

C++

Термины и определения

argument - значение, передаваемое функции.

Блок - последовательность операторов, заключённых в фигурные скобки.

Буфер - область памяти, используемая для хранения данных. Обычно используется устройствами ввода/вывода. Сброс буфера - принудительная запись данных на диск. По умолчанию буфер объекта `cin` сбрасывается при обращении к объекту `cout`. Буфер объекта `cout` сбрасывается по завершению программы.

Встроенный тип данных - тип данных, определённый в языке C++. Например `int`.

Выражение - наименьшая единица вычислений. Состоит из одного или нескольких операндов и оператора (`i + j`) - арифметическое выражение. 0

Возврат значений - функции в C++ должны возвращать значения, если иное не указано явно. Функция `main()` - всегда и обязательно возвращает целое число. Это бывает полезно, т.к. большинство ОС предусматривают возможность другим приложениям обращаться к возвращаемому значению программы.

Директива `#include` - делает код в указанном заголовке доступным в программе (`#include "Sales.h"`).

Заголовок - механизм, позволяющий сделать определения классов или других имён доступными в нескольких программах. Заголовок включается в программу при помощи директивы `#include`.

Заголовок `iostream` - библиотечный тип для потокового ввода/вывода.

Имя функции - имя, под которым функция известна и может быть вызвана.

Инициализация (объявление) - присвоение значения объекту или переменной в момент их создания.

Константа - объект, который не может изменять своё значение в процессе исполнения алгоритма.

Класс - средство определения собственной структуры данных, а так же связанных с ними действий.

Комментарий - игнорируемый компилятором текст в исходном коде.

Манипулятор - объект манипулирующий вводом/выводом (`endl`);).

Массив - коллекция элементов. Все элементы массива - одного типа.

Метод - синоним функции.

Объект cerr - управляет потоком вывода. Используется для вывода сообщений об ошибках. Не буферизируется.

Объект cin - чтение данных с устройства ввода.

Объект clog - используется для записи информации о ходе выполнения программы в файл журнала. Буферизируется.

Объект cout - запись на стандартное устройство вывода. Используется для вывода данных программы.

() - вызов функции. Может иметь аргументы, записываемые внутри скобок.

!= - не равно. Проверяет неравенство левого и правого операнда.

Оператор - часть программы, определяющее действие. Выражение, завершающееся точкой с запятой (;) является оператором. Такие операторы как for, while, if - имеют блоки, которые могут содержать другие операторы.

. (точка) - оператор, получающий два операнда. Левый - объект, правый имя метода класса этого объекта. Обеспечивает доступ к методу класса именованного объекта.

:: - оператор области видимости. Кроме прочего используется для доступа к элементам по именам в пространстве имён. Например: std::cout указывает, что используемое имя cout определено в пространстве имён std.

Оператор += - составной оператор присвоения. Эквивалентен $a = a + b$.

Оператор << - оператор вывода.

Оператор = - оператор присвоения. Присваивает значение правого оператора левому.

Оператор >> оператор ввода.

Оператор for - оператор цикла, обеспечивающий итерационное выполнение. Часть используется для повторения вычислений определённое количество раз.

Оператор if - управляющий условный оператор, обеспечивающий выполнения определённого условия. Если условие истинно (true), то выполняется тело оператора if. В противном случае (false) управление переходит к оператору else.

Оператор while - оператор цикла, обеспечивающий итерационное выполнение кода тела цикла, пока условие является истинным (true).

Переменная (variable) - именованный объект. Объект, который может изменить своё значение в процессе исполнения алгоритма

Присвоение (assignment) - удаляет текущее значение объекта, заменяя его новым. Тип Переменной Имя переменной = значение.

Пространство имён (namespace) - механизм применения имён, определённых в библиотеках. Позволяет избежать случайных конфликтов в имени. Имена из стандартной библиотеки C++ находятся в пространстве имён std.

Пространство имён std - пространство имён, используемое стандартной библиотекой. Запись std::cout указывает, что cout определён в пространстве имён std.

Строковый литерал - последовательность символов, заключённых в кавычки.

Структура данных - логическое объединение типов данных и возможных для них операций.

Тело функции - блок операторов, определяющий выполняемые функцией действия.

Тип istream - библиотечный тип, обеспечивающий потоковый ввод.

Тип ostream - библиотечный тип, обеспечивающий потоковый вывод.

Типы данных - это основа любой программы: они указывают, что именно означают эти данные и какие операции с ними можно выполнять. Тип определяет назначение данных и операции, которые с ними можно выполнять.

Функция - именованный блок операторов.

Функция main() - вызывается операционной системой при запуске программы на C++. У каждой программы C++ должна быть только одна обязательная функция main().

Типы данных - основа любой программы на C++: они указывают, что именно означают эти данные и какие операции с ними можно выполнять.

void - специальный тип данных, который ничего не возвращает.

В язык C++ допустимые для объекта операции определяет его тип.

string - библиотечный тип, представляющий последовательность символов переменной длины. Подобно классу istream он определён в пространстве имён std.

Объект - область памяти, способная содержать данные и обладающая типом. Область памяти, для которой указан тип.

Value - значение.

Инициализация - это не присвоение. Инициализация переменной происходит

при её создании. Присвоение же удаляет предыдущее значение, заменяя его новым.

Объявление (declaration) - делает имя известным программе. Объявление определяет тип и имя.

Определение (definition) - создаёт соответствующую сущность. Определение переменной - это её объявление. Кроме задания имени и типа, определение резервирует место для её хранения и может снабдить исходным значением (value).

Идентификатор - имена. Могут состоять из символов, цифр и символов подчёркивания.

Препроцессор - программа, которая выполняется перед компилятором.

Парсер - программа, которая анализирует программное выражение для построения схемы их вычисления

Язык C++ предоставляет набор встроенных типов данных, операторы для манипулирования ими и набор операторов для управлением процессом выполнения программы. Из этих элементов формируется алфавит языка. Это базовый уровень языка C++ который довольно прост.

Важнейшим компонентом языка C++ является класс. Он позволяет определять собственные типы данных (типы класса), которые отличаются от встроенных в язык типов данных. Это есть одна из главных задач проекта C++.

Язык C++ позволяет определять типы классов, в состав которых можно включать операции, выполняемые с этими данными.

В C++ действует правило - всюду, где только можно ставьте const. Это повышает надёжность программы.

Если при написании программы ты затрудняешься с реализацией конкретной функции, напиши вместо неё комментарий, в котором простыми словами опиши ей работу.

В знаки < > заключаются стандартные библиотеки языка. А в " " собственные файлы, такие как motor.h

По учебнику - Составные типы стр. 111 (113). Присвоение и указатели.

Темы с указателями и ссылками мне остались не понятны!

Введение

Развиваем свои навыки <https://proglib.io/p/prog-skill/>

Обязательная функция main()

```
int main() // Обязательная функция (Параметры)
{ // Блок операторов
    return 0; // Оператор завершения функции
}
```

Функцию main() вызывает операционная система. В программе C++ должна быть только одна функция main().

Блок - это последовательность из любого количества операторов, заключённых в фигурные скобки.

Из командной строки:

echo \$? - проверка состояния выполненной программы

Некоторые параметры сборки:

g++ -o [имя создаваемого файла программы] [имя исходного файла]

например:

```
$: g++ -o cals main.cpp
```

g++ -Wall -o - вывод предупреждения компилятора в случае обнаружения им проблемных конструкций

Стандартная библиотека ввода-вывода iostream.

Input/Output - IO

Поток (stream) - последовательность символов, записываемая или считываемая с устройств ввода-вывода. Подразумевается, что символы поступают и передаются последовательно на протяжении определённого времени.

Простая программа. Предлагает пользователю ввести два любых числа, после чего выдают их сумму.

```
#include <iostream>

int main()
{
    std::cout << "Введите два числа: " << std::endl;
    int v1 = 0, v2 = 0;
    std::cin >> v1 >> v2;
    std::cout << "Сумма " << v1 << " и " << v2 << " равно " << v1 + v2 <<
std::endl;
    return 0;
}
```

cin (си-ин) - ввод

cout (си-аут) - вывод

std - пространство имён. Все имена, определённые в стандартной библиотеке `iostream` находятся в пространстве имён `std`. Позволяет избежать вероятных конфликтов совпадения имён в разных библиотеках.

`::` - оператор области видимости

`<<` - оператор ввода

endl - манипулятор. При его записи в поток происходит переход на новую строку программы и сброс буфера, связанного с данным устройством. Сброс буфера гарантирует, что весь вывод будет немедленно записан в поток, на не будет ожидать записи находясь в памяти.

"Введите два числа: " - строковый литерал.

В выражении

```
std::cout << "Введите два числа: " << std::endl;
```

есть левый и правый операнды. Левый это объект класса `ostream` (вывода), а правый операнд - подлежащее выводу значение. Оператор `<<` заносит переданное значение в объект `cout` класса `ostream`.

