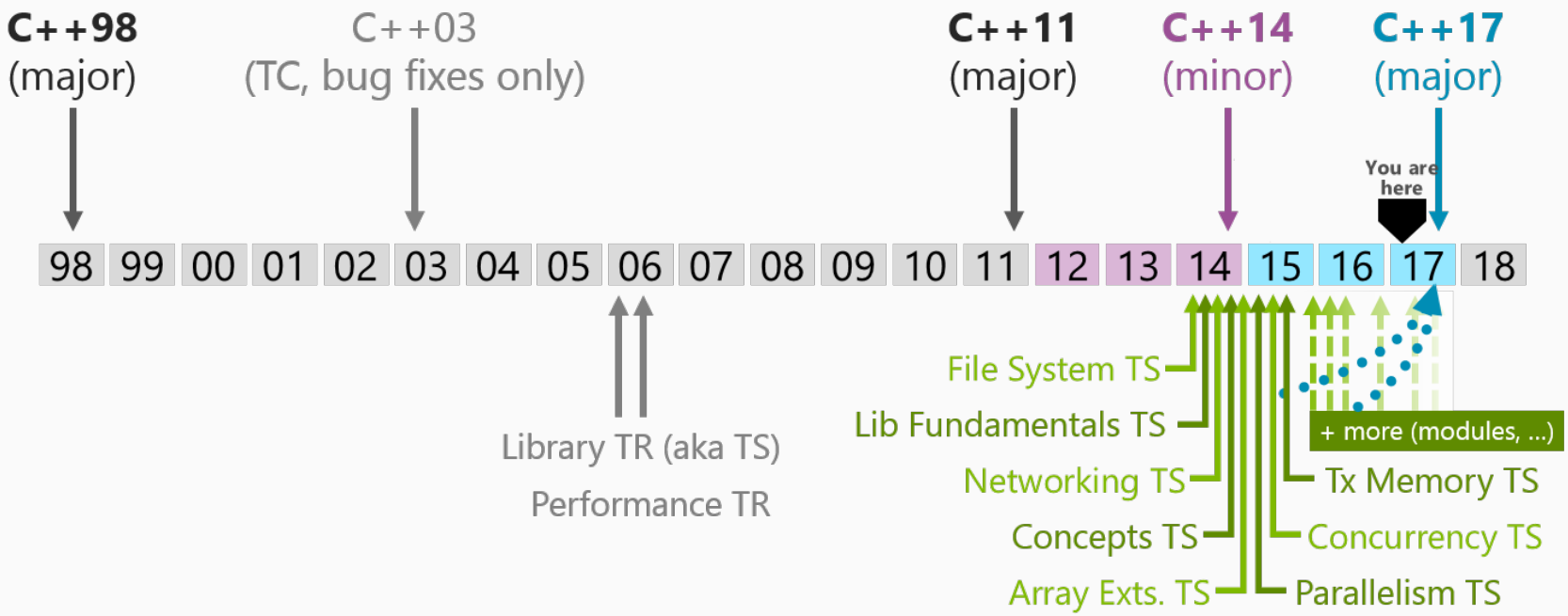
Стандарты C++ 11/14: вывод типов и move семантика

Александр Смаль

CS центр
29 марта 2017
Санкт-Петербург

Стандарты C++


C++98
(major)
C++03
(TC, bug fixes only)
C++11
(major)
C++14
(minor)
C++17
(major)
You are here
98 99 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18
Library TR (aka TS)
Performance TR
File System TS
Lib Fundamentals TS
Networking TS
Concepts TS
Array Exts. TS
Tx Memory TS
Concurrency TS
Parallelism TS
+ more (modules, ...)

Стандарт C++ 11/14

- поддержка стабильности и совместимость с C++98;
- предпочитается введение новых возможностей через стандартную библиотеку, а не через ядро языка;
- изменения, улучшающие технику программирования;
- совершенствовать C++ с точки зрения системного и библиотечного дизайна;
- увеличивать типобезопасность для обеспечения безопасной альтернативы для нынешних опасных подходов;
- увеличивать производительность и возможности работать напрямую с аппаратной частью;
- обеспечивать решение реальных, распространённых проблем;
- реализовать принцип «не платить за то, что не используешь»;
- сделать C++ проще для изучения без удаления возможностей, используемых программистами-экспертами.

Мелкие улучшения

1. Исправлена проблема с угловыми скобками: `T<U<int>>`.
2. Тип `std::nullptr_t` и литерал `nullptr`.
3. Перечисления со строгой типизацией:

```
enum class Enum1 { Val1, Val2, Val3 = 100, Val4 };
enum class Enum2 : unsigned int { Val1, Val2i };
```

4. Ключевое слово `explicit` для оператора приведения типа.
5. Шаблонный typedef

```
template<class First, class Second, int third> class SomeType;

template<typename Second>
using TypedefName = SomeType<OtherType, Second, 5>;

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

Мелкие улучшения ч.2

1. Добавлен `static_assert`

```
#include <type_traits>

template<class T>
void run(T * data, size_t n) {
    static_assert(std::is_integral<T>::value, "T isn't integral");
}
```

2. Добавлена возможность запрашивать и указывать выравнивание с помощью операторов `alignof` и `alignas`.

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

3. Методы по-умолчанию и удаление функций:

```
    A() = default; // только для специальных методов
    A & operator=(A const& a) = delete;
```

Удалять можно и свободные функции!

Вывод типов

