МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
МОСКОВСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ

Хлебников А.А., Каширская Е.Н., Набатов С.И., Розов Н.О.

Язык C/C++

Методическое пособие для учащихся

Москва — 2017

УДК ___
ББК ___
_ ___

Хлебников А.А. Язык C/C++ [Электронный ресурс]: Методическое пособие/ Хлебников А.А., Каширская Е.Н., Набатов С.И., Розов Н.О.

Аннотация. (указывается вид учебно-научного материала) …
…
…
Указывается предназначение учебно-научного материала (определяется место в учебно-научном процессе в МИРЭА)
…
…
(Указывается вид учебно-научного материала) издается в авторской редакции.

Авторский коллектив: Хлебников Андрей Александрович, Каширская Елизавета Натановна, Набатов Сергей Игоревич, Розов Никита Олегович.

Рецензенты:

Фамилия Имя Отчество, ученая степень, ученое звание, занимаемая должность, структурное подразделение, название организации

Фамилия Имя Отчество, ученая степень, ученое звание, занимаемая должность, структурное подразделение, название организации

Минимальный системные требования:
Поддерживаемые ОС: Windows 2000 и выше.
Память: ОЗУ 128МБ.
Жесткий диск: 20 Мб.
Устройства ввода: клавиатура, мышь.
Дополнительные программные средства: Программа Adobe Reader.
Подписано к использованию по решению Редакционно-издательского совета Московского технологического университета от (дата в формате ДД.ММ.ГГГГ)
Объем: __ Мб
Тираж: 1

ISBN _____

Оглавление

Список использованной литературы и материалов

1. Р. Лафоре «Объектно-ориентированное программирование в С++», 4-е издание, изд. «Питер»
2. http://netlib.narod.ru/library/book0003/ch04_11. htm
3. http://natalia.appmat.ru/c&c++/compito1. html.

Сведения об авторах

Фамилия_1 Имя Отчество, ученая степень, ученое звание, занимаемая должность, структурное подразделение (возможны дополнительные сведения, включая фотографию)

Фамилия_2 Имя Отчество, ученая степень, ученое звание, занимаемая должность, структурное подразделение (возможны дополнительные сведения, включая фотографию)

Фамилия_3 Имя Отчество, ученая степень, ученое звание, занимаемая должность, структурное подразделение (возможны дополнительные сведения, включая фотографию)

Введение

Среди современных языков программирования язык С является одним из наиболее распространенных. Язык С универсален, однако наиболее эффективно его применение в задачах системного программирования – разработке трансляторов, операционных систем, инструментальных средств. Язык С хорошо зарекомендовал себя эффективностью, лаконичностью записи алгоритмов, логической стройностью программ. Во многих случаях программы, написанные на языке С, сравнимы по скорости с программами, написанными на Ассемблере, при этом они более наглядны и просты в сопровождении.

Язык С имеет ряд существенных особенностей, которые выделяют его среди других языков программирования. В значительной степени на формирование идеологии языка повлияла цель, которую ставили перед собой его создатели – обеспечение системного программиста удобным инструментальным языком, который мог бы заменить Ассемблер. В результате появился язык программирования высокого уровня, обеспечивающий необычайно легкий доступ к аппаратным средствам компьютера. С одной стороны, как и другие современные языки высокого уровня, язык С поддерживает полный набор конструкций структурного программирования, модульность, блочную структуру программы. С другой стороны, язык С имеет ряд низкоуровневых черт.

Перечислим некоторые особенности языка С.

В языке С реализован ряд операций низкого уровня. Некоторые из таких операций напрямую соответствуют машинным командам, например, поразрядные операции или операции ++ и −−. Базовые типы данных языка С отражают те же объекты, с которыми приходится иметь дело в программе на Ассемблере – байты, машинные слова и т.д. Несмотря на наличие в языке С развитых средств построения составных объектов (массивов и структур), в нем практически отсутствуют средства для работы с ними как с единым целым. Язык С поддерживает механизм указателей на переменные и функции. Указатель – это переменная, предназначенная для хранения машинного адреса некоторой переменной или функции. Поддерживается арифметика указателей, что позволяет осуществлять непосредственный доступ и работу с адресами памяти практически так же легко, как на Ассемблере. Использование указателей позволяет создавать

высокоэффективные программы, однако требует от программиста особой осторожности. Как никакой другой язык программирования высокого уровня, язык С «доверяет» программисту. Даже в таком существенном вопросе, как преобразование типов данных, налагаются лишь незначительные ограничения. Однако это также требует от программиста осторожности и самоконтроля. Несмотря на эффективность и мощность конструкция языка С, он относительно мал по объему. В нем отсутствуют встроенные операторы ввода/вывода, динамического распределения памяти, управления процессами и т.п., однако в системное окружение языка С входит библиотека стандартных функций, в которой реализованы подобные действия. Язык С++ – это язык программирования общего назначения, цель которого – сделать работу серьёзных программистов более приятным занятием. За исключением несущественных деталей, язык С++ является надмножеством языка С. Помимо возможностей, предоставляемых языком С, язык С++ обеспечивает гибкие и эффективные средства определения новых типов.

Язык программирования служит двум взаимосвязанным целям: он предоставляет программисту инструмент для описания подлежащих выполнению действий и набор концепций, которыми оперирует программист, обдумывая, что можно сделать. Первая цель в идеале требует языка, близкого к компьютеру, чтобы все важные элементы компьютера управлялись просто и эффективно способом, достаточно очевидным для программиста. Язык С создавался на основе именно от этой идеи. Вторая цель в идеале требует языка, близкого к решаемой задаче, чтобы концепции решения могли быть выражены понятно и непосредственно. Эта идея привела к пополнению 3 языка С свойствами, превратившими его в язык С++.

Ключевое понятие в языке С++ – класс. Классы обеспечивают сокрытие информации, гарантированную инициализацию данных, неявное преобразование определяемых пользователем типов, динамическое определение типа, контроль пользователя над управлением памятью и механизм перегрузки операторов. Язык С++ предоставляет гораздо лучшие, чем язык С, средства для проверки типов и поддержки модульного программирования. Кроме того, язык содержит усовершенствования, непосредственно не связанные с классами, такие как: символические константы, встраивание функций в место вызова, параметры функций по умолчанию, перегруженные имена функций, операторы управления свободной памятью и ссылки. Язык С++ сохраняет способность языка С

эффективно работать с аппаратной частью на уровне битов, байтов, слов, адресов и т.д. Это позволяет реализовывать пользовательские типы с достаточной степенью эффективности.

1.Основные понятия языков C и C++. Типы данных

1.1Типы данных языка C и C++

Имя	Размер	Представляемые Значения	Диапазон
bool	1 байт	Логические	False,true
char	1 байт	Символ ASCII	От -128 до 127
wchar_t	2 байта	Символы Unicode	От 0 до 65535
short int	2 байта	Целые числа	От -32768 до 32767
int	4 байта	Целые числа	От – 2147483648 до 2147483647
long int	4 байта	Целые числа	От – 2147483648 до 2147483647
unsigned char	1 байт	Символы и целые числа	От 0 до 255
unsigned short int	2 байта	Целые числа	От 0 до 65535
unsigned int	4 байта	Целые числа	От 0 до 4294967295
unsigned long int	4 байта	Целые числа	От 0 до 4294967295
float	4 байта	Вещественные числа	От 3.4e–38 до 3.4e+38.
double	8 байт	Вещественные числа	От 2.2e-308 до 1.8e+308

1.2 Комментарии

Комментарий – это последовательность символов, которая игнорируется компилятором языка C/C++.

В C и C++ есть два вида комментариев: первый – построчный, второй многострочный.

```
// Пример построчного комментария
```

Для объявления построчного комментария нужно написать последовательность символов «//». Закрывать ее не требуется.

```
/* Пример
   многострочного
   комментария */
```

Для создания многострочного комментария нужно объявить такую последовательность символов - /* */. Первая группа символов открывает многострочный комментарий, вторая – закрывает.

Комментарии не могут быть вложенными!

1.3.Операторы

Арифметические операторы:

Операция (выражение)		Оператор	Синтаксис выражения
Присваивание		=	a=b
Сложение		+	a+b
Вычитание		-	a-b
Умножение		*	a*b
Деление		/	a/b
Остаток от деления целых чисел		%	a%b
Инкремент	Префиксная	++	++a
	Постфиксная		a++
Декремент	Префиксная	--	--a
	Постфиксная		a--

Все вышеописанные операции вам знакомы, есть смысл пояснить лишь, как работает префиксная и постфиксная форма записи инкремента и декремента.

Префиксная форма записи означает, что инкремент (декремент) будет выполнен первым. Постфиксная форма записи, в свою очередь, означает, что инкремент (декремент) будет выполнен последним.

Пример:

a= b* ++c - в данном случае инкрементирование выполнится перед умножением.

Операторы сравнения:

Операция (выражение)	Оператор	Синтаксис выражения
Равенство	==	a==b
Неравенство	!=	a!=b
Больше	>	a>b
Меньше	<	a<b
Больше или равно	>=	a>=b
Меньше или равно	<=	a<=b

Логические операторы:

Операция (выражение)	Оператор	Синтаксис выражения
Логическое НЕ	!	!a
Логическое И	&&	a&&b
Логическое ИЛИ	||	a||b

Перед приоритетом операций есть смысл поговорить о составном присваивании. В языке C/C++ есть одна удобная вещь. Часто приходится увеличивать некую переменную и присваивать новое значение переменной a. Это неудобно. Потому было создано составное присваивание.

Пример: a+=b - Это означает, что к старому значению априбавили b, а после - присвоили a.

Приоритет операций:

Тип операций	Операции	Приоритет
Унарные	!, ++, --, +, -	Высший
Арифметические	Мультипликативные: *, /, % Аддитивные: +, -	
Отношения	Неравенства: <, >, <=, >= Равенства: ==, !=	
Логические	&&, ||	
Присваивания	=, +=, -=, *=, /=, %=	Низший

1.4 Структура программы на языке С:

```
#include < stdio.h >
int main ( )
{
  printf ( "Let's start smth else\n" );
  return 0;
}
```

Функция main():

Первым исполненным оператором становится первый оператор функции main(). При запуске программы управление всегда передается функции main(). При попытке запустить программу без данной функции, компилятор выдаст сообщение об ошибке.

Директивы:

Первая строка, с которой начинается верхний код, является директивами. Эта строка – директива препроцессора.

#include<stdio.h> является директивой препроцессора, и в отличие от оператора, который дает компьютеру указание что-либо сделать, директива указывает компилятору.

Файл, включаемый с помощью директивы #include –заголовочный файл.

Функции ввода и вывода:

В языке С используются функции printf() и scanf() для вывода и ввода соответственно. Для того, чтобы эти функции работали, надо подключить библиотеку <stdio.h>.

1.5. Управляющие последовательности

Управляющая последовательность означает, что символ \ «управляет» интерпретацией следующих за ним символов последовательности.

Управляющая последовательность	Символ
\a	Звонок
\b	Возврат на шаг
\t	Горизонтальная табуляция
\n	Переход на новую строку
\v	Вертикальная табуляция
\r	Возврат каретки
\f	Перевод формата
\”	Кавычки
\’	Апостроф
\0	Ноль-символ
\\	Обратная дробная черта
\ddd	Восьмеричный код символа
\xddd	Шестнадцатеричный код символа

2.Циклы и ветвления

Важным элементом любого языка программирования являются циклы. Допустим, вам нужно выполнить некоторый участок код n-ное число раз, было бы довольно неудобно писать его n раз(а если учесть, что

иногда требуется повторы в десятки тысяч раз и более, то вообще невозможно). Поэтому язык С предоставляет возможность использования циклов.

2.1 Цикл For

Давайте взглянем на следующую программу:

```c
#include <stdio.h>

int main ( ) {
    int i;
    for ( i = 1; i <= 10; i ++ ) {
        printf( "%d", i );
    }
    return 0;
}
```

При запуске она выдает: 1 2 3 4 5 6 7 8 9 10. Что же происходит? После ключевого слова for открываются скобки, и до 1 точки с запятой располагается определение переменной счетчика – int i = 1 –это инициализирующее выражение. Далее ставится условие проверки – i<=10, логично, что как только i станет равно 11 цикл прекратиться. Изменением i занимается инкрементирующее выражение – i++. Чтобы это было легче понять, взглянем на блок-схему:

Как видно из примера данный цикл чаще всего используется, когда количество повторений уже известно.

Так же следует добавить, что не всегда цикл for имеет инкрементирующее выражение, условие проверки или инициализирующее выражение. Иногда одно, два или даже все три части могут пропускаться, при этом точки с запятой всегда должны быть на своих местах!

2.2 Цикл while

Следующим на очереди является цикл с предусловием - while. Далее код:

```
#include <stdio.h>

int main ( ) {
    int n = 1;
    int m = 0;
    printf ( "Please enter a nubmer of integers. In the end, enter 0: " );
    while ( n != 0 ) {
        scanf ( "%d", &n );
        m += n;
    }
    printf ( "The sum equals: &d", m );
    return 0;
}
```

Данный цикл, выглядит как упрощение for, так как имеет лишь условие проверки.

В данном случае программа просит пользователя ввести ряд целых чисел, а когда он будет завершен ввести 0. И выводит на экран сумму, что посчитал.

2.3 Цикл do

Последний из циклов- do. Этот цикл с постусловием часто применяется, когда участок кода должен быть выполнен как минимум 1 раз. В остальном же сильно походит на while.

Пример кода:

```
#include <stdio.h>

int main ( ) {
   int n = 1;
   int m = 0;
   printf ( "Enter a number of integers. In the end, enter 0: " );
   do {
     scanf ( "%d", &n );
     m += n;
   } while ( n != 0 );
   printf ( "The sum equals: %d", m );
   return 0;
}
```

Обратите внимание на точку с запятой, которая в данном цикле является необходимой.

Данный код делает то же, что и предыдущий.

2.4 Оператор if.. else

Еще одним базовым аспектом любого языка программирования являются ветвления. Они помогают программе выбрать путь исполнения, основываясь на имеющихся у нее данных. Самым простым оператором ветвления является if, он просто проверяет, выполняется ли условие, и если да –то исполняет свой блок.

Пример:

```
#include <stdio.h>

int main ( ) {
    int n;
    printf ( "Input a number: " );
    scanf ( "%d", &n );
    if ( n > 5 ) {
      printf ( "Your number is more than 5" );
    }
    return 0;
}
```

Этот код проверяет больше ли число, которое ввел пользователь, цифры 5.

Логично, что если есть реакция на ввод числа больше 5, то должна быть и на меньшее число, здесь поможет добавление else:

```
#include <stdio.h>

int main ( ) {
    int n;
    printf ( "Enter a number: " );
    scanf ( "%d", &n );
    if ( n > 5 ) {
        printf ( "Your number is more than 5" );
    } else {
        printf ( "Your number is less than 5" );
    }
    return 0;
}
```

Этот код проверяет больше или меньше число, которое ввел пользователь, цифры 5.

2.5 Оператор switch

Конечно, два варианта развития событий это хорошо, но что делать, если требуется несколько больше реакций. Можно поступить следующим образом:

```
if (…) {
        ...
    } else if (…) {
        ...
    } else if (…) {
        ...
    } else if (…) {
        ...
}
```

Но это довольно неудобно, поэтому лучше использовать оператор switch:

```
#include <stdio.h>

int main ( ) {
    int n;
    printf ( "Enter a number: " );
    scanf ( "%d", &n );
    printf ( "\nYou entered" );
    switch ( n ) {
        case 20 : printf ( "20" );
                  break;
        case 15 : printf ( "15" );
                  break;
        case 10 : printf ( "10" );
                  break;
        default : printf ( "nor 10 or 15 or 20");
    }
    return 0;
}
```

Этот код проверяет, что какое число ввел пользователь, и в случае если это число – 15, 10 или 20, он выводит на экран то, что ввел пользователь.

Здесь есть несколько моментов, на которых нужно остановиться. Во-первых, оператор break, предназначенный для выхода из вложения, будь то цикл или оператор switch. Если его не ставить после каждого case, то на экран будет выведено не только, то что нужно, но и все написанное ниже. Во-вторых, default, который исполняется, если ни одно из условий не является верным.

Практическая работа №1

1. Напишите программу, принимающую на вход число, и определяющую является ли это число простым. Для решения этой задачи используйте решето Эратосфена.

2. Выведите на экран размеры стандартных типов. Чтобы решить эту непростую задачу, используйте библиотечную функцию sizeof. Программа не принимает никаких значений с клавиатуры и выводит размеры стандартных типов.

3. С помощью символов “*” и “=” выведите на экран американский флаг размером 8 на 12.

4. Посчитайте объем усеченного конуса. Реализуйте эту программу так, чтобы пользователь НЕ мог ввести ничего, кроме числа. Если он вводит не число, то пусть он повторит ввод. Не забудьте про несуществующие значения (деление на 0 и прочее)

   Примечание:

   $$V = \frac{1}{3}\pi h(r1^2 + r1r2 + r2^2)$$

5. Напишите программу, которая решает квадратные уравнения, коэффициенты вводятся с клавиатуры. Сделайте ее в цикле, чтобы она завершалась, когда пользователь отказывался больше вводить коэффициенты.

6. Билет называют «счастливым», если в его номере сумма первых трех цифр равна сумме последних трех. Подсчитать число тех «счастливых» билетов, у которых сумма трех цифр равна 13. Номер билета может быть от 000000 до 999999.

7. Напишите простой калькулятор. Его функционал должен включать в себя произведение, сложение, вычитание, деление.

8. На ввод по очереди подаются координаты 5 точек. Определить, сколько из них попадает в круг радиусом R с центром в точке (a,b).

