

# ГЛАВА 15. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

## 15.1. Понятие о самоссыльчных структурах

Динамические структуры данных могут создаваться, изменять свой размер и уничтожаться в процессе выполнения программы.

Рассмотрим следующие наиболее часто используемые динамические структуры:

- очереди;
- стеки;
- двусвязные списки;
- двоичные (бинарные) деревья.

Как и обычные структуры, динамические структуры состоят из полей, или членов структур. Но, кроме информационных полей, они обязательно содержат поля-указатели на свой собственный тип структуры. Поэтому динамические структуры часто называются *самоссыльчными* структурами.

Самоссыльчную структуру можно объявить следующим образом:

```
struct node { int info; // здесь может находиться любое информационное
                     // поле, и их может быть больше чем одно
    struct node *next; //указатель на структуру типа node
}
```

Структура объявленного типа имеет информационное поле целого типа и указатель на такую же структуру, с помощью которой структуры объединяются в списки. Списки очень удобны для хранения различной информации.

Следует обратить внимание на то, что указатель указывает на структуру (иногда называемую *узлом*) целиком, а не на отдельные компоненты, находящиеся внутри ее.

## 15.2. Формирование очереди

Для создания списка нужно объявить структуру аналогично структуре node:

```
struct node { int info;
    struct node *next;
}

```

и указатель на объявленный структурный шаблон:

```
typedef node *NodePtr; //указатель на тип node

```

После этого объявляют указатель на начало списка, который иногда называют заголовком списка. Назовем его head и объявим следующим образом:

```
NodePtr head = NULL;
```

Нулевой указатель на начало списка является признаком того, что список пустой. Для начала формирования списка, т. е. для размещения первого элемента в списке, следует выделить память под этот элемент, предварительно убедившись, что список пустой. Это достигается путем выполнения операторов

```
if (head == NULL)
{ head = new node;
  head->info = 1;           // или какому-то другому значению
  head->next = NULL;
}
```

Оператор *p->info = 1;* эквивалентен оператору *(\*p).info = 1;*. Первый из них называется *операцией косвенного выбора* и объединяет в себе действия, связанные с разыменованием и выбором поля структуры. Второй оператор осуществляет прямой выбор. Не следует забывать круглые скобки вокруг *\*p*, так как приоритет операции разыменования ниже, чем приоритет операции обращения к полю структуры.

Для выделения памяти можно пользоваться любым способом резервирования памяти.

Список из одного такого элемента имеет вид, представленный на рис. 15.1.

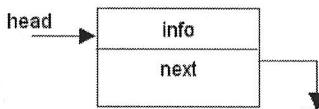


Рис. 15.1. Список из одного элемента

Но список редко состоит из одного элемента. Поэтому следующим шагом является добавление нового элемента в список. Для этого необходимо объявить указатель на текущий элемент p и выполнить следующие действия:

```
NodePtr p;           // указатель на текущий элемент
NodePtr tail;        // указатель на "хвост" очереди
if (head == NULL)
{ head = new node;
  head->info = 1;           // или какому-то другому значению
  head->next = NULL;
}
p = new node;
p->info = 2;
```

```
p->next = head->next; // в данном случае – NULL
head->next = p;
tail = p;
```

После вставки элемента список приобретает вид, показанный на рис. 15.2.

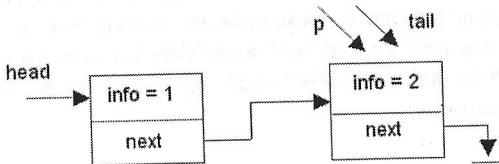


Рис. 15.2. Список из двух элементов

После вставки аналогичным образом еще нескольких элементов список приобретает вид, показанный на рис. 15.3. Если при этом известно количество элементов в очереди, используют оператор цикла с параметром:

```
NodePtr p; // указатель на текущий элемент
NodePtr tail; // указатель на "хвост" очереди
int N = 10; // количество элементов в очереди
int cnt = 1; // счетчик элементов в очереди
head = NULL;

if (head == NULL)
{ head = new node;
  head->info = cnt++;
  head->next = NULL;
  tail = head;
}

for (int i = 2; i <= N; i++)
{ p = new node;
  p->info = cnt++;
  tail->next = p; // в данном случае – NULL
  p->next = NULL;
  tail = p;
}
```

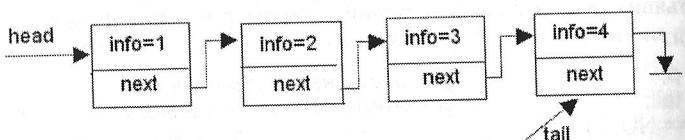


Рис. 15.3. Список из нескольких элементов

Мы создали список, или *очередь* (queue, FIFO, First Input-First Output). Первым Вшел – Первым Вышел), поэтому заголовочный элемент (*head*) всегда указывает на первый элемент списка.

**Пример 15.1.** Программа формирования очереди из 10 элементов и вывода ее на экран.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ struct node {
    node *next;
};

typedef node *NodePtr; // указатель на тип node
NodePtr head = NULL;
NodePtr p;
NodePtr tail;
int N = 10; // количество элементов в очереди
int cnt = 1; // счетчик элементов в очереди
if (head == NULL)
{head = new node;
  head->info = cnt++;
  head->next = NULL;
  tail = head;
}
for (int i = 2; i <= N; i++)
{ p = new node;
  p->info = cnt++;
  tail->next = p; // в данном случае – NULL
  p->next = NULL;
  tail = p;
}
// Вывод очереди на экран
p = head;
for (int i = 1; i <= N; i++)
{ cout << p->info << ' ';
  p = p->next;
}
cout << endl;
getch();
return 0;
}
```

### 15.3. Формирование стека

Стеком называется структура данных, организованная по принципу «Последним вошел – первым вышел» (LIFO, Last Input – First Output). Аналог стека – бусинки, которые нанизаны на нитку. Последнюю из нанизанных бусинок снять легко, а чтобы добраться до самой первой, нужно снять все. Как и очередь, стек организован на основе самоссыльочных

структур (см. рис. 15.1). Поэтому для формирования стека объявим такую же структуру, как и для формирования очереди:

```
struct node { int info;
    struct node *next;
};
typedef node *NodePtr; //указатель на тип Node
```

Первым в стек, показанный на рис. 15.4, помещен элемент с информационным полем, равным 1, вторым – 2 и т. д. Как видно из рисунка, следуя по указателям next от элемента head, мы доберемся до элемента 1 в последнюю очередь.

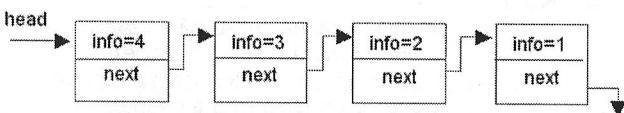


Рис. 15.4. Стек

Формирование стека можно осуществить с помощью следующих операторов:

```
NodePtr head = NULL;
if (head == NULL)
{ head = new node;
    head->info = rand()%100 - 50;
    head->next = NULL;
}
else { p = new node;
    p->info = rand()%100 - 50;
    p->next = head;
    head = p;
}
```

Выбор оператора цикла при создании стека происходит в зависимости от того, известно ли количество элементов в стеке или признак завершения его формирования.

**Пример 15.2.** Программа формирования стека из 10 элементов и вывода его на экран.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ struct node { int info;
    node *next;
};
typedef node *NodePtr; // указатель на тип node
NodePtr head = NULL;
NodePtr p; // указатель на текущий элемент
```

```
int N = 10; // количество элементов в стеке
int cnt = 1; // счетчик элементов в стеке

if (head == NULL)
{ head = new node;
    head->info = cnt++;
    head->next = NULL;
}
for (int i = 2; i <= N; i++)
{ p = new node;
    p->info = cnt++;
    p->next = head;
    head = p;
}
// Вывод стека на экран
p = head;
for (int i = 1; i <= N; i++)
{ cout << p->info << ' ';
    p = p->next;
}
cout << endl;
_getch();
return 0;
```

#### 15.4. Добавление и удаление элементов в односвязных списках

При формировании очереди было показано добавление элемента в ее конец, при формировании стека – в его конец. Но реально может возникнуть необходимость добавить элемент в любое место формируемого списка. При этом если список уже сформирован, то не важно, что он собой представляет – стек или очередь (рис. 15.5, 15.6).

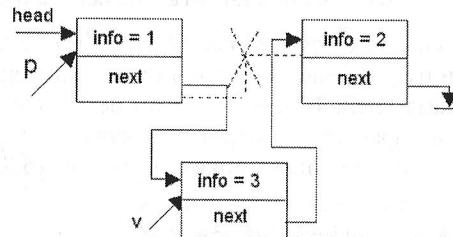


Рис. 15.5. Добавление элемента в середину списка

Действия, связанные с добавлением и удалением элементов стека и очереди, аналогичны, поскольку, когда список уже сформирован, доступ к его элементам осуществляется одинаково, путем прохождения от

элемента, на который указывает заголовочный указатель, до последнего элемента, поле-указатель которого равно NULL.

Для добавления элемента в середину списка нужно перенаправить указатели таким образом, чтобы поле next элемента p, за которым будет располагаться добавляемый элемент v, указывало на него, а поле next элемента v указывало на следующий за ним элемент.

```
NodePtr v, p;
...
v = new node;
v->info = 3;
v->next = p->next;
p->next = v;
```

Добавить элемент в начало списка еще проще. Нужно направить указатель head на добавляемый элемент v, а v->next – на элемент, который прежде был первым в списке.

```
NodePtr v, p;
...
v = new node;
v->info = 0;
v->next = head->next;
head = v;
```

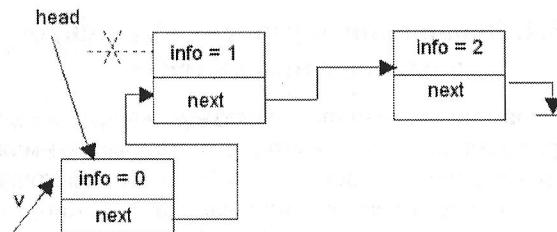


Рис. 15.6. Добавление элемента в начало списка

При работе со списками весьма распространено не только добавление элементов, но и их удаление, которое также выполняется с помощью перенаправления указателей. Только при этом не следует забывать освобождать память и возвращать ее обратно в кучу. Удаление элементов из начала, середины и конца списка показано на рис. 15.7, 15.8 и 15.9 соответственно.

```
NodePtr d; // указатель на удаляемый элемент
```

```
d = head;
head = head->next;
delete d;
```

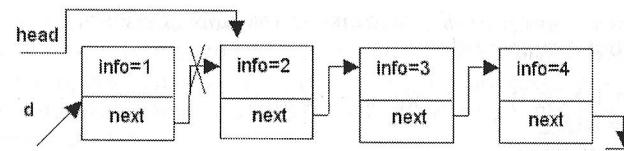


Рис. 15.7. Удаление элемента из начала списка

```
NodePtr d; // указатель на удаляемый элемент
...
p->next = d->next;
delete d;
```

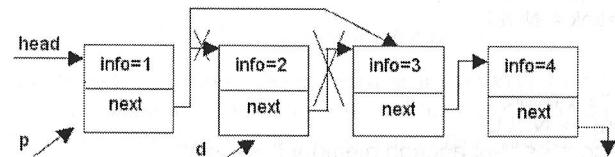


Рис. 15.8. Удаление элемента из середины списка

```
NodePtr d; // указатель на удаляемый элемент
...
p->next = d->next; // фактически это p->next = NULL
delete d;
```

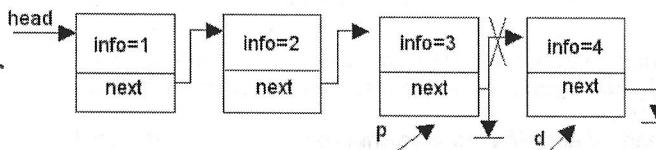


Рис. 15.9. Удаление элемента из конца списка

Не меньшее значение, чем добавление и удаление элементов, в связном списке имеет поиск нужного элемента.

