Н.В. Пацей, Н.Н. Дорожкина

КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Практикум для студентов
специальности 1-40 01 02
«Информационные системы и технологии»

В 2-х частях

Часть 2

Минск БГТУ 2006

Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Н.В. Пацей, Н.Н. Дорожкина

КОНСТРУИРОВАНИЕ ПРОГРАММ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ

**Практикум для студентов
специальности 1-40 01 02
«Информационные системы и технологии»**

В 2-х частях

Часть 2

Минск 2006

УДК 004.4(076.5)
ББК
П

Рассмотрено и рекомендовано к изданию редакционно-издательским советом университета

Рецензенты:
доцент кафедры мат. обеспечения АСУ БГУ, кандидат технических наук *И. В. Совпель*;
доцент кафедры ИТАС БГУИР, кандидат технических наук *О. В. Герман*

Пацей Н.В., Дорожкина Н.Н.

Конструирование программ и языки программирования: практикум для студентов специальности 1-40 01 02. «Информационные системы и технологии»: в 2 ч. Ч 2. / Пацей Н.В., Дорожкина Н.Н. – Мн.: БГТУ, 2006. – 117 с.

ISBN

В пособии рассматриваются основные возможности языка С++ при работе с динамическими структурами данных и разработке объектно-ориентированных программ по дисциплине «Конструирование программ и языки программирования», приводятся примеры и задания для самостоятельного выполнения.

УДК 004.4(076.5)
ББК

ISBN
ISBN

ПРЕДИСЛОВИЕ

Объектно-ориентированное программирование (ООП) является способом организации программы. Основное внимание при его изучении уделяется организации программы, а не вопросам написания кода. Главным компонентом объектно-ориентированной программы является объект, содержащий данные и функции для их обработки. Класс является формой или образцом для множества сходных между собой объектов.

Механизм наследования позволяет создавать новые классы на основе существующих классов, не внося изменений в последние. Наследование делает возможным повторное использование кода, то есть включение однажды созданного класса в любые другие программы.

C++ является расширением языка C, позволяющим реализовать концепцию ООП, а также включающим в себя некоторые дополнительные возможности.

Пособие является второй частью практикума по дисциплине «Конструирование программ и языки программирования» для специальности «Информационные системы и технологии» и предназначено для студентов, освоивших основы программирования на языке C/C++, позволит изучить динамические структуры и объектно-ориентированные средства языка C++.

Практикум содержит теоретический материал по темам: динамические структуры данных, рекурсия, бинарные деревья, объекты и классы, наследование, полиморфизм, виртуальные функции, шаблоны, потоки, STL, обработка исключений, создание Win32 API, и варианты заданий для самостоятельного выполнения.

ВВЕДЕНИЕ

Объектно-ориентированное программирование (ООП) является способом организации программы. Основное внимание при его изучении уделяется организации программы, а не вопросам написания кода. Главным компонентом объектно-ориентированной программы является объект, содержащий данные и функции для их обработки. Класс является формой или образцом для множества сходных между собой объектов.

Механизм наследования позволяет создавать новые классы на основе существующих классов, не внося изменений в последние. Порожденный класс наследует все данные и методы своего родителя, но имеет также и свои собственные. Наследование делает возможным повторное использование кода, то есть включение однажды созданного класса в любые другие программы.

C++ является расширением языка C, позволяющим реализовать концепцию ООП, а также включающим в себя некоторые дополнительные возможности.

Пособие является второй частью практикума по дисциплине «Конструирование программ и языки программирования» для специальности «Информационные системы и технологии» и предназначено для студентов, освоивших основы программирования на языке C/C++, позволяет изучить динамические структуры и объектно-ориентированные средства языка C++.

Практикум содержит теоретический материал по темам: динамические структуры данных, рекурсия, бинарные деревья, объекты и классы, наследование, полиморфизм, виртуальные функции шаблоны, потоки, STL, обработка исключений, создание Win32 API, и варианты заданий для самостоятельного выполнения.

Практикум №1. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

Краткие теоретические сведения

Динамическая структура данных представляет собой множество переменных, связанных между собой указателями. Каждый элемент такой структуры содержит один или несколько указателей на аналогичные элементы:

```
struct      list
{     int   value;
      list *next;       // указатель
      list *next,*pred; // два указателя
      list *links[10];  // ограниченное кол-во указателей
      list **plinks;    // произвольное кол-во указателей
};
```

Из данного описания нельзя определить ни количества переменных в структуре данных, ни характера связи между ними (последовательный, циклический, произвольный). Следовательно, конкретный тип структуры данных (список, дерево, граф) зависит от функций, которые работают с этой структурой.

Рассматриваемые ниже структуры данных являются динамическими по двум причинам: сами переменные таких структур создаются как динамические переменные; количество связей между переменными и их характер также определяются динамически в процессе работы программы.

Последовательность обхода зависит не от физического размещения элементов в памяти, а от последовательности их связывания указателями. Нумерация элементов списка - *логический номер* элемента в списке - номер, получаемый им в процессе движения по списку.

Списковые структуры данных доступны, как правило, через указатель на некоторый ее элемент, который называется *Заголовком*.

- Списки как динамические структуры данных

Наиболее простыми динамическими структурами данных являются списки. Список представляет собой линейную последовательность переменных, каждая из которых связана указателями со своими соседями. Списки подразделяют на:

односвязные - каждый элемент списка имеет указатель на следующий (рис. 1);

двусвязные - каждый элемент списка имеет указатель на следующий и на предыдущий элементы;

двусвязные циклические - первый и последний элементы списка ссылаются друг на друга (рис. 2).

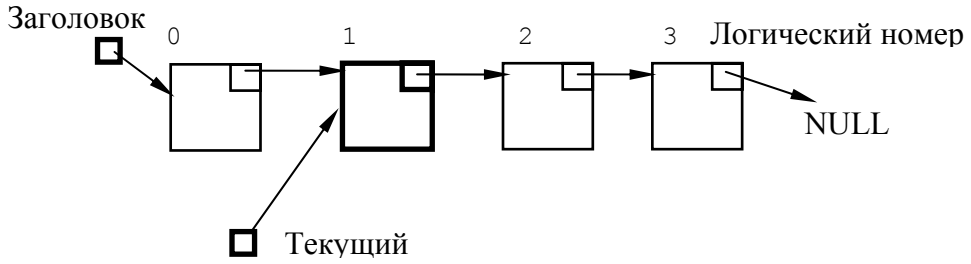


Рис. 1

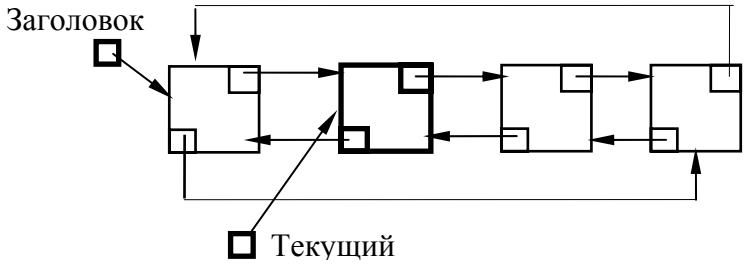


Рис. 2

Односвязные списки используются для моделирования таких структур данных, как стеки и очереди. Двусвязные списки дают возможность просмотра элементов в обоих направлениях и являются наиболее универсальными. Они используются для создания цепочек элементов, которые допускают частые операции включения, исключения, упорядочения, замены и прочие. В односвязных и двусвязных списках последний элемент содержит указатель `NULL` для обозначения факта окончания последовательности.

- Работа со списками

Особенности работы с динамическими структурами данных заключаются в частом использовании операции косвенного обращения по указателю к структуре или к ее элементу - «`*`» или «`->`». При этом обычно используется указатель, который ссылается на текущий элемент структуры данных. Весь просмотр структуры данных заключается в циклическом переходе от одного текущего элемента к другому. Ниже приведен фрагмент кода, выполняющий различные действия с односвязным списком:

```
struct      list
      {
```

```
      list *next;
      int   val;
      }     *ph;         // заголовок списка
struct      list  *p;    // указатель на текущий элемент
p = ph;                  // текущий указатель - на первый
p->next ...              // указатель на следующий элемент
p = p->next;             // переход к следующему элементу
p !=NULL ...             // проверка на конец списка
p->next ==NULL ...       // проверка на последний элемент
for (p=ph; p !=NULL; p=p->next)...
                         // просмотр элементов списка
if (ph !=NULL)
      {
      for (p=ph; p->next !=NULL; p=p->next);
      }                  // поиск последнего элемента
list *pred;              // просмотр с сохранением
                         // указателя на предыдущий элемент
for (pred=NULL, p=ph; p !=NULL; pred=p, p=p->next) ...
                         // указатель на второй элемент от текущего
ph->next = pnew;         // включение в начало непустого списка
ph = pnew;
pnew->next = NULL;       // включение в конец непустого списка
pred->next = pnew;
pred->next = p->next;    // исключение текущего элемента
                         // списка (если он не первый)
pnew->next = p->next;    // включение после текущего элемента
p->next = pnew;
```

Далее приведен фрагмент кода, выполняющий различные действия с двухсвязным списком:

```
struct      list2
      {
      list2 *next,*pred;
      int   val;
      } *ph, *p, *pnew;
pnew->pred = p;          // включение после текущего элемента
pnew->next = p->next;
p->next->pred = pnew;
p->next = pnew;
p->pred->next = p->next; // исключение текущего элемента
p->next->pred = p->pred; // из середины списка.
```

- Способы формирования списков

Хотя списки представляют собой динамические структуры данных, способы их формирования могут быть различными. Рассмотрим их на примере односвязного списка.

Способ 1. Элементы - обычные переменные, связи инициализируются транслятором:

```
struct list
      {
      list *next;
```

```
      int   val;
      } a={0,NULL}, b={1,&a}, c={2,&b}, *ph = &c;
```

Способ 2. Элементы списка создаются в обычном массиве, поэтому их количество ограничено. Связи устанавливаются динамически:

```
struct      list  S[100],*ph; // создать список из
for (i=0; i<99; i++)          // элементов, размещенных
      {                       // в статическом массиве
      S[i].next = S+i+1;
      S[i].val = i;
      }
S[99].next = NULL;
ph = S;
```

Способ 3. Элементы списка являются динамическими переменными, связи между ними устанавливаются программно:

```
list  *InsertFirst(list *ph, int n)
//       Включить в начало списка
//    ph - заголовок списка ,  n - значение элемента списка
{ list *pnew;
      pnew = new list;              // создать элемент списка -
      pnew->val = n;                // динамическую переменную,
      pnew->next = *ph;             // заполнить его и
      ph = pnew;                    // поместить в начало списка
      return ph;                    // вернуть новый заголовок
}
```

- Представление очереди и стека односвязным списком

При помощи односвязного списка можно моделировать такие структуры как стек и очередь. *Стек* или *LIFO* (Last In First Out) структура - упорядоченный набор элементов, в котором размещение новых элементов и удаление существующих производится с одного конца. Допустимы операции: добавление в стек - push; удаление из стека - pop; чтение элемента с вершины стека (без извлечения) - peek.

В представленном ниже примере стек представлен динамической структурой:

```
#define STACK struct stack
STACK {
      int info;
      STACK * next;
      };
extern void push (STACK **PPStack, int nItem);
extern int pop (STACK **PPStack, int *nError);
extern int peek (STACK **PPStack, int *nError);
void push (STACK **PPStack,int nItem)
      {
        STACK *pNewItem; // запрашиваем память под структуру для стека
        pNewItem=(STACK*)malloc(sizeof(STACK));
```

```
        pNewItwem->info=nItem; //заполяем поля структуры
        pNewItwem->next=*ppStack;
                  // устанавливаем новый указатель на вершину стека
        *ppStack=pNewItem;
      }
int pop (STACK **PPStack,int *nError)
      {
        STACK *pOldItem=*ppStack; // запоминаем старый адрес вершины
        int nOldInfo=0;
        if(*ppStack)
            {
            nOldInfo=pOldInfo->info; // если стек не пустой, извлекаем
            *ppStacj=(*ppStack)->next;
            free(pOldItem); // и освобождаем память
            *nError=0;
            }
        else            // в противном случае ошибка
            *nError=1;
      return nOldInfo;
      }
int peek (STACK **PPStack,int *nError)
      {
        if (*ppStack)
            { *nError=0;
            return (*ppStack)->info;
            }
        else
            {*nError=1;
            return 1;
            }
      }
```

Очередь или *FIFO* (First In First Out) структура - упорядоченный набор элементов, которые удаляются с одного конца, а помещаются в другой конец. Допустимы операции включения и исключения из очереди.

- Двусвязные списки

В качестве примера рарботы с двухсвязным списком рассмотрим включение в список нового элемента с сохранением упорядоченности значений:

```
struct      list2
      {
      list2 *next, *pred;
      int   val;
      };
list2 *InsertSort(list2 *ph, int v)
{
list2 *q, *p=new list2;
p->val = v;
p->pred = p->next = NULL;
if (ph == NULL) return p;// включение в пустой список
for (q=ph; q !=NULL &&  v > q->val; q=q->next);
```

```
                              // поиск места включения - q
if (q == NULL)                // включение в конец списка
      {                       // восстановить указатель на
      for (q=ph; q->next!=NULL; q=q->next);
      p->pred = q;            // последний
      q->next = p;
      return ph;
      }                 // включить перед текущим
p->next=q;              // следующий за новым = текущий
p->pred=q->pred;        // предыдущий нового = предыдущий текущего
if (q->pred == NULL)    // включение в начало списка
      ph = p;
else                    // включение в середину
      q->pred->next = p;// следующий за предыдущим = новый
q->pred=p;              // предыдущий текущего = новый
return ph;
}
```

- Проблема концов списка и циклические списки

Основной сложностью при работе со списками является необходимость проверки множества вариантов при выполнении операций над элементом списка: список пустой; элемент единственный; элемент в начале списка; элемент в конце списка; элемент в середине списка.

Для решения проблемы «концов списка» удобно бывает сделать его циклическим, то есть вместо указателей `NULL` записывать:

- указатель на последний элемент в качестве указателя на предыдущий в первом элементе списка;

- указатель на первый элемент в качестве указателя на последующий в последнем элементе списка.

Тогда, единственный элемент в циклическом списке будет иметь указатели на самого себя, а в процессе просмотра конец списка определяется фактом возвращения на его начало. В качестве примера рассмотрим функции просмотра циклического списка и включения элемента в его конец:

```
void  scan(list2 *ph)
{
      list2 *p;
      if (ph ==NULL) return;
      p = ph;
      do    { /* ... */
            p = p->next;
            }
      while (p != ph);  // возврат к началу списка
}
//включение в конец списка
list2 *insert(list2 *ph, int v)
{
list2 *p = new list;
```

```
p->val = v;
p->next = p->pred = p;  // указатели на самого себя
if (ph ==NULL)
      return p;         // включение в пустой список
list2 *pr;              // текущий и предыдущий элементы списка
pr = ph->pred;          // в циклическом списке  предыдущий
                        // от первого - последний
pr->next = p;           // следующий за последним = новый
ph->pred = p;           // предыдущий для первого = новый
p->pred = pr;           // предыдущий для нового = последний
p->next = ph;           // следующий за новым = первый
return ph;
}
```

Задания для выполнения

Таблица 1

Записи в линейном списке содержат ключевое поле типа `int`. Сформировать однонаправленный список.	
1	Удалить из него элемент с заданным номером, добавить элемент с заданным номером.
2	Удалить из него элемент с заданным ключом, добавить элемент перед элементом с заданным ключом.
3	Удалить из него `k` элементов, начиная с заданного номера, добавить элемент перед элементом с заданным ключом.
4	Удалить из него элемент с заданным номером, добавить `k` элементов, начиная с заданного номера.
5	Удалить из него `k` элементов, начиная с заданного номера, добавить `k` элементов, начиная с заданного номера.
Записи в линейном списке содержат ключевое поле типа `int`. Сформировать двунаправленный список.	
6	Удалить из него элемент с заданным номером, добавить элемент в начало списка.
7	Удалить из него первый элемент, добавить элемент в конец списка.
8	Удалить из него элемент после элемента с заданным номером, добавить `k` элементов в начало списка.
9	Удалить из него `k` элементов перед элементом с заданным номером, добавить `k` элементов в конец списка.
10	Добавить в него элемент с заданным номером, удалить `k` элементов из конца списка.
Записи в линейном списке содержат ключевое поле типа `*char`(строка символов). Сформировать двунаправленный список.	
11	Удалить из него элемент с заданным ключом, добавить элемент с указанным номером.
12	Удалить из него элементы, с одинаковыми ключевыми полями. Добавить элемент после элемента с заданным ключевым полем.

Продолжение таблицы 1

13	Удалить из него `k` первых элементов. Добавить элемент после элемента, начинающегося с указанного символа.
14	Удалить из него `k` элементов с указанными номерами. Добавить `k` элементов с указанными номерами.
15	Удалить `k` элементов из конца списка. Добавить элемент после элемента с заданным ключом.
16	Удалить элемент с заданным ключом. Добавить `k` элементов в конец списка. Напишите программу вывода текса (символов) «елочкой».
Выполнить задание	
17	Реализовать шейкер-сортировку двусвязного циклического списка, используя указатели на границы отсортированных частей списка.
18	Элемент односвязного списка содержит указатель на строку. Строки упорядочены по возрастанию. Вставить строку в список с сохранением упорядоченности. В список помещается копия входной строки в динамической памяти.
19	Элемент двусвязного циклического списка содержит указатель на строку. Строки упорядочены по возрастанию. Вставить строку в список с сохранением упорядоченности. В список помещается копия входной строки в динамической памяти.
20	Элемент односвязного списка содержит указатель на строку. Отсортировать список путем исключения максимального элемента и включения в начало нового списка.
21	Массив указателей содержит заголовки односвязных списков, элемент списка содержит целое. Целые в списках и в массиве указателей упорядочены. Включить новое целое с сохранением упорядоченности.
22	Элемент односвязного списка содержит массив указателей на строки. Написать функцию возвращения указателя на строку максимальной длины.
23	Элемент двусвязного циклического списка содержит массив указателей на строки, последовательность строк в структуре данных упорядочена. Написать функцию поиска строки по заданному образцу, с которым совпадает ее начало, не сравнивая все строки (с учетом возрастания). Функция возвращает указатель на копию найденной строки в динамической памяти.
24	Написать функции для работы со стеком: добавление элемента в стек; удаление элемента из стека. Стек реализовать с использованием динамического массива.
25	Написать функцию сортировки односвязного списка путем исключения элемента с минимальным значением и включения его в начало нового списка.

Окончание таблицы 1

26	Элемент двусвязного циклического списка содержит указатель на строку в динамической памяти. Написать функции просмотра списка и включения очередной строки с сохранением упорядоченности по длине строки и по алфавиту.
27	Элемент односвязного списка содержит массив из 4 целых переменных. Массив может быть заполнен частично. Все значения целых переменных хранятся в порядке возрастания. Написать функцию включения значения в элемент списка с сохранением упорядоченности. При переполнении массива создается новый элемент списка и в него включается половина значений из переполненного.
28	Элемент односвязного списка содержит массив указателей на строки. Строки читаются из текстового файла функцией fgets и указатели на них помещаются в структуру данных. Элементы списка и сами строки должны создаваться в динамической памяти в процессе чтения файла. В исходном состоянии структура данных - пуста.
29	Элемент односвязного списка содержит указатель на строку. Вставить строку в конец списка. В список помещается копия входной строки в динамической памяти.
30	Элемент односвязного списка содержит указатель на строку. Строки упорядочены по возрастанию. Вставить строку в список с сохранением упорядоченности. В список помещается копия входной строки в динамической памяти.

Контрольные вопросы

1. Как можно получить доступ к списку?

2. В чем суть удаления элемента из односвязного списка? Поясните на примере.

3. Перечислите преимущества и недостатки списков в сравнении с обычными массивами и массивами указателей.

4. Обращение к полю динамической структуры выглядит следующим образом: `p->key=i`. Какова вторая возможность обращения?

5. Нарисуйте схему построения цикличнского, односвязного и двусвязного списков.

Практикум №2. РЕКУРСИЯ

Краткие теоретические сведения

Рекурсивным называется способ построения объекта (понятия, системы), в котором определение объекта включает аналогичный объект в виде некоторой его части. Примеры: рекурсивное определение в синтаксисе языка; рекурсивная структура данных - элемент структуры данных содержит один или несколько указателей на такую же структуру данных (односвязный список); рекурсивная функция (тело функции содержит прямой или косвенный собственный вызов).

Очевидно, что рекурсия не может быть безусловной, в этом случае она становится бесконечной. Рекурсия должна иметь внутри себя условие завершения, по которому очередной рекурсивный вызов уже не производится.

- Особенности программирования рекурсивных функций

Функция называется рекурсивной, если ее значение для данного аргумента определяется через значения той же функции для предшествующих аргументов.

Рассмотрим приемр вызова рекурсивной функции `s`. Ддвижемся по ее тексту до тех пор, пока не встретим ее вызова, после чего мы опять начинает выполняться та же самая функция сначала, при этом ее первый вызов еще не закончился. Создается впечатление, что текст функции воспроизводится (копируется) всякий раз, когда функция сама себя вызывает:

```
.void main()      void S()    void S()    void S()
{                 {           {           {
S();              ..if()S();  ...if()S(); ...if()S();
}                 }           }           }
```

Копируется при этом не весь текст функции (не вся функция), а только ее части, связанные с данными (формальные, фактические параметры, локальные переменные и точка возврата). Создается копия локальных данных в следующей последовательности:

- в стеке резервируется место для формальных параметров, в которые записываются значения фактических параметров. Обычно в порядке, обратном их следованию в списке;

- при вызове функции в стек записывается точка возврата - адрес той части программы, где находится вызов функции;

- в начале тела функции в стеке резервируется место для локальных (автоматических) переменных.

Алгоритм (операторы, выражения) рекурсивной функции не меняется, поэтому он присутствует в памяти компьютера в единственном экземпляре. Таким образом, формальные параметры рекурсивной функции, внешние и локальные переменные не могут быть взаимозаменяемы. Кроме того, каждый новый рекурсивный вызов порождает новый «экземпляр» формальных параметров и локальных переменных, причем старый «экземпляр» не уничтожается, а сохраняется в стеке по принципу вложенности. Здесь имеет место единственный случай, когда одному имени переменной в процессе работы программы соответствуют несколько ее «экземпляров» (рис. 3).

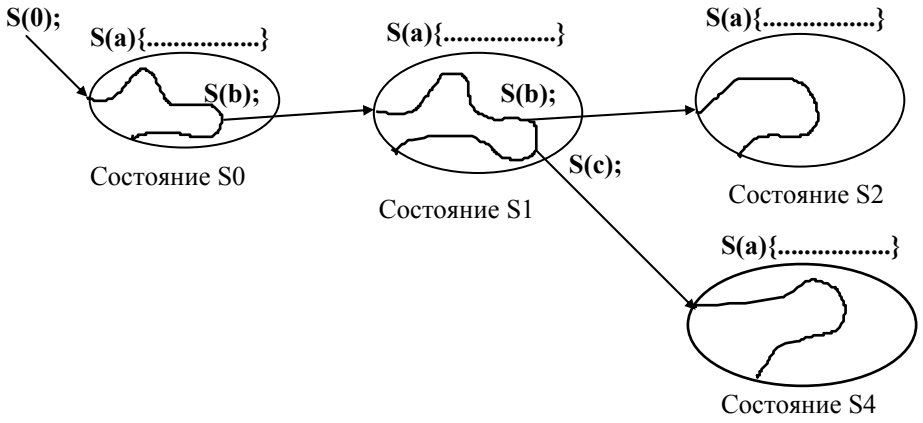


Рис. 3

При завершении рекурсии программа возвращается к предыдущей версии рекурсивной функции и к предыдущему фрейму в стеке.

Рекурсивные функции используются для обработки рекурсивных структур данных. Здесь обработка вложенного элемента структуры данных естественным образом предполагает рекурсивный вызов функции с передачей указателя на него. Однако рекурсивным может быть и любой алгоритм, в котором имеется линейная или разветвляющаяся цепочка шагов, в каждом из которых выполняется одинаковая последовательность действий.

- Линейная рекурсия

Простейшим примером рекурсии является линейная рекурсия, когда функция содержит единственный условный вызов самой себя. В таком случае рекурсия становится эквивалентной обычному циклу. Любой циклический алгоритм можно преобразовать в линейно-рекурсивный и наоборот. Пример вычисления факториала:

```
int fact(int n)
      {
      if (n==1) return 1;
      return n * fact(n-1);
      }
```

- Рекурсия и поисковые задачи

С помощью рекурсии легко решаются задачи, связанные с поиском, основанном на полном или частичном переборе возможных вариантов. Принцип рекурсивности заключается здесь в том, что процесс поиска разбивается на шаги, на каждом из которых выбирается и проверяется очередной элемент из множества, а алгоритм поиска повторяется, но уже для «оставшихся» данных. При этом вовсе не важно, каким образом цепочка шагов достигнет цели и сколько вариантов будет перебираться. Единственное, что важно - корректность очередного шага.

Само множество, в котором производится поиск, обычно реализуется в виде глобальных данных, в которых каждый шаг выбирает необходимые элементы, а по завершении поиска возвращает их обратно.

