МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное образовательное
учреждение высшего профессионального образования
«Чувашский государственный университет имени И.Н. Ульянова»

# ОБЪЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

## Методические указания к лабораторным работам

Чебоксары 2014

УДК 004.657(075.8)
С??

Составитель: канд. техн. наук, доцент И.А Обломов

**С?? Объектно-ориентированное программирование**: методические указания к лабораторным работам/ Сост. И. А. Обломов; Чуваш. ун-т. Чебоксары, 2014. 40с.

Рассмотрены теоретические сведения по темам лабораторных работ. Для каждой темы приведены примеры и варианты индивидуальных заданий. Даны указания, рекомендации к выполнению, основные требования к содержанию и оформлению лабораторных работ.

Для студентов II курса факультета Информатики и вычислительной техники направления 230100 «Информатика и вычислительная техника» (профиль «Программное обеспечение средств вычислительной техники и автоматизированных систем»).

Работа выполнена при финансовой поддержке федеральной программы «Кадры для регионов».

Отв. редактор канд. техн. наук, доцент А. Л. Симаков

Утверждено Методическим советом университета
УДК 004.657(075.8)

*Учебное издание*

**ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ**

**Методические указания к лабораторным работам**

Редактор _____

Подписано в печать __.__.2014. Формат 60×84/16. Бумага газетная.
Печать оперативная. Гарнитура Times New Roman. Усл. печ. л. _,__.
Уч.-изд. л. _,_. Тираж 100 экз. Заказ № ____.

Чувашский государственный университет
Типография университета 428015 Чебоксары, Московский просп

## Лабораторная работа №1.
## Типы данных языка С++.

Язык С++ является строго типизированным. Это означает, что любой программный объект должен быть определенного типа. Данные различных типов хранятся и обрабатываются по-разному. Тип данных определяет:

- внутреннее представление данных в памяти машины;
- множество значений, которые могут применять величины этого типа;
- операции (функции), которые можно применить к величинам этого типа.

От типа величины зависят машинные команды, которые будут использоваться для обработки данных.

Все типы языка С++ делятся на основные и составные. В языке С++ определено шесть основных типов для представления целых, вещественных, символьных и логических величин. На их основе программист может вводить описание составных типов. К ним относятся массивы, перечисления, структуры, указатели и ссылки, объединения и классы.

***Основные типы данных.***

Основные (стандартные) типы часто называют арифметическими, поскольку их можно использовать в арифметических операциях. Для описания основных типов используют следующие ключевые слова: int (целый), char (символьный), wchar_t (расширенный символьный), bool (логический), float (вещественный), double (вещественный с двойной точностью). Первые четыре типа называют целочисленными, т.к. они представляются в машине с помощью целых чисел. Два последних типа называют типами с плавающей точкой. Коды, формируемые компилятором для обработки целочисленных и вещественных данных, будут различными.

Существует четыре спецификатора типа, уточняющих внутреннее представление и диапазон значений стандартных типов: short (короткий), long (длинный), signed (знаковый), unsigned (беззнаковый).

***Целый тип (int).*** Размер типа int стандартом языка не регламентирован, он зависит от реализации компилятора и разрядности процессора. Как правило, размер целого типа совпадает с разрядностью процессора. Так для 16-и разрядного процессора компилятор выделяет 2 байта (16 двоичных разрядов), а для 32-х разрядного – 4 байта.

Спецификатор short перед именем типа указывает компилятору, что под число требуется отвести 2 байта, независимо от разрядности процессора. Спецификатор long требует 4 байта. Так, для 32-х разрядного процессора данное тип int потребует 4 байта, short int – 2 байта, long int – 4 байта.

Использование спецификатора signed (по умолчанию) предполагает, что старший разряд числа интерпретируется как знаковый (0- для положительных чисел, 1- для отрицательных). Спецификатор unsigned позволяет представлять только положительный числа. В зависимости от спецификаторов перед типом int диапазон представления различен.

| 31 | 0 |
|---|---|
| Знак | Значение числа |

Рис. 1. Внутреннее представление типов int, signed int и long для 32-х разрядной платформы

| 31<br>0 |
|---|
| Значение числа |

Рис. 2. Внутренне представление типа unsigned int для 32-х разрядной платформы

| 16 | 0 |
|---|---|
| Знак | Значение числа |

Рис. 3. Представление типа sort int

| 16<br>0 |
| --- |
| Значение числа |

Рис. 4. Представление типа unsigned short int

Возможны и более сложные комбинации типов и спецификаторов, например, unsigned long long int.

***Символьный тип (char).*** Как правило под данные типа char выделяется 1 байт. Этого достаточно для хранения до 256 различных символов кода ASCII. Тип char, как и другие целые типы может быть со знаком или без знака. В величинах со знаком можно хранить величины от -128 до 127. Если используется спецификатор unsigned, то величины могут принимать значения от 0 до 255.

Величины типа char могут использоваться для хранения целых чисел в указанных диапазонах.

***Расширенный символьный тип (wchar_t).*** Тип wchar_t предназначен для хранения символов, для которых 1 байта не достаточно, например, для символов в кодировке Unicode. Размер этого типа соответствует типу short ( 2 байта).

***Логический тип (bool).*** Множество значений этого типа – это true (1) и false (0). Любое другое значение интерпретируется как true. При преобразовании к целому типу true имеет значение 1, а false – 0.

Для величин этого типа компилятор выделяет 1 байт.

***Типы с плавающей точкой (float, double, long double).*** Типы данных с плавающей точкой хранятся в памяти компьютера иначе, чем целочисленные. Под тип float обычно выделяется 4 байта, один разряд из которых отводится под знак мантиссы, 8 разрядов под порядок и 23 – под мантиссу. Мантисса – это число, большее 1 и меньшее 2.

Под величины типа double выделяется 8 байт. Спецификатор long перед double гарантирует выделение 10 байт.

Константы с плавающей точкой имеют по умолчанию тип double.

Диапазоны изменений величин арифметических типов представлены в таблице 1.

Таблица 1.

| Тип данных | Размер памяти, бит | Диапазон значений |
|---|---|---|
| `char` (символьный) | 8 | от–128 до 127 |
| `signed char` (знаковый символьный) | 8 | от–128 до 127 |
| `unsigned char` (беззнаковый символьный) | 8 | от 0 до 255 |
| `short int` (короткое целое) | 16 | от–32768 до 32767 |
| `unsigned int` (беззнаковое целое) | 16 | от 0 до 65535 (16-битная платформа)<br>от 0 до 4294967295 (32-битная платформа) |
| `int` (целое) | 16<br>32 | от–32768 до 32767 (16-битная платформа)<br>от –2147483648 до 2147483647 (32-битная платформа) |
| `long` (длинное целое) | 32 | от–2147483648 до 2147483647 |
| `unsigned long` (длинное целое без знака) | 32 | от 0 до 4294967295 |
| `long long int` (C99) | 64 | от–($2^{63}$–1) до $2^{63}$–1 |
| `unsigned long long int` (C99) | 64 | от 0 до $2^{64}$–1 |
| `float` (вещественное) | 32 | от 3.4E–38 до 3.4E38 |
| `double` (двойное вещественное) | 64 | от 1.7E–308 до 1.7E308 |
| `long double` (длинное | 80 | от 3.4E–4932 до |

| | | |
|---|---|---|
| вещественное) | | 3.4E4932 |
| `bool (C++)` | 8 | `true(1), false(0)` |

Минимальные и максимальные допустимые значения для целых типов зависят от реализации и приведены в заголовочном файле <limits.h> (<climits>), значения вещественных типов – в файле <floath> (cfloat>), а также в шаблоне клсса numeric_limits.

***Тип void.*** Кроме перечисленных стандартных типов, к основным типам относят тип void, множество значений которого пусто. Он используется для определения функций, не возвращающих никакого результата, а также для представления пустого списка параметров. Кроме того этот тип используется как базовый тип для указателей.

***Составные типы данных.***

Как уже было отмечено, к составным типам относят массивы, перечисления, структуры, указатели и ссылки, объединения и классы. Из перечисленных типов будут рассмотрены переименование типов, перечисления и частично структуры. Остальные типы будут подробно рассмотрены в следующих лабораторных работах.

***Переименование типов(typedef).*** В некоторых случаях типу можно дать новое имя. Это позволяет сделать текст программы более ясным. Общий формат преобразования можно представить следующим образом:

typedef тип новое_имя [ размерность ];

В данном контексте квадратные скобки и содержимое между ними следует рассматривать как необязательный элемент конструкции.

Классический пример использования преобразования типа – сокращение длинных имен существующих типов, например:

typedef unsigned int UINT;

С момента объявления этого переименования слово UINT компилятором будет интерпретироваться как unsigned int.

Еще один пример, заменяющий объявление анонимной структуры на конкретное имя:

```
typedef  struct
 {
        char *Name;
        int Age;
        float Ball;
 } Students;
```

Введенные имена теперь можно использовать таким же образом, как и имена стандартных типов:

```
UINT  _x, _y;            // две переменные типа unsigned int
Students group[20];      // массив из 20 структур
```

***Перечисления(enum).*** В некоторых случаях необходимо иметь конечное множество именованных констант, имеющих различные значения. Для этого удобно воспользоваться перечислимым типом данных. Формат перечислимого типа следующий:

```
enum [имя_типа] {список_констант};
```

Необязательное имя типа требуется в случае, если требуется определить переменные этого типа. Компилятор обеспечит, чтобы переменные принимали значения только из списка констант. Константы должны быть обязательно целочисленными и могут быть инициализированы обычным образом. При отсутствии инициализации первая константа обнуляется, а каждая последующая принимает значение большее на 1, чем предыдущая.

Например:

```
enum error{ERR_READ, ERR_WRITE, ERR_CONVERT};
// объявление типа error
error err;  // объявление переменной err типа уккщк
```

Константы ERR_READ, ERR_WRITE, ERR_CONVERT получают соответственно значения 0, 1, 2. Переменная err может принимать значения только из этого диапазона, т. е. 0, 1, 2.

Константам можно присвоить другие значения при объявлении:

```
enum error{ERR_READ = 3, ERR_WRITE = 38, ERR_CONVERT = 99};
```

Над переменными типа перечисления можно выполнять арифметические операции, при этом они автоматически преобразуются к целому типу.

Структуры (struct). В отличие от других типов данных, например массивов, представляющих величины одного типа, структуры могут содержать в себе элементы различных типов. В языке С++ структура является видом класса и обладает всеми его свойствами. Общий формат объявления структуры следующий:

```
struct [ имя_типа ]
 { тип_1 поле_1;
   тип_2 поле_2;
   ……
   тип_n поле_n;
  } [ список_описателей ];
```

Имя структуры (необязательное, при этом такая структура называется анонимной) представляет новый тип, определенный пользователем. Этот тип в дальнейшем может быть использован наравне со стандартными. При этом компилятор не дает никаких привилегий стандартным типам. Если имя структуры отсутствует, должен быть указан список описателей переменных, указателей или массивов.

Элементы структуры, именуемые полями, могут имеет любой тип, кроме типа определяемой структуры, но могут быть указателями на этот тип.

Описание структуры, представляющее собой блок (последовательность операторов, заключенного в фигурные скобки) обязательно должно заканчиваться точкой запятой.

Рассмотрим пример структуры.

```
struct Students
{
        char *Name;
        int Age;
      bool  Sex;
        float Ball;
};
```

Здесь представлен новый тип Students через объявление структуры. Каждый объект этого типа (переменная типа

структуры) в своем составе будет иметь четыре поля типов char *, int, bool и float.

Для данной структуры не определено ни одного действия, кроме операции присваивания, при которой происходит копирование одного объекта в другой методом «поле за полем». Эта операция генерируется компилятором автоматически без участия программиста. Если необходимо специфическая форма копирования, то программист должен об этом позаботиться сам. Все остальные операции над объектами данного типа – обязанность программиста.

Обращение или доступ к полям структуры осуществляется с помощью операции выбора (‘.’ - символ точки) при обращение к полю через имя объекта или с помощью последовательности символов’-‘ и ‘>’, если обращение осуществляется через указатель. Например,

```
Student st, *ptr_st = &st;
st.Name = “ Ivan”;
st.Age = 20;
ptr_st->Sex = 1;
ptr_st->Ball = 4.75;
```

Структура может выступать в качестве типа аргумента функции, т. е. некоторой функции можно передать в качестве аргумента объект типа структура. Также структура может выступать в качестве типа возвращаемого результата некоторой функции.

***«Операции» над типами данных.***

Слово операции в названии заключено в скобки, поскольку операции определены над данными определенных типов, а не над самими типами. Язык С++, являясь объектно-ориентированным, не предполагает, что стандартный тип или тип объявленный пользователем в свою очередь является объектом (переменной) какого-нибудь типа.

Однако, для работы с типами С++ предоставляет следующие возможности:

- определить размер типа в байтах оператором sizeof, в качестве аргумента может быть имя типа, имя переменной или более сложное выражение;

- задать альтернативное имя (синоним) существующему типу оператором typedef;

- средства преобразования одних типов в другие;

- механизм определения типа объекта в реальном времени (RTTI);

***Оператор sizeof.***

Оператор sizeof- это унарный оператор, возвращающий длину в байтах переменной или типа, помещенных в скобки. Например:

```
float var_float =  3.14;
// ……
cout << " Количество байт для  вещественного типа = "
      << sizeof(var_float) << endl;
// ……..
cout << " Количество байт для целого типа = "
      << sizeof(int);
```

В качестве аргумента операции может выступать любой тип, как стандартный, так и определенный пользователем, а также объект или переменная любого типа.

Этот оператор может использоваться при выполнении программы на машинах с процессорами, имеющими различную разрядность, например, 16 и 32. Это позволяет создавать машинно-независимые программы.

С помощью оператора sizeof можно определить размер поля структуры или класса, например для структуры Students:

```
cout  << " Количество байт поля структуры = " << sizeof(st.Age)
Б << endl;
```

***Преобразование типов.*** В языке C++ возможно несколько способов преобразования данных одного типа в другой, если это допустимо. Один из способов состоит в указании необходимого типа, заключенного в скобки перед данным другого типа, в двух формах: тип(выражение) или тип(выражение). Например, оператор (float)var_int; гарантирует преобразование переменной var_int целого типа в вещественный формат. Явное преобразование следует использовать осторожно, т.к. оно является источником возможных ошибок. Понижающее преобразование может привести к потере точности, например, (int)var_float отсечет дробную часть вещественного числа.

В некоторых случаях компилятор автоматически переводит данные одного типа в другой, например, в выражении var_int+var_float, переменная целого типа var_int приводится к вещественному типу, т.е., компилятор выполняет следующие действия (float)var_int+var_float. Такие преобразования относятся к стандартным. К стандартным преобразованиям также относятся и преобразования указателей и ссылок.

Второй способ характерен для последних версий языка C++ и осуществляется с помощью специфических операторов const_cast, static_cast, dynamic_cast, reinterpret_cast, которые будут рассмотрены в следующих работах.

***Механизм определения типа объекта в реальном времени (RTTI).*** Этот механизм позволяет определить, на какой тип в текущий момент времени ссылается указатель. Для доступа к RTTI в стандарт языка введена операция typeid и класс type_info. Класс содержится в заголовочном файле <typeinfo>.

Основная операция typeid, которая применима к основным и производным типам. Метод name возвращает указатель на строку, представляющую имя типа, а метод before выполняет побуквенное сравнение двух имен типов. Для сравнения используется следующая конструкция:
tpeid(T1).before(typeid(T2)). Если имя типа T1 лексикографически предшествует имени T2, результат будет истинным.

Несколько примеров:

```
cout << " Имя типа = " << typeid(var_int).name() << endl;
        // int
cout << " Имя типа = " << typeid(2+3.4).name() << endl;
        // double
cout << typeid(Base).before(typeid(Derved)) << endl;
        // true
```

Информация о типе бывает полезной в диагностических целях.

***Задания для выполнения лабораторной работы***.

Для выполнения работы необходимо открыть пустой проект в среде Visual Studio, набрать программу, содержащую объявление переменных основных типов, а также структуру, состоящую из 3-4 полей.

