

# Алгоритмизация и программирование

## Контрольные точки

### 3 модуль:

1. 6 лекций (проверочные работы или тесты)
2. 3 лабораторные работы
3. 10 семинаров (проверочные работы или тесты)
4. Контрольная работа (на семинаре)

### 4 модуль:

1. 5 лекций (проверочные работы)
2. 3 лабораторные работы
3. 10 семинаров (проверочные работы или тесты)
4. Контрольная работа (на семинаре)

Оценка за текущий контроль в 3 и 4 модуле учитывает результаты студента следующим образом.

Модуль 3.  $O_{\text{текущая } 1} = O_{\text{лекция}} + O_{\text{семинар}} + O_{\text{лаб. работа}} + O_{\text{ответы у доски}} + O_{\text{контр. работа}}$

Модуль 4.  $O_{\text{текущая } 2} = O_{\text{лекция}} + O_{\text{семинар}} + O_{\text{лаб. работа}} + O_{\text{ответы у доски}} + O_{\text{контр. работа}}$

Все оценки рассматриваются без округления.

Промежуточная оценка за 3 и 4 модуль вычисляется по формуле

$O_{\text{промежуточная 3 и 4}} = 0,4 * O_{\text{текущая } 3} + 0,4 * O_{\text{текущая } 4} + 0,2 * O_{\text{экзамен 4 модуль}}$

где  $O_{\text{текущая } 3}$ ,  $O_{\text{текущая } 4}$  — оценки текущего контроля 3 и 4 модуля, без округления.

Округление производится один раз, после вычисления промежуточной оценки, по правилам арифметики.

Экзаменационная оценка не является блокирующей. Промежуточная оценка за 3 и 4 модуль не может превышать 10 баллов, в случае превышения ставится промежуточная оценка 10 баллов.

Результирующая оценка за дисциплину вычисляется по формуле

$$O_{\text{результирующая}} = 0,4 * O_{\text{промежуточная 1 и 2}} + 0,6 * O_{\text{промежуточная 3 и 4}}$$

Округление производится по правилам арифметики. В диплом выставляется результирующая оценка.

Для вычисления текущей оценки по дисциплине используется следующая таблица (для групп БИВ).

	Работа на семинарском занятии	Работа на лекции	Выполнение лабораторного практикума	Контрольная работа
1 модуль	2	1	7 (3+4)	-
2 модуль	1	1	5(2+2+1)	3
3 модуль	1	1	5(2+1,5+1,5)	3
4 модуль	1	1	5(1,5+1,5+2)	3

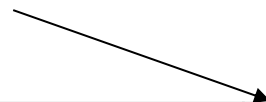
В скобках указано распределение баллов по лабораторным работам.

- Ни один из элементов текущего контроля не является блокирующим.
- На некоторых семинарах и лекциях проводится тест или проверочная работа. Каждый вид работы оценивается от 1 до 4 баллов. В итоговую оценку эти баллы входят с коэффициентом, получаемым делением числа занятий, на которых проводилось оценивание, на общее количество занятий.
- При пропуске лекции или семинарского занятия по любой причине студент не может решить дополнительное задание для компенсации баллов, которые он мог бы получить на этом занятии.
- Кроме того, преподаватель может оценивать дополнительными баллами ответ студента у доски (максимум 2 балла) и активное участие в решении задач семинаров (например, выявление и исправление неточностей и ошибок в алгоритмах и при кодировании программ, внесение усовершенствований в алгоритм и т.п.) (максимум по 0.2 балла за каждый ответ).

- Для каждой лабораторной работы устанавливается срок защиты отчета (в 3 модуле на 4, 8 и 12 занятия, считая обе подгруппы). При своевременной защите работа оценивается полученным баллом, при опоздании на 1 неделю балл снижается на 40%, при опоздании на 2 недели балл снижается на 60% от полученной оценки. При опоздании более чем на 2 недели работа не оценивается.
- В случае пропуска занятий по уважительной причине (обязательно предоставление справки) срок сдачи лабораторной работы может быть перенесен на соответствующее количество рабочих дней.
- В случае пропуска занятий по уважительной причине (обязательно предоставление справки) предоставляется дополнительное время для написания контрольной работы (единственная дата переписывания заранее сообщается через старост).
- Переписывание контрольной работы с целью повышения полученной оценки не допускается.

Для выбора набора заданий используйте формулы:

Пример



№ варианта	X – номер варианта	X=13
Задание 1	$(\text{Остаток от деления } x \text{ на } 7) + 1$	7
Задание 2	$(\text{Остаток от деления } x \text{ на } 9) + 1$	5
Задание 3	$(\text{Остаток от деления } x \text{ на } 10) + 1$	4

7, 9, 10 – количество вариантов заданий в л.р. 1.  
Номера вариантов указаны в журнале на страницах групп. Также список номеров вариантов есть в Smart lms

# Характеристики C++

- **C++ является ISO-стандартизированным языком программирования.**

В течение некоторого времени, C++ не имел официального стандарта, однако с 1998 года, C++ был стандартизирован комитетом ISO.

- **C++ компилируемый язык.**

C++ компилируется непосредственно в машинный код, что позволяет ему быть одним из самых быстрых в мире языков.

- **C++ является строго типизированным языком.**

C++ подразумевает, имеет большое количество возможностей использования типов.



- **С++ поддерживает статические и динамические типы данных.**

Таким образом, проверка типов данных может выполняться во время компиляции или во время выполнения. И это ещё раз доказывает его гибкость.

- **С++ поддерживает множество парадигм.**

С++ поддерживает процедурную, структурную и объектно-ориентированную парадигмы программирования, а также многие другие парадигмы.

- **C++ является портативным языком программирования.**

Это один из наиболее часто используемых языков в мире. Как открытый язык, C++ имеет широкий спектр компиляторов, которые работают на различных платформах. Код стандартной библиотеки C++ будет работать на многих платформах.

- **C++ является полностью совместимым с языком C**

В C++ можно использовать библиотеки языка C и они будут исправно работать.

# История создания языка программирования C++

- История создания
- Язык возник в начале 1980-х годов, когда сотрудник фирмы Bell Labs Бьёрн Страуструп придумал ряд усовершенствований к языку C под собственные нужды. В первую очередь в C были добавлены классы (с инкапсуляцией), наследование классов, строгая проверка типов, inline-функции и аргументы по умолчанию. Ранние версии языка, первоначально именовавшегося «C with classes» («Си с классами»), стали доступны с 1980 года.

При создании C++ Бьёрн Страуструп ставил следующие задачи.

- Получить универсальный язык со статическими типами данных, эффективностью и переносимостью языка C.
- Непосредственно и всесторонне поддерживать множество стилей программирования, в том числе процедурное программирование, абстракцию данных, объектно-ориентированное программирование и обобщённое программирование.
- Дать программисту свободу выбора, даже если это даст ему возможность выбирать неправильно.

- Максимально сохранить совместимость с C, тем самым делая возможным лёгкий переход от программирования на C.
- Избежать разночтений между C и C++: любая конструкция, допустимая в обоих языках, должна в каждом из них обозначать одно и то же и приводить к одному и тому же поведению программы.
- Избегать особенностей, которые зависят от платформы или не являются универсальными.
- «Не платить за то, что не используется» — никакое языковое средство не должно приводить к снижению производительности программ, не использующих его.

- Не требовать слишком усложнённой среды программирования.
- Несмотря на ряд известных недостатков языка С, Страуструп пошёл на его использование в качестве основы, так как «в С есть свои проблемы, но их имел бы и разработанный с нуля язык, а проблемы С нам известны». Кроме того, это позволило быстро получить прототип компилятора (cfront), который лишь выполнял трансляцию добавленных синтаксических элементов в оригинальный язык С.

# ПОТОКОВЫЙ ВВОД/ВЫВОД

```
#include <iostream> //библиотека потокового ввода/вывода
int a,b;
```

```
cout <<" input b,a:" <<
endl;
cin >> b >> a;
cout << "a=" << a
<< endl << "b=" << b;
```

```
input b,a:
6
8
a=8
b=6
```

```
printf("input b,a:\n");
scanf("%d%d", &b, &a);
printf("a=%d\nb=%d",a,
b);
```

<< – операция записи в поток

>> – операция чтения из потока

cin – стандартный поток для ввода с клавиатуры

cout – стандартный поток для вывода на экран

endl – функция, включающая в поток символ конца строки  
(аналог “\n”, но универсальный).

При форматном вводе (scanf) указываются адреса переменных. При потоковом вводе/выводе (cin/cout) и форматном выводе (printf) указываются имена переменных.

**Задача 1.** В заданном одномерном целочисленном массиве поменять местами минимум и максимум с использованием указателей. Используем потоковый ввод-вывод.

```
#include <iostream>
using namespace std;
int main ()
{int a[10],na,*ua,*umin,*umax,b;
/*na -длина,ua,umin,umax - указатели на текущий
  элемент, минимум и максимум*/
cout << "введите длину массива A:";cin>>na;
cout<<"введите массив A\n";
for (ua=a;ua<a+na;ua++)  cin>>*ua;
umin=umax=a; /*инициализация указателей - настроены на
  начало массива */
for(ua=a;ua<a+na;ua++) //поиск минимума и максимума
{
  if (*ua>=*umax) umax=ua ; //запоминаем адрес
  //максимума
  if (*ua<*umin) umin=ua ; //запоминаем адрес минимума
}
```



```
cout<<"значение min "<<*umin<<" адрес  
"<<umin<<endl;  
cout<<"значение max "<<*umax<<" адрес  
"<<umax<<endl;  
if(*umin==*umax)  
    cout<<" Одинаковые значения";  
else  
    {b=*umin;*umin=*umax; *umax=b;  
    cout<<"Массив A после перестановки "<<endl;  
    for (ua=a;ua<a+na;ua++)  
        cout<<*ua<<"    ";  
    cout<<"\n";  
}  
return 0;  
}
```

# Флаги форматирования в C++

Флаг	Назначение	Пример	Результат
<b>boolalpha</b>	Вывод логических величин в текстовом виде (true, false)	<pre>cout.setf(ios::boolalpha); bool log_false = 0, log_true = 1; cout &lt;&lt; log_false &lt;&lt; endl &lt;&lt; log_true &lt;&lt; endl;</pre>	false true
<b>oct</b>	Ввод/вывод величин в восьмеричной системе счисления (сначала снимаем флаг dec, затем устанавливаем флаг oct).	<pre>cout.unsetf(ios::dec); cout.setf(ios::oct); int value; cin &gt;&gt; value; cout &lt;&lt; value &lt;&lt; endl;</pre>	ввод:99 вывод:143
<b>dec</b>	Ввод/вывод величин в десятичной системе счисления (флаг установлен по умолчанию).	<pre>cout.setf(ios::dec); int value = 148; cout &lt;&lt; value &lt;&lt; endl;</pre>	148
<b>hex</b>	Ввод/вывод величин в шестнадцатеричной системе счисления (сначала снимаем флаг dec, затем устанавливаем флаг hex).	<pre>cout.unsetf(ios::dec); cout.setf(ios::hex); int value; cin &gt;&gt; value; cout &lt;&lt; value &lt;&lt; endl;</pre>	ввод:99 вывод:63

# Флаги форматирования в C++

<b>showbase</b>	Выводить индикатор основания системы счисления	<pre>cout.unsetf(ios::dec); cout.setf(ios::oct   ios::showbase); int value; cin &gt;&gt; value; cout &lt;&lt; value &lt;&lt; endl;</pre>	ввод:99 вывод:0143
<b>uppercase</b>	В шестнадцатеричной системе счисления использовать буквы верхнего регистра (по умолчанию установлены буквы нижнего регистра)	<pre>cout.unsetf(ios::dec); cout.setf(ios::hex   ios::uppercase); int value; cin &gt;&gt; value; cout &lt;&lt; value &lt;&lt; endl;</pre>	ввод:255 вывод:FF
<b>showpos</b>	Вывод знака плюс + для положительных чисел	<pre>cout.setf(ios::showpos); int value = 15; cout &lt;&lt; value &lt;&lt; endl;</pre>	+15
<b>scientific</b>	Вывод чисел с плавающей точкой в экспоненциальной форме	<pre>cout.setf(ios::scientific); double value = 1024.165; cout &lt;&lt; value &lt;&lt; endl;</pre>	1.024165e+003
<b>fixed</b>	Вывод чисел с плавающей точкой в фиксированной форме (по умолчанию)	<pre>double value = 1024.16; cout &lt;&lt; value &lt;&lt; endl;</pre>	1024.16

# Флаги форматирования в C++

<b>right</b>	Выравнивание по правой границе (по умолчанию). Сначала необходимо установить ширину поля (ширина поля должна быть заведомо большей чем, длина выводимой строки). При недостаточной ширине она игнорируется.	<pre>cout.width(5); cout &lt;&lt; "one" &lt;&lt; endl;</pre>	<code>__one</code>
<b>left</b>	Выравнивание по левой границе. Сначала необходимо установить ширину поля (ширина поля должна быть заведомо большей чем, длина выводимой строки).	<pre>cout.setf(ios::left); cout.width(5); cout &lt;&lt; "two" &lt;&lt; endl;</pre>	<code>two__</code>

Для использования манипуляторов форматирования в C++ необходимо подключить заголовочный файл **`#include <iomanip>`**

# Манипуляторы форматирования в C++

Манипулятор	Назначение	Пример	Результат
<b>endl</b>	Переход на новую строку при выводе	<code>cout &lt;&lt; "Hello, " &lt;&lt; endl &lt;&lt; "world";</code>	Hello, world
<b>boolalpha</b>	Вывод логических величин в текстовом виде (true, false)	<code>bool log_true = 1; cout &lt;&lt; boolalpha &lt;&lt; log_true &lt;&lt; endl;</code>	true
<b>nboolalpha</b>	Вывод логических величин в числовом виде (true, false)	<code>bool log_true = true; cout &lt;&lt; nboolalpha &lt;&lt; log_true &lt;&lt; endl;</code>	1

# Манипуляторы форматирования в C++

<b>oct</b>	Вывод величин в восьмеричной системе счисления	<pre>int value = 64; cout &lt;&lt; oct &lt;&lt; value &lt;&lt; endl;</pre>	100
<b>dec</b>	Вывод величин в десятичной системе счисления (по умолчанию)	<pre>int value = 64; cout &lt;&lt; dec &lt;&lt; value &lt;&lt; endl;</pre>	64
<b>hex</b>	Вывод величин в шестнадцатеричной системе счисления	<pre>int value = 64; cout &lt;&lt; hex &lt;&lt; value &lt;&lt; endl;</pre>	40
<b>showbase</b>	Выводить индикатор основания системы счисления	<pre>int value = 64; cout &lt;&lt; showbase &lt;&lt; hex &lt;&lt; value &lt;&lt; endl;</pre>	0x40
<b>noshowbase</b>	Не выводить индикатор основания системы счисления (по умолчанию).	<pre>int value = 64; cout &lt;&lt; noshowbase &lt;&lt; hex &lt;&lt; value &lt;&lt; endl;</pre>	40

# Манипуляторы форматирования в C++

<b>uppercase</b>	В шестнадцатеричной системе счисления использовать буквы верхнего регистра (по умолчанию установлены буквы нижнего регистра).	<pre>int value = 255; cout &lt;&lt; uppercase &lt;&lt; hex &lt;&lt; value &lt;&lt; endl;</pre>	FF
<b>nouppercase</b>	В шестнадцатеричной системе счисления использовать буквы нижнего регистра (по умолчанию).	<pre>int value = 255; cout &lt;&lt; nouppercase &lt;&lt; hex &lt;&lt; value &lt;&lt; endl;</pre>	ff
<b>showpos</b>	Вывод знака плюс + для положительных чисел	<pre>int value = 255; cout &lt;&lt; showpos &lt;&lt; value &lt;&lt; endl;</pre>	+255
<b>noshowpos</b>	Не выводить знак плюс + для положительных чисел (по умолчанию).	<pre>int value = 255; cout &lt;&lt; noshowpos &lt;&lt; value &lt;&lt; endl;</pre>	255
<b>scientific</b>	Вывод чисел с плавающей точкой в экспоненциальной форме	<pre>double value = 1024.165; cout &lt;&lt; scientific &lt;&lt; value &lt;&lt; endl;</pre>	1.024165e +003

# Манипуляторы форматирования в C++

<b>fixed</b>	Вывод чисел с плавающей точкой в фиксированной форме (по умолчанию).	<pre>double value = 1024.165; cout &lt;&lt; fixed &lt;&lt; value &lt;&lt; endl;</pre> 1024.165000
<b>setw(int number)</b>	Установить ширину поля, где <i>number</i> — количество позиций, символов (выравнивание по умолчанию по правой границе). Манипулятор с параметром.	<pre>cout &lt;&lt; setw(12) &lt;&lt; _strawberry "strawberry" &lt;&lt; endl;</pre>
<b>right</b>	Выравнивание по правой границе(по умолчанию). Сначала необходимо установить ширину поля (ширина поля должна быть заведомо большей чем, длина выводимой строки).	<pre>cout &lt;&lt; setw(5) &lt;&lt; right &lt;&lt; _one "one" &lt;&lt; endl;</pre>
<b>left</b>	Выравнивание по левой границе. Сначала необходимо установить ширину поля (ширина поля должна быть заведомо большей чем, длина выводимой строки).	<pre>cout &lt;&lt; setw(5) &lt;&lt; left &lt;&lt; two_ "two" &lt;&lt; endl;</pre>



# Манипуляторы форматирования в C++

## **setprecision (int count)**

Задаёт количество знаков после запятой, где *count* — количество знаков после десятичной точки

```
cout << fixed << setprecision(3) << (13.5 / 2) << endl;
```

6.750

## **setfill(int symbol)**

Установить символ заполнитель. Если ширина поля больше, чем выводимая величина, то свободные места поля будут наполняться символом *symbol* — это символ заполнитель

```
cout << setfill('0') << setw(4) << 15 << endl;
```

0015

```
#include <iostream>
```

```
#include <iomanip>
```

```
using namespace std;
```

```
void output_matr(int a[][3], int n, int m)
```

```
{int i, j; //использование манипуляторов при выводе  
//матрицы
```

```
    for(i=0; i<n; i++)
```

```
    {    for(j=0; j<m; j++)
```

```
        cout << setfill('_') << setw(4) << a[i][j]
```

```
<<"_|" ;
```

```
        cout << endl;
```

```
    }
```

```
}
```

```
int main()
```

```
{int a[2][3]={ {1,2,3}, {4,5,6}}, n=2, m=3;
```

```
    output_matr(a,n,m);
```

```
    return 0; }
```

```
1 | 2 | 3 |  
4 | 5 | 6 |
```

# С++ без классов, или "улучшенный" С .

Типы данных, операции и функции в С++

## 1. Новые заголовочные файлы

```
#include <iostream.h> /*заголовочный файл  
библиотеки ввода/вывода С++ в достандартном  
стиле */
```

```
#include <iostream> /*заголовочный файл  
библиотеки ввода/вывода С++ в стандартном  
стиле */
```

```
using namespace std; /*директива включения  
экспортируемых имен стандартной библиотеки в  
глобальное пространство имен */
```

```
using std::cout; /*объявление using  
включает только указанное имя using std::cin;  
using std::endl; */
```

```
#include <stdlib.h> /*заголовочный файл  
библиотеки языка C      в стандартном  
стиле */
```

```
#include "ftpl_swap.h" /*"свои"  
заголовочные файлы по-прежнему      имеют  
расширение .h */
```

```
double d = 12.345; /*имя d принадлежит  
глобальному пространству имен */
```

```
// Альтернативная библиотека
//ввода/вывода

int main() {
std::cout << "Hello, world!\n";
/*std::cout - выходной поток (объект
из пространства имен std)

:: - оператор доступа к контексту -
уточняет, что cout принадлежит
пространству имен std

<< - оператор "вставки в поток" */
    cout << "Hello, world!\n";
    /*здесь уточнение происходит
благодаря директиве using */
```

```
cout << "Hello, world!" << endl;  
// *манипулятор endl - конец строки  
(std::endl) операторы << часто  
применяют "цепочкой" */
```

Результат:

Hello, world!

Hello, world!

Hello, world!

—

Инструкции описания и исполнения могут чередоваться

```
cout << "Enter int and float values: ";  
int a; double b; /*инструкции описания могут  
располагаться в любом месте кода */  
cin >> a >> b;  
/*std::cin - входной поток (объект)  
оператор >> - "извлечение из потока"  
тоже часто применяют "цепочкой */  
cout << "a=" << a << ", b=" << b << endl;  
cout << "a == b is " << (a == b) << endl;
```

```
Enter int and float values: 5 5.25  
a=5, b=5.25  
a == b is 0
```

```
a=5, b=5.25;
cout << "Half of b:" << b/2 << endl
      << "Half of a:" << a/2 << endl;
cout << "Casting: "
      << (double)a/2 << '\t' /*(double) -
оператор явного приведения типа в стиле C */
/* функциональная форма оператора явного
приведения типа */
      << double(a)/2 << endl;
/*double() - функциональный стиль C++
оператора явного приведения типа */
```

```
Half of b:2.625 Half of a:2
Casting: 2.5 2.5
```



## Новое в описании констант и массивов

```
const int n = 10;  
int arr[n]; /*константные переменные можно  
использовать для указания размера массива*/
```

```
for(int i=0; i<n; ++i)  
    arr[i] = i;  
/*определять переменные можно даже в  
инструкциях for, if, while */
```

Новый тип - логический, со значениями true и false

```
bool t = false;  
t = a > b;  
cout << a << (t == true ? " > " : " <= ")  
<< b << endl;
```

5 <= 5.25
-----------

**Задача 1.** Дана последовательность целых чисел  $A[0:n-1]$ . Найти длину максимальной последовательности из нулей и начало этой последовательности. Используем указатели и потоковый ввод-вывод

Обозначения:

ntp – длина текущей последовательности из нулей

maxntp – максимальная длина последовательности

npmax – начало максимальной последовательности из нулей

n – номер текущего элемента

*//Алгоритм – вычислительная часть*

maxntp:=0; ntp:=0

цикл от n:=0 до n-1

если  $a[n]=0$  то ntp:=ntp+1

иначе

если ntp>maxntp то maxntp:=ntp; npmax:=n-ntp

все

ntp:=0

все

кц

если ntp>maxntp то maxntp:=ntp; npmax:=n-ntp;

все

```
#include <iostream>
int main()
{using namespace std;
const int lmax=100;
    int a[lmax], n, *ua, dtp, maxdp, npmax;
    do{
        cout<<"Введите 0<n<= "<<lmax; cin>> n;
        if (n<=0||n>lmax) cout<<"Введите n
повторно "<<endl;
        }while (n<=0||n>lmax);
    cout<<"Введите элементы массива\n";
    for (ua=a;ua< a+n; ua++) cin>>*ua;
```

```

maxdp=ntp=0;
for (ua=a; ua<a+n; ua++)
    if (*ua==0) ntp++;
    else
        {if (ntp>maxdp)
            maxdp=ntp, npmax=(ua-a)-ntp;
            ntp=0;
        }
if (ntp>maxdp)
    maxdp = ntp, npmax = (ua-a)-ntp;
if (!maxdp) cout<<"Нет нулей";
else
    cout<<"Maxdp = "<<maxdp<<" npmax =
"<<npmax<<endl;
return 0;}

```

При написании программ вы всегда должны учитывать, что пользователи могут вводить данные некорректно. Хорошо написанная программа либо грамотно обрабатывает эти случаи, либо вообще предотвратит их появление (если это возможно). Программа, которая хорошо обрабатывает случаи ошибок, называется **надежной**.

Когда пользователь вводит данные они помещаются в буфер внутри `std::cin`.

**Буфер** (также называемый буфером данных) – это часть памяти, отведенная для временного хранения данных. В этом случае буфер используется для хранения пользовательских входных данных, пока они ожидают извлечения в переменные (<<).

При использовании оператора извлечения происходит следующая процедура.

- Пользователя просят ввести данные для извлечения. Когда пользователь нажимает Enter, во входной буфер помещается символ '\n'.
- Оператор >> извлекает столько данных из входного буфера, сколько возможно, в переменную (игнорируя любые начальные пробельные символы, такие как пробелы, табуляции или '\n').
- Извлечение завершается успешно, если из входного буфера извлечен хотя бы один символ. Любые неизвлеченные входные данные остаются во входном буфере для дальнейшего извлечения.

Например,

```
using namespace std;
```

```
int n;
```

```
/*выделение памяти для переменной типа int*/
```

```
cin >> n;
```

Данные	Результат
ба	n=6, а остается в буфере для дальнейшей обработки
с	Извлечение не произошло, несоответствие типов

Как проверить правильность ввода данных?

Рассмотрим пример программы (калькулятор, без обработки ошибок).



```
#include <iostream>
using namespace std;
double getDouble()
{    cout << "Enter a double value: ";
    double x;
    cin >> x;
    return x;
}
char getOperator()
{    cout << "Enter one of the following: +,
-, *, or /: ";
    char op;
    cin >> op;
    return op;
}
```

```
void printResult(double x, char operation,
double y)
{
    switch (operation)
    {
        {case '+':cout << x << " + " << y << " is
" << x + y << '\n';          break;
        case '-':cout << x << " - " << y << " is
" << x - y << '\n';          break;
        case '*':cout << x << " * " << y << " is
" << x * y << '\n';          break;
        case '/':cout << x << " / " << y << " is
" << x / y << '\n';          break;
    }
}
```

```
int main()  
{  
    double x=getDouble();  
    char operation=getOperator();  
    double y=getDouble();  
  
    printResult(x, operation, y);  
  
    return 0;  
}
```

- Сначала мы просим пользователя ввести несколько чисел. Что, если он введет что-то, отличающееся от числа (например, 'a')? В этом случае извлечение не удастся.
- Во-вторых, мы просим пользователя ввести один из четырех возможных символов. Что, если он введет символ, отличный от ожидаемых? Мы сможем извлечь входные данные, но пока не обрабатываем то, что расположено после них.
- В-третьих, если мы попросим пользователя ввести символ, а он введет строку типа «\*hello". Хотя мы можем извлечь нужный нам символ '\*', в буфере останутся дополнительные входные данные, которые могут вызвать проблемы в будущем.

## Типы недопустимых входных данных

Можно разделить ошибки ввода текста на четыре типа:

- извлечение входных данных выполняется успешно, но входные данные не имеют смысла для программы (например, ввод 'k' в качестве математической операции);
- извлечение входных данных выполняется успешно, но пользователь вводит дополнительные данные (например, вводя "\* hello" в качестве математического оператора);
- ошибка извлечения входных данных (например, попытка ввести 'q' при запросе ввода числа);
- извлечение входных данных выполнено успешно, но пользователь выходит за пределы значения числа.

Чтобы сделать наши программы надежными, всякий раз при вводе данных мы должны определить, может ли произойти каждый из вышеперечисленных возможных вариантов, и если да, написать код для обработки этих случаев.

## Случай 1: извлечение успешно, но входные данные не имеют смысла

Рассмотрим следующий вариант выполнения приведенной выше программы:

```
Enter a double value: 6
Enter one of the following: +, -, *, or /: k
Enter a double value: 3

-----
Process exited after 12.78 seconds with return value 0
Для продолжения нажмите любую клавишу . . .
```

В этом случае мы попросили пользователя ввести один из четырех символов, но вместо этого он ввел 'k'. 'k' – допустимый символ, поэтому `std::cin` успешно извлекает его в переменную `op`, и она возвращается в `main`. Но наша программа не обрабатывает этот случай правильно (и, таким образом, ничего не выводит).

Решение таково: выполнить проверку ввода. Обычно она состоит из 3 шагов:

- убедитесь, что пользовательский ввод соответствует вашим ожиданиям;
- если да, верните значение вызывающей функции;
- если нет, сообщите пользователю, что произошла ошибка ввода, и попросите его повторить попытку.

Изменим код функции ввода операции с учетом необходимости проверки входных данных.

```
char getOperator()
{
    while (true) // Цикл бесконечный
    {
        cout << "Enter one of the following: +,
        -, *, or /: ";

        char operation;    cin >> operation;
        // Проверяем что данные верны
        switch (operation)
        {
            case '+': case '-': case '*': case '/':
                return operation;
        }
        // возвращаем символ вызывающей функции
        default: // сообщаем об ошибке
            cout << "Input is invalid.
            Please try again.\n";
        }
    }
}
```



После внесения изменений программа проверяет что символ операции введен верно. В противном случае происходит повторный ввод данных

```
Enter a double value: 2
Enter one of the following: +, -, *, or /: k
Input is invalid. Please try again.
Enter one of the following: +, -, *, or /: *
Enter a double value: 2
2 * 2 is 4
```

```
-----
Process exited after 15.1 seconds with return value 0
Для продолжения нажмите любую клавишу . . . _
```

## Случай 2: извлечение данных успешно, но с ошибочными сообщениями

Рассмотрим следующий вариант выполнения приведенной выше программы:

```
Enter a double value: 5*7
Enter one of the following: +, -, *, or /: Enter a double value: 5 * 7 is 35
-----
Process exited after 5.783 seconds with return value 0
Для продолжения нажмите любую клавишу . . . _
```

Программа выводит правильный ответ, но форматирование неверно.

Когда пользователь вводит "5\*7" в качестве вводных данных, эти данные попадают в буфер.

Затем оператор >> извлекает 5 в переменную x, оставляя в буфере "\*7\n".

Затем программа напечатает

"Enter one of the following: +, -, \*, or /:".

Однако когда был вызван оператор извлечения, он видит символы "\*7\n", ожидающие извлечения в буфере, поэтому он использует их вместо того, чтобы запрашивать у пользователя дополнительные данные.

Следовательно, он извлекает символ '\*', оставляя в буфере "7\n".

После запроса пользователя ввести другое значение `double`, из буфера извлекается 7 без ожидания ввода пользователя.

Поскольку у пользователя не было возможности ввести дополнительные данные и нажать Enter (добавляя символ новой строки), все запросы в выводе идут вместе в одной строке, даже если вывод правильный.

Было бы лучше, если бы любые введенные лишние символы просто игнорировались.

```
cin.ignore(100, '\n'); /* очищаем до 100  
символов из буфера или пока не будет удален  
символ '\n' */
```

```
#include <limits> // для std::numeric_limits
```

Чтобы игнорировать все символы до следующего символа '\n', мы можем передать

```
numeric_limits<streamsize>::max()
```

в `cin.ignore()`.

`numeric_limits<streamsize>::max()` возвращает наибольшее значение, которое может быть сохранено в переменной типа `streamsize`. Передача этого значения в `cin.ignore()` приводит к отключению проверки счетчика.

Чтобы игнорировать всё, вплоть до следующего символа '\n', мы вызываем

```
cin.ignore(numeric_limits<streamsize>::max(),  
'\n');
```

```
//Изменим функцию getDouble()
```

```
void ignoreLine()
{
    cin.ignore(numeric_limits<streamsize>::max(),
        '\n');
}

double getDouble()
{
    cout << "Enter a double value: ";
    double x;
    cin >> x;
    ignoreLine();
    return x;
}
```

После внесения изменений программа после ввода числа пропускает все символы до конца строки

```
Enter a double value: 5*7
Enter one of the following: +, -, *, or /: *
Enter a double value: 5
5 * 5 is 25

-----
Process exited after 11.74 seconds with return value 0
Для продолжения нажмите любую клавишу . . .
```

## Случай 3: сбой при извлечении

Рассмотрим следующий вариант ввода данных

```
Enter a double value: a_
```

После этого происходит бесконечный цикл, выводятся сообщения

```
Enter one of the following: +, -, *, or /:  
Input is invalid. Please try again.
```

Это очень похоже на случай ввода неверных символов, но немного отличается.

