost

- Это коллекция библиотек, расширяющих функциональность C++.
- Свободно распространяются по лицензии Boost Software License вместе с исходным кодом и документацией на `www.boost.org`.
- Лицензия позволяет использовать boost в коммерческих проектах.

Boost

- Это коллекция библиотек, расширяющих функциональность C++.
- Свободно распространяются по лицензии Boost Software License вместе с исходным кодом и документацией на `www.boost.org`.
- Лицензия позволяет использовать boost в коммерческих проектах.
- Библиотеки из boost являются кандидатами на включение в стандарт C++.

Boost

- Это коллекция библиотек, расширяющих функциональность C++.
- Свободно распространяются по лицензии Boost Software License вместе с исходным кодом и документацией на `www.boost.org`.
- Лицензия позволяет использовать boost в коммерческих проектах.
- Библиотеки из boost являются кандидатами на включение в стандарт C++.
- Некоторые библиотеки boost были включены в стандарты C++ 2011/14 года (`std::function`, `std::thread`, ...).

Boost

- Это коллекция библиотек, расширяющих функциональность C++.
- Свободно распространяются по лицензии Boost Software License вместе с исходным кодом и документацией на `www.boost.org`.
- Лицензия позволяет использовать boost в коммерческих проектах.
- Библиотеки из boost являются кандидатами на включение в стандарт C++.
- Некоторые библиотеки boost были включены в стандарты C++ 2011/14 года (`std::function`, `std::thread`, ...).
- При включении библиотеки в boost она проходит несколько этапов рецензирования.

Boost

- Это коллекция библиотек, расширяющих функциональность C++.
- Свободно распространяются по лицензии Boost Software License вместе с исходным кодом и документацией на `www.boost.org`.
- Лицензия позволяет использовать boost в коммерческих проектах.
- Библиотеки из boost являются кандидатами на включение в стандарт C++.
- Некоторые библиотеки boost были включены в стандарты C++ 2011/14 года (`std::function`, `std::thread`, ...).
- При включении библиотеки в boost она проходит несколько этапов рецензирования.
- Библиотеки boost позволяют обеспечить переносимость.

Boost

- Это коллекция библиотек, расширяющих функциональность C++.
- Свободно распространяются по лицензии Boost Software License вместе с исходным кодом и документацией на `www.boost.org`.
- Лицензия позволяет использовать boost в коммерческих проектах.
- Библиотеки из boost являются кандидатами на включение в стандарт C++.
- Некоторые библиотеки boost были включены в стандарты C++ 2011/14 года (`std::function`, `std::thread`, ...).
- При включении библиотеки в boost она проходит несколько этапов рецензирования.
- Библиотеки boost позволяют обеспечить переносимость.
- В текущей версии в boost более сотни библиотек.

Категории библиотек Boost

- String and text processing
- Containers,
- Iterators
- Algorithms
- Function objects and higher-order programming
- Generic Programming
- Template Metaprogramming
- Concurrent Programming
- Math and numerics
- Correctness and testing
- Data structures
- Domain Specific
- System
- Input/Output
- Memory
- Image processing
- Inter-language support
- Language Features Emulation
- Parsing
- Patterns and Idioms
- Programming Interfaces
- State Machines
- Broken compiler workarounds
- Preprocessor Metaprogramming

Программирование на языке C++
Лекция 11

Метапрограммирование: основы

Александр Смаль

Метапрограммирование

- Метапрограммированием называют создание программ, которые порождают другие программы.

Метапрограммирование

- Метапрограммированием называют создание программ, которые порождают другие программы.
- Шаблоны C++ можно рассматривать как функциональный язык для метапрограммирования.

Метапрограммирование

- Метапрограммированием называют создание программ, которые порождают другие программы.
- Шаблоны C++ можно рассматривать как функциональный язык для метапрограммирования.
- Метапрограммы C++ позволяют оперировать типами, шаблонами и значениями целочисленных типов.

Метапрограммирование

- Метапрограммированием называют создание программ, которые порождают другие программы.
- Шаблоны C++ можно рассматривать как функциональный язык для метапрограммирования.
- Метапрограммы C++ позволяют оперировать типами, шаблонами и значениями целочисленных типов.
- Метапрограммирование в C++ можно применять для широкого круга задач:
  - целочисленные compile-time вычисления,
  - compile-time проверка ошибок,
  - условная компиляция,
  - генеративное программирование,
  - …

Метапрограммирование

- Метапрограммированием называют создание программ, которые порождают другие программы.
- Шаблоны C++ можно рассматривать как функциональный язык для метапрограммирования.
- Метапрограммы C++ позволяют оперировать типами, шаблонами и значениями целочисленных типов.
- Метапрограммирование в C++ можно применять для широкого круга задач:
  - целочисленные compile-time вычисления,
  - compile-time проверка ошибок,
  - условная компиляция,
  - генеративное программирование,
  - …
- Для метапрограммирования существуют целые библиотеки, например, MPL из boost.