Запись выражения

```
std::cout << "Введите два числа: " << std::endl;
```

эквивалентно:

```
(std::cout << "Введите два числа: ") << std::endl;
```

и

```
std::cout << "Введите два числа: ";
std::cout << std::endl;
```

Запись выражения

```
std::cin >> v1 >> v2;
```

эквивалентно:

```
std::cin >> v1;  
std::cin >> v2;
```

Так же эту программу можно записать так (объявив пространство имён один раз в начале программы)

```
#include <iostream> // Стандартная библиотека  
using namespace std; // Пространство имён  
  
int main() // Обязательная функция  
{ // Блок операторов  
  // В операнд cout записывается оператором потока << строковый литерал "".  
  // Манипулятор endl - переход на новую строку с очисткой буфера  
  cout << "Введите два числа: " << endl;  
  int v1 = 0, v2 = 0; // Объявление переменных  
  // Операнд вывода cin при помощи оператора потока >> записывает в переменные v1 и  
  // v2 вводимые значения. Всё, кроме цифр игнорируется  
  cin >> v1 >> v2;  
  // Операнд cout выводит слово и значения переменных, складывает и выводит результат  
  cout << "Сумма " << v1 << " и " << v2 << " равно " << v1 + v2 << endl;  
  return 0; // Возвращаемое значение  
}
```

Итерация и While

Итерация - повторение, цикл

Оператор while осуществляет итерационное выполнение фрагмента кода, пока условие истинно.

Условие - это выражение, результатом которого является истина или ложь.

`+=` - составной оператор присвоения с суммой. Добавляет правый операнд к левому операнду.

Пример:

```
sum += val;
```

эквивалентно:

```
sum = sum + val;
```

`++` префиксный оператор инкремента, который осуществляет приращение

```
++val; // Добавить 1 к val
```

Оператор приращения добавляет 1 к своему операнду (val)

эквивалентно:

```
val = val + 1;
```

Пример кода:

```
#include <iostream>
using namespace std;

int main()
{
    int sum = 0, val = 1;
    // Продолжать выполнение цикла, пока val не превысит 10
    while (val <= 10)
    {
        sum += val; // Присвоить sum сумму val и sum
        ++val; // Добавить 1 к val
        cout << sum << endl; // вывести значение sum для каждой итерации
    }
    cout << "Сумма от 1 до 10 включает " << sum << endl; // Вывести результат
    // последней итерации
    return 0;
}
```

`--` префиксный оператор декремента

Пример, выводящий на экран числа от 10 до 0:


```

#include <iostream>
using namespace std;

int main()
{
    int val = 11;
    // Продолжать выполнение цикла, пока
    while (val >= 1)
    {
        --val;
        cout << val << endl;
    }
    return 0;
}

```

Мой пример (может содержать несуразности). Запрашивает два числа, выводит разность:

```

#include <iostream>
using namespace std;

int main()
{
    cout << "Введите два числа" << endl; // Приглашение
    int v1 = 0, v2 = 0;
    cin >> v1 >> v2; // Записываем в переменные вводимые значения
    cout << "Вы ввели: " << v1 << " и " << v2 << endl;
    int sum = v1 - v2;
    // Продолжать выполнение цикла, пока
    while (0 < sum) // Пока 0 меньше sum
    {
        --sum; // Декремент. Вычитается по 1
        cout << "--sum " << sum << endl;
    }
    while (0 > sum)
    {
        ++sum;
        cout << "++sum " << sum << endl;
    }
    return 0;
}

```

Ещё пример с использованием оператора цикла while:

```

#include <iostream>
using namespace std;

int main()
{
    int num = 1;
    int number;
    int total = 0;
    cout << "Введите число: " << endl;
    cin >> number; // Записываем введённое число в переменную
    while (num <= 6) // Пока 1 меньше или равно 6
    {
        total += number; // 0 + введённое пользователем число
        ++num; // Приращение на 1 (num + 1) количество раз, указанное в
скобках while
    }
    cout << total << endl; // Вывод значения total
    return 0;
}

```

Считать введённые пользователем данные и вычислить их

```

#include <iostream>
using namespace std;

int main()
{
    int sum = 0, value = 0;
    // Читать данные до конца файла и вычислить сумму всех значений
    while (cin >> value)
        sum += value;
    cout << sum << endl;
    return 0;
}

```

Оператор *for*

Оператор `for` обеспечивает условное циклическое выполнение действий.

```
for (П = НЗ; П != КЗ; П += Ш)
{
    операция;
}
```

где `П` — параметр, `НЗ` — начальное значение, `КЗ` — конечное значение (в общем случае — условие продолжения цикла, `Ш` — шаг):

Пример:

```
#include <iostream>
using namespace std;

int main()
{
    int sum = 0;
    for (int val = 1; val <= 10; ++val) // Сложить числа от 1 до 10 включительно
    {
        sum += val; // Иначе sum = sum + val
        cout << "Количество итераций: " << val << endl;
    }
    cout << "Результат: " << sum << endl;
    return 0;
}
```

Тело цикла `for` выполняется, пока условие истинно (в данном примере 10).

Оператор if

Оператор if обеспечивает условное или циклическое выполнение действий.

Пример программы, подсчитывающей введенные пользователем одинаковые числа.

```
#include <iostream>
using namespace std;

// Задача: подсчитывать количество числовых совпадений

int main()
{
    // Первая переменная для подсчитываемого числа, вторая для каждого числа
    int curVal, val;
    cout << "Enter your number" << endl;
    cout << "to exit the program, press any letter" << endl;
    if (cin >> curVal) // Если число введено
    {
        int checker = 1; // Создаём счётчик
        while (cin >> val) // Ожидаем и записываем второе число
        {
            if (curVal == val) // Сравниваем предыдущее и последнее
            {
                ++checker; // Если одинаковые, то инкрементируем
                // Счёт для предыдущего числа.
                // Если разное, то выводим на экран сообщение, сбрасываем
                // счётчик и обновляем 1-ю переменную
                // для вывода на экран предыдущего числа.
            }
            else
            {
                cout << "NumberB: " << curVal << " concurrencesB: "
                << checker << endl;
                checker = 1;
                curVal = val;
            }
        }
        // Относится к первому if (if (cin >> curVal))
        // Вывод счёта для самого последнего числа. Срабатывает, если
        // вместо числа введена буква или символ
        cout << "NumberA: " << curVal << " concurrencesA: " << checker <<
        endl;
    }
    return 0;
}
```

Описание программы.

Задача: запрашивать у пользователя числа и подсчитывать количество совпадений.

Сначала работает оператор, проверяющий введенное число (cin >> currVal) и сохраняет его в первую переменную (currVal), если число введено, то создаётся

локальная

переменная checker со значением 1. Это необходимо для начала отчёта.

После этого начинает работать оператор while, который сохраняет введённое пользователем второе число

во вторую переменную (val). Далее начинает работать следующий оператор if, который проверяет совпадение

значений двух ранее созданных переменных. Если есть совпадение

(пользователь ввёл два одинаковых числа,

то к локальной переменной checker инкрементно прибавляется единица. Если

нет (else) совпадения чисел, то

программа выводит сообщение из строкового литерала и переменной checker,

в которой храниться количество

совпадений. Новое число записывается в переменную val и сбрасывается счётчик.

Цикл while заканчивается и на экран выводится результат.

1. Запрашиваем число

2. Проверяем совпадения - Если нет, то выводим количество совпадений, запоминаем последнее число и сбрасываем счётчик

|
Если есть,
увеличиваем
счётчик на 1
и запрашиваем снова

Класс. Структура данных

Структура данных. Класс.

Класс - фундаментальный элемент языка C++.

Класс применяется для определения собственных структур данных. Он определяет тип данных и набор операций, связанный с этим типом. Механизм классов в C++ очень важен. Фактически, при проектировании программы на C++ основное внимание уделяют именно определению различных типов классов, которые ведут себя так же как встроенные типы данных.

Для использования класса необходимо знать:

1. Каково его имя?
2. Где он определён?
3. Что он делает?

Пример: имя класса `Sales_item` а определён он в заголовке `"Sales_item.h"` (обозначен двойными кавычками, а не угловыми скобками!).

Подобно встроенным типам данных (`int`, `bool`, `long` etc) можно создать объект (переменную) типа класса.

Пример:

```
Sales_item item;
```

где `Sales_item` - тип, а `item` - объект. // Создан объект типа `Sales_item` или объект класса `Sales_item` или экземпляр класса `Sales_item`.

С подобными объектами можно выполнять разные операции, например:

1. Вызывать функции (методы, содержащиеся в классе), например `isbn()`
2. Использовать различные операторы, например `<<`, `=`, `+`, `for` или `>>`

Действия, которые могут быть осуществлены с объектами класса определяет автор этого класса.

Объект `cerr` (си-err) - стандартная ошибка, которая используется для создания предупреждений и сообщений об ошибках.

Объект `clog` (си-лог) - для создания информационных сообщений.

