# Дополнение: краткая история и некоторые замечания по развитию языка C++

Язык C++ разработал сотрудник лабораторий [Bell](http://ru.wikipedia.org/w/index.php?title=Bell&action=edit) Бьерн Страуструп, который начал работать над ним в **1979** году. Первоначально язык назывался "Си с классами", так как в начале в Си были добавлены классы (с инкапсуляцией), производные классы, строгая проверка типов, inline-функции и аргументы по умолчанию.

В 1**983** году (Рик Маскитти) произошло переименование языка из "Си с классами" в C++. Кроме того, в него были добавлены новые возможности, такие как виртуальные функции, перегрузка функций и операторов, ссылки, константы, пользовательский контроль над управлением свободной памятью, улучшенная проверка типов и новый стиль комментариев (//).

В **1985** году вышло первое издание Бьерна Страуструпа "Язык программирования C++".

В **1989** году вышла версии 2.0. языка C++ . Его новые возможности включали множественное наследование, абстрактные классы, статические функции-члены, функции-константы и защищённые члены.

В **1998** году был ратифицирован международный стандарт языка C++: ISO/IEC 14882 "Standard for the C++ Programming Language".

Стандарт C++ 1998 года состоит из двух основных частей: ядра языка и стандартной [библиотеки](http://ru.wikipedia.org/wiki/%D0%91%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B0_%28%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5%29), которая включает [STL](http://ru.wikipedia.org/w/index.php?title=STL&action=edit) (Стандартную Библиотеку Шаблонов) и версию стандартной библиотеки Си.

В **2003** году была выпущена новая версия стандарта языка C++ — ISO/IEC 14882:2003.

В настоящее время нет единого языка программирования C++, это название относится к семейству языков. Никто не обладает правами на язык C++, он является свободным.

**Замечание**: При написании программ надо учитывать, на каком компиляторе будет прогоняться программа. Большая часть кода Си будет справедлива и для C++, но, все-таки, C++ не является надмножеством Си и не включает его в себя. То есть, код, справедливый для обоих Си и C++, может давать разные результаты в зависимости от того, где он скомпилирован.

**Пример**: // отличия Си от C++

#include <stdio.h>

int main()

{

printf("%s\n", (sizeof('a') == sizeof(char)) ? "C++" : "C");

return 0;

}

**Комментарий**: программа печатает "С", если компилируется компилятором Си, и "C++" — если компилятором C++. Так происходит из-за того, что символьные константы в Си (в примере 'a') представлены целым типом int, а в C++ — типом char.

Далее будут рассмотрены и некоторые другие отличия языка Си от языка C++.

Visual C++ позволяет разрабатывать приложения в терминах традиционного процедурного, объектно-ориентированного, консольного и визуального программирования.

В базовом курсе будет использовать консольное приложение, то есть приложение, которое не использует Windows-окна для обработки сообщений от пользователя (в дополнение могут быть даны элементы визуального программирования). Точкой входа в консольное приложение в языке C++ является метод main().

Каждая программа должна иметь точку входа и может содержать описание одного или нескольких классов.

**Недостатки ООП**

1. Документирование классов - задача более трудная, чем это было в случае процедур и модулей. Для абстрактных методов, которые пусты, в документации должно говориться о том, для каких целей предполагается использовать переопределяемый метод.
2. Реализация выполнения программы рассредоточивается по нескольким классам, и чтобы понять, как она работает, необходимо просматривать весь код.
3. Неоднозначность в использовании инкапсуляции данных.
4. Излишняя универсальность. В объектно-ориентированной программе обычно реализованы избыточные возможности. В библиотечном классе часто содержится больше методов, чем это реально необходимо, но они не могут быть удалены.
5. Рост объёма памяти для выполнения программы и уменьшение скорости выполнения программы с использованием ООП.

**Замечание**: Надежность объекто-ориентированного программного обеспечения и быстрота его написания дают определяющие преимущества перед использованием процедурного программного обеспечения.

**Классы (продолжение)**

**Перегрузка функций**

В C++ две или более функции могут иметь одно и то же имя, отличаясь либо типом, либо числом своих аргументов, либо и тем и другим.

