Министерство образования Российской Федерации

АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Энергетический факультет

А.Н. Рыбалев

ПРОГРАММИРОВАНИЕ И ОСНОВЫ АЛГОРИТМИЗАЦИИ

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Благовещенск

2002

ББК 31.21 я 73
Р 93

Печатается по решению редакционно-издательского совета энергетического факультета Амурского государственного университета

Рыбалев А.Н.

Программирование и основы алгоритмизации. Лабораторный практикум. Учебное пособие. Благовещенск: Амурский гос. ун-т, 2002.

Пособие предназначено для студентов специальности 210200 и других, изучающих дисциплину «Программирование и основы алгоритмизации» и выполняющих по данной дисциплине лабораторные работы. Может быть использовано при выполнении других работ, связанных с программированием на языке C++.

Рецензент: М.В. Исаев, главный специалист группы информационных технологий информационно-технического отдела Амурского регионального отделения Фонда социального страхования РФ.

ВВЕДЕНИЕ

В пособии приводятся краткие теоретические сведения и задания для лабораторных работ по курсу «Программирование и основы алгоритмизации». Предлагаемые работы охватывают следующие темы, изучаемые в данном курсе по языку программирования C++:

базовые типы данных;
пользовательские типы данных: структуры и классы;
динамические классы и перегрузка стандартных операций;
наследование и полиморфизм;

по структурам данных:

стеки и очереди;
связные списки;
деревья;

пример решения практической задачи:

имитационное моделирование.

Лабораторные работы выполняются с использованием любого доступного компилятора языка C++. Требования по оформлению ввода–вывода информации минимальны. Вполне достаточно средств стандартной библиотеки потокового ввода–вывода для платформы DOS.

Приведенные в пособии теоретические сведения не претендуют на полноту. Особенно в вопросах, связанных с языком программирования. Основной упор сделан на тех, без знания которых, по мнению автора, невозможно освоить язык и технику программирования вообще.

Автор заранее приносит извинения за возможные опечатки в текстах фрагментов программ.

1. БАЗОВЫЕ ТИПЫ ДАННЫХ

1.1. Базовые типы данных языка C++

Само понятие «тип данных» является центральным для любого языка программирования. Тип переменной определяет количество памяти, требуемой для ее хранения, интерпретацию содержимого этой памяти и совокупность разрешенных для переменной операций.

Базовые типы данных языка программирования C++ унаследованы им от своего предшественника – языка C. Это целые типы и перечисления (`char`, `int`, `long`, `enum`), вещественные типы (`float`, `double`), указатели и массивы, структуры (`struct`) и объединения (`union`).

Целые типы предназначены для хранения целых чисел. Число разрядов, отводимое для каждого типа, в общем случае зависит от разрядности вычислительной машины. Для 16-разрядной платформы тип `char` занимает 1 байт, `int` – 2 байта и `long int` – 4 байта. Все целые типы имеют две разновидности: знаковые (`signed`) и беззнаковые (`unsigned`).

Знаковые типы используют дополнительный код для представления чисел, поэтому диапазон значений, которые может принимать переменная, составляет от -2^{N-1} до $2^{N-1}-1$, где N – число разрядов, отведенных для типа. Беззнаковые типы интерпретируют содержимое переменной как целое число без знака, поэтому диапазон значений переменных – от 0 до 2^N-1.

Особое положение среди целых типов занимает тип `char`. Обычно он используется для хранения кодов символов в кодировке ASCII. Именно так компилятор обычно и интерпретирует содержимое байта памяти, отведенного для переменной данного типа.

Для целых типов определены (разрешены) все арифметические и логические операторы языка. Во всех логических операциях, за исключением поразрядных, значением целочисленной переменной считается TRUE, если она отлична от нуля, и FALSE в противном случае.

Перечисления – тип, определяющий набор целых констант, обозначаемых произвольными идентификаторами. Например, определение

```
enum DaysOfWeek {Sun, Mon, Tue, Wed, Thurs=10, Fri, Sat};
```

вводит перечислимый тип с именем `DaysOfWeek`, который в дальнейшем может использоваться в определениях и описаниях других объектов.

`DaysOfWeek` задает набор констант, соответствующих дням недели. По существу это обычные целочисленные константы (типа `int`), которым приписаны уникальные и удобные для использования обозначения. Значения констант по умолчанию начинаются с нуля и последовательно увеличиваются на единицу: `Sun = 0`, `Mon = 1` и т.д. Можно изменить этот порядок, указав числовое значение константы, как это сделано с `Thurs`. Последующие константы перечисления «начнут отчет» с нового значения: `Fri = 11`, `Sat = 12`.

Перечисления используются, например, для перегрузки функций. В этом случае несколько функций, имеющих одинаковые имена, будут разли-

чаться типами формальных параметров, в качестве которых используются перечислимые типы. Компилятор, выбирая функцию при вызове, «не ошибется», так как имя фактического параметра явно говорит о его принадлежности к тому или иному перечислимому типу.

Вещественные типы предназначены для хранения дробных чисел. Большинство компьютеров, в том числе ПК, хранит такие числа в формате с плавающей точкой. В персональном компьютере тип `float` занимает 4 байта. Переменные типа `float` могут принимать как положительные, так и отрицательные значения в диапазоне от $3{,}4\times10^{-38}$ до $3{,}4\times10^{38}$ по модулю. Переменные типа `double` занимают 8 байт и могут принимать значения от $1{,}7\times10^{-308}$ до $1{,}7\times10^{308}$ по модулю. Еще большими возможностями обладает тип `long double` (10 байт), однако его применение оправдано лишь в специальных приложениях, требующих очень большой точности.

Указатель – это беззнаковое целое, представляющее адрес памяти. Тип указателя есть тип данных, на которые он ссылается. Указатели используются тогда, когда с помощью одной переменной необходимо адресовать различные объекты памяти одного типа.

Если мы объявляем «обычную» переменную (не указатель), ее имя связывается компилятором с областью памяти, отведенной для переменной. В дальнейшем связать это имя с другой областью памяти невозможно. Имя переменной есть «константный указатель» (ссылка) на переменную. Эта ссылка не получает памяти, она присутствует только в коде программы в качестве относительного или абсолютного адреса, по которому располагается переменная.

С указателем дело обстоит иначе. Указатель – новая переменная с собственным именем и собственным адресом в памяти. Значением этой переменной является адрес другой переменной либо `NULL`, если указатель ни на что не указывает. `NULL` – константа, значение которой зависит от реализации (компилятора), чаще всего она равна нулю.

Значение указателя может быть изменено, тогда указатель будет настроен на новый объект памяти. Настроить его на некоторую переменную можно с помощью операции получения адреса «`&`». Обратиться к переменной, адресуемой указателем, можно посредством операции разыменования «`*`»:

```
int A;
int *p;
p = &a;
*p = 1;
```

В приведенном выше фрагменте объявляется указатель `p` типа `int`. Следует отметить, что «звездочка» в объявлении указателя не знак операции разыменования, – это разделитель, свидетельствующий, что объявляется именно указатель, а не просто переменная. Указатель `p` настраивается на переменную `A` с помощью операции получения адреса. Далее переменной `A`, адресуемой указателем p, присваивается значение, равное 1.

Как число, указатель может использоваться в некоторых арифметических операторах и операторах сравнения. Арифметические операции над ука-

зателем требуют особого внимания. Указатель может быть увеличен или уменьшен на целое число для ссылки на новые данные в памяти. При этом изменение указателя на число n означает изменение его значения на n размеров типа указателя в байтах. Например, в нашем случае указатель `p` имеет тип `int`, размер которого для 16-разрядной машины – 2 байта. Увеличение указателя `p` на 1 означает поэтому увеличение его значения на 2. Таким образом, указатель p будет указывать уже на следующие 2 байта в памяти, т.е. на следующее данное типа `int`.

Часто переменные, адресуемые указателем, не имеют собственного имени. Это касается, например, символов строковых массивов и динамических переменных.

Массив – область данных фиксированной длины, занятая списком однотипных элементов. Именем массива является константный указатель на первый его элемент. Объявление массива имеет примерно такой вид:

```
float M[10];
```

Здесь объявляется массив `M` из 10 элементов типа `float`. Обращение к элементам массива осуществляется посредством их индексов. В C++ нумерация элементов массива начинается с 0. Поэтому в нашем примере M[0] – это первый элемент массива, а M[9] – его последний, десятый, элемент. Такая, на первый взгляд, странная система индексации имеет естественное объяснение. Дело в том, что операция `M[i]` эквивалентна по реализации операции `*(M+i)`.

Работая с массивами, следует помнить следующее. Имя массива есть константа, значение которой – адрес его первого элемента. Изменить это значение невозможно. Вот почему операции вида `M++,M=…` запрещены. Кроме того, необходимо следить за тем, чтобы при обращениях к элементам массива не выйти за диапазон разрешенных индексов, так как компилятор обычно такую проверку не выполняет. Если программа будет использовать индексы, находящиеся вне разрешенного диапазона, может произойти изменение других переменных, что грозит многими неприятными последствиями.

Особое место среди массивов C++ занимают строки. Строки это массивы символов, т.е. элементов типа `char`, с `NULL`-символом в конце. `NULL`-символ – это байт с нулевым значением. Таким образом, строки, в отличие от массивов других типов, несут информацию о своей длине. При объявлении строки компилятор автоматически вычисляет ее длину и размещает строку в памяти, добавляя в конец `NULL`-символ. Строки могут адресоваться как константными указателями, так и указателями-переменными:

```
char S[] = "A string";
char *p = "A string";
```

В данном примере объявлены две одинаковых строки. Обе они занимают по 9 байт (учитывая `NULL`-символ).

Первая строка адресуется константным указателем `S`. Для доступа к символам этой строки необходимо применять операцию обращения по индек-

су. Например, `S[2]` адресует байт с символом «s». Достоинством такой адресации является то, что строка не может быть «потеряна», так как `S` – константа.

Вторая строка адресуется указателем-переменной `p`. Этой переменной выделяется память одновременно с символьным массивом. Первоначально `p` содержит адрес первого байта строки, но впоследствии этот указатель может быть перенастроен на другие данные типа `char`, – возможно, и не принадлежащие данной строке. При этом есть опасность потерять ссылку на память, отведенную под строку. Поэтому часто применяют «комбинированный» способ адресации строк:

```
char S1[] = "A string";
char *p1 = S1;
```

Константа `S1` всегда указывает на начало строки. Указатель `p1`, первоначально также настроенный на начало строки, в дальнейшем используется в вспомогательных целях, – например, для сканирования строки в поисках нужного символа.

Заметим, что функции для работы со строками всегда используют тот факт, что строки оканчиваются нулевым символом, и длина строки всегда может быть вычислена. Функции для работы с массивами других типов требуют дополнительной информации о фактической длине массива. Обычно длина массива передается функции в качестве параметра целого типа.

Структуры – пользовательские типы, объединяющие в качестве компонентов именованные данные различных типов.

C++ имеет ключевое слово `struct` для описания структур, заимствованное из языка C. В C++ возможности структур существенно расширены, и слово `struct` используется для определения особого вида классов, в которых все члены являются по умолчанию открытыми. Мы используем `struct` только для описания «традиционных» структур, т.е. таких, какими они были в языке C. Для определения классов мы будем использовать ключевое слово `class`. Классам посвящена следующая лабораторная работа.

Компоненты структур часто называются полями. Ниже приведен пример определения структурного типа `Student` с двумя полями: целым числом, хранящим идентификационный номер студента, и символьным массивом, в который заносится его фамилия:

```
struct Student
{
    int id;
    char name[30];
};
```

Объявление объектов структурных типов производится аналогично объектам других типов:

```
Student Smith; // объявление объекта Smith типа Student
Student *p; // объявление указателя p на объект типа Student
p = &Smith; // настройка указателя
```

Для структур определены операции обращения к компонентам (полям) по имени объекта «.» и через указатель на объект «->»:

```
Smith.id = 3587;
p->name = "Smith Adam";
```

Объекты структурных типов занимают в памяти объем, равный сумме объемов, необходимых для размещения всех полей структуры. В нашем случае для объекта `Smith` выделяется 32 байта.

Объединения (`union`) в нашей работе не рассматриваются.

Подробнее о базовых типах данных см. [5].

1.2. Функции

Программа на C/C++ представляет собой совокупность глобальных данных и функций. Одна из функций имеет имя `main` (главная), и ей передается управление после загрузки программы в оперативную память.

Функции представляют собой подпрограммы с набором входных и одним выходным параметром. Тип выходного параметра (возвращаемого значения) задает и тип функции.

Определение функции включает заголовок и тело функции, заключенное в фигурные скобки. Ниже приведен пример функции, вычисляющей сумму элементов массива:

```
double sum(double M[], int n)
{
    double s = 0;
    for(int i =0; i<n; i++) s+=M[i];
    return s;
}
```

Заголовок функции состоит из типа возвращаемого значения (`void` – если функция ничего не возвращает), имени функции и списка формальных параметров с обязательным указанием типа каждого из них. Уникальность функции в программе обеспечивается уникальностью «связки» имени функции и списка параметров (вернее, их типов). Поэтому в программе может быть несколько функций с одинаковыми именами, но различными наборами параметров (перегрузка функций). Вызов функции состоит в применении операции «`()`». Операндами служат имя функции и список фактических параметров вызова. При обращении к функции эти параметры замещают формальные параметры функции, причем соблюдается строгое соответствие их по типам.

```
// объявление и инициализация массива
double D[] ={2.3, 4.5, 24.9};
// вычисление длины массива
int n = sizeof(D)/sizeof(double);
// вычисление суммы элементов и вывод ее на экран
cout << sum(D, n);// вызов функции sum
```

До вызова функции в программе обязательно должно быть помещено либо ее полное определение, либо ее описание (прототип). Прототип функции – это просто ее заголовок. В отличие от определения функции в ее прототипе

необязательно (хотя и не запрещено) указывать имена формальных параметров, достаточно указать только их типы. Например, прототип приведенной выше функции мог бы иметь вид:

```
double sum(double [], int );
```

В своей работе функция может оперировать следующими переменными:

параметрами;

локальными переменными, определенными в теле функции (в нашей функции это переменная `s`);

глобальными переменными, определенными вне всяких функций.

Особого внимания заслуживает механизм передачи параметров функции. Существуют два способа передачи параметров подпрограмме: *по значению* и *по ссылке*.

Передача параметров по значению предполагает, что в функцию передается значение переменной (но не сама переменная). Функция оперирует с локальной копией данных, а переменная-«оригинал» остается неизменной. Передача параметра по ссылке означает, что в функцию передается сама переменная, вернее, ее адрес. Следовательно, функция работает с «оригиналом» и может изменить его значение.

В C++ параметры передаются функциям по значению. Исключение составляют массивы. При передаче функции массива фактически передается только адрес первого его элемента. Локальной копии массива в стеке функции не создается. Например, наша функция `sum` работает напрямую с массивом `D`. Если нужно передать массив по значению, следует объявить в теле функции локальный массив и скопировать в него элементы из исходного массива.

Другие переменные, в том числе и структурных типов, передаются только по значению. Например, когда целая переменная `n` передается функции `sum`, в стеке последней создается копия этой переменной, доступная функции также по имени `n`.

В языках C и C++ есть возможность «искусственно» организовать передачу по ссылке, используя такой элемент языка как указатель. Ниже приведены для сравнения два варианта функции `increment`, выполняющей инкрементирование целого числа. В первом случае параметр передается по значению, во втором – через указатель:

```
int increment(int x) { return ++x;}// 1. параметр-значение
void increment(int *x) { (*x)++;} // 2. параметр-указатель
```

Вызов этих функций:

```
int a;
a = increment(a); // передача и возврат значения a
increment(&a);    // передача адреса a
```

В C++, кроме того, появились собственно *ссылки* как особый тип переменных. Ссылка есть константный, автоматически разыменовывающийся указатель. Вариант функции `increment` с параметром типа ссылки имеет вид:

```
void increment(int &a) { a++;}
```

В теле функции работа с параметром, переданным по ссылке, ничем не отличается от работы с параметром, переданным по значению. Необходимо лишь помнить, что мы работаем не с локальной копией переменной, а с ней самой. Вызов такой функции также производится «обычным» способом:

```
increment(a);
```

Передача параметров по ссылке используется не только для того, чтобы функция получила доступ к переменной. Когда параметр функции имеет сложный структурный тип, создание его локальной копии в стеке занимает значительной время, поэтому предпочтительнее передавать параметр по ссылке. Если мы не хотим, чтобы функция изменяла этот параметр, мы можем подстраховаться, поместив перед ним в списке параметров функции ключевое слово `const`:

```
struct BigStrucrure {…};
void Anyfunction(const BigStructure &bs);
```

Функция может также возвращать ссылку на некоторую переменную. В приведенном ниже примере функция возвращает ссылку на максимальный элемент массива:

```
double& maxelem(double M[], int n)
{
    int maxindex = 0;
    for(int i =0; i<n; i++)
         if (M[i]>M[maxindex]) maxindex = i;
    return M[maxindex];
}
```

Вызов такой функции может стоять в левой части операции присваивания. Например, если мы хотим присвоить максимальному элементу массива D значение 100, следует поместить в программу код:

```
maxelem(D,n) = 100;
```

Часть (или все) параметры функции могут иметь так называемые *значения по умолчанию*, т.е. те значения, которые будут использоваться, если при вызове соответствующие параметры опущены:

```
void Anyfun(int a, int b = 24);
```

Параметр `b` функции `Anyfun` по умолчанию принимает значение 24. Поэтому, например, вызов `Anyfun(13)` равносилен вызову `Anyfun(13,24)`.

1.3. Классы памяти переменных

Класс памяти переменной определяет размещение объекта в памяти и продолжительность его существования. Различают *статические*, *автоматические* и *динамические* переменные.

Статические переменные определяются в исходном тексте программы вне всяких функций, а также внутри функций с использованием ключевого

слова `static`. В результате компиляции описание статических переменных заносится в специальную секцию исполнимого файла программы. При загрузке программы в оперативную память статические переменные размещаются в сегменте данных и существуют там все время выполнения программы. Их значения могут быть изменены, но память, отведенная под них, не может быть освобождена. По умолчанию статические переменные инициализируются нулевыми значениями.

Автоматические переменные – переменные, определенные внутри функций или блоков (фрагментов, ограниченных фигурными скобками), кроме тех, что описаны с использованием спецификатора `static`. Эти переменные «рождаются», когда вызывается функция или «открывается» блок, и «умирают», когда функция завершается или блок «закрывается».

Автоматические переменные размещаются в стеке программы и регистрах микропроцессора. Например, при вызове функции выполняется специальный код, который заносит значения передаваемых параметров в стек. Далее по ходу работы функции в стеке выделяется также память для локальных переменных функции. После завершения функции указатель стека принимает значение, которое он имел до ее вызова. Поэтому все локальные переменные функции окажутся «брошенными». Они потеряют свои значения при вызове другой функции, которая будет использовать те же участки стека, но уже для своих переменных.

Автоматические переменные по умолчанию не инициализируются. Их инициализация должна выполняться явно, иначе они будут использовать те значения, которые сохранились в ячейках памяти стека или регистрах микропроцессора после вызова предыдущей функции.

Автоматические переменные не описываются отдельно в исполнимом файле, но присутствуют там в виде кода, который их генерирует.

Динамические переменные – это переменные, создаваемые и уничтожаемые программой в ходе ее выполнения в специальной области данных, которая называется динамической памятью. Динамическая память может по необходимости запрашиваться программой у операционной системы и возвращаться ей. Создание динамических переменных осуществляется в C++ с помощью встроенного оператора `new` (можно использовать также библиотечную функцию `malloc` языка C). Параметром оператора `new` является тип создаваемой переменной, возвращаемым значением – указатель на нее (адрес). При создании динамической переменной она может быть инициализирована (не обязательно). Для этого после имени типа в круглых скобках приводится инициализатор:

```
int *p; // объявление указателя
p = new int(14); //создание динамической переменной и ее
                 //инициализация
```

Таким образом, динамические переменные не имеют имен и адресуются только с помощью указателей.

Удаление (уничтожение) динамических переменных производится с помощью оператора `delete` (можно пользоваться также библиотечной функцией `free` языка С). Параметром оператора является указатель на динамическую переменную:

```
delete p;
```

Необходимо помнить, что повторное применение операции `delete` к тому же указателю запрещено. Также запрещено применять эту операцию к указателю, получившему значение без использования `new`. Однако применение `delete` к нулевому указателю разрешается.

Некоторые особенности имеют создание и удаление динамических массивов. Во-первых, при создании динамического массива необходимо указать его размер:

```
p = new int[10]; //создание массива из 10 целых типа int
```

Во-вторых, инициализация динамических массивов запрещена.

При удалении динамических массивов используется особая форма оператора `delete`:

```
delete [] p; // удаление динамического массива
```

В исполнимом файле динамические переменные присутствуют в виде кодов их генерации и уничтожения.

1.4. Задания

1. Реализуйте и протестируйте функцию

```
void strinsert(char *s, char *t, int i);
```

которая вставляет строку `t` в строку `s` в позиции с индексом `i`. Если `i` больше длины `s`, вставка не выполняется.

2. Реализуйте и протестируйте функцию

```
void strdelete(char *s, int i, int n);
```

которая удаляет последовательность `n` символов из строки `s`, начиная с индекса `i`. Если `i` больше длины `s`, или равен ей, никакие символы не удаляются. Если `i+n` больше длины `s` или равно ей, удаляется конец строки, начиная с индекса `i`.

3. Реализуйте и протестируйте функцию

```
void PStrcat(char *s1, char *s2);
```

– аналог библиотечной функции C++ объединения строк для формата строк языка Pascal. (В Pascal, в отличие от C++, NULL-символ в конце строки отсутствует, длину строки указывает первый ее байт (счетчик).

4. Реализуйте и протестируйте функцию

```
void P2CStr(char *s);
```

выполняющую преобразование строки `s` из Pascal-формата в формат C++.

5. Реализуйте и протестируйте функцию

```
void C2PStr(char *s);
```

выполняющую преобразование строки `s` из формата C++ в формат Pascal.

6. Комплексные числа в программе имеют структурный тип:

```
struct Complex
{
    float real;
    float imag;
};
```

Реализуйте и протестируйте функции, выполняющие операции с комплексными числами:

```
Complex cadd(Complex &x, Complex &y); // x+y
Complex csub(Complex &x, Complex &y); // x-y
Complex cmul(Complex &x, Complex &y); // x*y
Complex cdiv(Complex &x, Complex &y); // x/y
```

7. Рассматривается перечислимый тип:

```
enum DaysOfWeek {Sun, Mon, Tue, Wed, Thurs, Fri, Sat};
```

Реализуйте и протестируйте функцию

```
void GetDay(DaysOfWeek &day);
```

которая читает имя дня с клавиатуры и присваивает параметру day значение элемента перечисления (дни недели: Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday).

