Содержание

[Практическая работа №12. Работа с бинарными файлами. 2](#_Toc501529995)

[Теоретические сведения. 2](#_Toc501529996)

[Контрольные вопросы. 7](#_Toc501530002)

[Задания к практической работе № 12 7](#_Toc501530003)

[Задания к практической работе № 13 12](#_Toc501530004)

[Практическая работа №14. Программирование и использование модулей 13](#_Toc501530005)

[Теоретический материал. 13](#_Toc501530006)

[Контрольные вопросы. 19](#_Toc501530007)

[Задания к практической работе 14. 19](#_Toc501530008)

[Практическая №15. Создание класса и использование его объектов 20](#_Toc501530009)

[Теоретические сведения 20](#_Toc501530010)

[Контрольные вопросы 27](#_Toc501530011)

[Задания к практической работе № 15 27](#_Toc501530012)

[Практическая работа №16. Иерархия объектов и группа. Итераторы 30](#_Toc501530013)

[Теоретические сведения 30](#_Toc501530014)

[Контрольные вопросы 35](#_Toc501530015)

[Задания к практической работе № 16 35](#_Toc501530016)

[Практическая работа №17. Перегрузка операций 38](#_Toc501530017)

[Теоретические сведения 38](#_Toc501530018)

[Контрольные вопросы 41](#_Toc501530019)

[Задания к практической работе 17 43](#_Toc501530020)

[ЛИТЕРАТУРА 47](#_Toc501530021)

# Практическая работа №12. Работа с бинарными файлами.

## Теоретические сведения.

Для работы с файлами в C++ можно воспользоваться стандартными библиотеками, унаследованными еще от C. Типы данных и функции для работы с файлами расположены в библиотеке <cstdio>. Для работы с файлом необходимо определить переменную типа FILE\*.

### Открытие файла при помощи fopen

Файл открывается при помощи fopen, которая возвращает информацию потока ввода-вывода, прикрепленного к указанному файлу или другому устройству, с которого идет чтение (или в который идет запись). В случае неудачи функция возвращает нулевой указатель.

Она определяются как

FILE \*fopen(**const** **char** \*path, **const char** \*mode);

Функция fopen по сути представляет из себя "обертку" более высокого уровня системного вызова open операционной системы. Аналогично, fclose является оберткой системного вызова close, а сама структура FILE языка С++ зачастую обращается к соответствующему файловому дескриптору ОС.

Параметр mode (режим) для fopen должен быть строковый и начинаться с одной из следующих последовательностей:

|  |  |  |
| --- | --- | --- |
| режим | описание | начинает с.. |
| r | rb |  | открывает для чтения | начала |
| w | wb |  | открывает для записи (создает файл в случае его отсутствия). Удаляет содержимое и перезаписывает файл. | начала |
| a | ab |  | открывает для добавления (создает файл в случае его отсутствия) | конца |
| r+ | rb+ | r+b | открывает для чтения и записи | начала |
| w+ | wb+ | w+b | открывает для чтения и записи. Удаляет содержимое и перезаписывает файл. | начала |
| a+ | ab+ | a+b | открывает для чтения и записи (добавляет в случае существования файла) | конца |

Значение "b" зарезервировано для двоичного режима С++. Стандарт языка С++ описывает два вида файлов — текстовые и двоичные — хотя операционная система не требует их различать. Текстовый файл – файл, содержащий текст, разбитый на строки при помощи некоторого разделяющего символа окончания строки или последовательности (в Unix – одиночный символ перевода строки; в Microsoft Windows за символом перевода строки следует знак возврата каретки). При считывании байтов из текстового файла, символы конца строки обычно заменяются символом ‘\0’ (NULL). Двоичный файл – файл, из которого байты считываются и выводятся в "сыром" виде без какого-либо связывания (подстановки).

При открытом файле в режиме обновления ( '+' в качестве второго или третьего символа аргумента обозначения режима) и ввод и вывод могут выполняться в одном потоке. Тем не менее, запись не может следовать за чтением без промежуточного вызова fflush или функции позиционирования в файле (fseek, fsetpos или rewind), а чтение не может следовать за записью без промежуточного вызова функции позиционирования в файле.

Режимы записи и добавления пытаются создать файл с заданным именем, если такого файла еще не существует. Как указывалось выше, если эта операция оканчивается неудачей, fopen возвращает NULL.

### Закрытие потока при помощи fclose

Функция fclose принимает один аргумент: указатель на структуру FILE потока для закрытия.

**int** fclose(FILE \*fp);

Функция возвращает нуль в случае успеха и EOF в случае неудачи. При нормальном завершении программы функция вызывается автоматически для каждого открытого файла.

### Запись в файл при помощи fwrite

fwrite определяется как

**int** fwrite(**const void**\* array,size\_t size,size\_t count,FILE \* stream );

Функция fwrite записывает блок данных в поток. Таким образом запишется массив размера count элементов в текущую позицию в потоке. Для каждого элемента запишется size байт. Индикатор позиции в потоке изменится на число байт, записанных успешно. Возвращаемое значение будет равно count в случае успешного завершения записи. В случае ошибки возвращаемое значение будет меньше count.

Следующая программа открывает файл пример.txt, записывает в него строку символов, а затем его закрывает.

#include <сstdio>
#include <сstring>
#include <сstdlib>
**using namespace** std;
**int** main(**int** argc, **char**\*\* argc)
{
 FILE \*fp;
 size\_t count;
 **char** str[] = "привет\n";
 fp = fopen(*"пример.txt",* *"wb"*);
 if(fp == NULL) return 1;
 count = fwrite(str, 1, **sizeof**(str)/**sizeof**(**char**), fp);
 fclose(fp);
 **return** 0;

}

### Чтение из файла при помощи fread

fread определяется как

**int** fread(**const void**\* array,size\_t size,size\_t count,FILE \* stream );

Функция fread читает блок данных из потока. Таким образом считается массив размера count элементов из текущей позиции в потоке. Для каждого элемента считается size байт. Индикатор позиции в потоке изменится на число байт, считанных успешно. Возвращаемое значение будет равно count в случае успешного завершения чтения. В случае ошибки возвращаемое значение будет меньше count.

Следующая программа открывает файл пример.txt, читает из него в него строку символов, а затем его закрывает.

#include <сstdio>
#include <сstring>
#include <сstdlib>
**using namespace** std;
**int** main(**int** argc, **char**\*\* argc)
{
 FILE \*fp;
 size\_t count;
 **char** str[64]= {0};
 fp = fopen(*"пример.txt",* *"rb"*);
 if(fp == NULL) return 1;
 count = fread(str, 1, **sizeof**(str)/**sizeof**(**char**), fp);
 fclose(fp);
 **return** 0;

}

### Файловый ввод ввод/вывод в библиотеке STL

Файловый ввод-вывод в STL ничем не отличается от консольного. За единственным исключением – если данные читаются из файла, то в любой момент можно вернуться к началу файла и считать все заново.

Для того, чтобы в C++ работать с файлами, необходимо подключить заголовочный файл fstream:

#include <fstream>

После этого можно объявлять объекты, привязанные к файлам: для чтения данных из файла используются объекты типа ifstream (аббревиатура от **i**nput **f**ile **stream**, для записи данных в файл используются объекты типа ofstream (**o**utput **f**ile **stream**). Например

ifstream in;// Поток in будем использовать для чтения
ofstream out;// Поток out будем использовать для записи

Чтобы привязать тот или иной поток к файлу (открыть файл для чтения или для записи) используется метод open, которому необходимо передать параметр – текстовую строку, содержащую имя открываемого файла.

in.open(*"input.txt"*);
out.open*("output.txt"*);

После открытия файлов и привязки их к файловым потокам, работать с файлами можно так же, как со стандартными потоками ввода-вывода cin и cout. Например, чтобы вывести значение переменной x в поток out используются следующая операция

out<<x;

А чтобы считать значение переменной из потока in

in>>x;

Для закрытия ранее открытого файла используется метод close() без аргументов:

in.close();
out.close();

Закрытый файловый поток можно переоткрыть заново при помощи метода open, привязав его к тому же или другому файлу.