Если две или более функции имеют одинаковое имя, то говорят, что они перегружены.

Для перегрузки функции надо просто объявить и определить все требуемые варианты. Компилятор автоматически выберет правильный вариант вызова на основании числа или типа используемых в функции аргументов.

Одно из основных применений перегрузки функций – это достижение полиморфизма (один интерфейс – множество методов).

***Перегрузка функции. Определение адреса перегруженной функции***

Также как и в Си можно присвоить адрес функции указателю и получить доступ к функции через указатель.

Адрес функции можно найти, если поместить имя функции в правой части инструкции присваивания без всяких скобок или аргументов.

**Пример:**// *Перегрузка функции. Определение адреса перегруженной функции*

#include <iostream>

using namespace std;

//Функция вычисления квадрата

void sq(float length)

{

cout << "Square=" << length \* length << endl;

}

//Функция вычисления прямоугольника

void sq(float length, float width)

{

cout << "Rectangle=" << width \* length << endl;

}

int main()

{

sq(10);

sq(3, 7);

// Вызов функции через указатели

void(\*fp1)(float);

void(\*fp2)(float, float);

fp1 = sq; // получение адреса функции вычисления площади квадрата

fp2 = sq; // получение адреса функции вычисления площади прямоугольника

fp1(5); // вызов функции вычисления площади квадрата

fp2(4, 5); // вызов функции вычисления площади прямоугольника

system("pause");

return(0);

}

***Встраиваемые функции (inline)***

В C++ существуют функции, которые не вызываются, а "встраиваются" в программу в место ее вызова (тело функции подставляется в каждую точку вызова).

Механизм "встраивания" функций подобен макросам языка Си, определенных с использованием директивы препроцессора #define. Однако между ними есть различия. Вызовы "встроенных" функций обрабатываются компилятором, а макросы обрабатываются препроцессором, который делает просто текстовую подстановку. При вызове "встраиваемых" функций компилятор проверяет типы аргументов.

*Преимущество использования "встроенных" функций*: нет вызова, поэтому выполняется быстрее.

*Недостаток*: если они большие, то занимают большой объем памяти.

*Объявление "встраиваемых" функций*: пишется спецификатор *inline* перед определением функции.

**Пример**: // *Использование встраиваемых функций*

#include<iostream>

using namespace std;

inline int choose(int x)

{

return (x % 2);

}

int main()

{

int addition = 0, multiplication = 1;

for (int i = 0; i <= 5; i++)

if (choose(i))

multiplication \*= i;

else

addition += i;

cout << "addition=" << addition << endl;

cout << "multiplication=" << multiplication << endl;

system("pause");

return 0;

}

*// Встраиваемые функции в объявлении класса*

#include<iostream>

using namespace std;

class sample\_inline {

public:

int choose(int);

};

inline int sample\_inline::choose(int x)

{

return (x % 2);

}

int main()

{

int addition = 0, multiplication = 1;

sample\_inline ob;

for (int i = 0; i <= 5; i++)

if (ob.choose(i))

multiplication \*= i;

else

addition += i;

cout << "addition=" << addition << endl;

cout << "multiplication=" << multiplication << endl;

system("pause");

return 0;

}

*// Компактный стиль встраиваемых функций в объявлении класса*

#include<iostream>

using namespace std;

class sample\_inline {

public:

int choose(int x) { return (x % 2); }

};

int main()

{

int addition = 0, multiplication = 1;

sample\_inline ob;

for (int i = 0; i <= 5; i++)

if (ob.choose(i))

multiplication \*= i;

else

addition += i;

cout << "addition=" << addition << endl;

cout << "multiplication=" << multiplication << endl;

system("pause");

return 0;

}

**То есть, если встраиваемая функция задана внутри класса, то ключевое слово inline не требуется.**

**Замечание:** спецификатор inline для компилятора является запросом, а не командой. Если компилятор не в состоянии выполнить запрос, функция будет компилироваться как обычная функция, а запрос inline будет проигнорирован. (В некоторых компиляторах не воспринимает циклы, goto, switch …).

# Конструкторы и деструкторы

*Рассмотрим* ***вопросы инициализации*** *при работе с классами в C++.*

Для инициализации при работе с объектами в C++ имеется функция-конструктор (constructor function), включаемая в описание класса.