Когда пользователь вводит 'a', этот символ помещается в буфер. Затем оператор >> пытается извлечь 'a' в переменную x, которая имеет тип double. Поскольку 'a' нельзя преобразовать в double, оператор >> не может выполнить извлечение.



В этот момент происходят две вещи: 'а' остается в буфере, а `std::cin` переходит в «режим отказа».

После перехода в «режим отказа», будущие запросы на извлечение входных данных будут автоматически завершаться ошибкой. Мы можем определить, завершилось ли извлечение сбоем, и исправить это:

```
if (cin.fail()) /* предыдущее извлечение не
удалось? */
{
    // да, исправляем ошибку
    cin.clear(); /* возвращаем ввод в
"нормальный" режим работы */
    ignoreLine();
// и удаляем неверные входные данные
}
```

//Изменим функцию getDouble() повторно

```
double getDouble()
{
    while (true)
    {
        cout << "Enter a double value: ";
        double x;
        cin >> x;
        if (cin.fail())
        {
            cin.clear();
            ignoreLine();
        }
        else
        {
            ignoreLine();
            return x;
        }
    }
}
```

После внесения изменений программа повторно вводит данные после ввода буквы вместо числа

```
Enter a double value: a  
Enter a double value: 2  
Enter one of the following: +, -, *, or /: _
```

## Случай 4: извлечение успешно, но пользователь выходит за пределы значения числа

```
#include <iostream>

using namespace std;

int main()
{short int x, y; /* x ,y - в диапазоне -32768
до 32767*/
    cout << "Enter a number between -32768 and
32767: ";
    cin >> x;
    cout << "Enter another number between -32768
and 32767: ";    cin >> y;
    cout << "The sum is: " << x + y << '\n';
    return 0;
}
```

При вводе числа, выходящего за пределы заданного диапазона, получим следующий результат

```
Enter a number between -32768 and 32767: 40000
Enter another number between -32768 and 32767: The sum is: 32767
-----
Process exited after 5.625 seconds with return value 0
Для продолжения нажмите любую клавишу . . .
```

В приведенном выше случае `cin` немедленно переходит в «режим отказа», но также присваивает переменной ближайшее значение в диапазоне. Следовательно, `x` будет присвоено значение 32767. Дополнительные входные данные пропускаются, оставляя `y` с инициализированным значением 0. Мы можем обрабатывать этот вид ошибки так же, как и неудачное извлечение.

```
void printResult(double x, char operation,
double y)
{    switch (operation)
    {case '+':
        cout << x << " + " << y << " is " <<
x + y << '\n';    break;
    case '-':
        cout << x << " - " << y << " is " <<
x - y << '\n';    break;
    case '*':
        cout << x << " * " << y << " is " <<
x * y << '\n';    break;
    case '/':
        cout << x << " / " << y << " is " <<
x / y << '\n';    break;
```

```
default: /* Надежность означает также
обработку неожиданных параметров,
даже если getOperator() гарантирует, что op в
этой конкретной программе корректен */
    cerr << "Something went wrong:
printResult() got an invalid operator.\n";
    }
}
```

# Префиксная и постфиксная форма инкремента и декремента

В префиксной форме сначала операнд увеличивается или уменьшается на единицу, а затем выражение вычисляется как значение операнда.

Постфиксные операторы инкремента/декремента немного сложнее. Сначала создается копия операнда. Затем операнд (не копия) увеличивается или уменьшается на единицу. И, наконец, вычисляется копия (а не оригинал).



```
#include <iostream>

int main()
{ int x = 5 ;
    int y = x++; /* x увеличивается до 6, копия
исходного x вычисляется в значение 5, а 5
присваивается y*/
    std::cout <<" x= " << x <<endl<< " y= " << y;
    return 0; }
```

x=6

y=5

Сначала создается временная копия  $x$ , которая имеет то же начальное значение, что и  $x$  (5). Затем реальная переменная  $x$  увеличивается с 5 до 6. Затем возвращается копия  $x$  (которая всё еще имеет значение 5) и присваивается переменной  $y$ . Затем эта временная копия отбрасывается.

Следовательно,  $y$  заканчивается значением 5 (значение до инкремента), а  $x$  заканчивается значением 6 (значение после инкремента).

Обратите внимание, что постфиксная версия требует гораздо большего количества шагов и, следовательно, может быть не такой производительной, как префиксная версия.

```

#include <iostream>
using namespace std;
int main()
{int x= 5;
  int y= 5;
  cout << x << ' ' << y << '\n';
  cout << ++x << ' ' << --y << '\n';
  // префиксная форма
  cout << x << ' ' << y << '\n';
  cout << x++ << ' ' << y-- << '\n';
  // постфиксная версия
  cout << x << ' ' << y << '\n';
  return 0;
}

```

5 5

6 4

6 4

6 4

7 3

```
#include <iostream>
using namespace std;
int add(int x, int y)
{ return x + y; }
int main()
{ int x= 5;
  int value = add(x, ++x) ;
  /* это 5 + 6, или 6 + 6? Зависит от
  того, в каком порядке компилятор
  вычисляет аргументы, передаваемые
  функции*/
  cout << value;
  // значение может быть 11 или 12
  return 0;
}
```

C++ не определяет порядок, в котором вычисляются аргументы функции.

Если первым вычисляется левый аргумент, это становится вызовом `add(5, 6)`, что равно 11.

Если первым вычисляется правый аргумент, это становится вызовом `add(6, 6)`, что равно 12!

Обратите внимание, что это становится проблемой только потому, что один из аргументов функции `add()` имеет побочный эффект.

Этих проблем можно избежать, если гарантировать, что любая переменная, в которой возникает побочный эффект, используется в какой-либо заданной инструкции не более одного раза.

# Сравнение значений с плавающей ТОЧКОЙ

```
#include <iostream>
using namespace std;
int main()
{ double d1=100.0-99. 999999999999999998;
// должно быть 0.000000000000000002
  double d2=100.0-99. 999999999999999999;
// должно быть 0.000000000000000001
  if (d1 == d2)
    cout << "d1 == d2" << '\n';
  else
    if (d1 > d2) cout << "d1 > d2" << '\n';
    else
      if (d1 < d2) cout << "d1 < d2" << '\n';
  return 0; }
```

```
d1 == d2
```

```
-----  
Process exited after 1.273 seconds with return value 0  
Для продолжения нажмите любую клавишу . . .
```

Если требуется высокий уровень точности, сравнение значений с плавающей запятой с использованием любого из операторов отношения может быть опасным. Это связано с тем, что значения с плавающей запятой неточны, а небольшие ошибки округления в операндах с плавающей запятой могут привести к неожиданным результатам.

Поскольку даже самая маленькая ошибка округления приведет к тому, что два числа с плавающей запятой не будут равны, операторы `==` и `!=` имеют высокий риск возврата `false`, когда можно было бы ожидать `true`.

Как можно корректно сравнить два операнда с плавающей точкой, чтобы увидеть, равны ли они?

Самый распространенный метод обеспечения равенства с плавающей запятой включает использование функции, которая проверяет, почти ли равны два числа. Если они «достаточно близки», то мы называем их равными. Значение, используемое для обозначения «достаточно близко», традиционно называется эпсилон. Эпсилон обычно определяется как небольшое положительное число (например, 0,00000001, иногда пишется  $1e-8$ ).



Рассмотрим следующую программу.

```
#include <iostream>
#include <cmath> // для abs()
using namespace std;
// эpsilon - абсолютное значение
bool isAlmostEqual(double a, double b,
double epsilon)
{ // если расстояние между a и b меньше
  эpsilon, тогда a и b "достаточно близки"
  return abs(a - b) <= epsilon;
}
```

```
int main()  
{double a=0.001, b=0.002,  
epsilon=0.01;  
if(isAlmostEqual( a,  b,  epsilon))  
    cout<<"Are equal"<<endl;  
else    cout<<"Are not equal"<<endl;  
return 0;  
}
```

Are equal

-----  
Process exited after 1.107 seconds with return value 0  
Для продолжения нажмите любую клавишу . . . \_

Рассмотрим следующую программу.

```
#include <iostream>
#include <cmath> // для abs()
using namespace std;
// эpsilon - абсолютное значение
bool isAlmostEqual(double a, double b,
double epsilon)
{ // если расстояние между a и b меньше
  эpsilon, тогда a и b "достаточно близки"
  return abs(a - b) <= epsilon;
}
```

```
int main()  
{double a=0.001, b=0.002,  
epsilon=0.01;  
if(isAlmostEqual( a,  b,  epsilon))  
    cout<<"Are equal"<<endl;  
else    cout<<"Are not equal"<<endl;  
return 0;  
}
```

Are equal

-----  
Process exited after 1.107 seconds with return value 0  
Для продолжения нажмите любую клавишу . . . \_

Хотя эта функция работает, но она не очень хороша. Эпсилон 0.00001 подходит для входных значений около 1.0, но слишком велико для входных значений около 0.0000001 и слишком мало для входных значений, таких как 10000. Это означает, что каждый раз, когда мы вызываем эту функцию, мы должны выбирать эпсилон, подходящий для наших входных данных. Если мы знаем, что нам придется масштабировать эпсилон пропорционально нашим входным данным, мы могли бы изменить функцию так, чтобы она делала это за нас.

Дональд Кнут, в своей книге «Искусство программирования, том 2: Получисленные алгоритмы» предложил следующий метод.

```
#include <iostream>
#include <cmath>      // для abs
#include <algorithm>  // для max
using namespace std;

/* возвращаем истину, если разница
между a и b находится в пределах
эпсилон-процента от большего из a и b
*/

bool approximatelyEqual(double a,
double b, double epsilon)
{
    return (abs(a - b) <=
(max(abs(a), abs(b)) * epsilon));
}
```

```
int main()  
{double a=0.001, b=0.001,  
epsilon=0.01;  
if(approximatelyEqual( a,  b,  
epsilon))  
    cout<<"Are equal"<<endl;  
else    cout<<"Are not equal"<<endl;  
return 0;  
}
```

Are equal

-----  
Process exited after 1.107 seconds with return value 0  
Для продолжения нажмите любую клавишу . . . \_

В этом случае, вместо абсолютного значения,  $\epsilon$  теперь зависит от величины  $a$  или  $b$ .

Левая часть оператора  $\leq$  (т.е.  $\text{abs}(a - b)$ ) сообщает нам расстояние между  $a$  и  $b$  как положительное число.

В правой части оператора  $\leq$  нам нужно вычислить наибольшее значение «достаточно близко», которое мы готовы принять. Для этого алгоритм выбирает большее из  $a$  и  $b$  (в качестве приблизительного показателя общей величины чисел), а затем умножает его на  $\epsilon$ . В этой функции  $\epsilon$  представляет собой процент. Например, если мы хотим сказать «достаточно близко» означает, что  $a$  и  $b$  находятся в пределах 1% от большего из  $a$  и  $b$ , мы передаем  $\epsilon = 0.01$  ( $1\% = 1/100 = 0,01$ ).



Чтобы выполнить сравнение на неравенство (!=) вместо равенства, просто вызовите эту функцию и используйте оператор логического отрицания (!), чтобы инвертировать результат:

```
if (!approximatelyEqual(a, b, 0.001))  
cout << a << " is not equal to " << b  
<< "\n";
```

Обратите внимание, что хотя функция `approximatelyEqual()` в большинстве случаев будет работать, она не идеальна, особенно когда числа близки к нулю.

```
int main()
{ /* а действительно близко к 1.0, но
имеет ошибки округления, поэтому оно
немного меньше 1.0 */
double a = 0.1 + 0.1 + 0.1 + 0.1 + 0.1
+ 0.1 + 0.1 + 0.1 + 0.1 + 0.1 ;
// сначала сравним "почти 1.0" с 1.0
    cout << approximatelyEqual(a,
1.0, 1e-8) << '\n';
/* а теперь сравним a-1.0 (почти 0.0)
с 0.0 (точное 0.0) ñ 0.0 */
    cout << approximatelyEqual(a-1.0,
0.0, 1e-8) << '\n';
    return 0; }
```

## Получаем результат

```
1
0

-----
Process exited after 1.203 seconds with return value 0
Для продолжения нажмите любую клавишу . . . _
```

Второй вызов не сработал, как ожидалось. Наши расчеты неверны при приближении почти до нуля.

Один из способов избежать этого – использовать как абсолютный эпсилон (как мы делали в первом примере), так и относительный эпсилон (как мы делали в подходе Кнута).

```
/* возвращаем true, если разница  
между a и b меньше, чем absEpsilon,  
или в пределах процента relEpsilon от  
большого значения a и b */
```

```
bool approximatelyEqualAbsRel(double  
a, double b, double absEpsilon,  
double relEpsilon)
```

```
{ // Проверяем, действительно ли  
числа близки - необходимо при  
сравнении чисел, близких к нулю */
```

```
double diff{abs(a - b) };
```

```
if (diff <= absEpsilon) return true;
```

```
/* В противном случае возвращаемся к  
алгоритму Кнута */  
return (diff <= (max(abs(a), abs(b))  
* relEpsilon));  
}
```

Проверим работу полученной функции

```
int main() { /* а близко к 1.0, но
имеет ошибки округления */

double a= 0.1 + 0.1 + 0.1 + 0.1 + 0.1
+ 0.1 + 0.1 + 0.1 + 0.1 + 0.1;

// сравниваем "почти 1.0" с 1.0
cout << approximatelyEqual(a, 1.0,
1e-8) << '\n';

// сравниваем "почти 0.0" с 0.0 cout
<< approximatelyEqual(a-1.0, 0.0, 1e-
8) << '\n';

// сравниваем "почти 0.0" с 0.0
cout << approximatelyEqualAbsRel(a-
1.0, 0.0, 1e-12, 1e-8) << '\n';
return 0;}
```

## Получаем результат

```
1
0
1
-----
Process exited after 1.221 seconds with return value 0
Для продолжения нажмите любую клавишу . . .
```

При правильно подобранном `absEpsilon` функция `approximatelyEqualAbsRel()` правильно обрабатывает маленькие входные значения.

Для сравнения чисел с плавающей запятой не существует универсального алгоритма, подходящего для всех случаев. Тем не менее, функция `approximatelyEqualAbsRel()` достаточно хорошо подходит для обработки большинства случаев.

# Преобразование типов

## Неявное преобразование типа

Неявное преобразование типа (также называемое автоматическим преобразованием типа или принуждением, англоязычный термин – «coersion») выполняется компилятором автоматически, когда требуется один тип данных, но предоставляется другой тип. Подавляющее большинство преобразований типов в C++ являются неявными преобразованиями типов.



Неявное преобразование типа происходит в следующих случаях.

1. При присвоении или инициализации переменной значением другого типа данных:

```
double d= 3 ; /* значение int 3  
неявно преобразуется в тип double*/  
d = 6;        /* значение int 6  
неявно преобразуется в тип double*/
```

2. При использовании бинарного оператора с операндами разных типов:

```
double division= 4.0 / 3;  
/* значение int 3 неявно  
преобразуется в тип double*/
```

3. При использовании небулевого значения в инструкции if:

```
if (5) /* значение int 5 неявно  
преобразуется в тип bool*/  
{  
}
```

4. При передаче аргумента функции с типом, отличающимся от типа параметра функции:

```
void doSomething(long l)  
{  
}  
  
doSomething(3); /* значение int 3  
неявно преобразуется в тип long*/
```

# Стандартные преобразования

Стандартные преобразования можно разделить на четыре категории:

- числовые продвижения;
- числовые преобразования;
- арифметические преобразования;
- другие преобразования (включая различные преобразования указателей и ссылок).

Когда требуется преобразование типа, компилятор знает, есть ли стандартные преобразования, которые он может использовать. В процессе преобразования компилятор может применить ноль, одно или несколько стандартных преобразований.

# Числовое продвижение

Числовое продвижение (расширяющее преобразование типа) – это преобразование более узкого числового типа (например, `char`) в более широкий числовой тип (обычно `int` или `double`), который может быть эффективно обработан и с меньшей вероятностью приведет к переполнению.

Все числовые продвижения сохраняют значения, что означает, что все значения в исходном типе могут быть представлены без потери данных или точности в новом типе. Поскольку такие продвижения безопасны, компилятор будет свободно использовать числовое продвижение по мере необходимости и при этом не будет выдавать предупреждение.

## Числовое продвижение снижает избыточность

Рассмотрим случай, когда нужно написать функцию для печати значения типа `int`.

```
#include <iostream>
using namespace std;
void printInt(int x)
{cout << x; }
```

А если мы захотим также иметь возможность печатать значение типа `short` или типа `char`? Если бы преобразования типов не существовало, нам пришлось бы написать еще одну функцию `print` для `short` и одну для `char`. И еще версию для `unsigned char`, `signed char`, `unsigned short`...

# Категории числового продвижения

Числовые правила продвижения делятся на две подкатегории: целочисленное продвижение и продвижение типов с плавающей запятой.

## **Продвижение типов с плавающей запятой**

Используя правила продвижения типов с плавающей запятой, значение типа `float` может быть преобразовано в значение типа `double`.

Это означает, что мы можем написать функцию, которая принимает значение типа `double`, а затем вызывать ее со значением типа `double` или `float`.

```
#include <iostream>
using namespace std;
void printDouble(double d)
{cout << d<<endl; }

int main()
{ printDouble(5.0) ;
// преобразование не требуется
printDouble(4.0f) ;
/* числовое продвижение float в
double */
return 0;
}
```

```
int main()  
{ printInt(2) ; //без преобразования  
  short s= 3 ;  
  printInt(s) ;  
  // числовое продвижение short в int  
  printInt('a') ;  
  // числовое продвижение char в int  
  printInt(true) ;  
  // числовое продвижение bool в int  
  return 0 ;  
}
```



# Целочисленные продвижения

Используя правила целочисленного продвижения, можно сделать следующие преобразования:

- `signed char` или `signed short` можно преобразовать в `int`;
- `unsigned char` и `unsigned short` могут быть преобразованы в `int`, если `int` может содержать весь диапазон типа, или в `unsigned int` в противном случае;
- `char` может быть преобразован в `int` (по умолчанию, если `char` со знаком) или `unsigned int` (по умолчанию, если `char` без знака);
- `bool` можно преобразовать в `int`, при этом `false` становится 0, а `true` становится 1.

Здесь стоит отметить две вещи.

Во-первых, в некоторых системах некоторые из целочисленных типов могут быть преобразованы в `unsigned int`, а не в `int`.

Во-вторых, некоторые более узкие беззнаковые типы (такие как `unsigned char`) будут преобразованы в более крупные типы со знаком (например, `int`).

Таким образом, хотя целочисленное продвижение способствует сохранению значения, оно не обязательно сохраняет отсутствие знака.

Некоторые преобразования типов с сохранением значений (например, `int` в `long` или `int` в `double`) в C++ не считаются числовыми продвижениями.

# Числовые преобразования

Существует пять типов числовых преобразований.

1. Преобразование одного целочисленного типа в любой другой целочисленный тип (за исключением целочисленных продвижений):

```
short s = 3; // конвертируем int в short  
long l = 3;  // конвертируем int в long  
char ch = s; /* конвертируем short в  
char*/
```

2. Преобразование типа с плавающей точкой в любой другой тип с плавающей точкой (за исключением продвижений с плавающей точкой):

```
float f = 3.0; /* конвертируем double в  
float*/
```

```
long double ld = 3.0; /* конвертируем  
double в long double*/
```

3. Преобразование типа с плавающей точкой в любой целочисленный тип:

```
int i = 3.5; // конвертируем double в int
```

4. Преобразование целочисленного типа в любой тип с плавающей запятой:

```
double d= 3; // конвертируем int в double
```

5. Преобразование целочисленного типа или типа с плавающей запятой в bool:

```
bool b1 = 3; // конвертируем int в bool
```

```
bool b2 = 3.0; /* конвертируем double в  
bool*/
```

# Сужающие преобразования

В отличие от числового продвижения (которое всегда безопасно), числовое преобразование может (или не может) привести к потере данных или точности.

Некоторые числовые преобразования всегда безопасны (например, `int` в `long` или `int` в `double`). Другие числовые преобразования, такие как `double` в `int`, могут привести к потере данных (в зависимости от конкретного преобразуемого значения и/или диапазона базовых типов):

```
int i1 = 3.5; /* 0.5 отбрасывается, что  
приводит к потере данных*/
```

```
int i2 = 3.0; /* будет преобразовано в  
значение 3, поэтому данные не будут  
потеряны*/
```

В C++ сужающее преобразование – это числовое преобразование, которое может привести к потере данных. К таким сужающим преобразованиям относятся:

- из типа с плавающей точкой в целочисленный тип;
- из более широкого типа с плавающей точкой в более узкий тип с плавающей точкой, если преобразовываемое значение не `constexpr` и не находится в диапазоне целевого типа (даже если оно не может быть представлено точно); (`constexpr` — спецификатор типа, введённый в стандарт программирования языка C++11 для обозначения константных выражений, которые могут быть вычислены во время компиляции кода.)

- из более широкого целочисленного типа в более узкий целочисленный тип, если преобразовываемое значение не `constexpr` и после целочисленного продвижения не будет соответствовать целевому типу;

Компилятор обычно выдает предупреждение (или ошибку), когда он определяет, что требуется неявное сужающее преобразование.

Следует избегать сужающих преобразований, но есть случаи, когда вам может потребоваться это сделать. В таких случаях вы должны сделать неявное сужающее преобразование явным, используя `static_cast`.

```
void someFcn(int i) { }

int main()
{ double d= 5.0 ; /* плохо: сгенерирует
предупреждение компилятора о сужающем
преобразовании */
someFcn(d) ;

/*хорошо:мы явно сообщаем компилятору,
что ожидается сужающее преобразование,
предупреждения не генерируются */
someFcn(static_cast<int>(d)) ;return 0 ; }
```



# Еще о числовых преобразованиях

Во всех случаях преобразование значения в тип, диапазон которого не поддерживает это значение, приведет к неожиданным результатам. Например:

```
int main()
{
    int i = 30000;

    char c = i; /* char имеет диапазон
от -128 до 127 */

    cout << static_cast<int>(c) ;
    return 0;
}
```

В этом примере мы присвоили большое целое число переменной с типом char (диапазон значений от -128 до 127). Это вызывает переполнение char и приводит к неожиданному результату: 48

Преобразование большего целочисленного типа или типа с плавающей запятой в меньший тип из того же семейства обычно будет работать, пока значение попадает в диапазон меньшего типа. Например:

```
int i= 2;  
short s=i;//конвертируем из int в short  
cout << s << '\n';  
double d= 0.1234;  
float f = d;  
cout << f << '\n';
```

Это дает ожидаемый результат:

2

0.1234

В случае значений с плавающей точкой может произойти некоторое округление из-за потери точности в меньшем типе. Например:

```
/* значение double 0.123456789 имеет 9  
значащих цифр, но float может  
поддерживать только около 7*/  
float f = 0.123456789;  
/* setprecision определена в заголовке  
iomanip */  
cout << setprecision(9) << f << '\n';
```

В этом случае мы видим потерю точности, потому что float не может содержать такой же точности, как double:

0.123456791

Преобразование из целого числа в число с плавающей запятой обычно работает до тех пор, пока значение попадает в диапазон типа с плавающей запятой.

Например:

```
int i= 10 ;  
float f = i;  
cout << f;
```

Это дает ожидаемый результат:

10.0

Преобразование из числа с плавающей запятой в целое число работает до тех пор, пока значение попадает в диапазон целого числа, но любые дробные части теряются.

Например:

```
int i = 3.5;
```

```
cout << i << '\n';
```

В этом примере дробная часть (.5) теряется, в итоге остается следующий результат:

3

# Преобразования при вычислении арифметических выражений

Что происходит, когда операнды бинарного оператора имеют разные типы?

```
float z=2+2.5;  
cout<<"z="<<z;
```

В C++ некоторые операторы требуют, чтобы их операнды были одного типа. Если один из этих операторов вызывается с операндами разных типов, один или оба операнда будут неявно преобразованы в соответствующие типы с использованием набора правил, называемых **обычными арифметическими преобразованиями**.

# Операторы, которым требуются операнды одного типа

- бинарные арифметические операторы: +, -, \*, /
- бинарные операторы отношения: <, >, <=, >=, ==, !=
- бинарные побитовые арифметические операторы: &, ^, |
- условный оператор ?: (исключая условие, которое, как ожидается, будет иметь тип bool).

# Правила обычного арифметического преобразования

Существует список типов по приоритету:

1. long double (высший приоритет)
2. double
3. float
4. unsigned long long
5. long long
6. unsigned long
7. long
8. unsigned int
9. int (низший приоритет)



Здесь есть только два правила:

- если тип любого из операндов находится в списке приоритетов, операнд с более низким приоритетом преобразуется в тип операнда с более высоким приоритетом;
- в противном случае (ни один из типов операндов не указан в списке) над обоими операндами выполняется числовое продвижение .

Например,

```
int i= 2 ;    double d= 3.5;  
cout << "i+d =" << ' ' << i + d << '\n';
```

5.5

Результат типа double, т.к. int i имеет более низкий приоритет.

```
short a= 4 ; short b= 5 ;  
cout << "a+b =" << ' ' << a + b << '\n';
```

9

Поскольку ни один из операндов не входит в список приоритетов, оба операнда проходят целочисленное продвижение до типа int.

Результатом сложения двух чисел int является int

## Проблемы со знаком / без знака

Эта иерархия приоритетов может вызвать некоторые проблемы при смешивании значений со знаком и без знака. Например:

```
cout << "5u - 10 =" << ' ' << 5u - 10 << '\n';
```

5u-10 = 42949672913.5

Поскольку операнд `unsigned int` имеет более высокий приоритет, операнд `int` преобразуется в `unsigned int`.

А поскольку значение `-5` выходит за пределы диапазона `unsigned int`, мы получаем результат, которого не ожидаем.

```
cout << " (-3 < 5u) =" << ' ' << (-3 < 5u) <<
'\n' ;
```

$$(-3 < 5u) = 0$$

Хотя нам ясно, что 5 больше, чем -3, при вычислении этого выражения -3 преобразуется в большое число `unsigned int`, которое больше 5. Таким образом, приведенный выше код выводит `false`, а не ожидаемый результат, равный `true`.

Это одна из основных причин избегать целочисленных типов без знака — когда в арифметических выражениях вы смешиваете их с целочисленными типами со знаком, вы рискуете получить неожиданные результаты. А компилятор, вероятно, даже не выдаст предупреждения.

# Приведение типа

```
double d = 10/4; /* выполняет целочисленное  
деление, инициализируя d значением 2.0*/  
cout << "d =" << ' ' << d << '\n';
```

Поскольку 10 и 4 принадлежат целочисленному типу `int`, целочисленного продвижения не происходит. Выполняется целочисленное деление  $10/4$ , в результате получается значение 2, которое затем неявно преобразуется в 2.0 и присваивается переменной `d`.

```
double d = 10.0 / 4.0; /* выполняет деление  
с плавающей точкой, инициализируя d  
значением 2.5 */
```

В случае, когда вы используете литеральные значения (например, 10 или 4), замена одного или обоих целочисленных литеральных значений на литеральное значение с плавающей точкой (10.0 или 4.0) приведет к преобразованию обоих операндов в значения с плавающей точкой, и деление будет выполнено с использованием математики с плавающей точкой(и, таким образом, сохранится дробная часть).

**Литерал** — запись в исходном коде компьютерной программы, представляющая собой фиксированное значение. Литералами также называют представление значения некоторого типа данных.

C++ поставляется с рядом различных операторов приведения типов (чаще называемых приведениями или англоязычный термин «cast»), которые могут использоваться программистом для запроса компилятора на выполнение преобразования типа.

Поскольку приведение типов является явным запросом программиста, эту форму преобразования типа часто называют **явным преобразованием типа** (в отличие от **неявного преобразования типа**, когда компилятор выполняет преобразование типа автоматически).

# Приведение типа в стиле C и статическое приведение

## Приведение типа в стиле C

В стандартном программировании на C приведение типов выполняется с помощью оператора `()`, при этом имя типа, в который необходимо преобразовать значение, помещается в круглые скобки. Например:



```
int x =10 ;int y = 4 ;
```

```
    /* преобразует x в double, поэтому  
    получаем деление с плавающей запятой */
```

```
double d =(double)x / y ;
```

```
cout << "d =" << ' ' << d << '\n';
```

d=2.5
-------

В приведенном выше коде мы используем приведение в стиле C для типа с плавающей запятой, чтобы указать компилятору, преобразовать x в double. Поскольку левый операнд у operator/ теперь вычисляется как значение с плавающей запятой, правый оператор также будет преобразован в значение с плавающей запятой, и деление будет выполняться с использованием деления с плавающей запятой вместо целочисленного деления!

C++ также позволяет вам использовать приведение в стиле C с синтаксисом, более похожим на вызов функций:

```
/* преобразует x в double, поэтому получаем  
деление с плавающей точкой*/
```

```
double d = double(x) / y ;
```

Это работает идентично предыдущему примеру, но тут преимущество в том, что преобразуемое значение заключено в скобки (что упрощает определение того, что конвертируется).

Хотя приведение в стиле C выглядит как единое преобразование, на самом деле оно может выполнять множество различных преобразований в зависимости от контекста.

В результате приведение типов в стиле C подвержено риску непреднамеренного неправильного использования и не приводит к ожидаемому поведению, чего легко избежать, если вместо этого использовать приведение типов согласно C++.

# Приведение типа в стиле C++

В C++ появился оператор приведения типов `static_cast`, который можно использовать для преобразования значения одного типа в значение другого типа.

Например, `static_cast` можно использовать для преобразования `char` в `int`, чтобы `std::cout` печатал его как целое число, а не как символ:

```
char c = 'a' ;  
    cout << c << ' ' << static_cast<int>(c)  
<< '\n' ; // печатает 97
```

Оператор `static_cast` принимает в качестве входного одно значение и выводит это значение, преобразованное в тип, указанный в угловых скобках. `static_cast` лучше всего использовать для преобразования одного базового типа в другой.

```
int x = 10;  
int y = 4;  
// преобразовываем x в double, и поэтому  
// получаем деление с плавающей точкой*/  
double d = static_cast<double>(x) / y;  
cout << d; // печатает 2.5
```

Основное преимущество `static_cast` заключается в том, что он обеспечивает проверку типа во время компиляции, что затрудняет случайную ошибку. `static_cast` также (намеренно) менее мощный, чем приведение типов в стиле C, поэтому вы не можете случайно удалить `const` или выполнить другие вещи, которые вы, возможно, не собирались делать.

Используйте `static_cast`, когда вам нужно преобразовать значение из одного типа в другой.

## Использование приведения типов, чтобы сделать неявное преобразование типов явным

Компиляторы выдают предупреждение при небезопасном (сужающем) неявном преобразовании типа. Например:

```
int i = 48;
```

```
char ch = i; //неявное сужающее преобразование
```

Преобразование int (4 байта) в char (1 байт) потенциально небезопасно (поскольку компилятор не может определить, будет ли целое число выходить за пределы диапазона char или нет), поэтому компилятор обычно выводит предупреждение.

Чтобы обойти это, мы можем использовать статическое приведение для явного преобразования нашего числа `int` в `char`:

```
int i = 48;  
/* явное преобразование из int в char, и  
далее этот char присваивается переменной ch */  
char ch = static_cast<char>(i);
```

Мы явно сообщаем компилятору, что это преобразование преднамеренное, и принимаем на себя ответственность за последствия (например, за превышение диапазона `char`, если оно произойдет). Поскольку выходное значение этого `static_cast` имеет тип `char`, присвоение переменной `ch` не генерирует несоответствия типов и, следовательно, нет никаких предупреждений или ошибок.



