

# 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
```