### Пример 15.3. Поиск элемента в связном списке.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
    int N; // количество элементов в списке
    int target; // искомое значение
    int count; // местоположение искомого элемента
    struct node { int dat; // информационное поле
                  node *link; // поле-указатель
    };
    typedef node *NodePtr; // указатель на тип Node
    NodePtr head = NULL;
```

```

NodePtr here;           // указатель на текущий элемент
cout << "Input a quantity elements: ";
cin >> N;
for (int i = 0; i<N; i++)
{ if (head == NULL)    // формируем стек
    { head = new node;
      if (head == NULL)
        { cout << "Not enough memory !" << endl;
          exit(1);
        }
      head->dat = rand()%100 - 50;
      cout << head->dat << ',';
      head->link = NULL;
    }
    else
    { here = new node;
      if (here == NULL)
        { cout << "Not enough memory !" << endl;
          exit(2);
        }
      here->dat = rand()%100 - 50;
      cout << here->dat << ',';
      here->link = head;
      head = here;
    }
}
cout << endl;
cout << "Input target ";
cin >> target;
here = head; // встали в начало списка
count = 1;
if (here == NULL)
    cout << "List is empty!" << endl;
else {while (here->dat != target && here->link != NULL)
    { here = here->link;
      count++;
    }
    if (here->dat == target)
        cout << "Target element has a number" << count << endl;
    else cout << "Target element is not found!" << endl;
}
_getch();
return 0;
}

```

### 15.5. Двусвязные списки

Рассмотренные нами очереди и стеки имеют по одному полю-  
указателю, поэтому их называют односвязными списками. Односвязные  
списки, хотя и очень удобны для хранения и обработки данных, имеют

один существенный недостаток: их можно просмотреть только в одном направлении, от заголовочного элемента до конца списка. Вернуться для обработки к предыдущему элементу относительно текущего нельзя. Поэтому приходится усложнять алгоритмы обработки, прибегать к повторным просмотрам и т. д.

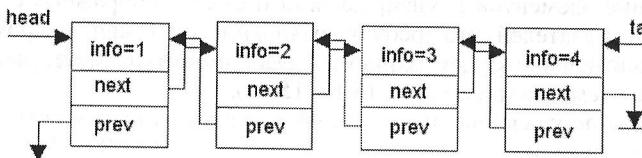


Рис. 15.10. Двусвязный список

Эти недостатки устраняются введением в структуру еще одного по-  
ля-указателя, указывающего на предыдущий элемент списка (рис. 15.10).  
Такие списки называются *двусвязными*, или *двунаправленными*, или *деками* (*deque*).

Действия над двусвязными списками очень похожи на действия над  
односвязными списками, но необходимо учитывать и корректировать на-  
правление обоих указателей.

#### Пример 15.4. Формирование двусвязного списка.

Создадим узел для двусвязного списка.

```

struct node { int info;
    struct node *next;   // указатель на следующий узел
    struct node *prev;   // указатель на предыдущий узел
};
typedef node *NodePtr;           //указатель на тип Node
NodePtr head = NULL, tail = NULL, p, d;

```

При формировании двусвязного списка также воспользуемся опера-  
цией new для резервирования памяти.

```

int N=10;
for (int i = 0; i<N; i++)
{ p = new node;
  if (p == NULL)
    { cout << "No memory!" << endl;
      exit(1);
    }
  p->info = rand()%100 - 50;
  cout << p->info << ',';
  if (head == NULL)
    { head = tail = p;
      p->prev=NULL;
      p->next=NULL;
    }
  else { p->prev = tail;
    tail->next = p;
  }
}

```

```

    tail = p;
    tail->next=NULL;
}
cout << endl;

```

Удаление элементов двунаправленного списка сопряжено с перенаправлением указателей, что требует повышенного внимания. Кроме того, это перенаправление будет разным в зависимости от местоположения удаляемого элемента (рис. 15.11, 15.12, 15.13).

Действия по удалению элементов двусвязного списка можно описать следующим образом.

#### Пример 15.5. Удаление элементов двусвязного списка.

```

while (head!=NULL)
{ d=head;
  if (d->prev != NULL)           // элемент – не первый
    { d->prev->next = d->next;
      if (d->next != NULL)        // элемент – не последний
        { d->next->prev = d->prev;
          cout << p->info << ' ';
        }
      else                         // элемент – последний
        { tail = d->prev;
          cout << p->info << ' ';
        }
    }
  else                           // элемент – первый
    { if (d->next != NULL)        // список не пустой
      { d->next->prev = NULL;
        head = d->next;
      }
      else { head = tail = NULL;
              cout << d->info << ' ';
            }
        }
      }
  delete d;
}

```

Просмотр двусвязного списка с целью дальнейшей его обработки возможен в любом из двух направлений: с начала и с конца списка.

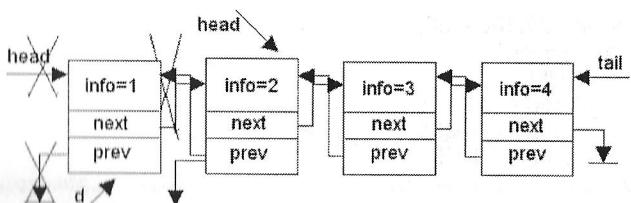


Рис. 15.11. Удаление первого элемента списка

#### Пример 15.6. Просмотр списка от начала к концу.

```

NodePtr p;
p = head;
while(p) { cout << p->info << ' ';
  p = p->next;
}
cout << endl;

```

#### Пример 15.7. Просмотр списка от конца к началу.

```

NodePtr p;
p = tail;
while(p) { cout << p->info << ' ';
  p = p->prev;
}
cout << endl;

```

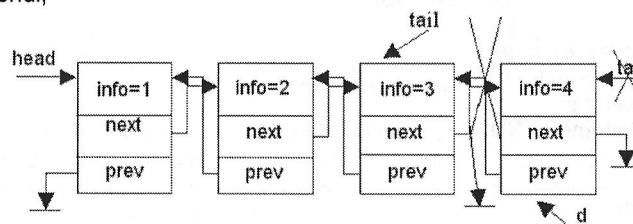


Рис. 15.12. Удаление последнего элемента списка

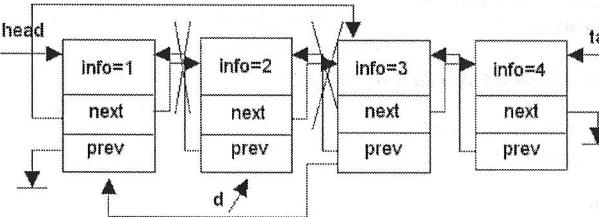


Рис. 15.13. Удаление среднего элемента списка

#### Пример 15.8. Поиск элемента в списке.

```

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <time.h>
#include <conio.h>
using namespace std;
struct node { int info;
  node *next; // указатель на следующий узел
  node *prev; // указатель на предыдущий узел
};
int _tmain(int argc, _TCHAR* argv[])
{time_t t;
 srand((int)time(&t));

```

```

setlocale(LC_ALL, "Russian");
typedef node *NodePtr; //указатель на тип Node
NodePtr head = NULL, tail = NULL, p, d;
int N=10, key;
for (int i = 0; i<N; i++)
{ p = new node;
if (p == NULL)
    { cout << "No memory!" << endl;
      exit(1);
    }
p->info = rand()%10;
cout << p->info << ' ';
if (head == NULL) { head = tail = p;
                    p->prev=NULL;
                    p->next=NULL;
}
else { p->prev = tail;
       tail->next = p;
       tail = p;
       tail->next=NULL;
}
}
cout << endl;
// Просмотр списка от начала к концу:
p = head;
while(p) { cout << p->info << ' ';
            p = p->next; }
cout << endl;
// Просмотр списка от конца к началу:
p = tail;
while(p) { cout << p->info << ' ';
            p = p->prev;
}
cout << endl;
// Поиск элементов в списке:
cout << "Введите искомое значение: ";
cin>>key;
p = head;
int cnt = 0;
while(p) { if (key == p->info) cnt++;
            p = p->next;
}
if (cnt == 0) cout << "Совпадений не найдено" << endl;
else cout << "Количество искомых значений - " << cnt << endl;
// Удаление элементов двусвязного списка.
while (head!=NULL)
{ d=head;
  if (d->prev != NULL) // элемент - не первый
    { d->prev->next = d->next;
      if (d->next != NULL) // элемент - не последний
        { d->next->prev = d->prev;
          delete d;
        }
      else
        { tail=d->prev;
          tail->next=NULL;
        }
    }
  else
    { tail=d->next;
      tail->prev=NULL;
    }
  head=d->next;
}
}

```

```

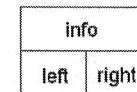
    { d->next->prev = d->prev; }
else                                //элемент - последний
    { tail = d->prev; }
}
else                                // элемент - первый
{if (d->next != NULL)   // список не пустой
    { d->next->prev = NULL;
head = d->next;
    }
else { head = tail = NULL;
    }                                // список пустой
}
delete d;
}
_getch();
return 0;
}

```

## 15.6. Бинарные деревья

Стеки и очереди являются линейными структурами данных. Деревья (в нашем случае – бинарные) представляют собой нелинейную (двумерную) структуру. Узел бинарного дерева состоит из информационного поля и двух указателей (рис. 15.14).

Первый узел дерева называется **корневым узлом**. Каждый указатель в корневом узле ссылается на **узел-потомок**, или **дочерний узел**. Левый дочерний узел является первым узлом в левом поддереве, правый — в правом поддереве (рис. 15.15).



**Рис. 15.14.** Узел бинарного дерева

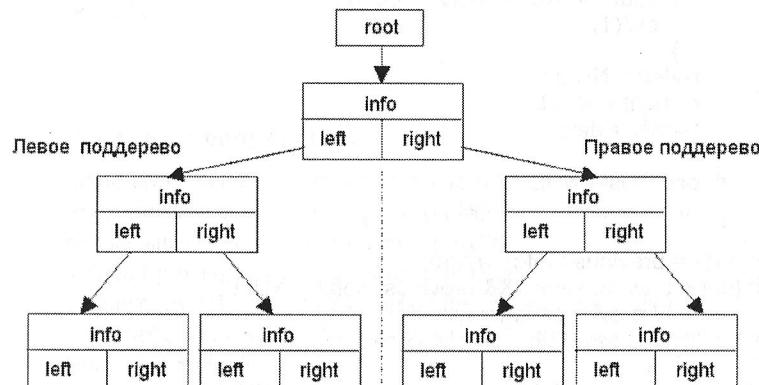


Рис. 15.15. Бинарное дерево

Мы будем рассматривать бинарное дерево, которое называется *деревом двоичного поиска (дихотомии)*. Особенности этого дерева состоят в том:

- что оно не имеет узлов с одинаковым информационным полем;
- значения в узлах левого поддерева меньше, чем значение родительского узла;
- значения в узлах правого поддерева больше, чем значение родительского узла.

Деревья по своей природе рекурсивны, так как состоят из поддеревьев, каждое из которых также представляет собой поддерево. Поэтому функции, используемые для работы с деревьями, также рекурсивные.

#### Пример 15.10. Формирование двоичного дерева.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
// Узел дерева имеет вид:
struct tree
{
    char info;
    tree *left;
    tree *right;
};
tree *root; // указатель на корень дерева
// Функция insert() добавляет в состав дерева новый элемент.
void insert (tree *r, tree *previous, char dat)
{
    if (r==NULL)
    {
        r = new tree;
        if (r==NULL)
            { cout << "No memory" << endl;
              exit(1);
            }
        r->left = NULL;
        r->right = NULL;
        r->info = dat;
    }
    if (previous==NULL) { root = r;
                          return;
                      }
    if(dat == previous->info) return;
    if (dat < previous->info && previous->left == NULL)
        { previous->left = r;
          return;
        }
    if (dat > previous->info && previous->right == NULL)
        { previous->right = r;
          return;
        }
    if (dat < previous->info) insert(r, previous->left, dat);
    else insert(r, previous->right, dat);
}
```

```
else insert(r, previous->right, dat);
}
// Для отображения дерева можно написать функцию:
void print_tree(tree *r, int l)
{
    int i;
    if (r==NULL) return;
    print_tree(r->right, l+1);
    for(i = 0; i < l; ++i) cout << " ";
    cout << r->info << endl;
    print_tree(r->left, l+1);
}
/* Как и в связных списках, в дереве может возникнуть необходимость найти узел, который содержит заданное значение информационного поля, если такой узел существует */
tree * search(tree *r, char key)
{
    if (r == NULL) return r;
    while (r->info != key)
    {
        if (key < r->info) r = r->left;
        else r = r->right;
        if (r == NULL) break;
    }
    return r;
}

int _tmain(int argc, _TCHAR* argv[])
{
    char s;
    root=NULL;
    do { cout << "Input symbol(. - for exit):";
          cin >> s;
          if (s !='.') insert(NULL,root, s);
      } while (s !='.');
    print_tree(root,0);
    getch();
    return 0;
}
```

#### Контрольные вопросы

- Что такое динамические структуры?
- Почему динамические структуры называют самоссыльочными?
- На что указывает указатель на структуру?
- Приведите примеры динамических структур.
- Перечислите действия, необходимые для создания списка.
- Сколько информационных полей и какого типа может иметь самоссыльная структура?
- По какому принципу организован стек и по какому очередь?
- В чём заключаются особенности организации двусвязных списков?

## Практические задания 18

Следующие задания предполагают написание фрагментов программы или функций, связанных с формированием стека или очереди, называемых далее списками.

1. Вычислить среднее арифметическое значение элементов списка.
2. Удалить из списка все вхождения элемента, имеющего заданное значение.
3. Вставить в упорядоченный по неубыванию список элемент с заданным значением так, чтобы не нарушить упорядоченность.
4. Дописать в конец очереди содержащиеся в ней нечетные элементы.
5. Удалить из списка первый, последний и средний его элементы.
6. Объединить два упорядоченных по возрастанию списка в один, упорядоченный по неубыванию.
7. Вставить в список элемент с заданным значением, расположив его за элементом, значение которого попадает в указанный диапазон.