В следующей программе компилятор обычно будет предупреждать о том, что преобразование `double` в `int` может привести к потере данных:

```
int i = 100 ;  
i = i / 2.5;
```

Чтобы сообщить компилятору, что мы преднамеренно хотим это сделать, можно написать так:

```
int i = 100 ;  
i = static_cast<int>(i / 2.5) ;
```

# Ключевое слово typedef

В C++ typedef (сокращенно от «type definition», «определение типа») – это ключевое слово, которое создает псевдоним для существующего типа данных. Для этого используем ключевое слово typedef, за которым следует существующий тип данных для псевдонима, затем имя для псевдонима. Например:

```
typedef double distance_t; /* определяем  
distance_t как псевдоним для типа double*/
```

По соглашению имена typedef объявляются с использованием суффикса "\_t". Это указывает, что идентификатор представляет собой тип, а не переменную или функцию, и помогает предотвратить конфликты имен с другими типами идентификаторов.

# Преобразование типов и перегрузка функций

Рассмотрим следующую функцию:

```
int add(int x, int y)
{    return x + y;}
```

Эта простая функция складывает два числа `int` и возвращает результат `int`. Однако что, если нам нужно сложить два числа с плавающей ? Эта функция `add()` не подходит, поскольку любые параметры с плавающей будут преобразованы в целые числа, что приведет к потере дробных частей аргументов с плавающей .

Один из способов обойти эту проблему – определить несколько функций с немного разными именами:

```
int addInteger(int x, int y)
{
    return x + y;
}
```

```
double addDouble(double x, double y)
{
    return x + y;
}
```

Однако для достижения наилучшего эффекта это требует, чтобы вы определили единый стандарт именования для аналогичных функций, которые имеют параметры разных типов, запомнили названия всех различных вариантов функций и вызывали правильную из них.

А что тогда произойдет, когда мы захотим иметь аналогичную функцию, которая складывает три числа `int` вместо двух? Управление уникальными именами для каждой функции быстро становится утомительным.

В C++ есть решение для таких случаев. Перегрузка функций позволяет нам создавать несколько функций с одним и тем же именем, при условии, что все функции с одинаковыми именами имеют разные параметры (или функции могут различаться иным образом). Все функции, имеющие одинаковые имена (в одной и той же области видимости), называются перегруженными функциями (иногда для краткости называемыми **перегрузками**).

Чтобы перегрузить нашу функцию `add()`, мы можем просто определить другую функцию `add()`, которая принимает параметры `double`:

```
double add(double x, double y)
{
    return x + y;
}
```

Теперь у нас есть две версии `add()` в одной области видимости:

```
int add(int x, int y)
// целочисленная версия
{
    return x + y;
}
double add(double x, double y)
// версия с плавающей
{
    return x + y;
}
```

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

Показанный выше код будет компилироваться. Хотя вы можете ожидать, что эти функции приведут к конфликту имен, здесь дело обстоит не так.

Поскольку типы параметров этих функций различаются, компилятор может различать эти функции и рассматривать их как отдельные функции, которые просто имеют общее имя.

# Введение в разрешение перегрузки

Кроме того, когда выполняется вызов для функции, которая была перегружена, компилятор на основе аргументов, используемых в вызове функции, попытается сопоставить этот вызов функции с соответствующей перегрузкой. Это называется **разрешением перегрузки**.

Вот пример, демонстрирующий это:



```
#include <iostream>
int add(int x, int y)
{    return x + y;}
double add(double x, double y)
{    return x + y;}
int main()
{    std::cout << add(1, 2);
//ВЫЗЫВАЕТ add(int, int)
    std::cout << '\n';
    std::cout << add(1.2, 3.4);
// ВЫЗЫВАЕТ add(double, double)
    return 0;
}
```

Показанная выше программа компилируется и дает следующий результат:

3
4.6

Когда мы предоставляем целочисленные аргументы в вызове `add(1, 2)`, компилятор определит, что мы пытаемся вызвать `add(int, int)`. А когда мы предоставляем аргументы с плавающей в вызове `add(1.2, 3.4)`, компилятор определит, что мы пытаемся вызвать `add(double, double)`.

## Выполнение компиляции

Чтобы программа могла компилировать перегруженные функции, должны выполняться два условия:

- Каждую перегруженную функцию нужно отличать от других.
- Каждый вызов перегруженной функции должен решаться в перегруженную функцию.

Если перегруженная функция не различается или вызов перегруженной функции не может быть преобразован в перегруженную функцию, результатом будет ошибка компиляции.

# Как различаются перегруженные функции

Самый простой способ дифференцировать перегрузку функций – убедиться, что каждая перегруженная функция имеет отличающийся набор (количество и/или тип) параметров.

Свойство функции	Используется для перегрузки	Примечания
Количество параметров	Да	
Тип параметров	Да	Не включает себя использование typedef, псевдонимы типов и квалификатор const для значений параметров. Включает в себя многоточия.
Тип возвращаемого значения	Нет	

# Как различаются перегруженные функции

Перегрузка по количеству параметров

Перегруженные функции различаются до тех пор, пока каждая перегруженная функция имеет разное количество параметров. Например:

```
int add(int x, int y)
{
    return x + y;
}
int add(int x, int y, int z)
{
    return x + y + z;
}
```

Компилятор может легко сказать, что вызов функции с двумя параметрами `int` должен идти на `add(int, int)`, а вызов функции с тремя параметрами `int` должен идти на `add(int, int, int)`.

## Перегрузка по типу параметров

Функции также можно различать, если различаются наборы типов параметров каждой перегруженной функции. Например, различаются все следующие перегрузки:

```
int add(int x, int y);  
// целочисленная версия  
double add(double x, double y);  
// версия с плавающей  
double add(int x, double y);  
// смешанная версия  
double add(double x, int y);  
// смешанная версия
```

Поскольку псевдонимы типов (или определения typedef) не являются отдельными типами, перегруженные функции, использующие псевдонимы типов, не отличаются от перегрузок, использующих исходные типы. Например, все следующие перегрузки не различаются (и приведут к ошибке компиляции):

```
typedef int height_t; // typedef
using age_t = int;    // псевдоним типа
void print(int value);
void print(age_t value);
// не отличается от print(int)
void print(height_t value);
// не отличается от print(int)
```

Для параметров, передаваемых по значению, квалификатор `const` также не учитывается. Поэтому следующие функции не считаются разными:

```
void print(int) ;
```

```
void print(const int) ;
```

```
// не отличается от print(int)
```

Тип возвращаемого значения функции не учитывается в уникальности

Тип возвращаемого значения функции не учитывается при различении перегруженных функций. Рассмотрим случай, когда вы хотите написать функцию, возвращающую случайное число, но вам нужна одна версия, которая вернет `int`, и другая версия, которая вернет `double`.



У вас может возникнуть соблазн сделать так:

```
int getRandomValue();  
double getRandomValue();
```

В Visual Studio 2019 это приводит к следующей ошибке компилятора:

```
error C2556: 'double getRandomValue(void)':  
overloaded function differs only by return  
type from 'int getRandomValue(void)'
```

Эту ошибку можно понять. Если бы вы были компилятором и увидели следующую инструкцию:

```
getRandomValue();
```

Какую из двух перегруженных функций вы бы вызвали? Не понятно.

Лучший способ решить эту проблему – дать функциям разные имена:

```
int getRandomInt();
```

```
double getRandomDouble();
```

Альтернативный способ – сделать так, чтобы функции возвращали void, а возвращаемое значение передавалось обратно вызывающему в качестве выходного параметра

```
void getRandomValue(int &out);
```

```
void getRandomValue(double &out);
```

Поскольку эти функции имеют разные параметры, они считаются уникальными.

Однако у этого есть свои недостатки. Во-первых, синтаксис неудобен, и вы не можете направить вывод этой функции непосредственно на ввод другой. Рассмотрим:

```
// метод 1: getRandomInt() возвращает int
printValue(getRandomInt()); // легко

/* метод 2: getRandomValue() имеет выходной
параметр int */
int temp;

// теперь нам нужна временная переменная
getRandomValue(temp);

// чтобы вызвать здесь getRandomValue(int)
printValue(temp); // это должна быть
отдельная строка, поскольку getRandomValue()
возвращает void*/
```

Кроме того, тип переданного аргумента должен точно соответствовать типу параметра. По этим причинам мы не рекомендуем этот метод.

Когда компилятор компилирует функцию, он выполняет изменение имени, что означает, что скомпилированное имя функции изменяется на основе различных критериев, таких как **количество** и **тип параметров**, поэтому компоновщик получает для работы **уникальные имена**.

Например, функция с прототипом `int fcn()` может компилироваться с именем `__fcn_v`, тогда как `int fcn(int)` может компилироваться с именем `__fcn_i`. Таким образом, хотя в исходном коде две перегруженные функции имеют одинаковые имена, в скомпилированном коде имена на самом деле уникальны.

**Стандарта** того, как следует изменять имена, не существует; поэтому разные компиляторы будут создавать разные измененные имена.

# Разрешение перегрузки функций и неоднозначные совпадения

В случае **неперегруженных функций** (функций с уникальными именами) существует только одна функция, которая потенциально может соответствовать вызову функции. Эта функция либо соответствует (или может быть выполнено соответствие после применения преобразования типов), либо нет (и возникает ошибка компиляции). С **перегруженными функциями** может быть много функций, которые могут соответствовать вызову. Вызов функции может разрешить только одну из них, компилятор должен определить, какая перегруженная функция лучше всего подходит.

Процесс сопоставления вызовов функций с конкретной перегруженной функцией называется **разрешением перегрузки**.

Когда тип аргументов функции и тип параметров функции точно совпадают, это (обычно) просто:

```
#include <iostream>
void print(int x) {      std::cout << x; }
void print(double d) {   std::cout << d; }
int main()
{
    print(5);
    // 5 - это int, поэтому это print(int)
    print(6.7);
    // 6.7 - это double, поэтому это print(double)
    return 0; }
```

Но что происходит в случаях, когда типы аргументов в вызове функции не совсем соответствуют типам параметров ни в одной из перегруженных функций? Например:

```
#include <iostream>

void print(int x) {          std::cout << x;}
void print(double d) {      std::cout << d;}

int main()
{
    print('a');
    // char не совпадает с int или double
    print(51);
    // long не совпадает с int или double
    return 0;
}
```

Тот факт, что здесь нет точного совпадения, не означает, что соответствие не может быть найдено — в конце концов, тип `char` или `long` можно неявно преобразовать в `int` или `double`. Но какое преобразование лучше всего выполнить в каждом конкретном случае?

Рассмотрим, как компилятор сопоставляет заданный вызов функции с конкретной перегруженной функцией.



## **Разрешение вызовов перегруженных функций**

Когда выполняется вызов перегруженной функции, компилятор выполняет последовательность правил, чтобы определить, какая из перегруженных функций (если она есть) лучше всего подходит.

На каждом шаге в вызове функции компилятор применяет к аргументу(ам) несколько различных преобразований типов. Для каждого примененного преобразования компилятор проверяет, соответствует ли какая-либо из перегруженных функций. После того, как все преобразования различных типов были применены и проверены на совпадения, этап завершен.

Результатом будет один из трех возможных исходов:

- **Подходящих функций не найдено.** Компилятор переходит к следующему шагу в последовательности.
- **Обнаружена единственная соответствующая функция.** Эта функция считается наиболее подходящей. Теперь процесс сопоставления завершен, и последующие шаги не выполняются.
- **Найдено более одной соответствующей функции.** Компилятор выдаст ошибку компиляции о неоднозначном совпадении. Об этом случае поговорим чуть позже.

Если компилятор **достигает конца всей последовательности, не найдя совпадения**, он сгенерирует ошибку компиляции, что для вызова функции не может быть найдена соответствующая перегруженная функция.

# Последовательность сопоставления аргументов

**Шаг 1)** Компилятор пытается найти точное совпадение. Это происходит в два этапа. Во-первых, компилятор смотрит, существует ли перегруженная функция, в которой тип аргументов в вызове функции точно соответствует типу параметров в перегруженных функциях. Например:

```
void print(int) {}  
void print(double) {}  
int main()  
{ print(0); // точное совпадение с print(int)  
  print(3.4); /* точное совпадение с  
  print(double) */  
  return 0;  
}
```

Поскольку 0 в вызове функции `print(0)` является `int`, компилятор будет проверять, была ли объявлена перегрузка `print(int)`. Поскольку это так, компилятор определяет, что `print(int)` является точным совпадением.

Во-вторых, компилятор применит ряд тривиальных преобразований к аргументам в вызове функции.

**Тривиальные преобразования** – это набор определенных правил преобразования, которые изменяют типы (без изменения значения) с целью поиска совпадения.

Например, неконстантный тип можно тривиально преобразовать в константный тип:

```
void print(const int) {}  
void print(double) {}  
int main()  
{  
    int x =0 ;  
    print(x) ;  
    // x тривиально конвертируется в const int  
    return 0 ;  
}
```

В приведенном выше примере мы вызвали `print(x)`, где `x` – это число `int`. Компилятор тривиально преобразует `x` из `int` в `const int`, который затем соответствует `print(const int)`.

- Преобразование не ссылочного типа в ссылочный тип (или наоборот) также является тривиальным преобразованием).
- Совпадения, полученные с помощью тривиальных преобразований, считаются точными совпадениями.

**Шаг 2)** Если точное совпадение не найдено, компилятор пытается найти совпадение, применяя к аргументу(ам) числовое продвижение. Ранее мы рассмотрели, как некоторые узкие целочисленные типы и типы с плавающей запятой могут автоматически преобразовываться в более широкие типы, такие как `int` или `double`. Если после числового продвижения совпадение найдено, вызов функции разрешен.

Например:

```
void print(int) {} void print(double) {}  
int main()  
{   print('a'); /* расширяющее преоб-  
разование для соответствия print(int) */  
    print(true); /* расширяющее преоб-  
разование для соответствия print(int) */  
    print(4.5f); /* расширяющее преобразо-  
вание для соответствия print(double) */  
return 0;}
```

Для `print('a')`, поскольку точное совпадение для `print(char)` не может быть найдено на предыдущем шаге, компилятор преобразует `char 'a'` в `int` и ищет совпадение. Это соответствует `print(int)`, поэтому вызов функции разрешается как `print(int)`.



**Шаг 3)** Если совпадение не найдено с помощью числового продвижения, компилятор пытается найти совпадение, применяя к аргументам числовые преобразования. Например:

```
#include <string> // для std::string
void print(double) {}
void print(std::string) {}
int main()
{    print('a'); /* 'a' преобразовано для
соответствия с print(double) */
    return 0; }
```

В этом случае, поскольку нет `print(char)` (точное совпадение) и `print(int)` (совпадение при числовом продвижении), `'a'` численно преобразуется в `double` и сопоставляется с `print(double)`.

**Шаг 4)** Если совпадение не найдено с помощью числового преобразования, компилятор пытается найти совпадение с помощью любых пользовательских преобразований.

Хотя мы еще не рассмотрели пользовательские преобразования, некоторые типы (например, классы) могут определять преобразования в другие типы, которые могут быть вызваны неявно.

Например:

```
/* Мы еще не рассмотрели классы, они будут  
позже */  
class X  
{  
public:  
    operator int() { return 0; }  
/* пользовательское преобразование из X в  
int */  
};  
void print(int) {}  
void print(double) {}  
int main()  
{    X x;  
    print(x); //X преобразуется в int  
    return 0;}
```

В этом примере компилятор сначала проверит, существует ли точное совпадение с `print(X)`. Мы не определили такой функции.

Затем компилятор проверит, можно ли применить к `x` числовое продвижение, чего он не может.

Затем компилятор проверит, можно ли применить к `x` числовое преобразование, чего он также не может.

Наконец, компилятор будет искать любые пользовательские преобразования. Поскольку мы определили пользовательское преобразование из `X` в `int`, компилятор преобразует `X` в `int`, чтобы соответствовать `print(int)`.

**Шаг 5)** Если совпадение не найдено с помощью пользовательского преобразования, компилятор будет искать соответствующую функцию, которая использует многоточие.

Функции, использующие многоточие, имеют вид:

**возвращаемый\_тип**

**имя\_функции(список\_аргументов, ...)**

**Список\_аргументов** – это один или несколько обычных параметров функции. Функции, использующие многоточие, должны иметь хотя бы один параметр до многоточия. Любые аргументы, переданные в функцию, должны сначала соответствовать параметрам списка\_аргументов.

**Многоточие** (которое представлено тремя точками подряд) всегда должно быть последним параметром функции. Многоточие захватывает любые дополнительные аргументы (если они есть). Можно представить себе многоточие как массив, который содержит любые дополнительные параметры. помимо тех, что указаны в списке\_аргументов.

**Шаг 6)** Если к этому моменту совпадений не найдено, компилятор сдастся и выдает ошибку компиляции о невозможности найти подходящую функцию.

# Неоднозначные совпадения

С неперегруженными функциями каждый вызов функции либо будет преобразован в функцию, либо совпадение не будет найдено, и компилятор выдаст ошибку компиляции:

```
void f () {}  
int main()  
{  
    f (); // совпадение найдено  
    g ();  
    // ошибка компиляции: совпадений не найдено  
    return 0;  
}
```

С перегруженными функциями есть третий возможный результат: может быть найдено *неоднозначное совпадение*.

**Неоднозначное совпадение** возникает, когда компилятор находит две или более функции, которые могут быть сопоставлены на одном шаге. Когда это произойдет, компилятор прекратит сопоставление и выдаст ошибку компиляции, заявив, что он обнаружил неоднозначный вызов функции.

Поскольку каждая перегруженная функция для компиляции должна быть различаться, как возможно, что вызов функции может привести к более чем одному совпадению. Давайте посмотрим на пример, который это иллюстрирует:



```
void print(int x) {}  
void print(double d) {}  
int main()  
{    print(5l); // 5l имеет тип long  
    return 0;  
}
```

Поскольку литерал 5l имеет тип long, компилятор сначала проверяет, может ли он найти точное совпадение для print(long), но не найдет его. Затем компилятор попытается выполнить числовое продвижение, но значения типа long не могут быть расширены, поэтому здесь тоже нет совпадений.

После этого компилятор попытается найти совпадение, применив числовые преобразования к аргументу `long`.

В процессе проверки всех правил преобразования чисел компилятор найдет два возможных совпадения.

Если аргумент `long` численно преобразован в `int`, тогда вызов функции будет соответствовать `print(int)`.

Если вместо этого аргумент `long` преобразуется в `double`, тогда он будет соответствовать `print(double)`.

Поскольку посредством числового преобразования были обнаружены два возможных совпадения, вызов функции считается неоднозначным.

Вот еще один пример неоднозначных совпадений:

```
void print(unsigned int x){}
void print(float y){}
int main()
{
    print(0);          /* int можно численно
преобразовать в unsigned int или float*/
    print(3.14159); /* double можно численно
преобразовать в unsigned int или float */
    return 0;}

```

Можно ожидать, что 0 приведет к `print(unsigned int)`, и 3.14159 приведет к `print(float)`. Но значение 0 типа `int` может быть численно преобразовано в `unsigned int` или `float`, поэтому обе перегрузки соответствуют одинаково хорошо, и результатом является неоднозначный вызов функции.

То же самое относится к преобразованию числа типа `double` в тип `float` или `unsigned int`. Оба являются числовыми преобразованиями, поэтому обе перегрузки соответствуют одинаково, и результат снова неоднозначен.

## Разрешение неоднозначных совпадений

Поскольку неоднозначные совпадения являются **ошибкой времени компиляции**, их необходимо устранить, прежде чем ваша программа будет компилироваться. Есть несколько способов разрешить неоднозначные совпадения:

Часто лучший способ – просто определить **новую перегруженную функцию**, которая принимает параметры именно того типа, с которым вы пытаетесь ее вызвать. Тогда C++ сможет найти точное совпадение для вызова функции.

В качестве альтернативы явно приведите неоднозначные аргументы к типу функции, которую хотите вызвать.

Например, чтобы `print(0)` вызывал `print(unsigned int)`, вы должны сделать это:

```
int x= 0 ;  
print(static_cast<unsigned int>(x)) ;  
    // вызовет print(unsigned int)
```

Если ваш аргумент является литералом, вы можете использовать суффикс литерала, чтобы убедиться, что ваш литерал интерпретируется как правильный тип:

```
print(0u) ;  
/* вызовет print(unsigned int) , поскольку  
суффикс 'u' = unsigned int */
```

# Суффиксы литералов

Если тип литерала по умолчанию не соответствует необходимому, вы можете изменить тип литерала, добавив суффикс:

Тип данных	Суффикс	Назначение
int	u или U	unsigned int
int	l или L	long
int	ul, uL, Ul, UL, lu, lU, Lu или LU	unsigned long
int	ll или LL	long long
int	ull, uLL, Ull, ULL, llu, llU, LLu или LLU	unsigned long long
double	f или F	float
double	l или L	l

## **Сопоставление функций с несколькими аргументами**

Если аргументов несколько, компилятор по очереди применяет правила сопоставления к каждому аргументу. Выбирается так функция, для которой каждый аргумент соответствует по крайней мере так же хорошо, как и у всех других функций, причем по крайней мере один аргумент соответствует лучше, чем у всех других функций. Другими словами, выбранная функция должна обеспечивать лучшее соответствие, чем все другие функции-кандидаты, по крайней мере, по одному параметру и не хуже по всем остальным параметрам.



В случае, если такая функция будет найдена, это явный и однозначно лучший выбор. Если такая функция не найдена, вызов будет считаться неоднозначным (или без совпадений). Например:

```
#include <iostream>

void print(char c, int x) //лучшее совпадение
{
    std::cout << 'a';}

void print(char c, double x)
{
    std::cout << 'b';}

void print(char c, float x)
{
    std::cout << 'c';
}

int main()
{
    print('x', 'a'); return 0;}
```

В приведенной выше программе все функции точно соответствуют по первому аргументу. Однако верхняя функция соответствует по второму параметру через числовое продвижение, тогда как другие функции требуют числового преобразования. Таким образом, `print(char, int)` однозначно является лучшим совпадением.

# Строки в стиле C++.

## Знакомство с std::string

Чтобы использовать строки в C++, нам сначала нужно включить через `#include` заголовочный файл `<string>`, чтобы ввести объявления для `std::string`. Как только это будет сделано, мы сможем определить переменные типа `std::string`. В примерах используем стандартное пространство имен

```
using namespace std;
```

```
string myName = ""; // пустая строка
```

```
myName = "John";
```

```
// присваиваем переменной myName строковый
```

```
//литерал "John"
```

Обратите внимание, что строки также могут содержать цифры:

```
string myID= "45" ;
```

// "45" не то же самое, что целое число 45!

В строковой форме числа обрабатываются как текст, а не как числа, и поэтому ими нельзя манипулировать как числами (например, вы не можете их умножать). С++ не будет автоматически преобразовывать строковые числа в целочисленные значения или значения с плавающей точкой.

## Вывод строк

Строки можно выводить с помощью `std::cout`:

```
#include <iostream>
```

```
#include <string>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    string myName= "Alex ";
```

```
    cout << "My name is: " << myName <<  
'\n';
```

```
    return 0;
```

```
}
```

## Ввод строк с помощью std::cin

Рассмотрим пример:

```
#include <iostream>
#include <string>
using namespace std;
int main()
{cout << "Enter your full name: ";
string name, age= "";
cin >> name; /* это не будет работать
должным образом, так как std::cin
прерывается на пустых символах*/
cout << "Enter your age: "; cin >> age;
cout << "Your name is " << name << " and
your age is " << age << '\n';
return 0;}
```

Enter your full name: John Doe

Enter your age: Your name is John and your age is Doe

При использовании оператора для извлечения из `cin` строки оператор возвращает только символы до первого попавшегося пробела. Все остальные символы остаются внутри `std::cin` в ожидании следующего извлечения.

Поэтому, когда мы использовали `operator>>` для извлечения строки в переменную `name`, было извлечено только "John", оставив "Doe" внутри `std::cin`. Когда мы снова использовали `operator>>` для получения переменной `age`, он извлек "Doe" вместо того, чтобы ждать, пока мы введем возраст.

# Использование `std::getline()` для ввода текста

Чтобы прочитать всю строку входных данных в переменную строки, лучше использовать функцию `std::getline()`. `std::getline()` принимает два параметра: первый – это `std::cin`, а второй – ваша строковая переменная.

Вот та же программа, что и выше, с использованием `std::getline()`:



```
#include <string>
//для std::string и std::getline
#include <iostream>
#include <iomanip>      // для std::ws
int main()
{
    std::cout << "Enter your full name: ";
    std::string name, age= "";
    std::getline(std::cin >> std::ws, name);
// считываем полную строку текста в name
    std::cout << "Enter your age: ";
    std::getline(std::cin >> std::ws, age);
// считываем полную строку текста в age
    std::cout << "Your name is " << name <<
" and your age is " << age << '\n';
    return 0;}
```

Теперь наша программа работает как и ожидалось:

```
Enter your full name: John Doe
```

```
Enter your age: 23
```

```
Your name is John Doe and your age is 23
```

## Что такое `std::ws`?

C++ поддерживает манипуляторы ввода (определенные в заголовке `<iomanip>`), которые изменяют способ приема входных данных.

Манипулятор ввода `std::ws` говорит `std::cin` игнорировать любые начальные пробелы. Обратите внимание, что `std::ws` не является функцией.

```
#include <string>
#include <iostream>
int main(){
    std::cout << "Pick 1 or 2: ";
    int choice;
    std::cin >> choice;
    std::cout << "Now enter your name: ";
    std::string name{};
    std::getline(std::cin, name);
    // примечание: здесь нет std::ws
    std::cout << "Hello, " << name << ", you
picked " << choice << '\n';

    return 0;
}
```

Вот пример работы программы:

```
Pick 1 or 2: 2
```

```
Now enter your name: Hello, , you picked 2
```

Данная программа сначала просит вас ввести 1 или 2 и ждет, когда вы это сделаете. Пока всё хорошо. Затем она просит вас ввести свое имя. Однако на самом деле она не будет ждать, пока вы введете свое имя! Вместо этого она печатает строку "Hello", а затем завершит работу.

Когда вы вводите значение с помощью оператора, `std::cin` не только захватывает значение, но также захватывает символ новой строки ('\n'), который появляется, когда вы нажимаете клавишу Enter. Итак, когда мы набираем 2 и нажимаем Enter, `std::cin` получает строку "2\n". Затем он извлекает 2 в переменную `choice`, оставляя символ новой строки на потом. Затем, когда `std::getline()` переходит к чтению имени, он видит, что "\n" уже находится в потоке, и полагает, что мы, должно быть, ввели пустую строку.

Мы можем изменить приведенную выше программу, чтобы использовать манипулятор ввода `std::ws`, чтобы указать `std::getline()` игнорировать любые начальные пробельные символы:

```
#include <string>
#include <iostream>
int main(){
    std::cout << "Pick 1 or 2: ";
    int choice;
    std::cin >> choice;
    std::cout << "Now enter your name: ";
    std::string name;
    std::getline(std::cin >> std::ws, name);
    // примечание: здесь добавлен std::ws
    std::cout << "Hello, " << name << ",
you picked " << choice << '\n';

    return 0;
}
```

Теперь эта программа будет работать так:

```
Pick 1 or 2: 2
```

```
Now enter your name: Alex
```

```
Hello, Alex, you picked 2
```

Если для чтения строк используется функция `std::getline`, используйте манипулятор ввода `std::ws`, чтобы игнорировать начальные пробелы.

Использование оператора извлечения (`>>`) с `std::cin` игнорирует начальные пробелы.

`std::getline` не игнорирует начальные пробелы, если вы не используете манипулятор ввода `std::ws`.

После определения имя typedef можно использовать везде, где требуется тип. Например, мы можем создать переменную с именем typedef в качестве типа:

```
distance_t milesToDestination= 3.4;  
// определяет переменную типа double
```

Когда компилятор встречает имя typedef, он подставляет тип, на который указывает typedef. Например:

```
#include <iostream>  
  
int main()  
{    typedef double distance_t; /*определяем  
distance_t как псевдоним для типа double */  
    distance_t milesToDestination= 3.4;  
// определяет переменную типа double  
    std::cout << milesToDestination << '\n';  
// выводит значение типа double  
    return 0;}
```



## typedef не определяет новый тип

Обратите внимание, что typedef не определяет новый тип. Он создает новый идентификатор (псевдоним) для существующего типа. typedef можно использовать как замену везде, где можно использовать обычный. Например:

```
int main()
{
    typedef long miles_t; /* определяет
miles_t как псевдоним для типа long */
    typedef long speed_t; /* определяет
speed_t как псевдоним для типа long*/
    miles_t distance =5; // distance это long
    speed_t mhz      = 3200; // mhz тоже long
    /* Следующее синтаксически корректно
(но семантически бессмысленно) */
    distance = mhz;
    return 0;}
```

Хотя мы предполагаем, что `miles_t` и `speed_t` имеют разные значения, оба они являются просто псевдонимами для типа `long`.

Это фактически означает, что значения типа `miles_t`, `speed_t` и `long` могут использоваться взаимозаменяемо.

И действительно, когда мы присваиваем значение типа `speed_t` переменной типа `miles_t`, компилятор видит только то, что мы присваиваем значение типа `long` переменной типа `long`, и он не будет предупреждать об ошибке.

## **#ifdef и #ifndef**

`#ifdef` и `#ifndef` означает «если определено» и «если не определено». Стандартный вид `#ifdef` следующий:

```
#ifdef имя_макроса
```

```
//последовательность операторов
```

```
#endif
```

Если имя макроса определено ранее в операторе `#define`, то последовательность операторов, стоящих между `#ifdef` и `#endif`, будет компилироваться. Стандартный вид `#ifndef` следующий:

```
#ifndef имя_макроса
```

```
//последовательность операторов
```

```
#endif
```

Если имя макроса не определено ранее в операторе `#define`, то последовательность операторов, стоящих между `#ifdef` и `#endif`, будет компилироваться.

Как `#ifdef`, так и `#ifndef` могут использовать оператор `#else`, но не `#elif`. Например:

```
#include <stdio.h>
#define TED 10
int main(void)
{
#ifdef TED
printf("Hi Ted\n");
#else
printf("Hi anyone\n");
#endif
#ifndef RALPH
printf("RALPH not defined\n");
#endif
return 0;}
```

Hi Ted RALPH not defined
-----------------------------

выводит «Hi Ted» и «RALPH not defined». Если TED не определен, то выведется «Hi anyone», а за ним «RALPH not defined».

`#ifdef` и `#ifndef` можно вкладывать друг в друга так же, как и `#if`.

## Область видимости typedef

Поскольку область видимости является свойством идентификатора, идентификаторы typedef подчиняются тем же правилам области видимости, что и идентификаторы переменных: typedef, определенный внутри блока, имеет область видимости блока и может использоваться только внутри этого блока, тогда как typedef, определенный в глобальном пространстве имен, имеет область видимости файла и может использоваться до конца файла. В приведенном выше примере `miles_t` и `speed_t` можно использовать только в функции `main()`.

Если вам нужно использовать один или несколько typedef в нескольких файлах, их можно определить в заголовочном файле и включить через `#include` в любые файлы исходного кода, которые должны использовать это определение:

```
// файл mytypes.h:
```

```
#ifndef MYTYPES
```

```
#define MYTYPES
```

```
    typedef long miles_t;
```

```
    typedef long speed_t;
```

```
#endif
```

Определения typedef, включенные таким образом, будут импортированы в глобальное пространство имен и, следовательно, будут иметь глобальную область видимости.