При считывании данных из файла может произойти ситуация достижения конца файла (**end of file**, сокращенно **EOF**). После достижения конца файла никакое чтение из файла невозможно. Для того, чтобы проверить состояние файла, необходимо вызвать метод eof(). Данный метод возвращает true, если достигнут конец файла или false, если не достигнут. Кроме того, состояние файлового потока можно проверить, если просто использовать идентификатор потока в качестве логического условия:

**if**(in)
{
}

Также можно использовать в качестве условия результат, возвращаемой операцией считывания. Если считывание было удачным, то результат считается истиной, а если неудачным – ложью. Например, организовать считывание последовательности целых чисел можно так:

**int** d;
**while**(in>>d)
{
}

А организовать считывание файла построчно (считая, что строка заканчивается символом перехода на новую строку) так:

string S;
**while**(getline(in,S))
{
}

## Контрольные вопросы.

1. Что такое файл? Какие существуют виды файлов?
2. Какими стандартными функциями располагает С++ для работы с файлами?
3. Каково должно быть содержание программы по созданию файла и его корректировки (замена элементов, добавление элементов, удаление элементов)?
4. Каковы особенности работы с текстовыми файлами?
5. Каковы особенности работы с типизированными файлами?
6. Как подсчитать число строк в текстовом файле?

## Задания к практической работе № 12

1. Даны текстовые файлы f1 и f2. Переписать с сохранением порядка следования компоненты файла f1 в файл f2, а компоненты файла f2 в файл f1. Использовать вспомогательный файл h.
2. Дан текстовый файл f. Записать в файл g компоненты файла f в обратном порядке.
3. Даны текстовые файлы f и g. Записать в файл h сначала компоненты файла f, затем – компоненты файла g с сохранением порядка.
4. Дан файл f, компоненты которого являются целыми числами. Получить в файле g все компоненты файла f: а) являющимися четными числами; б) делящиеся на 3 и не делящиеся на 7; в) являющимися точными квадратами.
5. Дан файл f, компоненты которого являются целыми числами. Получить файл g, образованный из файла f исключением повторных вхождений одного и того же числа.
6. Дан файл f, компоненты которого являются целыми числами. Никакая из компонент файла не равна нулю. Файл f содержит столько же отрицательных чисел, сколько и положительных. Используя вспомогательный файл h, переписать компоненты файла f в файл g так, чтобы в файле g:
	1. не было двух соседних чисел с одинаковым знаком;
	2. вначале шли положительные, затем отрицательные числа;
	3. числа шли в следующем порядке: два положительных, два отрицательных, два положительных, два отрицательных и т.д. (предполагается, что число компонент в файле f делится на 4).
7. Дан файл f, компоненты которого являются целыми числами. Никакая из компонент файла не равна нулю. Числа в файле идут в следующем порядке: десять положительных, десять отрицательных, десять положительных, десять отрицательных и т.д. Переписать компоненты файла f в файл g так, чтобы в файле g числа шли в следующем порядке: а) пять положительных, пять отрицательных, пять положительных, пять отрицательных и т.д.; б) двадцать положительных, двадцать отрицательных, двадцать положительных, двадцать отрицательных и т.д. (предполагается, что число компонент в файле f делится на 40).
8. Дан файл f, компоненты которого являются целыми числами. Записать в файл g наибольшее значение первых пяти компонент файла f, затем – следующих пяти компонент и т.д. Если в последней группе окажется менее пяти компонент, то последняя компонента файла g должна быть равна наибольшей из компонент файла f, образующих последнюю (неполную) группу.
9. Дан символьный файл f: а) подсчитать число вхождений в файл сочетаний 'ab'; б) определить входит ли в файл сочетание 'abcdefgh'; в) подсчитать число вхождений в файл каждой из букв 'a','b','c','d', 'e','f' и вывести результат в виде таблицы
a --> Na b --> Nb c --> Nc
d --> Nd e --> Ne f --> Nf
где Na, Nb, Nc, Nd, Ne, Nf – числа вхождений соответствующих букв.
10. Дан символьный файл f. Группы символов, разделенные пробелами (одним или несколькими) и не содержащие пробелов внутри себя, будем называть словами. Удалить из файла все однобуквенные слова и лишние пробелы. Результат записать в файл g. Типизированные файлы
11. Багаж пассажира характеризуется количеством вещей и общим весом вещей. Дан файл f, содержащий сведения о багаже нескольких пассажиров. Сведения о багаже каждого пассажира представляют собой запись с двумя полями: одно поле целого типа (количество вещей) и одно – действительное (вес в килограммах).
	1. Найти багаж, средний вес одной вещи в котором отличается не более, чем на 0.3 кг от общего среднего веса одной вещи.
	2. Найти число пассажиров, имеющих более двух вещей и число пассажиров, количество вещей которых превосходит среднее число вещей.
	3. Определить, имеются ли два пассажира, багажи которых совпадают по числу вещей и различаются по весу не более чем на 0,5 кг.
	4. Выяснить, имеется ли пассажир, багаж которого превышает багаж каждого из остальных пассажиров и по числу вещей, и по весу.
	5. Выяснить, имеется ли пассажир, багаж которого состоит из одной вещи весом менее 30 кг.
12. Прямая на плоскости задается уравнением ax+by+c=0, где a и b одновременно не равны нулю. Будем рассматривать только прямые, для которых коэффициенты a,b,c – целые числа. Пусть f – файл, содержащий коэффициенты нескольких прямых (не менее трех). Переписать из файла f в файл g коэффициенты тех прямых, которые:
	1. параллельны первой из прямых, заданной в файле f;
	2. указаны в пункте а), но дополнительно требуется, чтобы все прямые были различны;
	3. пересекают первую из прямых, заданных в файле f;
	4. указаны в пункте b), но дополнительно требуется, чтобы среди прямых не было параллельных.
13. Условие предыдущей задачи сохраняется. Требуется получить в файле g коэффициенты всех различных прямых файла f.
14. Сведения об ученике состоят из его имени и фамилии и названия класса (года обучения и буквы), в котором он учится. Дан файл f, содержащий сведения об учениках школы:
	1. выяснить, имеются ли в школе однофамильцы;
	2. выяснить, имеются ли однофамильцы в каких-либо параллельных классах;
	3. выяснить, имеются ли однофамильцы в каком-нибудь классе;
	4. ответить на вопросы а)-в), но в отношении учеников, у которых совпадают и имя, и фамилия;
	5. выяснить, в каких классах насчитывается более 35 учащихся;
	6. выяснить на сколько человек в восьмых классах больше, чем в десятых;
	7. собрать в файле g сведения об учениках 9-х и 10-х классов, поместив вначале сведения об учениках класса 9а, затем 9б и т.д., затем 10а, 10б и т.д.
15. Дан файл f, содержащий те же сведения об учениках, что и в предыдущей задаче, и дополнительно отметки, полученные учениками в последней четверти.
	1. выяснить, сколько учеников школы не имеют отметок ниже четырех;
	2. собрать в файле g сведения о лучших учениках школы, т.е. об учениках, не имеющих отметок ниже четырех и по сумме баллов не уступающих другим ученикам своего и параллельных классов.
16. Сведения об автомобиле состоят из его марки, номера и фамилии владельца. Дан файл f, содержащий сведения о нескольких автомобилях. Найти: а) фамилии владельцев и номера автомобилей данной марки; б) количество автомобилей каждой марки. Найденные данные записать в файл g.
17. Дан файл f, содержащий различные даты. Каждая дата – это число, месяц и год. Найти: а) год с наименьшим номером; б) все весенние даты; в) самую позднюю дату. Найденные данные записать в файл g.
18. Дан файл f, содержащий сведения о книгах. Сведения о каждой из книг – это фамилия автора, название и год издания. 1) Найти названия книг данного автора, изданных с 1960 г. 2) Определить, имеется ли книга с названием "Информатика". Если да, то сообщить фамилию автора и год издания. Если таких книг несколько, то сообщить имеющиеся сведения обо всех книгах.
19. Дан файл f, содержащий сведения о кубиках: размер каждого кубика (длина ребра в сантиметрах), его цвет (красный, зеленый, желтый или синий) и материал (деревянный, металлический, картонный). Найти: а) количество кубиков каждого из перечисленных цветов и их суммарный объем; б) количество деревянных кубиков с ребром 3 см и количество металлических кубиков с ребром, большим 5 см.
20. Дан файл f, содержащий сведения о веществах: указывается название вещества, его удельный вес и проводимость (проводник, полупроводник, изолятор). 1) Найти удельные веса и названия всех полупроводников. 2) Выбрать данные о проводниках и упорядочить их по убыванию удельных весов.

