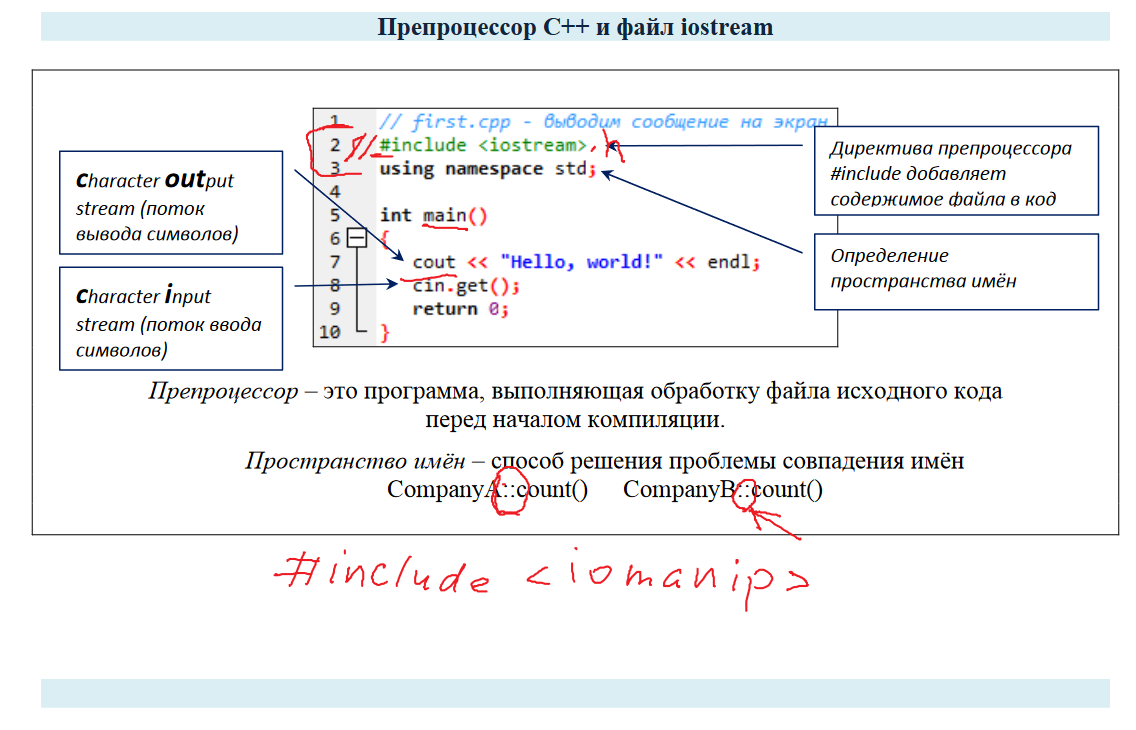
**Вопросы для самоконтроля по дисциплине «Программирование»  
за 1 семестр обучения.**

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.

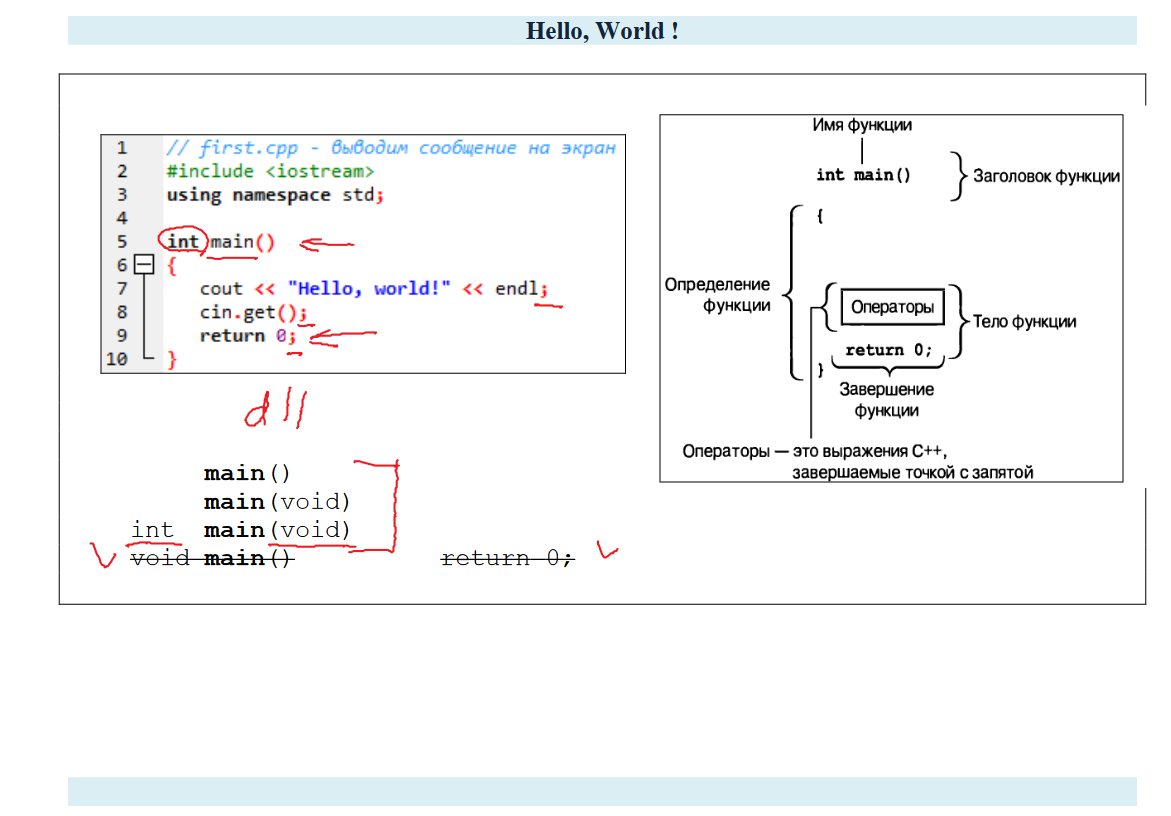
В действительности, эти правила не всегда соблюдаются и компиляторы могут реализовывать свою логику поиска файлов. Мало того, в последних стандартах языка Си алгоритм поиска файлов несколько изменен и стал таким неопределенным, что разработчики компиляторов этот пункт часто просто игнорируют. Благо, 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:

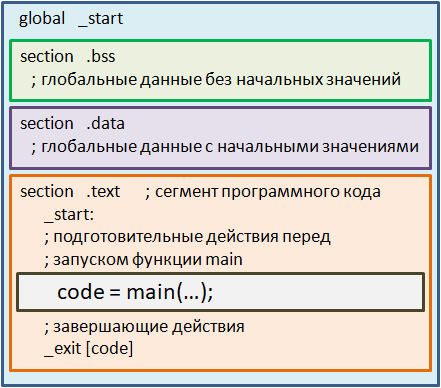
[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Hello, World**\n**");   *// вывод строки на экран*

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.



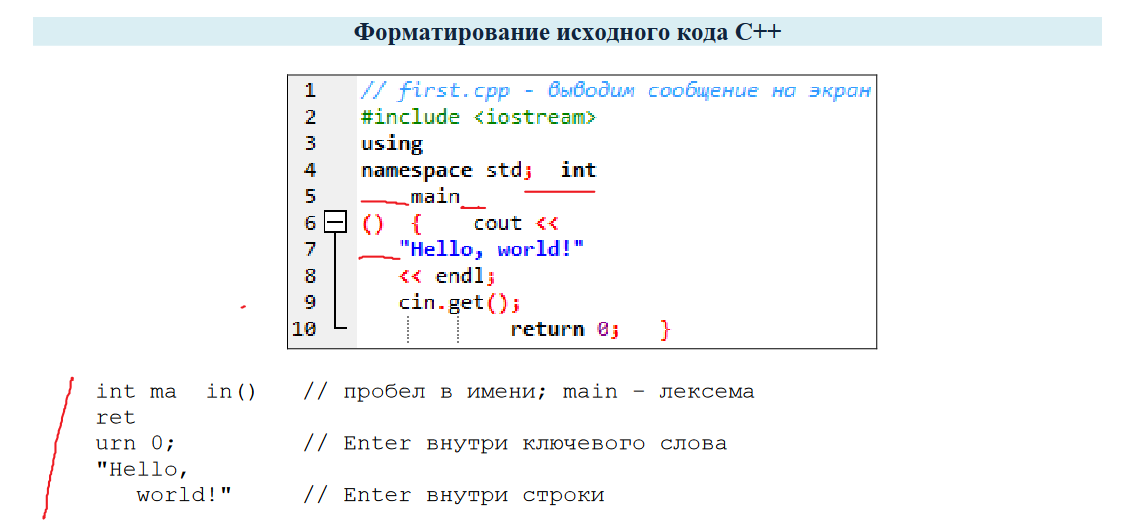
Итак, в программе на языке Си обязательно должна присутствовать функция с именем main, которой передается управление после запуска программы и выполнения некоторых подготовительных действий. Условно этот процесс показан на рисунке.

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;

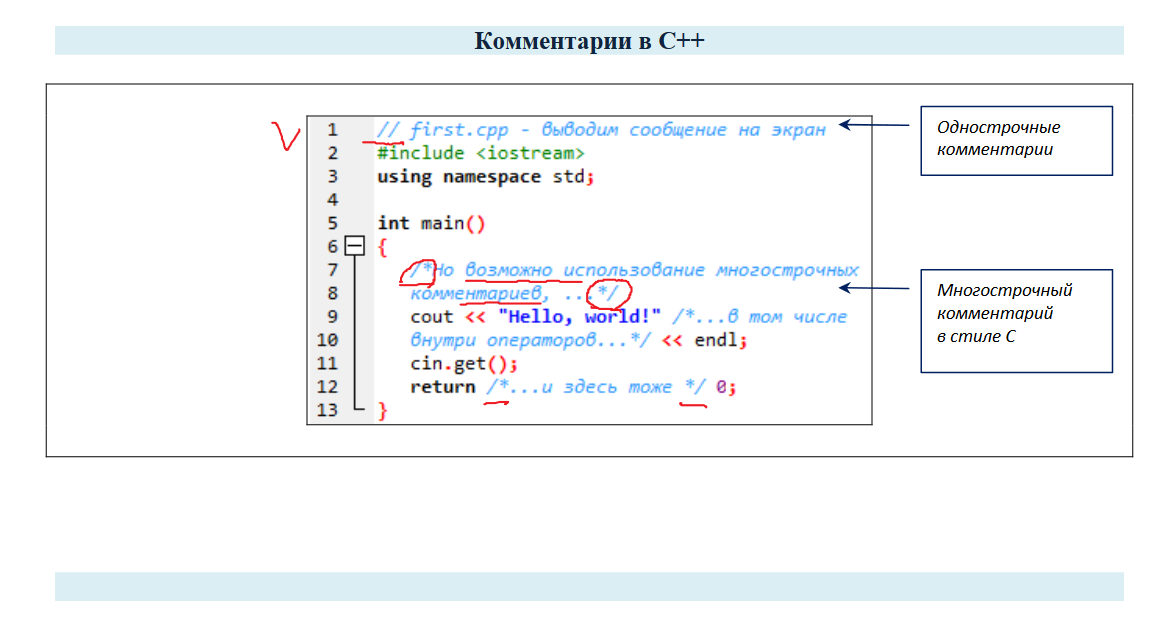
– вокруг круглых скобок, связанных с именем функции, пробельные символы

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;

}

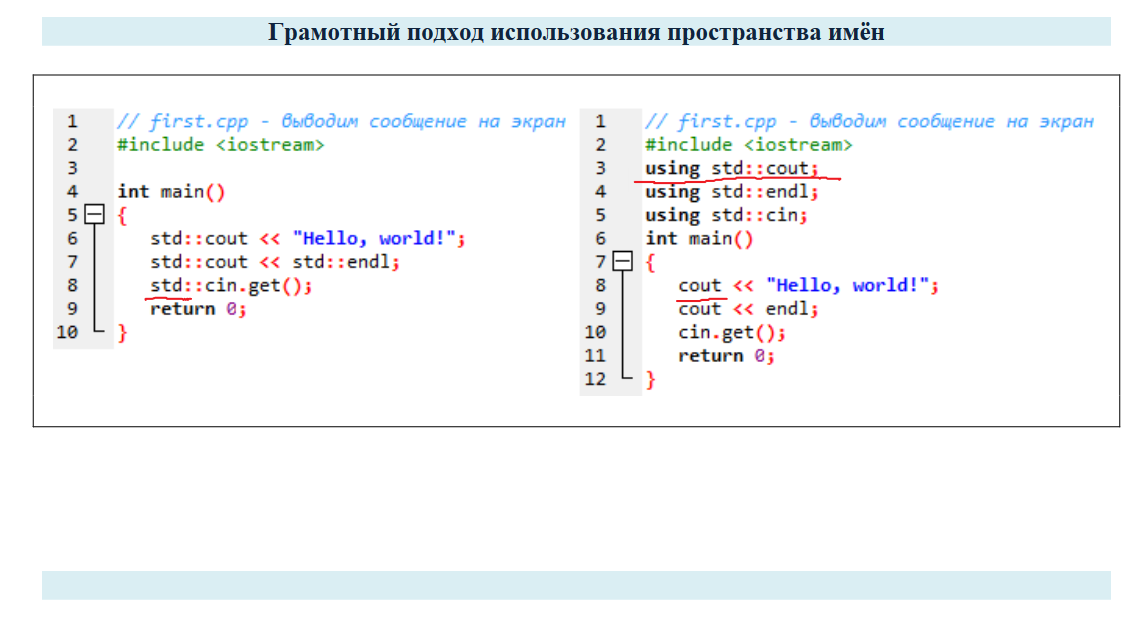
Часто, далеко не все, что прописано в std, используется в текущем модуле. И программист вполне может случайно объявить переменную или функцию или что-либо еще с тем же именем, что и импортированный элемент. Возникнет конфликт имен и компиляция программы завершится с ошибкой. Как раз чтобы этого избежать,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

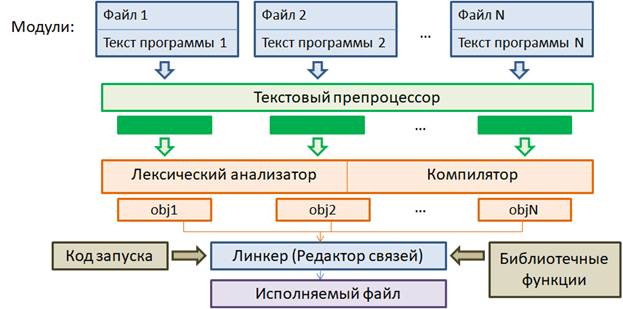
Но, конечно, главное преимущество using перед typedef проявляются при работе с шаблонами (templates). Но это уже выходит за пределы нашего базового курса по С++. Отмечу лишь, что в современных программах на С++ нет смысла использовать typedef и от него лучше отказываться в пользу оператора using. Хотя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

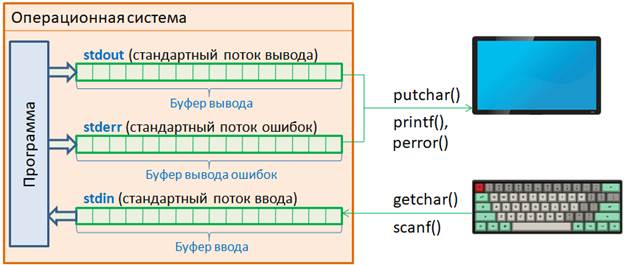
cout) и операции для работы с ними.

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.

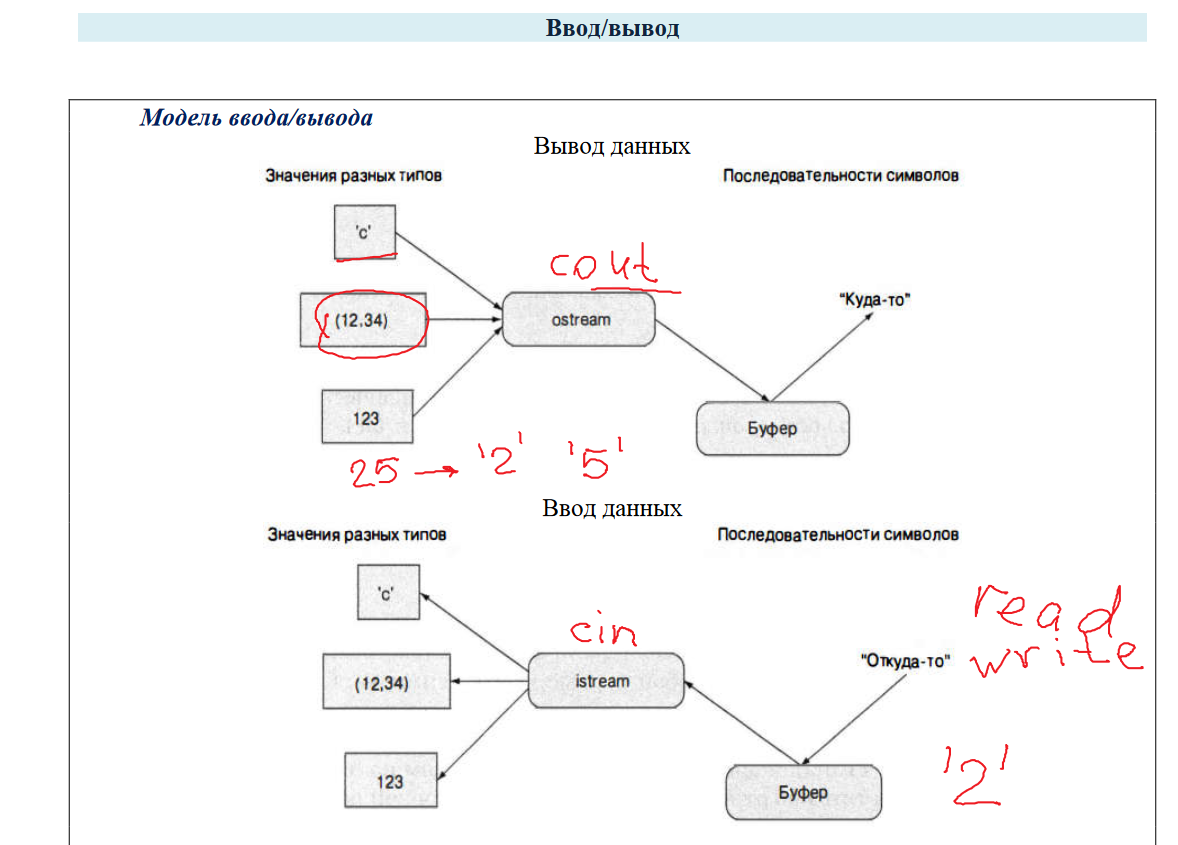
Вот наглядный эффект работы входного буфера. Мало того, если бы мы ввели не два, а, скажем, три символа, то после чтения первых двух, последний так бы и остался во входном буфере до момента завершения программы. При завершении, все буферы автоматически очищаются.

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 >>.

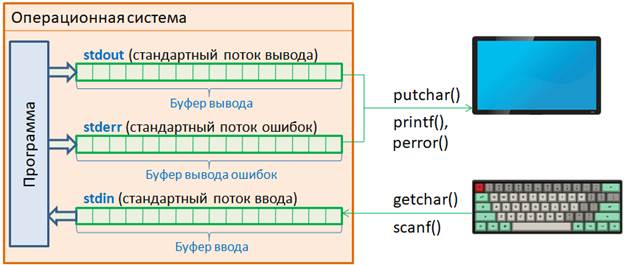


давайте сразу разберем работу двух таких объектов:

* 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().

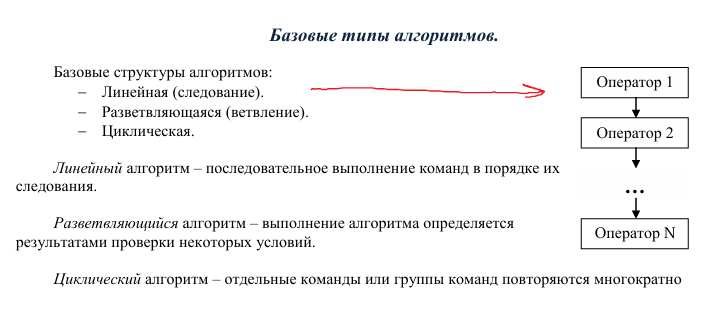
Контроль считываемых данных. Функция fail().