## Использование псевдонимов типов для повышения читабельности

Псевдонимы типов также могут помочь в документации и понимании кода. Что касается переменных, у нас есть идентификатор переменной, который помогает документировать назначение переменной. Но рассмотрим случай значения, возвращаемого функцией. Типы данных, такие как `char`, `int`, `long`, `double` и `bool`, хороши для описания того, какой тип возвращает функция, но чаще мы хотим знать, какой цели служит возвращаемое значение.

Например:

```
int GradeTest ( ) ;
```

Мы видим, что возвращаемое значение является целым числом, но что означает целое число? Буквенная оценка? Количество пропущенных вопросов? Идентификационный номер студента? Код ошибки? Кто знает! Тип возвращаемого значения `int` мало что говорит нам.



Возможно, где-то существует документация по этой функции, к которой мы можем обратиться. Если нет, мы должны прочитать код и определить назначение сами.

Создадим эквивалентную версию, используя псевдоним типа:

```
typedef int testScore_t;  
testScore_t GradeTest();
```

Использование типа возвращаемого значения `testScore_t` делает очевидным, что функция возвращает тип, представляющий результат теста.

Создание псевдонима типа только для документирования типа возвращаемого значения отдельной функции не стоит усилий (вместо этого используйте комментарий). Но если вы уже создали `typedef` по другим причинам, это может быть приятным дополнительным преимуществом.

## Использование псевдонимов типов для упрощения поддержки кода

Псевдонимы типов также позволяют изменять базовый тип объекта без изменения большого количества кода.

Например, если вы использовали `short` для хранения идентификационного номера студента, но позже решили, что вам нужен `long`, вам придется во всем коде заменить `short` на `long`. Вероятно, будет сложно понять, какой `short` используется для хранения номеров ID, а какой – для других целей.

Однако с псевдонимом типа всё, что вам нужно сделать, это изменить `studentID_t = short;` на `studentID_t = long;`.

```
typedef short studentID_t;
```

```
typedef long studentID_t;
```

Хотя это кажется приятным преимуществом, необходимо соблюдать осторожность при изменении типа, поскольку поведение программы также может измениться. Это особенно верно при изменении типа псевдонима типа на тип из другого семейства типов (например, целочисленный тип на значение с плавающей точкой или наоборот)! Новый тип может иметь проблемы со сравнением или делением целых чисел на числа с плавающей точкой или другие проблемы, которых не было у старого типа. Если вы измените существующий тип на какой-либо другой, ваш код следует тщательно повторно протестировать.

# Передача аргументов по ссылке

У передачи по значению есть несколько ограничений. Во-первых, при передаче в функцию большой структуры передача по значению создаст копию аргумента в параметре функции. Во многих случаях это ненужное снижение производительности, поскольку исходного аргумента было бы достаточно. Во-вторых, при передаче аргументов по значению единственный способ вернуть значение вызывающей функции – через возвращаемое значение функции. Есть случаи, когда было бы более понятно и эффективно, чтобы функция изменяла переданный аргумент. Передача по ссылке решает обе эти проблемы.

**Ссылка** — это тип переменной в языке C++, который работает как псевдоним другого объекта или значения.

```
#include <iostream>

int main()
{int value = 7; // обычная переменная
int &ref = value; //ссылка на переменную value
  value = 8; // value теперь 8
ref = 9; // value теперь 9
std::cout << value << std::endl;
// выведется 9
++ref;
std::cout << value << std::endl;
// выведется 10
return 0;
}
```

9
10

## Передача по ссылке

Чтобы передать переменную по ссылке, мы просто объявляем параметры функции как ссылки, а не как обычные переменные:

```
void addOne(int &ref)  
// ref - переменная-ссылка  
{ ref = ref + 1; }
```

Когда функция вызывается, `ref` станет ссылкой на аргумент. Поскольку ссылка на переменную обрабатывается точно так же, как и сама переменная, любые изменения, внесенные в ссылку, передаются аргументу.

```
void addOne(int &ref)
```

```
{
```

```
    ref = ref + 1;
```

```
}
```

```
int main()
```

```
{    int value= 5;
```

```
    cout << "value = " << value << '\n';
```

```
    addOne(value);
```

```
    cout << "value = " << value << '\n';
```

```
    return 0;
```

```
}
```

value = 5

value = 6

Параметр является ссылкой, а не обычной переменной.

Когда мы вызываем addOne(value), ref становится ссылкой на переменную value из main.

## Возврат нескольких значений через выходные параметры

Иногда нам нужно, чтобы функция возвращала несколько значений. Однако функции могут иметь только одно возвращаемое значение. Один из способов вернуть несколько значений – использовать параметры-ссылки:

```
#include <iostream>
#include <cmath>//для std::sin() и std::cos()
void getSinCos(double degrees, double
&sinOut, double &cosOut)
{/* sin() и cos() принимают радианы, а не
градусы, поэтому нам нужно преобразование*/
    static const double pi =
3.14159265358979323846 ; // значение пи
    double radians=degrees * pi / 180.0;
    sinOut = std::sin(radians);
    cosOut = std::cos(radians);}
```



```
int main()
{
    double sin= 0.0 ;
    double cos= 0.0 ;

    /* getSinCos вернет sin и cos в
переменных sin и cos*/
    getSinCos(30.0, sin, cos);
    std::cout << "The sin is " << sin <<
'\n';
    std::cout << "The cos is " << cos <<
'\n';
    return 0;
}
```

Эта функция принимает один параметр (по значению) в качестве входных данных и «возвращает» два параметра (по ссылке) в качестве выходных данных. Параметры, которые используются только для возврата значений вызывающей стороне, называются **выходными параметрами**. Эти параметры с суффиксом `out`, чтобы обозначить, что эти параметры выходные. Это помогает напомнить вызывающему, что начальное значение, переданное этим параметрам, не имеет значения и что мы должны ожидать их перезаписи. По соглашению выходные параметры обычно являются крайними правыми параметрами.

Давайте рассмотрим, как это работает, более подробно. Сначала функция `main` создает локальные переменные `sin` и `cos`. Они передаются в функцию `getSinCos()` по ссылке (а не по значению). Это означает, что функция `getSinCos()` имеет доступ к реальным переменным `sin` и `cos`, а не к копиям. `getSinCos()` присваивает новые значения `sin` и `cos` (через ссылки `sinOut` и `cosOut` соответственно), которые перезаписывают старые значения в `sin` и `cos`. Затем `main` печатает эти обновленные значения.

Если бы `sin` и `cos` были переданы по значению, а не по ссылке, `getSinCos()` изменила бы копии `sin` и `cos`, что привело бы к отмене любых изменений в конце функции. Но поскольку `sin` и `cos` были переданы по ссылке, любые изменения, внесенные в `sin` и `cos` (через ссылки), сохраняются за пределами функции. Таким образом, мы можем использовать этот механизм для возврата значений обратно вызывающей стороне.

Этот метод, хотя и работает, имеет несколько незначительных недостатков. Во-первых, вызывающий должен передать аргументы для хранения обновленных выходных данных, даже если он не намеревается их использовать. Что еще более важно, синтаксис немного неестественный: и входные, и выходные параметры объединяются в вызове функции. Со стороны вызывающего не очевидно, что `sin` и `cos` являются выходными параметрами и будут изменены. Это, наверное, самая опасная часть этого метода (так как может привести к ошибкам). Некоторые программисты и компании считают, что это достаточно большая проблема, чтобы посоветовать вообще избегать выходных параметров или использовать для выходных параметров вместо этого передачу по адресу (что имеет более четкий синтаксис, указывающий, можно ли изменять параметр или нет).

Принято обозначать параметры (и выходные аргументы) суффиксом (или префиксом) "out", чтобы помочь прояснить, что значение может быть изменено.

Эта функция принимает один параметр (по значению) в качестве входных данных и «возвращает» два параметра (по ссылке) в качестве выходных данных. Параметры, которые используются только для возврата значений вызывающей стороне, называются **выходными параметрами**. Мы назвали эти параметры с суффиксом out, чтобы обозначить, что эти параметры выходные. Это помогает напомнить вызывающему, что начальное значение, переданное этим параметрам, не имеет значения и что мы должны ожидать их перезаписи. По соглашению выходные параметры обычно являются крайними правыми параметрами.

## Краткий обзор l-value и r-value

**l-value** — это объект, который имеет определенный адрес памяти (например, переменная  $x$ ) и сохраняется за пределами одного выражения.

**r-value** — это временное значение без определенного адреса памяти и с областью видимости выражения (т.е. сохраняется в пределах одного выражения).

В качестве r-values могут быть как результаты выражения (например,  $2 + 3$ ), так и литералы.

## Инициализация ссылок

Ссылки должны быть инициализированы при создании:

```
int value = 7;
```

```
int &ref = value; /* корректная ссылка:  
инициализирована переменной value */
```

```
int &invalidRef; /* некорректная ссылка:  
ссылка должна ссылаться на что-либо*/
```

В отличие от указателей, которые могут содержать нулевое значение, ссылки нулевыми быть не могут.

Ссылки на **неконстантные** значения могут быть инициализированы только **неконстантными l-values**. Они не могут быть инициализированы **константными l-values** или **r-values**:

```
int a = 7;  
int &ref1 = a; /* верно: a - это  
неконстантное l-value */  
const int b = 8;  
int &ref2 = b; /* неверно: b - это  
константное l-value*/  
int &ref3 = 4; // неверно: 4 - это r-value
```

Обратите внимание, во втором случае вы не можете инициализировать неконстантную ссылку константным объектом. В противном случае, вы бы могли изменить значение константного объекта через ссылку, что уже является нарушением понятия «константа».

После инициализации изменить объект, на который указывает ссылка — нельзя. Рассмотрим следующий фрагмент кода:

```
int value1 = 7;  
int value2 = 8;  
int &ref = value1; /* верно, ref - теперь  
псевдоним для value1*/  
ref = value2; /* присваиваем 8 (значение  
переменной value2) переменной value1. Здесь  
НЕ изменяется объект, на который ссылается  
ссылка!*/
```

Обратите внимание, в четвертом присваивании (`ref = value2;`) выполняется не то, что вы могли бы ожидать! Вместо переприсваивания `ref` (ссылаться на переменную `value2`), значение из `value2` присваивается переменной `value1` (на которое и ссылается `ref`).



## Ссылки vs. Указатели

Ссылка — это тот же указатель, который неявно разыменовывается при доступе к значению, на которое он указывает («под капотом» ссылки реализованы с помощью указателей). Таким образом, в следующем коде:

```
int value = 7;  
int *const ptr = &value;  
int &ref = value;
```

\*ptr и ref обрабатываются одинаково. Т.е. это одно и то же:

```
*ptr = 7; ref = 7;
```

Поскольку ссылки должны быть инициализированы корректными объектами (они не могут быть нулевыми) и не могут быть изменены позже, то они, как правило, безопаснее указателей (так как риск разыменования нулевого указателя отпадает). Однако они несколько ограничены в функциональности по сравнению с указателями.

Если определенное задание может быть решено с помощью как ссылок, так и указателей, то лучше использовать ссылки. Указатели следует использовать только в тех ситуациях, когда ссылки являются недостаточно эффективными (например, при динамическом выделении памяти).

Ссылки позволяют определять псевдонимы для других объектов или значений. Ссылки на неконстантные значения могут быть инициализированы только неконстантными l-values. Они не могут быть переприсвоены после инициализации. Ссылки чаще всего используются в качестве параметров в функциях, когда мы хотим изменить значение аргумента или хотим избежать его затратного копирования.

## **Передача по константной ссылке**

Ссылки позволяют функции изменять значение аргумента, что нежелательно, если мы хотим, чтобы аргумент был доступен только для чтения. Если мы знаем, что функция не должна изменять значение аргумента, но не хотим передавать по значению, лучшим решением будет передача по константной ссылке.

Константная ссылка – это ссылка, которая не позволяет изменять переменную, на которую она ссылается.

Следовательно, если мы используем константную ссылку в качестве параметра, мы гарантируем вызывающему, что функция не изменит аргумент.

## Использование **const** полезно по нескольким причинам:

- Он обращается к компиляторам за помощью в обеспечении того, чтобы значения, которые нельзя изменять, не изменились (компилятор выдаст ошибку, если вы попытаетесь это сделать, как в приведенном выше примере).
- Он сообщает программисту, что функция не изменит значение аргумента. Это может помочь с отладкой.
- Вы не можете передать константный аргумент неконстантному параметру-ссылке. Использование константных параметров гарантирует, что вы можете передавать функции как неконстантные, так и константные аргументы.
- Ссылки на константные значения могут принимать аргументы любого типа, включая неконстантные l-значения, константные l-значения и r-значения.

Неконстантные ссылки не могут связываться с r-значениями. Функция с неконстантным параметром-ссылкой не может быть вызвана с литералами или временными значениями.

```
#include <string>

void f (std::string& text) {}

int main()
{
    std::string text= "hello";

    f(text); // верно
    f(text + " world"); /* недопустимо,
неконстантные ссылки не могут связываться с
r-значениями*/
    return 0;
}
```

## Ссылки на указатели

Указатель можно передать по ссылке, и функция в этом случае может изменить адрес, на который указывает указатель:

```
#include <iostream>
void foo(int *&ptr)
/* передать указатель по ссылке*/
{
    ptr = 0;
/* это изменяет фактический переданный
аргумент ptr, а не копию */
}
```

```
int main()
{
    int x=5;
    int *ptr= &x;
    std::cout << "ptr is: " << (ptr ? "non-
null" : "null") << '\n'; // выводит non-null
    foo(ptr);
    std::cout << "ptr is: " << (ptr ? "non-
null" : "null") << '\n'; // выводит null

    return 0;
}
```

Можно передать по ссылке массив в стиле C. Это полезно, если в функции вам нужна возможность изменять массив (например, для функции сортировки) или вам нужен доступ к информации о типе фиксированного массива (для выполнения `sizeof()`). Однако обратите внимание, что для того, чтобы это работало, в параметре вам необходимо явно указать размер массива:

```
#include <iostream>
```

```
/* Примечание: вам нужно указать размер  
массива в объявлении функции */
```

```
void printElements(int (&arr)[4])  
{int length= sizeof(arr) / sizeof(arr[0]);  
  for (int i= 0; i < length; ++i)  
  {    std::cout << arr[i] << '\n';  
    }  
}
```



```
int main()  
{  
    int arr[]={ 99, 20, 14, 80 };  
  
    printElements(arr);  
  
    return 0;  
}
```

Это означает, что это работает только с фиксированными массивами одной конкретной длины.

# Плюсы и минусы передачи по ссылке

## Преимущества передачи по ссылке:

- Ссылки позволяют функции изменять значение аргумента, что иногда бывает полезно. В противном случае можно использовать константные ссылки, чтобы гарантировать, что функция не изменит аргумент.
- Поскольку копия аргумента не создается, передача по ссылке выполняется быстро, даже при использовании с большими структурами или классами.
- Ссылки могут использоваться для возврата нескольких значений из функции (через выходные параметры).
- Ссылки должны быть инициализированы, поэтому о нулевых значениях можно не беспокоиться.

## Недостатки передачи по ссылке

- Поскольку неконстантная ссылка не может быть инициализирована константным l-значением или r-значением (например, литералом или выражением), аргументы неконстантных ссылочных параметров должны быть обычными переменными.
- Может быть трудно определить, предназначен ли аргумент, переданный неконстантной ссылкой, для входных данных, выходных данных или для того и другого. Разумное использование `const` и суффикса именования выходных переменных проясняет это.
- По вызову функции невозможно сказать, может ли аргумент измениться. Аргументы, переданные по значению и переданные по ссылке, выглядят одинаково. Мы можем определить, передан ли аргумент по значению или по ссылке, только взглянув на объявление функции. Это может привести к ситуациям, когда программист не понимает, что функция изменит значение аргумента.

## **Когда использовать передачу по ссылке**

- При передаче структур или классов (используйте `const`, если они доступны только для чтения).
- Когда вам нужна функция для изменения аргумента.
- Когда вам нужен доступ к информации о типе фиксированного массива.

## **Когда не использовать передачу по ссылке**

При передаче базовых типов, которые не нужно изменять (используйте передачу по значению).

## Длина строки

Если нужно определить длину строки, используем переменную `std::string`.

```
#include <iostream>
#include <string>
int main()
{
    std::string myName= "Alex" ;
    std::cout << myName << " has " <<
myName.length() << " characters\n";
    return 0;
}
```

Alex has 4 characters

Обратите внимание, что вместо того, чтобы запрашивать длину строки как `length(myName)`, мы говорим `myName.length()`.

Функция `length()` не является обычной автономной функцией – это особый тип функции, которая принадлежит `std::string` и называется функцией-членом.

К отдельным символам строки можно обращаться по индексу, как к элементам массива или C-строк.

Например `S[0]` - это первый символ строки.

# Арифметические операторы

Со строками можно выполнять следующие арифметические операции.

- = - присваивание значения.
- += - добавление в конец строки другой строки или символа.
- + - конкатенация двух строк, конкатенация строки и символа.
- ==, != - посимвольное сравнение.
- <, >, <=, >= - лексикографическое сравнение.

```
#include <string>
#include <iostream>
int main(){
    // тип string, операции над строками
    string s1 = "s-1";
    string s2 = "s-2";
    string s3;
    bool b;

    // операция '=' (присваивание строк)
    s3 = s1; // s3 = "s-1"

    // операция '+' - конкатенация строк
    s3 = s3 + s2; // s3 = "s-1s-2"
```



// операция '+' - присваивание с конкатенацией

```
s3 = "s-3";  
s3 += "abc"; // s3 = "s-3abc"
```

// операция '==' - сравнение строк

```
b = s2==s1; // b = false  
b = s2=="s-2"; // b = true
```

// операция '!=' - сравнение строк (не равно)

```
s1 = "s1";  
s2 = "s2";  
b = s1 != s2; // b = true
```

// операции '<' и '>' - сравнение строк

s1 = "abcd";

s2 = "de";

b = s1 > s2; // b = false

b = s1 < s2; // b = true

// операции '<=' и '>=' - сравнение строк

// (меньше или равно, больше или равно)

s1 = "abcd";

s2 = "ab";

b = s1 >= s2; // b = true

b = s1 <= s2; // b = false

b = s2 >= "ab"; // b = true

// операция [] - индексация

char c;

s1 = "abcd";

c = s1[2]; // c = 'c'

c = s1[0]; // c = 'a'

return 0;

}

# Конструкторы строк

Строки можно создавать с использованием следующих конструкторов:

- `string();` -конструктор по умолчанию (без параметров) создает пустую строку;
- `string(const char * str);` - инициализация;
- `string(const string & str)` – инициализация.

## Примеры инициализации с помощью

```
string s0; // инициализация - конструктор
//string()
string s1("Hello!");
// инициализация - конструктор string(const
//char * str)
string s2 = "Hello!";
// инициализация - конструктор string(const
//char * str)
char ps[] = "Hello";
string s3(ps); // инициализация
string s4(s3);
// инициализация - конструктор string(const
string & str)
```

# Присваивание строк. Функция assign()

Чтобы присвоить одну строку другой, можно применить один из двух методов:

- использовать оператор присваивания '=';
- использовать функцию assign() из класса string.

Функция assign() имеет несколько перегруженных реализаций.

**Первый вариант** – это вызов функции без параметров

```
string &assign(void) ;
```

В этом случае происходит простое присваивание одной строки другой.

**Второй вариант** позволяет копировать заданное количество символов из строки:

```
string &assign(const string & s, size_type  
st, size_type num);
```

где

s – объект, из которого берется исходная строка;

st – индекс (позиция) в строке, из которой начинается копирование num символов;

num – количество символов, которые нужно скопировать из позиции st;

size\_type – порядковый тип данных.

**Третий вариант** функции `assign()` копирует в вызывающий объект первые `num` символов строки `s`:

```
string & assign(const char * s, size_type  
num) ;
```

где

`s` — строка, которая завершается символом `'\0'`;

`num` — количество символов, которые копируются в вызывающий объект. Копируются первые `num` символов из строки `s`.



```
// присваивание строк, функция assign()  
string s1 = "bestprog.net";  
string s2;  
string s3;  
char ps[] = "bestprog.net";
```

```
s3 = s1; // s3 = "bestprog.net"  
s2.assign(s1); // s2 = "bestprog.net"  
s2.assign(s1, 0, 4); // s2 = "best"  
s2.assign(ps, 8); // s2 = "bestprog"
```

# Объединение строк. Функция append().

Для объединения строк используется функция append(). Для добавления строк также можно использовать операцию '+', например:

```
string s1;  
string s2;  
s1 = "abc";  
s2 = "def";  
s1 = s1 + s2; // s1 = "abcdef"
```

Функция append() хорошо подходит, если нужно добавлять часть строки.

Функция имеет следующие варианты реализации:

```
string &append(const string & s, size_type  
start, size_t sublen) ;
```

```
string &append(const char * s, size_type  
num) ;
```

В первом варианте реализации функция получает ссылку на строчный объект `s`, который добавляется к вызывающему объекту.

Во втором варианте реализации функция получает указатель на строку типа `const char *`, которая завершается символом `'\0'`.

```
string s1 = "abcdef";  
s2 = "1234567890";  
s1.append(s2, 3, 4); // s1 = "abcdef4567"
```

```
char ps []= "1234567890";  
s1 = "abcdef";  
s1.append(ps, 3); // s1 = "abcdef123"
```

## Вставка символов в строке. Функция insert()

Чтобы вставить одну строку в заданную позицию другой строки нужно использовать функцию insert().

**Первый вариант** функции позволяет вставить полностью всю строку s в заданную позицию start вызывающей строки (вызывающего объекта):

```
string & insert(size_type start, const  
string &s) ;
```

**Второй вариант** функции позволяет вставить часть (параметры insStart, num) строки s в заданную позицию start вызывающей строки:

```
string & insert(size_type start, const  
string &s, size_type insStart, size_type  
num) ;
```

В вышеприведенных функциях:

`s` – строка, которая вставляется в вызывающую строку;

`start` – позиция в вызывающей строке, из которой осуществляется вставка строки `s`;

`insStart` – позиция в строке `s`, из которой происходит вставка;

`num` – количество символов в строке `s`, которые вставляются с позиции `insStart`.

```
string s1 = "abcdef";  
string s2 = "1234567890";
```

```
s1.insert(3, s2); /* s1 =  
"abc"+"1234567890"+"def"="abc1234567890def"*  
/
```

```
s2.insert(2, s1, 1, 3);  
// s2 = "12bcd34567890"
```

## Замена символов в строке. Функция replace()

Функция replace() выполняет замену символов в вызывающей строке. Функция имеет следующие варианты реализации:

```
string &replace(size_type start, size_type  
num, const string &s);
```

```
string &replace(size_type start, size_type  
num, const string &s, size_type replStart,  
size_type replNum);
```

В первом варианте реализации вызывающая строка заменяется строкой s. Есть возможность задать позицию (start) и количество символов (num) в вызывающей строке, которые нужно заменить строкой s.



Второй вариант функции `replace()` отличается от первого тем, что позволяет заменять вызывающую строку только частью строки `s`. В этом случае задаются два дополнительных параметра:

позиция `replStart` и

количество символов в строке `s`, которые образуют подстроку, которая заменяет вызывающую строку.

// замена символов, функция replace()

```
string s1 = "abcdef";
```

```
string s2 = "1234567890";
```

```
s2.replace(2, 4, s1); // s2 = "12abcdef7890"
```

```
s2 = "1234567890";
```

```
s2.replace(3, 2, s1);
```

```
// s2 = "123abcdef67890"
```

```
s2 = "1234567890";
```

```
s2.replace(5, 1, s1);
```

```
// s2 = "12345abcdef7890"
```

```
// замена символов, функция replace()
```

```
s2 = "1234567890";
```

```
s2.replace(5, 1, s1, 2, 3);
```

```
// s2 = "12345cde7890"
```

```
s2 = "1234567890";
```

```
s2.replace(4, 2, s1, 0, 4);
```

```
// s2 = "1234abcd7890"
```

# Удаление заданного количества символов из строки. Функция erase()

Для удаления символов из вызывающей строки используется функция erase():

```
string & erase(size_type index=0, size_type  
num = npos);    где
```

index – индекс (позиция), начиная из которой нужно удалить символы в вызывающей строке;

num – количество символов, которые удаляются.

Пример.

```
string s = "01234567890";  
s.erase(3, 5); // s = "012890"  
s = "01234567890";  
s.erase(); // s = ""
```

# Поиск символа в строке. Функции `find()` и `rfind()`

В классе `string` поиск строки в подстроке можно делать двумя способами, которые отличаются направлением поиска:

- путем просмотра строки от начала до конца с помощью функции `find()`;
- путем просмотра строки от конца к началу функцией `rfind()`.

Прототип функции find() имеет вид:

```
size_type find(const string &s, size_type  
start = 0) const;
```

где

s – подстрока, которая ищется в строке, что вызывает данную функцию. Функция осуществляет поиск первого вхождения строки s. Если подстрока s найдена в строке, что вызвала данную функцию, тогда возвращается позиция первого вхождения. В противном случае возвращается -1;

start – позиция, из которой осуществляется поиск.

Прототип функции `rfind()` имеет вид:

```
size_type rfind(const string &s, size_type  
start = npos) const;
```

где

`s` – подстрока, которая ищется в вызывающей строке. Поиск подстроки в строке осуществляется от конца к началу. Если подстрока `s` найдена в вызывающей строке, то функция возвращает позицию первого вхождения. В противном случае функция возвращает `-1`;

`npos` – позиция последнего символа вызывающей строки;

`start` – позиция, из которой осуществляется поиск.

**Пример 1.** Фрагмент кода, который демонстрирует результат работы функции find()

```
// тип string, функция find()
```

```
string s1 = "01234567890";
```

```
string s2 = "345";
```

```
string s3 = "abcd";
```

```
int pos;
```

```
pos = s1.find(s2); // pos = 3
```

```
pos = s1.find(s2, 1); // pos = 3
```

```
pos = s1.find("jklmn", 0); // pos = -1
```

```
pos = s1.find(s3); // pos = -1
```

```
pos = s2.find(s1); // pos = -1
```



**Пример 2.** Фрагмент кода, который демонстрирует результат работы функции `rfind()`

*// тип string, функции find() и rfind()*

`string s1 = "01234567890";`

`string s2 = "345";`

`string s3 = "abcd";`

`string s4 = "abcd---abcd"; int pos;`

`pos = s1.rfind(s2); // pos = 3`

`pos = s1.rfind(s2, 12); // pos = 3`

`pos = s1.rfind(s2, 3); // pos = 3`

`pos = s1.rfind(s2, 2); // pos = -1`

`pos = s2.rfind(s1); // pos = -1`

`pos = s1.rfind(s3, 0); // pos = -1`

// разница между функциями find() и rfind()

```
pos = s4.rfind(s3); // pos = 7
```

```
pos = s4.find(s3); // pos = 0
```

## Сравнение частей строк. Функция `compare()`

Чтобы сравнить две строки между собой можно использовать операцию `'= '`. Если две строки одинаковы, то результат сравнения будет `true`. В противном случае, результат сравнения будет `false`. Но если нужно сравнить часть одной строки с другой, то можно использовать функцию `compare()`.

Прототип функции compare():

```
int compare(size_type start, size_type num,  
const string &s) const;
```

где

s – строка, которая сравнивается с вызывающей строкой;

start – позиция (индекс) в строке s, из которой начинается просмотр символов строки для сравнения;

num – количество символов в строке s, которые сравниваются с вызывающей строкой.

Функция работает следующим образом.

Если вызывающая строка меньше строки  $s$ , то функция возвращает -1 (отрицательное значение).

Если вызывающая строка больше строки  $s$ , функция возвращает 1 (положительное значение).

Если две строки равны, функция возвращает 0.

Пример. Демонстрация работы функции compare():

```
// тип string, функция compare()
string s1 = "012345";
string s2 = "0123456789";
int res;

res = s1.compare(s2); // res = -1
res = s1.compare("33333"); // res = -1
res = s1.compare("012345"); // res = 0
res = s1.compare("345"); // res = -1
res = s1.compare(0, 5, s2); // res = -1
res = s2.compare(0, 5, s1); // res = -1
res = s1.compare(0, 5, "012345"); // res = -1
res = s2.compare(s1); // res = 1
res = s2.compare("456"); // res = -1
res = s2.compare("000000"); // res = 1
```

# Получение строки с символом конца строки '\0' (char \*). Функция c\_str()

Чтобы получить строку, которая заканчивается символом '\0' используется функция c\_str().

Прототип функции:

```
const char * c_str() const;
```

Функция объявлена с модификатором const. Это означает, что функция не может изменять вызывающий объект (строку).

Пример. Преобразование типа string в const char \*.

```
char s[10], *p; p=s;  
string a="welcome";  
strcpy(p, a.c_str());  
cout<<p;
```

Задача. Дана символьная строка. Удалить из неё все «лишние» (парные) пробелы, оставив по одному пробелу между словами.

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string s, blanc=" "; //два пробела
    int d;
    cout <<"Enter one string\n";
    getline(std::cin >> std::ws, s);
```



```
bool change=false;
while ((d=s.find( blanc))!=-1)
{
    change=true;
    s.erase (d,1);
}
if (change)
{cout<<"New string\n";
cout<<s;
}
else cout<<"No changes";
return 0;
}
```

## **resize**

`s.resize(n)` - Изменяет длину строки, новая длина строки становится равна `n`.

При этом длина строки может как уменьшится, так и увеличиться.

Если вызвать в виде `S.resize(n, c)`,

где `c` - символ,

то при увеличении длины строки добавляемые символы будут равны `c`.

## **clear**

`S.clear()` - очищает строчку, строка становится пустой.

Выделенная память не будет освобождена.

```
#include <iostream>
using namespace std;
#include <string>
int main()
{string s="Hello. I am string";
cout<<"\noriginal string "<<s<<endl;
s.resize(5);cout<<"Length of string =5. New
string is "<<s<<endl;
s.resize(8,'!');cout<<"Length of string =8.
We added ! Now string is "<<s<<endl;
s.clear();cout<<"Final string "<<s<<"is
empty"<<endl;
    return 0;}
```

```
original string Hello. I am string
Length of string =5. New string is Hello
Length of string =8. We added ! Now string is Hello!!!
Final string is empty
```

## **empty**

`S.empty()` - возвращает `true`, если строка пуста, `false` - если непуста.

## **push\_back**

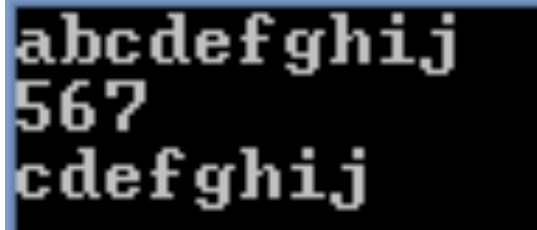
`S.push_back(c)` - добавляет в конец строки символ `c`, вызывается с одним параметром типа `char`.

## **substr**

`S.substr(pos)` - возвращает подстроку данной строки начиная с символа с индексом `pos` и до конца строки.

`S.substr(pos, count)` - возвращает подстроку данной строки начиная с символа с индексом `pos` количеством `count` или до конца строки, если `pos + count > S.size()`.