При реализации поисковых задач следует обратить внимание на результат рекурсивной функции. Он непосредственно связан с организацией процесса поиска и перебора вариантов:

- если рекурсивная функция имеет результат `void`, то она не может повлиять на характер протекания процесса поиска, и реализуемый алгоритм выполнит полный перебор всех возможных вариантов;

- если рекурсивная функция выполняет поиск первого попавшегося варианта, то результатом ее является, как правило, логическое значение. При этом «ИСТИНА» соответствует успешному завершению поиска, а «ЛОЖЬ» - неудачному;

-если в процессе поиска производится более сложный анализ и сравнение вариантов, то рекурсивная функция и, соответственно, шаг процесса должны производить выбор между подходящими вариантами, а целью выбора наиболее оптимального. Обычно для этого используется минимум или максимум какой-либо характеристики выбираемого варианта.

Открытым остается вопрос о том, как возвратить саму цепочку выбранных из множества элементов, дающих оптимальный результат. Здесь также возможны варианты:

- рекурсивная функция сама выводит выбранный элемент в случае успешного поиска;

- выбранный элемент записывается в область глобальных данных, в которых моделируется стек для возврата результата;

- в качестве результата функции используются более сложные, динамические структуры данных, например, список результатов.

- Поиск выхода в лабиринте

Представим лабиринта в виде двумерного массива, в котором значение 1 будет обозначать «стенку», а 0 - «проход»:

```
int LB[10][10]={
{1,1,0,1,1,1,1,1,1,1},
{1,1,0,1,1,1,1,1,1,1},
{1,1,0,0,1,0,0,0,1,1},
{1,1,1,0,0,0,1,0,1,1},
{1,0,1,1,1,0,0,0,1,1},
{1,0,0,0,0,0,1,1,1,1},
{1,1,1,1,1,0,1,1,1,1},
{1,1,1,1,1,0,0,0,1,1},
{1,1,1,1,1,1,1,1,1,1},
{1,1,1,1,1,1,1,1,1,1}};
```

Шаг алгоритма состоит в проверке возможности сделать ход в одном из четырех направлений. Рекурсивный характер алгоритма состоит в том, что в каждой соседней точке реализуется тот же самый алгоритм поиска. Формальными параметрами рекурсивной функции являются координаты точки, из которой в данный момент осуществляется поиск. Фактические параметры - координаты соседней точки, в которой реализуется рекурсивный алгоритм:

```
void step(int x,int y)
{
      step(x+1,y);...
      step(x,y+1);...
      step(x-1,y);...
      step(x,y-1);...
}
```

Естественно, что алгоритм не должен возвращаться в ту точку, из которой он только что пришел. Необходимо отмечать все пройденные точки лабиринта.

- Обход шахматной доски

В качестве примера рассмотрим проектирование функции для поиска последовательности обхода конем шахматной доски.

Алгоритм обхода можно разбить на последовательность шагов. Каждый шаг характеризуется начальным состоянием - текущим положением коня. Алгоритм на каждом шаге предполагает проверку возможности сделать ход в каждом из 8 допустимых направлений. Схема алгоритма имеет следующий вид:

```
void  step(int x0, int y0)
{                  // формальные параметры -
                   // текущая позиция коня
                   // приращения координат для 8 ходов
static      int xy[8][2] = {{ 1,-2},{ 1, 2},{-1,-2},{-1, 2},
                   { 2,-1},{ 2, 1},{-2, 1},{-2,-1}
                   };
int   i;           // локальный параметр - номер хода
if (x < 0 || x >=7 || y < 0 || y >=7 )
      return;      // выход за пределы доски
for (i=0; i<8; i++)
      step(x0+xy[i][0], y0+xy[i][1]);
                   // выполнить следующий ход
}
```

Предыдущий шаг дает полный перебор возможных последовательностей ходов коня. Функция имеет результат `void`, то есть является безусловной. Необходимо определить, как производить выбор необходимой последовательности. Очевидно, функция должна давать логический результат - 0, если при выполнении очередного хода в данном направлении задача не решается и 1, если решается успешно. При этом цикл, в котором происходит рекурсивный вызов - просмотр очередных ходов, выполняется до первого успешного вызова.

Необходимо определить условия начала и завершения рекурсивного процесса, а также условия взаимодействия шагов между собой. Условием завершения процесса является обход всех 64 полей доски. При этом не должно производиться повторное попадание на то же самое поле. Необходимо фиксировать и отмечать количество пройденных полей. Это можно сделать используя внешние переменные, так как они проверяются на разных шагах рекурсии. Номер шага будем использовать также для отметки поля. По завершению программы массив полей будет содержать номера шагов коня:

```
int   desk[8][8];             // поля доски
int   nstep;                  // номер шага
int   step(int x0, int y0)
{
static      int xy[8][2] = {{ 1,-2},{ 1, 2},{-1,-2},{-1, 2},
                   { 2,-1},{ 2, 1},{-2, 1},{-2,-1}
                   };
int   i;                // локальный параметр - номер хода
if (x0 < 0 || x0 >=7 || y0 < 0 || y0 >=7 )
      return(0);        // выход за пределы доски
if (desk[x0][y0] !=0)
      return(0);        // поле уже пройдено
desk[x0][y0] = nstep++; // отметить свободное поле
if (nstep == 64)
      return(1);        // все поля отмечены - успех
```

```
for (i=0; i<8; i++)
    if (step(x0+xy[i][0], y0+xy[i][1]))
      return(1);         // поиск успешного хода
nstep--;                       // вернуться на ход назад
desk[x0][y0] = 0;        // стереть отметку поля
return(0);               // последовательность не найдена
}
void  Path(int x0, int y0)
{
int   i,j;               // очистка доски
for (i=0; i<8; i++)
for (j=0; j<8; j++) desk[i][j] =0;
nstep = 1;               // установить номер шага
step(x0,y0);             // вызвать функцию для исходной позиции
}
```

- Линейный кроссворд

Для заданного набора слов требуется построить линейный кроссворд. Если окончание одного слова совпадает с началом следующего более чем в одной букве (например, *МАТРАС-РАСИСТ*), то такие слова можно объединить в цепочку. Первоначально ставится задача - получить любую такую цепочку, окончательно - цепочку минимальной длины.

Рекурсивная функция выполняет попытку присоединения очередного слова к уже выстроенной цепочке. Результатом функции является логическое значение. Условием завершения рекурсии является отсутствие еще не присоединенных к цепочке слов (успешное завершение), либо невозможность завершения цепочки ни через одно из оставшихся слов (неудача).

Множество возможных вариантов строится на основе обычного комбинаторного перебора всех допустимых сочетаний (последовательностей) из элементов множества. Для представления множества можно использовать массив указателей на строки. Исключение строки из множества будет заключаться в установке указателя на строку нулевой длины:

```
char *w[]={"РАСИСТ","МАТРАС","МАСТЕР","СИСТЕМА","СТЕРВА",NULL};

int step(char *lw)            // параметр - текущее слово цепочки
{ int n;
for (n=0; w[n]!=NULL;n++)
    if (*w[n]!=0) break;
if (w[n]==NULL)               // цепочка выстроена, все слова
    return 1;                 // из w[] присоединены
for (n=0; w[n]!=NULL;n++)
    {                         // проверка на присоединение
    char *pw;                 // очередного слова
    if (*w[n]==0) continue;
    pw=w[n];                  // пустое слово - пропустить
    w[n]="";                  // исключить проверяемое слово из
```

```
    if (TEST(lw,pw))          // множества
      {                       // попытка присоединить слово
      if (step(pw))           // присоединено - попытка вывести
          {                   // цепочку из нового слова,
          cout << pw << "\n";
          return 1;           // удача - вывести слово и выйти
          }
      }
    w[n]=pw;                  // возвратить исключенное слово
    }
return 0;
}
```

Функция TEST проверяет, не совпадает ли окончание первой строки с началом второй путем обычного сравнения строк при заданной длине «хвоста» первой строки:

```
int TEST(char *s, char *r)
    { int n,k;
      n=strlen(s);
      if (n==0) return 1;
      for (;*s!=0 && n>1; s++,n--)
        if (strncmp(s,r,n)==0) return 1;
     return 0; }
```

- Результат функции рекурсивного поиска

Выше мы рассматривали варианты рекурсивных функций с логическим результатом. Данные самого варианта (например, путь к выходу из лабиринта) могли храниться в глобальных переменных. Если же допустимых вариантов несколько, то алгоритм усложнится.

Изменения касаются самого принципа формирования результата. Каждый шаг рекурсивного алгоритма получает несколько различных результатов от рекурсивных вызовов, обрабатывает их и формирует собственный результат аналогичного типа. Обработка может заключаться в выборе варианта, соответствующего заданному критерию и его передаче (ретрансляции).

Следующий важный принцип заключается в том, что результат рекурсивной функции должен представлять структуру данных, полностью содержащую все данные о найденном варианте, использование глобальных переменных здесь невозможно. Обычно такие данные имеют переменную размерность и для их представления используются динамические структуры данных.

В C++ для возврата рекурсивной функцией совокупности параметров можно использовать результата функции - структурированную переменную, то есть возвращать структурированный тип по значению. Тогда все проблемы по распределению памяти для временного хранения ре-

зультата функции решаются транслятором. При этом в самой функции можно использовать операции присваивания структурированных переменных.

Задания для выполнения

Таблица 2

1	Написать рекурсивную функцию поиска файлов с одинаковыми именами во всех подкаталогах диска. (Структуру данных, содержащую имена найденных файлов можно реализовать в виде глобального односвязного списка).
2	Реализовать рекурсивный алгоритм построения цепочки из имеющегося набора костей домино.
3	Рекурсивная программа обхода дерева и изображения его вершин на экране. Для равномерного размещения вершин на экране программа должна «знать» для каждой вершины интервал позиций экрана, который выделен для данного поддерева и количество вершин в нем. Само дерево можно задать статически (инициализация).
4	Определить рекурсивную функцию, вычисляющую сумму цифр заданного натурального числа.
5	Определить, является ли заданная строка палиндромом, т.е. читается одинаково слева направо и справа налево. Используйте рекурсию.
6	Определить, является ли заданное натуральное число палиндромом
7	Определить рекурсивную функцию `C(m, n)`, где `0 <= m <= n`, для вычисления биномиального коэффициента C_n^m по следующей формуле $C_n^0 = C_n^n = 1;\ C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$ при $0 < m < n$.

8	Определить рекурсивную функцию Аккермана `Ack(m, n)` по следующей формуле:
	`Ack(0, n) = n+1,`
	`Ack(m, 0) = Ack(m-1,1) при m > 0,`
	`Ack(m, n) = Ack(m-1, Ack(m,n-1)) при m, n > 0.`

9	Расстояния между городами заданы матрицей (если между городами i,j есть прямой путь с расстоянием `N`, то элементы матрицы `A(i,j)` и `A(j,i)` содержат значение `N`, иначе *0*). Написать программу поиска минимального пути для произвольной пары городов.
10	Расстояния между городами заданы матрицей (Если между городами `i,j` есть прямой путь с расстоянием `N`, то элементы матрицы `A(i,j)` и `A(j,i)` содержат значение `N`, иначе *0*). Написать программу поиска минимального пути обхода всех городов без посещения дважды одного и того же города (задача коммивояжера).
11	Задача о восьми ферзях. Разместить на шахматной доске восемь ферзей так, чтобы они не находились «под боем».

Окончание таблицы 2

12	Разместить на шахматной доске максимальное количество коней так, чтобы они не находились друг у друга «под боем».
13	Программа генерирует текст из строки, содержащей определения циклических фрагментов вида «...(12 У попа была собака)...». Константа определяет количество повторений следующей за ней строки. Допускается вложенность фрагментов. Полученный текст помещается в выходную строку.
14	Программа генерирует текст из строки, содержащей определения циклических фрагментов вида «...(Иван, Петр, Федор=Жил-был * у самого синего моря)...» Символ “*” определяет место подстановки имени из списка в очередное повторение фрагмента. Допускается вложенность фрагментов. Полученный текст помещается в выходную строку.
15	Задан набор слов (массив указателей на строки). Построить из них любую цепочку таким образом, чтобы символ в конце слова совпадал с символом в начале следующего.
16	Задан набор слов (массив указателей на строки). Построить из них любую цепочку таким образом, чтобы символ в начале следующего совпадал с одним из символов в середине предыдущего (не первым и не последним).
17	Задан массив целых. Построить из них любую последовательность таким образом, чтобы последняя цифра предыдущего числа совпадала с первой цифрой следующего.
18	Разместить на шахматной доске максимальное количество слонов и ладей так, чтобы они не находились друг у друга «под боем».
19	Задача раскраски карты. Страны на карте заданы матрицей смежности. Если страны `i,j` имеют на карте общую границу, то элемент матрицы `A[i,j]` равен 1, иначе 0. Смежные страны не должны иметь одинакового цвета. «Раскрасить» карту минимальным количеством цветов.
20	Задача проведения границы на карте («создание военных блоков»). Страны на карте заданы матрицей смежности. Если страны `i,j` имеют на карте общую границу, то элемент матрицы `A[i,j]` равен 1, иначе 0. Необходимо разбить страны на две группы так, чтобы количество пар смежных стран из противоположных групп было минимальным.

Контрольные вопросы

1. В каких случаях применяется рекурсия?
2. Какая функция называется рекурсивной?
3. Дает ли рекурсия экономию памяти?
4. Приводит ли рекурсия к созданию более быстрых программ?
5. Как возвращает результат рекурсивная функция?

Практикум №3-4. БИНАРНЫЕ ДЕРЕВЬЯ

Краткие теоретические сведения

- Рекурсивные структуры данных

По аналогии с рекурсивным вызовом функции существуют структуры данных, допускающие рекурсивное определение: элемент структуры данных содержит один или несколько указателей на элементы такого же типа. Из подобных структур данных были рассмотрены списки. Рекурсивные структуры данных естественным образом обрабатываются рекурсивными функциями. В качестве параметра такие функции получают указатель на некоторый элемент. Если этот элемент содержит корректные (не равные NULL) указатели на другие элементы и если алгоритм требует их просмотра, то данная функция вызывается рекурсивно с параметром - указателем на новый элемент. Для списка это выглядит следующим образом:

```
struct list
      {
      list *next;
      int   val;
      };
void  scan(list *p)     // просмотр списка
{
      if (p == NULL) return;  // указатель NULL - конец списка
      cout << p->val << endl;
      scan(p->next);          // рекурсивный вызов для указателя
}                             // на следующий элемент
void  insert(list **ph, int v) //Включение в конец списка
{
if (*ph == NULL)              // включить новый элемент по адресу
      {                       // указателя со значением NULL
      list *pnew=new list;
      pnew->val=v;
      pnew->next=NULL;
      *ph=pnew;
      }
else   insert(& (*ph)->next, pnew);
}
list *insord(list *ph, int v) // включение с сохранением порядка
{
if (ph==NULL || ph->val > v)
      { list *pnew=new list;
        pnew->val=v;
        pnew->next=ph;
        return pnew;
      }
return insord(ph->next); // функция возвращает возможно
}                 //измененный указатель на оставшуюся часть списка
```

- Деревья

Определение дерева имеет исключительно рекурсивную природу. Элемент этой структуры данных называется *вершиной*. Дерево представляет собой либо отдельную вершину, либо вершину, имеющую ограниченное число указателей - другие деревья (ветвей). Нижележащие деревья для текущей вершины называются поддеревьями, а их вершины - потомками. По отношению к потомкам текущая вершина называется предком (рис. 4).

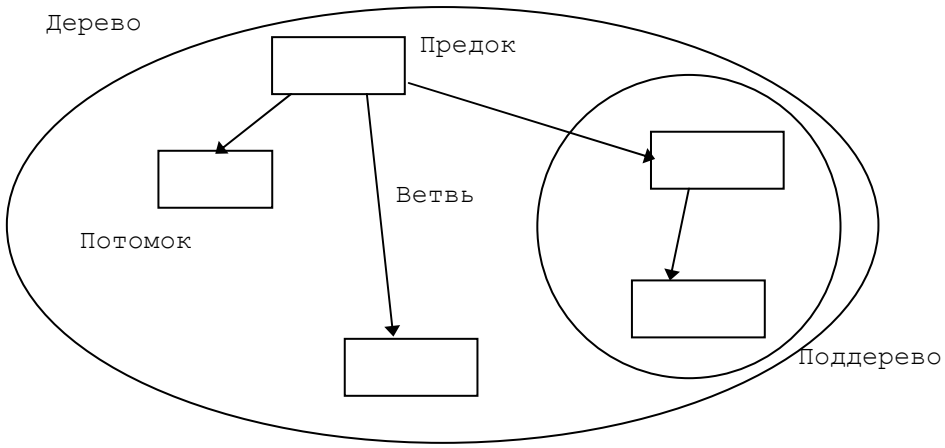


Рис.4

Вершину дерева можно определить таким образом:

```
struct      tree
{
      int   val;                    // значение элемента
      tree  *child[4];              // указатели на потомков
};
```

Из определения элемента дерева следует, что оно имеет ограниченное число указателей на подобные элементы. Как и во всех динамических структурах данных, характер связей между элементами определяется функциями, которые их устанавливают. Рекурсивное определение дерева требует и рекурсивного характера функций, работающих со связями. Простейшей функцией является функция обхода всех вершин дерева:

```
void  ScanTree(tree *p)             // обход дерева
{ int i;
  if (p == NULL)  return;           // следующей вершины нет
  for (i=0; i<4; i++)               // рекурсивный обход
      ScanTree(p->child[i]);        // потомков с передачей
}                                   // указателей на них
```

Само дерево обычно задается в программе указателем на его главную вершину. Часто обход дерева используется для получения информации, которая затем возвращается через результат рекурсивной функции, например, функция определения минимальной длины ветвей дерева:

```
int   MinDepth(tree *p)
{ int i, min, nn;
if (p == NULL)    return 0;          // следующей вершины нет
for (min = MinDepth(p->child[0], i=1; i<4; i++)
      {                              // обход потомков
      nn = MinDepth(p->child[i]);
      if (nn > max) max = nn;
      }                              // возвращается глубина с
return min + 1;                      // учетом текущей вершины
}
```

Другой распрстраненной функцией является включение нового элемента в дерево. Здесь есть проблема, общая для всех деревьев и рекурсивных алгоритмов. Войдя в поддерево, невозможно производить какие-либо действия для вершин, расположенных на том же уровне, но в других поддеревьях. Поэтому используется функция включения с просмотром дерева на заданную глубину, а сама глубина просмотра, в свою очередь, задается равной длине минимальной ветви дерева:

```
int Insert(tree *ph, int v, int d)
// ph - указателя на текущую вершину
// d  - текущая глубина включения
{
if (d == 0) return 0;   // ниже не просматривать
for (int i=0; i<4; i++)
      if (ph->child[i] == NULL)
            {
            tree *pn=new tree;
            ph->child[i]=pn;
            pn->val=v;
            for (i=0; i<4; i++) pn->child[i]=NULL;
            return 1; // результат логический - вершина включена
            }
      else
      if (Insert(ph->child[i], v , d-1)) return 1;
return 0;
}
void main()
{
      tree PH={1,{NULL,NULL,NULL,NULL}};  // пример вызова функции
      Insert(&PH, 5, MinDepth(&PH));
}
```

Из последнего примера видно, что количество просматриваемых вершин от уровня к уровню растет в геометрической прогрессии. Таким

образом, что деревья можно эффективно использовать для поиска данных: если включать в вершины дерева данные, таким образом, что наиболее часто используемые будут располагаться ближе к корню:

```
struct tree1
      {
      char *key;         // ключевое слово
      char *data;        // искомая информация
      tree1 *child[4];   // потомки
      };
char  *find(tree1 *ph, char *keystr)
{ char *s;
if (ph==NULL) return NULL;
if (strcmp(ph->key,keystr)==0) return ph->data;
                               // вершина найдена
if (strlen(keystr)<strlen(ph->key)) return NULL;
                               // короткие строки - ближе к корню
for (int i=0; i<4; i++)
      if ((s=find(ph->child[i],keystr))!=NULL) return s;
return NULL;
}
```

Функция включения в такое дерево ради сохранения свойств дерева при включении новой строки должна «вытеснять» более длинные строки из текущих вершин в поддеревья и заменять их на новые, более короткие.

- Двоичное дерево

Двоичным деревом называется дерево, каждая вершина которого имеет не более двух потомков. На данные, хранимые в вершинах дерева, вводится следующее правило упорядочения: значения вершин левого поддерева всегда меньше, а значения вершин правого поддерева - больше значения в самой вершине (рис. 5).

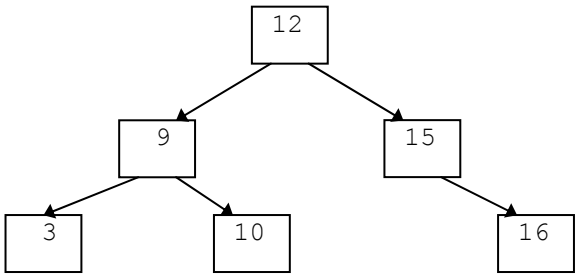


Рис. 5

```
struct      btree
      {
      int   val;
      btree *left,*right;
      };
```

Это свойство позволяет применить в двоичном дереве алгоритм двоичного поиска. Алгоритмы включения и исключения вершин дерева не должны нарушать указанное свойство: при включении вершины дерева поиск места ее размещения производится путем аналогичных сравнений:

```
btree *Search(btree *p, int v) //Рекурсивный поиск в двоичном дереве
{                 //    Возвращается указатель на найденную вершину
if (p==NULL) return(NULL);          // ветка пустая
if (p->val == v) return(p);         // вершина найдена
if (p->val > v)                     // сравнение с текущим
    return(Search(p->left,v));      // левое поддерево
else
    return(Search(p->right,v));     // правое поддерево
}
btree *Insert(btree *pp, int v) //включение значения в дерево
{     // функция возвращает указатель на созданную вершину,
      // либо на существующее поддерево
if (pp == NULL)               // найдена свободная ветка
    {                         // создать вершину дерева
    btree *q = new btree;     // и вернуть указатель
    q->val = v;
    q->left = q->right = NULL;
    return q;
    }
if (pp->val == v) return pp;
if (pp->val > v)                    // перейти в левое или
      pp->left=Insert(pp->left,v);  // правое поддерево
else
      pp->right=Insert(pp->right,v);
return pp;
}
void main()
{
btree *ss=Search(ph,5);             // пример вызова
ph=Insert(ph,6);
}
void  Scan(btree *p)    //рекурсивный обход двоичного дерева с выводом
{                       //значений вершин в порядке возрастания
if (p==NULL) return;
Scan(p->left);
cout << p->val << endl;
Scan(p->right);
}
```

- Нумерация вершин в деревьях. Способы обхода дерева

В любой структуре данных имеется естественная нумерация элементов по их расположению в ней. Так каждый элемент списка или массива имеет свой логический номер в линейной последовательности, соответствующей их размещению в памяти (массив) или направлению последовательного обхода (списки). В деревьях обход вершин возможен только

с использованием рекурсии, поэтому и логическая нумерация вершин производится согласно последовательности их рекурсивного обхода. Рекурсивная функция в этом случае получает текущий счетчик вершин, который она увеличивает на 1 при обходе текущей вершины и который она передает и получает обратно из поддеревьев. Далее рассмотрен пример обхода вершин дерева слева-направо, снизу-вверх, он обеспечивает просмотр значений вершин в порядке возрастания:

```
int Scan(btree *p, int n)
       //   Рекурсивный обход двоичного дерева с нумерацией вершин
{     //    снизу-вверх слева-направо, n - текущий номер вершины
if (p==NULL) return n;
Scan(p->left,n);
n++;
cout << n <<  p->val << endl;
n=Scan(p->right,n);
return n;
}
```

Аналогичный обход в порядке справа-налево, снизу-вверх дает в двоичном дереве последовательность в порядке убывания. Возможны и другие варианты обхода. Обход с нумерацией двоичного дерева используется для извлечения вершины по логическому номеру, соответствующему упорядочению значений этих вершин в порядке возрастания. В этом случае для нумерации вершин приходится использовать глобальный счетчик:

```
// Рекурсивный обход двоичного дерева с последовательной нумерацией
// вершин и с возвратом указателя на вершину с заданным номером
btree *ScanNum(btree *p, int *n)
{
      btree *q;
      if (p==NULL) return NULL;
      q=ScanNum(p->left,n);
      if (q!=NULL) return q;
      if ((*n)-- ==0) return p;
      return ScanNum(p->right,n);
}
void main()
{
      btree *ph; int N=26;          // пример вызова
      btree *s = ScanNum(ph,&N);
}
```

- Структуры данных с произвольными связями. Графы

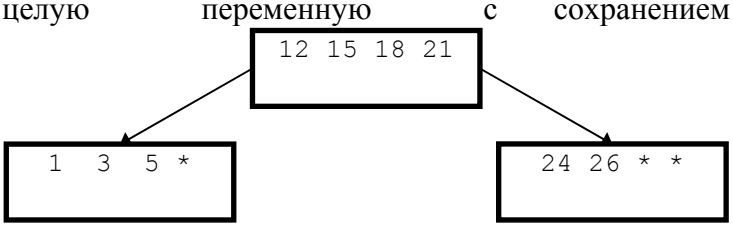
Граф представляет собой структуру с произвольным характером связей между элементами. Функции работы с графом также могут предусматривать его рекурсивный обход. При этом необходимо отмечать уже

пройденные вершины для исключения зацикливания алгоритма. В приведенном ниже примере каждая вершина графа имеет счетчик проходов, который проверяется каждый раз при входе в вершину:

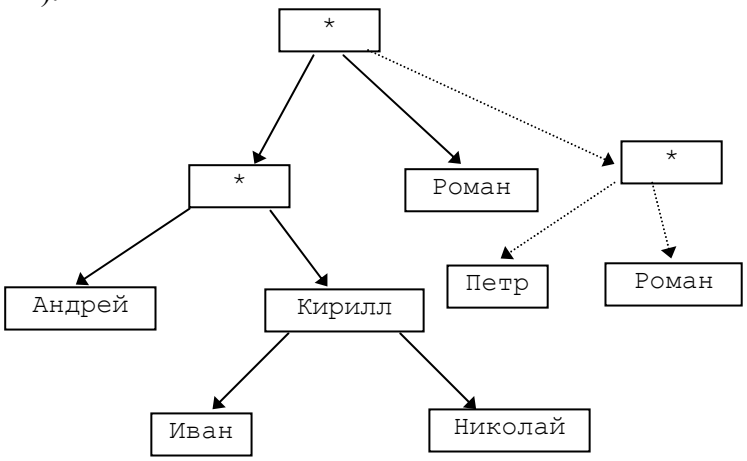
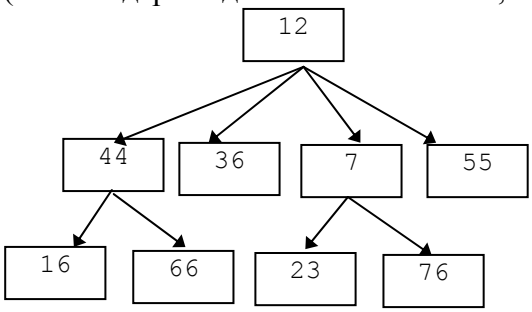
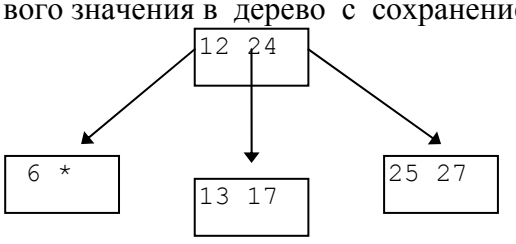
```
struct      graph
      {
      int   cnt;                    // счетчик обходов вершин
      graph **pl;                   // динамический массив
      };                            // указателей
void  ScanGraph(graph *p)
{ int i;
p->cnt++;                           // увеличить счетчик
for (i=0; p->pl[i] !=NULL; i++)
      {
      if (p->pl[i]->cnt !=p->cnt)   // вершина не просмотрена
      ScanGraph(p->pl[i]);          // рекурсивный обход
      }
}
```

Задания для выполнения

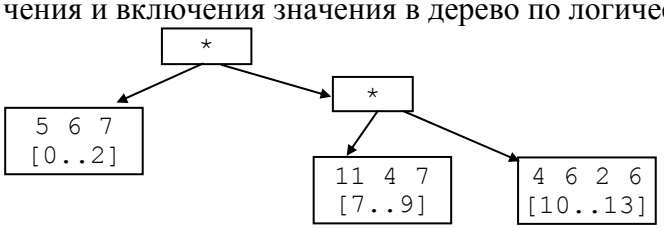
Таблица 3

Программа должна содержать функцию обхода дерева с выводом его содержимого, функцию добавления вершины дерева (ввод), а также указанную в варианте функцию.	
1	Вершина дерева содержит указатель на строку. Строки в дереве не упорядочены. Функция включает вершину в дерево с новой строкой в ближайшее свободное к корню дерева место. (То есть дерево должно иметь ветви, отличающиеся не более чем на 1).
2	Вершина двоичного дерева содержит массив целых и два указателя на правое и левое поддерево. Массив целых в каждом элементе упорядочен, дерево в целом также упорядочено. Функция включает в дерево целую переменную с сохранением упорядоченности. 
3	Вершина двоичного дерева содержит указатель на строку и указатели на правое и левое поддерево. Строки в дереве упорядочены по возрастанию. Написать функций включения строки и получения указателя на строку по заданному номеру, который она имеет в упорядоченной последовательности обхода дерева.

Продолжение таблицы 3

4	Элемент дерева содержит либо данные (строка ограниченной длины), либо указатели на правое и левое поддеревья. Строки в дереве упорядочены. Написать функцию включения новой строки. (Обратить внимание на то, что элемент с указателями не содержит данных, и при включении новой вершины вершину с данными следует заменить на вершину с указателями). 
5	Вершина дерева содержит целое число и массив указателей на поддеревья. Целые в дереве не упорядочены. Функция включает вершину в дерево с новой целой переменной в ближайшее свободное к корню дерева место. (То есть дерево должно иметь ветви, отличающиеся не более чем на 1). 
6	Вершина дерева содержит два целых числа и три указателя на поддеревья. Данные в дереве упорядочены. Написать функцию включения нового значения в дерево с сохранением упорядоченности. 

Продолжение таблицы 3

7	Вершина дерева содержит указатель на строку и N указателей на потомков. Функция помещает строки в дерево так, что строки с меньшей длиной располагаются ближе к корню. Если новая строка “проходит” через вершину, в которой находится более длинная строка, то новая занимает место старой, а алгоритм включения продолжается для старой строки. Функция включения выбирает потомка минимальным количеством вершин в поддереве.
8	Вершина дерева содержит либо 4 целых значения, либо 2 указателя на потомков, причем концевые вершины содержат данные, а промежуточные - указатели на потомков. Естественная нумерация значений производится при обходе концевых вершин слева направо. Разработать функции получения и включения значения в дерево по логическому номеру. 
9	Двоичное дерево представлено в массиве «естественным образом»: если вершина-предок имеет номер индекс n, то потомки - соответственно `2*n` и `2*n+1`. Нумерация начинается с `n=1`. Ячейка со значением 0 (или `NULL`) обозначает отсутствие вершины. Разработать функцию сортировки строк с использованием способа представления такого дерева в массиве указателей на строки.
10	Вершина дерева содержит `N` целых значений и два указателя на потомков. Запись значений производится таким образом, что меньшие значения оказываются ближе к корню дерева (то есть все значения в поддеревьях больше самого большого значения у предка). Разработать функции включения и поиска данных в таком дереве. Если новое значение “проходит” через вершину, в которой находится большее, то оно замещает большее значение, а для последнего алгоритм включения. Функция включения выбирает потомка максимальным значением в поддереве.
11	Выражение, содержащее целые константы, арифметические операции и скобки, может быть представлено в виде двоичного дерева. Концевая вершина дерева должна содержать значение константы. Промежуточная - код операции и указатели на правый и левый операнды - вершины дерева. Функция получает строку, содержащую выражение, и строит по ней дерево. Другая функция производит вычисления по полученному дереву.
12	Вершина дерева содержит указатель на строку и динамический массив указателей на потомков. Размерность динамического массива в корневой вершине - `N`, на каждом следующем уровне - в 2 раза больше. Функция при включении строки создает вершину, наиболее близкую к корню.

Окончание таблицы 3

13	Вершина дерева содержит динамический массив целых значений и два указателя на потомков. Значения в дереве не упорядочены. Размерность динамического массива в корневой вершине - N, на каждом следующем уровне - в 2 раза больше. Функция включает новое значение в свободное место в массиве ближайшей к корню вершины.
14	Вершина дерева содержит массив целых и два указателя на правое и левое поддерево. Значения в дереве не упорядочены. Естественная нумерация значений производится путем обхода дерева по принципу “левое поддерево - вершина - правое поддерево”. Разработать функции включения и получения значения элемента по заданному логическому номеру.
15	Код Хаффмана, учитывающий частоты появления символов, строится следующим образом. Для каждого символа подсчитывается частота его появления и создается вершина двоичного дерева. Затем из множества вершин выбираются две с минимальными частотами появления и создается новая - с суммарной частотой, к которой выбранные подключаются как правое и левое поддерево. Созданная вершина включается в исходное множество, а выбранные - удаляются. Затем процесс повторяется до тех пор, пока не останется единственная вершина. Код каждого символа - это последовательность движения к его вершине от корня (левое поддерево - 0, правое - 1). Опередить функцию построения кода Хаффмана для символов заданной строки.
Построчно ввести различные целые числа. Построить бинарное дерево поиска Т.	
16	Определить число листьев дерева. Вывести элементы дерева на экран.
17	Заменить все отрицательные элементы на их абсолютные значения деленные на среднее арифметическое этих элементов. Вывести элементы дерева на экран.
18	Определить число ветвей от корня до ближайшей вершины с заданным элементом. Вывести часть дерева от вершины до данного элемента на экран.
19	Удалить вершину с максимальным элементом. Вывести элементы дерева на экран.
20	Распечатать все элементы дерева по уровням: корень дерева, вершины 1-го уровня, вершины 2-го уровня, ...

Контрольные вопросы

1. Дать определение структуре данных - дерево?
2. В чем выражается рекурсивный характер дерева?
3. Поясните принцип построения двоичного дерева.
4. Как нумеровать вершины дерева и от чего это зависит?
5. Что такое связанный граф?

Практикум №5. ОБЪЕКТ И КЛАСС

Краткие теоретические сведения

Объект - структурированная переменная, содержащая всю информацию о некотором физическом предмете или реализуемом в программе понятии. **Класс** - описание множества таких объектов и выполняемых над ними действий.

- Классы

Классы C++ предусматривают создание расширенной системы предопрделнных типов. Каждый тип класса представляет уникальное множество объектов, операций над ними, а также операций, используемых для создания, манипулирования и уничтожения таких объектов. С классами связан ряд новых понятий.

Наследование. Новый, или производный класс может быть определен на основе уже имеющегося, или базового. При этом новый класс сохраняет все свойства старого: данные объекта базового класса включаются в данные объекта производного, а методы базового класса могут быть вызваны для объекта производного класса.

Полиморфизм. Он основывается на возможности включения в данные объекта также и информации о методах их обработки (в виде указателей на функции). Будучи доступным в некоторой точке программы, даже при отсутствии полной информации о типе объекта, он всегда может корректно вызвать свойственные ему методы. *Полиморфной* называется функция, определенная в нескольких производных классах и имеющая в них общее имя.

Для определеняи класса используются три ключевых слова: `struct`, `union` и `class`. Каждый класс включает в себя функции - называемые методами, компонентными функциями или функциями членами (member function), и данные - элементы данных (class members) (рис. 6).

Рис.6

Имя класса становится индентификатором новго типа данных и может использоваться для объявления объектов.

С точки зрения синтаксиса, класс в C++ - это структурированный тип, образованный на основе уже существующих типов. Класс можно определить с помощью формы:

```
тип_класса имя_класса  <:базовый_ класс>
            {список_членов_класса или список компонентов};
```

где тип_класса - одно из служебных слов class, struct, union;

имя_класса - идентификатор;

необязательный параметр базовый_класс - содержит класс или классы, из которого имя_класса заимствует элементы и методы, при необходимости спецификаторы доступа;

список_членов_класса - определения и описания типизированных данных и принадлежащих классу функций. Пример:

```
class STUDENT     {
char name[25];                  // имя
int age;                        // возраст
float grade;                    // рейтинг
char * GetName() ;              //метод – получить имя
int GetAge() const;             //метод – получить возраст
float GetGrade() const;        //метод – получить рейтинг
void SetName(char*);           //метод – установить имья
void SetAge(int);              //метод – установить возраст
void Show(); };                //метод – печать
```

Тело элемента-функции может быть определено в самом классе или в определении класса дается только прототип функции (заголовок с перечислением типов формальных параметров), а определение самой функции дается отдельно, при этом полное имя функции имеет вид:

```
имя_класса::имя_функции
```

Для структурированной переменной вызов функции - элемента этой структуры имеет вид:

```
имя_переменной.имя_функции (список_параметров);
```

Для описания объекта класса (экземпляра класса) используется форма:

```
имя_класса имя_объекта;
```

Пример:

```
STUDENT my_friend,me;
STUDENT *point = &me;                 // указатель на объект STUDENT
STUDENT group [30];                  // массив объектов
STUDENT &girle = my_friend;     // ссылка на объект
```

Обращаться к данным объекта или компонентам класса и вызывать функции для объекта можно двумя способами. Первый с помощью «квалифицированных» имен:

```
имя_объекта. имя_данного
имя_объекта. имя_функции
```

Пример:

```
STUDENT x1,x2;
x1.age = 20;
x1.grade = 2.3;
x2.Set("Петрова",18,25.5);
x1.Show ();
```

Второй способ доступа использует указатель на объект:

```
указатель_на_объект->имя_компонента
```

Пример:

```
STUDENT *point = &x1;  //  или  point = new STUDENT;
point ->age = 22;
point ->grade = 7.3;
point ->Show();
```

- Доступность компонентов класса

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

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

```
имя_объекта.имя_члена_класса
ссылка_на_объект.имя_члена_класса
указатель_на_объект->имя_члена_класса
```

Собственные, `private` компоненты локализованы в классе и не доступны извне. Они могут использоваться функциями - членами данного класса и функциями-друзьями того класса, в котором они описаны.

Защищенные, `protected` компоненты доступны внутри класса и в производных классах.

По умолчанию для `class` все элемнты являются собственными или частными , т.е. `private`. Пример:

```
class STUDENT{
     char name[25];                  // private по умолчанию
     int age;
     float grade;
public:
     void SetName(char*);
     void SetAge(int);};
```

Спецификатор доступа действителен пока не встретиться другой спецификатор достпуа, они могут пересекатся и группироваться:

```
class STUDENT{
     char name[25];
     int age;
     float grade;
private
protected:
     void SetName(char*);
public:
     void SetAge(int);}Me;
Me.age=18;        // ошибка
Me->SetAge(18);   // правильно
```

- Создание и уничтожение объектов

Недостатком рассмотренных ранее классов является отсутствие автоматической инициализации создаваемых объектов. Для каждого вновь создаваемого объекта необходимо вызвать функцию типа `Set` либо явным образом присваивать значения данным объекта.

Элементы-функции, неявно вызываемые при создании и уничтожении объектов класса называются *конструкторами* и *деструкторами*. Они определяются как элементы-функции с именами, совпадающими с именем класса. Конструкторов для данного класса может быть сколь угодно много, если они отличаются формальными параметрами, деструктор же всегда один и имеет имя, предваренное символом "~".

- Конструктор

Формат определения конструктора:

имя_класса(список_форм_параметров){операторы_тела_конструктора}

```
class STUDENT{
char name[25];
int age;
float grade;
public:
STUDENT();                        // конструктор без параметров
STUDENT(char*,int,float);         // конструктор с параметрами
STUDENT(const STUDENT&);          // конструктор копирования
...};
STUDENT::STUDENT(char*NAME,int AGE,float GRADE)
      {
```

```
      strcpy(name,NAME); age=AGE; grade=GRADE;
cout<< \nКонструктор с параметрами вызван для объекта  <<this<<endl;
      }
```

Момент вызова конструктора и деструктора определяется временем создания и уничтожения объектов. Конструкторы обладают некоторыми уникальными свойствами:

- конструктор имеет то же имя, что и сам класс;
- для конструктора не определяется тип возвращаемого значения (даже тип `void` не допустим);
- конструкторы не наследуются, хотя производный класс может вызывать конструкторы базового класса;
- конструкторы могут иметь аргументы по умолчанию или использовать списки инициализации элментов;
- конструкторы не могут быть описаны с ключевыми словами `virtual, static, const, mutuable, volatile`;
- нельзя работать с их адресами;
- если он не был задан явно, то генерируется компилятором;
- конструктор нельзя вызывать как обычную функцию;
- конструктор автоматически вызывается при определении или размещении в памяти с помощью неявного вызова оператора new каждого объекта класса и `delete`;
- конструктор выделяет память для объекта и инициализирует данные-члены класса.

По умолчанию создается общедоступный (`public`) конструктор без параметров и конструктор копирования. Если конструктор описан явно, то конструктор по умолчанию не создается. Параметром конструктора не может быть его собственный класс, но может быть ссылка на него (`T&`). Без явного указания программиста конструктор всегда автоматически вызывается при определении (создании) объекта. Для явного вызова конструктора используются две формы:

```
имя_класса  имя_объекта (фактические_параметры);
имя_класса (фактические_параметры);
```

Первая форма допускается только при не пустом списке фактических параметров. Вторая форма вызова приводит к созданию объекта без имени.

Существуют два способа инициализации данных объекта с помощью конструктора: передача значений параметров в тело конструктора; применение списка инициализаторов данного класса. Каждый инициализатор списка относится к конкретному компоненту и имеет вид:

имя_данного (выражение)

Пример:

```
class CLASS_A
{  int i; float e; char c;
public:
  CLASS_A(int ii,float ee,char cc) : i(8),e( i * ee + ii ),c(cc){}
. . .};
```

При определении объекта будет запущен тот конструктор, с которым совпадает заданный список аргументов.

Конструктор копирования (вида `T::T(const T&)`, где `T` – имя класса) вызывается всякий раз, когда выполняется копирование объектов, принадлежащих классу. Он вызывается:

а) когда объект передается функции по значению:

```
void View(STUDENT a){a.Show;}
```

б) при построении временного объекта как возвращаемого значения функции:

```
STUDENT NoName(STUDENT & student)
{STUDENT temp(student);
temp.SetName("NoName");
return temp;}
STUDENT c=NoName(a);
```

в) при использовании объекта для инициализации другого объекта:

```
STUDENT a("Иванов",19,50), b=a;
```

Если класс не содержит явным образом определенного конструктора копирования, то при возникновении одной из этих трех ситуаций производится побитовое копирование объекта (не во всех случаях является адекватным).

- Деструктор

Динамическое выделение памяти для объекта создает необходимость освобождения этой памяти при уничтожении объекта. Такую возможность обеспечивает специальный компонент класса – *деструктор*. Его формат:

~имя_класса(){операторы_тела_деструктора}

Свойства деструктора:

- имя деструктора совпадает с именем его класса, но предваряется символом "~" (тильда);

- деструктор не имеет параметров и возвращаемого значения;

- вызов деструктора выполняется не явно (автоматически), как только объект класса уничтожается.

Если в классе деструктор не определен явно, то компилятор генерирует деструктор по умолчанию, который просто освобождает память, занятую данными объекта. Так же, как и для конструктора, не может быть определен указатель на деструктор:

```
class Circle
(public:
...
~Circle (); //Деструктор private:
...
};
```

- Указатели на компоненты-функции

Можно определить указатель на компоненты-функции:

```
тип_возвр_значения(имя_класса::*имя_указателя_на_функцию)
            (специф_параметров_функции);
```

Пример:

```
void (STUDENT::*pf)();
pf=&STUDENT::Show;
(p[1].*pf)();
```

- Указатель this

Когда функция-член класса вызывается для обработки данных конкретного объекта, этой функции автоматически и неявно передается указатель на тот объект, для которого функция вызвана. Этот указатель имеет имя `this` и неявно определен в каждой функции класса следующим образом:

```
имя_класса *const this = адрес_объекта
```

Указатель `this` является дополнительным скрытым параметром каждой нестатической компонентной функции. При входе в тело принадлежащей классу функции `this` инициализируется значением адреса того объекта, для которого вызвана функция. В результате этого объект становится доступным внутри этой функции.

В большинстве случаев использование `this` является неявным. В частности, каждое обращение к нестатической функции-члену класса неявно использует `this` для доступа к члену соответствующего объекта. Примером широко распространенного явного использования `this` являются операции со связанными списками.

- Вставляемые (inline) функции

Если функция (обычная или функция-элемент класса) объявлены `inline`-функциями, то при вызове таких функций транслятор выполняет подстановку по тексту программы тела функции с соответствующей за-

меной формальных параметров на фактические. Функция-элемент также считается `inline` по умолчанию, если ее тело находится непосредственно в определении класса, например:

```
struct  dat
{int     day,month,year;
void    Setdat(char *p)       // Функция inline по умолчанию
        { ... }               // тело функции
};
```

Функцию-член можно описать как `inline` вне описания класса. Например:

```
  char char_stack {
      int size;
      char* top;
      char* s;
  public:
       char pop();
      // ...  };
  inline char char_stack::pop()
  {      return *--top;  }
```

- Друзья классов

Иногда требуются исключения из правил доступа, когда некоторой функции или классу требуется разрешить доступ к личной части объекта класса. Тогда в определении класса, к объектам которого разрешается такой доступ, должно быть объявление функции или другого класса как «дружественных».

Объявление дружественной функции представляет собой прототип функции, объявление переопределяемой операции или имя класса, которым разрешается доступ, с ключевым словом `friend` впереди. Общая схема объявления такова:

```
class   A
        {        int     x;   // Личная часть класса
        ...
friend  class   B;  // Функции класса B дружественны A
friend  void  C::fun(A&);
      // Элемент-функция fun класса C имеет доступ к приватной части A
friend  void  xxx(A&,int);    // Функция xxx дружественна классу A
friend  void  C::operator+(A&);// Переопределяемая в классе C операция
        };         // <объект C>+<объект A> дружественна классу A
  class   B       // Необходим доступ к личной части A
        {public: int     fun1(A&);
        void    fun2(A&);         };
class   C
        { public: void    fun(A&);
        void    operator+(A&);         };
```

К средствам контроля доступа относятся также объявления функций-элементов постоянными (const). В этом случае они не имеют права изменять значение текущего объекта, с которым вызываются. Заголовок такой функции имеет вид:

```
void    dat::put() const { ... }
```

По аналогии с функциями и методами можно объявить дружественными весь класс. При этом все методы такого дружественного класса смогут обращаться ко всем компонентами класса:

```
Class Myclass
{… Friend class MyFriend; …}
```

Дружба классаов не транзитивна и односторонняя.

- Статические элементы класса

Иногда требуется определить данные, которые относятся ко всем объектам класса. Например, контроль общего количества объектов класса или одновременный доступ ко всем объектам или части их, разделение объектами общих ресурсов. Тогда в определение класса могут быть введены статические элементы-переменные. Такой элемент сам в объекты класса не входит, зато при обращении к нему формируется обращение к общей статической переменной с именем. Доступность ее определяется стандартным образом в зависимости от размещения в личной или общей части класса. Сама переменная должна быть явно определена в программе модификатором static и инициализирована.

Статическими могут быть объявлены методы класса. Их вызов не связан с конкретным объектом и может быть выполнен по полному имени. Соответственно в них не используются неявный указатель на текущий объект this. Статические методы не могут быть константными (const) и выиртуальными (virtual). Например:

```
class   list
        { ...
static  void    show();       // статическая функция просмотра
        }                     // списка объектов
static  void    list::show()
{list    *p;
for (p=fst; p !=NULL; p=p->next)
        { ...вывод информации об объекте... }}
void    main()
{ ... list::show(); }         // вызов функции по полному имени
```

- Мастер Class Wizard

Среда разработки предлагает существенную помощь при определении нового класса, декларации методов и переменных, исходного кода

методов. Мастер **Class Wizard** позволяет добавлять в программу новые классы или изменять существующие. Использовать **Class Wizard** можно после того, как при помощи *AppWizard* создан прототип приложения. Доступ к **Class Wizard** осуществляется через меню **View**. Панель **Class Wizard Bar** выглядит так, как показано на рис. 7.

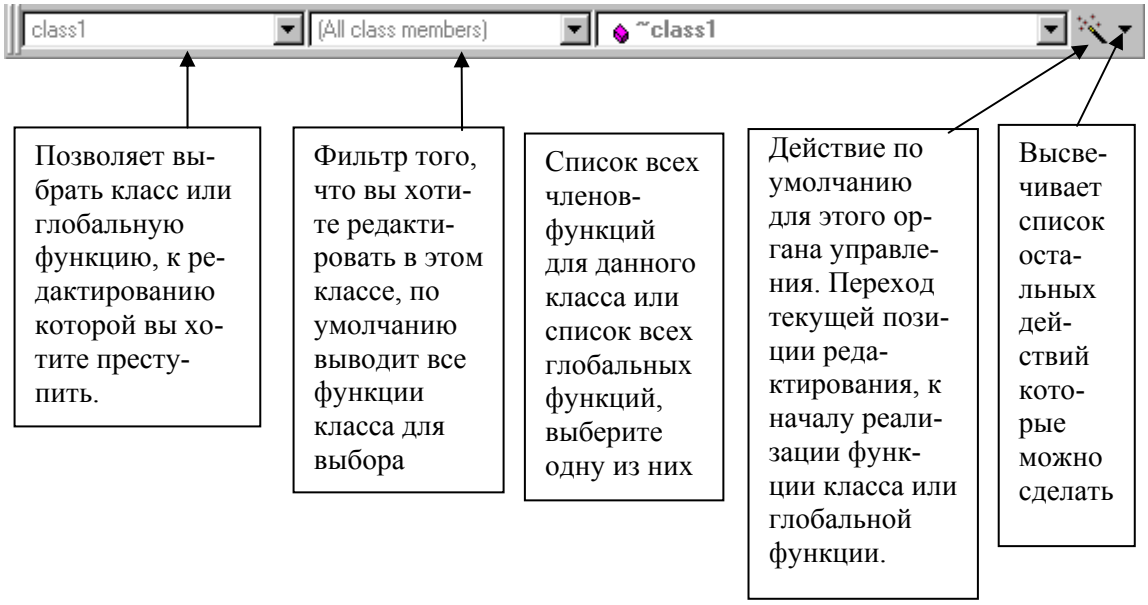


Рис.7

Выберается класс, затем функция класса, либо просто имя глобальной функции, щелкается по кнопке «**действие по умолчанию**» и получится переход в редакторе к тому месту в программе, где находится начало реализации выбранной функции. Если необходимо изменить что-то в декларации функции или класса, то сделать это можно или вручную, или автоматически.