8. Реализуйте и протестируйте функцию

```
void PutDay(DaysOfWeek & day);
```

которая выводит строку – имя дня по элементу перечисления `day` (см. пред. задание).

9. Минимальный и максимальный элементы массива описываются структурой:

```
struct ArrMinMax
{
    int MinElem;
    int MaxElem;
};
```

Реализуйте и протестируйте функцию

```
ArrMinMax  FindMinMax(int M[],int n);
```

определяющую минимальный и максимальный элементы массива `M` из `n` элементов и возвращающую структуру типа `ArrMinMax`.

10. Реализуйте и протестируйте функцию

```
int Search(int list[], int start, int n, int key);
```

просматривающую `n` элементов массива `list`, начиная с индекса `start`, и возвращающую индекс первого встретившегося элемента, равного `key`. Если

элемент не найден, возвращается -1. С помощью функции `Search` найдите все вхождения числа 5 в массив

```
int M[] = {1, 2, 4, 5, 8, 9, 5, 3, 5, 1};
```

Результат сформируйте в виде массива индексов, который должен в данном случае содержать элементы: `3, 6, 8`.

11. Реализуйте и протестируйте функцию

```
int str_comp(char *s1, char *s2);
```

сравнивающую посимвольно строки `s1` и `s2` и возвращающую 1, если они равны, и 0 – в противном случае.

12. Квадратная матрица представляется одномерным массивом с последовательной записью строк. Реализуйте и протестируйте функцию

```
int Trace(int M[], int n);
```

вычисляющую след квадратной матрицы размером `n` на `n`, заданной массивом `M` (следом матрицы называется сумма ее диагональных элементов).

13. Реализуйте и протестируйте функцию

```
void MaxArr(int M1[], int M2[], int n);
```

сравнивающую `n` элементов массивов `M1` и `M2` и помещающую большие элементы в массив `M1`.

14. Напишите функцию, подсчитывающую и выводящую на экран счетчики количеств появлений знаков пунктуации «.», «,», «!», «?» в строке.

15. Реализуйте и протестируйте функцию

```
int WordCount(char *s);
```

вычисляющую число слов в строке `s` (слова разделяются пробелами).

16. Реализуйте и протестируйте функцию

```
float Str2Float(char *s, int n);
```

вычисляющую значение вещественного типа `(float)`, соответствующее двоичному представлению числа с фиксированной точкой с `n` разрядами под целую часть, которое содержится в строке `s`.

Например, следующий вызов

```
char *s  = "1011101101";
int n = 5;
cout<< Str2Float(s,n);
```

должен вывести на экран число 23.40625.
($10111.01101 = 2^4+2^2+2^1+2^0+2^{-2}+2^{-3}+2^{-5} = 23.40625$.)

В случае обнаружения недопустимого символа в строке `s` функция должна вывести на экран сообщение об ошибке и вернуть -1.

17. Реализуйте и протестируйте функцию

```
char PopulSymb(char *s);
```

возвращающую наиболее часто встречающийся символ в строке `s`.

18. Реализуйте и протестируйте функцию

```
void ChangeReg(char *s);
```

изменяющую регистр буквенных символов строки s.

19. Реализуйте и протестируйте функцию

```
char* RevStr(char *s);
```

изменяющую порядок размещения символов в строке s на обратный и возвращающую указатель на эту строку. Например, следующий вызов

```
char *s  = "НОС";
cout<< RevStr(s);
```

должен вывести на экран строку «СОН».

20. Реализуйте и протестируйте функцию

```
char* DelSymb(char *s, char symb);
```

удаляющую из строки s все символы symb и возвращающую указатель на эту строку.

2. КЛАССЫ

2.1. Классы как абстрактные типы данных

Класс – пользовательский тип, содержащий в качестве компонентов (членов) данные и функции (методы) для работы с этими данными. Переменная типа класс называется объектом класса.

Как элемент языка программирования классы представляют абстрактные типы данных, т.е. типы, описывающие некие абстракции, – модели реальных объектов и процессов предметной области, с которыми имеет дело компьютерная программа.

Абстрактные типы данных предусматривают определение как организации данных, так и операций по их обработке. В этом состоит их отличие от просто структурных типов, которые описывают только строение объектов, не затрагивая их поведения.

Члены класса имеют различные уровни доступа к ним извне класса: 1) закрытые (private) – доступны только изнутри класса, т.е. только его методам; 2) открытые (public) – доступны как изнутри, так и извне класса любым функциям; 3) защищенные (protected) – доступны методам класса и методам классов-потомков данного класса. Защищенные компоненты используются при наследовании и в данной работе не рассматриваются.

Закрытые и защищенные члены класса составляют его внутреннее содержание/состояние, а открытые – внешнее поведение, или, иначе, интерфейс класса. Таким образом, реализуется одна из основных концепций объектно-ориентированного программирования – инкапсуляция данных.

Инкапсуляция данных подразумевает, что взаимодействие клиентов класса с объектами класса должно осуществляться только посредством специально предназначенных для этого интерфейсных компонентов (чаще всего методов). Клиенты класса не могут «напрямую» изменять состояние объекта.

Вместо этого они обращаются к открытым методам класса, которые обеспечивают корректный доступ к данным объекта и «правильное» его поведение. Защита информации также отличает классы от структур, имеющих только открытые данные.

Синтаксис определения класса рассмотрим с помощью приведенного ниже примера. Пример демонстрирует основные элементы определения, но не описывает никакой абстракции. Ознакомиться с примерами практически значимых классов можно в лаб. работе №3.

```
class ClassName //имя_класса
{
    private:
    //закрытые данные и методы
         int data1;
         char data2;
         static long data3;
         //…
    public:
         //открытые данные и методы
         //конструктор с параметрами по умолчанию
         ClassName(int a = 1, char b = 'a');
         float data4;
         void fun1(void) //внутреннее определение метода
         {
              // тело функции fun1
              //…
         }
         void fun2(void); //прототип метода
         void fun3(void); // прототип метода
         ~ClassName(); //деструктор (прототип)
         //…
};
```

Компонентные данные описываются внутри класса обыкновенным образом (подобно тому, как это делается в структурах).

Методы также обязательно должны быть описаны внутри класса, но их определение может быть как внутренним, так и внешним. В первом случае (см. `fun1`) в определении класса помещается вся реализация функции (т.е. ее тело – код). Компилятор «пытается» оформить такие методы как подставляемые (`inline`) функции, т.е. встроить их код непосредственно в точку вызова. Если это невозможно, выдается соответствующее предупреждение. При внешнем определении метода внутри класса помещается только прототип функции (`fun2`). Само внешнее определение метода обязательно включает указание класса, которому он принадлежит:

```
void ClassName::fun2(void)
{
    // тело функции
    //…
}
```

Среди компонентных функций особое место занимают специальные методы: *конструктор* и *деструктор*. Эти методы необязательны и вводятся программистом только для того, чтобы обеспечить нужное поведение объекта на этапах его создания и уничтожения.

Конструктор обязательно имеет имя, совпадающее с именем класса. Он не имеет возвращаемого параметра (даже типа `void`) и вызывается при создании объектов класса для инициализации их данных и любых других действий. В нашем случае, например, определение конструктора могло быть следующим:

```
ClassName::ClassName(int a, char b)
{
    data1 = a;
    data2 = b;
    //…
}
```

Часто для инициализации данных объекта используется специальная форма конструктора вида:

```
ClassName::ClassName(int a, char b):data1(a), data2(b)
{
    //…
}
```

Обе приведенные формы конструктора равноценны.

Класс может иметь несколько конструкторов, различающихся типами параметров, но только один с умалчиваемыми их значениями.

Деструктор класса имеет имя, начинающееся со знака «~» и заканчивающееся именем класса. Деструктор не имеет принимаемых и возвращаемых параметров (даже типа `void`). Он вызывается неявно при уничтожении объектов класса (статических – по окончании работы программы, автоматических – по выходе из функции или блока, динамических – по освобождении памяти, например, оператором `delete`). Деструктор может выполнять любые задачи, но чаще всего он используется в классах, имеющих в качестве компонентов указатели на динамические переменные. Эти переменные и удаляются деструктором при уничтожении объектов класса.

Создание объектов классов означает выделение им памяти. Для каждого объекта класса выделяется участок памяти, в котором размещаются все компонентные данные, т.е. каждый объект имеет свой экземпляр данных. Исключение составляют так называемые статические компоненты, которые существуют в программе в единственном числе и разделяются всеми объектами класса. Статические компоненты объявляются в классе с использованием ключевого слова `static`. В нашем классе такой компонентой является `data3`.

Код компонентных функций естественно не дублируется для каждого объекта и содержится в сегменте кода в единственном экземпляре. Поэтому на этапе выполнения программы данные объектов никак не связаны с методами. Такая связь существует только в исходном коде. (Это утверждение не касается полиморфных классов, объекты которых на этапе выполнения содержат ссыл-

ки на виртуальные методы. Полиморфизм будет рассматриваться в лабораторной работе №7).

Создание объектов классов отличается от создания переменных структурных типов тем, что оно связано с вызовом конструктора. Если класс не имеет явно определенного конструктора либо имеет конструктор с параметрами по умолчанию, объект класса может быть создан «обычным» образом:

```
ClassName A; // A.data1= 1, A.data2 ='a'
```

Для «явной» инициализации данных объекта конструктору необходимо передать параметры:

```
ClassName B(25, 'd'); // B.data1=25, B.data2 ='d'
```

Доступ к компонентам класса (как к данным, так и к методам) зависит от их «уровня защиты» (`private`, `protected` или `public`). Контроль за соблюдением прав доступа ведется компилятором.

Если компонента доступна, обратиться к ней извне класса можно так же, как к полю структуры:

```
ClassName C, *p; //объявление объекта C и указателя p
p = &C;          // настройка указателя p на объект C
C.data4 = 5;     //обращение к компоненте по имени объекта
C.fun3();        //то же, вызов метода
p->fun3();//обращение к компоненте через указатель на объект
```

Внутри класса любая компонента доступна любому его методу. При этом методы класса обычно работают с компонентами того объекта, для которого они были вызваны (объекта вызова). Например, в приведенном выше фрагменте метод `fun3` дважды вызывается для объекта `C` (первый раз по его имени, второй – через указатель `p`).

Здесь необходимо заметить следующее. Программа может содержать несколько классов, имеющих одноименные методы. В этом случае компилятор, формируя вызов, должен выбрать между ними. Если вызывается обычный, не виртуальный, метод, компилятор поступает просто: выбирается функция того класса, к которому принадлежит объект, для которого вызывается метод, или указатель, если метод вызывается через указатель. Другими словами, выбор метода осуществляется по типу объекта или указателя. Такое связывание кода вызова с кодом метода получило название «раннее связывание», т.е. связывание на этапе компиляции. Противоположностью такому подходу является «позднее связывание», или связывание на этапе выполнения программы, которое характерно для виртуальных методов (см. лаб. раб. №7).

В определении методов класса обращение к компонентам объекта вызова производится только по их именам и не требует указания имени самого объекта:

```
void ClassName::fun3(void)
{
    // обращение к компоненте объекта вызова по ее имени
    data1++;
    //…
```

```
}
```

Возникает естественный вопрос: если на этапе выполнения программы компонентные функции никак не связаны с объектами, каким образом они обращаются к их данным, пользуясь только именами компонентов? Дело в том, что каждому методу без указания программиста, скрытно, передается дополнительный параметр – указатель на объект вызова. Этот указатель имеет фиксированное имя `this`. В нашем случае фактически вызов и определение функции `fun3` выглядят следующим образом.

Вызов:

```
fun3(&C); // функции fun3 передается адрес C
fun3(p); // то же самое
```

Определение:

```
void ClassName::fun3(ClassName *this)
{
    this->data1++;
    //…
}
```

Указатель `this` может использоваться программистом и явным образом. В теле метода он всегда содержит адрес того объекта, для которого этот метод был вызван.

Как уже было сказано, взаимодействие клиентов класса с его объектами осуществляется посредством открытых методов класса. Расширить интерфейс класса позволяют так называемые *друзья* классов. Они объявляются внутри определения класса с помощью ключевого слова `friend`. Друзьями класса могут быть любые функции, в т.ч. методы других классов. Не являясь компонентами класса, дружественные функции все же имеют доступ к его закрытым и защищенным компонентам.

```
class AnyClass
{
    // "свободная" дружественная функция
    friend void freefun(AnyClass);
    // дружественная функция-компонент класса Class1
    friend void Class1::fun(AnyClass);
    // дружественный класс Class2
    friend Class2;
    private:
         int a;
    //…
};
```

Выше приведен фрагмент определения класса `AnyClass`, в котором объявлены его друзья: свободная функция `freefun`, метод `fun` класса `Class1` и класс `Class2`. Объявление другом целого класса равносильно объявлению дружественными всех его методов.

Объявление друзей может быть сделано в любом месте тела класса. Естественно, перед этим в программе должно быть приведено их определение

или, по крайней мере, описание, иначе они будут неизвестны компилятору, и он зафиксирует ошибку.

Особенностью вызова дружественных функций является то, что объекты классов, друзьями которых они являются, должны быть переданы функциям явно, через аппарат параметров (по значению, по ссылке или через указатель). Это объясняется тем, что дружественные функции не получают указателя `this` на эти объекты, так как не являются компонентами их классов.

Следующий фрагмент демонстрирует данную особенность:

```
AnyClass A;
A.freefun(); // ошибка: freefun – не компонент AnyClass
freefun(A); // верный вызов
```

По той же причине в теле дружественной функции обращение к компоненте объекта должно включать «полное» ее имя:

```
void freefun(AnyClass B)
{
    B.a = 25; // обращение к компоненте по имени объекта
    //…
}
```

Подробнее о классах см. [2,5].

2.2.Задания

1. Разработайте, реализуйте и протестируйте класс, описывающий цилиндр. Сделайте возможным изменение радиуса и высоты, вычисления площади поверхности и объема.

2. Разработайте, реализуйте и протестируйте класс `Event`, содержащий данные для нижнего и верхнего пределов времени наступления какого-либо события. Операции: конструктор, переустановка пределов, `GetEvent` (получить время события в заданном диапазоне).

3. Разработайте, реализуйте и протестируйте класс `Coins` для набора из `n` монет. Данные включают: общее количество монет `n`, общее количество лицевых сторон в последнем бросании. Операции: конструктор, получение и изменение `n`, бросание монет, возвращение общего количества лицевых сторон в последнем бросании.

4. Разработайте, реализуйте и протестируйте класс `Box` для коробки. Операции: конструктор, получение и изменение длин сторон, вычисление площади поверхности и объема.

5. Реализуйте методы класса `CardDeck` для колоды карт:

```
class CardDeck
{
    private:
         //колода карт реализуется как массив
         //целых от 0 до 51
         int cards[52];
         //положение текущей карты (индекс в массиве)
         int currentCard;
```

```
    public:
        //конструктор, тасование колоды карт
        CardDeck(void);
        //тасование колоды карт
        void Shuffle(void);
        //возвращение текущей карты (currentCard++)
        int GetCard(void);
        //печать карты (масть и значение)
        //пики 0-12, трефы 13-25, бубны 26-38,
        //черви 39-52, значения карт от 2 до туза.
        void PrintCard(int c);
};
```

СОВЕТ. Для тасования карт используйте цикл со сканированием 52 карт. Для карты `i` выберете случайное число в диапазоне от `i` до 51 и поменяйте местами карту с этим случайным индексом и карту с индексом `i`.

6. Разработайте, реализуйте и протестируйте класс `Calendar`, который содержит элементы данных `year` (год) и логическое значение `leapyear` (високосный или невисокосный год). Его операции следующие:

конструктор;

`unsigned NumDays(unsigned mm,unsigned dd)` – возвращает количество дней с начала года до заданного числа `dd` заданного месяца `mm`;

`int Leapyear(void)` – указывает, является ли год високосным;

`void PrintDate(ndays)` – печатает дату `ndays` (номер дня в году) в формате mm/dd/yy.

7.Реализуйте и протестируйте класс `OrderedList`, описывающий последовательный список целых чисел, представленный в порядке возрастания:

```
class OrderedList
{
    private:
        //массив для хранения списка и текущее число
        //элементов в нем.
        int list[100];
        int size;
    public:
        // конструктор ( size = 0)
        OrderdList(void);
        // возвратить текущее число элементов
        int ListSize(void) const;
        // пуст ли список?
        int ListEmpty(void) const;
        // Найти индекс элемента
        int Find(int &item) const;
        // вернуть элемент по индексу
        int GetElem(int pos) const;
        // вставить элемент
        void Insert(const int &item);
        // удалить элемент
        void Delete(const int &item);
```

```
        // очистить список
        void ClearList(void);
};
```

8. Реализуйте и протестируйте класс `FixedPointNumber`, описывающий представление вещественного числа в формате с фиксированной точкой:

```
class FixedPointNumber
{
    private:
        //знак числа (-1 число отрицательное)
        int signum;
        // целая часть
        unsigned integ;
        // дробная часть
        unsigned ration;
    public:
        // конструктор
        FixedPointNumber(double x);
        // печать числа
        void PrintNumber(void) const;
        // Преобразование в формат double
        double Double(void) const;
};
```

СОВЕТ. При реализации пользуйтесь битовыми операциями (сдвиг, дизъюнкция, конъюнкция) и библиотечными функциями из `math.h`.

9. Реализуйте и протестируйте класс `MovedPoint`, описывающий положение и движение материальной точки в двумерном пространстве под действием постоянной силы. При определении класса используйте структуру `xyValues`:

```
struct xyValues
{
    float x, y;
};
class MovedPoint
{
    private:
        //текущее время, масса точки
        float time, mass;
        // координаты, скорость
        xyValues place, speed;
    public:
        // конструктор (time = 0)
        MovedPoint(xyValues p, float m);
        // Получить координаты
        xyValues GetPlace(void) const;
        // Получить скорости
        xyValues GetSpeed(void) const;
        // Переместить точку под действием постоянной
        // силы в течение заданного промежутка времени
        void Move(xyValues force, float timeInterval);
};
```

10. Реализуйте и протестируйте класс `Triangle`, описывающий треугольник. При определении класса используется структура `ThreeValues`:

```
struct ThreeValues
{
    float v1, v2, v3;
};
class Triangle
{
    private:
         // Длины сторон по порядку: AB,BC CA
         ThreeValues lengths;
         // Углы при вершинах A, B, C
         ThreeValues angles;

    public:
         // конструктор: построение треугольника по трем
         // сторонам
         Triangle(ThreeValues ls = {1, 1, 1});
         // перепостроение треугольника по одной стороне
         //(AB)и двум  прилегающим к ней углам
         Build1(float ab, float anA, float anB);
         // перепостроение треугольника по двум сторонам
         // (AB и AC) и углу между ними
         Build2(float ab, float ac, float anA);
         // Получить длины сторон
         ThreeValues GetLengths(void) const;
         // Получить углы
         ThreeValues GetAngles(void) const;
};
```

11. Реализуйте и протестируйте класс `Calculator`, описывающий простейший калькулятор. При определении класса используется перечислимый тип `Action` (действие): плюс, минус, умножить, разделить, равно.

```
enum Action{plus, minus, multy, div, eqv};
class Calculator
{
    private:
         // Аккамулятор (накопитель)
         double accum;
         // Последнее введенное действие
         Action act;
    public:
         // конструктор (accum=0)
         Calculator(void);
         // ввод числа, возвращает введенное число
         double Input(double num);
         // ввод действия "+", "-", "*", "/", "="
         // возвращает accum
         double Input(char action);
};
```

12. Разработайте, реализуйте и протестируйте класс `MusicBox`, описывающий «музыкальную шкатулку». Данные-члены: массив высот звуков, массив длительностей звуков, число повторений мелодии.

Функции-члены: конструктор; `play()` - проигрывающая мелодию с помощью библиотечных функций `sound()`, `delay()(dos.h)`; `record()` - «записывающая» мелодию.

13. Разработайте и реализуйте класс `Circle`, описывающий окружность. Сделать возможным изменение радиуса, вычисления площади, длины окружности, площади сектора. Используйте класс для решения следующей задачи.

Проектируется круговая игровая площадка с радиусом 100 м. Стоимость ограждения 100 руб/м. Площадка травяная, за исключением бетонного сектора 30 град. Стоимость травяного покрытия 200 руб/кв. м, бетонного покрытия – 100 руб/кв. м. Определить стоимость материалов для строительства площадки.

14. Разработайте, реализуйте и протестируйте класс `Temperature`, содержащий информацию о значениях самой высокой и самой низкой температуры, а также о времени их наблюдения. Методы класса: конструктор (инициализация); `UpdateTemp()` – обновление данных (принимает в качестве параметров текущие значения времени и температуры); `GetHighTemp()` – возвращает максимальную температуру и время ее наблюдения; `GetLowTemp()` – возвращает минимальную температуру и время ее наблюдения.

15. Реализуйте и протестируйте класс `Date`, описывающий некоторую дату:

```
class Date
{
    private:
         int day, month, year;
    public:
         // конструктор, дата по умолчанию 01.01.2001
         Date(int m=1, int d = 1, int y = 2001);
         //конструктор, дата задается в виде
         //строки типа 01.04.1999
         Date(char *dstr);
         // Печать данных в формате "День месяца года"
         //(например, "3 марта 2000 года")
         void PrintDate(void);
};
```

16. Реализуйте и протестируйте класс `Height`, описывающий рост человека в одной из двух систем измерения (сантиметрах или дюймах; один дюйм равен 2,54 см):

```
enum unit{metric, English};
class Height
{
    private:
         char name[20];
         unit measureType;
```

```
        float h; //высота в единицах measureType
    public:
        // конструктор: имя, высота, единицы измерения
        Height(char nm[], float ht, unit m);
        //печать высоты в соответствующих единицах
        void PrintHeight(void);
        //ввод имени, типа измерений и высоты
        void ReadHeight(void);
        // Возвращение высоты
        float GetHeight(void);
        // Преобразование h в измерение m
        void Convert(unit m);
};
```

17. Реализуйте класс `Dice`, описывающий игральные кости:

```
class Dice
{
    private:
        int DiceTotal; // сумма двух костей
        int DiceList[2]; // список очков двух костей
    public:
        // конструктор
        Dice(void);
        // Бросание костей
        void Toss(void);
        // Возвращение суммы
        int Total(void) const;
        // Печать результата бросания
        void DisplayToss(void) const;
};
```

С помощью данного класса смоделируйте 10000 бросаний костей и вычислите частоту появления результатов 2, 3, 4,...,12 в процентах.

18. Разработайте, реализуйте и протестируйте класс, описывающий правильную треугольную пирамиду. Сделайте возможным изменение длин ребер, вычисление площади поверхности и объема.

19. Разработайте, реализуйте и протестируйте класс `BitSymbol`, описывающий «побитовое» представление символа в прямоугольнике размером 10 на 8 пикселов массивом `char [10]`. Операции включают: конструктор `BitSymbol(char[10])`, функции установки и снятия отдельных битов, функцию вывода на экран (в текстовом режиме установленный бит выводить в виде «`*`», снятый – в виде пробела).