## ГЛАВА 16. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Объектно-ориентированное программирование (ООП) основано на трех концепциях:

- инкапсуляции,
- наследовании,
- полиморфизме.

Базовой идеей объектно-ориентированного подхода является объединение данных и действий, которые производятся над этими данными, в единую языковую конструкцию, называемую *объектом*. Механизм, объединяющий данные и код, реализующий их обработку, называется *инкапсуляцией*.

Объект является переменной некоторого типа, определенного пользователем, и называемого классом. Другими словами, объект – это экземпляр класса. *Класс* – это объектно-ориентированный инструмент для создания новых типов данных, являющихся объектами.

*Наследование* – это процесс, посредством которого один объект может наследовать (приобретать) свойства другого объекта и добавлять к ним свои собственные черты. Наследование позволяет поддерживать концепцию иерархии классов.

Синтаксис описания класса практически идентичен синтаксису описания структуры:

```
class имя_класса { данные-члены_класса;
                    функции-члены_класса;
} список_объектов;
```

Класс представляет собой тип, создаваемый пользователем. Список объектов может отсутствовать, тогда объекты объявляют в программе по мере необходимости, так же, как это происходит при объявлении структурного шаблона и переменных структурного типа.

Члены класса делятся на две категории:

- данные-члены класса (data members);
- программный код, представляющий собой функции-члены класса (member functions) или методы.

Данные-члены классов не могут определяться как auto (локальные переменные), extern (внешние) или register (регистровые). В качестве данных-членов класса не может выступать представитель самого этого класса, но допускается использовать указатель или ссылку на представитель этого класса.

Данные-члены класса и функции-члены класса иногда называют *данными-элементами и функциями-элементами класса*.

По умолчанию все данные-члены класса и функции-члены класса являются закрытыми, т. е. доступными только для элементов своего класса. Чтобы сделать элементы класса открытыми, т. е. доступными для других частей программы, в которой содержится класс, используется модификатор public:

```
class AnyClass { int data; // закрытый (по умолчанию) элемент класса
public:
    void set_data(int num); // открытые (по модификатору)
                           // public
    int get_data();         // функции-элементы класса
};
```

Открытые функции-члены класса имеют доступ ко всем членам своего класса, в том числе и к закрытым, осуществляя таким образом связь класса с окружающей программной средой (рис. 16.1) [19].

В приведенном описании класса AnyClass функции-члены объявлены, но пока не определены. Для определения функции-члена нужно связать имя класса, к которому принадлежит функция, с именем функции. Для этого указывают тип возвращаемого значения функции, а затем имя класса и имя функции, разделенные знаком «::» (операция расширения области видимости) и список аргументов в скобках. После этого заголовка располагается тело функции в фигурных скобках.

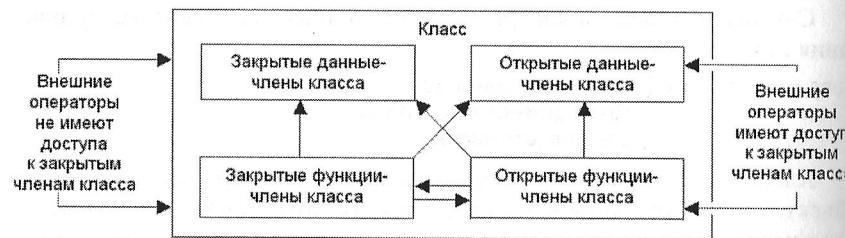


Рис. 16.1. Открытые и закрытые члены класса

```
void AnyClass::set_data (int num)
{ data = num; // тело функции
}
int AnyClass::get_data ()
{ return data; // тело функции
}
```

Обявление класса только определяет тип объекта, но не создает ни одного объекта, следовательно, не производит выделения памяти под объект. Для создания объекта используют имя класса как спецификатор типа данных:

```
AnyClass ob1, ob2; // создали два объекта типа AnyClass
```

После создания объекта можно обращаться к открытым элементам класса, используя операцию «точка» (.):

```
ob1.set_data(10); // ob1.data приобретает значение 10
ob2.set_data(99); // ob2.data приобретает значение 99
```

Каждый объект класса имеет собственную копию всех переменных, объявленных внутри класса.

Теперь можно вывести значения данных-элементов на экран.

```
cout << ob1.get_data() << endl;
cout << ob2.get_data() << endl;
```

Операторы вида

```
ob1.data = 10;
ob2.data = 100;
```

приводят к возникновению ошибок компиляции, так как data является закрытым элементом класса.

При объявлении класса в виде

```
class AnyClass { public:
    int data; // открытый (по модификатору
              // public) элемент класса
    void set_data(int num); // открытые (по модификатору
    int get_data(); // public) функции-элементы класса
};
```

Операторы ob1.data = 10;  
ob2.data = 100;

компилируются и работают корректно, так как доступ осуществляется к открытым элементам класса. В данном случае нет необходимости в открытых функциях-элементах класса set\_data() и get\_data().

**Пример 16.1.** Создание класса «мобильный телефон», установка данных и вывод их на экран.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class phone
{
    int kode; // код
    unsigned long int number; // номер
    int PIN; // PIN-код SIM-карты
    unsigned long int IMEI; // идентификационный номер аппарата
public:
    void set_phone(int k, unsigned long int n, int p, unsigned long int i);
    void print_phone();
}
int _tmain(int argc, _TCHAR* argv[])
```

```

{ setlocale(LC_ALL, "Russian");
  phone MyLG; // объект класса phone
  MyLG.set_phone(916, 3751831, 3182, 58316897);
  MyLG.print_phone();
  _getch();
  return 0;
}
void phone::set_phone(int k, unsigned long int n, int p, unsigned long int i)
{
  kode = k;
  number = n;
  PIN = p;
  IMEI = i;
}
void phone::print_phone()
{
  cout << "Код: " << kode << endl;
  cout << "Номер: " << number << endl;
  cout << "PIN-код: " << PIN << endl;
  cout << "Идентификатор: " << IMEI << endl;
}

```

Функции-элементы класса могут быть встраиваемыми. Встраиваемые функции-элементы должны удовлетворять требованиям, предъявляемым к встраиваемым функциям, не являющимся членами класса. Перепишем пример 16.1 таким образом, чтобы функции-члены класса оказались встраиваемыми.

**Пример 16.2.** Создание класса «мобильный телефон» со встраиваемыми функциями-элементами.

```

#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <math.h>
#include <conio.h>
using namespace std;
class phone
{
  int kode; // код
  unsigned long number; // номер
  int PIN; // PIN-код SIM-карты
  unsigned long IMEI; // идентификационный номер аппарата
public:
  void set_phone(int k, unsigned long n, int p, unsigned long i)
  {
    kode = k;
    number = n;
    PIN = p;
    IMEI = i;
  }
  void print_phone()
  {
    cout << "Код: " << kode << endl;
    cout << "Номер: " << number << endl;
  }
}

```

```

cout << "PIN-код: " << PIN << endl;
cout << "Идентификатор: " << IMEI << endl;
}
int _tmain(int argc, _TCHAR* argv[])
{
  setlocale(LC_ALL, "Russian"); // подключение русификатора
  phone MyLG; // объект класса phone
  MyLG.set_phone(916, 3751831, 3182, 583168971);
  MyLG.print_phone();
  _getch();
  return 0;
}

```

Каждый объект имеет собственные копии данных-элементов. Но функции-элементы (методы) у них общие. То есть, методы класса создаются и размещаются в памяти ЭВМ при создании класса.

```

class AnyCls { static type st_d;
  type d1;
  type d2;
  type d2;
  specification:
  type fn1(list_of_parameters);
  type fn2(list_of_parameters);
} ob1, ob2;

```

Размещение данных-элементов и функций-элементов объектов ob1 и ob2 в памяти ЭВМ показано на рис. 16.2.

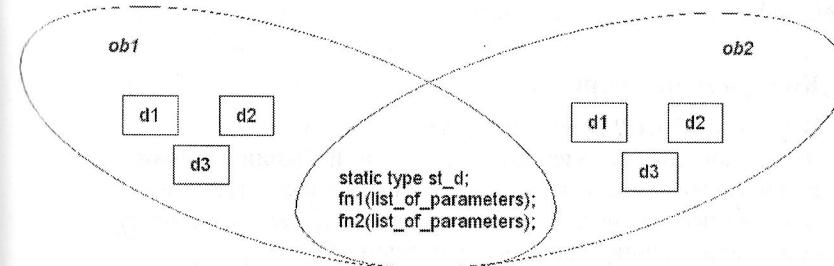


Рис. 16.2. Функции-элементы и статические данные-элементы

Если данное-элемент описано как статическое, т. е. со спецификатором static, то его значение будет одинаковым для всех объектов данного класса, так как для всех объектов этого класса существует только одна общая копия этой переменной, которая, подобно методам, используется совместно всеми объектами. Эта же статическая переменная будет применяться и для всех классов, производных от того класса, в котором она содержится. Статическое поле видимо только внутри класса, но существует оно в течение всего времени жизни программы. Статические пере-

менные класса используются для хранения данных, общих для объектов класса.

Объявление статического поля находится внутри определения класса, а его определение, как правило, вне класса и часто является определением глобальной переменной. Поэтому доступ к статической переменной-элементу класса возможен без связи с объектом.

#### *Пример 16.3*

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class stat_var
{
public:
    static int st_i;
    void set_st_i(int n) { st_i = n; }
    int get_st_i() { return st_i; }
};
int stat_var :: st_i;
int _tmain(int argc, _TCHAR* argv[])
{
    stat_var ob1, ob2;
    stat_var :: st_i = 125; // присваивание значения
    // статической переменной без упоминания имени объекта
    cout << ob1.get_st_i() << endl;
    cout << ob2.get_st_i() << endl;
    _getch();
    return 0;
}
```

#### Контрольные вопросы

- Что такое объект? Дайте определение класса.
- Что такое данные-элементы класса и функции-элементы класса? В чем заключаются особенности открытых и закрытых членов класса?
- Как называется механизм, объединяющий данные и программный код, реализующий обработку этих данных?
- Что такое статические данные-элементы? Как они объявляются и где используются?

## ГЛАВА 17. СИСТЕМА ВВОДА-ВЫВОДА В C++

### 17.1. Основные понятия

Программа на C++ автоматически открывает 4 потока:

cin – стандартный ввод – клавиатура;

cout – стандартный вывод – экран;

cerr – стандартная ошибка – экран;

clog – буферизованная стандартная ошибка – экран.

В C++ система ввода-вывода поддерживается заголовочным файлом iostream, в котором задана иерархия классов, применяемых для создания потоков ввода-вывода.

С понятием потока также связан класс ios, который определяет три основные составляющие потокового ввода-вывода:

- флаги форматирования,
- флаги ошибок,
- режимы работы с файлами.

### 17.2. Форматируемый ввод-вывод

Здесь нас будут интересовать только флаги форматирования, которые задают способ отображения данных. Они занимают 16 младших разрядов в длинном целом и имеют идентификаторы, заданные с помощью перечислимого типа:

```
enum {skipws      = 0x0001,
       left        = 0x0002,
       right       = 0x0004,
       internal    = 0x0008,
       dec         = 0x0010,
       oct         = 0x0020,
       hex         = 0x0040,
       showbase   = 0x0080,
       showpoint  = 0x0100,
       uppercase  = 0x0200,
       showpos    = 0x0400,
       scientific = 0x0800,
       fixed       = 0x1000,
       unitbuf    = 0x2000,
       stdio       = 0x4000,
};
```

Каждому идентификатору присвоено его шестнадцатеричное значение, соответствующее номеру его разряда (рис. 17.1).

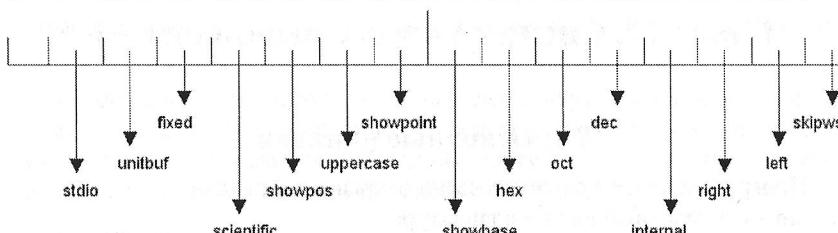


Рис. 17.1. Флаги формата ios

Если флаг формата установлен, реализуется соответствующая ему функция (табл. 17.1). При сбросе флага используется формат по умолчанию.