Метафункции

Метафункция — это шаблонный класс, который определяет имя типа `type` или целочисленную константу `value`.

- Аргументы метафункции — это аргументы шаблона.
- Возвращаемое значение — это `type` или `value`.

Метафункции могут возвращать типы:

```
template<typename T>
struct AddPointer
{
    using type = T *;
};
```

и значения целочисленных типов:

```
template<int N>
struct Square
{
    static int const value = N * N;
};
```

Вычисления в compile-time

```
template<int N>
struct Fact
{
    static int const
        value = N * Fact<N - 1>::value;
};

template<>
struct Fact<0>
{
    static int const value = 1;
};

int main()
{
    std::cout << Fact<10>::value << std::endl;
}
```

(Это вычисление можно реализовать через `constexpr`.)

Определение списка

Шаблоны позволяют определять алгебраические типы данных.

```
// определяем список
template <typename ... Types>
struct TypeList;

// специализация по умолчанию
template <typename H, typename... T>
struct TypeList<H, T...>
{
    using Head = H;
    using Tail = TypeList<T...>;
};

// специализация для пустого списка
template <>
struct TypeList<> { };
```

Длина списка

```
// вычисление длины списка
template<typename TL>
struct Length
{
    static int const value = 1 +
        Length<typename TL::Tail>::value;
};

template<>
struct Length<TypeList<>>
{
    static int const value = 0;
};

int main()
{
    using TL = TypeList<double, float, int, char>;
    std::cout << Length<TL>::value << std::endl;
    return 0;
}
```

Операции со списком

```
// добавление элемента в начало списка
template<typename H, typename TL>
struct Cons;

template<typename H, typename... Types>
struct Cons<H, TypeList<Types...>>
{
    using type = TypeList<H, Types...>;
};

// конкатенация списков
template<typename TL1, typename TL2>
struct Concat;

template<typename... Ts1, typename... Ts2>
struct Concat<TypeList<Ts1...>, TypeList<Ts2...>>
{
    using type = TypeList<Ts1..., Ts2...>;
};
```

Вывод списка

```
// вывод списка в поток os
template<typename TL>
void printTypeList(std::ostream & os)
{
    os << typeid(typename TL::Head).name() << '\n';
    printTypeList<typename TL::Tail>(os);
};

// вывод пустого списка
template<>
void printTypeList<TypeList<>>(std::ostream & os) {}

int main()
{
    using TL = TypeList<double, float, int, char>;
    printTypeList<TL>(std::cout);
    return 0;
}
```

Программирование на языке C++
Лекция 11

Метапрограммирование: генерация классов и проверка свойств

Александр Смаль

Генерация классов

```
struct A {
    void foo() {std::cout << "struct A\n";}
};
struct B {
    void foo() {std::cout << "struct B\n";}
};
struct C {
    void foo() {std::cout << "struct C\n";}
};

using Bases = TypeList<A, B, C>;
```

Генерация классов

```
struct A {
    void foo() {std::cout << "struct A\n";}
};
struct B {
    void foo() {std::cout << "struct B\n";}
};
struct C {
    void foo() {std::cout << "struct C\n";}
};

using Bases = TypeList<A, B, C>;
```

```
template<typename TL>
struct inherit;

template<typename... Types>
struct inherit<TypeList<Types...>> : Types... {};

struct D : inherit<Bases> { };
```

Генерация классов

```
struct D : inherit<Bases>
{
    void foo() { foo_impl<Bases>(); }

    template<typename L> void foo_impl();
};
template<typename L>
inline void D::foo_impl()
{
    // приводим this к указателю на базу из списка
    static_cast<typename L::Head *>(this)->foo();

    // рекурсивный вызов для хвоста списка
    foo_impl<typename L::Tail>();
}
template<>
inline void D::foo_impl<TypeList<>>()
{
}
```

Как определить наличие метода?

```
struct A {  void foo() { std::cout << "struct A\n"; } };
struct B {  }; // нет метода foo()
struct C {  void foo() { std::cout << "struct C\n"; } };
```

```
template<typename L>
inline void D::foo_impl()
{
    // приводим this к указателю на базу из списка
    static_cast<typename L::Head *>(this)->foo();

    // рекурсивный вызов для хвоста списка
    foo_impl<typename L::Tail>();
}
```

Как проверить наличие родственных связей?

```
typedef char YES;
struct NO { YES m[2]; };

template<class B, class D>
struct is_base_of
{
    static YES test(B * );
    static NO  test(...);

    static bool const value =
        sizeof(YES) == sizeof(test((D *)0));
};

template<class D>
struct is_base_of<D, D>
{
    static bool const value = false;
};
```