9. Радиус и координаты центра выберите сами. Посчитайте НОД и НОК, числа вводятся с клавиатуры.

10. Каждая бактерия делится на две в течение одной минуты. В начальный момент имеется одна бактерия. Составьте программу, которая рассчитывает количество бактерий на заданное вами целое значение момента времени (15 минут, 7 минут и т.п.).

3. Функции

3.1 Объявление и определение функций

Определяющей особенностью любого структурного языка являются функции. Так как обычная программа на любом языке состоит из множества строк кода, то для облегчения написания/чтения было решено объединять некоторые участки программы в отдельные блоки – функции.

Сам по себе, язык С предоставляет некоторое количество библиотек со стандартными функциями. Однако, для понимания работы, лучше сначала научиться создавать свои. Итак, пример функции:

```c
#include <stdio.h>

void example( );

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

void example ( )
{
    printf ( "It's example" );
}
```

Что же, это был самый простой пример функции - с типом возвращаемого значения void и без аргументов. Здесь мы можем увидеть 3 упоминания функции: объявление, вызов и определение. Объявление перед main() предупреждает компилятор о том, что где-то далее в программе есть тело функции с таким же полным именем. Полное имя включает в себя возвращаемое значение, типы параметров и, собственно, имя. Объявление же содержит тело функции. Для лучшего понимания рассмотрим еще небольшой код:

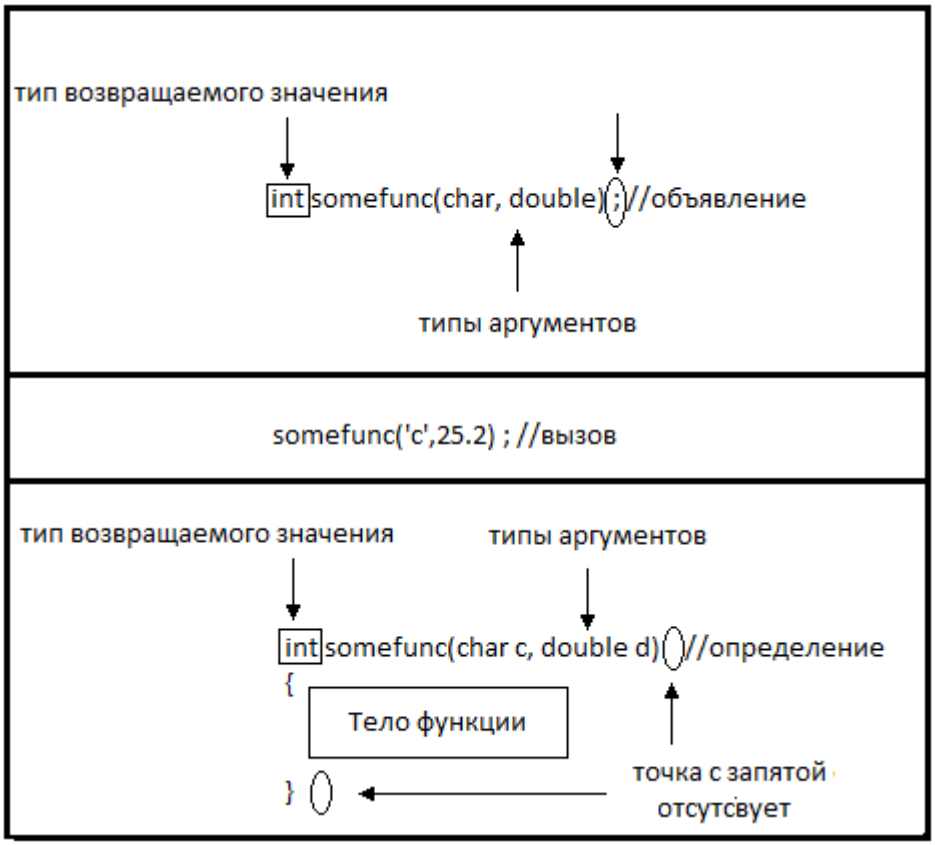
```c
#include <stdio.h>

int example (int, int);

int main ( )
{
    int answer;
    answer = example ( 4, 5 );
    printf ( "4*5 = %d", answer );
    return 0;
}

int example ( int i, int n )
{
  return i*n;
}
```

Чтобы обобщить все сказанное выше построим примерный план написания любой функции:



Примечание: Можно поставить определение на место объявления, а определения убрать, этот подход практикуется, если функция небольшая. Однако лучше им не пользоваться, т.к. когда функций много, вы можете запутаться.

3.2 Передача аргументов по ссылке

В предыдущей теме, были рассмотрены общие моменты создания функций. Разумеется, что существует некоторое число нюансов, которые будут описаны далее. Первым является передача аргументов по ссылке.

Ранее в примере аргументы передавались в функцию по значению, это значило, что в функции мы работали с копиями этих объектов, а если размер объектов слишком большой или важна память, очевидно, что лишние расходы ее на копии ни к чему. Другим аспектом, заслуживающим внимания, является то, что иногда от функции требуется вернуть больше одного значения, но синтаксис С не позволяет этого сделать. И тут на помощь приходят ссылки.

Ссылка представляет собой механизм, который не создает копии, а позволяет работать с самой переменной, указанной в качестве аргумента.

Пример программы, возводящей число в степень:

```
#include <stdio.h>

void power ( int *, int );

int main ( )
{
    int n = 2;
    int s = 7;
    power ( &n, s );
    printf ( "The result is: %d", n );
    return 0;
}

void power ( int *number, int stepen )
{
    int k = *number;
    for ( ; stepen > 1; stepen -- ) {
        *number *= k;
    }
}
```

В листинге данного кода функция power ( ) принимает адрес, который присваивается локальной переменной-указателю (о которых будет рассказано чуть ниже). В теле функции происходит изменение значения по адресу, содержащемуся в указателе. Однако по факту это адрес переменной n, из функции main ( ), следовательно меняется и значение n.

ВАЖНО: Передавать по адресу можно лишь переменные, если вы вместо переменной передадите какое-то значение, к примеру, в нашем случае вместо &nпередать n, то вызов этой функции приведет к ошибке.

3.3 Рекурсия

Если просто, рекурсия-это вызов функцией самой себя. Это мощный инструмент в руках опытного программиста. Часто для примера берут программу, вычисляющую числа Фибоначчи, что же посмотрим:

(Данный пример считает до 20 числа Фибоначчи, приведен чуть ниже).

```c
#include <stdio.h>

int fibo ( int );

int main ( )
{
    int q;
    int m;
    q = 20;
    m = fibo ( q );
    printf ( "The result is: %d", m );
    return 0;
}

int fibo ( int y )
{
    if ( y < 1 ) {
        return 0;
    }
    if ( y == 1 ) {
        return 1;
    }
    return fibo ( y-1 ) + fibo ( y - 2 );
}
```

3.4. Встраиваемые функции

Использование функций хоть и сокращает размер программы, однако увеличивает время выполнения программы. Для выполнения функции должны быть сгенерированы команды переходов, команды, сохраняющих значения регистров процессора, команды, помещающие в стек и извлекающие из стека аргументы функции (в том случае, если они есть), команды, восстанавливающие значения регистров после выполнения функции и команда перехода функции обратно в программу.

Обычно в функцию выносится повторяющийся участок кода, однако если мы вынесем маленький повторяющийся участок кода, то мы не выиграем по времени работы программы, хоть и сэкономим память.

Для работы с маленьким участком кода и были придуманы встраиваемые функции. Они так названы, т.к. при компиляции программы их исполняемый код «встраивается» в код программы, но при этом остается независимой частью программы.

Рассмотрим пример:

```
#include <stdio.h>

inline int example_of_funct ( int sh )
{
    return sh * 10;
}

int main ( )
{
      int n = 10;
      printf ( "%d", example_of_funct ( n ) );
      return 0;
}
```

Функция example_of_funct – встроенная функция.

Ключевое слово inline НЕ является указанием компилятору, а является лишь рекомендацией. Если функция слишком большая, то компилятор проигнорирует inline и скомпилирует эту функцию, как и обычную

4. Структуры:

Структура – объединение простых переменных с некими стандартными типами: int, float, double и т.д. Создаются для объединения неких данных. Типы переменных, присутствующих в структуре могут быть как различны, так и одинаковы.

Рассмотрим пример:

```
#include <stdio.h>

struct Bookcase {
    int number_of_bookshelf;
    char the_name_of_book [10];
    float width;
    float hight;
};

int main( )
{
    struct Bookcase x = { 5, "IT", 25.9, 12.0 };
    struct Bookcase y;
    printf ( "input the name of book: " );
    scanf ( "%s", y.the_name_of_book );
    printf ( "1: the number of bookshelf: %d  the name of book: %s ",
                                        x.number_of_bookshelf,
                                        x.the_name_of_book );
    printf ( "\n2: the name of book: %s", y.the_name_of_book );
    return 0;
}
```

Мы создали структуру bookcase (книжный шкаф), в которой хранятся данные (поля). Это ее определение:

```
struct Bookcase {
    int number_of_bookshelf;
    char the_name_of_book [10];
    float width;
    float hight;
};
```

В строке structBookcasex = { 5, "IT", 25.9, 12.0 } – структурная переменная инициализируется, а в struct Bookcase y–определение структурной переменной.

Члены структуры заключены в фигурные скобки. Структура (в нашем случае) хранит данные о книжном шкафе: ширина и высота книжного шкафа, название книги и номер книжной полки (для упрощения структуры, предположим, что одна книжная полка = одной книге).

Доступ к полям структуры возможен благодаря применению операции точки.

4.1 Структуры в памяти:

Обратите внимание на следующие структуры, они являются анонимными, т.е. имена этих структур не смогут использоваться в дальнейшем. Иначе говоря, у нас будет существовать лишь 1 экземпляр данных структур:

```
struct Test1 {
    char a;
    char b;
    int c;
} A;
struct Test2 {
    int x;
    int y;
} B;
struct Test3 {
    char a;
    char b;
    char c;
    int d;
} C;
```

А теперь вопрос: как вы думаете, сколько выделяется памяти для переменных этих структур? 6-для A, 8-для B, 7-для C ?

Нет. Все для всех этих структур будет выделено 8 байт. Правило состоит в том, что если структура занимает меньше 4 байт, то ее размер равен сумме размеров переменных, иначе же он должен быть кратен 4. Это делается для того чтобы, выровнять структуры в памяти, чтобы обеспечить удобную работу с ними. Однако кратность можно изменить с помощью директивы #pragma pack(push,n), где n-кратность памяти. (Что такое директивы будет рассказано чуть позже). Возможные параметры- степени двойки. Следует использовать эту директиву только тогда, когда памяти может не хватить, ведь при уменьшении расхода памяти увеличивается время работы со структурной переменной.

Пример:

```
#pragma pack(push, 1)

struct Str
{
    char a;
    int b;
};

#pragma pack(pop)
```

Обязательно ставьте #pragma pack(pop), иначе память программы может потерять свою структурированность, и может случиться что угодно (и вряд ли хорошее).

5. Перечисления

Это также еще один способ создания пользовательского типа данных. В отличие от структур этот способ используется, когда переменные создаваемого типа принимают известное конечное множество значений.

Пример:

```
#include <stdio.h>

enum Months { Jan = 1, Feb, Mar, Apr, May,
              June, July, Aug, Sep, Oct, Nov, Dec };

int main( )
{
    months x,y;
    x = Jan;
    y = May;
    printf ( "Result: %d", y-x );
    return 0;
}
```

Перечисления перечисляют все возможные значения переменных создаваемого типа. Обычно перечисление начинается с 0, однако в данном случае мы определили Jan=1 и, соответственно, перечисление началось с 1.

Переменным, типом которых является заданное перечисление, нужно присваивать только те значения, которые указаны в перечислении.

Перечисляемые типы данных (как видно из примера выше) разрешают применение основных арифметических операций.

Также перечисление удобно использовать как замена типу bool. Выглядит это так:

enum click {No, Yes}

Именно так удобно записывать, так как в таком случае No=0 и Yes=1, как и в bool.

Практическая работа №2

1. Посчитайте факториал переменной, которая вводится с клавиатуры.

   Примечание: n! = 1*2*3*…*n

2. У вас есть комната, у нее есть параметры – длина и ширина. Посчитайте ее площадь и периметр. Используйте структуры и функции.

3. Номер телефона можно разделить на 3 части: код города, номер телефонной станции и номера абонента. Напишите программу, которая хранит эти части телефонного номера, ищет соответствие по какой-то из частей номера и выводит этот номер на экран.

4. У вас есть декартова система координат, в которой, как известно, существуют 2 координаты – x,y. Пользователь задает вектор его координатами (1 координата – начало вектора, 2 координата – его конец), ваша программа должна вывести его модуль. Указание: использовать структуры.

5. Мы высчитываем время в часах, минутах и секундах. Создать код, который бы высчитывал сумму и вычитание двух значений времени. Указание: делать с помощью структур и функций

6. Вычислить заданную функциюy = 45*e^(2x^2) в диапазоне [-5,5] с шагом ½.

7. Понять, когда данное выражение выполняться не будет

$$\left|\sin(x)_n - \sin(x)_{n-1}\right| < A$$

Где A – некая бесконечно малая окрестность (вводится пользователем). Вывести n, при котором это уже не выполняется.

Указание: Нужно знать, как раскладывается sinx, чтобы решить эту задачу. В точке x = 0, эта функция раскладывается так:

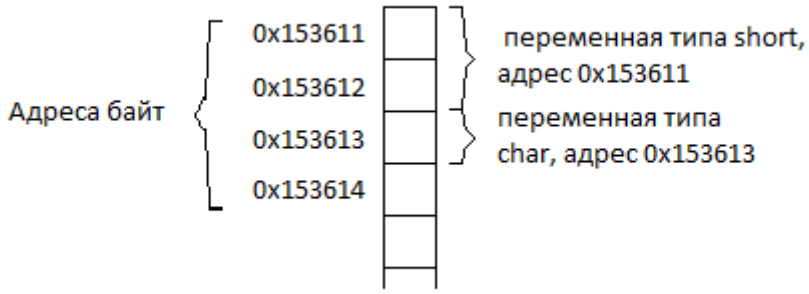
$$\sin(x)=\sum_{0}^{inf}\frac{-1^{n}*x^{2n+1}}{(2n+1)!}$$

6. Указатели

Указатели являются характерной особенностью языка Си. Эта особенность является одновременно и преимуществом языка, и его проблемой. Здесь будут рассмотрены лишь необходимые основы, поскольку формат пособия не предполагает углубленного изучения.

Насколько вы знаете, память компьютера представляет собой набор из битов, но это очень маленькая ячейка, и для повышения быстродействия и по ряду технических причин, наименьшей доступной ячейкой в памяти является байт. Все байты имеют адрес, по которому к ним происходит обращение.

Программа, загружаясь в память, занимает некоторое количество этих адресов. Следовательно, каждая переменная или функция начинается с определенного адреса. Небольшой рисунок для понимания:



Видно, что, переменная типа short занимает 2 байта, а типа char 1 байт.

Объявление указателя происходит таким образом:

```
int *pointer;
```

Где pointer – указатель, который хранит адрес ячейки памяти, в которой лежит целое число (int).

Есть два способа использования указателя:

- Использовать имя указателя без символа *, таким образом можно получить фактический адрес ячейки памяти, куда ссылается указатель.

- Использовать имя с указателем с символом *, это позволит получить значение, хранящееся в памяти. В рамках указателей, у символа * есть техническое название – операция разыменования. По сути, мы принимаем ссылку на какой-то адрес памяти, чтобы получить фактическое значение.

```
#include <stdio.h>

int main ( ) {
    int cell;
    int *pointer;
    pointer = &cell;
    cell = 100 + 26;
    printf ( "%d", *pointer );
    return 0;
}
```

В строчке pointer = &cellмы получаем адрес ячейки, куда ссылается указатель. В 8 строчке *pointer– это процесс разыменования. 8 строчка выводит 126 на экран.

Указатели, инициализированные значением 0, называются нулевыми и ни на что не указывают. Бывают случаи, когда нас интересует просто значение адреса, а не тип указываемого объекта. С этой целью в язык C++ введены указатели void *,способные хранить адрес любого объекта. В отличие от типизированных указателей, рассмотренных выше, указатели void* - нетипизированные.

Если p является указателем на структуру s с полем a, то оператор -> в записи p->a используется для сокращения записи (*p).a. Например:

```
#include <stdio.h>

struct Point {
    double x;
    double y;
};

int main ( ) {
    struct Point t;
    struct Point *pt = &t;
    pt->x = pt->y = 1;
    printf ( "%f, %f", pt->x, pt->y );
    return 0;
}
```

Попытка разыменования нулевого указателя приводит к ошибке при выполнении программы, попытка же разыменования указателя void * вызовет ошибку на этапе компиляции.

6.1 Выделение памяти

Конечно, вышеперечисленная информация только знакомит с концепцией указателей. Однако сейчас разберем самое частое использование указателей (и основную идею, для которой они были придуманы) – работа с памятью.

