Практика:

Ссылочка на все мои лабы. Там есть комменты и все такое, ошибки конечно же тоже, наверняка. Но основные алгоритмы и примеры задач в общем и и в целом разобраны.

Советую глянуть 5-ю и 6-ю лабы. Перестановка строк массива через переменную и указатель, перестановка столбцов, рекуррентные последовательности.

Вопросы

1. Понятие алгоритма, определение и правила построения алгоритмов.

(в методе нет, в конспекте тоже. Навр - лежать + сосать. **Опер**деление из инета, ваша фантазия приветствуется)
Алгоритм - это последовательность команд. набор инструкций, описывающих порядок действий для достижения результата.
этой хуеты нет в норм виде, сорян

2. Состав языка Си: алфавит, идентификаторы, ключевые слова, знаки операций, константы, комментарии. Структура простейшей программы.

- **Алфавит** языка Си состоит из прописных и строчных букв латинского алфавита, арабских чисел, специальных символов, пробельных и разделительных символов.

Из символов алфавита формируются **лексемы** (элементарные конструкции языка). К лексемам относятся: идентификаторы, зарезервированные слова, знаки операций, константы, разделители.

- **Идентификатор** – последовательность цифр и букв латинского алфавита, а также специальных символов при условии, что первой стоит буква или знак подчеркивания. (обычное слово короче, *Пр.* ааа, баа, АуЕ). Допустимо использовать любое количество символов, однако значимыми считаются только первые 32.

При выборе идентификатора необходимо:
1. Следить, чтобы идентификатор не совпадал с ключевыми зарезервированными словами и именами библиотечных функций;
2. Использовать с осторожностью символ подчеркивания в качестве первого символа идентификатора и комбинацию «_t» в конце идентификатора.

- **Ключевые слова** -- зарезервированы стандартом ANSI Си (для компилятора):
auto, double, int, struct, break, long, switch, register, typedef, char, extern, return, void, case, float, unsigned, default , for, signed, union, do, if, sizeof, else, while, volatile, continue, enum, short.

- **Комментарии** – текстовая или символьная информация, используемая для пояснения участков программы. Комментарии не влияют на ход выполнения программы, т. к. не являются лексемами и не включаются в содержимое исполняемого файла.

В С++ комментарии:
1. Начинаются последовательностью «//» и заканчиваются концом строки.
2. Начинаются последовательностью «/*» и заканчиваются последовательностью «*/».

- **Знак операции** – один или несколько символов, определяющих действие над операндами. Использование пробелов внутри знака операции не допускается.

- **Структура программы** -- http://informatics-lesson.ru/c/structure-program.php

3. Операция присваивания, ее различные формы (инкремент, декремент). Стандартные библиотеки языка C/C++, функции библиотеки math.h.

- Формат операции присваивания
  оператор_1=оператор_2;
  В переменную оператор_1 заносится значение оператор_2. В качестве оператора_1 можно использовать только переменную. В качестве оператора_2 можно использовать константу, переменную, выражение или функцию.
  Допустимо использовать: a=b=c=d;
  это равнозначно a=b; b=d; d=c;
  Часто в программировании используются операции такого типа:
  оператор_1=оператор_2 знак_операции оператор_2;
  для сокращения можно и нужно использовать короткую форму записи:
  оператор_1 знак_операции=оператор_2;
  a=a+2; равносильно записи a+=2;
  Частный случай, когда оператор_2 равен еденице называется операцией инкремента(++) или декремента(--):
  оператор_1++; **инкремент(++)** a++; равносильно a=a+1 равносильно a+=1
  оператор_1--; **декремент(--)** a--; равносильно a=a-1 равносильно a-=1
  ЕСТЬ ДВЕ ФОРМЫ (чекайте начало конспекта, Навр много распрягал про это)
  1)префиксная ++a;
  2)постфиксная a++;
  при **префиксной** сначала выполняется инкремент/декремент, а затем арифметич. операции.
  при **постфиксной** сначала выполняется арифметич. операции, а затем инкремент/декремент.

1.7. Стандартные библиотеки C++

При создании исполняемого файла к исходной программе подключаются файлы, содержащие различные библиотеки. Такие файлы содержат уже откомпилированные функции и, как правило, имеют расширение *lib*. На этапе компиляции компоновщик извлекает из библиотечных файлов ипользуемые в программе функции. Для осуществления связи с библиотечным файлом к программе подключается заголовочный файл, который содержит информацию об именах и типах функций, расположенных в библиотеке. Подключение заголовочных файлов осуществляется с помощью директив препроцессора.

> В стандартном C++ заголовочные файлы не имеют расширения, а для файлов, унаследованных от Си, следует указывать расширение. Например, #include <math.h>.

1.8. Функции библиотеки math.h

Все аргументы в тригонометрических функциях задаются в радианах. Параметры и аргументы всех остальных функций (кроме функции abs) имеют тип **double**. Математические функции перечислены в табл. 1.1.

Таблица 1.1

Математическая функция	Функция библиотеки math.h	Содержание вычислений
1	2	3
$\|x\|$	abs(x)	Вычисление абсолютного значения целого числа. Например: s = abs(−3) → Результат s = 3 s = abs(3) → Результат s = 3 s = abs(−3.9) → Результат s = 3 s = abs(3.2) → Результат s = 3
arccos(x)	acos(x)	Вычисление значения арккосинуса числа x. Значение x может быть задано только из диапазона −1...1. В результате выполнения функции возвращается число из диапазона $-\pi/2 \ldots \pi/2$. Например: s = acos (−1) → Результат s = 3.14159 s = acos (0.4) → Результат s = 1.15928 s = acos (1.5) → Результат s = −1.#IND

1	2	3
arcsin(x)	asin(x)	Вычисление значения арксинуса числа x. Значение x может быть задано только из диапазона –1...1. В результате выполнения функции возвращается число из диапазона 0…π. Например: s = asin (−1) → Результат s = −1.5708 s = asin (0.9) → Результат s = 1.11977
arctg(x)	atan(x)	Вычисление значения арктангенса. В результате выполнения функции возвращается число из диапазона −π/2…π/2. Например: x = atan (3.5) → Результат s = 1.2925
arctg(x/y)	atan2(x,y)	Вычисление значения арктангенса двух аргументов. В результате выполнения функции возвращается число из диапазона −π…π. Если y равняется 0, то функция возвращает π/2, если $x > 0$; 0, если $x = 0$; −π/2, если $x < 0$. Например: s = atan2(4.5, 9.2) → Результат s = 0.454914 s = atan2(0, 0) → Результат s = 0 s = atan2(5.4, 0) → Результат s = 1.5708 s = atan2(−7.3, 0) → Результат s = −1.5708
Округление к большему	ceil(x)	Функция возвращает действительное значение соответствующее наименьшему целому числу, которое больше или равно x. Например: s = ceil(−3.4) → Результат s = −3 s = ceil(3.4) → Результат s = 4
cos(x)	cos(x)	Вычисление cos(x)
ch(x)	cosh(x)	Вычисление косинуса гиперболического.
e^x	exp(x)	Вычисление экспоненты числа x
$\lvert x \rvert$	fabs(x)	Вычисление абсолютного значения x
Округление к меньшему	floor(x)	Функция возвращает действительное значение, соответствующее наибольшему целому числу, которое меньше или равно x. Например: s = floor (−3.4) → Результат s = −4 s = floor (3.4) → Результат s = 3

Остаток от деления x на y	fmod(x,y)	Функция возвращает действительное значение, соответствующее остатку от деления x на y. Например: s = fmod (3, 4) → Результат $s = 3$ s = fmod (8, 3) → Результат $s = 2$ s = fmod (6.4, 3.1) → Результат $s = 0.2$
$\ln(x)$	log(x)	Вычисление натурального логарифма x
$\lg_{10}(x)$	log10(x)	Вычисление десятичного логарифма x
x^y	pow(x, y)	Возведение x в степень y
$\sin(x)$	sin(x)	Вычисление $\sin(x)$
$\mathrm{sh}(x)$	sinh(x)	Вычисление синуса гиперболического x
$\sqrt{x}$	sqrt(x)	Вычисление квадратного корня x
$\mathrm{tg}(x)$	tan(x)	Вычисление тангенса x
$\mathrm{tgh}(x)$	tanh(x)	Вычисление тангенса гиперболического x

4. Базовые типы данных. Понятие операции и выражения.

- **Тип данных** позволяет определить, какие значения могут принимать переменные, какая структура и какое количество ячеек используется для их размещения и какие операции допустимо над ними выполнять.

1. Скалярный тип -- данные, представляемые одним значением (числом, символом) и размещаемые в одной ячейке из нескольких байтов.
2. Структурированные типы определяются пользователем как комбинация скалярных и описанных ранее структурированных типов.

- **Базовыми типами** данных являются: целый, действительный (вещественный) и символьный тип. Могут быть константами и переменными. В отличие от переменных константы не могут изменять свое значение во время выполнения программы.