Простой пример класса:

```

#include <iostream>
using namespace std;

class BankAccount
{
    public:
        void sayHi() // Метод sayHi
        {
            cout << "Hi" << endl;
        }
};

int main()
{
    BankAccount test; // Объект test класса BankAccount
    test.sayHi(); // Вызов метода объекта. Точка (.) используется для получения
доступа
}

```

Пример использования класса:

```

#include <iostream>
#include "Sales_item.h"
using namespace std;

int main()
{
    Sales_item item1, item2; // Объявляем два объекта класса
    cin >> item1 >> item2; // Считываем ввод двух номеров ISBN
    // item1.isbn() и item2.isbn() - это методы из класса Sales_item
    if (item1.isbn() == item2.isbn()) // Сначала проверяем, что два ISBN
совпадают
    {
        // Если совпадают, то:
        cout << item1 + item2 << endl; // Складываем количество проданных,
выводим общую сумму
        // И среднюю сумму

        return 0; // Возвращаем успех
    }
    else // Иначе выводим сообщение об ошибке
    {
        cerr << "Данные должны относиться к тому же ISBN" << endl;
        return -1; // Возвращаем ошибку
    }
}

```

Ещё пример использования класса:

```

#include <iostream>
#include "Sales_item.h"

using namespace std;

int main()
{
    Sales_item total; // Объект класса для предыдущего значения
    if (cin >> total) // Введено значение
    {
        Sales_item trans; // Последнее значение
        while (cin >> trans) // Введено последнее значение
        {
            // Сравниваем два метода из класса
            if (total.isbn() == trans.isbn())
            {
                total += trans; // total = total +
trans
            }
            else
            {
                cout << total << endl;
                total = trans; // Присваиваем последнее значение
            }
        }
        cout << total << endl; // Первое значение
    }
    else
    {
        cerr << "No data?!" << endl; // Данных нет?
        return -1;
    }
    return 0;
}

```

Описание программы:

Данный код, как обычно начинается с подключаемых заголовков: `iostream` (библиотека) и `Sales_item` (собственный - из класса).

В функции `main()` объявлен объект `total` (для суммирования данных по текущему ISBN). Чтение начинается с первой транзакции в переменную `total` с проверкой успешности `if (cin >> total)`. Если условие терпит неудачу, то управление переходит к удалённому оператору `else`, который выводит сообщение об отсутствии данных (`cerr << "No data?!"`).

Если условие успешно, то управление переходит к блоку наиболее удалённого оператора `if`, который начинается с объявления объекта `trans` (`Sales_item trans;`) предназначенного для хранения считываемых транзакций. Оператор `while` читает все остальные транзакции (`while (cin >> trans)`). Тело цикла `while` выполняется, пока условие истинно. В теле цикла `while` один оператор `if`, который проверяет тождество ISBN. Если они равны, то выполняется составной оператор суммирования объектов (`total += trans;`). Если не равны - отображается значение объекта `total`, которому затем присваивается значение объекта `trans` (`total = trans;`). После выполнения тела второго оператора `if` управление возвращается к условию цикла `while`,

читающему

следующую транзакцию (`while (cin >> trans)`) до тех пор, пока записи не исчерпаются.

После выхода из цикла `while` объект `total` содержит значение последнего ISBN. В последнем операторе `if` отображается значение последнего ISBN (`cout << total << endl`).

Методы - это функции, определённые в составе класса.

`isbn()` - где `()` -оператор вызова без аргументов (`arguments`)

Типы данных

Арифметические типы данных

Простые встроенные типы.

Арифметические типы.

Вопросы и ответы.

1. Каковы различия между типами int, long, long long и short?

Ответ: int - целое число 16 bit.

long - длинное целое число 32 bit.

long long - длинное целое число 64 bit.

short - короткое целое число 16 bit.

Представленные типы являются знаковыми типами (-/+ , положительные и отрицательные числа).

2. Какие типы вы бы использовали для коэффициента основной суммы и платежей?

Ответ: double или float.

1. Символы
2. Целые числа
3. Логические значения
4. Числа с плавающей запятой

Есть две разновидности арифметических типов: целочисленные типы (включая символьные и логические) и типы с плавающей запятой.

Знаковый тип (signed) способен представлять отрицательные и положительные числа.

Беззнаковый (unsigned) - только положительные числа и 0.

Типы int, short, long и long long являются знаковыми. Соответствующий беззнаковый тип получают добавлением части unsigned к названию типа. Например: unsigned int, unsigned short и т.д.

Тип unsigned int может быть сокращён до unsigned.

unsigned int может быть сокращён до unsigned

Несколько эмпирических правил, способных помочь при выборе используемого типа.

1. Используйте беззнаковый тип, когда точно знаете, что значения не могут быть отрицательными;
2. Используйте тип int для целочисленной арифметики. Если ваши значения больше, чем минимально гарантируемый тип int, то используйте long long;
3. Не используйте базовый тип char и bool в арифметических выражениях. Используйте их только для хранения символов и логических значений;
4. Используйте тип double для вычислений с плавающей точкой. У типа float

обычно не хватает точности. Точность, определяемая типом `long double` обычно чрезмерна и не нужна.

Преобразование типов

Тип объекта определяет данные, которые он может содержать. и операции, которые с ним можно выполнять. Среди операций, поддерживаемых множеством типов, есть возможность преобразовать (convert) объект данного типа в другой.

Преобразование типов происходит автоматически, когда объект одного типа используется там, где ожидается объект другого типа.

Пример того, когда значения одного арифметического типа присваивается другому:

```
bool b = 42; // b содержит true  
bool b = 0; // b содержит false
```

Когда значение одного из не логических арифметических типов присваивается объекту типа `bool`, результат будет `false`, если значением является 0, в противном случае - `true`.

```
int i = b; // i содержит значение 1
```

Это плохо понял!

Когда значение типа `bool` присваивается одному из других арифметических типов, будет получено значение 1, если логическим значением было `true`, и 0, если это было `false`.

```
i = 3.14; // i содержит значение 3
```

Когда значение с плавающей точкой присваивается объекту целочисленного типа, оно усекается до части перед десятичной точкой.

```
double pi = i; // pi содержит значение 3.0
```

Плохо понял!

Когда целочисленное (интегральное) значение присваивается объекту типа с плавающей точкой, дробная часть равна нулю. Если у целого числа больше битов, чем может вместить объект с плавающей точкой, то точность может быть потеряна.

```
unsigned char c = -1; // при 8-битовом char c содержит 255
```

Это надо проверить программно, чтобы развеять сомнения!

Если к объекту беззнакового типа присваивается значение не из его диапазона, результатом будет остаток деления по модулю значения, которые способен содержать тип назначения. Например, 8-битовый тип `unsigned char` способен содержать значения от 0 до 255 включительно. Если присвоить ему значение вне этого диапазона, то компилятор присвоит ему остаток от деления по модулю 256. Поэтому в результате присвоения значения -1 переменной 8-битового типа `unsigned char` будет получено значение 255.

```
signed char c2 = 256; // при 8-битовом значение char c2 неопределенно
```

Если к объекту знакового типа присваивается значение не из его диапазона, результат оказывается не определён.

Пример ошибочного но компилируемого кода с преобразованием типов

```
#include <iostream>
using namespace std;

int main()
{
    unsigned u = 10;
    int i = -42;
    cout << "i + i: " << i + i << endl;
    cout << "u + i: " << u + i << endl;
}
```

Литералы

Литерал - это элемент программы, который представляет значение.

Типы литералов:

1. Целые (int, 42)\$
2. С плавающей запятой, логические (float, 42.3)\$
3. Указатели (тип данных, который содержит адрес переменной);
4. Строковые (string?)\$
5. Символьные (char?);

Целочисленные литералы и литералы с плавающей запятой

Целочисленные литералы могут быть представлены в восьмеричной, десятичной и шестнадцатеричной форме:

```
024 // Восьмеричная форма
20 // Десятеричная форма
0x14 // Шестнадцатеричная форма
```

Тип целочисленного литерала зависит от его значения и формы. По умолчанию десятичные литералы считаются знаковыми (знак - не является частью литерала - он оператор), а восьмеричные и шестнадцатеричные могут быть знаковыми и беззнаковыми.

Для десятичного литерала принимается наименьший тип int, long или long long.

Для восьмеричного и шестнадцатеричного литерала принимается int, unsigned int, long, unsigned long, long long, unsigned long long.

Нет литералов типа short.

Литералы с плавающей запятой имеют либо десятичную точку 3,14, либо экспоненту 3,14E0

```
char x = 'a'; // Символьный литерал
char x = "Hello World"; // Строковый литерал
```

Типом строкового литерала является массив константных символов. К каждому строковому литералу компилятор добавляет нулевой символ '\0'. Два строковых литерала, разделённых пробелами, табуляцией, двойными кавычками и символами новой строки конкретизируются в единый литерал.

```
cout << "Hi Alex!"
      "Omg Ubuntu!" << endl;
```

Управляющие последовательности.

Начинаются с символа наклонной черты влево (/).