```
#include <string>
#include <iostream>
using namespace std;
int main()
{ string a = "0123456789abcdefghijklmnopqrstuvwxyz";
  string sub1 = a.substr(10);
  cout << sub1 << '\n';
  string sub2 = a.substr(5, 3);
  cout << sub2 << '\n';
  string sub3 = a.substr(12, 25);
  cout << sub3 << '\n'; return 0;
}
```



```
abcdefghijklmnopqrstuvwxyz
567
abcdefghijklmnopqrstuvwxyz
```

## **find\_first\_of**

Ищет в данной строке первое появление любого из символов данной строки `str`. Возвращается номер этого символа или значение `string::npos` (-1).

Если задано значение `pos`, то поиск начинается с позиции `pos`, то есть возвращаемое значение будет не меньше, чем `pos`. Если значение `pos` не указано, то считается, что оно равно 0 - поиск осуществляется с начала строки.

`S.find_first_of(str, pos = 0)` - искать первое вхождение любого символа строки `str` начиная с позиции `pos`. Если `pos` не задано - то начиная с начала строки `S`.

```
#include <string>
#include <iostream>
using namespace std;
int main(){
    string s = "0123456789abcdefghij",
    dig="0123456789", alf="abcd";    int pos;
    pos=s.find_first_of(dig);
    cout<<"pos of dig="<<pos<<endl;
    pos=s.find_first_of(alf);
    cout<<"pos of alf="<<pos<<endl;
    pos=s.find_first_of(dig,11);
    cout<<"pos of dig from 11="<<pos<<endl;
    return 0;
}
```

```
pos of dig=0
pos of alf=10
pos of dig from 11=-1
```

## **find\_last\_of**

Ищет в данной строке последнее появление любого из символов данной строки `str`. Способы вызова и возвращаемое значение аналогичны методу `find_first_of`.

## **find\_first\_not\_of**

Ищет в данной строке первое появление символа, отличного от символов строки `str`. Способы вызова и возвращаемое значение аналогичны методу `find_first_of`.

## **find\_last\_not\_of**

Ищет в данной строке последнее появление символа, отличного от символов строки `str`. Способы вызова и возвращаемое значение аналогичны методу `find_first_of`.



```
#include <string>
#include <iostream>
using namespace std;
int main()
{
    string s = "00hh!!!";
    int pos;
    pos=s.find_first_not_of('0');
    cout<<"pos of first not
'0'="<<pos<<endl;
    pos=s.find_last_not_of('!');
    cout<<"pos of las not '!' "<<pos<<endl;
    return 0;
}
```

```
pos of first not '0'=2
pos of las not '!' 3
```

## Оценивание результатов 4 модуля

- Лекции 1 раз в 2 недели (1 балл)
- Семинар 1 раз в неделю (1 балл)
- Лабораторные работы (5 баллов ,  
 $1.5+1.5+(0.5+1.5)$ ). Дедлайн на 4, 8, 12 занятия л.р.  
(считая обе подгруппы)
- Контрольная работа (3 балла).

Оценка за текущий контроль в 3 и 4 модуле учитывает результаты студента следующим образом.

Модуль 3.  $O_{\text{текущая } 3} = O_{\text{лекция}} + O_{\text{семинар}} + O_{\text{лаб. работа}} + O_{\text{ответы}} + O_{\text{контр. работа}}$

Модуль 4.  $O_{\text{текущая } 4} = O_{\text{лекция}} + O_{\text{семинар}} + O_{\text{лаб. работа}} + O_{\text{ответы}} + O_{\text{контр. работа}}$

Все оценки рассматриваются без округления.

Промежуточная оценка за 3 и 4 модуль вычисляется по формуле

$O_{\text{промежуточная } 3 \text{ и } 4} = 0,4 * O_{\text{текущая } 3} + 0,4 * O_{\text{текущая } 4} + 0,2 * O_{\text{экзамен } 4 \text{ модуль}}$ ,

где  $O_{\text{текущая } 3}$ ,  $O_{\text{текущая } 4}$  — оценки текущего контроля 3, 4 модуля, без округления.

Оценка экзамена округляется до ближайшего целого по правилам арифметики.

Окончательное округление производится после вычисления промежуточной оценки, по правилам арифметики.

Экзаменационная оценка не является блокирующей.  
Промежуточная оценка за 3 и 4 модуль не может превышать 10 баллов, в случае превышения ставится промежуточная оценка 10 баллов.

Результирующая оценка за дисциплину вычисляется по формуле

$$O_{\text{результирующая}} = 0,4 * O_{\text{промежуточная 1 и 2}} + 0,6 * O_{\text{промежуточная 3 и 4}}$$

Округление промежуточных и результирующей оценок производится по правилам арифметики.

В диплом выставляется результирующая оценка.

# Пример решения задачи лабораторной работы 9.

Дан массив символьных строк.

1. Выделить из каждой строки и напечатать подстроки, ограниченные с обеих сторон одной или несколькими точками.
2. Среди выделенных подстрок найти подстроку с максимальным числом заглавных русских букв.
3. В исходной строке, которой принадлежит найденная подстрока, вместо первой цифры вставить столько нулей, какова эта цифра.

Каждую часть оформить как отдельную подпрограмму.

# Например,

Mas1[0:k1-1]

..3 Поросёнка..или 4
....
И..серый...волк
Такая... сказка
...Знаете её...?.

Mas2[0:k2-1]

3 Поросёнка
серый
Знаете её
?

nom[0:k2-1]

0
2
4
4

Num=0

Преобразуем 0-ю строку  
исходного массива

..000 Поросёнка..или 4
------------------------

```

#include <iostream>
using namespace std;
#include <string>
#include <cctype>
int substr(int k1,string mas1[],
string mas2[],int nom[])
{int  n, i, j, j1, k2=0;
  for( i=0; i<k1;++i)
  { j = 0;
    n = mas1[i].length();
    j1 = -1;
    while (j < n )
      if (mas1[i][j] == '.')
      { if (j1 != -1)
        {
          mas2[k2].assign(mas1[i], j1, j-j1);
          nom[k2++]=i;
        }
        while (mas1[i][j] == '.' && j < n )
          j ++;
      }
  }
}

```

```

j1 = j;

    }
    else
        j ++;
}
return k2;
}
int nompodstr(int k2, string mas2 [])
{string  rus = "ЁЙЦУКЕНГШЩЗХЪФЫВАПРОЛДЖЭЯЧСМИТЬБЮ";
int  i, j, max=0, maxi= -1, num;

for( i=0; i<k2;++i)
{ num = 0;
  for( j=0; j<mas2[i].length();++j)
    if (rus.find(mas2[i][j])!=-1)  num++;
  if (num > max)
  {
    max = num;          maxi = i;
  }
}
return maxi;
}

```



```
bool change(string &s)
{bool f;
  int i, j, k, L;
  f = false;
  i = 0;
  L = s.length();
  while(i < L && ! (isdigit(s[i]))) i++;
  if (i < L && isdigit(s[i]))
  {
    f = true;
    k = s[i] - '0';
    s.resize( L + k - 1);
    for (j = L-1; j >= i + 1; --j) s[j + k - 1] =
s[j];
    for (j = i; j < i+k; j++) s[j] = '0';
  }
  return f;
}
```

```
int main()
{int num,i,k1,k2;
string s;
do
    {cout << "Enter number of lines: ";
        cin >> k1;
        if (cin.fail())      cin.clear();
        getline(cin,s) ;
    }
while (k1<=0);
int nom[k1]; string mas1[k1],mas2[5*k1];
cout<<"Enter "<<k1<<" strings"<<endl;
for( i=0; i<k1;++i)
    getline(cin,mas1[i]);
k2=substr(k1, mas1,  mas2, nom);
if (k2 ==0 )
    cout<<"No substrings"<<endl;
else
    {
        cout<<"Substrings"<<endl;
        for( i=0; i<k2;++i)
            cout<<mas2[i]<<endl;
        num=nompodstr(k2, mas2);
    }
```

```
    if (num==-1)
        cout<<"No necessary substrings"<<endl;
    else
    {
        cout<<"Necessary substring
"<<mas2[num]<<endl;
        if (change(mas1[nom[num]]))
            cout<<"Changed string"<<
mas1[nom[num]] ;
        else
            cout<<"No changes in string"<<endl;
    }
}
return 0;
}
```

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

int main()
{
    puts("System default locale\n");
    for (int i=128; i<256; i++) printf("%c
%d | ",i,i);
    printf("\n\n\n");
    setlocale(LC_ALL,".1251");
    puts("Locale 1251\n");
    for(int i=128; i<256; i++)
    {
        if (i==149) printf(" ");
        printf("%c %d | ",i,i);
    }
    return 0;
}
```

## System default locale

А	128	:	Б	129	:	В	130	:	Г	131	:	Д	132	:	Е	133	:	Ж	134	:	З	135	:	И	136	:	Й	137	:
К	138	:	Л	139	:	М	140	:	Н	141	:	О	142	:	П	143	:	Р	144	:	С	145	:	Т	146	:	У	147	:
Ф	148	:	Х	149	:	Ц	150	:	Ч	151	:	Ш	152	:	Щ	153	:	Ъ	154	:	Ы	155	:	Ь	156	:	Э	157	:
Ю	158	:	Я	159	:	а	160	:	б	161	:	в	162	:	г	163	:	д	164	:	е	165	:	ж	166	:	з	167	:
и	168	:	й	169	:	к	170	:	л	171	:	м	172	:	н	173	:	о	174	:	п	175	:	р	176	:	с	177	:
и	178	:	й	179	:	к	180	:	л	181	:	м	182	:	н	183	:	о	184	:	п	185	:	р	186	:	с	187	:
и	188	:	й	189	:	к	190	:	л	191	:	м	192	:	н	193	:	о	194	:	п	195	:	р	196	:	с	197	:
и	198	:	й	199	:	к	200	:	л	201	:	м	202	:	н	203	:	о	204	:	п	205	:	р	206	:	с	207	:
и	208	:	й	209	:	к	210	:	л	211	:	м	212	:	н	213	:	о	214	:	п	215	:	р	216	:	с	217	:
и	218	:	й	219	:	к	220	:	л	221	:	м	222	:	н	223	:	о	224	:	п	225	:	р	226	:	с	227	:
и	228	:	й	229	:	к	230	:	л	231	:	м	232	:	н	233	:	о	234	:	п	235	:	р	236	:	с	237	:
и	238	:	й	239	:	к	240	:	л	241	:	м	242	:	н	243	:	о	244	:	п	245	:	р	246	:	с	247	:
и	248	:	й	249	:	к	250	:	л	251	:	м	252	:	н	253	:	о	254	:	п	255	:	р	256	:	с	257	:

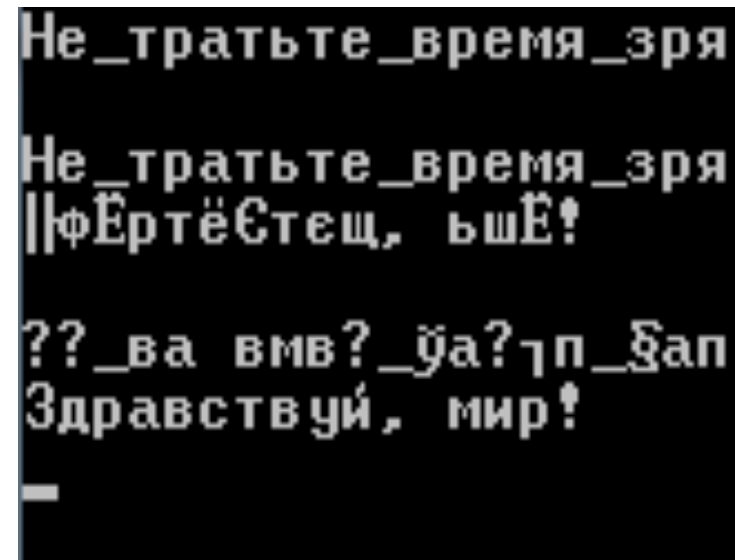
## Locale 1251

?	128	:	?	129	:	'	130	:	?	131	:	"	132	:	:	133	:	+	134	:	+	135	:	?	136	:	%	137	:
?	138	:	<	139	:	?	140	:	?	141	:	?	142	:	?	143	:	?	144	:	?	145	:	?	146	:	"	147	:
"	148	:	<	149	:	?	150	:	?	151	:	?	152	:	?	153	:	?	154	:	?	155	:	?	156	:	?	157	:
?	158	:	?	159	:	?	160	:	?	161	:	?	162	:	?	163	:	?	164	:	?	165	:	?	166	:	?	167	:
Е	168	:	с	169	:	Є	170	:	<	171	:	?	172	:	?	173	:	Р	174	:	?	175	:	?	176	:	+	177	:
?	178	:	?	179	:	?	180	:	ч	181	:	Ч	182	:	?	183	:	ё	184	:	№	185	:	е	186	:	>	187	:
?	188	:	?	189	:	?	190	:	й	191	:	А	192	:	Б	193	:	В	194	:	Г	195	:	Д	196	:	Е	197	:
Ж	198	:	З	199	:	И	200	:	Й	201	:	К	202	:	Л	203	:	М	204	:	Н	205	:	О	206	:	П	207	:
Р	208	:	С	209	:	Т	210	:	У	211	:	Ф	212	:	Х	213	:	Ц	214	:	Ч	215	:	Ш	216	:	Щ	217	:
Ъ	218	:	Ы	219	:	Ь	220	:	Э	221	:	Ю	222	:	Я	223	:	а	224	:	б	225	:	в	226	:	г	227	:
д	228	:	е	229	:	ж	230	:	з	231	:	и	232	:	й	233	:	к	234	:	л	235	:	м	236	:	н	237	:
о	238	:	п	239	:	р	240	:	с	241	:	т	242	:	у	243	:	ф	244	:	х	245	:	ц	246	:	ч	247	:
ш	248	:	щ	249	:	ь	250	:	ы	251	:	ь	252	:	э	253	:	ю	254	:	я	255	:			:			:

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

int main ()
{
char sInput[200] = "Здравствуй, мир!";
char sOutput[200];

cin >> sOutput;
cout << endl << sOutput << endl;
cout << sInput << endl;
    setlocale(LC_ALL, ".1251"); // ru - russian
cout << endl << sOutput << endl;
cout << sInput << endl;
return 0;
}
```



```
Не_тратьте_время_зря
Не_тратьте_время_зря
||фЁртёЕтещ, ьшЁ!
??_ва вmv?_ўа?п_Зап
Здравствуй, мир!
_
```

В дальнейшем рекомендуется для вывода сообщений использовать английский язык.

```
#include <iostream>
#include <windows.h>
using namespace std;

int main ()
{
    setlocale(LC_ALL, ".1251");
    //для использования кириллицы (из библиотеки <iostream>)
    SetConsoleCP(1251);
    /для потокового вывода с использованием русского языка
    SetConsoleOutputCP(1251);
    char sInput[200] = "Здравствуй, мир!";
    char sOutput[200];

    cin >> sOutput;
    cout << endl << sOutput << endl;
    cout << sInput << endl;
    return 0;
}
```

Необходимо выбрать шрифт Lucida console

# Работа с файлами в C++

Файл – именованный набор байтов, который может быть сохранен на некотором накопителе.

Полное имя файлов – это полный адрес к директории файла с указанием имени файла, например:

D:\docs\file.txt.

В C++ есть 3 основных класса файлового ввода/вывода:

**ifstream** (производный от **istream**), **ofstream** (производный от **ostream**) и **fstream** (производный от **iostream**). Эти классы выполняют файловый ввод, вывод и ввод/вывод соответственно. Чтобы использовать классы файлового ввода/вывода, вам необходимо включить заголовочный файл **fstream**.



В отличие от потоков `cout`, `cin`, `cerr` которые уже готовы к использованию, файловые потоки должны быть настроены программистом явно. Чтобы открыть файл для чтения и/или записи, просто создайте экземпляр объекта соответствующего класса файлового ввода/вывода с именем файла в качестве параметра. Затем используйте операторы вставки и извлечения для записи или чтения данных из файла. Для завершения работы с файлом его необходимо закрыть - вызвать функцию `close()`.

# Явное открытие файлов с помощью `open()`

Файловый поток можно явно открыть с помощью `open()` и явно закрыть его с помощью `close()`.

`open()` принимает имя файла и необязательный параметр режима открытия файла.

Например:

```
std::ofstream outf( "Sample.txt" );  
//Для вывода  
outf << "This is line 1" << '\n';  
outf << "This is line 2" << '\n';  
outf.close(); // явно закрываем файл  
  
// добавим ещё одну строку  
outf.open("Sample.dat", std::ios::app);  
//пояснения режима далее  
outf << "This is line 3\n";  
outf.close();
```

```
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
    // ofstream используется для записи в файл
    // создадим в текущей директории файл
    //Sample.txt
    ofstream outf;
    outf.open ("Sample.txt") ;
    //или ofstream outf("Sample.txt") ;
    // если файловый поток для записи не открыт
    if (!outf)
    {
        // Выводим сообщение об ошибке и выходим
        cerr << "Oh no, Sample.dat could not be
opened for writing!" << endl;
        return 1;
    }
}
```

```
// Записываем данные в файл
```

```
    outf << "This is line 1" << '\n';  
    outf << "This is line 2" << '\n';  
    outf.close();  
    return 0;
```

```
}
```

## Ввод из файла

Теперь возьмем файл, который мы написали в предыдущем примере, и прочитаем его с диска. Обратите внимание, что `ifstream` возвращает 0, если мы достигли конца файла (EOF, «end of the file»). Мы воспользуемся этим фактом, чтобы определить, когда чтение будет завершено. Добавляем приведенный ниже фрагмент кода в программу (после закрытия файла).

```
// ifstream используется для чтения файлов
// Мы будем читать из файла с именем Sample.dat
ifstream inf("Sample.txt" );
    // Если мы не смогли открыть входной
//файловый поток для чтения
    if (!inf)
    { // сообщаем об ошибке и выходим
        cerr << "Oh no, Sample.txt could not be
opened for reading!" << std::endl;
        return 1;
    }
    // Пока конец файла не достигнут
    while (inf)
    { // считываем информацию из файла inf в
//строку и распечатываем ее
        string strInput;
        inf >> strInput;
        cout << strInput << '\n';
    }
inf.close();
```

Эта программа дает следующий результат:

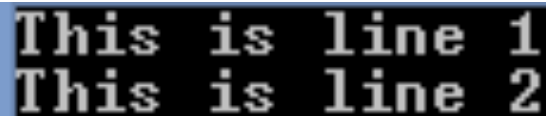
```
This  
is  
line  
1  
This  
is  
line  
2
```

оператор извлечения разделяет входные строки пробелами.  
Чтобы читать строки полностью, нам нужно использовать функцию `getline()`.

Изменим цикл, который читает данные из файла

```
while (inf)
{
    // считываем информацию из файла в
    // строку и распечатываем ее
    std::string strInput;
    std::getline(inf, strInput);
    std::cout << strInput << '\n';
}
```

Теперь программа дает другой результат:

A screenshot of a terminal window with a black background and white text. It displays two lines of output: "This is line 1" followed by "This is line 2".

```
This is line 1
This is line 2
```



Для проверки того, что файл успешно открыт, также можно использовать метод `is_open()` :

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
    ifstream file ("d:\\file.txt");
    if (file.is_open()) // вызов метода is_open()
        cout << " All right\n" << endl;
    else
    {
        cout << "File was not opened!\n\n" << endl;
        return -1;
    }
    file.close();
    return 0;}
```

## Буферизованный вывод

Вывод в C++ может буферизоваться. Это означает, что всё, что выводится в файловый поток, может сразу не записываться на диск. Вместо этого несколько операций вывода могут быть объединены и обработаны вместе. Это сделано в первую очередь из соображений производительности. Когда буфер записывается на диск, это называется очисткой/сбросом буфера (англоязычный термин «flushing»). Один из способов вызвать сброс буфера – закрыть файл: содержимое буфера будет сброшено на диск, а затем файл будет закрыт.

Буферизация при неосторожности может вызвать сложности. Главная проблема в этом случае – когда в буфере есть данные, а программа немедленно завершается (либо из-за сбоя, либо из-за вызова `exit()`).

В этих случаях файлы не закрываются, что означает, что буферы не сбрасываются. В этом случае данные в буфере не записываются на диск и теряются навсегда. Следует явно закрывать все открытые файлы перед вызовом `exit()`.

Можно очистить буфер вручную, используя функцию `ostream::flush()` или отправив `std::flush` в выходной поток. Любой из этих методов обеспечит немедленную запись содержимого буфера на диск.

`std::endl`; также очищает выходной поток. Следовательно, чрезмерное использование `std::endl` может повлиять на производительность при выполнении буферизованного ввода/вывода, когда операции очистки дороги (например, запись в файл). Программисты, заботящиеся о производительности, для вставки новой строки в выходной поток часто используют `'\n'` вместо `std::endl`, чтобы избежать ненужной очистки буфера.

## Режимы открытия файлов

Что произойдет, если мы попытаемся записать данные в уже существующий файл? Повторный запуск примера вывода в файл показывает, что при каждом запуске программы исходный файл полностью перезаписывается. Что, если вместо этого мы захотим добавить еще немного данных в конец файла? Конструкторы файловых потоков принимают необязательный второй параметр, который позволяет указать, как следует открывать файл. Этот параметр называется **режимом**, и допустимые флаги, которые он принимает, находятся в классе `ios`.

# Режимы открытия файла

Режим открытия файла ios	Значение
app	Открывает файл в режиме добавления
ate	Ищет конец файла перед чтением/записью
binary	Открывает файл в двоичном режиме (вместо текстового режима)
in	Открывает файл в режиме чтения (по умолчанию для ifstream)
out	Открывает файл в режиме записи (по умолчанию для ofstream)
trunc	Стирает файл, если он уже существует

Режимы открытия файлов можно комбинировать с помощью поразрядной логической операции или `|`, например:  
`ios_base::out | ios_base::trunc` — открытие файла для записи, предварительно очистив его.

Объекты класса `ofstream`, при связке с файлами по умолчанию содержат режимы открытия файлов `ios_base::out | ios_base::trunc`. То есть файл будет создан, если не существует. Если же файл существует, то его содержимое будет удалено, а сам файл будет готов к записи. Объекты класса `ifstream` связываясь с файлом, имеют по умолчанию режим открытия файла `ios_base::in` — файл открыт только для чтения. Режим открытия файла ещё называют — флаг, для удобочитаемости в дальнейшем будем использовать именно этот термин.

Обратите внимание на то, что флаги `ate` и `app` по описанию очень похожи, они оба перемещают указатель в конец файла, но флаг `app` позволяет производить запись, только в конец файла, а флаг `ate` просто переставляет флаг в конец файла и не ограничивает места записи.

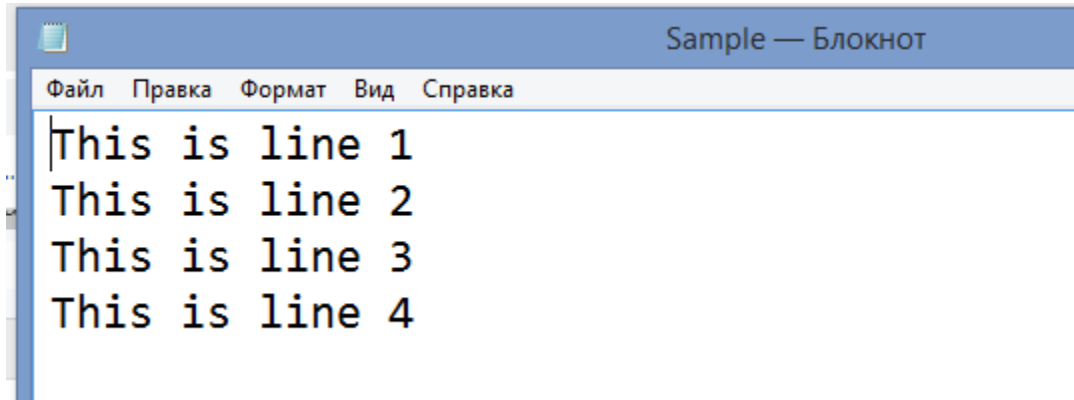
```
#include <iostream>
#include <fstream>
int main()
{
    // Мы передадим флаг ios::app, чтобы
    // сообщить ofstream о необходимости добавления
    // вместо того, чтобы перезаписывать файл.
    // Нам не нужно передавать std::ios::out,
    // поскольку std::ios::out для ofstream
    // используется по умолчанию
    std::ofstream outf( "Sample.txt",
std::ios::app );
    // Если мы не смогли открыть выходной
    // файловый поток для записи
    if (!outf)
    {
        // Сообщаем об ошибке и выходим
        std::cerr << "Oh no, Sample.txt could
not be opened for writing!\n";
        return 1;    }
```



```
outf << "This is line 3" << '\n';  
outf << "This is line 4" << '\n';  
outf.close();  
return 0;
```

```
}
```

Теперь в файле четыре строки



**Пример 1.** Сформировать файл, содержащий вещественные числа. Признак окончания ввода данных – буква. Затем открыть файл и подсчитать количество чисел в файле.

```
#include <fstream>
#include <iostream>
#include<string>
using namespace std;
int main()
{string filename;
//имя файла, включая путь, вводит пользователь
cout<<"Input file name :";
getline(cin,filename);
int i, n;
double a;
ofstream f; //для вывода
f.open(filename);
cout<<"Input numbers, end of input is letter :\n";
do
{ //признак окончания ввода - буква
cout<<"a=";
cin>>a;
if(!cin.fail()) //если ввод удался
f<<" "<<a;
}
while(!cin.fail());
cin.clear();
f.close();
```

```
fstream F; //открываем на чтение или запись
F.open(filename);
if (!F)
    cout<<"File not found";
else
{
    n=0;
    while (!F.eof())
    {
        F>>a; //читаем число
        cout<<a<<"\t"; //выводим на экран
        n++; //увеличиваем количество чисел
    }
    F.close(); //закрываем файл
    cout<<"n="<<n<<endl;
}

return 0;
}
```

## Метод write

Используется в бинарных файлах для записи блока памяти (массива байт) в файл как они есть. Любая переменная так же является массивом байт, вернее ее так можно рассматривать. Соответственно этот метод запишет в файл ее машинное представление (тот вид как она выглядит в памяти).

Этот метод принимает два параметра: **указатель на блок данных** и **количество байт**, который этот блок занимает. В примере строка занимает `strlen()` байт, целое `sizeof()` (которое даст 4 на 32-х битных операционных системах для целого и 8 для вещественного).

В отличии от форматированного вывода оператором `<<`, метод `write()` не выводит данные в текстовом представлении.

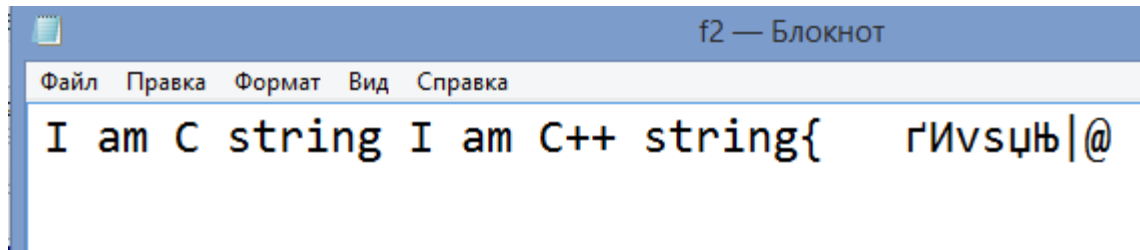
```
#include <fstream>
#include <iostream>
#include<string>
using namespace std;
int main()
{string filename;
cout<<"Input file name for output :";
getline(cin,filename);
ofstream fileo;
fileo.open(filename, ios::binary);

    // Запись побайтно
    // строки в стиле C
    char sc1[] = "I am C string\n";
    fileo.write(sc1,sizeof(sc1));
```

```
// строки в стиле C++
string sc2 = "I am C++ string";
fileo.write(&sc2[0],sc2.length());

// целого в машинном представлении
int k = 123;
fileo.write((char*)&k,sizeof(k));

// вещественного в машинном представлении
double dd = 456.789;
fileo.write((char*)&dd,sizeof(dd));
fileo.close();
return 0;
}
```



## Метод read

Используется в бинарных файлах для чтения блока памяти (массива байт). Имеет те же параметры, что и метод write

## Метод seekg

Производит установку текущей позиции в нужную, указываемую числом. В этот метод также передается способ позиционирования.

- `ios_base::end` – Отсчитать новую позицию с конца файла.
- `ios_base::beg` – Отсчитать новую позицию с начала файла (абсолютное позиционирование).
- `ios_base::cur` – Перескочить на n байт начиная от текущей позиции в файле (по умолчанию).



```
file.seekg(0,ios_base::end) ;  
//Стать в конец файла  
file.seekg(10,ios_base::end) ;  
//Стать на 10 байтов с конца  
file.seekg(30,ios_base::beg) ;  
//Стать на 30-й байт  
file.seekg(3,ios_base::cur) ;  
//перепрыгнуть через 3 байта  
file.seekg(3) ;  
//перепрыгнуть через 3 байта - аналогично
```

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    char PATH[] = "f:\\f.txt";
    ofstream f1(PATH); //для записи
    for (int i = 5; i < 15; i++) {
        f1.write((char*)&i, sizeof(int));
    } //записываем числа от 5 до 14
    f1.close();
    int value = 0;
    ifstream f2(PATH); //для чтения
    f2.seekg(5 * sizeof(int), ios::beg);
    f2.read((char*)&value, sizeof(int));
    cout <<"value " << value << '\n';
    //результат 10
    f2.close(); return 0;
}
```

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    char PATH[] = "f:\\\\let.txt";
    ofstream f1(PATH);
    for (char i = 'A'; i <= 'Z'; i++) {
        f1 << i;
    } //буквы от 'A' до 'Z'
    f1.close();
    char value = 0;
    ifstream f2(PATH);
    f2.seekg(5 * sizeof(char), ios::beg);
    f2 >> value;
    cout <<"letter " << value;        //result  F
    f2.close();
    return 0;
}
```

# Динамическое распределение памяти C++

## Операторы new и delete

**new** – оператор языка C++ выделяющий участок памяти под переменную или объект класса. Возвращает указатель на выделенную область памяти с типом, заданным в параметре оператора.

```
//myVar – указатель на переменную типа <type>
```

```
myVar1 = new <type1>(<value>) //число равное <value>
```

```
myVar2 = new <type2>[<size>] //массив размера <size>
```

```
int arraySize[5]={3,6,2,1,4};
```

```
int *myArray[5];
```

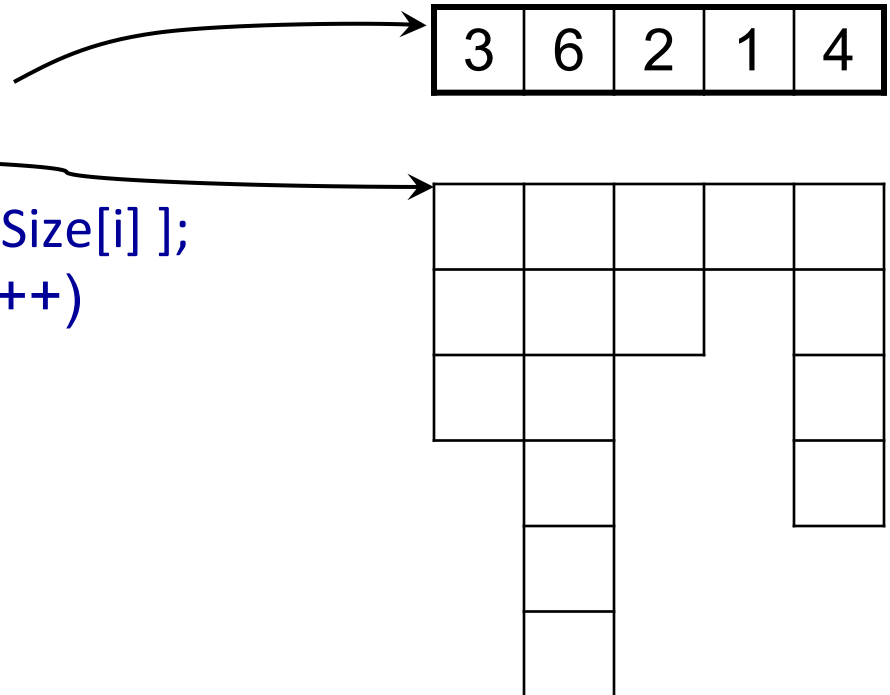
```
for(int i=0; i<5; i++){
```

```
myArray[i] = new int [ arraySize[i] ];
```

```
for(int j=0; j<arraySize[i]; j++)
```

```
cin >> myArray[i][j];
```

}



# Динамическое распределение памяти C++

## Операторы new и delete

**delete** – оператор языка C++, освобождает память выделенную под переменную или объект класса, в качестве параметра требует указатель на переменную или объект, которые необходимо удалить.