Тип	байт	Диапазон принимаемых значений
целочисленный (логический) тип данных		
bool	1	0 / 255
целочисленный (символьный) тип данных		
char	1	0 / 255
целочисленные типы данных		
short int	2	-32 768 / 32 767
unsigned short int	2	0 / 65 535
int	4	-2 147 483 648 / 2 147 483 647
unsigned int	4	0 / 4 294 967 295
long int	4	-2 147 483 648 / 2 147 483 647
unsigned long int	4	0 / 4 294 967 295
типы данных с плавающей точкой		
float	4	-2 147 483 648.0 / 2 147 483 647.0
long float	8	-9 223 372 036 854 775 808 .0 / 9 223 372 036 854 775 807.0
double	8	-9 223 372 036 854 775 808 .0 / 9 223 372 036 854 775 807.0

- **Константы целого типа** – последовательность цифр, начинающаяся со знака минус для отрицательных констант, и со знака плюс или без него для положительных констант. Для обозначения констант типа long после числа ставят букву L или l.

Десятичные константы: последовательность чисел от 0 до 9, начинающаяся не с нуля, например, 334.

Восьмеричные константы: последовательность чисел от 0 до 7, начинающаяся с нуля, например, 045.

Шестнадцатеричные константы: последовательность чисел от 0 до 9 и букв от A до F, начинающаяся с символов 0x, например 0xF5C3.

- **Символьный тип данных**

Символьный тип предназначен для хранения одного символа, для чего достаточно выделить 1 байт памяти. Данные такого типа рассматриваются компилятором как целые, поэтому в переменных типа signed char можно хранить целые числа из диапазона –128…127. Для хранения символов используется unsigned char, который позволяет хранить 256 символов кодовой таблицы *ASCII* (*American Standard Code for Information Interchange* – Американский стандартный код для обмена информацией). Стандартный набор символов *ASCII* использует только 7 битов для каждого символа (диапазон 0…127). Добавление 8-го разряда позволило увеличить количество кодов таблицы *ASCII* до 255. Коды от 128 до 255 представляют собой расширение таблицы *ASCII* для хранения символов национальных алфавитов, а также символов псевдографики.

Значения кодовой таблицы *ASCII* с номерами 0…32 и 127 содержат непечатаемые символы, которые не имеют графического представления, но влияют на отображение текста. Символы с кодами 32…127 представлены в табл. 2.1. Символы с кодами 128…255 (кодовая таблица 866 – *MS-DOS*) представлены в табл. 2.2.

- **Логический тип данных.** 2 значения: true (1) или false (0). Но так как диапазон допустимых значений типа данных bool от 0 до 255, то необходимо было как-то сопоставить данный диапазон с определёнными в языке программирования логическими константами true и false. Таким образом, константе true эквивалентны все числа от 1 до 255 включительно, тогда как константе false эквивалентно только одно целое число — 0.

- **Операции и выражения.** Любое выражение языка состоит из операндов (переменных, констант и др.), соединенных знаками операций.

Знак операции - это символ или группа символов, которые сообщают компилятору о необходимости выполнения определенных арифметических, логических или других действий.

Операции выполняются в строгой последовательности. Величина, определяющая преимущественное право на выполнение той или иной операции, называется **приоритетом**. Порядок выполнения операций может регулироваться с помощью круглых скобок. Таблица приоритетов:

Уровень приоритета	Тип операции	Операторы
1	Разрешение области действия	::
2	Унарные	префикс ++, префикс --, –>
	Другие	( ), [ ], . (точка)
3	Унарные	+, –, !, *, &, sizeof, new, delete, явное преобразование типа
4	Арифметические	*, /, %
5	Арифметические	+, –
6	Поразрядный сдвиг	<<, >>
7	Сравнение	>, <, >=, <=
8	Сравнение	==, !=
9	Поразрядные логические	&
10	Поразрядные логические	^
11	Поразрядные логические	|
12	Логические	&&
13	Логические	||
14	Условная	?:
15	Присваивания	=, *=, /=, %=, +=, –=, <<=, >>=, &=, ^=, |=
16	Унарные	постфикс ++, постфикс --
	Последовательность	, (запятая)

5. Арифметические операции, преобразование типов при выполнении операций, операция явного приведения типа. Понятие переменной.

Арифметич операции. Хз че тут рассказывать, просто из таблицы запомните какие есть команды и всё.

3. Операции в языке C++

3.1. Арифметические операции

Самыми простыми арифметическими операциями являются +, –, *, /. Данные операции применимы как к целым, так и к вещественным типам данных и правила их использования аналогичны их использованию в математических вычислениях. Порядок следования операций можно изменять с помощью скобок.

Для работы только с целыми числами существует операция получения остатка от деления – %.

Например: 10 % 6 = 4, 7 % 10 =7, 10 % 5 =0.

2.7. Неявное преобразование типов

В большинстве случаев преобразование типов происходит автоматически с использованием приоритета типов. Типы имеют следующий порядок приоритетов:

char → short → int → long → float → double → long double

Приоритет увеличивается слева направо (в сторону увеличения занимаемой типом памяти). При проведении арифметических операций действует правило: операнд с более низким приоритетом преобразуется в операнд с более высоким приоритетом, а значения типов char и short всегда преобразуются к типу int.

При использовании операции присваивания преобразование типов не происходит. Поэтому при присваивании переменным, имеющим меньший приоритет типа, значений переменных, имеющих больший приоритет типа, возможна потеря информации.

Например, необходимо вычислить: s = a + b, где s – переменная типа double; a – символ; b – переменная типа int. Пусть a = 'd', b = 45. Выражение будет рассчитываться следующим образом. Так как в арифметическом выражении присутствуют две переменные различных типов, то переменная с меньшим приоритетом (a) будет приведена к типу int. Для этого в памяти компьютера создается временная переменная типа int, которая будет хранить номер символа 'd' равный 100 (см. табл. 2.1). После этого выполняется операция суммирования, результат которой будет равен 145 (100+45). Полученное значение (145) присваивается переменной s. При выполнении операции присваивания преобразование типа не происходит, но размер переменной типа double больше переменной типа int, поэтому потеря информации в этом случае не происходит.

2.8. Явное преобразование типов

Если неявное преобразование типов не приводит к требуемому результату, программист может задать преобразование типов явным образом:

static_cast <*тип*> (*переменная*)

Из языка Си сохранилась устаревшая форма приведения типов

(*тип*) *переменная*

или

тип (*переменная*)

Пример:

```
int a,b,s,f;
        a = b = 2147483647;
        s = (a*b)/a;
        f = (static_cast <double> (a)*b)/a;
```

Результат: $s = 0, f = 2147483647$.

При расчете переменной **s** вычисленное значение произведения **a** на **b** выходит за границы диапазона значений, которые могут храниться в переменной типа **int**. Создаваемая для хранения промежуточного результата временная переменная типа **int** получает ошибочную информацию, поэтому результат вычисления будет неверным.

В следующей строке переменная **a** явным образом приводится к типу **double**. Следовательно, результат вычисления произведения будет храниться во временной переменной наибольшего по длине типа (из **int** и **double**) – типа **double**. Полученный результат не выходит за границы диапазона значений типа **double**, поэтому ошибка не возникает.

Переменная — именованный участок памяти для хранения данных определенного типа

6. Операции сравнения, логические операции, побитовые операции. Оператор безусловной передачи управления goto. Понятие «блок».

3.3. Операции сравнения

Операции сравнения применяются при работе с двумя операндами и возвращают *true* (1), если результат сравнения – истина, и *false* (0) – если результат сравнения – ложь. В языке Си определены следующие операции сравнения:

< *(*меньше*)*, **<=** (меньше или равно), **>** (больше),

>= (больше или равно), **!=** (не равно), **= =** (равно).

Операнды должны иметь одинаковый тип (допустимо сравнивать целый и действительный тип).

3.4. Логические операции

Логические операции работают с операндами скалярных типов и возвращают результат логического типа. Определены три логические операции «!», «&&» и «||».

Унарная логическая операция НЕ (!) возвращает *true* (1), если операнд имеет нулевое значение, и *false*(0), если операнд отличен от нуля.

Например:

```
k = 5;
a = !(k > 0);
```

Результат: 0 (*false*), т. к. операнд имеет значение 1 (*true*), которое изменяется операцией «!» на обратное – 0 (*false*).

Логическая операция И (&&) возвращает *true* (1), если операнды имеют ненулевые значения, и *false*(0) – если хотя бы один операнд имеет нулевое значение.

Например:

```
k = 5;
a = (k > 0 && k <= 10 && k != 5);
```

Результат: 0 (*false*), т. к. два первых операнда имеют значение 1 (*true*), а последний операнд имеет значение 0 (*false*).

Если ввести:

```
k = 5;
a = (k > 0 && k <= 10 && k == 5);
```

Результат: 1 (*true*), т. к. все операнды имею значение 1 (*true*).

Логическая операция ИЛИ (||) возвращает *true* (1), если хотя бы один операнд имеет ненулевое значение, и *false*(0), если все операнды имеют нулевое значение.

Например:

```
k = 5;
```

3.5. Поразрядные логические операции

Поразрядные логические операции работают с двоичным представлением целых чисел.