Таблица 17.1. Флаги формата ios

Установленный флаг	Описание
skipws	При выполнении ввода в поток начальные невидимые символы (' ', TAB, \n) отбрасываются. Флаг используется при чтении дисковых файлов определенного формата
left	Выравнивание вывода по левому краю
right	Выравнивание вывода по правому краю (по умолчанию при сброшенных флагах left, internal)
internal	Вставка пробелов между знаком и цифрами числа для обеспечения выравнивания числа по всей ширине поля вывода
dec	Вывод целых чисел в десятичной системе счисления (по умолчанию)
oct	Вывод чисел в восьмеричной системе счисления. Для возврата в десятичную систему счисления нужна явная установка флага dec
hex	Вывод чисел в шестнадцатеричной системе счисления. Для возврата в десятичную систему счисления нужна явная установка флага dec
showbase	На экран будет выводиться основание системы счисления
showpoint	Наличие десятичной точки и последующих нулей для всех выводимых чисел с плавающей точкой
uppercase	Символы «е» в экспоненциальной форме и «х» в шестнадцатеричном представлении чисел выводятся в верхнем регистре (по умолчанию – в нижнем)
showpos	Наличие знака «+» перед положительными десятичными числами
scientific	Вывод чисел в экспоненциальной форме

## Окончание табл. 17.1

Установленный флаг	Описание
fixed	Числа с плавающей точкой выводятся в обычной системе обозначений с шестью десятичными знаками
unitbuf	Флеширование буфера потока вывода после каждой операции вывода. Этот флаг имеет особенности, зависящие от конкретного компилятора
stdio	Потоки cout и cerr автоматически флешируются после каждой операции вывода. Этот флаг неопределен стандартом ANSI C++ и поддерживается не каждым компилятором

Флеширование – процесс записи на физическое устройство содержащего буферной памяти потока.

Для установки флага формата используется функция setf(), являющаяся элементом класса ios. Ее синтаксис имеет следующий вид:

long setf(long флаги);

Эта функция возвращает предыдущие установки флагов формата и устанавливает заданные флаги, которые также задаются внутри класса ios.

Установка флага происходит с помощью оператора вида  
поток.setf(ios::флаг);

при этом любой вызов функции setf() происходит относительно конкретного потока. Поэтому каждый поток отдельно поддерживает свое собственное состояние формата.

Функция setf() может устанавливать сразу несколько флагов:

поток.setf(ios::флаг1 | ios::флаг2 | ios::флаг3 ...);

Функцией, обратной setf(), является функция-элемент ios unsetf(), которая сбрасывает один или несколько флагов формата. Ее синтаксис:

long unsetf(long флаги);

Флаги, перечисленные в качестве параметров, сбрасываются, состояние остальных флагов не изменяется.

Функция-элемент ios flags() при отсутствующих аргументах возвращает текущее значение флагов формата, при наличии аргументов – устанавливает флаги формата, являющиеся аргументами. Эта функция также работает относительно конкретного потока. Ее синтаксис:

long flags(); или long flags(long f);

В качестве параметра этой функции используется битовый шаблон, представляющий собой длинное целое, в котором в шестнадцатеричном виде представлен набор флагов формата.

**Пример 17.1.** Работа с флагами формата.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{ int i;
    double f;
    cout << "Input i ";
    cin >> i;
    cout << "Input f ";
    cin >> f;
    cout.setf(ios::hex);
    cout << i << endl;
    cout.setf(ios::uppercase | ios::scientific);
    cout << f << endl;
    cout.unsetf(ios::uppercase);
    cout << f << endl;
    _getch();
    return 0;
}
```

**Пример 17.2.** [29]. Определение состояния флагов формата.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;
void showflags();
int _tmain(int argc, _TCHAR* argv[])
{ setlocale(LC_ALL, "Russian"); //подключение русификатора
    long f = 0x048C;
    showflags(); // флаги по умолчанию
    cout.setf(ios::oct | ios::showbase | ios::fixed);
    showflags();
    cout.flags(f);
    showflags();
    _getch();
    return 0;
}
void showflags()
{ long f, i;
    int j;
    char flags[15][12] = { "skipws",
                           "left",
                           "right",
                           "internal",
                           "dec",
                           "oct",
                           "hex",
                           "showbase",
                           "showpoint",
                           "uppercase",
                           "showpos",
                           "scientific",
                           "fixed",
                           "unitbuf",
                           "stdio"
                           };
    f = cout.flags(); // получение состояния флагов
    // контроль состояния каждого из флагов
    for (i = 1, j = 0; i <= 0x4000; i = i << 1, j++)
        if (i & f) cout << flags[j] << " - установлен\n";
        else cout << flags[j] << " - сброшен\n";
    cout << endl;
}
```

```
"hex",
"showbase",
"showpoint",
"uppercase",
"showpos",
"scientific",
"fixed",
"unitbuf",
"stdio"
};

f = cout.flags(); // получение состояния флагов
// контроль состояния каждого из флагов
for (i = 1, j = 0; i <= 0x4000; i = i << 1, j++)
    if (i & f) cout << flags[j] << " - установлен\n";
    else cout << flags[j] << " - сброшен\n";
cout << endl;
}
```

Результат работы программы:

skipws - установлен	skipws - сброшен
left - сброшен	left - сброшен
right - сброшен	right - сброшен
internal - сброшен	internal - установлен
dec - сброшен	dec - сброшен
oct - сброшен	oct - сброшен
hex - сброшен	hex - сброшен
showbase - сброшен	showbase - сброшен
showpoint - сброшен	showpoint - сброшен
uppercase - установлен	uppercase - установлен
showpos - сброшен	showpos - установлен
scientific - сброшен	scientific - сброшен
fixed - сброшен	fixed - сброшен
unitbuf - сброшен	unitbuf - установлен
stdio - сброшен	stdio - сброшен

skipws - сброшен
left - сброшен
right - установлен
internal - установлен
dec - сброшен
oct - сброшен
hex - сброшен
showbase - установлен
showpoint - сброшен
uppercase - сброшен
showpos - установлен
scientific - сброшен
fixed - сброшен
unitbuf - сброшен
stdio - сброшен

Операция поразрядного AND позволяет определить, установлен ли флаг с порядковым номером j.

### 17.3. Функции установки ширины поля, точности и символа заполнения

Класс ios содержит 3 функции форматирования, позволяющие определить:

ширину поля вывода – int width(int w);

точность – int precision(int p);

символ заполнения – char fill(char ch).

По умолчанию при вводе некоторого значения, оно занимает количество позиций, соответствующее количеству вводимых символов. Используя функцию `width()`, можно задать ширину поля (параметр `w`).

По умолчанию при выводе значений с плавающей точкой после десятичной точки ставится 6 цифр. Точность вывода можно изменить с помощью функции `precision()` (параметр `p`).

По умолчанию при выводе в качестве символа заполнения используется пробел. Функция `fill()` позволяет использовать в качестве символа заполнения любой другой символ (параметр `ch`).

Все перечисленные функции устанавливают новое значение параметра и возвращают его прежнее значение.

#### 17.4. Манипуляторы ввода-вывода

При вводе-выводе данных можно воспользоваться *манипуляторами*, т. е. специальными функциями форматирования, которые могут находиться в теле оператора ввода-вывода (табл. 17.2). Если в манипуляторе используются параметры, то необходимо подключение заголовочного файла `<iomanip.h>`.

Таблица 17.2. Манипуляторы ввода-вывода

Манипулятор	Назначение	Ввод-вывод
dec	Вывод числовых данных в десятичной системе счисления	Вывод
hex	Вывод числовых данных в шестнадцатеричной системе счисления	Вывод
oct	Вывод числовых данных в восьмеричной системе счисления	Вывод
endl	Вывод символа новой строки и флеширование	Вывод
ends	Вывод нуля (NULL)	Вывод
flush	Флеширование	Вывод
ws	Пропуск начальных пробелов	Ввод
resetiosflags (long f)	Сброс флагов, задаваемых в f	Ввод-вывод
setbase (int основание)	Устанавливает основание системы счисления для вывода данных	Вывод
setfill(char ch)	Устанавливает символ заполнения ch	Вывод
setiosflags (long f)	Установка флагов, задаваемых в f	Вывод
setprecision(int p)	Задает число символов после десятичной точки, равное p	Вывод
setw(int w)	Задает ширину поля, равное w позициям	Вывод

**Пример 17.3.** Вывод данных с использованием манипуляторов.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <math.h>
#include <conio.h>
using namespace std;
void showflags();
int _tmain(int argc, _TCHAR* argv[])
{
    double x, y;
    cout << "Input x ";
    cin >> x;
    y = sin(x);
    cout << setprecision(3);
    cout << setw(7) << x;
    cout << setw(7) << y;
    getch();
    return 0;
}
```

#### Контрольные вопросы

- Что задают флаги форматирования?
- Какие функции используются для управления флагами форматирования?
- Что такое битовый шаблон и как он формируется?
- Перечислите манипуляторы ввода-вывода, не имеющие параметров.

# ГЛАВА 18. КОНСТРУКТОРЫ И ДЕСТРУКТОРЫ

## 18.1. Понятие о конструкторах и деструкторах

В процессе работы программы можно присвоить значения данным-элементам класса. Допускается также их инициализация с помощью специальной функции-элемента класса, называемой конструктором. *Конструктор* – это функция-элемент класса, автоматически выполняющаяся в момент создания объекта. Таким образом, инициализация объекта при использовании конструктора выполняется автоматически. То есть в C++ оператор объявления переменной является «оператором действия».

Для глобальных объектов конструктор вызывается в момент начала выполнения программы. Для локальных объектов конструктор вызывается всякий раз при выполнении оператора, объявляющего переменную.

Необходимость использования конструктора обусловлена тем, что данные-элементы класса не могут непосредственно получать начальные значения в определении класса.

Конструктор является функцией-элементом класса с тем же именем, что и класс, и вызывается каждый раз при создании объекта этого класса. Если конструктор не определен программистом, он создается компилятором как функция без параметров и с пустым телом.

Конструктор обладает следующими свойствами:

- для конструктора не указывается тип возвращаемого значения;
- конструктор не может возвращать значение;
- конструктор не наследуется;
- конструктор не может быть объявлен как const, volatile, virtual, static.

Конструкторы можно перегружать, чтобы обеспечить множество наборов начальных значений объектов класса.

Функцией, обратной конструктору, является *деструктор*, вызываемый при удалении объекта. В момент удаления объекта должны выполняться некоторые действия, например освобождение выделенной для него памяти. Для этого в объявление класса включается деструктор, имя которого совпадает с именем класса, но имеет префикс «~» (тильда). Если деструктор не объявлен явно, он, как и конструктор, создается компилятором как пустая функция.

Деструктор обладает следующими свойствами:

- деструктор не имеет аргументов;
- деструктор не может возвращать значение;
- деструктор не наследуется, кроме случая виртуального деструктора;
- деструктор не может быть объявлен как const, volatile, static;
- деструктор может быть объявлен как virtual.

Деструктор класса вызывается при уничтожении объекта, но сам объект не уничтожает, а только выполняет подготовительные действия перед тем, как система начнет освобождать занимаемую им область памяти.

Поскольку деструктор не принимает никаких параметров, класс может иметь только один деструктор, перегрузка деструкторов запрещена.

Деструктор класса вызывается при удалении объекта: локальные объекты удаляются в тот момент, когда они выходят из области видимости, глобальные – при завершении программы.

При создании объектов класса могут создаваться неявные функции-элементы. Автоматическое определение функций-элементов происходит в случае, когда они не определены явно. Автоматически могут определяться:

- конструктор, заданный по умолчанию;
- конструктор копирования;
- оператор присваивания;
- деструктор;
- оператор адресации.

Автоматически созданный конструктор, заданный по умолчанию, превращает объект класса в подобие автоматической переменной, значение которой при инициализации неизвестно.

### Пример 18.1. Использование конструктора без параметров.

```
class MyCls { int a;
public:
    MyCls() { a = 0; } // конструктор без параметров
    void Fn1() { // тело функции Fn1(); }
    void Fn2() { // тело функции Fn2(); }
    ~MyCls() { cout << "Destructor"; } // деструктор
};
```

## 18.2. Конструкторы с параметрами

Конструкторы позволяют задавать инициализирующее значение не в теле функции-конструктора, а передавать их в качестве параметров.

### Пример 18.2. Использование конструкторов с параметрами.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class MyCls { int a, b;
public:
    MyCls(int x, int y) { a = x;
                           b = y;
    }
```

```

void Show_Dat()
{ cout << a << ' ' << b << endl; }
};

int _tmain(int argc, _TCHAR* argv[])
{ MyCls ob(10, 100);           // эта строка является сокращением
    // записи MyCls ob = MyCls(10, 100);
ob.Show_Dat();
_getch();
return 0;
}

```

**Пример 18.3.** Создание класса, закрытым элементом которого является строка.

```

#include "stdafx.h"
#include <iostream>
#include <string>
#include <conio.h>
using namespace std;
class str { char *s;           // указатель на строку
public:
    str(char *word)        // конструктор
    { s=new char[strlen(word)+1];
      strcpy_s(s,strlen(word)+1, word);
    }
    ~str(){ delete [] s; }   //деструктор
    void write();           // прототип функции-элемента
};
void str::write()             // определение функции-элемента
{ cout << s; }
int _tmain(int argc, _TCHAR* argv[])
{ str ob("Good morning");
ob.write();
_getch();
return 0;
}

```

### 18.3. Конструкторы по умолчанию

Конструктор может содержать значения аргументов по умолчанию. Задание в конструкторе значений по умолчанию позволяет гарантировать, что объект будет находиться в непротиворечивом состоянии, даже если в вызове конструктора не указаны никакие значения. Конструктор, у которого все аргументы являются аргументами по умолчанию (или который совсем не требует аргументов), называется *конструктором по умолчанию*. Для каждого класса может существовать только один конструктор, имеющий умолчания.