20. Разработайте, реализуйте и протестируйте класс `Statistic`, описывающий последовательность результатов каких-либо измерений случайной величины, представленную в виде массива типа `double`. Данными-членами класса должны являться указатель на первый элемент массива (но не сам массив!) и общее число его элементов. Операции: конструктор, вычисление среднего и среднеквадратического отклонения, определение ссылок на максимальный и минимальный элементы массива.

3. СТЕКИ И ОЧЕРЕДИ

3.1. Стеки

Стек (stack) – это список элементов, доступный для добавления и удаления только с одного его конца (вершины). Основные операции стека: добавление элемента (Push) и удаление элемента (Pop) работают с вершиной стека: первая передвигает вершину стека «вперед» (увеличивая тем самым размер списка) и помещает на вершину новый элемент, вторая удаляет элемент из вершины стека и перемещает вершину стека «назад» (уменьшая размер списка). Таким образом, последний вставленный в стек элемент является первым удаляемым элементом. По этой причине говорят, что стек имеет LIFO (last in / first out – последний пришел / первый вышел) порядок сохранения элементов.

Стек является одной из наиболее используемых и важных структур данных. В программных приложениях стеки применяются очень часто, – например, при распознавании синтаксиса в компиляторе, при оценке выражений и т.д. На нижнем уровне стек используется самим микропроцессором (системный стек) для сохранения информации при вызове подпрограмм, передачи параметров подпрограммам и других целей.

В лабораторных работах стек реализуется классом `Stack`:

```
#include <stdlib.h>
class Stack
{
    private:
         // Массив стека и индекс вершины
         SDataType Stacklist[MaxStackSize];
         int top;
    public:
         // конструктор
         Stack(void): top(-1){}
         // Вставить, извлечь из стека, очистить стек
         void Push(const SDataType& item);
         SDataType Pop(void);
         void ClearStack(void) {top = -1;}
         // Считать из стека
         SDataType Peek(void) const;
         // Стек пуст?, полон?
         int StackEmpty(void) const {return top ==-1;}
         int StackFull(void) const
         {return top == MaxStackSize-1;}
};
```

Класс использует массив для хранения списка. Размер массива определяет максимальный размер стека и задается константой `MaxStackSize`, а его элементы имеют некоторый, заранее неизвестный, тип `SDataType`. Определение класса и реализация его методов помещаются в файл `stacklib.h`. Перед включением этого файла в исходный текст программы необходимо определить константу `MaxStackSize` и тип `SDataType`, – например, следующим образом:

```
const int MaxStackSize = 50; // макс. размер стека 50
typedef int SDataType; // элементы стека имеют тип int
#include “stacklib.h” // включаем определение класса
```

Кроме массива элементов, определение класса включает индекс `top` элемента массива – вершины стека. Содержание стека задается элементами массива с индексами от `0` до `top` включительно, остальные элементы не рассматриваются. В этом состоит один из недостатков данной реализации стека: для хранения элементов мы вынуждены резервировать память возможно большего размера, чем может понадобиться. Более «экономичное» решение состоит в использовании связного списка, узлы которого размещаются в динамической памяти (связным спискам посвящена работа № 5).

Компонентные функции класса представляют его интерфейс.

Конструктор устанавливает индекс вершины стека в –1, тем самым подготавливая стек к заполнению.

Функция `Push` вставляет в стек элемент, переданный ей по ссылке. Перед этой операцией она проверяет, не является ли стек полным, а если это так, выдает на экран сообщение об ошибке и завершает программу вызовом библиотечной функции `exit` (прототип которой находится в `stdlib.h`):

```
void Stack::Push(const SDataType& item)
{
    // если стек полон, завершить выполнение программы
    if (top == MaxStackSize-1)
    {
         cerr << "Stack full!" << endl;
         exit(1);
    }
    // увеличить top и копировать item в Staklist
    top++;
    Stacklist[top] = item;
}
```

Метод `Pop` удаляет элемент из стека и возвращает его значение. Попытка удаления из пустого стека приведет к сообщению об ошибке и завершению программы:

```
SDataType Stack::Pop(void)
{
    SDataType temp;
    // если стек пуст, завершить выполнение программы
    if (top ==-1)
    {
         cerr << "Stack Empty!" << endl;
         exit(1);
    }
    // считать элемент с вершины стека
    temp = Stacklist[top];
    // уменьшить top и возвратить результат
    top--;
    return temp;
}
```

Функция Peek позволяет получить доступ к вершине стека без удаления элемента. Ее определение в основном дублирует определение Pop, за исключением того, что индекс top не уменьшается и стек остается нетронутым.

```
SDataType Stack::Peek(void) const
{
    // если стек пуст, завершить выполнение программы
    if (top ==-1)
    {
         cerr << "Stack empty!" << endl;
         exit(1);
    }
    // считать элемент с вершины стека
    return Stacklist[top];
}
```

Методы StackEmpty и StackFull, реализованные внутри определения класса, проверяют, является стек пустым или полным. В первом случае из стека невозможно удалить или считать элемент, и вызов функций Pop или Peek приведет к аварийному завершению программы. Во втором случае в стек невозможно вставить новый элемент, и к аварийному завершению программы приведет вызов метода Push. Поэтому перед каждой операцией доступа к стеку клиент класса должен проверить его состояние вызовом функций StackEmpty или StackFull.

3.2. Очереди

Очередь (Queue) – структура данных, которая сохраняет элементы в списке и обеспечивает доступ к ним только в двух концах списка. Элемент вставляется в конец списка и удаляется из его начала. Иными словами, очередь сохраняет элементы в FIFO (first in/ first out – первым вошел/ первым вышел) порядке.

В лабораторных работах очередь описывается классом Queue:

```
#include <stdlib.h>
class Queue
{
    private:
         // Массив очереди
         QDataType qlist[MaxQSize];
         // Индекс первого элемента, места вставки и
         //общее количество элементов
         int front, rear, count;
    public:
         // конструктор
         Queue(void): front(0), rear(0), count(0){}
         // Вставить, извлечь, очистить очередь
         void QInsert (const QDataType& item);
         QDataType QDelete(void);
         void ClearQueue(void)
         { front = 0; rear = 0; count = 0;}
         // считать первый элемент без удаления
```

```
        QDataType QFront(void) const;
        // Очередь пуста?, полна?
        int QEmpty(void) const {return !count;}
        int QFull(void) const {return count == MaxQSize;}
        // Возвратить длину очереди
        int QLength(void) const {return count;}
};
```

Данными класса являются: массив элементов очереди `qlist` и переменные:

`front` – индекс элемента массива, занятого первым элементом очереди;

`rear` (тыл) – индекс элемента массива, следующего за последним занятым элементом;

`count` – общее число элементов очереди.

Максимальный размер очереди (размер массива `qlist`) задается константой `MaxQSize`. Элементы очереди имеют тип `QDataType`. Перед объявлением класса константа `MaxQSize` и тип `QDataType` должны быть определены в программе (аналогично определению `MaxStackSize` и `SDataType` для стека).

Конструктор класса заполняет компонентные данные `front`, `rear` и `count` нулевыми значениями. Операции класса включают: добавление элемента в очередь (`QInsert`), удаление элемента из очереди (`QDelete`), очистку очереди (`ClearQueue`), считывание первого элемента без его удаления (`QFront`), а также тестирующие методы:

`QEmpty` – проверяет, пуста ли очередь;

`QFull` – проверяет, полна ли очередь;

`QLength` – возвращает длину очереди.

Прежде чем приступить к реализации методов класса, рассмотрим так называемую *круговую модель* очереди, которая будет нами использоваться. Эта модель состоит в том, что элементы очереди организованы логически в окружность (рис.1).

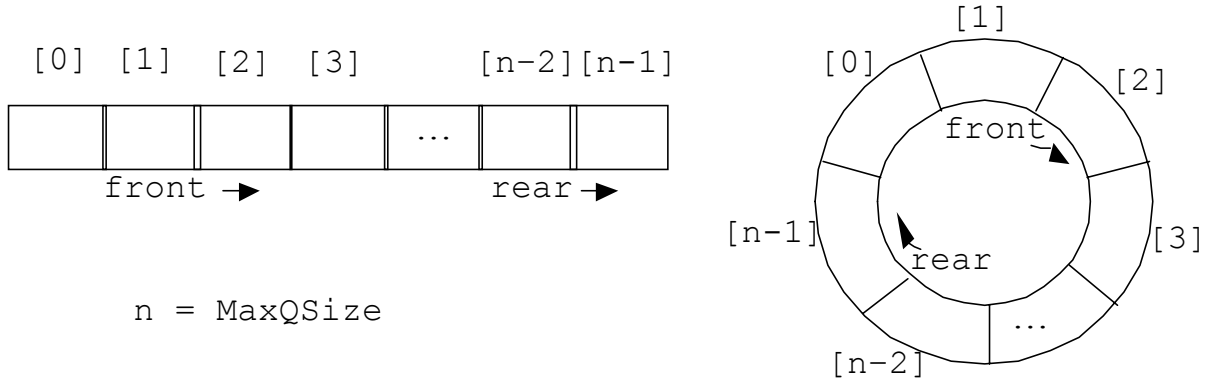


Рис.1. Круговая модель очереди

Переменная `front` всегда является местоположением первого элемента очереди, она продвигается по кругу по мере выполнения удалений. Переменная `rear` является местоположением вставки элемента. После вставки она также продвигается по кругу по часовой стрелке. Переменная `count` поддерживает запись количества элементов в очереди, и если `count == MaxQSize`, очередь заполнена.

Круговое движение реализуется с использованием операции получения остатка от деления:

перемещение конца очереди вперед (добавление элемента):

```
rear = (rear+1)%MaxQSize;
```

перемещение начала очереди вперед (удаление элемента):

```
front = (front+1)%MaxQSize;
```

Методы `QInsert`, `QDelete` и `QFront` используют круговую модель для операций вставки, извлечения и считывания элементов очереди:

```
// Вставить элемент в конец очереди
void Queue::QInsert(const QDataType& item)
{
    // если заполнена, завершить выполнение программы
    if (count == MaxQSize)
    {
         cerr << "Queue full!" << endl;
         exit(1);
    }
    // увеличить count и копировать item в Qlist
    count++;
    qlist[rear] = item;
    rear = (rear+1)%MaxQSize;
}
// Извлечь элемент из начала очереди
QDataType Queue::QDelete(void)
{
    QDataType temp;
    // если пуста, завершить выполнение программы
    if (!count)
    {
         cerr << "Queue empty!" << endl;
         exit(1);
    }
    // Считать значение из начала очереди
    temp = qlist[front];
    // уменьшить count и продвинуть начало очереди
    count--;
    front = (front+1)%MaxQSize;
    // Вернуть результат
    return temp;
}
// Считать первый элемент очереди
QDataType Queue::QFront(void) const
```

```
{
    // если пуста, завершить выполнение программы
    if (!count)
    {
         cerr << "Queue empty!" << endl;
         exit(1);
    }
    // Вернуть результат
    return qlist[front];
}
```

Определение класса `Queue` и его методов поместим в файл `queuelib.h`.

3.3. Очереди приоритетов

Очередь приоритетов (Priority Queue) – структура, поддерживающая извлечение элементов в порядке их приоритета. Операция вставки элемента просто вставляет элемент данных в список, а операция удаления извлекает из списка элемент, имеющий максимальный приоритет. Очереди приоритетов находят применение во многих приложениях, – например, в многозадачной операционной системе, которая записывает процессы в список и затем выполняет их в порядке приоритетов.

При удалении элемента из очереди приоритетов в списке могут находиться несколько элементов с одним и тем же уровнем приоритета. В этом случае мы можем потребовать, чтобы они рассматривались как очередь (т.е. извлекались в порядке их поступления). В нашем определении класса очереди приоритетов мы не будем учитывать это требование, чтобы не усложнять реализацию класса и не снижать эффективность его методов. Оставим данную задачу в качестве варианта задания на лабораторную работу.

Очередь приоритетов реализуется в данной лабораторной работе классом `PQueue`:

```
# include <stdlib.h>
class PQueue
{
    private:
         // Массив очереди
         PQDataType pqlist[MaxPQSize];
         // Общее количество элементов
         int count;
    public:
         // Конструктор
         PQueue(void): count(0){}
         // Вставить, извлечь, очистить очередь
         void PQInsert (const PQDataType& item);
         PQDataType PQDelete(void);
         void ClearPQ(void) {count = 0;}
         // Очередь пуста?, полна?
         int PQEmpty(void) const {return !count;}
         int PQFull(void) const {return count == MaxPQSize;}
         // Возвратить длину очереди
```

```
        int PQLength(void) const {return count;}
};
```

Элементы очереди имеют тип `PQDataType` и размещаются в массиве `pqlist` размером `MaxPQSize`. Тип `PQDataType` и константа `MaxPQSize` должны быть определены в программе до определения класса. Переменная `count` содержит общее количество элементов очереди.

Конструктор класса устанавливает `count=0`. Класс содержит методы вставки (`PQInsert`) и удаления (`PQDelete`) элемента, а также ряд вспомогательных методов для проверки состояния очереди, аналогичных методам класса `Queue`.

Метод `PQInsert` вставляет элемент в конец очереди. При попытке записи в полную очередь выводится сообщение об ошибке, и программа завершается:

```
void PQueue::PQInsert(const PQDataType& item)
{
    // если заполнена, завершить выполнение программы
    if (count == MaxPQSize)
    {
        cerr << "PQueue full!" << endl;
        exit(1);
    }
    // поместить элемент в конец списка
    // и увеличить count на единицу
    pqlist[count] = item;
    count++;
}
```

Метод `PQDelete` удаляет элемент с высшим уровнем приоритета из списка. В нашей реализации мы полагаем, что элемент с высшим приоритетом – это элемент с меньшим значением. Наименьшее значение определяется с использованием оператора «<».

Сначала метод определяет, не является ли список пустым, и завершает программу, если это условие выполняется. Таким образом, перед удалением элемента из очереди приоритетов клиенты класса должны убедиться, что очередь не пуста. Это можно сделать, например, с помощью вызова функции `PQEmpty`.

Далее метод ищет элемент с минимальным значением и удаляет его из списка, заменяя этот элемент последним элементом в списке и уменьшая длину очереди (`count`).

```
PQDataType PQueue::PQDelete(void)
{
    PQDataType min;
    int i, minindex =0;
    if (count > 0)
    {
    // найти минимальное значение
    // и его индекс в массиве
```

```
            min = pqlist[0];
            // просмотреть остальные элементы,
            //изменяя минимум и его индекс
            for (i=1;i<count; i++)
                 if (pqlist[i]< min)
                 {
                      min = pqlist[i];
                      minindex = i;
                 }
            // переместить хвостовой элемент на место
            // минимального и уменьшить на единицу count
            pqlist[minindex] = pqlist[count-1];
            count--;
       }
       else
       // массив pqlist пуст, завершить программу
       {
            cerr << "PQueue empty!" << endl;
            exit(1);
       }
       //возвратить минимальное значение
       return min;
  }
```

Наша реализация метода `PQDelete` предполагает, что операция сравнения «<» определена для типа `PQDataType`. Если в качестве элементов очереди используются объекты типов, для которых по умолчанию данная операция не разрешена, пользователь должен самостоятельно определить ее для используемого типа.

Пусть, например, в качестве данных рассматриваются объекты структурного типа, описывающего сотрудника некоторой фирмы. Полями структуры являются символьный массив, содержащий фамилию сотрудника, и целое, задающее «ранг» его должности согласно принятой в фирме классификации (например, директор – 1, менеджер – 2 и т.д.):

```
struct Person
{
    char name[20];
    unsigned rank;
};
```

Пусть, далее, извлечение записей о сотрудниках из очереди приоритетов должно происходить в порядке, соответствующем должностному положению сотрудника. Иными словами, «сравнивая» сотрудников, мы должны сравнивать ранги их должностей, т.е. поле `rank` должно выступать в качестве *ключа* для сравнения. Чтобы это было возможным, перегрузим операцию «<» для объектов типа `Person` следующим образом:

```
int operator <(Person a, Person b)
{return a.rank < b.rank;}
```

Теперь можно определить тип `PQDataType` как синоним `Person`:

```
typedef Person PQDataType
```

Подробнее перегрузка стандартных операций для объектов структурных типов и классов будет нами рассматриваться в лабораторной работе № 4.

Определение класса `PQueue` и его методов поместим в файл `pqueuelib.h`.

3.4. Задания

1. Реализовать и протестировать функцию:

```
int Ispalindrome(char *s);
```

которая проверяет, является ли строка `s` палиндромом (если да, то возвращается 1, иначе 0), используя стек.

Палиндромом является строка, которая читается одинаково как в прямом, так и в обратном направлениях, например: «кабан упал и лапу на бак». Пробелы не учитываются.

УКАЗАНИЕ. Функция действует следующим образом:

строка `s` копируется в строку `s1` с удалением пробелов;

строка `s1` посимвольно помещается в стек;

символы считываются из стека и сравниваются с символами `s1`. При первом несовпадении – выход из функции и возвращение 0, при полной «очистке» стека – возвращаем 1.

2. Реализуйте и протестируйте функцию:

```
void MultibaseOutput(unsigned num, unsigned base);
```

выводящую на экран представление с основанием `base` целого положительного числа `num`. Функция должна использовать стек. Порядок ее работы иллюстрирует следующий пример.

Исходное число 3553(по основанию 10).Напечатать его с основанием 8.

```
1)  n = 3553, n%8 = 1 -> стек;
2)  n/=8;  n = 444,  n%8 = 4  -> стек;
3)  n/=8; n = 55,   n%8 = 7  -> стек;
4)  n/=8; n = 6,    n%8 = 6 -> стек;
4)  n/=8; n = 0 - условие окончания.
5)  извлекаем данные из стека, выводя их на экран: 6741
```

3. Реализуйте и протестируйте функцию:

```
void Sort2_10(unsigned M[], unsigned n);
```

выполняющую поразрядную сортировку массива `M` двухзначных чисел из `n` элементов с использованием очередей.

Метод поразрядной сортировки состоит в следующем (для двухзначных чисел).

Создается 10 очередей: от 0-й по 9-ю.

В первом проходе элементы массива распределяются по очередям в соответствии со значением младшего разряда (единицы).

Далее элементы последовательно извлекаются из очередей и заносятся опять в массив.

Во втором проходе элементы массива распределяются по очередям в соответствии со значением старшего разряда (десятки).

Далее элементы последовательно извлекаются из очередей и заносятся опять в массив. И он уже упорядочен.

Пример. Исходный массив: 91, 46, 85, 15, 92, 35, 31, 22.

1) распределяем элементы по очередям в соответствии с младшим разрядом:

Очереди

0	1	2	3	4	5	6	7	8	9
	91	92			85	46			
	31	22			15				
					35				

2) считываем в массив: 91, 31, 92, 22, 85, 15, 35, 46;

3) распределяем элементы по очередям в соответствии со старшим разрядом:

Очереди

0	1	2	3	4	5	6	7	8	9
	15	22	31	46				85	91
			35						92

4) считываем в массив: 15, 22, 31, 35, 46, 85, 91, 92 (упорядочен).

4. `SDataType = int`. Реализуйте и протестируйте функцию

```
void SelectItem(Stack &S, int n);
```

которая находит первое появление элемента `n` в стеке `S` и перемещает его на вершину стека. Остальные элементы стека не изменяют свой порядок.

5. Добавьте в класс `Queue` метод, вставляющий элемент в начало очереди:

```
void Queue::QInsertFront(const QDataType &item);
```

Реализуйте и протестируйте метод.

6. Добавьте в класс `Queue` метод, удаляющий заданный элемент из очереди:

```
void Queue::QDelete(const QDataType &item);
```

Реализуйте и протестируйте метод.

7. Добавьте в класс `Queue` метод `Test`, позволяющий получить полное представление о данных-членах класса:

```
void Queue::Test(void);
```

Предположим, `MaxQSize = 6`, элементы очереди: `1, 2, 3, 4, 5`, `front = 3, rear = 2`. Функция должна выводить на экран примерно следующее:

```
index:      0    1    2    3    4    5
qlist:      4    5    88   1    2    3
```

```
queue:                          1    2    3
                4    5
front = 3, rear = 2
```

8. Массив может использоваться для сохранения двух стеков. Первый растет с левого конца (рост индекса), второй – с правого конца (уменьшение индекса).

Реализуйте и протестируйте класс `DualStack`:

```
const int MaxDualStackSize = 100;
class DualStack
{
    private:
         // Массив стеков и индексы вершин
         DSDataType StackStorage[MaxDualStackSize];
         int top1, top2;
    public:
         // конструктор
         DualStack(void);
         // Вставить item в стек n
         void Push(const DSDataType& item, int n);
         // извлечь из стека n,
         DSDataType Pop(int n);
         // очистить стек n
         void ClearStack(int n);
         // считать из стека n
         DSDataType Peek(int n) const;
         // Стек n пуст?
         int StackEmpty(int n) const;
         // Стеки заполнены?
         int StackFull(void)  const;
};
```

9. Реализуйте и протестируйте функцию

```
void StackCopy(Stack &S1, Stack &S2);
```

копирующую содержание стека `S1` в стек `S2` с сохранением порядка элементов. Стек `S1` не должен измениться после вызова функции.

10. Перепишите и протестируйте класс `PQueue` для обеспечения FIFO порядка элементов на одном уровне приоритетов.

11. Реализуйте и протестируйте функцию

```
void QueueCopy(Queue &Q1, Queue &Q2);
```

копирующую содержание очереди `Q1` в очередь `Q2` с сохранением порядка элементов. Очередь `Q1` не должна измениться после вызова функции.

12. В одном немецком городке каждую пятницу вечером организуются танцы. Каждый тур танцев начинается с того, что мужчины выстраиваются в один ряд, а женщины в другой. Когда тур начинается, партнеры по танцу выбираются по одному из каждого ряда. Каждый приходящий на танцы, как истинный немец, заранее знает, сколько туров он намерен провести. Напишите

программу, печатающую имена партнеров по танцам во всех турах. Участники описываются структурами типа:

```
struct Person
{
    char name[];
    char sex; // 'M' или 'F'
    int num; // число туров;
};
```

Программа должна:

ввести 7-8 структур `Person`;

организовать две очереди (мужскую и женскую);

последовательно извлекать пары из очередей, печатать их имена, уменьшать поля `num` структур;

если `num > 0`, вновь помещать человека в очередь;

закончиться, когда одна из очередей окончательно опустеет.