SFINAE

SFINAE = Substitution Failure Is Not An Error.
Ошибка при подстановке шаблонных параметров
не является сама по себе ошибкой.

```
// ожидает, что у типа T определён
// вложенный тип value_type
template<class T>
void foo(typename T::value_type * v);

// работает с любым типом
template<class T>
void foo(T t);
```

```
// при инстанциировании первой перегрузки
// происходит ошибка (у int нет value_type),
// но это не приводит к ошибке компиляции
foo<int>(0);
```

Используем SFINAE

```
template<class T>
struct is_foo_defined
{
    // обёртка, которая позволит проверить
    // наличие метода foo с заданой сигнатурой
    template<class Z, void (Z::*)() = &Z::foo>
    struct wrapper {};

    template<class C>
    static YES check(wrapper<C> * p);

    template<class C>
    static NO  check(...);

    static bool const value =
        sizeof(YES) == sizeof(check<T>(0));
};
```

Проверяем наличие метода

```
template<bool b>
struct Bool2Type          { using type = YES; };

template<>
struct Bool2Type<false> { using type = NO;  };
```

```
template<class L>
void foo_impl()
{
    using Head = typename L::Head;

    constexpr bool has_foo =
            is_foo_defined<Head>::value;

    using CALL =
        typename Bool2Type<has_foo>::type;

    call_foo<Head>(CALL());
    foo_impl<typename L::Tail>();
}
```

Проверяем наличие метода (продолжение)

```
struct D : inherit<Bases>
{
    // ... foo, foo_impl

    template<class Base>
    void call_foo(YES)
    {
        static_cast<Base *>(this)->foo();
    }

    template<class Base>
    void call_foo(NO) { }
};
```

std::enable_if

```
// <type_traits>
namespace std {
    template<bool B, class T = void>
    struct enable_if {};

    template<class T>
    struct enable_if<true, T> { using type = T; };
}
```

```
template<class T>
typename std::enable_if<
    std::is_integral<T>::value, T>::type
    div2(T t) { return t >> 1; }

template<class T>
typename std::enable_if<
    std::is_floating_point<T>::value, T>::type
    div2(T t) { return t / 2.0; }
```

std::enable_if

```
template<class T>
T div2(T t, typename std::enable_if<
        std::is_integral<T>::value, T>::type * = 0)
{ return t >> 1; }

template<class T, class E = typename std::enable_if<
    std::is_floating_point<T>::value, T>::type>
T div2(T t)
{ return t / 2.0; }
```

```
template<class T, class E = void>
class A;

template<class T>
class A<T, typename std::enable_if<
           std::is_integral<T>::value>::type>
{};
```

Программирование на языке C++
Лекция 9

Стандартная библиотека шаблонов

Александр Смаль

STL: введение

- STL = Standard Template Library.
- STL является частью стандартной библиотеки C++, описана в стандарте, но не упоминается там явно.
- Авторы: Александр Степанов, Дэвид Муссер и Менг Ли (сначала для HP, а потом для SGI).
- Основана на разработках для языка Ада.

STL: введение

- STL = Standard Template Library.
- STL является частью стандартной библиотеки C++, описана в стандарте, но не упоминается там явно.
- Авторы: Александр Степанов, Дэвид Муссер и Менг Ли (сначала для HP, а потом для SGI).
- Основана на разработках для языка Ада.

Основные составляющие

- контейнеры (хранение объектов в памяти),
- итераторы (доступ к элементам контейнера),
- адаптеры (обёртки над контейнерами),
- алгоритмы (для работы с последовательностями),
- функциональные объекты, функторы (обобщение функций).

Преимущества стандартной библиотеки

- стандартизированность,
- общедоступность,
- эффективность,
- общеизвестность,
- …

Преимущества стандартной библиотеки

- стандартизированность,
- общедоступность,
- эффективность,
- общеизвестность,
- …

Чего нет в стандартной библиотеке?

- сложных структур данных,
- сложных алгоритмов,
- работы с графикой/звуком,
- …

Программирование на языке C++
Лекция 9

Последовательные контейнеры STL

Александр Смаль

Общие сведения о контейнерах

Контейнеры библиотеки STL можно разделить на две категории:

- последовательные,
- ассоциативные.

Общие сведения о контейнерах

Контейнеры библиотеки STL можно разделить на две категории:

- последовательные,
- ассоциативные.

Общие вложенные типы

- `Container::value_type`
- `Container::iterator`
- `Container::const_iterator`

Общие сведения о контейнерах

Контейнеры библиотеки STL можно разделить на две категории:

- последовательные,
- ассоциативные.

Общие вложенные типы

- `Container::value_type`
- `Container::iterator`
- `Container::const_iterator`

Общие методы контейнеров

- Все „особенные методы“ и swap.
- `size`, `max_size`, `empty`, `clear`.
- `begin`, `end`, `cbegin`, `cend`.
- Операторы сравнения: `==`, `!=`, `>`, `>=`, `<`, `<=`.

Примечание: вся STL определена в пространстве имён `std`.

Шаблон `array`

Класс-обёртка над статическим массивом.

