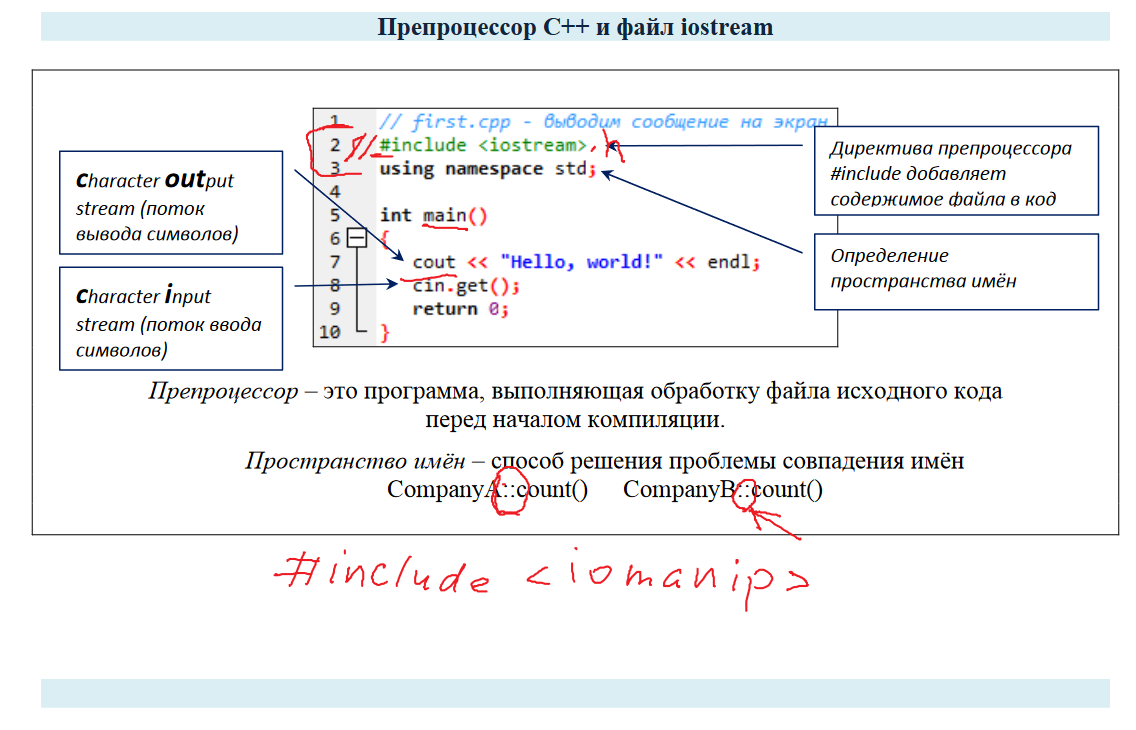
**Вопросы для самоконтроля по дисциплине «Программирование»  
за 1 семестр обучения.**

1. Структура программы на С++.

Директива #include.

Директива работает по такому принципу, что как только макропроцессор встречает эту директиву, то вставляет содержимое указанного файла на ее место.



В целом, этого понимания работы директивы #include вполне достаточно для программирования на Си. Отмечу только один нюанс. Имя файла после директивы #include можно заключать или в угловых скобках, или в кавычках. Например, если ее записать в виде:

#include "stdio.h"

то ничего не изменится и программа также успешно откомпилируется. Но отличия в работе директивы все же есть. Согласно стандарту C99 файл прописанный в угловых скобках сначала ищется в системных каталогах, и если не будет найден, то в текущем рабочем каталоге проекта. Если же прописываются двойные кавычки, то поиск работает наоборот, сначала файл ищется в рабочем каталоге проекта, а затем только в системных каталогах.

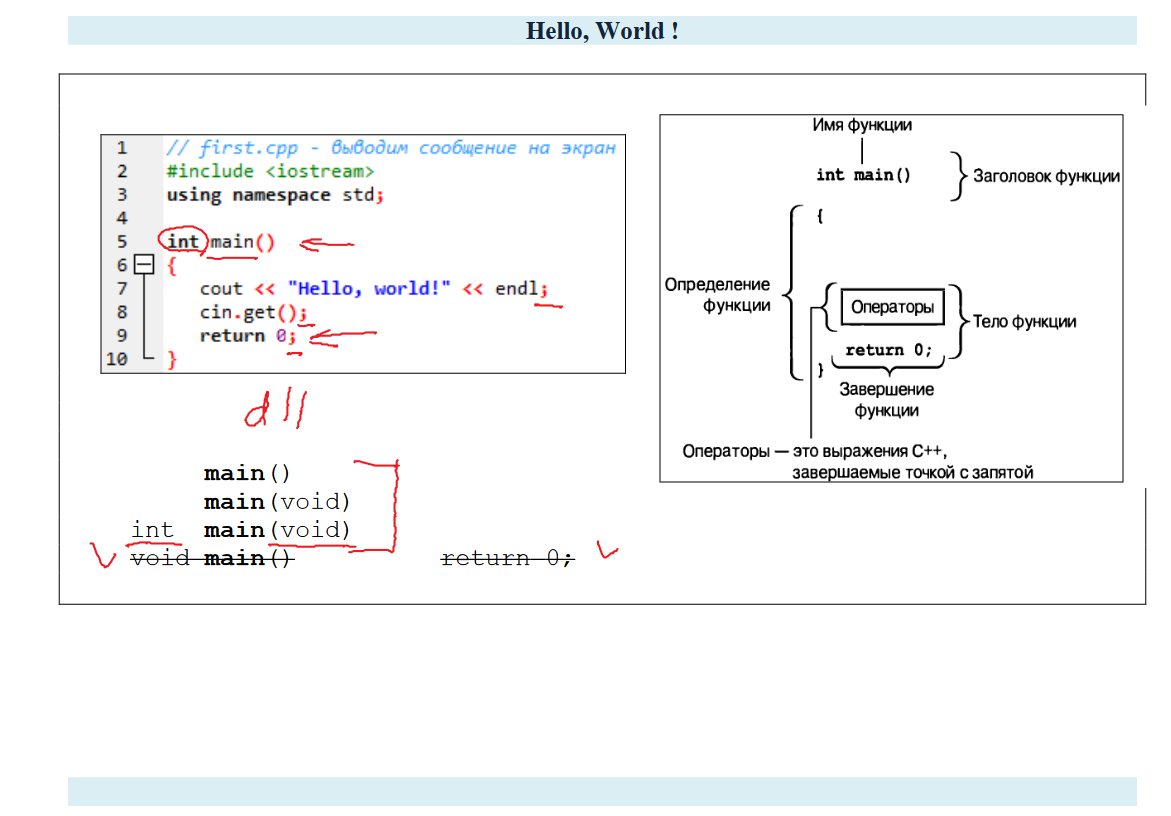
В действительности, эти правила не всегда соблюдаются и компиляторы могут реализовывать свою логику поиска файлов. Мало того, в последних стандартах языка Си алгоритм поиска файлов несколько изменен и стал таким неопределенным, что разработчики компиляторов этот пункт часто просто игнорируют. Благо, нам, как программистам на Си важно лишь, что макропроцессор будет искать указанный файл и добавлять его содержимое в наш проект. А правило использования угловых скобок и кавычек в сообществе разработчиков стало таким:

*Если нужно подключить стандартные файлы, поставляемые с компилятором языка Си, то следует писать угловые скобки. Если же подключаются свои собственные заголовочные файлы, то их имена следует заключать в двойные кавычки.*

Функция main().

Теперь пришло время познакомиться и разобрать структуру простой, но типовой программы.

**2.Функция** **main** - особая **функция**, определяющая точку входа программы (по сути, программа представляет собой именно **функцию** **main**, вызывающую другие **функции**) При написании программы на языке С++ требуется специальная функция **main()**, потому что она является отправной точкой для выполнения всех программ на этом языке.



Часто в учебниках или на лекциях для этой цели приводят программу, печатающую на экране строку «Hello, World!». Я не стану нарушать эту традицию и поступлю ровно также, тем более, что пример достаточно удачный.