**Например:**

```
int *a;  
a = new int(18);  
delete a;
```

Если память была выделена под массив, тогда перед указателем необходимо поставить квадратные скобки [ ].

**Например:**

```
for(int i=0; i<5; i++)  
    if (arraySize[i]==0) //те массивы, размер которых стал нулевым  
        delete [ ] myArray[i]; //очистка памяти от массивов
```

```
int *d, a=20;  
int c[10]; /* при такой записи в скобках можно ставить только константу */  
d=new int[a*2]; /*создание динамического массива из 2*a элементов */
```

При выделении памяти для многомерного массива все измерения, кроме первого, должны быть константными выражениями.

```
int *e = *new int[a][5]; //двумерный массив a строк на 5 столбцов  
int *w = new int; //выделение памяти для переменной w (указатель)  
float *h = new float(25.804);  
/* в круглых скобках задаётся значение новой переменной, однако задать  
множество значений для массива таким способом нельзя*/  
int *v = new int(); /* пустыми круглыми скобками тоже можно задать  
нулевое начальное значение для переменной. */  
delete h; //освобождение памяти от переменной h  
delete d; //освобождение памяти от динамического массива //некорректно  
delete [ ] e; //правильно освобождение памяти от массива e
```

Операторы **new** и **delete** корректно работают только друг с другом. Выделение (удаление) памяти другими способами одновременно с ними приведёт к непредсказуемым действиями программы.

**Так делать нельзя!!!**

```
int *h = new int;  
free(h);
```

```
int* buffer = (int*) malloc(1);  
delete buffer;
```

//верные варианты

```
int* buffer = (int*) malloc(1);  
free(buffer); //C
```

```
int *h = new int;  
delete h; //C++
```

**new** и **delete** можно использовать только в C++.

В C использование этих функций невозможно, так как они описаны с использованием синтаксиса C++.

# Динамические структуры данных

Динамические структуры данных - структуры, размер которых в программе явно не определяется. Память под элементы, входящие в эти структуры, выделяется в процессе работы программы.

Пример динамических структур – связанные списки.

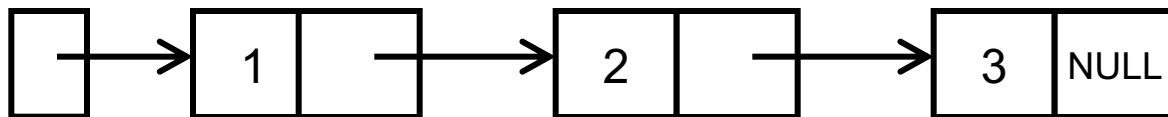
Основные типы списков уже изучались в языке С.



Пусть исходные данные для создания списка – числа 1, 2, 3 и 0 (0-признак окончания ввода данных).

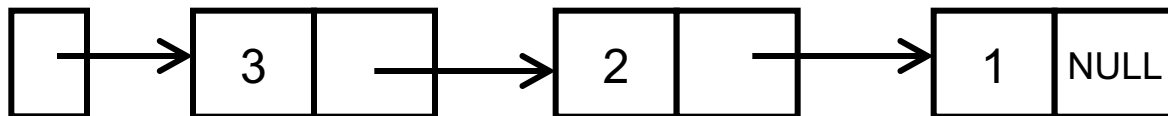
## 1. Линейные однонаправленные списки

### 1. Очередь



lst

### 2. Стек

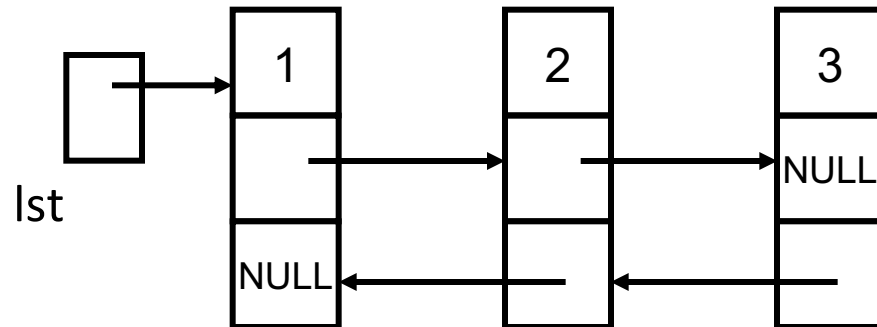


lst

Элемент двунаправленного списка содержит указатель не только на следующий, но и на предыдущий элемент списка.

## 2. Линейные двунаправленные списки

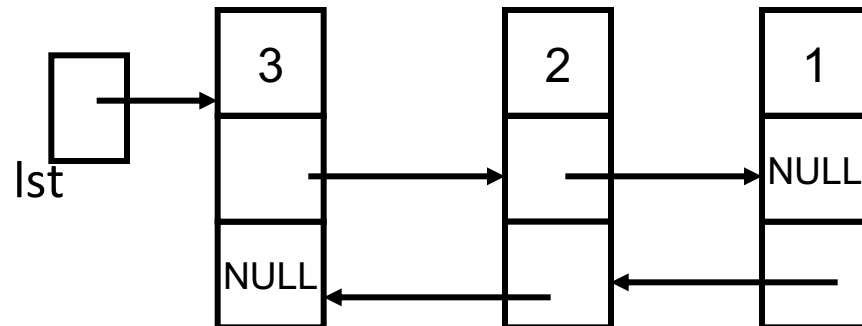
### 1. Очередь



поле данных  
указатель на след.

указатель на пред.

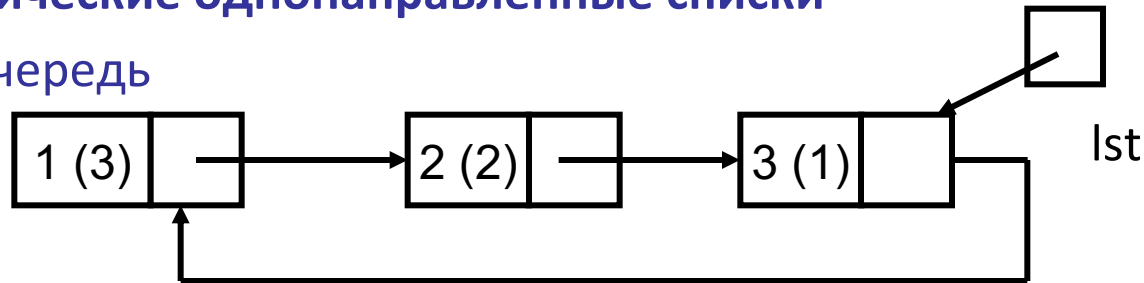
### 2. Стек



В циклическом списке последняя запись указывает на первую. Для циклического списка удобно возвращать адрес последней записи.

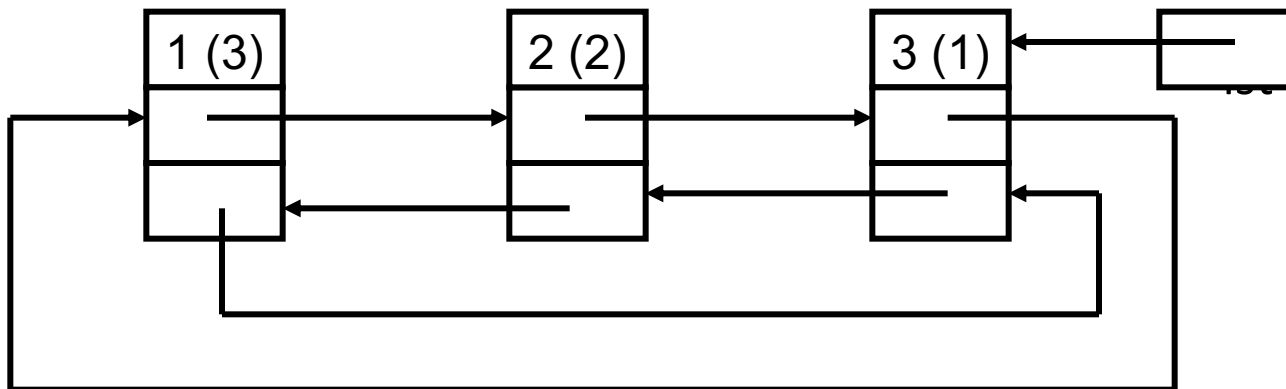
### 3. Циклические однонаправленные списки

#### 1. Очередь



2. **Стек** отличается от очереди порядком расположения полей данных. Порядок данных для стека указан на предыдущем рис. в скобках.

#### 4. Циклические двунаправленные списки – очередь (стек).



**Пример 1:** Написать программу, которая вводит целые положительные числа в стек, а затем вычисляет их сумму.

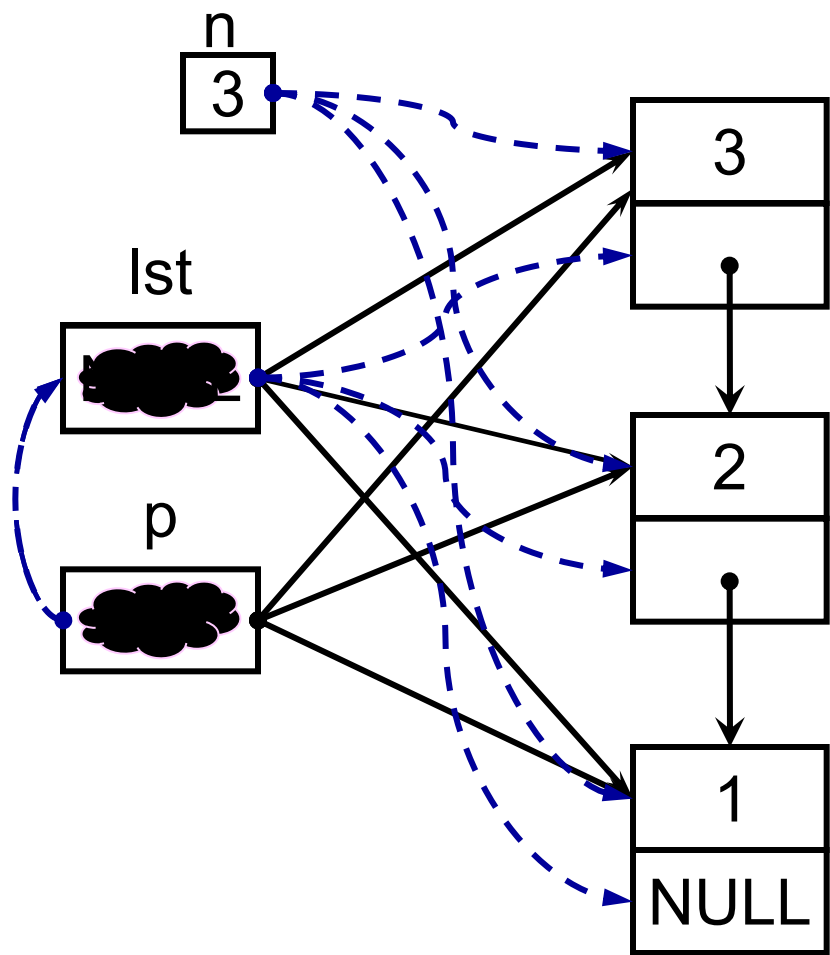
```
#include <iostream>
using namespace std; struct node
{node *next; int info; };
```

```
node *stack()
```

```
//формирование стека
```

```
{int n=1;
node *p, *lst; /* p - указатель на
текущую добавляемую запись,
lst - указатель на предыдущую
запись (на вершину стека) */
//В C++ аналог NULL nullptr
```

```
lst=NULL; //пустой стек
cout<<"Enter positive integers\n";
while(n>0)
//ввод до не числа или n<=0
{cin>>n;
if(!cin.fail()&& n>0)
{
p= new node;
p->info=n;
p->next=lst;
lst=p;
}
}
return(lst);/*вернет указатель на
вершину стека*/
}
```



```
int sumspisok (node *lst)
{
int sum=0;
node *p; //указатель на текущую запись
```

```

p=lst;
while(p!=NULL) //лучше while(p)
{
    sum+=p->info;
    p=p->next;
}
return(sum);
}
```

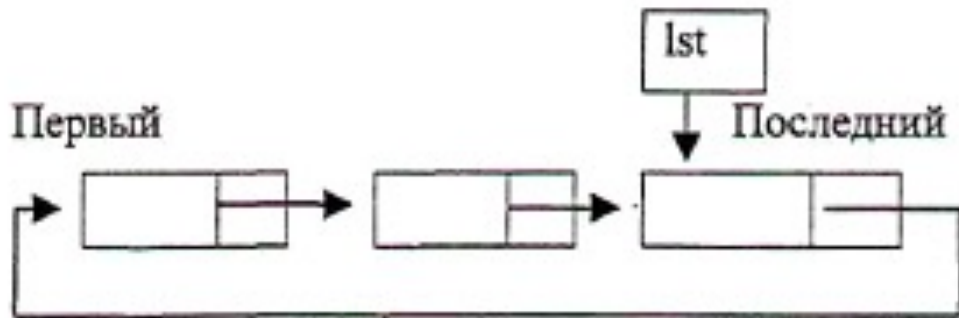
//освобождение памяти

```
void free_memory(node *lst)
{
    node *now=lst, *next=lst;
    while (next)
    {next=now->next;
    delete(now);
    now=next;
    }
    cout<<"\nNow memory is free\n";}

int main()
{node*lst=stack();
if(!lst) cout<<"List is empty\n";
else
{
    cout<<" summ "<< sumspisok(lst) <<endl; ;
    free_memory(lst);
}
return 0;}
```

# Циклический список

В циклическом списке поле указателя последнего элемента содержит указатель на первый элемент:

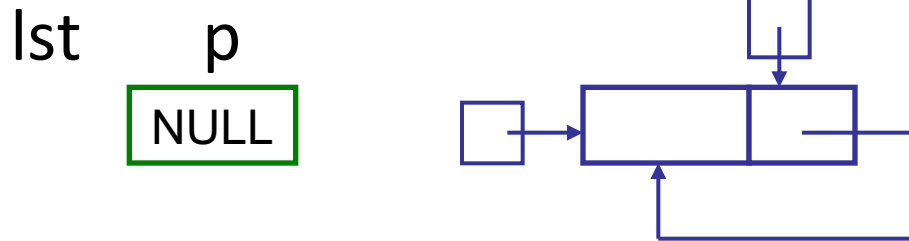


Указатель `lst` используется для доступа к циклическому списку.

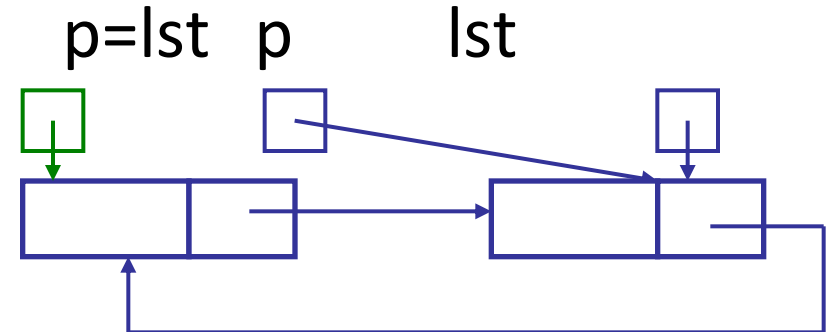
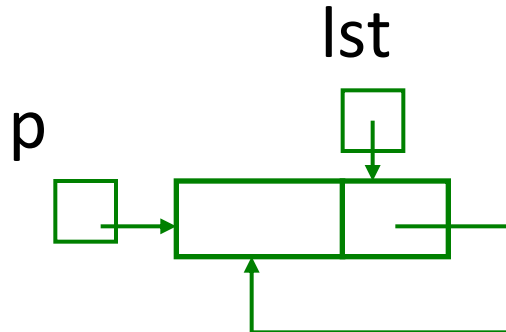
Обычно такой список определяется с помощью указателя на последний элемент. Если список пуст, то `lst=NULL`. Циклический список может быть использован для реализации стека и очереди.



Пустая очередь:



Очередь не пуста:



Новый элемент добавляется после последнего, так как это очередь.

Подпрограмма занесения значения X в очередь на базе циклического списка.

Подпрограмма занесения значения X в очередь на базе циклического списка.

```
node *add_elem(node *lst, int x)
{
    node *p;
    p= new node;
    p->info=x;
    if (lst==NULL) p->next=p; //очередь пуста
    else //очередь не пуста
    {
        p->next=lst->next;
        lst->next=p;
    }
    lst=p; return(lst);
}
```

```

void printspisok_1(node *lst) //вывод списка на экран
{
    node *p; p=lst;
    do{p=p->next;
        cout<<p->info<<" ";
    }
    while(p!=lst);
    cout<<endl;
}

void free_memory_1(node *lst) //освобождение памяти
{
    node *curr, *pred; pred=lst->next;
    if (lst==lst->next) delete (lst); //единственный элемент
    else
    {
        do
        {
            {curr=pred->next;      delete (pred);
              pred=curr;
            }
            while(pred!=lst);
            delete (lst); //последний оставшийся элемент
        }
        puts("\nNow memory is free");
    }
}

```

```
int main()
{node *lst=NULL;
    int n=1;

    cout<<"Enter positive integers \n";
    while(n>0)
    {cin>>n;
        if(!cin.fail()&& n>0)
            lst=add_elem(lst,n);
    }
    if(lst) //список создан
    { printspisok_1(lst);
        free_mamory_1(lst);
    }
    else
        cout<<"list is empty";
    return 0;
}
```

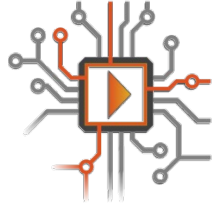
# Двунаправленные связанные списки

Каждый элемент содержит два указателя: на следующий и на предыдущий элемент.

Такие списки могут быть использованы для организации очереди и стека.

**Пример :** Занесение X в стек на базе линейного двунаправленного связанного списка.

```
#include <iostream>
using namespace std;
struct node2
{
    node2 *next, *prev;
    int info;
};
```



```
node2 *addstack (node2 *lst, int x)
```

```
{
```

```
node2 *p = new node2;
```

```
p->info=x;
```

```
if(lst!=NULL)
```

```
lst->prev=p; /* новая запись размещается перед той, на которую  
указывал lst */
```

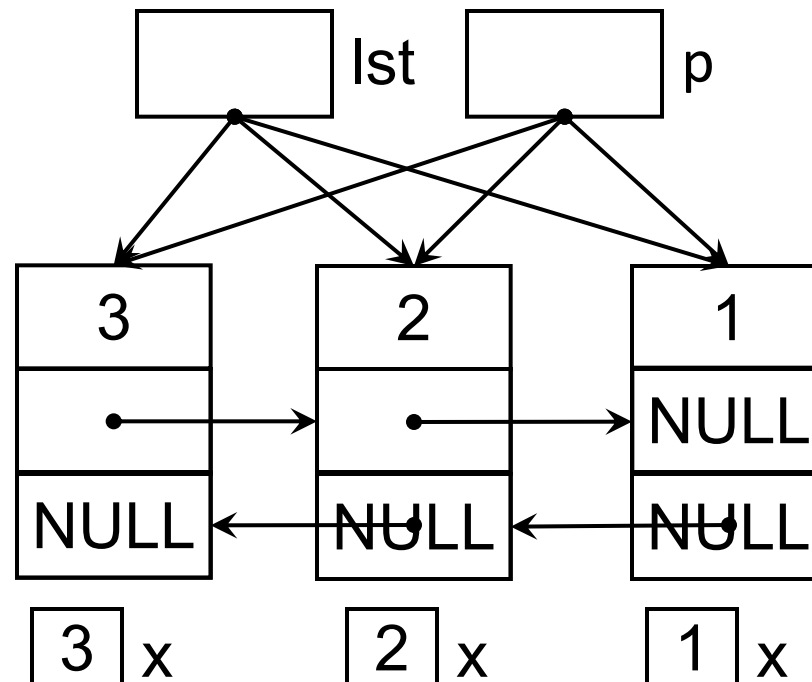
```
p->next=lst;
```

```
p->prev=NULL;
```

```
lst=p;
```

```
return (lst) ;
```

```
}
```



```
void printlist_2(node2 *lst)
{ node2 *p,*t;

p=lst;
cout<<"forward";
while(p)
{
    cout<<"    "<<p->info);
    t=p;p=p->next;
}
cout<<"\nback";
p=t;
while(p)
{
    cout<<"    "<<p->info);
    p=p->prev;
}
cout<<endl;
}
```

void free\_memory2(node2 \*lst) //аналогично односвязному списку

```
{  node2 *now=lst, *next=lst;
  while (next)
  {next=now->next;
   delete(now);
   now=next;
  }
  cout<<"\nNow memory is free";
}
```

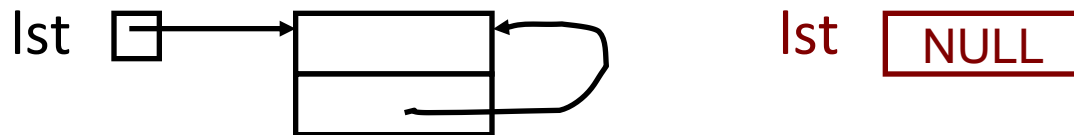
int main()

```
{  node2 *lst=NULL;  int n=1;
  cout<<"Enter positive integers \n";
  while(n>0)
  {cin>>n;
   if(!cin.fail()&& n>0)    lst=addstack(lst,n);}
  if (lst)
  {  printlist_2(lst);    free_memory2(lst);  }
  else cout<<" No list";
  return 0;}
```

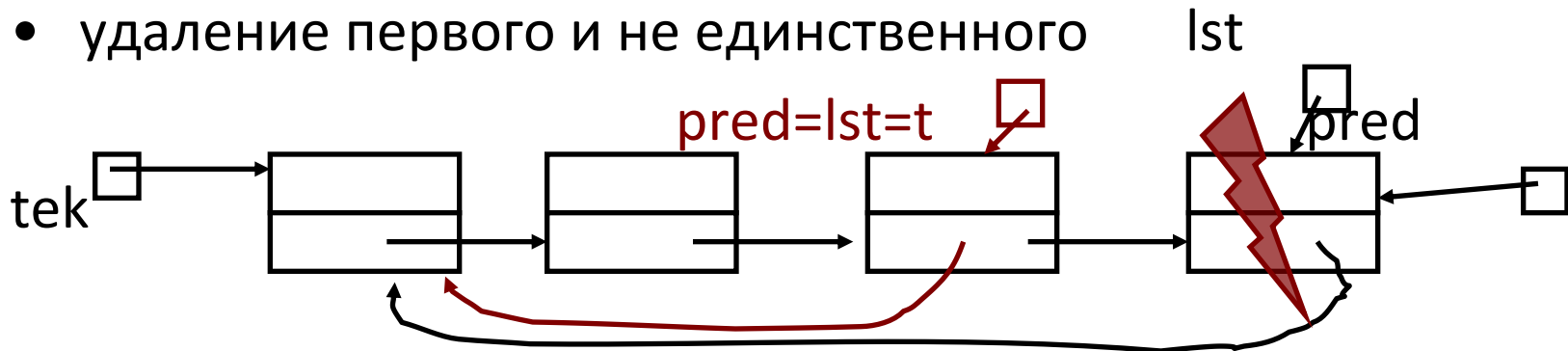


**Задача.** Написать\_функцию для удаления элементов меньших чем X1 из циклического однонаправленного списка. Используем указатели: *lst* – указатель на последний элемент, *pred* – предшествующий, *next* – следующий, *t* – новый последний элемент списка. Рассмотрим случаи:

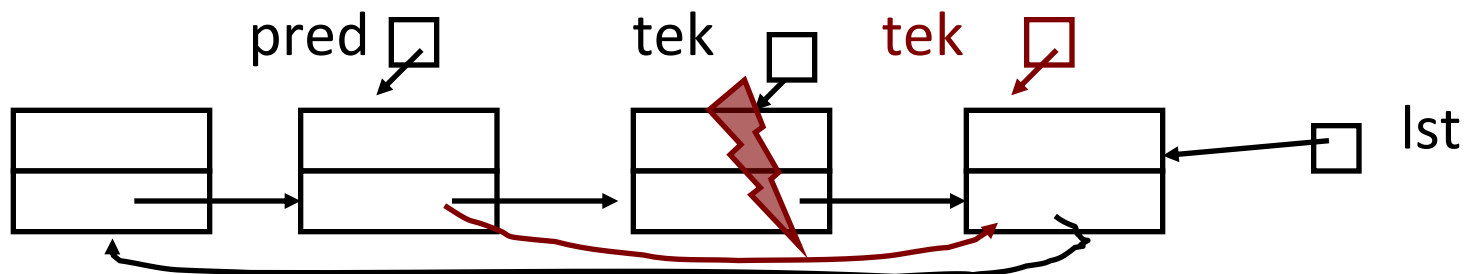
- удаление первого и единственного



- удаление первого и не единственного



- удаление элемента из середины списка



```
node * del (node *lst, int x1)
{
node *tek,*pred, *t;
tek=lst->next; //указатель на текущий элемент
pred=lst; //указатель на предыдущий элемент
while (lst&&lst->info<x1) //пока удаляется первый элемент
    if (lst->next==lst) //если элемент первый и единственный
    {
        delete(lst); lst=NULL;
    }
else //если первый и не единственный
    { t=lst; //поиск нового lst – он будет перед исходным lst
do
    t=t->next;
while (t->next!=lst);
t->next=pred->next;
delete(lst);
lst=t;
pred=t;
    }
```

```
if (lst!=NULL)
    //если список еще не удален полностью
    do
        if (tek->info<x1)
            { //удаление элемента из середины списка
                pred->next=tek->next;
                delete(tek);
                tek=pred->next;
            }
        else //движение по списку
            pred=tek,tek=tek->next;
    while (tek!=lst);

return(lst);
}
```

# Бинарные деревья

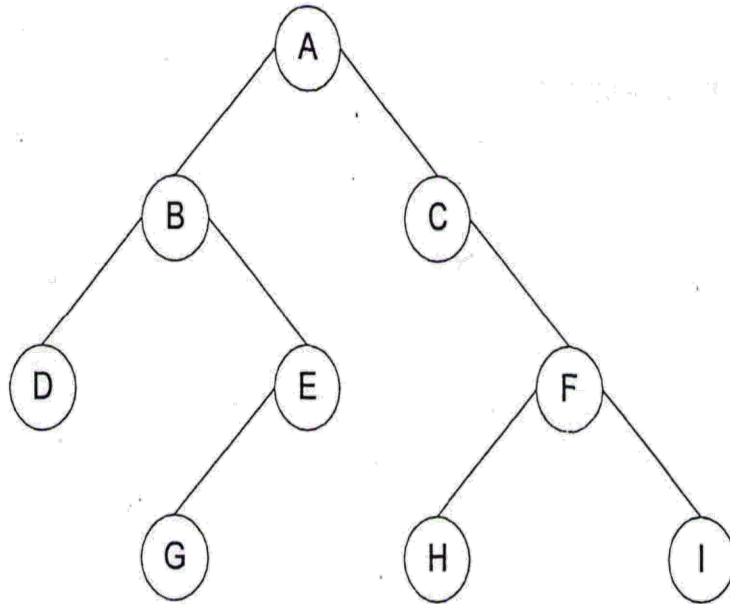
**Бинарное дерево** - конечное множество элементов, которое либо пусто, либо содержит один элемент, называемый **корнем** дерева. Остальные элементы множества делятся на два непересекающихся подмножества, каждое из которых является бинарным деревом. Эти подмножества называются **левым и правым поддеревьями** исходного дерева. Каждый элемент бинарного дерева называется **узлом**.

Если  $X$  - корень бинарного дерева и  $Y$  - корень его левого или правого поддерева, то говорят, что  $X$  - **отец**  $Y$ ,  $Y$  - левый или правый **сын**  $X$ .

Узлы, не имеющие сыновей, называются **листьями**.

Два узла называются **братьями**, если они сыновья одного отца.

# Пример бинарного дерева:



В дереве девять узлов.

*A* - корень дерева.

*B* - корень левого поддерева

*C* - корень правого поддерева.

Левое поддерево бинарного дерева с корнем *C* пусто.

*A* - отец для *B* и *C*.

*B*-левый сын *A*.

*E* - отец *G*.

*G* - левый сын правого сына *B*.

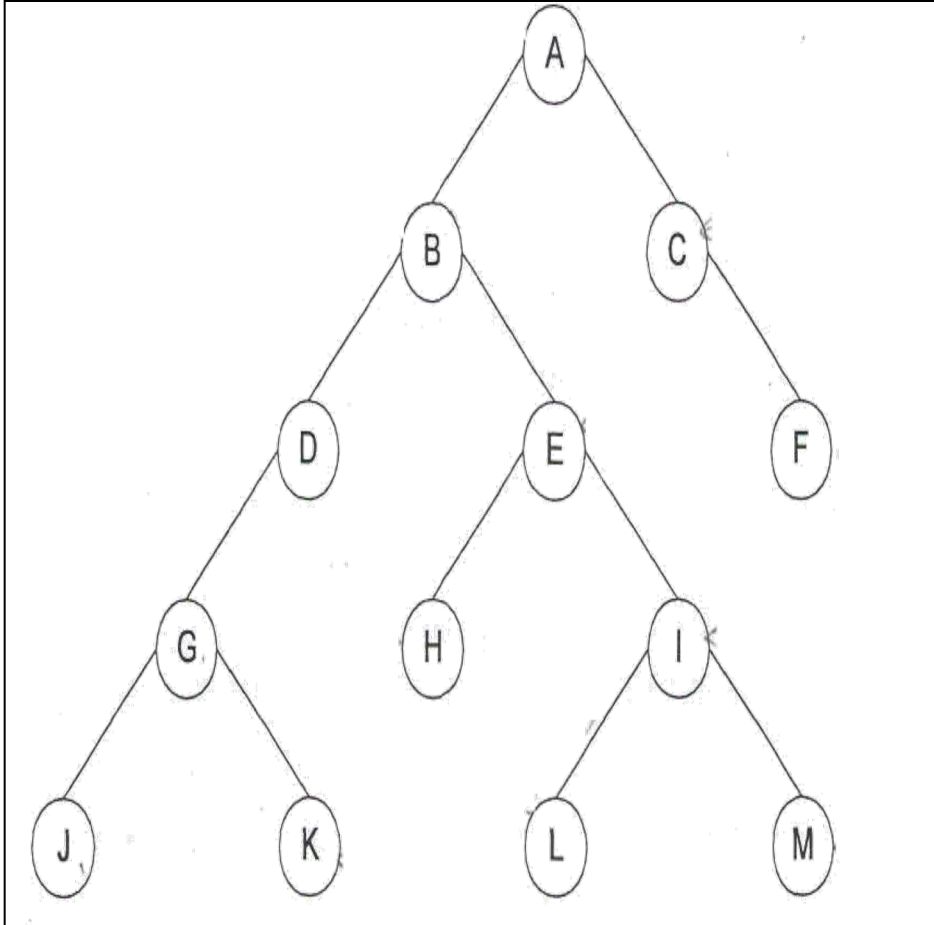
Листья: *D, G, H, I*.