## Задания к практической работе № 13

Написать программу, реализующую ввод данных о студентах: фамилию, имя, оценки по предметам «Базы данных», «Программирование» и теория информации». Вывод всех списка студентов на отчисление (хотя бы одна двойка), а также список круглых отличников. Данные о студенте организовать с помощью структуры. Список студентов хранить в файле.

# Практическая работа №14. Программирование и использование модулей

## Теоретический материал.

При создании больших проектов, в особенности если в разработке участвуют несколько человек, не возможно написать всю программу целиком в одном файле. Еще в 60-70 гг. прошлого века был разработан принцип модульности в программировании, который предполагает написание программы в нескольких файлах и последующей совместной их компиляцией. Это также было обусловлено тем, что со временем при проектировании программ акцент сместился с организации процедур на организацию структур данных.

Модулем обычно называют совокупность связанных процедур и тех данных, которыми они управляют. Парадигма программирования приобрела вид:

Определите, какие модули нужны; поделите программу так, чтобы данные были скрыты в этих модулях.

Эта парадигма известна также как «принцип сокрытия данных». Если в языке нет возможности сгруппировать связанные процедуры вместе с данными, то он плохо поддерживает модульный стиль программирования.

Итак, программный модуль – это любой фрагмент описания процесса, оформляемый как самостоятельный программный продукт, пригодный для использования в описаниях процесса. Это означает, что каждый программный модуль программируется, компилируется и отлаживается отдельно от других модулей программы, и тем самым, физически разделен с другими модулями программы. Более того, каждый разработанный программный модуль может включаться в состав разных программ, если выполнены условия его использования, декларированные в документации по этому модулю. Таким образом, программный модуль может рассматриваться и как средство борьбы со сложностью программ, и как средство борьбы с дублированием в программировании (т.е. как средство накопления и многократного использования программистских знаний).

Не всякий программный модуль способствует упрощению программы. Выделить хороший с этой точки зрения модуль является серьезной творческой задачей. Для оценки приемлемости выделенного модуля используются некоторые критерии. Так, Хольт предложил следующие два общих таких критерия:

* хороший модуль снаружи проще, чем внутри;
* хороший модуль проще использовать, чем построить.

**Размер** модуля измеряется числом содержащихся в нем операторов или строк. Модуль не должен быть слишком маленьким или слишком большим. Маленькие модули приводят к громоздкой модульной структуре программы и могут не окупать накладных расходов, связанных с их оформлением. Большие модули неудобны для изучения и изменений, они могут существенно увеличить суммарное время повторных трансляций программы при отладке программы. Обычно рекомендуются программные модули размером от нескольких десятков до нескольких сотен операторов.

**Прочность** модуля – это мера его внутренних связей. Чем выше прочность модуля, тем больше связей он может спрятать от внешней по отношению к нему части программы и, следовательно, тем больший вклад в упрощение программы он может внести. Для оценки степени прочности модуля Майерс предлагает упорядоченный по степени прочности набор из семи классов модулей. Самой слабой степенью прочности обладает модуль, прочный по совпадению. Это такой модуль, между элементами которого нет осмысленных связей. Такой модуль может быть выделен, например, при обнаружении в разных местах программы повторения одной и той же последовательности операторов, которая и оформляется в отдельный модуль. Необходимость изменения этой последовательности в одном из контекстов может привести к изменению этого модуля, что может сделать его использование в других контекстах ошибочным. Такой класс программных модулей не рекомендуется для использования. Вообще говоря, предложенная Майерсом упорядоченность по степени прочности классов модулей не бесспорна. Однако, это не очень существенно, так как только два высших по прочности класса модулей рекомендуются для использования.

**Функционально прочный модуль** – это модуль, выполняющий (реализующий) одну какую-либо определенную функцию. При реализации этой функции такой модуль может использовать и другие модули. Такой класс программных модулей рекомендуется для использования.

**Информационно прочный модуль** – это модуль, выполняющий (реализующий) несколько операций (функций) над одной и той же структурой данных (информационным объектом), которая считается неизвестной вне этого модуля. Для каждой из этих операций в таком модуле имеется свой вход со своей формой обращения к нему. Такой класс следует рассматривать как класс программных модулей с высшей степенью прочности. Информационно прочный модуль может реализовывать, например, абстрактный тип данных.

**Сцепление** модуля – это мера его зависимости по данным от других модулей. Характеризуется способом передачи данных. Чем слабее сцепление модуля с другими модулями, тем сильнее его независимость от других модулей. Для оценки степени сцепления Майерс предлагает упорядоченный набор из шести видов сцепления модулей. Худшим видом сцепления модулей является сцепление по содержимому. Таким является сцепление двух модулей, когда один из них имеет прямые ссылки на содержимое другого модуля (например, на константу, содержащуюся в другом модуле). Такое сцепление модулей недопустимо. Не рекомендуется использовать также сцепление по общей области – это такое сцепление модулей, когда несколько модулей используют одну и ту же область памяти. Единственным видом сцепления модулей, который рекомендуется для использования современной технологией программирования, является параметрическое сцепление (сцепление по данным по Майерсу) – это случай, когда данные передаются модулю либо при обращении к нему как значения его параметров, либо как результат его обращения к другому модулю для вычисления некоторой функции. Такой вид сцепления модулей реализуется на языках программирования при использовании обращений к процедурам (функциям).

**Рутинность** модуля – это его независимость от предыстории обращений к нему. Модуль называется рутинным, если результат (эффект) обращения к нему зависит только от значений его параметров (и не зависит от предыстории обращений к нему). Модуль называется зависящим от предыстории, если результат (эффект) обращения к нему зависит от внутреннего состояния этого модуля, изменяемого в результате предыдущих обращений к нему. Майерс не рекомендует использовать зависящие от предыстории (непредсказуемые) модули, так как они провоцируют появление в программах хитрых (неуловимых) ошибок. Однако такая рекомендация является неконструктивной, так как во многих случаях именно зависящий от предыстории модуль является лучшей реализаций информационно прочного модуля. Поэтому более приемлема следующая (более осторожная) рекомендация:

* всегда следует использовать рутинный модуль, если это не приводит к плохим (не рекомендуемым) сцеплениям модулей;
* зависящие от предыстории модули следует использовать только в случае, когда это необходимо для обеспечения параметрического сцепления;
* в спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость таким образом, чтобы было возможно прогнозировать поведение (эффект выполнения) данного модуля при разных последующих обращениях к нему.