*/\* Моя первая программа \*/*

#include <stdio.h>

int main(void)

{

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Hello, World**\n**");

    return 0;

}

Первое, что сразу же следует отметить, это то, что текст программы пишется программистом для себя и/или других программистов. И только во вторую очередь для компилятора. Это означает, что в программе следует соблюдать форматирование текста для лучшего визуального восприятия, а также не использовать сложные синтаксические конструкции, если без ущерба для производительности программы можно записать их в более простой форме. Чаще всего искусственное усложнение программного кода можно наблюдать у новичков, которые этим хотят показать свое виртуозное владение языком программирования. В рабочей среде, в лучшем случае, это вызывает только улыбку, а в худшем – требование переписать код в понятном для человека виде. Не совершайте таких ошибок и сразу правильно оформляйте тексты программ. Как вариант, можно ориентироваться на оформление в текущем курсе.

Итак, у нас имеется простейшая программа. Давайте теперь построчно разберем, что здесь к чему. В первой строчке записан комментарий. Обычно, это некий текст, который пишется для программистов с разъяснением работы отдельных операторов или фрагментов кода. Компилятор игнорирует все комментарии и не переводит их в машинный код.

До стандарта C99 комментарии можно было оформлять только с помощью комбинаций символов /\* \*/. И все, что заключено в них, считается комментарием. Его можно записывать и в одну строчку и в несколько строчек, например, так:

*/\* Это*

*многострочный*

*комментарий*

*\*/*

С появлением стандарта C99 добавились однострочные комментарии, обозначаемые символами //. Например, комментарий в нашей программе можно оформить и так:

*// Моя первая программа*

Но, обычно, однострочные комментарии используют для разъяснения отдельных операторов программы, например:

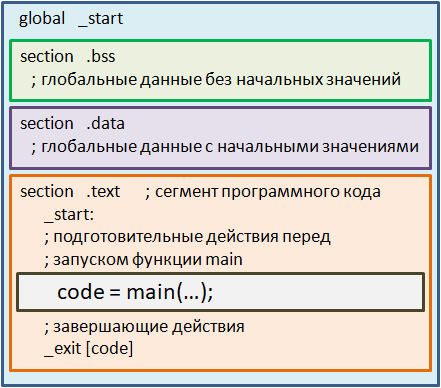
[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Hello, World**\n**");   *// вывод строки на экран*

Следующая строчка программы – это, так называемая, директива препроцессора. Подробно о них мы еще будем говорить. Здесь отмечу только, что все директивы начинаются с символа шарп ‘#’ и следом идет ее имя. В частности, директива #include выполняет включение содержимого заголовочного файла stdio.h в то место, где записана эта директива. Сразу возникает вопрос, зачем это потребовалось в программе? Дело в том, что база языка Си весьма ограничена. Изначально в нем не реализовано даже функций ввода/вывода. Поэтому для вывода какой-либо информации на экран нам необходимо воспользоваться внешней библиотечной функцией printf(), которая записана ниже в программе. Так вот, в текстовом файле stdio.h, в частности, прописано объявление функции printf(). Обратите внимание, объявление функции, но не ее реализация. Компилятор языка Си так устроен, что для формирования объектного файла текущего модуля ему достаточно дать определение (объявление) функции printf(), а конкретная реализация добавится в программу на этапе линковки кода, то есть, редактором связей. Эта реализация будет взята из библиотечного откомпилированного модуля, поставляемого вместе с компилятором языка Си. Поэтому, все что нужно будет сделать линковщику – это найти фрагмент машинного кода для функции printf() и добавить в исполняемый файл. Или же, при использовании динамических библиотек (DLL), реализация многих стандартных функций не добавляется в скомпилированный файл, а берется из соответствующей динамической библиотеки, хранимой в ОС. Но на данном этапе, для нас важно лишь одно: для успешного транслирования программы в машинный код компилятору достаточно дать лишь описание используемых функций, а их реализации добавит позже редактор связей. Вот для этого и нужна в программе директива #include с подключением файла stdio.h, имя которого является сокращением от английских слов:

standard input ouput

(стандартный ввод/вывод). А расширение h – это первая буква от слова **header** (заголовок). Получается заголовочный файл стандартного ввода/вывода.

В следующей строчке программы идет объявление функции с именем main. Опять же, зачем это понадобилось? Забегая вперед, отмечу, что функции – это активные элементы программы, которые выполняют, заключенные в них операторы (инструкции). Вся программа на языке Си – это, по сути, наборы разных функций и их вызов в запрограммированном порядке. Так вот, первая функция, с которой начинается выполнение программы, должна называться **main**. Почему именно main? Потому, что так решил разработчик этого языка Деннис Ритчи в далеком 1972 году. Все компиляторы языка Си формируют машинный код так, что после загрузки программы в память компьютера, ОС передает управление сначала, так называемой, секции кода, как правило, с переходом по метке \_start. Там выполняются некоторые подготовительные действия, а затем, вызывается функция main(). И это всегда так. Мало того, если в программе будет отсутствовать такая функция, то при компиляции появится сообщение об ошибке.



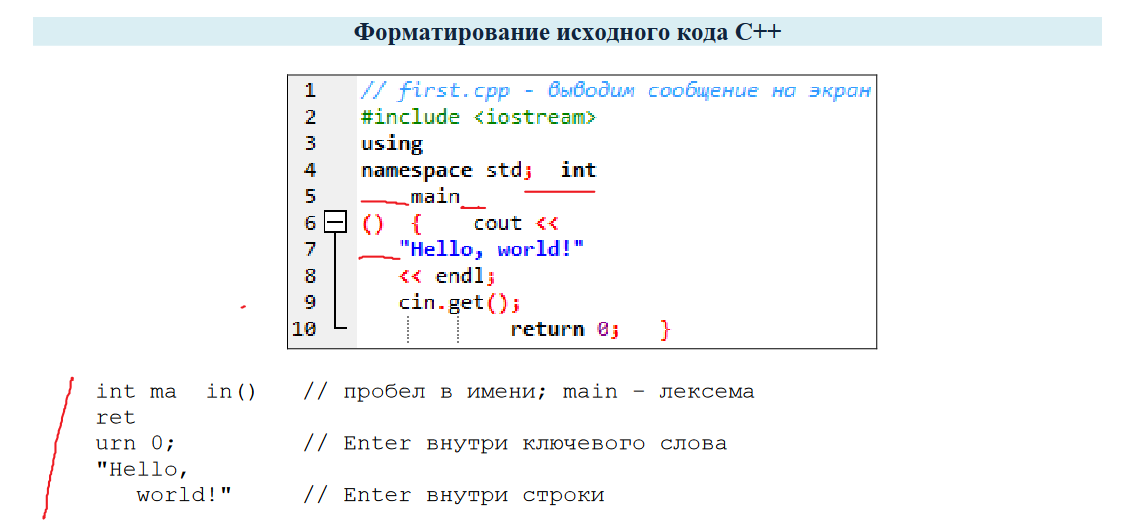
Итак, в программе на языке Си обязательно должна присутствовать функция с именем main, которой передается управление после запуска программы и выполнения некоторых подготовительных действий. Условно этот процесс показан на рисунке.

Далее, перед именем этой функции записано ключевое слово int. Оно означает целочисленный тип данных и то, что функция main() возвращает целочисленное значение. Обо всем этом мы еще подробно будем говорить, здесь я лишь даю краткие пояснения. После имени main должны идти круглые скобки и в них прописаны параметры функции. Так как в нашем примере функция main() не имеет параметров, то в круглых скобках указано ключевое слово void, которое можно перевести с английского как «пустой». Далее, в фигурных операторных скобках прописываются операторы, которые будут последовательно выполняться (сверху-вниз) при вызове функции main(). В частности, в нашей программе, сначала будет вызвана библиотечная функция printf() для вывода указанной строки в консоль, а затем, сработает оператор:

return 0;

Опять же, забегая вперед, оператор return выполняет два действия. Первое, он завершает текущий вызов функции main(). И второе, в нем указывается значение, которое будет возвращено этой функцией. Почему здесь прописано значение 0? Дело в том, что это значение, затем, через системный вызов \_exit будет передано операционной системе. Собственно, этот системный вызов и завершает программу, а вовсе не функция main(). Так вот, значение 0 говорит ОС о том, что программа выполнила свою функцию и успешно решила поставленную перед ней задачу. Все другие значения, отличные от нуля, будут означать ошибки выполнения программы. Например, мы что то планируем записать в файл, а ОС не позволяет открыть указанный файл на запись. Тогда программа может выдать сообщение об ошибке и вернуть код, отличный от нуля. Хотя, в современной практике программирования, обычно, так не поступают и просто выводят сообщение об ошибке с кодом завершения 0.