Определены следующие операции:

«~» – поразрядное отрицание;

«&» – поразрядное И;

«|» – поразрядное ИЛИ;

«^» – поразрядное исключающее ИЛИ;

«<<» – поразрядный сдвиг влево;

«>>» – поразрядный сдвиг вправо.

Унарная поразрядная операция «~» инвертирует каждый бит операнда.

Таблица истинности для операций «&», «|», «^» представлена в табл. 3.1.

Таблица 3.1

Значение бит	b_1 & b_2	b_1 | b_2	b_1 ^ b_2
$b_1 = 0, b_2 = 0$	0	0	0
$b_1 = 0, b_2 = 1$	0	1	1
$b_1 = 1, b_2 = 0$	0	1	1
$b_1 = 1, b_2 = 1$	1	1	0

Операция поразрядного сдвига «>>» сдвигает биты левого операнда на число разрядов, указанное правым операндом. Правые биты теряются. Если левый операнд является беззнаковым числом, то левые освободившиеся биты заполняются нулями. Если есть знак символа, то ячейки заполняются этим символом. Сдвиг целого числа эквивалентен целочисленному делению на 2^n.

Операция поразрядного сдвига «<<» сдвигает биты левого операнда на число разрядов, указанное правым операндом. Правые биты теряются. Левые биты теряются, а правые заполняются нулями. Если есть знак символа, то ячейки заполняются этим символом. Сдвиг целого числа эквивалентен умножению на 2^n.

- **goto.** Операторы безусловной передачи управления изменяют нормальную последовательность программы и показывают, какой оператор должен выполняться следующим в результате некоторого вычисления и сколько раз. Их всего 3: оператор завершения **break**, оператор продолжения **continue**, оператор перехода **goto**.
    - Оператор завершения **break** обеспечивает возможность выхода из оператора цикла или оператора-переключателя switch, внутри которого он был употреблен, в ближайший оператор, охватывающий этот оператор цикла или оператор-переключатель switch.
    - Оператор продолжения **continue** обеспечивает возможность перехода к следующему шагу цикла, внутри которого он был употреблен.
    - Оператор перехода **goto** обеспечивает возможность перехода на метку, по форме совпадающую с идентификатором, внутри одной функции. Переход допускается в любое место из любого. (но есть функции и поэтому он -- **бесполезное говно**)

- **Блок** -- Группа операторов, заключенная в фигурные скобки, рассматривается компилятором как один составной оператор.

7. Оператор условного перехода if-else, оператор альтернативного выбора switch. Условная тернарная операция «?:». Операторы передачи управления

- **тернарная операция «?:»**

Типа пишите чет вроде **a*b==2 ? cout<<"хуй соси" : cout<<"Губой тряси";**
Эта херота значит, что
Если произведение **a** на **b** равно двум, то выводится "хуй соси", в ином случае выводится "губой тряси"

- **Оператор условного перехода if-else**

4.1. Оператор условной передачи управления if

Формат оператора выбора:

if (*логическое_выражение***)** *оператор_1*;
else *оператор_2*;

Если *логическое_выражение* истинно, то выполняется *оператор_1*, иначе – *оператор_2*.

Например:

if (f > 10) x = 3;
else x = 34;

Оператор имеет сокращенную форму:

if (*логическое_выражение***)** *оператор_1*;

Например: **if** (f == 0) x = 4;

Логическое_выражение всегда располагается в круглых скобках. Если *оператор_1* или *оператор_2* содержат более одного оператора, то используется блок.

В качестве *оператора_1* и *оператора_2* могут быть использованы операторы if. Такие операторы называют *вложенными*. Во вложенных операторах if ключевое слово else принадлежит ближайшему предшествующему ему if.

Например:

if (*логическое_выражение_1***)** *оператор_1*;
if (*логическое_выражение_2***)** *оператор_2*;
else *оператор_3*;

Оператор_3 будет выполняться, если *логическое_выражение_2* ложно. Значение *логического_выражения_1* не оказывает влияния на работу *оператора_3*.

Изменить порядок проверки можно, используя фигурные скобки:

if (*логическое_выражение_1***) {**
оператор_1;
if (*логическое выражение_2***)** *оператор_2*;
}
else *оператор_3*;

Оператор_3 будет выполняться, если *логическое_выражение_1* ложно. Значение *логического_выражения_2* не оказывает влияния на работу *оператора_3*.

4.2. Условная операция

В программировании достаточно часто распространена операция сравнения двух аргументов, например:

```
if (a > b) max = a;
    else max = b;
```

В связи с этим была разработана специальная условная операция. Ее общая форма:

условие **?** *оператор_1* **:** *оператор_2*;

Если значение *условие* истинно, то результатом операции является *оператор_1*, иначе – *оператор_2*.

Например, найти наибольшее из двух чисел:

```
max = a > b ? a : b;
```

Условие может быть любым скалярным выражением, а операторы могут иметь практически любой тип.

Применение условной операции сокращает код, однако не оказывает никакого влияния на скорость выполнения программы.

4.3. Оператор множественного выбора switch

Если в программе большое число ветвлений, которые зависят от значения одной переменной, то можно вместо вложенной последовательности конструкций if .. else использовать оператор switch. Общая форма оператора следующая:

```
switch(переменная выбора) {
        case const1: операторы_1 ;   break;
                                  …
        case constN: операторы_n;   break;
        default : операторы_n+1;
                                  }
```

Работает оператор следующим образом. Сначала анализируется значение переменной выбора и проверяется, совпадает ли оно со значением одной из констант. При совпадении выполняются операторы этого case. Конструкция default (может отсутствовать) выполняется, если результат выражения не совпал ни с одной из констант. Тип переменной выбора может быть целым, символьным или перечисляемым. Тип констант сравнения должен совпадать с ти-

- **Операторы передачи управления**

5.4. Операторы и функции передачи управления

Операторы и функции передачи управления позволяют изменить стандартный порядок выполнения операторов.

5.4.1. Оператор continue

Используется при организации циклических процессов. Операторы тела цикла, находящиеся после оператора continue, пропускаются, а управление передается следующему циклу. Оператор continue обычно используется вместе с

оператором if для того, чтобы при определенных значениях данных завершить текущий цикл и передать управление следующему циклу.

5.4.2. Оператор break

Позволяет перейти к следующему за блоком оператору. Например, в циклах он обеспечивает досрочный выход из цикла, а в операторе switch – выход из блока выбора. Следует обратить внимание на то, что оператор break выходит только из текущего блока, т. е. в случае вложенных циклов выход происходит только из одного цикла.

5.4.3. Оператор return

Завершает выполнение текущей функции и передает управление вызывающей функции. Управление передается следующему за вызовом текущей функции оператору.

Формат оператора:

return *выражение*;

Если значение выражения задано, то результат возвращается в вызывающую функцию в качестве значения вызываемой функции.

5.4.4. Функция exit

Находится в библиотеке *stdlib.lib*. Корректно прерывает выполнение программы, записывая все буферы, закрывая все потоки. Формат функции:

void exit(**int**)

Параметр является служебным сообщением системе. Как правило, 0 говорит об успешном завершении программы, ненулевые значения – об ошибке.

5.4.5. Функция abort

Находится в библиотеке *stdlib.lib*. Генерирует «молчаливое» исключение, не связанное с сообщением об ошибке. Корректно прерывает выполнение программы, записывая все буферы, закрывая все потоки. Формат функции:

```
void abort(void)
```

5.4.6. Оператор безусловного перехода goto

Передает управление оператору, помеченному меткой. Использование оператора goto существенно снижает читабельность программы и увеличивает вероятность ошибки. Поэтому использование goto в программах нежелательно.

Примером обоснованного применения оператора безусловного перехода может служить необходимость организации выхода сразу из нескольких вложенных циклов:

```
for ( i = 0; i < n; i++)
  for (j = 0; j < m; j++) {
        if (логическое_выражение) goto met;
                    }
met:  …
```

8. Операция последовательного вычисления «,» (запятая). Оператор цикла с предусловием while.

- **Операция последовательного вычисления** последовательно вычисляет два своих операнда, сначала первый, затем второй. Оба операнда являются выражениями. Синтаксис операции:
  <*выражение1*>, <*выражение2*>

Знак операции - запятая, разделяющая операнды. Результат операции имеет значение и тип второго операнда. Ограничения на типы операндов (т. е. типы результатов выражений) не накладываются, преобразования типов не выполняются.

Операция последовательного вычисления обычно используется для вычисления нескольких выражений в ситуациях, где по синтаксису допускается только одно выражение(например цикле for: for(i=j=1; i+j<20; **i+=i, j--**))

-

5.2. Оператор цикла while

Оператор цикла с предусловием

while (*логическое_выражение*)
{
Тело_цикла
}

организует повторение операторов тела цикла до тех пор, пока значение логического выражения истинно. Как только значение логического выражения становится равным 0 (*false*), циклический процесс прекращается и выполняется первый после цикла оператор. Если условие цикла сразу равно 0 (*false*), то тело цикла не выполняется ни разу.

9. Операторы цикла: оператор цикла с постусловием do-while, оператор цикла с постусловием и коррекцией for. Операторы break, continue.