1. Определить число байт, необходимых для хранения основных типов для данной реализации компилятора и разрядности процессора. По возможности выполнить программу на машинах с разной разрядностью и оценить полученные результаты.
2. Осуществить преобразование отдельных типов с целью улучшения читаемости программы, а также сокращения длинных имен типов.
3. Объявить переменные перечислимого типа без инициализации констант и с их инициализацией. Выполнить допустимые для них операции. Определить число байт, требуемое для хранения таких переменных.
4. Объявить структуру. Определить число байт, требуемое для хранения всех полей структуры. Оценить полученные результаты и сопоставить с результатами по пункту 1.Определить число байт, необходимых для хранения отдельных полей.
5. Пользуясь преобразованием типов, осуществить преобразования объектов к другим типам. Оценить возможность или невозможность преобразований.

6. Пользуясь механизмом определения типа в реальном времени, определить тип фактических объектов и выражений.

## Лабораторная работа №2.
## Базовые конструкции структурного программирования.

Известно, что для решения задачи любой сложности достаточно трех структур, называемых следованием, ветвлением и циклом. Эти конструкции называют базовыми конструкциями структурного программирования. *Следование* – это конструкция, представляющая собой последовательность двух или более операторов, обеспечивающих выполнение программы «сверху вниз». *Ветвление* задает выполнение одного или другого оператора в зависимости от выполнения какого-либо условия (предиката). *Цикл* задает многократное выполнение оператора или последовательности операторов.

***Условный оператор if.*** Используется для разветвления вычислительного процесса два альтернативных направления. Общий формат оператора if следующий:

if(expr) operator_1; else operator_2;

В первую очередь вычисляется выражении expr, которое может относится к арифметическому типу или к типу указателя. Если оно отлично от нуля (значение true), выполняется оператор operator_1, иначе – operator_2. После чего управление передается на оператор, следующий за условным оператором. Альтернативная ветвь, начинающаяся со слова else, может отсутствовать. Каждый из операторов может быть как простым, так и составным.

***Оператор switch***. Оператор switch используется для разветвления процесса на несколько направлений. Его формат следующий:

```
switch(expr)
{
        case конст_выражение_1: список_операторов_1;
        case конст_выражение_2: список_операторов_2;
        ……
        case конст_выражение_N: список_операторов_N;
```

default: операторы;

}

Выполнение оператора начинается с вычисления выражения (результат должен быть целочисленным) и управление передается первому оператору из списка, значение которого совпало с вычисленным. После чего последовательно выполняются оставшиеся операторы, если выход из оператора не указан явно. Обычно для выхода используется оператор break. Если совпадения не найдено, выполняются операторы, следующие за словом default.

***Цикл с предусловием – оператор while.*** Оператор while имеет следующий вид:

while(expr) оператор;

Выражение expr определяет условие повторения тела цикла, которое представляется простым или составным оператором. Для выполнения оператора в теле цикла необходимо чтобы значение выражения не равнялось 0 (false). Оператор может не выполнится ни одного раза.

***Цикл с постусловием – оператор do while.*** Формат этого оператора:

do оператор while (expr);

Этот оператор подобен оператору while, с той лишь разницей, что выражение вычисляется в последнюю очередь. Цикл завершается, если результат выражения равен 0. Оператор do while может выполниться хотя бы один раз.

***Цикл с параметром – for.*** Цикл с параметром имеет следующий формат:

for(инициализация; выражение; модификация) оператор;

В инициализации объявляются и инициализируются переменные - параметры цикла. Здесь можно перечислить несколько переменных, разделенных запятыми. Выражение (точнее его значение) определяет условие продолжения или окончания цикла. Если значение не равно 0, цикл повторяется, иначе завершается. В области модификации описываются выражения, задающие изменения параметров цикла.

К операторам передачи управления относятся следующие: goto, break, continue, return.

Вычислить и вывести на экран в виде таблицы значения функции, заданной с помощью ряда Тейлора на интервале от $x_{нач}$ до $x_{кон}$ с шагом dx с точностью ε. Таблицу снабдить заголовком. Каждая строка таблицы должна содержать значение аргумента, значение функции и количество просуммированных элементов ряда.

***Задания для выполнения лабораторной работы***.

1. **Error!**= 2Σ**Error!** = **Error!** |x| > 1
2. $e^{-x}$ = Σ**Error!** = **Error!**- … |x|<∞
3. ln(x+1) = Σ**Error!** = x**Error!**- … -1 < x ≤ 1
4. ln **Error!** = 2Σ **Error!** = 2**Error!** |x| < 1
5. ln(1 − x) = -Σ**Error!** = **Error!**-1 ≤ x < 1
6. arcctg x = **Error!**+Σ**Error!** = **Error!**- … |x| ≤ 1
7. arctg x = π/2 + Σ**Error!**= **Error!** + … x > 1
8. arctg x = Σ **Error!** = **Error!** + |x| ≤ 1
9. arth x = Σ **Error!** = **Error!** … |x| ≤ 1
10. arth x = Σ **Error!** = **Error!** + … |x| > 1
11. cos x = Σ**Error!**=**Error!** + … |x| < ∞
12. **Error!**= Σ**Error!**= **Error!**- … |x| < ∞
13. ln x = 2Σ**Error!** = **Error!**x > 0
14. ln x = Σ**Error!**= **Error!** 0 < x ≤ 2
15. ln x = Σ **Error!**= **Error!**+ … x > **Error!**
16. arcsin x = x + Σ **Error!** =
    = x + **Error!** + **Error!** +**Error!** + **Error!** + … |x|<1
17. arcos x = **Error!**- (**Error!**+ **Error!** + **Error!** + … ) |x| < 1

## Лабораторная работа №3.
## Одномерные и многомерные массивы.

Массив в С++ представляет собой именованную область памяти, содержащую конечную последовательность однотипных элементов. Описание одномерного массива имеет следующий формат:

тип имя_массива [размерность] = {инициализация};

Для многомерных массивов необходимо задавать каждую размерность в отдельных скобках, например двумерный массив описывается следующим образом:

тип имя_массива [размерность_1] [размерность_2] = {инициализация};

Размерность всегда должна быть целочисленной константой. В общем случае размерность можно не указывать (но пустые скобки обязательны), тогда должна обязательно присутствовать инициализация массива. Компилятор автоматически определит объем памяти необходимый для хранения элементов массива в соответствии с их типом по их фактическому количеству. Элементы массива нумеруются с нуля, а каждый последующий элемент имеет индекс на единицу больший, чем предшествующий.

Обращение к элементам осуществляется по имени массива и указанием индекса. Выход за границы массива компилятором не отслеживается, поэтому при работе с массивами следует проявлять осторожность, особенно при использовании операторов цикла. Неверное задание границ в циклических операторах может привести к неожиданному результату или даже к сбою в работе программы.

При работе с массивами удобно пользоваться указателями (см. лабораторную работу №4), поскольку имя массива компилятор понимает как указатель на его первый элемент.

***Задания для выполнения лабораторной работы***.

**Вариант 1.**

А) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. сумму отрицательных элементов массива;
2. произведение элементов массива, расположенных между максимальным и минимальным элементами;
3. упорядочить элементы массива по возрастанию;

В) Дана целочисленная квадратная матрица. Определить:

1. количество строк, не содержащих ни одного нулевого элемента:

2. максимальное число, встречающееся в заданной матрице более одного раза.

**Вариант 2.**

А) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. сумму положительных элементов массива;
2. произведение элементов массива, расположенных между максимальным по модулю и минимальным по модулю элементами;
3. упорядочить элементы массива по убыванию.

В) Дана целочисленная прямоугольная матрица. Определить количество столбцов, не содержащих ни одного нулевого элемента.

Характеристикой строки целочисленной матрицы называется сумма ее положительных четных элементов. Переставляя строки заданной матрицы, расположить их в соответствии с ростом характеристик.

**Вариант 3.**

А) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. произведение элементов массива с четными номерами;
2. сумму элементов массива, расположенных между первым и последним нулевым элементами;
3. преобразовать массив таким образом, чтобы сначала располагались все положительные элементы, а потом – все отрицательные;

В) Дана целочисленная квадратная матрица. Определить:

1. количество столбцов, содержащих хотя бы один нулевой элемент:
2. номер строки, в которой находится самая длинная серия одинаковых элементов.

**Вариант 4.**

А) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. сумму элементов массива с нечетными элементами;
2. сумму элементов массива, расположенных между первым и последним отрицательным элементами;
3. сжать массив, удалив из него элементы, модуль которых не превышает 1. Освободившиеся в конце массива элементы заполнить нулями.

В) Дана целочисленная квадратная матрица. Определить:

1. произведение элементов в тех строках, которые не содержат отрицательных элементов;
2. максимум среди сумм элементов диагоналей, параллельных главной диагонали матрицы.

**Вариант 5.**

А) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. максимальный элемент массива;
2. сумму элементов массива, расположенных до последнего положительного элемента;
3. сжать массив, удалив из него все элементы, модуль которых находится в интервале [a,b]. Освободившиеся элементы в конце массива заполнить нулями.

В) Дана целочисленная квадратная матрица. Определить:

1. сумму элементов в тех столбцах, которые не содержат отрицательных элементов;
2. минимум среди сумм модулей элементов диагоналей, параллельных побочной диагонали матрицы.

**Вариант 6.**

А) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. минимальный элемент массива;
2. сумму элементов массива, расположенных между первым и последним элементами;
3. преобразовать массив таким образом, чтобы сначала располагались все элементы равные нулю, а затем – все остальные;

В) Дана целочисленная квадратная матрица. Определить:

1. сумму элементов в тех строках, которые содержат хотя бы один отрицательный элемент;
2. номера строк и столбцов всех седловых точек матрицы.

Матрица A имеет седловую точку $A_{ij}$ , если $A_{ij}$ является минимальным элементом в i-й строке и максимальным в j-м столбце.

**Вариант 7.**

A) В одномерном массиве, состоящем из n целых элементов вычислить:

1. номер максимального элемента массива;
2. произведение элементов массива, расположенных между первым и вторым отрицательным элементами;
3. преобразовать массив так, чтобы в первой половине располагались элементы, стоящие в нечетных позициях, а во второй – элементы, стоящие в четных позициях.

B) Для заданной матрицы размером 8 на 8, найти такие k, что k-я строка матрицы совпадает с k-м столбцом. Найти сумму элементов в тех строках, которые содержит хотя бы один отрицательный элемент.

**Вариант 8.**

A) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. номер минимального элемента массива;
2. сумму элементов массива, расположенных между первым и вторым отрицательным элементами;
3. преобразовать массив таким образом, чтобы сначала располагались все элементы, модуль которых не превышает 1, а потом – все остальные.

B) Характеристикой столбца целочисленной матрицы называется сумма модулей его отрицательных нечетных элементов. Переставляя столбцы заданной матрицы, расположить их в соответствии с ростом характеристик.

**Вариант 9.**

A) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. максимальный по модулю элемент;
2. сумму элементов массива, расположенных между первым и вторым положительными элементами;
3. преобразовать массив таким образом, чтобы элементы равные нулю, располагались после всех.

В) Соседями элемента $A_{ij}$ в матрице называются элементы $A_{kl}$ с i-1 <= k <= i+1, j-1 <= l <= j+1, (k,l) != (i,j). Операция сглаживания матрицы дает новую матрицу того же размера, каждый элемент которой получается как среднее арифметическое имеющихся соседей соответствующего элемента исходной матрицы. Построить результат сглаживания матрицы размером 10 на 10.

**Вариант 10.**

А) В одномерном массиве, состоящем из n целых элементов вычислить:

1. минимальный по модулю элемент массива;
2. сумму модулей элементов массива, расположенных после первого элемента, равного нулю;
3. преобразовать массив таким образом, чтобы в первой его половине располагались элементы, стоявшие в четных позициях, а во второй половине – элементы, стоявшие в нечетных позициях.

В) Элемент матрицы называется локальным минимумом, если он строго меньше всех имеющихся у него соседей. Подсчитать количество локальных минимумов заданной матрицы размером 10 на 10.

**Вариант 11.**

А) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. номер минимального по модулю элемента массива;
2. сумму модулей элементов массива, расположенных после первого отрицательного элемента;
3. сжать массив, удалив из него все элементы, величина которых находится в интервале [a,b]. Освободившиеся в конце массива элементы заполнить нулями.

В) Коэффициенты системы линейных уравнений заданы в виде прямоугольной матрицы. С помощью допустимых преобразований привести матрицу к треугольному виду. Найти количество строк, среднее арифметическое элементов которых меньше заданной величины.

**Вариант 12.**

А) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. номер максимального по модулю элемента массива;
2. сумму элементов массива, расположенных после первого положительного элемента;
3. преобразовать массив таким образом, чтобы сначала располагались элементы, целая часть которых лежит в интервале [a.b], а потом – все остальные.

В) Уплотнить заданную матрицу, удаляя из нее строки и столбцы, заполненные нулями. Найти номер первой из строк, содержащей хотя бы один положительный элемент.

**Вариант 13.**

А) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. количество элементов, лежащих в диапазоне от A до B;
2. сумму элементов массива, расположенных после максимального элемента;
3. упорядочить элементы массива по убыванию модулей элементов.

В) Осуществить циклический сдвиг элементов прямоугольной матрицы на n элементов вправо и вниз.

**Вариант 14.**

А) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. количество элементов массива, больших заданного числа;
2. сумму элементов массива, расположенных после минимального элемента;
3. упорядочить массив по возрастанию модулей элементов.

В) Осуществить циклический сдвиг элементов квадратной матрицы размерности M*N вправо на k элементов таким образом: элементы 1-й строки сдвигаются в последний столбец сверху вниз, из него – в последнюю строку справа налево, из нее в первый столбец снизу вверх, из него – в первую строку; для остальных элементов – аналогично.

**Вариант 15.**

А) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. количество элементов массива, равных 0;
2. произведение элементов массива, расположенных после максимального по модулю элемента;
3. преобразовать массив таким образом, чтобы сначала располагались все отрицательные элементы, а потом все положительные (0 считается положительным числом).

В) Упорядочить строки целочисленной прямоугольной матрицы по возрастанию элементов в каждой строке.

Найти номер первого из столбцов, не содержащих ни одного отрицательного элемента.

**Вариант 16.**

А) В одномерном массиве, состоящем из n вещественных элементов вычислить:

1. количество отрицательных элементов массива;
2. сумму модулей элементов массива, расположенных после минимального по модулю элемента:
3. заменить все отрицательные элементы массива их квадратами и упорядочить массив по возрастанию.

В) Путем перестановки элементов матрицы добиться того, чтобы ее максимальный элемент находился в верхнем левом углу, следующий по величине элемент в позиции (2,2)б следующий – в позиции (3,3) и т. д., то есть заполнить главную диагональ матрицы.

Найти номер первой из строк, не содержащей ни одного положительного элемента.

## Лабораторная работа №4.
## Указатели.

Указатели – особый вид переменных языка С++, содержащих в себе адреса ячеек памяти. Различают три вида указателей: указатель на объект какого-либо типа (стандартного или определенного пользователем), указатель на функцию и указатель на тип void. Рассмотрим их вкратце.

***Указатель на объект*** в общем виде описывается следующим образом:

тип_указателя *имя_указателя = инициализация;

Инициализация указателя при объявлении не обязательна, но желательна. Инициализация выполняется с помощью операции взятия адреса объекта - &имя_объекта, или же путем присваивания значения уже инициализированного указателя. Например:

```
int  var_int = 38;                  // переменная целого типа
int *ptr_int_1 = &var_int;          // первый указатель на переменную целого типа
int *ptr_int_2 = ptr_int_1;         // второй указатель на переменную целого типа
```

Для обращения к значению, на который указывает указатель, используют операцию разадресации (разыменования), используя символ ‘*’ перед именем указателя, например,

```
cout <<  “ Значение переменной целого типа = “ << *ptr_int_1 << endl;
```

Адрес переменной (т. е. значение самого указателя) можно получить по имени указателя без его разыменования, например,

```
cout << “ Адрес переменной целого типа = “ << ptr_int_1 << endl;
```

***Задание №1***. Пользуясь результатами, полученными в первой лабораторной работе, объявить переменные стандартных типов. Объявить указатели на них, получить значения адресов объектов стандартного типа и значений этих объектов.

***Задание №2.*** Объявить переменные типа перечисление и указатели на объекты типа перечисления. Показать пример работы с этими переменными через указатели.

Обращение к полям структуры, объявленной через указатель, осуществляется через операцию доступа, обозначаемую через символ ‘->’, стоящим между именем указателя на структуру и именем поля.