Новая строка \n

Вертикальная табуляция \v
Наклонная черта влево \\
Возврат каретки \r
Горизонтальная табуляция \t
Возврат на один символ \b
Вопросительный знак \?
Прогон страницы (formfeed) \f
Оповещение, звонок \a
Двойная кавычка \"
Одинарная кавычка \'

Пример:

```
#include<iostream>
using namespace std;

void printHi()
{
    cout << "Hi Alex!"
          "Omg Ubuntu!" << endl;
    cout << '\n' << endl; // Отобразить новую строку
    cout << "\tHi!\n" << endl; // Отобразить "Hi!" и новую строку
}

int main()
{
    printHi();
    return 0;
}
```

Логический литерал

```
bool test = false;
```


Переменные

Переменная - именованная область памяти, которой могут манипулировать программы.

У каждой переменной в C++ есть определённый тип. Тип определяет размер и расположение в памяти, диапазон возможных значений, набор применимых операций.

В C++ переменная и объект - синонимы.

string - библиотечный тип, представляющий последовательность символов переменной длины. Подобно классу `iostream` он определён в пространстве имён `std`.

Объект - область памяти, способная содержать данные и обладающая типом.

Инициализация - это не присвоение. Инициализация переменной происходит при её создании. Присвоение же удаляет предыдущее значение, заменяя его новым.

Списочная инициализация

```
int arm = 0;  
int arm = {0};  
int arm {0};  
int arm (0);
```

При использовании этой формы инициализации компилятор не позволит инициализировать переменные встроенного типа, если это может привести к потере информации.

Пример:

```
long double ld = 3.1415926536;  
int a{ld}, b = {ld}; // Ошибка: преобразование с потерей данных  
int c{ld}, d = ld;   // Ок: но значение будет усечено
```

Объявление (declaration) - делает имя известным программе. Объявление определяет тип и имя.

Определение (definition) - создаёт соответствующую сущность. Определение переменной - это её объявление. Кроме задания имени и типа, определение резервирует место для её хранения и может снабдить исходным значением (value).

Пример:

```
extern int i; // Объявляем, но не определяем переменную. Ключевое слово extern  
int i; // Объявляем и определяем переменную
```

Область видимости (scope) - это часть программы. Разграничиваются фигурными скобками.

Объект (переменную) имеет смысл определять ближе к месту его первого использования.

Инициализация — присваивание переменной определённое значение в момент его создания.

Если переменная определена без инициализатора, то происходит её инициализация по-умолчанию. Таким переменным присваивается значение по-умолчанию.

В C++ инициализация и присвоение (=) разные операции. Инициализация переменной происходит при её создании. Присвоение удаляет текущее значение переменной и заменяет его новым.

```
int sun = 0, value, // sum, value, и units_sold имеют тип int
units_sold = 0; // sum и units_sold инициализированы значением 0
```

Объявление переменной определяет ей тип и имя. **Определение** переменной - это её объявление. кроме задания имени и типа, определение резервирует в памяти место для её хранения и может снабдить переменную исходным значением.

Чтобы получить объявление, не являющееся также определением, добавляем ключевое слово `extern` и можно не предоставлять явный инициализатор.

```
extern int i; // Объявить, но не определить переменную i
int j; // Объявить и определить переменную j
```

Идентификаторы - имена переменных, могут состоять из символов, цифр, и символов подчёркивания. Ограничений на длину имён не накладывается. Символы в верхнем и нижнем регистре различаются.

Ключевое слово - это зарезервированные идентификаторы, которые имеют специальное значение для компилятора. Их можно использовать только в том смысле, в котором они определены. Ключевые слова делятся на:

спецификаторы типов: `char`, `double`, `enum`, `float`, `int`, `long`, `short`, `struct`, `signed`, `union`, `unsigned`, `void`, `typedef`

квалификаторы типов: `const`, `volatile`

квалификаторы классов памяти: `auto`, `extern`, `register`, `static`

операторы языка и идентификаторы специального назначения: `break`, `continue`, `do`, `for`, `goto`, `if`, `return`, `switch`, `while`; `default`, `case`, `sizeof`

модификаторы и псевдопеременные: конкретный набор зависит от компилятора

Соглашение об именах (идентификаторах) переменных: 1) идентификатор должен быть осмысленным, 2) имена обычно состоят из строчных символов,

например index, а не Index или INDEX, 3) имена классов обычно начинаются с прописной буквы, например Sales_item, 4) несколько слов в идентификаторе разделяют либо символом подчёркивания, либо прописными буквами в первых символах каждого слова, например student_loan или studentLoan, но не studentloan!

Область видимости (scope) - это часть программы, в которой у имени есть конкретное значение. Как правило, область видимости в C++ разграничивается фигурными скобками.

Имена видимы с момента их объявления и до конца области видимости, в которой они объявлены.

Пример с глобальной и локальной одноимёнными переменными

```
#include <iostream>
using namespace std;

int value = 42; // Глобальная видимость переменной

int main()
{
    /* Программа предназначена в демонстративных целях
    * использование одноимённой глобальной и локальной переменных - плохой
    стиль!
    */
    int data = 0; // Локальная видимость (в пределах блока фигурных скобок)
    cout << value << " " << data << endl; // 42 0
    int value = 0; // Новая локальная одноимённая переменная
    cout << value << " " << data << endl; // 0 0
    // Явное обращение к глобальной переменной.
    cout << ::value << " " << data << endl; // 42 0

    return 0;
}
```

Пример с использованием разных областей видимости переменных

```

#include <iostream>
using namespace std;

int main()
{
    int i = 100, sum = 0; // i и sum в пределах видимости блока main() { }
    for (int i= 0; i != 10; ++i) // Получаем от 0 до 9
        sum += i; // аналогично sum = sum + i. i в пределах видимости for ()
                    // sum принимает значение от i в конструкции for

    /* 0 = 0 + 0
       * 1 = 0 + 1
       * 3 = 1 + 2
       * 6 = 3 + 3
       * 10 = 6 + 4
       * 15 = 10 + 5
       * 21 = 15 + 6
       * 28 = 21 + 7
       * 36 = 28 + 8
       * 45 = 36 + 9
       */
    cout << i << " " << sum << endl; // Получаем 100 и 45

    return 0;
}

```

Составные типы

Составной тип - это тип, определённый в терминах другого типа. У языка C++ несколько составных типов, два из которых ссылки и указатели.

В простейшем случае в момент объявления (объявление переменной определяет ей тип и имя) переменных не было ничего, кроме имён переменных. Такие переменные имеют базовый тип объявления.

Более сложные операторы позволяют определять переменные с составными типами, которые состоят из объявлений базового типа.

Ссылка (reference) - является альтернативным именем объекта (переменной). Ссылочный тип ссылается на другой тип. В определении (определение переменной - это её объявление. кроме задания имени и типа, определение резервирует в памяти место для её хранения и может снабдить переменную исходным значением.) ссылочного типа используется оператор объявления в формате `&d`, где `d` - `j,]zdkztvjt bvz/`

Обычно при инициализации (присваивание переменной определённое значение в момент его создания) переменной значение инициализатора копируется в создаваемый объект. При определении ссылки вместо копирования значения инициализатора происходит связывание (bind) ссылки с её инициализатором. После инициализации ссылка остаётся связанной с исходным объектом. Нет никакого способа изменить привязку ссылки так, чтобы она ссылалась на другой объект, поэтому ссылки следует инициализировать.

```
int ival = 1024;
int &refVal = ival; // refVal ссылается на другое имя, ival
int &refVal2; // ошибка: ссылку следует инициализировать!
```

После того, как ссылка определена, все операции с ней фактически осуществляются с объектом, с которым связана ссылка.

```
refVal = 2; // Присваивает значение 2 объекту (переменной), на который
            // ссылается ссылка refVal, т.е. ival
int ii = refVal // То же, что и ii = ival
```

Ссылка - не объект, а только другое имя уже существующего объекта.

Поскольку ссылки не объекты, нельзя определить ссылку на ссылку.
Поскольку ссылки не объекты, у них нет адресов.

За несколькими исключениями, типы ссылки и объекта, на который она ссылается, должны совпадать точно.

Ссылка может быть связана только с объектом, а не с литералом (это элемент программы, который представляет значение) или результатом более общего выражения.

Указатели

Указатель () - составной тип, переменная которого указывает на объект другого типа. Подобно ссылкам указатели используются для доступа к другим объектам. В отличие от ссылок, указатель - это настоящий объект. Указатели могут быть присвоены и скопированы: один указатель за время своего существования может указывать на несколько разных объектов. В отличие от ссылки, указатель можно не инициализировать (присваивание переменной определённое значение в момент его создания) в момент определения.

```
int *ip1, *ip2; // ip1 и ip2 - указатели на тип int
double dp, *dp2; // dp2 - указатель на тип double
                // dp - переменная типа double
```

Получение адреса объекта.

Указатель содержит адрес другого объекта. Для получения адреса объекта используется оператор обращения к адресу, или оператор &

```
int ival = 42;
int *p = &ival; // p содержит адрес переменной ival
                // p - указатель на переменную ival
```

За несколькими исключениями типы указателя и объекта, на который ссылается указатель, должны совпадать.

Использование указателя для доступа к объекту.