- `operator[]`, `at`,
- `back`, `front`.
- `fill`,
- `data`.

Позволяет работать с массивом как с контейнером.

```
#include <array>

std::array<std::string, 3> a = {"One", "Two", "Three"};
std::cout << a.size() << std::endl;
std::cout << a[1] << std::endl;

// ошибка времени выполнения
std::cout << a.at(3) << std::endl;
```

Общие методы остальных последовательных контейнеров

- Конструктор от двух итераторов.
- Конструктор от `count` и `defVal`.
- Конструктор от `std::initializer_list<T>`.
- Методы `back, front`.
- Методы `push_back, emplace_back`
- Методы `assign`.
- Методы `insert`.
- Методы `emplace`.
- Методы `erase` от одного и двух итераторов.

Шаблон vector

Динамический массив с автоматическим изменением размера при добавлении элементов.

- operator[], at,
- resize,
- capacity, reserve, shrink_to_fit,
- pop_back,
- data.

Позволяет работать со старым кодом.

```
#include <vector>

std::vector<std::string> v = {"One", "Two"};
v.reserve(100);
v.push_back("Three");
v.emplace_back("Four");
legacy_function(v.data(), v.size());
std::cout << v[2] << std::endl;
```

Шаблон deque

Контейнер с возможностью быстрой вставки и удаления элементов на обоих концах за $O(1)$. Реализован как список указателей на массивы фиксированного размера.

- `operator[]`, `at`,
- `resize`,
- `push_front`, `emplace_front`
- `pop_back`, `pop_front`,
- `shrink_to_fit`.

```
#include <deque>

std::deque<std::string> d = {"One", "Two"};
d.emplace_back("Three");
d.emplace_front("Zero");
std::cout << d[1] << std::endl;
```

Шаблон `list`

Двусвязный список. В любом месте контейнера вставка и удаление производятся за $O(1)$.

- `push_front`, `emplace_front`,
- `pop_back`, `pop_front`,
- `splice`,
- `merge`, `remove`, `remove_if`, `reverse`, `sort`, `unique`.

```
#include <list>

std::list<std::string> l = {"One", "Two"};
l.emplace_back("Three");
l.emplace_front("Zero");
std::cout << l.front() << std::endl;
```

Итерация по списку

У списка нет методов для доступа к элементам по индексу.
Можно использовать range-based for:

```
using std::string;
std::list<string> l = {"One", "Two", "Three"};
for (string & s : l)
    std::cout << s << std::endl;
```

Для более сложных операций нужно использовать *итераторы*.

```
std::list<string>::iterator i = l.begin();
for ( ; i != l.end(); ++i) {
    if (*i == "Two")
        break;
}
l.erase(i);
```

Итератор списка можно перемещать в обоих направлениях:

```
auto last = l.end();
--last; // последний элемент
```

Шаблон forward_list

Односвязный список. В любом месте контейнера вставка и удаление производятся за $O(1)$.

- `insert_after` и `emplace_after` вместо `insert` и `emplace`,
- `before_begin`, `cbefore_begin`,
- `push_front`, `emplace_front`, `pop_front`,
- `splice_after`,
- `merge`, `remove`, `remove_if`, `reverse`, `sort`, `unique`.

```
#include <forward_list>
using std::string;

std::forward_list<string> fl = {"One", "Two"};
fl.emplace_front("Zero");
fl.push_front("Minus one");
std::cout << fl.front() << std::endl;
```

Шаблон basic_string

Контейнер для хранения символьных последовательностей.

```
typedef basic_string<char>      string;
typedef basic_string<wchar_t>   wstring;
typedef basic_string<char16_t>  u16string;
typedef basic_string<char32_t>  u32string;
```

- Метод c_str() для совместимости со старым кодом,
- поддержка неявных преобразований со строками в стиле C,
- `operator[]`, `at`,
- `reserve`, `capacity`, `shrink_to_fit`,
- `append`, `operator+`, `operator+=`,
- `substr`, `replace`, `compare`,
- `find`, `rfind`, `find_first_of`, `find_first_not_of`, `find_last_of`, `find_last_not_of` (в терминах *индексов*)

Адаптеры и псевдоконтейнеры

Адаптеры:

- `stack` — реализация интерфейса стека.
- `queue` — реализация интерфейса очереди.
- `priority_queue` — очередь с приоритетом на куче.

Псевдо-контейнеры:

- `vector<bool>`
  - ненастоящий контейнер (не хранит `bool`-ы),
  - использует proxy-объекты.
- `bitset`
  Служит для хранения битовых масок.
  Похож на `vector<bool>`.
- `valarray`
  Шаблон служит для хранения числовых массивов и оптимизирован для достижения повышенной вычислительной производительности.

Ещё о vector

- Самый универсальный последовательный контейнер.
- Во многих случаях самый эффективный.
- Предпочитайте vector другим контейнерам.
- Интерфейс вектора построен на итераторах, а не на индексах.
- Итераторы вектора ведут себя как указатели.