Дополнительно **Class Wizard** предоставляет возможности, список которых можно получить, нажав на стрелку вниз (рис.8). При удалении какой либо функции класса с помощью средств Visual C++, сам код функции не удаляется, а только заключается в комментарии, окончательное решение на удаление - за разработчиком.

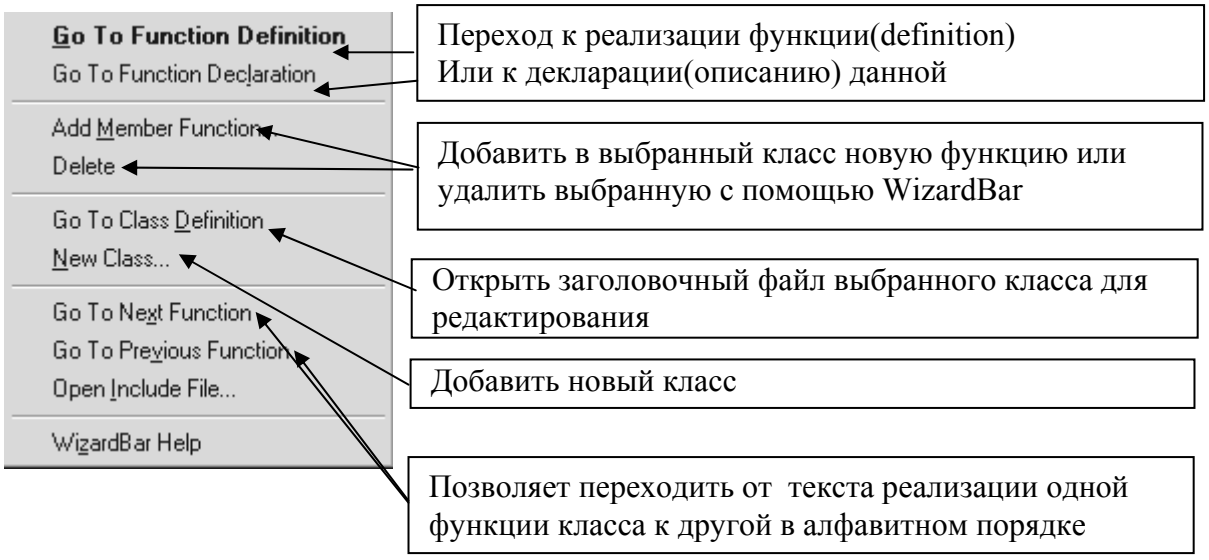


Рис. 8

Чтобы добавить новую функцию-член класса, можно воспользоваться средствами либо вкладки *Classes*, либо средствами ***Class Wizard Bar***. В любом случае появится следующее окно (рис. 9).

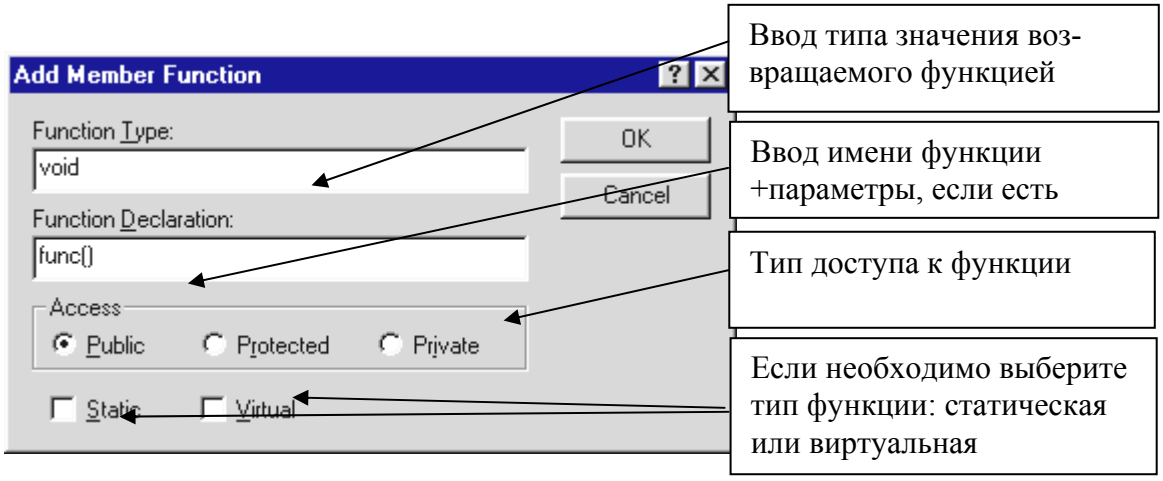


Рис. 9

Сгенерированная функция всегда появляется в декларации класса. В конце того файла, где находится реализация функций данного класса, появится «заготовка» для данной функции (просто пустая функция). Практически аналогично выполняется добавление новой переменной в класс. Для этого надо в **Workspace Viwer** выбрать вкладку *Class View* и щелкнуть на имени класса правой кнопкой мыши. В появившемся всплывающем меню выбрать пункт **Add Member Variable…** (рис.10).

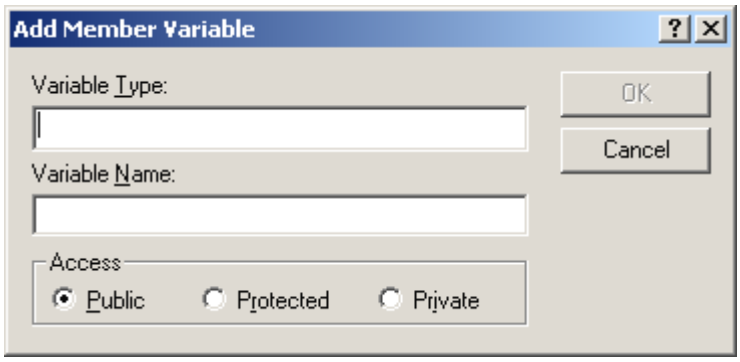


Рис. 10

Для внесения в проект данных о новом классе можно воспользоваться **Class Wizard Bar**, вызвав одну из возможностей с именем **New Class**. Открывается окно, представлееное на рис. 11. При создании генерируется прототип класса, пустого конструктора и деструктора в заданных файлах. Затем в класс можно добавить члены-данные и члены-функции.

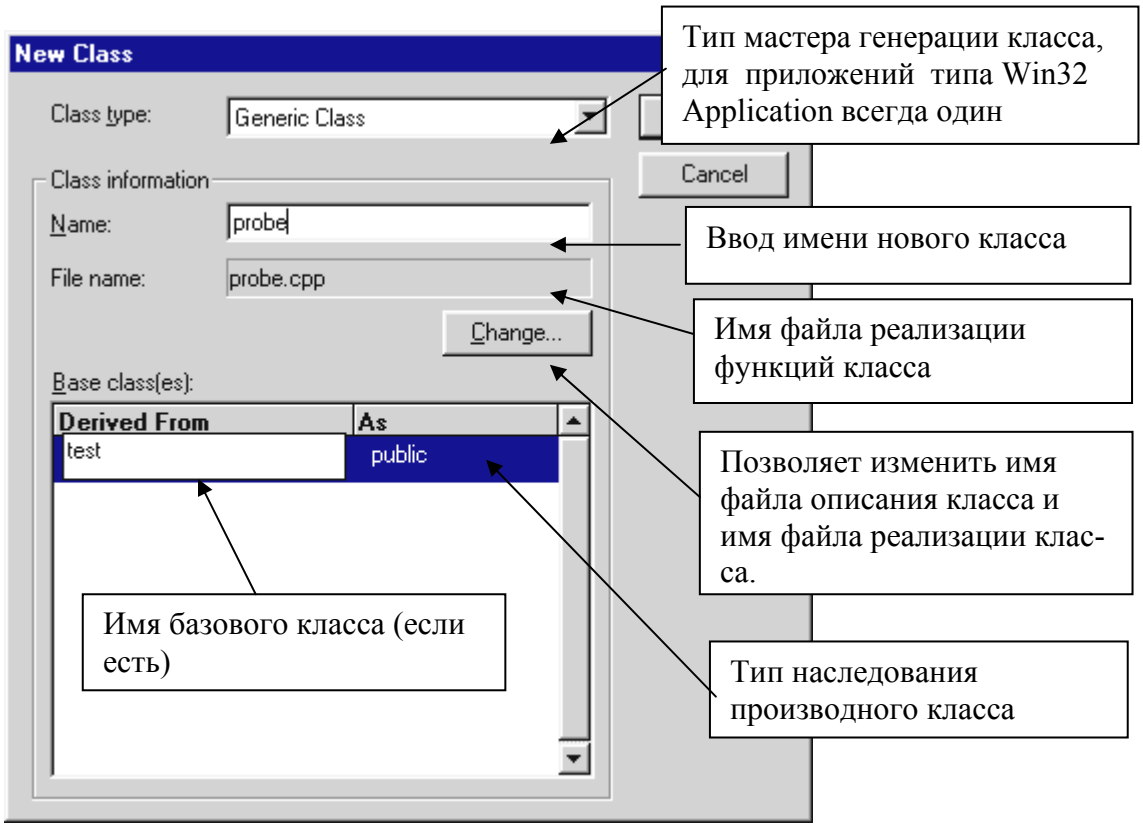


Рис. 11

Задания для выполнения

Таблица 4

Определить пользовательский класс в соответствии с вариантом задания (смотри варианты). Определить в классе следующие конструкторы: без параметров, с параметрами, копированиия. Определить в классе деструктор, компоненты-функции для просмотра и установки полей данных, указатель на компоненту-функцию. Определить указатель на экземпляр класса. Написать демонстрационную программу, в которой создаются и разрушаются объекты пользовательского класса и каждый вызов конструктора и деструктора сопровождается выдачей соответствующего сообщения (какой объект какой конструктор или деструктор вызвал). Показать в программе использование указателя на объект и указателя на компоненту-функцию. При работе с классами пользоваться Class Wizard. Для описания членов-данных классов использовать:	
1	СТУДЕНТ: имя – char*; курс – int; пол – int(bool)
2	ИЗДЕЛИЕ: имя – char*; шифр – char*; количество – int
3	АДРЕС: имя – char*; улица – char*; номер дома – int
4	ЦЕХ: имя – char*; начальник – char*; количество работающих – int
5	СТРАНА: имя – char*; форма правления – char*; площать – float
6	СЛУЖАЩИЙ: имя – char*; возраст – int; рабочий стаж – int
7	БИБЛИОТЕКА: имя – char*; автор – char*; стоимость – float
8	ТОВАР: имя – char*; количество – int; стоимость – float
9	ПЕРСОНА: имя – char*; возраст – int; пол – int(bool)
10	ЖИВОТНОЕ: имя – char*; класс – char*; средний вес – int
11	КАДРЫ: имя – char*; номер цеха – int; разряд – int
12	ЭКЗАМЕН: имя студента – char*; дата – int; оценка – int
13	КВИТАНЦИЯ: номер – int; дата – int; сумма – float
14	АВТОМОБИЛЬ: марка – char*; мощность – int; стоимось – float
15	КОРАБЛЬ: имя – char*; водоизмещение – int; тип – char*
Создать класс матрица. Определить конструктор без параметров, конструктор с одним параметром и конструктор с двумя параметрами, деструктор. Определить методы доступа: возвращать значение элемента (i,j) и адрес этого элемента. Определить функции сложения и вычитания (матрицы с матрицей), умножение матрицы на матрицу. Определить умножение матрицы на число и функцию печати. Проверить работу этого класса. В случае нехватки памяти, несоответствия размерностей, выхода за пределы устанавливать код ошибки.	
16	Класс содержит указатель на int, размер строк, столбцов, состояние ошибки.

Продолжение таблицы 4

17	Класс содержит указатель на long, размер строк и столбцов и состояние ошибки.
18	Класс содержит указатель на float, размер строк и столбцов и состояние ошибки.
19	Класс содержит указатель на double, размер строк и столбцов и состояние ошибки.
Выплнить задание	
20	Создать класс вектор, который имеет указатель на int, число элементов и переменную состояния. Определить конструктор без параметров, конструктор с параметром, конструктор с двумя параметрами. Определить функцию, которая присваивает элементу массива некоторое значение (параметр по умолчанию), функцию, которая получает некоторый элемент массива. В переменную состояния устанавливать код ошибки, при не хватке памяти, выхода за пределы массива. Определить функции печати, сложения, умножения, вычитания, которые производят эти арифметические операции с данными этого класса и встроенного int. Определить методы сравнения: больше, меньше или равно. Предусмотреть возможность подсчета числа объектов данного типа.
21	Создать класс типа - прямоугольник. Поля - высота и ширина. Функции-члены вычисляют площадь, периметр, устанавливает поля и возвращают значения. Функции-члены установки полей класса должны проверять корректность задаваемых параметров. Функция печати.
22	Создать класс типа - стек. Функции-члены вставляют элемент в стек, вытаскивают элемент из стека. Проверяют вершину стека.
23	Создать класс типа - двухсвязный список. Функции-члены добавляют элемент к списку, удаляют элемент из списка. Отображают элементы списка от начала и от конца. Найти элемент в списке.
24	Создать класс типа - дата с полями: день (1-31), месяц (1-12), год (целое число). Класс имеет конструктор. Функции-члены установки дня, месяца и года. функции-члены получения дня, месяца и года, а также две функции-члены печати: печать по шаблону: “5 января 1997 года” и “05.01.1997”. Функции-члены установки полей класса должны проверять корректность задаваемых параметров.
25	Создать класс типа - циклическая очередь. Функции-члены получают элемент и вставляют элемент.
26	Создать класс представления матриц произвольного размера. Поля- число строк и столбцов. Функции-члены изменяют число строк и столбцов, выводят подматрицы любого размера и всю матрицу.
27	Создать класс типа - окружность. Поля - радиус. Функции-члены вычисляют площадь, длину окружности, устанавливает поля и возвращают значения. Функции-члены установки полей класса должны проверять корректность задаваемых параметров. Функция печати.

Окончание таблицы 4

28	Создать класс типа - время с полями: час (0-23), минуты (0-59), секунды (0-59). Класс имеет конструктор. Функции-члены установки времени,. функции-члены получения часа, минуты и секунды, а также две функции-члены печати: печать по шаблону: “16 часов 18 минут 3 секунды” и “4 p.m. 18 минут 3 секунды”. Функции-члены установки полей класса должны проверять корректность задаваемых параметров.
29	Создать класс типа - квадрат. Поля - сторона. Функции-члены вычисляют площадь, периметр, устанавливает поля и возвращают значения. Функции-члены установки полей класса должны проверять корректность задаваемых параметров. Функция печати.
30	Создайте класс, который использует массив из 40 элементов для хранения целых чисел вплоть до больших целых, содержащих по 40 цифр. Функции-члены должны вводить, выводить, складывать и вычитать эти большие, целые. Сравнение больших целых чисел. Деление и умножение целых больших чисел.

Контрольные вопросы

1. Что такое деструктор и каково его назначение?

2. Истинно ли следующее утверждение: поля класса должны быть всегда закрытыми?

3. Определите класс `lev`, включающего одно закрытое поле типа `int` с именем `bar` и один открытый методом с прототипом `void Pr ()`.

4. На что ссылается указатель `this`?

5. Какаво назначение константного метода, вызванного для объекта класса?

Практикум № 6. НАСЛЕДОВАНИЕ. ВЗАИМОДЕЙСТВИЕ ОБЪЕКТОВ

Краткие теоретические сведения

Простейшим случаем введения иерархии в систему классов является использование объектов ранее определенных классов в качестве элементов данных нового класса. Другой способ создания иерархии классов заключается в том, что новый класс автоматически включает в себя все свойства старого класса, а затем развивает их.

- Производные классы

Сохранение в новом классе свойств старого называется *наследованием*. Принцип наследования состоит в том, что элементы данных старого класса автоматически становятся элементами данных нового класса, а все функции-элементы старого класса применимы к объекту нового класса, точнее к его старой составляющей. Старый класс при этом называется *базовым классом*, новый - *производным классом*.

Синтаксис определения производного класса имеет вид:

```
class производный : базовый_1, базовый_2,...базовый_n
      { определение личной и общей частей производного класса
      }
```

Основные свойства базового и производного классов:

- объект базового класса определяется в производном классе как неименованный (он не может быть использован в явном виде);

- элементы данных базового класса включаются в объект производного класса, однако личная часть базового класса закрыта для прямого использования в производном классе;

- функции-элементы базового класса наследуются в производном классе, то есть вызов функции, определенной в базовом классе возможен для объекта производного класса и понимается как вызов ее для входящего в него объекта базового класса;

- в производном классе можно переопределить (перегрузить) наследуемую функцию, которая будет вызываться вместо нее. При этом для выполнения соответствующих действий над объектом базового класса она может включать явный вызов переопределенной функции по полному имени:

```
class   a
      {
public:
      void    f() {}
```

```
      void    g() {}
      };
class   b : a           // производный класс : базовый класс
      {
public:
      void  f()         // "f" переопределяется
            { ...
            a::f();     // явный вызов "f" для БК
            }           // "g" наследуется из  БК
      void  h() {}      // собственная функция в ПК
      };
void    main()
{
a       A1;
b       B1;
B1.f();                 // вызов переопределенной b::f()
B1.g();                 // вызов наследуемой a::f()
}
```

Предполагается, что при вызове в производном классе функций, наследуемых из базового, транслятор производит преобразование указателя `this` объекта производного класса в указатель на входящий в него объект базового класса, учитывая размещение второго в первом.

- Конструкторы и деструкторы производных классов

Поскольку конструкторы не наследуются, при создании производного класса наследуемые им данные-члены должны инициализироваться конструктором базового класса. Конструктор базового класса вызывается автоматически и выполняется до конструктора производного класса. Параметры конструктора базового класса указываются в определении конструктора производного класса:

```
class Basis
{     int a,b;
      public:
      Basis(int x,int y){a=x;b=y;}
};
class Inherit:public Basis
{     int sum;
      public:
      Inherit(int x,int y, int s):Basis(x,y){sum=s;}
};
```

Взаимоотношение конструкторов и деструкторов базового и производного классов:

- объекты класса конструируются снизу вверх: сначала базовый, потом компоненты-объекты (если они имеются), а потом сам производный класс. Деструкторы вызываются в обратном порядке - сначала для производного, затем для базового;

- в заголовке конструктора производного класса может быть явно указан вызов конструктора базового класса с параметрами. Он может быть без имени, а может быть с именем базового класса. Если базовых классов несколько, то в вызовы конструкторов базовых классов должны быть перечислены через запятую и должны быть поименованы.

Существуют следующие варианты наследования методов базового класса в производном (способы изменения свойств объекта):

- «новое свойство» - имя определяемого в производном классе метода не совпадает ни с одним из известных в базовом классе:

```
class a {
      public: void f() {}
        };
class b : public a
            {
      public: void newb() {}  // newb() - новое свойство (метод)
            };
```

- «полное неявное наследование» - если в производном классе метод не переопределяется, то по умолчанию он наследуется из базового класса. Определенное в базовом классе свойство не меняется:

```
class a {
      public: void f() {}
            };
class b : public a
            {
      public:     // f() - унаследованное свойство (метод)
            };    // эквивалентно void f() { a::f(); }
```

- «полное перекрытие» - если в производном классе определяется метод, совпадающий по имени с методом базового класса, причем в теле метода отсутствует вызов одноименного метода в базовом классе. В этом случае свойство объекта базового класса в производном классе отрицается, а метод производного класса «перекрывает» метод базового:

```
class a {
      public: void f() {}
            };
class b : public a
            {
      public:
            void f() {...}    // переопределенное свойство (метод)
            };
```

- «условное наследование» - в производном классе переопределяется метод, перекрывающий одноименный метод базового класса, но в методе базового класса обязательно имеется вызов перекрытого метода ба-

зового класса - условный или безусловный. Этот прием наиболее полно соответствует принципу развития свойств объекта, поскольку свойство в производном классе является усложненным вариантом аналогичного свойства объекта базового класса:

```
class a {
      public: void f() {}
            };
class b : public a
            {
      public:
            void f()
                  {... a::f(); .... }
            };
```

- Права доступа в производных классах

Производный класс включает в себя как личную, так и общую части базового класса. Возможны следующие варианты (рис. 12):

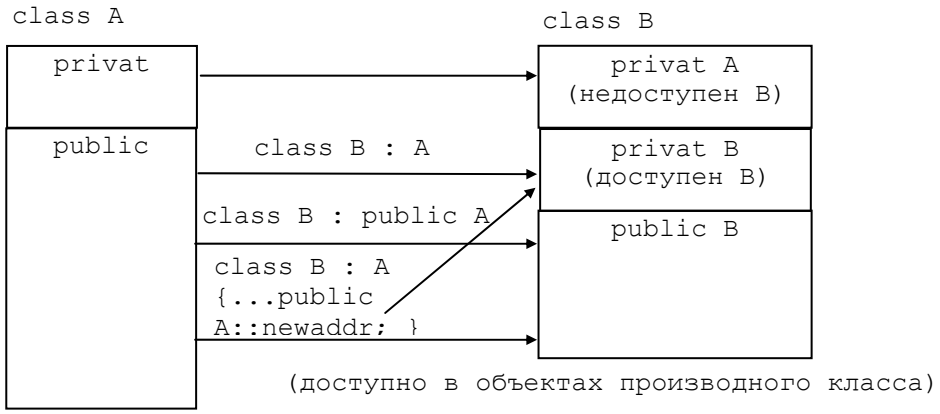


Рис. 12

- личная часть базового класса `A` всегда включается в личную часть производного класса `B`, но при этом непосредственно недоступна в классе `B`. Соответствует общему принципу защиты данных класса от вмешательства извне.

- по умолчанию, то есть при использовании заголовка `class B : A { }` общая часть класса `A` попадает в личную часть класса `B`. Это значит, что функции-элементы класса `A` доступны из функций-элементов класса `B`, но не могут быть вызваны извне при обращении к объектам класса `B`. То есть для внешнего пользователя класса `B` интерфейс класса `A` закрывается;

- при использовании заголовка `class B : public A { }` общая часть класса `A` попадает в общую часть класса `B`, и внешнему пользователю при работе с объектами класса `B` доступны интерфейсы обоих классов;

- в определении общей части класса `B` можно явно указать функции-элементы (а также данные) общей части базового класса `A`, которые попадают в общую часть класса `B`:

```
class B : A
            {
        public:
        public  A::fun;
            };
```

Из рассмотренных вариантов видно, что личная часть базового класса недоступна в любом производном классе. Однако, базовый класс может разрешить доступ к своим элементам личной части в производных классах. Это делается при помощи объявления защищенных (`protected`) элементов.

Элемент с меткой `protected` в базовом классе входит в личную часть базового класса. Кроме того, он доступен и в личной части производного класса. Если же базовый класс включается в производный как `public`, то защищенный элемент становится защищенным и в производном классе, то есть может использоваться в последующих производных классах (рис. 13).

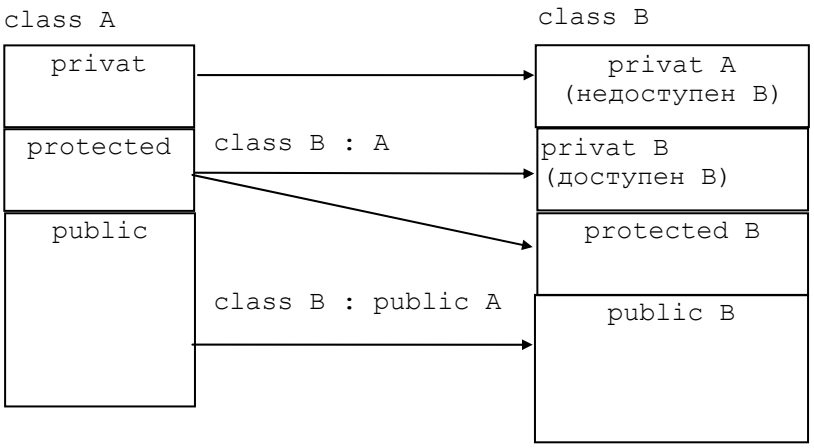


Рис.13

Пример:

```
class A
      { int     a1;     // Обычный личный элемент
protected:
        int     a2;     // Защищенный личный элемент
public:
      };
//наследование без public
```

```
class B : A              // a1,a2 в личной части B
      { void    x();
      };
void  B::x()
{ a1 = 5;                // Ошибка: a1 недоступен в B
  a2 = 3; }              // a2 доступен в личной части B
//наследование с public
class B : public A    // a2 доступен и защищен в личной
      {                  // части B, неявно имеет место
      protected: int a2;
      };
```

Таблица 5

Тип наследования доступа	Доступность в базовом классе	Доступность компонентов базового класса в производном
public	public protected private	public protected private
private	public protected private	private private не доступны

В иерархии классов существует следующее соглашение относительно доступности компонентов класса:

`private` – член класса может использоваться только функциями-членами данного класса и функциями-друзьями своего класса. В производном классе он недоступен;

`protected` – то же, что и `private`, но дополнительно член класса с данным атрибутом доступа может использоваться функциями-членами и функциями -друзьями классов, производных от данного;

`public` – член класса может использоваться любой функцией, которая является членом данного или производного класса, а также к `public` -членам возможен доступ извне через имя объекта.