***Задание №3.*** Пользуясь структурой, объявленной в работе №1, объявить указатель на нее. Через указатель вывести на экран монитора значения всех полей структуры.

***Задание №4.*** Исследовать, возможны ли преобразования переменных стандартных типов через указатели на них.

С указателями очень часто упоминают о массивах, так как имя массива в С++ понимается как указатель на его первый элемент. Обращение к элементам массива через указатели сокращает время обработки массивов и предпочтительнее с точки зрения семантики программы. Рассмотрим пример, имеется массив целых чисел, необходимо вывести элементы массива на экран монитора.

```
int array_int[10] = {23, 4 -378, 0, -64, 5 ,11};
// …
for(int i=0; i<10; i+)
  cout << “ Элемент [ “ << i << “] = “ << *(array_int+i) << ‘ ‘;
  cout << endl;
//…
```

Пользуясь системным таймером можно приблизительно оценить время обработки массива с помощью операции индексирования и сопоставить со временем обработки с помощью указателей.

***Задание №5.*** Используя одномерный массив, описанный в лабораторной работе №3, организовать его обработку, пользуясь указателями.

***Задание №6.*** Выполнить задание лабораторной работы №3 для многомерного массива, объявив его в динамической области памяти, используя операции new и delete.

Передача массивов в качестве параметров функции осуществляется только через указатели. Общий формат выглядит следующим образом:

тип_результата имя_функции(тип_массива имя_массива[размерность]); или

тип_результата имя_функции(тип_массива имя_массива[], const int размерность); или

тип_результата имя_функции(тип_массива *имя_массива, const int размерность);

***Задание №7.*** Для пункта А) лабораторной работы №3 выполнить подпункт 1, передав одномерный массив как параметр функции. Выполнить обработку массива внутри функции согласно заданию.

***Указатель на функцию.*** Этот вид указателя позволяет передать в качестве параметра функции другую функцию, то есть осуществлять косвенный вызов функции. Следующий пример показывает способ использования указателя на функцию.

```
// объявление функции
int fun(int arg)
{
        return arg*2;
}

// объявление указателя на функцию, имеющую
//один параметр целого типа и тип возвращаемого значения int
typedef int (*PF)(int);

// определение функции, получающей указатель на другую функцию как параметр
void exemple(PF ptr_fun, int number)
{
        cout  << " Вызов функции через указатель " << ptr_fun(number) << endl;
}
```

```
//…

// передача функции другой функции через указатель при вызове
exemple(fun, 10);
//
```

***Задание №8.*** Определить функцию, обрабатывающую массив по пункту А) и подпункту 2, передав его в качестве параметра. Объявить указатель на эту функцию и передать его некой другой функции, имеющей один из возможных параметров указатель на первую функцию.

***Указатель на тип void.*** Является базовым для всех типов указателей. К нему можно привести любой тип указателя (но не, наоборот!), в случае, если при написании программы не известен тип фактического указателя. При работе с указателем на тип void, его необходимо привести к типу фактического указателя. Следующий пример показывает способы работы через указатель на тип void.

```
int var_int = -38;
double var_double = 3.64;
int *ptr_int = &var_int;
double *ptr_double = &var_double;
void *ptr_void;
//
ptr_void  = ptr_int;
cout << " Целая переменная через указатель на void = "
     << *(int *)ptr_void << endl;
ptr_void  = ptr_double;
cout << " Вещественная переменная с дв. Точностью через указатель на void = "
     << *(double *)ptr_void << endl;

// int *ptr_int= ptr_void;  Это ошибочное действие, но
// int *ptr_int= (int *)ptr_void;   Можно, объясните почему? И что можно дальше делать с //указателем  ptr_int?
```

***Задание №9.*** Объявить объекты стандартных типов и указатели на них, а так же объекты тип структура, вывести значения этих объектов через указатели на них, и через указатель на тип void. Осуществить все необходимые преобразования.

## Лабораторная работа №5.

## Функции. Перегрузка функций. Шаблоны функций.

*Функция* – это именованная последовательность описаний и операторов, выполняющая законченное действие. Функция в С++ понимается как простейший способ модульности программы. Для эффективной работы функции достаточно знать ее интерфейс, т.е. имя функции, количество аргументов и их тип, а так же тип возвращаемого результата. Реализация функции может быть скрыта от пользователя, это свойство в ООП называется *инкапсуляцией*.

Функция, перед ее вызовом, должна быть объявлена и определена. Объявление функции (прототип, заголовок) содержит имя функции, тип возвращаемого результата и список параметров. Определение помимо объявления содержит еще тело функции, представляющее собой блок, заключенный в фигурные скобки.

Общий формат объявления функции следующий:

```
[класс] тип_результата имя_функции ([список параметров])[throw]
{
        // тело функции – последовательность
        // описаний и операторов
}
```

Необязательный модификатор класс задает область видимости функции, используя ключевые слова extern и static:

- extern – глобальная видимость во всех модулях проекта (по умолчанию);
- static – видимость только в пределах данного модуля.

Тип возвращаемого результата может быть любым, кроме массива и функции. Если функция не должна возвращать результата (аналог процедуры), указывается тип void.

Список параметров определяет тип и количество передаваемых параметров. Элементы разделяются запятыми, перед каждым из них должен присутствовать тип параметра. Если функция не должна принимать параметров указывается тип void или пустые скобки – ().

Если функция участвует в обработке исключительных ситуаций, указывается ключевое слово throw, о чем речь пойдет в одной из следующих лабораторных работ.

Пример функции суммирования двух целых чисел.

```
#include<iostream>
using namespace std;
// прототип функции
int summa(int,int);        // имена параметров указывать не обязательно

int main()
{
        int var_1, var_2;
        cout << " введите первое слагаемое: "; cin >> var_1;
        cout << " введите второе слагаемое: "; cin >> var_2;
        // вызов функции
        cout << " сумма = " << summa(var_1, var_2) << endl;
        return 0;
}

// определение функции
int summa(int arg_1, int arg_2)
{ return arg_1+arg_2; }
```

Функция может быть определена как встроенная с помощью модификатора inline. Отличие состоит в том, что при вызове inline-функции не происходит передача управления на

код, представляющий эту функцию, а осуществляется подстановка самого кода в точку вызова. Это позволяет снизить накладные расходы, связанные с вызовом функции, но может увеличить объем исполняемого модуля. Использование inline-функций носит рекомендательный характер.

Все величины, объявленные внутри функции, а также ее формальные параметры считаются локальными. Областью их действия является функция, время жизни – время работы функции. Ссылаться на них вне функции нельзя.

Если необходимо сохранить значение некоторой локальной переменной от вызова к вызову, ее снабжают модификатором static, например:

```
void function()
{
        static int count = 0;
        while(exp)
        {
                // …
                cout << “  статическая переменная = “ << count++ << endl
        }
// …
}
```

Значение переменной count будет храниться до следующего вызова в сегменте данных. Это позволяет организовать подсчет числа вызовов той или иной функции.

В теле функции видны все объекты, являющиеся глобальными, а, следовательно, их можно изменять. Подобный прием используется для расширения интерфейса между отдельными функциями. Тем не менее, делать этого не рекомендуется, поскольку затрудняет отладку программы и препятствует помещению подобных функций в общую библиотеку.

Функция не может вернуть в качестве результата указатель на локальную переменную, например,

```
int *function()
{
        int var = 38;
```

```
        return &var;
}
```

Это происходит по той причине, что после выхода из функции, память, занятая ее локальными параметрами освобождается.

Механизм передачи параметров функции является основным способом обмена информацией между функциями. Различают два способа передачи параметров в функцию: *по значению и по адресу*. При передаче по значению в стек заносятся копии фактических параметров и функция работает именно с копиями. Любые изменения параметров не приведут к изменению исходных значений. При передаче по адресу в стеке формируются копии адресов параметров, а функция осуществляет доступ к параметрам по полученным копиям. Как результат функция может изменить значения исходных параметров. Передача параметров по адресу может быть реализована через указатели или через ссылки. Следующий пример показывает использование различных способов передачи параметров.

```
// прототип функции
void function(int, int *, int &);


int main()
{
        int i = 10, j = 20, k = 30;
        cout << " До:  i= " << i << "  "  << " j= " << j <<  "
" << " k= " << k << endl;
        function(i, &j, k);
        cout << " После:  i= " << i << "  "  << " j= " << j <<
" " << " k= " << k << endl;
        return 0;
}
// определение функции
void function(int a, int *b, int &c)
{
```

```
        a++; (*b)++; c++;
}
```

Первый параметр передается по значению; его изменение в теле функции (a++ ) никак не отразится на исходном (передаваемом) значении переменной i. Второй и третий параметры передаются по адресу, в частности, второй с помощью указателя, а третий – с помощью ссылки. Различие между указателем и ссылкой очевидно, указатель необходимо разыменовывать. Изменение фактических переданных параметров j и k будет. Выполните эту программу и убедитесь в этом.

*Передача массивов* в функцию как параметров, а также получение массива как результата работы функции, разрешается только через указатели. Рассмотрим передачу массива в качестве параметра на простом примере – суммирования элементов целочисленной матрицы.

```
int summa(const int *arr, const int size)
{
        // или int summa(int arr[], const int size)
        // или int summa(int arr[size]) этот вариант менее предпочтителен
        int sum =0;
        for(int i = 0; i<size; i++)
        sum += arr[i];
        return sum;
}
```

*При передаче многомерных массивов* все размерности должны передаваться в качестве параметров, например,

```
int summa(const int *arr, const int nstr, const int nstb);, как варианты
int summa(const int **arr, const int nstr, const int nstb);
int summa(const int arr[][], const int nstr, const int nstb);
```

В первом случае объявляется как одномерный массив указателей на массив, во втором – как указатель на указатель, в третьем – явное объявление двумерного массива.

В языке С++ нельзя передать функцию в качества параметра другой функции, но *указатель на функцию* передать можно. Для этого необходимо объявить указатель на функцию. Рассмотрим на примере.

```
//  прототип функции
int summa(int, int);

// указатель на функцию
typedef int ( *ptr_fun)(int,int);

// функция, получающая в качестве параметра указатель на другую функцию
void function(ptr_fun pf)
{
        cout << " сумма = " << pf(38,64) << endl;
}
```

Функция называется *рекурсивной*, если она вызывает саму себя. Классический пример рекурсивной функции – вычисление факториала числа.

```
long factorial(long n)
{
        If(n==0||n++1) return 1;
        return (n + factorial (n -1));
}
```

Любая рекурсивная функция может быть преобразована в не рекурсивную функцию с помощью операторов цикла. Использование рекурсивных функций предпочтительно из-за их более лаконичного описания, однако накладные расходы возрастают.

*Перегружаемые функции*, то есть, функции с одним именем – одна из особенностей языка С++, позволяющая выполнить один и тот же алгоритм для параметров разного типа. Перегружаемые функции позволяют реализовать слабую форму *полиморфизма* – одну из основ ООП. Компилятор самостоятельно определяет, какую именно функцию вызвать по

типу фактических параметров. Рассмотрим пример перегруженных функций.

```
int summa(int arg_1, int arg_2)
{
        return arg_1+arg_2;
}
double summa(double arg_1, double arg_2)
{
        return arg_1+arg_2;
}
```

Первое, что выполняет компилятор, осуществляет выбор, соответствующий типу фактических параметров вариант функции. Если точного соответствия типов не найдено, выполняется продвижение типов в соответствии с общими правилами преобразования типов, например, bool и char в int, float, double, указателей в тип void * и т. д. Следующим шагом является преобразование типов, заданных пользователем. Если соответствия не нашлось, вызов считается неоднозначным и выдается сообщение об ошибке.

При определении перегружаемых функций следует руководствоваться следующими правилами:

- функции должны находиться в одной области видимости;
- функции могут иметь параметры по умолчанию, при этом значения одного и того же параметра в разных функциях должны совпадать;
- функции не считаются перегруженными, если описание их параметров отличается модификатором const или использованием ссылки;
- функции не считаются перегруженными, если они возвращают результаты различных типов.

Более универсальным средством параметризации алгоритма являются *шаблоны функций*. С помощью шаблона функции можно определить алгоритм, который будет применяться к данным различных типов, а конкретный тип данных передается функции на этапе компиляции. Компилятор

автоматически генерирует код, соответствующий переданному типу. Общий формат объявления шаблона функции следующий:

```
template <class Type> заголовок _функции_шаблона
{
        // тело функции
}
```

Вместо слова Type можно использовать произвольный идентификатор пользователя. Конструкция <class Type>, именуемая списком шаблона может содержание произвольного числа типов, например, <class T1, class T2, class T3>. Слово class здесь можно заменить на слово typename. Рассмотрим пример:

```
template<class Type>  Type summa(Type arg_1, Type arg_2)
{
        return arg_1+arg_2;
}
```

Вызов шаблона функции ничем не отличается от вызова обычной функции, для данного случая, cout << " Сумма = " << summa(20,30). Всю работу по генерации последовательности кода, соответствующую типу фактических параметров, берет на себя компилятор. Программист может "облегчить участь" компилятора, указав явно передаваемый тип в угловых скобках между именем функции и списком фактических параметров , например, cout << " Сумма целых чисел = " << summa<int>(20,30); или cout << " Сумма вещественных чисел = " << summa<float>(20.5,30.6);. Подобный прием называют специализацией шаблона.

Шаблоны функций можно перегружать как обычные функции.

Для каждого варианта выполнить следующие задания.

***Задание №1.*** *Передача в функцию параметров стандартных типов.* Написать функцию вывода таблицы значения функции из лабораторной работы №2 для аргументов, изменяющихся в заданных пределах с заданным шагом, с точностью ε. Значение аргумента и точность передать в качестве параметров функции.

***Задание №2.*** *Передача в функцию указателя на функцию.* Пользуясь функцией из задания №1, объявить указатель на нее и передать его как параметр некоторой другой функции.

***Задание №3.*** *Передача одномерных массивов в функцию.* Пользуясь массивом, определенным в пункте А лабораторной работы №3, определить функции, реализующие подпункты данного пункта.

***Задание №4.*** *Передача строк в функцию*. Определить функцию, считывающую строку символов (длина строки не более 100 символов), подсчитать, сколько в каждой строке числовых символов.

***Задание №5.*** *Передача многомерных массивов в функцию.* Пользуясь массивом, определенным в пункте В лабораторной работы №3, определить функции, реализующие подпункты данного пункта.

***Задание №6.*** *Передача структур в функцию*. Определить функцию, получающую в качестве аргумента структуру и выводящую поля данной структуры.

***Задание №7***. *Рекурсивные функции*. Написать функцию упорядочивания массива по возрастанию, используя рекурсию.

***Задание №8.*** *Перегружаемые функции*. Пользуясь заданием №3 данной работы, перегрузить функцию для массивов типов int и double.

***Задание №9.*** *Шаблоны функции*. Определить шаблон функции, реализующий подпункт 1 пункта В (или пункт В) лабораторной работы №3 для произвольных арифметических типов. Вызвать шаблон как обычную функцию и со спецификатором шаблона.

## Лабораторная работа №6.
## Классы.

***Основные сведения.***

Класс – абстрактный тип данных, определяемый пользователем, и представляющий собой модель реального мира в виде данных и функций для работы с ними. Данные, содержащиеся в классе, называются полями, а функции – методами. Общий формат объявления класса выглядит следующим образом:

```
class имя
{
        private:
        // описание скрытых компонентов класса
        public:
        // описание доступных компонентов
};
```

Точка с запятой в конце описания – обязательный элемент.

Ключевые слова (спецификаторы) private, public управляют видимостью компонентов класса. Слово private (для класса этот спецификатор по умолчанию) запрещает обращение к полям класса из вне, а public (обязательный спецификатор) - объявляет поля как общедоступные. Все компоненты, описанные после public, часто называют интерфейсом класса. Количество и порядок следования спецификаторов стандартом не оговаривается.

Поля класса имеют следующие специфические особенности:

- могут иметь любой тип, кроме типа объявляемого класса, но могут быть указателями или ссылками на данный класс;
- могут быть объявлены с модификатором const, при этом инициализируются с помощью конструкторов специального вида;
- могут быть объявлены с модификатором static, но никак не auto, register или extern.

Инициализация полей класса при описании не допускается.