*B, C* – братья.

# Прохождение бинарного дерева

Обычно дерево сначала строится, а затем обходится. Как отметить каждый узел один раз? В линейном списке элементы просматриваются от начала до конца. Для узлов дерева не существует такого естественного порядка. Рассмотрим три метода прохождения, которые определяются рекурсивно.

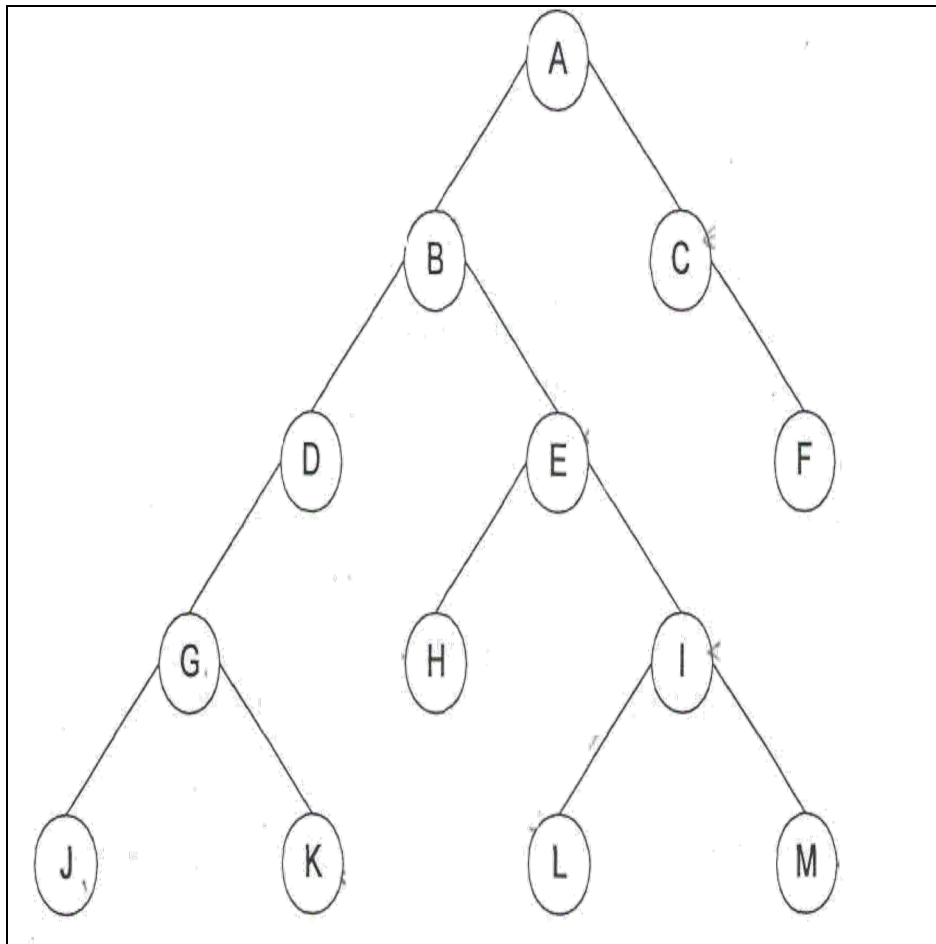
# Прохождение в прямом порядке



- Обработать корень.
- Пройти в прямом порядке левое поддерево.
- Пройти в прямом порядке правое поддерево.

Для представленного дерева получим следующий порядок узлов: ABDGJKENILMCF

# Прохождение в обратном порядке

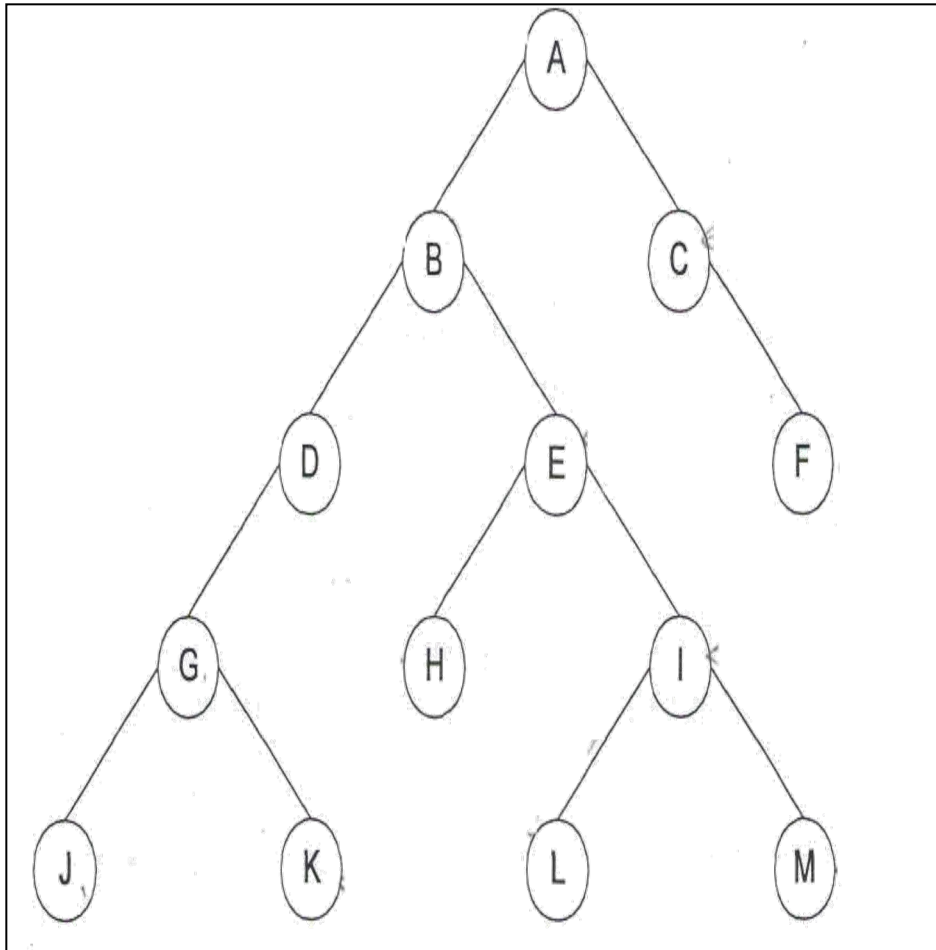


- Пройти в обратном порядке левое поддерево.
- Пройти в обратном порядке правое поддерево.
- Обработать корень.

Получим:  
JKGDHLMIEBFCSA



# Прохождение в симметричном порядке



- Пройти в симметричном порядке левое поддерево.
- Обработать корень.
- Пройти в симметричном порядке правое поддерево.

Порядок узлов:  
JGKDBHNELIMACF

# Свойства рекурсивных алгоритмов

**Наличие тривиального случая.** Правильный рекурсивный алгоритм не должен создавать бесконечную последовательность вызовов самого себя. Для этого обязательно должен содержаться нерекурсивный выход (тривиальный случай).

**Примеры:**  $n=0$  в алгоритме вычисления  $n!$ ,  $p=NULL$  в алгоритме прохождения дерева.

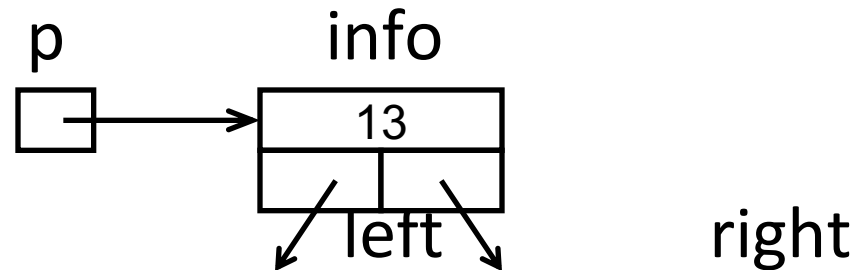
**2. Определение сложного случая в терминах простого.** При любых исходных данных нерекурсивный выход должен достигаться за конечное число рекурсивных вызовов. Для этого каждый новый вызов рекурсивного алгоритма должен решать более простую задачу.

**Примеры:** уменьшение  $n$  в задаче вычисления факториала; уменьшение размера бинарного дерева в задаче его прохождения.

# Операции над бинарными деревьями

Узел бинарного дерева содержит поля: info, left, right.

Пусть p - указатель на узел бинарного дерева



Тогда:

`p->info` - содержимое узла с указателем `p`;

`p->left` -указатель на левого сына узла с указателем `p`;

`p->right`- указатель на правого сына узла с указателем `p`.

При создании бинарного дерева используются следующие операции:

- **p=newn(x)** - создание нового бинарного дерева, состоящего из одного узла с информационным полем x; p - указатель на этот узел.
- **setleft(p,x)** - создание нового левого сына x для узла с указателем p.
- **setright(p,x)** - создание нового правого сына для узла с указателем p.

```
#include <iostream>
using namespace std;
struct node
{int info;
 node *left,*right;
};
//новый узел с числом p
node *newn(int p)
{
 node *uk = new node;
 uk->info=p;
 uk->left=uk->right=NULL;
 return(uk);
}
```

```
//присоединение нового узла
// с числом x к узлу p слева
void setleft(node*p, int x)
{p->left=newn(x);
}
void setright(node*p, int x)
{p->right=newn(x);
}
node *form()
{int n;
 node *der=NULL,*next,*tek;

cout<<"Input integer
      numbers\nletter means that
      input is over\n";
cin>>n;
if(!cin.fail())
{der=newn(n);
```

```

cin>>n;
while(!cin.fail())
{next=tek=der;
while (next!=NULL)
{
tek=next;
if (n<tek->info)
next=tek->left;
else next=tek->right;
}
if (n<tek->info) setleft(tek,n);
else setright(tek,n);
cin>>n;
}
}
return der;
}
//вывод дерева на экран
void print_tree(node *der, int h)
{ if (der)

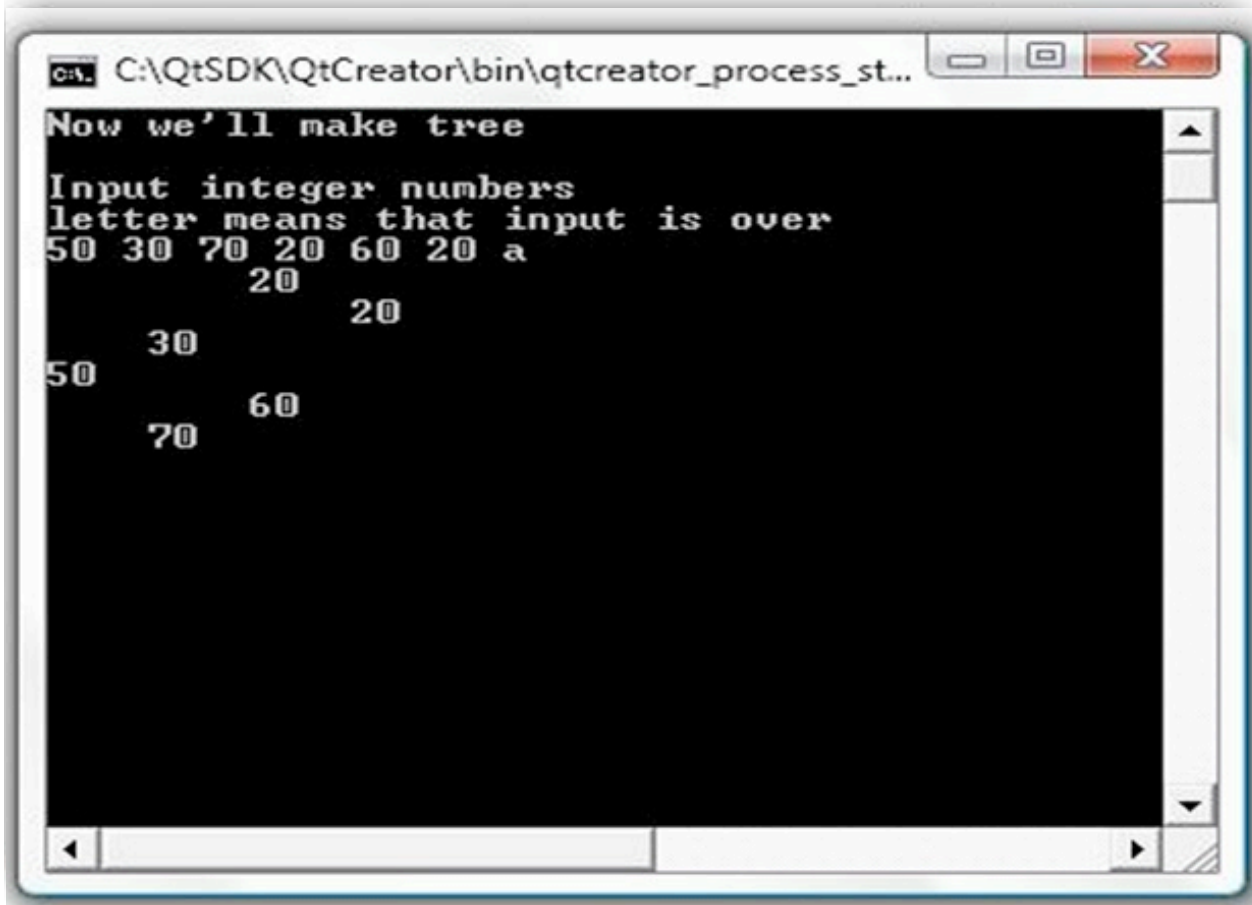
```

```

{
print_tree(der->left,h+1);
for(int i=0; i<h;i++) cout <<"  ";
cout << der->info << endl;
print_tree(der->right,h+1);
}
}
int main()
{
node *der;
cout<<"Now we'll make tree\n";
der=form();
if (!der)
cout<<"Tree is empty";
else
print_tree(der,0);
return 0;
}

```

Результат вывода дерева на экран. Числа располагаются слева направо.



The screenshot shows a console window with the following text and a binary tree diagram:

```
Now we'll make tree
Input integer numbers
letter means that input is over
50 30 70 20 60 20 a
```

The tree structure is visualized as follows:

```

      20
     /  \
    30   20
   /  \
  50   60
 /  \
30  70
```

The tree is a binary search tree. The root node is 20. Its left child is 30, and its right child is 20. The left child 30 has a left child 50 and a right child 60. The node 50 has a left child 30 and a right child 70. The input sequence 50 30 70 20 60 20 a results in this tree structure.

Рассмотрим задачи обработки бинарного дерева.

1. Посчитать число элементов (узлов) в бинарном дереве.

```
int NumEl (node *der) //прямой обход
{ if (der==NULL) return (0) ;
return (NumEl (der->left)+NumEl (der->right)+1) ; }
```

Вызов:

```
int k;
//считаем, что эта переменная везде определена
k= NumEl (der) ;
cout<<"number of elements is equal "<<k;
```

2. Посчитать число листьев в бинарном дереве.

```
int NumLeaf (node *der)
{if (der==NULL) return (0) ;
if (der->left==NULL&&der->right==NULL) return (1) ;
return (NumLeaf (der->left)+NumLeaf (der->right)) ;
}
```

Вызов:

```
k=NumLeaf (der) ;
cout<<"number of leafs is equal "<<k;
```



3. Посчитать число однодетных отцов.

```
int Num1 (node *der)
{ int k=0;

  if (der==NULL) return (0) ;
      if ( (der->left==NULL && der->right!=NULL) ||
          (der->left!= NULL && der->right==NULL) )
k=1;
      return (k+Num1 (der-> left) +Num1 (der->
right) ) ;
}
```

Вызов:

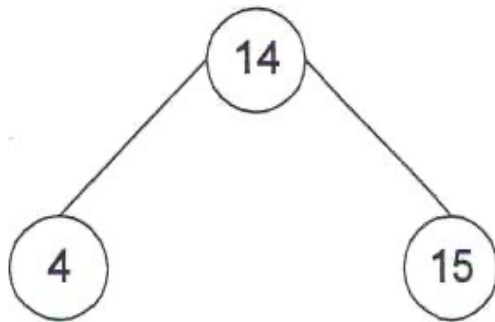
```
k=Num1 (der) ;

cout<<"number of one-child fathers is equal
"<<k;
```

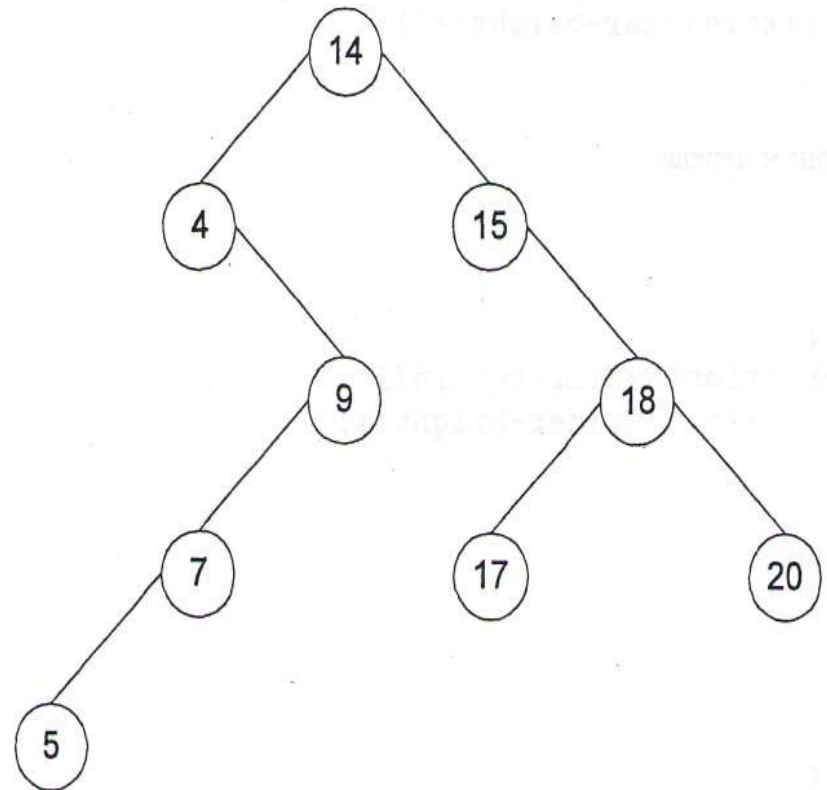
## 5. Посчитать глубину бинарного дерева.

- Нет дерева - глубина=0.
- Для дерева, состоящего из одного узла глубина=0.

Глубина=1



- Глубина=4



```
/* объявление глобальных переменных, чтобы  
не копировать их при каждом вызове */
```

```
int gl, gr, gmax;
```

```
int depth(node *der)
```

```
{ //тривиальный случай
```

```
if (der==NULL) return (0) ;
```

```
//второй случай
```

```
if (der->left==NULL&&der->right==NULL)  
return (0) ;
```

```
gl=depth(der->left) ;
```

```
gr=depth(der->right) ;
```

```
gmax= (gl>gr) ? gl : gr ;
```

```
return (gmax+1) ;
```

```
}
```

5. Удалить листья из бинарного дерева.

```
void DelLeaf(node **der) //прямой обход
{ /*указатель на указатель, т.к. корень может
меняться*/
    if (*der==NULL) return; //тривиальный случай
    if ((*der)->left==NULL&&(*der)->right==NULL)
    { //узел является листом
        cout<<"Leaf was deleted " <<
            (*der)->info<<endl;
        delete(*der);
        *der=NULL;
        return;
    }
    DelLeaf(&((*der)->left));
    DelLeaf(&((*der)->right));
} //рекурсия для левого и правого поддерев
```

Вызов:

```
DelLeaf(&der);
cout<<"the tree after deleting of the leafs\n";
if(der)print_tree(der, 0);
else cout<<"empty";
```

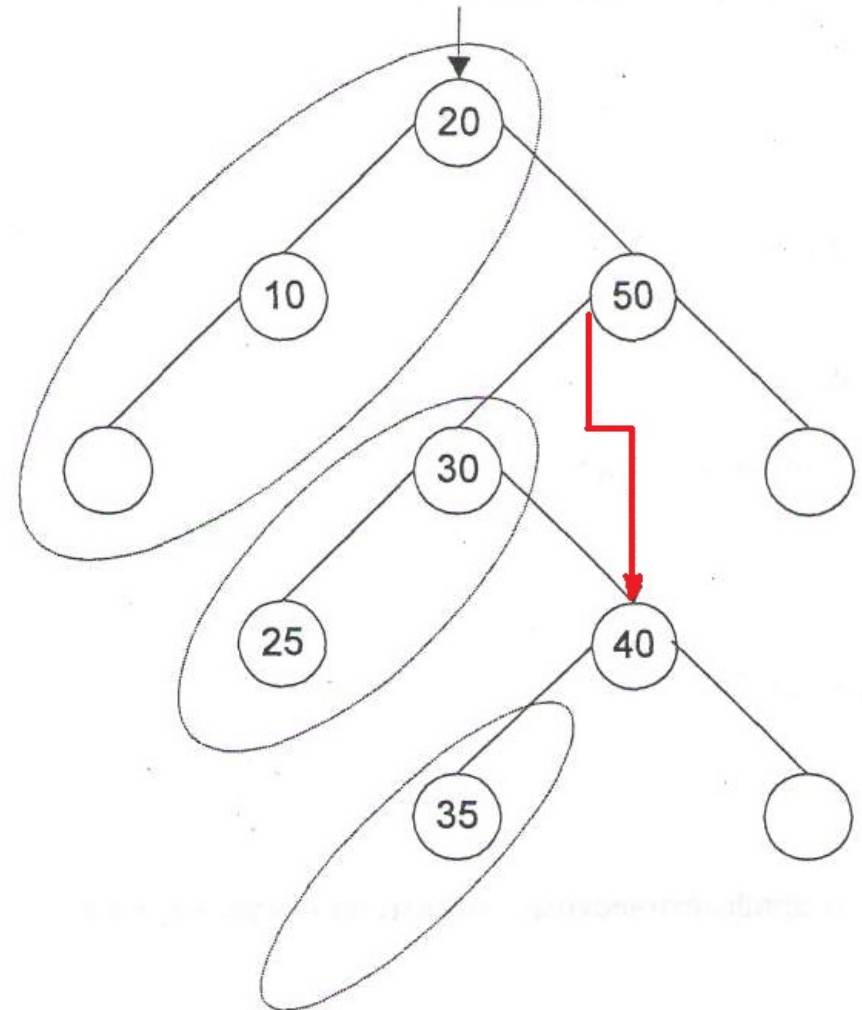
6. Поиск максимума в бинарном дереве.

```
int searchmax(node *der, int max)
{
    if(!der)
        return max;
    else
    {
        if(der->info>max) max=der->info;
        max=searchmax(der->left, max);
        //можно опустить
        max=searchmax(der->right, max);
        return max;
    }
} //вызов
if(der){int max=der->info;
max=searchmax(der, max);
cout<< " max= " << max);} else cout<<"tree is
empty";
```

7. Удалить из бинарного дерева все числа меньше X.

Пусть  $X=40$ .

Тогда для представленного бинарного дерева с корнем 20 придется удалить корень и левое поддерево, а затем числа из правого поддерева.



```
//прототип функции удаления поддерева
void udalder(node *der);
void udalx(int x, node **ader)
//удаление чисел меньших x
{node *der=*ader; //корень исходного дерева
if(der==NULL) return;
if(der->info<x)
{ //оборвать правую ветвь и удалить левое поддерево
    *ader=der->right; //новый корень
    der->right=NULL;
    udalder(der); //удаление левого поддерева
    udalx(x,ader); }
//удаление узлов из правого поддерева
else if(der->info>x)
    udalx(x,&der->left); /* передаем адрес
    указателя на левого потомка в корне дерева */
```

```
else //корень=x
```

```
{udalder(der->left); der->left=NULL; }
```

```
}
```

```
//удаление поддерева
```

```
void udalder(node *der)
```

```
{
```

```
if(der==NULL) return;
```

```
//удаление левого поддерева
```

```
udalder(der->left);
```

```
//удаление правого поддерева
```

```
udalder(der->right) ;
```

```
//удаление корня в последнюю очередь
```

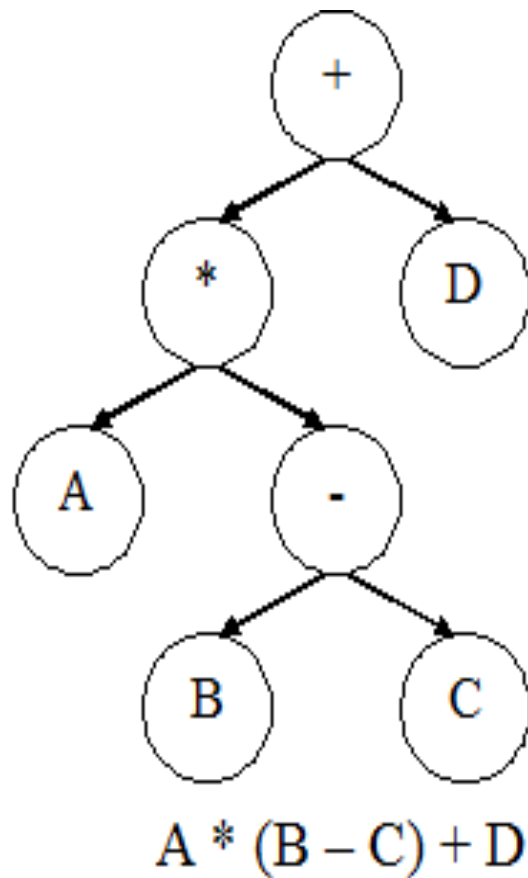
```
delete(der);
```

```
}
```

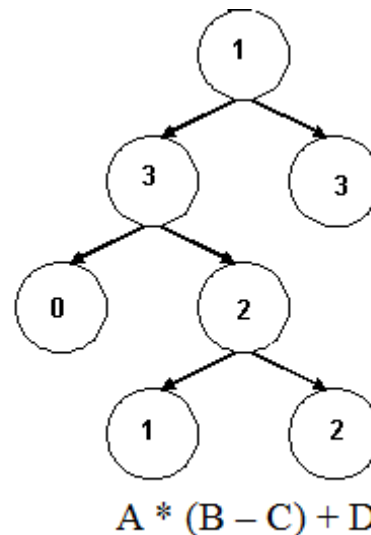


**Задача.** Написать рекурсивную подпрограмму для вычисления значения арифметического выражения, представленного в виде бинарного дерева

Информационное поле каждого узла содержит число – номер оператора или операнда. Значения операндов хранятся в массиве num.



A	B	C	D
---	---	---	---



**Номера операторов:**

1.+

2.-

3.\*

4./

Алгоритм count(root)

нач

если root->left=NULL то  
count:=num[root->info]

иначе

//Вычислить левое поддерево

oper1:=count( root->left)

//Вычислить правое поддерево

oper2:=count( root->right)

//Вычислить оператор

op:=root->info

/\* Вычислить результат  
применения оператора\*/

выбор (op)

1: count:= oper1+oper2;

2: count:= oper1- oper2;

3: count:= oper1\*oper2;

4: count:= oper1/oper2;

квыбор

все

кон

## Определение интервалов. Часы

Для работы со временем, начиная со стандарта C++11, появилась библиотека времени – chrono. До выхода C++11 была только одна библиотека работающая со временем – это Си-библиотека ctime. Она доступна и сейчас. Библиотеку chrono составляют три основных типа:

- Интервалы (duration)
- Моменты (time\_point)
- Часы (clock)

Используем часы system\_clock – системные часы реального времени.

Используется три метода:

- `now()` – текущий момент времени
- `to_time_t()` – преобразует момент времени в тип `time_t`
- `from_time_t()` – преобразует тип `time_t` в момент времени системных часов

**`time_t`** – арифметический тип, представляет собой целочисленное значение – число секунд, прошедших с 00:00, 1 января 1970. Определим время работы определенного фрагмента программы. Вычисляем первый момент времени до исследуемого фрагмента кода, а второй после. Вычисляем разницу обоих моментов и можем оценить скорость работы программы.

```
#include <iostream>
#include <chrono>
using namespace std;
using namespace chrono;
int main() {
    // Получаем момент времени_1
    system_clock::time_point start =
system_clock::now();
    // Выполняем некоторый код
    for (long i = 1; i < 10000000000; i += 1);
    // Получаем момент времени_2
    system_clock::time_point end =
system_clock::now();
    /* Определяем тип объекта интервала и
вычисляем его значение */
    duration<double> sec = end - start;
```

```
// вычисляем количество тактов в интервале
// и выводим итог
cout << sec.count() << " сек." << endl;
return 0;
}
```

Чтобы вывести итоговое значение нам нужно знать количество тактов. За такое вычисление берется метод `count()`, который и позволяет вывести окончательный результат.

Если нам нужно определить интервалы с очень большой точностью, `chrono` позволяет использовать на выводе различные единицы – от часов до наносекунд (в зависимости от величины интервала).