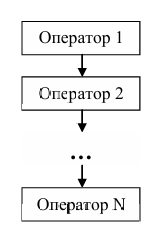
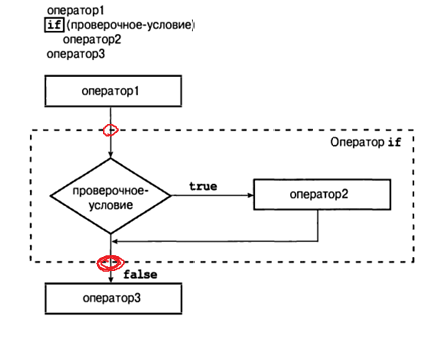
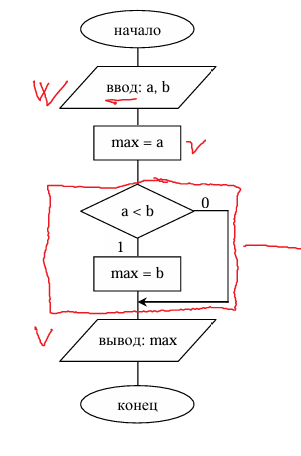
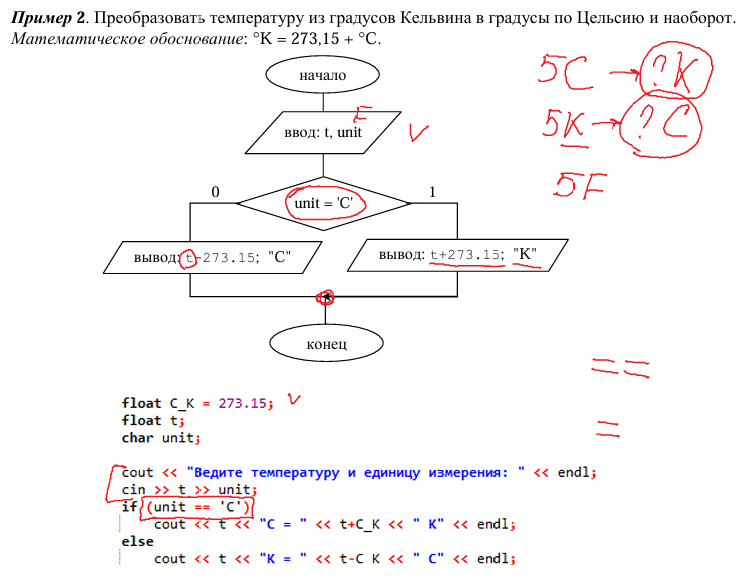
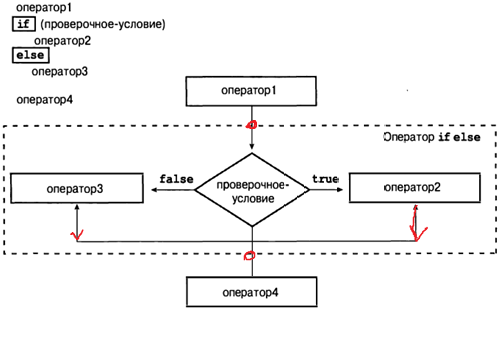
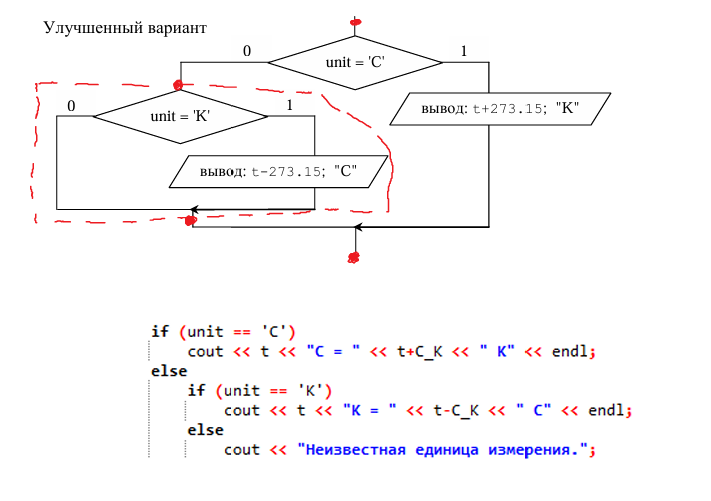
Форматированный файловый вывод данных.

Отличия консольного ввода/вывода от файлового.

1. Алгоритмы.

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



Алгоритмы бывают: линейные, разветвляющиеся, циклические. Линейный алгоритм не содержит логических условий, имеет одну ветвь обработки и изображается линейной последовательностью связанных друг с другом блоков. Разветвляющийся алгоритм содержит одно или несколько логических условий и имеет несколько ветвей обработки. Разветвляющиеся алгоритмы могут иметь несколько структур:

- неполная альтернатива, обработка производится при выполнении условия в противном случае обработка не производится;

- полная альтернатива, обработка производится при выполнении условия по ветви 1, в противном случае по ветви 2;

- конструкция выбора, обработка производится при выполнении одного из нескольких различных условий по соответствующей ему ветви.

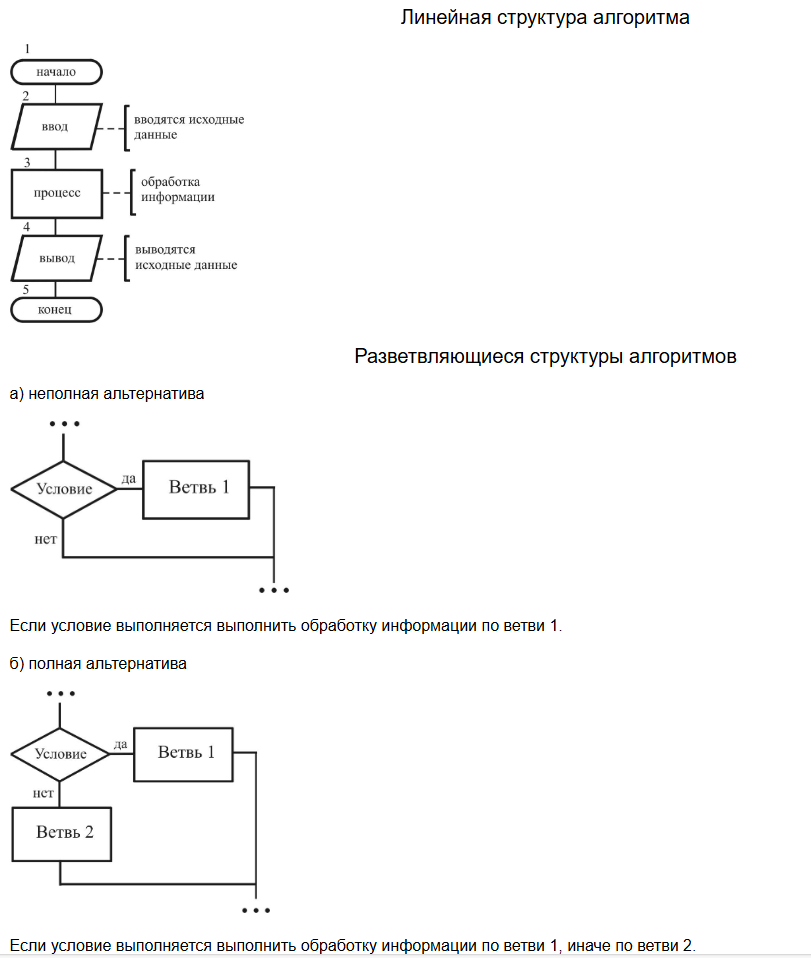
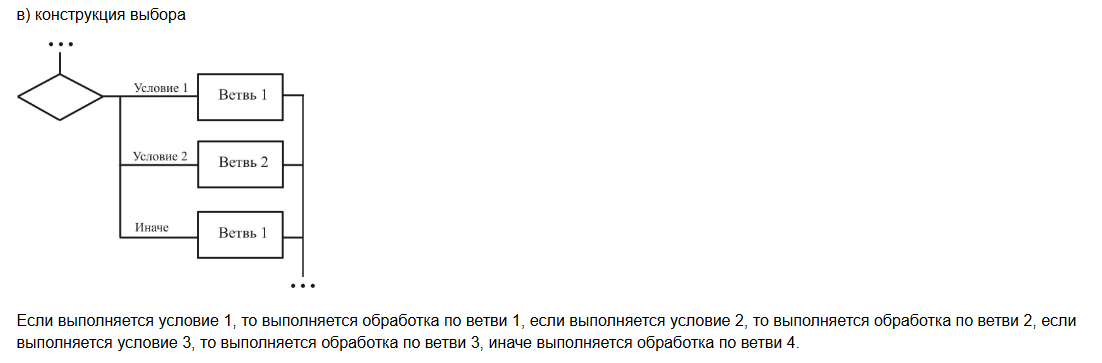
Запись алгоритмов с помощью граф-схем алгоритмов. ГОСТ 19.701-90 (ISO 5807-85)

Блок - Решение имеет один вход и несколько выходов, которые следует показывать:

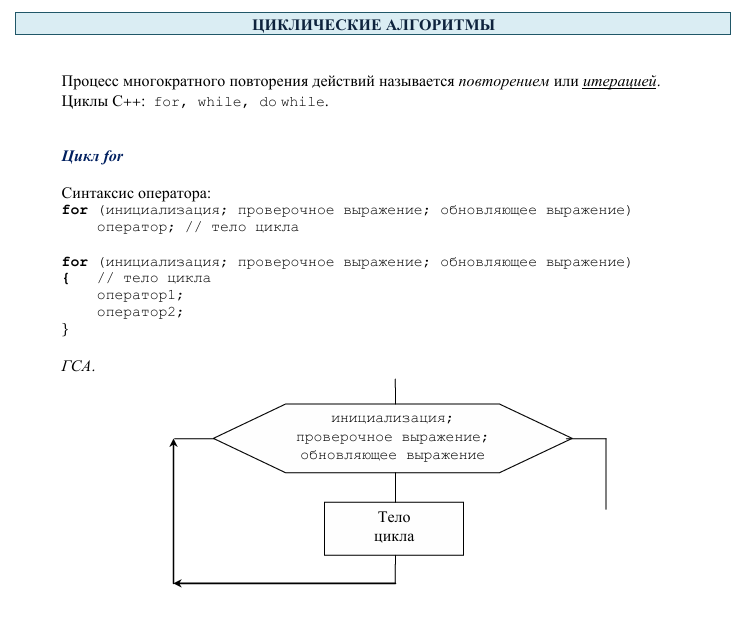
1) несколькими линиями от данного символа к другим символам;

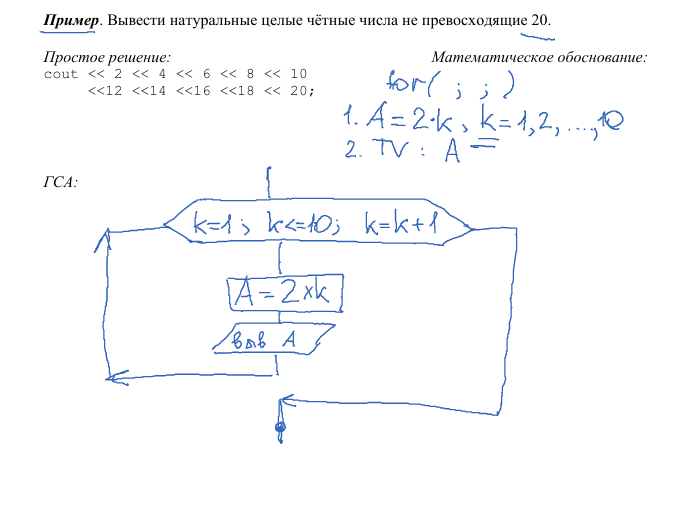
2) одной линией от данного символа, которая затем разветвляется в соответствующее число линий.

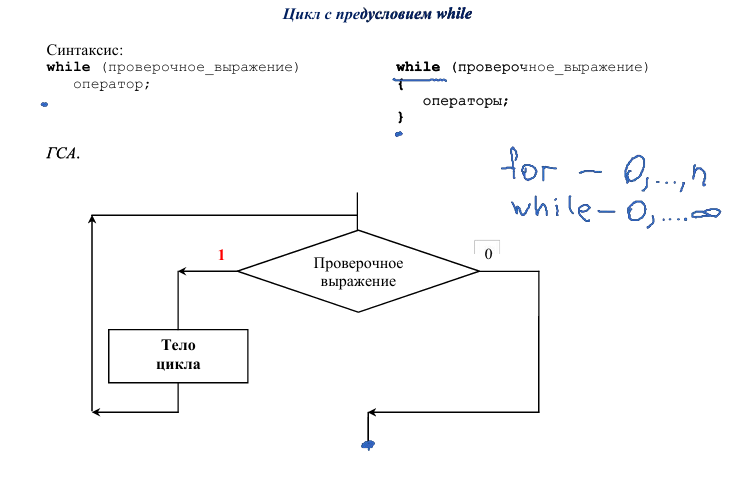
3) каждый выход из символа должен сопровождаться соответствующими значениями условий, чтобы показать логический путь, который он представляет, с тем, чтобы эти соответствующие ссылки были идентифицированы.

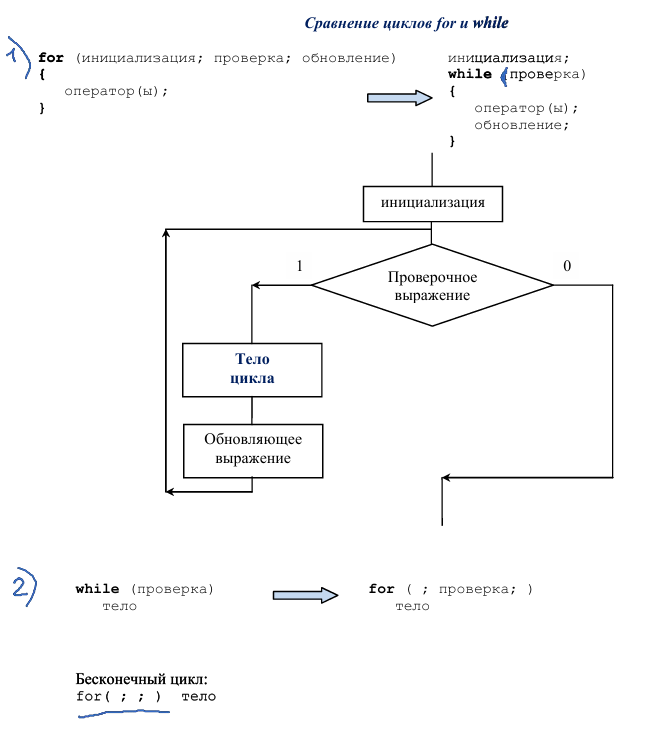
 

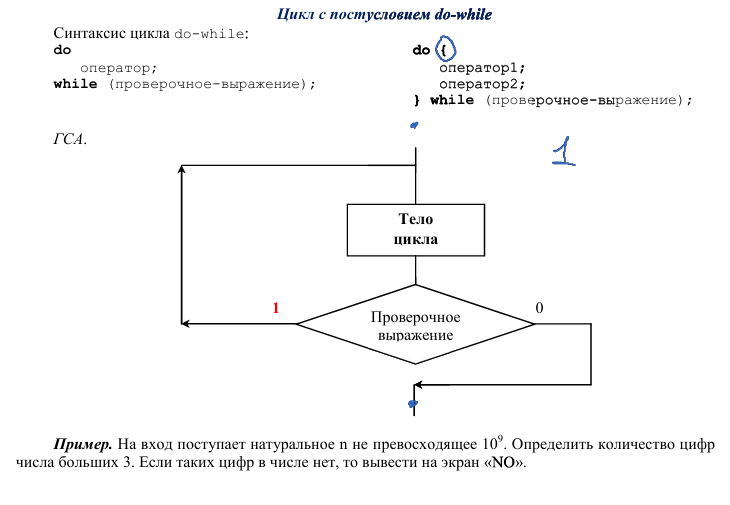
Циклические алгоритмы











ЕСПД. Схемы алгоритмов, программ, данных и систем. Условные обозначения и правила выполнения.

|  |
| --- |
| **1. Символы данных** |
|  | **1.1. Основные символы данных** |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-x9AKww.jpg | 1.1.1. Данные. Символ отображает данные, носитель данных не определен. |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-rte8Ht.jpg | 1.1.2. Запоминаемые данные. Символ отображает хранимые данные в виде, пригодном для обработки, носитель данных не определен. |
|  | **1.2. Специфические символы данных** |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-t9p00T.jpg | 1.2.1. Оперативное запоминающее устройство. Символ отображает данные, хранящиеся в оперативном запоминающем устройстве. |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-epRvUo.jpg | 1.2.2. Запоминающее устройство с последовательным доступом. Символ отображает данные, хранящиеся в запоминающем устройстве с последовательным доступом (магнитная лента, кассета с магнитной лентой, магнитофонная кассета). |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-_XO88g.jpg | 1.2.3. Запоминающее устройство с прямым доступом. Символ отображает данные, хранящиеся в запоминающем устройстве с прямым доступом (магнитный диск, магнитный барабан, гибкий магнитный диск). |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-s3M6ul.jpg | 1.2.4. Документ. Символ отображает данные, представленные на носителе в удобочитаемой форме (машинограмма, документ для оптического или магнитного считывания, микрофильм, рулон ленты с итоговыми данными, бланки ввода данных). |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-5AU6F7.jpg | 1.2.5. Ручной ввод. Символ отображает данные, вводимые вручную во время обработки с устройств любого типа (клавиатура, переключатели, кнопки, световое перо, полоски со штриховым кодом). |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-KSrznI.jpg | 1.2.6. Карта. Символ отображает данные, представленные на носителе в виде карты (перфокарты, магнитные карты, карты со считываемыми метками, карты со сканируемыми метками). |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-QYKEpr.jpg | 1.2.7. Бумажная лента. Символ отображает данные, представленные на носителе в виде бумажной ленты. |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-F2dt6e.jpg | 1.2.8. Дисплей. Символ отображает данные, представленные в человекочитаемой форме на носителе в виде отображающего устройства (экран для визуального наблюдения, индикаторы ввода информации). |
|  | **2. Символы процесса** |
|  | **2.1. Основные символы процесса** |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-9FuHPx.jpg | 2.1.1. Процесс. Символ отображает функцию обработки данных любого вида (выполнение определенной операции или группы операций, приводящее к изменению значения, формы или размещения информации или к определению, по которому из нескольких направлений потока следует двигаться). |
|  | **2.2. Специфические символы процесса** |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-mIXLEt.jpg | 2.2.1. Предопределенный процесс. Символ отображает предопределенный процесс, состоящий из одной или нескольких операций или шагов программы, которые определены в другом месте ( в подпрограмме, модуле). |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-idny0g.jpg | 2.2.2. Ручная операция. Символ отображает любой процесс, выполняемый человеком. |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-YcvES1.jpg | 2.2.3. Подготовка. Символ отображает модификацию команды или группы команд с целью воздействия на некоторую последовательную функцию (установка переключателя, модификация индексного регистра или инициализация программы). |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-hZCkhj.jpg | 2.2.4. Решение. Символ отображает решение или функцию переключаемого типа, имеющую один вход и ряд альтернативных выходов, один из которых может быть активизирован после вычисления условий, определенных внутри этого символа. |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-JZPGFn.jpg | 2.2.5. Параллельные действия. Символ отображает синхронизацию двух или более параллельных операций. |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-Jgcn8J.jpg | 2.2.6. Граница цикла. Символ, состоящий из двух частей, отображает начало и конец цикла. Обе части символа имеют один и тот же идентификатор. Условия для инициализации, приращения, завершения и т.д. помещаются внутри символа в начале или конце в зависимости от расположения операции, проверяющей условие. |
|  | **3. Символы линий** |
|  | **3.1. Основной символ линий** |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-B7W20s.jpg | 3.1.1. Линия. Символ отображает поток данных или управления. |
|  | **3.2. Специфические символы линий** |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-4VRamm.jpg | 3.2.1. Передача управления. Символ отображает непосредственную передачу управления от одного процесса к другому, иногда с возможностью прямого возвращения к инициирующему процессу после того, как инициированный процесс завершит свои функции. Тип передачи управления должен быть назван внутри символа (например, запрос, вызов, событие). |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-cfD3yp.jpg | 3.2.2. Канал связи. Символ отображает передачу данных по каналу связи. |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-84bKAU.jpg | 3.2.3. Пунктирная линия. Символ отображает альтернативную связь между двумя или более символами. Кроме того, символ используют для обведения аннотированного участка. |
|  | **4. Специальные символы** |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-E8_f0w.jpg | 4.1. Соединитель. Символ отображает выход в часть схемы и вход из другой части этой схемы и используется для обрыва линий и продолжения ее в другом месте. Соответствующие символы-соединители должны содержать одно и то же уникальное обозначение. |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-tjGbRY.jpg | 4.2. Терминатор. Символ отображает выход во внешнюю среду и вход из внешней среды (начало или конец схемы программы, внешнее использование и источник или пункт назначения данных). |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-Pt5yQS.jpg | 4.3. Комментарий. Символ используют для добавления описательных комментариев или пояснительных записей в целях объяснения или примечаний. |
| https://studfile.net/html/760/278/html_vsmsDp2GFp.DtC1/img-yR8hsZ.jpg | 4.4. Пропуск. Символ (три точки) используют схемах для отображения пропуска символа или группы символов, в которых не определены ни тип, ни число символов. Символ используют только в символах линий или между ними. Он применим равным образом в схемах, изображающих общие решения с неизвестным числом повторений. |

ПРИМЕНЕНИЕ СИМВОЛОВ

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Символ | Наименование символа | 1 | 2 | 3 | 4 | 5 |
| Символы данных |  |  |  |  |  |  |
| Основные | Данные | + | + | + | + | + |
|  | Запоминаемые данные | + | - | + | + | + |
|  | Специфические ОЗУ | + | - | + | + | + |
|  | ЗУ с послед. выборкой | + | - | + | + | + |
|  | ЗУ с прямым доступом | + | - | + | + | + |
|  | Документ | + | - | + | + | + |
|  | Ручной ввод | + | - | + | + | + |
|  | Карта | + | - | + | + | + |
|  | Бумажная лента | + | - | + | + | + |
|  | Дисплей | + | - | + | + | + |
|  |  |  |  |  |  |  |
| Символы процесса |  |  |  |  |  |  |
| Основные | Процесс | + | + | + | + | + |
| Специфические | Предопределенный процесс | - | + | + | + | - |
|  | Ручная операция | + | - | + | + | - |
|  | Подготовка | + | + | + | + | - |
|  | Решение | - | + | + | - | - |
|  | Параллельные действия | - | + | + | + | - |
|  | Граница цикла | - | + | + | - | - |
|  |  |  |  |  |  |  |
| Символы линий |  |  |  |  |  |  |
| Основные | Линия | + | + | + | + | + |
| Специфические | Передача управления | - | - | - | + | - |
|  | Канал связи | + | - | + | + | + |
|  | Пунктирная линия | + | + | + | + | + |
| Специальные символы | Соединитель | + | + | + | + | + |
|  | Терминатор | + | + | + | - | - |
|  | Комментарий | + | + | + | + | + |
|  | Пропуск | + | + | + | + | + |

Примечание. Знак "+" указывает, что символ используют в данной схеме, знак "-" - не используют.