13. Сотрудники компании разделяются по категориям: рабочий, супервайзер, менеджер. Представители всех категорий могут подавать запросы (заявки) на выполнение работ общей секретарской группе компании, которая выполняет работы в очередности, соответствующей приоритетам запроса (наибольший приоритет, естественно, имеют запросы менеджеров, наименьший – рабочих). Приоритет запроса задается перечислимым типом:

```
enum staff{ Manager, Supervisor, Worker};
```

Запрос описывается структурой

```
struct JobRequest
{
    staff StaffPerson; //категория запроса
    int jobid; // ID-номер запроса
    int jobTime;  // время выполнения запроса;
};
```

Для сравнения приоритетов запросов перегружается оператор «<»:

```
int operator<(const JobRequest &a, const JobRequest &b)
{
    return a.stuffPerson < b.staffPerson;
}
```

Напишите программу, вводящую 7-8 запросов (структур `JobRequest`), помещающих их в очередь приоритетов и извлекающую их из нее (выполнение работ). При этом печатается информация о выполняемых заданиях в формате «категория – ID – время». Кроме того, программа должна подсчитать и вывести время, затраченное на выполнения работ по каждой категории запросов:

```
Managers - ?? мин.
Supervisors - ?? мин.
Workers - ?? мин.
```

14. Напишите и протестируйте функцию, удаляющую из стека заданный элемент. Остальные элементы не изменяют свой порядок.

15. Реализуйте и протестируйте функцию

```
int Str2Num(char* &s, double& x);
```

считывающую символьное представление вещественного числа из строки `s`, преобразующую его в значение типа `double` и помещающую в переменную `x`.

По окончании работы функции указатель `s` должен передвинуться «за число». В случае успеха функция возвращает 1, иначе (если `s` указывает, например, на букву) – 0.

Пример:

```
char* s= "6.30-подъем";
double n;
int res;
res = GetNumFromStr(s, n);
```

Результат работы функции: `s` указывает на тире, `n = 6.3, res=1`;

Если вызвать функцию еще раз, `s` и `n` не изменятся, `res = 0`;

При считывании числа используйте стек (для сохранения целой части) и очередь (для сохранения дробной части).

16. Реализуйте и протестируйте функцию

```
double EvalStr(char *s);
```

производящей вычисление формулы, представленной в виде строки `s`.

Для упрощения считаем, что в формулу входят только: 1) целые числа от 0 до 9; 2) знаки сложения и вычитания; 3) скобки.

При реализации функции используйте стек(и).

Логику работы функции рассмотрим на примере.

«Вычислить» строку: "3-((3+4-2)-2)-2"

1) вычисляем 3-;
2) встретили '(', помещаем в стек(и) 3-;
3) встретили '(', помещаем в стек(и) 0+;
4) вычисляем 3+4-2 = 5;
5) встретили ')', извлекаем из стека(ов) 0+;
6) прибавляем 5 и отнимаем 2: 0+5-2 = 3;
7) встретили ')', извлекаем из стека(ов) 3-;
8) отнимаем 3, отнимаем 2: 3-3-2 = -2;
9) возвращаем -2.

17. Реализуйте и протестируйте функцию

```
int StackSize(Stack& s);
```

возвращающую количество элементов в стеке `s` (не изменяя его содержимого).

УКАЗАНИЕ. Функция должна использовать собственный стек.

18. Реализуйте и протестируйте функцию

```
void ReverseString(char *s);
```

изменяющую порядок слов в строке `s` на обратный с использованием стека.

УКАЗАНИЕ. Слова в строке разделяются пробелами, в стеке сохранять указатели на начало каждого слова (`SDataType = char*`).

19. `QDataType = int`. Реализуйте и протестируйте функцию

```
void SelectItem(Queue &Q, int n);
```

которая находит первое появление элемента `n` в очереди `S` и перемещает его в начало очереди. Остальные элементы очереди не изменяют свой порядок.

20. Напишите и протестируйте функцию, сортирующую массив элементов типа int в порядке возрастания элементов с помощью очереди приоритетов.

4. ПЕРЕГРУЗКА СТАНДАРТНЫХ ОПЕРАТОРОВ

4.1. Перегрузка стандартных операторов для объектов классов

Термин «перегрузка операторов» (operator overloading) подразумевает, что операторы языка – такие как «+», «`!=`», «`[]`», «=» могут быть переопределены для типа класса. Язык C++ предоставляет различные методы определения перегрузки операторов, включающие определяемые пользователем внешние функции, функции-члены класса и дружественные функции. При этом требуется, чтобы, по крайней мере, один аргумент перегруженного оператора был объектом класса или ссылкой на объект класса.

Имя функции перегрузки оператора всегда состоит из ключевого слова `operator` и знака операции. Например, для перегрузки операции суммирования необходимо определить функцию с именем `operator+`. Число и состав параметров функции зависят от арности операции (унарная операция, бинарная операция и т.д.) и способа перегрузки (внешняя функция, компонент класса, дружественная классу функция). Тип возвращаемого значения в общем случае может быть любым и определяется потребностью приложения. В табл. 1 показаны примеры перегрузки некоторых операторов языка внешними, в том числе дружественными, функциями и методами класса.

Рассмотрим правила перегрузки операторов, используя табл.1 для демонстрации.

Все унарные операторы, за исключением постфиксных, перегружаются:

внешними, в том числе дружественными классу, функциями с одним параметром, тип которого – класс либо ссылка на класс. Объект-операнд становится фактическим параметром при вызове;

функциями-компонентами класса без параметров. Функция вызывается для объекта-операнда.

Постфиксные унарные операторы перегружаются:

внешними, в том числе дружественными классу, функциями с двумя параметрами. Первый параметр имеет тип класс либо ссылка на класс. Тип второго параметра `int`. Объект-операнд становится первым фактическим параметром при вызове. Второй параметр обычно равен нулю, он никак не используется в теле функции и вводится только для того, что отличить постфиксную форму операции от ее же префиксной формы;

функциями-компонентами класса с одним параметром типа `int`. Функция вызывается для объекта-операнда, параметр при вызове равен нулю и не используется функцией.

Таблица 1)

Перегрузка стандартных операций (`A`, `B` – объекты класса `T`)

Оператор (пример)	Способ перегрузки (вид функции)	Возможные прототипы функции	Вызов функции
Унарные операции			
`-A;`	внешняя (дружественная)	`T operator-(T);` `T operator-(T&);`	`operator-(A);`
`!A`	компонент класса	`T T::operator!();`	`A.operator!();`
`++A;`	внешняя (дружественная)	`T operator++(T);` `T operator++(T&);`	`operator++(A);`
`--A;`	компонент класса	`T T:: operator--();`	`A.operator--();`
`A--;`	внешняя (дружественная)	`T operator--(T,int);` `T operator--(T&,int);`	`operator--(A,0);`
`A++;`	компонент класса	`T T::operator++(int);`	`A.operator++(0);`
Бинарные операции			
`A+B;`	внешняя (дружественная)	`T operator+(T,T);` `T operator+(T&,T&);`	`operator+(A,B);`
`A[3];`	компонент класса	`тип T::operator[](int);`	`A.operator[](3);`
`A(7);`	внешняя (дружественная)	`тип operator()(T,int);` `тип operator()(T&,int);`	`operator()(A,7);`
`A=B;`	компонент класса	`T& T::operator=(T);` `T& T::operator=(T&);`	`A.operator=(B);`
Операция с тремя аргументами			
`A(B,9);`	внешняя (дружественная)	`тип operator()(T,T,int);` `тип operator()(T&,T&,int);`	`operator()(A,B,9);`
`A(B,9);`	компонент класса	`тип T::operator()(T,int);` `тип T::operator()(T&,int);`	`A.operator()(B,9);`

Бинарные операторы перегружаются:

внешними, в том числе дружественными классу, функциями с двумя параметрами. Тип одного из параметров обязательно класс либо ссылка на класс, тип другого параметра может быть любым. Оба операнда становятся фактическими параметрами при вызове;

функциями-компонентами класса с одним параметром любого типа. Крайний слева операнд обязательно должен быть объектом класса, именно для него вызывается функция. Второй операнд может иметь любой тип и передается функции как параметр.

Более двух операндов может иметь операция «`()`» , в стандартном варианте означающая вызов функции. Она перегружается в основном подобно

бинарным операциям. Внешняя функция перегрузки должна иметь число параметров, равное числу операндов, при этом один из параметров обязательно должен быть объектом класса. Если операция перегружается с помощью функции-компонента, крайний слева операнд должен быть объектом класса, именно для него вызывается функция. Остальные операнды передаются как параметры. Следовательно, число параметров функции на единицу меньше числа операндов.

Механизм перегрузки выбирается в каждом конкретном случае в зависимости от приложения.

Перегрузка операторов с помощью компонентных функций осуществляется на этапе разработки класса. При этом мы должны быть уверены в том, что при выполнении бинарных операций крайним левым операндом будет объект класса. В противном случае для перегрузки следует использовать дружественные функции. Внешние недружественные классу функции при перегрузке обладают тем недостатком, что не имеют доступа к закрытым компонентам класса. Поэтому их применение оправданно тогда, когда у нас нет возможности использовать другие варианты (например, в случае перегрузки операторов для объектов библиотечных классов).

При перегрузке операторов следует учитывать, что

1) приоритет операций, ассоциативность и количество операндов, диктуемые встроенным определением оператора, не могут быть изменены;

2) такие операторы, как «`.`» , «`.*`» , «`?:`», «`::`», «`sizeof`», не могут быть перегружены;

3) функции перегрузки не могут иметь аргументов по умолчанию, т.е. все операнды должны присутствовать в выражении.

4.2. Классы и динамическая память

Объекты классов, подобно объектам других типов данных, могут быть динамическими, т.е. размещаться в динамической области данных, создаваться оператором `new` и уничтожаться оператором `delete`. С другой стороны, динамические переменные, а вернее указатели на них, могут являться компонентами классов. Такие структуры данных находят важное применение в приложениях, потребности в памяти которых становятся известными только во время их исполнения. При этом использование динамических структур эффективно снимает ограничения на размеры, характерные для статических переменных. Например, в классе `Stack` максимальный размер стека ограничивается значением `MaxStackSize`, а от клиента требуется выполнение проверки условия заполненности стека перед добавлением нового элемента. Динамическая память повысила бы возможности использования класса `Stack`, поскольку при этом класс запрашивал бы достаточный ресурс памяти для удовлетворения потребностей приложения.

При разработке классов, имеющих динамические компоненты, особое внимание следует уделить таким методам как *конструктор, конструктор копирования, перегруженный оператор присваивания* и *деструктор*.

Конструктор класса должен выделять память под динамические переменные (с помощью оператора `new`). Размер этой памяти обычно передается конструктору через параметры.

Конструктор копирования вызывается всегда, когда создается новый объект, являющийся копией другого объекта. Это имеет место, например, в следующих случаях.

1. Инициализация объекта при его создании:

```
class AnyClass
{
    //…
}
AnyClass A;
// …
AnyClass B=A;
```

В данном примере сначала создается объект `A` класса `AnyClass`, далее, возможно, с ним выполняются какие-то действия, изменяющие его состояние. После этого создается объект `B`, и он инициализируется значениями данных объекта `A`. Для инициализации и вызывается конструктор копирования.

2. При передаче объекта функции в качестве параметра по значению:

```
void AnyFunction(AnyClass ); //прототип функции
AnyClass C;
//…
AnyFunction(C); // вызов функции
```

Объект `C` класса `AnyClass` передается функции `AnyFunction` как параметр по значению. В стеке функции создается локальная копия объекта `C`. Инициализацию этой копии и выполняет конструктор копирования.

Конструктор копирования имеет вид:

```
T::T(const T&);
```

где `T` – имя класса. Единственный параметр конструктора – ссылка на объект того же класса, т.е. на объект, из которого копируются данные.

Конструктор копирования существует для любого класса, причем он может быть создан без указаний программиста (по умолчанию). Конструктор копирования по умолчанию просто копирует данные из одного объекта в другой побитно.

Однако когда класс содержит указатели на динамические переменные, нам необходимо более гибкое поведение конструктора копирования. Действительно, если не принять никаких дополнительных мер, мы получаем копию исходного объекта – объект, данные которого ссылаются на ту же область динамической памяти, что и данные исходного объекта. Это может привести к катастрофическим для программы последствиям, так как в дальнейшем изменение объекта-копии приведет к изменению исходного объекта. Для приведенного выше примера это означает, что после выполнения функции `AnyFunction` изменится объект `C`, чего мы никак не ожидали, передавая этот объект функции по значению. Более того, если класс `AnyClass` имеет дест-

руктор, освобождающий память, занятую динамическими переменными, выход из функции приведет к вызову этого деструктора и уничтожению динамических данных, связанных с объектом C. В дальнейшем попытка вызова деструктора уже для самого объекта C приведет к краху программы, так как повлечет повторное выполнение операции delete для уже уничтоженных переменных.

Решение данной проблемы состоит в явном определении конструктора копирования, который создавал бы копию динамических данных для нового объекта.

Перегруженный оператор присваивания в принципе выполняет ту же задачу, что и конструктор копирования. Отличие состоит в том, что он вызывается не для создаваемого, а для уже существующего объекта, когда ему присваивается значение другого объекта того же класса.

```
AnyClass A, B;
// …
B=A; // вызов оператора присваивания
```

По умолчанию операция присваивания для объектов классов состоит в побитовом копировании данных одного объекта в другой. Для объектов, содержащих динамические данные, такое поведение операции неприемлемо по тем же причинам, что и поведение конструктора копирования по умолчанию. Поэтому оператор присваивания следует перегрузить функцией, обеспечивающей создание копии динамических данных. Обычно функция перегрузки операции присваивания является методом класса. В этом случае она вызывается для объекта класса, стоящего в левой части операции присваивания и имеет один параметр – ссылку на объект класса, стоящий в правой части операции присваивания. Возвращаемым значением является ссылка на объект вызова. Это необходимо, чтобы сделать возможным каскадный вызов типа A=B=C . Прототип функции имеет вид:

```
T& T::operator=(T &);
```

где T – имя класса.

Деструктор класса должен освобождать память, отведенную под динамические данные. В противном случае произойдет так называемая утечка памяти, – ситуация, при которой в динамической области располагаются данные, связь с которыми потеряна.

4.3. Класс Matrix

В данной работе мы реализуем класс Matrix, описывающий прямоугольные матрицы чисел с плавающей точкой двойной точности (тип double). Основными компонентными данными класса будут указатель на динамический массив, в котором хранятся элементы матрицы, а также целочисленные переменные: размеры матрицы по вертикали (число строк) и по горизонтали (число столбцов). Для работы с динамическими данными класс будет содержать полный набор обязательных методов: конструктор, конструктор копирования, перегруженный оператор присваивания и деструктор.

Кроме того, мы определим ряд функций перегрузки стандартных операторов для объектов класса Matrix с целью обеспечить возможность применения этих операторов для матриц, – например, возможность складывать и вычитать матрицы, пользуясь знаками «+» и «–». Большинство из перегруженных операторов будут методами класса.

Определение класса Matrix:

```
#include <iostream.h>
#include <stdlib.h>
class Matrix
{
    private:
         // Размеры матрицы
         unsigned xsize, ysize;
         // Указатель на начало массива
         // в динамической памяти
         double *Array;
         // Флаг "временности"
         int temporary;
    public:
         // Возвратить размеры матрицы
         unsigned getxsize(void) {return xsize;}
         unsigned getysize(void) {return ysize;}
         // конструктор, заполняет матрицу значением v
         Matrix(unsigned y=1, unsigned x=1, double v=0);
         // конструктор копирования
         Matrix(Matrix &);
         // деструктор
         ~Matrix();
         // "Примитивный" вывод
         void show(void);
         // Последовательный доступ к элементу
         double & operator()(unsigned n) const;
         // Остальные методы – в заданиях
         //…
};
```

Среди компонентов класса имеется переменная temporary. Цель ее введения требует пояснений, которые затронут основные принципы реализации класса.

Мы предполагаем, что объекты класса Matrix будут интенсивно использоваться в качестве операндов в различных вычислениях. Рассмотрим следующую последовательность операций, считая, что соответствующие функции их перегрузки реализованы:

```
Matrix A,B,C;
//…
C = A+B;
```

В данном примере создаются три объекта класса Matrix, далее над ними выполняются некие действия, после чего объекту C присваивается зна-

чение суммы объектов A и B. При этом сначала вызывается функция operator+, а затем функция operator=. Обе функции мы сделаем компонентами класса, следовательно, они будут иметь по одному параметру.

Функция operator+ вызывается для объекта A, в качестве параметра ей передается объект B. Как передается этот параметр? Здесь вопрос не вызывает затруднений: чтобы исключить копирование данных и вызов конструктора для локальной копии объекта, очевидно, следует передавать его по ссылке.

Сложнее дело обстоит с возвращаемым параметром функции, которым должен быть объект – результат операции суммирования. Необходимо решить, как этот объект будет сформирован и какой тип будет иметь возвращаемый параметр.

Можно сформировать результат операции как локальный объект функции operator+ и вернуть (скопировать) его значение вызывающей стороне. В данном случае прототип функции будет следующим:

```
Matrix Matrix::operator+(Matrix&);
```

Однако это неэффективный и опасный подход. Во-первых, он требует копирования данных и в ряде случаев двойного вызова конструктора (для локального объекта и его возвращаемой копии). Во-вторых, если не принять никаких мер, при выходе из функции деструктор локального объекта, следуя своему назначению, уничтожит динамический массив, на который должен ссылаться объект – результат операции.

Поэтому выберем другой путь. Объект-результат будет создаваться функцией operator+ в динамической памяти, а возвращаться будет ссылка на этот объект. Таким образом, прототип функции будет иметь вид:

```
Matrix& Matrix::operator+(Matrix&);
```

Однако и такое решение не свободно от недостатков. Вызывающая сторона обязана позаботиться о том, чтобы не потерять ссылку на безымянный динамический объект. В нашем случае функция operator= должна, очевидно, уничтожить объект-параметр после того, как он будет ею использован. Но такое поведение функции не универсально, так как не всегда требуется уничтожение объекта-параметра. Например, если мы присваиваем объекту C значение объекта A (C = A), последний, естественно, не должен (и не может!) быть уничтожен. Уничтожаться должны только «временные» динамические объекты, созданные методами класса.

Таким образом, нам требуется средство, позволяющее перегруженному оператору присваивания и другим методам класса определять «вид» объекта–параметра. С этой целью в определение класса и вводится компонентное данное temporary. Если temporary!=0, объект считается «временным» и должен быть уничтожен после использования.

Рассмотрим компонентные функции класса Matrix.

Конструктор класса инициализирует компонентные данные. В качестве параметров ему передаются размеры матрицы и элемент данных. По умолчанию создается матрица с единственным элементом (1×1), равным 0.

Конструктор выделяет участок динамической памяти нужных размеров и настраивает на него указатель `Array`. В случае, если память выделить не удалось, выводится сообщение об ошибке и программа аварийно завершается.

Все объекты класса `Matrix` создаются как «постоянные» (`temporary=0`). Методы класса при необходимости должны сами изменить эту переменную для «временных» объектов.

```
Matrix::Matrix(unsigned y, unsigned x, double v)
{
    unsigned asize = x*y;
    Array = new double[asize];
    if (!Array)
    {
         cerr << "Memory allocation error";
         exit(1);
    }
    xsize = x;
    ysize = y;
    temporary = 0;
    for(int i = 0; i<asize; i++) Array[i] = v;
}
```

Конструктор копирования создает копию объекта, переданного ему в качестве параметра. Если объект «временный», указатель `Array` копии настраивается на уже имеющийся динамический массив, указатель `Array` «оригинала» обнуляется и сам «оригинал» удаляется. Обнуление указателя объекта-параметра необходимо для корректной работы деструктора (см. далее). Если объект не «временный», в динамической памяти создается массив – копия, на которую настраивается указатель `Array`.

Описанное поведение конструктора копирования продемонстрируем следующим примером:

```
Matrix A,B;
//…
Matrix C = A+B; //1. копирование «временного» объекта
Matrix D = A;   //2. копирование не «временного» объекта
```

В первом случае динамический массив для объекта C «заимствуется» конструктором копирования у «временного» объекта – результата операции суммирования (который далее уничтожается). Во втором случае «оригинал» не «временный», и конструктор копирования вынужден создать новый динамический массив.

```
Matrix::Matrix(Matrix & M)
{
   xsize = M.xsize;
   ysize = M.ysize;
   temporary = 0;
   if (M.temporary)
   {
      Array = M.Array;
      M.Array = NULL;
```

```
      delete &M;
   }
   else
   {
         unsigned asize = xsize*ysize;
         Array = new double[asize];
         if (!Array)
         {
              cerr << "Memory allocation error";
              exit(1);
         }
         for(int i = 0; i<asize; i++) Array[i] = M.Array[i];
   }
}
```

Деструктор класса освобождает память, выделенную под динамический массив, если только этот массив не перешел в пользование другому объекту и указатель `Array` не был обнулен.

```
Matrix::~Matrix()
{
    if (Array) delete []Array;
}
```

Компонентная функция `show` предназначена для организации простейшего вывода матрицы на экран. Если выводится «временный» объект, – например, по вызову `(A+B).show()`, то после вывода он уничтожается.

```
void Matrix::show(void)
{
   cout << '\n';
   for (int i = 0; i<ysize; i++)
   {
         for (int j = 0; j<xsize; j++)
              cout << Array[i*xsize+j]<< '\t';
         cout << '\n';
   }
   if (temporary) delete this;
}
```

Перегруженный оператор «`()`» с одним операндом целого типа позволяет получить доступ к элементу матрицы по его индексу в последовательной системе индексации. При этом элементы нумеруются «обычным» порядком (начиная с 1). При выходе за границу массива формируется сообщение об ошибке, и программа аварийно завершается.

Результат операции – ссылка на элемент. Поэтому с помощью операции можно не только считывать, но и изменять элементы матрицы, присваивая им новые значения, – например, следующим образом:

```
Matrix A;
//…
A(3)=4.5;//присвоить третьему элементу матрицы A  4.5
```

Операция не разрешена для «временных» объектов, так как по определению «временные» объекты должны быть уничтожены после использования, и, следовательно, недопустимо ссылаться на их элементы.

```
double & Matrix::operator()(unsigned n) const
{
   if (n > xsize*ysize)
   {
       cerr << "Index exceeds matrix dimensions!";
       exit(1);
   }
   if (temporary)
   {
       cerr << "Missing operator!";
       exit(1);
   }
   return Array[n-1];
}
```

Остальные методы класса `Matrix`, в том числе перегруженный оператор присваивания, оставлены в качестве заданий.

4.4. Задания

1. Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки оператора «`()`» для доступа к элементу матрицы при параллельной индексации:

```
double & Matrix::operator()(unsigned y, unsigned x) const;
```

Результат – ссылка на элемент. Операция не может быть применена к «временному» объекту. При таком вызове фиксируется ошибка с выходом из программы.