Когда указатель указывает на объект, для доступа к этому объекту можно использовать оператор обращения к значению, или оператор *

```
int ival = 42;
int *p = &ival // p содержит адрес ival; p - указатель на ival
cout << *p;    // * возвращает объект, на который указывает p. Выводит 42
```

При присвоении значения *p оно присваивается объекту, на который указывает указатель p.

```
*p = 0;        // * возвращает объект; присвоение нового значения ival через
указатель p
cout << *p;    // выводит 0
```

Некоторые символы, такие как & и *, используются и как оператор в выражении, и как часть объявления.

Нулевые указатели.

Нулевой указатель (null pointer) не указывает ни на какой объект.

```
int *p1 = nullptr; // эквивалентно int *p1
int *p2 = 0;        // непосредственно инициализирует p2 литеральной
константой 0, необходимо include cstdlib
```

```
int *p3 = NULL; // эквивалентно int *p3 = 0;
```

Проще всего инициализировать указатель, используя литерал nullptr, который был введён новым стандартом. Лучше вообще избегать применение переменного NULL, используя вместо неё литерал nullptr.

Рекомендуется инициализировать (присваивание переменной определённое значение в момент его создания) все переменные, особенно указатели. Рекомендуется определять (определение переменной - это её объявление. Кроме задания имени и типа, определение резервирует место для её хранения и может снабдить исходным значением (value) указатель только после определения объекта, на который он должен указывать. Если связываемого объекта ещё нет, то инициализируйте указатель nullptr или 0.

Принципиальная разница между ссылкой и указателем?

главных отличий два:

- ссылка, в отличие от указателя, не может быть неинициализированной;
- ссылка не может быть изменена после инициализации.

Отсюда и получаем плюсы и минусы использования того и другого:

- ссылки лучше использовать когда нежелательно или не планируется изменение связи ссылка → объект;
- указатель лучше использовать, когда возможны следующие моменты в течении жизни ссылки:
 - ссылка не указывает ни на какой объект;
 - ссылка указывает на разные объекты в течении своего времени жизни.

Присвоение и указатели.

И указатели и ссылки предоставляют косвенный доступ к другим объектам.

Ссылка - не объект!

Между указателем и содержащимся в нём адресом нет жесткой связи.

Спецификатор *const*

Значение переменной можно сделать неизменным, используя в её определении спецификатор `const`

- **Определение** (definition) - создаёт соответствующую сущность. Определение переменной - это её объявление. Кроме задания имени и типа, определение резервирует место для её хранения и может снабдить исходным значением (value).

```
const int i = get_size(); // ок: инициализация во время выполнения программы
const int j = 42;          // ок: инициализация во время компиляции
const int k;               // ошибка: k - не инициализированная константа
```

Чтобы совместно использовать константный объект в нескольких файлах, его необходимо определить с использованием ключевого слова `extern`.

```
// Файл file_1.cc. Определение и инициализация константы, которая доступна
// для других файлов
extern const int bufSize = fcn();
// Файл file_1.h
extern const int bufSize; // та же bufSize, определённая в file_1.cc
```


Overflow

Избегайте переполнения выбирая подходящие типы.

Пример переполнения

```
#include<iostream>
using namespace std;

/* Если происходит переполнение беззнакового (unsigned) short, то значение
 * переменной сбрасывается на 0
 * Если происходит переполнение знакового (signed) short, то значение
 * переменной сбрасывается на -32767
 */

int main() {
    unsigned short uShortValue = 65535;
    cout << "Увеличение unsigned short: " << uShortValue << endl; // Вывод на экран
    cout << ++uShortValue << endl; // Прибавляется единица

    short signedShort = 32767;
    cout << "Увеличение signedShort: " << signedShort << endl;
    cout << ++signedShort << endl;

    return 0;
}
```

Size of data types

Пример кода

```
#include<iostream>
using namespace std;

int main() {
    cout << "bool: " << sizeof(bool) << endl;

    cout << "char: " << sizeof(char) << endl;

    cout << "short: " << sizeof(short) << endl;

    cout << "short int: " << sizeof(short int) << endl;
    cout << "unsigned short int: " << sizeof(unsigned short int) << endl;

    cout << "int: " << sizeof(int) << endl;
    cout << "unsigned int: " << sizeof(unsigned int) << endl;

    cout << "long int: " << sizeof(long int) << endl;;
    cout << "unsigned long int: " << sizeof(unsigned long int) << endl;

    cout << "long long: " << sizeof(long long) << endl;
    cout << "unsigned long long: " << sizeof(unsigned long long) << endl;

    cout << "float: " << sizeof(float) << endl;
    cout << "double: " << sizeof(double) << endl;

    cout << "Вывод зависит от комплятора и ОС" << endl;

    return 0;
}
```

P.S. Вывод зависит от комплятора и ОС

Use of Auto

Ключевое слово auto делегирует компилятору принятие решения о типе переменной

Пример использования auto

```
#include<iostream>
using namespace std;

int main() {
    auto coinFlipperHeads = true;
    auto largeNumber = 250'000'000'000'000;
    cout << "coinFlipperHeads = " << coinFlipperHeads;
    cout << " , sizeof(coinFlipperHeads) = " << sizeof(coinFlipperHeads) << endl;

    cout << "largeNumber = " << largeNumber;
    cout << " , sizeof(largeNuber) = " << sizeof(largeNumber) << endl;

    cout << "Вывод зависит от комплятора и ОС" << endl;

    return 0;
}
```

Постинкремент и декремент

Префиксный инкремент ++

```
++/*имя переменной*/;  
++i;  
++val;
```

При использовании операции преинкремента значение переменной сначала увеличивается на 1, а затем используется в выражении.

При использовании операции преинкремента (++i) значение переменной сначала увеличивается на 1, а затем используется в выражении (сначала делаем, потом думаем).

При использовании операции постинкремента значение переменной сначала используется в выражении, а потом увеличивается на 1.

Switch ... case

Ресурсы

https://www.tutorialspoint.com/cplusplus/cpp_switch_statement.htm

<http://easy-code.ru/lesson/switch-case-cpp>

Логические выражения в C++

<https://codelessons.ru/cplusplus/lessons/logical-operators-in-cpp.html>

Логические выражения логические операторы в C++

Логические переменные

Для того, чтобы мы могли хранить данные логического типа, нам надо знать о логических переменных. Мы уже встречались и ими в нашем прошлом уроке. Но давайте повторим:

Логические данные хранятся в переменных типа `bool`.

Хранить они могут только два значения:

- «Верно» — это `true`;
- «Лож» — это `false`;

Теперь давайте узнаем какие логические операторы существуют в C++.

Операторы сравнения

Язык C++ имеет 5 различных операторов сравнения в своем арсенале. Также существуют такие операторы, которые являются комбинациями других. Все они вам должны быть знакомы из курса математики, поэтому их изучение не должно вызвать у вас проблем.

Давайте разберем по порядку каждый из них:

- `A < B` — сравнивает две переменные и возвращает `true`, если `A` меньше `B`.
- `A > B` — возвращает `true`, если `A` строго больше `B`.
- `A == B` — проверяет на равенство переменные `A` и `B`.
- `A != B` — проверяет переменные `A` и `B` на неравенство.
- `A >= B` — нестрогое неравенство. Возвращает `true`, если `A` больше или равно `B`.
- `A <= B` — противно неравенству `A > B`.

Теперь давайте разберем пару примеров, тем самым подкрепим теорию практикой:

```
bool r; int a = 5, b = 7; // создали переменные с которыми будем работать
r = a > b; // r содержит false, поскольку 5 < 7
r = a <= b; // r содержит true
r = a <= 5 // r равен true
r = b == 9 // r содержит false, поскольку 7 != 9
```

Из примера видно, что в качестве `A` и `B` мы можем использовать не только переменные, но и простые числа.

Мы немного поэкспериментировали с операторами сравнения, однако пока не

можем сгруппировать несколько из них и следственно создать серьезное логическое выражение. Для этих целей мы и будем применять логические операторы.

Логические операторы

Для комбинации сразу нескольких логических выражений мы должны использовать один или набор логических операторов.

Давайте рассмотрим следующий список:

- `A && B` — эквивалент «И». Соответственно возвращает `true`, если `A` и `B` являются истиной.
- `A || B` — эквивалент логического «ИЛИ». Вернет `true` ели хотя бы одно из выражений является истинным.
- `A xor B` — этот оператор можно сравнить с «ТОЛЬКО ОДИН», соответственно вернет `true` если `A == true` и `B == false`, или наоборот.
- `!A` — данный оператор инвертирует значение `A`. То есть, если `A == true`, то он вернет `false` и наоборот.

Здесь самая главная «причуда» логических операторов — это их обозначения в C++. В остальном они интуитивно понятны.