Оъявление `friend` не является атрибутом доступа и не наследуется.Явно изменить умалчиваемый статус доступа при наследовании можно с помощью атрибутов доступа - `private`, `protected` и `public`, которые указываются непосредственно перед именами базовых классов.

- Указатели на объекты базового и производного классов

Преобразуя указатель на объект производного класса к указателю на объект базового класса, мы получаем доступ к вложенному объекту базового класса. В результате значение указателя (адрес памяти) на объект базового класса может оказаться не равным исходному значению указателя на объект производного. Преобразования может быть выполнено

неявно (остальные преобразования типов указателей должны быть явными):

```
class A
      {
      public: void    f1();
      };
class B : A {
public: void    f1();   // Переопределена в классе B
        void    f2();   };
A     *pa;
B     *pb;
B     x;
pa = &x;
// Неявное преобразование указателя на объект
// класса B в указатель на объект класса A
pa->f1();
// Вызов функции из вложенного объекта базового
// класса A::f1(), хотя она переопределена
```

Обратное преобразование от указателя на базовый класс к указателю на производный может быть сделано только явно. Преобразование будет корректно, если данный объект базового класса действительно входит в объект того производного класса, к типу указателя которого оно выполняется:

```
pb = (B*) pa;      // Обратное преобразование - явное
pb ->f2();         // Корректно, если под "pa" был объект класса "B"
```

Задания для выполнения

Таблица 6

Определить иерархию классов (в соответствии с вариантом). Компонентные данные класса специфицировать как protected. Указать, какие классы должны быть реализованы, какие должны быть в них конструкторы, компоненты-функции и т.д. Определить в классе статическую компоненту - указатель на начало связанного списка объектов и статическую функцию для просмотра списка (инициализировать вне определения класса, в глобальной области). Статический метод просмотра списка вызывать не через объект, а через класс. **Реализовать классы. Написать демонстрационную программу, в которой создаются объекты различных классов и помещаются в список (методом класса), после чего список просматривается. Реализовать вариант, когда объект добавляется в список при создании, т.е. в конструкторе.** **Определение классов, их реализацию, демонстрационную программу поместить в отдельные файлы.** **Далее приведен перечень классов:**

Окончание таблицы 6

1	Классы - студент, преподаватель, персона, завкафедрой;
2	Классы -служащий, персона, рабочий, инженер;.
3	Классы - рабочий, кадры, инженер, администрация;
4	Классы - деталь, механизм, изделие, узел;
5	Классы - организация, страховая компания, судостроительная компания, завод;
6	Классы - журнал, книга, печатное издание, учебник;
7	Классы - тест, экзамен, выпускной экзамен, испытание;
8	Классы - место, область, город, мегаполис;
9	Классы - игрушка, продукт, товар, молочный продукт;
10	Классы - квитанция, накладная, документ, чек;
11	Классы - автомобиль, поезд, транспортное средство, экспресс;
12	Классы - двигатель, двигатель внутреннего сгорания, дизель, турбореактивный двигатель;
13	Классы - республика, монархия, королевство, государство;
14	Классы - млекопитающие, парнокопытные, птицы, животное;
15	Классы - корабль, пароход, парусник, корвет;
16	Классы – дерево, клен, трава, растение;
17	Классы – книга, буква, слово, печатная продукция;
18	Классы – животные, грызуны, насекомые, мышь;
19	Классы – дом, комната, здание, кухня;
20	Классы – человек, женщина, ребенок, мальчик;

Контрольные вопросы

1. В чем состоит смысл наследования?

2. Истинно ли следующее утверждение: класс `D` может быть производным класса `C`, который в свою очередь является производным класса `B`, производного от класса `A`?

3. Истинно ли следующее утверждение: если конструктор производного класса не определен, то объекты этого класса будут использовать конструкторы базового класса?

4. Напишите первую строку описания класса `Tool`, который является производным классов `Color` и `Material`.

5. Допустим, что базовый и производный классы включают в себя методы с одинаковыми именами. Какой из методов будет вызван объектом производного класса, если не использована операция разрешения

имени?

Практикум №7-8. ПОЛИМОРФИЗМ И ВИРТУАЛЬНЫЕ ФУНКЦИИ

Краткие теоретические сведения

С понятием производных классов тесно связан полиморфизм. Прежде всего, *полиморфизм* - это свойство функции определенной в множестве производных классов, построенных на основе общего базового. В каждом из классов функция может быть переопределена, а может быть унаследована из базового. Свойство полиморфности заключается в том, что при отсутствии полной информации о том, к какому из классов относится объект, функция в состоянии идентифицировать его класс и корректно выполниться в этом классе.

Пусть имеется базовый класс `A` и производные классы `B,C`. В классе `A` определена функция-элемент `f()`, в классах `B,C` -унаследована и переопределена. Пусть имеется массив указателей на объекты базового класса - `p`. Он инициализирован как указателями на объекты класса `A`, так и на объекты производных классов `B,C` (точнее, на вложенные в них объекты базового класса `A`) (рис. 14):

```
class a
                { ...   void f(); };
class b : public a
                { ...   void f(); };
class c : public a
                { ...   void f(); };
        a       A1;
        b       B1;
        c       C1;
        a       *p[3] = { &B1, &C1, &A1 };
```

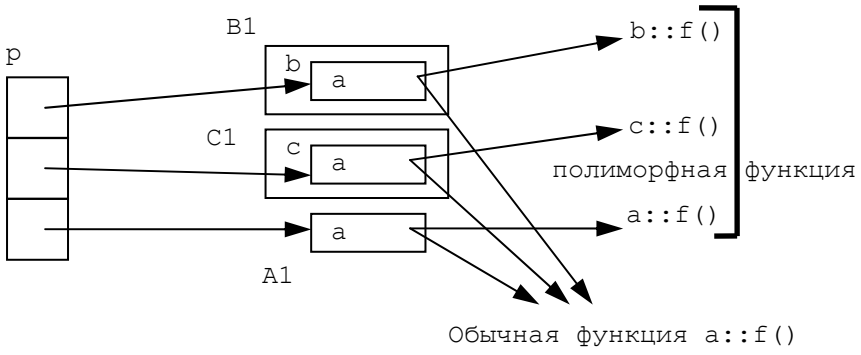


Рис. 14

Наличие указателя на объект базового класса `A` свидетельствует о том, что в данной точке программы транслятор не располагает информа-

цией о том, объект какого из производных классов расположен под указателем. Тем не менее, если функция является полиморфной, то при вызове ее по указателю на объект базового класса она должна идентифицировать его производный класс и вызвать переопределенную функцию именно для этого класса:

```
p[0]->f();              // вызов b::f() для B1
p[1]->f();              // вызов c::f() для C1
p[2]->f();              // вызов a::f() для A1
for (i=0; i<=2; i++)    // вызов b::f(),c::f(),a::f()
      p[i]->f();        // в зависимости от типа объекта
```

В C++ полиморфная функция называется виртуальной функцией. *Виртуальная функция* – функция, определенная в базовом и переопределенная (унаследованная) в группе производных классов. При вызове виртуальной функции через указатель на объект базового класса происходит вызов функции, соответствующей классу окружающего его объекта производного класса. Виртуальной может быть не только функция-элемент, но и переопределяемая операция.

Виртуальные функции предоставляют механизм позднего (отложенного) или динамического связывания. Любая нестатическая функция базового класса может быть сделана виртуальной, для чего используется ключевое слово `virtual`.

Пример:

```
class base
{
public:
  virtual void print(){cout<<"\nbase";}
. . .
};
class dir : public base
{
public:
  void print(){cout<<"\ndir";}
};
void main()
{
  base B,*bp = &B;
  dir D,*dp = &D;
  base *p = &D;
  bp ->print(); // base
  dp ->print(); // dir
  p ->print(); // dir
}
```

Таким образом, интерпретация каждого вызова виртуальной функции через указатель на базовый класс зависит от значения этого указате-

ля, т.е. от типа объекта, для которого выполняется вызов. Выбор того, какую виртуальную функцию вызвать, будет зависеть от типа объекта, на который фактически (в момент выполнения программы) направлен указатель, а не от типа указателя.

Виртуальными могут быть только нестатические функции-члены. Виртуальность наследуется. После того как функция определена как виртуальная, ее повторное определение в производном классе (с тем же самым прототипом) создает в этом классе новую виртуальную функцию, причем спецификатор `virtual` может не использоваться.

Конструкторы не могут быть виртуальными, в отличие от деструкторов. Практически каждый класс, имеющий виртуальную функцию, должен иметь виртуальный деструктор.

- Абстрактные классы

Если базовый класс используется только для порождения производных классов, то виртуальные функции в базовом классе могут быть «пустыми», поскольку никогда не будут вызваны для объекта базового класса. Базовый класс в котором есть хотя бы одна такая функция, называется *абстрактным*.

Чистой виртуальной функцией называется компонентная функция, которая имеет следующее определение:

```
virtual тип имя_функции (список_формальных_параметров) = 0;
```

Чистая виртуальная функция ничего не делает и недоступна для вызовов. Ее назначение - основа для подменяющих ее функций в производных классах. Абстрактный класс может использоваться только в качестве базового для производных классов.

Виртуальные функции в определении класса обозначаются следующим образом:

```
class base
{
      public:
      virtual print()=0;
      virtual get() =0;
};
```

Механизм абстрактных классов разработан для представления общих понятий, которые в дальнейшем предполагается конкретизировать. При этом построение иерархии классов выполняется по следующей схеме. Во главе иерархии стоит абстрактный базовый класс. Он используется для наследования интерфейса. Производные классы будут конкретизиро-

вать и реализовать этот интерфейс. В абстрактном классе объявлены чистые виртуальные функции, которые, по сути, есть абстрактные методы.

Объект абстрактного класса не может быть формальным параметром функции, однако формальным параметром может быть указатель на абстрактный класс. В этом случае появляется возможность передавать в вызываемую функцию в качестве фактического параметра значение указателя на производный объект, заменяя им указатель на абстрактный базовый класс. Таким образом, мы получаем полиморфные объекты.

- Множественное наследование

Множественным наследованием называется процесс создания производного класса из двух и более базовых. В этом случае производный класс наследует данные и функции всех своих базовых предшественников:

```
class d : public a,public b, public c { };
        d       D1;
        pd =    &D1;          // #define db sizeof(a)
        pa =    pd;           // #define dc sizeof(a)+sizeof(b)
        pb =    pd;           // pb = (char*)pd + db
        pc =    pd;           // pc = (char*)pd + dc
```

Механизм виртуальных функций при множественном наследовании имеет свои особенности. Во-первых, на каждый базовый класс в производном классе создается свой массив виртуальных функций. Во-вторых, если функция базового класса переопределена в производном, то при ее вызове требуется преобразовать указатель на объект базового класса в указатель на объект производного.

- Виртуальные базовые классы

В процессе иерархического определения производных классов может получиться, что в объект производного класса войдут несколько экземпляров объектов базового класса, например:

```
class base {}
class aa : public base {}
class bb : public base {}
class cc : aa, bb {}
```

В классе `cc` присутствуют два объекта класса `base`. Для исключения такого дублирования объект базового класса должен быть объявлен виртуальным:

```
class a : virtual public base {}
class b : virtual public base {}
class c : public a, public b {}
a       A1;
b       B1;
```

```
c       C1;
```

Объект обычного базового класса располагается, как правило, в начале объекта производного класса и имеет фиксированное смещение. Если же базовый класс является виртуальным, то требуется его динамическое размещение. Тогда в объекте производного класса на соответствующем месте размещается не сам объект базового класса, а указатель на него, который устанавливается конструктором (рис. 15 для вышерассмотренного примера).

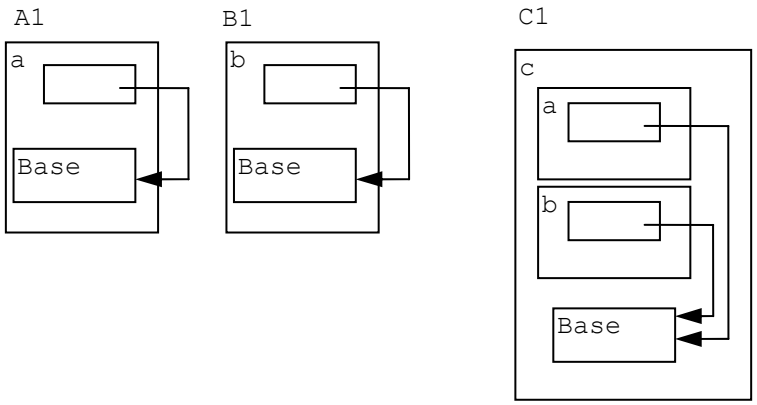


Рис. 15

- Группа

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

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

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

- Иерархия объектов

Иерархия классов есть иерархия по принципу наследования. *Иерархия объектов* – это иерархия по принципу вхождения. Объекты нижнего уровня иерархии включаются в объекты более высокого уровня, которые являются для них группой.

- Виртуальные функции как элемент объединения классов

Один из наиболее распространенных приемов использования виртуальных функции - создание базовых классов, объединяющих в единую

группу различные классы на основе некоторого общего свойства. Базовый класс при этом заключает в себе общие свойства группы, а весь набор действий, которые одинаково применимы к объектам из любого класса, реализуется через виртуальные функции.

В качестве примера рассмотрим группу классов - типов данных. Допустим, проектируется база данных, предназначенная для хранения произвольных объектов (типов данных). Прежде всего, определяется ряд общих действий, которые обязательно должны быть выполнимы для объекта любого класса:

```
class ADT
{
public:
virtual int  Get(char *)=0;          // Загрузка объекта из строки
virtual char *Put()=0;               // Выгрузка объекта в строку
virtual long Append(BinFile&)=0;     // Добавить объект в двоичный файл
virtual int  Load(BinFile&)=0;       //
virtual int  Type()=0;               // Возвращает идентификатор
                                     // типа объекта
virtual     char  *Name()=0;         // Возвращает имя типа объекта
virtual     int    Cmp(ADT *)=0;     // Сравнивает значения объектов
virtual     ADT   *Copy()=0;         // Создает динамический объект -
                                     // копию с себя
virtual ~ADT(){};                    // Виртуальный деструктор
};
```

Базовый класс получился абстрактным, то есть его объект не содержит данных, а функции являются «пустыми». Это значит, что объекты базового класса не могут создаваться в программе. Базовый класс может содержать данные и непустые функции, если в самой группе классов можно выделить некоторую общую часть.

Естественно, что при проектировании любого производного класса должен соблюдаться приведенный шаблон, то есть в первую очередь в нем должны быть реализованы виртуальные функции, которые поддерживают в нем перечисленные действия. Остальная часть класса может быть какой угодно.

Базовый класс и набор виртуальных функций используются как общий интерфейс доступа к объектам - типам данных при проектировании базы данных. В качестве примера рассмотрим фрагмент класса - массив указателей, который здесь выступает аналогом базы данных:

```
class MU
      {
      int sz;
      ADT **p;
public:...
```

```
ADT  *min();            //  Поиск минимального
      void  sort();     // Сортировка
      int   test();           // Проверка на идентичность типов
};
// Вызов виртуальных функций отмечен "***"
int   MU::test()
{
for (i=1; p[i]!=NULL; i++)
if (p[i]->Type()!=p[i-1]->Type()) return 0;     //***
return 1;
}
ADT *MU::min()
{ ADT *pmin; int i;
if (p[0]==NULL || !test()) return NULL;
for (i=0, pmin=p[0]; p[i]!=NULL; i++)
      if (pmin->Cmp(p[i]) > 0) pmin=p[i];       //***
return pmin;
}
void MU:sort()
{ int d,i; void *q;
if (p[0]==NULL || !test()) return;
do    {
      for (d=0, i=1; p[i]!=NULL; i++)
      if (p[i-1]->Cmp(p[i]) > 0)                //***
            {d++; q=p[i-1]; p[i-1]=p[i]; p[i]=q; }
      }
while (d!=0);}
```

- Динамическая идентификация типов

Динамическая идентификация типа характерна для языков, в которых поддерживается полиморфизм. В этих языках возможны ситуации, в которых тип объекта на этапе компиляции неизвестен. Пусть группа содержит объекты различных классов и необходимо выполнить некоторые действия только для объектов определенного класса. Тогда в итераторе необходимо распознавать тип очередного объекта.

В стандарт языка C++ включены средства RTTI (Run-Time Type Idendification) - динамическая идентификация типов. Информацию о типе объекта получают с помощью оператора `typeid`, определение которого содержит заголовочный файл `<typeinfo.h>`.

Имеется две формы оператора `typeid`:

```
typeid (объект)
typeid (имя_типа)
```

Оператор `typeid` возвращает ссылку на объект типа `type_info`. В классе `type_info` перегруженные операции == и != обеспечивают сравнение типов. Функция `name()` возвращает указатель на имя типа.

Оператор `typeid` работает корректно только с объектами, у которых определены виртуальные функции. Большинство объектов имеют вирту-

альные функции, хотя бы потому, что обычно деструктор является виртуальным для устранения потенциальных проблем с производными классами. Когда оператор `typeid` применяют к неполиморфному классу (в классе нет виртуальной функции), получают указатель или ссылку базового типа. Пример:

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

Если при обращении `typeid(*p), p=NULL`, то возникает исключительная ситуация `bad_typeid`.

Задания для выполнения

Таблица 7

Использовать проект созданный в практикуме №6. Написать виртуальный метод просмотра списока. Дополнить созданную иерархию классами «группа». Рекомендуется создать абстрактный класс – «подразделение», который будет предком всех групп и абстрактный класс Object, находящийся во главе всей иерархии. Написать для класса-группы метод-итератор. Написать функцию, которая выполняется для всех объектов, входящих в группу. Написать демонстрационную программу, в которой создаются, показываются и разрушаются объекты-группы, а также демонстрируется использование итератора. Методы-итераторы определяется в неабстрактных классах-группах на основе приведенных в вариантах запросов. Далее приведены варианты запросов:	
1	запросы - имена студентов указанного курса, имена и должность преподавателей указанной кафедры.
2	запросы - имена служащих со стажем не менее заданного, имена служащих заданной профессии
3	запросы - имена рабочих заданного цеха, количество инженеров в заданном подразделении.
4	запросы - наименование всех деталей (узлов), входящих в заданный узел

	(механизм).

Окончание таблицы 7

5	запрсы - количество служащих со стажем не менее заданного.
6	запросы - наименование всех книг в библиотеке (магазине), вышедших не ранее указанного года; суммарное количество учебников в библиотеке (магазине).
7	запросы - имена студентов, сдавших все (заданный) экзамены на отлично (хорошо и отлично), имена студентов, не сдавших все (хотя бы один) экзамен.
8	запросы - названия всех городов заданной области, суммарное количество жителей всех городов в области.
9	запросы - наименование всех товаров в заданном отделе магазина, суммарная стоимость товара заданного наименования.
10	запросы - суммарная стоимость продукции заданного наименования по всем накладным, количество чеков.
11	запросы - количество указанного транспортного средства в автопарке (на автостоянке), количество пассажиров во всех вагонах экспресса.
12	запросы - средняя мощность всех дизелей, обслуживаемых заданной фирмой.
13	запросы - имена всех монархов на заданном континенте, количество демократических государств.
14	запросы - средний вес животных заданного вида в зоопарке, количество хищних птиц.
15	запросы - среднее водоизмещение всех парусников на верфи (в порту).
16	запросы - средний возраст деревьев, высота кленов.
17	запросы - количество страниц в книгах.
18	запросы - количество грыцзунов определенного цвета.
19	запросы - средний размер комнат, среднее количество квартир в здании.
20	запросы - имена всех детей, количество всех женщин.

Контрольные вопросы

1. Когда используется абстрактный класс?
2. Что такое чистая виртуальная функция?
3. Определите виртуальную функцию `dang()`, возвращающую результат типа `void` и имеющую аргумент типа `int`.
4. Как называется отношение между классами, когда один класс содержит в качестве составной части объекты другого класса?
5. Есть ли какие либо особенности виртуальных функций при множественном наследовании?

Практикум №9. ПЕРЕГРУЗКА ОПЕРАЦИЙ

Краткие теоретические сведения

Следующим шагом в использовании класса как базового типа данных является переопределение или перегрузка операций языка, в которых один или несколько операндов могут быть объектами класса. Это достигается введением функции-элемента специального вида, обращение к которой компилятор формирует при трансляции такой операции. Такая функция должна иметь результат, отличный от `void`, если предполагается использование этой операции внутри другого выражения.

Для каждой комбинации типов операндов в переопределяемой операции необходимо ввести отдельную функцию, то есть транслятор не может производить перестановку операндов местами, даже если базовая операция допускает это.

Для перегрузки операции используется особая форма функции-элемента с заголовком такого вида:

```
operator операция( список_параметров-операндов)
```

Имя функции состоит из ключевого слова `operator` и символа данной операции в синтаксисе языка. Список формальных параметров функции соответствует списку операндов, определяя их типы и способы передачи. Результат функции (тип, способ передачи) является одновременно результатом перегружаемой операции.

Имеется два способа описания функции, соответствующей перегружаемой операции:

- если функция задается как обычная функция-элемент класса, то первым операндом операции является объект класса, указатель на который передается неявным параметром `this`;

- если первый операнд перегружаемой операции не является объектом некоторого класса, либо требуется передавать в качестве операнда не указатель, а сам объект (значение), то соответствующая функция должна быть определена как дружественная классу с полным списком аргументов. Полное имя такой функции не содержит имени класса.

В качестве примера рассмотрим переопределение операций расботы со строками:

```
class String
{
public:
     . . .
     String operator+(const String& s) const;//операция сложения строк
```

```
     bool operator<(const String& s) const; //операция сравнения строк
};
String::operator+(const String& s) const
{
     String result;
     result.length = length + s.length;
     result.str = new char [result.length + 1];
     strcpy(result.str, str);
     strcat(result.str, s.str);
     return result;
}
bool
String::operator<(const String& s) const
{
     char* cp1 = str;
     char* cp2 = s.str;
     while (true) {
          if (*cp1 < *cp2)
               return true;
          else if (*cp1 > *cp2)
               return false;
          else {
               cp1++;
               cp2++;                 if (*cp2 == 0)
                    return false;
               else if (*cp1 == 0)
                    return true;
          }
     }
}
```

- Перегрузка унарных операций

Любая унарная операция может быть определена двумя способами: либо как компонентная функция без параметров, либо как глобальная (возможно дружественная) функция с одним параметром.

Унарные операции, перегружаемые в рамках определенного класса, могут перегружаться только через нестатическую компонентную функцию без параметров. Вызываемый объект класса автоматически воспринимается как операнд. Формат:

тип_возвр_значения operator знак_операции

Пример:

```
class person
{ int age;
  ...
public:
  …
  void operator++(){ ++age;}
};
void main()
```

```
{class person jon;
    ++jon;
}
```

Унарные операции, перегружаемые вне области класса (как глобальные функции), должны иметь один параметр типа класса. Передаваемый через этот параметр объект воспринимается как операнд. Формат:

тип_возвр_значения operator знак_операции(идентификатор_типа)

Пример:

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

- Перегрузка бинарных операций

Любая бинарная операция может быть определена либо как компонентная функция с одним параметром, либо как глобальная (возможно дружественная) функция с двумя параметрами. В первом случае x⊕y означает вызов x.operator⊕(y), во втором - вызов `operator` ⊕(x,y).

- Перегрузка операции присваивания

Для операции присваивания характерно следующее:

- операция не наследуется;

- операция определена по умолчанию для каждого класса в качестве операции поразрядного копирования объекта, стоящего справа от знака операции, в объект, стоящий слева;

- операция может перегружаться только в области определения класса. Это гарантирует, что первым операндом всегда будет леводопустимое выражение.

Формат перегруженной операции присваивания:

имя_класса& operator=(имя_класса&);

Существуют две важные особенности функции `operanor=`. Во-первых, в ней используется параметр-ссылка. Это необходимо для предотвращения создания копии объекта, передаваемого через параметр по значению. Во-вторых, функция `operator=()` возвращает не объект, а ссыл-

ку на него. Смысл этого тот же, что и при использовании параметра-ссылки. Пример:

```
class String {
      public:
      String& operator=(const String& s);
      };
String&    String::operator=(const String& s)
{
     if (this == &s)            return *this;
     if (str != 0)
      {
            delete [] str;
      }
     length = s.length;
     str = new char[length + 1];
     if (str == 0) {         // обработка ошибок       }
     strcpy(str, s.str);
     return *this;
}
```

Задания для выполнения

Таблица 8

Создать заданный в варианте класс. Определить в классе конструкторы, деструктор, необходимые функции и заданные перегруженные операции. Написать программу тестирования, в которой проверяется использование перегруженных операций.	
1	Класс – одномерный массив. Дополнительно перегрузить следующие операции: * – умножение массивов; [] – доступ по индексу.
2	Класс – одномерный массив. Дополнительно перегрузить следующие операции: int() – размер массива; [] – доступ по индексу.
3	Класс – одномерный массив. Дополнительно перегрузить следующие операции: [] – доступ по индексу; = = – проверка на равенство; != – проверка на неравенство.
4	Класс – множество set. Дополнительно перегрузить следующие операции: + – добавить элемент в множество (типа set+item); + – объединение множеств; * – пересечение множеств.
5	Класс – множество set. Дополнительно перегрузить следующие операции: + – добавить элемент в множество (типа item + set); + – объединение множеств; = = – проверка множеств на равенство.