Форматирование исходного кода С++. Пробельные символы.



Правила хорошего стиля оформления кода:

– один оператор в одной строке;

– открывающая и закрывающая фигурные скобки для функции, каждая из которых

находится в своей строке;

– операторы в операторных скобках, функциях записаны с отступом от фигурных скобок;

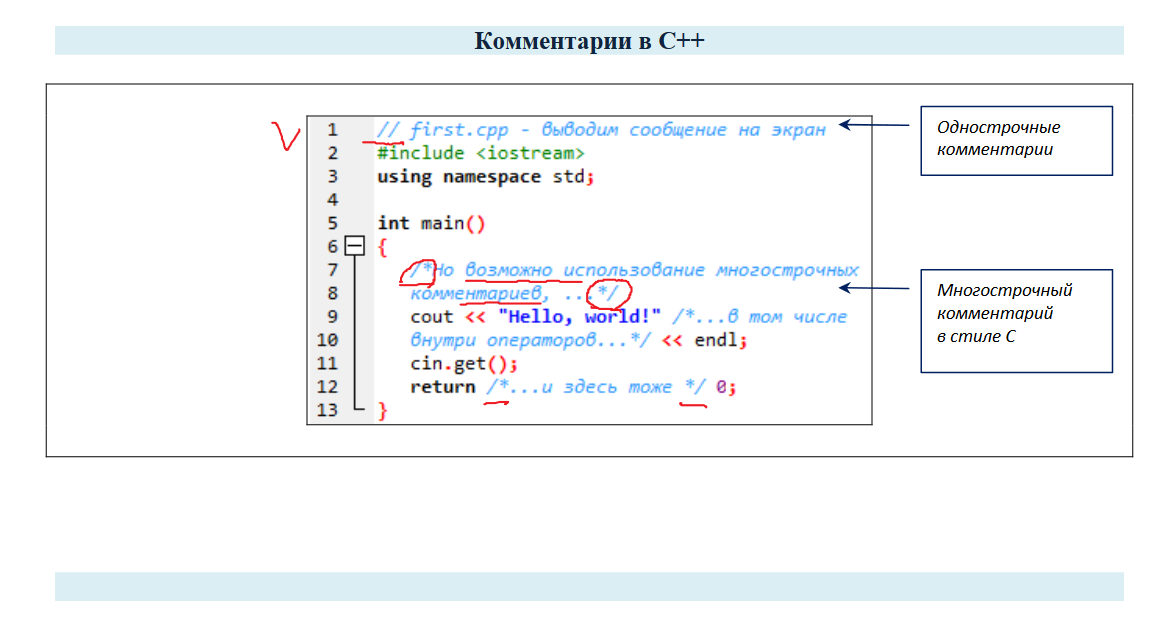
– вокруг круглых скобок, связанных с именем функции, пробельные символы

отсутствуют

и др.

В отличие от других языков программирования, C++ не имеет каких-либо ограничений в форматировании кода со стороны программистов. Основное правило заключается в том, чтобы использовать только те способы, которые максимально улучшают читабельность и логичность кода. **Пробелы**относятся к символам, которые используются в форматировании кода, вместе с символами табуляции и, иногда, разрывом строки. Компилятор, как правило, игнорирует пробелы, но все же есть небольшие исключения. Исключением, где компилятор учитывает пробелы, является цитируемый текст, например: "Hello, world!". "Hello, world!"отличается от "Hello,     world!" **Разрыв/перевод строки** не допускается в цитируемом тексте. Еще одним исключением, где компилятор обращает внимание на пробелы, являются **однострочные комментарии**: они занимают только одну строку. Перенос однострочного комментария на вторую строку вызовет ошибку компиляции.

Комментарии в стиле С и С++.



Компилятор игнорирует все комментарии и не переводит их в машинный код.

До стандарта C99 комментарии можно было оформлять только с помощью комбинаций символов /\* \*/. И все, что заключено в них, считается комментарием. Его можно записывать и в одну строчку и в несколько строчек, например, так:

*/\* Это*

*многострочный*

*комментарий*

*\*/*

С появлением стандарта C99 добавились однострочные комментарии, обозначаемые символами //. Например, комментарий в нашей программе можно оформить и так:

*// Моя первая программа*

Но, обычно, однострочные комментарии используют для разъяснения отдельных операторов программы, например:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Hello, World**\n**");   *// вывод строки на экран*

Пространства имён. Директива using.



Пространство имён – способ решения проблемы совпадения имён

CompanyA::count() CompanyB::count()

На этом занятии мы с вами познакомимся с новой концепцией языка С++ **пространством имен**. Из курса по языку Си мы знаем, что в глобальной области видимости можно объявлять функции, переменные, структуры и так далее:

void foo()

{

    std::cout << "function: foo()" << std::endl;

}

int global\_a = 5;

struct point {

    double x, y;

};

Так вот, в языке С++ все подобные объявления попадают в так называемое **глобальное пространство имен**. В частности, это означает, что внутри функции main() мы можем обращаться к этим определениям следующим образом:

int main()

{

    point pt {};

    foo();

    std::cout << global\_a << std::endl;

    return 0;

}

или так:

int main()

{

    ::point pt {};

    ::foo();

    std::cout << ::global\_a << std::endl;

    return 0;

}

Обратите внимание на два двоеточия (программисты еще иногда их называют «четвероточием»). Формально, это **символ раскрытия области видимости**. Если перед ним (слева) не указана какая-либо область видимости, то подразумевается глобальная область (глобальное пространство имен).

Что нам дает этот символ? Смотрите, если, к примеру, внутри функции main() объявить еще одну переменную с именем global\_a:

int global\_a = 10;

то мы можем их различать с помощью «четвероточия»:

std::cout << ::global\_a << " " << global\_a << std::endl;

Увидим значения 5 и 10. Поэтому, если мы хотим быть уверены, что обращаемся к глобальной переменной, то перед ее именем достаточно прописать два двоеточия. В ряде случаев это бывает очень полезно.

Далее, из курса по Си мы знаем, что если в глобальном пространстве имен определить два определения с одинаковыми именами, например, две функции с именем foo:

void foo() ...

void foo() ...

то компилятор выдаст сообщение об ошибке. Но, несмотря на всю очевидность этого факта, такое все же вполне может произойти. Например, когда над большим проектом работает группа разработчиков, то независимо друг от друга программисты могут создавать совершенно одинаковые прототипы функций с разными реализациями. И при попытке собрать проект целиком возникает описанная проблема.

Или же мы написали свою библиотеку функций или классов, подключаем ее к проекту, где используются другие стандартные библиотеки, и видим тот же конфликт имен. Хотя при разработке своего отдельного модуля мы с этим никак не сталкивались. И подобных ситуаций может быть множество. Как же этого избежать? Для в С++ существует возможность создавать свои пространства имен, которые позволяют разделять один набор определений от другого набора. Для этого используется ключевое слово namespace, после которого указывается название нового или существующего пространства. Например, так:

namespace firstSpace {

    void foo()

    {

        std::cout << "function from firstSpace: foo()" << std::endl;

    }

}

void foo()

{

    std::cout << "function: foo()" << std::endl;

}

Здесь название firstSpace мы придумываем сами подобно именам переменных и функций, а, затем, в фигурных скобках следует содержимое этого пространства. В нашем примере там располагается функция с именем foo. Теперь программа будет компилироваться без проблем, так как функции foo() находятся в разных пространствах имен. Но давайте вторую функцию тоже поместим в отдельное пространство с именем secondSpace:

namespace secondSpace {

    void foo()

    {

        std::cout << "function from secondSpace: foo()" << std::endl;

    }

}

Отлично, но как теперь следует вызывать эти функции? Делается это следующим образом:

firstSpace::foo();

То есть, сначала указывается название пространства имен, а затем, через два двоеточия элемент этого пространства. В данном случае вызывается функция foo().