В памяти компьютера есть особая область, называемая кучей (англ. heap), которая позволяет выделять память динамически, т.е. во время работы программы (Строго говоря, есть еще 2 типа хранения памяти стек и непосредственно тело исполняемого модуля). Для этого существует специальная функция – malloc(). То, как она работает проще понять на примере программы, подсчитывающей среднее арифметическое у массива произвольной длины:

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

int main ( )
{
    int *array;
    int quantity;
    int i;
    double average = 0;
    printf ( "Enter the size of array: " );
    scanf ( "%i", &quantity );
    array = ( int * ) malloc ( quantity );
    printf ( "\nEnter the each element: " );
    for ( i = 0; i < quantity; i++ ) {
        printf ( "\n" );
        scanf ( "%i", &array [i] );
        average = average + ( double ) array [i];
    };
    printf ( "%e", average / quantity );
    free ( array );
    return 0;
}
```

Первую строчку main ( ) можно переписать так:

```
int *array, quantity, i;
```

Эта«строчка»main(), это пример того что для того чтобы объявить
указатели друг за другом через запятую, нужно использовать * перед именем каждого указателя, иначе (как в данном случае) вы получите один указатель на целый тип и несколько переменных целого типа.

Делать так не следует, так как человеку, читающему ваш код, будет труднее понять его листинг.

Как видно для использования перед malloc нужно в скобках ставить тип указателя, для которого нужно выделить память, а аргументом нужно
указывать количество необходимых ячеек.

Честно говоря, функция free(), здесь не нужна, так как по завершению программы память освободится автоматически, но в больших проектах об этом не стоит забывать.

Возможно, вы удивились, увидев здесь выражение (double)array[i], данная конструкция является явным преобразованием типов. О ней будет рассказано в следующем пункте.

Примечание.

При выделении память для двумерного массива используется следующая конструкция:

```
int **ptr = ( int** ) malloc ( size1 );
for( i = 0; i < size1; i ++ ) {
    ptr [i] = ( int* ) malloc ( size2 );
}
```

А для освобождения обратный порядок:

```
for ( i = 0; i < size1; i ++ ) {
   free ( ptr [i] );
}
 free ( ptr );
```

Возможно, вы удивились, увидев здесь операцию [], но как уже было сказано массив является указателем (константным, его адрес нельзя изменить), а конструкция a[i] это не что иное, как измененный вид конструкции *(a+i), по этой причине индексация массива начинается с нуля, ведь имя массива-указатель на его первый элемент.

Следующая функция для выделения памяти calloc достаточно похожа на malloc, единственное различие, что при применении calloc ячейки памяти, которые она выделяет заполняются нулями, так же различается количеством аргументов, здесь добавляется вес переменной в байтах:

```
array = ( int * ) calloc ( quantity, sizeof ( int ) );
```

В остальном же если подставить это в предыдущую программу, то ничего не изменится.

Говоря о выделении памяти нельзя также не сказать о функции, realloc(). Она предназначена для перераспределения памяти. Например, если у вас есть некая выделенная память под процесс, но может случится так, что этой памяти не хватит, и тогда будет достаточно просто с помощью realloc() выделить дополнительную память.

Следующая программа дает возможность ввести строку неограниченной длины до символа ‘=’:

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

int main ( )
{
    int i = 0;
    char *str1;
    char symbol;
    str1 = ( char * ) malloc ( sizeof ( char ) );
    do {
        symbol = getchar ( );
        str1 = ( char * ) realloc ( str1, ( i+2 ) );
        str1 [i] = symbol;
        i ++;
    } while ( symbol != '\n' );
    str1 [i] = '\0';
    printf ( "The string is: %s", str1 );
    return 0;
}
```

6.2. Явное и неявное преобразование типов.

В языках С и С++ преобразование одного типа в другой называется приведением типа. Оно бывает явным и неявным:

- При неявном приведении преобразование происходит автоматически

- При явном приведении перед выражение, необходимо использовать оператор приведения типа static_cast<>.

```
int *pi1 = static_cast < int* > ( pi );
```

Для языкаС++. В языке Сэтого оператора нет!

В некоторых старых компиляторах оператор static_cast отсутствует, поэтому приходится использовать приведение типа в старом стиле

```
int *pi1 = ( int * ) pi;
```

Работает и в С, и в С++

В новой редакции С++ старый способ приведения заменен на четыре новых, различающихся по степени безопасности. О них будет рассказано в разделе С++.

6.3 Указатели на функции

Хотя функция - это не переменная, она по-прежнему имеет физическое положение в памяти, которое может быть присвоено указателю. Адрес, присвоенный указателю, является входной точкой в функцию. Далее этот указатель можно использоваться для вызова функции.Адрес функции получается при использовании имени функции без каких-либо скобок или аргументов.

```
#include <stdio.h>

int nod ( int, int );

int main ( )
{
    int ( *ptrnod ) ( int, int ) = NULL;
    int a;
    int b;
    int result;
    ptrnod = nod;
    printf ( "Enter first number: " );
    scanf ( "%i", &a );
    printf ( "\nEnter second number: " );
    scanf ( "%i", &b );
    result = ptrnod ( a, b );
    printf ( "The greatest common divisor: %i", result );
    return 0;
}
```

```
int nod ( int number1, int number2 )
{
    if ( number2 == 0 ) {
        return number1;
    }
    return nod ( number2, number1 % number2);
}
```

Как вы можете видеть для объявления указателя на функцию вам достаточно знать лишь прототип функции, т. е. список параметров и тип возвращаемого значения. Кроме того, при объявлении указателя на функцию имя указателя вместе с символом * заключаются в скобки.

6.4. Арифметические действия над указателями

Над указателями можно проводить только такие действия, как сложение и вычитание. Предполагается, что если указатель p относится к типу T *, то p указывает на элемент некоторого массива типа T. Тогда p + 1 является указателем на следующий элемент этого массива, а p - 1 – указателем на предыдущий элемент. Аналогично определяются выражения p + n, n + p и p - n, а также действия p++, p–, ++p, –p, p += n, p -= n, где n – целое число. Важно отметить, что арифметические действия с указателями выполняются в единицах того типа, к которому относится указатель. То есть p + n, преобразованное к целому типу, содержит на sizeof(T)* n большее значение, чем p. Из равенства p + n == p1 следует, что p1 - p == n. Именно так вводится оператор разности двух указателей: его значением является целое, равное количеству элементов массива от p до p1. Отметим, что это – единственный случай в языке, когда результат бинарного оператора с операндами одного типа принадлежит к принципиально другому типу. Сумма двух указателей не имеет смысла и поэтому не определена. Не определены также арифметические действия над нетипизированными указателями void *. Наконец, все указатели, в том числе и нетипизированные, можно сравнивать, используя операторы отношения:>, <, >=, <=, ==, !=.

Пример (сумма элементов массива):

```
#include <stdio.h>

int main ( )
{
     int a[10];
     int sum;
     int i;
     int *p;
     for  ( i = 0; i < 10; i++ ) {
         scanf  (" %d", &a[i] );
     }
     for ( p = &a[0]; p < &a[10]; p++) {
        sum += *p;
     }
     printf ( "%d", sum );
     return 0;
}
```

Указатели и массивы тесно взаимосвязаны. Имя массива может быть неявно преобразовано к константному указателю на первый элемент этого массива. Так, &a[0] равноценно a. Вообще, верна формула a[n] == a + n; то

есть адрес n-того элемента массива есть увеличенный на n элементов указатель на начало массива. Разыменовывая левую и правую части, получаем основную формулу, связывающую массивы и указатели:

```
a [n] == *( a + n );
```

Данная формула, несмотря на простоту, требует нескольких пояснений. Во-первых, компилятор любую запись вида a[n] интерпретирует как *(a + n). Во-вторых, формула поясняет, почему в C, C++ массивы индексируются с нуля, и почему нет контроля выхода за границы диапазона. Наконец, используя эту формулу, мы можем записать следующую цепочку равенств:

```
a[n] == *(a + n)== *(n + a)== n[a];
```

Таким образом, элемент массива a с индексом 2 можно обозначить не только как a[2], но и как 2[a]

7. Массивы и строки

7.1 Одномерные массивы

Массив — это структура данных, представленная в виде группы ячеек одного типа, объединенных под одним единым именем. Массивы используются для обработки большого количества однотипных данных. Имя массива является указателем. Что такое указатели будет рассказано немного позже. Отдельная ячейка данных массива называется элементом массива. Элементами массива могут быть данные любого типа. Массивы могут иметь как одно, так и более одного измерений. В зависимости от количества измерений массивы делятся на одномерные массивы, двумерные массивы, трёхмерные массивы и так далее до n-мерного массива. Ниже можно увидеть программу, демонстрирующую основные аспекты определения и объявления массивов:

```
#include<stdio.h>

int main ( )
{
    int array[10];
    int i;
    for ( i = 0; i < 10; i++) {
        array [i] = i;
        printf ( "%d ", array[i] );
     }
     return 0;
}
```

Строка intarray[10] – объявляет массив, состоящий из 10 элементов.

Из примера видно, что для использования массива, необходимо его объявить, т.е. указать тип ячеек, имя массива и количество элементов. Далее нужно обращаться к элементам используя имя и индекс. Заметьте, что нумерация начинается с 0, т.е для обращения к 1 элементу в скобках вам нужно указать 0, следовательно для n-нного элемента n-1.

Массив также можно инициализировать вручную при объявлении следующим образом:

```
int array[5] = { 1, 2, 3, 4, 5 };
```

Если вы не указываете первоначальное значение для какого-либо элемента массива, большинство компиляторов C++ будут инициализировать такой элемент нулем.

Например, следующее объявление инициализирует первые три из пяти элементов массива

```
int values[5] = { 100, 200, 300 };
```

7.2 Двумерные массивы

Работа с двумерными массивами ничем не отличается от работы с одномерными, можете сами убедится:

```
#include <stdio.h>

int main ( )
{
    int array[2][3] = { { 1, 3, 5 }, { 2, 4, 6 } };
    int i;
    int j;
    for ( i = 0; i < 2; i++ ) {
        for ( j = 0; j < 3; j++ ) {
            printf ( "%d ", array[i][j] );
        }
    }
    return 0;
}
```

При передаче функции массива как аргумента, копия объекта не создается, и работаем мы с тем же массивом. Связано это с тем, что имя массива - это указатель.

```
#include <stdio.h>

void func ( int array[2][3] )
{
    int i;
    int j;
    for ( i = 0; i < 2; i++ ) {
        for ( j= 0; j < 3; j++ ) {
            printf ( "%d ", array[i][j] );
        }
    }
};

void gunc( int array[2][3] )
{
    int i;
    int j;
    for ( i = 0; i < 2; i++ ) {
        for ( j = 0; j < 3; j++ ) {
            scanf ( "%d ", &array[i][j] );
        }
    }
};


 int main ( )
 {
     int array[2][3];
     gunc ( array );
     func ( array );
     return 0;
 }
```

Откровенно говоря, при определении/объявлении кроме имени нужно указать количество элементов только в первых скобках, вторые можно оставить пустыми. Но чтобы не путаться, лучше писать все.

7.3 Строки

Особым видом массива в С является строка. Она представляет собой массив типа char, завершающийся нулевым символом- ‘\0’. Также строки могут объявляться иначе чем массивы другого типа:

```
#include <stdio.h>

int main ( )
{
    char string [ ] = "This is string. ";
    char string1 [11] = {'I','t',' ','i','s',' ','t','o','o','.','\0'};
    char string2 [11];
    printf ( "Enter the string (no more than 10 symbols)\n" );
    scanf ( "%s", &string2 );
    printf ( "%s %s %s", string, string1, string2);
    return 0;
}
```

Функции для манипулирования С – строками объявлены в заголовочном файле string.h (или по стандарту 1998 г. в cstring).

Прототипы основных функций для работы со строками:

```
char *strcpy ( char *dst, const char *src );
```

Она копирует строку src в строку dst, включая нулевой символ, и возвращает указатель на строку dst. При этом выход за границы массива dst не контролируется:

Для определения длины строки служит функция strlen с прототипом:

```
size_t *strlen ( const char *src );
```

Для добавления одной строки к другой используется функция strcat с прототипом

```
char *strcat ( char *dst, const char *src );
```

Функция strcmp предназначена для лексикографического сравнения строк. Она имеет прототип

```
int strcmp ( const char *p, const char *q );
```

Практическая работа №3:

Строки:

1. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Циклически сдвинуть все слова влево на k слов, удалив при этом лишние пробелы (k заведомо меньше количества слов).

2. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Циклически сдвинуть все слова вправо на k слов, удалив при этом лишние пробелы (k заведомо меньше количества слов).

3. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Скопировать в новую строку два самых коротких слова исходной строки. Алгоритм просмотра исходной строки должен быть однопроходным.

4. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале, в конце строки и между словами). Сформировать новую строку, в которой содержатся все слова-перевертыши (палиндромы) исходной строки. Алгоритм просмотра исходной строки должен быть полуторапроходным (полпрохода на проверку того, является ли слово перевертышем).

5. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами), а также целочисленный массив перестановок слов. По данной строке и массиву перестановок сформировать новую строку, удалив при этом лишние пробелы. Например, если задана строка « aa bbb c dd eeee» и массив

перестановок 5 2 4 3 1, то итоговая строка должна иметь вид: «eeee bbb dd c aa». Указание: вначале сформировать массив указателей на начала слов.

6. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Сформировать строку, в которой слова из исходной строки упорядочены по алфавиту, удалив при этом лишние пробелы. Указание: для сравнения строк можно воспользоваться библиотечной функцией strcmp(s, s1).

7. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале, в конце строки и между словами). Сформировать строку, в которой слова из исходной строки упорядочены по длине (а при равной длине порядок их следования остается таким же, как и в исходной строке), удалив при этом лишние пробелы

8. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале, в конце строки и между словами). Сформировать строку, в которой слова из исходной строки упорядочены по количеству гласных (а при равном количестве гласных порядок их следования остается таким же, как и в исходной строке), удалив при этом лишние пробелы.

9. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Сформировать строку, в которой удалены лишние пробелы и повторявшиеся ранее слова. Порядок слов не менять.

10. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Сформировать строку, в которой слова упорядочены по повторяемости. Дубликаты слов следует удалить. При одинаковой повторяемости первым должно следовать слово, первое вхождение которого встречается раньше в исходной строке.

11. Вводится строка слов, разделенных пробелами (возможны лишние пробелы в начале и в конце строки и между словами). Выдать таблицу слов и количество их повторений в строке. Дубликаты слов не выдавать.

12. Вводится строка. Заменить в ней все цифры их словесными обозначениями: 0 на «zero», 1 на «one», 2 на «two» и т.д.

13. Удалить из строки s каждое вхождение подстроки s1.

14. Заменить в строке s каждое вхождение подстроки s1 на подстроку s2.

15. Даны две строки. Найти индексы (их может быть несколько) и длину самого длинного одинакового участка в обоих массивах. Например, в строках «abracadabra» и «sobrat» самым длинным одинаковым участком будет «bra».

16. Дан массив строк. Сформировать по нему массив подстрок, удовлетворяющих условию, передаваемому в качестве параметра (условие должно представлять собой функцию, принимающую параметр char * и возвращающую значение логического типа).

17. Реализовать функцию char *mystrstr(const char *p, const char *q); возвращающую первое вхождение подстроки q в строку p (или 0, если подстрока не найдена).

18. Реализовать функцию char *mystrpbrk(const char *p, const char *q);| возвращающую указатель на первое вхождение в строку p какого-либо символа из строки q (или 0, если совпадений не обнаружено)

19. Реализовать функцию size_t mystrspn(const char *p, const char *q); возвращающую число начальных символов в строке p, которые не совпадают ни с одним из символов из строки q.

20. Реализовать функцию size_t mystrcspn(const char *p, const char *q); возвращающую число начальных символов в строке p, которые совпадают с одним из символов из строки q.21. Реализовать функцию char* mystrtok(const char *p, const char *q, char* t); пропускающую символы разделителей, хранящихся в строке q, считывающую первую лексему в строке p в строку t (до следующего символа разделителя или до конца строки) и возвращающую указатель на первый, не просмотренный символ.

Массивы

Считая, что переменная a описана как a[3][4], расшифруйте выражения (пример Указатели. Ссылки):

- *a;

- **a;

- *a+1

- (*a)[2];

- *(a[2])

- *a[2];

- *2[a];

- *1[a + 1];

- &a[0][0];

- 2[a][3].

1. Дан массив целых. Оформить функцию count_if, вычисляющую количество элементов в массиве, удовлетворяющих данному условию, передаваемому в качестве параметра (условие должно представлять собой функцию, принимающую параметр типа int и возвращающую значение логического типа).

2. Дан массив целых. Заполнить его, передавая в качестве параметра функцию, задающую алгоритм генерации следующего значения, вида int f(). Для генерации данная функция может запоминать значения, сгенерированные на предыдущем шаге, либо в глобальных переменных, либо в статических локальных переменных.

3. Дан массив целых, отсортированный по возрастанию. Удалить из него дубликаты.

4. Дан массив целых. Составить функцию remove_if, удаляющую из него все элементы, удовлетворяющие условию, передаваемому в качестве параметра.

5. Дан массив чисел и число a. Переставить элементы, меньшие a, в начало, меняя их местами с предыдущими. Порядок элементов, меньших a, а также порядок элементов, больших a, не менять.

6. Дан массив целых. Найти в нем пару чисел a и b с минимальным значением f(a, b), где f передается в качестве параметра.

7. Дан массив целых. Составить функцию accumulate, применяющую функцию f(s, a), передаваемую в качестве параметра, к каждому элементу a массива и записывающую результат в переменную s. С ее помощью найти минимальный элемент в массиве, сумму и произведение элементов массива.

8. Дан массив целых. Сформировать по нему массив, содержащий длины всех серий (подряд идущих одинаковых элементов). Одиночные элементы считать сериями длины 1.

9. Дан массив целых. Из каждой серии удалить один элемент.

10. Дан массив целых. Удалить все серии, длина которых меньше k.

11. Слить n массивов целых, упорядоченных по возрастанию, в один, упорядоченный по возрастанию. Указание 1. Для упрощения алгоритма следует записать в конец каждого массива барьер – самое большое число соответствующего типа. Барьер, в частности, будет определять, где заканчиваются данные в массиве

8. Работа с файлами в С

При открытии файла с нами связывается поток ввода-вывода. Выводимая информация записывается в поток, вводимая информация считывается из потока.