Юзайте обычный for и не ебите головы. Но, для общего развития можно и нужно знать другие варианты этой хероты.

break, continue. Были в операторах передачи управления

5.1. Оператор цикла for

Общий вид оператора:

for (*инициализирующее_выражение*; *логическое_выражение*;
инкрементирующее_выражение)

{

Тело цикла

}

Чаще всего все три выражения содержат одну переменную, которую называют счетчиком цикла.

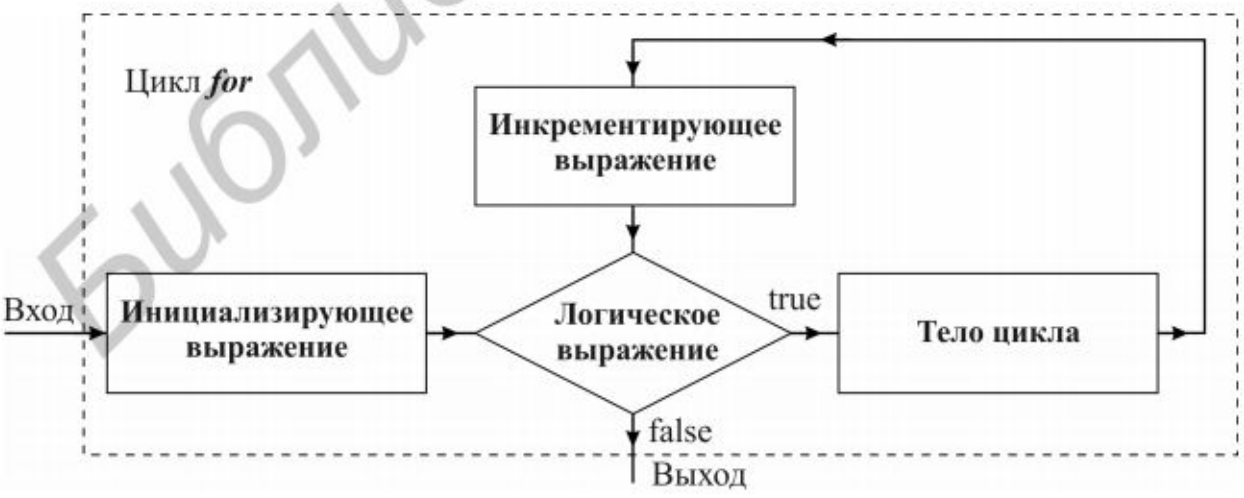
Инициализирующее_выражение выполняется только один раз в начале выполнения цикла. Вычисленное значение инициализирует счетчик цикла.

Логическое_выражение проверяется в начале каждого цикла. Если результат равен 1 (*true*), то цикл повторяется, иначе выполняется следующий за телом цикла оператора.

Инкрементирующее_выражение предназначено для изменения значения счетчика цикла. Модификация счетчика происходит после каждого выполнения тела цикла.

Тело цикла – последовательность операторов, которая выполняется многократно, до тех пор, пока не будет выполнено условие выхода из цикла. Тело цикла может содержать внутри себя любы конструкции языка C++, в том числе – любое количество вложенных циклов.

Схема работы цикла for представлена на рис. 5.1:



Рассмотрим работу следующего оператора:

```
for (i = 1;  i < 10;  i++) cout << i << endl;
```

В начале цикла в переменную *i* будет занесено число 1. Затем будет проверено значение логического выражения, и т. к. оно имеет значение *true* (1 < 10), то будет выполнено тело цикла (значение *i* будет выведено на экран). После этого выполняется инкрементирующее выражение i++ и снова будет проверено значение логического выражения. Тело цикла будет выполняться до тех пор, пока логическое выражение не примет значения *false* (10 < 10). В результате на экран будут выведены цифры от 1 до 9.

Если необходимо вывести цифры от 1 до 10, то можно использовать конструкцию:

```
for (i = 1;  i <= 10;  i++)
```

Однако в C++ обычно используют конструкции со строгим неравенством:

```
for (i = 1;  i < 11;  i++)
```

Удобно совмещать выполнение инкрементирующего выражения с описанием счетчика цикла.

```
for ( int i = 1;  i < 11;  i++)
```

Такое написание удобно тем, что объявленная переменная по стандарту C++ будет существовать только внутри цикла, и далее имя этой переменной можно использовать для других целей.

В своей версии языка C++ фирма Microsoft область видимости переменной, объявленной внутри оператора for, расширила до конца блока, содержащего этот for.

Любая из секций в операторе for не является обязательной, поэтому может отсутствовать одно или несколько выражений. Допустимо такое написание бесконечного цикла:

```
for ( ; ; )
```

Для размещения в одной секции оператора for несколько операторов используется операция «запятая», которая позволяет в тех местах, где допустимо использование только одного оператора, размещать несколько операторов. Формат операции

Оператор_1, Оператор_2, …, Оператор_N

Программа для вычисления факториала числа *n* может выглядеть следующим образом:

```
for (int f=1, i=1;  i<=n;  f*=i, i++);
```

Точка с запятой в конце оператора for означает, что тело цикла отсутствует. Так как в Visual C++ область видимости переменных f и i продлена до конца следующего блока, то значение f можно вывести во внешнем по отношению к for блоке.

5.2. Оператор цикла while

Оператор цикла с предусловием

```
while (логическое_выражение)
   {
Тело_цикла
    }
```

организует повторение операторов тела цикла до тех пор, пока значение логического выражения истинно. Как только значение логического выражения становится равным 0 (*false*), циклический процесс прекращается и выполняется первый после цикла оператор. Если условие цикла сразу равно 0 (*false*), то тело цикла не выполняется ни разу.

5.3. Оператор цикла do-while

Оператор цикла с постусловием

```
do {
Тело_цикла
   }
       while (логическое_выражение);
```

организует повторение операторов тела цикла до тех пор, пока значение логического выражения истинно. Как только значение логического выражения становится равным 0 (*false*), циклический процесс прекращается и выполняется первый после цикла оператор. Вне зависимости от значения логического выражения тело цикла будет выполнено не менее одного раза.

> Оператор цикла do-while опасен тем, что тело цикла обязательно выполняется хотя бы один раз. Следовательно, необходимо проверять условие его завершения до входа в цикл. Поэтому, если это возможно, следует избегать использования этого оператора.

10. Декларация статических массивов, размещение данных в памяти, правила обращения к элементам массивов.

- **Массив --** структура данных, хранящая набор значений (элементов массива), идентифицируемых по индексу или набору индексов, принимающих целые (или приводимые к целым) значения из некоторого заданного непрерывного диапазона.

Массив характеризуется: именем массива, типом хранимых данных, **размером** (количеством элементов) и **размерностью** (формой представления элементов массива). Номер ячейки массива называется индексом. Индексы массивов должны иметь целый тип, а элементы массивов могут иметь любой тип.

- Статический массив

6.1. Одномерные массивы

Объявление одномерного массива:

тип имя_массива [**размер**];

Пример объявления массива:

int c[4];

Размер статического массива задается константой или константным выражением целого типа.

Индексы в языке C/C++ начинаются с 0. Например, вышеобъявленный массив состоит из четырех элементов: c[0], c[1], c[2] и c[3]. Расположение элементов массива в памяти указано на рис. 6.1.

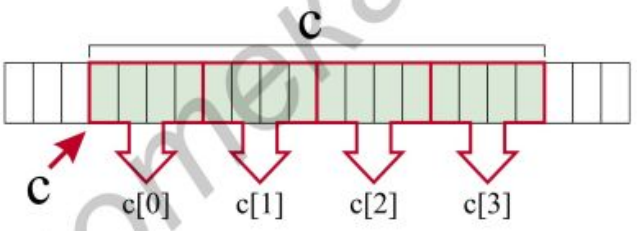


Рис. 6.1

Одновременно с объявлением можно инициализировать элементы массива:

double b[4] = {1.5, 2.5, 3.75, 3.04};

int a[4] = {1, 4};

Если в группе инициализации не хватает начальных значений, то оставшиеся элементы заполняются нулями, например, массив a: a[0] = 1, a[1] = 4, a[2] = 0 и a[3] = 0. // Зависит от компилятора, но в вижле gcc, который делает именно так.

Памяти занимается столько, сколько размерность умноженная на размер типа данных. **НО** когда объявляется массив в виде int array[25], то этим определяется не только выделение памяти для двадцати пяти элементов массива, но и **для указателя** с именем array, значение которого равно адресу первого по счету (нулевого) элемента массива.

- Для доступа к элементам массива существует два различных способа.
  - Первый способ связан с использованием обычных индексных выражений в квадратных скобках, например, array[16]=3 или array[i+2]=7.
  - Второй способ доступа к элементам массива связан с использованием адресных выражений и операции разадресации в форме *(array+16)=3 или *(array+i+2)=7. При таком способе доступа адресное выражение равное адресу шестнадцатого элемента массива тоже может быть записано разными способами *(array+16) или *(16+array).
- При реализации на компьютере первый способ приводится ко второму, т.е. индексное выражение преобразуется к адресному. Для приведенных примеров array[16] и 16[array] преобразуются в *(array+16). Для доступа к начальному элементу массива (т.е. к элементу с нулевым индексом) можно использовать просто значение указателя array или ptr. Любое из присваиваний

*array = 2; array[0] = 2; *(array+0) = 2; *ptr = 2; ptr[0] = 2; *(ptr+0) = 2;

присваивает начальному элементу массива значение 2, но быстрее всего выполнятся присваивания *array=2 и *ptr=2, так как в них не требуется выполнять операции сложения.

