C++

Fall 2020

compscicenter.ru

Башарин Егор, t.me/egorbasharin

Филипп Грабовой, t.me/phil_grab

Лекция VIII

Namespaces. Unions

Static members

Non-const static members

```
struct X {
  static int n = 13; // Error
};
```

Non-const static members

```
struct X {
  static int n = 13; // Error
};
```

- В теле класса только объявление статической переменной
- Что бы было, если бы там было разрешено определение?

Non-const static members

```cpp
// x.hpp
struct X {
  static int n; // declaration
};
```

```cpp
// x.cpp
#include "x.hpp"
int X::n = 12; // definition
```

```
$ clang++ -c x.cpp -o x.o
$ nm x.o
0000000000000000 D __ZN1X1nE
```

- n имеет внешнюю линковку (external linkage)

Non-const static members

В объявлении статических переменных можно использовать incomplete типы

```
// x.hpp
struct F; // Forward declaration
struct X {
    static F f; // incomplete type
    static X x; // incomplete type
};
```

```
// x.cpp
#include "x.hpp"
#include "f.hpp" // class F definition here

F X::f; // definition
X X::x; // definition
```

const static members

```
int make_j();

struct X {
  // integral/enum types are OK
  // with constexpr initializer
  static const int i = 1;

  static const int j = make_j(); // Error: not constexpr
  static const double d = 3.2;   // Error
};
```

constant expression

```
const double d1 = 10.0;
const double d2 = 10.0;
static_assert(d1 == d2); // Error

const int i1 = 10;
const int i2 = 10;
static_assert(i1 == i2); // OK
```

- Если переменная, инициализированная значением константного выражения, имеет тип, отличный от целочисленного или типа перечисления, то даже const квафилификатор не позволит использовать ее в константных выражениях

const static members

```
struct X {
  static const int i = 1;
};
```

- Значение `i` известно в compile-time, поэтому оно будет подставлено туда, где это возможно. Переменная по факту не будет создана.
- Невозможно подставить `i` в случае `odr-used`:
  - взятие адреса
  - использование ссылки
- no linkage

odr-used static const members

```
struct X {
  static const int i = 1;
};

void f(const int& ref);

int main()
{
    f(X::i); // Linkage error: symbols not found
}
```

odr-used static const members

```
struct X {
  static const int i = 10;
};

const int X::i; // definition without initializer

void f(const int& ref);

int main()
{
    f(X::i); // OK
}
```

inline (since C++17)

Определение допустимо прямо в теле класса

```
struct X {
  inline static double d = 10.0;
  inline static const float f = 10.0;
};
```

const reference member

Расширение времени жизни

Пример 1

```
struct T {
  T() { std::cout << "T"; }
  ~T() { std::cout << "~T"; }
};

int main()
{
    {
        const T& r = T();
        std::cout << "|";
    }
    std::cout << "0";
}
```

Output: T|~T0

Расширение времени жизни

Пример 2.1

```
struct T {
  T() { std::cout << "T"; }
  ~T() { std::cout << "~T"; }

  int i_;
  const int& i() const { return i_; }
};

int main()
{
    const int& ref1 = T().i(); // dangling reference
    std::cout << "x";
}
```

Расширение времени жизни

Пример 2.2

```
struct T {
  T() { std::cout << "T"; }
  ~T() { std::cout << "~T"; }

  int i_;
  const int& i() const { return i_; }
};

int main()
{
    const int& ref1 = T().i_; // OK
    std::cout << "x";
}
```

const reference member

```
struct A
{
  A() = default;
  A(double) : v(10) { } // error
  A(const int& v) : v(v) { }
  const int& v = 10;
};

int main()
{
    A a1; // error
    A a2(1.0); // error
    A a3(1); // dangling reference
}
```

member functions & inline

```
struct S {
  void f() { } // inline by default
};
```

member functions & inline

```
struct S {
  void f();
};

inline void S::f() {}
```

Namespaces

- Позволяют различать одинаковые имена
  - до неймспейсов — префиксы имен: `struct XML_Parser`, `int XML_Get…`
- Лучше структурируется код
- …

names

Имена в C++ — это обозначение конкретных сущностей:

- variables, constants
- functions
- structs, classes, enums, unions
- templates
- typedefs, usings
- namespaces

namespace + inner

```
namespace ns {
    int A = 42;

    namespace ns_inner {
        void foo(int) { ... }
    }
}

// использование
int B = ns::A;
ns::ns_inner::foo(ns::A);
```

namespace extension

```
namespace ns {
    namespace ns_inner {
        int A = 42;
    }

    namespace ns_inner_other {
        int A = 42;
    }

    namespace ns_inner {
        int B = 24;
    }
}
```

usage syntax

- `operator::` — ищет имя в соответствующем простанстве имен
  - `my_namespace::my_func`
  - `::func` — поиск в глобальном пространстве имен
- `std::string::npos` — классы и структуры задают свое пространство имен

алгоритм поиска имен

В процессе компиляции, когда нужно *разрешить* имя:

1. Если оно есть в текущем неймспейсе — остановиться и выдать все одноименные сущности
2. Если текущий неймспейс глобальный - выдать ошибку
3. Перейти к родительскому неймспейсу
4. Повторить сначала

example

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

namespace ns {
    int foo(float f) { return  2; }
    int foo(double a, double b) { return  3; }
    namespace ns_inner {
        int global = foo(5);
    }
}
```

- когда какое-либо имя найдено — остановка
- выбор функции из перегрузок — из найденных имен

Koenig lookup (ADL)

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

int main() {
    ns::Point a(1,2);
    ns::Point b(3,4);
    b = ns::operator+(a, b); // ок
    b = a + b;               // тоже ок
    return 0;
}
```

- для имен функций (⇒ и операторов)
- на первой фазе алгоритма поиска
- дополнительно рассматриваем пространства имен, *из которых аргументы*

using namespace <smth>

включает все имена из неймспейса в текущий

```
namespace my_global { namespace ns {
int A = 42;
namespace ns_inner {
    int foo(int i ) { return i; }
}
} // namespace ns

using namespace ns;
int B = A;                          // -> ns::A
int C = ns::ns_inner::foo(B);       // -> ns::ns_inner:foo

using namespace ns::ns_inner;
int D = foo(B);                     // -> ns::ns_inner:foo

} // namespace my_global
```

using <smth>

включает одно имя из неймспейса в текущий

```
namespace ns {
int A = 42;

namespace ns_inner {
int foo(int i ) { return i; }
}

} // namespace ns

using ns::A;
int B = A;                          // -> ns::A
int C = ns::ns_inner::foo(B);
```

anonymous namespace

Имя безымянного пространства имен уникально (генерируется компилятором)

```
namespace {
    int A = 42;
    /* ... */
}
```

```
// translated to:
namespace $uniq {
    int A = 42;
    /* ... */
}

using namespace $uniq;
```

- Зачастую заменяет static-переменные (в заголовнчых файлах)

`new` and `delete`

`new`-expression выполняет (верхнеуровнево):

1. memory allocation
2. object construction
3. address returning

`delete`-expression ответственен за:

1. object destruction
2. memory deallocation

notes

- `delete <ptrValue>;` — аргумент должен быть результатом соответствующего оператора `new`
- `delete[] <ptrValue>;` — не передаем size
- `new/delete` выражения используют соответствующие операторы для выделения памяти. Их можно переопределять, в т.ч. для каждого класса
  - ⇒ в коде возможны `::new`, `::delete`

overloading

Возможны два переопределения:

- global operator new/delete
- class member operator new/delete
  - static dy default: нет this

Зачем нужно:

1. Debugging/Logging
2. Perf (custom allocations, arenas)
3. Garbage collecting

class member new/delete

```
struct T {
    static void* operator new(size_t size) {
        std::cout << "My new operator with size: "
                  << size << std::endl;
        return ::operator new(size);
    }

    static void operator delete(void *p) {
        std::cout << "My delete operator" << std::endl;
        ::operator delete(p);
    }
};

int main() {
    T* t = new T;
    delete t;
}
```

Output:

```
My new operator with size: 1
My delete operator
```

global new/delete

```
void* operator new(size_t size) {
    std::cout << "My new operator with size: "
        << size << std::endl;
    return malloc(size);
}

void operator delete(void *p) noexcept {
    std::cout << "My delete operator" << std::endl;
    free(p);
}

struct T {};
int main() {
    T* t = new T;
    delete t;
}
```

Output:

```
My new operator with size: 1
My delete operator
```

placement new example

```
Foo* p = new Foo();
```

```
// by steps
Foo* p;
void* raw = operator new(sizeof(Foo));

try {
    p = new(raw) Foo();     // in-place contruction
} catch (...) {             // exception in ctor
    operator delete(raw);
    throw;                  // rethrow
}
```

placement new

- конструирует объект на уже подготовленном участке памяти
  - буффер должен быть корректным для размещения объекта, иначе UB
- нужен в редких случаях, примеры:
  - union (variant) — чтобы хранить *один из* объектов на одной памяти
  - vector (или Arena в общем случае) — когда выделение памяти и конструкция/деструкция разнесены по времени
    - size vs capacity

placement new example

```
struct Foo { };
struct Bar { int i; char c; };
const size_t MAX_SIZE
    = std::max(sizeof(Foo), sizeof(Bar));
void foobar() {
    alignas(Foo) alignas(Bar)
        unsigned char buf[MAX_SIZE];

    // Construct a `Foo` object in-place
    // at the address pointed by `buf`
    Foo* fooPtr = new(buf) Foo;
    // **Manually** calling the object's destructor (!)
    fooPtr->~Foo();

    Bar* barPtr = new(buf) Bar{1, 'a'};
    barPtr->~Bar();
}
```

Unions

- хранит только один из своих объектов в каждый момент времени
- в какой member последний раз записали — тот и можно читать (остальные — UB)
  - но некоторые компиляторы это специфицируют
- можно объявлять конструкторы (с [c++11])

Unions syntax

```
union S {
    std::string str;
    std::vector<int> vec;
    ~S() {};                    // ?
};

void example() {
    S s = {"Hello, world"};
    // use s.str...
    s.str.~basic_string();

    // switching active member
    new (&s.vec) std::vector<int>{10};
    s.vec.push_back(10);
    std::cout << s.vec.size() << std::endl;
    s.vec.~vector();
}
```

`std::variant`

Проблемы union:

- нужно "помнить", какой объект сейчас сохранен
- не забывать звать деструктор явно

Решение: используйте `std::variant` [c++17]