**Пример 18.4.** Определение класса Cubic, присваивающего единичные значения переменным length, width, height.

```

#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class Cubic
{ public:
    // конструктор с умолчаниями:
    Cubic (int = 1, int = 1, int = 1);
    void setCubic (int, int, int);
    void printCubic();
    int volumeCubic();
private:
    int length;
    int width;
    int height;
};

Cubic::Cubic(int len, int wid, int hei)
// по умолчанию значения равны 1:
{ setCubic(len, wid, hei); }
void Cubic::setCubic(int l, int w, int h)
// некорректные значения явно (без умолчаний)
// устанавливаются равными 1
{ length = (l > 0) ? l : 1;
  width = (w > 0) ? w : 1;
  height = (h > 0) ? h : 1;
}
void Cubic::printCubic()
{ cout << length << "x" << width << "x" << height << endl;
}
int Cubic::volumeCubic()
{ return length * width * height; }

int _tmain(int argc, _TCHAR* argv[])
{ Cubic c1, c2(3), c3(15, 12), c4(5, 6, 7), c5(-2, -3, -5);
cout << "Все аргументы по умолчанию" << endl;
c1.printCubic();           // 1x1x1
c1.volumeCubic();
cout << "Два аргумента по умолчанию" << endl;
c2.printCubic();           // 3x1x1
c2.volumeCubic();
cout << "Один аргумент по умолчанию" << endl;
c3.printCubic();           // 15x12x1
c3.volumeCubic();
cout << "Все аргументы заданы явно" << endl;
c4.printCubic();           // 5x6x7
c4.volumeCubic();
cout << "Все аргументы заданы явно, но неправильно" << endl;
c5.printCubic();           // 1x1x1
c5.volumeCubic();
_getch();
}

```

```
return 0;
}
```

Аргументы конструктора по умолчанию можно объявлять только в прототипе функции-конструктора внутри определения класса.

## 18.4. Указатели на объекты

Указатели на объекты объявляются так же, как и указатели на переменные других типов.

*Пример 18.4*

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class MyCls { int a;
public:
    MyCls(int x) { a = x; }
    int get() { return a; }
};
int _tmain(int argc, _TCHAR* argv[])
{MyCls ob(150);           // создание объекта
MyCls *pCls;              // создание указателя на объект
pCls = &ob;                // передача адреса ob в pCls
cout << "Значение, получаемое при использовании объекта:"
    << ob.get() << endl;
cout << "Значение, получаемое при использовании указателя:"
    << pCls->get() << endl;
_getch();
return 0;
}
```

## 18.5. Перегрузка конструкторов

Необходимость перегрузки конструкторов обусловлена тремя причинами:

- обеспечением гибкости программ,
- поддержкой массивов,
- созданием конструкторов копирования.

Допускается объявление объектов класса несколькими способами. При этом каждому способу объявления должна соответствовать своя версия конструктора. В противном случае возникают ошибки трансляции.

*Пример 18.5.* Перегрузим конструктор для случая инициализации объекта явно указанным значением и значением, заданным по умолчанию.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
```

```
using namespace std;
class double_init
{
    int x;
    int y;
public:
    double_init() { x = 0; y = 0; }
    double_init(int m, int n) { x = m; y = n; }
    void show_x_y() { cout << x << ' ' << y << endl; }
};
int _tmain(int argc, _TCHAR* argv[])
{ double_init ob1(10, 100); // объявление с явными значениями
double_init ob2;          // объявление без явного задания значений
ob1.show_x_y();
ob2.show_x_y();
_getch();
return 0;
}
```

*Пример 18.6.* Перегрузка конструктора для инициализации одиночного объекта и массива объектов.

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <conio.h>
using namespace std;
class MyCls { int x;
public:
    MyCls() { x = 0; }
    MyCls(int n) { x = n; }
    int get_x() { return x; }
};
int _tmain(int argc, _TCHAR* argv[])
// объявление объекта без явного задания значений
MyCls ob1;
// объявление объекта с явным значением
MyCls ob2(10);
// объявление массива без явного задания значений
MyCls ob3[10];
// объявление массива с явными значениями
MyCls ob4[10] = {1,2,3,4,5,6,7,8,9,10};
cout << "Объект ob1: " << endl;
cout << ob1.get_x() << endl;
cout << "Объект ob2: " << endl;
cout << ob2.get_x() << endl;
cout << "Массив ob3: " << endl;
for (int i = 0; i < 10; i++) cout << setw(3) << ob3[i].get_x();
cout << endl;
cout << "Массив ob4: " << endl;
for (int i = 0; i < 10; i++) cout << setw(3) << ob4[i].get_x();
cout << endl;
```

```
_getch();
return 0;
}
```

**Пример 18.7.** Перегрузка конструктора для создания динамического массива.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <iomanip>
using namespace std;
class MyCls { int x;
public:
    MyCls() { x = 0; }
    MyCls(int n) { x = n; }
    int get_x() { return x; }
};

int _tmain(int argc, _TCHAR* argv[])
{ MyCls ob(12525);
    MyCls *obPtr; // объявление указателя
    // объявление массива с явными значениями:
    MyCls ob_arr[10] = {1,2,3,4,5,6,7,8,9,10};
    cout << "Массив об_arr: " << endl;
    for (int i = 0; i < 10; i++) cout << setw(3) << ob_arr[i].get_x();
    cout << endl;
    obPtr = new MyCls[10]; // динамические массивы не инициализируются!
    if (!obPtr) { cout << "Ошибка выделения памяти" << endl;
        return 1; }
    cout << "Массив обPtr: " << endl;
    for (int i = 0; i < 10; i++) obPtr[i] = ob;
    for (int i = 0; i < 10; i++) cout << setw(3) << obPtr[i].get_x();
    cout << endl;
    getch();
    return 0;
}
```

Инициализация динамических массивов невозможна. Поэтому если в классе есть конструктор с инициализацией, то необходимо перегрузить конструктор под версию без инициализации. В противном случае возникает ошибка трансляции.

## 18.6. Присваивание объектов

Один объект можно присвоить другому объекту, если оба они одинакового типа. По умолчанию, когда объект A присваивается объекту B, происходит побитовое копирование всех данных-элементов A в данные-элементы B. Объекты A и B при этом приобретают одинаковые значения данных-элементов, оставаясь совершенно независимыми.

Присваивать можно лишь объекты одного типа (т. е. с одним именем типа), а не типов, одинаковых физически.

### Пример 18.8

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class MyClass { int a, b;
public:
    void set(int i, int j) { a = i; b = j; }
    void show() { cout << a << ' ' << b << endl; }
};

int _tmain(int argc, _TCHAR* argv[])
{ MyClass ob1, ob2;
    ob1.set(10, 5);
    ob2 = ob1; // все правильно
    ob1.show();
    ob2.show();
    getch();
    return 0;
}
```

### Пример 18.9

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class MyClass { int a, b;
public:
    void set(int i, int j) { a = i; b = j; }
    void show() { cout << a << ' ' << b << endl; }
};

class YourClass { int a, b;
public:
    void set(int i, int j) { a = i; b = j; }
    void show() { cout << a << ' ' << b << endl; }

int _tmain(int argc, _TCHAR* argv[])
{ MyClass ob1;
    YourClass ob2;
    ob1.set(10, 5);
    ob2 = ob1; // ошибка трансляции
    ob1.show();
    ob2.show();
    getch();
    return 0;
}
```

## 18.7. Конструктор копирования

Конструктор копирования (copy constructor) представляет собой одну из наиболее важных форм перегрузки конструктора, которая в состоянии решить одну из проблем, которая возникает при побитовом копировании объектов.

Конструктор копирования вызывается всякий раз, когда создается новый объект, и инициализируется существующим объектом такого же типа. При явном задании конструктора копирования этот процесс становится управляемым.

В C++ четко разделяются два типа ситуаций, при которых значение одного объекта присваивается другому:

- присваивание,
- инициализация.

Конструктор копирования используется только при инициализации. Конструктор копирования никогда не участвует в процессе присваивания и никак не влияет на него.

Инициализация имеет место в трех случаях:

- когда в операторе объявления один объект используется для инициализации другого;
- при передаче объекта в качестве параметра функции;
- при создании временного объекта для возврата значения из функции.

**Пример 18.10.** Пусть объявлен класс AnyClass и объект этого класса ob1. Тогда конструктор копирования может быть вызван следующими операторами:

```
AnyClass ob2 = ob1; // ob1 явно инициализирует ob2
AnyFn1(ob1);      // ob1 передается в качестве параметра
ob2 = AnyFn2();    // ob2 получает значение возвращаемого объекта
```

Конструктор копирования, заданный по умолчанию, выполняет по-элементное копирование нестатических элементов. При этом копирование выполняется по значению (побитовое копирование). В результате такого копирования при последующем вызове деструкторов могут возникнуть проблемы, обусловленные неопределенностью в момент освобождения памяти.

**Пример 18.11.** Пусть объявлен класс, одним из элементов которого является указатель int \*p. Тогда при присваивании объекта этого класса second = first;

наиболее критическим будет оператор присваивания, присутствующий в листинге программного кода:

```
second.p = first.p;
```

поскольку он приводит не к копированию массива, определяемого указателем p, в массив-копию, а к копированию указателя.

Проблема возникает в момент вызова операции

```
delete [] second.p;
```

удаляющей массив, на который указывает указатель p. После этого результат выполнения операции

```
delete [] first.p;
```

неопределен.

Для решения подобных проблем применяют явно заданные конструкторы копирования. В частности, вместо того, чтобы копировать адрес массива, конструктор копирования должен создавать еще один массив, присваивать его элементам значения исходного массива и присваивать элементу p нового объекта адрес массива-дубликата. В результате при каждом вызове деструктора освобождается другая строка, а не несколько раз одна и та же.

Синтаксис конструктора копирования имеет вид:

```
имя_класса (const имя_класса &obj)
{ тело_конструктора
}
```

где obj – ссылка на объект, используемый для инициализации другого объекта.

**Пример 18.12.** Создание конструктора копирования для объекта, содержащего в качестве одного из своих членов динамический массив.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <iostream>
using namespace std;
class array { int *Arr_Ptr;
             int size;
public:
array (int sz)      // конструктор
{ Arr_Ptr = new int[sz];
  if (!Arr_Ptr) exit (1);
  size = sz;
  cout << "Ordinary constructor" << endl;
}
array(const array &a); // прототип конструктора копирования:
~array()           // деструктор
{ delete [] Arr_Ptr; }
void put(int i, int j)
{ if (i >= 0 && i < size) Arr_Ptr[i] = j; }
int get(int i)
{ return Arr_Ptr[i]; }
```