В связи с последней рекомендацией может быть полезным определение внешнего представления (ориентированного на информирование человека) состояний зависящего от предыстории модуля. В этом случае эффект выполнения каждой функции (операции), реализуемой этим модулем, следует описывать в терминах этого внешнего представления, что существенно упростит прогнозирование поведения данного модуля.

В C++ единицей компиляции является файл исходных текстов программы с расширением cpp, cxx, c (обычно c – используется в программах на языке C). Для того, чтобы обеспечить взаимодействие между различными модулями используются файлы заголовков (файлы с расширением h, hpp, hxx).

В файлах заголовков обычно описывают пользовательские типы данных, прототипы функций, глобальные константы и переменные, а также глобальные макросы.

Файлы срр содержат реализацию функций, объявленных в соответствующем заголовочном файле, локальные для данного модуля функции, типы данных, переменные и константы.

Заголовочные файлы подключаются директивой препроцессора #iclude, как, например, мы подключали ранее модули стандартной библиотеки. При этом выполнятся следующее правило: если после директивы #include имя файла пишется в кавычках, то компилятор будет искать заголовочный файл сначала в текущем каталоге, а потом в библиотеках, описанных в переменных окружения; если же за #include следует имя файла в треугольных скобках <>, то текущий каталог в списке поиска опускается.

Чтобы исключить повторное подключение заголовочного файла (а это необходимо, поскольку нельзя в одной области видимости дважды определять один и тот же идентификатор) обычно в заголовочном файле используют директивы условной компиляции.

Таким образом, заголовочный файл обычно выглядит следующим образом

#ifndef <макрос>
#define <макрос>
…<определения>…
#endif

Где <макрос> – некоторое макроопределение. В качестве имени макроса может использоваться, в принципе, любой набор букв, цифр и знаков «\_», начинающийся не с цифры. Однако многие компиляторы (в часности, gcc) используют эти макроопределения для индексации файлов заголовков. Поэтому обычно имя макроса содержит в себе имя файла заголовка и имеет вид ИМЯ\_РАСШИРЕНИЕ, например, заголовочный файл stack.h будет иметь вид:

#ifndef STACK\_H
#define STACK\_H
…<определения>…
#endif //STACK\_H

При использовании модулей необходимо учитывать область видимости переменных, констант и функций.

Если мы хотим, чтобы переменная, функция или константа не были доступны в других модулях, то их необходимо определить с ключевым словом **static**.

**static int** x;
**static const int** y;
**static double** avg(**double** x, **double** y)
{
 **return** (x+y)/2;
}

Если мы хотим сообщите компилятору, что переменная, константа или функция будут объявлены в другом модуле, необходимо объявить их с ключеым словом **extern**.

**extern int** x;
**extern const int** y;
**extern double** avg(**double** x, **double** y);

С ключевым словом **extern** может использоваться только прототип функции, но не ее определение.

Определение функции, константы или переменной без ключевого слова означает, что они доступны для использования в других модулях. Прототип функции без ключевого слова подразумевает использование ключевого слова **extern**.

Для переменных и констант, определенных в теле функции, действуют правила локальной области видимости внутри функции. Типы данных, структуры, смеси, классы определенные внутри модуля будут виды в других модулях только если их объявления будут вынесены в заголовочный файл.

В качестве примера можно привести реализацию работы с комплексными числами.

Файл complex.h

#ifndef COMPLEX\_H
#define COMPLEX\_H
**struct** Comlex
{
 **double** image, real;
}
**extern double** abs(**const** Compex& c);
#endif

Файл complex.cpp

#include “complex.h”
#include <cmath>
**using namespace** std;
**double** abs(**const** Compex& c)

{
 **double** res = pow(c.image\* c.image + c.real\*c.real, 0.5f);
 **return** res;
}

## Контрольные вопросы.

1. Зачем нужны заголовочные файлы?
2. Для чего используется директива #include?
3. Для чего используется ключевое слово **extern?**
4. Как определить переменную или функцию, существующую только внутри одного модуля?

## Задания к практической работе 14.

Выполнить лабораторные работы 9 и 10, расположив все функции, кроме main, в отдельном модуле.

# Практическая №15. Создание класса и использование его объектов

## Теоретические сведения

**Класс.**

Класс – фундаментальное понятие С++, он лежит в основе многих свойств С++. Класс предоставляет механизм для создания объектов. В классе отражены важнейшие концепции объектно-ориентированного программирования: инкапсуляция, наследование, полиморфизм.

С точки зрения синтаксиса, класс в С++ – это структурированный тип, образованный на основе уже существующих типов.

В этом смысле класс является расширением понятия структуры. В простейшем случае класс можно определить с помощью конструкции:

типкласса имя\_класса{список\_членов\_класса};

где

типкласса – одно из служебных слов **class, struct, union**;

имя класса – идентификатор;

списокчленов класса – определения и описания типизированных данных и принадлежащих классу функций.

Функции – это методы класса, определяющие операции над объектом.

Данные – это поля объекта, образующие его структуру. Значения полей определяет состояние объекта.

**Примеры.**

struct date // дата
{int month,day,year; // поля: месяц, день, год
void set(int,int,int); // метод – установить дату
void get(int\*,int\*,int\*); // метод – получить дату
void next(); // метод – установить следующую дату
void print(); // метод – вывести дату
};
class complex // комплексное число
{double re,im;
double real(){return(re);}
double imag(){return(im);}
void set(double x,double y){re = x; im = y;}
void print(){cout<< *"re = "*<<re; cout<<*"im = "*<<im;}
};

Для описания объекта класса (экземпляра класса) используется конструкция

имякласса имяобъекта;

date today,my\_birthday;

date \*point = &today; // указатель на объект типа date

date clim[30]; // массив объектов

date &name = mybirthday; // ссылка на объект

В определяемые объекты входят данные, соответствующие членам -данным класса. Функции – члены класса позволяют обрабатывать данные конкретных объектов класса. Обращаться к данным объекта и вызывать функции для объекта можно двумя способами. Первый с помощью «квалифицированных» имен:

имяобъекта.имяданного; имяобъекта.имя\_функции

Например:

complex x1,x2;

xl.re = 1.24;

x1.im = 2.3;

x2.set(5.1,1.7);

x1.print();

Второй способ доступа использует указатель на объект указатель\_на\_объект->имя\_компонента

complex \*point = &x1; // или point = new complex;

point ->re = 1.24;

point ->im = 2.3;

point ->print();

Доступность компонентов класса.

В рассмотренных ранее примерах классов компоненты классов являются общедоступными. В любом месте программы, где «видно» определение класса, можно получить доступ к компонентам объекта класса. Тем самым не выполняется основной принцип абстракции данных – инкапсуляция (сокрытие) данных внутри объекта. Для изменения видимости компонент в определении класса можно использовать спецификаторы доступа: public, private, protected.

Общедоступные (public) компоненты доступны в любой части программы. Они могут использоваться любой функцией как внутри данного класса, так и вне его. Доступ извне осуществляется через имя объекта:

имя\_объекта.имя\_члена\_класса

ссылка\_на\_объект.имя\_члена\_класса

указатель\_на\_объект->имя\_члена\_класса

Собственные (private) компоненты локализованы в классе и не доступны извне. Они могут использоваться функциями – членами данного класса и функциями – «друзьями» того класса, в котором они описаны.