Пример:

```
class Small
//определение класса
{
        private:
           int small;                    // поле класса
        public:
           void setdata(int s)           //методы класса
             { small = s; }
           void showdata()
             { cout << “  Значение поля: “ << small << endl; }
};
```

Этот простой класс содержит всего одно поле и два метода для доступа к нему. Первый метод позволяет присвоить полю некоторое значение, второй – выводит это значение на экран. Объединение данных и функций (инкапсуляция) является стержневой идеей объектно-ориентрованного программирования.

Объявление объекта класса аналогично объявлению переменных стандартных типов, например,

```
int var_int;
double var_double;          // объявление переменных стандартных типов
Small small_1, small_2;     // объявление переменных (объектов) типа Small
```

Объект класса Small находится в таком же отношении к своему классу, в каком переменная находится по отношению к своему типу. Объект является

экземпляром класса так же, как автомобиль является экземпляром класса средств передвижения.

Обращение к объектам класса допускается только через компоненты, описанные после спецификатора public. Например, для объявленного объекта small_1.setdata(38); производится присваивание значения 38 полю small. Непосредственное обращение к скрытому полю, например, small_1.small = 664;, запрещено. Сокрытие данных класса защищает данные класса от несанкционированного доступа через операцию присваивания или через функции, не принадлежащие данному классу.

Число полей и методов класса теоретически может быть любым.

Определение методов класса Small – setdata(int) и showdata() осуществлено непосредственно в самом классе. По умолчанию функции, определенные в теле класса, считаются как встроенные, то есть со спецификатором inline. Подобным образом желательно определять функции небольшого объема (2-5 строк), а функции, имеющие больший объем лучше выносить за пределы класса, оставляя внутри класса их прототипы. Общий формат определения функции за пределами класса выглядит следующим образом: тип_результата имя_класса :: имя_функции(список параметров)

```
{
        // тело функции;
}
```

Для рассмотренного класса Small:

```
class Small
{
        private:
           int small;
        public:
           void setdata(int);
```

```
            void showdata();
};

void Small :: setdata(int s)
      { small = s; }
void Small :: showdata()
       { cout << “  Значение поля: “ << small << endl;
}
```

Подобный прием позволяет существенно уменьшить объем объявляемого класса, а вынесенные методы записать в заголовочные файлы и файлы реализации. Следует отметить, что методы, определение которых вынесено за пределы класса не являются встраиваемыми. Перед их определением необходимо указывать явно ключевое слово inline при необходимости.

***Задание №1.***

Определить простой класс, представляющий описание членов семьи, объявив поля для хранения фамилии, имени, отчества, возраста, пола, статуса (учащийся, рабочий, пенсионер). Объявить методы, позволяющие вводить данные для каждого член семьи, а также обеспечить вывод этих данных на экран. Попытайтесь обратиться к скрытым полям класса не через методы класса, а напрямую, объясните реакцию компилятора.

***Локальные классы.***

Класс может быть глобальным, то есть объявленным вне какого-либо блока и локальным, объявленным в пределах блока (например, функции или другого класса).

Некоторые особенности локальных классов:

- внутри локального класса можно использовать статические и внешние переменные, внешние функции из области (блока), в которой он описан. Автоматические переменные из данной области использовать нельзя;

- локальный класс не может иметь статических компонентов;

- методы локального класса могут быть описаны только в теле самого класса;

- если некоторый класс описан в теле другого, то доступ к элементам внешнего класса осуществляется по общим правилам.

Пример:

```
int func(int val)                    // определение функции
{
        class Test                   // локальный класс Test
        {
                int test;
        public:
                Test(){};
                Test(int t)
                {
                        test = t;
                }
                int ret()
                {
                        return test;
                }
};                                   // конец определения
класса Test
        Test tst(val);               // объявление объекта
типа Test
        return(tst.ret());
        }
```

Необходимо отметить, что областью видимости класса Test является блок функции funct.

Аналогично можно определить класс в теле другого класса, общая схема подобного действия описана ниже.

```
class Extern    // определение внешнего класса
{
   private:
        //  приватные поля внешнего класса;
   public:
        // открытые поля и функции
        // …
        class Inter      // определение внутреннего класса
        {
                         // определение полей и методов локального класса
        };               // конец определения локального класса Inter
};                       // конец определения внешнего класса Extern
```

***Задание №2.***

Определить класс из задания №1 в теле произвольной функции, которая последовательно выводит данные о каждом члене семьи.

Расширьте определение класса из первого задания, добавив в определение внешнего класса локальный класс, содержащий элементы, описывающие доходы членов семьи.

***Константные методы класса.***

Среди методов класса необходимо выделить отдельный вид – константные методы. Эти методы

объявляются с ключевым словом const, следующим сразу за списком формальных параметров. Константные методы (их чаще называют безопасными) предназначены для работы с константными объектами определяемого класса, изменять значение полей которых не допускается. Они имеют специфические особенности:

- объявляется с ключевым словом const;
- не может менять значение полей класса;
- может вызываться для любых объектов, в том числе и не константных.

Для класса Small, рассмотренного в первом примере, безопасным методом может быть только showdata(), так как он только выводит значения полей, не меняя их значений. Для полноты использования константного метода дополним класс Small конструктором (специального вида функцией), который даст возможность инициализации константного объекта.

```
class Small
{
        private:
           int small;
        public:
          Small(int sm)
            { small = sm;}
           void setdata(int s)
            { small = s; }
           void showdata() const                    //
константный метод
             { cout << “  Значение поля: “ << small <<
endl; }
};
```

Теперь мы вправе объявить константный объект класса Small:

const Small const_small(3864);

и вызвать метод const_small.showdata();.

Применять не константный метод по отношению к const_small нельзя, так вызов const_small.setdata(745); приведет к ошибке.

В отдельных изданиях константные методы называют привилегированными составляющими класса.

***Задание №3.***

Дальнейшая модификация класса семейных отношений состоит в переопределении (добавлении) методов, не меняющих значения полей класса в константные. Объявите константные объекты данного класса и испытайте действие константных и не константных методов. Оцените и объясните реакцию компилятора на подобные действия.

***Конструкторы.***

Инициализация полей класса при объявлении объектов данного типа производится с помощью составной функции класса, называемой конструктором. Имя этой функции идентично имени класса, а ее идентификатор может быть перегружен произвольное число раз. Это дает возможность для объектов класса применять инициализаторы с разными типами аргументов и в разных количествах. Если пользователь при определении класса не объявил ни одного конструктора, компилятор автоматически создает конструктор без параметров с заголовком Name(), где Name – имя создаваемого класса. Такой конструктор называется *конструктором по умолчанию*.

Вызов конструктора осуществляется автоматически при объявлении объекта. Если такой вызов является инициализатором объекта, то заключенные в круглые

скобки аргументы могут быть размещены сразу же за идентификатором объявляемого объекта. Предположим, что пользователь определил класс для работы с комплексными числами Complex. Следующие примеры показывают способы инициализации объектов при их объявлении:

Complex complex_number_1 = Complex(23.4, -4.5);, что эквивалентно объявлению

Complex complex_number_1(23.4, -4.5);.

Complex complex_number_2 = Complex(); или

Complex complex_number_2; .

Определим основные свойства конструкторов:

- конструктор не возвращает значение, даже типа void;

- класс может содержать несколько конструкторов с разным количеством и типом параметров для разных видов инициализации объектов. Можно сказать, что конструкторы класса, перегружаемые функции;

- конструктор, не имеющий параметров, называется конструктором по умолчанию. Компилятор автоматически создает таковой, если пользователь не создал ни одного конструктора;

- конструкторы могут иметь параметры любого типа кроме типа создаваемого класса. Конструктор, имеющий единственный параметр – константную ссылку на определяемый класс называется конструктором *копирования (методом «поле за полем»)*. Компилятор автоматически создает такой конструктор, если пользователь не определил его самостоятельно;

- конструкторы не наследуются;

- конструкторы нельзя описывать с модификаторами const, virtual, static;

- конструкторы глобальных объектов вызываются до вызова функции main. Локальные объекты создаются

сразу после того, как становится активна их область действия, например, при передаче объекта в качестве фактического параметра некоторой функции.

Примеры объявления конструкторов.

```
class Child
{
    private:
        char *Name;
        int Age;
    public:
        Child(char *name, int age)              //
конструктор с двумя параметрами
        {
          Name = name; Age = age;
        }
        void Out()
        {
            cout << " Name = " << Name << " Age =  "
<< Age << endl;
        }
};

// объявление объектов типа Child
Child child_1(" Ваня ", 6), child_2(" Ира ", 15);
```

Объявление объектов без инициализации, например, Child child_2; для данного типа не возможно, так как уже описан конструктор с параметрами и компилятор не создает автоматически конструктор по умолчанию. Исправить эту «ошибку» можно, самостоятельно, добавив в тело класса, конструктор по умолчанию: Child(){};.

***Задание №4.***

Как уже говорилось, число конструкторов может быть произвольным. Расширьте класс семейных отношений объявлением 3-4 конструкторов, имеющих различное количество параметров и различные их типы, позволяющих инициализировать объекты различными способами.

***Список инициализации конструктора.***

Инициализация полей объекта может проводиться не только с помощью операторов, содержащихся в теле конструктора, но также с помощью списка инициализации, который находится в заголовке конструктора. Список инициализации отделяется от заголовка двоеточием (:) и состоит из записей вида field(list), где field – идентификатор поля, a list – список выражений.

*Объектные, константные и ссылочные поля класса инициализируются только с помощью списка инициализации!*

*Статические поля класса инициализируются вне класса!*

Выполнение действий, определенных конструктором, состоит из обработки списка инициализации, а затем – тела конструктора.

Список инициализации обязателен также при определении производного класса от некоторого базового, если в базовом классе определен хотя бы один конструктор, требующий инициализации.

В качестве примера рассмотрим определение класса Child, в котором конструктор имеет список инициализации.

```
class Child
{
    private:
        char *Name;
```

```
        int Age;
    public:
        // конструктор с двумя параметрами и списком инициализации
        Child(char *name, int age): Name(name), Age(age){};
        // тело конструктора пусто
        void Out()
        {
            cout << " Name = " << Name << " Age =  " << Age << endl;
        }
};
```

Наличие в данном классе конструктора со списком инициализации не обязательно, так как нет ни константных, ни объектных и ни ссылочных полей. Следующий пример содержит константные и ссылочные поля.

```
class Pair
{
    private:
        const int Fix;              // константное поле
        const int &Ref;             // константное и ссылочное поле
    public:
        Pair(int fix, int ref): Fix(fix), Ref(ref){};
        void  Out()
         {
                cout << Fix + Ref;
         }
};
int Num = 64;
int main()
```

```
{
        Pair Twin(330, Num);
        Twin.Out();
        return 0;
}
```

Конструктор Pair::Pair(int,int) не может быть представлен в следующем виде

```
Pair(int fix, int ref)
{
        Fix = fix;
        Ref = ref;
}
```

***Задание №5.***

Определить собственный класс, моделирующий целочисленные (вещественные) величины. Предусмотреть наличие константных и ссылочных полей. Проанализировать сообщения компилятора при попытке инициализации их операторами в теле конструкторов.

***Инициализация объектных полей.***

Поле считается объектным, если оно имеет тип класса (структуры), определенного или объявленного ранее. Следующий пример показывает использование объектного поля.

```
class  Integral
{
    private:
        int Fix;
    public:
        Integral(int val)
        {
           Fix = val;
        }
        Integral()
```

```
        {
            Fix  = 10;
        }
        int  Get()
        {
                Return Fix;
        }
};

class Rational
{
    private:
        Integral Num, Den;          // объектные поля типа Integral
    public:
        Rational(int num, int) : Num(num){};
        void  Out()
        {
                cout << Num.Get() << " / " << Den.Get() << endl;
        }
};
// Объявление объектов типа Rational  с объектными полями
Rational Num(13,64);
```

Список инициализации конструктора класса Rational трактуется так, как если бы он был записан в виде Num(num), Den().

***Задание №6.***

Определить класс комплексные числа Complex, объектными полями которого являются поля Re и Im типа Float (вещественный тип), объявленный ранее. Определить конструктор со списком инициализации, обеспечивающим

инициализацию объявленных полей. Проанализируйте реакцию компилятора при попытке инициализации полей другим способом.

***Копирующие конструкторы***.

Среди конструкторов данного класса особую роль выполняют копирующие конструкторы. Предположим, что имя класса – Name, тогда конструктор вида Name::Name(const Name &), есть копирующий конструктор.

Если программист не определяет подобный конструктор явно, компилятор формирует его автоматически. Использование предполагаемого копирующего конструктора приводит к инициализации методом «поле – за - полем» всех полей инициализируемого объекта класса.

Пример использования предполагаемого конструктора.

```
class Test
{
	int test;
public:
	Test(int t)
	{
		test = t;
	}
	void Out()
	{
		cout << " Test: " << test << endl;
	}
};

int main()
```

```
{
        Test tst_1(100), tst_2 = tst_1;
        tst_1.Out();
        tst_2.Out();
        return 0;
}
```

Как видно, в классе Test не определен копирующий конструктор в явном виде, однако, есть возможность инициализировать новый объект tst_2 инициализируется полями уже существующего объекта tst_1. Это возможно потому, что компилятор создал свою версию конструктора по методу «поле – за – полем»:

```
Test(const Test &t)
{
        Test = t.test;
```

}, или с использованием списка инициализации:

```
Test(const Test &t) : test(t.test){};
```

Наберите этот пример, проанализируйте результаты работы программы.

***Задание №7.***

Для класса, оперделенного в задании №6, определить копирующий конструктор, инициализирующий все поля объявляемого объекта.

Если в классе определено несколько полей, их нициализация производится последовательно. Рассмотрим пример класса Student, имеющий три поля различного типа.

```
class Student
{
        char *Name;
        int Age;
        double Ball;
```

```
        // остальные поля
public:
        //
        Student(const Student &);
        // остальные методы
};
Student :: Student(const Student &st)
{
        Name = st.Name;
        Age = st.Age;
        Ball = st.Ball;
}
```

Попытайтесь самостоятельно определить аналогичный конструктор, но со списком инициализации.

Копирующие конструкторы вызываются автоматически в случае если объявляемый объект инициализируется уже существующим, а так же в случае передачи параметра типа класс в какую-либо функцию или же функция возвращает объект типа класс. Для класса Student определим внешнюю функцию, в качестве ее парамеира передадим ссылку на объект типа Student.

```
class Student
{
 public:
        char *Name;
        int Age;
        double Ball;
        // остальные поля
        Student(const Student &);
        // остальные методы
};
Student :: Student(const Student &st)
{
```

```
        Name = st.Name;
        Age = st.Age;
        Ball = st.Ball;
}
void ExtFunc(Student st)
{
        cout << " Name: " << st.Name << ' '
             << " Age: " << st.Age << ' '
             << " Ball: " << st.Ball << endl;
}
```

Заметьте, что все поля класса объявлены как открытые. Это связано с тем, что обращение вида st.Name недопустимо если поле объявлено под спецификатором private. Прблема решается открытием полей или же объявлением методов, возвращающих значения закрытых полей класса.

***Задание №8.***

Для класса, определенного в задании №6, определит внешнюю функцию, возвращающую в качестве результата объект типа Complex. В копирующем конструкторе предусмотреть вывод «Работает копирующий конструктор», который позволит убедится в в каких случаях вызывается данный вид конструктора.

***Преобразования, задаваемые конструктором***.

Поскольку каждый конструктор задает преобразование в своем классе, следовательно, если в некоторм месте программы используется выражение Exp, представляющее данное, которое должно быть преобразовано к типу класса, то сразу же после обработки этого выражения неявно активизируется конструктор с инициализатором (Exp), что выполняет необходимые

преобразования. Конструктором преобразования считается любой конструктор, имеющий один параметр, тип которого отличен от типа определяемого класса. Следующий пример показывает использование конструктора преобразования.

```
class Test
{
	int test;
public:
	Test(){};
	Test(int t)
	{
		cout << " Конструктор преобразования" <<
endl;
		test = t;
	}
	Test(const Test &t):test(t.test)
	{
		cout << " Копирующий конструктор " <<
endl;
	}

	void Out()
	{

		cout << " Test: " << test << endl;
	}
};


int main()
{
	setlocale(0,"RUS");
```

```
        Test tst;
        tst = 300;        // или tst = Test(300);
        tst.Out();
        return 0;
}
```

Преобразующий конструктор класса Test:

```
        Test(int t)
        {
                cout << " Конструктор преобразования" <<
endl;
                test = t;
        }
```

осуществляет преобразование параметра типа int к типу Test и оператор tst = 300; в теле основной функции трактуется как оператор tst = Test(300);. Такое преобразование называется типичным.