2. Добавить в определение класса `Matrix`, реализовать и протестировать функции перегрузки оператора присваивания матрицы матрице, скаляра матрице, элементов массива матрице (в последнем случае из массива последовательно извлекается `xsize*ysize` элементов):

```
Matrix & Matrix::operator=(Matrix &M);
Matrix & Matrix::operator=(double v);
Matrix & Matrix::operator=(double v[]);
```

Результат – ссылка на матрицу, которой присваивается новое значение. В первой функции операнд объект-матрица уничтожается, если он «временный» (например, в выражении `n = n+m`).

3. Добавить в определение класса `Matrix`, реализовать и протестировать функции перегрузки аддитивных операций:

```
Matrix & Matrix::operator+(const Matrix& M) const;
Matrix & Matrix::operator+(double v) const;
Matrix & Matrix::operator-(const Matrix& M) const;
Matrix & Matrix::operator-(double v) const;
```

Результат – ссылка на новую «временную» матрицу. Операнды объекты-матрицы в первой и четвертой функциях уничтожаются, если они «временные» (например, в выражении `n+(m+n)`).

4. Добавить в определение класса `Matrix`, реализовать и протестировать функции перегрузки операции сложения скаляра с объектом-матрицей как дружественные классу `Matrix`:

```
Matrix & operator+(double v, const Matrix& M);
Matrix & operator-(double v, const Matrix& M);
```

Результат – ссылка на новую «временную» матрицу. Операнд объект-матрица уничтожается, если он «временный» (например, в выражении `4.5+(n+m)`).

5. Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки операции умножения матрицы на матрицу как компонентную функцию класса `Matrix`:

```
Matrix & Matrix::operator*(const Matrix& M) const;
```

Результат – ссылка на новую «временную» матрицу. Операнды объекты-матрицы уничтожаются, если они временные (например, в выражении `(n+3)*(m+n)`).

6. Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки операции умножения матрицы на скаляр как компонентную функцию класса `Matrix`:

```
Matrix & Matrix::operator*(double v) const;
```

Результат – ссылка на новую «временную» матрицу. Операнд объект-матрица уничтожается, если он «временный» (например, в выражении `(n+m)*9`).

Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки операции умножения скаляра на матрицу как дружественную функцию для класса `Matrix`:

```
Matrix & operator*(double v, const Matrix& M);
```

Результат – ссылка на новую «временную» матрицу. Операнд объект-матрица уничтожается, если он «временный» (например, в выражении `6*(n+m)`).

7. Добавить в определение класса `Matrix`, реализовать и протестировать функции перегрузки операций сравнения матрицы с матрицей как компонентные функции класса `Matrix`:

```
Matrix & Matrix::operator==(const Matrix& M) const;
Matrix & Matrix::operator>(const Matrix& M) const;
Matrix & Matrix::operator<(const Matrix& M) const;
```

Результат – ссылка на новую «временную» матрицу, состоящая из нулей и единиц – результатов логических операций над элементами. Операнды

объекты-матрицы уничтожаются, если они «временные» (например, в выражении `(n+3)==(m+n)`).

8. Добавить в определение класса `Matrix`, реализовать и протестировать функции перегрузки операций сравнения матрицы со скаляром как компонентные функции класса `Matrix`:

```
Matrix & Matrix::operator==(double v) const;
Matrix & Matrix::operator>(double v) const;
Matrix & Matrix::operator<(double v) const;
```

Результат – ссылка на новую «временную» матрицу, состоящая из нулей и единиц – результатов логических операций над элементами матрицы и скаляром. Операнд объект-матрица уничтожается, если он «временный» (например, в выражении `(n+m)==8`).

9. Добавить в определение класса `Matrix`, реализовать и протестировать функции перегрузки операций сравнения скаляра с матрицей как дружественные функции для класса `Matrix`:

```
Matrix & operator==(double v, const Matrix& M);
Matrix & operator>(double v, const Matrix& M);
Matrix & operator<(double v, const Matrix& M);
```

Результат – ссылка на новую «временную» матрицу, состоящая из нулей и единиц – результатов логических операций над элементами матрицы и скаляром. Операнд объект-матрица уничтожается, если он «временный» (например, в выражении `3==(n+m)`).

10. Добавить в определение класса `Matrix`, реализовать и протестировать функции перегрузки унарных операций логического отрицания и изменения знака как компонентные функции класса `Matrix`:

```
Matrix & Matrix::operator!() const;
Matrix & Matrix::operator-() const;
```

Результат – ссылка на новую «временную» матрицу. Операнд объект-матрица уничтожается, если он «временный» (например, в выражении `!(n>m)`).

11. Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки операции потокового вывода как дружественную функцию для класса `Matrix`:

```
ostream& operator << (ostream & out, const Matrix & M);
```

Результат – ссылка на поток вывода. Операнд объект-матрица уничтожается, если он «временный» (например, в выражении `cout << (n+m)`).

12. Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки операции потокового ввода как дружественную функцию для класса `Matrix`:

```
istream& operator >> (istream & out, const Matrix & M);
```

Результат – ссылка на поток ввода.

13. Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки унарной операции «~» для вычисления транспонированной матрицы как компонентную функцию класса `Matrix`:

```
Matrix & Matrix::operator~() const;
```

Результат – ссылка на новую «временную» матрицу. Операнд объект-матрица уничтожается, если он «временный» (например, в выражении `~(n+m)`).

14. Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки операции «%» для вычисления объединения матриц по вертикали как компонентную функцию класса `Matrix`:

```
M1 % M2 = [M1]
          [M2]
Matrix & Matrix::operator % (Matrix &) const;
```

Результат – ссылка на новую «временную» матрицу. Операнды объекты-матрицы уничтожаются, если они «временные» (например, в выражении `(n+m)%(n+1)`).

15. Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки операции «||» для вычисления объединения матриц по горизонтали как компонентную функцию класса `Matrix`:

```
M1 || M2 = [M1 M2]
Matrix & Matrix::operator || (Matrix &) const;
```

Результат – ссылка на новую «временную» матрицу. Операнды объекты-матрицы уничтожаются, если они «временные» (например, в выражении `(n+m)||(n+1)`).

16. Используется перечислимый тип:

```
enum ElemStatus {minimal, maximal};
```

Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки операции «[]» для вычисления минимального или максимального элемента матрицы:

```
double & Matrix::operator [](ElemStatus es)  const;
```

Результат – ссылка на этот элемент. Операция будет использоваться в выражениях типа `M[minimal] = 0`. Операция запрещена для «временных» объектов.

17. Используется перечислимый тип:

```
enum Action {sum, prod};
```

Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки операции «[]» для вычисления суммы или произведения элементов матрицы:

```
double & Matrix::operator [](Action act)  const;
```

Операция будет использоваться в выражениях типа `M[sum]`. Операнд объект-матрица уничтожается, если он «временный» (например, в выражении `double x = (n+m)[sum]`).

18. Используется перечислимый тип:

```
enum ConvertType{row, column};
```

Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки операции «`[]`» для преобразования матрицы в матрицу-строку (объединением строк) или в столбец (объединением столбцов):

```
Matrix & Matrix::operator [](ConvertType ct)  const;
```

Операция будет использоваться в выражениях типа `M[row]`. Результат – ссылка на новую «временную» матрицу. Операнд объект-матрица уничтожается, если он «временный» (например, в выражении `(n+m)[row]`).

19. Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки операции «`[]`» для нахождения элемента матрицы ближайшего по значению к заданному числу:

```
double & Matrix::operator [](double v)  const;
```

Операция будет использоваться в выражениях типа `M[1.0]`. Результат – ссылка на этот элемент. Операция не может быть применена к «временному» объекту. При таком вызове фиксируется ошибка с выходом из программы.

20. Добавить в определение класса `Matrix`, реализовать и протестировать функцию перегрузки операции «`()`» для вычисления матрицы, получаемой путем вырезки прямоугольного фрагмента из исходной матрицы:

```
Matrix&  Matrix::operator ()(unsigned y1, unsigned y2,
                             unsigned x1, unsigned x2)  const;
```

Вырезаемый фрагмент в исходной матрице имеет координаты `y1...y2, x1...x2`. Результат – ссылка на новую «временную» матрицу. Операнд объект-матрица уничтожается, если он «временный» (например, в выражении `a =(n+m)(2,3,2,3)`).

5. СВЯЗНЫЕ СПИСКИ

5.1. Структура связных списков

Существует множество приложений, требующих хранения некоторого количества однотипных данных в виде последовательного списка.

Естественный способ организации последовательного списка предоставляет массив. Массив сохраняет элементы в смежных ячейках памяти, что делает возможным прямой и максимально быстрый доступ к элементу по его индексу.

Однако часто списки, основанные на массиве, недостаточно эффективны для приложений, требующих гибких методов обработки данных. Одна из проблем связана с тем, что массив – структура фиксированной длины. Из этого следует, в частности, что список, основанный на массиве, независимо от фак-

тического количества элементов, требует для хранения фиксированной области памяти. Ситуация усугубляется тем, что часто заранее неизвестно максимальное число элементов списка. Поэтому мы вынуждены определять размер массива «с запасом», неэффективно расходуя память, а также принимать меры, которые исключили бы выход за пределы массива в непредвиденных случаях. С этой проблемой нам пришлось столкнуться при разработке классов `Stack`, `Queue` и `PQueue` в лабораторной работе №3.

Другая проблема состоит в том, что массив не позволяет эффективно выполнять добавление и удаление элементов, если только эти операции не производятся в конце списка. Например, удаление первого элемента из списка, размещенного в массиве, приводит к необходимости сдвига влево (копирования) всех оставшихся элементов. Если число элементов велико, такая операция займет значительное время. Для классов `Stack` и `Queue` это не имело значения, так как операции добавления и удаления элементов стека производятся только в конце списка, а при реализации очереди мы использовали круговую модель. Однако для очереди приоритетов отмеченная сложность заставила отказаться от поддержания FIFO-порядка сохранения элементов на одном уровне приоритетов.

В подобных случаях, очевидно, необходима структура, не требующая непрерывного размещения элементов в памяти.

В данной лабораторной работе для реализации последовательного списка будем использовать структуру, называемую связным списком (linked list). Элементы связного списка называются узлами. Каждый узел содержит поле данных `data` и указатель на следующий узел `next` (рис. 2).

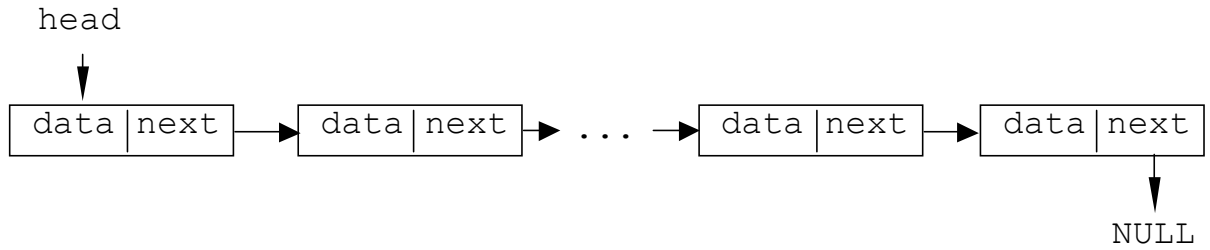


Рис.2. Структура связного списка

Первый элемент списка называется *головой*, и в нашей работе он будет адресоваться указателем `head`. Последний элемент называется *хвостом*. Поле `next` хвоста списка имеет значение `NULL`.

Приложения, работающие с таким списком, последовательно проходят по узлам, используя указатели `next`, начиная с головы, до хвоста, признаком которого является `next = NULL`.

В отличие от массива связный список не требует непрерывной области памяти, так как узлы могут располагаться в ней произвольным образом. Чаще всего узлы связного списка являются динамическими объектами, они создаются по необходимости операцией `new` и уничтожаются операцией `delete`.

Процедуры добавления и удаления узлов связного списка не требуют копирования данных, ограничиваясь только изменением указателей. Вставка

узла в связный список сводится к «настройке» указателей `next` двух узлов: вставляемого и предшествующего вставляемому. Удаление узла состоит лишь в изменении поля `next` узла, предшествующего удаляемому.

5.2. Шаблоны

Разрабатывая функцию или класс, мы стремимся сделать их отвечающими потребностям как можно большего числа приложений. Однако на этом пути сталкиваемся с рядом трудностей. Одна из них – неопределенность типов данных на этапе разработки. Это касается, в частности, и типа данных поля `data` узлов связного списка. В зависимости от приложения связный список может содержать данные как простых типов (например, целые числа), так и сколь угодно сложных (например, структуры, описывающие сотрудников компании). Наше определение класса узлов не должно зависеть от типа поля `data`.

Один из подходов к решению этой проблемы применен в лабораторной работе №3, где для классов `Stack`, `Queue` и `PQueue` мы использовали типы `SDataType`, `QDataType` и `PQDataType`. Перед включением в программу определения классов эти типы обязательно должны быть заданы с помощью `typedef`. К сожалению, у такого подхода есть один существенный недостаток: так как тип данных определяется один раз и не может быть изменен, в программном модуле не может быть нескольких объектов с данными различных типов, – например, стека целых чисел и стека указателей на строки.

Язык C++ предоставляет более мощный механизм для решения данной проблемы – использование шаблонов. Шаблон есть описание множества функций или классов, имеющих родственное строение и поведение, а отличающихся только используемыми или обрабатываемыми типами. Таким образом, шаблон задает целое семейство функций или классов.

Общее описание шаблона имеет вид:

```
template <class тип1, class тип2,..>
// Определение функции или класса
```

Тип1, тип2 – имена типов – параметров шаблона. В определении функции или класса все эти имена должны быть использованы. Описание параметров шаблона действительно только до конца определения функции или класса, т.е. перед каждым шаблоном должно помещаться собственное описание типов, начинающееся с ключевого слова `template`. Различные шаблоны могут использовать одинаковые имена типов.

Приведем, например, шаблон функции, определяющей сумму двух слагаемых:

```
template <class T>
T sum(T a, T b) {return a + b;}
```

В этом определении `T` – имя типа – параметра шаблона. По данному шаблону компилятор формирует функцию, анализируя типы фактических параметров при ее вызове. Например, вызов `sum(2,4)` приведет к формирова-

нию функции с прототипом: `int sum(int,int)`. Программный модуль может содержать сколько угодно функций, формируемых по шаблону и отличающихся только типами параметров.

Шаблоны классов в основном аналогичны шаблонам функций. Рассмотрим, например, следующее определение:

```
template <class T>
class AnyClass
{
    private:
         T AnyData;
         //...
    public:
         AnyClass(T a): AnyData(a){}
         //...
};
```

Это определение шаблона класса с компонентой `AnyData` и конструктором, инициализирующим эту компоненту.

В отличие от объектов обычных классов объявление объекта класса, основанного на шаблоне, должно включать спецификацию типа данных, который используется этим объектом. Спецификация типа приводится в угловых скобках после имени класса. Например, для нашего случая, если объект должен содержать компоненту `AnyData` целого типа, это объявление будет следующим:

```
AnyClass<int> a(5);
```

5.3. Класс `Node`

Приведем определение шаблона класса `Node` для узла связного списка.

```
template <class T>
class Node
{
    private:
         // указатель на адрес следующего узла
         Node<T> *next;
    public:
         T data;
         //конструктор
         Node(const T& item, Node<T> *ptrnext = NULL):
                        data(item), next(ptrnext) {}
         // Вставка следующим
         void InsertAfter(Node<T> *p)
         {
              p->next = next;
              next = p;
         }
         // Удаление следующего
         Node<T> *DeleteAfter(void);
         // Получение адреса следующего
         Node<T> *NextNode(void) const {return next;}
```

```
};
```

Класс содержит закрытый указатель `next` на следующий узел – объект класса `Node`, а также открытое данное `data`. Таким образом, клиенты класса могут напрямую обращаться к данным узлов связного списка, но не могут «самостоятельно» изменять его структуру. Добавление и удаление узлов возможно только с помощью компонентных функций класса. Такая структура класса упрощает построение на основе связного списка различных классов коллекций – стеков, очередей и т.д.

Конструктор класса инициализирует компонентные данные `data` и `next` и может быть вызван с одним или двумя параметрами. Вызов конструктора с одним параметром приводит к созданию узла с полем `next`, равным `NULL`.

Метод `InsertAfter` вставляет узел, указатель на который передан ему в качестве параметра, в связный список непосредственно после текущего узла. Сначала он настраивает указатель `next` вставляемого узла на узел, следующий за текущим, а потом – указатель `next` текущего узла на вставляемый.

Метод `DeleteAfter` удаляет из связного списка узел, следующий за текущим. Он определяется вне класса следующим образом:

```
template <class T>
Node<T>* Node<T>::DeleteAfter(void)
{
    // если нет следующего, возвратить NULL
    if (next==NULL) return NULL;
    // сохранить адрес удаляемого узла
    Node<T> *tempPtr = next;
    // текущий указывает на узел,
    // следующий за удаляемым
    next = tempPtr->next;
    // возвратить указатель на удаляемый узел
    return tempPtr;
}
```

Метод `NextNode` возвращает указатель на следующий узел связного списка. С его помощью клиенты класса получают возможность проходить по списку от узла к узлу.

Для работы со связными списками, в частности, для выполнения лабораторных работ, могут потребоваться также вспомогательные функции, не являющиеся компонентами класса. Минимальный набор таких функций включает функцию создания списка из элементов массива и функцию печати списка. Их определение дается ниже.

```
// Создание связного списка из массива
// возвращает указатель на голову списка
template <class T>
Node<T> *BuildNodeList(T M[], int n)
{
   Node<T> *head=NULL;
   for (int i = n-1; i>=0; i--)
```

```
            head= new Node<T>(M[i],head);
      return head;
}
// Печать связного списка
template <class T>
void PrintNodeList(Node<T> * head)
{
      if (!head) {cout <<"\nList empty"; return;}
      cout << '\n';
      while (head)
      {
         cout << head->data << '\t';
         head = head->NextNode();
      }
}
```

Определение шаблонного класса `Node` и его компонентных функций, а также вспомогательных функций `BuildNodeList` и `PrintNodeList` помещается в файл `nodelib.h`.

5.4. Двусвязные списки. Класс `DNode`

Связный список, состоящий из узлов – объектов класса `Node` допускает только прямое прохождение узлов (от головы к хвосту). В некоторых приложениях требуется, чтобы было возможно и обратное прохождение (от хвоста к голове). В таком случае, очевидно, каждый узел списка должен, кроме указателя на следующий узел (`next`), содержать и указатель на предыдущий узел (`prev`). Такой список называется двусвязным (рис. 3).

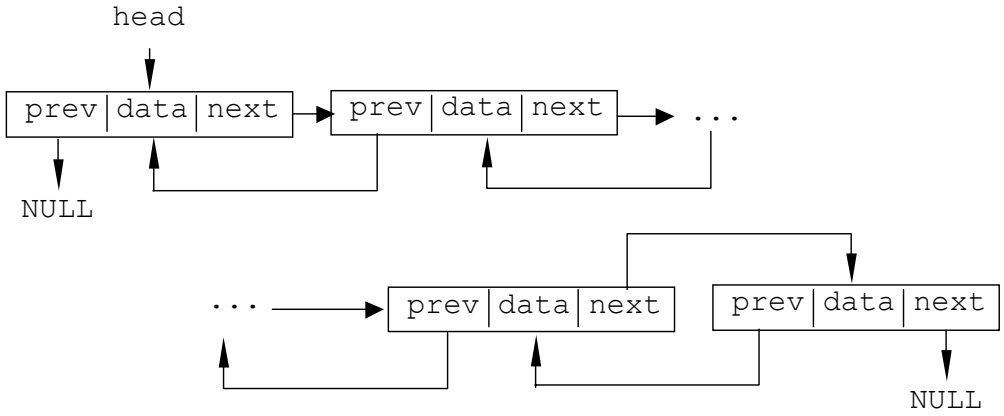


Рис. 3. Структура двусвязного списка

Поле `prev` головы списка имеет значение `NULL`.

В данной работе узлы двусвязного списка описываются классом `DNode`, определение которого приведено ниже.

```
template <class T>
class DNode
{
    private:
         // указатели на предыдущий и следующий узлы
         DNode<T> *prev, *next;
```

```
    public:
         T data;
         //конструктор
         DNode(const T& item, DNode<T> *ptrprev = NULL,
                              DNode<T> *ptrnext = NULL):
               data(item), prev(ptrprev),next(ptrnext)
               {
                  // присоединить текущий узел к предыдущему
                   // и следующему
                   if (prev) prev->next = this;
                   if (next) next->prev = this;
               }
         // Вставка предыдущим и следующим
         void InsertBefor(DNode<T> *p);
         void InsertAfter(DNode<T> *p);
         // Удаление предыдущего и следующего
         DNode<T> *DeleteBefor(void);
         DNode<T> *DeleteAfter(void);
         // Получение адреса предыдущего и следующего
         DNode<T> *PrevNode(void) const {return prev;}
         DNode<T> *NextNode (void) const {return next;}
};
```

Конструктор класса, кроме инициализации компонентных данных созданного объекта, изменяет также указатель `next` предыдущего узла и указатель `prev` следующего узла, настраивая их на этот объект, и, таким образом, полностью встраивает вновь созданный узел в двусвязный список. Методы `PrevNode` и `NextNode` возвращают указатели на предыдущий и следующий узлы, обеспечивая возможность прохождения списка в обоих направлениях. Методы вставки и удаления узлов оставлены для реализации в качестве заданий на лабораторную работу.

Определение класса `DNode` помещается в файл `dnodelib.h`.

5.5. Задания

1. Реализуйте и протестируйте функцию:

```
template <class T>
int CountKey(const Node<T> *head, T key);
```

подсчитывающую количество вхождений ключа `key` в связный список объектов класса `Node`, начиная головы `head`.

2. Реализуйте и протестируйте функцию:

```
template <class T>
void DeleteKey(Node<T> *head, T key);
```

удаляющую из связного списка объектов класса `Node` все вхождения ключа `key`, начиная с головы `head`.

3. Реализуйте и протестируйте функцию:

```
template <class T>
Node<T> *GetNode(const T &item, Node<T> *nextPtr = NULL);
```

создающую в динамической памяти узел-объект класса Node с данным item и указателем на следующий узел nextPtr.

Реализуйте и протестируйте функцию:

```
template <class T>
void InsertFront(Node<T>* &head, const T & item);
```

создающую узел-объект класса Node с данным item и вставляющую его в начало связного списка, на которое указывает head. Функция может вызывать GetNode.

Реализуйте и протестируйте функцию:

```
template <class T>
void InsertRear(Node<T>* &head, const T &item);
```

создающую узел-объект класса Node с данным item и вставляющую его в конец связного списка с указателем на голову head. Функция может вызывать GetNode, InsertFront.

4. Реализуйте и протестируйте функцию:

```
template <class T>
void DeleteFront(Node<T>* &head);
```

удаляющую первый узел из связного списка, на который указывает head.

Реализуйте и протестируйте функцию:

```
template <class T>
void DeleteRear(Node<T>* &head);
```

удаляющую последний узел из связного списка, на начало которого указывает head.