Защищенные (protected) компоненты доступны внутри класса и в производных классах.

Изменить статус доступа к компонентам класса можно и с помощью использования в определении класса ключевого слова class. В этом случае все компоненты класса по умолчанию являются собственными.

Пример.

class complex
{
double re, im; // private по умолчанию
public:
double real(){return re;}
double imag(){return im;}
void set(double x,double y){re = x; im = y;}
};

**Конструктор.**

Недостатком рассмотренных ранее классов является отсутствие автоматической инициализации создаваемых объектов. Для каждого вновь создаваемого объекта необходимо было вызвать функцию типа set (как для класса complex) либо явным образом присваивать значения данным объекта. Однако для инициализации объектов класса в его определение можно явно включить специальную компонентную функцию, называемую конструктором. Формат определения конструктора следующий:

имякласса (список\_форм\_параметров)
{операторы\_тела\_конструктора}

Имя этой компонентной функции по правилам языка С++ должно совпадать с именем класса. Такая функция автоматически вызывается при определении или размещении в памяти с помощью оператора new каждого объекта класса.

Пример.

сomplex(double re1 = 0.0,double im1 = 0.0)
{re = re1; im = im1;}

Конструктор выделяет память для объекта и инициализирует данные – члены класса.

Конструктор имеет ряд особенностей:

* Для конструктора не определяется тип возвращаемого значения. Даже тип void не допустим.
* Указатель на конструктор не может быть определен, и соответственно нельзя получить адрес конструктора.
* Конструкторы не наследуются.
* Конструкторы не могут быть описаны с ключевыми словами virtual, static, const, mutuable, valatile.

Конструктор всегда существует для любого класса, причем, если он не определен явно, он создается автоматически. По умолчанию создается конструктор без параметров и конструктор копирования. Если конструктор описан явно, то конструктор по умолчанию не создается. По умолчанию конструкторы создаются общедоступными (public).

Параметром конструктора не может быть его собственный класс, но может быть ссылка на него (T&). Без явного указания программиста конструктор всегда автоматически вызывается при определении (создании) объекта. В этом случае вызывается конструктор без параметров. Для явного вызова конструктора используются две формы:

имякласса имяобъекта (фактические параметры);

имя класса (фактические параметры);

Первая форма допускается только при не пустом списке фактических параметров. Она предусматривает вызов конструктора при определении нового объекта данного класса:

complex ss (5.9,0.15);

Вторая форма вызова приводит к созданию объекта без имени: complex ss = complex (5.9,0.15);

Существуют два способа инициализации данных объекта с помощью конструктора. Ранее мы рассматривали первый способ, а именно, передача значений параметров в тело конструктора. Второй способ предусматривает применение списка инициализаторов данного класса. Этот список помещается между списком параметров и телом конструктора. Каждый инициализатор списка относится к конкретному компоненту и имеет вид:

имяданного (выражение)

Примеры.

class CLASS\_A
{
int i; float e; char c;
public:
CLASS\_A(int ii,float ee,char cc)
: i(8),e( i \* ee + ii ),с(сс){}/
;

Класс «символьная строка».

*#include <string.h> #include <iostream.h>*class string
{
char \*ch; // указатель на текстовую строку
int len; // длина текстовой строки
public: // конструкторы
// создает объект – пустая строка
string(int N = 80): len(0)
{ch = new char[N+1]; ch[0] = '\0';}
// создает объект по заданной строке
string(const char \*arch)
{len = strlen(arch);ch = new char[len+1];
strcpy(ch,arch);}
// компоненты-функции // возвращает ссылку на длину строки
int& len\_str(void){return len;} // возвращает указатель на строку
char \*str(void){return ch;}
};

Здесь у класса string два конструктора – перегружаемые функции.

По умолчанию создается также конструктор копирования вида T::T(const T&), где Т – имя класса. Конструктор копирования вызывается всякий раз, когда выполняется копирование объектов, принадлежащих классу. В частности он вызывается:

1. когда объект передается функции по значению;
2. при построении временного объекта как возвращаемого значения функции;
3. при использовании объекта для инициализации другого объекта.

Если класс не содержит явным образом определенного конструктора копирования, то при возникновении одной из этих трех ситуаций производится побитовое копирование объекта. Побитовое копирование не во всех случаях является адекватным. Именно для таких случаев и необходимо определить собственный конструктор копирования. Например, в классе string:

string(const string& st)
{
len=strlen(st.len);
ch=new char[len+1];
strcpy(ch,st.ch);
}

Можно создавать массив объектов, однако при этом соответствующий класс должен иметь конструктор по умолчанию (без параметров).

Массив объектов может инициализироваться либо автоматически конструктором по умолчанию, либо явным присваиванием значений каждому элементу массива.

class demo{
int x;
public:
demo(){x=0;}
demo(int i){x=i;}
};
void main()
{
demo a[20]; //вызов конструктора без параметров(по умолчанию)
demo b[2]={demo(10),demo(100) //явное присваивание Деструктор.

};

Динамическое выделение памяти для объекта создает необходимость освобождения этой памяти при уничтожении объекта. Например, если объект формируется как локальный внутри блока, то целесообразно, чтобы при выходе из блока, когда уже объект перестает существовать, выделенная для него память была возвращена. Желательно, чтобы освобождение памяти происходило автоматически. Такую возможность обеспечивает специальный компонент класса – деструктор класса. Его формат:

~имя\_класса(){операторы\_тела\_деструктора}

Имя деструктора совпадает с именем его класса, но предваряется символом "~" (тильда).

Деструктор не имеет параметров и возвращаемого значения. Вызов деструктора выполняется не явно (автоматически), как только объект класса уничтожается.

Например, при выходе за область определения или при вызове оператора delete для указателя на объект. string \*p=new string "строка"); delete p;

Если в классе деструктор не определен явно, то компилятор генерирует деструктор по умолчанию, который просто освобождает память, занятую данными объекта. В тех случаях, когда требуется выполнить освобождение и других объектов памяти, например область, на которую указывает ch в объекте string, необходимо определить деструктор явно: ~string(){delete []ch;}

Так же, как и для конструктора, не может быть определен указатель на деструктор.

Указатели на компоненты-функции. Можно определить указатель на компоненты-функции. тип\_возвр\_значения(имя класса::\*имя указателя\_на\_функцию) (специф\_параметров\_функции);

Пример.

double(complex : :\*ptcom)(); // Определение указателя

ptcom = &complex : : real; // Настройка указателя

// Теперь для объекта А можно вызвать его функцию

complex A(5.2,2.7);

cout<<(A.\*ptcom)();

Можно определить также тип указателя на функцию typedef double&(complex::\*PF)(); а затем определить и сам указатель PF ptcom=&complex::real;

## Контрольные вопросы

1. Дайте определения классу.
2. Дайте определение экземпляру класса.
3. Каковы функции конструктора класса?
4. Каковы функции деструктора класса.
5. Как объявляется конструтор класса в С++?
6. Как объявляется деструтор класса в С++?

## Задания к практической работе № 15

Написать программу, в которой создаются и разрушаются объекты, определенного пользователем класса. Выполнить исследование вызовов конструкторов и деструкторов.

Порядок выполнения работы.