```
for (std::vector<string>::const_iterator i  = v.cbegin();
                                         i != v.cend(); ++i){
    std::vector<string>::const_reverse_iterator j = i;
}
// C++11
for (auto i = myvec.cbegin(); i != myvec.cend(); ++i) {
    decltype(myvec.rbegin()) j = i;
}
    // примеры
    const std::vector<int> v(1);
    auto a = v[0];          // a - int
    decltype(v[0]) b = 1; // b - const int&
    auto c = 0;             // c - int
    auto d = c;             // d - int
    decltype(c) e;          // e - int
    decltype((c)) f = c;   // f - int&
    decltype(0) g;          // g - int
```

Альтернативный синтаксис функций

```
// RETURN_TYPE = ?
template <typename A, typename B>
    RETURN_TYPE Plus(const A &a, const B &b) {
    return a + b;
}
// некорректно
template <typename A, typename B>
    decltype(a + b) Plus(const A &a, const B &b) { return a + b; }
// корректно, но неудобно
template <typename A, typename B>
    decltype(std::declval<const A&>() + std::declval<const B&>())
        Plus(const A &a, const B &b) {
    return a + b;
}
// OK
template <typename A, typename B>
    auto Plus(const A &a, const B &b) -> decltype(a + b) {
    return a + b;
}
```

Шаблоны с переменным числом аргументов

```
void printf(const char *s) {
    while (*s) {
        if (*s == '%' && *(++s) != '%')
            throw std::runtime_error("invalid format");
        std::cout << *s++;
    }
}
template<typename T, typename... Args>
void printf(const char *s, T value, Args... args) {
    while (*s) {
        if (*s == '%' && *(++s) != '%') {
            std::cout << value;
            printf(++s, args...);
            return;
        }
        std::cout << *s++;
    }
    throw std::logic_error("extra arguments provided to printf");
}
```

Rvalue Reference/Move semantics

```
struct string {
    string (const string & s): size_(s.size()), data_(0) {...}

    string & operator = (const string & s) {...}

    string (string && s) : size_(0), data_(0) {
        s.swap(*this);
    }
    string & operator = (string && s) {
        s.swap(*this);
        return *this;
    }
};
    v.push_back(string("Hello, ")); // rvalue
    string s("world!");
    v.push_back(std::move(s));      // lvalue
```

Перемещающие особые методы

- Добавлены два особых метода перемещающий конструктор и перемещающий оператор присваивания (аналоги копирующих).
- В отличие от копирующих конструктора и оператора присваивания перемещающие методы генерируются только, если в классе нет пользовательских копирующих операций, перемещающих операций и пользовательского деструктора.
- Генерация копирующих методов для классов с пользовательским конструктором признана устаревшей.

Пример: unique_ptr

```
struct Foo {
    void bar() { std::cout << "Foo::bar\n";  }
};

void f(const Foo &) { std::cout << "f(const Foo&)\n"; }

int main() {
    std::unique_ptr<Foo> p1(new Foo);  // p1 owns Foo
    if (p1) p1->bar();

    std::unique_ptr<Foo> p2(std::move(p1));  // now p2 owns Foo
    f(*p2);

    p1 = std::move(p2);  // ownership returns to p1
    if (p1) p1->bar();

    // p1 = p2; // error
}
```

Rvalue reference в шаблонах

"Склейка" ссылок:

- `A& &` → `A&`
- `A& &&` → `A&`
- `A&& &` → `A&`
- `A&& &&` → `A&&`

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`.

Perfect forwarding

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg const& arg) {
  return shared_ptr<T>(new T(arg));
}

template<typename T, typename Arg>
shared_ptr<T> factory(Arg && arg) {
  return shared_ptr<T>(new T(std::move(arg)));
}
```

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

Perfect forwarding

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg&& arg) {
  return shared_ptr<T>(new T(arg));
}
```

```
template<typename T, typename Arg>
shared_ptr<T> factory(Arg&& arg) {
  return shared_ptr<T>(new T(std::move(arg)));
}
```

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

Perfect forwarding in-depth

Давайте разберёмся как разворачиваются определения в случае lvalue и в случае rvalue.

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

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

Perfect forwarding in-depth: lvalue

```
    X x;
    auto p = factory<A>(x);
```

```
shared_ptr<A> factory(X& && arg) {
  return shared_ptr<A>(new A(std::forward<X&>(arg)));
}
X& && forward(remove_reference<X&>::type& a) noexcept {
  return static_cast<X& &&>(a);
}
```

```
shared_ptr<A> factory(X& arg) {
  return shared_ptr<A>(new A(std::forward<X&>(arg)));
}
X& forward(X& a) {
  return static_cast<X&>(a);
}
```

Perfect forwarding in-depth: rvalue

```
    X foo();
    auto p = factory<A>(foo());
```

```
shared_ptr<A> factory(X&& arg) {
  return shared_ptr<A>(new A(std::forward<X>(arg)));
}
X&& forward(remove_reference<X>::type& a) noexcept {
  return static_cast<X&&>(a);
}
```

```
shared_ptr<A> factory(X&& arg) {
  return shared_ptr<A>(new A(std::forward<X>(arg)));
}
X&& forward(X& a) noexcept {
  return static_cast<X&&>(a);
}
```

`std::move` in-depth

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

`std::move` in-depth: lvalue

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

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

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

std::move in-depth: rvalue

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

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

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

Замечание

std::move и std::forward не выполняют никаких действий времени выполнения.

Variadic templates + rvalue reference

```
template <typename... BaseClasses>
struct ClassName : BaseClasses... {
    ClassName (BaseClasses&&... base_classes)
        : BaseClasses(base_classes)...
    {}
};

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

template<typename ...Args> struct SomeStruct {
    static const int count = sizeof...(Args);
};

auto p = make_unique<std::vector>(10, 13);
```