Когда поток открывается для ввода-вывода, он связывается со стандартной структурой типа FILE, которая определена в <stdio.h>.

Открытие файла осуществляется с помощью функции fopen(), которая возвращает указатель на структуру типа FILE, который можно использовать для последующих операций с файлом.

```
FILE *f = fopen ( "name.txt", "type" );
```

name – имя файла, который вы открываете (включая путь до него)

type – способ доступа к файлу:

“r”– открыть файл для чтения (файл обязан существовать)

“w” – открыть файл для записи (файл должен быть пустым)

“a” – дописывает информацию к файлу

“rb” – открывает двоичный файл для чтения

“wb” – открывает двоичный файл для записи

“ab”- дополняет двоичный файл

“rt” – открывает текстовый файл (символьный) для чтения

“wt” – открывает текстовый файл (символьный) для записи. Файл должен быть пустым

“at” – дописывает информацию к текстовому символьному файлу

После окончания работы с файлом нужно всегда закрывать данный файл. В противном случае произойдет потеря данных. Почему? Будет объяснено чуть ниже.

```
fclose ( FILE* f );
```

Прототип функции, которая закрывает файл.

Почему произойдет потеря данных? Данные в файл попадают не сразу (из-за специфики строения жесткого диска), они попадают в буфер памяти (cash). И лишь cashпереполняется, они записываются в

файл. Поэтому, если мы не закроем файл, данные просто останутся в данном буфере памяти.

Предположим, что по какой-то причине, ваша программа, которая работает с файлами, рушится. Однако какая-то часть данных застряла в cash'е (предположим, что вы использовали функцию fprintf(), она работает аналогично printf(), только с файлом). То, что мы считали с клавиатуры с помощью fprintf(), останется в буфере памяти. Но программа опять же «упала», а данные в файл должны поместиться. Что делать?

Есть такая функция fflush (FILE *f), которая «проталкивает» данные в файл. Ниже приведен отрывок из кода, где она используется.

```
fprintf ( f, "%i", d);
fflush ( f );
```

Также рассмотрим несколько функций, которые тоже важны для работы с файлами:

1.

```
size_t fread ( void* p, size_tsize, size_tcount, FILE* f );
```

Данная функция считывает из файла f количество символов count размером size и помещает их в массив p. Возвращает количество считанных блоков размером (count*size). Возвращает тип size_t. Перемещает указатель в файле на количество считанных им символов.

Пример использования (многие еще неизвестные вам функции объясняются чуть ниже в этом же пункте):

```
#include <stdio.h>

int main ( )
{
    FILE *pFile;
    long lSize;
    char *buffer;
    size_t result;
    pFile = fopen ( "myfile.bin" , "rb" );
    if ( pFile == NULL ) {
        fputs ("File error", stderr);
        exit (1);
    }
```

//Получаем размер файла

```
fseek ( pFile , 0, SEEK_END );
lSize = ftell (pFile);
rewind (pFile);
```

//Выделяем память для хранения всего файла

```
buffer = ( char * ) malloc ( sizeof ( char ) *lSize );
if ( buffer == NULL ) {
    fputs ("Memory error", stderr);
    exit (2);
 }
```

//Копируем файл в буфер

```
result = fread ( buffer, 1, lSize, pFile) ;
if ( result != lSize ) {
   fputs ( "Reading error", stderr );
   exit (3);
}
```

/* Теперь весь файл находится в буфере */

// Завершение

```
fclose (pFile);
free (buffer);
return 0;
}
```

2.

```
size_t fwrite ( void *p, size_t size, size_t count, FILE* f );
```

Эта функция записывает в файл f количество символов count размером size из массива p. Оставшаяся часть аналогична fread(…);

Примериспользования:

```
#include <stdio.h>

int main ()
{
    FILE * pFile;
    char buffer [] = { 'x' , 'y' , 'z' };
    pFile = fopen ( "myfile.bin" , "wb" );
    fwrite (buffer , 1, sizeof(buffer) , pFile );
    fclose (pFile);
    return 0;
}
```

3.

```
int fseek ( FILE* stream, size_toffset, int place );
```

Данная функция помещает указатель в файловом потоке stream на место place, смещая его каждый раз на значение offset. У place есть 3 состояния: 0,1,2, которые соответствуют SEEK_SET, SEEK_CUR, SEEK_END – началу файла, текущей позиции и концу файла. В случае успеха функция возвратит нулевое значение.

Пример использования:

```
#include <stdio.h>

int main ( )
{
      FILE *pFile;
      pFile = fopen ( "example.txt" , "wb" );
      fputs ( "This is an apple." , pFile );
      fseek ( pFile , 9, SEEK_SET );
      fputs ( " sam" , pFile );
      fclose ( pFile );
      return 0;
}
```

4.

```
long ftell ( FILE* stream );
```

Возвращает значение указателя текущего положения потока.

Пример использования:

```
#include <stdio.h>

int main ( )
{
    FILE *pFile;
    long size;
    pFile = fopen ( "myfile.txt", "rb" );
    if (pFile == NULL) {
        perror ( "Error opening file" );
    } else {
        fseek ( pFile, 0, SEEK_END );
        size = ftell (pFile);
        fclose (pFile);
        printf ( "Size of myfile.txt: %ld bytes.\n", size );
    }
    return 0;
}
```

5.

```
void rewind ( FILE* stream );
```

Устанавливает внутренний указатель в начальное положение, то есть в начало файла. Эквивалентен fseek(…). Но в отличие от fseek(…) его возвращаемое значение void.

Пример использования:

```
#include <stdio.h>

int main ()
{
    int n;
    FILE *pFile;
    char buffer [27];
    pFile = fopen ("myfile.txt", "w+");
    for ( n = 'A' ; n <= 'Z' ; n++ )
        fputc ( n, pFile );
    rewind (pFile);
    fread (buffer, 1, 26, pFile);
    fclose (pFile);
    buffer [26] = '\0';
    puts (buffer);
    return 0;
}
```

6.

```
int fgetpos ( FILE* stream, fpos_t* pos );
```

Функция получения текущего положения в потоке stream. Результат помещается в pos. 0 – в случае успеха, иначе отличное от 0 значение и записывается в переменную errno.

Пример использования:

```c
#include <stdio.h>

int main ( )
{
    FILE *pFile;
    int c;
    int n;
    fpos_t pos;
    pFile = fopen ( "myfile.txt", "r" );
    if ( pFile == NULL ) {
        perror ( "Error opening file" );
    } else {
        c = fgetc (pFile);
        printf ( "1st character is %c\n", c );
        fgetpos ( pFile, &pos );
        for ( n = 0; n < 3; n++ ) {
            fsetpos (pFile, &pos);
            c = fgetc (pFile);
            printf ( "2nd character is %c\n", c );
         }
         fclose (pFile);
    }
    return 0;
}
```

7.

```c
int fsetpos ( FILE* stream, const fpos_t* pos );
```

Устанавливает позицию pos в потоке stream. Возвращаемый результат аналогичен fgetpos(…)

Пример использования:

```c
#include <stdio.h>

int main ( )
{
    FILE *pFile;
    fpos_t position;
    pFile = fopen ( "myfile.txt", "w" );
    fgetpos ( pFile, &position );
    fputs ( "That is a sample", pFile );
    fsetpos ( pFile, &position );
    fputs ( "This", pFile );
    fclose (pFile);
    return 0;
}
```

8.

```
int feof ( FILE *stream );
```

Проверяет, достигнут ли конец файла, связанного с потоком stream

Пример использования:

```
#include <stdio.h>

int main ( )
{
    FILE *pFile;
    long n = 0;
    pFile = fopen ( "myfile.txt", "rb" );
    if (pFile == NULL) {
        perror ("Error opening file");
    } else {
        while ( !feof(pFile) ) {
            fgetc (pFile);
            n++;
        }
        fclose (pFile);
        printf ("Total number of bytes: %d\n", n - 1);
    }
    return 0;
}
```

9.

```
int rename ( const* char oldname, const* char newname );
```

Меняет название файла с oldname на newname.Возвращает 0, если удалось переименовать, иначе возвращает любое ненулевое значение и устанавливает переменную err no.

Пример использования:

```
#include <stdio.h>

int main ( )
{
    int result;
    char oldname [ ] = "oldname.txt";
    char newname [ ] = "newname.txt";
    result= rename ( oldname , newname );
    if ( result == 0)
        puts ( "File successfully renamed" );
    else
        perror ( "Error renaming file" );
    return 0;
}
```

10.

```
int fscanf ( FILE *stream, const char *format, arg-list );
int fprintf ( FILE * stream, const char * format, arg-list );
```

Работают аналогично scanf() и printf().

Функция fscanf ( ) возвращает количество аргументов, которым действительно были присвоены значения. В их число не входят пропущенные поля (пробелы). Возврат EOF означает, что при чтении была сделана попытка пройти маркер конца файла.

Функция fprintf ( ) выполняет форматированный вывод в поток. Записывает в указанный поток последовательность символов в формате, указанном аргументом format. После параметра format, функция ожидает, по крайней мере, многие дополнительные аргументы, как указано в прототипе.

Пример использования одной из них:

```
#include <stdio.h>

int main ()
{
    FILE *pFile;
    int n;
    char name[100];
    pFile = fopen ("myfile.txt", "w");
    for ( n = 0; n < 3; n++ ) {
        puts ("please, enter a name: ");
        gets (name);
        fprintf (pFile, "Name %d [%-10.10s]\n", n, name);
    }
    fclose (pFile);
    return 0;
}
```

11.

```
int fputs ( const char* string, FILE* stream );
```

Кладет в поток stream строку string, введенную с клавиатуры

Пример использования:

```c
#include<stdio.h>

int main()
{
    FILE *ptrFile = fopen ( "log.txt", "a" );
    char sentence [256];
    printf ( "Please enter a string to put it into the file: " );
    fgets ( sentence, 255, stdin );
    fputs ( sentence, ptrFile );
    fclose ( ptrFile );
    return 0;
}
```

Полагаем, что данный кусок кода требует небольших пояснений:

8 строчка записывает строку из стандартного потока ввода stdin в символьный массив sentence.

9 строка добавляет строку в файл.

12.

```c
char* fgets ( char* string, int num, FILE* stream );
```

Получает символы из файлового потока stream и сохраняет их в строку string, до тех пор, пока не наступит конец файла или строки. num– максимальное количество символов доступных для чтения. Нулевой символ включается!

Пример использования:

```c
#include <stdio.h>

int main ( )
{
    FILE *ptrFile = fopen ( "file.txt" , "r" );
    char mystring [100];
    if (ptrFile == NULL)
        perror("Error with opening the file");
    else {
        if ( fgets(mystring, 100, ptrFile) != NULL )
            puts(mystring);
        fclose (ptrFile);
    }
    return 0;
}
```

13.

```
FILE* freopen ( const char* fname, const char* mode, FILE *stream );
```

Используется для связывания существующего потока с новым файлом. Новое имя файла указывается аргументом fname, режим доступа — аргументом mode, а поток, который надо связать с новым именем, — аргументом stream

Пример использования:

```
#include <stdio.h>

int main ( )
{
    freopen ( "myfile.txt", "w", stdout );
    printf ("This sentence is redirected to a file.");
    fclose (stdout);
    return 0;
}
```

9. Препроцессор

9.1 Включение файла

Включение файлов (помимо других полезных вещей) позволяет легко управлять наборами директив #define и объявлений. Любая строка вида

```
#include "filename"
```

или

```
#include <filename>
```

заменяется содержимым файла с именем filename. Если имя файла заключено двойные кавычки, то, как правило, файл ищется среди исходных файлов программы; если такового не оказалось, или имя файла заключено в угловые скобки < и >, то поиск осуществляется по определяемым реализацией правилам. Включаемый файл сам может содержать в себе строки #include. Часто исходные файлы начинаются с нескольких строк #include, ссылающихся на файлы, содержащие общие директивы #define, объявления extern или прототипы нужных библиотечных функций из заголовочных файлов вроде <stdio.h>. (Строго говоря, эти включения не обязательно являются файлами; технические детали того, как осуществляется доступ к заголовкам, зависят от конкретной реализации.) Директива #include — хороший способ собрать вместе объявления большой программы. Он гарантирует, что все исходные файлы будут пользоваться одними и теми же определениями, и объявлениями переменных, благодаря чему предотвращаются особенно неприятные ошибки. Естественно, при внесении изменений во включаемый файл все зависимые от него файлы должны перекомпилироваться.

9.2 Макроподстановка

Директива макроподстановки имеет вид:

#define имя-замещающий-текст

Макроподстановка используется для простейшей замены: во всех местах, где встречается имя, вместо него будет помещен замещающий текст. Имена в #define задаются по тем же правилам, что и имена обычных переменных. Замещающий текст может быть произвольным. Обычно замещающий текст целиком помещается в строке, в которой расположено слово #define, однако длинные определения можно разбивать на несколько строк, поставив в конце каждой продолжаемой строки обратную наклонную черту *n*. Область видимости имени, определенного директивой #define, простирается от определения до конца файла. В определении макроподстановки могут использоваться предшествующие ему макроопределения. Подстановка осуществляется только для тех имен, которые расположены вне текстов, заключенных в кавычки и не являются частью другого слова. Например, если YES определено с помощью директивы #define, то никакой подстановки в printf(”YES”) или в YESMAN выполнено не будет. Любое имя можно определить с произвольным замещающим текстом.

#define forever for(;;) /*бесконечный цикл*/

К примеру, он определяет новое слово forever для бесконечного цикла. Можно определить макрос с аргументами, чтобы замещающий текст варьировался в зависимости от задаваемых параметров. Например, определим max следующим образом:

#define max (A, B) ((A) > (B) ? (A) : (B))

Хотя обращения к max выглядят как обычные обращения к функции, они будут вызывать только текстовую замену. Каждый формальный параметр (в данном случае A и B)будет заменяться соответствующим ему аргументом. Так, строка

```
x = max ( p + q, r + s );
```

будет заменена на строку

```
x = ( (p + q) > (r + s) ? (p + q) : (r + s) );
```

Поскольку аргументы просто подставляются в текст, указанное определение max подходит для данных любого типа, так что не нужно писать различные варианты max для данных разных типов, как это было бы в случае определения функции. Если вы внимательно проанализируете работу max, то обнаружите некоторые подводные камни. Выражения вычисляются дважды, и если они вызывают побочный эффект (из-за инкрементных операций или функций ввода-вывода), это может привести к нежелательным последствиям. Например,

```
max (i++, j++)   /*It's wrong!*/
```

вызовет увеличение i и j дважды. Кроме того, следует позаботиться о скобках, чтобы обеспечить нужный порядок вычислений. Задумайтесь, что случится, если при определении

#define square(x) x * x/* НЕВЕРНО*/

вызвать square (z + 1). Тем не менее макросы имеют свои достоинства. Один из примеров можно найти в файле <stdio.h>, где getchar и putchar часто реализуют с помощью макросов, чтобы избежать расходов времени на вызов функции для каждого обрабатываемого символа. Функции в <ctype.h> обычно также реализуются помощью макросов. Действие #define можно отменить с помощью директивы #undef:

#undef getchar

```
int getchar ( void ) { ... }
```

Как правило, это делается, чтобы заменить макроопределение настоящей функцией с тем же именем. Имена формальных параметров не заменяются, если встречаются заключенных в кавычки строках. Однако, если в замещающем тексте перед формальным параметром стоит знак #, этот параметр будет заменен на аргумент, заключенный в кавычки. Это можно сочетать с конкатенацией строк, например, чтобы создать макрос отладочного вывода:

#define dprint(expr) printf(#expr ” = %g\n”, expr)

обращение к

```
dprint (x/y);
```

развернется в

```
printf ("x/y" " = %g\n", x/y);
```

в результате конкатенации двух соседних строк, которая будет автоматически выполнена компилятором, получим

```
printf ("x/y = %g\n", x/y);
```

Внутри фактического аргумента каждый знак” заменяется на *n*”, а каждая обратная наклонная черта *n* на *nn*, так что в результате подстановки получается правильная символьная константа. Оператор ## позволяет конкатенировать аргументы в макрорасширениях.

Если в замещающем тексте параметр соседствует с ##, то он заменяется соответствующим ему аргументом, а оператор ## и окружающие его символы-разделители выбрасываются.

Например, в макроопределении paste конкатенируются два аргумента

```
#define paste(front, back) front ## back
```

так что запись paste(name, 1) будет заменена на name1.

9.3. Условная компиляция

Работой препроцессора можно управлять с помощью условных инструкций. Они представляют собой средство для выборочного включения того или иного текста программы в зависимости от значения условия, вычисляемого во время компиляции. Вычисляется константное целое выражение, заданное в строке #if.

Это выражение не должно содержать операторы sizeof, приведения типов и констант из перечислений enum. Если оно имеет ненулевое значение, то будут включены все последующие строки вплоть до ближайшей директивы #endif, #elif, или #else.(Директива препроцессора #elif действует как else if.) Выражение defined(имя) #if равно 1,если имя было определено, и 0 в противном случае. Например, чтобы застраховаться от повторного включения заголовочного файла hdr.h, его можно оформить следующим образом:

```
#if !defined(HDR)
#define HDR
/* здесь содержимое hdr.h */
#endif
```

При первом включении файла hdr.h будет определено имя HDR, а при последующих включениях препроцессор обнаружит, что имя HDR уже определено, и перескочит сразу на #endif. Этот прием может оказаться полезным, когда нужно избежать многократного включения одного и того же файла. Если им пользоваться систематически, то в результате каждый заголовочный файл будет сам включать заголовочные файлы, от которых он зависит, освободив от этого занятия пользователя. Вот пример цепочки проверок имени SYSTEM, позволяющей выбрать нужный файл для включения:

```
#if SYSTEM == SYSV
#define HDR ”sysv.h”
#elif SYSTEM == BSD
#define HDR ”bsd.h”
#elif SYSTEM == MSDOS
#define HDR ”msdos.h”
```

```
#else
#define HDR ”default.h”
#endif
#include HDR
```

Инструкции #ifdef и #ifndef специально предназначены для проверки того, определено или нет заданное в них имя. И, следовательно, первый пример, приведенный выше для иллюстрации #if, можно записать и в таком виде:

```
#ifndef  HDR
#define HDR
/* здесь содержимое hdr.h */
#endif
```

9.4 Директива #pragma

Директивы pragma определяют функции компилятора для конкретного компьютера или операционной системы.