1. Определить пользовательский класс в соответствии с вариантом задания.
2. Определить в классе следующие конструкторы: без параметров, с параметрами, копирования.
3. Определить в классе деструктор.
4. Определить в классе компоненты-функции для просмотра и установки полей данных.
5. Определить указатель на компоненту – функцию.
6. Определить указатель на экземпляр класса.
7. Написать демонстрационную программу, в которой создаются и разрушаются объекты пользовательского класса и каждый вызов конструктора и деструктора сопровождается выдачей соответствующего сообщения (какой объект какой конструктор или деструктор вызвал).
8. Показать в программе использование указателя на объект и указателя на компоненту-функцию.

**Варианты заданий**



# Практическая работа №16. Иерархия объектов и группа. Итераторы

## Теоретические сведения

**Группа.**

Группа – это объект, в который включены другие объекты. Объекты, входящие в группу, называются элементами группы. Элементы группы, в свою очередь, могут быть группой.

Примеры групп:

1. Окно в интерактивной программе, которое владеет такими элементами, как поля ввода и редактирования данных, кнопки, списки выбора, диалоговые окна и т.д
2. Агрегат, состоящий из более мелких узлов.
3. Огород, состоящий из растений, системы полива и плана выращивания.
4. Некая организационная структура (например, ФАКУЛЬТЕТ, КАФЕДРА, СТУДЕНЧЕСКАЯ ГРУППА).

Мы отличаем «группу» от «контейнера». Контейнер используется для хранения других данных. Примеры контейнеров: объекты контейнерных классов библиотеки STL в C++ (массивы, списки, очереди).

В отличие от контейнера мы понимаем группу как класс, который не только хранит объекты других классов, но и обладает собственными свойствами, не вытекающими из свойств его элементов.

Группа дает второй вид иерархии (первый вид – иерархия классов, построенная на основе наследования) – иерархию объектов (иерархию типа целое/часть), построенную на основе агрегации.

Реализовать группу можно несколькими способами:

1. Класс «группа» содержит поля данных объектного типа. Таким образом, объект «группа» в качестве данных содержит либо непосредственно свои элементы, либо указатели на них

class WindowDialog: public Group
{
protected:
InputLine input1; Edit edit1; Button button1;
/\*другие члены класса\*/

2. Группа содержит член-данное last типа TObject\*, который указывает на начало связанного списка объектов, включенных в группу. В этом случае объекты должны иметь поле next типа TObject\*, указывающее на следующий элемент в списке.

3. Создается связанный список структур типа TItem:

struct TItem
{
TObject\* item;
TItem\* next;
};

Поле item указывает на объект, включенный в группу. Группа содержит поле last типа TItem \*, которое указывает на начало связанного списка структур типа TItem.

Если необходим доступ элементов группы к ее полям и методам, объект типа TObject должен иметь поле owner типа TGroup\*, которое указывает на собственника этого элемента.

Методы группы.

Имеется два метода, которые необходимы для функционирования группы:

1. void Insert(TObject\* p); Вставляет элемент в группу.
2. void Show(); Позволяет просмотреть группу.

Кроме этого группа может содержать следующие методы:

1. int Empty();

Показывает, есть ли хотя бы один элемент в группе.

1. TObject\* Delete(TObject\*p);

Удаляет элемент из группы, но сохраняет его в памяти.

1. void DelDisp(TObject\* p);

Удаляет элемент из группы и из памяти.

**Иерархия объектов.**

Иерархия классов есть иерархия по принципу наследования, т.е. типа «это есть разновидность того». Например, «рабочий есть разновидность персоны», «автомобиль» есть разновидность «транспортного средства». В отличие от этого иерархия объектов – это иерархия по принципу вхождения, т.е. типа «это есть часть того». Например, «установка – часть завода», «двигатель» – часть «автомобиля». Таким образом, объекты нижнего уровня иерархии включаются в объекты более высокого уровня, которые являются для них группой

**Итератор.**

Итераторы позволяют выполнять некоторые действия для каждого элемента определенного набора данных.

For all элементов набора { действия}

Такой цикл мог бы быть выполнен для всего набора, например, чтобы напечатать все элементы набора, или мог бы искать некоторый элемент, который удовлетворяет определенному условию, и в этом случае такой цикл может закончиться, как только будет найден требуемый элемент.

Мы будем рассматривать итераторы как специальные методы класса-группы, позволяющие выполнять некоторые действия для всех объектов, включенных в группу. Примером итератора является метод Show.

Нам бы хотелось иметь такой итератор, который позволял бы выполнять над всеми элементами группы действия, заданные не одним из методов объекта, а произвольной функцией пользователя. Такой итератор можно реализовать, если эту функцию передавать ему через указатель на функцию.

Определим тип указателя на функцию следующим образом: typedef void(\*PF)(TObject\*,< дополнительные параметры>); Функция имеет обязательный параметр типа TObject или TObject\*, через который ей передается объект, для которого необходимо выполнить определенные действия.

Метод-итератор объявляется следующим образом:

void TGroup::ForEach(PFaction,< дополнительные параметры >);

где

action – единственный обязательный параметр-указатель на функцию, которая вызывается для каждого элемента группы;

дополнительные параметры – передаваемые вызываемой функции параметры.

Затем определяется указатель на функцию и инициализируется передаваемой итератору функцией.

PF pf=myfunc;

Тогда итератор будет вызываться, например, для дополнительного параметра типа int, так:

gr.ForEach(pf,25); Здесь gr – объект-группа.

**Динамическая идентификация типов.**

Динамическая идентификация типа характерна для языков, в которых поддерживается полиморфизм. В этих языках возможны ситуации, в которых тип объекта на этапе компиляции неизвестен.

В С++ полиморфизм поддерживается через иерархии классов, виртуальные функции и указатели базовых классов. При этом указатель базового класса может использоваться либо для указания на объект базового класса, либо для указания на объект любого класса, производного от этого базового.

Пусть группа содержит объекты различных классов и необходимо выполнить некоторые действия только для объектов определенного класса. Тогда в итераторе мы должны распознавать тип очередного объекта.

В стандарт языка С++ включены средства RTTI (Run-Time Type Idendification) – динамическая идентификация типов. Эти средства реализованы в последних системах Borland C++ (версий 4.0 и выше).

Информацию о типе объекта получают с помощью оператора typeid, определение которого содержит заголовочный файл <typeinfo>.

Имеется две формы оператора typeid:

typeid (объект)

typeid (имятипа)

Оператор typeid возвращает ссылку на объект типа typeinfo.

В классе typeinfo перегруженные операции == и != обеспечивают сравнение типов.

Функция name () возвращает указатель на имя типа.

Имеется одно ограничение. Оператор typeid работает корректно только с объектами, у которых определены виртуальные функции. Большинство объектов имеют виртуальные функции, хотя бы потому, что обычно деструктор является виртуальным для устранения потенциальных проблем с производными классами. Когда оператор typeid применяют к неполиморфному классу (в классе нет виртуальной функции), получают указатель или ссылку базового типа.

Примеры.

1.

#include<iostream.h>
#include<typeinfo.h>
class Base
{
virtual void f(){}; //...
};
class Derived: public Base
{
};
void main()
{
int i;
Base ob,\*p; Derived obi;
cout<<typeid(i).name(); //Выводится int p=&ob1;
cout«typeid(\*p).name(); // Выводится Derived
}

2.

//начало см. выше
void WhatType(Base& ob)
{
cout<< typeid(ob).name()<<endl;
}
void main()
{
Base ob; Derived obi;
WhatType(ob);//Выводится Base WhatType(obi)//Выводится Derived
}

3.