Продолжение таблицы 8

6	Класс – множество set. Дополнительно перегрузить следующие операции:

	- – удалить элемент из множества (типа set-item); * – пересечение множеств; < – сравнение множеств.
7	Класс – множество set. Дополнительно перегрузить следующие операции: - – удалить элемент из множества (типа set-item); > – проверка на подмножество; != – проверка множеств на неравенство.
8	Класс – множество set. Дополнительно перегрузить следующие операции: + – добавить элемент в множество (типа set+item); * – пересечение множеств; int()– мощность множества.
9	Класс – множество set. Дополнительно перегрузить следующие операции: () – конструктор множества (в стиле конструктора для множественного типа в языке Pascal); + – объединение множеств; <= – сравнение множеств.
10	Класс – множество set. Дополнительно перегрузить следующие операции: > – проверка на принадлежность (типа операции in множественного типа в языке Pascal); * – пересечение множеств; < – проверка на подмножество.
11	Класс – однонаправленный список list. Дополнительно перегрузить следующие операции: + – добавить элемент в начало (list+item); -- – удалить элемент из начала (--list); = = – проверка на равенство.
12	Класс – однонаправленный список list. Дополнительно перегрузить следующие операции: + – добавить элемент в начало (item+list); -- – удалить элемент из начала (--list); != – проверка на неравенство.
13	Класс – однонаправленный список list. Дополнительно перегрузить следующие операции: + – добавить элемент в конец (list+item); -- – удалить элемент из конца (типа list--); != – проверка на неравенство.

14	Класс – однонаправленный список list. Дополнительно перегрузить следующие операции: [] – доступ к элементу в заданной позиции, например:
	`Type c;`
	`int i;`
	`list L;`
	`c=L[i];`
	+ – объединить два списка; = = – проверка на равенство.

15	Класс – однонаправленный список list. Дополнительно перегрузить следующие операции: [] – доступ к элементу в заданной позиции, например:
	`int i; Type c;`
	`list L;`
	`c=L[i];`
	+ – объединить два списка; != – проверка на неравенство.

Окончание таблицы 8

16	Класс – однонаправленный список list. Дополнительно перегрузить следующие операции: () – удалить элемент в заданной позиции, например:
	`int i;`
	`list L;`
	`L(i);`
	() – добавить элемент в заданную позицию, например:
	`int i;`
	`Type c;`
	`list L;`
	`L(c,i);`
	!= – проверка на неравенство.

17	Класс – стек stack. Дополнительно перегрузить следующие операции: + – добавить элемент в стек; -- – извлечь элемент из стека; bool() – проверка, пустой ли стек.
18	Класс – очередь queue. Дополнительно перегрузить следующие операции: + – добавить элемент; -- – извлечь элемент; bool() – проверка, пустая ли очередь.
19	Класс – одномерный массив. Дополнительно перегрузить следующие операции: + – сложение массивов; [] – доступ по индексу; + – сложить элемент с массивом.
20	Класс – одномерный массив. Дополнительно перегрузить следующие операции: - – вычитание массивов; [] – доступ по индексу; - – вычесть из масива элемент

Контрольные вопросы

1. Какаво назаначение переопределения операций в клаасах?
2. Зачем при переопределении опертора в форме внешней глобальной функции она объявляется дружественной классу?
3. Могут ли унарные операции, переопределяемые в рамках определенного класса, перегружаться через статическую компонентную функцию без параметров?
4. Перечислите особенности переопределения операции присваивания.
5. Почему операция присваивания может перегружаться только в области определения класса?

Практикум № 10-11. ШАБЛОНЫ

Краткие теоретические сведения

Шаблон (template) - макроопределение (текстовая заготовка) класса с параметром - типом данных. *Макроопределение* - способ определения множества классов или функций с параметризованным внутренним типом данных. Функции и классы часто называю соответственно *параметрезированная функция* (generic functions) и *параметризированными классами* (generic classes).

Назначение шаблонов - сокращение времени разработки, числа уникальных функций и имен классов (в одном проекте), требующих определения.

- Шаблоны классов

Общая форма объявления параметризованного класса:

```
template <class тип_данных> class имя_класса { . . . };
```

Основные свойства шаблонов классов:

- компонентные функции параметризованного класса автоматически являются параметризованными. Их не обязательно объявлять как параметризованные с помощью `template`;

- дружественные функции, которые описываются в параметризованном классе, не являются автоматически параметризованными функциями, т.е. по умолчанию такие функции являются дружественными для всех классов, которые организуются по данному шаблону;

- если дружественная функция содержит в своем описании параметр типа параметризованного класса, то для каждого созданного по данному шаблону класса имеется собственная дружественная функция;

- в рамках параметризованного класса нельзя определить дружественные шаблоны (дружественные параметризованные классы);

- с одной стороны, шаблоны могут быть производными (наследоваться) как от шаблонов, так и от обычных классов, с другой стороны, они могут использоваться в качестве базовых для других шаблонов или классов;

- шаблоны функций, которые являются членами классов, нельзя описывать как `virtual`;

- локальные классы не могут содержать шаблоны в качестве своих элементов.

Пример:

```
template < class T > class Funct
```

```
{           // <class T> - параметр шаблона
            //класс "T", внутренний тип данных
            // Funct    -  имя группы шаблонных классов
      public:
            Funct(T value)
            {
                  x=value;
            }
            T GetValue()
            {
                  return (x*x)-(2*x);
            }
      private:
            T x;
};
void main()
{
  Funct< int > cI(25);
  Funct< double > cD(3.12);
  cout << "cI " << cI.GetValue() << endl;
  cout << "cD " << cD.GetValue() << endl;
}
```

- Шаблоны функций

Функции-элементы шаблона классов в свою очередь также являются шаблонными функциями с тем же самым параметром. Общая форма объявления:

```
template <class тип_данных> тип_возвр_значения
            имя_функции(список_параметров){тело_функции}
```

Основные свойства шаблонов функций:

- имена параметров шаблона должны быть уникальными во всем определении шаблона;

- список параметров шаблона не может быть пустым;

- в списке параметров шаблона может быть несколько параметров, и каждому из них должно предшествовать ключевое слово `class`;

- имя параметра шаблона имеет все права имени типа в определенной шаблоном функции;

- определенная с помощью шаблона функция может иметь любое количество непараметризованных формальных параметров. Может быть непараметризованно и возвращаемое функцией значение;

- в списке параметров прототипа шаблона имена параметров не обязаны совпадать с именами тех же параметров в определении шаблона;

- при конкретизации параметризованной функции необходимо, чтобы при вызове функции типы фактических параметров, соответствующие

одинаково параметризованным формальным параметрам, были одинаковы;

- шаблоны могут иметь также и параметры-константы, которые используются для статического определения размерностей внутренних структур данных. Кроме того, шаблон может использоваться для размещения не только указателей на параметризованные объекты, но и самих объектов.

В качестве примера рассмотрим шаблон замены переменных местами:

```
template <class T> void swap  (T& left, T& right)
      {
      T temp=left;
      left=right;
      right=temp;
     }
```

Задания для выполнения

Таблица 9

Модифицировать проект, созданный в предыдущем практикуме №9, следующим образом. Создать шаблон заданного класса. Проверить использование шаблона для стандартных типов данных (в качестве стандартных типов использовать целые и вещественные типы). Определить пользовательский класс, который будет использоваться в качестве параметра шаблона (для пользовательского типа взять класс из проекта практикума №5). Реализацию шаблона следует разместить вместе с определением в заголовочном файле. Написать функцию-шаблон в соответсвии с вариантом. Проверить работу функции (с int, long, double и char).	
1	Написать функцию-шаблон, вычисляющую максимальное значение в массиве. Аргументами функции должны быть имя и размер массива.
2	Написать функцию-шаблон, переставляющую элементы в массиве.
3	Написать параметризованную функцию сортировки методом отбора.
4	Написать параметризованную функцию сортировки методом быстрой сортировки.
5	Написать параметризованную функцию сортировки методом вставки.
6	Создать шаблон функции прореживания: три параметра: откуда выбирать, куда выбирать и через сколько выбирать.
7	Написать функцию-шаблон, вычисляющую среднее значение в массиве. Аргументами функции должны быть адрес и размер массива.
8	Написать функцию-шаблон, вычисляющую максимальное значение в массиве. Аргументами функции должны быть адрес и размер массива.
9	Написать функцию-шаблон, вычисляющую минимальное значение в

	массиве. Аргументами функции должны быть имя и размер массива

Окончание таблицы 9

10	Написать родовую функцию в виде функции-шаблон. Функция меняет местами два аргумента
11	Написать параметризованную функцию - сортировка методом Шелла.
12	Написать функцию-шаблон бинарного поиска.
13	Написать функцию-шаблон, вычисляющую минимальное значение в массиве. Аргументами функции должны быть адрес и размер массива.
14	Создать шаблон функции прореживания: два параметра: откуда выбирать и через сколько выбирать.
15	Создать шаблон функции min, которая возвращает минимальное из трех заданных значений.Создать несколько экземпляров шаблона для данных различных типов.
16	Создать шаблон функции, которая возвращает максимально близкое к заданному элементу из значений массива. Аргументами функции должны быть значение, адрес и размер массива.
17	Написать функцию-шаблон, удваивающую элементы в массиве. Аргументами функции должны быть имя и размер массива.
18	Написать функцию-шаблон, удаляющую элементы равные заданному в массиве. Аргументами функции должны быть значение, адрес и размер массива.
19	Написать функцию-шаблон, которая подсчитывет количество элементов больших заданному в массиве. Аргументами функции должны быть значение, имя и размер массива.
20	Написать параметризованную функцию - сортировка методом пузырька.

Контрольные вопросы

1. Каково назначение шаблонов?
2. Определите шаблон класса «вектор» - одномерный массив.
3. Как записать параметр шаблона?
4. Можно ли перегружать параметризованные функции?
5. Как определяются компонентные функции параметризованных классов вне определения шаблона класса?

Практикум №12. ПОТОКОВЫЕ КЛАССЫ

Краткие теоретические сведения

Потоковые классы представляют объектно-ориентированный вариант функций ANSI-C.

- Потоковые классы в C++

Библиотека потоковых классов C++ построена на основе двух базовых классов: `ios` и `streambuf` .

Класс `streambuf` обеспечивает организацию и взаимосвязь буферов ввода-вывода, размещаемых в памяти, с физическими устройствами ввода-вывода (см. схему иерархии рис. 16). Методы и данные класса `streambuf` обычно явно не используются. Этот класс нужен другим классам библиотеки ввода-вывода.

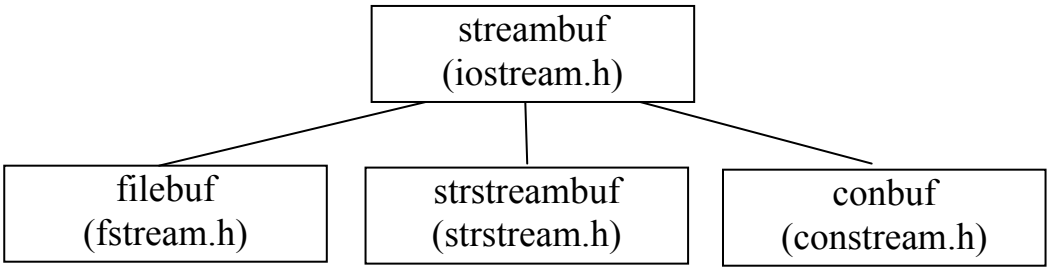


Рис. 16

Класс `ios` содержит средства для форматированного ввода-вывода и проверки ошибок (рис. 17): `istream` - класс входных потоков; `ostream` - класс выходных потоков; `iostream` - класс ввода-вывода; `istrstream` - класс входных строковых потоков; `ifstream` - класс входных файловых потоков и т.д.

Потоковые классы, их методы и данные становятся доступными в программе, если в неё включены заголовочные файлы:

`iostream.h` - для `ios, ostream, istream`;

`strstream.h` - для `strstream, istrstream, ostrstream;`

`fstream.h` - для `fstream, ifstream, ofstream.`

- Базовые потоки ввода-вывода

Для ввода с потока используются объекты класса `istream`, для вывода в поток - объекты класса `ostream`..

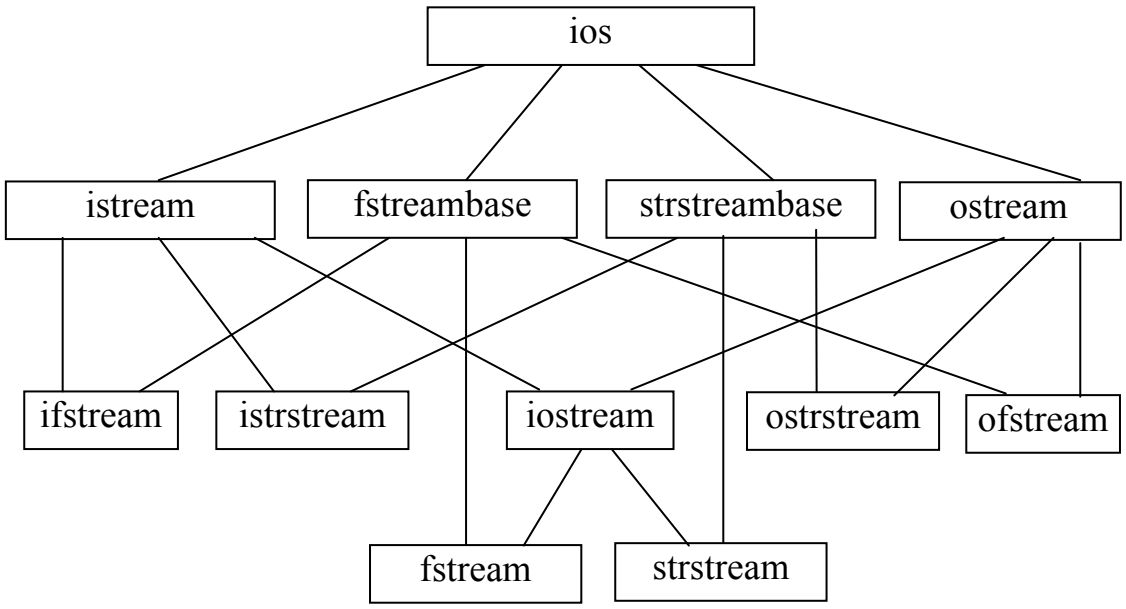


Рис. 17

В классе `istream` определены следующие функции:

```
istream& get(char* buffer,int size,char delimiter='\n');
                              //извлечение символов
istream& read(char* buffer,int size); //чтение символов
istream& getline(char* buffer,int size, char delimiter='\n');
                                    //извлечение строк
istream& get(streambuf& s,char delimiter='\n');
                                   //извлечение данных из файла
istream get (char& C);             // извлечение символа
int get();                         // извлечение символа
int peek();                        //возврат символа
int gcount();                      //количество считанных символов
istream& putback(C);
//размещение символа в область get при наличии свободного пространства
istream& ignore(int count=1,int target=EOF);
           //извлечение символа из istream при определенных условиях
```

В классе `ostream` определены следующие функции:

```
ostream& put(char C); // размещение символа в ostream
ostream& write(const char* buffer,int size);
                 //запись в ostream содержимого буферв
ostream& flush();// сброс буфера streambuf.
istream& seekg(long p);
           // позиционирование указателя при чтении от начала буфера
istream& seekg(long p,seek_dir point);
```

```
            // указание начальной точки перемещение при чтении
enum seek_dir{beg,curr,end};// позиционирование указателя при чтении
long tellg();// возрат текущего положения указателя при чтении
ostream& seekp(long p);
            //позиционирование указателя при записи от начала буфера
ostream& seekp(long p,seek_dir point); //указание точки отсчета
long tellp();//возврат текущего положения указателя
```

Помимо этих функций в классе `istream` перегружена операция `>>`, а в классе `ostream` `<<`. Операции `<<` и `>>` имеют два операнда. Левым операндом является объект класса `istream (ostream)`, а правым - данное, тип которого задан в языке.

Для того чтобы использовать операции `<<` и `>>` для всех стандартных типов данных используется соответствующее число перегруженных функций `operator<<` и `operator>>`. При выполнении операций ввода-вывода в зависимости от типа правого операнда вызывается та или иная перегруженная функция `operator`.

Поддерживаются следующие типы данных: целые, вещественные, строки (`char*`). Для вывода - `void*` (все указатели, отличные от `char*`, автоматически переводятся к `void*`). Перегрузка операции `>>` и `<<` не изменяет их приоритета.

Функции `operator<<` и `operator>>` возвращают ссылку на тот потоковый объект, который указан слева от знака операции. При вводе-выводе можно выполнять форматирование данных.

Чтобы использовать операции `>>` и `<<` с данными пользовательских типов, определяемых пользователем, необходимо расширить действие этих операций, введя новые операции-функции. Первым параметром операции-функции должна быть ссылка на объект потокового типа, вторым - ссылка или объект пользовательского типа.

В файле `iostream.h` определены следующие объекты, связанные со стандартными потоками ввода-вывода: `cin` - объект класса `istream`, связанный со стандартным буферизированным входным потоком; `cout` - объект класса `ostream`, связанный со стандартным буферизированным выходным потоком; `cerr` - не буферизированный выходной поток для сообщения об ошибках; `clog` - буферизированный выходной поток для сообщения об ошибках.

- Форматирование

Непосредственное применение операций ввода `<<` и вывода `>>` к стандартным потокам `cout, cin, cerr, clog` для данных базовых типов приводит к использованию «умалчиваемых» форматов внешнего представления пересылаемых значений.

Форматы представления выводимой информации и правила восприятия данных при вводе могут быть изменены с помощью флагов форматирования. Они унаследованы всеми потоками из базового класса `ios`, реализованы в виде отдельных фиксированных битов и хранятся в `protected` компоненте класса `long x_flags`.

- Манипуляторы

Несмотря на гибкость и большие возможности управления форматами с помощью компонентных функций класса `ios`, их применение достаточно громоздко. Более простой способ изменения параметров и флагов форматирования обеспечивают манипуляторы.

Манипуляторы - специальные функции, позволяющие модифицировать работу потока. Их можно использовать в качестве правого операнда операции `>>` или `<<`. В качестве левого операнда, как обычно, используется поток (ссылка на поток), и именно на этот поток воздействует манипулятор.

При использовании манипуляторов следует включить заголовочный файл `<iomanip.h>`, в котором определены встроенные манипуляторы.

- Определение пользовательских манипуляторов

Рассмотрим пример создания пользовательского манипулятора с параметрами для вывода. Определяется класс (`my_man`) с полями: параметры манипулятора, указатель на функцию `(*f)`. Определяется конструктор класса (`my_man`) с инициализацией полей. Определяется дружественная функция - `operator<<`, котоая в качестве правого аргумента принимает объект класса (`my_man)`, левого аргумента (операнда) поток `ostream` и возвращает поток `ostream` как результат выполнения функции (`*f)`:

```
typedef far ostream&(far *PTF)(ostream&,int,int,char);
class my_man{
            int w;int n;char fill;
            PTF f;
      public:
            //конструктор
      my_man(PTF F,int W,int N,char FILL):f(F),w(W),n(N),fill(FILL){}
            friend ostream& operator<<(ostream&,my_man);
      };
ostream& operator<<(ostream& out,my_man my)
      {
      return my.f(out,my.w,my.n,my.fill);
      }
```

Определяется функция типа `*f (fmanip)`, принимающая поток и параметры манипулятора и возвращающя поток. Эта функция собственно и выполняет форматирование:

```
ostream& fmanip(ostream& s,int w,int n,char fill)
      {     s.width(w);
            s.flags(ios::fixed);
            s.precision(n);
            s.fill(fill);
            return s;
      }
```

Определяется собственно манипулятор `(wp)` как функция, принимающая параметры манипулятора и возвращающаяю объект `my_manip`, поле `f` которой содержит указатель на функцию `fmanip`:

```
my_man wp(int w,int n,char fill)
      {return my_man(fmanip,w,n,fill); }
```

Для создания пользовательских манипуляторов с параметрами можно использовать макросы, которые содержатся в файле `<iomanip.h>`.

- Состояние потока

Каждый поток имеет связанное с ним состояние. Состояния потока описываются в классе `ios` в виде перечисления `enum`:

```
public:
enum io_state{
      goodbit, //нет ошибки 0X00
      eofbit,     //конец файла 0X01
      failbit,    //последняя операция не выполнилась 0X02
      badbit,     //попытка использования недопустимой операции 0X04
      hardfail    //фатальная ошибка 0X08
};
```

Флаги, определяющие результат последней операции с объектом `ios`, содержатся в переменной `state`. Получить значение этой переменной можно с помощью функции `int rdstate()`.

Кроме того, проверить состояние потока можно функциями:

```
int bad();     //        1, если badbit или hardfail
int eof();     //        1, если eofbit
int fail();    //        1, если failbit, badbit или hardfail
int good();    //        1, если goodbit
```

Если операция `>>` используется для новых типов данных, то при её перегрузке необходимо предусмотреть соответствующие проверки.

- Файловый ввод-вывод

Потоки для работы с файлами создаются как объекты следующих классов: `ofstream` - запись в файл; `ifstream` - чтение из файла; `fstream` - чтение/запись.

Для создания потоков имеются следующие конструкторы:

```
fstream(); //создает поток, не присоединяя его ни к какому файлу
fstream(const char* name,int mode,int p=filebuf::openprot);
            // создает поток, присоединяет его к файлу
```

Флаги режима определены в классе `ios` и имеют следующие значения: `in` – для чтения; `out` – для записи; `ate` – индекс потока помещен в конец файла; `app` – поток открыт для добавления данных в конец; `trunc` – усечение существующего потока до нуля; `nocreate` - открытие потока будет завершена неудачно, если файл не существует; `noreplace` - открытие потока будет завершена неудачно, если файл существует; `binary` - поток открывается для двоичного обмена.

Если при создании потока он не присоединен к файлу, то присоединить существующий поток к файлу можно функцией:

```
void open(const char* name,int mode,int p=filebuf::openprot);
```

Длс сбраса буфера потока, отсоединения потока от файла и закрытия файла используется функция:

```
void fstreambase::close();
```

Создать поток и связать его с файлом можно тремя способами:

1. Создается объект `filebuf: filebuf fbuf;` объект `filebuf` связывается с устройством (файлом): `fbuf.open("имя",ios::in);` создается поток и связывается с `filebuf: istream stream(&fbuf);`

2. Создается объект `fstream (ifstream, ofstream): fstream stream;` открывается файл, который связывается через `filebuf` с потоком: `stream.open("имя",ios::in);`

3. Создается объект `fstream`, одновременно открывается файл, который связывается с потоком: `fstream stream("имя",ios::in);`

- Строковые потоки. Строки класса string

Работу со строковыми потоками обеспечивают классы `istringstream, ostringstream, stringstream`. Для их испольования необходимо подключить файл `<sstream>`. Применение потоков аналогично применению файловых потоков, с той лишь разницей, что размещение информации происходит в оперативной памяти.

Класс string берет на себя управление памятью при первоначальном размещении строки и при всех ее модификациях, увеличивающих и уменьшающих длину строки. Чтобы использовать строки необходимо подключить заголовочный файл `<string>`. Класс содержит несколько конструкторов, три операции присваивания, конкатенации, равенства, неравенства, меньше, больше, индексации, ввод-вывод, добавления и т.д. и ряд методов:

```
    ...
ifstream fin ("a.txt",ios::in|ios::nocreated); //открываем файл
string word;
ostingstream sentence;
```

```
while (!fin.eof())
      {
      char symb;
      while(iLimit(symb=f in.peek()))
            {
            sentence << symb;
            if (symb == '\n') break;
            fin.seekg(1,ios::cur);
            }
      fin >> word;
      sentence << word;
      char last=word[word.size()-1];
      if ((last == '.')||(last == '!')
            sentence.str(""); // очистка потока
      }
      ...
```

Задания для выполнения

Модифицировать проек, созданный в практикумах №6, 7-8: релизовать ввод-вывод данных на экран и в файл через потоковые классы, решить проблему ввода-выода кириллицы посредством разработанных классов.

Контрольные вопросы

1. Объяснить понятие буфера в операциях ввода/вывода.
2. Каково назначение класса `ios`?
3. Для чего используются перегруженные операции << и >>?
4. Как создать поток для файлового ввода-вывода?
5. Как проверить состояние потока?

Практикум №13-14. СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ (STL). АЛГОРИТМЫ

Краткие теоретические сведения

STL (Standart Template Library) - стандартная библиотека шаблонов, которя представляет большой набор данных структур и алгоритмов. В STL большое количество шаблонов, как классов, так и функций. Их можно использовать с ООП или без него.

Библиотека содержит пять основных видов компонентов: алгоритм (algorithm) - определяет вычислительную процедуру; контейнер (container) - управляет набором объектов в памяти; итератор (iterator)- обеспечивает для алгоритма средство доступа к содержимому контейнера; функциональный объект (function object) - инкапсулирует функцию в объекте для использования другими компонентами; адаптер (adaptor) - адаптирует компонент для обеспечения различного интерфейса.

Контейнеры (containers) - это объекты, предназначенные для хранения других элементов (вектор, линейный список, множество). В каждом классе-контейнере определен набор функций для работы с ними. Например, список содержит функции для вставки, удаления и слияния элементов. У каждого контейнера имеется определенный для него распределитель памяти (allocator), который управляет процессом выделения памяти для контейнера. По умолчанию распределителем памяти является объект класса allocator. Можно определить собственный распределитель.

