Ещё о нововведениях C++ 11/C++ 14

Александр Смаль

CS центр
27 февраля 2018
Санкт-Петербург

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
```

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`

Специально 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;
};
```

```
CStyleStruct    s1 = {19, 72.0};// инициализация по-старому
CPPStyleStruct  s2(19, 83.0);   // вызов конструктора по-старому
```

```
CStyleStruct    s1{19, 72.0}; // инициализация по-новому
CPPStyleStruct  s2{19, 83.0}; // вызов конструктора по-новому
```

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

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.)"
```

Семантика перемещения

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() {
    std::unique_ptr<Unit> p1(new Elf()); // захват указателя

    // 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
}
```

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)));
}
```

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"));
```