В некотрых случаях преобразования, задаваемые конструкторами можно запретить, так как они могут привести к нежелательным результатам. Для этого перед именем конструктора записывается стандартное слово explicit, которе предупреждает компилятор о том, что при объявлении некоторго объекта и инициализации его значением не относящимся к типу определяемого класса, неявных преоборазований производить не следует.

Рассмотрим еще раз класс Test

```
class Test
{
public:
        int test;
public:
        Test(){};
        explicit Test(int t)
```

```
        {
                cout << " Конструктор преобразования,
запрещающий
                        неявные преобразования типа int  в
                тип Test" << endl;
                test = t;
        }
        Test(const Test &t):test(t.test)
        {
                cout << " Копирующий конструктор " <<
endl;
        }

        void Out()
        {

                cout << " Test: " << test << endl;
        }
};


int main()
{
        setlocale(0,"RUS");
        Test tst_1(38);
        Test tst_2 = tst_1;
        Test tst_3 = 64;          // Здесь ошибка!
        return 0;
}
```

Откомпилируйте этот фрагмент и посмотрите сообщения компилятора. В операторе Test tst_3 = 64; компилятор не сможет преобразовать тип int к типу Test. Ошибку можно «исправить», если использовать явное преобразование типа Test tst_3 = (Test)64; , если это

необходимо. Этот пример не единственный, более того, он очень прямолинеен. Больших неприятностей можно ожидать в случае передачи некоторой внешней функции параметра типа определяемого класса. Например,

```
void FunOut(Test t)
{
        cout << " Test: ";
        t.Out();
}
```

Если вызвать эту функцию с объектом типа Test, например, FunOut(tst_1),никаких проблем не возникнет. Еси же вызвать с переменной типа int, например, FunOut(300);, то неявно вызывается конструктор преобразования, котрый приводит число 300 к типу Test. Если перед конструктором стоит спецификатор explicit, неприятностей преобразования не будет, так как компилятор выдаст соответствующее сообщение.

***Задание №9.***

Для класса комплексных чисел добавить конструктор с одним аргументом, инициализирующий действительную или мнимую часть комплексного числа, запрещающий преобразования.

***Переменная this.***

Возможность неквалфицированного обращения в теле составной функции (метода) класса к компонентам класса обуславливается следующими утверждениями:

- в теле каждой нестатической составной функции класса с именем Name доступна переменная this типа Name с присвоенным ей указателем того объекта класса Name, применительно к которому вызвана данная составная функция;

- каждое неквалифицированное обращение к нестатическому компоненту класса Nane, например Field, трактуется как сокращенная запись выражения this->Field.

Рассмотрим модифицированный класс Student, используя переменную this.

```
class Student
{
 private:
        char *Name;
        int Age;
        double Ball;
public:
        Student(char *, int, double);
        void Out();
};

Student :: Student(char *Name, int Age, double Ball)
{
        this->Student::Name = Name;
        this->Student::Age = Age;
        this->Student::Ball = Ball;
}
void Student::Out()
{
        cout << " Name:" << this->Student::Name
              << " Age:" << this->Student::Age
             << "Ball:" << this->Student::Ball << endl;
}
```

Объявления вида this->Student::Name = Name; интерпретируются легко: this- указатель на объект, применительно которого вызывается данный конструктор, Student::Name – опреатор разрешения области видимости

компоненты Name класса, последнее слово Name – это имя аргумента конструктора.

Этот пример искусственно усложнен, его более читаемый вид:

```
Student :: Student(char *name, int age, double ball)
{
        Name = name;
        Age = age;
        Ball = ball;
}
```

Использование переменной this здесь так же не обязательно. Необходимость в ней возникает в случае если на момент написания кода программы фактический объект -получатель данного метода не известен, например, при перегрузке операций.

***Задание №10.***

Для класса комплексные числа определить методы, использующие переменную this.

***Дружественные функции и классы.***

Функции, объявленные в классе, делятся на составные и дружественные. Каждая нестатическая функция, объявленная со спецификатором friend, является дружественной классу. Такая функция имеет доступ ко всем компонентам класса, к которым имеют доступ составные функции класса. Рассмотрим пример.

```
class Fixed
{
        Int Fix;
public:
        Fixed(int val):Fix(val){};
        friend void Show(Fixed);
};
```

```
void Show(Fixed Arg)
{
        cout << Arg.Fix;
}

 int main()
{
        Fixed Num(100);
        Show(Num);
        return 0;
}
```

Поскольку функция Show дружественная для класса, в ней можно использовать идентификатор приватного поля этого класса.

Использовать дружественные функции следует осторожно, так как они нарушают инкапсуляцию полей класса.

Составная функция некоторого класса может быть дружественной по отношению к другому классу. Это достигается путем передачи составной функции параметра типа класс, по отношению к которому она должна быть дружественной.

***Задание №11.***

Определить два простых класса, объявив составную функцию одного из них, дружественной для другого. Например,

```
class Test2;                    // прототип класса Test2
class Test1
{//
public:
        void fun(Test2);
};
class Test2
```

```
{//
public:
        friend void Test1::fun(Test2);
}
```

Все составные функции одного класса можно сделать дружественными по отношению ко второму, если в описании второго объявить дружественный класс. Например, для задания №11 класс Test2 определить дружественным классом по отношению к Test1 по следующей схеме:

```
class Test2;
class Test1
{//
public:
        friend class Test2;
        //
};
class Test2
{
        Test1 tst1;
public:
        fun_1();
        fun_2();
};
```

Дружественные функции используются для придания объектам класса свойств не характерных их типу, например, ввод из стандартного потока и вывод в стандартный поток. Типовой пример вывода показан ниже.

```
#include<ostream>
class Name
{
        int name;
```

```
public:
        Name(int val)
{
        name = val;
}
//
friend ostream &operator <<(ostream &, const Name &);
};
ostream &operator <<(ostream &out, const Name &n)
{
        out << " Message: " << n.name << endl;
}
```

***Задание №12.***

Для задания №11 класс Test1 дополнить дружественными функциями – операциями ввода и вывода.

***Перегрузка операций.***

Функцией-операцией является каждая функция с именем operator #, где # представляет одну из следующих операций: new, delete, +, - , *, /, %, &, !, = , <, >, !=, (), [], -> и так далее.

Функции-операции могут быть одноместные (унарные) и двуместные (бинарные), при этом число операндов, приоритет и ассоциативность определенной функции, такие же, как в базовом языке.

Не разрешается перегружать следующие операции: *, sizeof, #, ##, ::, ?:.

Необходимо, что специальные операции, а именно =, [], () были нестатическими составными функциями класса. Среди них особого внимания заслуживает операция присваивания, так как она не подлежит наследованию.

Если пользователь не переопределяет самостоятельно операцию присваивания, то неявно создается предполагаемое присваивание

Class &Class::operator =(const Class &),

реализующее присваивание методом «поле – за - полем».

Функции-операции можно определять тремя способами: как составную функцию класса, как внешнюю функцию или как дружественную. В двух последних случаях функции передается хотя бы один параметр типа класс, указатель или ссылка на класс.

***Перегрузка унарных операций.***

Унарная функция-операция, определяемая внутри класса, должна быть представлена с помощью нестатического метода без параметров. Рассмотрим пример простого класса.

```
class Test
{
        int test;
public:
        Test(){};
        Test(int tst)
        {
                test = tst;
        }
        Test &operator ++()
        {
                ++test;
                return *this;
        }
        void Out()
        {
                cout << " Test: " << test << endl;
```

```
        }
};

int main()
{
        Test tst(200);
        ++tst;
        tst.Out();
        return 0;
}
```

***Задание №13.***

Переопределите самостоятельно для данного класса унарную префиксную операцию как дружественную и как внешнюю.

Постфиксную унарную операцию отличают от префиксной путем передачи пустого параметра типа int, причем, если операция определяется как внешняя, этот параметр должен быть вторым. Для класса Test:

```
Test operator ++(Test &tst, int)
{
        tst.test++;
        return tst;
}
```

Обратите внимание на то, что для работы такой внешней функции, необходимо открыть доступ к полю test.

В продолжение задания №13, определите унарную постфиксную операцию как дружественную классу Test.

***Перегрузка бинарных операций.***

Если бинарная операция определяется как нестатическая составная функция класса, то она должна иметь один параметр, как правило, константная ссылка на объект определяемого класса. Этот параметр представляет

собой правый операнд, а вызвавший объект считается левым операндом операции.

Для класса Test перегрузка операции сложения может быть представлена следующим образом:

```
Test Test:: operator+(const Test &tst)
{
        return this->test+tst.test;
}
```

Определить самостоятельно перегрузку бинарных арифметических операций как внешнюю и дружественную функцию, с учетом того, что передавть им необходимо два параметра типа класс.

Бинарные арифметические операции всегда должны возвращать в качестве результата объект типа класс. Операции сравнения возвращают в качестве результата объект типа bool. Дополните класс Test операциями сравнения.

Важное место в определении класса является перегрузка операции присваивания. При определении данной операции следует руководствоваться следущими рассуждениями:

-эту операцию имеет смысл переопределять в том случае, если класс содержит поля, память под которые выделяется динамически;

-если программист не оперделял операцию самостоятельно, компилятор автоматически сгенерирует ее код как поэлементное копирование полей одного объекта в другой;

-операция не наследуется в производных классах;

-операцию можно определить только как составную функцию класса.

Разберите следующий пример, обратив особое внимание на операцию присваивания.

```
class Text
```

```
{
        int Len;
        char *Ref;
public:
        Text()
        {
                Len = 0;
                Ref = new char[Len];
        };
        Text(char *Str)
        {
                Ref = new char[Len=strlen(Str)+1];
                strcpy_s(Ref,Len,Str);
        }
        ~Text()
        {
                delete Ref;
        }
        Text &operator =(const Text &);
        void Out()
        {
                cout << " Out text = "  << Ref << endl;
        }
};

Text &Text::operator =(const Text &Par)
{
        if(this == &Par) return *this;
        else
        {
                delete Ref;
                Ref = new char[strlen(Par.Ref)+1];
                strcpy_s(Ref,Len-1,Par.Ref);
        }
```

```
        return *this;
}

int main()
{
        Text head("Text Text"), tail;
        tail = head;
        tail.Out();
        return 0;
}
```

Операция strcpy_s, спецефичная для стандарта языка С, в последних версиях VC может работать не корректно.

***Задание №14.***

Определите собственный класс, содержащий поля-указатели на стандартные типы, требующие выделения места в динамической области памяти. Переопределить операцию присваивания для данного класса.

***Перегрузка операции индексирования***.

Операция индексирования [] так же специфична и ее перегружают в тех случаях, когда тип класса представляет собой некое множество, для которого индексирование имеет определенный смысл. Например, в классе, моделирующем работу массива. Необходимо учитывать то, что эта операция возвращает в качестве результата объект того типа, который содержится во множестве.

Рассмотрим класс Vector, описывающий модель стандартного массива.

```
#include<iostream>
```

```
#include<stdlib.h>
using namespace std;

class Vector
{
protected:
	int size;
	int *p;
public:
	explicit Vector(int n=5);
	Vector(const int arr[], int n):size(n)
	{
		p = new int[size];
		for(int i=0; i<size; i++)
			p[i] = arr[i];
	}
	~Vector()
	{
		delete []p;
	}
	int &operator [](int);
	void Out()
	{
		for(int i=0; i<size; i++)
			cout << p[i] << ' ';
		cout << endl;
	}
};
int &Vector::operator [](int i)
{
	if(i<0 || i>size)
	{
		cout << " Выход за границы массива! "
<< endl;
```

```
                exit(0);
        }
        return p[i];
}


int main()
{
        setlocale(0,"RUS");
        int array_int[10]={23,-45,6,0,435,-6};
        Vector vector(array_int,10);
        vector.Out();
        cout << vector[1] << endl;
        cout << vector[11] << endl;

        return 0;
}
```

В этом примере показана модель стандартного массива, котрая не в полной мере воспроизводит реальный массив. Например, объявление объекта класса Vector, по аналогии с обычными массивами Vector vector[10]={1,2,3,4,5}; , в данном случае приведет к ошибке.

***Задание №15.***

Определите класс однонаправленный список, к элементам которого можно обращаться по индексу.

***Перегрузка операции доступа к компонентам класса ->.***

Эта операция используется для доступа к полям и методам класса. Обращение x -> m трактуется как (x.operator())->m, причем x – это объект, а не указатель на него. Операция возврвщает указатель на объект , либо ссылку на объект

класса, поскольку оригинальное значение операции -> не теряется, а только задерживается. Перегрузка операции -> допускается только через нестатическую составную функцию класса.

Рассмотрим подробнее определение этого оператора.

```
struct Test
{
        int test;
        Test(){};
        Test(int t)
        { test = t;}
        Test *operator ->()
        {
                cout << " Собственный оператор -> "
<< endl;
                return this;
        }
        void Out()
        {  cout << " Test: " << test << endl;  }

};

int main()
{
        int int_obj = 100;
        Test tst(int_obj);
        cout << tst->test << endl;      // или cout <<
(tst.operator->())->test << endl;
        tst->Out();                     // или
(tst.operator->())->Out();
        return 0;
}
```

Ипользование ключевого слова struct вместо class связано с тем, что обращение к скрытым полям (public:)в классе не допускается и доступ ptr_tst->test приведет к ошибке.

Следует отметить, что этот оператор может быть перегружен только как нестатический метод класса.

Обратите внимание на следующий момент, если объявить указатель на объект класса и вызвать операцию доступа, работать будет стандартная операция. Оцените это на следующем примере:

```
Test *ptr_Test = new Test(int_obj);
cout << ptr_Test->test << endl;
```

***Задание №16.***

Для класса комплекных чисел переопределить оператор доступа, возвращающий значения действительной и мнимой части комплексного числа.

Для любопытных: перегрузку операции ->* - выделение указателя на компоненту можно посмотреть в Microsoft Developer Network (MSDN).

***Перегрузка операций new и delete.***

Перегрузку операций new и delete выполняют для обеспечения альтернативности управления динамической памятью. Эти функции должны соответствовать требованиям описаным в международном стандарте ISO/IEC 14882. К особенностям операций следует отнести:

1. -им не требуется передавть параметр типа класс;
   - первым параметром функции new должен передаваться размер объекта типа size_t (тип

определен в заголовочном файле stddef.h), который при вызове передается неявно;
- операция new возвращает *void;
- операция delete возвращает результат типа void;
- опрерация delete имеет первый параметр типа *void;
- операции new и delete, определенные как составные функции класса, по умолчанию считаются статическими функциями.

Перегружать можно как глобальные операции new и delete, так и для определяемых типов данных. Рассмотрим типовой пример перегрузки глобальных операций.

```
#include<iostream>
#include<stddef.h>
#include<malloc.h>

using namespace std;

void* operator new(size_t sz)
{
        cout << "Оператор new " << " Байт: " << sz << endl;
        void* m = malloc(sz);
        if(!m) cout << " Нехватка памяти " << endl;
        return m;
}
void operator delete(void* m)
{
         cout << "Оператор delete" << endl;
         free(m);
}
```

```
int main()
{
	setlocale(0,"RUS");

	char *ptr_char =  new char;
	int *ptr_int = new int;
	double *ptr_double = new double;
	delete ptr_char;
	delete ptr_int;
	delete ptr_double;

	return 0;
}
```

Протестируйте данный пример, оцените результаты. Следующий пример показывает перегрузку операций внутри класса.

```
#include<iostream>
#include<stddef.h>
#include<malloc.h>

using namespace std;

class Test
{
	int test;
public:
	Test(){};
	Test(int tst):test(tst)    {};
	void *operator new(size_t);
	void operator delete(void*);
	void Out()
	{
		cout << " Test: " << test << endl;
```

```
        }

};

void* Test::operator new(size_t size)
{
        cout << " New операция для класса Test,
необходимое число байт: " << size          << endl;
   void *storage = malloc(size);
   if(NULL == storage)
        {
                cout << " Нехватка памяти " << endl;
        }
        return storage;
}


void Test::operator delete(void *p)
{
        cout << " Delete операция для класса Test " <<
endl;
        free(p);
}

int main()
{
        setlocale(0,"RUS");
        // Вызов собственной операции new
        Test *ptr_Test_1 = new Test(100);
        ptr_Test_1->Out();
        // Вызов глобальной операции new
        Test *ptr_Test_2 = ::new Test(200);
        ptr_Test_2->Out();
        // Вызов собственной операции delete
```

```
        delete ptr_Test_1;
        // Вызов глобальной операции delete
        ::delete ptr_Test_2;

        return 0;
}
```

Операции new и delete могут иметь другие форматы списка параметров, описанные в стандарте ISO/IEC 14882.