```

};

array :: array(const array &a) // конструктор копирования
{ int i;
  Arr_Ptr = new int[a.size]; // выделение памяти для копии массива
  if (!Arr_Ptr) exit(1);
  // копирование содержимого:
  for (i = 0; i < a.size; i++) Arr_Ptr[i] = a.Arr_Ptr[i];
  cout << "Copy constructor" << endl;
}
int _tmain(int argc, _TCHAR* argv[])
{ array arr(10); // вызов обычного конструктора
  int i;
  for (i = 0; i < 10; i++) arr.put(i, i);
  for (i = 0; i < 10; i++) cout << arr.get(i);
  cout << endl;
  array new_arr = arr; // вызов конструктора копирования
  for (i = 0; i < 10; i++) cout << new_arr.get(i);
  _getch();
  return 0;
}

```

Следует помнить, что передача объекта в функцию по ссылке предотвращает вызов конструктора копирования. Это позволяет сберечь время, необходимое для его вызова и сэкономить память, используемую для хранения нового объекта.

### Контрольные вопросы

1. Что представляет собой класс?
2. Какие спецификации доступа используются при описании класса?
3. Что являются элементами класса?
4. Как осуществляется доступ к элементам класса?
5. Для чего используется указатель this?
6. Что такое конструктор?
7. Что такое деструктор?

### Практические задания 18

1. Определить класс-строку. В класс включить два конструктора: для определения класса строки строкой символов и путем копирования другой строки (объекта класса строки). Предусмотреть функции поиска слова в строке и добавления другой строки начиная с позиции N.
2. Определить класс-строку. В класс включить два конструктора: для определения класса строки строкой символов и путем копирования другой строки (объекта класса строки). Предусмотреть функции слияния двух строк и функцию подсчета предложений в строке.
3. Определить класс-строку. В класс включить два конструктора: для определения класса строки строкой символов и путем копирования другой строки (объекта класса строки). Предусмотреть функции сортировки слов в строке по алфавиту и подсчета количества слов.

4. Определить класс-вектор. В класс включить два конструктора для определения вектора по его размеру и путем копирования другого вектора. При задании вектора по его размеру предусмотреть его заполнение случайными числами. Предусмотреть функции умножения векторов и подсчета суммы элементов вектора.
5. Определить класс-вектор. В класс включить два конструктора для определения вектора по его размеру и путем копирования другого вектора. При задании вектора по его размеру предусмотреть его заполнение случайными числами. Предусмотреть функции нахождения максимального и минимального из элементов вектора и умножения вектора на число.
6. Определить класс-список элементов. В определение класса включить два конструктора: для определения списка по его размеру и путем копирования другого списка. Предусмотреть функции подсчета количества элементов списка и добавления одного списка в другой список начиная с позиции N.

# ГЛАВА 19. ФУНКЦИИ И ОБЪЕКТЫ

## 19.1. Указатели на объекты

Арифметика указателей на объекты аналогична арифметике указателей на любой другой тип данных: она выполняется относительно объекта.

**Пример 19.1.** Объявление и использование указателя на объект класса.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class AnyCls { int a, b;
public:
    AnyCls (int m, int n) { a = m; b = n; }
    int get_a() { return a; }
    int get_b() { return b; }
};
int _tmain(int argc, _TCHAR* argv[])
{ AnyCls ob[4] = { AnyCls(1, 2), AnyCls(3, 4), AnyCls(5, 6), AnyCls(7, 8) };
int i;
AnyCls *Ptr;           // объявление указателя на объект класса
Ptr = ob;              // адрес начала массива объектов
for (i = 0; i < 4; i++)
{ cout << Ptr -> get_a() << ' ' << Ptr -> get_b() << endl;
  Ptr++;
}                      // переход к следующему объекту
cout << endl;
_getch();
return 0;
}
```

C++ содержит специальный указатель `this`, который автоматически передается любой функции-элементу класса при ее вызове и указывает на объект, делающий вызов. Указатель `this` передается только функциям-элементам и никогда не передается дружественным функциям. Если функция-элемент работает с другими элементами этого же класса, ей не требуется уточнение имени класса благодаря тому, что при вызове функции-элемента ей автоматически передается указатель `this` на тот объект, который является источником вызова.

Два варианта написания программного кода, представленные в примере 19.2, полностью эквивалентны, поэтому никто не пользуется вторым (более длинным) способом записи.

Необходимость явного применения указателя `this` возникает при перегрузке операций.

### Пример 19.2

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <string>
using namespace std;
class PhotoModel
{ char name[10];           // имя
  char c_hair[10];          // цвет волос
  char c_eyes[10];          // цвет глаз
  double weight;            // вес
  int height;               // рост
public:
  PhotoModel(char *n,char *c_h,char *c_y, double w, int h)
  { strcpy(name, n);         //strcpy(this->name, n);
    strcpy(c_hair, c_h);     //strcpy(this->c_hair,c_h);
    strcpy(c_eyes, c_y);     //strcpy(this->c_eyes,c_y);
    weight = w;              //this->weight = w;
    height = h;              //this->height = h;
  }
  void show()
  { cout << name << ''
    << c_hair << ''
    << c_eyes << ''
    << weight << ''
    << height
    << endl;
  }
  void show()
  { cout << this->name
    << '' << this->c_hair
    << '' << this->c_eyes
    << '' << this->weight
    << '' << this->height
    << endl;
  }
}
int _tmain (int argc, _TCHAR* argv[])
{ PhotoModel Olga("Olga", "black", "black", 56.5, 170);
  Olga.show();
  _getch();
  return 0;
}
```

## 19.2. Передача объектов в функции

### 19.2.1. Передача объектов по значению

Синтаксически объекты передаются в функции так же, как и переменные других типов. Для этого объявляют параметр функции, который имеет тип класса. Затем используют объект этого класса в качестве аргумента при вызове функции. По умолчанию происходит передача объектов по значению. При этом в стеке создается копия аргумента, которая используется при работе функции.

**Пример 19.3.** Передача объекта в функцию по значению.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <math.h>
using namespace std;
class circle
{
    int x, y, r;
public:
    circle(int kx, int ky, int rad)
    { x = kx; y = ky; r = rad; }
    void get_c(int &kx, int &ky, int &rad)
    { kx = x; ky = y; rad = r; }
    double get_square()
    { return 3.14159 * r * r; }
};
// функция point_in() определяет, принадлежит ли
// точка с координатами (xp, yp) кругу
// с координатами центра и радиусом, определенными
// в объекте класса circle
int point_in(circle ob, int xp, int yp)
{
    int res, r_x, r_y, r_r;
    ob.get_c(r_x, r_y, r_r);
    if (pow(double(xp - r_x), 2) +
        pow(double(yp - r_y), 2) <
        pow(double(r_r), 2)) res = 1;
    else res = 0;
    return res;
}
int _tmain(int argc, _TCHAR* argv[])
{
    setlocale(LC_ALL, "Russian");
    int result, xt = 7, yt = 10;
    circle c1(5, 2, 17), c2(8, 15, 3);
    cout << c1.get_square() << endl;
    cout << c2.get_square() << endl;
    result = point_in(c1, xt, yt);
    if (result == 1) cout << "Принадлежит" << endl;
    else cout << "Не принадлежит" << endl;
    result = point_in(c2, xt, yt);
    if (result == 1) cout << "Принадлежит" << endl;
    else cout << "Не принадлежит" << endl;
    getch();
    return 0;
}
```

#### 19.2.2. Передача объектов по указателю

При передаче объекта в функцию по указателю происходит передача его адреса, а не значения. При этом функция может изменять значение аргумента, используемого в вызове.

**Пример 19.4.** Передача объекта в функцию по указателю.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <math.h>
using namespace std;
class my_cls { int x;
public:
    my_cls(int n) { x = n; }
    void set_x(int n) { x = n; }
    int get_x() { return x; }
};
void cube_obj(my_cls *object)
{ object->set_x(pow(double(object->get_x()), 3));
    cout << object->get_x() << endl; // 1000
}
int _tmain(int argc, _TCHAR* argv[])
{ my_cls ob(10);
    cube_obj(&ob);
    cout << ob.get_x() << endl; // 1000
    getch();
    return 0;
}
```

При передаче объекта в функцию по значению создается копия этого объекта. При завершении работы функции созданная копия объекта удаляется. При вызове функции, в момент создания копии объекта, конструктор не вызывается, так как он используется для инициализации элементов объекта, а копия создается для уже существующего (а значит – проинициализированного) объекта.

Но при завершении работы функции, т. е. при удалении копии объекта, вызывается деструктор, так как иначе могут оказаться невыполнеными некоторые необходимые действия, связанные с его удалением, например освобождение памяти.

Здесь возникает проблема освобождения динамической памяти (рис. 19.1).

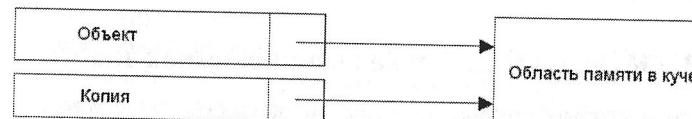


Рис. 19.1. Результат создания побитовой копии объекта

Поскольку объект и его копия, если они содержат указатели, обращаются к одной и той же области памяти, то все изменения, производимые с этой памятью копией, будут влиять на объект-оригинал. Кроме то-

го, освобождение памяти деструктором копии приводит к тому, что указатель объекта может стать неопределенным.

Одним из способов обойти эту проблему также является передача объектов в функцию по указателю или по ссылке.

### 19.2.3. Передача объектов по ссылке

Ссылка является скрытым указателем и всегда работает просто как другое имя переменной. При передаче параметра по ссылке изменения объекта внутри функции влияют на исходный объект и сохраняются при завершении работы функции.

Ссылка не является аналогом указателя, поэтому при передаче объекта по ссылке для доступа к его элементам используется операция «точка» (.), а не «стрелка» (->).

**Пример 19.5.** Передача объекта в функцию по ссылке.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <math.h>
using namespace std;
class my_cls { int x;
public:
    my_cls(int n) { x = n; }
    void set_x(int n) { x = n; }
    int get_x() { return x; }
};
void cube_obj(my_cls &object)
{ object.set_x(pow(double(object.get_x()), 3));
cout << object.get_x() << endl; // 1000
}
int _tmain(int argc, _TCHAR* argv[])
{ my_cls ob(10);
cube_obj(ob);
cout << ob.get_x() << endl; // 1000
_getch();
return 0;
}
```

### 19.3. Объекты в качестве возвращаемых значений

Для того чтобы функция могла возвращать объект, нужно объявить ее так, чтобы ее возвращаемое значение имело тип класса, а затем поставить объект этого типа в операторе return.

**Пример 19.6.** Объявление класса, закрытыми элементами которого являются часы и минуты. Будем прибавлять к минутам и часам некоторое значение и возвращать полученный результат.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <math.h>
using namespace std;
class time { int hour;
int minute;
public:
time() { hour = 0; minute = 0; }
time(int h, int m) { hour = h; minute = m; }
void showtime() { cout << hour << ":" << minute << endl; }
time add_time(time);
time time :: add_time(time t2)
{ time tmp;
tmp.minute = minute + t2.minute;
if (tmp.minute >= 60)
{ tmp.minute -= 60;
tmp.hour = 1 + hour + t2.hour;
}
else tmp.hour = hour + t2.hour;
return tmp;
}
int _tmain(int argc, _TCHAR* argv[])
{ time t1(12, 15), t2(17, 50), t3;
t3 = t1.add_time(t2);
t1.showtime();
t2.showtime();
t3.showtime();
_getch();
return 0;
}
```

Когда функция возвращает объект, для его хранения автоматически создается временный объект. После того как значение возвращено, этот временный объект удаляется. Если возвращаемый объект содержит указатели, то удаление его временной копии может привести к неожиданным и печальным последствиям, связанным с особенностями вызова конструктора и деструктора.

Функция может возвращать ссылку, но для закрытых элементов класса реализовать это можно единственным способом: организовать возврат ссылки на закрытый элемент класса из открытой функции-элемента.

Но ссылка на объект является его псевдонимом и, следовательно, может быть использована с левой стороны оператора присваивания. Поэтому не следует возвращать из открытой функции-элемента неконстантную ссылку (или указатель) на закрытый элемент данных. Возвращение такой ссылки нарушает инкапсуляцию класса.

## 19.4. Дружественные функции

Дружественные функции, не являясь элементами класса, имеют доступ к его закрытым и защищенным элементам. Для того чтобы объявить функцию дружественной некоторому классу, нужно в этот класс включить ее прототип, перед которым поставить ключевое слово `friend`.

Дружественная функция может быть членом другого класса, или не принадлежать ни к какому классу.

Синтаксис объявления дружественных функций:

```
class имя_класса
{ friend тип имя_другого_класса :: имя_функции(список_параметров);
  friend тип имя_функции(список_параметров);
}
```

### Пример 19.7

```
class MyCls
{ friend void AnotherClass :: MemberFnName(int, float);
  friend void RegularFnName(char);
}
```

Можно объявить целый класс дружественным данному классу, включив в определение этого класса описание другого класса с ключевым словом `friend`. Тогда всем элементам дружественного класса будет разрешен доступ ко всем элементам того класса, для которого он объявлен дружественным:

```
class MyCls
{ ...
  friend AnotherClass;
}
```

### Пример 19.8. Использование дружественной функции.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <math.h>
using namespace std;
class MyCls
{ int div, val;
public:
  MyCls(int d, int v) { div = d; val = v; }
  friend int Divisibility(MyCls ob);
};
int Divisibility(MyCls ob)
{ if ( !(ob.div % ob.val) ) return 1;
  else return 0;
}
int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian"); // подключение русификатора
```

```
MyCls ob1(10, 5), ob2(13, 7);
if (Divisibility(ob1)) cout << " ob1.div кратно ob1.val" << endl;
else cout << " ob1.div не кратно ob1.val" << endl;
if (Divisibility(ob2)) cout << " ob2.div кратно ob2.val" << endl;
else cout << " ob2.div не кратно ob2.val" << endl;
_getch();
return 0;
}
```

Так как дружественная функция не является членом того класса, для которого она дружественна, то ее нельзя вызвать, используя имя объекта.

Доступ дружественной функции к закрытым элементам класса можно осуществить только через объект класса, поэтому в отличие от функции-элемента, в которой можно упоминать закрытые элементы класса, не указывая имя объекта, дружественная функция может иметь доступ к этим переменным только в связи с объектом, который объявлен внутри функции или передан ей.