Аналогично можно вызвать функцию foo() из другого пространства:

secondSpace::foo();

После запуска программы видим, что были вызваны две разные функции foo() из соответствующих пространств имен. Как видите, все достаточно просто.

Теперь несколько важных деталей. Во-первых, определение одного и того же пространства имен определять в разных местах программы (проекта), например, следующим образом:

namespace firstSpace {

    void foo()

    {

        std::cout << "function from firstSpace: foo()" << std::endl;

    }

}

namespace firstSpace {

    struct point {

        double x, y;

    };

}

В результате будет создано пространство firstSpace с функцией foo и структурой point. Это эквивалентно тому, если бы мы их прописали в одном определении firstSpace:

namespace firstSpace {

    void foo()

    {

        std::cout << "function from firstSpace: foo()" << std::endl;

    }

    struct point {

        double x, y;

    };

}

Данный прием бывает полезен, когда программист разбивает свой проект на несколько файлов и в каждом нужно прописывать свою реализацию одного и того же пространства имен. Тогда в каждом файле достаточно прописать одно и то же имя после ключевого слова namespace.

Следующее важное замечание касается вложенных пространств имен. Например, так:

namespace secondSpace {

    namespace functions {

        void foo()

        {

            std::cout << "function from secondSpace: foo()" << std::endl;

        }

    }

    namespace params {

        int global\_a = 5;

    }

}

Соответственно, обращение к элементам такого пространства будет выглядеть следующим образом:

secondSpace::functions::foo();

std::cout << secondSpace::params::global\_a << std::endl;

И последний штрих. Если перед именем вложенного пространства прописать ключевое слово inline, например, так:

namespace secondSpace {

**inline** namespace functions {

        void foo()

        {

            std::cout << "function from secondSpace: foo()" << std::endl;

        }

    }

    ...

}

то доступ к функции foo() можно делать следующим образом:

secondSpace::functions::foo();

secondSpace::foo();

То есть, имя inline-пространства указывать не обязательно.

На этом мы с вами завершим текущее занятие, а на следующем рассмотрим оператор using, который позволяет включать все или отдельные определения одного пространства имен в другое.



На прошлых занятиях для обращения к элементам пространства имен std нам приходилось прописывать std::cin, std::cout и так далее. Это не очень удобно. Если в модуле (файле) предполагается постоянное использование каких-либо элементов из того или иного пространства имен, то эти элементы можно импортировать непосредственно в этот модуль. В частности, для этого предназначен оператор using. Использовать его можно в соответствии со следующим синтаксисом:

using <пространство имен>::<элемент>;

Например, чтобы импортировать объекты cin/cout и функцию endl в текущий модуль в глобальное пространство имен, достаточно прописать:

#include <iostream>

using std::cout;

using std::cin;

using std::endl;

int main()

{

    char str[50];

    cin >> str;

    cout << "Hello, " << str << "!" << endl;

    return 0;

}

Если же этот оператор прописать в каком-либо блоке, например, в теле функции main(), то импорт будет сделан в пределах этого блока:

#include <iostream>

int main()

{

    using std::cout;

    using std::cin;

    using std::endl;

    char str[50];

    cin >> str;

    cout << "Hello, " << str << "!" << endl;

    return 0;

}

Соответственно, определения cout, cin и endl доступны теперь только внутри тела функции main() и не доступны за его пределами.

Обычно, на практике, отдельные элементы импортируются в глобальную область или в пространство имен, где их активно предполагается использовать. А вот делать импорт всех определений того или иного пространства имен следует исключительно в локальные области.

Давайте посмотрим, как синтаксически выполняется импорт всего пространства. Для этого также прописывается ключевое слово using, за которым следует еще одно ключевое слово namespace с указанием имени импортируемого пространства. Например:

int main()

{

    using namespace std;

*// using std::cout;*

*// using std::cin;*

*// using std::endl;*

    char str[50];

    cin >> str;

    cout << "Hello, " << str << "!" << endl;

    return 0;

}

В результате, все определения из std становятся доступными напрямую в теле функции main(). И, еще раз, обратите внимание. Писать подобный импорт в глобальной области – крайне плохая практика:

#include <iostream>

using namespace std;

int main()

{

    char str[50];

    cin >> str;

    cout << "Hello, " << str << "!" << endl;

    return 0;

}

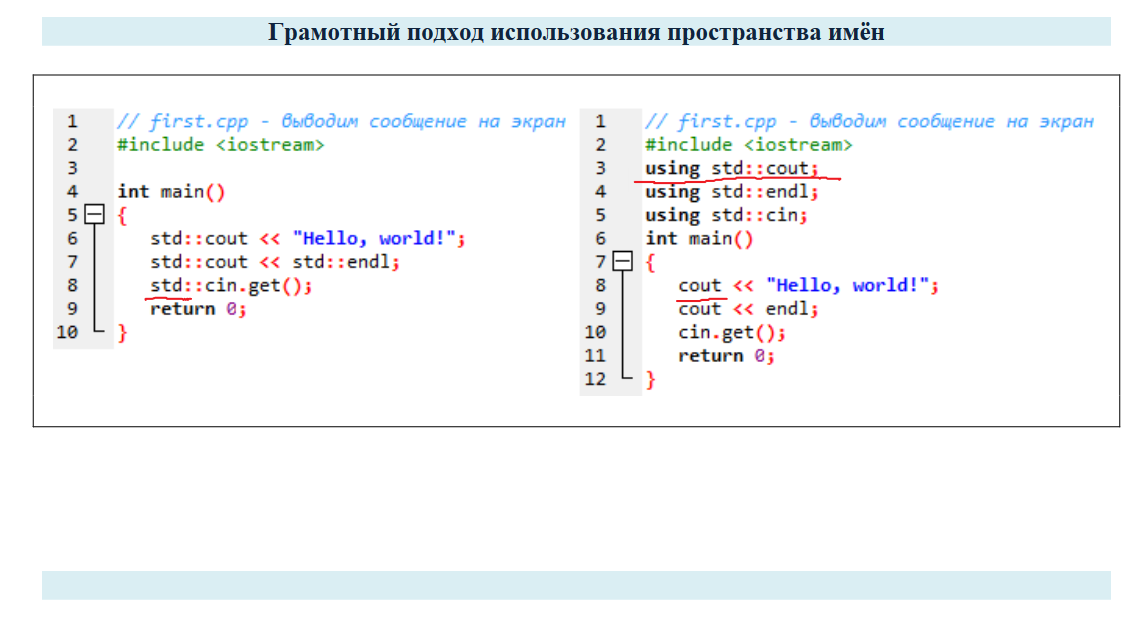
Часто, далеко не все, что прописано в std, используется в текущем модуле. И программист вполне может случайно объявить переменную или функцию или что-либо еще с тем же именем, что и импортированный элемент. Возникнет конфликт имен и компиляция программы завершится с ошибкой. Как раз чтобы этого избежать, лучше явно указывать импортируемые элементы:

using std::cout;

using std::cin;

using std::endl;

Или, в крайнем случае, делать полный импорт в локальную область видимости.



**Определение псевдонимов типов**

Но это еще не все, что умеет делать оператор using. С его помощью можно создавать псевдонимы (алиасы) существующих в программе типов данных. Делается это по следующему синтаксису:

using <alias> = <тип данных>;

Например, в самом простом варианте, можно прописать что-то вроде:

using byte\_8 = unsigned char;

В программе появляется новое имя byte\_8 базового типа unsigned char, которое полноценно можно использовать следующим образом:

byte\_8 byte;

byte\_8\* byte\_ptr;

Или, такой пример. Пусть у нас в программе объявлено пространство имен с определением функции и структуры:

namespace firstSpace {

    void foo()

    {

        cout << "function from firstSpace: foo()" << endl;

    }

    struct point {

        double x, y;

    };

}

Тогда для типа firstSpace::point можно создать псевдоним следующим образом:

using point2D = firstSpace::point;

и использовать его для объявления соответствующей переменной на структуру:

point2D pt;

При этом прежний тип firstSpace::point, конечно же продолжает существовать. Мы лишь создаем еще одно имя этого типа не более того. И using создает псевдонимы именно для типов данных. Например, запись вида:

using func = firstSpace::foo;

приведет к ошибке, т.к. foo – это имя функции, а не тип данных.