//начало см. выше
void main()
{
Base \*p; Derived ob; p=&ob;
if(typeid(\*p)==typeid(Derived)) cout<<"p указывает на объект типа Derived";
}

Если при обращении typeid(\*p), p=NULL, то возбуждается исключительная ситуация badtypeid

## Контрольные вопросы

1. Дайте определение понятию группа?
2. Что такое итератор?
3. Укажите способы динамической идентификации типов объектов.
4. Чем отличается группа от контейнера?

## Задания к практической работе № 16

Порядок выполнения работы.

1. Дополнить иерархию классов практической работы № 2 классами «группа». Например, для предметной области ФАКУЛЬТЕТ можно предложить классы «факультет», «студенческая группа», «кафедра». Рекомендуется создать абстрактный класс – «подразделение», который будет предком всех групп и абстрактный класс TObject, находящийся во главе всей иерархии.
2. Написать для класса-группы метод-итератор.
3. Написать процедуру или функцию, которая выполняется для всех объектов, входящих в группу (смотри примеры в приложении).
4. Написать демонстрационную программу, в которой создаются, показываются и разрушаются объекты-группы, а также демонстрируется использование итератора.

Варианты запросов.

1. Имена всех лиц мужского (женского) пола.
2. Имена студентов указанного курса.
3. Имена и должность преподавателей указанной кафедры.
4. Имена служащих со стажем не менее заданного.
5. Имена служащих заданной профессии.
6. Имена рабочих заданного цеха.
7. Имена рабочих заданной профессии.
8. Имена студентов, сдавших все (заданный) экзамены на отлично (хорошо и отлично).
9. Имена студентов, не сдавших все (хотя бы один) экзамен.
10. Имена всех монархов на заданном континенте.
11. Наименование всех деталей (узлов), входящих в заданный узел (механизм).
12. Наименование всех книг в библиотеке (магазине), вышедших не ранее указанного года.
13. Названия всех городов заданной области.
14. Наименование всех товаров в заданном отделе магазина.
15. Количество мужчин (женщин).
16. Количество студентов на указанном курсе.
17. Количество служащих со стажем не менее заданного.
18. Количество рабочих заданной профессии.
19. Количество инженеров в заданном подразделении.
20. Количество товара заданного наименования.

#

# Практическая работа №17. Перегрузка операций

## Теоретические сведения

**Абстрактный тип данных (АТД).**

АТД – тип данных, определяемый только через операции, которые могут выполняться над соответствующими объектами безотносительно к способу представления этих объектов.

АТД включает в себя абстракцию как через параметризацию, так и через спецификацию. Абстракция через параметризацию может быть осуществлена так же, как и для процедур (функций); использованием параметров там, где это имеет смысл. Абстракция через спецификацию достигается за счет того, что операции представляются как часть типа.

Для реализации АТД необходимо, во-первых, выбрать представление памяти для объектов и, во-вторых, реализовать операции в терминах выбранного представления.

Примером абстрактного типа данных является класс в языке С++.

**Перегрузка операций.**

Возможность использовать знаки стандартных операций для записи выражений как для встроенных, так и для АТД.

В языке С++ для перегрузки операций используется ключевое слово operator, с помощью которого определяется специальная операция-функция (operator function).

Формат операции-функции:

тип\_возвр\_значения operator знакоперации (
специфпараметров) {
операторы\_тела\_функции
}

Перегрузка унарных операций

* Любая унарная операция $⊕$ может быть определена двумя способами: либо как компонентная функция без параметров, либо как глобальная (возможно дружественная) функция с одним параметром. В первом случае выражение $⊕Z$ означает вызов Z.operator $⊕()$ , во втором – вызов $operator ⊕(Z)$.
* Унарные операции, перегружаемые в рамках определенного класса, могут перегружаться только через нестатическую компонентную функцию без параметров. Вызываемый объект класса автоматически воспринимается как операнд.
* Унарные операции, перегружаемые вне области класса (как глобальные функции), должны иметь один параметр типа класса. Передаваемый через этот параметр объект воспринимается как операнд.

Синтаксис:

1. в первом случае (описание в области класса):
тип\_возвр\_значения operator знак\_операции
2. во втором случае (описание вне области класса):
тип\_возвр\_значения operator знак\_операции(идентификатор\_типа)
3. Примеры.

class person{
int age;
public:
friend void operator++(person&);
void person::operator++(person& ob){++ob.age;}
};
void main() {
person jon;
++jon;
}

class person {
int age;
public:
void operator++(){ ++age;}
};
void main() {
person jon;
 ++jon;
}

**Перегрузка бинарных операций**

* Любая бинарная операция $⊕$ может быть определена двумя способами: либо как компонентная функция с одним параметром, либо как глобальная (возможно дружественная) функция с двумя параметрами. В первом случае x$⊕y$ означает вызов $x.operator⊕(y)$, во втором – вызов $operator ⊕(x,y)$.
* Операции, перегружаемые внутри класса, могут перегружаться только нестатическими компонентными функциями с параметрами. Вызываемый объект класса автоматически воспринимается в качестве первого операнда.
* Операции, перегружаемые вне области класса, должны иметь два операнда, один из которых должен иметь тип класса.

Примеры.

class person{...};
class adresbook{
//содержит в качестве компонентных данных множество объектов типа
//person, представляемых как динамический массив, список или дерево
public:
person& operator[](int); //доступ к i-му объекту
};
person& adresbook::operator[](int . .}
void main(){
adresbook persons;
person record;
record = persons [3];
}

class person{…};
class adresbook{
// содержит в качестве компонентных данных множество объектов типа
//person, представляемых как динамический массив, список или дерево
public:
friend person& operator[](const adresbook&,int);
//доступ к i-му объекту
};
person& operator[](const adresbook& ob ,int {…}
};
void main(){
adresbook persons;
person record;
record = persons [3];
}

Перегрузка операции присваивания Операция отличается тремя особенностями:

* операция не наследуется;
* операция определена по умолчанию для каждого класса в качестве операции поразрядного копирования объекта, стоящего справа от знака операции, в объект, стоящий слева.
* операция может перегружаться только в области определения класса. Это гарантирует, что первым операндом всегда будет леводопус-тимое выражение.

Формат перегруженной операции присваивания:

имя\_класса& operator=( имя\_класса&);

Отметим две важные особенности функции operanor=. Во-первых, в ней используется параметр-ссылка. Это необходимо для предотвращения создания копии объекта, передаваемого через параметр по значению. В случаи создания копии, она удаляется вызовом деструктора при завершении работы функции. Но деструктор освобождает распределенную память, еще необходимую объекту, который является аргументом. Параметр-ссылка помогает решить эту проблему.

Во-вторых, функция operator=() возвращает не объект, а ссылку на него. Смысл этого тот же, что и при использовании параметра-ссылки. Функция возвращает временный объект, который удаляется после завершения ее работы. Это означает, что для временной переменной будет вызван деструктор, который освобождает распределенную память. Но она необходима для присваивания значения объекту. Поэтому, чтобы избежать создания временного объекта, в качестве возвращаемого значения используется ссылка.