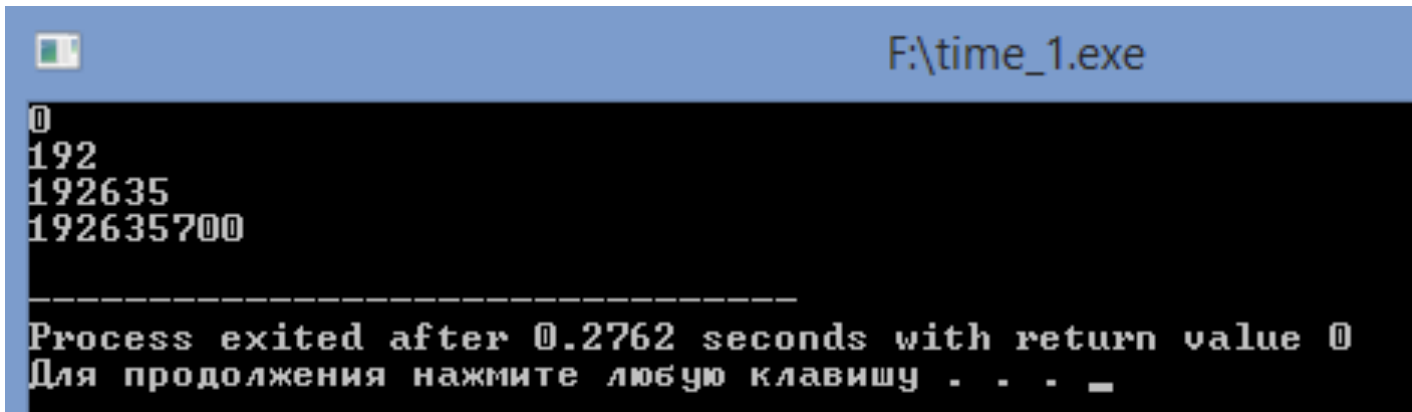
Для изменения единиц времени используется формат преобразования в стиле C++ `duration_cast<>`, которому, в качестве шаблонного параметра, необходимо передать один из возможных типов перечисленных ниже:

- `std::chrono::nanoseconds`
- `std::chrono::microseconds`
- `std::chrono::milliseconds`
- `std::chrono::seconds`
- `std::chrono::minutes`
- `std::chrono::hours`

ИЗМЕНИМ ВЫВОД В КОНЦЕ ПРОГРАММЫ

```
cout << duration_cast<seconds>(end -  
start).count() << "\n"  
    << duration_cast<milliseconds>(end -  
start).count() << "\n"  
    << duration_cast<microseconds>(end -  
start).count() << "\n"  
    << duration_cast<nanoseconds>(end -  
start).count() << "\n";
```

Результат



```
0  
192  
192635  
192635700  
  
-----  
Process exited after 0.2762 seconds with return value 0  
Для продолжения нажмите любую клавишу . . . _
```

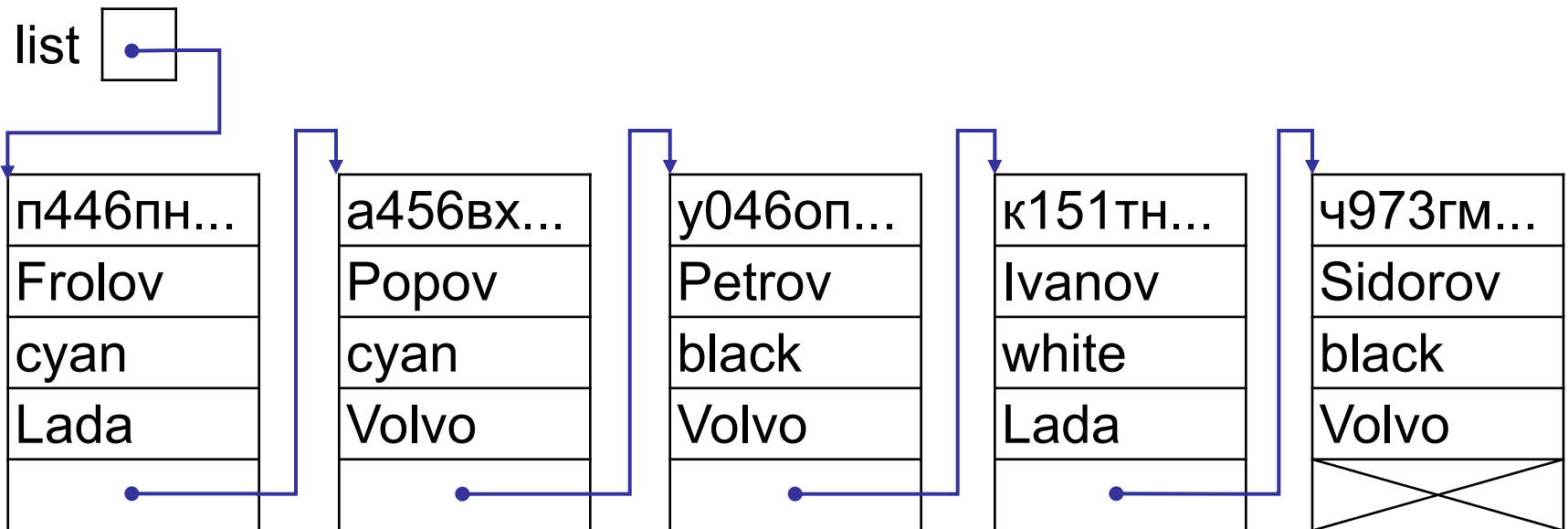


# Мультиписки

Рассмотрим множество автомобилей в городе. О каждой машине имеется следующая информация:

- марка;
- цвет;
- номер;
- владелец.

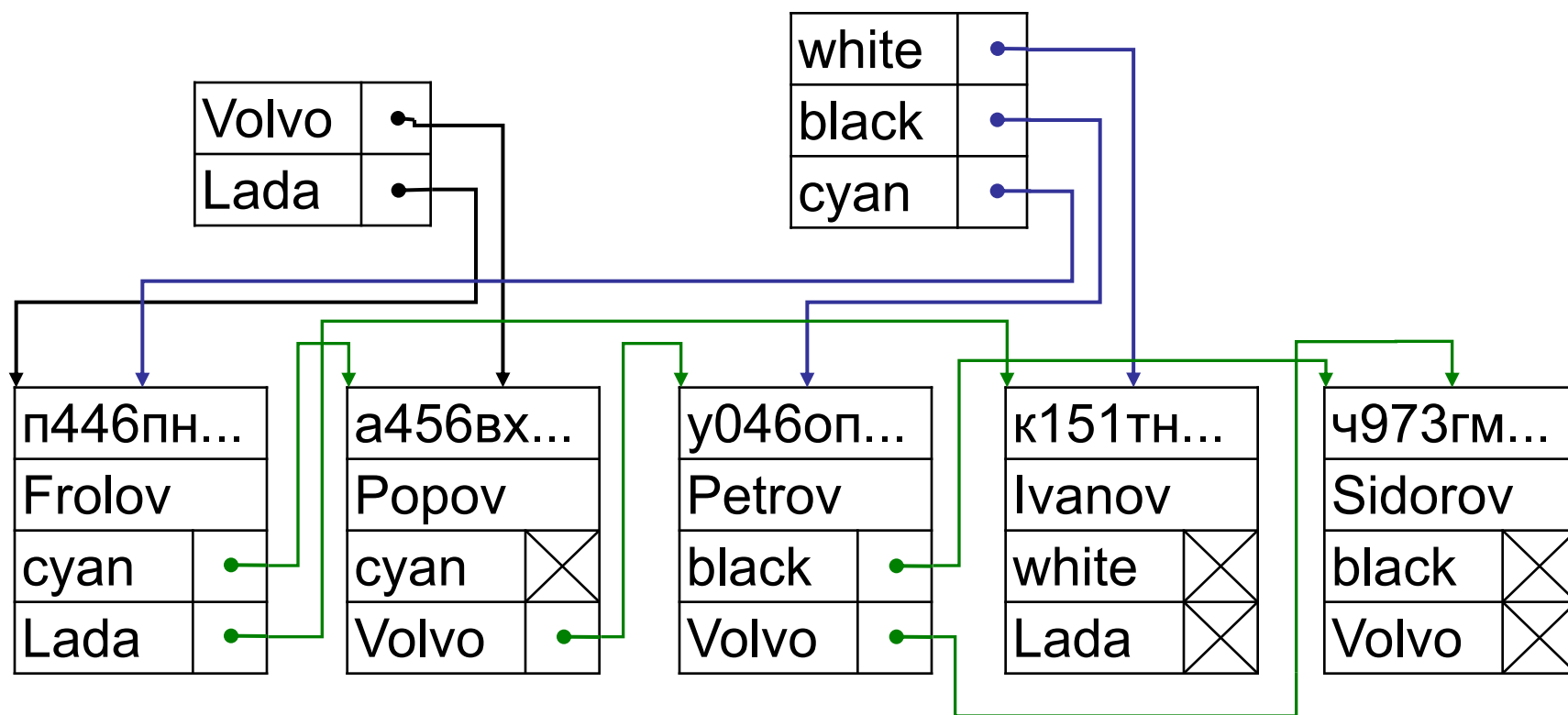
Эти данные можно представить в виде связанного списка.



- обозначает NULL

Если нужна информация об автомобилях «Lada», то необходимо просматривать весь список. Можно включить в каждый элемент указатель на следующую машину той же марки. Объем хранимой информации увеличится незначительно, но доступ существенно ускорится. Для каждой марки необходимо использовать внешний указатель на первую машину этой марки. Аналогично можно поступить с цветом машины, то есть ввести в каждый элемент указатель на следующую машину того же цвета.

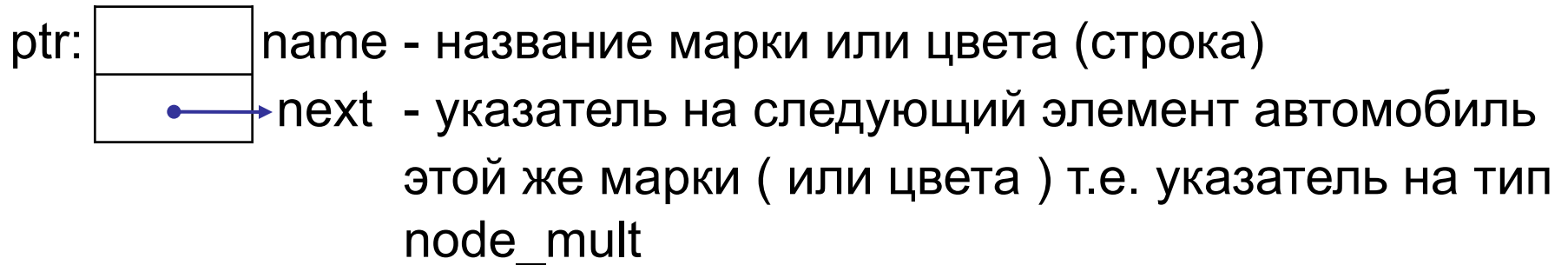
Здесь объединены четыре списка, в каждом элементе которых два указателя – на тот же цвет и на машину той же марки.



- обозначает NULL

При формировании мультисписка используем указатели двух типов.

1) Указатель на автомобиль заданной марки (или цвета).



2) Тип для автомобиля (включает указатель (1) на следующий автомобиль).

п446пн...		number
Frolov		owner
cyan	●	color
Lada	●	mark

# Замечания:

- Элемент в связанном мультисписке содержит не менее двух указателей (иначе мультисписок распадается на несколько линейных списков - список "Lada", список "Volvo" и т.д.).
- С увеличением числа указателей упрощается доступ к информации, хранящейся в списке, но усложняются операции вставки и удаления.
- Мультисписки широко используются при организации баз данных, так как позволяют избегать дублирования информации в памяти.

Формирование мультисписка.

Вывод списка на экран.

```
#include <stdio.h>
```

```
#include <malloc.h>
```

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

```
#define max_num 5
```

```
/*максимальное число различных марок и  
цветов*/
```

//структурные типы для мультисписка

struct node\_mult; //прототип для типа данных об автомобиле

/\*тип для массива указателей на автомобиль заданной марки или цвета\*/

struct ptr

{

char name[20]; //наименование марки или цвета

node\_mult \*next; /\*указатель на следующий автомобиль этой марки или цвета\*/

};

struct node\_mult /\*тип для автомобиля\*/

{

ptr mark, color;

char number[12], owner[20];

};

```
node_mult *input_auto(void)
{ //ввод полей автомобиля
  node_mult *t=(node_mult *)malloc(sizeof(node_mult));
  //выделение памяти
  printf("number :");gets(t->number);
  printf("color :");gets(t->color.name);
  printf("owner :");gets(t->owner);
  printf("mark :");gets(t->mark.name);
  t->color.next=t->mark.next=NULL;
  return t;
}

void print_auto(node_mult *t)
{ //вывод полей автомобиля
  printf("number %s\n",t->number);
  printf("color %s\n",t->color.name);
  printf("owner %s\n",t->owner);
  printf("mark %s\n\n",t->mark.name);
}
```



```
//добавление автомобиля к списку
void add_auto(node_mult* new_auto, ptr marks[], int *n_mark,
ptr colors[], int *n_color)
{
    node_mult *t;
    //указатель на текущий автомобиль
    int i; //номер в массиве цветов или марок

    //формирование массива марок и списка автомобилей
    for(i=0;i<*n_mark&&strcmp(new_auto->mark.name,
marks[i].name)!=0;i++);
    //ищем марку, совпадающую с текущей
    if(i==*n_mark) //новая марка
    {
        strcpy(marks[*n_mark].name,new_auto->mark.name);
```

```
//заноcим ее в массив марок
```

```
//поле указателя настраиваем на этот (первый) автомобиль
```

```
marks[*n_mark].next=new_auto; (*n_mark)++;
```

```
//увеличиваем число марок
```

```
}
```

```
else //такая марка уже была, на i-м месте
```

```
{
```

```
//указывает на первый автомобиль этой марки
```

```
t=marks[i].next;
```

```
while (t->mark.next)
```

```
    //ищем последний в списке автомобиль этой марки
```

```
    t=t->mark.next;
```

```
//новый автомобиль размещаем в списке за найденным
```

```
t->mark.next=new_auto;
```

```
}
```

```
//аналогичные действия для цветов пропущены
```

```
}
```

//вывод списка марок на экран

```
void print_mark (ptr marks[],int n_mark)
{
    node_mult *t;
    int i;
    for(i=0;i<n_mark;i++)
    {
        printf("auto %d, mark=%s\n",i,marks[i]. name);
        t=marks[i].next;
        while(t)
        {
            print_auto(t);
            t=t->mark.next;
        }
    }
    puts("");
}
```

// вывод списка цветов на экран аналогично

```
int main()
{
    //структура с данными об автомобиле
    node_mult*new_auto;
    //массив марок и указателей на первый
    ptr marks[max_num]; //автомобиль этой марки
    ptr colors[max_num]; //аналогично для цветов
    //количество различных марок и цветов, инициализация
    int n_mark=0, n_color=0; char ans; //ответ пользователя
    //формирование списка
    do
    {
        //вводим данные об очередном автомобиле
        new_auto=input_auto();
        //и добавляем его в список
        add_auto(new_auto, marks, &n_mark,colors, &n_color);
    }
```

```
printf("continue ? (y/n) ");
/* продолжаем ввод пока пользователь отвечает на вопрос 'Y'
или 'y' */
ans=getchar();
while(getchar()!='\n');
}
//пропуск символов до конца строки
while (ans=='y' || ans=='Y');
//вывод списка марок и автомобилей этой марки
puts("list of marks");
print_mark (marks,n_mark);
//аналогично для цветов
return 0;
}
```

Оценка за текущий контроль в 3 и 4 модуле учитывает результаты студента следующим образом.

Модуль 3.  $O_{\text{текущая } 1} = O_{\text{лекция}} + O_{\text{семинар}} + O_{\text{лаб. работа}} + O_{\text{ответы у доски}} + O_{\text{контр. работа}}$

Модуль 4.  $O_{\text{текущая } 2} = O_{\text{лекция}} + O_{\text{семинар}} + O_{\text{лаб. работа}} + O_{\text{ответы у доски}} + O_{\text{контр. работа}}$

Все оценки рассматриваются без округления.

Промежуточная оценка за 3 и 4 модуль вычисляется по формуле

$$O_{\text{промежуточная 3 и 4}} = 0,4 * O_{\text{текущая } 3} + 0,4 * O_{\text{текущая } 4} + 0,2 * O_{\text{экзамен 4 модуль}},$$

где  $O_{\text{текущая } 3}$ ,  $O_{\text{текущая } 4}$  — оценки текущего контроля 3 и 4 модуля, без округления.

Округление производится один раз, после вычисления промежуточной оценки, по правилам арифметики.

Экзаменационная оценка не является блокирующей. Промежуточная оценка за 3 и 4 модуль не может превышать 10 баллов, в случае превышения ставится промежуточная оценка 10 баллов.

Результирующая оценка за дисциплину вычисляется по формуле

$$O_{\text{результирующая}} = 0,4 * O_{\text{промежуточная 1 и 2}} + 0,6 * O_{\text{промежуточная 3 и 4}}$$

Округление производится по правилам арифметики. В диплом выставляется результирующая оценка.

- Ни один из элементов текущего контроля не является блокирующим.
- На некоторых семинарах и лекциях проводится тест или проверочная работа. Каждый вид работы оценивается от 1 до 4 баллов. В итоговую оценку эти баллы входят с коэффициентом, получаемым делением числа занятий, на которых проводилось оценивание, на общее количество занятий.
- При пропуске лекции или семинарского занятия по любой причине студент не может решить дополнительное задание для компенсации баллов, которые он мог бы получить на этом занятии.
- Кроме того, преподаватель может оценивать дополнительными баллами ответ студента у доски (максимум 2 балла) и активное участие в решении задач семинаров (например, выявление и исправление неточностей и ошибок в алгоритмах и при кодировании программ, внесение усовершенствований в алгоритм и т.п.) (максимум по 0.2 балла за каждый ответ).



- Для каждой лабораторной работы устанавливается срок защиты отчета (даты дедлайнов указаны в журналах). При своевременной защите работа оценивается полученным баллом, при опоздании на 1 неделю балл снижается на 40%, при опоздании на 2 недели балл снижается на 60% от полученной оценки. При опоздании более чем на 2 недели работа не оценивается.
- В случае пропуска занятий по уважительной причине (обязательно предоставление справки) срок сдачи лабораторной работы может быть перенесен на соответствующее количество рабочих дней.
- В случае пропуска занятий по уважительной причине (обязательно предоставление справки) предоставляется дополнительное время для написания контрольной работы (единственная дата переписывания заранее сообщается через старост).
- Переписывание контрольной работы с целью повышения полученной оценки не допускается.

# Распределение памяти и управление памятью

**Управление памятью** — это целый набор механизмов, которые позволяют контролировать доступ программы к оперативной памяти компьютера.

Когда программа выполняется в операционной системе компьютера, она нуждается в доступе к оперативной памяти Random Access Memory, с англ. — «Запоминающее устройство с произвольным доступом») для того, чтобы:

- загружать свой собственный код для выполнения;
- хранить значения переменных и структуры данных, которые используются в процессе работы;
- загружать внешние модули, которые необходимы программе для выполнения задач.

Помимо места, используемого для загрузки своего собственного байт-кода, программа использует две области в оперативной памяти — **стек** (stack) и **кучу** (heap).

# Стек

Стек используется для статического выделения памяти. Он организован по принципу «последним пришёл — первым вышел» (**LIFO**). Можно представить стек как стопку книг — разрешено взаимодействовать с самой верхней книгой: прочитать её или положить на неё новую.

Стек позволяет очень быстро выполнять операции с данными — манипуляции производятся с «верхней книгой в стопке». Книга добавляется в самый верх, если нужно сохранить данные, либо берётся сверху, если данные требуется прочитать.

В стеке вызовов разрешено взаимодействовать с любой книгой в стопке. Например, функция может передать адрес/ссылку на свои локальные данные на стеке другим вызываемым ею функциям, а те передать ещё дальше. В результате можно взаимодействовать с данными не только на вершине стека, но и где-то в глубине.

- существует ограничение: данные, которые предполагается хранить в стеке, обязаны быть конечными и статичными — их размер должен быть известен ещё на этапе компиляции;
- в стековой памяти хранится стек вызовов — информация о ходе выполнения цепочек вызовов функций в виде стековых кадров. Каждый стековый кадр это набор блоков данных, в которых хранится информация, необходимая для работы функции на определённом шаге — её локальные переменные и аргументы, с которыми её вызывали. Когда функция объявляет новую переменную, она добавляет её в верхний блок стека. Затем, когда функция завершает свою работу, очищаются все блоки памяти в стеке, которые функция использовала — иными словами, очищаются все блоки ее стекового кадра;

- каждый поток многопоточного приложения имеет доступ к своему собственному стеку;
- управление стековой памятью простое и прямолинейное; оно выполняется операционной системой;
- в стеке обычно хранятся данные вроде локальных переменных и указателей;
- при работе со стеком есть вероятность получать ошибки переполнения стека (stack overflow), так как максимальный его размер строго ограничен. Например, ошибка при составлении граничного условия в рекурсивной функции совершенно точно приведёт к переполнению стека;
- в большинстве языков существует ограничение на размер значений, которые можно сохранить в стеке.

# Куча

Куча используется для динамического выделения памяти, однако, в отличие от стека, данные в куче первым делом требуется найти с помощью «оглавления». Можно представить, что куча это такая большая многоуровневая библиотека, в которой, следуя определённым инструкциям, можно найти необходимую книгу. Операции на куче производятся несколько медленнее, чем на стеке, так как требуют дополнительного этапа для поиска данных.

- в куче хранятся данные динамических размеров, например, список, в который можно добавлять произвольное количество элементов;
- куча общая для всех потоков приложения;
- вследствие динамической природы, куча нетривиальна в управлении и с ней возникает большинство всех проблем и ошибок, связанных с памятью. Способы решения этих проблем предоставляются языками программирования;

- типичные структуры данных, которые хранятся в куче — это глобальные переменные (они должны быть доступны для разных потоков приложения, а куча как раз общая для всех потоков), ссылочные типы, такие как строки или ассоциативные массивы, а так же другие сложные структуры данных;
- при работе с кучей можно получить ошибки выхода за пределы памяти (out of memory), если приложение пытается использовать больше памяти, чем ему доступно;
- размер значений, которые могут храниться в куче, ограничен лишь общим объёмом памяти, который был выделен операционной системой для программы.

# Управление памятью. Различные подходы

## Ручное управление памятью

Язык не предоставляет механизмов для автоматического управления памятью. Выделение и освобождение памяти для создаваемых объектов остаётся полностью на совести разработчика.

Пример такого языка — **C**. Он предоставляет ряд методов (*malloc*, *realloc*, *calloc* и *free*) для управления памятью — разработчик должен использовать их для выделения и освобождения памяти в своей программе. Этот подход требует большой аккуратности и внимательности. Так же он является в особенности сложным для новичков.



# Сборщик мусора

Сборка мусора — это процесс автоматического управления памятью в куче, который заключается в поиске не использующихся участков памяти, которые ранее были заняты под нужды программы. Это один из наиболее популярных вариантов механизма для управления памятью в современных языках программирования. Подпрограмма сборки мусора обычно запускается в заранее определённые интервалы времени и бывает, что её запуск совпадает с ресурсозатратными процессами, в результате чего происходит задержка в работе приложения. Сборщик мусора используется в **JVM (Java/Scala/Groovy/Kotlin), JavaScript, Python, C #, Golang, OCaml и Ruby.**

# Сборщик мусора на основе алгоритма пометок (Mark & Sweep)

Это алгоритм, работа которого происходит в две фазы: первым делом он помечает объекты в памяти, на которые имеются ссылки, а затем освобождает память от объектов, которые пометки не получили. Этот подход используется, например, в JVM, C#, Ruby, JavaScript и Golang. В JVM существует на выбор несколько разных алгоритмов сборки мусора, а JavaScript-движки, такие как V8, используют алгоритм пометок в дополнение к подсчёту ссылок. Такой сборщик мусора можно подключить в C и C++ в виде внешней библиотеки.

# Сборщик мусора с подсчётом ссылок

Для каждого объекта в куче ведётся счётчик ссылок на него — если счётчик достигает нуля, то память высвобождается. Данный алгоритм в чистом виде не способен корректно обрабатывать циклические ссылки объекта на самого себя. Сборщик мусора с подсчётом ссылок, вместе с дополнительными ухищрениями для выявления и обработки циклических ссылок, используется, например, в **PHP**, **Perl** и **Python**. Этот алгоритм сборки мусора так же может быть использован и в **C++**.

Получение ресурса есть инициализация  
(RAII- *Resource Acquisition Is Initialization (RAII)*)

**RAII** — это программная идиома в ООП, смысл которой заключается в том, что выделяемая для объекта область памяти строго привязывается к его времени существования. Память выделяется в конструкторе и освобождается в деструкторе. Данный подход был впервые реализован в **C++**, а так же используется в **Ada** и **Rust**.

## Автоматический подсчёт ссылок (ARC - *automatic reference counting*)

Данный подход весьма похож на сборку мусора с подсчётом ссылок, однако, вместо запуска процесса подсчёта в определённые интервалы времени, инструкции выделения и освобождения памяти вставляются на этапе компиляции прямо в байт-код. Когда же счётчик ссылок достигает нуля, память освобождается как часть нормального потока выполнения программы.

Автоматический подсчёт ссылок всё так же не позволяет обрабатывать циклические ссылки и требует от разработчика использования специальных ключевых слов для дополнительной обработки таких ситуаций. **ARC** является одной из особенностей транслятора **Clang**, поэтому присутствует в языках **Objective-C** и **Swift**. Так же автоматический подсчет ссылок доступен для использования в **Rust** и новых стандартах **C++** при помощи умных указателей.

# Владение

Это сочетание **RAII** с концепцией владения, когда каждое значение в памяти должно иметь только одну переменную-владельца. Когда владелец уходит из области выполнения, память сразу же освобождается. Можно сказать, что это примерно как подсчёт ссылок на этапе компиляции. Данный подход используется в **Rust**.

# Процесс компиляции программ на C++

Все действия будут производиться на **Ubuntu** версии **16.04**.

Используя компилятор **g++** версии:

```
$ g++ --version g++ (Ubuntu 5.4.0-  
6ubuntu1~16.04.9) 5.4.0 20160609
```

Состав компилятора **g++**

- **cpp** — препроцессор
- **as** — ассемблер
- **g++** — сам компилятор
- **ld** — линкер

## Этапы компиляции:

Перед тем, как приступить, создадим исходный .cpp файл, с которым и будем работать в дальнейшем.

**driver.cpp:**

```
#include <iostream>
using namespace std;
#define RETURN return 0
int main()
{ cout << "Hello, world!" << endl;
RETURN; }
```



# **Зачем нужно компилировать исходные файлы?**

**Исходный C++ файл** — это всего лишь код, но его невозможно запустить как программу или использовать как библиотеку. Поэтому каждый исходный файл требуется скомпилировать в исполняемый файл.

# Препроцессинг

Самая первая стадия компиляции программы.

Препроцессор — это макро процессор, который преобразовывает вашу программу для дальнейшего компилирования. На данной стадии происходит работа с препроцессорными директивами.

Например, препроцессор добавляет хэдеры в код (`#include`), убирает комментирования, заменяет макросы (`#define`) их значениями, выбирает нужные куски кода в соответствии с условиями `#if`, `#ifdef` и `#ifndef`.

Хэдеры, включенные в программу с помощью директивы `#include`, рекурсивно проходят стадию препроцессинга и включаются в выпускаемый файл. Однако, каждый хэдер может быть открыт во время препроцессинга несколько раз, поэтому, обычно, используются специальные препроцессорные директивы, предохраняющие от циклической зависимости.

Получим препроцессированный код в выходной файл `driver.i` (прошедшие через стадию препроцессинга C++ файлы имеют расширение `.i`), используя флаг `-E`, который сообщает компилятору, что компилировать (об этом далее) файл не нужно, а только провести его препроцессинг:

```
g++ -E driver.cpp -o driver.ii
```

Взглянув на тело функции `main` в новом сгенерированном файле, можно заметить, что макрос `RETURN` был заменен:

```
int main() {  
    cout << "Hello, world!" << endl;  
    return 0;  
}  
//driver.ii
```

В новом сгенерированном файле также можно увидеть огромное количество новых строк, это различные библиотеки и хэдер `iostream`.

# Компиляция

На данном шаге g++ выполняет свою главную задачу — компилирует, то есть преобразует полученный на прошлом шаге код без директив в ассемблерный код. Это промежуточный шаг между высокоуровневым языком и машинным (бинарным) кодом.

**Ассемблерный код** — это доступное для понимания человеком представление машинного кода.

Процесс компиляции состоит из следующих этапов:

- **Лексический анализ.** Последовательность символов исходного файла преобразуется в последовательность лексем.
- **Синтаксический анализ.** Последовательность лексем преобразуется в дерево разбора.
- **Семантический анализ.** Дерево разбора обрабатывается с целью установления его семантики (смысла) — например, привязка идентификаторов к их декларациям, типам, проверка совместимости, определение типов выражений и т. д.
- **Оптимизация.** Выполняется удаление излишних конструкций и упрощение кода с сохранением его смысла.
- **Генерация кода.** Из промежуточного представления порождается объектный код.

Используя флаг -S, который сообщает компилятору остановиться после стадии компиляции, получим ассемблерный код в выходном файле driver.s:

```
$ g++ -S driver.ii -o driver.s
```

driver.s

Мы можем все также посмотреть и прочесть полученный результат. Но для того, чтобы машина поняла наш код, требуется преобразовать его в машинный код, который мы и получим на следующем шаге.

# Ассемблирование

Так как x86 процессоры исполняют команды на бинарном коде, необходимо перевести ассемблерный код в машинный с помощью ассемблера.

**Ассемблер** преобразовывает ассемблерный код в машинный код, сохраняя его в объектном файле.

**Объектный файл** — это созданный ассемблером промежуточный файл, хранящий кусок машинного кода. Этот фрагмент машинного кода, который еще не был связан вместе с другими фрагментами машинного кода в конечную выполняемую программу, называется **объектным кодом**.



Далее возможно сохранение данного объектного кода в статические библиотеки для того, чтобы не компилировать данный код снова.

Получим машинный код с помощью ассемблера (as) в выходной объектный файл driver.o:

```
$ as driver.s -o driver.o
```

Но на данном шаге еще ничего не закончено, ведь объектных файлов может быть много и нужно их все соединить в единый исполняемый файл с помощью компоновщика (линкера). Поэтому мы переходим к следующей стадии.

# Компоновка

**Компоновщик** (линкер) связывает все объектные файлы и статические библиотеки в единый исполняемый файл, который мы и сможем запустить в дальнейшем. Для того, чтобы понять как происходит связка, следует рассказать о таблице СИМВОЛОВ.

**Таблица символов** — это структура данных, создаваемая самим компилятором и хранящаяся в самих объектных файлах. Таблица символов хранит имена переменных, функций, классов, объектов и т.д., где каждому идентификатору (символу) соотносится его тип, область видимости.

Также таблица символов хранит адреса ссылок на данные и процедуры в других объектных файлах.

Именно с помощью таблицы символов и хранящихся в них ссылок линкер будет способен в дальнейшем построить связи между данными среди множества других объектных файлов и создать единый исполняемый файл из них.

Получим исполняемый файл driver:

```
$ g++ driver.o -o driver // также тут можно добавить  
//и другие объектные файлы и библиотеки
```

# Загрузка

Последний этап, который предстоит пройти нашей программе — вызвать загрузчик для загрузки нашей программы в память. На данной стадии также возможна подгрузка динамических библиотек.

Запустим нашу программу:

```
$ ./driver
```

```
// Hello, world!
```

# Учебные ассистенты

- Учебными ассистентами могут стать студенты и аспиранты, не имеющие оценок ниже 8 баллов по итогам промежуточной аттестации по учебной дисциплине, к реализации которой они привлекаются, а также оценок ниже 6 баллов по другим дисциплинам.
- Учебный ассистент не должен:
  - обучаться на курсе, студентам которого преподается данная дисциплина;
  - работать одновременно с двумя и более преподавателями или по нескольким разным дисциплинам;
  - иметь академические задолженности и участвовать в пересдачах;
  - иметь дисциплинарные взыскания;
  - работать в Университете на преподавательской должности или оказывать университету преподавательские услуги на основании гражданско-правового договора (для студентов магистратуры или аспирантов);
  - обучаться по индивидуальному учебному плану специального типа.

# Работы для учебных ассистентов

- Сопровождение пар, помощь в приеме лабораторных работ.
- Ведение журналов, т.е. заполнение данных о проверочных работах и выполнении лабораторного практикума. Вести журналы нужно еженедельно. Ассистенты, ведущие журналы, приходят на лабораторные только по дедлайнам (в случае необходимости), т.е. гораздо меньше остальных. Также эти ассистенты освобождены от проверки контрольных и проверочных работ.
- Проверка проверочных и контрольных работ. Распределяется между всеми учебными ассистентами.
- Ассистенты, которые не могут сопровождать л.р. ни очно, ни дистанционно, получают больше работ на проверку.
- Помощь в размещении материалов в Smart Lms.

# Данные об учебных ассистентах

Для подачи заявки на работу учебным ассистентом пришлите мне письмо на [eroxinaea@yandex.ru](mailto:eroxinaea@yandex.ru). В письме укажите.

- Номер телефона.
- Email (корпоративный и личный).
- Курс, на котором вы учитесь
- ФИО.
- Оценки по дисциплине за 1, 2, 3, 4 модуль.
- Группу.

данные следует прислать после сдачи летней сессии, до 30 июня включительно.