Использование reserve и capacity:

```
std::vector<int> v;
v.reserve(N); // N — верхняя оценка на размер
...
if (v.capacity() == v.size()) // реаллокация
```

Сжатие и очистка в C++03:

```
std::vector<int> & v = getData();
// shrink_to_fit
std::vector<int>(v).swap(v);
// clear + shrink_to_fit
std::vector<int>().swap(v);
```

Программирование на языке C++
Лекция 9

Ассоциативные контейнеры

Александр Смаль

Общие сведения

Ассоциативные контейнеры делятся на две группы:

- *упорядоченные* (требуют отношение порядка),
- *неупорядоченные* (требуют хеш-функцию).

Общие методы

1. `find` по ключу,
2. `count` по ключу,
3. `erase` по ключу.

Шаблоны set и multiset

set хранит упорядоченное множество (как двоичное дерево поиска).
Операции добавления, удаления и поиска работают за $O(\log n)$.
Значения, которые хранятся в set, неизменяемые.

- lower_bound, upper_bound, equal_range.

```
#include <set>
```

```
std::set<int> primes = {2, 3, 5, 7, 11};
// дальнейшее заполнение
if (primes.find(173) != primes.end())
    std::cout << 173 << " is prime\n";

// std::pair<iterator, bool>
auto res = primes.insert(3);
```

В multiset хранится упорядоченное мультимножество.

```
std::multiset<int> fib = {0, 1, 1, 2, 3, 5, 8};
// iterator
auto res = fib.insert(13);
// pair<iterator, iterator>
auto eq = fib.equal_range(1);
```

Шаблоны map и multimap

Хранит упорядоченное отображение (как дерево поиска по ключу).
Операции добавления, удаления и поиска работают за $O(\log n)$.

```
    typedef std::pair<const Key, T> value_type;
```

- lower_bound, upper_bound, equal_range,
- operator[], at.

```
#include <map>
```

```
std::map<std::string, int> phonebook;
phonebook.emplace("Marge", 2128506);
phonebook.emplace("Lisa",  2128507);
phonebook.emplace("Bart",  2128507);
// std::map<string,int>::iterator
auto it = phonebook.find("Maggie");
if ( it != phonebook.end())
    std::cout << "Maggie: " << it->second << "\n";
```

```
std::multimap<std::string, int> phonebook;
phonebook.emplace("Homer", 2128506);
phonebook.emplace("Homer", 5552368);
```

Особые методы map: operator[] и at

```
auto it = phonebook.find("Marge");
if (it != phonebook.end())
     it->second = 5550123;
else
    phonebook.emplace("Marge", 5550123);
// или
phonebook["Marge"] = 5550123;
```

Метод operator[]:

1. работает только с неконстантным map,
2. требует наличие у T конструктора по умолчанию,
3. работает за $O(\log n)$ (не стоит использовать map как массив).

Метод at:

1. генерирует ошибку времени выполнения,
   если такой ключ отсутствует,
2. работает за $O(\log n)$.

Использование собственного компаратора

Отношение строгого порядка: $\neg(x \prec y) \wedge \neg(y \prec x) \Rightarrow x = y$

```
struct Person { string name; string surname; };

bool operator<(Person const& a, Person const& b) {
    return a.name < b.name ||
           (a.name == b.name && a.surname < b.surname);
}
// уникальны по сочетанию имя + фамилия
std::set<Person> s1;

struct PersonComp {
    bool operator()(Person const& a,
                    Person const& b) const {
        return a.surname < b.surname;
    }
};
// уникальны по фамилии
std::set<Person, PersonComp> s2;
```

Шаблоны unordered_set и unordered_multiset

unordered_set хранит множество как хеш-таблицу.
Операции добавления, удаления и поиска работают за $O(1)$ в среднем.
Значения, которые хранятся в unordered_set, неизменяемые.

- equal_range, reserve,
- методы для работы с хеш-таблицей.

```
#include <unordered_set>
```

```
unordered_set<int> primes = {2, 3, 5, 7, 11};
// дальнейшее заполнение
if (primes.find(173) != primes.end())
    std::cout << 173 << " is prime\n";

// std::pair<iterator, bool>
auto res = primes.insert(3);
```

В unordered_multiset хранится мультимножество.

```
unordered_multiset<int> fib = {0, 1, 1, 2, 3, 5, 8};
// iterator
auto res = fib.insert(13);
```

Шаблоны unordered_map и unordered_multimap

Хранит отображение как хеш-таблицу.
Операции добавления, удаления и поиска работают за $O(1)$ в среднем.

