Стандарт C++ 11

Ссылки на временные объекты

Позволяют реализовать семантику переноса

```
Vector(const Vector& ref);
Vector(Vector&& ref);
Vector& operator=(const Vector& right);
Vector& operator=(Vector&& right);
```

constexpr

Функция возвращает константу времени компиляции

Ограничения на действия функции:

1. такая функция должна возвращать значение;
2. тело функции должно быть вида return выражение;
3. выражение должно состоять из констант и/или вызовов других constexpr-функций;
4. функция, обозначенная constexpr, не может использоваться до определения в текущей единице компиляции.

constexpr

```
double accelerationOfGravity = 9.8;
double moonGravity = accelerationOfGravity / 6; // ошибка
```

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

POD

- Тип простых данных (plain old data type (POD))
- Класс рассматривается как тип простых данных, если он *тривиальный* (*trivial*), *со стандартным размещением*(*standard-layout*) и если типы всех его нестатических членов-данных также являются типами простых данных
- Типы простых данных, могут использоваться в реализации объектного слоя, совместимого с C.

Тривиальный класс

- содержит тривиальный конструктор по умолчанию,
- не содержит нетривиальных копирующих конструкторов,
- не содержит нетривиальных перемещающих конструкторов,
- не содержит нетривиальных копирующих операторов присваивания,
- не содержит нетривиальных перемещающих операторов присваивания,
- содержит тривиальный деструктор.

Класс со стандартным размещением

- не содержит нестатических членов-данных, имеющих тип класса с нестандартным размещением (или массива элементов такого типа) или ссылочный тип,
- не содержит виртуальных функций,
- не содержит виртуальных базовых классов,
- имеет один и тот же вид доступности (public, private, protected) для всех нестатических членов-данных,
- не имеет базовых классов с нестандартным размещением,
- не является классом, одновременно содержащим унаследованные и неунаследованные нестатические члены-данные, или содержащим нестатические члены-данные, унаследованные сразу от нескольких базовых классов,
- не имеет базовых классов того же типа, что и у первого нестатического члена-данного (если таковой есть)

Списки инициализации

```
class SequenceClass
{
public:
        SequenceClass(std::initializer_list<int> list);
};
```

```
SequenceClass someVar = {1, 4, 5, 6};
```

Списки инициализации

```
void FunctionName(std::initializer_list<float> list);
FunctionName({1.0f, -3.45f, -0.4f});
```

```
std::vector<std::string> v = { "aaa", "bbb", "cccc" };
std::vector<std::string> v{"aaa", "bbb", "cccc" };
```

Универсальная инициализация

```
struct BasicStruct
{
        int x;
        double y;
};
```

`BasicStruct var1{5, 3.2};`	Инициализация агрегатов

Универсальная инициализация

```
struct AltStruct
{
        AltStruct(int x, double y) : x_(x), y_(y) {}
private:
        int x_; double y_;
};
```

`AltStruct var2{2, 4.3};`	Вызов конструктора

Универсальная инициализация

```
IdString GetString()
{
        return {"SomeName", 4};
}
```

```
std::vector<int> theVec{4};
```

auto

```
auto someStrangeCallableType = std::bind(
                                &SomeFunction, _2, _1, someObject);
auto otherVariable = 5;
```

```
for (vector<int>::const_iterator itr = myvec.cbegin();
        itr != myvec.cend(); ++itr) {}
```

```
for (auto itr = myvec.cbegin(); itr != myvec.cend(); ++itr) {}
```

decltype

```
#include <vector>
int main()
{
        const std::vector<int> v(1);
        auto a = v[0];          // тип a - int
        decltype(v[0]) b = 1; // тип b - const int& (возвращаемое значение
                              // std::vector<int>::operator[](size_type) const)
        auto c = 0;           // тип c - int
        auto d = c;           // тип d - int
        decltype(c) e;        // тип e - int, тип сущности, именованной как c
        decltype((c)) f = c; // тип f - int&, так как (c) является lvalue
        decltype(0) g;       // тип g - int, так как 0 является rvalue
}
```