Теперь давайте попробуем на примере скомбинировать несколько логических выражений и вывести их значения на экран. Заранее расскажу про следующую строку:

```
cout.setf(ios::boolalpha);
```

она отвечает за форматный вывод `bool` переменных (вывод слов вместо чисел). Дело в том, что по умолчанию C++ при выводе логических значений используются два значения:

- 1 для `true`;
- 0 для `false`;

Таким образом мы «приукрасим» вывод нашей программы и сделаем его более читабельным.

```

#include <iostream>
using namespace std;

int main() {
    cout.setf(ios::boolalpha);

    bool r; // создаем переменную bool типа
    int a = 10, b = 7; // a также две переменные типа int

    r = (a < b) && (b == 7); // r равно false, поскольку a > b
    cout << "r = " << r << endl; // вывод результата

    r = a < b || b == 7; // r равен true
    cout << "r = " << r << endl; // вывод результата

    r = (a < b) xor (b == 7); // r равен true, поскольку только b == 7 верно
    cout << "r = " << r << endl; // вывод результата

    r = !(a == 10 && (b <= 8 || true)); // комбинируем целую кучу операторов
    cout << "r = " << r << endl; // и снова выводим результат

    return 0;
}

```

Как видите мы можем пользоваться скобками, чтобы указать порядок выполнения логических операций также, как и при арифметических операциях. Также скобки можно и вовсе опустить, но это может создать путаницу в программе (именно поэтому я не рекомендую так делать).

Давайте посмотрим, что же выводит наша программа:

```

r = false
r = true
r = true
r = false

```

Выражение, вычисляющее значение логического типа может иметь следующий вид:

```

#include<iostream>
using namespace std;

int main() {
    int userSelection = 12;
    bool deleteFile = (userSelection == 12);
    cout << deleteFile << endl;

    return 0;
}

```


do ... while

do { } while (true);
бесконечный цикл

do { } while (false);
сразу выходит

false равно 0
true всё кроме нуля

```
do {  
  a(); b(); c();  
  if (conditionA) break;  
  e(); f(); g();  
  if (conditionB) break;  
  h(); i(); j();  
} while (false);
```

таким образом можно соскакивать с кода, это, так сказать, вежливый goto!

Пример № 1

```
void setup() {  
  Serial.begin(9600);  
  pinMode(9, OUTPUT);  
}  
  
void loop() {  
  int pot = A7;  
  int data = 0;  
  do {  
    Serial.println(data);  
    delay(50);  
  }  
  while((data = analogRead(pot)) != 0); // Присваивание с условием проверки  
}
```

Комментарий к коду.

Считается плохим тоном присваивание совмещать с условием. Если это необходимо, что нужно явно указывать то, что программист хочет проверить в условии.

Основные алгоритмические структуры

<https://prog-cpp.ru/algorithm-structure/>

<http://electricalschool.info/electronica/1918-logicheskie-jelementy-i-ili-ne-i-ne-ili.html>

<https://instrumentationtools.com/boolean-arithmetic/>

http://book.kbsu.ru/theory/chapter7/1_7_9.html

[https://pikabu.ru/story/](https://pikabu.ru/story/vyipusk_6_uslovnyie_operatoryi_i_tsiklyi_osnovyi_arduino_dlya_nachinayushchikh_45773)

[vyipusk_6_uslovnyie_operatoryi_i_tsiklyi_osnovyi_arduino_dlya_nachinayushchikh_45773](https://pikabu.ru/story/vyipusk_6_uslovnyie_operatoryi_i_tsiklyi_osnovyi_arduino_dlya_nachinayushchikh_45773)

Алгоритм - последовательность действий, исполнение которых приводит к искомому результату.

Свойства алгоритма:

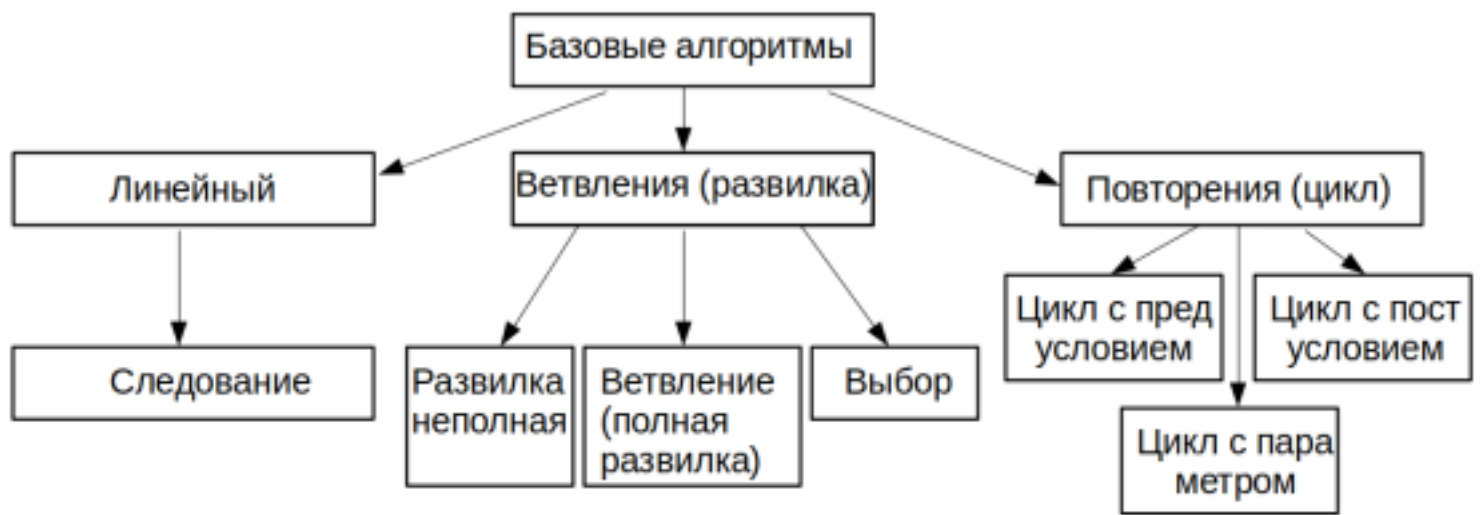
1. Определённость (однозначность) - каждое выражение определено и понятно исполнителю (компилятору и процессору) ;
2. Результативность (конечность) - приводит к конкретному результату за конечное количество шагов;
3. Дискретность - если алгоритм можно разложить на конечное количество простых шагов, понятных исполнителю;
4. Массовость - если он применим для любого набора исходных данных из множества допустимых.

Структура алгоритма:


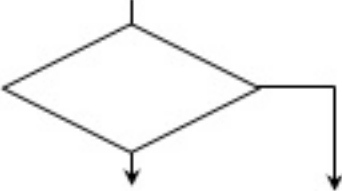
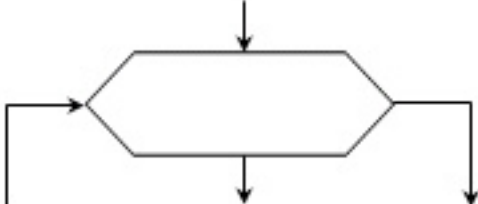
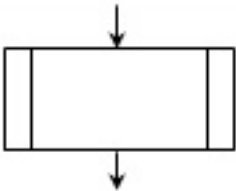





1. Заголовок алгоритма;
2. Определение используемых типов данных и констант;
3. Определение (ввод) начальных значений переменных;
4. Описание действий, их последовательное выполнение которых приводит к искомому результату;
5. Вывод искомого результата на устройство вывода;
6. Конец алгоритма.

Операция присваивания: Тип Переменной Имя переменной = значение.

Базовые алгоритмы

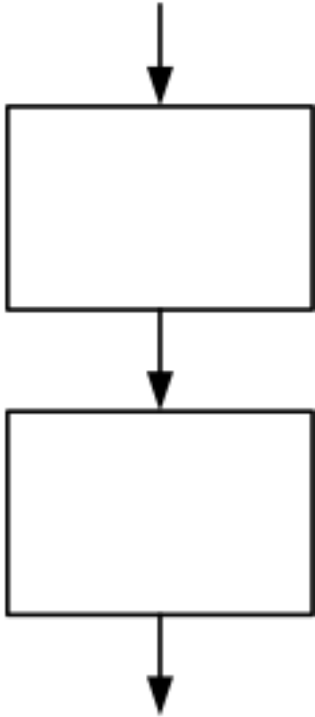


Язык блок-схем

Символ	Описание символа
	Процесс —формирование новых значений, выполнение арифметических или логических операций или действий, результаты которых запоминаются в оперативной памяти ЭВМ
	Решение — проверка условия: выбор одного из двух направлений выполнения алгоритма в зависимости от некоторого условия
	Модификация — организация циклических конструкций (начало цикла)
	Предопределенный процесс — вычисление по подпрограмме, использование ранее созданных и отдельно описанных алгоритмов
	Начало-конец программы или вход и выход в подпрограммах
	Ввод – вывод данных – связь алгоритма с внешним миром. Вывод может осуществляться на бумагу, экран монитора на магнитный диск или ленту
	Соединитель – разрыв линий потока
	Соединитель – перенос на другую страницу
	Комментарий – пояснения, содержание подпрограмм

Следование

Следование представляет собой последовательное выполнение операций и представляется алгоритмически последовательностью блоков «Процесс»:



Развилка (условие)

Развилка, в свою очередь, делится на

- неполную развилку;
- полную развилку;
- ветвление.

Развилка представляет собой блок выбора (проверка условия).