Вообще конструкция

using <alias> = <тип данных>;

очень напоминает оператор typedef языка Си. Например, с его помощью мы также можем записать:

typedef unsigned char byte\_8;

typedef firstSpace::point point2D;

На первый взгляд никаких отличий. Но они все же имеются. Оператор using полностью покрывает функциональность оператора typedef и привносит некоторые дополнительные возможности и улучшения. Например, объявление типа указателя на функцию через typedef выглядит так:

typedef float (\*func\_ptr)(int);

а с использованием using несколько понятнее и красивее:

using func\_ptr = float (\*)(int);

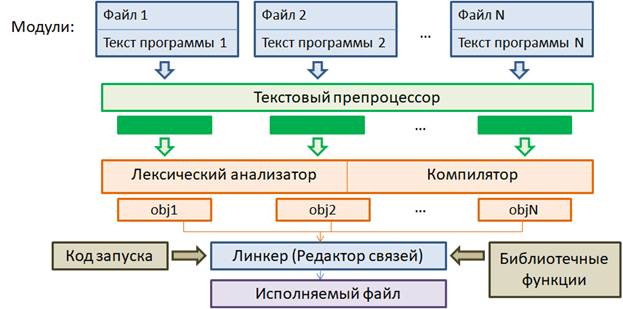
Но, конечно, главное преимущество using перед typedef проявляются при работе с шаблонами (templates). Но это уже выходит за пределы нашего базового курса по С++. Отмечу лишь, что в современных программах на С++ нет смысла использовать typedef и от него лучше отказываться в пользу оператора using. Хотя это не строгое правило и применение typedef все же допустимо.

Компиляция программы: исходный и исполняемый код, виды ошибок.

## 

## Этапы перевода (трансляции) текстов программы в исполняемый код

Сначала каждый модуль независимо пропускается через текстовый препроцессор. Его задача в программе найти все, так называемые, директивы (указания) для этого препроцессора и выполнить их. В результате, исходный текст программы несколько меняется. После этого, преобразованные тексты подаются на компилятор (также независимо друг от друга), которые сначала проходят через лексический анализ программы. На этом этапе выделяются возможные синтаксические ошибки. Если ошибок не обнаружено, то программа далее переводится непосредственно в машинные коды. На выходе получаются объектные файлы модулей. Но в этих объектных файлах отсутствуют связи с другими модулями (если они были прописаны в тексте программы), реализации библиотечных функций (если они были использованы), код запуска всей программы. Все это делает на последнем этапе линкер (по-простому, редактор связей). Он связывает все объектные файлы модулей в единый исполняемый файл, добавляет в него необходимые реализации библиотечных функций и код запуска для текущей операционной системы. На выходе получается окончательный результат в виде исполняемого файла.



Линкер добавляет в итоговые исполняемые файлы только те реализации библиотечных функций, которые используются в программе. Ничего дополнительного, лишнего на выходе не образуется.

Использование компиляторов для разных архитектур процессоров и разных ОС позволяет относительно просто и быстро переносить ранее написанную программу с одной ОС на другую, или с одной архитектуры процессора на другую. И, так как компиляторы языка Си реализованы практически везде, на всех платформах, то программа на языке Си оказывается самой переносимой среди многих других языков высокого уровня.

## Стандарты языка Си

Конечно, компиляторы языка Си пишутся разными людьми в разных фирмах. А, значит, они и работают с некоторыми отличиями. В то же время, программист на Си не должен задумываться об особенностях этих компиляторов и программы, написанные разными программистами, должны корректно переводиться (транслироваться) в машинный код разными компиляторами. Добиться этого можно только стандартизацией. Программисты по всему миру должны договориться между собой о синтаксисе и наборе команд, используемых в языке Си.

Первым таким неформальным стандартом стала книга Брайана Кернигана и Денниса Ритчи «Язык программирования Си». На первых этапах становления языка Си этого было достаточно.

### Стандарт ANSI C (ISO C)

Но по мере того, как все больше и больше программистов по всему миру стали использовать этот язык программирования, остро встал вопрос создания вполне официального стандарта, который бы, к тому же, включал все полезные новшества этого нового языка. С этой целью в 1983 году национальный институт стандартизации США – ANSI (American National Standards Institute) образовал специальную рабочую группу (комитет) под названием X3J11. И в 1989 году миру был предъявлен новый стандарт, который сейчас часто называют **ANSI C** или **C89**. Правда, официально он был принят только в 1990 году, поэтому вместо C89 иногда используют запись C90. Но это означает, по сути, одно и то же.

По итогам своей работы комитет X3J11 сформулировал несколько довольно важных принципов:

* доверять программисту;
* не мешать программисту делать то, что он считает необходимым;
* без необходимости не усложнять язык, сохранять его простоту;
* каждая операция языка должна иметь только один способ выполнения;
* операция должна выполняться максимально быстро, даже в ущерб переносимости языка.

Конструкции языка Си, так или иначе, отражают эти принципы, заложенные в самом начале становления этого языка.

### Стандарт C99

Совсем скоро, уже в 1994 году появилась необходимость в создании нового стандарта языка Си. В первую очередь это было обусловлено растущей популярностью самого языка и его применение в самых разных задачах. В частности, возникла необходимость:

* в интернационализации (поддержке различных международных языков) на программном уровне;
* в устранении некоторых неточностей предыдущей версии языка стандарта ANSI C;
* в повышении стабильности математических вычислений для возможности безопасного использования языка в научных проектах.

В 1999 году этот новый стандарт был принят и стал известен под кодовым названием **C99**.

### Другие стандарты

Язык Си стал хорошей базой для развития и создания нового подобного и во многом похожего на него языка программирования C++. Его автором считается Бьёрн Страуструп – сотрудник Bell Laboratories. В этом языке появилась объектно-ориентированная составляющая, шаблоны классов и функций и некоторые другие полезные инструменты для современной разработки программ любого уровня сложности.

Новые стандарты языка Си появляются с завидной регулярностью. И, как вы догадываетесь, этот безудержный процесс, скорее всего, будет продолжаться, пока существует язык Си. Но я остановлюсь на стандарте C99, принятом в 1999-м году. На мой взгляд, он удачно сочетает возможности и красоту языка Си, а также сохраняет его дух, заложенный создателем Деннисом Ритчи в далеком 1972 году.

1. Консольный ввод/вывод данных.

Заголовочный файл iostream.

Ввод/вывод данных на С++ организован с использованием потоков. Поток – это

абстрактное понятие, относящееся к любому переносу данных от источника к приемнику.

Потоки C++ обеспечивают надежную работу как со стандартными, так и с определёнными

пользователем типами данных, а также единообразный и понятный синтаксис.

В С++ реализованы следующие виды потоков:

а) стандартные – для передачи данных от клавиатуры и на экран дисплея;

б) файловые – для обмена информацией с файлами на внешних носителях данных;

в) строковые – для работы с массивами символов в оперативной памяти.

Рассмотрим работу со стандартными потоками ввода/вывода. Для этого в программе

требуется подключение библиотеки iostream (см. 2-ую строку кода в п. «Простейшая

программа на С++»). Эта библиотека содержит стандартные объекты ввода/вывода (cin и

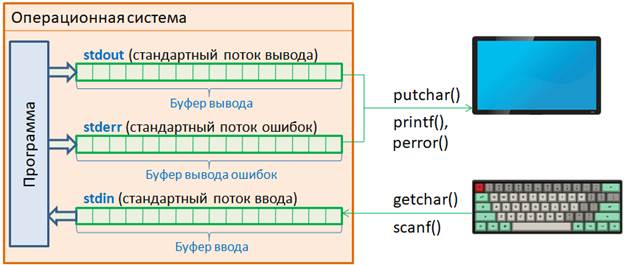
cout) и операции для работы с ними.

Понятие потока ввода/вывода.

На этом занятии затронем тему стандартных потоков ввода/вывода. Вначале ответим на вопрос, что это за потоки и для чего они нужны?