5. Реализуйте и протестируйте функцию:

```
template <class T>
Node<T> * Delete(Node<T> * &head, T key);
```

удаляющую первый узел с вхождением ключа key из связного списка с указателем на голову head и возвращающую указатель на удаленный узел.

6. Реализуйте и протестируйте функцию:

```
template <class T>
void InsertOrder(Node<T> * &head, T item);
```

создающую узел-объект класса Node с данным item и вставляющую его в связный список с указателем на голову head с поддержанием порядка узлов (по возрастанию data).

7. Разработайте и реализуйте класс Stack, использующий связный список объектов Node для хранения элементов стека.

8. Разработайте и реализуйте класс Queue, использующий связный список объектов Node для хранения элементов очереди.

9. Разработайте и реализуйте класс PQueue, использующий связный список объектов Node для хранения элементов очереди приоритетов.

10. Реализуйте и протестируйте функцию:

```
template <class T>
Node <T>* Copy(Node<T> * head);
```

создающую в динамической памяти новый связный список – копию списка, на начало которого указывает параметр `head`.

11. Реализуйте и протестируйте функцию:

```
template <class T>
void Head2Rear(Node<T> * &head);
```

перемещающую первый элемент связного списка в его конец.

12. Используя класс `Node`, создайте связный список из узлов, полем данных в каждом из которых является структура `Emplyee`:

```
struct Emplyee
{
    char name[20]; // Имя служащего
    int idnumber;  // id-номер
    float hourlypay;    // почасовая оплата
}
```

Реализуйте и протестируйте функции:

перегрузки операции вывода объекта типа `Emplyee` на экран

```
ostream& operator<<(ostream& out, Emplyee&);
```

перегрузки операции ввода объекта типа `Emplyee` с клавиатуры

```
istream& operator>>(istream& in, Emplyee&);
```

С помощью данных функций в основной программе организуете ввод информации о 3-4 сотрудниках, помещение ее в связный список и вывод всего списка.

13. Реализуйте и протестируйте функцию:

```
template <class T>
Node<T> * cat(Node<T> * head1, Node<T> *head2);
```

присоединяющую связный список с указателем на голову `head2` к концу связного списка с указателем на голову `head1` и возвращающую указатель на голову получившегося списка.

14. Реализуйте и протестируйте функцию:

```
template <class T>
void Rear2Head(Node<T> * &head);
```

перемещающую последний элемент связного списка в его начало.

15. Реализуйте и протестируйте функцию:

```
template <class T>
void reverse(Node<T> * &head);
```

изменяющую порядок размещения узлов в связном списке на обратный (от конца к началу).

16. Реализуйте и протестируйте функцию:

```
template <class T>
Node<T> *BuildNodeList1(T M[], int n);
```

создающую связный список объектов класса `Node` при помощи компонентной функции `InsertAfter`.

17. Реализуйте и протестируйте функции-компоненты класса `DNode`:

```
template <class T>
void DNode<T>::InsertAfter(DNode<T> *p);
```

вставляющую узел в двусвязный список после данного узла,

```
template <class T>
DNode<T>* DNode<T>::DeleteAfter(void);
```

удаляющую узел, следующий за данным узлом, из двусвязного списка.

18. Реализуйте и протестируйте функции-компоненты класса DNode:

```
template <class T>
void DNode<T>::InsertBefor(DNode<T> *p);
```

вставляющую узел в двусвязный список перед данным узлом,

```
template <class T>
DNode<T>* DNode<T>::DeleteBefor(void);
```

удаляющую узел, предшествующий данному узлу, из двусвязного списка.

19. Реализуйте и протестируйте функцию:

```
template <class T>
DNode<T> *BuildDNodeList(T M[], int n);
```

создающую двусвязный список обектов класса `DNode<T>`, извлекая данные из массива `M` длиной `n`. Для построения списка функция должна использовать конструктор класса.

20. Реализуйте и протестируйте функцию:

```
template <class T>
int IsDNodeListHead(DNode<T> *head);
```

проверяющую, является ли параметр `head` указателем на голову двусвязного списка объектов класса `DNode`. Необходимо, кроме головы, проверить правильность указателей каждого объекта списка.

6. ДЕРЕВЬЯ

6.1. Структура деревьев

Массивы и связные списки – примеры линейных структур, доступ к элементам которых осуществляется последовательно. Во многих приложениях, однако, обнаруживаются структуры, поддерживающие нелинейный порядок элементов. Одной из нелинейных структур, нашедших широкое применение, является древовидная структура (дерево). Она используется для описания самых различных объектов, – например, генеалогического древа человека, иерархической структуры управления предприятием и т.д.

Дерево состоит из узлов и ветвей и имеет направление от корня к внешним узлам, называемым листьями. Основные термины деревьев понятны

без определения: корень (root), предок, потомок, родитель, сын (непосредственный потомок), лист (узел, не имеющий потомков), поддерево и т.д.

В общем случае узлы деревьев могут иметь различное число сыновей. Мы сосредоточимся на ограниченном классе деревьев, где каждый родитель имеет не более двух сыновей. Такие деревья называются бинарными, имеют унифицированную структуру и допускают разнообразные алгоритмы прохождения и эффективный доступ к элементам. Пример бинарного дерева приведен на рис.4.

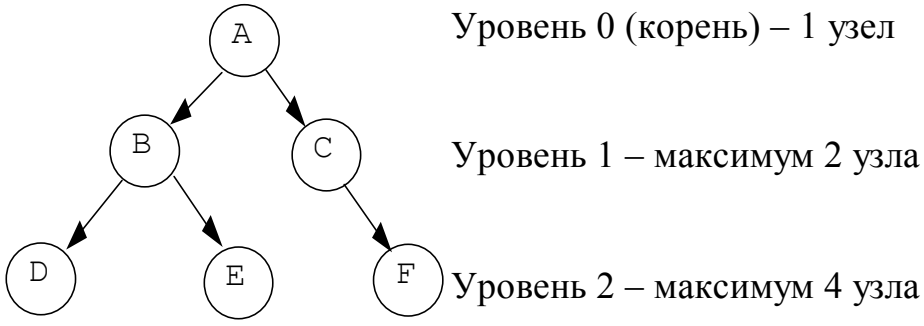


Рис. 4. Бинарное дерево

Путь от корня к узлу дерева дает меру, называемую *уровнем* (level) узла. Уровень корня принято считать за 0. Максимальный уровень узлов дерева есть его *глубина* (depth).

На любом уровне n бинарное дерево может содержать от 1 до 2^n узлов. Число узлов, приходящихся на уровень, является показателем плотности дерева. Плотность можно определить как отношение величины дерева (числа узлов) к его глубине. Крайними мерами плотности являются:

вырожденное (degenerate) бинарное дерево, имеющее на каждом уровне по одному листу;

полное (full) бинарное дерево, имеющее полный набор узлов на каждом уровне.

Вырожденное бинарное дерево глубины N содержит N+1 узлов и эквивалентно связному списку. Полное бинарное дерево глубины N содержит $2^{N+1}-1$ узлов.

На практике в качестве структур данных чаще применяются деревья с большой плотностью, так как они содержат больше элементов вблизи корня, т.е. с более короткими путями от него. Такие деревья позволяют хранить большие объемы данных и осуществлять быстрый доступ к элементам. Приложения, например, часто оперируют так называемыми *законченными* (complete) бинарными деревьями, у которых все уровни, кроме максимального, имеют полный набор узлов и все листья максимального уровня расположены слева относительно корня.

6.2. Класс TreeNode

В нашей реализации узлы бинарного дерева описываются классом `TreeNode`. Каждый узел – объект класса `TreeNode` – содержит поле данных

data и два поля с указателями на левого (left) и правого (right) сыновей. Тип указателей TreeNode*. Значение NULL какого-либо из указателей говорит о том, что узел не имеет соответствующего сына.

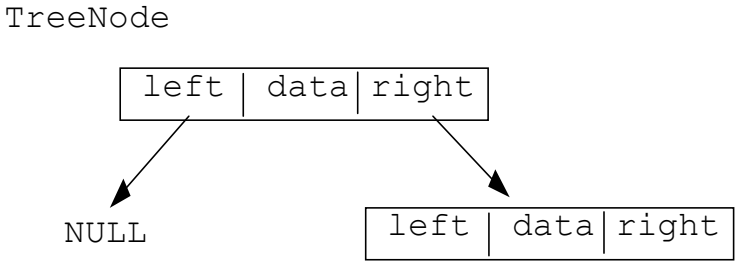


Рис. 5. Структура узлов бинарного дерева

Спецификация класса TreeNode:

```
#include <iostream.h>
//Класс BinSTree будет дружественным для TreeNode
template <class T>
class BinSTree;

//Класс узла бинарного дерева
template <class T>
class TreeNode
{
   friend class BinSTree<T>;
   private:
      TreeNode<T> *left;
      TreeNode<T> *right;
   public:
      T data;
      TreeNode(const T& item, TreeNode<T> *lptr = NULL,
                              TreeNode<T> *rptr = NULL):
      data(item), left(lptr), right(rptr) {}
      TreeNode<T> *Left(void) const {return left;}
      TreeNode<T> *Right(void) const {return right;}
};
```

Перед определением класса TreeNode помещено объявление другого класса – BinSTree. Это класс так называемых бинарных деревьев поиска, который будет определен ниже. BinSTree объявлен дружественным классу TreeNode, что в дальнейшем позволит получить методам BinSTree доступ к закрытым данным TreeNode.

Класс TreeNode основан на шаблоне, параметр T которого – тип данных поля data. Это поле является открытым, тогда как указатели left и right описаны в закрытой части класса. Таким образом, клиенты класса имеют прямой доступ к данным узла, но не могут изменить структуру дерева. Конструктор класса инициализирует поля данных и указателей. По умолчанию указателям присваивается NULL, и, таким образом, узел инициализируется как лист. Методы Left и Right возвращают значения указателей, позволяя клиентам класса проходить дерево.

Для работы с деревом, состоящим из объектов класса `TreeNode`, должны использоваться внешние функции. Одной из задач является *прохождение дерева*, т.е. обход всех его узлов. Методы прохождения по сути своей рекурсивны, так как имеют дело с рекурсивной структурой дерева. Действительно, каждый узел дерева является корнем своего собственного дерева, и, «попадая» на этот узел, функция прохождения как бы начинает свою работу сначала, используя один и тот же алгоритм. Существует три основных метода прохождения деревьев: *симметричный*, *прямой* и *обратный*.

Порядок операций при симметричном методе прохождения бинарного дерева следующий:

1) прохождение левого поддерева;

2) посещение узла;

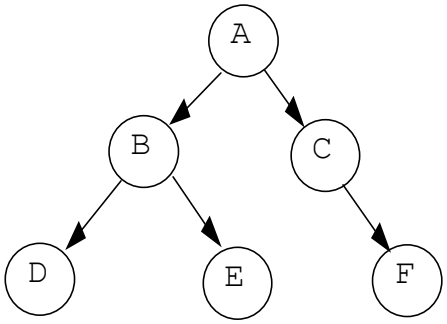
3) прохождение правого поддерева.

Такое прохождение называется LNR–прохождением (left, node, right).

Прямой метод прохождения определяется посещением узла в первую очередь и последующим прохождением сначала левого, а потом правого его поддеревьев (NLR–прохождение (node, left, right)).

При обратном методе прохождения посещение узла откладывается до тех пор, пока не будут пройдены оба поддерева. Порядок операций задает LRN(left, right, node) сканирование.

Рисунок демонстрирует порядок посещения узлов для всех трех рассмотренных методов.



Методы прохождения дерева:

Симметричный	`D–B–E–A–C–F`
Прямой	`A–B–D–E–C–F`
Обратный	`D–E–B–F–C–A`

Рис.6. Методы прохождения бинарного дерева

Напишем функцию, реализующую симметричный метод прохождения дерева. Функция должна обойти все узлы и для каждого выполнить определенное действие, связанное с доступом к данным узла. Это действие будет указано функции посредством параметра:

```
template <class T>
void Inorder(TreeNode<T> *t, void (*visit)(T &item))
{
  // прохождение завершается на пустом поддереве (t=NULL)
  if (t)
  {
    Inorder(t->Left(), visit); // пройти левое поддерево
    (*visit)(t->data); // посетить узел
    Inorder(t->Right(), visit); // пройти правое поддерево
  }
}
```

Параметрами функции Inorder являются указатель t на узел дерева и указатель visit на функцию, непосредственно выполняющую работу с данными узла. Единственный параметр функции – ссылка на данные типа T. Для каждого узла Inorder сначала рекурсивно спускается по левому поддереву, потом вызывает функцию, адресуемую указателем visit, которая выполняет некоторые действия с данными этого узла. После этого Inorder рекурсивно спускается по правому поддереву.

Допустим, у нас имеется дерево, состоящее из узлов, данные которых имеют тип char*, т.е. являются указателями на строки. Корень дерева определяется указателем root. Нам необходимо вывести на экран данные дерева, пользуясь симметричным методом прохождения. Функция печати строки в таком случае должна быть примерно следующей:

```
void Print(char* &item) { cout << '\n' << item;}
```

Тогда для вывода на экран данных узлов необходимо выполнить следующий вызов:

```
Inorder(root, Print);
```

Очевидно, что функции, реализующие прямой и обратный методы прохождения дерева, аналогичны Inorder, за исключением порядка следования рекурсивных вызовов и вызова функции обработки.

Рекурсивные методы прохождения используются также в таких операциях как копирование и удаление деревьев.

Функция CopyTree создает в динамической памяти копию дерева, корень которого передан ей в качестве параметра. Функция реализует обратный метод прохождения копируемого дерева. Для каждого узла сначала рекурсивным вызовом создаются копии его левого и правого поддеревьев, потом создается сам узел:

```
template <class T>
TreeNode<T> * CopyTree(TreeNode<T> *t)
{
   if(!t) return NULL;
   TreeNode<T> * newlptr=NULL, * newrptr = NULL, * newnode;
   if (t->Left()) newlptr = CopyTree(t->Left());
   if (t->Right()) newrptr = CopyTree(t->Right());
   newnode = new TreeNode<T>(t->data, newlptr, newrptr);
   if (!newnode)
   {
         cout << "Memory allocation error";
         exit(1);
   }
   return newnode;
}
```

Функция DeleteTree удаляет из динамической памяти узлы дерева, указатель на корень которого передан ей в качестве параметра. Как и CopyTree, в DeleteTree применяется обратный метод прохождения дерева: сначала удаляются левое и правое поддеревья, а потом и сам узел.

```
template <class T>
void DeleteTree(TreeNode<T> * t)
{
   if(t)
   {
    DeleteTree(t->Left());
    DeleteTree(t->Right());
    delete t;
   }
}
```

6.3. Бинарные деревья поиска

Бинарным деревом поиска (binary search tree) называется бинарное дерево, строящееся по принципу: для каждого узла значение данных в левом поддереве меньше, чем в этом узле, а в правом – больше либо равно. Таким образом, бинарное дерево поиска упорядочивает элементы посредством оператора отношения «<». Сравнивая узлы деревьев, мы подразумеваем, что часть или все поле данных узла определено в качестве ключа и оператор «<» сравнивает ключи, когда размещает элемент в дереве (см. лаб. работу №3 «Очереди приоритетов»).

Бинарные деревья поиска предоставляют очень эффективные методы поиска элементов (отсюда и такое их название), значительно превосходящие возможности последовательных структур: массивов и связных списков. Однако эффективность поиска в дереве зависит от его плотности. Чем плотнее дерево, тем быстрее осуществляется поиск данных. Вырожденное дерево не дает никаких преимуществ перед последовательными списками. Наибольшая эффективность достигается для полного и законченного деревьев. Максимальное число сравнений для поиска элемента в них составляет $\log_2 N$, где N – число узлов дерева. Пусть, например, у нас имеется список из 10000 элементов. Среднее число сравнений при последовательном поиске элемента в списке составляет 5000. Если разместить элементы списка в узлах законченного дерева, поиск любого элемента потребует не более 14 сравнений.

Класс бинарного дерева поиска:

```
#include <stdlib.h>
#include "treenodelib.h"
template <class T>
class BinSTree
{
   private:
      TreeNode<T> *root, *current;
      int size;
      TreeNode<T> * FindNode(const T& item,
                        TreeNode<T> * &parent) const;
   public:
      // конструктор
      BinSTree(void): root(NULL), current(NULL), size(0){}
      // конструктор копирования
      BinSTree(const BinSTree<T> & tree);
      // деструктор
```

```
        ~BinSTree(void) {DeleteTree(root);}
        // перегруженный оператор присваивания
        BinSTree<T>& operator=(const BinSTree<T> & tree);
        // поиск узла
        int Find(T& item);
        // вставка данных в дерево
        void Insert(const T & item);
        // удаление данных из дерева
        void Delete(const T & item);
        // очистка дерева
        void ClearTree(void){DeleteTree(root);
                             root=current = NULL;size=0;}
        // дерево пусто?
        int TreeEmpty(void) const {return !size;}
        // размер дерева
        int TreeSize(void) const {return size;}
        //обновить дерево
        void Update(const T & item);
};
```

Класс `BinSTree` основан на шаблоне, параметр которого `T` – тип данных узла. Данными класса являются: `root` – указатель на корень дерева, `current` – указатель на текущий узел, `size` – число узлов в дереве.

Закрытый метод `FindNode` осуществляет поиск узла по значению данных. Возвращаемым значением является указатель на найденный узел. Одновременно через параметр `parent`, переданный по ссылке, возвращается указатель на родителя этого узла. Указатель на родителя может потребоваться, например, при удалении найденного узла для восстановления структуры дерева.

Поиск элемента в дереве осуществляется с применением операции «==» для сравнения данных узла с параметром `item`. Предполагается, что эта операция определена для типа `T`. В случае, если `T` – структурный тип, оператор «==» должен быть перегружен для данного типа. При этом может быть организовано сравнение структур по ключу и функция `FindNode` будет искать данные по ключу.

```
TreeNode<T> * BinSTree<T>::FindNode(const T& item,
                                    TreeNode<T> * &parent) const
{
   TreeNode<T> *t = root;
   parent = NULL;
   while(t)
   {
      if (item==t->data) break;
      else
      {
         parent = t;
         if (item < t->data)
              t = t->left;
         else
              t = t->right;
```

```
        }
    }
    return t;
}
```

Так как класс `BinSTree` содержит указатель на динамические данные, он должен иметь конструктор копирования. Конструктору копирования в качестве параметра передается ссылка на дерево – объект класса `BinSTree`. Конструктор вызывает внешнюю функцию `CopyTree` для создания копии дерева в динамической памяти, настраивает указатели `root` и `current` созданного объекта и инициализирует переменную `size`:

```
template <class T>
BinSTree<T>::BinSTree(const BinSTree<T> & tree)
{
    root = CopyTree(tree.root);
    current = root;
    size = tree.size;
}
```

Перегруженный оператор присваивания работает аналогично конструктору копирования, за исключением того, что перед созданием копии дерева он удаляет «старое» дерево вызовом `DeleteTree`. Функция возвращает ссылку на объект – левый операнд операции присваивания, что позволяет осуществить каскадный вызов операции.

```
template <class T>
BinSTree<T>& BinSTree<T>::
                    operator=(const BinSTree<T> & tree)
{
    DeleteTree(root);
    root = CopyTree(tree.root);
    current = root;
    size = tree.size;
    return *this;
}
```

Метод `Find` осуществляет поиск данных в дереве по ключу. Ключ должен входить в состав полей параметра `item`, передаваемого функции `Find` по ссылке. После вызова метода, если поиск увенчался успехом, `item` будет содержать «полные» данные узла, найденного по этому ключу.

Для поиска узла метод использует закрытую функцию `FindNode`, результат вызова которой присваивается указателю `current`. Если узел найден, в `item` заносятся «полные» данные узла и функция `Find` возвращает 1. В противном случае возвращается 0.

```
template <class T>
int BinSTree<T>::Find(T& item)
{
    TreeNode<T> *parent;
    current = FindNode(item,parent);
    if (current)
    {
```

```
      item = current->data;
      return 1;
   }
   else return 0;
}
```

Метод `Insert` принимает в качестве параметра новый элемент данных и вставляет его в подходящее место на дереве. Функция итеративно проходит путь по дереву, пока не найдет точку вставки. В этот момент становится известным родитель вставляемого узла. Далее функция определяет, каким сыном – левым или правым – будет вставляемый узел и присоединяет его к родителю.

```
template <class T>
void BinSTree<T>::Insert(const T& item)
{
   // t – текущий узел, parent – предыдущий,
   //newNode – вставляемый
   TreeNode<T> *t = root, *parent=NULL,
           *newNode = new TreeNode<T>(item);
   while(t) // остановиться на пустом дереве
   {
      // обновить указатель parent и идти направо или налево
      // в зависимости от значения данных
      parent = t;
      if (item < t->data)
         t = t->left;
      else
         t = t->right;
   }
   // если дерево пусто, вставить в качестве корня
   if (!parent)
         root = newNode;
   // если item меньше данных родителя, вставить левым сыном
   else if (item < parent->data)
              parent ->left = newNode;
         else
              //вставить правым сыном
              parent->right = newNode;
   // сделать новый узел текущим и увеличить size
   current = newNode;
   size++;
}
```

Операция `Delete` удаляет из дерева узел с заданным ключом. Ключ является одним из полей параметра `item`. Сначала с помощью метода `FindNode` устанавливается место этого узла на дереве и определяется указатель на его родителя. Если искомый узел отсутствует, операция завершается.

Удаление узла требует ряда проверок, чтобы определить, куда присоединить сыновей удаляемого узла. Поддеревья должны быть присоединены таким образом, чтобы сохранилась структура бинарного дерева поиска.

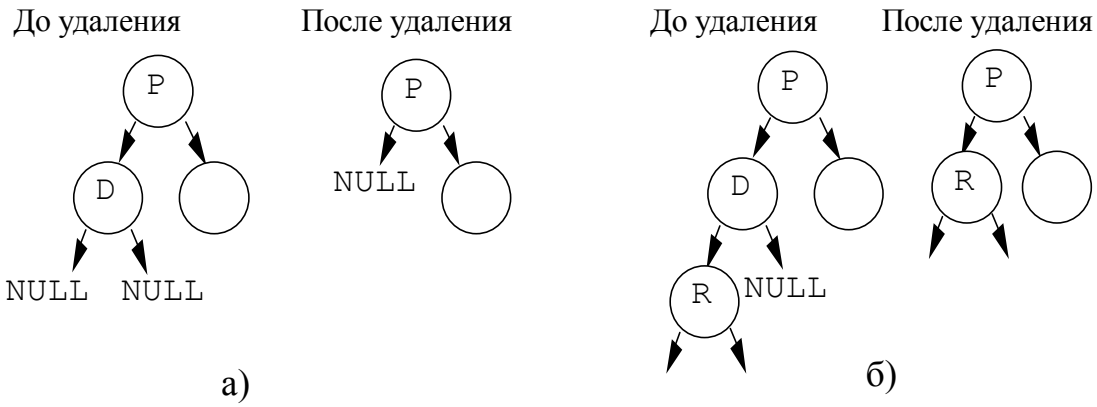


Рис. 7. Удаление узла бинарного дерева:
а) листа; б) имеющего только левого сына

При удалении узла возможны четыре ситуации:

1. Удаляемый узел `D` не имеет сыновей (рис.7а). Необходимо обновить родительский узел `P`, обнулив указатель на удаленного сына.