***Задание №17.***

Для класса комплексных чисел переопределить свои собственные операции new и delete.

***Указатели на компоненты класса.***

На любую нестатическую компонента класса можно содать указатель. Этот указатель отличаетстся от обычного наличием спецификатором типа класс. В частности, указатель на тип int объявляется следующим образом: int *, а указатель на поле целого типа класса Class – int Class::*.

Примеры:

| | |
|---|---|
| int Fix; | int Class:: * |
| float *Num; | float Class::* |
| long (*Ref)[2]; | long (*Class::*)[2] |
| int Sub(int); | int (Class::*)(int) |

Благодаря связи компонента класса с идентификатором класса возможно осуществление контроля правильности обращения к компонентам класса через указатели.

С точки зрения синтаксиса обращений применяется простая нотация: если Ptr – выражение, указывающее на компоненту класса, то *Ptr – имя этого компонента. Это приводи к тому, что если Obj имя объекта, а Ref –

указатель на объект класса, справедливы следующие записи:

Obj.*Ptr; и
Ref->*Ptr;

Рассмотрим пример указателей на поля класса.

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

class Test
{
public:
	int Tst;

};
// Указатель на поле целого типа класса Test
int Test::*ptr_Test = &Test::Tst;
// Похоже на объявление обычного указателя int
*Ptr_Test = &Tst; , только
// присутствует имя класса Test::
int main()
{
	setlocale(0,"RUS");

	// Объект класс Test
	Test Num;
	Num.Tst = 63;
	cout << " Обычным способом через объект
класса: " << Num.Tst << endl;
	cout << " Через указатель на компоненту: "
<< Num.*ptr_Test << endl;

	// Указатель на объект типа Test
```

```
        Test *Ptr_Test = &Num;
        Ptr_Test->Tst = 38;
        cout << " Обычным способои через указатель
на объект класса: " << Ptr_Test->Tst << endl;
        cout << " Через указатель на компоненту: "
<< Ptr_Test->*ptr_Test << endl;

        return 0;
}
```

Внимательно изучите этот пример, возможно потребуется вспомнить указатели на стандартные типы данных. Создать указатель на закрытое (защищенное) поле класса можно, но обратиться аналогичным способом не разарешит компилятор.

А теперь разберемся с указателями на методы класса.

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

class Test
{
        int Tst;
public:
        Test(){};
        Test(int tst):Tst(tst){};
        int *TestPtr()
        {
                return &Tst;
        }
        int TestSumm(int arg)
        {
                Tst += arg;
```

```
                return Tst;
        }
};

// Указатель на метод TestSumma класса Test
int (Test::*Ptr_TestSumm)(int) = &Test::TestSumm;
// Указатель на метод *TestPtr() класса Test
int *(Test::*Ptr_TestPtr)() = &Test::TestPtr;

int main()
{
        setlocale(0,"RUS");
        // Для метода TestSumm(int);
        Test Num(100);
        // Через объект класса
        cout << (Num.*Ptr_TestSumm)(30) << endl;
        // Через указатель
        Test *ptr_Test = &Num;
        cout << (ptr_Test->*Ptr_TestSumm)(50) <<
endl << endl;

        // Для метода *TestPtr()
        // Через объект класса
        cout << *(Num.*Ptr_TestPtr)() << endl;
        // Через указатель
        cout << *(ptr_Test->*Ptr_TestPtr)() << endl;
        return 0;
}
```

Нельзя создать указатель на конструктор класса, следущее выражение компилятор посчитает ошибочным: (Test::*PtrTest)(int) = &Test::Test(int);. Нельзя так же создать указатели на статические компоненты класса.

***Задние №18.***

Для класса компексные числа определит указатели на поля и методы класса. Показать их использование.

## Лабораторная работа №7. Наследование. Полиморфизм.

***Общие сведения.***

Наиболее значимой особенностью ООП является наследование. Это процесс создания новых классов, называемых наследниками или производными, из уже существующих или базовых классов. Наследование заключается в приме в производном классе компонент базового класса. Кроме того, производный класс дополняется своими собственными компонентами.

Объявление производного класса представляется следующим образом:

```
class Имя_класса : [private|protected|public] Имя_базового_класса
{
        Тело класса;
};
```

По определению считается, что все компоненты базового класса являются компонентами производного, за исключением конструкторов, деструкторов и операции присваивания (=). Предполагается, что первое поле производного класса расположено после всех полей, наследованных им от базового класса.

Если базовых полей несколько, в заголовке производного класса они должны быть перечислены все через запятую, каждый со своим ключом доступа.

Очередность активизации конструкторов следующая:

- конструктор базового класса;

- конструктор подобъектов;
- конструктор производного класса.

Соответствующие деструкторы активизируются в обратом порядке.

***Ключи доступа.***

Перед именем базового класса ставят ключи доступа: private (для классов по умолчанию), protected или public (для структур по умолчанию). Слово private указывает на приватный базовый класс, а public – на обобществленный.

Не следует путать эти ключевые слова с одноименными словами в теле класса, описывающими уровень доступа к компонентам класса. Рассмотрим и разберем смысл этих слов на примерах.

*Пример 1. Производный класс с обобществленным базовым классом.*

```
class Base
{
        int base;
public:
        Base(){};
        Base(int b)
        {
                base = b;
        }
        void Out()
        {
                cout << " Base: " << base << endl;
        }
};
```

```
class Derived :public Base          // производный класс с обобществленным
                                    // базовым классом Base
{
        int derived;
public:
        Derived(){};
        Derived(int b, int d):Base(b)
        {
                derived = d;
        }
        void Out()
        {
                cout << " Derived: " << derived << endl;
        }
};
```

В этом случае все компоненты производного класса имеют доступ к элементам базового класса в соответствии с их ключами доступа. Например, следующее обращение в методе производного класса приведет к ошибке:

```
        void Out()
                {
                        cout << " Base: " <<  base << endl;
                        cout << " Derived: " << derived << endl;
                }
```

visual studio 2010\projects\inher_1\inher_1\inher_1.cpp(31): error C2248: base: невозможно обратиться к private члену, объявленному в классе "Base".

Прямой доступ к закрытым компонентам базового класса не допустим. Ситуация изменится, если обратиться через метод базового класса или разместить поле base в области protected базового класса. Эта область, подобно области private, не допускает обращение к полю извне, но делает поле доступным для всех компонентов производного класса. В этом и состоит отличие слов private и protected.

*Пример 2. Производный класс с защищенным базовым классом.*

```
class Derived : protected Base//производный класс с
защищенным базовым классом Base
{
        int derived;
public:
        Derived(){};
        Derived(int b, int d):Base(b)
        {
                derived = d;
        }
        void Out()
        {
                cout << " Base: " << base << endl;
                //  если поле base в области protected
                // или Base::Out(); если поле base в
области private
                cout << " Derived: " << derived << endl;
        }
};
```

В этом случае поле private базового класса остается таким же, а поля protected и public, понимаются как

protected, то есть защищенными от внешнего воздействия, но доступными из производного класса.

*Пример 3. Производный класс с приватным базовым классом.*

Если в заголовке базовый класс описывается с ключевым словом private или спецификатор отсутствует, считается, что базовый класс приватный. В этом случае все поля базового класса без исключения считаются по умолчанию private.

```
class Derived : private Base
        //производный класс с закрытым
(приватным) базовым классом Base
{
        int derived;
public:
        Derived(){};
        Derived(int b, int d):Base(b)
        {
                derived = d;
        }
        void Out()
        {
                cout << " Base: " << base << endl; //
```

если пол

e base в области protected

```
            Base::Out();
            cout << " Derived: " << derived << endl;
        }
};
```

Обратиться к отдельным компонентам можно средствами оператора разрешения области видимости,

например, Base::Out();. Обращение к области private по-прежнему допустимо только через методы базового класса.

***Конструктор производного класса.***

При определении конструктора производного класса необходимо обеспечить инициализацию полей базового класса. Осуществляется это путем использования конструктора со списком инициализации. Другим способом передать параметр и инициализировать базовый класс нельзя. Рассмотрим еще раз конструктор класса Derived:

```
Derived(int b, int d):Base(b)
{
        derived = d;
}
```

Здесь в списке инициализации используется выражение :Base(b), обеспечивающее инициализацию поля базового класса. Если в базовом классе есть конструктор требующий несколько параметров, в конструкторе производного класса необходимо обеспечить их инициализацию через список инициализации. Попытка передачи параметров в теле конструктора, например, Base = b; , приведет к ошибке.

***Задание №1.***

Объявить базовый и производный классы, моделирующие слово (базовый класс) и строку (производный класс). В базовом классе предусмотреть поле для хранения одного слова произвольной длины (можно использовать тип string), а в производном, кроме наследованного слова, необходимо объявить поле, содержащее количество слов в строке. Для классов определить методы, обеспечивающие ввод слов и строк и

вывод их на экран. Проанализировать случаи приватного, защищенного и обобществленного наследования.

***Задание №2.***

В конструкторах базового и производного классов обеспечить вывод в стандартный поток сообщения, идентифицирующего принадлежность объекта тому или иному классу. Проанализировать последовательность активизации конструкторов при объявлении объектов производного класса. Дополнить классы деструкторами, оценить порядок их активации.

***Простое наследование***.

Наследование называется простым, если у производного класса один базовый класс. Рассмотрим примеры наследования составных частей базового класса при простом наследовании.

*Наследование статических компонентов базового класса.*

```
class Base
{
protected:
        int base;
public:
        Base(){};
        Base(int b)
        {
                base = b;
        }
        static int st_base;
        static  int Get()
        {
                return st_base;
        }
```

```
        friend ostream &operator <<(ostream &, const
Base &);
};

int Base::st_base = 64;

ostream &operator <<(ostream &out, const Base &b)
{
        out << b.base << endl;
        return out;
}
class Derived : public Base
{
        int derived;
public:
        Derived(){};
        Derived(int b, int d):Base(b)
        {
                derived = d;
                // можно и здесь  st_base = 33;
        }
        /*

        static int Get()
        {
                return st_base;
                // можно переопределить, а можно
наследовать из базового
        }
        */
        static void SetStat()
        {
                st_base = 33;
        }
```

```
};
```

В теле основной функции наберите следующий код и проанализируйте результаты.

```
Base bas(2000);
Derived der(33,64);
cout << " Статическое поле базового класса: " <<
bas.Get() << endl;
der.SetStat();
cout << " Статическое поле базового класса: " <<
der.Get() << endl;
```

Доступ к статическому полю базового класса возможен непосредственно в методе производного класса, если оно объявлено в открытой области или через метод базового класса, если объявлено в закрытой. Наследование статических переменных имеет практический смысл, так как в иерархии возможны элементы, доступные для любого объекта данной иерархии. Жизненным примером может служить наследование потомками фамилии родителей (предков). Необходимо заметить, что объект производного класса может не только воспользоваться значением статического поля, но и изменить его при необходимости. В приведенном примере в производном классе описан метод SetStat(), позволяющий менять статическое поле.

Метод static int Get() в производном классе переопределен, но закомментирован, в этом случае будет вызываться метод базового класса. Для вызова своего метода, с него необходимо снять комментарии.

***Задание №3.***

Дополните объявление класса слово статическим полем, приведите пример его использования объектами и методами производного класса строка.

Самостоятельно разобрать и привести пример наследования константных и ссылочных полей.

*Наследование дружественных функций.*

Рассмотрим возможность использования функции дружественной базовому классу объектами производного класса. Это допустимо, поскольку объект производного класса по умолчанию считается объектом базового класса. Необходимо заметить, что функция дружественная базовому классу, имеющая неограниченный доступ к его полям, не имеет доступа к полям производного класса. Для устранения этого недостатка необходимо переопределить аналогичную функцию, дружественную производному классу, которая будет дружественной и базовому классу. В качестве примера рассмотрим дружественную функцию-операцию вывода в стандартный поток.

```
class Base
{
protected:
        int base;
public:
        Base(){};
        Base(int b)
        {
                base = b;
        }
```

```
        friend ostream &operator <<(ostream &, const
Base &);
};

ostream &operator <<(ostream &out, const Base &b)
{
        out << "  Функция-операция дружественная
классу Base: " << b.base  << endl;
        return out;
}

class Derived : public Base
{
        int derived;
public:
        Derived(){};
        Derived(int b, int d):Base(b)
        {
                derived = d;
        }
};

int main()
{
        setlocale(0,"RUS");
        Base bas(2000);
        Derived der(33,64);
        cout << bas;
        cout << der;
        system("pause");
        return 0;
}
```

В этом примере объект производного класса пользуется услугами операций вывода: cout << der;,

объявленной в базовом классе. Разберите этот пример и проанализируйте результаты.

***Задание №4.***

В классе слово определить дружественную функцию, выводящую отдельное слово в стандартный поток. Перегрузите аналогичную функцию в классе строка. Приведите примеры использования дружественных функций.

***Переменная this в условиях наследования.***

У каждой нестатической составной функции класса есть доступ к переменной this, указывающую на ту часть объекта, которая является производной этого класса. Рассмотрим следующий пример иерархии классов.

```
class One
{
        int one_1, one_2;
public:
        One(int o_1, int o_2):one_1(o_1),
one_2(o_2){};
        int Get()
        {
                cout << " Адрес объекта: " << this << '
';
                return one_1;
        }
};

class Two_1 :public One
```

```
{
        long two_1;
public:
        Two_1(int o_1, int o_2):One(o_1, o_2){};
};

class Two_2 :public One
{
        char two_2;
public:
        Two_2(int o_1, int o_2):One(o_1, o_2){};
};

class All :public Two_1, public Two_2
{
public:
        All(int o_1, int o_2, int o_3, int o_4):
          Two_1(o_1, o_2), Two_2(o_3, o_4){};
};

All  Var(1,2,3,4);
int main()
{
        setlocale(0,"RUS");
        cout << " Значение поля: " <<
Var.Two_1::Get() << endl;
        cout << " Значение поля: " <<
Var.Two_2::Get() << endl;
        system("pause");
        return 0;
}
```

В каждом из двух обращений переменная this указывает на различные части объекта Var. Выполнение

программы покажет различные адреса и значения частей объекта.

***Задание №5. ****

Создать иерархию классов, показать пример наследования переменной this объектами производных классов.

***Преобразование указателей.***

Важной особенностью производного класса с обобществленным базовым классом является возможность преобразования указателей объекта этого класса в указатель объекта базового класса.

Разрешается также преобразование ссылки на производный класс в ссылку на объект базового класса, в том числе связывание параметра ссылки на базовый класс с выражением, идентифицирующим объект производного класса.

Оба преобразования относятся к стандартным и не требуют явного использования операции преобразования.

Допускается также присвоение объекту базового класса объекта его производного класса.

*Пример 1. Преобразование указателя объекта производного класса на указатель объекта базового класса.*

```
class Base
{
protected:
        int base;
public:
        Base(){};
        Base(int b)
        {
```

```
                base = b;
        }
        int Out()
        {
                return base;
        }
};

class Derived : public Base
{
        int derived;
public:
        Derived(){};
        Derived(int b, int d):Base(b)
        {
                derived = d;
        }


};

Derived der(33,64);
int main()
{
        setlocale(0,"RUS");
        Base *ptr_Base = (Base *)&der;
        cout << ptr_Base->Out() << endl;
        system("pause");
        return 0;
}
```

В этой программе объявление Base *ptr_Base = &der; равносильное Base *ptr_Base = (Base *)&der;, относится к стандартным и явного преобразования можно

не делать. В случае, если базовый класс описывается с ключом private или protected, преобразования нужно задавать явно.