Я думаю, вы все прекрасно понимаете, что программы часто выводят результаты своей работы на монитор, а считывают информацию с клавиатуры. Хотя это и не обязательно так, но чаще всего мы имеем дело именно с монитором и клавиатурой. Так вот, на уровне операционной системы, как правило, имеются три стандартных потока ввода/вывода:

* stdout – поток вывода информации (как правило, на монитор);
* stderr – поток вывода ошибок (как правило, на монитор);
* stdin – поток ввода информации (как правило, с клавиатуры).



Конечно, все эти потоки можно настроить на любые другие устройства. Например, в некоторых случаях поток ошибок stderr ассоциируется с самописцем (принтером), который сразу на бумаге выдает ошибки, произошедшие в процессе работы программы. А поток вывода stdout можно связать с файлом, куда будет выводиться вся информация. Аналогично поток ввода stdin. Вместо клавиатуры можно использовать любое устройство ввода, вплоть до магнитных лент, информация с которых в битовом представлении поступает в поток ввода. То есть, монитор и клавиатура – это всего лишь частные, хотя и частые случаи. Благодаря использованию стандартных потоков ввода/вывода, программы способны универсальным образом работать с любыми устройствами, связанными с этими потоками. И это очень удобно.

На программном уровне все эти потоки организованы в виде буферов приема или передачи информации. То есть, данные сначала поступают в буфер, а затем, уже либо на устройство вывода, либо в переменные программы. Это очень важный момент. Данные поступают в программу, например, с клавиатуры, не напрямую, а через буфер ввода. То же самое с выводом. Сначала данные из программы попадаю в буфер вывода и только потом, например, отображаются на мониторе или записываются в файл. Запомним этот момент. Он нам в будущем пригодится.

## Функции для работы со стандартными потоками

Язык Си предоставляет набор библиотечных функций для работы со стандартными потоками ввода/вывода. Мы рассмотрим некоторые из них, которые наиболее часто используются на практике:

* putchar() – вывод символа через поток stdout;
* printf() – форматный вывод строки через поток stdout;
* perror() – вывод ошибок в виде строки через поток stderr;
* getchar() – чтение одного байта (символа) из потока stdin;
* scanf() – форматный ввод данных из потока stdin.

Описания (прототипы) всех этих функций даны в заголовочном файле **stdio.h**. То есть, для их использования в программе вначале должна быть прописана директива:

#include <stdio.h>

## Функция getchar()

Давайте рассмотрим эти функции и начнем с getchar(). Она имеет следующее определение:

int getchar(void);

Целочисленный тип int перед функцией означает, что она возвращает целое число, а void в круглых скобках говорит об отсутствии каких-либо параметров. Поэтому мы можем вызвать ее в программе следующим образом:

#include <stdio.h>

int main(void)

{

    int value = [getchar](http://www.opengroup.org/onlinepubs/009695399/functions/getchar.html)();

    return 0;

}

После запуска этой программы будет ожидаться ввод значения в поток stdin. Так как он по умолчанию ассоциирован с клавиатурой, то нам нужно набрать один символ, любой, например, буквы ‘s’ и нажать клавишу Enter. После этого программа продолжится, перейдет к следующему оператору return и завершится.

Я напомню, что функция getchar() предназначена для чтения одного байта из входного потока stdin. И здесь возникает вопрос, почему эта функция возвращает целое число типа int, а не данные типа char, который и описывает один байт памяти? Дело в том, что стандартный поток ввода работает несколько более сложным образом, нежели просто выдача очередного байта из буфера. В частности, он дополнительно генерирует некоторые служебные значения. Например, значение:

EOF (End of File – конец файла)

которое определено как -1 в заголовочном файле stdio.h. То есть, помимо байтового диапазона [0; 255] целых чисел функция getchar() дополнительно может вернуть значения вне этого диапазона, в частности, -1.

Вам может показаться странным, что мы говорим про константу EOF, когда речь идет о вводе данных с клавиатуры? Но, во-первых, стандартный поток ввода stdin вполне можно связать с файлом и тогда данные будут читаться из него, а не с клавиатуры и при достижении конца файла будет сгенерировано значение EOF. И, во-вторых, при вводе с клавиатуры мы также можем симитировать достижение конца файла путем ввода специального символа комбинацией клавиш Ctrl+Z для ОС Windows и Ctrl+D для ОС Linux.

Таким образом, функции getchar() нужно возвращать целые значения, превышающий байтовый диапазон [0; 255]. Поэтому разработчик языка Си решил использовать тип int.

## Функция putchar()

Следующая аналогичная функция – это putchar(), которая служит для вывода одного байта (символа) в выходной поток stdout и определена следующим образом:

int putchar(int ch);

Она также возвращает целое число типа int и в качестве аргумента принимает целое значение этого же типа int. В действительности, тип int здесь использован для сопряжения (по типам данных) с функцией getchar(). Иначе бы можно было прописать тип char, так как функция putchar() в качестве аргумента принимает код символа в диапазоне [0;255]. Любое другое значение за пределами этого диапазона просто будет приводиться к восьми битам и затем помещаться в выходной поток stdout. Возвращает эта функция код символа, переданного в выходной поток:

#include <stdio.h>

int main(void)

{

    int value = [getchar](http://www.opengroup.org/onlinepubs/009695399/functions/getchar.html)();

    int res = [putchar](http://www.opengroup.org/onlinepubs/009695399/functions/putchar.html)(value);

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("**\n**%d**\n**", res);

    return 0;

}

При выполнении этой программы, нам необходимо будет ввести какой-либо символ с клавиатуры, и затем, он продублируется вызовом функции putchar(). Возвращаемое значение (код введенного символа) будет выведено на экран с помощью функции printf().

Конечно, на практике функцию putchar() обычно вызывают исключительно для вывода информации в стандартный поток stdout. Поэтому возвращаемое значение просто игнорируют:

[putchar](http://www.opengroup.org/onlinepubs/009695399/functions/putchar.html)(value);

То есть, если функция что-либо возвращает, нет необходимости в программе учитывать это значение. В этом случае говорят, что **функция вызвана ради побочного эффекта**. Надо сказать, что в языке Си это обычная практика.

## Буферы приема/передачи стандартных потоков

На протяжении всего занятия я акцентрирую ваше внимание на наличие буферов приема/передачи информации у стандартных потоков ввода/вывода. При запуске программы они пустые, в них нет никаких посторонних значений. Но, в процессе ввода или вывода информации они заполняются и это может повлиять на ход исполнения программы.

Давайте я это покажу на конкретном примере. Запишем два подряд идущих вызова функции getchar() следующим образом:

#include <stdio.h>

int main(void)

{

    int value1 = [getchar](http://www.opengroup.org/onlinepubs/009695399/functions/getchar.html)();

    int value2 = [getchar](http://www.opengroup.org/onlinepubs/009695399/functions/getchar.html)();

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("%c %c**\n**", value1, value2);

    return 0;

}

И после запуска этой программы введем с клавиатуры два символа: ds. В результате, оба символа помещаются во входной буфер, первый считывается при первом вызове функции getchar(), а второй – при втором вызове функции getchar(). Поэтому программа не ждет от нас ввода какой-либо дополнительной информации, а сразу переходит к функции printf(). Соответственно в переменной value1 будет храниться код символа d, а в переменной value2 – код символа s. Затем, функция printf() выводит на экран оба прочитанных символа.