Алгоритмы (algorithms) выполняют операции над содержимым контейнера. Существуют алгоритмы для инициализации, сортировки, поиска, замены содержимого контейнеров. Многие алгоритмы предназначены для работы с последовательностью (sequence), которая представляет собой линейный список элементов внутри контейнера.

Итераторы (iterators) - это объекты, которые по отношению к контейнеру играют роль указателей. Они позволяют получить доступ к содержимому контейнера примерно так же, как указатели используются для доступа к элементам массива. С итераторами можно работать так же, как с указателями. К ним можно применить операции *, инкремента, декремента.

- Пространство имен

Пространство имен (namespace) используется для программ созданых из многих файлов, в которых есть опасность конфликта имен. Объявляется пространство имен командой `namespace`. Формат определения:

```
namespace [идентификатор]
        {
         описание для этой рабочей области
        }
```

Для использования рабочей области применяется команда `using namespace`:

```
using namespace [идентификатор]
```

Пример:

```
#include "stdafx.h"
#include "iostream.h"
namespace spaceA
{
        int MyVal=10;
}
namespace spaceB
{
        int MyVal=20;
}
namespace spaceC
{
        int MyVal=30;
}
void Test()
{
        using namespace spaceB;
        cout << MyVal << " " << "spaceB" << endl;
}
void main()
{
 using namespace spaceA;
 cout << MyVal << " " << "spaceA" << endl;
 Test();
 cout << spaceC::MyVal << " " << "spaceC" << endl;
}
```

- Итераторы

Существует пять типов итераторов:

1. Итераторы ввода (input_iterator) поддерживают операции равенства, разыменования и инкремента: `==, !=, *i, ++i, i++, *i++`. Специальным случаем итератора ввода является istream_iterator.

2. Итераторы вывода (output_iterator) поддерживают операции разыменования, допустимые только с левой стороны присваивания, и инкремента: `++i, i++, *i=t, *i++=t`. Специальным случаем итератора вывода является ostream_iterator.

3. Однонаправленные итераторы (forward_iterator) поддерживают все операции итераторов ввода/вывода и, кроме того, позволяют без огра-

ничения применять присваивание: `==, !=, =, *i, ++i, i++, *i++`.

4. Двунаправленные итераторы (biderectional_iterator) обладают всеми свойствами forward-итераторов, а также имеют дополнительную операцию декремента (`--i, i--, *i--`), что позволяет им проходить контейнер в обоих направлениях.

5. Итераторы произвольного доступа (random_access_iterator) обладают всеми свойствами biderectional-итераторов, а также поддерживают операции сравнения и адресной арифметики, то есть непосредственный доступ по индексу: `i+=n, i+n, i-=n, i-n, i1-i2, i[n], i1<i2, i1<=i2, i1>i2, i1>=i2`.

В STL поддерживаются обратные итераторы (reverse iterators). Обратными итераторами могут быть либо двунаправленные итераторы, либо итераторы произвольного доступа, но проходящие последовательность в обратном направлении.

- Контейнеры

В STL определены два типа контейнеров: последовательности и ассоциативные.

Ключевая идея для стандартных контейнеров заключается в том, что когда это представляется разумным, они должны быть логически взаимозаменяемыми. Например, если часто требуется поиск по ключу, можно воспользоваться map (ассоциативным массивом). По умолчанию пользователь должен использовать vector; он реализован, чтобы хорошо работать для самого широкого диапазона задач. STL содержит множество обобщенных алгоритмов, которые избавляют от необходимости знать подробности отдельных контейнеров.

В табл. 10 приведены классы-контейнеры, определенные в STL.

Таблица 10

Класс- контейнер	Описание	Заголовочный файл с определнными классами
`bitset`	множество битов	`<bitset.h>`
`vector`	динамический массив	`<vector.h>`
`list`	линейный список	`<list.h>`
`deque`	двусторонняя очередь	`<deque.h>`
`stack`	стек	`<stack.h>`
`queue`	очередь	`<queue.h>`
`priority_queue`	очередь с приоритетом	`<queue.h>`

Окончание таблицы 10

Класс- контейнер	Описание	Заголовочный файл с оп-

		ределнными классами
`map`	ассоциативный список для хранения пар ключ / значение, где с каждым ключом связано одно значение	`<map.h>`
`multimap`	с каждым ключом связано два или более значений	`<map.h>`
`set`	множество	`<set.h>`
`multiset`	множество, в котором каждый элемент не обязательно уникален	`<set.h>`

В табл. 11 приведен обзор основных операций с контейнерами.

Таблица 11

Типы	
value_type	тип элемента
allocator_type	тип распределителя памяти
size_type	тип индексов, счетчика элементов и т.д
iterator	ведет себя как value_type*
reverse_iterator	просматривает контейнер в обратном порядке
reference	ведет себя как value_type&
key_type	тип ключа (только для ассоциативных контейнеров)
key_compare	тип критерия сравнения (только для ассоциативных контейнеров)
mapped_type	тип отображенного значения
Итераторы	
begin()	указывает на первый элемент
end()	указывает на элемент, следующий за последним
rbegin()	указывает на первый элемент в обратной последовательности
rend()	указывает на элемент, следующий за последним в обратной последовательности
Доступ к элементам	
front()	ссылка на первый элемент
back()	ссылка на последний элемент
operator[](i)	доступ по индексу без проверки
at(i)	доступ по индексу с проверкой
Включение элементов	
insert(p,x)	добавление x перед элементом, на который указывает p
insert(p,n,x)	добавление n копий x перед p

Окончание таблицы 11

insert(p,first,last)	добавление элементов из [first:last] перед p
push_back(x)	добавление x в конец
push_front(x)	добавление нового первого элемента (только для списков и очередей с двумя концами)
Удаление элементов	
pop_back()	удаление последнего элемента
pop_front()	удаление первого элемента (только для списков и очередей с двумя концами)
erase(p)	удаление элемента в позиции p
erase(first,last)	удаление элементов из [first:last]
clear()	удаление всех элементов
Другие операции	
size()	число элементов
empty()	контейнер пуст?
capacity()	память, выделенная под вектор (только для векторов)
reserve(n)	выделяет память для контейнера под n элементов
resize(n)	изменяет размер контейнера (только для векторов, списков и очередей с двумя концами)
swap(x)	обмен местами двух контейнеров
==, !=, <	операции сравнения
Операции присваивания	
operator=(x)	контейнеру присваиваются элементы контейнера x
assign(n,x)	присваивание контейнеру n копий элементов x (не для ассоциативных контейнеров)
assign(first,last)	присваивание элементов из диапазона [first:last]
Ассоциативные операции	
operator [](k)	доступ к элементу с ключом k
find(k)	находит элемент с ключом k
lower_bound(k)	находит первый элемент с ключом k
upper_bound(k)	находит первый элемент с ключом, большим k
equal_range(k)	находит lower_bound (нижнюю границу) и upper_bound (верхнюю границу) элементов с ключом k

- Алгоритмы

Каждый алгоритм выражается шаблоном функции или набором шаблонов функций. Таким образом, алгоритм может работать с очень разными контейнерами, содержащими значения разнообразных типов. Алгоритмы в STL реализуют большинство распространенных универсальных операций с контейнерами, такие как просмотр, сортировка, поиск, вставка и удаление элементов. Они определены в заголовочном файле `<algorithm.h>`.

В табл. 12 приведены имена некоторых наиболее часто используе-

мых функций-алгоритмов STL.

Таблица 12

Немодифицирующие операции	
for _earch()	выполняет операции для каждого элемента последовательности
find()	находит первое вхождение значения в последовательность
find_if()	находит первое соответствие предикату в последовательности
count()	подсчитывает количество вхождений значения в последовательность
count_if()	подсчитывает количество выполнений предиката в последовательности
search()	находит первое вхождение последовательности как подпоследовательности
search_n()	находит n-е вхождение значения в последовательность
Модифицирующие операции	
copy()	копирует последовательность, начиная с первого элемента
swap()	меняет местами два элемента
replace()	заменяет элементы с указанным значением
replace_if()	заменяет элементы при выполнении предиката
replace_copy()	копирует последовательность, заменяя элементы с указанным значением
replace_copy_if()	копирует последовательность, заменяя элементы при выполнении предиката
fill()	заменяет все элементы данным значением
remove()	удаляет элементы с данным значением
remove_if()	удаляет элементы при выполнении предиката
remove_copy()	копирует последовательность, удаляя элементы с указанным значением
remove_copy_if()	копирует последовательность, удаляя элементы при выполнении предиката
reverse()	меняет порядок следования элементов на обратный
random_shuffle()	перемещает элементы согласно случайному равномерному распределению
transform()	выполняет заданную операцию над каждым элементом последовательности
unique()	удаляет равные соеднии элементы
unique_copy()	копирует последовательность, удаляя равные соседние элементы
Сортировка	
sort()	сортирует последовательность с хорошей средней эф-

	фективностью

Окончание аблицы 12

partial_sort()	сортирует часть последовательности
stable_sort()	сортирует последовательность, сохраняя порядок следования равных элементов
lower_bound()	находит первое вхождение значения в отсортированной последовательности
upper_bound()	находит первый элемент, больший чем заданное значение
binary_search()	определяет, есть ли данный элемент в отсортированной последовательности
merge()	сливает две отсортированные последовательности
Работа с множествами	
includes()	проверка на вхождение
set_union()	объединение множеств
set_intersection()	пересечение множеств
set_difference()	разность множеств
Минимумы и максимумы	
min()	меньшее из двух
max()	большее из двух
min_element()	наименьшее значение в последовательности
max_element()	наибольшее значение в последовательности
Перестановки	
next_permutation()	следующая перестановка в лексикографическом порядке
pred_permutation()	предыдущая перестановка в лексикографическом порядке

Все алгоритмы отделены от деталей реализации структур данных и используют в качестве параметров типы итераторов. Поэтому они могут работать с определяемыми пользователем структурами данных, когда эти структуры данных имеют типы итераторов, удовлетворяющие предположениям в алгоритмах.

- Вектор (Vector)

Вектор (vector) в STL определен как динамический массив с доступом к его элементам по индексу:

```
template<class T,class Allocator=allocator<T>>class std::vector{…};
//где T – тип предназначенных для хранения данных.
```

`Allocator` задает распределитель памяти, который по умолчанию является стандартным. В классе vector определены следующие конструкторы:

```
explicit vector(const Allocator& a=Allocator());
explicit vector(size_type число, const T&значение= T(),
                  const Allocator&a= =Allocator());
```

```
vector(const vector<T,Allocator>&объект);
template<class InIter>vector(InIter начало, InIter конец,
                  const Allocator&a= =Allocator());
```

Пример:

```
vector<int> a;
vector<double> x(5);
vector<char> c(5,'*');
vector<int> b(a);       //b=a
```

Для любого объекта, который будет храниться в векторе, должен быть определен конструктор по умолчанию. Для класса вектор определены операторы сравнения: ==, <, <=, !=, >, >=; оператор индекса []; функции добавления: `insert(), push_back(), resize(), assign();` функции удаления: `erase(), pop_back(), resize(), clear();`итераторы доступа: `begin(), end(), rbegin(), rend().`

Манипулирование контейнером, сортировка, поиск в нем и тому подобное возможно с помощью глобальных функций файла-заголовка `<algorithm.h>`.

Пример:

```
#include<iostream.h>
#include<vector.h>
       using namespace std;
void main()
       {
       vector<int> v; //объявление пустого вектора
       int i;
       for(i=0;i<10;i++)v.push_back(i);
       //заполнить вектор последовательными цифрами, добавляя в конец
       cout<<"size="<<v.size()<<"\n";
       for(i=0;i<10;i++)cout<<v[i]<<" ";
       cout<<endl;
       for(i=0;i<10;i++)v[i]=v[i]+v[i];
       for(i=0;i<v.size();i++)cout<<v[i]<<" ";
       cout<<endl;
}
```

Пример вставки и удаления элементов:

```
#include<iostream.h>
#include<vector.h>
       using namespace std;
void main()
       {
vector<int> v(5,1);
            //объявление вектора 5 элементов, инициализация цифрой  1
int i;
       for(i=0;i<5;i++)cout<<v[i]<<" "; //вывод
       cout<<endl;
       vector<int>::iterator p=v.begin();
       p+=2;
```

```
        v.insert(p,10,9); //вставить 10 элементов со значением 9
        p=v.begin();//вывод
        while(p!=v.end())
             {cout<<*p<<" ";
             p++;
             }
        p=v.begin();//удалить вставленные элементы
        p+=2;
        v.erase(p,p+10);
        p=v.begin();//вывод
        while(p!=v.end())
             {cout<<*p<<" ";p++;}
}
```

Пример вектора, с объектами пользовательского класса:

```
#include<iostream.h>
#include<vector.h>
#include"student.h"
using namespace std;
void main()
{
        vector<STUDENT> v(3); //начальный размер вектора -3
        int i;
        v[0]=STUDENT("Иванов",45.9);
        v[1]=STUDENT("Петров",30.4);
        v[0]=STUDENT("Сидоров",55.6);
        for(i=0;i<3;i++)cout<<v[i]<<" "; //вывод
        cout<<endl;
}
```

- Функциональные объекты

Объекты-функции – это экземпляры класса, в котором определен `operator()`. В ряде случаев удобно заменить функцию на объект-функцию:

```
class less{
        public:
             bool operator()(int x,int y)
                   {return x<y;}
            };
```

Использование функциональных объектов вместе с функциональными шаблонами как делает результирующий код более эффективным. Например, если необходимо поэлементно сложить два вектора `a` и `b`, содержащие `double`, и поместить результат в `a`, можно:

```
transform(a.begin(), a.end(), b.begin(), a.begin(), plus<double>());
```

Библиотека обеспечивает базовые классы функциональных объектов для всех арифметических, логических операторов и операторов сравнения языка.

Задания для выполнения

Таблица 13

Создать три проекта демонстрацити: использования контейнерных классов для хранения встроенных типов данных; использования контейнерных классов для хранения пользовательских типов данных; использования алгоритмов STL.

<u>В проекте № 1</u> выполнить следующее:

1. Создать объект-контейнер в соответствии с вариантом задания и заполнить его данными, тип которых определяется вариантом задания. Просмотреть контейнер.

2. Изменить контейнер, удалив из него одни элементы и заменив другие.

3. Просмотреть контейнер, используя для доступа к элементам итераторы.

4. Создать второй контейнер этого же класса и заполнить его данными того же типа, что и первый контейнер.

5. Изменить первый контейнер, удалив из него n элементов после заданного и добавив затем в него все элементы из второго контейнера. Просмотреть первый и второй контейнеры.

<u>В проекте № 2</u> выполнить то же самое, но для данных пользовательского типа. В качестве пользовательского типа данных использовать пользовательский класс практикума № 6. Для вставки и удаления элементов контейнера использовать соответствующие операции, определенные в классе контейнера. Для ввода-вывода объектов пользовательского класса следует перегрузить операции >> и <<.

<u>В проекте № 3</u> выполнить следующее:

1. Создать контейнер, содержащий объекты пользовательского типа. Тип контейнера выбирается в соответствии с вариантом задания. Отсортировать его по убыванию элементов. Если алгоритмы не поддерживают используемые контейнеры, написать свой алгоритм. Просмотреть контейнер.

2. Используя подходящий алгоритм, найти в контейнере элемент, удовлетворяющий заданному условию.

3. Переместить элементы, удовлетворяющие заданному условию в другой (предварительно пустой) контейнер. Тип второго контейнера определяется вариантом задания (при перемещении элементов ассоциативного контейнера в неассоциативный перемещаются только данные, ключи не перемещаются и наоборот, при перемещении элементов неассоциативного контейнера в ассоциативный должен быть сформирован ключ). Просмотреть второй контейнер.

4. Отсортировать первый и второй контейнеры по возрастанию элементов. Просмотреть их.

5. Получить третий контейнер путем слияния первых двух. Просмотреть третий контейнер.

6. Подсчитать, сколько элементов, удовлетворяющих заданному условию, содержит третий контейнер. Определить, есть ли в третьем контейнере элемент, удовлетворяющий заданному условию.

Окончание таблицы 13

№	Первый контейнер	Второй контейнер	Встроенный тип данных
1	stack	set	char
2	vector	list	int
3	list	deque	long
4	deque	stack	float
5	stack	queue	double
6	queue	vector	char
7	vector	stack	string
8	map	list	long
9	multimap	deque	float
10	set	stack	int
11	multiset	queue	char
12	vector	map	double
13	list	set	int
14	deque	multiset	long
15	vector	set	long
16	queue	stack	int
17	map	vector	char
18	set	vector	double
19	list	multiset	string
20	stack	set	float

Контрольные вопросы

1. Перечислите основные функции итераторов STL.

2. Верно ли, что итератор всегда может сдвигаться как в прямом, так и в обратном направлении по контейнеру?

3. Для чего используется контейнер STL?

4. Перечислите последовательные контейнеры STL.

5. Верно ли, что алгоритмы могут использоваться только с контейнерами STL?

Практикум №15. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ

Краткие теоретические сведения

В языке C++ реализован специальный механизм для сообщения об ошибках - механизм исключительных ситуаций, который может применяться и для обработки плановых ошибок.

Исключительная ситуация возникает при выполнении оператора `throw`. В качестве аргумента `throw` задается любое значение. Это может быть значение одного из встроенных типов (число, строка символов и т.п.) или объект любого определенного в программе класса.

При возникновении исключительной ситуации выполнение текущей функции или метода немедленно прекращается, созданные к этому моменту автоматические переменные уничтожаются, и управление передается в точку, откуда была вызвана текущая функция или метод. В точке возврата создается та же самая исключительная ситуация, прекращается выполнение текущей функции или метода, уничтожаются автоматические переменные, и управление передается в точку, откуда была вызвана эта функция или метод. Происходит своего рода откат всех вызовов до тех пор, пока не завершится функция `main` и, соответственно, вся программа.

Пусть, из `main` была вызвана функция `foo`, которая вызвала метод `Open`, а он в свою очередь возбудил исключительную ситуацию:

```
class Database
{
public:
     void Open(const char* serverName);
};
void
Database::Open(const char* serverName)
{
     if (connect(serverName) == false)
          throw 2;
}
foo()
{
     Database database;
     database.Open("db-server");
     String y;
     . . .
}
main()
{
     String x;
     foo();
}
```

В этом случае управление вернется в функцию `foo`, будет вызван деструктор объекта `database`, управление вернется в `main`, где будет вызван деструктор объекта `x`, и выполнение программы завершится. Таким образом, исключительные ситуации позволяют аварийно завершать программы с некоторыми возможностями очистки переменных. В таком виде оператор `throw` используется для действительно исключительных ситуаций, которые практически никак не обрабатываются.

- Обработка исключительных ситуаций

В программе можно объявить блок, в котором будут отслеживаться исключительные ситуации с помощью операторов `try` и `catch`:

```
try   {
      . . .
      } catch (тип_исключительной_операции) {
      . . .
}
```

Если внутри блока `try` возникла исключительная ситуация, то она первым делом передается в оператор `catch`. Тип исключительной ситуации - это тип аргумента `throw`. Если тип исключительной ситуации совместим с типом аргумента `catch`, выполняется блок `catch`. Тип аргумента `catch` совместим, если он либо совпадает с типом ситуации, либо является одним из ее базовых типов. Если тип несовместим, то происходит описанный выше откат вызовов, до тех пор, пока либо не завершится программа, либо не встретится блок `catch` с подходящим типом аргумента.

Например:

```
foo()
{
     Database database;
     int attempCount = 0;
again:
     try {
          database.Open("dbserver");
          } catch (int& ex) {
          cerr << "Ошибка соединения номер " << x << endl;
          if (++attemptCount < 5)
                goto again;
          throw;
     }
     String y;
      . . .
}
```

Ссылка на аргумент `throw` передается в блок `catch`. Этот блок гасит исключительную ситуацию. Во время обработки в блоке `catch` можно создать или ту же самую исключительную ситуацию с помощью операто-

ра `throw` без аргументов, или другую, или же не создавать никакой. В последнем случае исключительная ситуация считается погашенной, и выполнение программы продолжается после блока `catch`.

С одним блоком `try` может быть связано несколько блоков `catch` с разными аргументами. В этом случае исключительная ситуация последовательно "примеряется" к каждому `catch` до тех пор, пока аргумент не окажется совместимым. Этот блок и выполняется. Специальный вид `catch` совместим с любым типом исключительной ситуации. Однако, в него нельзя передать аргумент.

- Примеры обработки исключительных ситуаций

Механизм исключительных ситуаций предоставляет гибкие возможности для обработки ошибок. Рассмотрим некоторые приемы обработки исключительных ситуаций.

Прежде всего, желательно определить для них специальный класс. Простейшим вариантом является класс, который может хранить код ошибки:

```
class Exception
{
public:
     enum ErrorCode {
          NO_MEMORY,
          DATABASE_ERROR,
          INTERNAL_ERROR,
          ILLEGAL_VALUE
     };
     Exception(ErrorCode errorKind,
               const String& errMessage);
     ErrorCode GetErrorKind(void) const { return kind; };
     const String& GetErrorMessage(void) const
          { return msg;};
private:
     ErrorCode kind;
     String msg;
};
```

Создание исключительной ситуации будет выглядеть следующим образом:

```
if (connect(serverName) == false)
     throw Exception(Exception::DATABASE_ERROR, serverName);
```

а проверка на исключительную ситуацию:

```
try {
     . . .
    } catch (Exception& e) {
     cerr << "Произошла ошибка " << e.GetErrorKind()
          << " Дополнительная информация: "
```

```
          << e.GetErrorMessage();
}
```

Преимущества класса перед просто целым числом состоят, во-первых, в том, что передается дополнительная информация и, во-вторых, в операторах `catch` можно реагировать только на ошибки определенного вида. Это особенно существенно при сопряжении нескольких различных программ и библиотек - каждый набор классов отвечает только за собственные ошибки.

В данном случае код ошибки записывается в объекте типа `Exception`. Если в одном блоке `catch` ожидается несколько разных исключительных ситуаций, и для них необходима разная обработка, то в программе придется анализировать код ошибки с помощью операторов `if` или `switch`:

```
try {
     . . .
} catch (Exception& e) {
     cerr << "Произошла ошибка " << e.GetErrorKind()
           << " Дополнительная информация: "
          << e.GetErrorMessage();
     if (e.GetErrorKind() == Exception::NO_MEMORY
          || e.GetErrorKind() == Exception:: INTERNAL_ERROR)
          throw;
     else if (e.GetErrorKind() == Exception::DATABASE_ERROR)
          return TRY_AGAIN;
     else if (e.GetErrorKind() == Exception::ILLEGAL_VALUE)
          return NEXT_VALUE;
}
```

Другим методом разделения различных исключительных ситуаций является создание иерархии классов - по классу на каждый тип исключительной ситуации.

Чтобы облегчить обработку ошибок и сделать запись о них более наглядной, описания методов и функций можно дополнить информацией, какого типа исключительные ситуации они могут создавать:

```
class Database
{
public:
     Open(const char* serverName) throw ConnectDbException;
};
```

Такое описание говорит о том, что метод `Open` класса `Database` может создать исключительную ситуацию типа `ConnectDbException`. Соответственно, при использовании этого метода желательно предусмотреть обработку возможной исключительной ситуации.

Следует отметить, что при возникновении исключительной ситуации остаток функции или метода не выполняется. Более того, при обработке ее не всегда известно, где именно возникла исключительная ситуация. Поэтому прежде чем выполнить оператор `throw`, освободите ресурсы, зарезервированные в текущей функции. Например, если какой-либо объект был создан с помощью **new**, необходимо явно вызвать для него `delete`.

Желательно избегать использования исключительных ситуаций в деструкторах. Деструктор может быть вызван в результате уже возникшей исключительной ситуации при откате вызовов функций и методов. Повторная исключительная ситуация не обрабатывается и завершает выполнение программы. Если исключительная ситуация возникла в конструкторе объекта, считается, что объект сформирован не полностью, и деструктор для него вызван не будет

Задания для выполнения

Написать программу, в которой перехватываются исключения типа int, char *. Сгенерировать исключительную ситуацию. Создать тип ошибка - error. Добавить к программе перехват исключительной ситуации созданного Вами типа. Создать тип ошибок памяти и тип ошибок с файлами, наследуемые от error. Добавить к программе перехват исключительной ситуации ваших типов.

Контрольные вопросы

1. Что такое исключение? Приведите пример.
2. Каково назначение блока `catch`?
3. Правильным ли будет перечисление обработчиков исключений в порядке, который предотвращает их вызов.
4. Сколько блоков `catch` с разными аргументами может быть связано с одним блоком `try`?
5. Почему желательно избегать использования исключительных ситуаций в деструкторах?

Практикум № 16-17. ОСНОВЫ ПРОГРАММИРОВАНИЯ ПОД WINDOWS. СОЗДАНИЕ WIN32 API

Краткие теоретические сведения

Архитектура Windows-программ основана на принципе сообщений, а все программы содержат некоторые общие компоненты.