1 - Схема данных;

2 - Схема программы;

3 - Схема работа системы;

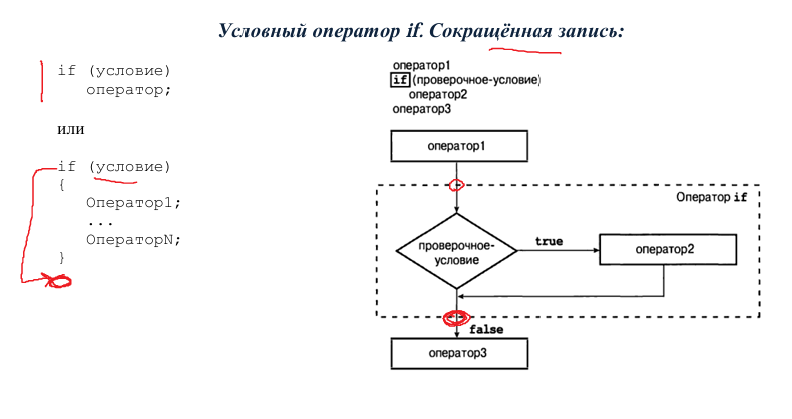
4 - Схема взаимодействия программ;

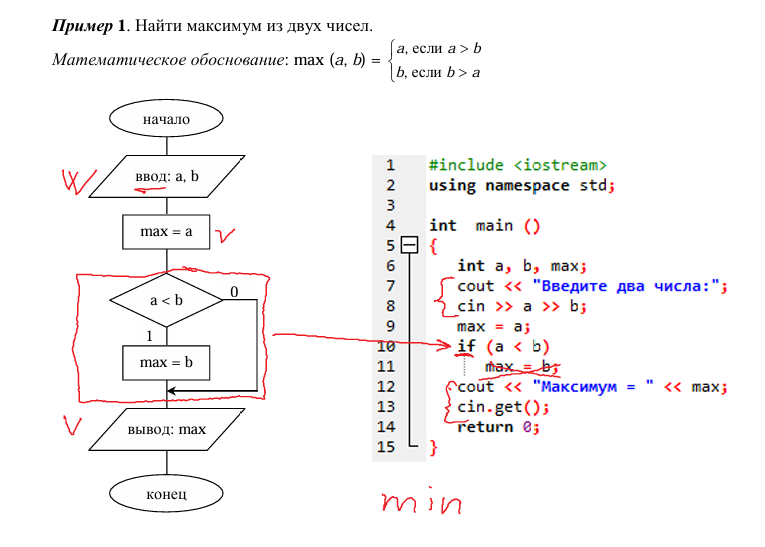
5 - Схема ресурсов системы;

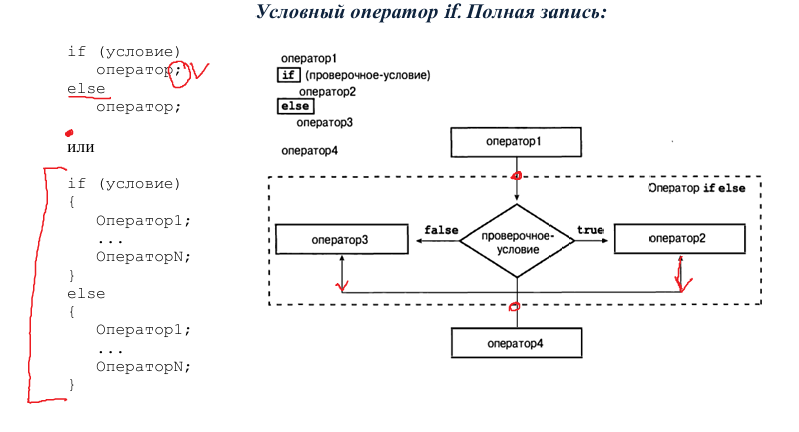
ОЗУ - оперативное запоминающее устройство;

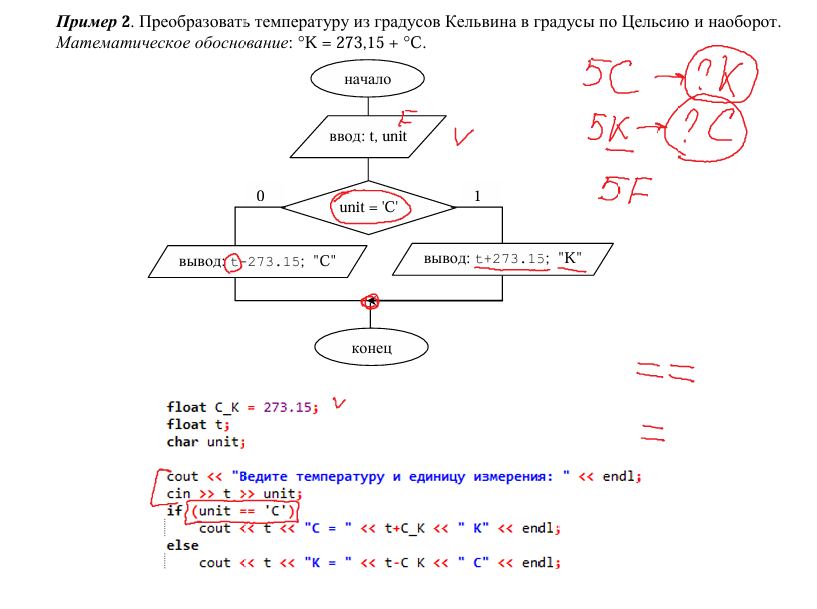
ЗУ - запоминающее устройство.

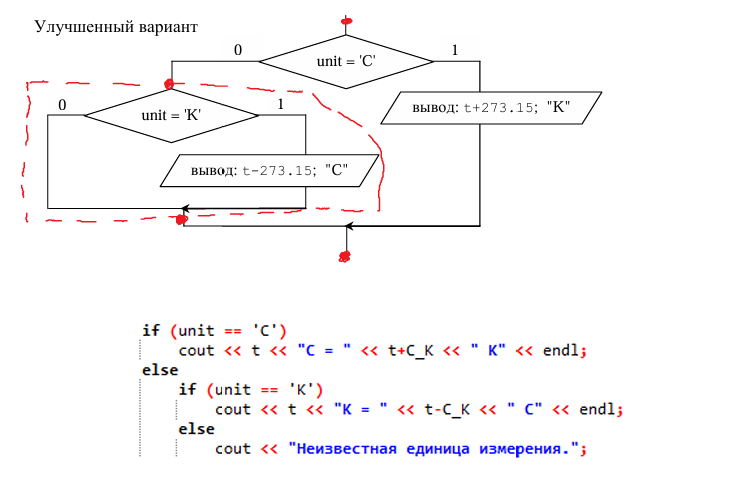
Условный оператор: сокращённая и полная записи, ГСА, синтаксис. Пример программы.

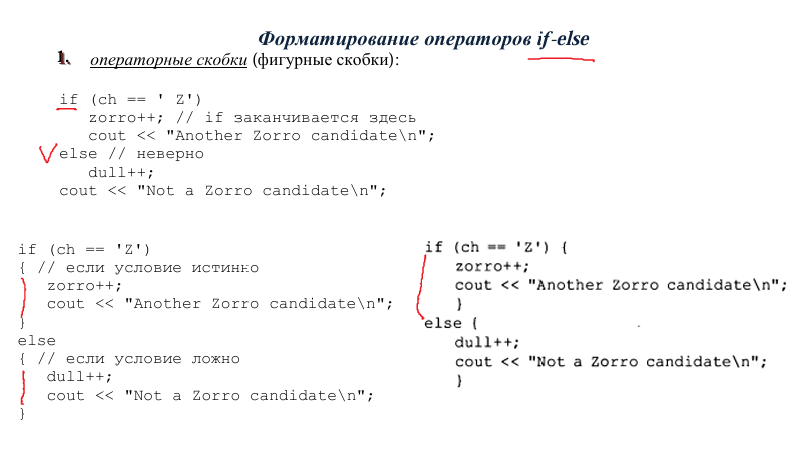






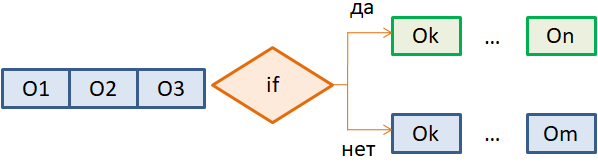






На этом занятии мы познакомимся с условным оператором if. Что он делает и для чего нужен? На предыдущем занятии мы с вами рассматривали операции сравнения, которые выдают либо 0 (false), либо 1 (true). Так вот, чтобы программа могла менять свое поведение в зависимости от этих значений, как раз можно воспользоваться оператором if.

Принцип его работы очень прост. Процессор последовательно выполняет операторы (команды) и, как только доходит до условного оператора, то при истинности условия выполняется одна группа операторов, а иначе – другая:



Это похоже на процесс ветвления при выполнении команд. Поэтому условные операторы еще иногда называют **операторами ветвления**.

Чтобы мы могли использовать оператор if в своих программах, нам нужно знать его синтаксис. В самых простых вариациях он имеет следующий вид:

if(<выражение>) оператор;

или

if(<выражение>) {  
    оператор\_1;  
    ...  
    оператор\_N;  
}

Здесь выражение – это любая конструкция языка Си, которая возвращает числовые значения. Число 0 будет интерпретировано как false, а любое не нулевое – как true. После круглых скобок должен идти оператор, который выполняется при истинности выражения. Если же нужно выполнить группу операторов, то для этого используются операторные фигурные скобки, которые можно воспринимать, как единый составной оператор. Поэтому, формально, после оператора if всегда следует один оператор.

Давайте для примера посмотрим, как можно использовать этот оператор для вычисления модуля числа:

#include <stdio.h>

int main(void)

{

    int x;

[scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &x);

    if(x < 0) x = -x;

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

    return 0;

}

Вначале в переменную x вводится с клавиатуры некоторое целое значение, а затем, делается проверка: если число x меньше нуля (то есть, отрицательное), то выполняется команда x = -x и знак меняется на противоположный, число становится положительным. Причем, эта команда будет выполнена только при условии, что x меньше нуля. Если же x больше или равен нулю, то команда пропускается и число остается положительным. В результате мы этой проверкой формируем модуль числа.

Обратите внимание, что при истинности условия выполняется только один оператор x = -x. Функция printf(), которая записана следующей строчкой, находится вне условного оператора и выполняется всегда. С условным оператором if связана только одна команда x = -x.

Кстати, в этой же программе было бы правильно проверить корректность введенного значения. Для этого достаточно проверить, что функция scanf() вернула единицу. Но я пропишу обратное условие:

    if([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &x) != 1) {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Error input");

        return 0;

    }

Смотрите, в качестве выражения здесь выступает вызов функции scanf() с последующей проверкой на неравенство единице. Так тоже вполне можно делать. Затем, если вводится некорректное значение, то это условие оказывается истинным и выполняются операторы внутри фигурных скобок. Будет выведена строка с сообщением об ошибке и оператор return 0, который завершит функцию main() и, как следствие, всю программу.

Не редко в программах на языке Си можно встретить использование оператора if с одной переменной в качестве условия, например, так:

if(x) x = -x;

Это тоже допустимо, т.к. в круглых скобках можно указывать любое выражение, которое возвращает числовое значение. И легко догадаться, если значение x равно 0, то условие считается ложным, а при любом другом – истинным.

А если нам нужно инвертировать это условие, то достаточно воспользоваться логической операцией НЕ следующим образом:

if(!x) x = -x;

Тогда все будет наоборот, условие истинно при x равном 0, и ложно при любом другом числовом значении.

Вообще, используя логические операции, можно формировать произвольные составные условия. Например, проверить попадание точки с координатой x в диапазон [3; 11]:

    if(x >= 3 && x <= 11)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x in [3; 11]**\n**");

А в качестве самостоятельного задания напишите проверку непопадания значения x в этот же диапазон [3; 11]. Вся информация для этого у вас уже есть.

## Конструкция if-else

Следующим шагом напишем программу различия положительных и отрицательных введенных чисел. Используя имеющиеся знания, это можно сделать следующим образом:

#include <stdio.h>

int main(void)

{

    int x;

    if([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &x) != 1) {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Error input");

        return 0;

    }

    if(x < 0) [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x < 0**\n**");

    if(x >= 0) [printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x >= 0**\n**");

    return 0;

}

У нас здесь два подряд идущих условия. Причем, взаимоисключающие. Действительно, x не может быть одновременно и меньше нуля и больше либо равно. После ввода значения возможен только один вариант. В таких ситуациях было бы логично проверить первое условие, если оно истинно, выполнить соответствующий оператор, а иначе, выполнить другой оператор без какой-либо проверки. Такую конструкцию в языке Си можно записать следующим образом:

if(<выражение>) оператор\_1;  
else оператор\_2;

или с использованием операторных скобок:

if(<выражение>) {  
    оператор\_1;  
    ...  
    оператор\_N;  
}  
else {  
    оператор\_1;  
    ...  
    оператор\_M;  
}

Здесь ключевое слово else как раз и соответствует ветке «иначе», то есть, оператор\_2 (или операторы от 1 до M в фигурных скобках) выполняется, если не сработало (оказалось ложным) выражение в условном операторе if. Таким образом, если условие в операторе if истинно, то выполняется оператор\_1 (или аналогичные операторы в фигурных скобках), а иначе – оператор\_2 (или аналогичные операторы в фигурных скобках).

Давайте перепишем нашу программу с использованием этой конструкции. Получим:

#include <stdio.h>

int main(void)

{

    int x;

    if([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &x) != 1) {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Error input");

        return 0;

    }

    if(x < 0)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x < 0**\n**");

    else

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x >= 0**\n**");

    return 0;

}

Из-за того, что здесь остался только один условный оператор, программа будет работать несколько быстрее. И везде, где мы имеем набор взаимоисключающих условий, их следует оформлять в виде конструкции if-else. Мало того, в нашем примере мы можем пойти дальше и после ключевого слова else прописать еще один условный оператор, например, так:

    if(x < 0)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x < 0**\n**");

    else if(x > 0)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x > 0**\n**");

        else

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x == 0**\n**");

Здесь после первого else записан еще один условный оператор if с проверкой x > 0 и у этого второго условного оператора также имеется свой блок else. Получилась такая вложенная конструкция из операторов if-else.

Я специально в программе второй блок if-else сместил вправо, чтобы визуально показать его вложенность в первый блок else. То есть, здесь последний оператор else относится именно ко второму условному оператору if. Или, это же вложение можно было бы для лучшего понимания оформить и так:

    if(x < 0)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x < 0**\n**");

    else {

        if(x > 0)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x > 0**\n**");

        else

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x == 0**\n**");

    }

В операторных скобках мы определили вложенный оператор if. Но в практике программирования фигурные скобки в этом случае не пишут, а оформляют такие вложения следующим образом:

    if(x < 0)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x < 0**\n**");

    else if(x > 0)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x > 0**\n**");

    else

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x == 0**\n**");

Читается и понимается такая конструкция достаточно просто. Сначала идет проверка первого условия x < 0. Если оно истинно, то выполняется оператор по условию и дальнейшие блоки пропускаются. Если же первое условие ложно, то попадаем в блок else первого оператора if и там осуществляется проверка вложенного условия x > 0. Если это условие истинно, то выполняется оператор printf("x > 0\n"), а если ложно, то переходим в последний блок else, который гарантированно срабатывает с выполнением оператора printf("x == 0\n").

Но всегда следует помнить, что конструкция if-else, или вложенные конструкции if-else, как правило, используются при проверке взаимоисключающих условий. Хотя это не всегда так. Например, нам нужно определить размер неотрицательного числа: является ли оно однозначным, двухзначным, трехзначным или каким-либо еще. Такую проверку можно прописать следующим образом:

    if(x < 10)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x in [0; 9]**\n**");

    else if(x < 100)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x in [10; 99]**\n**");

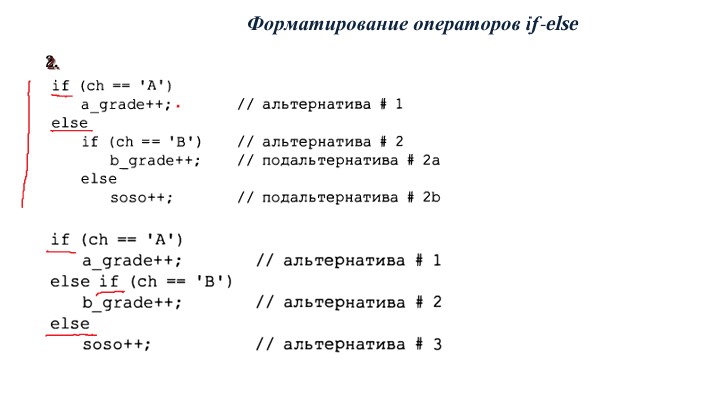
    else if(x < 1000)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x in [100; 999]**\n**");

    else

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x > 999**\n**");

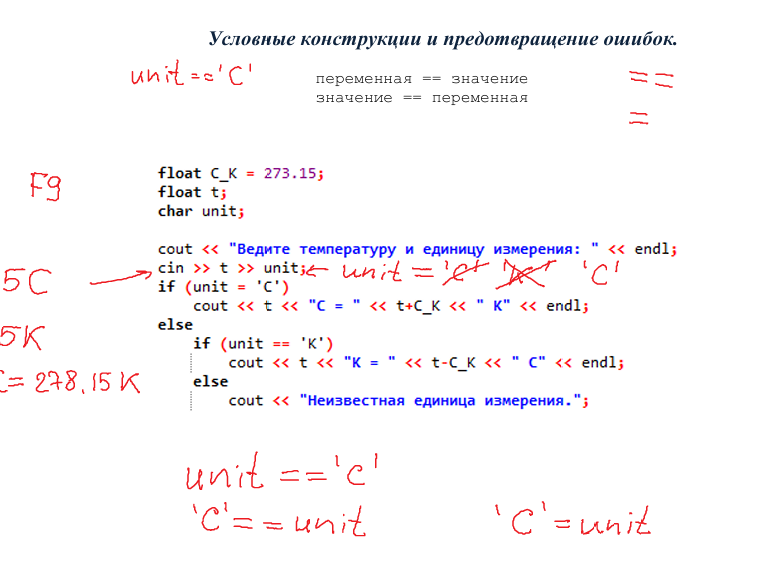
Формально условия x < 10; x < 100; x < 1000 не взаимоисключающие, но порядок их следования таков, что они позволяют выделять нам нужные независимые группы чисел. То есть, творческое мышление, при использовании различных операторов, никто не отменял. Иначе бы программы уже давно писались бы не людьми, а теми же компьютерами. Хотя такие попытки предпринимаются, в частности, с помощью нейронных сетей. Но пока до человеческого уровня им далеко и в ближайшие 10-20 лет грамотный программист в этой области незаменим.



Составной оператор. Операторные скобки.

Если же нужно выполнить группу операторов, то для этого используются операторные фигурные скобки, которые можно воспринимать, как единый составной оператор.

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



На этом занятии вы увидите, как в программах можно делать логические выводы на уровне:

* true – истина;
* false – ложь.

Так как компьютер – это вычислительная машина, то понятия «истина» и «ложь» должны выражаться на уровне чисел. В большинстве, а может быть во всех языках программирования, значение false определяется как 0, а true – как любое ненулевое значение. Поэтому константы true и false часто  определяют как:

true = 1; false = 0

До стандарта C99 язык Си не предполагал наличия какого-либо специального булевого типа данных. Вместо этого использовался любой целочисленный тип, и если значение переменной равнялось нулю, это означало false (ложь), а любое другое значение, отличное от нуля – true (истина). Например, так:

char fl\_view = 0;        *// false*

int fl\_open\_file = 1;     *// true*

Но стандарт C99 предоставляет нам новый тип (новое ключевое слово):

\_Bool

И булевы переменные стало возможно определять следующим образом:

\_Bool fl\_view = 0;           *// false*

На самом деле, тип \_Bool определен на базе обычного целочисленного типа, который соответствует наименьшему допустимому размеру, то есть, char. В этом легко убедиться, воспользовавшись оператором sizeof:

#include <stdio.h>

int main(void)

{

    \_Bool fl\_view = 0;       *// false*

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

    return 0;

}

После запуска программы увидим строчку:

Size of \_Bool: 1

К сожалению, ключевые слова true или false по-прежнему не определены. Поэтому булевой переменной fl\_view нужно явно присваивать значение 0 в качестве false и 1 – для true. Причем, никакие другие значения она принимать не может. Если попытаться присвоить, например, число 10:

\_Bool fl\_view = 10;

то fl\_view его преобразует к единице. И, вообще, любое ненулевое значение приравнивается единице. В этом ключевое отличие типа \_Bool, например, от типа char, который позволяет представлять любые целые числа в диапазоне [0; 255].