В классе может быть несколько конструкторов и только одна функция, обратная конструктору, называемая деструктором (destructor). Количество конструкторов определяется количеством имеющихся способов создания объекта. Конструктор выполняет инициализацию переменных-членов класса и резервирует динамическую память для объекта.

Конструктор класса вызывается всякий раз при создании объекта этого класса. Таким образом, необходимая объекту инициализация, при наличии конструктора, выполняется автоматически.

*Конструктор имеет то же имя*, что и класс, частью которого он является, и не имеет возвращаемого значения.

*Функция – деструктор (деструктор)*, вызывается при удалении объекта и служит для освобождения памяти. Имя деструктора совпадает с именем класса, но с символом ~ (тильда) в начале.

***Конструктор без параметров***

Конструктор без параметров инициализирует переменные-члены, присваивая им устанавливаемые по умолчанию значения. Если конструктор не определен, компилятор генерирует его по умолчанию (*таким образом, конструктор существует всегда*).

**Замечание**: Если аргументы конструктору не передаются, то в определении объекта не следует включать пустые скобки (иначе будет определена функция, возвращающая тип класса).

**Пример**: // *Работа конструктора без параметров*

#include<iostream>

using namespace std;

class myclass {

int a;

public:

myclass(); // конструктор

void show();

};

myclass::myclass()

{

cout << "Пример инициализации в конструкторе \n";

a = 25;

}

void myclass::show()

{

cout << a << endl;

}

int main()

{

setlocale(LC\_ALL, "Russian");

myclass ob;

ob.show();

system("pause");

return 0;

}

Для глобальных объектов конструктор объекта вызывается тогда, когда начинается выполнение программы. Для локальных объектов конструктор вызывается всякий раз при выполнении инструкции объявления переменной.

**Пример**:// фрагмент программы работы деструктора

~myclass(); // объявление деструктора в классе

myclass::~myclass() // определение деструктора вне класса

{

… // тело функции

}

Локальные объекты удаляются тогда, когда они выходят из области видимости. Глобальные объекты удаляются при завершении программы. Адреса конструктора и деструктора получить невозможно.

**Замечание**: Создать экземпляр класса можно при условии, что конструктор является функцией-членом класса типа *public*. Если класс предназначен лишь для создания других классов, конструктор можно сделать защищенным членом, поместив его в раздел *protected*.

***Конструктор с параметрами***

Конструктору можно передавать аргументы.

**Пример**:// *Работа конструктора с параметром*

#include<iostream>

using namespace std;

class myclass {

int a;

public:

myclass(int x); // конструктор

void show();

};

myclass::myclass(int x)

{

cout << "Инициализация в конструкторе \n";

a = x;

}

void myclass::show()

{

cout << a << endl;

}

int main()

{

setlocale(LC\_ALL, "Russian");

myclass ob(25);

ob.show();

system("pause");

return 0;

}

***Значение, передаваемое в myclass(), используется для инициализации параметра a.***

**Замечание**: Выражение myclass ob(25); эквивалентно выражению

myclass ob = myclass(25);

Существует еще одна форма инициализации конструктора – инициализация расположена между прототипом функции и телом функции. От прототипа функции список инициализации отделен двоеточием, а инициализирующие значения помещены в скобках, после соответствующих переменных класса, через запятую. Инициализация переменных класса происходит до начала исполнения тела конструктора. Такой подход позволяет задавать, например, начальные значения констант и ссылок.

**Пример**:

#include<iostream>

using namespace std;

class myclass {

int a, b;

float c;

const int d;

public:

myclass(int x, int y, float z, int w) : a(x), b(y), c(z), d(w) { cout << "Инициализация в конструкторе \n"; }

void show();

};

void myclass::show()

{

cout << "a=" << a << " b=" << b << " c=" << c << " d=" << d << endl;

}

int main()

{

setlocale(LC\_ALL, "Russian");

myclass ob(25, 20, 15.5, 55);

ob.show();

system("pause");

return 0;

}

**Задание**: Проверить невозможность инициализации константы в конструкторе другим способом (без списка инициализации).