#pragma string

Каждая реализация C и C++ поддерживает некоторые функции, уникальные для хост-компьютера или операционной системы. Например, некоторые программы должны осуществлять точный контроль над областями памяти, в которых размещаются данные, или управлять способом получения параметров некоторыми функциями. Директивы #pragma позволяют компилятору использовать особенности конкретного компьютера или операционной системы, сохраняя при этом совместимость языков и C++. Директивы pragma характерны для конкретного компьютера или операционной системы по определению и обычно отличаются для каждого компилятора. Директивы pragma можно использовать в условных операторах для обеспечения новой функциональности препроцессора или для предоставления компилятору сведений, определенных реализацией.

string ——это последовательность символов, которые предоставляют определенную инструкцию компилятора и аргументы, если таковые имеются. Символ решетки (#) должен быть первым отличным от пробела символом в строке, которая содержит директиву pragma; символы пробела могут разделять знак числа и слово pragma. После #pragma введите любой текст, который преобразователь может проанализировать как токены предварительной обработки. Аргумент #pragma подлежит расширению макроса. Если компилятор обнаруживает нераспознаваемую директиву pragma,он выдает предупреждение и продолжает компиляцию.

Компиляторы Microsoft C и C++ распознают следующие директивы pragma.

alloc_text	auto_inline	bss_seg
check_stack	code_seg	comment
component	conform	const_seg
data_seg	deprecated	detect_mismatch
fenv_access	float_control	fp_contract
function	hdrstop	include_alias
init_seg	inline_depth	inline_recursion
intrinsic	loop	make_public
managed	сообщение	omp
once	optimize	pack
pointers_to_members	pop_macro	push_macro
region, endregion	runtime_checks	section
setlocale	strict_gs_check	unmanaged
vtordisp	warning	

9.5 Директива#pragma pack

Задает выравнивание упаковки для членов структуры, объединения и класса.

```
#pragma pack( [ show ] | [ push | pop ] [, identifier ] , n)
```

Директива pack обеспечивает контроль на уровне объявления данных. В этом состоит отличие от параметра компилятора /Zp, который предоставляет контроль только на уровне модуля. Директива pack действует на первое объявление struct, union или class после этой директивы #pragma. Директива pack не действует на определения. При вызове директивы pack без аргументов для параметра n задается значение, указанное в параметре компилятора /Zp. Если этот параметр компилятора не указан, по умолчанию используется значение 8. При изменении выравнивания структурыона может занимать меньше места в памяти, но возможно снижение производительности или даже возникновение аппаратного исключения для не выровненного доступа. Поведение этого исключения можно изменить с помощью директивы SetErrorMode.show(необязательно) —Отображает текущее байтовое значение выравнивания упаковки. Значение отображается в предупреждении. push(необязательно) — Помещает текущее значение выравнивания упаковки внутренний стек компилятора и задает для текущего выравнивания упаковки значение n. Если значение n не указано, текущее значение выравнивания упаковки не помещается в стек. pop(необязательно) —Удаляет запись из вершины внутреннего стека компилятора. Если значение n не указано вместе с pop, то новым значением выравнивания упаковки становится значение упаковки, связанное с результирующей записьюв вершине стека. Если значение n указано (например, #pragma pack(pop,16)), n становится новым значением выравнивания упаковки. Если извлечение из стека производится вместе с идентификатором identifier (например, #pragma pack(pop, r1)),из стека извлекаются все записи,пока не будетнайдена запись identifier. Эта запись извлекается из стека, и новым значением выравнивания упаковки становится значение упаковки, связанное с результирующей записью в вершине стека. Если при извлечении из стека с идентификатором identifier этот идентификатор не найден ни в одной из записей стека, директива pop игнорируется. identifier(необязательно) —При использовании с

директивой push присваивает имя записи во внутреннем стеке компилятора. При использовании с директивой pop записи из внутреннего стека извлекаются до тех пор, пока не будет удален идентификатор identifier; если идентификатор identifier во внутреннем стеке не найден, ничего не извлекается.

n (необязательно) —Указывает значение(в байтах),используемое для упаковки. Если для модуля не задан параметр компилятора /Zp, по умолчанию для n используется значение 8. Допустимые значения: 1, 2, 4, 8 и 16. Выравнивание члена будет производиться по границе, кратной значению n или кратной размеру члена, в зависимости от того, какое из значений меньше.

Значение #pragma pack(pop, identifier, n) не определено.

```c
/* pragma_directives_pack.c */
#include <stddef.h>
/** offsetof */
#include <stdio.h>

struct S {
    int i;  /* size 4*/
    short j;/* size2*/
    double k;/* size8*/

};

#pragma pack(2)

struct T {
    int i;
    short j;
    double k;
};


int main ( ) {
    printf ( "%d ", offsetof ( struct S, i ) );
    printf ( "%d ", offsetof ( struct S, j ) );
    printf ( "%d -", offsetof ( struct S, k ) );
    printf ( "%d ", offsetof ( struct T, i ) );
    printf ( "%d ", offsetof ( struct T, j ) );
    printf ( "%d\n", offsetof ( struct T, k ) );
    return 0;
}
```

Результат: 0 4 8 – 0 4 6

9.6 Директива #pragma section

Создает раздел в OBJ-файле.

```
#pragma section( "section-name" [, attributes] )
```

Термины сегмент и раздел в этом разделе взаимозаменяемы. После определения раздел остается допустимым для остальной части компиляции. Однако следует использовать __declspec(allocate),так как иначе никакие данные не будут помещены в раздел.

section-name —обязательный параметр, который будет именем раздела. Имя не должно конфликтовать со стандартными именами раздела. Список имен, которые не следует использовать при создании раздела, см. в разделе/SECTION.

attributes —необязательный параметр, состоящий из одного или несколькихразделенных запятыми атрибутов, которые требуется присвоить разделу. Ниже перечислены возможные attributes.

- read (чтение) — Позволяет выполнять операции чтения данных.
- write (запись) — Позволяет выполнять операции записи данных.
- execute —Позволяет выполнять код.
- shared —Предоставляет совместный доступ к разделу всем процессам, загружающим образ.
- nopage —Отмечает раздел как невыгружаемый; используются для драйверов устройств Win32.
- nocache —Отмечает раздел как некэшируемый; используются для драйверов устройств Win32.
- discard —Отмечает раздел как удаляемый; используются для драйверов устройств Win32.
- remove —Отмечает раздел как нерезидентный; только драйверы виртуальных устройств (VxD).

Если не задать атрибуты, раздел будет иметь атрибуты чтения и записи.

В следующем примере первая инструкция определяет раздел и его атрибуты.

Целое число j не помещается в mysec, поскольку оно не было объявлено с __declspec(allocate),j переходит в раздел данных. Целое число i переходит в mysec как результат атрибута класса хранения __declspec(allocate).

```
/* pragma_section.cpp */
#pragma section(”mysec”, read, write)

  int j =0;
  __declspec(allocate(”mysec”))
  int i =0;

int   main(){}
```

9.7. Директива#pragmacomment

Вставляет запись комментария в объектный или исполняемый файл.
#pragma comment(comment-type [, ”commentstring”])

comment-type обозначает один из предопределенных идентификаторов(см.ниже), которые задают тип записи комментария. Необязательный параметр commentstring обозначает строковый литерал, который содержит дополнительную информацию (для некоторых типов комментариев). Поскольку параметр commentstring обозначает строковый литерал, он соблюдает все правила, действующие для строковых литералов в отношении escape-символов, внедренных кавычек (”) и конкатенации.

- compiler —Задает в объектном файле имя и номер версии компилятора. Эта запись комментария игнорируется компоновщиком. Если для этого типа записи будет указан параметр commentstring, компилятор выведет предупреждение.

- exestr —Задает в объектном файле commentstring, которая во время компоновки помещается в исполняемый файл. Эта строка не загружается в память вместе с исполняемым файлом, однако ее можно обнаружить при помощи программы, которая находит в файлах печатаемые строки. Записи комментариев этого типа позволяют, в частности, вставлять в исполняемый файл информацию о номере версии и т. д. Использовать параметр exestr не рекомендуется; в следующих выпусках он будет удален. Компоновщик не обрабатывает эту запись комментария.

- lib —Задает в объектном файле запись поиска библиотеки. Этот тип комментария должен сопровождаться commentstring с именем библиотеки, которую должен найти компоновщик. В нем также можно указать путь к ней. Имя библиотеки указывается в объектном файле после записей поиска по библиотекам по умолчанию. Компоновщик ищет эту библиотеку точно так же, как если бы она была указана в командной строке (если библиотека не была задана при помощи параметра /NODEFAULTLIB). В один и тот же исходный файл можно вставить несколько записей поиска библиотеки. В объектном файле они будут располагаться в том же порядке, что и в исходном.

Если для вас важно, в каком порядке расположены и добавленные библиотеки, и библиотека по умолчанию, задайте при компиляции ключ /Zl. В этом случае в объектный модуль не будет вставлено имя библиотеки по умолчанию. Далее можно вставить еще одну директиву #pragma comment и с ее помощью добавить имя библиотеки по умолчанию уже после добавленной библиотеки.

Библиотеки, добавленные при помощи этих директив, будут находиться в объектном модуле в том же порядке, в каком они указаны в исходном коде.

- linker — Задает параметр компоновщика в объектном файле. Благодаря этому параметр компоновщика можно не передавать из командной строки и не указывать в среде разработки, а задать непосредственно в комментарии. Например, в нем можно задать параметр /INCLUDE, чтобы принудительно включить символ:

#pragma comment(linker, ”/INCLUDE:__mySymbol”)

Идентификатору компоновщика могут передаваться только следующие параметры (comment-type)

- /DEFAULTLIB
- /EXPORT
- /INCLUDE
- /MANIFESTDEPENDENCY
- /MERGE
- /SECTION

- user —Вставляет в объектный файл комментарий общего рода. commentstring содержит текст комментария. Эта запись комментария игнорируется компоновщиком.

Следующая директива pragma указывает компоновщику найти библиотеку EMAPI.LIB во время компоновки. Сначала компоновщик ищет ее в текущем рабочем каталоге, а затем по пути, заданном в переменной среды LIB.

#pragma comment(lib, ”emapi”)

Следующая директива pragma указывает компилятору вставить в объектный файл имя и номер версии компилятора:

```
#pragma comment( compiler )
#pragma comment( user, ”Compiled on ” __DATE_)
```

10. Аргументы командной строки

Иногда бывает необходимо запустить приложение через консоль, и заодно передать ему некоторые аргументы. До этого мы оставляли функцию mainбез аргументов, однако в коде многих программ можно встретить аргументы int argc и char* argv[]. Первый- это количество аргументов, а второй- массив строк. Примечательно, что первый аргумент- это всегда полный путь к программе. При передаче через консоль аргументы разделяются пробелами. Вот самая простая демонстрация:

```
#include <stdio.h>

int main ( int argc, char *argv [ ] )
{
    int i;
    printf ( "\nargc = %i", argc );
    for ( i = 0; i < argc; i ++ ) {
        printf ( "\nArgument  %s", argv[i] );
    }
    return 0;
}
```

Программа просто выводит полученные аргументы.

Примечание

С помощь данного метода удобно передавать имя файла функцию.

11. Введение в объектно-ориентированное программирование

11.1 Объекты

При изучении объектно-ориентированного программирования (ООП) наибольшей проблемой является использование новой терминологии и понимание нового подхода к решению старых задач. Для начала дадим определение: ООП – система принципов и способов организации представления программы в виде совокупности объектов, каждый из которых является экземпляром определенного класса.

C++ представляет именно другую методологию программирования, если в C вы использовали функцию, как основополагающую единицу (процедурное программирование), то теперь ее заменит класс. Поясним, что такое классы и объекты:

Классы в C++ — это абстракция, описывающая методы, свойства, ещё не существующих объектов. Объекты — конкретное представление абстракции, имеющее свои свойства и методы. Методы - функции классов.

Приведем простой пример класс – Автомобиль, объекты класса Автомобиль: ВАЗ-2105, BMWX5. Класс задает основные характеристики и функции, который должен иметь любой его экземпляр и действительно мы можем сказать, что и BMW, и ВАЗ являются автомобилями.

11.2 Наследование

Еще одним столпом ООП является наследование. Наследование - концепция объектно-ориентированного программирования, согласно которой абстрактный тип данных может наследовать данные и функциональность некоторого существующего типа, способствуя повторному использованию компонентов программного обеспечения. Проще говоря, если у вас есть класс, но вам нужно его расширить или добавить дополнительные функции/переменные, то вы можете сделать еще один класс на основе этого. Например, класс Внедорожник, который будет являться наследником класса Автомобиль, т.е. Внедорожник будет иметь все те же функции/данные, что и класс родитель, однако будет дополнен некоторыми особенностями.

Немного терминологии,

Класс, опредёленный через наследование от другого класса, называется: производным классом, классом потомком (англ. derived class) или подклассом (англ. subclass). Класс, от которого новый класс наследуется, называется: предком (англ. parent), базовым классом (англ. base class) или суперклассом (англ. parent class).

11.3 Инкапсуляцияи полиморфизм

Если говорить кратко инкапсуляция – это сокрытие данных. На самом деле за кажущейся очевидностью кроется нечто большее. Этот принцип позволяет нам работать с объектами непосредственно через интерфейс класса, даря возможность не думать, как он устроен и, что более важно, не давая как-то вмешаться в уже отлаженный механизм. Как вы вскоре увидите принято большинство данных классов делать скрытыми.

Опять же аналогия с автомобилями производитель сделал машину, которая выполняет необходимые требования владельца, однако для того, чтобы ей пользоваться, не нужно знать, как она работает.

Способность объекта использовать методы производного класса, который не существует на момент создания базового, называется полиморфизмом (греч, polymorphos многообразный). Чтобы было понятнее, представим, что вы хотите использовать объекты 3х различных классов, которые являются наследниками одного класса. Вы можете работать с каждым из них по отдельности, но это не очень удобно. Гораздо проще работать с ними как с объектами одного класса, вот здесь и поможет полиморфизм, он поможет работать с разными объектами, как с элементами одного класса.

Возьмем мастерскую, занимающуюся ремонтом, где ремонтируются мотоциклы, легковые автомобили, катера и т.д. Очевидно, будет проще вводить данные по унифицированной системе для всех транспортных средств, а также осуществлять поиск и совершать иные операции. Так что создав класс, Транспортное_средство и объявив его предком (настроить общие функции), можно существенно облегчить процесс.

11.4 Абстракция

Абстракция стоит отдельно от «трех китов ООП»: полиморфизма, наследования и инкапсуляции. Однако о ней тоже следует упомянуть.

Абстракция данных – выделение существенных характеристик объекта, существенных в рамках решаемой нами задачи.

Возьмем тот же автомобиль. Допустим, нашей задачей будет определить внешнее различие одного автомобиля от какого-то другого (да, пример своеобразен, но он легко объясняет принцип абстракции).

Внешние признаки автомобиля: цвет, форма автомобиля и т.д. Для этих признаков мы и создадим поля класса для будущего объекта автомобиля.

Остальные признаки сравнения, такие как двигатель, размер колес автомобиля, его вес и т.д. нам не нужны. Так как мы сравниваем только по внешним признакам, соответственно и поля класса для будущих объектов создаются только по внешним признакам (в нашем случае).

Комментарии

Конечно, приведенные выше определения и примеры не являются точными и подробными, они лишь призваны познакомить с основными концепциями, большинство из прочитанного будет раскрыто позднее, так как создание большого разрыва между теорией и практикой не поспособствует успешному освоения материала

12. Классы

Основным отличием C++ от C является наличие классов, которые предоставляют возможность моделирования объектов реального мира. Это происходит посредством совмещения методов, т.е. функций классов и данных(+ еще несколько особенностей, но об этом позже).

Лучше увидеть это на практике:

```
#include <iostream>

using namespace std;

class example {
    private:
        int length;
        int width;
    public:
        void input ( ) {
            cout << "\nEnter the length: ";
            cin >> length;
            cout << "\nEnter the width: ";
            cin >> width;
        }
        int square ( ) {
            return length * width;
        }
        int perimeter ( ) {
            return 2 * ( length + width );
        }
};


int main ()
{
    example ex;
    ex.input( );
    cout << "\nThe square: " << ex.square();
    cout << "\nThe square: " << ex.perimeter();
    return 0;
}
```

Объявление класса начинается с ключевого слова class. После имени открываются фигурные скобки – это тело цикла, здесь располагаются данные (переменные) и функции класса (методы). Открывает тело слово private, оно означает, что к следующим методам/данным могут получить доступ только методы данного класса, например в данной программе нельзя записать ex.length=7, однако методы ex.square() и ex.perimeter() будут работать. Ключевое слово publicозначает, что следующие методы или данные пользователь может вызывать из других функций. Ключевые слова

могут быть объявлены и не в таком порядке, кроме того, privateможно не объявлять, т.к. элементы класса по умолчанию скрыты. После объявления и определения методов в классе, можно приступать к работе с элементами класса.