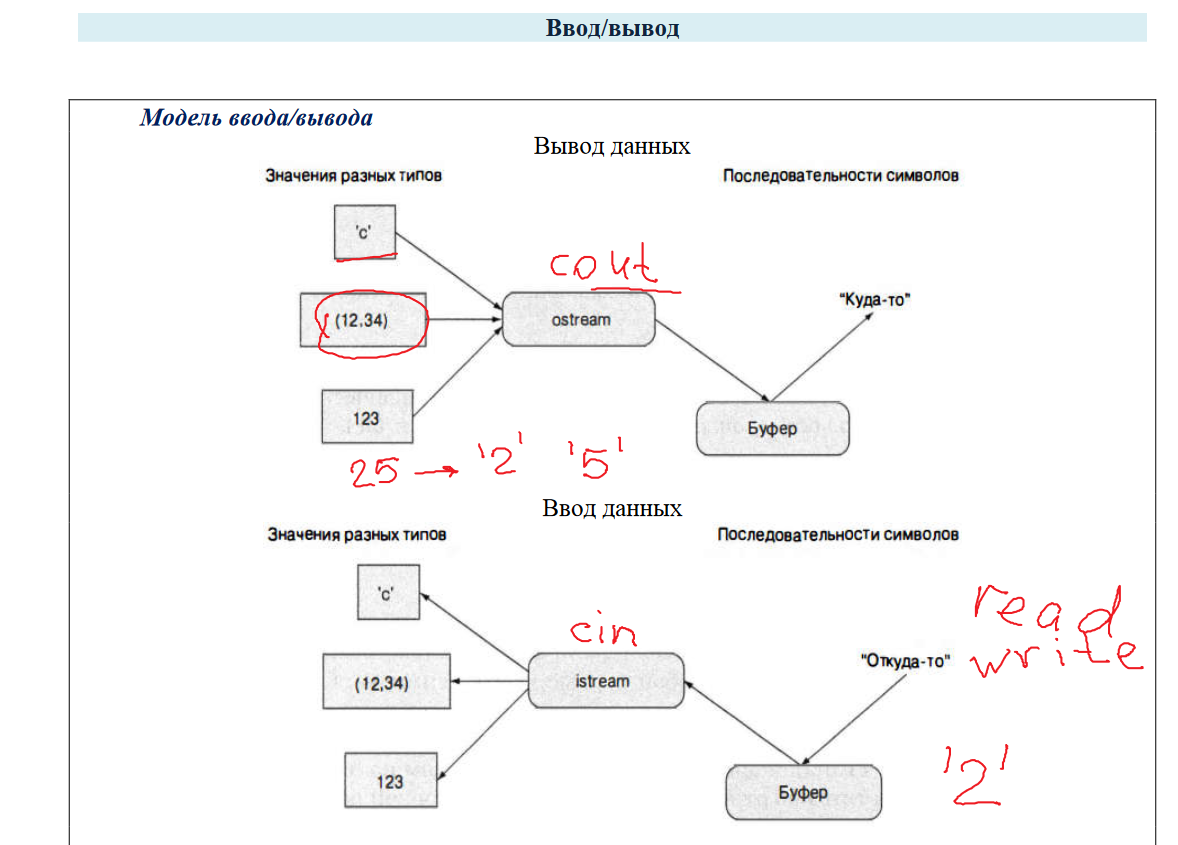
Вот наглядный эффект работы входного буфера. Мало того, если бы мы ввели не два, а, скажем, три символа, то после чтения первых двух, последний так бы и остался во входном буфере до момента завершения программы. При завершении, все буферы автоматически очищаются.

На этом мы завершим вводное занятие по стандартным потокам ввода/вывода, а также по функциям getchar() и putchar(). На следующем занятии продолжим эту тему и поговорим о функции printf().

Понятие буфера.

это область [памяти](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D0%B0%D0%BC%D1%8F%D1%82%D1%8C), используемая для временного хранения данных при вводе или выводе. Обмен данными (ввод и вывод) может происходить как с внешними устройствами, так и с [процессами](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81_(%D0%B8%D0%BD%D1%84%D0%BE%D1%80%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)) в пределах [компьютера](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80). Буфера могут быть реализованы в [аппаратном](https://ru.wikipedia.org/w/index.php?title=%D0%90%D0%BF%D0%BF%D0%B0%D1%80%D0%B0%D1%82%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5&action=edit&redlink=1) или [программном обеспечении](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B5_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D0%B5), но подавляющее большинство буферов реализуется в программном обеспечении. Буфера используются, когда существует разница между скоростью получения данных и скоростью их обработки или в случае, когда эти скорости переменны, например, при буферизации печати.

Объекты cin и cout, операторы << и >>.

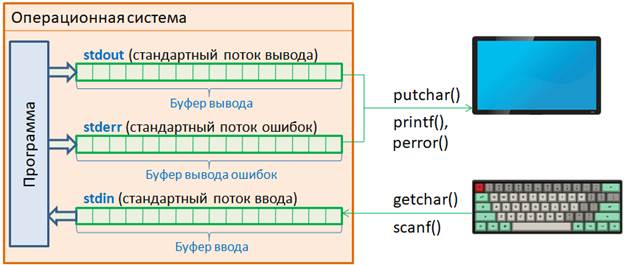


давайте сразу разберем работу двух таких объектов:

* cin – объект класса istream для работы с потоком stdin;
* cout – объект класса ostream для работы с потоком stdout.

На данном этапе будет вполне достаточным воспринимать объекты cin/cout, как некие переменные, через которые можно работать со стандартными потоками ввода/вывода.

Я напомню, что в курсе по языку Си мы подробно с вами говорили об этих потоках, которые часто обозначаются через определения stdin/stdout. И я приводил следующую схему их работы:



Так вот, в С++ вместо функций printf(), scanf() и им подобных предлагается использовать более дружественные объекты cout и cin. В чем их дружественность и отличия от уже знакомых нам функций ввода/вывода? Начнем с более простого объекта cout. Чтобы им воспользоваться, в программе на С++ вначале необходимо подключить файл iostream. После этого в пространстве имен std будет доступен объект cout. Затем, объявим в программе несколько переменных разных типов и выведем их в стандартный поток с помощью cout. Сделать это можно следующим образом:

#include <iostream>

int main()

{

    char str[100] = "Hi, Sergey!";

    short old = 99;

    double weight = 82.54;

    std::cout << str << "**\n**";

    std::cout << old << "**\n**";

    std::cout << weight << std::endl;

}

Первое, что бросается в глаза – это вывод значения переменной без дополнительного указания ее типа. Например, если бы мы эту же операцию захотели бы повторить с помощью функции printf(), то пришлось бы прописывать форматную строку со спецификаторами:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("%s**\n**%d**\n**%.2f**\n**", str, old, weight);

При использовании объекта cout ничего этого делать не нужно. Просто прописываем переменную, значение которой хотим вывести в поток stdout и все.

Если кому интересно, как это все работает «под капотом», то скажу буквально пару слов. У каждого класса в С++ можно делать переопределение стандартных операций. В частности, операция << для класса ostream переопределена для каждого стандартного, базового типа и компилятор в соответствии с типом переменной (или, в общем случае, по данным в правом операнде), выбирает соответствующую реализацию операции <<. Сама же операция << выполняет передачу в выходной поток stdout соответствующих данных и возвращает объект cout. А раз так, то мы можем записать вывод трех переменных следующим образом:

std::cout << str << '**\n**' << old << "**\n**" << weight << std::endl;

Получим абсолютно тот же самый результат.

Второе, на что следует обратить внимание – это использование управляющих символов ‘\n’. Сначала я его прописал, как символ, а второй раз – как строку. Для объекта cout это не имеет большого значения, но, конечно, если нам нужно вывести отдельный символ, то лучше это делать через одинарные кавычки.

В самом конце вывода прописана функция endl. Она выполняет два действия: переводит курсор на новую строку и очищает выходной буфер, указывая конечному устройству, связанному с выходным потоком, что все данные переданы и их можно обрабатывать. В случае с выводом на монитор очистка мало что дает, т.к. информация сразу отображается на экране. Но, вполне могут быть и другие ситуации, когда вместо монитора подключено другое устройство и вывод осуществляется только в момент очистки выходного буфера.

Следующий объект cin работает похожим образом, но осуществляет уже считывание информации из стандартного входного потока stdin и запись прочитанных данных в указанную переменную или, в общем случае, lvalue выражение.

Давайте, для примера выполним считывание данных в переменные old и weight с помощью cin следующим образом:

#include <iostream>

int main()

{

    char str[100] = "Hi, Sergey!";

    short old = 99;

    double weight = 82.54;

    std::cin >> old;

    std::cin >> weight;

    std::cout << str << '**\n**' << old << '**\n**' << weight << std::endl;

}

Обратите внимание, что для объекта cin переопределена операция >> также для каждого стандартного типа языка С++. Поэтому нам достаточно прописать переменную без указания ее типа. Причем после имени переменной не нужно прописывать операцию взятия адреса, как это мы делали в функции scanf(). В операции >> для этого применяется механизм ссылок. О них мы с вами еще будем подробнее говорить.