*Пример 2. Присвоение объекту базового класса объекта производного класса.*

```
class Base
{
protected:
        int base;
public:
        Base(){};
        Base(int b)
        {
                base = b;
        }
        int Out()
        {
                return base;
        }
};

class Derived :  public Base
{
        int derived;
public:
        Derived(){};
        Derived(int b, int d):Base(b)
        {
                derived = d;
        }
```

```
};

Derived der(33,64);
int main()
{
	setlocale(0,"RUS");
	Base bas;
	bas = *(Base *)&der;
	cout << bas.Out() << endl;
	system("pause");
	return 0;
}
```

В этом примере оператор bas = *(Base *)&der; можно заменить его эквивалентом bas = &der;, так как подобное преобразование компилятор выполнит автоматически. Проверьте работоспособность примера. Замените ключ доступа в объявлении производного класса и вновь проверьте. Проанализируйте сообщения компилятора.

***Виртуальные функции.***

Виртуальной (полиморфной) функцией является каждая нестатическая составная функция класса, объявленная со спецификатором virtual, а также каждая функция такого же типа, содержащаяся в произвольной последовательности производных классов, происходящих от этого класса. Использование в объявлении класса хотя бы одной полиморфной функции приводит к расширению полей класса на скрытое идентификационное поле, в котором хранится информация о типе объекта данного класса. Полиморфным может быть деструктор, несмотря на то, что в разных классах у него разные имена.

При обработке виртуальных методов компилятор формирует таблицу виртуальных методов (vtbl), в которой для каждого виртуального метода записывается его адрес в памяти. Каждый объект данной иерархии содержит дополнительной скрытое поле ссылки на vtbl, называемое vptr. Это поле заполняется конструктором класса при создании объекта. При компиляции программы ссылки на виртуальные методы замещаются на обращения к таблице виртуальных методов (vtbl) через указатель (vptr) объекта. На этапе выполнения в момент обращения к методу его адрес выбирается из таблицы. Таким образом, вызов виртуального метода, в отличие от обычных методов, выполняется через дополнительный этап получения адреса функции из таблицы. Описанный механизм носит название механизма позднего или динамического связывания. Через виртуальные методы в C++ реализуется полиморфизм.

Рассмотрим пример полиморфной функции.

```
class First
{
        int first;
public:
        First(){};
        First(int f):first(f){};
        virtual void Out()
        {
                cout << " First: " << first << endl;
        }
};

class Second :public First
{
        int second;
public:
```

```
        Second(){};
        Second(int f, int s):First(f)
                { second = s;}
        void Out()
        { cout << " Second: " << second << endl; }
};

class Third :public Second
{
        int third;
public:
        Third(int f, int s, int t): Second(f,s)
        {
                third = t;
        }
        void Out()
        { cout << "Third: " << third << endl;}
};



int main()
{

        First *ptr_first = new First;
        Second sec(100,200);
        ptr_first = &sec;
        ptr_first->Out();                   // (1)
        Third trd(300,400,500);
        ptr_first = &trd;                   // (2)
        ptr_first->Out();
        return 0;
}
```

Функция Out() в данной иерархии классов является виртуальной. Вызов функции ptr_first->Out(); приведет к появлению различных результатов, так как в первом случае будет работать функция Out() класса Second, а во втором - класса Third.

Механизм виртуальных функций работает только через указатели или ссылки на объекты. Объект, определенный через указатель, называется полиморфным, что означает выполнение различных действий, в зависимости, на какой фактический объект ссылается указатель.

***Задание №6.***

Разработать иерархию классов «человек, служащий, студент», в которой класс человек имеет поля имя, фамилия, возраст. Класс служащий дополняет его полем специальность, а класс студент – полями группа и средний балл. Предусмотреть полиморфные методы, позволяющие получить информацию о субъекте в зависимости от его типа.

***Абстрактный класс.***

Если первые полиморфные функции данной иерархии являются избыточными, их можно определить как чисто виртуальные. Функция является чисто виртуальной, если вместо тела будет размещена запись =0, например, void Out()=0;. Класс, содержащий, по меньшей мере, одну чисто виртуальную функцию, называется абстрактным классом. Запрещается объявлять объекты его типа, но допускаются указатели. Абстрактный класс обычно выступает как прародитель большой иерархии, содержащий в себе наиболее общие свойства, характерные для всей иерархии.

***Задание №7.***

В задании №6 класс человек определить как абстрактный. Привести примеры использования абстрактного класса.

***Операция dynamic_cast.***

Эта операция применяется для преобразования указателей родственных классов, в основном указателя базового класса в указатель на производный класс. Формат операции следующий: dynamic_cast<тип *>(выражение);.

Выражение – это указатель или ссылка на класс, тип – производный или базовый для данной иерархии. Операция dynamic_cast предусматривает проверку допустимости преобразования. В случае успешного выполнения операции формирует результат заданного тип, в противном случае в качестве результата возвращает 0 для указателя и bad_cast для ссылки.

Преобразование производного класса в базовый класс называется *повышающим*, относится к стандартным преобразованиям. Явное преобразование можно опустить. Приведение базового класса в производный носит название *понижающего* преобразования, требуется явный вызов оператора dynamic_cast. Возможно также преобразование между базовыми классами для одного производного класса, а также между производными от одного базового класса. Это *перекрестное* преобразование.

*Пример 1. Повышающее преобразование.*

```
class Base
{
protected:
        int base;
public:
        Base(){};
```

```
        Base(int b):base(b){};
        void Out()
        {
                cout << " Base: " << base << endl;
        }
};

class Derived :public Base
{
        int derived;
public:
        Derived(){};
        Derived(int b, int d):Base(b), derived(d){};
        void Out()
        {
                cout << " Derived: " << derived << endl;
        }
};
int main()
{
        setlocale(0,"RUS");
        Base *ptr_Base = new Base(100);
        Derived *ptr_Derived = new Derived(222,333);
        ptr_Base =  dynamic_cast<Base
*>(ptr_Derived);
        //  или  ptr_Base = ptr_Derived;
        ptr_Base->Out();
        system("pause");
        return 0;
}
```

Здесь выражение ptr_Base = dynamic_cast<Base *>(ptr_Derived); преобразует указатель ptr_Derived на класс Derived к указателю класса Base. Результатом работы будет число 222, так как относится к базовой части объекта

производного типа. При удалении этого оператора из программы, результат изменится на 100.

*Пример №2. Понижающее преобразование.*

Операция dynamic_cast чаще всего используется для понижающего преобразования. Производные классы могут иметь методы, которых нет в базовых классах. Для их вызова через указатель на базовый класс необходимо иметь уверенность в том, что указатель действительно ссылается на объект производного класса. Такая проверка осуществляется в момент выполнения приведения типа с использованием RTTI (run time type information) – информации о типе во время исполнения. Для допустимости такой проверки, аргумент операции dynamic_cast должен быть полиморфного типа. Для использования RTTI необходимо подключить заголовочный файл <typinfo>. Рассмотрим модифицированный случай предыдущего примера.

```
class Base
{
protected:
        int base;
public:
        Base(){};
        Base(int b):base(b){};
        virtual void Out()
        {
                cout << " Base: " << base << endl;
        }
};
class Derived :public Base
{
        int derived;
```

```
public:
        Derived(){};
        Derived(int b, int d):Base(b), derived(d){};
        void Out()
        {
                cout << " Derived: " << derived << endl;
        }
};
int main()
{
        setlocale(0,"RUS");
        Base *ptr_Base = new Base;
        Derived *ptr_Derived = new Derived(20, 30);
        ptr_Base = new Derived(222,333);
        dynamic_cast<Derived *>(ptr_Base)->Out();
        system("pause");
        return 0;
}
```

В этом примере в качестве результата следует ожидать число 333. Оператор dynamic_cast<Derived *>(ptr_Base)->Out(); осуществляет приведение указателя базового класса к указателю на производный и поэтому работает метод производного класса. Заметим еще раз, что функция Out() в базовом классе должна быть виртуальной, иначе преобразование не состоится. В этом примере не осуществляется контроля преобразования, что может привести к ошибочным ситуациям.

Контроль преобразования лучше осуществить в какой-либо функции (желательно обрабатывающую исключительную ситуацию). Простой пример приведен ниже. Классы заимствованы из предыдущего примера.

```
void fun(Base *pBase)
{
```

```
        Derived *pDerived = dynamic_cast<Derived
*>(pBase);
        if(pDerived) pDerived->Out();
        else cout << " Передан объект не
производного класса" << endl;
}
```

В теле основной функции выполните следующую последовательность операторов и оцените результаты.

```
Base *ptr_Base = new Base(100);
Derived *ptr_Derived = new Derived(20, 30);
fun(ptr_Base);
fun(ptr_Derived);
```

***Задание №7.***

В иерархии «Человек, служащий, студент» преобразовать указатель на класс человек в указатель на класс служащий и студент. Показать результаты работы.

*Пример №3. Преобразование ссылок.*

Преобразование ссылок производных классов в ссылку на базовый класс относятся к стандартным и не рассматриваться не будет. Понижающее преобразование ссылок имеет несколько другой смысл, чем преобразование указателей. Поскольку ссылка всегда указывает на конкретный объект, операция dynamic_cast должна выполнять преобразование именно к типу этого объекта. Корректность преобразования проверяется автоматически, в случае невозможности преобразования порождается исключение bad_cast. Пример возможного понижающего преобразования без обработки исключений рассмотрен ниже.

```
class A
{
public:
        virtual void f1()
        { cout << " Class A " << endl;}
};

class B :public A
{
public:
        virtual void f2()
        { cout << " Class B " << endl;  }
};


int main()
{
        B b;
        B &b_ref = b;
        A &a_ref = b_ref;
        dynamic_cast<B &>(a_ref).f2();

        system("pause");
        return 0;
}
```

Выражение dynamic_cast<B &>(a_ref).f2(); преобразует ссылку a_ref к ссылке на класс A. Важно наличие оператора A &a_ref = b_ref;, который обеспечивает понижающее преобразование. При его отсутствии система сгенерирует исключение.

Этот пример демонстрационный, для его полноты необходимо отследить тип передаваемого объекта.

***Задание №8.***

В соответствие с заданием №7 обеспечить понижающее преобразование ссылок класса человек в ссылку на класс служащий, ссылки на класс служащий в ссылку на класс студент.

## Лабораторная работа №8.

## Шаблоны Классов. Стандартные шаблоны классов (STL).

***Шаблоны классов. Общие положения.***

Шаблоны классов позволяют создавать параметризированные классы. Параметризированный класс создает семейство родственных классов, которые можно применять к любому типу данных, передаваемых в качестве параметра шаблона. Наиболее широкое применение шаблоны получили при создании контейнерных классов.

Преимуществом шаблонов классов состоит в том, что он может применяться к любым типам данных без переопределения кода.

Синтаксис описания шаблона следующий:

```
template<class Type>  NameClass
{
        // тело шаблона
};
```

Список шаблона <class Type> состоит из служебного слова class или typename после чего идет произвольный идентификатор формального типа. Если предполагается несколько типов, то они перечисляются через запятую. Действие формального типа ограничено рамками данного шаблона.

Рассмотрим простой пример шаблона стека, предназначенного для хранения объектов различного типа.

```
const int Max = 100;
template<typename Type> class Stack
{
protected:
        int top;
        Type st[Max];
public:
        Stack()
        {
                top = -1;
        }
        void Push(Type);
        Type Pop();
        friend ostream &operator << <>(ostream &,
const Stack<Type> &);
};

template<typename Type> ostream &operator
<<(ostream &out, const Stack<Type> &s)
{
        out << " Element: " << s.st[s.top] << endl;
        return out;
}

template<typename Type> void Stack<Type>::
Push(Type var)
{
        st[++top] = var;
}
template<typename Type>Type Stack<Type>::Pop()
{
        return st[top--];
}
```

Здесь приведен пример шаблона класса Stack. Для простоты методы Pop и Push не снабжены проверками на пустоту и переполнение стека. Сам стек представляется как массив объектов. Обратим внимание на то, как определяются методы за пределами классов. Во-первых, все методы шаблонов класса по умолчанию считаются шаблонами функций. Во-вторых, после имени шаблона класса в обязательном порядке должен присутствовать спецификатор типа. Если это определение метода, то в качестве спецификатора выступает имя формального типа, например, Type. Если же объявление реального объекта, то спецификатором должен быть конкретный тип, в том числе и тип, определенный пользователем.

Обратите внимание на объявление дружественной функции (оператора) в теле и за пределами шаблона. В отличие от дружественной функции обычного класса, в шаблоне в объявлении должен присутствовать спецификатор шаблона, идущий сразу за именем функции: friend ostream &operator << <Type>(ostream &, const Stack<Type> &);, который в общем случае может быть пустым. При определении дружественной функции, она объявляется как шаблон, а спецификатор типа после имени не допустим. Он здесь используется для передачи типа шаблона во втором аргументе: const Stack<Type> &;.

Объявление объекта типа шаблон должно обязательно быть со спецификатором конкретного типа, например, Stack<long> st_1; или Stack<double> st_2;. Дальнейшие обращения к объектам шаблонов ничем не отличаются от обращений к объектам обычных классов.

Общие правила описания шаблонов классов:

1. локальные классы не могут содержать шаблоны в качестве своих элементов;
2. шаблоны могут содержать статические компоненты, дружественные функции и классы;

3. шаблоны могут входить в иерархию классов и шаблонов.

Следующий пример показывает использование в качестве параметра шаблона тип, определенный пользователем. Добавьте к предыдущему примеру определение простого класса и передайте его тип в качестве параметра шаблона.

```
class Test
{
        int test;
public:
        Test(){};
        Test(int t):test(t){};
        void Out()
        {
                cout << " Test: " << test << endl;
        }
        friend ostream &operator <<(ostream &, const
Test &);
};
ostream &operator <<(ostream &out, const Test &t)
{
        out << t.test << endl;
        return out;
}
```

Теперь можно использовать тип Test в качестве параметра:

```
Stack<Test> st;
st.Push(100);
st.Push(200);
cout << st.Pop();
```

```
cout << st.Pop();
```

***Задание №1.***

Создать шаблон класса, моделирующий линейный однонаправленный список. Предусмотреть операции добавлении, удаления и поиск заданного элемента. Привести примеры хранения в данном списке объектов стандартного и пользовательского типов.

***Шаблоны и наследование.***

Как было сказано выше, шаблоны классов могут участвовать в иерархии классов, при этом могут выступать как базовые для шаблонов, а так же и для обычных классов и могут быть производными от шаблонов и обычных классов.

Рассмотрим пример шаблонов классов, используя принципы наследования и полиморфизма.

```
// базовый класс
template<class Type> class Base
{
        Type base;
public:
        Base(){};
        Base(Type);
        virtual void Out();
};

template <class Type> Base<Type>:: Base(Type
b):base(b){};
template<class Type> void Base<Type>::Out()
{
        cout << " Template base: " << base << endl;
}
// производный класс
```

```
template<class Type_1, class Type_2> class Derived
:public Base<Type_1>
{
        Type_2 derived;
public:
        Derived(){};
        Derived(Type_1, Type_2);
        void Out();
};
template<class Type_1, class Type_2>
Derived<Type_1, Type_2>::Derived(Type_1 b, Type_2
d):Base<Type_1>(b)
{
        derived = d;
}
template<class Type_1, class Type_2> void
Derived<Type_1, Type_2>::Out()
{
        Base<Type_1>::Out();
        cout << " Template derived: " << derived<<
endl;
}
```

Следующие выражения показывают примеры наследования и полиморфизма:

```
// объект шаблона базового класса
Base<int> bas(100);
// объект шаблона производного класса
Derived<int, double> der(200, 3.1);
// метод Out базового класса
bas.Out();
// метод Out производного класса
der.Out();
```

```
// наследование метода базового класса объектом производного класса
der.Base<int>::Out();
// указатель на шаблон базового класса
Base <int> *ptr_Base = &der;
// работа полиморфной функции производного класса
ptr_Base->Out();
```

В шаблонах можно осуществлять как повышающее преобразование, так и понижающее. Дополним предыдущую программу функцией, получающей в качестве параметра указатель на шаблон базового типа, который внутри преобразуется к указателю на шаблон производного класса:

```
void fun(Base<int> *pBase)
{
    Derived <int, double> *pDerived =
    dynamic_cast<Derived<int, double> *>
(pBase);
    if(pDerived) pDerived->Out();
    else cout << " Передан объект не
производного класса" << endl;
}
```

Оттранслируйте следующие выражения и проанализируйте полученные результаты.

```
Derived<int, double> *ptr_Derived = new Derived<int, double>(100, 200);
Base<int> *ptr_Base = new Base<int>();
fun(ptr_Base);
fun(ptr_Derived);
```

Не забывайте о том, что подобное преобразование допустимо при наличии полиморфных функций.