В mainсоздается объект exкласса example, далее через операцию точки (.) получается доступ к 3 методам. 2 из них возвращают значения целочисленного типа. Так что можно вывести результат через поток вывода.

При создании нового объекта класса создаются и новые данные (если не объявлены со словом static), однако не методы, что обусловлено экономией памяти, да и просто логикой.

Примечание.

Никогда не путайте структуры C и C++, большинство программистов используют структуры только для хранения данных (и это правильно); в C++, в отличие от C, их возможности намного выше. В C++ структуры отличаются от классов только тем, что по умолчанию имеют тип public, в то время как классы – private.

13. Конструкторы и деструкторы

При создании экземпляра класса всегда вызывается функция, называемая конструктором. Если, как в предыдущей главе, она не определена то, такой конструктор называется конструктором по умолчанию. Конструктор не может возвращать значения, однако передавать ему аргументы, при условии, что они прописаны в объявлении, можно. Характерной особенностью конструктора помимо обязательного вызова при создании, является то, что имя этой функции совпадает с именем класса, таким образом, его всегда можно отличить.

Как можно догадаться, деструктор является, в некоторой степени, противоположностью конструктора, то есть вызывается при уничтожении объекта, из этого логически следует, что деструктор не может иметь ни параметров, ни возвращаемого значения. Найти данную функцию в коде очень просто, так как она имеет имя класса со знаком «~»перед. Основной задачей деструктора обычно является работа с памятью.

Модифицируем немного предыдущий класс:

```
#include<iostream>

using namespace std;

class example
{
    private:
        int length;
        int width;
    public:
        example( ): length(0), width(0)
            { /*There is nothing*/ }
        example ( int l, int w ): length ( l ), width ( w )
            {/*There is nothing too*/}
        ~example ( ) {
            cout << "Object was destucted";
        };
        void input ( ) {
            cout << "\nEnter the length: ";
            cin >> length;
            cout << "\nEnter the width: ";
            cin >> width;
        }
        int square ( ) {
            return length * width;
        }
```

```
        int perimeter ( ) {
            return 2 * ( length + width );
        }
};

int main()
{
    example ex;
    example ex1 ( 12, 13 );
    ex.input ( );
    cout << "\nThe 1 square: " << ex.square( );
    cout << "\nThe 1 perimeter: " << ex.perimeter( );
    cout << "\nThe 2 square: " << ex1.square( );
    cout << "\nThe 2 perimeter: " << ex1.perimeter( );
    return 0;
}
```

Строка: example(): length(0), width(0) – это конструкторбезаргументов.

Строка: example(intl, intw) : length(l), width(w) – это конструкторсдвумяаргументами

В этом примере представлено два конструктора: без аргументов и с 2 аргументами. Как вы могли заметить, присваивание параметров в конструкторах происходит не стандартно, такое решение программисты зачастую принимают из-за того, что это повышает удобочитаемость и логичность кода, однако обычный подход, в котором присваивание происходит в теле функции, здесь также сработает.

В программе деструктор, конечно, является просто примером и не выполняет тех функций, которые ложатся на него в проектах.

В заключении стоит лишь указать, что конструкторы и деструкторы являются важной особенностью классов, первые удобно использовать если вам необходимо сделать некоторые операции сразу при определении(или хотя бы из-за того, что это удобно), что до вторых то их использование еще более важно, и в следующих главах это можно будет увидеть отчетливо.

14. Наследование

Пришло время рассказать еще об одной особенности языка C++, которая существенно облегчает работу программиста. Итак, наследование – это процесс создания новых классов называемых наследниками или производными классами, из уже существующих или базовых классов. То есть, допустим, если у вас есть класс с нужными свойствами, но вам нужно написать еще несколько, при этом вам нельзя изменять базовый класс (возможно, его долго отлаживали или он необходим для каких-то частных объектов). Тогда вы можете наследовать этот класс, приписав к нему необходимые свойства. На рисунке 14.1 это представлено наглядно:

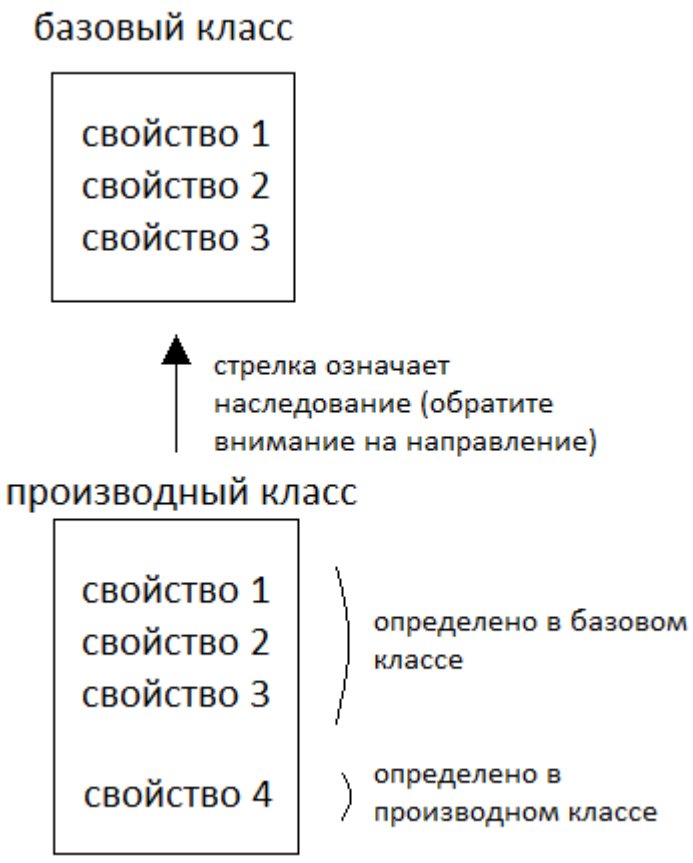


Рис 14.1

Что же, попробуем увидеть эту схему на примере:

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

class Summ
{
 protected:
    int first;
    int second;
 public:
    Summ(int a,int b): first(a),second(b)
    {}
    int plus()
    {
     return first+second;
    }
};

class SummAndDif : public Summ
{
 public:
    SummAndDif(int a,int b): Summ(a,b)
    {}


    int minus()
    {
     return first-second;
    }
};

int main()
{
    SummAndDif numbers(8,5);
    cout<<"Summ is: "<<numbers.plus();
    cout<<"\nDifference is: "<<numbers.minus();
    return 0;
}
```

Сразу можно заметить, что вместо ключевого слова private используется protected. Разница в том, что с помощью protected из наследников можно получить доступ к данным/методам, а из private – нет. Для того, чтобы сделать класс наследником нужно после имени класса поставить двоеточие, а далее перечислить родителей (да их может быть несколько). Вы могли заметить немного необычное определение конструктора у наследника: после двоеточия вызывался конструктор родителя, это довольно встречающаяся практика, конструктор из производного класса получает данные, а для обработки полной или частичной, вызывается конструктор базового класса. При работе с созданным объектом все методы вызываются так, как будто объявлены в теле класса.

Представленная выше информация несет в себе цель познакомить студента с наследованием и дать минимальные практические навыки, однако в вопросах наследования есть много тонкостей, которым нужно было бы выделить отдельную книгу, поэтому лучший способ изучения языка – практика.

15. Явное и неявное преобразование типов в C++

Среди них уже рассмотренный нами оператор static_cast (в разделе 6.2), оператор снятия константности (и volatile, но это ключевое слово используется редко) const_cast, а также операторы reinterpret_cast (для преобразования принципиально различных типов) и dynamic_cast (для преобразования полиморфных типов, связанных иерархией наследования).

Рассмотрим их синтаксис:

```
TYPE = static_cast<TYPE> (object);
```

Оператор static_cast преобразует выражение одного статического типа в объекты и значения другого статистического типа

```
TYPE* = dynamic_cast< TYPE* > (object);
```

Оператор dynamic_castиспользуется для динамического приведениятипов во время выполнения программы

```
TYPE = const_cast<TYPE> (object);
```

```
TYPE = reinterpret_cast < TYPE > (object);
```

Что делают эти операторы было написано в первом абзаце.

16. Перегрузка операций

Когда уже определенная операция (к примеру, + или -) наделяется возможностью совершать действия над операндами нового типа, то такая операция – перегруженная. Перегрузка операций помогает упростить код программы.

Рассмотрим это утверждение на примере:

```
#include <iostream>

using namespace std;

class Room
{
    private:
        int width_meters;
        int length_meters;
        float width_sm;
        float length_sm;
    public:
        Room ( ) : width_meters ( 0 ),
                   length_meters ( 0 ),
                   width_sm ( 0.0 ),
                   length_sm ( 0.0 )
        {  }
        Room (int w, int l, float wm, float lm) : width_meters ( w ),
                                                  length_meters ( l ),
                                                  width_sm ( wm ),
                                                  length_sm ( lm )
        {  }



         void enter ( )
         {
             cout << "Enter the values";
             cin >> width_meters;
             cin >>length_meters;
             cin >> width_sm;
             cin >> length_sm;
         }
         void output ( )
         {
             cout << width_meters << endl <<
                     length_meters << endl <<
                     width_sm << endl
                     << length_sm << endl;
         }
         Room operator+ ( Room ) const;
};
```

```
Room Room :: operator+ ( Room x ) const
{
    int n1 = width_meters + x.width_meters;
    int n2 = length_meters + x.length_meters;
    float f1 = width_sm + x.width_sm;
    float f2 = length_sm + x.length_sm;
    if ( f1 >= 100.0 ) {
        f1 -=100.0;
        n1++;
    }
    if ( f2 >= 100.0 ) {
    f2 -=100.0;
    n2++;
    }
    return Room ( n1, n2, f1, f2 );
}


int main ( )
{
    Room iz1, iz2, iz3;
    iz1.enter ( );
    iz2.enter ( );
    iz3 = iz1 + iz2;
    iz3.output ( );
    return 0;
}
```

В листинге этой программы используется ключевое слово operator, с которым вы, вероятнее всего, столкнулись впервые. Именно это ключевое слово позволяет нам перегружать операции.

Сначала пишется возвращаемый тип (в данном случае Room), далее ключевое слово operator, а затем сама операция, которую следует перегрузить (мы перегружаем «+»). Такой синтаксис говорит компилятору о том, что если операнд принадлежит классу Room, то нужно вызвать функцию с таким именем, встретив эту арифметическую операцию в тексте программы.

17. Перегрузка функций

В C++ стала возможна перегрузка функций. Это понятие означает, что вы можете иметь несколько функций с одинаковыми именами. В самом деле, если вам нужно несколько функций, выполняющих приблизительно одну и ту же задачу, то будет намного проще иметь одно имя для всех. Но как же компилятор различит, какую функцию надо использовать? Ответ прост – по параметрам.

Ниже представлена программа, возводящая целые или вещественные числа в квадрат:

```
#include <iostream>

using namespace std;

int root ( int );
float root ( float );

int main ( )
{
    int n;
    float f;
    cout << "Enter the integer and float: " << endl;
    cin >> n >> f;
    cout << "\n The results are: " << root ( n ) << " " << root( f );
  ì return 0;
}

int root ( int numb )
{
    return numb * numb;
}

float root ( float numb )
{
    return numb * numb;
}
```

В листинге этой программы видно, что прототип функции rootобъявлен как и для вещественных, так и для целых чисел.

18. Потоки ввода и вывода

В C для организации ввода и вывода (по крайней мере в этом пособии) были использованы функции scanf() илиprintf(), а также их аналоги для работы с файлами fprintf() иfscanf(). Однако в C++ практикуется несколько иной подход, а конкретнее: потоковые объекты. Они имеют несколько преимуществ перед функциями. Первым является простота использования, вспомните, как нужно было форматировать ввод и вывод с помощью символов управления форматом (%d, %f, и т.д.), ничего подобного вы не встретите при использовании потоков, ибо каждый объект сам отвечает за то, как он выглядит на экране. Вторым весомым аргументом является возможность перегрузки операторов вставки(<<) и извлечения(>>), это позволяет работать с классами как со стандартными типами. Нельзя также не упомянуть о том, что использование потоков является лучшим способом записывать данные в файл, лучшим способом организовывать данные в памяти (что важно при работе с графическим интерфейсом).

Пример использования потоковых объектов можно увидеть ниже:

```
#include <iostream>

using namespace std;

int main ( )
{
    int n;
    float f;
    char c;
    cout << "Enter the integer: ";
    cin >> n;
    cout << "\nEnter the float: ";
    cin >> f;
    cout << "\nEnter the symbol: ";
    cin >> c;
    cout << "\nWas entered: " << n << " " << f << " " << c;
    return 0;
}
```

Итак, с первой строки мы сталкиваемся с неизвестной, пока что, библиотекой на самом деле, как не трудно догадаться по названию, она содержит все необходимое для работы с данными. Далее идет строка: using namespace std. Дело в том, что в C++ есть такое понятие, как пространство имен, если вкратце, то оно помогает данным не пересекаться, т. е. если у вас есть две функции, объявленные в разных пространствах, то можно не переживать, что будет использована одна, вместо другой. Директива

usingобъявляет, что все объекты в коде принадлежат одному пространству имен. Есть еще один вариант:

```
std::cout << "\nEnter the symbol: ";
std::cin >> c;
```

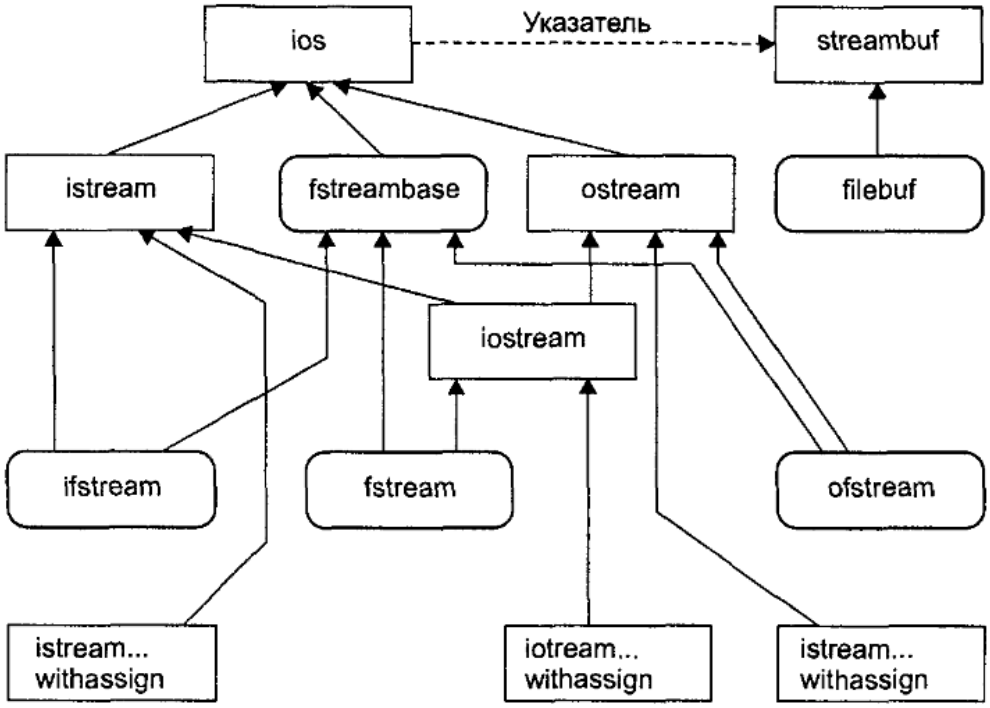
Операция :: называется операцией глобального разрешения и отражает, что cin, cout являются объектами пространства std.

Так часто пишут, если нужно работать с множеством библиотек и, следовательно, с множеством пространств.

Далее в программе идет несколько cin (объектов ввода),cout(объектов вывода), а также операций вставки в поток(<<)и извлечения(>>). Смысл их работы выражается следующем: объект представляет собой поток, в который с помощью операции вставки помещаются данные, или с помощью операции извлечения считываются.

18.1 Иерархия библиотек ввода вывода

Говоря о вводе и выводе нельзя не упомянуть о связи библиотек, отвечающих за ввод и вывод. Как уже известно, одним их основных отличий программирования на C++ от C является применение принципов ООП, в частности – наследования. Несколько наследований могут образовывать своеобразную иерархию:



Как видно класс iosнаследуется всеми классами.

18.2 Форматирование ввода и вывода

Как вы помните, в C для форматирования вывода использовались следующие символы: ‘\n’, ‘\t’, и т. д. Они также, использовались в вышеуказанном примере, тем не менее, потоки представляют больше способов организации ввода и вывода.

Основными являются манипуляторы, флаги форматирования и функции для работы с потоковыми объектами и установки флагов форматирования.

18.3 Манипуляторы

Манипуляторы – это инструкции форматирования, которые вставляются прямо в поток примеры:

```
cout << "Helloworld" << endl;
cout << setiosflags ( ios::fixed ) << var;
```

endlявляется манипулятором разделителя строк строки, его действие напоминает “\n”.

setiosflags() – устанавливает (подождите, подождите) флаги форматирования, в данном случае флаг фиксированного вывода.

Ниже приведен список самых используемых манипуляторов:

Dec	Ввод/вывод данных в десятичной форме	ввод и вывод
Endl	Вывод символа новой строки с передачей в поток всех данных из буфера	вывод
Ends	Вывод нулевого символа	вывод
Flush	Передача в поток содержимого буфера	вывод
Hex	Ввод/вывод данных в шестнадцатеричной системе	ввод и вывод
Oct	Ввод/вывод данных в восьмеричной форме	ввод и вывод
resetiosflags(long f)	Сбрасывает флаги, указанные в f	ввод и вывод
setbase(int base)	Устанавливает базу счисления равной параметру base	вывод
setfill(int ch)	Устанавливает символ заполнения равным ch	вывод
setiosflags(long f)	Устанавливает флаги, указанные в f	ввод и вывод
setprecision(int p)	Устанавливает число цифр после запятой	вывод
setw(int w)	Устанавливает ширину поля равной w	вывод
Ws	Пропускает начальный символ-разделитель	ввод

При использовании данных манипуляторов не забывайте, что манипуляторы действуют только на те данные, которые следуют за ними в потоке.

18.4 Функции форматирования ввода и вывода

В отличие от манипуляторов, функция форматирования не должна вставляться в поток, ее применение полностью соответствует применению любой другой функции. Каждая библиотека из семейства ios имеет свои собственные функции (хотя доступ к ним всем можно всегда получить из iostream, если речь идет о вводе и выводе данных).

Функции ios

Функция	Назначение
setf(flags)	Устанавливает флаг форматирования
unsetf(fags)	Сбрасывает флаг форматирования
width(w)	Устанавливает ширину текущего поля
w =width()	Возвращает текущее значение ширины поля
precision(p)	Возвращает число вводимых знаков для формата с плавающей запятой
fill(ch)	Устанавливает символ заполнения
ch=fill()	Возвращает символ заполнения
setf(flags, field)	Очищает поле, затем устанавливает флаги форматирования

Функции iostream

get(ch)	Извлекает один символ в ch
get(str)	Извлекает символы в массив str до ‘\n’
get(str, Max)	Извлекает до Махчисла символов в массив
get(str, lim)	Извлекает символы в массив до ограничителя lim
get(str, Max, lim)	Извлекает символы в массив до Махсимволов или до ограничителя lim
getline(str, Max, lim)	Извлекает символы в массив до Max символа или до ограничителя lim. Извлекает ограничитель из потока
putback(ch)	Вставляет последний прочитанный символ обратно во входной поток
ignore(Max, lim)	Извлекает и удаляет доМахчисла символов до ограничителя ‘\n’ в массив
peek(ch)	Читает один символ, оставляя его в потоке
count=gcount()	Возвращает число символов, прочитанныхтолько что встретившимися вызовами get(), getline(), read()
read(str, Max)	(Для файлов) Извлекает до Махчисла символов в массив до EOF(признак конца файла)
seekg()	Устанавливает расстояние (в байтах) от начала до файлового указателя
seek(pos, seek_dir)	Устанавливает расстояние (в байтах) от указанной позиции до файлового указателя. seek_dirможет принимать значения ios::beg, ios::cur, ios::end;
pos=tellg(pos)	Возвращает позицию (в байтах) указателя файла от начала файла

19.Шаблоны

19.1. Шаблоны функций

Возвращаясь к предыдущей теме, можно сказать, что помимо перегрузки функций, есть более элегантный способ решения. Действительно ведь, функции возведения в корень отличаются только типом аргумента. Это настолько часто встречающееся явление в программировании, что C++ предоставляет программисту возможность использовать шаблоны. Шаблоны функций — это инструкции, согласно которым создаются локальные версии функции с шаблоном для определенного набора параметров и типов данных. Если проще, то вместо указания типа аргумента, можно написать шаблон и вызывать функцию для работы со всеми типами (конечно, если к этим типам применимы операции, которые вы описали в функции). Это немного трудно для понимания, поэтому небольшой пример

```
#include<iostream>

using namespace std;

template <typenameT>
T root(T);

int main ( )
{
    int n;
    float f;
    cout << "Enter the integer and float: " << endl;
    cin >> n >> f;
    cout << "\n The results are: " << root ( n ) << " " << root ( f );
    return 0;
}

template <typenameT>
T root ( T numb )
{
    return numb * numb;
}:
```

Третья строка - template<typename T> - это шаблон функции перед объявлением.

Также template<typename T>необходимо поставить и перед определением

Итак, первое отличие этой программы от предыдущей состоит в том, что появляется ключевое слово template, с помощью него мы определяем шаблон функции. Далее идут угловые скобки, в которых стоит еще одно ключевое слово typename, на самом деле его можно заменить либо на type, либо на class. И, наконец, переменная T (может называться как угодно, T– просто случайное название, не несущее смысловой нагрузки). При вызове функции, Tзаменяется на тот тип данных, который был передан в качестве аргумента, при каждом новом вызове иного типа создается еще новая функция, то есть, если вы вызвали шаблонную функцию для integer и float, то на самом деле, у вас будет две функции, шаблоны помогают в удобстве написания кода, но не в экономии памяти. Подстановка соответствующего типа называется реализацией шаблона функции. Важное замечание – функция с шаблоном должна иметь определение этого самого шаблона перед объявлением и определением.

Примечание

Аргументы шаблона должны быть согласованы, следующий пример демонстрирует, как не надо делать:

```
Template <class F>

void somefunc ( F, F )
{ … }

int main ( )
{
    int n;
    char c;
    somefunc(c,n)  //ERROR!
    return 0;
}
```

Если вы хотите использовать различные аргументы, то можно сделать так:

```
Template <class F, class E>

void somefunc(F,E)
{ … }

int main ( )
{
    int n;
    char c;
    somefunc( c, n ) //It's allright
    return 0;
}
```

Небольшой совет: не начинайте написание функции сразу с шаблонов, сначала лучше убедиться в ее работе с фиксированным типом данных.

19.2. Шаблоны классов

Как можно было догадаться, шаблоны могут применяться не только в функциях, но и в классах. Рассмотрим пример реализации стека для разных типов данных:

```
#include <iostream>

using namespace std;

const int MAX = 20;

template <typename T>

class Stack {
    private:
        T mas[MAX];
        int top;
    public:
        Stack();
        void push(T);
        T pop();
    };

template <typename T>
Stack<T>::Stack()
    {
        top = -1;
    }


template <typename T>
void Stack<T> :: push(T var)
{
    mas[++top] = var;
}

template <typename T>
T Stack<T>::pop()
{
    return mas[top--];
}
```

```
int main ( )
{
    Stack<float> st;
    st.push ( 1.1 );
    st.push ( 2.2 );
    st.push ( 3.3 );
    cout << "\n1 : " << st.pop( );
    cout << "\n2 : " << st.pop( );
    cout << "\n3 : " << st.pop( );

    Stack<int> sti;
    sti.push ( 11 );
    sti.push ( 22 );
    sti.push ( 33 );
    cout << "\n1 : " << sti.pop( );
    cout << "\n2 : " << sti.pop( );
    cout << "\n3 : " << sti.pop( );
    return 0;
}
```

Разберем же, что здесь происходит. Сначала перед объявлением класса ставится уже знакомая конструкция: template<typename T>. Так же она ставится перед каждым определением функции вне класса (что схоже с логикой объявления / определения шаблонных функций). Обратите внимание на то, что после template<typename T> в определении идет – void Stack<T> :: push(T var), где Stack<T> помогает идентифицировать класс.

Создание объекта происходит же посредством написания имени класса и указания в угловых скобках имя типа : Stack<float>st.

Приложение

Правила оформления кода

Отступы

∗ Для обозначения отступа используйте 4 пробела подряд;

∗ Используйте пробелы, а не табуляцию.

Объявление переменных

∗ Объявляйте по одной переменной в строке;

∗ Избегайте, если это возможно, коротких и запутанных названий переменных (Например: «a», «rbarr», «nughdeget»);

∗ Односимвольные имена переменных подходят только для итераторов циклов, небольшого локального контекста и временных переменных. В остальных случаях имя переменной должно отражать ее назначение;

∗ Заводите переменные только по мере необходимости:

```
// Wrong
int a, b;
char *c, *d;
// Correct
int height;
int width;
char *nameOfThis;
char *nameOfThat;
```

∗ Функции и переменные должны именоваться с прописной буквы, а если имя переменной или функции состоит из нескольких слов, то первое слово должно начинаться с прописной буквы, остальные – со строчных;

∗Избегайтеаббревиатур:

```
// Wrong
short Cntr;
char ITEM_DELIM = '\t';
// Correct
short counter;
char itemDelimiter = '\t';
```

∗ Имена классов всегда начинаются с заглавной буквы.

Пробелы

∗ Используйте пустые строки для логической группировки операторов, где это возможно;

∗ Всегда используйте одну пустую строку в качестве разделителя;

∗ Всегда используйте один пробел перед фигурной скобкой:

```
 // Wrong
if(foo){ }
// Correct
if (foo) { }
```

∗ Всегда ставьте один пробел после * или &, если они стоят перед описанием типов. Но никогда не ставьте пробелы после * или & и именем переменной:

```
char *x;
const Class &myClass;
const char * const y = "hello";
```

∗Бинарные операции отделяются пробелами с 2-х сторон;

∗После преобразования типов не ставьте пробелов;

∗Избегайте преобразования типов в стиле C:

```
 // Wrong
 char *blockOfMemory = (char *) malloc(data.size());
// Correct
 char *blockOfMemory = reinterpret_cast( malloc(data.size()) );
```

Фигурные скобки

∗Возьмитезаосновурасстановкуоткрывающихфигурныхскобокнаоднойстроке свыражением, которому они предшествуют:

```
// Wrong
if (codec)
{
}
// Correct
if (codec) {
}
```

∗ Исключение: Тело функции и описание класса всегда открывается фигурной скобкой, стоящей на новой строке:

```
static void foo ( int g )
{
 fprintf ( stdout, "foo: %i", g );
}
class Moo
{
};
```

∗Используйте фигурные скобки в условиях, если тело условия в размере превышает одну линию, или тело условия достаточно сложное, и выделение скобками действительно необходимо:

```
// Wrong
if (address.isEmpty()) {
 return false; }
for (int i = 0; i < 10; ++i) {
fprintf(stdout, "%i", i);
}
// Correct
if (address.isEmpty())
  return true;
for (int i = 0; i < 10; ++i)
  fprintf(stdout, "%i", i);
```

∗ Исключение 1: Используйте скобки, если родительское выражение состоит из нескольких строк или оберток:

```
// Correct
if (address.isEmpty() || !isValid())
|| !codec) {
return false;
}
```

∗ Исключение 2: Используйте фигурные скобки, когда тела ветвлений if-then-else занимают несколько строчек:

```
// Wrong
if (address.isEmpty())
  return false;
else {
fprintf(stdout, "%s", address.c_str());
++it;
}
// Correct
if (address.isEmpty()) {
  return false; }
else {
  fprintf(stdout, "%s", address.c_str());
++it;
 }
 // Wrong
if (a)
  if (b)
    ...
else
// Correct
if (a) {
   if (b)
...
else
    ... }
```

∗Используйте фигурные скобки для обозначения пустого тела условия:

```
// Wrong
while (a);
 // Correct
while (a) {}
```

Круглые скобки

∗Используйте круглые скобки для группировки выражений:

```
// Wrong
if (a && b || c)
// Correct
if ((a && b) || c)
// Wrong
a + b & c
 // Correct
(a + b) & c
```

Использование конструкции switch

∗Операторы case должны быть в одном столбце со switch

∗Каждый оператор case должен иметь закрывающийbreak (илиreturn) или комментарий, который предполагает намеренное отсутствие break или return:

```
switch (myEnum) {
    caseValue1:
        doSomthing();
        break;
    caseValue2:
        doSomthingElse();
        // continue
    default:
        defaultHandling();
        break;
}
```

Разрыв строк

∗ Длина строки кода не должна превышать 100 символов. Если надо – используйте разрыв строки. ∗ Запятые помещаются в конец разорванной линии; операторы помещаются в начало новой строки. В зависимости от используемой вами IDE, оператор на конце разорванной строки можно проглядеть:

```
// Correct
     if (longExpression
     + otherLongExpression
    + otherOtherLongExpression) {
     }
// Wrong
     if (longExpression + otherLongExpression + otherOtherLongExpression) {
}
```

Наследование и ключевое слово virtual

∗ При переопределении virtual-метода, ни за что не помещайте слово virtual в заголовочный файл.

Главное исключение

∗ Не бойтесь нарушать описанные выше правила, если вам кажется, что они только запутают ваш код.

Лабораторные работы

Лабораторная работа №1

Задача с 3 массивами

В каждом варианте необходимо обработать три массива разного размера. Необходимо написать функцию, которая выполняет требуемые действия, применить её ко всем трём массивам и сравнить полученные результаты. Все исходные данные задаются на этапе компиляции.

Пример решения

Определить в каком массиве больше среднее арифметическое элементов, меньших заданного числа. Если в двух или трёх массивах значения среднего арифметического совпадают, вывести соответствующее сообщение.

```c
#pragma warning(disable : 4996)
#include <cstdio>
#include <cstdlib>
#include <cstring>

#define MATRIX1 4
#define MATRIX2 2
#define MATRIX3 6

static double m1[] = { 1., 1., 1., 1. };

static double m2[] = { 1., 1. };

static double m3[] = { 4., 4., 4., 4., 4., 4. };

static double
calculate(double matrix[], int elements, const double part)
{
  double result = 0;
  int i;
  int n = 0;

  for (i = 0; i < elements; ++i) {
    if (matrix[i] < part) {
      result += matrix[i];
      ++n;
    }
  }
  return result == 0|| n == 0? 0: result / n;
}

int
main(int argc, char **argv)
{
  const double part = 4.;
  double mr1 = calculate(m1, MATRIX1, part);
  double mr2 = calculate(m2, MATRIX2, part);
  double mr3 = calculate(m3, sizeof(m3)/sizeof(m3[0]), part);

  if (mr1 == mr2)
    fprintf(stdout, "Matrix1 == Matrix2\n");
  if (mr1 == mr3)
    fprintf(stdout, "Matrix1 == Matrix3\n");
  if (mr2 == mr3)
    fprintf(stdout, "Matrix2 == Matrix3\n");
  system("pause");
  return EXIT_SUCCESS;
}
```

1. Определить в каком массиве меньше среднее арифметическое элементов, больших заданного числа. Если в двух или трёх массивах значения среднего арифметического совпадают, вывести соответствующее сообщение.

2. Определить в каком массиве больше количество элементов, меньших заданного числа. Если в двух или трёх массивах количества искомых элементов совпадают, вывести соответствующее сообщение.

3. Определить в каком массиве меньше количество элементов, больших заданного числа. Если в двух или трёх массивах количества искомых элементов совпадают, вывести соответствующее сообщение.

4. Определить в каком массиве минимум элементов больше заданного числа. Если в двух или трёх массивах минимумы совпадают, вывести соответствующее сообщение.

5. Определить в каком массиве максимум элементов меньше заданного числа. Если в двух или трёх массивах максимумы совпадают, вывести соответствующее сообщение.

6. Определить в каком массиве больше сумма элементов, попадающих в заданный диапазон. Если в двух или трёх массивах суммы совпадают, вывести соответствующее сообщение.

7. Определить в каком массиве меньше сумма элементов, не попадающих в заданный диапазон. Если в двух или трёх массивах суммы совпадают, вывести соответствующее сообщение.

8. Определить в каком массиве больше произведение элементов, непопадающих в заданный диапазон. Если в двух или трёх массивах произведения совпадают, вывести соответствующее сообщение.

9. Определить в каком массиве меньше произведение элементов, попадающих в заданный диапазон. Если в двух или трёх массивах произведения совпадают, вывести соответствующее сообщение.

10. Определить в каком массиве меньше среднее арифметическое элементов, меньших заданного числа. Если в двух или трёх массивах значения среднего арифметического совпадают, вывести соответствующее сообщение.

11. Определить в каком массиве больше среднее арифметическое элементов, больших заданного числа. Если в двух или трёх массивах значения среднего арифметического совпадают, вывести соответствующее сообщение.

12. Определить в каком массиве меньше количество элементов, меньших заданного числа. Если в двух или трёх массивах количества искомых элементов совпадают, вывести соответствующее сообщение.

13. Определить в каком массиве больше количество элементов, больших заданного числа. Если в двух или трёх массивах количества искомых элементов совпадают, вывести соответствующее сообщение.

14. Определить в каком массиве меньше минимум элементов, больших заданного числа. Если в двух или трёх массивах минимумы совпадают, вывести соответствующее сообщение.

15. Определить в каком массиве больше максимум элементов, меньших заданного числа. Если в двух или трёх массивах максимумы совпадают, вывести соответствующее сообщение.

16. Определить в каком массиве меньше сумма элементов, попадающих в заданный диапазон. Если в двух или трёх массивах суммы совпадают, вывести соответствующее сообщение.

17. Определить в каком массиве больше сумма элементов, непопадающих в заданный диапазон. Если в двух или трёх массивах суммы совпадают, вывести соответствующее сообщение.

18. Определить в каком массиве меньше произведение элементов, непопадающих в заданный диапазон. Если в двух или трёх массивах произведения совпадают, вывести соответствующее сообщение.

19. Определить в каком массиве больше произведение элементов, попадающих в заданный диапазон. Если в двух или трёх массивах произведения совпадают, вывести соответствующее сообщение.

20. Определить в каком массиве больше среднее арифметическое элементов, меньших заданного числа. Если в двух или трёх массивах значения среднего арифметического совпадают, вывести соответствующее сообщение.

21. Определить в каком массиве меньше среднее арифметическое элементов, больших заданного числа. Если в двух или трёх массивах значения среднего арифметического совпадают, вывести соответствующее сообщение.

22. Определить в каком массиве больше количество элементов, меньших заданного числа. Если в двух или трёх массивах количества искомых элементов совпадают, вывести соответствующее сообщение.

23. Определить в каком массиве меньше количество элементов, больших заданного числа. Если в двух или трёх массивах количества искомых элементов совпадают, вывести соответствующее сообщение.

24. Определить в каком массиве больше минимум элементов, больших заданного числа. Если в двух или трёх массивах минимумы совпадают, вывести соответствующее сообщение.

Лабораторная работа №2

Обработка матриц

При выполнении этого задания необходимо написать две функции. Одна из этих функций должна получать и обрабатывать матрицу целиком. Другая функция должна обрабатывать одномерный массив. В качестве этого одномерного массива передаётся одна строка матрицы. Ввод осуществляется из файлов с использованием аргументов функции main. Вывод–на экран или в файл с обязательным выводом исходных данных.

Пример выполнения задания

0. Даны две матрицы разного размера. Для той из матриц, в которой больше максимальный элемент, найти максимальный элемент в каждой строке.