- Функция окна

Все Windows-программы должны содержать специальную функцию, которая не используется в программе, но вызывается операционной системой. Эту функцию обычно называют функцией окна, или процедурой окна. Она вызывается Windows, когда системе необходимо передать сообщение в программу. Именно через нее осуществляется взаимодействие между программой и системой. Функция окна передает сообщение в своих аргументах. Согласно терминологии Windows, функции, вызываемые системой, называются функциями обратного вызова. Таким образом, функция окна является функцией обратного вызова. Помимо принятия сообщения от Windows, функция окна должна вызывать выполнение действия, указанного в сообщении. В большинстве Windows-программ задача создания функции окна лежит на разработчике.

- Цикл сообщений

Все приложения Windows должны организовать так называемый цикл сообщений (обычно внутри функции `WinMain()`). В этом цикле каждое необработанное сообщение должно быть извлечено из очереди сообщений данного приложения и передано назад в Windows, которая затем вызывает функцию окна программы с данным сообщением в качестве аргумента. В традиционных Windows-программах необходимо самостоятельно создавать и активизировать такой цикл.

- Класс окна

Каждое окно в Windows-приложении характеризуется определенными атрибутами, называемыми классом окна (понятие «класс» означает стиль или тип). В традиционной программе класс окна должен быть определен и зарегистрирован прежде, чем будет создано окно. При регистрации необходимо сообщить Windows, какой вид должно иметь окно и какую функцию оно выполняет. В то же время регистрация класса окна еще не означает создание самого окна. Для этого требуется выполнить дополнительные действия.

Существует три основных типа окон - перекрывающиеся, всплывающие и дочерние, из которых можно создавать множество самых раз-

нообразных объектов, комбинируя биты стиля. Перекрывающиеся (overlapped window) - основной, наиболее универсальный тип окон. Для их создания используется стиль `WS_OVERLAPPEDWINDOW`. Вспомогательный или всплывающие окна (popup window) Используется стиль `WS_POPUP`. Используются для диалоговых окон и окон сообщений. Дочерние окна (child window) Они связаны некоторыми характеристиками с главным окном, из которого они были созданы.

- Типы данных в Windows

В Windows-программах вообще не слишком широко применяются стандартные типы данных такие как `int` или `char*`. Вместо них используются типы данных, определенные в различных библиотечных (header) файлах. Наиболее часто используемыми типами являются `HANDLE` (32-разрядное целое, используемое в качестве дескриптора), `HWND` (32-разрядное целое – дескриптор окна), `BYTE` (8-разрядное беззнаковое символьное значение), `WORD` (16-разрядное беззнаковое короткое целое), `DWORD` (беззнаковое длинное целое), `UNIT` (беззнаковое 32-разрядное целое), `LONG` (эквивалентен типу long), `BOOL` (целое и используется, когда значение может быть либо истинным, либо ложным), `LPSTR` (указатель на строку) и `LPCSTR` (константный (const) указатель на строку).

- Программа для Windows

Минимальная программа для Windows состоит из двух функций: функции `WinMain` и функции окна или оконной процеуры. Функция `WinMain` состоит их трех функциональных частей: регистрация класса окна, создания главного окна приложения и цикла обработки сообщений. На некотором псевдоязыке программу для Windows можно записать следующим образом:

```
WinMain(список аргументов)
{
        Создание класса окна
        Создание экземпляра класса окна
        Пока не произошло необходимое для выхода событие
             Выбрать из очереди сообщений очередное сообщение
             Передать сообщение оконной функции
        Возврат из программы
}
WindowsFunction(список аргументов)
{
        Обработать полученное сообщение
        Возврат
}
```

- Функция WinMain

Рассмотрим простейшее приложение Win32:

```
int WINAPI WinMain(HINSTANCE         hInstance,
                   HINSTANCE         hPrevInstance,
                   LPSTR             lpCmdLine,
                   int               nCmdShow)
{  return 0; }
```

Это приложение ничего не делает и сразу же прекращает свою работу, возвращая управление ОС с кодом возврата 0.

`WINAPI` перед телом функции `WinMain` указывает компилятору на необходимость сгенерировать перед выполнением этой функции специальный пролог и эпилог необходимый для функции, в которой запускается и завершается программа. Если это слово будет отсутствовать, то программа будет сгенерированна неправильно.

Рассмотрим параметры, которые получает приложение от ОС в функции `WinMain`:

`lpCmdLine` - указатель на командную строку;

`nCmdShow` - код режима начального отображения главного окна приложения;

`hInstance` - дискриптор, ассоциируемый с текущим приложением, некоторые функции API могут потребовать его в качестве параметра. В основном он необходим при работе с ресурсами приложений, организации многозадачности и при создании оконных обьектов;

`hPrevInstance` - параметр для совместимости с предыдущими версиями Win16; в Win32 не имеет никакого значения и всегда равен `NULL`.

Главное окно программы первое появляется и последним исчезает при работе с программой. Другие окна, которые создаются главным окном, взаимодействуют с ним и им же уничтожаются. Прежде чем его создать, необходимо зарегистрировать его класс, при этом имя класса главного окна должно быть уникальным, чтобы не возникало конфликта с классами окон других приложений.

При регистрации класса окна в нем задаются наиболее общие свойства тех оконных объектов, которые будут созданы на основе данного класса, и сведения о которые необходимы для системы.

- Регистрация класса окна

Для регистрации класса окна рекомендуется использовать функцию `RegisterClassEx` или `RegisterClass`.

```
     ATOM RegisterClassEx (
      const WNDCLASSEX *lpwcx;    );
//указатель на структуру, содержащую данные об регистрируемом классе
```

Функция возвращает уникальный целочисленный идентификатор (ATOM), которое уникально для каждого зарегистрированного класса окна или 0 - в случае неудачи. Параметр функции - указатель на структуру WNDCLASSEX или WNDCLASS, в которой содержатся все необходимые данные об классе окна. Прототип этой структуры:

```
typedef struct _WNDCLASSEX {
 UINT cbSize;           //размер структуры в байтах
 UINT style;            //стиль класса
 WNDPROC lpfnWndProc;   //адрес функции обратного вызова для для
                        //приема сообщений предназначенных для
                        // данного класса окна
 int cbClsExtra;        //число байт для хранения данных для   класса
 int cbWndExtra;        //число байт для хранения данных при создании
                        // каждого оконного объекта класса
 HANDLE hInstance;      // дескриптор программного модуля
                        //который регистрирует класс
 HICON hIcon;           // дескриптор большой иконки
 HCURSOR hCursor;       // дескриптор курсора
 HBRUSH hbrBackground;  //цвет кисти фона
 LPCTSTR lpszMenuName;  //имя меню для этого класса окна
 LPCTSTR lpszClassName; //имя класса окна
 HICON hIconSm;         // дескриптор маленькой иконки е
 } WNDCLASSEX;
```

Загрузить необходимую иконку можно с помощью функций LoadIcon (загружает только иконки размером 32*32) или LoadImage:

```
HICON LoadIcon(
  HINSTANCE hInstance,//Идентификатор программного модуля, из
                      //ресурсов которого необходимо загрузить иконку
                      //для загрузки из ресурсов ОС укажите  NULL
  LPCTSTR lpIconName  //имя иконки; для встроенных иконок этот
                      //параметр может быть например IDI_APPLICATION
);
```

Для загрузки курсора используется функция LoadCursor:

```
HCURSOR LoadCursor(
 HINSTANCE hInstance,// Идентификатор программного модуля, из ресурсов
                     // которого необходимо загрузить рисунок курсора,
                     // для загрузки из ресурсов ОС укажите NULL
 LPCTSTR lpCursorName // имя курсора; для задания идентификатора о
                      //дного из встроенных курсоров в виде флажка
                      // Windows укажем этот параметр как IDC_ARROW
);
```

Всего в системе есть около двадцати предопределенных цветов, доступных по своим константным номерам. При использование какого-либо из них обязательно прибавлять к номеру 1 (т.к. первое значение для них равно 0) и преобразовывать к типу HBRUSH.

Стили окна - это битовые флаги, которые могут комбинироваться с помощью логической операции ИЛИ, всего их более 10:

`CS_DBLCLKS` - если нажимать клавиши мыши достаточно быстро, то система будет отправлять сообщение для программы о двойном щелчке; если его не установить, то как бы быстро не щелкали на клавиши мышки, сообщения о двойном щелчке не появятся;

`CS_HREDRAW` - если пользователь изменил ширину данного окна, то посылается сообщение об его перерисовки;

`CS_VREDRAW` - если пользователь изменил высоту данного окна, то посылается сообщение об его перерисовки.

Например, можно установиить поле `style` равным: `CS_HREDRAW| CS_VREDRAW | CS_DBLCLKS`.

Теперь регистрируется опредленный класс окна с помощью функции `RegisterClassEx`. Функции вовращает `ATOM`, или шестнадцатиразрядное целое число по которому система будет различать класс окна. Его не нужно запоминать, но стоит проанализировать на предмет неравенства этого значения нулю (если возвращенное число равно нулю, то зарегистрировать класс окна не удалось).

Каждый зарегистрировнный класс окна необходимо обеспечить своей уникальной функцией обратного вызова и своим уникальным именем.

Пример регистрации класса окна:

```
ATOM MyRegisterClass(HINSTANCE hInstanse)
{WNDCLASSEX wcex;
 wcex.cbSize            =sizeof(WNDCLASSEX);
 wcex.style             =CS_HREDRAW|CSVREDRAW;
 wcex.lpfnWndProc       =(WNDPROC)WndProc;
 wcex.cbClsExtra        =0;
 wcex.cbWndExtra        =0;
 wcex.hInstance         =hInstance;
 wcex.HIcon             =LoadIcon(hInstance,IDI_APPLICATION);
 wcex.hCursor           =LoadCursor(NULL,IDCARROR);
 wcex.hbrBackground     =(HBRUSH)(COLOR_WINDOW+1);
 wcex.lpszMenuName      =NULL;
 wcex.lpClassName       =szWindowClass;
 wcex.hIconSm           =LoadIcon(wcex.hInstance,IDI_APPLICATION);
 ATOM atom=::RegisterClassEx(&wcex);
if(atom) return atom;
else   //если данная версия Windows не поддерживает расширенный
       //класс окна
{WNDCLASS wc;
 wc.style               =CS_HREDRAW|CSVREDRAW;
 wc.lpfnWndProc         =(WNDPROC)WndProc;
 wc.cbClsExtra          =0;
 wc.cbWndExtra          =0;
```

```
 wc.hInstance           =hInstance;
 wc.HIcon               =LoadIcon(hInstance,IDI_APPLICATION);
 wc.hCursor             =LoadCursor(NULL,IDCARROR);
 wc.hbrBackground       =(HBRUSH)(COLOR_WINDOW+1);
 wc.lpszMenuName        =NULL;
 wcex.lpClassName       =szWindowClass;
 return ::RegisterClass(&wc);
}  }
```

- Сообщения и функция обратного вызова

Если создается только один объект данного класса окон, то можно сразу приступать к обработке сообщения. Приложение может создать столько окон одинакового оконного класса, сколько нужно, но если приложение регистрирует внутри себя какой-то производный класс окна (не главное окно которое может быть создано только в одном экземпляре) и создает несколько объектов этого класса, то оно должно учитывать от какого оконного объекта пришло сообщение.

При регистрации оконного класса, для ОС передается указатель на функции, удовлетворяющий следующему прототипу (имя функции может быть любое):

```
LRESULT CALLBACK WndProc(
  HWND hwnd,      //дескриптор оконного объекта
  UINT uMsg,      //код сообщения
  WPARAM wParam,  //первый параметр сообщения
  LPARAM lParam   //второй параметр сообщения
);
```

Если создать объект зарегистрированного оконного класса, то все сообщения для объектов этого класса при передаче ему сообщений от объектов других классов, системы или пользователя, или от объекта того же класса, направятся в эту функцию.

Слово `CALLBACK` в объявлении функции говорит компилятору о необходимости генерации специального кода для входа и выхода из функции обратного вызова.

Тип значения, возвращаемый функции класса окна `LRESULT`, определен как 32-битное значащее целое, но в зависимости от сообщения в `uMsg`, его, возможно, потребуется преобразовать к указателю на некоторое значения или к другому целому типу. Это значение определяет результат обработки сообщения, поэтому оно всегда должно передаваться ОС при завершение работы функции.

Параметры функции:

`HWND hwnd` - декриптор объекта окна, для которого предназначено сообщение, если создано несколько объектов данного оконного класса, то

по нему обработчик выбирает нужный; для главного окна приложения, которое существует в одном экземпляре его можно проигнорировать;

`UINT uMsg` - код сообщения, беззнаковое целое;

`WPARAM wParam` и `LPARAM lParam` - 32-битное целые, в которых передаются параметры сообщения, их возможно потребуется преобразовать к указателям на некоторое значения или к другому целому типу.

Всего в Windows существует более 900 сообщений, которые может получить любая функция класса окна, и все они должны быть обработаны. Однако те сообщения, обрабатывать которые внутри функции класса окна нет необходимости, могут быть переданы ОС для их обработки по умолчанию. Для этого необходимо передать те параметры, которые были переданы функции обратного вызова, в специальную системную функцию обработки сообщения по умолчанию называемую `DefWindowProc`:

```
LRESULT DefWindowProc(
                  HWND hWnd,      // декриптор окна
                  UINT Msg,       // сообщение
                  WPARAM wParam,  // первый параметр сообщения
                  LPARAM lParam   // второй параметр сообщения
                  );
```

Эта функция принимает и обрабатывает по умолчанию любое сообщение, обрабатывать которое функция класса окна не будет; в любом приложение для любого класса окна, все сообщения, переданные этой функции будут обрабатываться одинаково.

Теперь можно написать простейшую функцию обработки сообщений:

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT message,
                         WPARAM wParam,
                         LPARAM lParam)
{ return DefWindowProc(hwnd,message,wParam,lParam);}
```

При использования этой функции для обработки сообщений для некоторого класса окна, получается закрашенный цветом кисти фона для данного класса окно, чьи размеры и заголовок можно изменять и не реагирующее не на какие ваши действия, кроме приказа на удаления. Поэтому надо добавить:

```
Switch (message)
{       case WM_DESTROY:
             PostOuitMessage(0);
             Break;
….}
```

Такое окно мало для чего пригодно, поэтому необходимо переопределить обработку сообщений по умолчанию.

- Создание главного окна приложения

После регистрации класса окна, приложение должно создать вначале главное окно приложения, затем возможно и еще несколько окон связанных с главным окном. Создание любого объекта оконного типа, выполняется функцией `CreateWindow`:

```
HWND CreateWindow(
LPCTSTR lpClassName,  //указатель на строку с именем класса окна
LPCTSTR lpWindowName, //указатель на строку с заголовком окна
DWORD dwStyle,        //стиль окна
int x,                //x-координата левого верхнего угла окна
int y,                //y-координата левого верхнего угла окна
int nWidth,           //ширина окна
int nHeight,          //высота окна
HWND hWndParent,      //декриптор родительского окна
HMENU hMenu,          //если есть у окна меню то его handle
HANDLE hInstance,     //дескриптор того программного модуля,
                      //который создает окно
LPVOID lpParam        //указатель на дополнительные параметры
);
```

Для главного окна приложения и для всплывающих окон координаты начала окна отсчитываются от левого верхнего угла экрана, а для дочерних окон от левого верхнего угла угла родительского окна.

Для главного окна приложения созданного со стилем `WS_OVERLAPPED` параметр `hWndParent` должен быть равен `NULL`, для дочернего окна должен быть равен дескриптору какого-нибудь из окон уже созданных данным приложением, для всплывающих окон или `NULL` или дескриптору одного из уже определенных окон.

Младшее слово в параметре стиля `dwStyle` специфицирует те необходимые свойства в поведение создаваемого объекта для окон данного класса, которые нужно знать операционной системе, а старшее слово в этом двойном слове специфицируют те свойства в поведение оконного объекта, которые необходимо знать функции обработчику сообщений для окон заданного класса. Для главного окна приложения старшее слово обычно не используется, поэтому рассмотрим те стили окна которые есть в младшем слове:

`WS_OVERLAPPED` - создать перекрывающееся окно имеющее рамку и заголовок, любое главное окно приложения должно иметь этот стиль;

`WS_POPUP` - создает всплывающее окно, то есть способное появляться в любой части экрана;

`WS_CHILD` - создает дочернее окно;

WS_CAPTIO N- создает окно, которое имеет заголовок, любое перекрывающееся окно создается с таким стилем автоматически;

WS_BORDER - создает окно, которое имеет толстую рамку вокруг окна, любое перекрывающееся окно создается с таким стилем автоматически;

WS_SIZEBOX - окно может изменять размер, если пользователь будет двигать рамку окна;

WS_SYSMENU - окно имеет системное меню;

WS_MAXIMIZEBOX и WS_MINIMIZEBOX - окно имеет кнопки увеличения размера до максимального и уменьшения размера до минимального соответственно;

WS_VISIBLE - после создание окно оно сразу появляется на экране и окну приходит сообщение на перерисовку, лучше его использовать для дочерних и всплывающих окон, для главного окна приложения стоит использовать другой метод появления на экране;

WS_OVERLAPPEDWINDOW - рекомендуемый стиль перекрывающегося окна для главного окна приложения, определен как битовая комбинация следующих стилей: WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBO, WS_MAXIMIZEBOX.

Для создания главного окна приложения можно использовать следующую функцию:

```
BOOL InitInstance(HINSTANCE hInstance,int nCmdShow)
{
HWND hWnd;
hInst=hInstance;
hWnd=CreateWindow(szWindowClass,szTitle,WS_OVERAPPENDWINDOW,
                                        CW_USERDEFAULT,0,
                                        CW_USERDEFAULT,0,
                                        NULL,NULL,hInstance,NULL);
if(!hWnd)
return FALSE;
ShowWindow(hWnd,nCmdShow) ;
UpdateWindow(hWnd);
return TRUE;
}
```

После возврата из функции CreateWindow система записывает в свою внутреннюю базу данных информацию, необходиую для сопровождения данного окна. Но при этом окно не появляется. Требуется вызов: ShowWindow(hWnd,nCmdShow) ; UpdateWindow(hWnd).

- Цикл обработки сообщений

Простейший цикл обработки сообщений выглядит следующим образом:

```
while(GetMessage(&msg,NULL,0,0))
```

```
        {     TranslateMessage(&msg);
              DispatchMessage(&msg)
        }
```

Из очереди приложения сообщение выбирается функцией `GetMessage()`. Первый ее параметр(`&msg`) - указатель на структуру типа `MSG`, т.е. на сообщение. Второй параметр – дескриптор окна, созданного программой. Сообщение, адресованное только этому окну будет выбираться функцией `GetMessage` и передаваться оконной функции. Третий и четвертый параметры позволяют передавать оконной функции не все сообщения, а только те, номера которых попадают в определенный интервал.

Функция `TranslateMessage()` преобразует некоторые сообщения в более удобный для обработки вид. Функция `DispatchMessage()` передает сообщение оконной функции.

Завершение цикла обработки сообщений происходит при выборке из очереди сообщения `WM_DESTROY`, в ответ на которое функция `GetMessage()` возвращает нулевое значение.

- Пример функции WinMain()

```
int WINAPI WinMain(HINSTANCE      hInstance,
                   HINSTANCE      hPrevInstance,
                   LPSTR          lpCmdLine,
                   int            nCmdShow)
                  {
MSG msg;
MyRegisterClass(hInstanse);
if(!InitInstance(hInstance,nCmdShow)) return FALSE;
while(GetMessage(&msg,NULL,0,0))
       {
       TranslateMessage(&msg);
       DispatchMessage(&msg)
       }
return msg.wParam; }
```

- Пример проекта Win32 API

```
#include <windows.h>
LONG WINAPI WndProc(HWND, UINT, WPARAM,LPARAM);
int WINAPI WinMain(     HINSTANCE hInstance,
                        HINSTANCE hPrevInstance,
                        LPSTR lpCmdLine,
                        int nCmdShow)
{   HWND hwnd;
    MSG msg;
    WNDCLASS w;
    memset(&w,0,sizeof(WNDCLASS));
    w.style = CS_HREDRAW|CS_VREDRAW;
    w.lpfnWndProc = WndProc;
    w.hInstance = hInstance;
```

```
    w.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    w.lpszClassName = "My Class";
    RegisterClass(&w);
hwnd    =    CreateWindow("My    Class",    "PRESS    DOUBLE    CLICK!!!",
WS_OVERLAPPEDWINDOW, 300, 200, 400, 280, NULL, NULL, hInstance, NULL);
    ShowWindow(hwnd,nCmdShow);
    UpdateWindow(hwnd);
    while(GetMessage(&msg,NULL,0,0))
    {
       TranslateMessage(&msg);
       DispatchMessage(&msg);
    }
    return msg.wParam;
}
LONG WINAPI WndProc(    HWND hwnd,
                        UINT Message,
                        WPARAM wparam,
                        LPARAM lparam)
{
   switch (Message)
   { case WM_DESTROY:
           PostQuitMessage(0);
      break;
         case WM_LBUTTONUP:
                  SetWindowText(hwnd, " NEW WINDOW-FullScreen");
          break;
            case WM_CLOSE:
            if(IDOK==MessageBox(hwnd, "Exit programm?", "Close",
                        MB_OKCANCEL|MB_ICONQUESTION|MB_DEFBUTTON2))
            SendMessage(hwnd, WM_DESTROY, NULL, NULL);
            break;
      default:  return DefWindowProc(hwnd, Message, wparam, lparam);
   }
   return 0;
}
```

Задания для выполнения

Таблица 14

Создать пустой проект типа Win32 Application. Создать главное окно приложения определенного размера с заголовком, цвета и стиля. Написать обработчики сообщений в соответсвии с вариантом. При закрытии приложения (WM_CLOSE) сделать вывод окна сообщения MessageBox с двумя кнопками - OK и Cancel. Добавить предупреждающий значок и активизировать вторую кнопку для четных вариантов и первую для нечетных.

Окончание таблицы 14

1	реакция на щелчок правой кнопки мыши - меняется заголовок
2	реакция на щелчок левой кнопки мыши - меняется заголовок
3	реакция на двойной щелчок левой кнопки мыши - меняется заголовок

4	реакция на ввод смвола - меняется позиция окна
5	реакция на щелчок правой кнопки мыши - меняется размер окна
6	реакция на перемещение мыши - меняется размер и позиция окна
7	реакция на двойной щелчок правой кнопки мыши - меняется размер окна и заголовок
8	реакция на двойной щелчок левой кнопки мыши - меняется размер и позиция окна
9	реакция на щелчок левой кнопки мыши - меняется размер и позиция окна, заголовок окна
10	реакция на ввод смвола - меняется заголовок
11	реакция на двойной щелчок правой кнопки мыши - меняется размер и позиция окна
12	реакция на ввод смвола - меняется размер, позиция окна, заголовок окна
13	реакция на щелчок левой или правой кнопки мыши - меняется позиция окна
14	реакция на двойной щелчок левой или правой кнопки мыши - меняется позиция окна
15	реакция на перемещение мыши - меняется заголовок
16	реакция на перемещение мыши - меняется позиция окна
17	реакция на ввод смвола - меняется заголовок окна
18	реакция на двойной щелчок правой кнопки мыши - меняется размер и позиция окна
19	реакция на двойной щелчок правой кнопки мыши - меняется заголовок окна
20	реакция на перемещение мыши – закрытие окна

Контрольные вопросы

1. Что такое класс окна?
2. Определите состав функции `WinMain`?
3. Каково назначение функции `DispatchMessage()` и где она употребляется?
4. Что означает строка `w.style = CS_HREDRAW|CS_VREDRAW;`?
5. Что такое функция обратного вызова, приведиете пример?

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ.....3
Практикум №1. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ.....8
Практикум №2. РЕКУРСИЯ.....17
Практикум №3-4. БИНАРНЫЕ ДЕРЕВЬЯ 26
Практикум №5. ОБЪЕКТ И КЛАСС 36
Практикум № 6. НАСЛЕДОВАНИЕ. ВЗАИМОДЕЙСТВИЕ ОБЪЕКТОВ 51
Практикум №7-8. ПОЛИМОРФИЗМ И ВИРТУАЛЬНЫЕ ФУНКЦИИ60
Практикум №9. ПЕРЕГРУЗКА ОПЕРАЦИЙ 69
Практикум № 10-11. ШАБЛОНЫ.....75
Практикум №12. ПОТОКОВЫЕ КЛАССЫ.....79
Практикум №13-14. СТАНДАРТНАЯ БИБЛИОТЕКА ШАБЛОНОВ (STL). АЛГОРИТМЫ.....86
Практикум №15. ОБРАБОТКА ИСКЛЮЧИТЕЛЬНЫХ СИТУАЦИЙ97
Практикум № 16-17. ОСНОВЫ ПРОГРАММИРОВАНИЯ ПОД WINDOWS. СОЗДАНИЕ WIN32 API 102

Учебное издание

Пацей Наталья Владимировна
Дорожкина Наталья Николаевна

КОНСТРУИРОВАНИЕ ПРОГАММ И ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Практикум

В 2-х частях

Часть 2

Редактор

Подписано в печать 02.04.2006. Формат 60x84 1/16
Бумага офсетная. Гарнитура Таймс. Печать офсетная.
Усл. печ. л. 6,8. Уч.-изд. л. 7,0.
Тираж 70 экз. Заказ

Учреждение образования «Белорусский государственный технологический университет». 220050. Минск, Свердлова, 13а.
ЛИ № 02330/0133255 от 30.04.2004.

Отпечатано в лаборатории полиграфии учреждения образования «Белорусский государственный технологический университет». 220050. Минск, Свердлова, 13.
ЛП № 02330/0056739 от 22.01.2004.