**Пример 19.9.** Объявим два класса, описывающие пассажирский и грузовой самолеты, и сравним их скорость и дальность полета.

```
#include "stdafx.h"
#include <iostream>
#include <math.h>
#include <conio.h>
using namespace std;
class aero_w; // ссылка вперед
class aero_p
{ int passengers;
  int dist_of_flight;
  int speed;
public:
  aero_p(int p, int d, int s)
  { passengers = p;
    dist_of_flight = d;
    speed = s;
  }
  friend int comp_char(aero_p a_p, aero_w a_w, int &charact);
};
class aero_w
{ int weight;
  int dist_of_flight;
  int speed;
public:
  aero_w(int w, int d, int s)
  { weight = w;
    dist_of_flight = d;
    speed = s;
  }
}
```

```

friend int comp_char(aero_p a_p, aero_w a_w, int &charact);
};

// программа сравнения характеристик самолетов
int comp_char(aero_p a_p, aero_w a_w, int &charact)
{ int result;
    switch (charact)
    { case 1: result = a_p.dist_of_flight - a_w.dist_of_flight;
        break;
    case 2: result = a_p.speed - a_w.speed;
        break;
    }
    return result;
}

int _tmain(int argc, _TCHAR* argv[])
{setlocale(LC_ALL, "Russian");
int res, p_p, p_d, p_s, w_w, w_d, w_s, condition;
cout << "Характеристики пассажирского самолета:";
cin >> p_p >> p_d >> p_s;
cout << "Характеристики грузового самолета: ";
cin >> w_w >> w_d >> w_s;
aero_p pas(p_p, p_d, p_s);
aero_w wei(w_w, w_d, w_s);
condition = 2;
res = comp_char(pas, wei, condition);
if (res > 0) cout << "Пассажирский летает быстрее";
else if (res < 0) cout << "Грузовой летает быстрее ";
else cout << "Скорости одинаковы";
_getch();
return 0;
}

```

К друзьям применимы следующие положения:

- на описания со спецификатором `friend` не распространяется действие спецификаторов `public`, `protected`, `private`;
- описания `friend` невзаимны: т. е. если А объявляет другом В, из этого не следует, что А является другом В;
- дружественность не наследуется, т. е. если В является другом А, классы, производные от В, не будут автоматически иметь доступ к закрытым элементам А;
- дружественность не является переходным свойством, т. е. если А объявляет другом В, классы, производные от А, не будут автоматически признавать дружественность В;
- дружественность не обладает свойством транзитивности, т. е. если А объявляет другом В, а В объявляет другом С, то из этого не следует, что А является другом С.

### Контрольные вопросы

1. Как может осуществляться передача объектов в функции?
2. Что такое указатель `this`? Для чего он используется?
3. Как организовать возвращение объекта из функции?
4. Что такое дружественные функции, дружественные классы?
5. Перечислите основные свойства дружественных функций и дружественных классов.
6. Когда используются ссылки вперед?

## ГЛАВА 20. ПЕРЕГРУЗКА ОПЕРАТОРОВ

Перегрузка операторов, в отличие от перегрузки функций, всегда связана с классом. При перегрузке оператора из его исходного назначения ничего не теряется, но он дополнительно приобретает новые свойства.

Для перегрузки оператора задается оператор-функция (или функция-операция), которая является либо элементом класса, либо дружественной классу, для которого она задана:

```
тип имя_класса :: operator #(список_аргументов)
{ выполняемые действия
}
```

Здесь вместо знака # ставится знак перегружаемого оператора.

Можно перегрузить операторы: +, -, \*, /, %, ^, &, |, ~, !, !=, ==, <, >, <=, >=, <<, >>, &&, ||, ++, --, +=, -=, \*=, /=, %=, ^=, &=, |=, <<=, >>=, ->, ->\*, ,, [], (), new, delete.

Существует несколько жестких ограничений на перегрузку операторов:

- нельзя перегружать операторы: ., .\*, ::, ?:, sizeof и операторы препроцессора;
- нельзя менять приоритет операторов;
- нельзя менять число operandов оператора относительно его первоначального состояния;
- нельзя создавать новые операции, можно только перегружать уже существующие;
- оператор-функции не могут иметь параметров, передаваемых по умолчанию.

Функции-операции могут быть одноместными (унарными) и двуместными (бинарными), при этом число operandов, приоритет и ассоциативность операции, определенной функцией-операцией, остаются такими же, как и в базовом языке. Операции, которые имеют унарный и бинарный варианты, нужно перегружать отдельно.

Порядок выполнения операций определяется их приоритетом. Но если приоритет операций одинаков, порядок их выполнения определяется их ассоциативностью. Ассоциативность «слева направо» означает, что из двух операций, имеющих одинаковый приоритет, первой выполняется та, что находится слева. При ассоциативности «справа налево» первой выполняется операция, находящаяся справа.

### 20.1. Перегрузка унарных операторов

Унарные операторы имеют один operand, поэтому унарная операция класса перегружается как нестатическая функция-элемент без аргументов

либо как дружественная функция с одним аргументом. Этот аргумент должен быть либо объектом класса, либо ссылкой на объект класса.

Функция-элемент может быть объявлена как static, если она не должна иметь доступ к нестатическим элементам класса. В отличие от нестатических функций-элементов статическая функция-элемент не имеет указателя this, так как статические данные-элементы и статические функции-элементы существуют независимо от конкретных объектов класса.

Обычно каждый объект класса имеет свою собственную копию всех данных элементов класса. Но в определенных случаях во всех объектах класса должна фигурировать только одна копия некоторых данных-элементов для всех объектов класса. Для этих целей используют статические данные, которые содержат информацию «для всего класса». Их объявление также начинается со слова static.

**Пример 20.1.** Перегрузка знака <-> в унарном варианте.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class twins { int t1, t2;
public:
    twins() { t1 = 0; t2 = 0; }
    twins(int a, int b) { t1 = a; t2 = b; }
    twins &operator -();
    void show_tw() { cout << t1 << " " << t2 << endl; }
};
twins &twins :: operator -()
{
    t1 = -t1;
    t2 = -t2;
    return *this;
}
int _tmain(int argc, _TCHAR* argv[])
{
    twins ob1, ob2(17, -19);
    ob1 = -ob2;
    ob1.show_tw();
    ob2.show_tw();
    getch();
    return 0;
}
```

**Пример 20.2.** Перегрузка оператора инкремента (++) в префиксной и постфиксной форме записи.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class twins
{ int t1, t2;
```

```

public:
twins() { t1 = 0; t2 = 0; }
twins(int a, int b) { t1 = a; t2 = b; }
void get_tw(int &a, int &b) { a = t1; b = t2; }
twins& operator ++();
twins& operator ++(int);
};
twins &twins :: operator ++()
{ t1++;
t2++;
return *this;
}
twins &twins :: operator ++(int)
{ ++t1;
++t2;
return *this;
}
int _tmain(int argc, _TCHAR* argv[])
{ twins ob, ob1(10, 15), ob2(17, 19);
int m, k;
ob = ob1++;
ob1.get_tw(m, k);
cout << m << ' ' << k << endl;// 11 16
ob.get_tw(m, k);
cout << m << ' ' << k << endl;// 11 16
ob = ++ob2;
ob2.get_tw(m, k);
cout << m << ' ' << k << endl;// 18 20
ob.get_tw(m, k);
cout << m << ' ' << k << endl;// 18 20
_getch();
return 0;
}

```

## 20.2. Перегрузка бинарных операторов

Бинарная операция класса перегружается как нестатическая функция-элемент с одним аргументом либо как дружественная функция с двумя аргументами. Один из этих аргументов должен быть либо объектом класса, либо ссылкой на объект класса.

**Пример 20.3.** Перегрузка операторов «+», «-» и «=».

```

#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class twins
{ int t1, t2;
public:

```

```

twins() { t1 = 0; t2 = 0; }
twins(int a, int b) { t1 = a; t2 = b; }
void get_tw(int &a, int &b) { a = t1; b = t2; }
twins operator +(twins o2);
twins operator -(twins o2);
twins& operator =(twins o2);
};
twins twins :: operator +(twins o2)
{ twins temp;
temp.t1 = t1 + o2.t1;
temp.t2 = t2 + o2.t2;
return temp;
}
twins twins :: operator -(twins o2)
{ twins temp;
temp.t1 = t1 - o2.t1;
temp.t2 = t2 - o2.t2;
return temp;
}
twins &twins :: operator =(twins o2)
{ twins temp;
t1 = o2.t1;
t2 = o2.t2;
return *this;
}
int _tmain(int argc, _TCHAR* argv[])
{ twins ob, ob1(10, 15), ob2(17, 19), ob3;
int m, k;
ob3 = ob1 + ob2;
ob3.get_tw(m, k);
cout << m << ' ' << k << endl;
ob3 = ob1 - ob2;
ob3.get_tw(m, k);
cout << m << ' ' << k << endl;
ob3 = ob1;
ob3.get_tw(m, k);
cout << m << ' ' << k << endl;
_getch();
return 0;
}

```

При перегрузке операторов «+» и «-» внутри перегружаемого оператора используется временный объект `temp`, который служит возвращаемым значением. Возвращение функциями `operator +()` и `operator -()` объекта, в данном случае типа `twins`, позволяет использовать результаты сложения и вычитания объектов в сложных выражениях, например:

`ob5 = ob1 + ob2 - ob3 + ob4;`

При создании функции `operator +()` порядок следования operandов не имеет значения, но для правильной перегрузки операции вычитания не-

обходимо учитывать порядок следования операндов. Следует помнить, что при перегрузке бинарного оператора левый операнд передается неявно, а правый выступает в качестве аргумента функции.

Оператор присваивания имеет две особенности:

- левый операнд (т. е. объект, которому присваивается значение) в теле оператора изменяется;
- функция возвращает `*this`. Это происходит потому, что функция `operator =()` возвращает объект, которому присваивается значение. Возвращение `*this` позволяет делать последовательности присваиваний объектов аналогично последовательности присваиваний обычных переменных: `ob1 = ob2 = ob3;`.

Можно перегрузить связанный с классом оператор таким образом, что правый операнд (передаваемый явно) будет переменной одного из стандартных типов, а левый операнд (передаваемый по умолчанию) – объектом класса.

**Пример 20.4.** Перегрузка операторов «+» для случая, когда оба операнда – объекты класса и когда правый операнд имеет тип `int`.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class twins
{ int t1, t2;
public:
    twins() { t1 = 0; t2 = 0; }
    twins(int a, int b) { t1 = a; t2 = b; }
    void get_tw(int &a, int &b) { a = t1; b = t2; }
    twins operator +(twins o2);
    twins operator +(int i);
};
twins twins :: operator +(twins o2)
{ twins temp;
    temp.t1 = t1 + o2.t1;
    temp.t2 = t2 + o2.t2;
    return temp;
}
twins twins :: operator +(int i)
{ twins temp;
    temp.t1 = t1 + i;
    temp.t2 = t2 + i;
    return temp;
}
int _tmain(int argc, _TCHAR* argv[])
{ twins ob1(10, 15), ob2(17, 19), ob3;
    int m, k;
    ob3 = ob1 + ob2;
```

```
    ob3.get_tw(m, k);
    cout << m << ' ' << k << endl;
    ob3 = ob1 + 100;
    ob3.get_tw(m, k);
    cout << m << ' ' << k << endl;
    getch();
    return 0;
}
```

При перегрузке оператор-функции для случая использования переменной стандартного типа нужно следить за тем, чтобы переменная встроенного типа находилась справа от знака операции, так как запись вида `ob3 = 100 + ob1;` вызывает ошибку трансляции.

### Использование параметров-ссылок при перегрузке операторов

В оператор-функции можно использовать параметр-ссылку.

**Пример 20.5.** Перегрузка оператора «+» для класса `twins` с использованием ссылки.

```
twins twins :: operator +(twins &o2)
{ twins temp;
    temp.t1 = t1 + o2.t1;
    temp.t2 = t2 + o2.t2;
    return temp;
}
```

Ссылка, используемая в качестве аргумента, улучшает характеристики программного кода, обеспечивая более высокое быстродействие и более эффективное использование памяти, чем при передаче по значению.

Однако, возвращаемое значение не может быть ссылкой, так как эта ссылка в данном случае указывала бы на временную переменную `temp`. Но эта переменная уничтожается в момент завершения функции, поэтому мы получили бы ссылку, которая указывает на несуществующий объект.

Никогда не следует создавать ссылки (и указатели) на локальные переменные и другие временные объекты.

### 20.3. Перегрузка операторов отношения и логических операторов

При перегрузке операторов отношения и логических операторов происходит возвращение не объектов класса, а переменной целого типа, интерпретируемой как `true` или `false`.

**Пример 20.6.** Перегрузка операторов `==`, `&&` и `||`.