- equal_range, reserve, operator[], at,
- методы для работы с хеш-таблицей.

```
#include <unordered_map>
```

```
unordered_map<std::string, int> phonebook;
phonebook.emplace("Marge", 2128506);
phonebook.emplace("Lisa",  2128507);
phonebook.emplace("Bart",  2128507);
// unordered_map<string,int>::iterator
auto it = phonebook.find("Maggie");
if ( it != phonebook.end())
    std::cout << "Maggie: " << it->second << "\n";
```

```
unordered_multimap<std::string, int> phonebook;
phonebook.emplace("Homer", 2128506);
phonebook.emplace("Homer", 5552368);
```

Использование собственной хеш-функции

```
struct Person { string name; string surname; };

bool operator==(Person const& a, Person const& b) {
    return a.name    == b.name
        && a.surname == b.surname;
}

namespace std {
    template <> struct hash<Person> {
        size_t operator()(Person const& p) const {
            hash<string> h;
            return h(p.name) ^ h(p.surname);
        }
    };
}

// уникальны по сочетанию имя + фамилия
unordered_set<Person> s;
```

Программирование на языке C++
Лекция 9

Итераторы и умные указатели

Александр Смаль

Категории итераторов

Итератор — объект для доступа к элементам последовательности, синтаксически похожий на указатель.
Итераторы делятся на пять категорий.

- Random access iterator: `++`, `--`, арифметика, `<`, `>`, `<=`, `>=`. (`array`, `vector`, `deque`)
- Bidirectional iterator: `++`, `--`. (`list`, `set`, `map`)
- Forward iterator: ++. (`forward_list`, `unordered_set`, `unordered_map`)
- Input iterator: ++, read-only.
- Output iterator: ++, write-only.

Функции для работы с итераторами:

```
void   advance  (Iterator & it, size_t n);
size_t distance (Iterator f, Iterator l);
void   iter_swap(Iterator i, Iterator j);
```

iterator_traits

```
// заголовочный файл <iterator>
template <class Iterator>
struct iterator_traits {
   typedef difference_type   Iterator::difference_type;
   typedef value_type        Iterator::value_type;
   typedef pointer           Iterator::pointer;
   typedef reference         Iterator::reference;
   typedef iterator_category Iterator::iterator_category;
};

template <class T>
struct iterator_traits<T *> {
    typedef difference_type   ptrdiff_t;
    typedef value_type        T;
    typedef pointer           T*;
    typedef reference         T&;
    typedef iterator_category
            random_access_iterator_tag;
};
```

iterator_category

```
// <iterator>
struct random_access_iterator_tag {};
struct bidirectional_iterator_tag {};
struct forward_iterator_tag {};
struct input_iterator_tag {};
struct output_iterator_tag {};
```

```
template<class I>
void advance_(I & i, size_t n,
              random_access_iterator_tag)
{ i += n; }
template<class I>
void advance_(I & i, size_t n, ... ) {
   for (size_t k = 0; k != n; ++k, ++i );
}
template<class I>
void advance(I & i, size_t n) {
   advance_(i, n, typename
     iterator_traits<I>::iterator_category());
}
```

reverse_iterator

У некоторых контейнеров есть обратные итераторы:

```
list<int> l = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// list<int>::reverse_iterator
for(auto i = l.rbegin(); i != l.rend(); ++i)
    cout << *i << endl;
```

Конвертация итераторов:

```
list<int>::iterator i = l.begin();
advance(i, 5); // i указывает на 5
// ri указывает на 4
list<int>::reverse_iterator ri(i);
i = ri.base();
```

Есть возможность сделать обратный итератор из random access или bidirectional при помощи шаблона `reverse_iterator`.

```
// <iterator>
template <class Iterator>
class reverse_iterator {...};
```

Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

1. Удаление делает некорректным итератор на удалённый элемент в любом контейнере.

Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

1. Удаление делает некорректным итератор на удалённый элемент в любом контейнере.
2. В `vector` и `string` добавление потенциально инвалидирует все итераторы (может произойти выделение нового буфера), иначе инвалидируются только итераторы на все следующие элементы.

Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

1. Удаление делает некорректным итератор на удалённый элемент в любом контейнере.
2. В `vector` и `string` добавление потенциально инвалидирует все итераторы (может произойти выделение нового буфера), иначе инвалидируются только итераторы на все следующие элементы.
3. В `vector` и `string` удаление элемента инвалидирует итераторы на все следующие элементы.

Инвалидация итераторов

Некоторые операции над контейнерами делают существующие итераторы некорректными (*инвалидация* итераторов).

1. Удаление делает некорректным итератор на удалённый элемент в любом контейнере.
2. В `vector` и `string` добавление потенциально инвалидирует все итераторы (может произойти выделение нового буфера), иначе инвалидируются только итераторы на все следующие элементы.
3. В `vector` и `string` удаление элемента инвалидирует итераторы на все следующие элементы.
4. В `deque` удаление/добавление инвалидирует все итераторы, кроме случаев удаления/добавления первого или последнего элементов.