11. Ввод-вывод одномерного и двумерного массивов, заполнение массива случайными равномерно распределенными числами.

Ну, лол. Тип воодите всякую вашу дристоту через цикл, границей которого будет размер массива. Помните, что все многомерные массивы можно юзать как адскисукапиздецкак длинный массив (бесполезно, но помнить надо, вроде)
****

```
Int mas[n];
For (int i=0; i<n; i++)
cin>>mas[i];
```

Это для одномерного.
****

```
Int mas[n][m];
For (int i=0; i<n; i++)
        {
                For (int j=0; j<m; i++)
                        {
                                cin>>mas[i][j];
                        }
        }
```

Это для двумерного.
Заполнять рандомно не надо ваще никак, есть риск отчисления в армию. Но, для проверорк самого себя почему бы и нет

Пример 6.7. Заполнение двумерного массива случайными действительными числами из диапазона *n*min…*n*max.

Для генерации случайных чисел используют функции библиотеки *stdlib.lib*:

rand() – генерирует псевдослучайное число из диапазона 0…RAND_MAX (32767).

srand() – устанавливает аргумент для создания новой последовательности псевдослучайных целых чисел, возвращаемых функцией **rand()**.

```
srand( time( NULL ) );
for (i=0; i<n; i++)
 for (j=0; j<m; j++)
 s[i][j] = static_cast <double> (rand()) /
                        (RAND_MAX + 1) * (nmax - nmin) + nmin;
```

12. Основные алгоритмы работы с элементами массива: нахождение суммы, произведения, минимального и максимального, среднего.

```
int n;
cout << "Size:";
cin >> n;
int *arr = new int[n];  // объявление массива

for (int i=0; i<n; i++){  // вывод эл-тов массива
   cout << " " << arr[i];
}

int pr=1;  // нахождение среднего арифм, суммы и пр-ния эл-тов
double sredn, sum=0;
for (int i=0; i<n; i++){
   sum += arr[i];
   pr *= arr[i];
}
sredn = sum/n;
```

```
int min,max; min = max = arr[0];  // нахождение макс и мин эл-тов
for (int i=1; i<n; i++){
   if (arr[i] < min) min = arr[i];
   if (arr[i] > max) max = arr[i];
}

for (int i=0; i<n; i++){  // удаление эл-та из массива
   if (arr[i] < 0)
       for (int j=i+1; j<n; j++)  // смещение всех эл-тов на 1 влево
           arr[j-1]=arr[j];
       n--; i--;   // чтобы не потерять 1 проверку
}

delete []arr; // освобождение памяти
```

13. Декларация указателя. Указатель на объект, указатель типа void. Инициализация указателя, значение NULL. Операции над указателями.

Земля металлом, если ты всё еще плохо шаришь за указатели. Так вот
Inr *piska, *ochko;
Char *pidor;
Звезда-пезда ставится именно перед самой переменной, которая будет указателем, а не после типа. Да, это важно. А еще указатель любого типа занимает 4 байта памяти, тоже вполне себе ловушка от Навра.
Если скрины ниже не помогли понять, то иди лазить в ютабе. Подбери видос на свой вкус и цвет, главное чтоб про указатели.

7. Использование указателей

7.1. Объявление указателя

Память компьютера представляет собой массив последовательно пронумерованных ячеек. При объявлении данных в памяти выделяется непрерывная область для их хранения. Например, для переменной типа **int** выделяется участок памяти размером 4 байта. Номер первого байта, выделенного под переменную участка памяти, называется адресом этой переменной.

Указатель – это переменная, значением которой является адрес участка памяти. Формат объявления указателя:

*Тип_данного *имя_указателя*;

Например:

```
int *a;    double *b, *d;    char *c;
```

На один и тот же участок памяти может ссылаться любое число указателей, в том числе и различных типов. Допустимо описывать переменные типа указатель на указатель (указатель на ячейку памяти, которая в свою очередь содержит адрес другой ячейки памяти). Например:

```
int *um1, **um2, ***um3;
```

В языке Си существует три вида указателей:

1. Указатель на объект известного типа.
2. Указатель типа void. Применяется в случаях, когда тип объекта заранее не определен.
3. Указатель на функцию. Позволяет обращаться с функциями, как с переменными.

7.2. Операции над указателями

7.2.1. Унарные операции

Над указателями можно провести две унарные операции:

1. «**&**» **(взять адрес)**. Операция позволяет получить адрес переменной.
2. «*****» **(операция разадресации)**. Позволяет получить доступ к величине, расположенной по указанному адресу.

7.2.2. Арифметические операции и операции сравнения

При выполнении арифметических операций с указателями автоматически учитывается размер данных, на которые они указывают.

Инкремент и декремент. Перемещает указатель к следующему или предыдущему элементу массива.

Например:

```
int *um, a[5] = {1,2,3,4,5};
um = a;
```

```
um++;
cout << *um << endl;   // Выводит: 2
```

Сложение. Перемещение указателя на число байт, равное произведению размера типа данного, на которое ссылается указатель, на величину добавляемой или вычитаемой константы. Например:

```
int *um, a[5] = {1,2,3,4,5};
um = a;
cout << *um << endl;   // Выводит: 1
   um += 3;
cout << *um << endl;   // Выводит: 4
```

Вычитание. Разность двух указателей равна числу объектов соответствующего типа, размещенных в данном диапазоне адресов. Например:

```
int *um, a[5] = {1,2,3,4,5};
 um = &a[0];
 un = &a[4];
  k = un - um;
cout << k << endl;   // Выводит: 4
```

Операции сравнения. Сравнивают адреса объектов.

7.3. Инициализация указателей

Инициализация пустым значением. Например:

```
int *a = NULL;
int *b = 0;
```

Присваивание указателю **адреса уже существующего объекта**. Например:

```
int k = 23;
int *uk = &k;   // или   int *uk(&k);
int *us = uk;
```

Присваивание указателю **адреса выделенного участка динамической памяти**:

```
int *s = new int;
int *k = (int *) malloc(sizeof(int));
```

В примере использована операция sizeof, которая определяет размер указанного параметра в байтах.

14. Связь указателей с массивами. Создание динамических массивов и правила работы с ними.

- Указатели топ тема при работе со строками матрицы. Пример замены 2х строк местами:

```
int *str1 = arr[0];
int *str3 = arr[2];

for (int i = 0; i < N; i++){
    swap(*(str1+i), *(str3+i));
}
```

15. Динамическое выделение памяти с помощью библиотечных функций (операции new, delete)

- **Динамическим** называется массив, размер которого может изменяться во время исполнения программы. Возможность изменения размера отличает динамический массив от статического, размер которого задаётся на момент компиляции программы.

Динамические массивы дают возможность более гибкой работы с данными, так как позволяют не прогнозировать хранимые объёмы данных, а регулировать размер массива в соответствии с реально необходимыми объёмами.

- Динамическая память (heap) – специальная область памяти, в которой во время выполнения программы можно выделять и освобождать место в соответствии с текущими потребностями. Доступ к выделяемым участкам памяти осуществляется через указатели. Для работы с динамической памятью в языке Си (библиотека malloc.lib) определены следующие функции:

void *malloc(size) – выделяет область памяти размером *size* байт. Возвращает адрес выделенного блока памяти. Если недостаточно свободного места для выделения заданного блока памяти, то возвращает NULL.

void *calloc(n, size) – выделяет область памяти размером *n* блоков по *size* байт. Возвращает адрес выделенного блока памяти. Если недостаточно свободного места для выделения заданного блока памяти, то возвращает NULL. Вся выделенная память заполняется нулями.

void *realloc(*u) – изменяет размер ранее выделенной памяти, связанной с указателем *u* на новое число байт. Если память под указатель не выделялась, то функция ведет себя как *malloc*. Если недостаточно свободного места для выделения заданного блока памяти, то функция возвращает значение NULL.

void free(*u) – освобождает участок памяти, связанный с указателем *u*.

В языке C++ для выделения и освобождения памяти определены операции new и delete. Имеются две формы операций:

тип *указатель = new *тип* (*значение*) – выделение участка памяти в соответствии с указанным типом и занесение туда указанного значения.

delete указатель – освобождение выделенной памяти.

тип *указатель = new *тип*[n] – выделение участка памяти размером *n* блоков указанного типа.

delete []*указатель* – освобождение выделенной памяти.

Операция delete не уничтожает значения, связанные с указателем, а разрешает компилятору использовать данный участок памяти.

- **Работа с динамическим массивом. (~15 вопрос)**. В C++ операции new и delete предназначены для динамического распределения памяти компьютера. Операция new выделяет память из области свободной памяти, а операция delete высвобождает выделенную память. Выделяемая память, после её использования должна высвобождаться, поэтому операции **new** и **delete** используются парами. Даже если не высвобождать память явно, то она

освободится ресурсами ОС по завершению работы программы. Рекомендую все-таки не забывать про операцию delete.