А теперь внимательнее посмотрим на написание этого типа. Вначале идет символ подчеркивания, а затем, с заглавной буквы слово «Bool». Довольно неудобная и «корявая» запись. Дело в том, что это была первая официальная попытка определить булевый тип в языке Си. И никто тогда точно не мог сказать, как эта идея будет воспринята сообществом программистов. Кроме того, для совместимости с прежними компиляторами, этот тип можно было легко заменить на любой другой базовый целочисленный, например, char. И программы бы компилировались без проблем.

В действительности, именно в такой записи тип \_Bool практически не использовался на практике. Позже его подменили более изящным словом bool. И, кроме того, такой тип официально появился в языке С++, который является естественным развитием языка Си. Так вот, чтобы в программах на языке Си в соответствии со стандартом C99 можно было бы использовать более приятную и общеупотребительную запись булевого типа bool, следует подключить заголовочный файл stdbool.h:

#include <stdbool.h>

В нем не только переопределен тип \_Bool как bool, но и введены две константы:

true = 1; false = 0

Поэтому переменную fl\_view в нашей программе теперь можно определить так:

bool fl\_view = **true**;

Эта строчка выглядит естественнее и понятнее для программиста, чем корявый тип \_Bool и числа 0 или 1.

## Операции сравнения

Я, думаю, с определением переменных булевого типа в языке Си все понятно. Давайте теперь посмотрим, как можно их использовать в операциях сравнения. Вначале приведу список этих операций.

|  |  |
| --- | --- |
| **Операция** | **Описание** |
| == | (Два равно). Сравнение на равенство. |
| != | Сравнение на неравенство. |
| < | Сравнение на меньше. |
| > | Сравнение на больше. |
| <= | Сравнение на меньше или равно. |
| >= | Сравнение на больше или равно. |

Все эти операции являются бинарными, то есть, слева и справа от них прописываются выражения (операнды):

<левый операнд> <операция сравнения> <правый операнд>

В качестве операндов может выступать любая конструкция языка Си, со значениями которых возможны сравнения. Часто это переменные и числовые литералы.

Обратите внимание, все операции сравнения являются именно операциями, а не операторами, то есть, они позволяют выполнять некоторое сравнение и возвращают вычисленный результат в виде значений:

0 – false; 1 – true.

Например:

#include <stdio.h>

#include <stdbool.h>

int main(void)

{

    double x = 5.67;

    bool fl\_view = x < 0;

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

    return 0;

}

Так как в нашем примере x больше нуля, то операция сравнения на отрицательное значение вернет 0, которое соответствует понятию «ложь» (false). А вот если вместо нуля указать, например, значение 10:

bool fl\_view = x < 10;

то переменная fl\_view будет равна 1, что соответствует понятию «истина» (true). И так работают все операции сравнения:

double x = 5.67;

int var\_i = 7;

bool fl\_view = x < 10;      *// true*

bool res\_1 = 5 > 7;         *// false*

bool res\_2 = x+2 >= 10.56;    *// false*

bool res\_3 = var\_i == 7;    *// true*

bool res\_4 = var\_i != 7;    *// false*

Причем, приоритет операций сравнения выше приоритета операции присваивания. Поэтому сначала выполняются сравнения и только потом – присваивания. А арифметические операции выше операций сравнения, поэтому x+2 будет выполнено до сравнения на больше или равно. Кроме того, обратите внимание на операцию сравнения на равенство. Она записывается как два символа равно (==). Как мы знаем, одно равно – это операция присваивания, поэтому для сравнения на равенство ввели обозначение из двух символов равно. И здесь начинающие программисты очень часто делают ошибку. Для сравнения на равенство значений они пишут не два, а, по привычке, одно равно, например, так:

bool res\_3 = var\_i = 7;

При этом компилятор может даже не выдать никаких предупреждений и в любом случае переведет программу в машинный код. Только работать она будет не так, как задумывалась программистом. И чисто визуально найти такую ошибку бывает не просто, особенно, если в голове не зафиксировано, что сравнение – это два знака равно. В общем, если вы только начинаете постигать азы программирования на языке Си, то выпишите этот момент на бумажке и прилепите ее к монитору. Я гарантирую, что найдется немало тех, кто ее совершит.

Также следует помнить, что операции:

<= и >=

нужно прописывать именно в таком виде. Иногда, правда редко, меняют местами символы и пытаются прописать:

=< и => (неверная запись)

Это приведет к синтаксической ошибке.

Давайте в качестве короткого примера приведу программу ввода числового значения и определение четности числа:

#include <stdio.h>

#include <stdbool.h>

int main(void)

{

    int digit;

[scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &digit);

    bool even = digit % 2 == 0;

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

    return 0;

}

После запуска и ввода целого значения мы увидим 0 – для нечетных чисел и 1 – для четных.

## Составные операции сравнения

Давайте теперь поставим более сложную задачу и определим, попадает ли число (значение переменной) в диапазон [-2; 5]? Какие логические умозаключения здесь нужно провести, чтобы ответить на этот вопрос? Для простоты представим, что у нас имеется переменная:

double y = 1.85;

Очевидно, чтобы она попадала в диапазон [-2; 5], нужно соблюдение двух условий:

y >= -2

и

y <= 5

В языке Си это можно реализовать следующим образом:

bool is\_range = y >= -2 && y <= 5;

Здесь операция && означает логическое И. При этом общее условие истинно, если истинно каждое из подусловий: y >= -2 и y <= 5. Благодаря этому мы будем получать значение true (единица), если y принадлежит диапазону [-2; 5] и false (ноль) в противном случае.

А теперь сделаем противоположную проверку, что переменная y не попадает в диапазон [-2; 5]. Очевидно, это будет происходить, если:

y < -2 или y > 5

В языке Си такая составная операция сравнения может быть записана в виде:

bool is\_not\_range = y < -2 || y > 5;

Операция || означает логическое ИЛИ и возвращает истину (true), если истинно хотя бы одно из подусловий. В нашем примере, это проверка, что y или меньше -2 или больше 5. Часто начинающие программисты здесь делают логическую ошибку и подобное сравнение записывают с использованием операции И следующим образом:

bool is\_not\_range = y < -2 && y > 5;

В этом случае я прошу назвать число, которое одновременно меньше -2 и больше 5. Как правило, после этого в головах все встает на свои места. Не путайте принцип действия этих двух операций: логическое ИЛИ и логическое И.

На самом деле противоположную проверку непопадания в диапазон [-2; 5] можно было бы реализовать путем инвертирования ранее вычисленного значения is\_range следующим образом:

bool is\_not\_range = !is\_range;

Здесь восклицательный знак – это унарная операция НЕ, которая может быть применена к любому выражению. Принцип ее работы заключается в инвертировании булевого значения:

true -> false; false->true.

## Приоритеты операций И, ИЛИ, НЕ

Приоритеты всех этих трех логических операций следующие:

|  |  |
| --- | --- |
| Логическое ИЛИ (||) | 1 |
| Логическое И (&&) | 2 |
| Логическое НЕ (!) | 3 |

То есть, наибольший приоритет имеет унарная операция НЕ, затем, операция И, и самый низкий – у операции ИЛИ.

Все эти приоритеты необходимо строго соблюдать для составления корректных условий. Например:

int x = 5;

bool is\_correct = x % 2 == 0 || x % 3 == 0 && x > 5;

Это составное условие эквивалентно следующему:

bool is\_correct = x % 2 == 0 || (x % 3 == 0 && x > 5);

то есть, сначала проверяется, что число x кратно 2 (четное) ИЛИ число кратно 3 и при этом больше 5. Обратите внимание здесь на два важных момента. Во-первых, стандартом языка Си определен строгий порядок проверок слева-направо при вычислении составных логических операций. Это значит, мы можем быть абсолютно уверены, что сначала выполнится проверка x % 2 == 0 и только после этого следующее подусловие x % 3 == 0 && x > 5. Причем, в нем также сначала проверяется первое x % 3 == 0 и только потом второе x > 5. Во-вторых, если в процессе проверки значение всей составной логической операции становится известным, то вычисления прерываются и не идут дальше. Например, в условии:

bool fl\_digit = x != 0 && 10 / x > 1;

сначала будет проверено, что переменная x не равна нулю, и если это не так (то есть это первое подусловие равно false), то дальше проверки делать не имеет смысла, т.к. при любом булевом значении второго подусловия (10/x > 1) общее все равно будет принимать значение false. Благодаря такому поведению, мы можем без проблем вычислять деление 10 / x после проверки, что x != 0. Этим на практике довольно часто пользуются.

Но все это нужно применять очень аккуратно. Например, если прописать следующее составное условие:

bool is\_read = x < 0 && [scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &x) == 1;

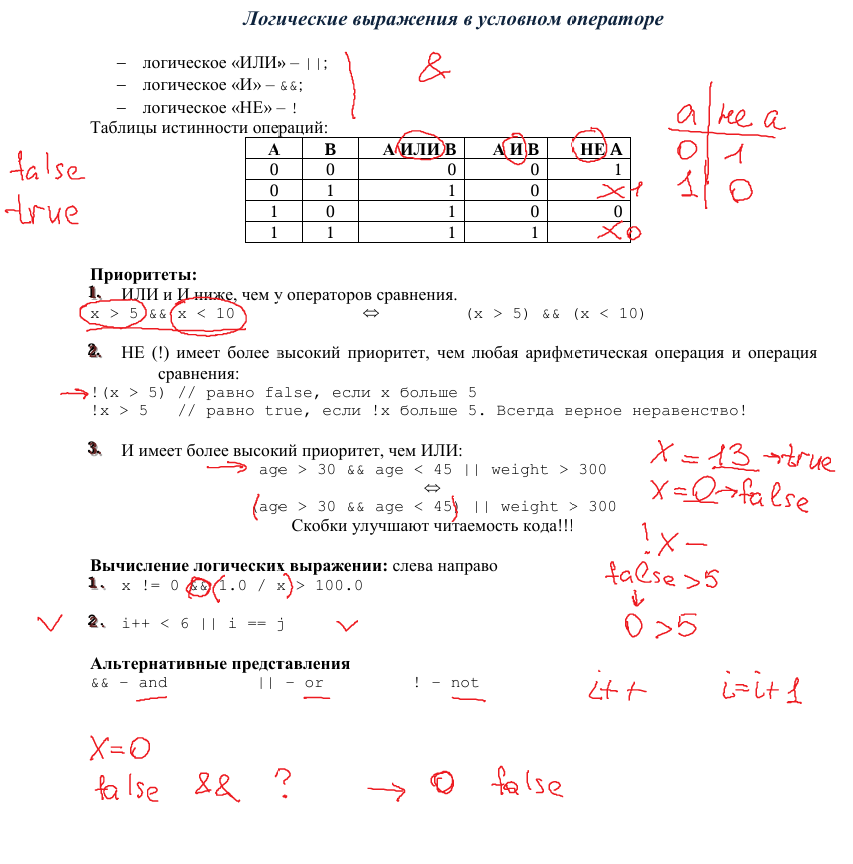
то функция scanf() не будет вызвана, если x больше или равен нулю.

И последнее, важный, хотя и очевидный момент. Если нам нужно поменять приоритеты операций, то для этого можно использовать все те же круглые скобки. Например:

bool is\_correct = (x % 2 == 0 || x % 3 == 0) && x > 5;

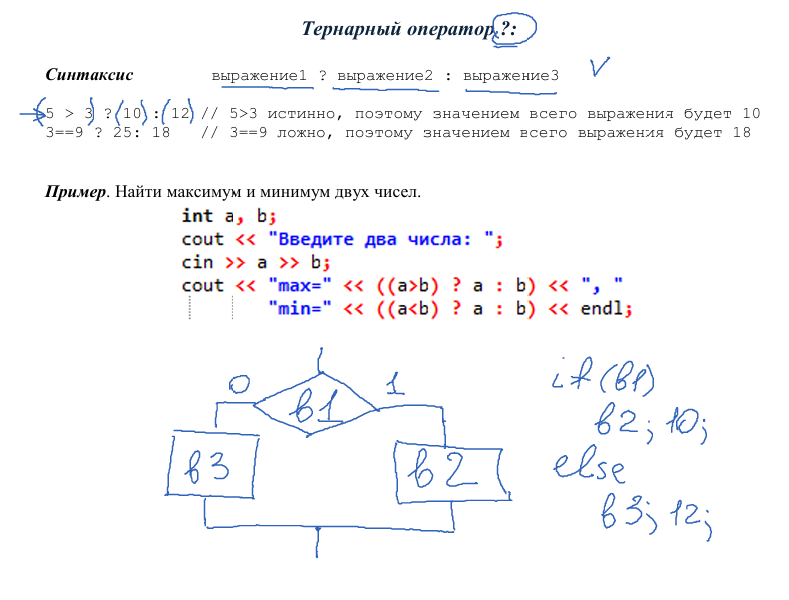
Теперь это условие будет истинно, если x кратно 2 или 3 и больше 5.

Логические выражения в условных операторах. Альтернативные представления логических операторов.



Выражения отношений. Примеры логических выражений.

Тернарный оператор: назначение, ГСА, синтаксис. Примеры



# Условное тернарное выражение

Практический курс по C/C++: <https://stepik.org/course/193691>

Смотреть материал на [YouTube](https://www.youtube.com/watch?v=_RztW4dtW9w&list=PLA0M1Bcd0w8w-mqVmBjt-2J8Z1gVmPZVz)| [RuTube](https://rutube.ru/video/4082e0baa33eac635386e504040356bc/?playlist=539096)

На этом занятии мы познакомимся с условной тернарной операцией, которая имеет следующий синтаксис:

<выражение 1> ? <выражение 2> : <выражение 3>

Если выражение 1 истинно, то возвращается значение выражения 2, иначе – значение выражения 3. Давайте поясню его работу на конкретном примере. Предположим, у нас имеются две переменные:

double a = 7.5, b = -3.43;

и мы хотим определить максимальное значение. Используя имеющиеся знания, это можно было бы сделать так:

double max\_ab;

if(a > b)

    max\_ab = a;

else

    max\_ab = b;

А с использованием тернарной условной операции это можно реализовать гораздо изящнее:

double max\_ab = a > b ? a : b;

Видите, какая простая, понятная и компактная запись в итоге получилась. Это одно из удобств данного оператора. Но, все же, между предыдущей программой с условным оператором if и тернарной операцией есть одно принципиальное отличие. Тернарная операция возвращает результат. В данном примере – это или переменная a или переменная b. Тогда как обычный условный оператор предназначен для выполнения блока кода по определенному условию и сам по себе не возвращает никаких значений.

Второе важное отличие – в тернарной условной операции нет внутренних блоков, где бы мы могли записывать несколько операторов. Вместо a и b можно указывать только одну какую-либо конструкцию. Часто это некое значение, или результат работы операторов, например, арифметических:

double res = (a > b) ? a + 2 : b - 5;

Так делать вполне допустимо. И, обратите внимание, я поместил условие в круглые скобки, чтобы визуально выделить его в этой конструкции. Конечно, делать это не обязательно, так как приоритет операций ? и : выше только операций присваивания и запятой, поэтому сначала будут вычислены операции сравнения и только потом осуществляется проверка условия.

Также в качестве выражений в тернарной операции можно использовать функции, например, так:

double res\_abs = (a < b) ? [fabs](http://www.opengroup.org/onlinepubs/009695399/functions/fabs.html)(a) : [fabs](http://www.opengroup.org/onlinepubs/009695399/functions/fabs.html)(b);

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("res\_abs = %.2f**\n**", res\_abs);

Забегая вперед, я здесь использовал функцию fabs() для вычисления модуля вещественного числа. Чтобы воспользоваться этой функцией вначале программы нужно подключить заголовочный файл math.h:

#include <math.h>

Из этого последнего примера хорошо видно, что в зависимости от истинности условия будет вызвана либо первая функция fabs(a), либо вторая fabs(b). И это очень важный момент. Следует всегда помнить, что в тернарной условной операции вычисляется только одно из двух последних выражений, но не оба вместе.

Так как тернарная операция – это операция, а не оператор, то ее можно прописывать как аргумент при вызове функций. Например:

#include <stdio.h>

int main(void)

{

    int x = 8;

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("x is %s digit**\n**", (x % 2 == 0) ? "even" : "odd");

    return 0;

}

Здесь тернарная операция записана как аргумент функции printf() и возвращает строковый литерал в зависимости от четности числа.

## Вложенная тернарная операция

Так как внутри тернарной условной операции можно использовать любые конструкции языка Си, то что нам мешает вложить одну тернарную операцию в другую, например, так:

#include <stdio.h>

int main(void)

{

    int a = 2, b = 3, c = -4;

    int max = (a > b) ? ((a > c) ? a : c) : (b > c) ? b : c;

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

    return 0;

}

Сразу скажу, что это делается крайне редко и лучше избегать таких вложений, так как восприятие и понимание текста программы резко снижается.

Работает приведенная программа следующим образом. Сначала проверяется условие a > b. Если это так, то далее с помощью вложенной тернарной операции возвращается максимально из двух переменных a и c. Иначе, максимум из переменных b и c. В результате получаем максимальное из трех чисел a, b, c.

Причем здесь обязательно вложенные тернарные операции следует записывать в круглых скобках, так как приоритет операций ? и : очень низкий.

Еще раз отмечу, что таких вложений в практике программирования следует избегать, так как в ней очень легко запутаться и допустить ошибку. Да и потом разбираться, что здесь было написано не так то просто. Поэтому в 99% случаях условную тернарную операцию используют без каких либо вложений.

Связь условного и тернарного операторов. Примеры.

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

В языке Си имеется еще один оператор ветвления switch, который позволяет реализовывать в программах множественный выбор.

Вначале я приведу синтаксис этого оператора, а затем, мы подробно посмотрим на примеры его работы. Общее определение можно записать следующим образом:

switch(<выражение>) {  
case <константа 1> :  
        [операторы]  
case <константа 2> :  
        [операторы]  
...  
[default : операторы]  
}

Здесь выражение – это любая конструкция языка Си, которая возвращает целочисленное значение (включая значение типа char). Соответственно, константы после ключевых слов case также должны быть целочисленными. Причем, вычисляемыми на этапе компиляции (например, целочисленные переменные использовать нельзя, а вот числовые литералы можно). После константы ставится двоеточие и могут быть прописаны операторы, которые выполняются по указанной метке. Обратите внимание, операторы здесь могут отсутствовать и далее в примерах мы увидим, когда это может быть полезно. Наконец, последнее необязательное ключевое слово default позволяет задавать набор операторов, которые выполняются, если не сработала ни одна из меток.

Работу оператора switch проще понять на конкретных примерах. Давайте предположим, что нам нужно делать выбор одного из пунктов меню:

1. Learning C language  
2. Learning Python language  
3. Learning Java language  
4. Learning C++ language  
5. Exit

Программа будет следующей:

#include <stdio.h>

int main(void)

{

    int item;

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("1. Learning C language**\n**"

           "2. Learning Python language**\n**"

           "3. Learning Java language**\n**"

           "4. Learning C++ language**\n**"

           "5. Exit**\n**");

    if([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &item) != 1) {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Error input");

        return 0;

    }

    switch(item) {

    case 1:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning C language**\n**");

    case 2:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning Python language**\n**");

    case 3:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning Java language**\n**");

    case 4:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning C++ language**\n**");

    case 5:

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

    }

    return 0;

}

Здесь после case прописаны целочисленные числовые литералы от 1 до 5. Конечно, в реальной практике программирование этого следует избегать и не прописывать конкретные числа в программном коде. Но, так как мы с вами еще не проходили определение констант, то эти числа следует воспринимать исключительно, как учебный вариант.

Итак, когда выполнение программы доходит до оператора switch, то в нем берется значение из целочисленной переменной item и, затем, оно последовательно (сверху-вниз) сравнивается на равенство с константами (метками), записанными после ключевого слова case.