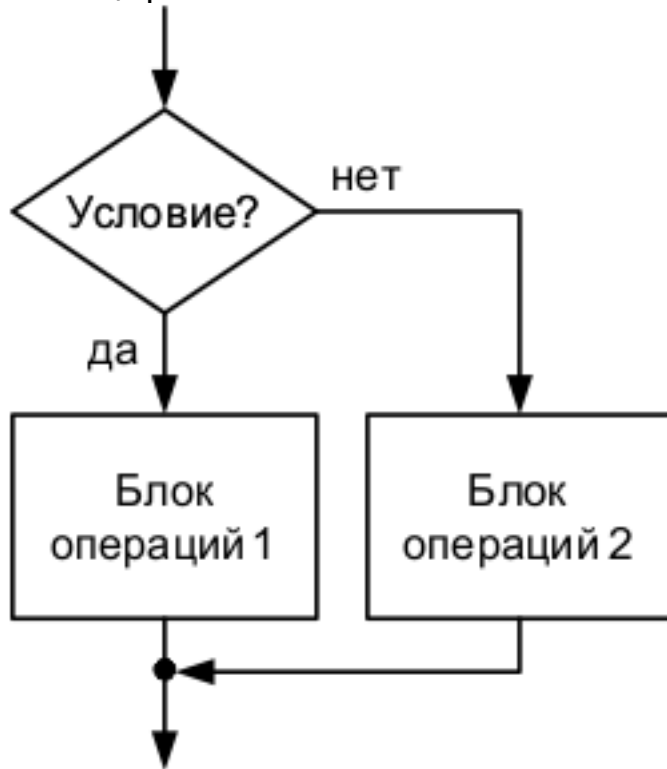
Неполная развилка выполняет последовательность операций только по одной из веток.



Реализация неполной развилки в Си имеет вид:

```
if (условие)
{
    операции;
}
```

Полная развилка выполняет последовательность операций по каждой из двух веток (при выполнении или невыполнении условия):

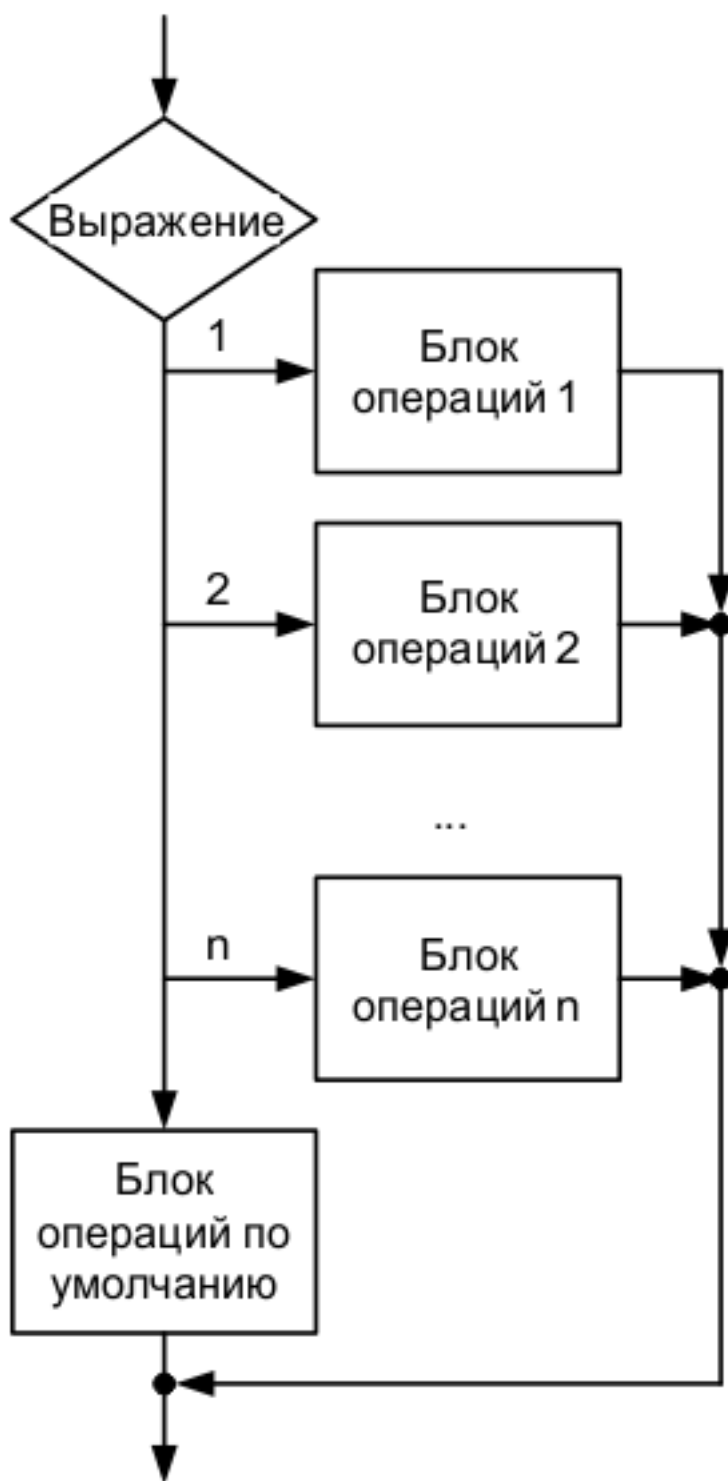


Реализация полной развилки в Си имеет вид:

```
if (условие)
{
    операции блока 1;
}
else
{
    операции блока 2;
}
```

Ветвление

Ветвление представляет собой операцию множественного выбора, при которой проверка условия может иметь более двух возможных вариантов:



Реализация ветвления в Си имеет вид:

```
switch (выражение)
{
    case 1:
        блок операций 1;
        break;
    case 2:
        блок операций 2;
        break;
    ...
    case n:
        блок операций n;
        break;
    default:
        блок операций по умолчанию;
```

}

Цикл

Существует 3 основных вида циклов:

- цикл с предусловием;
- цикл с постусловием;
- параметрический цикл.

Цикл с предусловием осуществляет проверку условия перед началом своего выполнения. В случае если условие не выполняется, происходит выход из цикла. Цикл с предусловием может не выполниться ни одного раза.

Реализация цикла с предусловием на Си:

```
while (условие)
{
    операции;
}
```

Цикл с постусловием всегда выполняется хотя бы один раз, поскольку проверка условия осуществляется после выполнения операций цикла.



Реализация на Си цикла с постусловием:

```
do {
```



```
операции;  
} while (условие);
```

Параметрический цикл — это цикл с заданным числом повторений.



Реализация параметрического цикла на Си:

```
for (П = НЗ; П != КЗ; П += Ш)  
{  
    операции;  
}
```

Где (П — параметр, НЗ — начальное значение, КЗ — конечное значение (в общем случае — условие продолжения цикла), Ш — шаг))

Тернарный (третичный) оператор

Тернарный оператор — операция, возвращающая свой второй или третий операнд в зависимости от значения логического выражения, заданного первым операндом.



```
var a = 10;  
var msg = (a == 10) ? "a = 10": "a != 10";  
alert(msg);
```

Флаги

<http://www.cyberforum.ru/cpp-beginners/thread1639876.html>

<https://toster.ru/q/470675>

http://www.tvd-home.ru/prog/7_1

https://ru.wikipedia.org/wiki/
%D0%A4%D0%BB%D0%B0%D0%B3_(%D0%BA%D0%BE%D0%BC%D0%BF%D1%8C%D

Constexpr

С помощью спецификатора constexpr можно создавать переменные и функции, которые будут рассчитаны на этапе компиляции. Это может ускорить работу созданной программы.

Пример 1

```
#include<iostream>
using namespace std;

constexpr double GetPi() { return 3.141593; }
constexpr double TwicePi() { return 2 * GetPi(); }

int main() {
    const double pi = 3.141593;

    cout << "Константа pi равна " << pi << endl;
    cout << "constexpr GetPi() возвращает " << GetPi() << endl;
    cout << "constexpr TwicePi()" << TwicePi() << endl;

    return 0;
}
```

Enum

Иногда некая переменная должна принимать значения только из определённого набора.

В данной ситуации подходит перечисление (Enumerations) - enum

Одной из основных ценностей перечисления является то, что числа расставляются на этапе компиляции и не требуют инициализации и оперативной памяти.