```c
#pragma warning(disable : 4996)
#include <cstdio>
#include <cstdlib>
#include <cstring>

static double **
__loadMatrix(
   const char * const szFileName,
   int * piRows,
   int * piCols);
static void
__destroyMatrix(double **pMatrix, int rows, int cols);

static int
__exception(const char * const szMessage)
{
 fprintf(stderr, "%s\n", szMessage);
 return EXIT_FAILURE;
}

static void
__printMatrix(double **pMatrix, int rows, int cols);
static double
__findMaxElement(double **pMatrix, int rows, int cols);
static void
__outputMaxElements(double **pMatrix, int rows, int cols);

int
main(int argc, char **argv)
{
 if (argc < 3) {
  return __exception("Not found input file");
 }
 int mRows1 = 0;
 int mCols1 = 0;
 int mRows2 = 0;
 int mCols2 = 0;
 double ** matrix1 = __loadMatrix(argv[1], &mRows1, &mCols1);
 double ** matrix2 = __loadMatrix(argv[2], &mRows2, &mCols2);

 /** Output */
 fprintf(stdout, "Matrix N1:\n");
 __printMatrix(matrix1, mRows1, mCols1);
 fprintf(stdout, "Matrix N2:\n");
 __printMatrix(matrix2, mRows2, mCols2);

 /** Find max element */
 double max1 = __findMaxElement(matrix1, mRows1, mCols1);
 double max2 = __findMaxElement(matrix2, mRows2, mCols2);
 if (max1 > max2) {
  fprintf(stdout, "Output matrix N1:\n");
```

```
    __outputMaxElements(matrix1, mRows1, mCols1);
  } else {
    fprintf(stdout, "Output matrix N2:\n");
    __outputMaxElements(matrix2, mRows2, mCols2);
  }

  __destroyMatrix(matrix1, mRows1, mCols1);
  __destroyMatrix(matrix2, mRows2, mCols2);
  system("pause");
  return EXIT_SUCCESS;
}


/** */
#include <string>
#include <vector>


class NumberFromFileParser
{
  typedef std::vector<double> __MatrixLine;
  typedef std::vector< __MatrixLine > __Matrix;

  friend double **
  __loadMatrix(
     const char * const szFileName,
     int * piRows,
     int * piCols);
public:
  NumberFromFileParser()
    : _ch(0), _fd(nullptr)
  {
  }
  ~NumberFromFileParser()
  {
  }

  bool eof() const
  {
    return feof(_fd) > 0;
  }
  bool next()
  {
    _ch = fgetc(_fd);
    return _ch != EOF;
  }
  bool isWhitespace() const
  {

    return isspace((int)_ch) > 0;
  }
  bool isDigit() const
  {
```

```
104     return isdigit((int)_ch) > 0;
105   }
106   bool isDot() const
107   {
108     return _ch == '.' ||
109       _ch == ',';
110   }
111   bool isSign() const
112   {
113     return _ch == '-' ||
114       _ch == '+';
115   }
116   bool isEndOfLine() const
117   {
118     return _ch == '\r' ||
119       _ch == '\n';
120   }
121   void skipWhitespace()
122   {
123     while (!eof() && isWhitespace() && next());
124   }
125   void skipEndOfLine()
126   {
127     while (!eof() && isEndOfLine() && next());
128   }
129   void parseNumber()
130   {
131     skipWhitespace();
132     while (!eof() && !isWhitespace() &&
133         (isSign() || isDot() || isDigit())) {
134       put();
135       next();
136     }
137   }
138   void put()
139   {
140     _buffer.push_back(_ch);
141   }
142
143   bool parse(const char * const name, __Matrix &matrix)
144   {
145     std::vector<double> row;
146     matrix.clear();
147     if (_fd == nullptr)
148       _fd = fopen(name, "r");
149     if (_fd == nullptr)
150       return false;
151     next();
152     while (!eof()) {
153       _buffer.clear();
154       parseNumber();
```

```
      if (_buffer.size() > 0) {
        row.push_back(atof(_buffer.c_str()));
      }
      if (isEndOfLine()) {
        skipEndOfLine();
        matrix.push_back(row);
        row.clear();
      }
    }
    if (row.size() > 0)
      matrix.push_back(row);
    fclose(_fd);
    _fd = nullptr;
    return true;
  }
private:
  char           _ch;
  FILE          *_fd;
  std::string    _buffer;
};

double **
__loadMatrix(
    const char * const szFileName,
    int * piRows,
    int * piCols)
{
  NumberFromFileParser parser;
  NumberFromFileParser::__Matrix matrix;
  auto retval = parser.parse(szFileName, matrix);
  if (retval) {
    (*piRows) = matrix.size();
    double **result = new double * [matrix.size()];
    for (
       NumberFromFileParser::__MatrixLine::size_type k = 0;
       k < matrix.size();
       k++) {
      NumberFromFileParser::__MatrixLine line = matrix.at(k);
      result[k] = new double [line.size()];
      (*piCols) = line.size();
      for (
         NumberFromFileParser::__Matrix::size_type i = 0;
         i < line.size();
         ++i) {
        result[k][i] = line.at(i);
      }
    }
    return result;
  }
  return nullptr;
}
```

```
206
207 void
208 __destroyMatrix(double **pMatrix, int rows, int cols)
209 {
210   if (pMatrix != nullptr) {
211     for (auto i = 0; i < rows; i++) {
212       if (pMatrix[i] != nullptr)
213         delete[] pMatrix[i];
214     }
215     delete[] pMatrix;
216   }
217 }
218
219 void
220 __printMatrix(double **pMatrix, int rows, int cols)
221 {
222   for (auto i = 0; i < rows; ++i) {
223     for (auto j = 0; j < cols; ++j) {
224       if (j > 0)
225         fprintf(stdout, " ");
226       fprintf(stdout, "%3.5f", pMatrix[i][j]);
227     }
228     fprintf(stdout, "\n");
229   }
230 }
231
232 double
233 __findMaxElement(double **pMatrix, int rows, int cols)
234 {
235   double result = pMatrix[0][0];
236   for (auto i = 0; i < rows; ++i) {
237     for (auto j = 0; j < cols; ++j) {
238       if (result <= pMatrix[i][j])
239         result = pMatrix[i][j];
240     }
241   }
242   return result;
243 }
244
245 void
246 __outputMaxElements(double **pMatrix, int rows, int cols)
247 {
248   for (auto i = 0; i < rows; ++i) {
249     double max = pMatrix[i][0];
250     for (auto j = 0; j < cols; ++j) {
251       if (max <= pMatrix[i][j])
252         max = pMatrix[i][j];
253     }
254     fprintf(stdout, "%3.5f\n", max);
255   }
256 }
```

Варианты:

1. Даны две матрицы разного размера. Для той из матриц, в которой больше максимальный элемент, проверить наличие положительных элементов в каждой строке.

2. Даны две матрицы разного размера. Для той из матриц, в которой больше максимальный элемент, найти количество положительных элементов в каждой строке.

3. Даны две матрицы разного размера. Для той из матриц, в которой больше максимальный элемент, найти сумму положительных элементов в каждой строке.

4. Даны две матрицы разного размера. Для той из матриц, в которой больше максимальный элемент, найти среднее арифметическое ненулевых элементов в каждой строке.

5. Даны две матрицы разного размера. Для той из матриц, в которой есть элементы, равные 0, найти минимальный элемент в каждой строке.

6. Даны две матрицы разного размера. Для той из матриц, в которой есть элементы, равные 0, проверить наличие отрицательных элементов в каждой строке.

7. Даны две матрицы разного размера. Для той из матриц, в которой есть элементы, равные 0, найти количество отрицательных элементов в каждой строке.

8. Даны две матрицы разного размера. Для той из матриц, в которой есть элементы, равные 0, найти произведение ненулевых элементов в каждой строке.

9. Даны две матрицы разного размера. Для той из матриц, в которой есть элементы, равные 0, найти среднее арифметическое положительных элементов в каждой строке.

10. Даны две матрицы разного размера. Для той из матриц, в которой меньше количество нулевых элементов, найти минимальный элемент в каждой строке.

11. Даны две матрицы разного размера. Для той из матриц, в которой меньше количество нулевых элементов, проверить наличие отрицательных элементов в каждой строке.

12. Даны две матрицы разного размера. Для той из матриц, в которой меньше количество нулевых элементов, найти количество отрицательных элементов в каждой строке.

13. Даны две матрицы разного размера. Для той из матриц, в которой меньше количество нулевых элементов, найти произведение ненулевых элементов в каждой строке.

14. Даны две матрицы разного размера. Для той из матриц, в которой меньше количество нулевых элементов, найти среднее арифметическое положительных элементов в каждой строке.

15. Даны две матрицы разного размера. Для той из матриц, в которой меньше среднее арифметическое положительных элементов, найти минимальный элемент в каждой строке.

16. Даны две матрицы разного размера. Для той из матриц, в которой меньше среднее арифметическое положительных элементов, проверить наличие отрицательных элементов в каждой строке.

17. Даны две матрицы разного размера. Для той из матриц, в которой меньше среднее арифметическое положительных элементов, найти количество отрицательных элементов в каждой строке.

18. Даны две матрицы разного размера. Для той из матриц, в которой меньше среднее арифметическое положительных элементов, найти произведение ненулевых элементов в каждой строке.

19. Даны две матрицы разного размера. Для той из матриц, в которой меньше среднее арифметическое положительных элементов, найти среднее арифметическое положительных элементов в каждой строке.

20. Даны две матрицы разного размера. Для той из матриц, в которой больше произведение ненулевых элементов, найти максимальный элемент в каждой строке.

21. Даны две матрицы разного размера. Для той из матриц, в которой больше произведение ненулевых элементов, проверить наличие положительных элементов в каждой строке.

22 Даны две матрицы разного размера. Для той из матриц, в которой больше произведение ненулевых элементов, найти количество положительных элементов в каждой строке.

23 Даны две матрицы разного размера. Для той из матриц, в которой больше произведение ненулевых элементов, найти сумму положительных элементов в каждой строке.

24 Даны две матрицы разного размера. Для той из матриц, в которой больше произведение ненулевых элементов, найти среднее арифметическое ненулевых элементов в каждой строке.

Лабораторная работа №3

Пример выполнения задания:

1. Строка состоит из слов, разделенных одним или несколькими пробелами. Поменяйте местами наибольшее по длине слово и наименьшее.

```
#pragma warning(disable : 4996)
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <cctype>

/** Задание*/
/** Строкасостоитизслов , разделенныходним
*** илинесколькимипробелами. Поменяйте
*** местаминаибольшееподлинесловоинаименьшее
***/

struct __Token
{
  int loc;
  int len;
```

```
};

static void
__do(char *in);
/** Найтиминимальноесловиимаксимальное   */
static void
__find(char *in, struct __Token *min, struct __Token *max);
/** Заменитьдваслова */
static void
__swap(char *in, struct __Token *first, struct __Token *secon
static int
__is_whitespace();

int
main(int argc, char **argv)
{
  char *buffer = strdup("Now that we've got our ServiceHost cr
  fprintf(stdout, "%s\n", buffer);
  __do(buffer);
  fprintf(stdout, "%s\n", buffer);
  free(buffer);
  system("pause");
  return EXIT_SUCCESS;
}

void
__do(char *buffer)
{
  struct __Token min;
  struct __Token max;

  __find(buffer, &min, &max);
  __swap(buffer, &min, &max);
}

#define SKIPWS(In) \
     while (*(In) != 0&& iswspace( (int)(*(In)) )) { \
    ++(In);                                          \
    ++id;                                            \
       }

void
__find(char *in, struct __Token *min, struct __Token *max)
 {
   char *it = in;
   char *next = it;
   int id = 0;


   memset(min, 0, sizeof(struct __Token));
   memset(max, 0, sizeof(struct __Token));

```

```
  SKIPWS(it);
  while (*it != 0) {
    if (iswspace((int)(*it))) {
      int len = it - next;
      if (min->len == 0) {
        min->len = len;
        min->loc = id - len;

        max->len = len;
        max->loc = id - len;
      } else {
        if (len > max->len) {
          max->len = len;
          max->loc = id - len;
        }
        if (len < min->len) {
          min->len = len;
          min->loc = id - len;
        }
      }
      SKIPWS(it);
      next = it;
    }
    ++it;
    ++id;
  }
}


void
__swap(char *in, struct __Token *first, struct __Token *secor
{
  char *out = (char *)calloc(1, strlen(in) + 1);
  struct __Token *d1 = first;
  struct __Token *d2 = second;
  int i = 0;
  int k = 0;

  /** Ищемближайший */
  if (first->loc > second->loc)
    d1 = second, d2 = first;

  /** Копируемдопервого */
  while (in[i] != 0&& i < d1->loc)
    out[i] = in[i++];

  /** Записываемпервоеслово */
  k = d2->loc;
  for (i = d1->loc; i < d1->loc + d2->len; ++i) {
    out[i] = in[k++];
  }
  /** Записываемпродолжение */
```

```
121   k = d1->loc + d1->len;
122   for (; i < d2->loc - d1->len + d2->len; ++i) {
123     out[i] = in[k++];
124   }
125   /** Записываемвтороеслово */
126   k = d1->loc;
127   for (; k < d1->loc + d1->len; ++k, ++i) {
128     out[i] = in[k];
129   }
130   /** Записываемпродолжение */
131   k = d2->loc + d2->len;
132   while (in[i] != 0)
133     out[i++] = in[k++];
134   /** Копируемвременнуюстроку */
135   strcpy(in, out);
136   /** Освобождаемвременнуюстроку */
137   free(out);
138 }
```

1. Строка состоит из слов, разделенных одним или несколькими пробелами. Переставьте слова в алфавитном порядке.

2. Строка состоит из слов, разделенных одним или несколькими пробелами. Переставьте слова по убыванию их длин.

3. Дана строка, представляющая из себя арифметическое выражение, состоящее из чисел, скобок и арифметических операций. Проверьте данное выражение на правильность расстановки скобок.

4. Даны две строки. Выделить из каждой строки наибольшей длины подстроки, состоящие только из цифр, и объедините эти подстроки в одну новую строку.

5. Даны три строки. Заменить все вхождения второй строки в первую на третью строку.

6. Дана строка. Придумать алгоритм шифрования данной строки и дешифрования.

7. Даны две строки, состоящие из слов, разделенных пробелами. Сформировать строку, состоящую из слов, которые:

а) встречаются хотя бы в одной строке;

б) (8) встречаются только в первой строке;

в) (9) встречаются только в одной из строк.

10. Дана строка, состоящая из слов, разделенных пробелами. Сформировать новую строку со следующими свойствами:

а) все слова в нижнем регистре, кроме первой буквы первого слова;

б) (11) все ссылки в словах заменяются на ”[ссылка запрещена]”;

в) (12) все email заменяются на”[контакты запрещены]”;

г) (13) все слова длины более 3 символов, содержащие только цифры, удаляются.

14. В сообщении, состоящем из одних русских букв и пробелов, каждую букву заменили ее порядковым номером в русском алфавите (А – 1, Б – 2, …, Я – 33),а пробел– нулем. Требуется по заданной последовательности цифр найти количество исходных сообщений, из которых она могла получиться.

15. Непустая строка, содержащая некоторое слово, называется палиндромом, если это слово одинаково читается как слева направо, так и справа налево. Пусть дана строка, в которой записано слово s, состоящее из n прописных букв латинского алфавита. Вычеркиванием из этого слова некоторого набора символов можно получить строку, которая будет палиндромом. Требуется найти количество способов вычеркивания из данного слова некоторого (возможно, пустого) набора таких символов, что полученная в результате строка является палиндромом. Способы, различающиеся только порядком вычеркивания символов, считаются одинаковыми.

16. Даны две строки s и w, вывести строку x максимальной длины, состоящую из букв, таких, что существует перестановках, являющаяся подстрокой перестановки s и одновременно являющаяся подстрокой перестановки w.

17. Исключить из строки группы символов, расположенные между символами «/*»,«*/» включая границы. Предполагается, что нет вложенных скобок.

18. Заданный текст является правильной записью римскими цифрами целого числа от 1 до 1999, получить это число.

19. Заданное натуральное число от 1 до 1999 вывести римскими цифрами.

20. Из заданного текста выбрать и напечатать те символы, которые встречаются в нем ровно два раза (в том порядке, как они встречаются в тексте).

21. Проверить, соблюдается ли в заданном тексте баланс открывающих и закрывающих круглых скобок, то есть можно ли установить взаимно однозначное соответствие открывающих и закрывающих скобок, причем открывающая скобка всегда предшествует соответствующей закрывающей.

22. Для встречающихся в заданном тексте пар рядом расположенных символов указать, сколько раз встречается в тексте каждое из таких двухбуквенных сочетаний.

23∗ Определить является ли введенный текст записью целого числа, записью вещественного числа.

24∗ Написать программу получения строки, в которой удалены все «лишние» пробелы, то есть из нескольких подряд идущих пробелов оставить только один.

25∗ Написать программу, которая осуществляет сравнение двух строк.

26∗ Написать программу, реализующую процедуру удаления k символов с позиции номер n из строки S.