Давайте запустим программу и введем с клавиатуры число 3. В результате увидим вывод следующих строк:

Learning Java language  
Learning C++ language  
Exit

И этот результат часто удивляет начинающих программистов. Они ожидают здесь увидеть только выполнение одной функции printf(), связанной с меткой 3, а выполняются три функции printf(), начиная с метки 3. Именно так работает оператор switch. Как только срабатывает какая-либо метка, то далее выполняются все операторы, которые записаны после этой метки, включая операторы в других нижестоящих метках. Причем, речь идет именно о нижестоящих метках. Например, если ключевое слово case со значением 3 прописать в самом начале:

    switch(item) {

    case 3:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning Java language**\n**");

    case 1:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning C language**\n**");

    case 2:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning Python language**\n**");

    case 4:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning C++ language**\n**");

    case 5:

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

    }

И снова выполнить программу, ввести значение 3, то увидим все выводы строк:

Learning Java language  
Learning C language  
Learning Python language  
Learning C++ language  
Exit

## Прерывание выполнения оператора switch. Ключевое слово break

Может для вас такое поведение оператора switch кажется несколько неожиданным, но он работает именно так. Тогда, спрашивается, как нам можно реализовать множественный выбор так, чтобы выполнялись операторы только по одной метке, а не по всем нижестоящим? Для этого можно воспользоваться оператором **break**, который прерывает работу оператора switch. Например, если прописать break после каждого вызова функции printf():

    switch(item) {

    case 3:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning Java language**\n**");

**break**;

    case 1:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning C language**\n**");

**break**;

    case 2:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning Python language**\n**");

**break**;

    case 4:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning C++ language**\n**");

**break**;

    case 5:

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

**break**;

    }

То, как только он встречается, управление переходит к следующему оператору после switch. В результате, все операторы в нижестоящих метках пропускаются.

Однако часто оператор switch выносят в отдельную функцию и тогда прервать его работу можно также и с помощью оператора return. В нашем примере тоже так можно сделать, если вместо break прописать «return 0;» следующим образом:

    switch(item) {

    case 3:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning Java language**\n**");

        return 0;

    case 1:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning C language**\n**");

        return 0;

    case 2:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning Python language**\n**");

        return 0;

    case 4:

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Learning C++ language**\n**");

        return 0;

    case 5:

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

        return 0;

    }

Кстати, оператор return чаще применяется в практике программирования для прерывания работы оператора switch.

Конечно, как только встречается «return 0;», то функция main() завершает свою работу и все операторы, стоящие после switch выполняться уже не будут. В нашем конкретном случае, это не критично, т.к. там нет никакого значимого программного кода. Однако, если программу нужно продолжить после оператора switch, прервав его работу, то следует использовать оператор break. В этом главное отличие между «return 0» и break.

## Ключевое слово default

Давайте для примера рассмотрим еще один вариант использования оператора switch. Мы с клавиатуры будем вводить символ буквы (малой или большой) и выводить соответствующее сообщение на экран:

#include <stdio.h>

int main(void)

{

    char item;

    if([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%c", &item) != 1) {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Error input");

        return 0;

    }

    switch(item) {

    case 'a':

    case 'A':

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

**break**;

    case 'b':

    case 'B':

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

**break**;

    case 'c':

    case 'C':

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

**break**;

    }

    return 0;

}

Обратите внимание на два момента. Первый, метки для малой и заглавной букв следуют друг за другом, причем, блок операторов записан только у второй метки, а первая идет без операторов. Благодаря этому мы можем и для малой и для большой буквы выполнять одни и те же операторы, что довольно удобно. И второй момент. В качестве меток используются символы. В действительности, как мы с вами уже говорили, символы при трансляции программы в машинный код, преобразуются в целые числа и, по сути, представляют собой целочисленные константы. Именно поэтому символы можно прописывать наряду с целыми числами после ключевого слова case.

Если мы запустим эту программу и введем символ, который не указан в метках, например, буквы d, то на экран ничего выведено не будет. Оператор switch просто завершит свою работу, а, затем, завершится и вся программа. Давайте сделаем так, чтобы при вводе не указанного символа на экран выводилась строка «Incorrect symbol». Сделать это очень просто, используя ключевое слово default, следующим образом:

    switch(item) {

    case 'a':

    case 'A':

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

**break**;

    case 'b':

    case 'B':

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

**break**;

    case 'c':

    case 'C':

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

**break**;

    default:

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

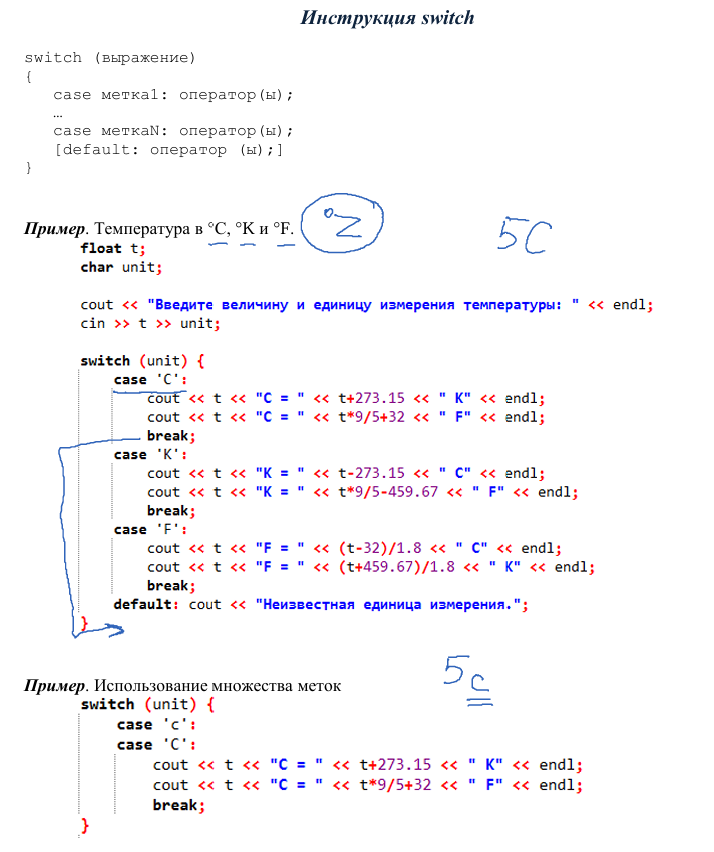
    }

То есть, операторы по метке default отрабатывают в том случае, когда не сработала ни одна из меток case. Причем определение default может быть записано в любом месте оператора switch, но, как правило, его пишут в самом конце.

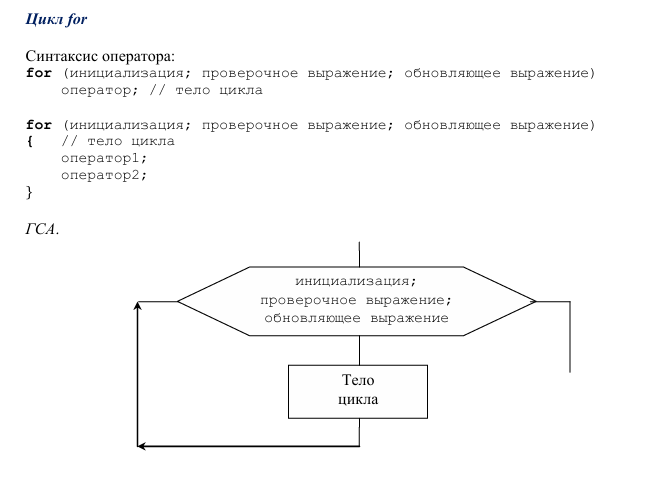
## Рекомендации по использованию оператора switch

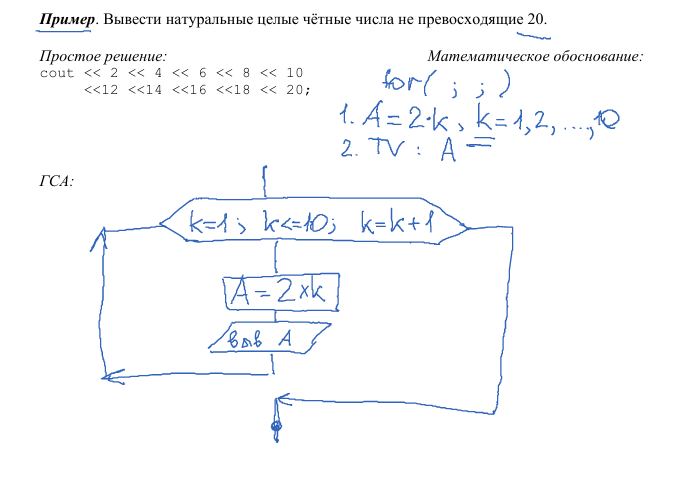
Вы могли заметить, что запись оператора switch занимает немало строк, даже в самых простых задачах. В реальной практике он нередко разрастается до больших размеров и затрудняет восприятие текста программы. Поэтому есть несколько рекомендаций, которых следует придерживаться при использовании оператора switch:

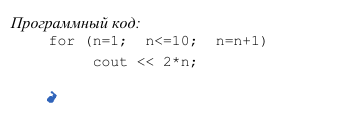
* оператор switch лучше выносить в отдельную функцию и прерывать его работы с помощью оператора return;
* программу следует организовывать так, чтобы после каждой метки case прописывался вызов некоторой функции вместо набора операторов;
* никогда в программах не используйте вложенные операторы switch.

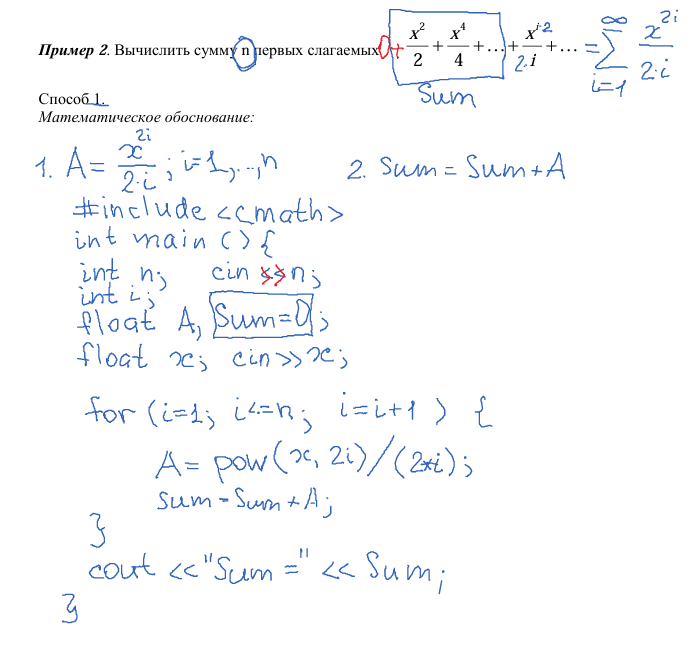


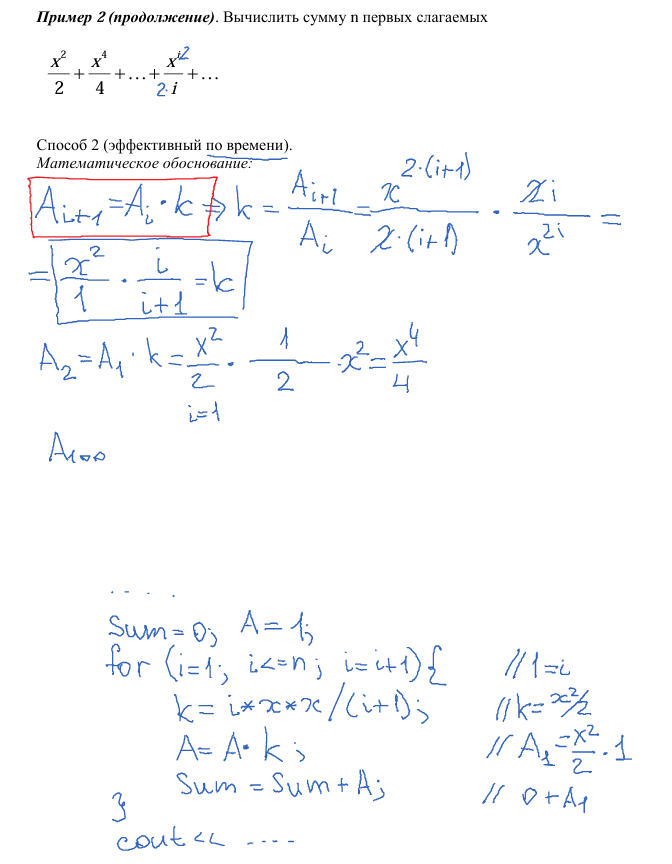
Цикл  параметром for: назначение, ГСА. Пример.

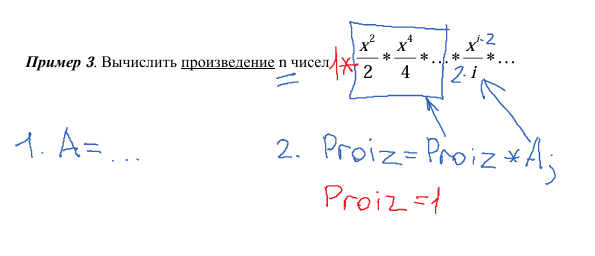












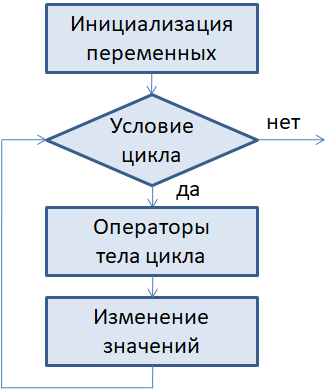
# Оператор цикла for

Практический курс по C/C++: <https://stepik.org/course/193691>

Смотреть материал на [YouTube](https://www.youtube.com/watch?v=Uq0VSGpqdDA&list=PLA0M1Bcd0w8w-mqVmBjt-2J8Z1gVmPZVz)| [RuTube](https://rutube.ru/video/428c06b7ce03416c1dfe65674c0afafb/?playlist=539096)

На прошлом занятии мы рассмотрели работу цикла while, на этом занятии речь пойдет об операторе for. Оба этих оператора образуют циклы с предусловием, то есть, сначала проверяется условие цикла и если оно истинно, то выполняется текущая итерация. И здесь возникает вопрос, зачем понадобился еще один оператор цикла с предусловием?

Смотрите, в практике программирования очень часто возникают задачи, когда нужно организовать циклы по следующей схеме:



То есть, перед циклом мы выполняет некоторую инициализацию переменных, используемых затем в цикле. Далее идет условие цикла, если оно истинно, то выполняются операторы, записанные внутри цикла. И в конце после основной группы операторов, выполняются некоторые изменения переменных для новой итерации. Например, ранее мы использовали эту схему в цикле while следующим образом:

*/\* Инициализация \*/*

int n;

int s = 0;

*/\* Цикл с предусловием \*/*

while(n > 0) {

         s += n\*n;    */\* Операторы тела цикла \*/*

         n--;         */\* Изменение значений \*/*

}

Так вот, чтобы иметь возможность в программах записывать подобные циклы в более краткой форме, и был введен оператор for, который имеет следующий синтаксис:

for([инициализация]; [условие]; [изменение значений])  
    оператор;

или

for([инициализация]; [условие]; [изменение значений]) {  
    операторы;  
}

Как видите, сразу в этом операторе можно прописать инициализацию переменных перед запуском цикла, условие цикла и порядок изменения значений после выполнения каждой итерации цикла. Обратите внимание, все эти элементы являются не обязательными, то есть, мы можем не прописывать инициализацию, условие и изменение значений. Также видим, что все эти элементы внутри цикла for разделены между собой точкой с запятой. Это, наверное, единственный оператор языка Си, который в своем определении использует символ точку с запятой как разделитель.

Давайте перепишем программу выше с циклом while через оператор for. Получим:

#include <stdio.h>

int main(void)

{

*/\* Объявление переменных \*/*

         int n, s;

[scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &n);

*/\* Цикл с предусловием for \*/*

         for(s = 0; n > 0; --n)

                   s += n\*n;    */\* Операторы тела цикла \*/*

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

         return 0;

}

Как видите, сам цикл имеет довольно краткую запись и визуально мы сразу можем выделить блок инициализации, блок проверки условия и блок изменения переменной. Если вам сейчас такая запись кажется несколько странной, то с опытом к ней очень быстро привыкаешь. И благодаря удобству оператор цикла for используется гораздо чаще оператора цикла while.

Давайте я приведу еще один пример с оператором цикла for для вычисления факториала числа:

#include <stdio.h>

int main(void)

{

         int n = 5, p = 1;

         for(int i = 1; i <= n; ++i)

                   p = p \* i;

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

         return 0;

}

Я напомню, что:

n! = 1 ∙ 2 ∙ 3 ∙ … ∙ n

В программе мы определяем n = 5 и через цикл for находим факториал этого числа. Для этого задается вспомогательная переменная p с начальным значением 1 и в цикле for счетчик (переменная) i также с начальным значением 1. Обратите внимание, внутри блока инициализации допустимо объявлять переменную, которая ранее нигде не существовала. Причем, используемый мной компилятор gcc, создает эту переменную исключительно внутри цикла for и за его пределами она не существует. Однако другие компиляторы, работающие по другим стандартам, вполне могут определять такие переменные за пределами оператора for. Соответственно, доступ к ним сохраняется после выполнения цикла. Но для нас сейчас важно лишь то, что в блоке инициализации можно объявлять переменные и использовать их в теле цикла данного оператора.

Работает цикл очень просто. Сначала проверяется условие цикла, т.к. оно истинно, то выполняется текущая итерация – оператор «p = p \* i;». После этого происходит увеличение счетчика i на единицу и снова проверяется условие цикла. В результате, мы получаем значение переменной p равное:

p = 1 ∙ 2 ∙ 3 ∙ 4 ∙ 5 = 120

А вот еще несколько вариаций записи этого же цикла:

         int i, p;

         for(i = 1, p = 1; i <= n; ++i)

                   p = p \* i;

Здесь использована новая операция запятая для инициализации двух переменных. Или:

         int n = 5;

         int i = 1, p = 1;

         for(; i <= n; ++i)

                   p = p \* i;

Здесь пустой блок инициализации, т.к. она прописана до оператора цикла. Или:

         int n = 5;

         int i = 1, p = 1;

         for(; i <= n;) {

                   p = p \* i;

                   ++i;

         }

Здесь два пустых блока: инициализации и изменения значений. Или:

         int n = 5, i, p;

         for(i = 1, p = 1; i <= n; p = p \* i, ++i)

                   { }

Вся логика вычислений прописана внутри оператора цикла for. Соответственно, в теле цикла ничего прописывать не нужно, но формально там все же должен быть указан хотя бы один оператор. Точка с запятой, как раз и воспринимается компилятором как оператор, который ничего не делает. Также в блоке изменения значений прописаны две операции, разделенные запятой. При этом сначала выполнится первая операция p = p \* i и только потом вторая ++i. Это поведение для операции запятая строго определено в стандарте языка Си. Вычисление выражений, разделенных запятой, происходит слева-направо и никак иначе.

Начинающий программист, глядя на все это разнообразие форм записей одного и того же оператора для решения одной и той же задачи, может задаться вполне резонным вопросом: какой же вариант лучше? В действительности, следует выбирать тот, который проще воспринимать и модифицировать (при необходимости). Компилятор все эти вариации переведет в машинный код примерно одинаковым образом. Лично я выбрал бы здесь первый вариант записи, т.к. он хорошо отражает логику работы программы.

Раз уж мы затронули вопрос различных вариаций записей оператора цикла for, то приведу еще одну без каких-либо блоков:

for( ; ; );

Так тоже можно записывать. В этом случае цикл будет работать «вечно» пока мы или операционная система не прервет выполнение программы.

Вообще цикл for используют не только для формирования каких-либо арифметических последовательностей в программах. Спектр его применения куда шире. Здесь следует исходить из того, что в блоках инициализации и изменения значений можно прописывать любые допустимые конструкции. Например, такие:

#include <stdio.h>

#include <stdlib.h>

int main(void)

{

         for(int x = [rand](http://www.opengroup.org/onlinepubs/009695399/functions/rand.html)() % 10; x != 0; x = [rand](http://www.opengroup.org/onlinepubs/009695399/functions/rand.html)() % 10)

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

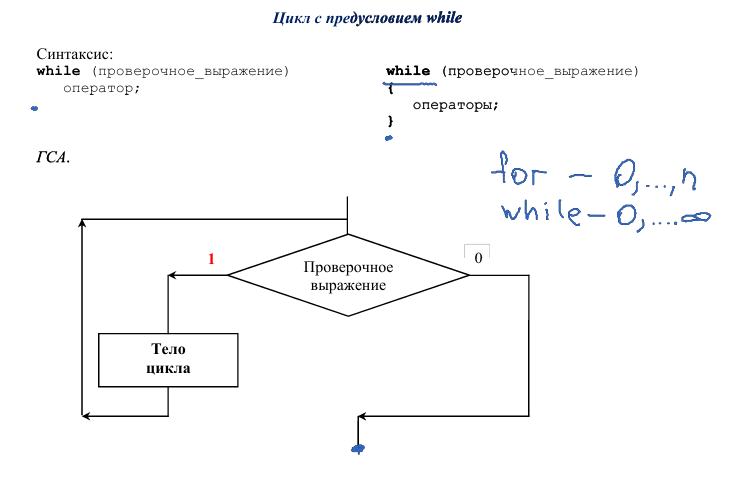
         return 0;

}

Как видите, в блоке инициализации формируется первое псевдослучайное значение в диапазоне [0; 9], а в блоке изменения следующее псевдослучайное значение из этого же диапазона. На каждой итерации на экран выводится полученное числовое значение, пока x не станет равен нулю.