Advanced итераторы

Для пополнения контейнеров:
`back_inserter, front_inserter, inserter`.

```
// в классе Database
    template<class OutIt>
    void findByName(string name, OutIt out);
```

```
// размер заранее неизвестен
vector<Person> res;
Database::findByName("Rick", back_inserter(res));
```

Для работы с потоками:
`istream_iterator, ostream_iterator`.

```
ifstream file("input.txt");
vector<double> v((istream_iterator<double>(file)),
                 istream_iterator<double>());

copy(v.begin(), v.end(),
     ostream_iterator<double>(cout, "\n"));
```

Как написать свой итератор

```
// <iterator>
template
<class Category, // iterator::iterator_category
 class T,        // iterator::value_type
 class Distance = ptrdiff_t,// iterator::difference_type
 class Pointer = T*,        // iterator::pointer
 class Reference = T&       // iterator::reference
> class iterator;
```

```
#include <iterator>

struct PersonIterator
    : std::iterator<forward_iterator_tag, Person>
{
// operator++, operator*, ...
};
```

Умные указатели

`unique_ptr`

- Умный указатель с уникальным владением.
- Нельзя копировать, можно перемещать.
- Не подходит для разделяемых объектов.

`shared_ptr`

- Умный указатель с подсчётом ссылок.
- Универсальный указатель.

`weak_ptr`

- Умный указатель с для создания *слабых ссылок*.
- Работает вместе с `shared_ptr`.

Программирование на языке C++
Лекция 9

Алгоритмы

Александр Смаль

Функторы и min/max алгоритмы

- *Функтор* — класс, объекты которого ведут себя как функции, т.е. имеет перегруженные `operator()`.
- *Предикат* — функтор, возвращающий `bool`.

Функторы и min/max алгоритмы

- *Функтор* — класс, объекты которого ведут себя как функции, т.е. имеет перегруженные `operator()`.
- *Предикат* — функтор, возвращающий `bool`.

Функторы в стандартной библиотеке:

- `less`, `greater`, `less_equal`, `greater_equal`, `not_equal_to`, `equal_to`,
- `minus`, `plus`, `divides`, `modulus`, `multiplies`,
- `logical_not`, `logical_and`, `logical_or`
- `bit_and`, `bit_or`, `bit_xor`,
- `hash`.

Функторы и min/max алгоритмы

- *Функтор* — класс, объекты которого ведут себя как функции, т.е. имеет перегруженные `operator()`.
- *Предикат* — функтор, возвращающий `bool`.

Функторы в стандартной библиотеке:

- `less`, `greater`, `less_equal`, `greater_equal`, `not_equal_to`, `equal_to`,
- `minus`, `plus`, `divides`, `modulus`, `multiplies`,
- `logical_not`, `logical_and`, `logical_or`
- `bit_and`, `bit_or`, `bit_xor`,
- `hash`.

Алгоритмы min/max

- `min`, `max`, `minmax`,
- `min_element`, `max_element`, `minmax_element`.

Немодифицирующие алгоритмы

- `all_of`, `any_of`, `none_of`,
- `for_each`,
- `find`, `find_if`, `find_if_not`, `find_first_of`,
- `adjacent_find`,
- `count`, `count_if`,
- `equal`, `mismatch`,
- `is_permutation`,
- `lexicographical_compare`,
- `search`, `search_n`, `find_end`.

Для упорядоченных последовательностей

- `lower_bound`, `upper_bound`, `equal_range`,
- `set_intersection`, `set_difference`, `set_union`, `set_symmetric_difference`,
- `binary_search`, `includes`.

Примеры

```
vector<int> v = {2,3,5,7,13,17,19};
size_t c = count_if(v.begin(), v.end(),
                    [](int x) {return x % 2 == 0;});

auto it = lower_bound(v.begin(), v.end(), 11);

bool has7 = binary_search(v.begin(), v.end(), 7);
```

```
vector<string> & db = getNames();
for_each(db.begin(), db.begin() + db.size() / 2,
        [](string & s){cout << s << "\n";});

auto w = find(db.begin(), db.end(), "Waldo");

string agents[3] = {"Alice", "Bob", "Eve"};
auto it = find_first_of(db.begin(), db.end(),
                        agents, agents + 3);
```

Модифицирующие алгоритмы

- `fill`, `fill_n`, `generate`, `generate_n`,
- `random_shuffle`, `shuffle`,
- `copy`, `copy_n`, `copy_if`, `copy_backward`,
- `move`, `move_backward`,
- `remove`, `remove_if`, `remove_copy`, `remove_copy_if`,
- `replace`, `replace_if`, `replace_copy`, `replace_copy_if`,
- `reverse`, `reverse_copy`,
- `rotate`, `rotate_copy`,
- `swap_ranges`,
- `transform`,
- `unique`, `unique_copy`,
- * `accumulate`, `adjacent_difference`, `inner_product`, `partial_sum`, `iota`.