- **Вкратце**:
    - Одномерный:

```
double *arr = new double[n]
arr = new double(n*sizeof(double))
arr = static_cast <double*>(malloc(n*sizeof(double)));
```

    - Двумерный:

```
int N = 0, M = 0; // строки, столбцы
      int **arr;

      arr = new int *[N]; // Выделение памяти на N строк массива
      for (int i = 0; i < N; i++){
          arr[i] = new int [M];  // Растягивание каждой строки
      }



      for(int i=0; i < N; i++)
          delete [ ]arr[i];
      delete [ ]arr;
      arr = nullptr;
```

Почитать **подробнее**: http://cppstudio.com/post/432/

- **Работа с двумерными массивами.**

```
  int n, m;

   cout << "Rows:";
   cin >> n;
   cout << "Columnes: ";
   cin >> m;
   int **arr = new int *[n];  //   объявление n строк массива
   for (int i=0; i<n;i++){
       arr[i] = new int[m];  // растягивание до m эл-тов
   }

   for (int i=0; i<n; i++){  // заполнение массива
       cout << endl;
       for (int j=0; j<m; j++){
           cout << "Value of " << "[" << i << "][" << j << "] element: ";
```

```
            cin >> arr[i][j];
        }
    }

    int s1=0, s2 = 0;  // Нахождение суммы элементов диагоналей.
    for (int i=0; i<n; i++){
        s1 += arr[i][n-i-1];
        s2 += arr[i][i];
    }

    int sum = 0;  // Нахождение суммы элементов, лежащих выше главной диаг
    for (int i=0; i<n-1; i++) // не делать лишних итераций по n+ не выйти за размеры в след цикле
        for (int j=i+1; j<m; j++)
            sum += arr[i][j];

    int k1=1, k2=2, t;
    for (int j=0; j<m; j++){  // Перестановка строк с номерами k1 и k2. Вариант 1.
        t = arr[k1][j];
        arr[k1][j] = arr[k2][j];
        arr[k2][j] = t;
    }

    int *pntr1=arr[1], *pntr2=arr[2];
    for (int j=0; j<m; j++){  // Перестановка строк с номерами k1 и k2. Вариант 2.
        int t = pntr1[j];
        pntr1[j] = pntr2[j];
        pntr2[j] = t;
    }

/*Получение из матрицы n-го порядка матрицы порядка n-1 путем удаления из исходной матрицы строк и столбцов,
* на пересечении которых расположен элемент с наименьшим значением
*/
    int imin=0, jmin=0;
    for (int i=0; i<n; i++) // нахождение координат минимального элемента
        for (int j=0; j<m; j++)
            if (arr[i][j] < arr[imin][jmin]){
                imin = i; jmin = j;
            }
```

```
  // Работает сдвигом сдвигом эл-тов по строкам и стобцам. Лучше нарисовать.
  for (int i=0; i<n; i++){
      for (int j=jmin; j<m-1; j++)
          arr[i][j] = arr[i][j+1]; // смещение всех эл-тов правее минимального в каждой строке на 1 влево.
  }m--; // это удаляет СТОЛБЕЦ, поэтому в СТРОКЕ становится на 1 элемент меньше

  for (int j=0; j<m; j++){ // удаляем строку
      for (int i=imin; i<n-1; i++)
          arr[i][j] = arr[i+1][j];
  }n--; // количество строк стало на 1 меньше


  for (int i=0; i<n; i++){  // вывод эл-тов массива
      cout << endl;
      for (int j=0; j<m; j++){
          cout << " " << arr[i][j];
      }
  }


  for (int i=0; i < n; i++){ // Очищение памяти
      delete[] arr[i];
  }
  delete[] arr;
```

16. Строка – массив типа char. Стандартные функции библиотеки string.h

- В методе норм. Функции -- тут.
- В использовании функций ничего такого нет, но в реальной жизни такой роскоши, как изменение или копирование данных у вас скорее всего не будет (представь бд с тексктом на несколько гб (которые мы кстати проходим на 3м курсе) и что тебе надо обработать текст из нее. Короче, насиловать входные данные -- говнокод.

- Примеры работы:

```
int main() {
     char str[100];
     fgets(str, 100, stdin);
     strcat(str, " "); // пробел в конце, да, так можно

     // 1. Подсчитать количество слов в строке.
```

```
Слова отделяются друг от друга одним пробелом.
Перед первым словом пробела нет
      int c = 0;
      for (int i = 0; str[i] != '\0' ; i++) {
          if (str[i] != ' '){
               c++;
               while (str[i] != ' ') i++;
          }
      }
      cout << endl << c;

      //2. Подсчитать, какое количество букв 'a' в
первом слове строки. Слова отделяются друг от
друга одним пробелом. Перед первым словом
пробела нет.
      int j=0, count=0;
      while (str[j] != ' '){ // зачем идти до конца, если мы
идем до 1го пробела?
      if (str[j] == 'a') count++; // TODO: придумайте, как
сделать это, если все же есть 1 пробел
      j++;
      }
      cout << endl << count;
      return 0;
}
```

- Полезно также быть тру-джуном и самом писать библиотечные функции:

```
void strwr(char *str){
      int c = 'a' - 'A'; // сдвиг по таблице

      for (int i=0; i < strlen(str)-1; i++){
      if (str[i] >= 'A' && str[i] <= 'Z')
            str[i] += c;
      }
}

char *cutstr(char *str, int n, int k){// вырезать k символов с
n-ной позиции
    static char returnstr[100];
    int i = 0
    for (int counter=n-1; counter < n - 1 + k; counter ++){
       returnstr[i] = str[counter];
       i++;
    }
}

int lengthString(char *str){
```

```
	int i = 0;
	while (str[i] != '\0'){
	i++;
	}

	return i;
}

char *reverseString(char *str){
	int length = lengthString (str); // находим длину строки
	char temp; // временная переменная для хранения переставляемого символа

	for (int counter = 0; counter < (length / 2); counter++){
		temp = str[length - 1 - counter]; // символ с конца строки сохраняем во временную переменную
		str[length - 1 - counter] = str[counter];
		str[counter] = temp;
	}
	return str;
}
```

- **Замечание** Для ввода в разных IDE используются разные функции:
  - Старые версии вижлы: gets(str);
  - Новая версия, там немного изменили библиотеку: gets_s(str);
  - Работает и cin >> str, но могут возникнуть проблемы с буфером.
  - Для линуксоводов или просто нормальных людей: fgets(str, 100, stdin);

9. Использование строковых переменных

В языке C++ имеется два способа работы со строковыми данными: использование массива символов типа char и использование класса *string*. В данном пособии рассматривается первый способ организации работы со строками.

9.1. Объявление строк

Объявление строки аналогично объявлению массива:

char имя строки [размер]

В отличие от массива строка должна заканчиваться нулевым символом '\0' – (нуль-терминатором). Длина строки равна количеству символов плюс нулевой символ. При наборе нулевой символ помещается в конец сроки автоматически. Например, в строке

char st1[10] = "123456789";

символы располагаются следующим образом:

'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'	'\0'

Если размер строки не объявлен явно, то он будет установлен автоматически и будет равен количеству введенных символов +1.

char st2[ ] = "1234";

Разрешено использовать указатели на строку:

char *st3 = st2;

Доступ к отдельным символам строки осуществляется по их индексам. Например: st[2] = 'e';

> В языке Си одиночные кавычки используются для обозначения символов, а двойные – для обозначения строк.

Массив строк объявляется как двумерный массив символов:

char *имя*[*количество строк*][*количество символов в строке*];

Например,

char str[10][5].

Обращение к нулевой строке массива строк: str[0].

17. Декларация структуры (struct). Создание структурных переменных. Обращение к элементам структуры. Вложенные структуры.

10. Типы данных, определяемых пользователем

10.1. Объявление и использование структур

Структура – составной тип данных, в котором под одним именем объединены данные различных типов. Отдельные данные структуры называются *полями*.

Объявление структуры:

```
struct им_структуры
{
        тип_элемента_1   имя_элемента_1;
        тип_элемента_2   имя_элемента_2;
           ...
        тип_элемента_n   имя_элемента_n;
};
```

Например:

```
struct struc1
{
    int m1;
    double m2, m3;
};
```

Поля структуры могут быть любого типа, в том числе массивами и структурами.

После фигурной скобки допустимо указывать переменные соответствующего структурного типа:

```
struct struc1
{
    int m1;
    double m2, m3;
} a, b ,c;
```

Объявление переменной структурного типа:

```
struc1 x;
```

К отдельным частям структуры можно *обращаться* через составное имя. Формат обращения:

имя_структуры.имя_поля

или

указатель_на_структуру -> имя_поля

Например, если структура объявлена следующим образом:

```
struct struc1
{
    int m1;
    double m2, m3;
} x, *y;
```

то обратиться к полю m1 можно (после выделения памяти для *y*):

```
x.m1 = 35;
```

или

```
(&x)->m1 = 35;
```

или

```
y->m1 = 35;
```

или

```
(*y).m1 = 35;
```

Правила работы с полями структуры идентичны работе с переменными соответствующих типов. Инициализировать переменные-структуры можно путем помещения за объявлением списка начальных значений.

```
struct struc1
{
    int m1;
    double m2, m3;
} a = {5, 2.6, 34.2};
```

В качестве полей могут быть использованы другие структуры.

```
struct struc1
{
    int m1;
    double m2, m3;
      struct
      {
            int mm1;
      } m4;
} s;
```

Обращение к полю mm1 в этом случае будет следующим:

```
s.m4.mm1 = 3;
```

Если имя структуры не указывается, то такое определение называется *анонимным*.