На этом мы завершим первое знакомство с оператором цикла for. На следующем продолжим эту тему и поговорим об операторе цикла с постусловием do-while.

Цикл с предусловием: назначение, ГСА. Пример.



# Оператор цикла while

Практический курс по C/C++: <https://stepik.org/course/193691>

Смотреть материал на [YouTube](https://www.youtube.com/watch?v=_xBLgSOGnU0&list=PLA0M1Bcd0w8w-mqVmBjt-2J8Z1gVmPZVz)| [RuTube](https://rutube.ru/video/1ae020d8fdbec1d6e28d10ae8cd4fb68/?playlist=539096)

На этом занятии начнем знакомиться с еще одной ключевой конструкцией – циклами. Вначале, давайте я на простом примере покажу, о чем идет речь. Представим, что пользователь вводит с клавиатуры целое положительное значение n и нам нужно вычислить следующую сумму:

1^2 + 2^2 + 3^2 + 4^2 + ... + n^2

Понятно, что мы не можем заранее прописать всю эту сумму, так как значение n попросту неизвестно на момент написания программы. Переменная n принимает конкретное значение только в процессе выполнения программы. Поэтому реализовать такую конструкцию можно только с помощью операторов циклов.

В общем случае циклы позволяют реализовывать некие повторяющиеся действия. Например, предположим, что маленькие панды прыгают с горки в течение часа, пока мама-панда не позовет всех к столу – кушать. Формализовать это можно, например, так:

цикл (пока не прошел час):  
      прыгаем с горки

То есть, пока истинно условие, цикл работает, как только условие становится ложным (прошел час) цикл завершается. Ровно так работает цикл while, о котором и пойдет речь на этом занятии. Он имеет, следующее определение (синтаксис):

while(<условие>)  
    оператор;

или

while(<условие>) {  
    оператор 1;  
    ...  
    оператор N;  
}

В программировании оператор или блок операторов, выполняющихся в цикле, называют **телом цикла**. А один проход выполнения оператора цикла – **итерацией**. Сам цикл while относится к **циклам с предусловием**, т.к. сначала проверяется условие цикла и только после этого (при истинности условия) выполняется текущая итерация.

Давайте вернемся к исходной задаче – вычисления суммы квадратов целых чисел от 1 до n и посмотрим, как здесь можно использовать цикл while:

#include <stdio.h>

int main(void)

{

         int n;

         int s = 0;

         if([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &n) != 1) {

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

                   return 0;

         }

         while(n > 0) {

                   s += n\*n;

                   n--;

         }

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

         return 0;

}

Вначале мы объявили две переменные n и s, причем, в s будет храниться сумма и она принимает начальное значение 0. Затем, запрашивается ввод в переменную n и если он некорректен, то программа завершается. Иначе запускается цикл while. В круглых скобках прописано условие продолжения цикла: пока n больше нуля. А в теле цикла определены два оператора: сначала к переменной s прибавляем квадрат наибольшего значения n, а потом, уменьшаем переменную n на единицу. В результате у нас будет образовываться следующая сумма:

s = n^2 + (n-1)^2 + … + 1^2

Как только значение n становится равным 0, условие цикла становится ложным и оператор while прекращает свою работу. Управление переходит к следующему оператору printf().

Давайте теперь несколько изменим нашу программу и запишем цикл while следующим образом:

         while(n-- > 0)

                   s += n\*n;

Будет ли это тем же самым или программа станет работать по другому? Проверим это. Запустим и введем число 4. Получим сумму:

s = 14

Очевидно это не вся прежняя сумма:

s = 1^2 + 2^2 + 3^2 + 4^2 = 30

У нас не хватает последнего слагаемого 4^2 = 16. Почему так произошло? Все просто. Сначала выполняется сравнение переменной n с нулем (4 > 0) и после этого операция декремента. То есть, когда выполнение программы переходит к оператору «s += n\*n;» тела цикла переменная n уже на единицу меньше. Возможно, некоторых из вас это удивит, так как операция декремента записана в постфиксной форме (после имени переменной), а значит, она должна выполняться в последнюю очередь. Но логика работы здесь несколько иная. Любое выражение, которое прописывается в условии, сначала полностью вычисляется и только затем осуществляется переход к телу цикла. И это всегда так. Этот важный момент нужно запомнить и знать. Именно поэтому переменная n гарантированно будет уменьшена на единицу при переходе к оператору «s += n\*n;».

Кстати, последнее слагаемое в нашем примере будет 0^2 = 0. Поэтому (если ноль нам не нужен) правильнее было бы прописать эту условие так:

         while(--n > 0)

                   s += n\*n;

Префиксная форма записи операции декремента сначала уменьшит значение n на единицу и только после этого будет осуществляться сравнение с нулем.

Давайте теперь немного усложним программу и сделаем ограничение на максимальное значение n на тот случай, если пользователь введет слишком большое число. Например, сделаем так, чтобы слагаемых было не больше 10:

         int i = 0;

         while(++i <= n && i <= 10)

                   s += i\*i;

Условие в цикле while пришлось переписать с использованием дополнительной вспомогательной переменной i, которая, по сути, является счетчиком итераций. Ее первое значение при подсчете суммы будет равно 1, затем, 2 и так пока либо не дойдет до n, либо до 10.

Давайте внимательнее посмотрим на условие. Вначале записана операция инкремента в префиксной форме. Это означает, что значение переменной i увеличится на единицу и только потом выполнится операция сравнения. Мало того, здесь мы можем точно гарантировать, что операция инкремента (в любой форме записи: префиксной или постфиксной) отработает до перехода к следующей проверке i <= 10. Так заложено в стандарте языка Си. И это правило справедливо для всех логических связок:

&& и ||

где бы они ни использовались: в операторах циклов или условных операторах.

Благодаря этому мы можем четко понимать, как отработает данный цикл. Вначале обязательно увеличится значение переменной i на единицу, затем, проверится условие i <= n и только после этого (при необходимости) будет проверяться второе подусловие i <= 10.

Вообще, в качестве условия цикла while можно записывать любое выражение. Например, мы хотим считывать целые числа с клавиатуры, пока пользователь не введет 0. Сделать это можно следующим образом:

#include <stdio.h>

int main(void)

{

         int s = 0;

         int x = 1;

         while([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &x) == 1 && x != 0)

                   s += x;

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

         return 0;

}

Здесь цикл while будет работать до тех пор, пока пользователь либо вводит корректные данные (целые числа), либо введет число 0. В самом цикле выполняется суммирование введенных числовых значений, а после цикла отображение полученного результата с помощью функции printf().

Или можно немного усложнить пример и выполнять суммирование только четных чисел (или только положительных). Для этого достаточно внутри цикла while указать нужное нам условие, например, следующим образом:

         while([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &x) == 1 && x != 0)

                   if(x % 2 == 0)

                            s += x;

То есть, в теле цикла while можно прописывать абсолютно любые операторы языка Си. И, обратите внимание, в данном случае мы не указывали фигурные скобки, т.к. в теле цикла формально записан один оператор if, а для одного оператора фигурные скобки не обязательны. Хотя, при желании их можно записать:

         while([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &x) == 1 && x != 0) {

                   if(x % 2 == 0)

                            s += x;

         }

Программа от этого не изменится.

В заключение занятия отмечу один важный момент использования операторных фигурных скобок. Обычная переменная, объявленная внутри них, например:

         while([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &x) == 1 && x != 0) {

                   int res = x % 2;

                   if(res == 0)

                            s += x;

         }

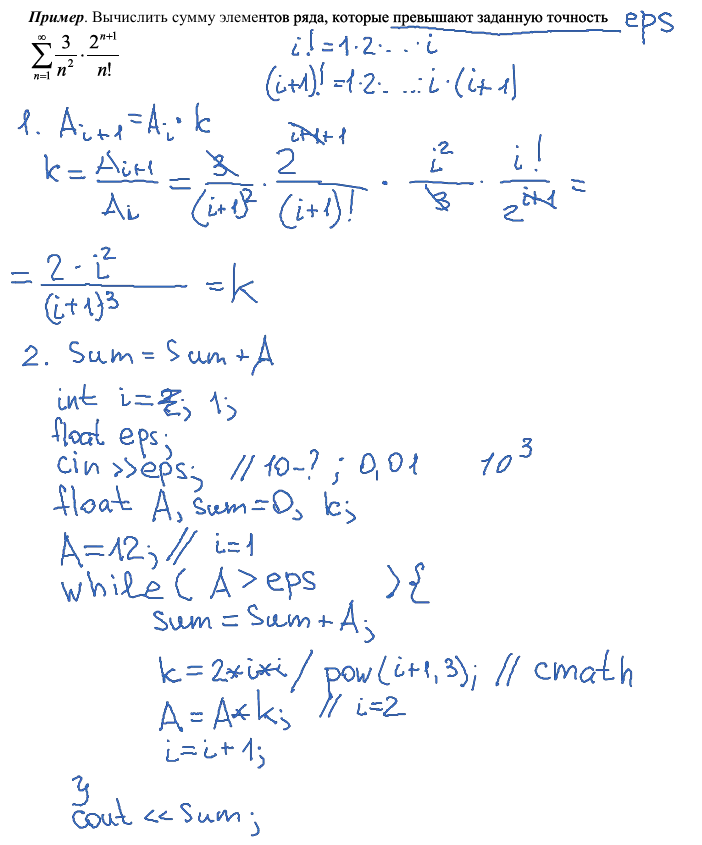
существует только в пределах этих операторных скобок и перестает существовать, когда выполнение программы перейдет к следующему оператору за пределами этих фигурных скобок. То есть, если мы захотим вывести значение переменной res после цикла while:

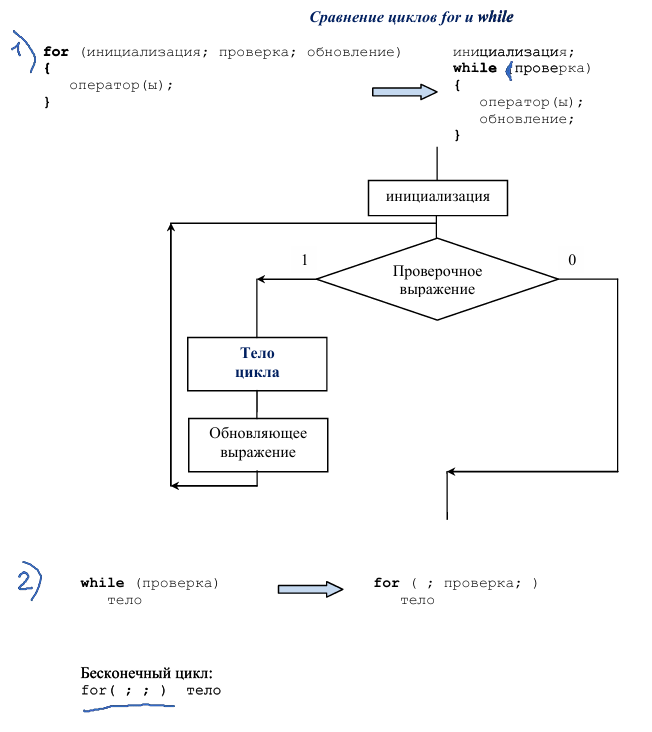
[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("res = %d**\n**", res);

то при компиляции возникнет ошибка, что переменная res не определена.

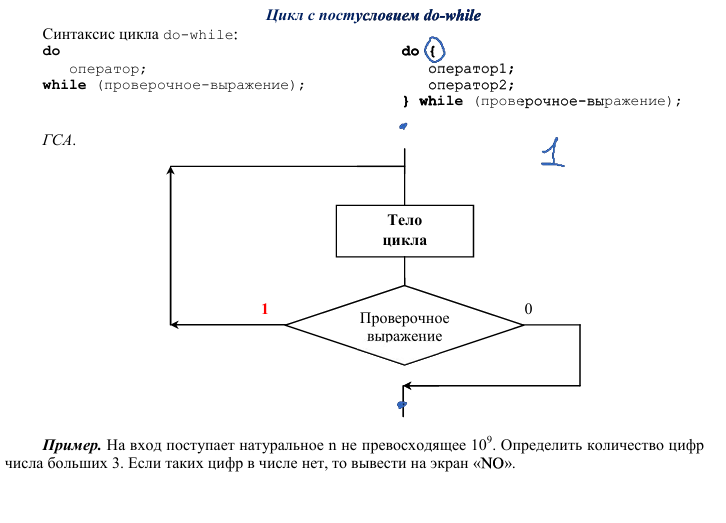
На самом деле, это очень полезное поведение, заложенное в язык Си. На практике часто внутри фигурных скобок объявляют временные переменные, необходимые для отработки внутренней логики. Потом они просто автоматически исчезают и не мешают работе остальной программы. Обо всем этом мы еще подробнее будем с вами говорить, когда речь пойдет о локальных и глобальных переменных.

На этом завершим первое занятие по циклам. Из него вам должно быть понятно для чего вообще нужны циклы и как работает оператор цикла while.





Цикл с постусловием: назначение, ГСА. Пример.



На этом занятии рассмотрим последний такой оператор – цикл с постусловием do-while, который имеет следующий синтаксис:

do {  
    [оператор (один или несколько)]  
} while(<условие>);

Фигурные скобки при записи этого оператора строго обязательны, даже если в них будет находиться всего один оператор. Также после ключевого слова whileдолжна стоять точка с запятой.

Само название этого оператора говорит, что цикл do-whileотличается от цикла while, фактически, только тем, что сначала выполняется итерация, а затем, проверяется условие для продолжения цикла. Приэтом цикл do-whileработает пока истинно условие. Соответственно, как только условие становится ложным, цикл завершается и управление передается следующему оператору в программе.

Давайте рассмотрим пример, где использование такого типа цикла оправдано. Предположим, пользователь должен ввести кодовое число, чтобы получить доступ к чему-либо. Для этого вначале нужно попросить его ввести числовое значение и только после этого проверить, совпадает ли ввод с кодовым числом. Если совпадает, то предоставить доступ, а иначе попросить ввести код еще раз. Эту логику проще всего реализовать с помощью оператора цикла do-while следующим образом:

#include <stdio.h>

int main(void)

{

         int pass\_code = 13;

         int enter\_code;

         do {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Please enter the secret code: ");

[scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &enter\_code);

         } while(enter\_code != pass\_code);

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Access is allowed**\n**");

         return 0;

}

Сначала будет выполнена итерация цикла, то есть, операторы, записанные в теле цикла. В результате пользователь получает возможность ввести кодовое значение. А затем, в цикле выполняется обратная проверка на неравенство значений. Почему так? Потому что мы здесь прописываем условие продолжения цикла. И цикл должен продолжаться, пока пользователь не введет верное кодовое число. В этом случае условие становится ложным и управление передается следующему оператору printf(), записанному после цикла do-while.

Как видите, цикл с постусловием полезен, когда нам в программе нужно вначале выполнить некоторые повторяющиеся действия и только после этого решить, продолжать цикл или завершить. В действительности, это не частая ситуация и по этой причине оператор цикла do-whileна практике применяется реже двух остальных: while и for, о которых мы с вами уже говорили на предыдущих занятиях.

Также обратите внимание, как записан в тексте программы этот оператор цикла. Ключевое слово whileстоитв одной строчке с закрывающейся фигурной скобкой.Так принято делать, чтобы визуально подчеркнуть, что while здесь – это завершение конструкции, которая начинается с ключевого слова do. И чтобы не спутать этот whileс другим таким же оператором уже цикла с предусловием. Например, давайте усовершенствуем нашу программу, и будем очищать буфер входного потока stdinпосле каждого ввода:

#include <stdio.h>

int main(void)

{

         int pass\_code = 13;

         int enter\_code;

         int c;

         do {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Please enter the secret code: ");

[scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &enter\_code);

                   while ((c = [getchar](http://www.opengroup.org/onlinepubs/009695399/functions/getchar.html)()) != '**\n**' && c != EOF) { }

         } while(enter\_code != pass\_code);

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Access is allowed**\n**");

         return 0;

}

Я здесь использую цикл whileс вызовом функции getchar(), которая читает один байт из потока stdin. Цикл останавливается, если прочитан символ переноса строки (\n) или конца файла (EOF). Это простейший вариант очистки буфера входного потока. И, смотрите, в тексте программы идут друг за другом два слова while. Но, тем не менее, они легко различаются, так как имеются отступы у операторов тела цикла.

Или может быть ситуация, когда после цикла do-whileидет цикл while:

#include <stdio.h>

int main(void)

{

         int pass\_code = 13;

         int enter\_code;

         int c;

         do {

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Please enter the secret code: ");

[scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &enter\_code);

                   while ((c = [getchar](http://www.opengroup.org/onlinepubs/009695399/functions/getchar.html)()) != '**\n**' && c != EOF) { }

         } while(enter\_code != pass\_code);

         while ((c = [getchar](http://www.opengroup.org/onlinepubs/009695399/functions/getchar.html)()) != '**\n**' && c != EOF) { }

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Access is allowed**\n**");

         return 0;

}

При нашем способе оформления текста программы все три цикла хорошо различимы и не возникает никакой путаницы. Именно поэтому рекомендуется оператор do-whileзаписывать в таком виде.

## Вложенные циклы

Из последних примеров вы видели, что в один оператор цикла можно заключать другой. В программировании такая конструкция называется вложенными циклами. И ее можно реализовывать с любыми операторами циклов. Давайте подробнее посмотрим, как она работает.

Чаще всего вложенные циклы можно увидеть с операторами for. Поэтому в нашем примере будут фигурировать именно они. А сам пример будет следующим. Представим, что нам нужно выполнить полный перебор документов (файлов), лежащих в нескольких ящиках (box). В каждом ящике одинаковое число файлов, а всего ящиков пусть будет три.

Для решения этой задачи воспользуемся вложенными циклами for следующим образом:

#include <stdio.h>

int main(void)

{

         int total\_boxs = 3;

         int total\_files = 6;

         for(int i = 0;i <total\_boxs; ++i)

                   for(int j = 0; j <total\_files; ++j)

[printf](http://www.opengroup.org/onlinepubs/009695399/functions/printf.html)("Box %d, file %d**\n**", i+1, j+1);

         return 0;

}

Смотрите, сначала срабатывает первый (внешний) цикл со счетчиком (переменной) i, который содержит номер текущего просматриваемого ящика. Затем, запускается итерация этого внешнего цикла (выполняются операторы, записанные в теле цикла). В результате запускается второй (внутренний) цикл, который перебирает документы текущего ящика. Для каждого документа выводится информация в формате:

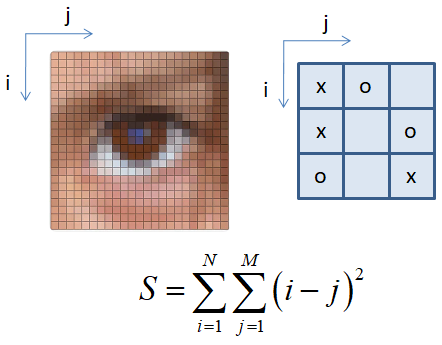
Box<номер ящика>, file<номер документа>

В результате, после просмотра всех документов первого ящика на экране появятся строчки:

Box 1, file 1  
Box 1, file 2  
Box 1, file 3  
Box 1, file 4  
Box 1, file 5  
Box 1, file 6

После этого работа внутреннего цикла завершается и управление передается внешнему циклу. В нем происходит увеличение счетчика i на единицу и проверяется условие цикла. Так как оно истинно, то запускается вторая итерация и снова срабатывает внутренний цикл, но уже для перебора документов второго ящика. И так для всех трех ящиков.

Вот общий принцип работы вложенных циклов. Конечно, пример с ящиками и файлами – это лишь частный пример. В общем случае подобным образом можно перебирать пиксели двумерного изображения, ячейки игрового поля, вычислять вложенные суммы и так далее.



Мало того, при необходимости, мы можем увеличивать число вложений операторов циклов и осуществлять перебор элементов не только в плоскости, но и в объеме и вообще в Nмерном пространстве. Хотя, на практике слишком увлекаться этим не стоит, т.к. сложность понимания и восприятия программы растет с числом этих вложений. Обычно, ограничиваются двумя, тремя уровнями. При правильной организации логики программы этого оказывается вполне достаточно.

Чтобы закрепить этот материал предлагаю самостоятельно написать программу для вычисления следующей суммы:

https://proproprogs.ru/htm/c_base/files/c_cikl-do-while-s-postusloviem-vlozhennye-cikly.files/image003.png

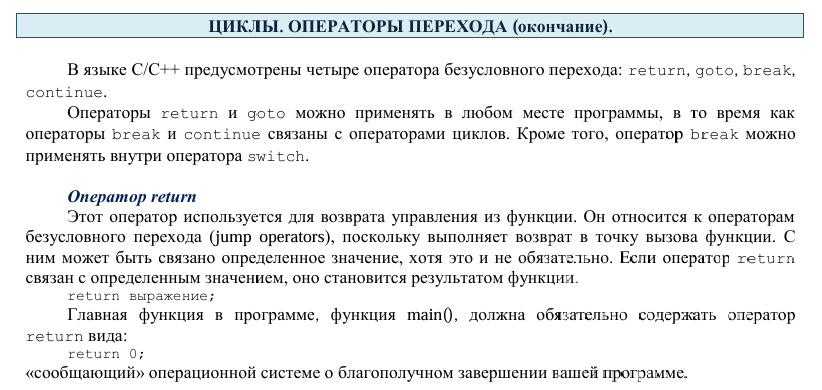
Если все правильно сделаете, то должны получить ответ 2750.