2. Удаляемый узел `D` имеет левого сына, но не имеет правого (рис. 7б). Необходимо присоединить его левое поддерево к родителю. Иными словами, заменяющим узлом `R` будет левый сын удаляемого.

3. Удаляемый узел имеет правого сына, но не имеет левого. Необходимо присоединить его правое поддерево к родителю. Ситуация аналогична предыдущей, за исключением того, что замещающим узлом будет правый сын удаляемого.

4. Удаляемый узел имеет обоих сыновей (рис.8).

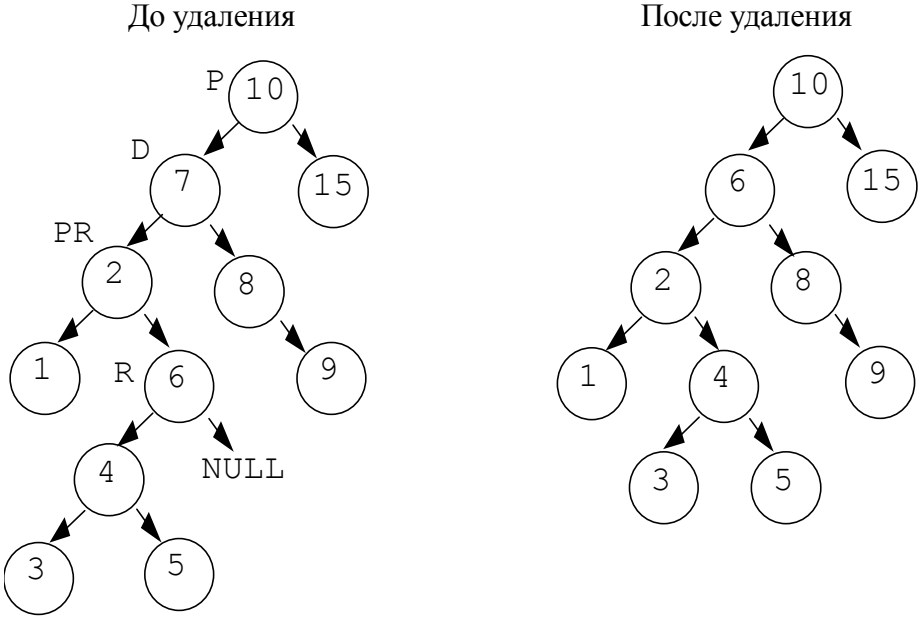


Рис. 8. Удаление узла с двумя сыновьями

Алгоритм должен выбрать такой заменяющий узел `R`, чтобы сохранилось правильное размещение элементов бинарного дерева поиска. Мы имеем две кандидатуры на роль замены:

1) самый правый узел левого поддерева удаляемого узла (узел 6);

2) самый левый узел правого поддерева удаляемого узла (узел 8).

Наш алгоритм будет использовать первый вариант. Таким образом, для отыскания замещающего узла мы должны спускаться по левому поддереву узла D, двигаясь все время вправо, до тех пор, пока не обнаружим узел, у которого отсутствует правый сын (`right == NULL`). Этот узел и будет замещающим.

Далее необходимо отсоединить замещающий узел R, присоединив его левого сына (узел 4) к родителю PR замещающего узла (узел 2) в качестве правого сына. Заметим, что правого сына у самого узла R не может быть по определению.

После этого остается заменить удаляемый узел D замещающим R, связав последний с сыновьями удаляемого (узлы 2 и 8) и его родителем P (узел 10).

Таков в общих чертах алгоритм замены удаляемого узла, когда он имеет обоих сыновей. В частных случаях может возникнуть ситуация, когда замещающим узлом является левый сын удаляемого узла (или, что то же самое, родителем замещающего узла является удаляемый: PR=D). Тогда алгоритм упрощается, так как мы можем, не отсоединяя узел R от потомков, сразу же присоединить его к узлу P – родителю D (рис.9).

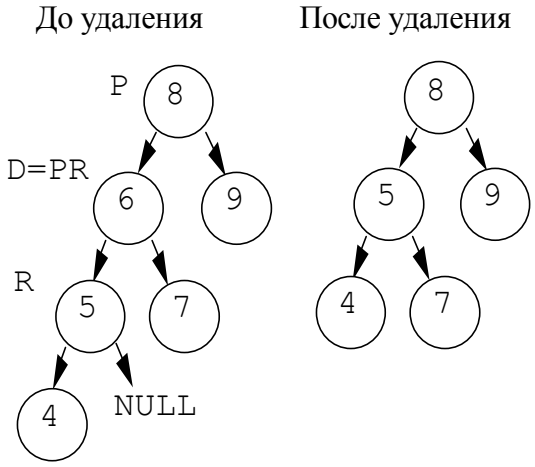


Рис. 9. Частный случай удаления узла: D=PR

После удаления узла D из дерева и его замены функция Delete уничтожает этот узел и уменьшает на единицу переменную size.

```
template <class T>
void BinSTree<T>::Delete(const T& item)
{
   // DNodePtr – указатель на удаляемый узел D
   // PNodePtr – указатель на родителя P удаляемого узла D
   // RNodePtr – указатель на замещающий узел R
   TreeNode<T> * DNodePtr, *PNodePtr, *RNodePtr;
   // найти узел, данные которого совпадают с item.
   // получить его адрес и адрес его родителя
   DNodePtr = FindNode(item,PNodePtr);
   // если такой узел не найден, выйти из функции
```

```
    if (!DNodePtr) return;
    // если D имеет NULL-указатель, заменяющим узлом
    // является тот, что находится на другой ветви
    if (!DNodePtr->right)
       RNodePtr = DNodePtr->left;
    else  if (!DNodePtr->left)
             RNodePtr = DNodePtr->right;
          else
          // узел D имеет двух сыновей
          {
             //PofRNodePtr – указатель на родителя PR узла R
             TreeNode<T> * PofRNodePtr = DNodePtr;
             // переходим в левое поддерево узла D
             RNodePtr = DNodePtr->left;
             // спускаемся «вправо», пока не найдем замену
             while (RNodePtr->right)
             {
                PofRNodePtr = RNodePtr;
                RNodePtr = RNodePtr->right;
             }
             if (PofRNodePtr == DNodePtr)
              // левый сын узла D является заменяющим
              // присоединить правое поддерево узла D к R
               RNodePtr->right = DNodePtr->right;
             else
             {
               // присоединить левое поддерево узла R
               // к его родителю
               PofRNodePtr->right = RNodePtr->left;
               // связать узел R с сыновьями D
               RNodePtr->left =  DNodePtr->left;
               RNodePtr->right =  DNodePtr->right;
             }
          }
    // осталось присоединить узел R к узлу P
    if (!PNodePtr)
          // у узла D нет родителя – он являлся корнем
          // назначить новый корень – узел R
          root = RNodePtr;
    // с какой стороны присоединить узел R к узлу P?
    else  if (DNodePtr->data < PNodePtr->data)
                // слева
                PNodePtr->left = RNodePtr;
           else
                // справа
                PNodePtr->right = RNodePtr;
    // удалить узел D из памяти и уменьшить размер дерева
    delete DNodePtr;
    size--;
}
```

Метод `Update` служит для обновления текущего узла и вставки новых данных. Он принимает в качестве параметра ссылку на элемент данных и, если

определен текущий узел, сравнивает его значение с этим элементом. Если они равны, производится обновление узла. В противном случае, а также если текущий узел не определен, новые данные включаются в дерево вызовом `Insert`. Напомним, что сравнение новых данных со значением текущего узла производится не по всем полям типа `T`, а только по ключевому полю, остальные поля могут отличаться и в результате обновления примут новые значения.

```
template <class T>
void BinSTree<T>::Update(const T& item)
{
   if (current && current->data==item)
    current->data = item;
   else
    Insert(item);
}
```

Остальные методы класса `BinSTree` типичны для классов коллекций. Все они реализованы внутри класса.

6.4. Задания

1. Реализуйте и протестируйте итерационную функцию

```
template <class T>
TreeNode * Max(TreeNode<T> *t);
```

которая возвращает указатель на максимальный узел бинарного дерева поиска.

2. Реализуйте и протестируйте рекурсивную функцию

```
template <class T>
TreeNode * Min(TreeNode<T> *t);
```

которая возвращает указатель на минимальный узел бинарного дерева поиска.

3. Реализуйте и протестируйте итерационную функцию

```
template <class T>
int NodeLevel(const BinSTree<T> &T, const T& elem);
```

которая определяет глубину узла с данными `elem` на бинарном дереве поиска и возвращает `-1`, если такого узла нет на дереве.

4. Пусть в узлах дерева находятся символьные строки. Постройте бинарное дерево поиска, которое получается в результате вставки следующих ключевых слов:

`for, case, while, class, protected, virtual, public, private, do, template, const, if, int.`

Осуществите прохождение этого дерева симметричным методом.

УКАЗАНИЕ. Для сравнения символьных строк в методах класса `BinSTree` необходимо, чтобы строки являлись объектами некоторого класса, для которого следует перегрузить оператор «<».

5. Реализуйте и протестируйте функцию

```
template <class T>
int CountEdges(TreeNode<T> *t);
```

которая подсчитывает число ребер (ненулевых указателей) бинарного дерева.

6. Реализуйте и протестируйте функцию

```
void TreeSort(double *M, int n)
```

которая сортирует массив `M` из `n` элементов путем включения их в бинарное дерево поиска, симметричного прохождения этого дерева и копирования отсортированных данных обратно в массив. Для симметричного прохождения и присвоения значений элементам массива напишите рекурсивную функцию

```
void InorderAssign(TreeNode<double> *t, double *M, int i);
```

7. Реализуйте и протестируйте функцию

```
template <class T>
TreeNode * ReverseCopy(TreeNode<T> *t);
```

которая копирует бинарное дерево, попутно меняя местами все левые и правые указатели.

8. Реализуйте и протестируйте функцию

```
template <class T>
void InsertOne(BinSTree<T> &t, T item);
```

которая включает данные `item` в бинарное дерево поиска, если его там еще нет. В противном случае функция завершается, не выполняя включения нового узла.

9. Реализуйте и протестируйте рекурсивную функцию

```
template <class T>
void CountLeaf(TreeNode<T> *t, int &count);
```

которая подсчитывает число узлов бинарного дерева, пользуясь обратным методом прохождения.

10. Реализуйте и протестируйте рекурсивную функцию

```
template <class T>
int Depth(TreeNode<T> *t);
```

вычисляющую глубину бинарного дерева с использованием обратного метода прохождения. Глубина дерева есть максимальный уровень его узлов. Глубина пустого дерева равна `-1`.

11,**12**. Узлы бинарного дерева поиска содержат данные типа

```
struct IntegerCount {int number; unsigned count;};
```

В переменной `number` хранится некоторое целое число, а поле `count` используется как счетчик появлений этого числа.

Напишите и протестируйте программу, которая выполняет следующие действия:

генерирует 100000 случайных чисел в диапазоне от 0 до 9, помещает каждое число в структуру `IntegerCount` и вставляет в бинарное дерево поиска, если такого числа в нем нет. Если число есть, то поле count соответствующего узла увеличивается на единицу;

симметрично проходит полученное дерево, выводя на экран значения чисел и счетчиков их появления.

УКАЗАНИЕ. В данной задаче в качестве ключа при сравнении данных узлов должно использоваться поле `number`. Учитывая это, необходимо соответствующим образом перегрузить оператор «<» для объектов структурного типа `IntegerCount`.

13,**14**. Разработайте и реализуйте класс `GTreeNode` для узлов дерева общего вида, имеющих неограниченное число сыновей. Функционально класс должен быть подобен классу узлов бинарного дерева `TreeNode`. Реализуйте также функцию симметричного прохождения дерева общего вида:

```
template <class T>
void Inorder(GTreeNode<T> *t, void (*visit)(T &item));
```

15. Реализуйте и протестируйте рекурсивную функцию

```
template <class T>
void DelDubNodes(BinSTree<T> &t);
```

удаляющую все дублирующие узлы из бинарного дерева поиска с использованием прямого метода прохождения. (Все дубликаты узла бинарного дерева поиска располагаются в правом его поддереве и не имеют левых сыновей.)

16. Реализуйте и протестируйте рекурсивную функцию

```
template <class T>
int BinSTreeTest(TreeNode<T> *t);
```

которая проверяет, имеет ли дерево, состоящее из узлов – объектов класса `TreeNode`, – структуру бинарного дерева поиска. Если это так, функция возвращает 1, в противном случае 0.

17. Реализуйте и протестируйте функцию

```
template <class T>
TreeNode<T> * BuildTree(T* M, int n);
```

которая строит бинарное дерево из элементов массива `M` длиной `n` с помощью конструктора класса `TreeNode`. Дерево должно быть максимально заполненным, т.е. иметь минимально возможное число уровней. Порядок размещения элементов массива в дереве произволен.

18. Реализуйте и протестируйте рекурсивную функцию

```
template <class T>
BuildTree(BinSTree<T> &t, T* M, int l, int h);
```

которая заносит в бинарное дерево поиска данные из массива `M`, упорядоченного по возрастанию элементов. Данные располагаются в диапазоне индексов `l…h` и заносятся таким образом, чтобы дерево было максимально заполненным, т.е. имело минимально возможное число уровней. Первоначально дерево пусто.

19,**20**. Бинарное дерево с n узлами имеет n+1 нулевых указателей, т.е. половина памяти, отведенной под указатели, расходуется напрасно. Хороший алгоритм использует эту память. Пусть при симметричном прохождении каж-

дый левый (в прошлом пустой) указатель теперь указывает на своего предшественника, а каждый правый – на преемника. Такая структура называется прошитым деревом, а рассмотренные выше указатели – нитями. Разработайте и реализуйте следующие функции:

```
template <class T>
TreeNode<T>  *ThreadedCopy(TreeNode<T> * t);
```

создающую «прошитую» копию бинарного дерева, корень которого передан функции в качестве параметра (выполняется при симметричном прохождении последнего);

```
template <class T>
void ThreadInorder(TreeNode<T> *t, void (*visit)(T &item));
```

осуществляющую симметричное прохождение дерева, начиная с узла `t`, с помощью итерационного алгоритма.

7. ИМИТАЦИОННОЕ МОДЕЛИРОВАНИЕ

7.1. Модели обслуживания

Задачей моделирования является создание модели реальной ситуации для лучшего ее понимания. Моделирование позволяет вводить различные условия и наблюдать результаты их выполнения.

Мы будем рассматривать так называемую *модель обслуживания*. Эта модель подходит для описания множества реальных процессов, – таких как обслуживание клиентов банка, магазина, парикмахерской и т.д. Основные действующие лица модели: клиенты и обслуживающий персонал. Клиенты приходят, встают в очередь, обслуживаются персоналом и уходят. При этом промежутки времени между приходами клиентов, а также время обслуживания каждого клиента являются случайными величинами, распределенными на некоторых интервалах. Скажем, время обслуживания имеет постоянную составляющую (например, минимальное время обслуживания) и случайную переменную составляющую. Постоянная составляющая для промежутка между приходами клиентов может быть принята равной нулю.

Результаты моделирования (прогона программы) должны отражать основные показатели, характеризующие эффективность работы учреждения: затраты времени клиентов на ожидание и обслуживание, занятость персонала и число обслуженных клиентов в течение рабочего дня. Анализируя эти показатели, руководитель, в частности, сможет принимать решения о приеме на работу (или увольнении) персонала.

Существуют три основных подхода к построению подобных моделей: *сканирования активностей*, *процессно-ориентированный* и *событийный* [3].

При использовании подхода сканирования активностей разработчик описывает все действия, в которых принимают участие элементы системы, и задает условия, определяющие начало и завершение действий. После каждого продвижения имитационного времени условия всех возможных действий про-

веряются, и если некоторое условие выполняется, то происходит имитация соответствующего действия. Достоинством подхода является простота реализации модели – мы описываем систему как набор условий и набор действий, соответствующих этим условиям, не заботясь об организации процесса имитации. Основной недостаток подхода – большие затраты времени для расчета из-за того, что на каждом шаге необходимо проверять условия всех действий (а их могут быть сотни). Другим недостатком является неточность фиксации событий во времени, связанная с дискретностью его изменения.

В процессно-ориентированном подходе исследователь описывает последовательность компонентов модели, которые возникают по определенной схеме. Все действия, предусмотренные в программе, принадлежат определенным процессам, которые функционируют параллельно и взаимодействуют между собой. Например, модель обслуживания при таком подходе состояла бы из самостоятельных процессов, каждый из которых описывал бы продвижение одного клиента по цепочке «приход–ожидание–обслуживание–уход». Число таких процессов в каждый момент времени равнялось бы числу клиентов, находящихся в этот момент в учреждении. Недостатком подхода является сложность синхронизации процессов. Поэтому для реализации такого подхода необходима среда (язык программирования, операционная система), поддерживающая параллельные вычисления.

При событийном подходе исследователь описывает события, которые могут изменять состояние системы, и определяет логические взаимосвязи между ними. Имитация происходит путем выбора из списка будущих событий ближайшего по времени и его выполнения. Выполнение события приводит к изменению состояния системы и генерации будущих событий, логически связанных с выполняемым. Эти события заносятся в список будущих событий и упорядочиваются в нем по времени наступления.

Наша реализация модели обслуживания использует событийный подход, так как он хорошо подходит для демонстрации возможностей объектно-ориентированного программирования. ОПП поощряет повторное использование кода, а следовательно, описание наиболее общих абстракций предметной области. Эти абстракции будут развиваться по ходу разработки с использованием механизма наследования.

7.2. Наследование и полиморфизм

Механизм наследования предполагает построение иерархии классов. Иерархия классов позволяет определять новые классы на основе уже имеющихся. Имеющиеся классы обычно называются базовыми, а новые – производными. Производные классы «получают в наследство» от базовых данные и методы и, кроме того, могут пополняться собственными компонентами. Определение производного класса обязательно содержит список всех базовых классов, из которых он непосредственно наследует:

```
class DerivedClass: public BaseClass1, private BaseClass2{…}
```

Здесь определяется класс `DerivedClass`, наследующий от классов `BaseClass1` и `BaseClass2`. Ключевые слова `public` и `private` перед именами базовых классов выступают в качестве *спецификаторов доступа*.

Спецификатор позволяет влиять на уровень доступа в производном классе к унаследованным компонентам базового класса. Необходимо помнить, что с помощью спецификатора можно только «усложнить» доступ, но не упростить его (см. табл.2).

Таблица 2)

Доступ к компонентам базового класса в производном

Доступ в базовом	Спецификатор доступа	Доступ в производном
public	отсутствует	private
protected	отсутствует	private
private	отсутствует	недоступны
public	public	public
protected	public	protected
private	public	недоступны
public	private	private
protected	private	private
private	private	недоступны

Так, открытые и защищенные компоненты класса `BaseClass1`, наследуемого со спецификатором `public`, не изменят статуса доступа в классе `DerivedClass`. Закрытые же компоненты `BaseClass1` будут недоступны в `DerivedClass`, несмотря на спецификатор `public`.

Отсутствие спецификатора доступа равносильно спецификатору `private`.

Кратко остановимся на особенностях конструкторов и деструкторов при наследовании. При создании объекта производного класса всегда сначала создается та часть его компонент, которая унаследована им от базовых классов. При этом вызываются конструкторы базовых классов в порядке их перечисления в определении производного класса. Производный класс также может иметь свой конструктор (или несколько конструкторов). Конструктор производного класса вызывается в последнюю очередь.

Если при создании объекта производного класса необходимо передать параметры конструктору его базового класса, это делается следующим образом. Параметры передаются конструктору производного класса, и он перед выполнением «своих» действий явно вызывает конструктор базового класса, передавая ему эти параметры.

Порядок вызова деструкторов при наследовании противоположен порядку вызовов конструкторов. Сначала вызывается деструктор производного класса, потом деструкторы всех базовых классов по порядку, обратному их перечислению в определении производного класса.

С наследованием связано также такое понятие объектно-ориентированного программирования как полиморфизм.

Полиморфизм означает, что один и тот же метод может быть по-разному определен для объектов различных классов – потомков одного базового класса. Конкретное поведение метода будет зависеть от типа объекта. Причем тип объекта определяется не на этапе компиляции, как это происходит при раннем связывании, а на этапе выполнения. Такой механизм выбора метода называется поздним (динамическим) связыванием.

C++ поддерживает полиморфизм и позднее связывание с помощью виртуальных функций-членов. Рассмотрим следующий фрагмент:

```
class BaseCL
{
    …
    public:
         …
         void fun1(void);          // обычный метод
         virtual void fun2(void); // виртуальный метод
};
class DerivedCL: public BaseCL
{
    …
    public:
         …
         void fun1(void); // обычный метод
         void fun2(void); // виртуальный метод
};
DerivedCL DCObject; //объект производного класса
BaseCL * pBC;       // указатель базового класса
pBC = & DCObject;  // настройка указателя
pBC->fun1(); // вызов BaseCL::fun1(void)
pBC->fun2(); // вызов DerivedCL::fun2(void)
```

В данном примере определены два класса: базовый `BaseCL` и производный `DerivedCL`. В производном классе переопределены открытые методы базового класса: «обычный» `fun1` и виртуальный `fun2`. Заметим, что переопределение виртуального метода не требует повторного применения ключевого слова `virtual`. Далее объявляется объект `DCObject` производного класса и указатель `pBC` типа базового класса, который настраивается на объект `DCObject`. Это вполне допустимо по правилам языка: указатель базового класса может быть настроен на объект производного класса.

После этого мы вызываем методы `fun1` и `fun2` для объекта `DCObject`, адресуемого указателем `pBC`. Какие методы будут при этом вызываться: базового класса или производного?

В первом случае речь идет об обычном, не виртуальном методе. Поэтому компилятор определяет тип указателя `pBC` – класс `BaseCL` и вызывает метод базового класса. То, что этот указатель настроен на объект производного класса, не имеет значения, компилятор этого уже «не помнит», ему важен только тип указателя. Это пример раннего связывания.

Во втором случае вызывается виртуальный метод, и компилятор действует иначе. В место вызова помещается специальный код, который при вы-

полнении программы производит обращение к объекту и «определяет его тип». Все объекты классов, включающих виртуальные функции, имеют скрытый указатель на специальную таблицу, в которой хранятся адреса всех виртуальных функций класса. С помощью этой таблицы и производится выбор метода. Это пример позднего связывания и полиморфного вызова.