## Контрольные вопросы

1. Что такое абстрактный тип данных?
2. Приведите примеры абстрактных типов данных.
3. Каковы синтаксис/семантика «операции-функции»?
4. Как можно вызвать операцию-функцию?
5. Нужно ли перегружать операцию присваивания относительно определенного пользователем типа данных, например класса? Почему?
6. Можно ли изменить приоритет перегруженной операции?
7. Можно ли изменить количество операндов перегруженной операции?
8. Можно ли изменить ассоциативность перегруженной операции?
9. Можно ли, используя дружественную функцию, перегрузить оператор присваивания?
10. Все ли операторы языка С++ могут быть перегружены?
11. Какими двумя разными способами определяются перегруженные операции?
12. Все ли операции можно перегрузить с помощью глобальной дружественной функции?
13. В каких случаях операцию можно перегрузить только глобальной функцией?
14. В каких случаях глобальная операция-функция должна быть дружественной?
15. Обязателен ли в функции operator параметр типа «класс» или «ссылка на класс»?
16. Наследуются ли перегруженные операции?
17. Можно ли повторно перегрузить в производном классе операцию, перегруженную в базовом классе?
18. В чем отличие синтаксиса операции-функции унарной и бинарной операции?
19. Приведите примеры перегрузки операций для стандартных типов.
20. Перегрузите операцию "+" для класса "комплексное число".
21. Перегрузите операции "<",">","==" для класса "строка символов".

## Задания к практической работе 17

Определить и реализовать класс – абстрактный тип данных. Определить и реализовать операции над данными этого класса. Написать и выполнить EasyWin-программу полного тестирования этого класса.

Порядок выполнения работы.

1. Выбрать класс АТД в соответствии с вариантом.

2. Определить и реализовать в классе конструкторы, деструктор, функции Input (ввод с клавиатуры) и Print (вывод на экран), перегрузить операцию присваивания.

3. Написать программу тестирования класса и выполнить тестирование.

4. Дополнить определение класса заданными перегруженными операциями ( в соответствии с вариантом).

5. Реализовать эти операции. Выполнить тестирование.

**Варианты заданий**

1. АТД – множество с элементами типа char. Дополнительно пере-

грузить следующие операции:

+ – добавить элемент в множество(типа char + set);

+ – объединение множеств;

= = – проверка множеств на равенство.

2. АТД – множество с элементами типа char. Дополнительно пере-

грузить следующие операции:

- - удалить элемент из множества (типа set-char); \* – пересечение множеств;

< – сравнение множеств.

3. АТД – множество с элементами типа char. Дополнительно пере-

грузить следующие операции:

- - удалить элемент из множества (типа set-char); > – проверка на подмножество;

!= – проверка множеств на неравенство.

4. АТД – множество с элементами типа char. Дополнительно пере-

грузить следующие операции:

+ – добавить элемент в множество (типа set+char);

\* - пересечение множеств; int()- мощность множества.

5. АТД – множество с элементами типа char. Дополнительно пере-

грузить следующие операции:

() – конструктор множества (в стиле конструктора Паскаля); + – объединение множеств; <= – сравнение множеств .

6. АТД – множество с элементами типа char. Дополнительно пере-

грузить следующие операции:

> – проверка на принадлежность(char in set Паскаля);

\* - пересечение множеств;

< – проверка на подмножество.

7. АТД – однонаправленный список с элементами типа char. Допол-

нительно перегрузить следующие операции:

+ – объединить списки (list+list); -- – удалить элемент из начала (типа --list); = = – проверка на равенство.

8. АТД – однонаправленный список с элементами типа char. Допол-

нительно перегрузить следующие операции:

+ – добавить элемент в начало(char+list); -- – удалить элемент из начала(типа -list); = = – проверка на равенство.

9. АТД – однонаправленный список с элементами типа char. Допол-

нительно перегрузить следующие операции:

+ – добавить элемент в конец (list+char); -- – удалить элемент из конца (типа list--); != – проверка на неравенство.

10. АТД – однонаправленный список с элементами типа char. До-

полнительно перегрузить следующие операции:

[] – доступ к элементу в заданной позиции, например:

int i; char c; list L; c=L[i];

+ – объединить два списка; = = – проверка на равенство.

11. АДТ – однонаправленный список с элементами типа char. До-

полнительно перегрузить следующие операции:

[] – доступ к элементу в заданной позиции, например: int i; char c; list L;

c=L[i];

+ – объединить два списка; != – проверка на неравенство.

12. АДТ – однонаправленный список с элементами типа char. До-

полнительно перегрузить следующие операции:

() – удалить элемент в заданной позиции, например : int i; list L;

L[i];

() – добавить элемент в заданную позицию, например : int i; char c; list L; LM;

!= – проверка на неравенство.

13. АДТ – стек. Дополнительно перегрузить следующие операции:

+ – добавить элемент в стек;

- - извлечь элемент из стека; bool()- проверка, пустой ли стек.

14. АДТ – очередь. Дополнительно перегрузить следующие опера-

ции:

+ – добавить элемент;

- - извлечь элемент;

bool() – проверка, пустая ли очередь.

15. АДТ – одномерный массив (вектор) вещественных чисел. Допол-

нительно перегрузить следующие операции:

+ – сложение векторов (a[i]+b[i] для всех i); [] – доступ по индексу;

+ – добавить число к вектору (double+vector).

16. АТД – одномерный массив (вектор) вещественных чисел. Допол-

нительно перегрузить следующие операции:

- - вычитание векторов (a[i]-b[i] для всех i); [] – доступ по индексу;

- - вычесть из вектора число (vector-double).

17. АТД – одномерный массив (вектор) вещественных чисел. Допол-

нительно перегрузить следующие операции:

\* - умножение векторов (a[i]\*b[i] для всех i); [] – доступ по индексу;

\* - умножить вектор на число (vector\*double).

18. АТД – одномерный массив (вектор) вещественных чисел. Допол-

нительно перегрузить следующие операции:

int() – размер вектора;

() – установить новый размер;

- - вычесть из вектора число (vector-double); [] – доступ по индексу;

19. АТД – одномерный массив (вектор) вещественных чисел. Допол-

нительно перегрузить следующие операции:

= – присвоить всем элементам вектора значение (vector=double);

[] – доступ по индексу;

= = – проверка на равенство;

!= – проверка на неравенство;

20. АТД – двухмерный массив (матрица) вещественных чисел. До-

полнительно перегрузить следующие операции:

() – доступ по индексу;

\* - умножение матриц;

\* - умножение матрицы на число;

\* - умножение числа на матрицу.

# ЛИТЕРАТУРА

Основные источники:

1. Огнева, М. В. Программирование на языке с++: практический курс : учебное пособие для СПО / М. В. Огнева, Е. В. Кудрина. — М. : Издательство Юрайт, 2017. — 335 с. — (Серия : Профессиональное образование). — ISBN 978-5-534-05780-5. — Режим доступа: www.biblio-online.ru/book/B76AB4A4-7623-4842-9136-B6ADC57B90BC.

Дополнительные источники:

1. Лубашева Т.В. Основы алгоритмизации и программирования [Электронный ресурс] : учебное пособие / Т.В. Лубашева, Б.А. Железко. — Электрон. текстовые данные. — Минск: Республиканский институт профессионального образования (РИПО), 2016. — 379 c. — 978-985-503-625-9. — Режим доступа: http://www.iprbookshop.ru/67689.html
2. Основы алгоритмизации и программирования [Электронный ресурс] : лабораторный практикум / . — Электрон. текстовые данные. — Ставрополь: Северо-Кавказский федеральный университет, 2015. — 211 c. — 2227-8397. — Режим доступа: http://www.iprbookshop.ru/63112.html
3. Устинов В.В. Основы алгоритмизации и программирование. Часть 2 [Электронный ресурс] : конспект лекций / В.В. Устинов. — Электрон. текстовые данные. — Новосибирск: Новосибирский государственный технический университет, 2013. — 32 c. — 978-5-7782-2337-0. — Режим доступа: http://www.iprbookshop.ru/44675.html
4. Журналы: Программирование

Интернет-ресурсы:

1. Электронный учебник по С++: http:/cgui.ru
2. Каталог книг по программированию на С, С++: http:/c2p.ru