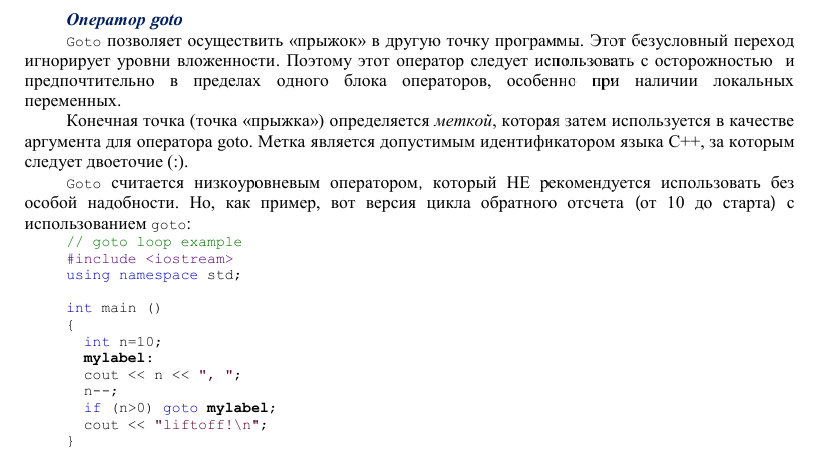
Представление цикла с параметром for через цикл с предусловием. Пример.

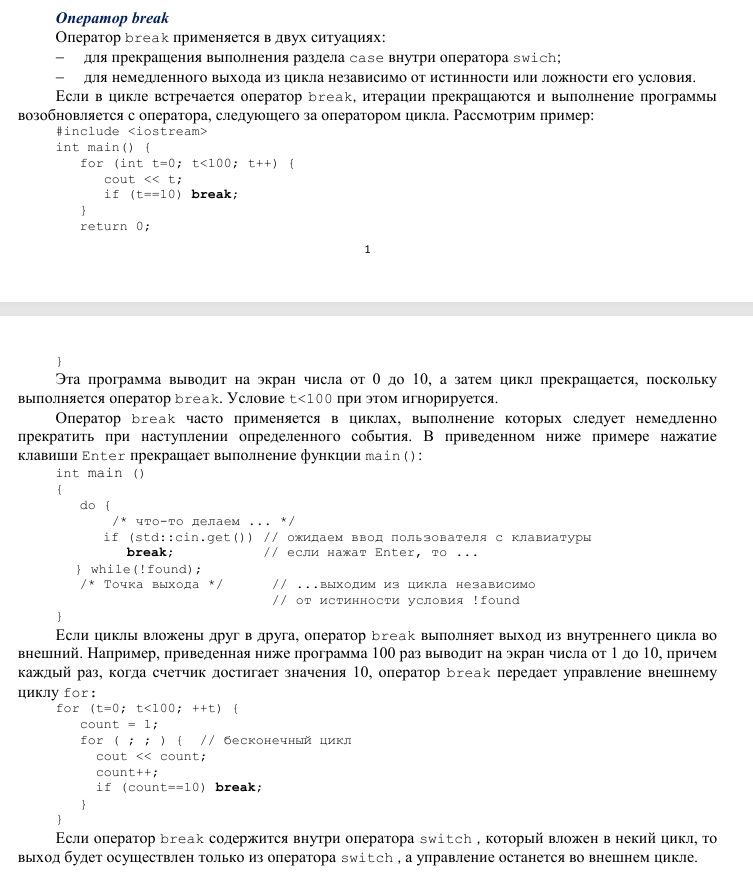
Представление цикла с параметром for через цикл с постусловием. Пример.

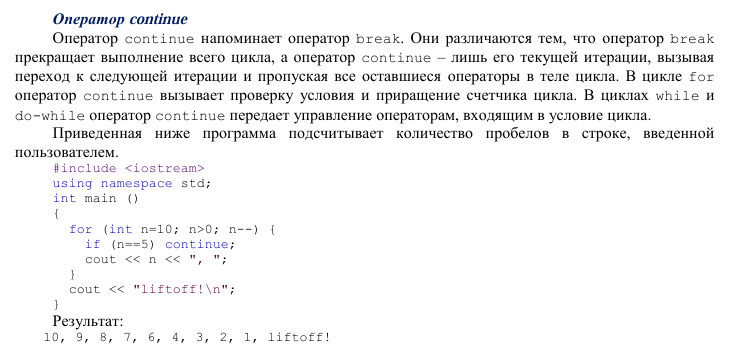
Исключение зацикливания.

Операторы перехода: return, break, continue. Примеры.









На прошлых занятиях мы подробно разобрали работу трех операторов цикла: while, for и do-while. Также увидели как можно формировать вложенные циклы и примеры задач, где это может пригодиться. На этом заключительном занятии по циклам разберем следующие управляющие операторы, которые можно использовать в любом операторе цикла:

* break – досрочное прерывание работы оператора цикла;
* continue – пропуск итерации тела цикла;
* goto – безусловный переход по метке.

## Оператор break

Начнем с оператора break. Предположим, нам нужно написать программу подсчета суммы вещественных чисел, пока пользователь вводит положительные числа. Как только он введет отрицательное значение, подсчет суммы завершается с выводом результата на экран. Такую программу можно реализовать следующим образом:

#include <stdio.h>

int main(void)

{

         double x, s = 0.0;

         while([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%lf", &x) == 1) {

                   if(x < 0)

**break**;

                   s += x;

         }

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

         return 0;

}

Вначале объявляются две вспомогательные переменные: x – для ввода текущего вещественного значения; s – для подсчета суммы вещественных положительных чисел. Далее запускается цикл while, который работает, пока вводятся корректные числовые значения. В теле цикла идет проверка, если введенное значение x отрицательное, то срабатывает оператор break. Этот оператор завершает работу цикла while и управление передается следующему оператору printf(). Если же введенное число неотрицательное, то выполняется подсчет суммы.

Как видите, все предельно просто. Для примера, если в этой программе убрать условие и записать оператор break в теле цикла, то цикл while гарантированно будет прерываться на первой же итерации.

Конечно, эту же программу можно было бы записать и без оператора break, изменив условие цикла:

         while([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%lf", &x) == 1 && x >= 0) {

                   s += x;

         }

Возможно, даже любую программу можно составить без оператора break. Но его использование может заметно облегчить написание и понимание логики программы. Именно с этой целью он и введен в язык Си, да и во многие другие языки высокого уровня.

Отмечу здесь еще один распространенный способ прерывания работы цикла. Если вместо break записать оператор return:

         while([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%lf", &x) == 1) {

                   if(x < 0)

                            return 0;

                   s += x;

         }

то для цикла while эффект будет такой же, как и от оператора break. Но, правда, есть и существенное отличие. Как мы уже знаем, оператор return завершает выполнение функции. В данном случае функции main(). Поэтому, как только пользователь введет отрицательное число, то программа просто завершится, результат на экран выведен не будет, так как все операторы после while уже не сработают. В этом ключевое отличие между break и return. Оператор break позволяет продолжить выполнение функции, и управление передается следующему оператору после цикла. А при return текущая функция завершается сразу. На самом деле, оператор return завершает работу функции и лишь, как следствие, прерывает работу цикла. По той же логике оператор return можно для завершения работы оператора switch, о котором мы с вами ранее уже говорили. И вообще, с помощью return можно прервать работу чего угодно в пределах функции.

## Оператор continue

Следующий оператор continue прерывает лишь текущую итерацию цикла. Давайте я покажу принцип его работы на конкретном примере. Пусть мы также вводим с клавиатуры целые числа и хотим вычислить сумму только нечетных чисел. Сделать это можно следующим образом:

#include <stdio.h>

int main(void)

{

         int x, s = 0;

         while([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &x) == 1 && x != 0) {

                   if(x % 2 == 0)

                            continue;

                   s += x;

         }

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

         return 0;

}

Начало вам уже знакомо, а дальше записан цикл while, который работает пока пользователь вводит корректные данные или не введет число 0. Затем, в теле цикла стоит проверка на четность введенного числа x и если оно четно, то выполняется оператор continue. Как только встретился этот оператор, все остальные после него операторы тела цикла пропускаются и мы переходим к следующей итерации цикла while. В результате, сумма будет вычисляться только для нечетных значений.

Конечно, опять же, эту программу можно было бы реализовать и без оператора continue, например, так:

         while([scanf](http://www.opengroup.org/onlinepubs/009695399/functions/scanf.html)("%d", &x) == 1 && x != 0) {

                   if(x % 2 != 0)

                            s += x;

         }

И это вполне рабочий вариант. Здесь все как и с оператором break. Оператор continue введен для удобства программирования и его следует использовать тогда, когда он позволяет упростить логику программы и сделать ее более читабельной.

Оператор continue, равно как и оператор break можно использовать не только в цикле while, но и в циклах for и do-while. Например, если нам нужно из диапазона целых чисел [1; 100] отобразить только те, что кратны 3 и 5 одновременно, то это можно сделать следующим образом:

#include <stdio.h>

int main(void)

{

         for(int i = 1; i <= 100; ++i) {

                   if(i % 3 != 0 || i % 5 != 0)

                            continue;

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

         }

         return 0;

}

Это учебный пример. В этой задаче использование оператора continue избыточно и проще было бы вызывать функцию printf() по обратному условию. Я лишь показываю, что операторы continue и break допустимо применять с любыми операторами циклов.

## Оператор goto

Последний оператор, который мы рассмотрим на этом занятии, это goto. Он позволяет передать управление любому другому оператору по указанной метке в пределах текущей функции (в пределах той функции, в которой вызывается данный оператор).

На многих обучающих курсах и учебной литературе по языку Си сразу оговаривается, что оператор goto не следует использовать в своих программах, особенно, начинающим программистам. И я с этим полностью согласен! Но, существует, по крайне мере, одна ситуация, когда применение этого оператора оправданно. Это прерывание работы вложенных циклов. Давайте я детально поясню о чем здесь речь, а позже отмечу, почему все же оператор goto лучше не использовать в своих программах.

Итак, предположим, что мы вычисляем двойную сумму вида:

https://proproprogs.ru/htm/c_base/files/c_operatory-break-continue-i-goto.files/image001.png

и хотим прервать вычисления как только слагаемое i-j будет больше нуля. Такую программу можно реализовать следующим образом:

#include <stdio.h>

int main(void)

{

         int s = 0;

         for(int i = 1; i <= 10; ++i)

                   for(int j = 7; j >= 5; --j) {

                            if(i-j > 0)

                                      goto exit\_sum;

                            s += i - j;

                   }

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

         return 0;

}

Смотрите, здесь два цикла, один вложен в другой. В теле второго цикла выполняется проверка на положительность значения i-j. Если это так, то по условию задания нам нужно прервать работу обоих циклов и завершить подсчет суммы. Как раз для этого удобно воспользоваться оператором goto и по метке exit\_sum (имя мы придумываем сами) переходим к оператору printf(), стоящему после этих циклов. В результате такого перехода работа циклов будет завершена.

Возможно, некоторые из вас сейчас в недоумении смотрят на эту программу и задаются вопросом: а почему бы здесь не использовать оператор break, он же как раз и позволяет прервать цикл? Все верно, цикл (один цикл) он может прервать и если записать программу в виде:

         for(int i = 1; i <= 10; ++i)

                   for(int j = 7; j >= 5; --j) {

                            if(i-j > 0)

**break**;

                            s += i - j;

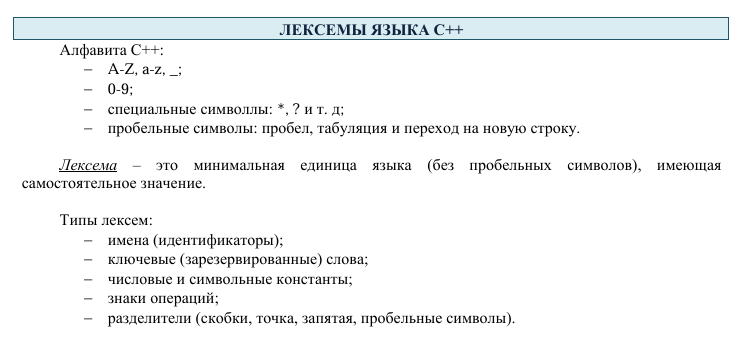
                   }

то по условию завершится работа внутреннего цикла, но внешний продолжит работать и на новой итерации опять запустит внутренний цикл. Очевидно, это не то, что нам нужно. Запомните, оператор break прерывает только один текущий цикл и не более того. Начинающие программисты здесь часто делают ошибку. Поэтому в нашей задаче вместо break используется оператор goto и, как мы видели, он позволяет прервать и два и три и более вложенных циклов при переходе по метке за их пределы, но в пределах текущей функции.

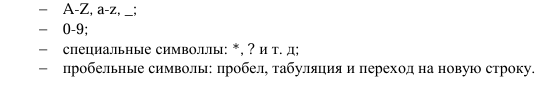
Это тот редкий случай, когда применение этого оператора оправданно. Но почему тогда к этому оператору такое негативное отношение со стороны программистского сообщества языка Си? Все очень просто. Если злоупотреблять использованием этого оператора, то логика работы программы очень быстро становится очень запутанной, настолько, что даже сам автор перестанет в ней ориентироваться. Соответственно читать и редактировать такие программы чрезвычайно сложно. Второй момент связан с компиляцией. Если программа написана без операторов goto, то, скорее всего, при более-менее грамотном подходе к программированию, на выходе получим качественный машинный код, который будет выполняться предельно быстро. Если же в программе присутствуют постоянные переходы goto, то никто не даст гарантии качественной трансляции такого текста в машинный код. Соответственно, главный плюс языка Си сходит на нет: исполняемый файл не оптимизирован для быстрого выполнения на текущей архитектуре компьютера.

Чтобы гарантированно избежать этих недостатков при построении программ, стоит совсем отказаться от применения оператора goto. Исключение может составлять необходимость прерывания сразу нескольких вложенных циклов. Хотя и здесь мы могли бы обойтись без этого оператора. Часто логику вложенных циклов реализуют в отдельных функциях. И как только появляется необходимость прервать их работу, завершают функцию оператором return. Это более частый вариант по сравнению с оператором goto. Именно так я рекомендую поступать при проектировании своих программ.

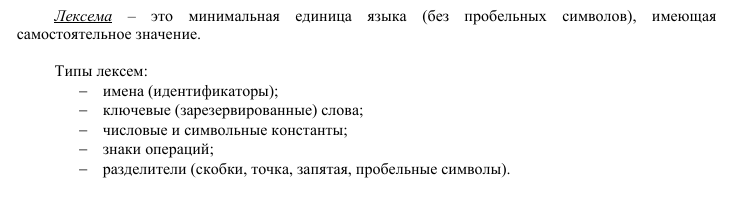
1. Основные понятия языка С++.



Алфавит языка С++.



Лексемы языка С++: определение, типы.



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

Первое, с чего следует начать – это познакомиться со способами хранения данных. На уровне программы они могут быть представлены в двух видах:

* переменные – изменяемые значения;
* константы и литералы – неизменяемые значения.

Начнем с переменных. Как мы с вами уже говорили, данные в памяти компьютера хранятся в виде неделимых ячеек в один байт, которые образуют непрерывную последовательность определенной длины:



В современных компьютерах каждой запущенной программе отводится своя независимая рабочая область памяти. И адреса ячеек становятся уже не физическими, то есть, реальными, а логическими – теми, которые определит для программы ОС. Это сделано для повышения надежности работы сразу нескольких программ в один и тот же момент времени. Но нам, как программистам, важно лишь, что есть ячейки в один байт, каждая ячейка имеет свой адрес (логический или физический – не имеет большого значения) и все они следуют друг за другом в пронумерованном порядке. Причем, адреса всегда увеличиваются на единицу при переходе к следующей ячейке памяти.

Так вот, в самом простом случае, мы можем в любую доступную для записи ячейку, поместить значение в виде целого числа из диапазона [0; 255]. Кратко напомню, что этот диапазон определяется тем, что один байт состоит из восьми бит, а каждый бит кодирует информацию двумя состояниями: 0 или 1. Получаем число возможных комбинаций 2^8 = 256. Значит, в один байт можно записывать произвольные целые числа в диапазоне от 0 до 255 включительно. Чтобы в программе на языке Си работать с отдельной ячейкой памяти, следует воспользоваться такой конструкцией:

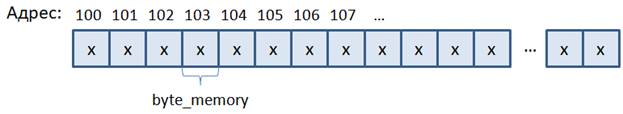
char byte\_memory;

Здесь char – это тип переменной; byte\_memory – имя переменной. Именно так следует объявлять переменные в языке Си: сначала указывается тип данных для переменной, а затем, ее имя. На уровне синтаксиса это выглядит следующим образом:

<тип переменной> <имя переменной>;

Обратите внимание на точку с запятой в конце объявления переменной. Точка с запятой в языке Си является неотъемлемой частью оператора, а не разделителем между операторами, как иногда полагают. В данном случае символ точка с запятой указывает компилятору, где заканчивается операция объявления переменной и превращает запись в полноценный, законченный оператор. Далее мы с вами увидим, как символ точка с запятой входит в состав синтаксиса оператора цикла for и там она играет свою особую роль. А на данный момент нужно просто запомнить, что запись оператора объявления переменной (или нескольких переменных) должна завершаться точкой с запятой.

Но вернемся к типу char. Он описывает хранилище данных размером в один байт и позволяет хранить 256 вариаций из целых чисел. А имя переменной – это, по сути, название хранилища, где расположено некоторое целое число.



На рисунке переменная byte\_memory условно находится в 103-й ячейке памяти. В действительности, расположение переменных определяет **менеджер памяти** ОС. Когда в программе объявляется очередная переменная, то ОС располагает ее в найденной свободной последовательности ячеек. И, как вы понимаете, в общем случае, адрес хранения переменной может быть самым разным.

Но на уровне программы для нас это не имеет особого значения. Мы всегда легко можем обратиться к хранилищу данных по имени переменной, прочитать или записать туда нужные значения. Делается это с помощью **операции присваивания** следующим образом:

byte\_memory = 100;

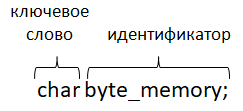
char a;

a = byte\_memory;

В конце каждого оператора ставится точка с запятой. Об операции присваивания мы с вами еще подробнее будем говорить. Здесь лишь отмечу, что в левый операнд заносится значение, записанное в правом операнде. То есть, при выполнении первой строчки, в переменную byte\_memory будет занесено целое значение 100. В нашем случае, это автоматически означает, что ячейка с адресом 103 будет содержать число 100. В этом и есть смысл переменных: они могут менять свои значения в процессе выполнения программы.

В двух следующих строчках происходит объявление еще одной байтовой переменной с именем a и **копирование** в эту переменную значения из переменной byte\_memory. Обратите внимание, я употребил слово «копирование». Так как в языке Си переменные – это непосредственно хранилища данных, поэтому в момент присваивания происходит копирование информации из одного хранилища в другое. То есть, переменные a и byte\_memory – совершенно независимы между собой.

Я думаю, что вы в целом поняли, что из себя представляют переменные языка Си, как можно объявлять байтовые переменные, заносить и считывать из них данные. Дополнительно укажу лишь некоторые терминологические моменты.



Тип char относится к одному из ключевых, то есть, зарезервированных слов языка Си. В программе оно воспринимается компилятором исключительно как байтовый тип переменной и не может быть использовано в каком-либо другом качестве, например, как имя переменной.

Далее, имя переменной также еще называют идентификатором, то есть, имя как бы идентифицирует область памяти, которую представляет. Забегая вперед, отмечу, что идентификаторами также являются имена функций и некоторых других конструкций языка Си. Применительно к имени переменной, в соответствии со стандартом С99, ее максимальная длина внутри модуля составляет 63 символа. Конечно, можно задавать имена и длиннее 63 символов, но компилятор будет учитывать только первые 63. Если же переменная используется за пределами модуля, где она объявлена, то ее максимальная длина составляет 31 символ. Это следует учитывать, хотя размер переменных более чем достаточен, чтобы не переходить указанные границы.

Теперь, как правильно задавать имена переменных. Так как они представляют собой хранилища, то они должны быть существительными, например:

data, name, total, count, size, fl\_view и т.п.

Следует иметь в виду, что язык Си различает большие и малые буквы в названиях переменных, да и вообще любых идентификаторов. Поэтому переменные:

Comp, comp, COMP, cOmp

- это разные переменные. В их именах можно использовать только символы латинского алфавита, цифры и символ подчеркивания:

a-z, A-Z, 0-9, \_

Причем, в качестве первого символа может идти или буква или символ подчеркивания. Цифра может появляться только, начиная со второго символа в названии переменной.