Если данные введены верно (целочисленное значение и вещественное), то переменные old и weight будут содержать корректные введенные значения. Причем ввести их можно каждый с новой строки или через пробел. Если во входном потоке stdin встречаются данные, не соответствующие типу переменной, то значение переменной обнуляется с неверной интерпретацией последующих операций считывания. Поэтому формат входных данных в stdin должен совпадать с типами читаемых данных.

Операция >> после чтения порции данных возвращает объект cin, поэтому два последовательных вызова cin можно объединить в один:

std::cin >> old >> weight;

Результат будет тем же самым.

Последний важный момент, связанный с использованием объекта cin, это чтение строк из входного потока. Допустим, мы в переменную str читаем строковые данные следующим образом:

std::cin >> str;

и вводим строку «hello world». В результате в массив str будет помещен только первый фрагмент до пробела, то есть, «hello», а второй останется во входном потоке. То есть, cin со строками работает так же, как и функция scanf() языка Си. Если вам нужно читать строку целиком (до символа перевода строки или до конца данных), то следует использовать новую функцию getline() языка С++. Она работает с объектом std::string и речь о ней еще впереди.

Я думаю, вы теперь в целом хорошо понимаете, как использовать объекты cout/cin языка С++ для работы со стандартными потоками ввода/вывода.

Форматированный вывод данных.

Функции форматирования: fill(), width(), precision().

Функции setf() и unsetf() и флаги форматирования (boolalpha, oct, dec, hex, showbase, showpos, scientific, fixed, uppercase, right, left).

Манипуляторы объекта cout.

Ввод данных разных типов с помощью cin. Примеры.

Ввод символов с помощью cin.get(). Примеры.

Ввод строк с помощью cin.getline(). Примеры.

1. Файловый ввод/вывод.

Заголовочный файл fstream.

Последовательность действий при файловом вводе данных.

Последовательность действий при файловом выводе данных.

Контроль ошибок открытия файлов. Функции good(), eof().

Контроль считываемых данных. Функция fail().

Форматированный файловый вывод данных.

Отличия консольного ввода/вывода от файлового.

1. Алгоритмы.

Понятие алгоритма. Базовые структуры алгоритмов: линейная, разветвляющаяся, циклическая.

Запись алгоритмов с помощью граф-схем алгоритмов. ГОСТ 19.701-90 (ISO 5807-85) ЕСПД. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения.

Условный оператор: сокращённая и полная записи, ГСА, синтаксис. Пример программы.

Составной оператор. Операторные скобки.

Оператор сравнения. Рекомендации по предотвращению ошибок и стилю написания условных операторов.

Логические выражения в условных операторах. Альтернативные представления логических операторов.

Выражения отношений. Примеры логических выражений.

Тернарный оператор: назначение, ГСА, синтаксис. Примеры

Связь условного и тернарного операторов. Примеры.

Оператор варианта switch: назначение, ГСА, синтаксис. Пример.

Цикл  параметром for: назначение, ГСА. Пример.

Цикл с предусловием: назначение, ГСА. Пример.

Цикл с постусловием: назначение, ГСА. Пример.

Представление цикла с параметром for через цикл с предусловием. Пример.

Представление цикла с параметром for через цикл с постусловием. Пример.

Исключение зацикливания.

Операторы перехода: return, break, continue. Примеры.

1. Основные понятия языка С++.

Алфавит языка С++.

Лексемы языка С++: определение, типы.

Переменные: объявление, инициализация. Инициализация в стиле С++ (списковая).

Идентификаторы.

Константы: объявление, инициализация.

Зарезервированные слова С++: ключевые слова, альтернативные лексемы, зарезервированные имена библиотек. Их отличительные особенности.

Символьные литералы: структура, значения, операции, функции.

Строковые литералы: структура, значения, операции, функции. Правила конкатенации строк.

Целочисленные литералы: структура, значения, операции, функции.

Числа с плавающей точкой: структура, значения, операции, функции.

1. Вычисления в языке С++.

Операторы в С++: объявления, инициализации, присваивания.

Выражения в С++.

Порядок выполнения и операций: приоритеты операций и ассоциативность.

Арифметические операции. Их особенности.

Преобразования типов в арифметических выражениях.

Логические операции.

Битовые операции.

Комбинация операции присваивания и арифметических операций.

Операция "запятая".

Операции инкремента и декремента. Их постфиксная и префиксная формы записи.

Применимость операций к данным разного типа.

1. Простые типы данных.

Понятие типа данных. Назначение типа данных.

Классификация типов данных.

Базовые и составные типы данных.

Символьный тип данных: назначение, особенности, диапазон представимых данных. Беззнаковый аналог.

Целый тип данных: назначение,  особенности, диапазон представимых данных. Беззнаковый аналог.

Вещественный тип данных: назначение,  особенности, диапазон представимых данных.

Логический тип данных.

Преобразование типов при инициализации: тип с плавающей точкой в меньший тип с плавающей точкой, тип с плавающей точкой в целочисленный тип, целочисленный тип в меньший целочисленный тип. Возможные проблемы таких преобразований. Примеры.

Оператор sizeof.

Заголовочный файл climits.

Заголовочный файл float.

Семейство функций cctype для работы с символами. Их реализация с использованием ASCII-кодов символов. (https://ru.cppreference.com/w/cpp/header/cctype)

Форматированный вывод целых чисел. Примеры.

Форматированный вывод вещественных чисел. Примеры.

Форматированный вывод символов и строк. Примеры.

Явное преобразование типов. Примеры.

Неявное преобразование типов. Примеры.

Префиксы и суффиксы типов данных.

1. Составные типы данных.

Одномерные массивы: определение, объявление, инициализация, особенности. Примеры.

Многомерные массивы: понятие, объявление, инициализация. Примеры.

Строки в стиле С: объявление, инициализация, особенности.

Некоторые функции библиотека cstring для работы со строками в стиле С: strlen(), strcmp(), strcat(), strcpy().

Особенности ввода массивов и строк.

Функции cin.get() и cin.getline().

1. Пользовательские типы данных.

Структуры: назначение, объявление. Примеры использования.

Ввод и вывод данных структурных переменных.

Массивы структур и массивы в структурах.

Битовые поля в структурах.

Перечисления: назначение, объявление. Примеры использования.

Установка значений перечислителей.

Диапазоны значений перечислителей.

Дублирование констант перечислений. Перечисления с областью видимости.

Объединения: назначение, объявление. Примеры использования.

Объявление и инициализация переменных пользовательских типов.

Анонимное объявление пользовательских типов. Особенности использования.

Оператор typedef. Примеры использования.

1. Базовые алгоритмы

Алгоритм поиска максимума и минимума в последовательности чисел, введённых с клавиатуры.

Алгоритм нахождения конечных сумм и произведений.

Алгоритм нахождения бесконечных сумм.

Ввод двухмерного массива чисел по столбцам с нумерацией вводимых столбцов и пропуском избыточных данных.

вывод двухмерного массива чисел с нумерацией строк и столбцов.

Поиск координат минимального числа в двумерном массиве с максимальным размером 10×10.

Анализ задачи и обработка треугольной части элементов двухмерного массива.

Умножение матриц: математическое решение и цикл получения матрицы-произведения.

Вывод тестового файла на экран.

Ввод текстового файла с клавиатуры.

Копирование содержимого текстового файла в другой файл.

Алгоритм удаления лишних пробелов между слов заданного текста.

Алгоритм подсчёта слов в заданном тексте.

Вычисление значения полинома произвольной степени по схеме Горнера.

Преобразование символьной записи целого числа в целый тип данных.

Преобразование целочисленного значения в строковое представление.

Генерация таблицы символов и их ASCII-кодов с помощью всех операторов цикла.

Структура для хранения даты с использованием перечислений для представления месяца.

Создание массива записей с данными о студентах (ФИО, пол, дата рождения) с помощью ввода данных пользователем и вывода информации о них с нумерацией по порядку на экран.