Примеры

```
// случайные
vector<int> a(100);
generate(a.begin(), a.end(), [](){return rand() % 100;});

// 0,1,2,3,...
vector<int> b(a.size());
iota(b.begin(), b.end(), 0);

// c[i] = a[i] * b[i]
vector<int> c(b.size());
transform(a.begin(), a.end(), b.begin(),
          c.begin(), multiplies<int>());

// c[i] *= 2
transform(c.begin(), c.end(), c.begin(),
          [](int x) {return x * 2;});

// сумма c[i]
int sum = accumulate(c.begin(), c.end(), 0);
```

Удаление элементов из последовательности

```
vector<int> v = {2,5,1,5,8,5,2,5,8};
remove(v.begin(), v.end(), 5);
```

Как изменится `v.size()`?

Какое содержимое вектора v?

Удаление элементов из последовательности

```
vector<int> v = {2,5,1,5,8,5,2,5,8};
remove(v.begin(), v.end(), 5);
```

Как изменится `v.size()`? Не изменится.

Какое содержимое вектора v? `{2,1,8,2,8,5,2,5,8}`

Удаление элементов из последовательности

```
vector<int> v = {2,5,1,5,8,5,2,5,8};
remove(v.begin(), v.end(), 5);
```

Как изменится `v.size()`? Не изменится.
Какое содержимое вектора v? `{2,1,8,2,8,5,2,5,8}`

Удаление элемента по значению:

```
vector<int> v = {2,5,1,5,8,5,2,5,8};
v.erase(remove(v.begin(), v.end(), 5), v.end());
```

```
list<int> l = {2,5,1,5,8,5,2,5,8};
l.remove(5);
```

Удаление элементов из последовательности

```
vector<int> v = {2,5,1,5,8,5,2,5,8};
remove(v.begin(), v.end(), 5);
```

Как изменится `v.size()`? Не изменится.
Какое содержимое вектора v? `{2,1,8,2,8,5,2,5,8}`

Удаление элемента по значению:

```
vector<int> v = {2,5,1,5,8,5,2,5,8};
v.erase(remove(v.begin(), v.end(), 5), v.end());
```

```
list<int> l = {2,5,1,5,8,5,2,5,8};
l.remove(5);
```

Удаление одинаковых элементов:

```
vector<int> v = {1,2,2,2,3,4,5,5,5,6,7,8,9};
v.erase(unique(v.begin(), v.end()), v.end());
```

```
list<int> l = {1,2,2,2,3,4,5,5,5,6,7,8,9};
l.unique();
```

Удаление из ассоциативных контейнеров

Неправильный вариант

```
map<string, int> m;
for (auto it = m.begin(); it != m.end(); ++it)
    if (it->second == 0)
        m.erase(it);
```

Удаление из ассоциативных контейнеров

Неправильный вариант

```
map<string, int> m;
for (auto it = m.begin(); it != m.end(); ++it)
    if (it->second == 0)
        m.erase(it);
```

Правильный вариант

```
for (auto it = m.begin() ; it != m.end(); )
    if (it->second == 0)
        it = m.erase(it);
    else
        ++it;
```

Удаление из ассоциативных контейнеров

Неправильный вариант

```
map<string, int> m;
for (auto it = m.begin(); it != m.end(); ++it)
    if (it->second == 0)
        m.erase(it);
```

Правильный вариант

```
for (auto it = m.begin() ; it != m.end(); )
    if (it->second == 0)
        it = m.erase(it);
    else
        ++it;
```

Альтернативный вариант (для старого стандарта)

```
for (map<string,int>::iterator it = m.begin();
                               it != m.end();)
    if (it->second == 0)
        m.erase(it++);
    else
        ++it;
```

Сортировка

- `is_sorted`, `is_sorted_until`,
- `sort`, `stable_sort`,
- `nth_element`, `partial_sort`,
- `merge`, `inplace_merge`,
- `partition`, `stable_partition`, `is_partitioned`, `partition_copy`, `partition_point`.

Сортировка

- is_sorted, is_sorted_until,
- sort, stable_sort,
- nth_element, partial_sort,
- merge, inplace_merge,
- partition, stable_partition, is_partitioned, partition_copy, partition_point.

```
vector<int> v = randomVector<int>();

auto med = v.begin() + v.size() / 2;
nth_element(v.begin(), med, v.end());
cout << "Median: " << *med;

auto m = partition(v.begin(), v.end(),
          [](int x){return x % 2 == 0;});
sort(v.begin(), m);
v.erase(m, v.end());
```

Что есть ещё?

- Операции с кучей:
  - `push_heap`,
  - `pop_heap`,
  - `make_heap`,
  - `sort_heap`
  - `is_heap`,
  - `is_heap_until`.
- Операции с неинициализированными интервалами:
  - `raw_storage_iterator`,
  - `uninitialized_copy`,
  - `uninitialized_fill`,
  - `uninitialized_fill_n`.
- Операции с перестановками
  - `next_permutation`,
  - `prev_permutation`.