Вот примеры правильных и неправильных имен:

|  |  |
| --- | --- |
| **Правильные имена** | **Неправильные имена** |
| total\_digits | 1\_house |
| a | don’t |
| b | K&R |
| fl\_mine | 100 |
| \_arg | find@10 |
| \_\_Apple1 | si# |
| Count\_Views100 | array[] |

Вообще, принято имена переменных записывать малыми буквами латинского алфавита без первого символа подчеркивания. Например, так:

swift, book\_si, text\_array, counter и т.д.

Именно такого написания я буду придерживаться в данном курсе.

Думаю излишне говорить, что имена переменных должны быть уникальными в пределах одной области видимости.

## Базовые типы данных

Итак, мы с вами использовали базовый тип char, который описывает один байт памяти устройства. Конечно, в языке Си есть и другие встроенные типы данных. Они следующие:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Тип данных** | **Для 16-битных** | **Для 32-битных** | **Для 64-битных** | **Описание** | **Диапазон** |
| char | 1 | 1 | 1 | Символьный и целочисленный | [0; 255] или  [-128; 127] |
| short [int] | 2 | 2 | 2 | Целочисленный | [-32768; 32767] |
| int | 2 | 4 | 4 | Целочисленный | [-32768; 32767]  или  [-2 147 483 648; 2 147 483 648] |
| long [int] | 4 | 4 | 8 | Целочисленный | … |
| long long [int] | 8 | 8 | 8 | Целочисленный | … |
| float | 4 | 4 | 4 | Вещественный | … |
| double | 8 | 8 | 8 | Вещественный | … |
| long double | 10 | 10 | 10 | Вещественный | … |

Тип long long помечен оранжевым, так как он поддерживается не всеми компиляторами. В стандарте С99 были добавлены некоторые другие типы, но они не часто используются на практике, поэтому я не стал их приводить, чтобы не перегружать материал.

Первое, что бросается в глаза, глядя на эту таблицу – это плавающий размер типов в зависимости от разрядности процессора и ОС. Да и они приведены лишь, как некий ориентир. В действительности стандарты языка Си никак не оговаривают конкретные размеры типов данных. А значит, компиляторы могут использовать такие, какие заложит в них разработчик. Конечно, большинство компиляторов согласуют размеры как в приведенной  таблице. Но гарантии никакой нет. И, конечно же, здесь возникает закономерный вопрос: почему бы эти размеры строго не определить и договориться использовать вполне определенные значения? Но на это есть одно веское основание. Вспоминаем, язык Си разработан как заменитель языка Ассемблер и программы на Си должны наилучшим образом переводиться в машинный код. А машинный код жестко привязан к архитектуре компьютера: разрядности регистров процессора, шины и ячеек памяти. Например, тип char введен для работы с отдельными ячейками памяти. Если в компьютере они составляют 8 бит, то и компилятор будет описывать тип char восьмью битами. Если же представить вычислительную технику с ячейками в 12 бит, то логично тип char также сделать 12-битным. Вот почему размеры типов жестко не регламентируются спецификациями языка Си. И так со всеми остальными. Например, тип int – это основной среди целочисленных. Почему так? Потому что вплоть до появления 64-битных процессоров он совпадал по размеру с машинным словом процессора. Поэтому в эпоху 16-битных процессоров int составлял 16 бит (2 байта), при 32-битных – 32 бита (4 байта). А вот при появлении 64-битных он остался 32-битным. Опять же, казалось бы, давайте его по логике увеличим до 64 бит? Но здесь возникла другая проблема. Если int сделать 64-битным, то у нас не будет целочисленного типа в 32 бита. Вводить новый было бы неразумно, т.к. существует масса ранее уже написанных программ с использованием только таких типов данных. А компиляторы стараются создавать так, чтобы программы на Си были переносимыми, то есть, чтобы их можно было впоследствии перекомпилировать под другую платформу или для другого процессора. Все это и обусловливает такую неопределенность типов данных.

Но одно мы все же можем сказать. Относительное распределение размеров типов следующее:

char <= short <= int <= long <= long long

float <= double <= long double

Но более ни на что рассчитывать не приходится. Хотя, как я уже сказал, большинство компиляторов для стандартной вычислительной техники, придерживаются размеров, указанных в таблице.

Итак, примем как данность неопределенность размеров типов и будем ориентироваться на данные из таблицы. В ней мы видим еще такую запись:

short [int] или long [int] или long long [int]

Что означает int в квадратных скобках? Это значит, что формально эти типы можно записывать как с int в конце, так и без него, например:

short int a;

short b;

long d;

long int m;

Причем, от добавления слова int тип данных никак не меняется, поэтому в современной практике программирования на Си это ключевое слово совместно с модификаторами short и long не указывают, а пишут просто:

short b;

long d;

Зачем тогда вообще появились эти сочетания short int, long int? Все это пришло из глубокой древности языка Си. Вероятно, Деннис Ритчи изначально задумывал один общий целочисленный тип int и два модификатора к нему: short и long, чтобы иметь возможность определять более короткие и длинные по размеру памяти целочисленные переменные. Но, так как short и long, кроме как к целочисленным переменным не применяется (кроме, разве что long double, который появился позже), то особого смысла дописывать int просто нет.

## Модификаторы signed и unsigned

Вообще подобные «неряшливые» моменты время от времени встречаются и их нужно правильно понимать и учитывать при написании программ, особенно, когда программа пишется под заказ. Поэтому важно отметить еще одну такую «неряшливость», связанную с типом char.

Когда мы в программе задаем переменную этого типа:

char fl\_view;

то в зависимости от компилятора она может принимать или целые числа в диапазоне [0; 255], или целые числа в диапазоне [-128; 127]. Стандартом это также никак не оговорено. Откуда взялись эти диапазоны? Первый (от 0 до 255) – это, по сути, беззнаковое представление целого числа в одном байте. А второй – это те же 256 вариаций, но для представления целых чисел со знаком. Спрашивается, как же нам в программе гарантированно описать переменную fl\_view как беззнаковую или, наоборот, как знаковую? Для этого в языке Си существуют два модификатора:

* unsigned – беззнаковый;
* signed – знаковый (появился в стандарте ANSI C90).

Если при объявлении явно указать:

unsigned char fl\_view;

то переменная fl\_view будет описывать целые числа в интервале [0; 255]. Если же указать:

signed char fl\_view;

то переменная fl\_view будет описывать целые числа со знаком в интервале [-128; 127].

А теперь, внимание! Все остальные целочисленные типы (short, int, long, long long) по умолчанию всегда идут как знаковые. Поэтому определение вида:

int arg;

создает переменную arg как целочисленную и знаковую. И так со всеми другими типами, кроме char. Кстати, именно по этой причине в языке Си появился модификатор signed, чтобы гарантированно объявлять переменную типа char со знаком. У других целых типов модификатор signed указывать не имеет смысла, он идет по умолчанию. А вот второй модификатор unsigned вполне имеет смысл. Например:

unsigned short d;

Получаем целочисленную беззнаковую переменную с именем d. И так со всеми целыми типами.

Обратите внимание, когда я говорю о модификаторах signed и unsigned, то отмечаю именно целочисленные типы данных. Дело в том, что вещественные типы float и double всегда знаковые и применять к ним эти модификаторы нельзя.

## Рекомендации по выбору типов для переменных

Итак, перед нами множество самых разных типов переменных. Спрашивается, как ориентироваться в этом многообразии и выбирать наиболее подходящий вариант? На этот счет есть довольно простые рекомендации.

Если нам требуется хранить отдельные символы или работать с отдельными ячейками памяти, то для этого существует только один тип **char**. Если предполагается оперировать целочисленными значениями, то изначально следует подумать о типе **int**. Этот тип нам может не подойти в двух ситуациях: если мы создаем в программе большое число целочисленных переменных (от 1000 и более), то следует посмотреть в сторону типа **short** в целях экономии памяти. Конечно, тип short можно использовать, только если его диапазона значений достаточно для представления данных в рамках решаемой задачи. Иначе, придется опять же обратиться к типу int, или даже типу long. Типы **long** или **long long** (если его поддерживает компилятор) следует использовать только в случае, если диапазон значений int недостаточно. Наконец, для описания вещественных чисел (чисел с плавающей точкой), следует изначально рассматривать тип **double**. И обращаться к типу **float** только если в программе предполагается создавать много вещественных чисел (от 1000 и более). Тип **long double** практически не используется за редким исключением сложных математических вычислений.

Итак, по результатам этого занятия вы должны хорошо себе представлять, как выбирать имена переменных, какие базовые типы данных существуют в языке Си и какого они бывают размера (в битах или байтах), как объявлять переменные и как правильно применять модификаторы signed и unsigned.

На этом занятии подробнее рассмотрим операцию присваивания, а также числовые и символьные литералы. Начнем с операции присваивания.

На предыдущем занятии мы с вами научились объявлять переменные. Например, так:

#include <stdio.h>

int main(void)

{

    int var;

    return 0;

}

Создается целочисленная переменная с именем var внутри функции main. Так как мы не присваивали ей пока никакого значения, то она будет содержать произвольные данные. Давайте следующей строчкой занесем в целочисленное хранилище var значение 100. Сделать это можно с помощью **операции присваивания**, которая в языке Си определена символом равно:

var = 100;

Обратите внимание, я говорю операция присваивания, а не оператор. Дело в том, что в языке Си присваивание реализовано как **выражение** и помимо копирования данных эта операция еще возвращает скопированное значение. В данном случае – это целое число 100. И только благодаря точки с запятой, записанной в конце, компилятор понимает, где заканчивается данная конструкция и начинается следующая. То есть, точка с запятой превращает операцию присваивания в оператор – завершенную конструкцию языка Си.

Как я уже отмечал на предыдущем занятии, операция присваивания копирует значение из правого операнда в левый операнд. Соответственно левый операнд должен явно или косвенно определять ячейки памяти, куда можно занести значение, указанное в правой части. Как говорят в программировании, слева должно стоять **модифицируемое лево-допустимое выражение** (l-value). В простейшем случае – это просто имя переменной. Но могут быть и другие конструкции, с которыми мы познакомимся в будущем.

Учитывая, что присваивание это операция, мы могли бы записать ее и в таком виде:

int var, size;

size = var = 100;

Смотрите, во-первых, в программе объявляются две целочисленные переменные, которые перечисляются через запятую после типа данных int. Так тоже можно делать и это вполне распространенная и допустимая практика объявления сразу нескольких переменных одного и того же типа. Во-вторых, в следующей строчке присваивается целое значение 100 сразу двум переменным. Такая запись допустима, как раз благодаря тому факту, что присваивание является операцией, а не законченным оператором. Вначале число 100 будет присвоено переменной var, а затем, возвращенное значение 100 присваивается переменной size. Точка с запятой в конце завершает эту конструкцию и определяет ее как оператор.

Конечно, эту же программу мы могли бы записать и так:

int var, size;

var = 100;

size = 100;

Современные компиляторы в обоих случаях сформируют оптимальный машинный код и корректно переведут эту программу в наборы машинных инструкций. Поэтому, какой вариант присваивания использовать решает сам программист, исходя из удобства оформления программы.

Сразу же отмечу еще один важный момент, связанный с объявлением переменных. Правило хорошего тона предполагает, что все значимые переменные следует объявлять вначале каждого блока операторов (то есть, фигурных скобок). А уже затем, выполнять операции с ними. Именно так сделано в нашей программе: сначала идет объявление двух переменных, а ниже описаны действия с ними. Благодаря этому упрощается понимание и редактирование программ. При необходимости, каждую переменную можно сопроводить поясняющим комментарием. Конечно, это относится только к значимым переменным. Любые другие, временные, можно объявлять в любом месте программы, чтобы визуально она не засорялась лишними деталями.

## Инициализация переменных

В языке Си, равно как и во многих других языках высокого уровня, можно определять значение переменной сразу в момент ее объявления. Например, так:

int total = 1024;

Визуально это выглядит как обычное присваивание переменной значения. Но в действительности – это совершенно другая операция, которая называется **инициализацией** переменной. То есть, когда мы объявляем какую-либо переменную (любого типа) и сразу указываем для нее некоторые начальные данные, то в этот момент запускается механизм инициализации этой переменной. Компилятор четко различает эти две ситуации: инициализация и обычное присваивание. На уровне машинных кодов они также могут быть реализованы по разному. Да и синтаксически, как мы в будущем увидим, при инициализации возможные конструкции, которые нельзя использовать при присваивании. А пока на данном этапе достаточно знать и запомнить, что есть процесс инициализации переменных, а есть присваивание данных и, в общем случае, это разные вещи.

Для полноты картины приведу еще один пример, когда мы комбинируем обычное объявление и инициализацию переменных:

int total = 1024, buffer;

Синтаксически так тоже можно делать и использовать в своих программах.

## Целочисленные и символьные литералы

Теперь давайте внимательнее посмотрим на число 100 в нашей программе. И зададимся вопросом, как его воспринимает компилятор и где оно хранится? В программировании явно прописанные числовые значения называются **числовыми литералами** и представляется как целочисленная константа типа int. Почему именно типом int? Так решил создатель языка Си Деннис Ритчи: все целочисленные константы, *записанные в программе в десятичном виде*, хранить на уровне типа int. Но только в том случае, если число умещается в диапазон этого типа. Напомню, что в современных 32- и 64-битных компьютерах тип int, как правило, составляет 4 байта (32 бита) и описывает диапазон значений:

[-2147483648; 2147483647]

Если целочисленный литерал положителен и не умещается в этот диапазон, то компилятор подбирает соответствующий размер типов данных в порядке возрастания:

int, unsigned int, long, unsigned long, long long, unsigned long long

Если же и самого большого недостаточно (что сложно себе представить в реальных задачах), то компилятор выдаст ошибку.

Но так только с целыми числами, записанными в десятичном виде. Язык Си позволяет определять в программе числовые литералы еще в шестнадцатеричной и восьмеричной форме. Делается это очень просто, например:

int dec, hex, oct;

dec = 100;

hex = 0x1FA;

oct = 0123;

Здесь 100 – это десятичная форма записи; 0x1FA – шестнадцатеричная (число 1FA); 0123 – восьмеричная. То есть, для записи шестнадцатеричных чисел перед ними ставится префикс в виде символов «0x», а для записи восьмеричных – префикс в виде нуля. Так можно прописывать любые числа в нужном нам формате. Разумеется, на уровне машинных кодов они представляются единым образом в виде набора бит и запись числовых литералов в той или иной форме служит исключительно для удобства восприятия программистом. Не более того.

Но с шестнадцатеричными и восьмеричными литералами есть один важный нюанс. Их компилятор изначально представляет не типом int, как десятичные, а типом **unsigned int**. Соответственно, если литерал не умещается в этот тип, то берутся другие больших размеров в порядке:

unsigned int, unsigned long, unsigned long long

При желании мы можем явно указать компилятору тип числового литерала. Для этого используются следующие суффиксы:

* U или u – использование модификатора unsigned в определении литерала;
* L или l – использование типа long при определении литерала;
* LL или ll – использование типа long long при определении литерала.

Обычно в программах прописывают большие буквы L, так как малые случайно можно спутать с изображением числа 1. Суффикс U можно записывать и отдельно, но можно комбинировать с суффиксами L и LL:

int dec\_i, dec\_ui, dec\_l, dec\_ul;

dec\_i = 100;        *// тип литерала int*

dec\_ui = 100U;      *// тип литерала unsigned int*

dec\_l = 100L;       *// тип литерала long*

dec\_ul = 100UL;     *// тип литерала unsigned long*

Хорошо, с запись литералов разобрались. Но как они сохраняются непосредственно в программе? Конечно же компилятор переводит их в машинный код и хранит непосредственно в скомпилированной программе как неизменяемые данные, то есть, как константы. Когда программа загружается в память компьютера, то вместе с ней загружаются все данные, которые мы явно определяем в тексте.

Отлично, с этим разобрались. Давайте теперь посмотрим на тип char, который формально определен и как символьный и как целочисленный. Первый вопрос, как такое может быть? На самом деле все очень просто. Объявим переменную этого типа, например:

char ch;

а, затем, присвоим ей какой-либо символ, например, буквы d:

ch = 'd';

Обратите внимание, как прописан символ в тексте программы: он заключен в одинарные кавычки (их еще называют апострофами). И только так можно определять символьные литералы в языке Си. Когда компилятор видит одинарные кавычки, то он воспринимает информацию в них, как символ. Никакие другие кавычки для этого использовать нельзя. Например, двойные зарезервированы для определения строк, поэтому запись:

ch = "d";

приведет к ошибке на этапе компиляции. Если же совсем убрать кавычки, то символ d в программе будет восприниматься как переменная:

ch = d;

Поэтому следует запомнить, что для определения символьного литерала используются только одинарные кавычки.

Хорошо, на уровне программы мы теперь знаем, как прописывать отдельные символы. Но спрашивается, как эти символы представляются в машинных кодах. Там же могут быть только числа. Все верно. И это очень важный момент. Все символьные литералы в программе автоматически переводятся в соответствующие коды. Как я вам уже говорил на самом первом занятии, каждому символу ставится в соответствии определенный код (целое число). И приводил фрагмент таблицы ASCII. Так вот, компилятор, используя текущую кодовую таблицу (не обязательно ASCII), переводит все числа в соответствующие коды. В частности, символ буквы ‘d’ в нашем случае имеет код 100. Любой другой символ будет иметь другой код. И, по сути, строчка программы:

ch = 'd';

в нашем конкретном случае, может быть заменена на:

ch = 100;

Хотя, явно прописывать так не следует, потому что при другой кодовой таблице код символа d может быть другим.

Далее, мы можем вывести это значение переменной ch с помощью функции printf() в двух форматах: символьном и числовом:

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

Об этой функции мы еще подробнее будем говорить, пока только отмечу, что вместо символов «%c» будет подставлено значение ch и переведено в символ. А вместо «%d» будет также подставлено также значение из ch, но выведено в виде целого десятичного числа.

После запуска программы увидим:

ch = d, code = 100

Этот факт показывает, что компьютеру важно лишь какое число хранится в переменной ch, а его интерпретация может быть самой разной: или как символ, или как целое число. Причем, обратите внимание, компилятор преобразовывает символьные литералы в числа типа int, а не char, как можно было бы ожидать. Это, как раз связано с тем, что он воспринимает любой символ, как десятичное число, а оно по умолчанию представляется типом int.

Конечно, при работе с символами мы формально можем использовать любой другой целочисленный тип. Например, прописать так:

int ch;

ch = 'd';

В данном случае на работу программы это никак не повлияет. Но, все же, если планируется в переменных хранить коды символов, то лучше использовать тип char в целях экономии памяти, т.к. этого типа достаточно для представления всех основных символов, включая и русские буквы.

## Вещественные литералы

Помимо целочисленных в программе можно прописывать и вещественные литералы. Определять их можно следующими способами:

double d1, d2, d3, d4;

d1 = 10.0;

d2 = -7.;

d3 = 1e2;

d4 = 5e-3;

В первых двух случаях используется символ точки. Причем, обратите внимание, несмотря на то, что числа 10.0 и -7.0 с математической точки зрения соответствуют целым числам 10 и -7, на уровне программы они будут вещественными и иметь тип double.

Все вещественные литералы компилятор языка Си по умолчанию представляет этим типом данных. Соответственно, математические операции с числами 10.0 и -7.0 будут выполняться несколько иначе, чем с аналогичными целыми числами. Это следует иметь в виду.

Последние два варианта – запись числа в экспоненциальной форме:

<число>e<степень десятки>

Например, запись:

https://proproprogs.ru/htm/c_base/files/c_operaciya-prisvaivaniya-chislovye-i-simvolnye-literaly-operaciya-sizeof.files/image001.png

Такую форму удобно использовать в научных расчетах, когда используются или очень маленькие или очень большие числа. В любом случае экспоненциальная форма переводится в вещественное число типа double, даже если оно математически является целым, а не дробным.

При желании мы можем явно указать компилятору переводить вещественный литерал в тип float. Для этого после числа следует прописать суффикс f, например, так:

d1 = 10.0f;

В такой записи вещественное число 10.0 будет представляться типом float, а не double. Это бывает полезно, когда используется переменная типа float и ей правильно было бы присвоить значение того же типа:

float var\_f;

var\_f = 10.0f;

Тогда компилятор не выдаст предупреждение (warning) о возможной потере данных в момент присваивания значения переменной var\_f.

## Операция sizeof

В заключение этого занятия отмечу довольно распространенную операцию sizeof языка Си. Она возвращает число байт, занимаемых в памяти переменной или, отведенных под тип данных. Синтаксис этой операции следующий:

sizeof(<тип | имя переменной>);

sizeof <имя переменной>;

Обратите внимание, во втором случае мы можем записать ключевое слово sizeof без круглых скобок, но тогда эта операция применяется только к переменным, но не к типам. Чтобы не запоминать эти тонкости, обычно sizeof записывают с круглыми скобками и указывают либо тип данных, либо имя переменных. Например:

int size\_float = sizeof(float);

int size\_var\_f = sizeof(var\_f);

На выходе получаем число байт, которое занимает тип float и переменная var\_f. Соответственно:

int size\_ch = sizeof(char);

всегда равно единице.

## Заключение

Итак, на этом занятии вы должны хорошо понимать, как работает операция присваивания и инициализация переменных в языке Си. Знать, как записываются и представляются на уровне машинных кодов числовые и символьные литералы, а также какие суффиксы у них можно пописывать. И, конечно же, уметь применять операцию sizeof для определения размера типа данных или переменной.

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.