Допустимо использовать операцию присваивания для структур одного типа. Например:

```
struc1 x, y;
    …
  x = y;
```

В этом случае все значения полей структуры y копируются в соответствующие поля структуры x.

Из структур, как правило, организуют массивы:

```
struct struc1
{     int m1;
      double m2, m3;
};
…
struc1 ms[100];  // Объявление массива структур
…
ms[99].m1 = 56;  // Обращение к полю массива структур
```

Пример. Имеется список жильцов многоквартирного дома. Каждый элемент списка содержит следующую информацию: фамилия, номер квартиры, количество комнат в квартире. Вывести в алфавитном порядке фамилии жильцов, проживающих в двухкомнатных квартирах. Память для хранения списка выделять динамически.

```
#include <iostream.h>
#include <string.h>

int main ()
{
  struct tzhilec
  {
      char fio[50];
      int nomer;
      int nkomnat;
  } *spisok;

 int n, i, j;
  cout << "Vvedite kolichestvo zhilcov: " << endl;
  cin >> n;
     spisok = new tzhilec[n];
  for (i=0; i<n; i++)
  {
```

```
        cout << "Vvedite familiju: ";
                cin >> spisok[i].fio;
        cout << "Vvedite nomer kvartiry: ";
                 cin >> spisok[i].nomer;
        cout << "Vvedite kolichestvo komnat: ";
                cin >> spisok[i].nkomnat;
        cout << endl;
      }

    tzhilec tmp;
       for (i=0; i< n-1; i++)
           for (j=i+1; j<n; j++)
          if (spisok[i].nkomnat == 2 && spisok[j].nkomnat == 2
                        && strcmp(spisok[i].fio, spisok[j].fio )== 1)
                        {
                                tmp   = spisok[i];
                                spisok[i] = spisok[j];
                                spisok[j] =tmp;
                        }
    for (i=0; i<n; i++)
           if (spisok[i].nkomnat == 2)
        cout << spisok[i].fio << "  nomer kvartiry - "
                                        << spisok[i].nomer << endl;
    delete [ ]spisok;
      return 0;
    }
```


18. Перечисления (enum), объединения (union).

Дзен и смысл этих двух придет чуть позже, но основы таковы:

- **Объединения (union) --** это объект, позволяющий нескольким переменным различных типов занимать **один участок памяти**. Объявление объединения похоже на объявление структуры:

```
union union_type {
      int a; double b;
};
```

Размер памяти, выделяемый под объединение равен **размеру самого большого поля**. Т.е. такая union_type будет занимать только 8 байт памяти(double), при этом такая же структура занимала бы 8+4=12 байт.

Объединения могут использоваться для: 1) Для экономии памяти (особенно во встроенных системах). 2) Для исследования значений отдельных байтов многобайтных величин. 3) Для интерпретации данных, расположенных в некоторой области памяти.

- **Перечисления (enum)** -- тип данных, который включает множество именованных целочисленных констант. Именованные константы, принадлежащие перечислению, называются перечислимыми константами. Объявление: enum color { r, g, b };
  Тут по порядку: r=0, g=1; b=2.
- Остальное -- тут. Пример использования:

```
enum days_of_week { Sun, Mon, Tue, Wed, Thu, Fri, Sat };
int main()
{
      days_of_week day1, day2; //define variables of type days_of_week
      day1 = Mon;
      day2 = Sat;

      int diff = day2 - day1; //can do integer arithmetic
      cout << "Days between = " << diff << endl;

      if(day1 < day2) //can do comparisons
      cout << "day1 comes before day2" << endl;
      return 0;
}

/*
 Days between = 6
 day1 comes before day2
*/
```

19. Понятие функции, описание и определение функции. Вызов функции.

- **Функция** - это совокупность объявлений и операторов (именованный блок кода), обычно предназначенная для решения определенной задачи. Каждая функция должна иметь имя, которое используется для ее объявления, определения и вызова.

20. Передача данных в функцию по значению, по указателю, по ссылке.

- Прежде всего -- повторить функции и указатели. Если мы передаем данные по значению, то мы передаем их **копию** --> сами данные не изменяться вне функции. Если же по ссылке или указателю (что почти одно и то же), то мы работаем непосредственно с переданными данными и любое их изменение в функции изменит их глобально. В принципе, технически разница между указателем и ссылкой лишь в том, что саму ссылку нельзя изменить (а указатель можно). [Вот тут бóльший список отличий.] Кроме этого, разница ещё синтаксическая: с ссылкой вы обращаетесь как будто это переменная, а с указателем нужно его правильно получать/разыменовывать:
- Примеры:

```
void Foo1(int a){ // передача по значению
    a = 1;  //  работа с копией
}

void Foo2(int &a){ // передача по ссылке
    a = 2; // не нужно разыменовывать
}

void Foo3(int *a){ // передача по указателю
    *a = 3; // разыменование
}
```

- При работе с этими функциями:

```
      int value = 5;
      cout << "value: " << value << endl;

      cout << "Foo1: " << value << endl;
      Foo1(value);
      cout << "value: " << value << endl << endl;

      cout << "Foo2: " << value << endl;
      Foo2(value);  // хоть функция и получит адрес,
передаем без &
      cout << "value: " << value << endl << endl;

      cout << "Foo3: " << value << endl;
      Foo3(&value);
      cout << "value: " << value << endl << endl;

// a = 5
// a = 5 // т.к. работаем с локальной копией
переменной
// a = 2
// a = 3
```

21. Параметры функций по умолчанию, функции с переменным числом параметров.

22. Встраиваемые функции. Перегрузка функций. Передача массивов в функцию. Указатель на функцию.

- Встроенные -- тут. (Вкратце -- сразу использование кода, быстрее)
- Перегрузка -- тут. (Вкратце -- одно имя на разные функции, потому что параметры)
- Передача массива. Одномерные -- тут и тут. Двумерные -- вот тут прям хорошо.

Примеры для динамических:

1. Одномерный

```
void Foo1(int *, int);

int main(){
      int N;
      cin >> N;
      int *arr = new int [N];

      for (int i=0; i<N; i++){  // з а п о л н е н и е  м а с с и в а
      cin >> arr[i];
      }
      Foo1(arr, N);

      delete []arr;  // о с в о б о ж д а й т е  п а м я т ь  б л я т ь
}

void Foo1(int *arr, int N){  // ф у н к ц и я  в ы в о д а
      for (int i=0; i<N; i++){
      cout << arr[i];
      }
}
```

2. Двумерный:

```
void Foo1(int **, int, int);
int main(){
      int N, M;
      cin >> N >> M;

      int **arr = new int *[N];
      for (int i = 0; i < N; i++) {
      arr[i] = new int [M];
      }

      for (int i=0; i<N; i++){  // з а п о л н е н и е  м а с с и в а
      cout << endl;
      for (int j = 0; j < M; j++) {
            cin >> arr[i][j];
      }
      }
      Foo1(arr, N, M);

      for(int i=0; i < N; i++)  // у д а л я й т е  и х  б л я т ь
          delete [ ]arr[i];
      delete [ ]arr;
}

void Foo1(int **arr, int N, int M){  // ф у н к ц и я  в ы в о д а
      for (int i=0; i<N; i++){
          cout << endl;
          for (int j = 0; j < M; j++) {
```

```
                cout << arr[i][j];
            }
        }
}
```

- **Указатель на функцию.** Сложная и не очень полезная фигня. Тыц и Тыц. (Вкратце -- функция тоже объект в памяти → на него можно ссылаться. Удобно только при создании массива функций. Например, для тестов.)

23. Классы памяти. Время жизни и область видимости переменных.

Переменная, которую ты объявил глобально, правильно называть **голая жопа в окне** © Рак + Шатилова.

12. Область видимости и классы памяти

Область видимости определяет, в каких частях программы возможно использование данной переменной, а *класс памяти* – время, в течение которого переменная существует в памяти компьютера. Период времени между созданием и уничтожением переменной называется *временем жизни* переменной.

В языке C++ определены 4 класса памяти:

Автоматический, локальный (***auto***) класс памяти. Область видимости локальных переменных ограничена функцией или блоком, в котором она объявлена. Время жизни локальной переменной − промежуток времени между ее объявлением и завершением работы функции или блока, в которых она объявлена. Ограничение времени жизни переменной позволяет экономить оперативную память. Этот класс памяти используется по умолчанию.

Статический, локальный (***static***) класс памяти. Переменная имеет такую же область видимости, как и автоматическая. Время жизни статической локальной переменной − промежуток времени между первым обращением к функции, ее содержащей, и окончанием работы программы. Инициализация переменной происходит только при первом обращении к функции. Компилятор хранит значение переменной от одного вызова функции до другого.