Как видно, использование виртуальных функций влечет за собой некоторые накладные расходы, связанные с увеличением объема памяти для хранения объектов и объема кода для вызова методов. Тем не менее, полиморфизм используется очень широко, и в ряде приложений без него действительно трудно обойтись.

Допустим, нам необходимо одновременно вывести на экран несколько графических фигур, причем сколько и каких именно, заранее не известно. Фигуры должны быть представлены в программе объектами различных классов. Предположим, что каждый из этих классов имеет метод `draw` для рисования фигуры на экране.

Чтобы вывести все фигуры на экран, мы должны сосредоточить графические объекты в одном массиве и потом в цикле вызвать для каждого объекта метод `draw`. Однако здесь возникает вопрос: элементы какого типа будет содержать наш массив, если графические объекты имеют различные типы? Подобные проблемы легко разрешаются с помощью использования так называемых абстрактных классов.

Абстрактные классы вводятся для представления наиболее общих понятий, которые в дальнейшем предполагается конкретизировать [5]. Эти понятия невозможно использовать непосредственно, но на их основе можно как на базе построить частные производные классы, пригодные для описания конкретных объектов. С точки зрения языка программирования, абстрактным классом называется класс, в котором есть хотя бы одна чистая (пустая) виртуальная функция, имеющая определение вида:

```
virtual тип имя_функции(список параметров) = 0;
```

Такая функция «ничего не делает» и недоступна для вызовов. Ее назначение – служить основой для подменяющих ее функций в производных классах.

Для нашего случая можно определить абстрактный класс `Figures`, описывающий графические фигуры «вообще» и имеющий чисто виртуальную функцию `draw`.

```
class Figures
{
    virtual void draw(void) = 0;
};
```

Все классы графических объектов будут наследовать от класса `Figures` и переопределять метод `draw`, задавая конкретное его поведение.

Теперь мы можем объявить массив указателей на объекты класса `Figures`, заполнить его адресами объектов производных классов (графических объектов) и вывести фигуры на экран:

```
Figures *GraphicFigures[10];
int n; // число объектов
// заполнение массива указателями на графические объекты
//…
// вывод на экран
for(int i = 0; i<n; i++)  GraphicFigures[i]->draw();
```

В данном случае, естественно, вызовы метода `draw` будут полиморфными, так как «настоящие» типы графических объектов, адреса которых помещены в массив, становятся известными только при выполнении цикла в работающей программе.

7.3. Разработка модели обслуживания

Нам требуется разработать и реализовать структуру классов для модели обслуживания. Наша модель, как уже было сказано, будет использовать событийный подход. Этого знания достаточно, чтобы уже на данном этапе, не вдаваясь в подробности задачи, определить одну из ключевых абстракций модели – генератор событий.

Задача генератора событий – заполнять и хранить список будущих событий, извлекать из него событие, ближайшее по времени, и передавать его на обработку. Обработка будет различной в зависимости от типа события. Генератор событий не «знает» и не должен «знать» типы событий и при генерации любого из них должен действовать единообразно. Выражаясь в терминах языка программирования, он должен вызывать компонентную функцию некоторого «класса обработчиков событий». Этот класс будет основой для всех классов, реально обрабатывающих события определенных типов, а функция-обработчик окажется виртуальной.

Объявим абстрактный класс обработчиков событий `EventsServers` с единственной чисто виртуальной функцией `ServeEvent` (обработать событие):

```
//Абстрактный класс обработчиков событий
class EventsServers
{
    public:
    virtual void ServeEvent()= 0;
};
```

Каждый класс, наследующий от `EventsServers`, будет располагать собственной реализацией `ServeEvent`, подходящей для событий конкретного типа.

Генератор событий выбирает из списка будущих событий ближайшее по времени. Поэтому естественным решением проблемы сохранения списка событий будет использование очереди приоритетов. Класс очереди приоритетов `PQueue` разработан нами в лабораторной работе №3, его определение помещено в файл `pqueuelib.h`. До подключения этого файла необходимо определить тип элементов очереди и задать максимальную ее длину.

Элементом очереди будет структура, содержащая запись о будущем событии. Она должна включать указатель на объект – обработчик события и время наступления события:

```
//Структура записи о будущем событии
struct EventRecord
{
    EventsServers *pEventServer;
    unsigned EventTime;
};
```

Значением указателя `pEventsServer` будет адрес объекта класса – потомка `EventsServers`, а генерация события заключаться в вызове: `pEventServer->ServeEvent()`. Этот вызов будет полиморфным, так как он распознается не на этапе компиляции, когда тип объекта-обработчика еще не известен, а на этапе выполнения программы.

Извлечение элемента из очереди приоритетов требует сравнения элементов (в нашем случае – для определения ближайшего по времени события). Поэтому необходимо перегрузить оператор «<» для объектов класса `EventRecord`:

```
int operator<(EventRecord ev1,EventRecord ev2)
{
    return ev1.EventTime < ev2.EventTime;
}
```

Максимальная длина очереди приоритетов определяется максимальным количеством событий, планируемых в любой момент времени. Рассматривая генератор событий как «самостоятельную» абстракцию, без привязки к конкретной задаче, невозможно точно определить эту величину. Поэтому зададим ее некоторым достаточно большим числом (например, 20). В будущем эта цифра может быть уточнена.

Теперь, наконец, в программу можно включить определение классов `PQueue` и генератора событий `EventsGenerators`:

```
// Определение класса PQueue
typedef EventRecord PQDataType; //тип элементов очереди
const int MaxPQSize = 20; // максимальная длина очереди
#include "pqueuelib.h"   // класс PQueue
// Класс EventsGenerators
class EventsGenerators
{
    private:
       unsigned Time;
       PQueue FutureEvents;
   public:
       //Конструктор
       EventsGenerators(void);
       //Предсказать новое событие
       void PlanNewEvent(EventsServers* , unsigned );
       // Вернуть время
       unsigned GetTime(void);
```

```
        //Главный процесс
        void Procces(void);
};
```

Компонентными данными класса являются беззнаковое целое `Time`, в котором хранится текущее время, и очередь приоритетов `FutureEvents`. Методы класса:

конструктор – устанавливает `Time=0`;

`PlanNewEvent` – формирует и заносит в очередь приоритетов запись о будущем событии, параметрами служат указатель на обработчика и время наступления события;

`GetTime` – возвращает текущее время;

`Procces` – «главная» функция класса, пока не окончилось время моделирования, извлекает из очереди приоритетов запись о ближайшем событии, изменяет текущее время и вызывает соответствующую функцию обработки события.

Перейдем к нашей конкретной задаче – построению модели. Зададим входные, или варьируемые, параметры модели, а также требования к результатам моделирования.

Варьируемыми параметрами модели будут постоянные и случайные составляющие промежутков времени между приходами клиентов, времени обслуживания, а также максимальный размер очереди клиентов и число обслуживающих работников. Для простоты зададим эти параметры в виде констант:

```
// Время между приходами клиентов = 0...RandClientTime
#define RandClientTime 30
// Время обслуживания =
// ServingTime...ServingTime+RandServingTime
#define ServingTime 30
#define RandServingTime 30
// Максимальный размер очереди
#define MaxQSize 6
// Число обслуживающих работников
#define NOS 2
```

Результаты прогона программы должны фиксироваться в текстовых файлах. В файле `Clients.txt` должна отражаться информация обо всех клиентах, посетивших учреждение в течение рабочего дня. Эта информация упорядочивается в виде таблицы с заголовком вида:

```
Name: Enter: Qlen: Serbeg: Server: Exit: Wait: Served:  All:
```

Позиции заголовка соответствуют следующему порядку внесения информации о клиентах в файл: номер клиента, время прихода, длина очереди на момент прихода, время начала обслуживания, номер обслуживающего работника, время ухода, длительность ожидания, длительность обслуживания, общее время нахождения в учреждении. Занесение информации в таблицу легко осуществляется с применением табуляции.

Кроме того, программа должна создавать несколько файлов, относящихся к работникам: `Server1.txt`, `Server2.txt` и т.д. (по одному файлу на

работника). В этих файлах будет фиксироваться информация о занятости работника в течение рабочего дня и тех клиентах, которые были им обслужены. Эта информация также упорядочивается в виде таблицы с заголовком:

```
Server 1:
Number: Client:  enter:   exit:    Service Time:
```

Позиции заголовка соответствуют следующему порядку внесения информации в файл: номер клиента по порядку поступления на обслуживание к данному работнику, номер клиента по порядку прихода в учреждение, время начала обслуживания, время окончания обслуживания, длительность обслуживания.

События. В нашей системе будут иметь место два вида событий:

1) приход очередного клиента,

2) окончание обслуживания работником одного из клиентов.

Следовательно, реальная длина очереди приоритетов будет не более чем число работников (для каждого из которых планируется событие «окончание обслуживания») плюс 1 (событие «приход клиента»).

Классы. Анализ задачи позволяет выделить следующие абстракции (классы):

класс `Clients` (клиенты) – отвечает за хранение и вывод информации о клиенте. Объекты класса накапливают сведения о пребывании клиента в учреждении по мере его продвижения по цепочке «приход–ожидание–обслуживание–уход» и перед уничтожением фиксируют эти сведения в файле `Clients.txt`;

класс `ClientsQueues` (очереди клиентов) – отвечает за хранение и обновление информации об очереди клиентов, дожидающихся обслуживания. Заказывает и обрабатывает событие «приход клиента», создавая при этом новые объекты класса `Clients`, по запросу поставляет первого в очереди клиента на обслуживание;

класс `Servers` (работники) – отвечает за хранение и обновление информации о работнике и его занятости в каждый момент времени. Заказывает и обрабатывает событие «окончание обслуживания»;

класс `Managers` (управляющие) – отвечает за хранение информации о всех работниках, служит связующим звеном между ними и очередью клиентов.

Класс `Clients`:

```
#include <fstream.h>
class Servers;
class ClientsQueues;
//Класс Clients
class Clients
{
    friend Servers;
    friend ClientsQueues;
    private:
         unsigned Name;
```

```
        unsigned QueueLength;
        unsigned Server;
        unsigned EnterTime, ServiceBeginTime, ExitTime;
        //общее число клиентов
        static unsigned NumberOfClients;
        static ofstream ClientsOutFile;
    public:
        Clients(void); // конструктор
        ~Clients();
};
```

Класс в основном предназначен для хранения информации о каждом клиенте. Формировать эту информацию будут объекты дружественных классов `Servers` и `ClientsQueues`. Данные класса включают:

`Name` – порядковый номер клиента;

`QueueLength` – длина очереди на момент прихода клиента;

`Server` – номер работника, обслуживающего клиента;

`EnterTime, ServiceBeginTime, ExitTime` – время прихода, начала обслуживания и ухода;

`NumberOfClients` – общее число клиентов;

`ClientsOutFile` – поток вывода, связанный с файлом, в который заносится информация о клиентах.

Последние два компонента являются статическими, т.е. общими для всех экземпляров класса. Они могут быть инициализированы до создания объектов класса следующим образом:

```
unsigned Clients::NumberOfClient = 0;
ofstream Clients::ClientsOutFile;
```

Методы класса:

конструктор `Clients` – увеличивает на 1 `NumberOfClients`, в случае, если клиент первый `(NumberOfClient==1)`, открывает файл `Clients.txt`, связывая его с потоком `ClientsOutFile`, и записывает в него заголовок таблицы;

деструктор `~Clients` – вносит информацию о клиенте в файл при уничтожении объекта класса.

Класс ClientsQueue поддерживает структуру типа «Очередь» для клиентов. Поэтому естественно сделать его наследником класса `Queue`, разработанного нами в лабораторной работе №3. Определение данного класса помещено в файл `queuelib.h`. Длина очереди задается константой `MaxQSize`, которая нами определена выше. Тип элементов очереди – указатель на объект класса `Clients`:

```
//Подключение класса Queue
typedef Clients* QDataType;
#include "queuelib.h"
```

Методы класса `Queue` (вставить в очередь, удалить из очереди) будут вызываться только методами класса `ClientsQueue` и не должны быть дос-

тупны извне, поэтому класс Queue наследуется со спецификатором доступа private.

Класс ClientsQueue наследует также поведение класса EventsServers, так как определяет обработку события «приход клиента», предоставляя собственный вариант виртуальной функции ServeEvent. Эта функция должна быть открытой, поэтому наследование производится со спецификатором public.

```
class Managers; //объявление используемого класса
//Класс ClientsQueue
class ClientsQueues: public EventsServers, private Queue
{
   private:
         EventsGenerators *pEventsGenerator;
         Managers *pManager;
   public:
         ClientsQueues(EventsGenerators *, Managers *);
         void ServeEvent();
         Clients * GetClient(void);
         ~ClientsQueues();
};
```

Данными класса являются указатель на генератор событий pEventsGenerator (для заказа следующего события «приход клиента») и указатель на менеджера pManager (для запроса на обслуживание пришедшего клиента при пустой очереди).

Методы класса:

конструктор ClientsQueues – инициализирует данные класса, рассчитывает время прихода первого клиента и заказывает это событие вызовом функции EventsGenerators::PlanNewEvent;

ServeEvent – обрабатывает событие «приход клиента»: если очередь не полна, создает нового клиента (в динамической памяти), заносит в объект информацию о времени прихода и длине очереди. Далее, если очередь не пуста, вставляет клиента в очередь, в противном случае (при пустой очереди) запрашивает менеджера о приеме клиента на обслуживание вызовом Managers::TakeClient и, если клиент не принимается, вставляет его в очередь. После всего этого заказывает приход следующего клиента вызовом EventsGenerators::PlanNewEvent;

GetClient – если очередь не пуста, возвращает указатель на клиента, извлекая его из очереди. В противном случае возвращает NULL;

деструктор ~ClientsQueues – очищает очередь, удаляя клиентов.

Класс Servers. Являясь обработчиком события «окончание обслуживания», класс наследует от класса EventsServers, предоставляя собственный вариант виртуальной функции ServeEvent.

```
//Класс Servers
class Servers: public EventsServers
{
```

```
        private:
        unsigned Name;
             EventsGenerators *pEventsGenerator;
             Managers *pManager;
             ofstream *pServerOutFile;
             Clients * pClient;
             unsigned NumberOfServedClients;
             unsigned ClientEnterTime, ClientExitTime;
             void SaveInformation();
        public:
             Servers(Managers* m, EventsGenerators * eg,
             unsigned );
             int TakeClient(Clients *);
             void ServeEvent();
};
```

Компонентными данными являются:

`Name` – имя (номер) работника;

`pEventsGenerator` – указатель на генератор событий (для заказа события «окончание обслуживания»);

`pManager` – указатель на менеджера (для запроса нового клиента);

`pServerOutFile` – указатель на поток вывода, связанный с файлом, в котором фиксируется информация о загрузке работника в течение рабочего дня);

`pClient` – указатель на обслуживаемого клиента (принимает значение `NULL`, если работник свободен);

`NumberOfServedClients` – число обслуженных клиентов;

`ClientEnterTime`, `ClientExitTime` – время прихода клиента на обслуживание и время окончания обслуживания.

Методы класса:

закрытый метод `SaveInformation` – вызывается по окончанию обслуживания очередного клиента и заносит в файл информацию о нем;

конструктор `Servers` – инициализирует компонентные данные, в том числе создает файл с именем `ServerI.txt` (где `I` – номер работника) и заносит в него заголовок таблицы;

`TakeClient` – если работник свободен, принимает клиента от менеджера, заносит информацию о клиенте в структуры клиента и собственную, рассчитывает время окончания обслуживания и заказывает у генератора событий соответствующее событие;

`ServeEvent` – заносит информацию о клиенте в структуры клиента и собственную, увеличивает `NumberOfServedClients`, вызывает `SaveInformation`, удаляет клиента, запрашивает у менеджера следующего клиента вызовом `Managers::GetClient`; если клиент получен, обновляет данные клиента и работника, рассчитывает время освобождения и заказывает событие «окончание обслуживания» вызовом `EventsGenerators::PlanNewEvent`.

Класс `Managers`

```
//Класс Managers
```

```
class Managers
{
    private:
         ClientsQueues *pClientsQueue;
         unsigned NumberOfServers;
         Servers * ServersRecord[10];
   public:
         Managers(EventsGenerators *, unsigned);
         int TakeClient(Clients *);
         Clients * GetClient(void);
         ~Managers();
};
```

Компонентными данными являются:

`pClientsQueue` – указатель на очередь клиентов;

`NumberOfServers` – число работников;

`ServersRecord` – массив указателей на работников – объектов класса `Servers`;

Методы класса:

конструктор `Managers` – создает в динамической памяти очередь клиентов – объект класса `ClientsQueues` и настраивает на него указатель `pClientsQueue`, создает в динамической памяти работников – объекты класса `Servers` и заносит указатели на них в массив `ServersRecord`;

`TakeClient` – принимает запрос от очереди об обслуживании клиента, опрашивает всех работников и передает клиента первому свободному работнику. Если такой работник найдется, возвращает 1, в противном случае 0 (клиент не принят на обслуживание);

`GetClient` – обслуживает запрос от освободившегося работника. Вызовом `ClientsQueues::GetClient` получает указатель на клиента от очереди или `NULL`, если очередь пуста, и возвращает это как результат.

деструктор `~Managers` – уничтожает очередь клиентов и работников.

Схема взаимодействия основных функций модели представлена на рис.10.

После определения всех методов класса запуск процесса моделирования можно осуществить следующей главной функцией:

```
void main(void)
{
    // инициализация генератора случайных чисел
    srand(time(0));
    EventsGenerators EG; // генератор событий
    Managers MN(&EG,NOS); // менеджер
    EG.Procces(); // запуск процесса
}
```

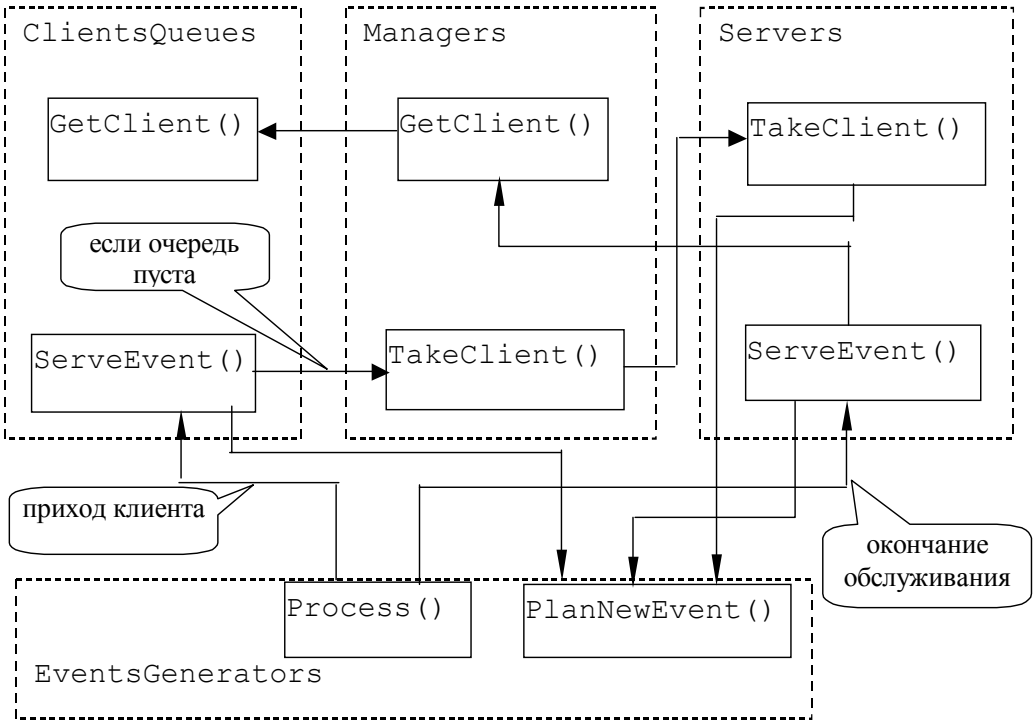


Рис.10. Схема взаимодействия функций модели

7.4. Задания

1,2. Реализовать функции:

```
EventsGenerators::EventsGenerators(void);
void EventsGenerators::PlanNewEvent(EventsServers *EvServer,
                                    unsigned EvTime);
```

3,4. Реализовать функции:

```
unsigned EventsGenerators::GetTime(void);
void EventsGenerators::Procces(void);
```

5,6. Реализовать функции:

```
Clients::Clients(void);
Clients::~Clients();
```

7. Реализовать функцию

```
ClientsQueues::ClientsQueues(EventsGenerators *pEvGen,
                             Managers *pMan);
```

8,9. Реализовать функцию

```
void ClientsQueues::ServeEvent();
```

10. Реализовать функции:

```
Clients * ClientsQueues::GetClient(void);
ClientsQueues::~ClientsQueues();
```

11,12. Реализовать функцию

```
void Servers::SaveInformation();
Servers::Servers(Managers *pM, EventsGenerators *pEvGen,
                 unsigned n);
```

13,14. Реализовать функцию

```
int Servers::TakeClient(Clients * pCl);
```

15,16. Реализовать функцию

```
void Servers::ServeEvent();
```

17. Реализовать функцию

```
Managers::Managers(EventsGenerators *pEvGen, unsigned sn);
```

18. Реализовать функцию

```
int Managers::TakeClient(Clients * pCl);
```

19,20. Реализовать функции:

```
Clients * Managers::GetClient(void);
Managers::~Managers();
```

ЛИТЕРАТУРА

1. *Амелина Н.И., Демьяненко Я.М.* и др. Задачи по программированию. М.: Вузовская книга, 2000. 104 с.

2 *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на С++. 2-е изд./ Пер с англ. М.: Изд-во «Бином»; СПб.: Невский диалект, 2000. 506 с.

3. *Емельянов В.В., Ясиновский С.И.* Введение в интеллектуальное имитационное моделирование сложных дискретных систем и процессов. Язык РДО. М.: АНВИК, 1998. 427 с.

4. Методы программирования: Учеб. пособие / Минакова Н.И., Невская Е.С., Угольницкий Г.А. и др.; под ред. Угольницкого Г.А. 2–изд. М.: Вузовская книга, 2000. 280 с.

5. *Подбельский В.В.* Язык С++: Учеб. пособие. 5–е изд. М.: Финансы и статистика, 2000. 506 с.

6. *Топп У., Форд У.* Структуры данных с С++/ Пер. с англ. М.: Изд-во «Бином», 1999. 816 с.

СОДЕРЖАНИЕ

Андрей Николаевич Рыбалев,
доцент кафедры АПП и Э АмГУ,
канд. техн. наук

ПРОГРАММИРОВАНИЕ И ОСНОВЫ АЛГОРИТМИЗАЦИИ.
Лабораторный практикум

Изд-во АмГУ. Подписано к печати ?.?.02. Формат 60×84/16. Усл. печ. л. 5,35, уч. - изд. л. 5,5. Тираж 100. Заказ 96.