Example #1

```
#include<iostream>
using namespace std;

enum CardinalDirections
{
    Nord = 25,
    South,
    East,
    West
};

int main() {
    cout << "Направления и их назначения" << endl;
    cout << "North: " << Nord << endl;
    cout << "South: " << South << endl;
    cout << "East: " << East << endl;
    cout << "West: " << West << endl;

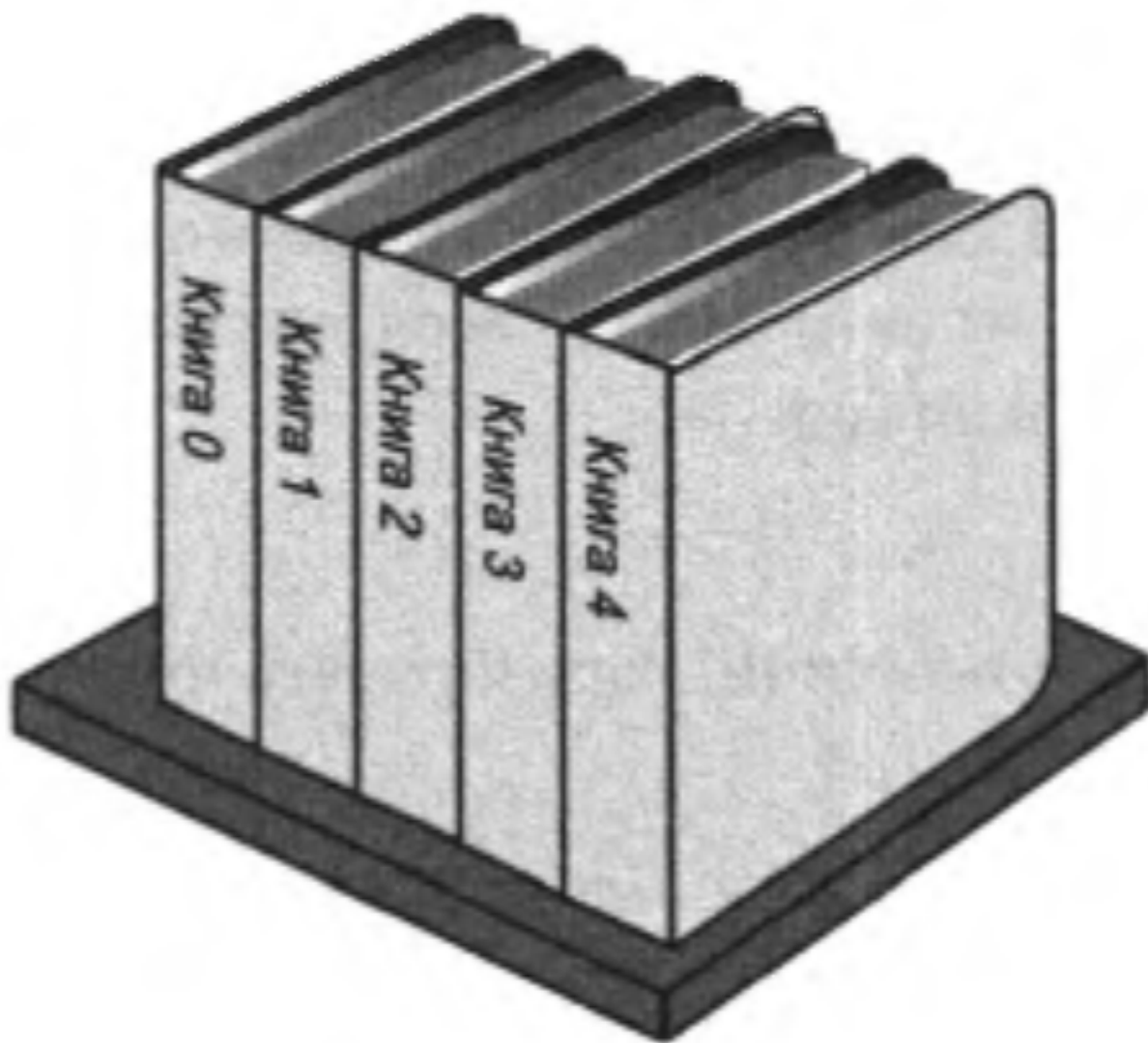
    CardinalDirections windDirection = South;
    cout << "windDirection = " << windDirection << endl;

    return 0;
}
```

Nord получила значение 25, что гарантирует то, что остальные за ней получат значение по порядку возрастания - 26,27,28.

Это одна из особенностей enum.

Array



Пример с применением доступа к элементам массива

```

#include<iostream>
using namespace std;

int main() {
    int myNumbers[5] = { 34, 56, -21, 5002, 365 };

    cout << "Элемент 0: " << myNumbers[0] << endl;
    cout << "Элемент 1: " << myNumbers[1] << endl;
    cout << "Элемент 2: " << myNumbers[2] << endl;
    cout << "Элемент 3: " << myNumbers[3] << endl;
    cout << "Элемент 4: " << myNumbers[4] << endl;

    return 0;
}

```

За пределы массива выходить нельзя!

Example #2

```

#include<iostream>
using namespace std;

int main() {
    // Устанавливаем размер массива
    const int ARRAY_LENGTH = 5;
    // Инициализированный массив из 5 целых чисел
    int myNumbers[ARRAY_LENGTH] = {5, 10, 0, -101, 20};

    cout << "Элемент 0: " << myNumbers[0] << endl;
    cout << "Элемент 1: " << myNumbers[1] << endl;
    cout << "Элемент 2: " << myNumbers[2] << endl;
    cout << "Элемент 3: " << myNumbers[3] << endl;
    cout << "Элемент 4: " << myNumbers[4] << endl;

    cout << "Введите индекс изменяемого элемента: " << endl;
    // Переменная для вводимого номера позиции из массива
    int elementIndex = 0;
    cin >> elementIndex;

    cout << "Введите новое значение: " << endl;
    int newValue = 0; // Переменная для нового значения
    cin >> newValue;

    myNumbers[elementIndex] = newValue; // Присваиваем элементу массива номер

    cout << "Элемент " << elementIndex << " myNumbers равен: ";
    cout << myNumbers[elementIndex] << endl;

    return 0;
}

```

Example #3

```

#include<iostream>
using namespace std;

constexpr int Square(int number) { return number * number; }

int main() {
    // Устанавливаем размер массива
    const int ARRAY_LENGTH = 5;
    // Инициализированный массив из 5 целых чисел
    int myNumbers [ARRAY_LENGTH] = {5, 10, 0, -101, 20};
    /* Использование спецификатора constexpr для
     * вычисления массива из 25 целых чисел
     */
    int moreNumbers[Square(ARRAY_LENGTH)] = {25}; // 5*5

    cout << "Введите индекс изменяемого элемента: " << endl;
    int elementIndex = 0; // Переменная для номера позиции из массива
    cin >> elementIndex;

    cout << "Введите новое значение: " << endl;
    int newValue = 0; // Переменная для значения
    cin >> newValue;

    myNumbers[elementIndex] = newValue; // Присваиваем элементу массива номер
    moreNumbers[elementIndex] = newValue; //

    cout << "Элемент " << elementIndex << " myNumbers равен: " << endl;
    cout << myNumbers[elementIndex] << endl;

    cout << "Элемент " << elementIndex << " moreNumbers равен: " << endl;
    cout << moreNumbers[elementIndex] << endl;

    return 0;
}

```

Example #4 с проверкой введенного индекса на соответствие длины массива


```

#include<iostream>
using namespace std;

int main() {
    // Устанавливаем размер массива
    const int ARRAY_LENGTH = 5;
    // Инициализированный массив из 5 целых чисел
    int myNumbers[ARRAY_LENGTH] = {5, 10, 0, -101, 20};

    cout << "Элемент 0: " << myNumbers[0] << endl;
    cout << "Элемент 1: " << myNumbers[1] << endl;
    cout << "Элемент 2: " << myNumbers[2] << endl;
    cout << "Элемент 3: " << myNumbers[3] << endl;
    cout << "Элемент 4: " << myNumbers[4] << endl;

    cout << "Введите индекс изменяемого элемента: " << endl;
    // Переменная для вводимого номера позиции из массива
    int elementIndex = 0;
    cin >> elementIndex;

    if(elementIndex <= 4) {
        cout << "Введите новое значение: " << endl;
        int newValue = 0; // Переменная для нового значения
        cin >> newValue;
        myNumbers[elementIndex] = newValue; // Присваиваем элементу массива номер
        cout << "Элемент " << elementIndex << " myNumbers равен: ";
        cout << myNumbers[elementIndex] << endl;
    }
    else {
        cout << "Вы вышли за пределы массива!" << endl;
    }

    return 0;
}

```

Example #5 Доступ к многомерным массивам

```

#include<iostream>
using namespace std;
// Доступ к элементам многомерного массива
/*
*      0      1      2
* 0 [-501][206] [2016]
* 1 [989] [101] [206]
* 2 [303] [456] [596]
*
*/

int main() {
    int threeRowsThreeColuns[3][3] = // Три массива по три элемента
    {{-501, 206, 2016}, {989, 101, 206}, {303, 456, 596}};
    cout << "Row 0: " << threeRowsThreeColuns[0][0] << " "
           << threeRowsThreeColuns[0][1] << " "
           << threeRowsThreeColuns[0][2] << endl;

    cout << "Row 1: " << threeRowsThreeColuns[1][0] << " "
           << threeRowsThreeColuns[1][1] << " "
           << threeRowsThreeColuns[1][2] << endl;

    cout << "Row 2: " << threeRowsThreeColuns[2][0] << " "
           << threeRowsThreeColuns[2][1] << " "
           << threeRowsThreeColuns[2][2] << endl;

    return 0;
}

```