Перебор коллекции

```
int my_array[5] = {1, 2, 3, 4, 5};
for(int &x : my_array)
{
        x *= 2;
}
```

Применимо к С-массивам, спискам инициализаторов и любым типам, для которых определены функции begin() и end()

Лямбда-функции

```
std::vector<int> someList;
int total = 0;
std::for_each(someList.begin(), someList.end(),
        [&total](int x) {
                total += x;
        }
);
std::cout << total;
```

Тип возвращаемого значения шаблона функции

```
template <typename LHS, typename RHS>
        decltype(std::declval<const LHS &>() +
                 std::declval<const RHS &>())
                AddingFunc(const LHS &lhs, const RHS &rhs)
{
        return lhs + rhs;
}
```

Тип возвращаемого значения шаблона функции

```
template <typename LHS, typename RHS>
        auto AddingFunc(const LHS &lhs, const RHS &rhs) ->
                decltype (lhs + rhs)
{
        return lhs + rhs;
}
```

Тип возвращаемого значения функции

```
struct SomeStruct
{
        auto FuncName(int x, int y) -> int;
};

auto SomeStruct::FuncName(int x, int y) -> int
{
        return x + y;
}
```

Вызов конструкторов

```
class SomeType
{
        int number;
public:
        SomeType(int new_number) : number(new_number) {}
        SomeType() : SomeType(42) {}
};
```

default

Спецификатор default означает реализацию по умолчанию
и может применяться только к специальным функциям-членам:

- конструктору по умолчанию;
- конструктору копий;
- конструктору перемещения;
- оператору присваивания;
- оператору перемещения;
- деструктору.

delete

Спецификатором delete помечают те методы, работать с которыми нельзя. Раньше приходилось объявлять такие конструкторы в приватной области класса

default и delete

```
class Foo
{
public:
        Foo() = default;
        Foo(const Foo&) = delete;
        void bar(int) = delete;
        void bar(double) {}
};
```

```
Foo obj;
obj.bar(5);    // ошибка!
obj.bar(5.42); // ok
```

override

```
class Shape
{
public:
    void Print()const;
    virtual double Area()const = 0;
    virtual double Perimetr()const =0;
};
```

```
class Circle: public Shape{
public:
    virtual void Print()const override;    //ошибка
    virtual double Area() override;         //ошибка
    virtual int Perimetr()const override; //ошибка
private:
    double radius;
};
```

final

```
class Shape
{
public:
     virtual void Print()const final;
};
```

```
class Circle: public Shape{
public:
    virtual void Print()const;    //ошибка
};
```

final

```
class Shape final
{
    …
};
```

```
class Circle: public Shape   //ошибка
{
    …
};
```

override и final

Идентификаторы `override` и `final` имеют специальное значение только при использовании в определённых ситуациях. В остальных случаях они могут использоваться в качестве нормальных идентификаторов (например, как имя переменной или функции).

nullptr

```
void foo(char *);
void foo(int);
```

```
foo(nullptr);   // вызывает foo(int), а не foo(char*)
```

nullptr

```
void foo(char *);
void foo(int);
```

```
char *pc = nullptr; // верно
int *pi = nullptr;  // верно
bool b = nullptr;   // верно. b = false.
int i = nullptr;    // ошибка

foo(nullptr);       // вызывает foo(char *), а не foo(int);
```

Перечисления со строгой типизацией

```
enum class Enumeration
{
        Val1,
        Val2,
        Val3 = 100,
        Val4, /* = 101 */
};
```

Создание шаблонного синонима

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

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

Использование using как typedef

```
typedef void (*OtherType)(double);  // Старый стиль
using OtherType = void (*)(double); // Новый синтаксис
```

Конец