Внешний, глобальный (***extern***) класс памяти. Глобальные переменные объявляются вне функций и доступны во всех функциях, находящихся ниже описания глобальной переменной. В момент создания глобальная переменная инициализируется нулем. Включение ключевого слова *extern* позволяет функции использовать внешнюю переменную, даже если она определяется позже в этом или другом файле. Память для глобальных переменных выделяется в начале программы и освобождается при завершении ее работы.

Регистровый, локальный (***register***) класс памяти. Является всего лишь «пожеланием» компилятору помещать часто используемую переменную в регистры процессора для ускорения скорости выполнения программы. Если компилятор отказался помещать переменную в регистры процессора, то переменная становится «автоматической».

24. Стандартные библиотечные функции для организации ввода-вывода информации (getc, gets, scanf, putc, puts, printf). Спецификации преобразований для данных различных типов.

- Функции языка С для работы с файлами. Текстовый ввод-вывод. Почитать подробно -- тут.
- Ссылочки отдельно:
  - getc, gets
  - putc, puts,
  - scanf, printf.
- Спецификации типов -- предмет документации, который определяет язык программирования, чтобы пользователи и разработчики языка могли согласовывать, что означают программы на данном языке [wiki].
- Преобразования типов: явное (ручное) и неявное(компилятор умный). Тыц и тыц.

2.7. Неявное преобразование типов

В большинстве случаев преобразование типов происходит автоматически с использованием приоритета типов. Типы имеют следующий порядок приоритетов:

char → short → int → long → float → double → long double

Приоритет увеличивается слева направо (в сторону увеличения занимаемой типом памяти). При проведении арифметических операций действует правило: операнд с более низким приоритетом преобразуется в операнд с более высоким приоритетом, а значения типов char и short всегда преобразуются к типу int.

При использовании операции присваивания преобразование типов не происходит. Поэтому при присваивании переменным, имеющим меньший приоритет типа, значений переменных, имеющих больший приоритет типа, возможна потеря информации.

Например, необходимо вычислить: s = a + b, где s – переменная типа double; a – символ; b – переменная типа int. Пусть a = 'd', b = 45. Выражение будет рассчитываться следующим образом. Так как в арифметическом выражении присутствуют две переменные различных типов, то переменная с меньшим приоритетом (a) будет приведена к типу int. Для этого в памяти компьютера создается временная переменная типа int, которая будет хранить номер символа 'd' равный 100 (см. табл. 2.1). После этого выполняется операция суммирования, результат которой будет равен 145 (100+45). Полученное значение (145) присваивается переменной s. При выполнении операции присваивания преобразование типа не происходит, но размер переменной типа double больше переменной типа int, поэтому потеря информации в этом случае не происходит.

25. Директивы препроцессора.

using namespace std -- использование пространства стандартных имен библиотеки.

1.6. Директивы препроцессора

Препроцессор – специальная часть компилятора, обрабатывающая директивы до начала процесса компиляции программы. Директива препроцессора начинается с символа #, который должен быть первым символом строки, затем следует название директивы. В конце директивы точка с запятой не ставится. В случае необходимости переноса директивы на следующую строку применяется символ '\'.

Для подключения к программе заголовочных файлов используется директива **include**. Если идентификатор файла заключен в угловые скобки, то поиск файла будет вестись в стандартном каталоге, если – в дойные кавычки, то поиск проводится в следующем порядке:

– каталог, в котором содержится файл, включивший директиву;
– каталоги файлов, которые были уже включены директивой;
– текущий каталог программы;
– каталоги, указанные опцией компилятора '\I';

– каталоги, заданные переменной окружения inclide.

Обработка препроцессором директивы include сводится к тому, что на место директивы помещается копия указанного в директиве файла.

Для определения символических констант используется директива define. Например, если определить в начале программы:

```
#define PI  3.14159265359
```

то во всем тексте при компиляции идентификатор PI будет заменен текстом 3.14159265359. Замена идентификатора константы не производится в комментариях и в строках. Если замещающий текст в директиве не задан, то во всем тексте соответствующий идентификатор стирается.

Описание констант с помощью директив препроцессора характерно для языка Си. В C++ рекомендуется использовать ключевое слово const, например:

```
const double pi = 3.14159265359;
```

Директива define используется также для написания макросов:

```
#define имя(параметры)реализация
```

В программе имя макроса заменяется на строку его реализации.

Например, имеется следующее макроопределение:

```
#define MAX(A,B) ((A)>(B)?(A):(B))
```

Если в программе встречается строка

```
s = MAX(a,b);
```

то перед компиляцией каждая макрокоманда заменяется макроопределением:

```
s = ((a)>(b)?(a):(b));
```

Желательно каждый параметр помещать в фигурные скобки, т. к. их отсутствие может спровоцировать ошибку.

Например создадим макрос:

```
#define SQR(A) (A*A)
```

При использовании в программе

```
s = SQR(a + b);
```

будет сформирована строка, содержащая ошибку:

```
s = a + b * a + b;
```

Надо писать следующим образом:

```
#define SQR(A) ((A) * (A))
```

тогда строка будет выглядеть следующим образом:

```
s= (a + b) * (a + b);
```

Для отмены действия директивы #define используется директива #undef. Синтаксис этой директивы следующий:

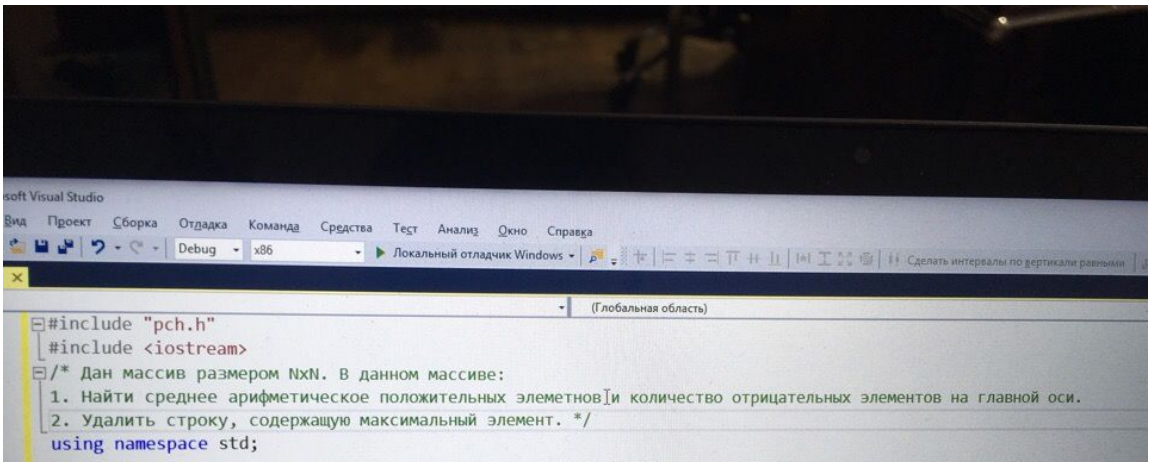
#undef идентификатор

Например: #undef MAX

Директивы могут использоваться также для условной компиляции и для изменения номеров строк и идентификатора файла.

Задачи с экза.

1)



2) Динамический массив. Вывести четные элементы массива. Те m[i] четный
Отрицательные элементы массива заменить на нули
Второе задание это элементы равные нулю заменить на полусумму соседних, кроме 1 и последнего элемента

3) Билет #20. 1) Динам.массив, вывести сумму его отрицательных элементов, ко всем четным числам массива прибавить 5. 2) Отсортировать так, чтобы сначала шли отриц.элементы, а потом положительные не меняя их взаиморасположения. 3) циклы фор и вайл

4) Мат NxN, найти кол-во ненулевых элементов матрицы, произведение положительных
Дан массив B.Если в строке матрицы все положительные, то массиву присваиваю 1, в противном - 0

5) У меня было задание с одномерным динамическим
Сначала найти мин и макс значения, потом вывести массив, в котором последние 4 числа заменяются на 9. И надо было найти наиболее повторяющееся число

6) Дана строка, слова разделены одним или несколькими пробелами, вывести слова, содержащие “www”. Вывести слова, у которых первый и последний символ одинаковые, а второй и третий разные. Связь указателей с массивами. Создание динамических массивов и правила работы с ними.

7) Билет 12
1. Дан динамический одномерный массив (число элементов ввести с клавиатуры). Подсчитать среднее арифметическое положительных элементов. Отрицательные элементы умножить на два. Вывести массив и результаты вычислений. Очистить кучу.

2. Элементам с чётными индексами(кроме последнего) присвоить значение полусуммы соседних элементов. Вывести массив.
3. Арифметические операции. Преобразование типов(явное и неявное)

8)
1. Вывести значение функции y=(e^x-e^(-x))/2 для икса от a=0.3 до b=0.9 с шагом h=0.07.
2. Посчитать рекуррентно сумму s=...(не помню, соре, мб в лабах есть, прочекай) при помощи функции s(x, N) для N=1200 слагаемых
3. Первый вопрос из списка теории

9)