***Задание №2.***

Создайте шаблон базового класса « символ », позволяющий хранить одиночные символы типа char, int и производный от него шаблон класса « строка », позволяющего хранить смесь произвольных символов определенной длины (например, не более 15 символов в строке). На примере данной иерархии продемонстрировать принципы наследования и полиморфизма в шаблонах классов, а так же возможность преобразования типов в пределах родственных классов.

***Задание №3.***

Дополните предыдущее задание дружественными перегруженными операциями вывода в стандартный поток. Продемонстрируйте вывод одиночного символа и строки в поток. Создайте статическое поле и метод(ы) работы с ним в шаблоне базового класса, покажите возможность наследования статических компонент.

***Задание №4.***

В созданной иерархии замените ключевое слово class, стоящее перед именем базового и производного шаблона классов на слово struct или union. Проанализируйте возможные сообщения компилятора и объясните их.

***Стандартные шаблоны классов (STL).***

Стандартные шаблоны классов являются мощным инструментом создания программного обеспечения в различных прикладных областях. Библиотек шаблонов классов входит в стандарт языка. Примечательно то, что

многие алгоритмы, разработанные в данной библиотеке, прекрасно работают с обычными массивами.

STL состоит из следующих частей: контейнер, алгоритм и итератор.

*Контейнер* – это способ организации хранения данных, например, стек, связный список, очередь. Массив языка C++ в какой-то степени относится к контейнерам. Контейнеры представлены в библиотеке как шаблонные классы, что позволяет быстро менять тип хранимых в них данных.

*Алгоритм* – это процедуры, применяемые к контейнерам для обработки хранящихся данных. Они представлены как шаблоны функций. Однако эти функции не принадлежат какому-либо классу контейнеру, они не являются дружественными функциями. Наоборот, это совершенно независимые функции. Это обеспечивает универсальность алгоритмов и позволяет использовать алгоритмы не только к контейнерам, но и к другим типам данных.

Итераторы – обобщенная концепция указателей, они ссылаются на элементы контейнеров. Итераторы – ключевая часть STL, поскольку они связывают алгоритмы с контейнерами. Иногда итераторы называют интерфейсом библиотеки шаблонов.

***Контейнеры.***

Контейнеры представляют собой хранилища данных. При этом не имеет значения, какого типа данные в них хранятся. Это могут быть базовые типы int, float, double, etc, а также типы, объявленные пользователем. Контейнеры STL делятся на две основные категории: последовательные контейнеры и ассоциативные. К последовательным контейнерам относятся векторы, списки и очереди с двусторонним доступом. К ассоциативным

контейнерам – множества, мультимножества, отображения и мультиотображения. Кроме того, наследниками последовательных контейнеров выступают стек, очередь и приоритетная очередь.

***Методы.***

Для выполнения простых операций над контейнерами используют методы. Методы проще алгоритмов, применять их можно к большей части классов контейнеров. Некоторые методы перечислены ниже.

size() – возвращает число элементов контейнера;

empty() – возвращает true, если контейнер пуст;

max_size() – возвращает максимально допустимый размер контейнера;

begin() – возвращает итератор на начало контейнера;

end() – возвращает итератор на последнюю позицию в контейнере;

rbegin() – возвращает реверсивный итератор на конец контейнера;

rend() – возвращает реверсивный итератор на начало.

***Алгоритмы.***

Алгоритм – это функция, которая производит некоторое действие над элементами контейнера. В стандарте языка C++ алгоритм независимая шаблонная функция, применимая даже к обычным массивам. Рассмотрим отдельные методы на примерах.

***Алгоритм find().***

Этот алгоритм ищет первый элемент в контейнере, значение которого равно указанному. Общий формат метода следующий: find(beg, end, val), где beg – итератор

(указатель) первого значения, которое нужно проверить; end – итератор последней позиции; val – искомое значение.

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

int arr[] = {11, 22, 33, 44, 55, 66, 77, 88, 99, 0};

int main()
{
	setlocale(0,"RUS");
	int *ptr;
	ptr = find(arr, arr+10, 55);
	cout << " Первый объект со значением 55 найден в позиции: " << (ptr-arr) << endl;
	system("pause");
	return 0;
}
```

Для использования алгоритмов необходимо подключить файл algorithm. В рассмотренном примере в качестве контейнера выступает обычный массив, в котором ищется число 55. Начало поиска – элемент с нулевым индексом, окончание – указатель на единицу больший, чем указатель на последний элемент. Подобный подход носит название «значение после последнего». Результатом работы программы будет число 4. Если в контейнере нет искомого значения, программа выдаст указатель на следующий элемент после последнего. Для рассмотренного примера это будет число 10.

Следующий фрагмент демонстрирует пример использования метода find для контейнера типа vector.

```
vector<int> v;
        for(int i = 1; i <= 100; i++)
        {
                v.push_back(i*i);
        }

        if(find(v.begin(), v.end(), 49) != v.end())
        {
                cout <<  " Число 49 найдено в списке
                квадратов натуральных чисел, не
                превосходящих 100 " << endl;
        }
        else
        {
                cout <<  "  число 49 не найдено "
        }
```

Диапазон поиска определяется с помощью методов begin и end.

***Задание №5.***

Продемонстрировать работу алгоритма find() для последовательных контейнеров списки (list), очереди с двусторонним доступом (deque) и стек (stack). В программе необходимо подключить соответствующие заголовочные файлы.

***Алгоритм count().***

Этот алгоритм подсчитывает, сколько элементов в контейнере имеют данное значение. Его формат подобен формату алгоритма find. Рассмотрим фрагмент программного кода, использующего алгоритм count().

```
int arr[] = {11, 22, 33, 22, 55, 22, 77, 22, 99, 0};

int main()
{
	setlocale(0,"RUS");
	int n = count(arr, arr+10, 22);
	cout << " Число 22 встречается " << n << " раз(а) "
<< endl;

	system("pause");
	return 0;
}
```

Результатом алгоритма count() будет число вхождений искомого элемента или число 0, если искомый элемент не входит в данный контейнер.

***Задание №6.***

Продемонстрировать работу алгоритма count() для последовательных контейнеров списки (list), очереди с двусторонним доступом (deque) и стек (stack).

***Алгоритм search().***

Алгоритм search() оперирует одновременно с двумя контейнерами. Он отыскивает некоторую последовательность, входящую в один контейнер в другом контейнере. Формат алгоритма следующий: search(beg, end, sbeg, send), где beg – начало первого контейнера, end – конец первого контейнера, sbeg – начало второго контейнера, send – конец второго контейнера. Результатом работы алгоритма будет позиция, начиная с которой обнаружено совпадение. Если совпадения не обнаружено, необходимо обеспечить вывод соответствующего

сообщения. Следующий фрагмент показывает пример работы алгоритма.

```
int arr1[] = {11, 24, 33, 11, 22, 33, 22, 66, 77, 8};
int arr2[] = {11, 22, 33};
int main()
{
	setlocale(0,"RUS");
	int *ptr;
	ptr = search(arr1, arr1+10, arr2, arr2+3);
	if(ptr == arr1+10)  cout << " Совпадений нет ";
	else cout << " Совпадения в позиции " << (ptr-arr1)
<< endl;
	system("pause");
	return 0;
}
```

Несложно заметить, что результатом данного примера будет число 3, так как совпадение двух массивов начинается с элемента со значением 11 первого массива, имеющего индекс 3.

Важно заметить, что параметрами алгоритма search() не обязательно должны быть контейнеры одного и того же типа. Второй параметр в этом случае играет роль маски. В этом состоит универсальность библиотеки STL.

***Задание №7.****

Объявить классы Person и Student, имеющие поля Name, Age, идентифицирующие имя и возраст человека и студента. В каждом из классов предусмотреть поля, характерные для его типов, например для класса Student таким полем может быть Ball, а для класса Person – поле Prof, идентифицирующее профессию человека. Объявить два контейнера типа vector, в одном из которых находятся

объекты типа Person, а во втором – объекты типа Student. Используя алгоритм search(), определить последовательность значений, заданных контейнером Student, попадающих в контейнер Person. Сравнение проводить по значениям полей Name. Показать пример работы алгоритма count() на заданных контейнерах, в качестве критерия поиска можно использовать поля Ball типа Student, подсчитывающего число студентов, сдавших последнюю сессию на 4 и на 5. Для класса Person критерием выбора может быть величина оклада.

***Алгоритм sort().***

Этот алгоритм сортирует элементы контейнера в порядке их увеличения. Следующий пример показывает возможность использования алгоритма сортировки для целочисленных контейнеров, например, для последовательности символов.

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

typedef unsigned int UINT;
vector<char> arr_char;

int main()
{
	arr_char.push_back('T');
	arr_char.push_back('E');
	arr_char.push_back('G');
	arr_char.push_back('L');
	arr_char.push_back('A');
	arr_char.push_back('Z');
```

```
	arr_char.push_back('K');
	arr_char.push_back('W');
	sort(arr_char.begin(), arr_char.end());
	for(UINT j=0; j < arr_char.size(); j++)
		cout << arr_char[j] << ' ';
	cout << endl;
	setlocale(0,"RUS");
	system("pause");
	return 0;
}
```

Алгоритм sort() имеет два параметра, идентифицирующие начало и конец сортируемого контейнера. В данном примере начало, и конец последовательности получаются как результат работы соответствующих методов.

***Алгоритм merge().***

Этот алгоритм работает с тремя контейнерами, объединяя два из них в третий. Рассмотрим пример использования этого алгоритма.

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

int arr1[] = {2, 3, 4, 6, 8};
int arr2[] = {1, 3, 5};
int dest[8];

int main()
{
	setlocale(0,"RUS");
	merge(arr1, arr1+5, arr2, arr2+3, dest);
	for(int i=0; i<8; i++)
```

```
            cout << dest[i] << ' ';
        cout << endl;
    system("pause");
    return 0;
}
```

***Задание №8.***

Используя алгоритм merge() объединить два вектора и списка в результирующие контейнеры.

Функциональные объекты.

Многие алгоритмы для своей работы требуют функциональные объекты. Функциональным объектом считается объект шаблонного класса, в котором есть единственный метод – перегруженная операция вызова функции – (). Рассмотрим пример использования алгоритма sort с функциональным объектом, позволяющим отсортировать последовательность в порядке убывания.

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

double ddata[] = {38.44, 64.22, -9.345, 0.453, 8.01, 23};

int main()
{
    setlocale(0,"RUS");

    sort(ddata, ddata+6, greater<double>() );
    for(int j=0; j<6; j++)
        cout << ddata[j] << ' ';
    cout << endl;
    system("pause");
```

```
        return 0;
}
```

Функциональным объектом здесь является метод greater<double>(), который сортирует последовательность по убыванию. Обычно алгоритм sort() сортирует по возрастанию.

Функциональные объекты могут успешно работать только со стандартными типами данных C++ или с классами, в которых определены (перегружены) операторы <, <=, >, >= , ==, etc. Проблему можно обойти, определив собственную функцию для функционального объекта. Следующий пример показывает это.

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

char *Names[] = { "Петя", "Маша", "Дима", "Вова", "Вася",
"Филимон"};

bool alpha(char *n1, char *n2)
{
        return(strcmp(n1,n2)<0) ? true : false;
}

int main()
{
        setlocale(0,"RUS");
        sort(Names, Names+6, &alpha);
        for(int i=0; i<6; i++)
                cout << Names[i] << ' ';
        cout << endl;
        system("pause");
```

```
    return 0;
}
```

Здесь вместо функционального объекта выступает функция bool alpha(), которая сравнивает две строки, используя стандартную функцию strcmp(). Наличие оператора взятия адреса перед именем функции alpha() не обязательно. Это скорее дань традициям языка С.

***Задание №9.***

Используя собственные функции как функциональные объекты, показать пример их применения к спискам, очередям с двусторонним доступом.

***Задание №10.***

Самостоятельно изучить последовательные контейнеры: векторы, списки и очереди с двусторонним доступом, а также алгоритмы и методы для работы с ними. Показать примеры их использования.

***Итераторы.***

Для обращения к элементам обычного массива используют операцию индексирования [] или обращение через указатель. Использовать указатели к более сложным контейнерам затруднительно. Во-первых, элементы контейнера могут храниться в памяти не последовательно, а сегментировано. Методы доступа к ним существенно усложняются. Нельзя просто инкрементировать указатель для получения следующего значения. Во-вторых, при добавлении или удаления элементов из середины контейнера, указатель может получить некорректное значение.

Одним из решений проблемы является создание класса «умных» указателей – итераторов. Объект такого класса является оболочкой для методов, оперирующих с

обычными указателями. Для них перегружены операции ++ и *, адекватно работающих даже в том случае, когда элементы сегментированы в памяти.

Итераторы играют важную роль в STL. Они определяют какие алгоритмы использовать к каким контейнерам. По категориям различают следующие виды итераторов: входные, выходные, прямые, двунаправленные и случайного доступа.

***Работа с итераторами.***

*Доступ к данным. Вставка данных.*

В контейнерах с итераторами произвольного доступа (векторах, очередях с двусторонним доступом), итерация осуществляется с помощью оператора []. Однако к спискам эту операцию применить нельзя. Рассмотрим пример работы со списками через итераторы.

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

int main()
{
	// доступ к данным
	double  arr[] = {2.33, 44.32, -6.54, .8};
	list<double> doubleList;
	for(int i=0; i<4; i++)
		doubleList.push_back(arr[i]);
	list<double> ::iterator iter = doubleList.begin();
	for(iter=doubleList.begin(); iter!=doubleList.end();
iter++)
		cout << *it++ << ' ';
		cout << endl;
```

```
        // вставка данных
        list<int> intList(5);
        list<int> ::iterator it;
        int data = 0;
        for(it=intList.begin(); it!=intList.end(); it++)
                *it = data += 2;
        it = intList.begin();
        while(it !=intList.end())
                cout << *it++ << ' ';
                cout << endl;
        system("pause");
        return 0;
}
```

Объявление итераторов в данной программе:

- оператор list<double> ::iterator iter = doubleList.begin(); - определяет двунаправленный итератор для работы со списком, содержащего вещественные числа и инициализирует его указателем на начало списка;

- оператор list<int> ::iterator it; - определяет двунаправленный итератор для списка с целочисленными элементами.

Итераторы для вектора или очереди автоматически делаются с произвольным доступом. Имена итераторов – обычные идентификаторы, подчиняющиеся законам языка. Еще одно важное замечание: обращение к итераторам, например, *it = data += 2; или cout << *it++ << ' '; осуществляется как к обычным указателям.

***Задание №11.***

Определить вектор, очередь содержащие элементы стандартного типа (целого, вещественного и т. д.), определить для них итераторы и опробовать алгоритмы и методы работы с ними. Определить класс, состоящий из небольшого числа компонент, сохранить объекты этого класса в очереди. Показать пример использования итераторов при работе с очередью объектов собственного типа.

***Специализированные итераторы.***

Под специализированными итераторами понимают адаптеры итераторов и потоковые итераторы.

Адаптеры итераторов бывают трех видов: обратные (реверсивные) итераторы, итераторы вставки и итераторы неинициализированного хранения.

*Обратные итераторы.*

Для осуществления реверсивного прохода по контейнеру используются обратные итераторы. Следующий пример показывает определение и применение реверсивного итератора для вывода списка в обратном порядке.

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

int main()
{
	int arr[] = {2, 4, 6, 8, 10, 12};
	list<int> intList;

	for(int i=0; i<6; i++)
```

```
            intList.push_back(arr[i]);

        list<int>::reverse_iterator revit;
        revit = intList.rbegin();
        while(revit != intList.rend())
                cout << *revit++ << ' ';
        cout << endl;
        system("pause");

        return 0;
};
```

Обратный проход списка в данном примере здесь обеспечивает объявление реверсивного итератора list<int>::reverse_iterator revit;. При его использовании следует использовать методы rbegin() и rend(). С обычными итераторами их использовать нельзя.

***Задание №12.***

Определить список, содержащий элементы собственного типа (например, из предыдущего примера), объявить обратный итератор и обеспечить вывод списка в обратном порядке.