```
#include "stdafx.h"
#include <iostream>
```

```
#include <conio.h>
using namespace std;
class twins
{ int t1, t2;
public:
    twins() { t1 = 0; t2 = 0; }
    twins(int a, int b) { t1 = a; t2 = b; }
    void get_tw(int &a, int &b) { a = t1; b = t2; }
    int operator ==(twins o2);
    int operator &&(twins o2);
    int operator ||(twins o2);
};
int twins :: operator ==(twins o2)
{ if ((t1 == o2.t1) && (t2 == o2.t2)) return 1;
  else return 0;
}
int twins :: operator &&(twins o2) { return ((t1 && o2.t1) && (t2 && o2.t2)); }
int twins :: operator ||(twins o2) { return ((t1 || o2.t1) && (t2 || o2.t2)); }

int _tmain(int argc, _TCHAR* argv[])
{ twins ob1(10, 15), ob2(17, 19), ob3(0, 0),
  ob4(10, 15), ob5(0, 0);
  if (ob1 == ob2) cout << "Objects are equal" << endl;
  else cout << "Objects are not equal" << endl;
  if (ob1 == ob4) cout << "Objects are equal" << endl;
  else cout << "Objects are not equal" << endl;
  if (ob1 && ob2) cout << "True" << endl;
  else cout << "False" << endl;
  if (ob1 && ob3) cout << "True" << endl;
  else cout << "False" << endl;
  if (ob1 || ob2) cout << "True" << endl;
  else cout << "False" << endl;
  if (ob5 || ob3) cout << "True" << endl;
  else cout << "False" << endl;
  getch();
  return 0;
}
```

## 20.4. Перегрузка оператора индексирования

Оператор индексирования массива можно перегрузить для реализации доступа к данным-элементам класса, даже если эти данные не образуют массив.

**Пример 20.7.** Пусть закрытыми данными-элементами класса являются несколько отдельных переменных, имеющих одинаковый тип, но не объединенных в массив. Перегрузим оператор индексирования так, чтобы можно было организовать к ним доступ аналогично доступу к элементам массива.

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class NotArr
{ int a0;
  int a1;
  int a2;
  int a3;
  int a4;
public:
  NotArr (int v0, int v1, int v2, int v3, int v4)
  {a0 = v0; a1 = v1; a2 = v2; a3 = v3; a4 = v4; }
  int get_a(unsigned i);
  int operator [](unsigned i);
};
int _tmain(int argc, _TCHAR* argv[])
{ NotArr na(1, 2, 3, 4, 5);
  for (int ind = 0; ind < 5; ind++) cout << na[ind] << ' ';
  cout << endl;
  getch();
  return 0;
}
int NotArr :: get_a(unsigned i)
{ switch (i)
  { case 0: return a0; // оператор break здесь не нужен, так как выход // происходит по оператору return и до break дело просто не дойдет
    case 1: return a1;
    case 2: return a2;
    case 3: return a3;
    case 4: return a4;
    default: return a0;
  }
}
int NotArr :: operator[](unsigned i)
{ return get_a(i); }
```

Индексы перегруженных массивов не обязательно должны быть типа int. Например, вполне корректной является перегружаемая функция-оператор `int operator[](char c);` с использованием символьных индексов. Это – существенное отличие от классического программирования на языках С и С++. Фактически перегрузка [ ] с использованием символов в качестве индексов создает так называемый хеш или ассоциативный массив.

## 20.5. Перегрузка оператора присваивания

Можно перегрузить оператор присваивания для случая, когда нежелательно побитовое копирование.

**Пример 20.8:**

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <string.h>
using namespace std;
class MyStr
{ char *sPtr;
  int length;
public:
  MyStr(char *s);
  ~MyStr() { cout << "Освобождаем память" << endl;
             delete []sPtr;
           }
  char *get() { return sPtr; }
  MyStr &operator = (MyStr &ob);
};
MyStr:: MyStr(char *s)
{ int len;
  len = strlen(s);
  sPtr = new char[len + 1];
  if (!sPtr)
  { cout << "Ошибка выделения памяти" << endl;
    exit(1);
  }
  length = len;
  strcpy(sPtr, s);
}
MyStr &MyStr:: operator = (MyStr &ob)
{if (length < ob.length) // необходима дополнительная память
 { delete []sPtr;
   sPtr = new char[ob.length + 1];
   if (!sPtr) { cout << "Ошибка выделения памяти" << endl;
                exit(1); }
 }
length = ob.length;
strcpy(sPtr, ob.sPtr);
return *this;
}
int _tmain(int argc, _TCHAR* argv[])
{ MyStr a("Hallo"), b("Good morning");
  cout << a.get() << endl;
  cout << b.get() << endl;
  a = b;
  cout << a.get() << endl;
  cout << b.get() << endl;
  getch();
  return 0;
}
```

Перегрузка оператора присваивания предотвращает побитовое копирование sPtr. В процессе работы программы определяется, достаточно ли памяти в объекте, находящемся слева от оператора присваивания, для размещения присваиваемой строки. Если недостаточно, то память освобождается и выделяется новый фрагмент. Затем в эту память копируется строка, а ее длина передается закрытому элементу length.

**20.6. Перегрузка операторов управления памятью****20.6.1. Перегрузка оператора new**

Оператор new перегружается, как и любой другой оператор, но ответственность за поддержку запросов выделения памяти для объектов этого класса ложится на программиста.

Для перегрузки new используют прототип функции

```
void *operator new(size_t size);
```

После этого обращения к оператору new перенаправляются перегруженной функции, которая должна возвращать адрес области памяти, выделенной объекту, или NULL при отказе выделения памяти.

**Пример 20.9. Перегрузка оператора new.**

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <string.h>
using namespace std;
class OverNew
{ int x;
public:
  OverNew();
  void *operator new(size_t size);
};
char buf[512]; // память выделяется не в куче, а в глобальном буфере.
// Можно выделять память, например, на диске.
int i;
int _tmain(int argc, _TCHAR* argv[])
{ cout << "Создание объекта в куче" << endl;
  OverNew ob1;
  cout << "Резервирование памяти с помощью new" << endl;
  OverNew *ob2 = new OverNew;
  OverNew *ob3 = new OverNew;
  OverNew *ob4 = new OverNew;
  getch();
  return 0;
}
OverNew :: OverNew()
{ cout << "Constructor" << endl;
  x = i;
```

```

}
void *OverNew :: operator new(size_t size)
{ cout << "Overload new: size = " << size << endl;
  if (i >= 512 - sizeof(OverNew)) return 0;
  else { int k = i;
    i += sizeof(OverNew);
    return &buf[k];
  }
}

```

### 20.6.2. Перегрузка оператора delete

Прототип функции перегрузки оператора delete имеет вид:

```
void operator delete(void *ptr);
```

где ptr ссылается на удаляемый объект.

Для передачи функции числа байтов в удаляемом объекте прототип функции перегрузки оператора delete приобретает такой вид:

```
void operator delete(void *ptr, size_t size);
```

**Пример 20.10.** Перегрузка оператора delete.

```

#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class OverNew
{ int x;
public:
  OverNew();
  void *operator new(size_t size);
  void operator delete(void *ptr);
};

char buf[512]; // память выделяется не в куче, а в глобальном буфере
int _tmain(int argc, _TCHAR* argv[])
{ cout << "Создание объекта в куче" << endl;
  OverNew ob1;
  cout << "Захват памяти с помощью new" << endl;
  OverNew *ob2 = new OverNew;
  OverNew *ob3 = new OverNew;
  OverNew *ob4 = new OverNew;
  delete ob2;
  delete ob3;
  delete ob4;
  getch();
  return 0;
}
OverNew :: OverNew()
{ cout << "Constructor" << endl;
  x = i;
}

```

```

}
void *OverNew :: operator new(size_t size)
{ cout << "Overload new: size = " << size << endl;
  if (i >= 512 - sizeof(OverNew)) return 0;
  else { int k = i;
    i += sizeof(OverNew);
    return &buf[k];
  }
}
void OverNew :: operator delete(void *ptr)
{ cout << "Deleting of object" << ptr << endl; }

```

В данном случае реального освобождения памяти не происходит, поскольку удаляемые объекты находятся не в куче.

Чтобы воспользоваться операторами new и delete, работающими с кучей и игнорирующими перегрузку, нужно поставить перед ними двойное двоеточие:

```
OverNew *p = ::new OverNew;
::delete p;
```

### 20.7. Использование дружественных оператор-функций

При перегрузке операторов с использованием дружественных оператор-функций передача указателя this невозможна. Поэтому при перегрузке бинарного оператора дружественной функции явно передаются два операнда, в случае перегрузки унарного оператора – один операнд. Перегрузка с использованием дружественных оператор-функций аналогична перегрузке с помощью функций-элементов, за исключением двух моментов:

- нельзя использовать дружественную функцию для перегрузки оператора присваивания;
- используя дружественную оператор-функцию, в операциях с объектами можно использовать встроенные типы данных, и при этом встроенный тип может располагаться слева от оператора.

**Пример 20.11.** Перегрузка операторов «+» для случая использования дружественной оператор-функции.

```

#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class twins
{ int t1, t2;
public:
  twins() { t1 = 0; t2 = 0; }
  twins(int a, int b) { t1 = a; t2 = b; }
  void get_tw(int &a, int &b) { a = t1; b = t2; }
  friend twins operator +(twins o1, twins o2);
  friend twins operator +(twins o1, int i);
}

```

```

friend twins operator +( int i, twins o1);
};

twins operator +( twins o1, twins o2)
{ twins temp;
temp.t1 = o1.t1 + o2.t1;
temp.t2 = o1.t2 + o2.t2;
return temp;
}

twins operator +( twins o1, int i)
{ twins temp;
temp.t1 = o1.t1 + i;
temp.t2 = o1.t2 + i;
return temp;
}

twins operator +( int i, twins o1)
{ twins temp;
temp.t1 = o1.t1 + i; temp.t2 = o1.t2 + i;
return temp;
}

int _tmain(int argc, _TCHAR* argv[])
{ twins ob1(10, 15), ob2(17, 19), ob3, ob4;
int m, k;
ob3 = ob1 + ob2;
ob3.get_tw(m, k);
cout << m << ' ' << k << endl;
ob3 = ob1 + 100;
ob3.get_tw(m, k);
cout << m << ' ' << k << endl;
ob4 = 100 + ob2;
ob4.get_tw(m, k);
cout << m << ' ' << k << endl;
_getch();
return 0;
}

```

При использовании дружественной оператор-функции для перегрузки унарного оператора «`++`» или «`--`» операнд передается в функцию как параметр-ссылка, поскольку в дружественную оператор-функцию не передается указатель `this`. В операциях инкремента и декремента предполагается, что операнд будет изменен.

**Пример 20.12.** Перегрузка оператора инкремента (`++`) с использованием дружественной оператор-функции.

```

#include "stdafx.h"
#include <iostream>
#include <conio.h>
using namespace std;
class twins { int t1, t2;
public:
twins() { t1 = 0; t2 = 0; }

```

```

twins(int a, int b) { t1 = a; t2 = b; }
void get_tw(int &a, int &b){a = t1; b = t2;}
friend twins operator ++( twins &o);

};

twins operator ++( twins &o)
{ o.t1++; o.t2++;
return o;
}

int _tmain(int argc, _TCHAR* argv[])
{ twins ob1(10, 15), ob2(17, 19);
int m, k;
ob = ob1++;
ob1.get_tw(m, k);
cout << m << ' ' << k << endl; // 11 16
ob.get_tw(m, k);
cout << m << ' ' << k << endl; // 11 16
ob2++;
ob2.get_tw(m, k);
cout << m << ' ' << k << endl; // 18 20
ob.get_tw(m, k);
cout << m << ' ' << k << endl; // 18 20
_getch();
return 0;
}

```

## 20.8. Перегрузка операторов вставки и извлечения

В языке C++ вывод иногда называют *вставкой информации в поток* или просто *вставкой*, а ввод – *извлечением информации из потока* или просто *извлечением*.

При перегрузке `<<` для вывода создается функция вставки (*inserter function* или *inserter*), синтаксис которой имеет вид:

```

ostream &operator <<(ostream &stream, имя_класса имя_объекта)
{
    // тело функции вставки
    return stream;
}

```

Тип `ostream` задается внутри класса `ios`. Первый параметр является ссылкой на объект типа `ostream`, т. е. параметр `stream` представляет собой поток вывода. В качестве второго параметра выступает выводимый объект. В случае необходимости второй параметр также может быть ссылкой. Функция вставки возвращает ссылку на `ostream`, что позволяет организовать сложные выражения вида

```
cout << ob1 << ob2 << ob3;
```

Функция вставки не может быть элементом того класса, для работы с которым она создается. Если перегруженные операции вставки должны

иметь доступ к закрытым элементам класса, то они объявляются друзьями этого класса.

При перегрузке `>>` для вывода создается функция извлечения (extractor function или extractor), синтаксис которой имеет вид:

```
istream &operator >>(istream &stream, имя_класса &имя_объекта)
{
    // тело функции извлечения
    return stream;
}
```

Здесь первый параметр – ссылка на поток ввода, второй – ссылка на объект, получающий информацию. Поскольку функция извлечения возвращает ссылку на поток ввода `istream`, возможна организация сложных выражений вида

```
cin >> ob1 >> ob2 >> ob3;
```

Свойства функции извлечения аналогичны свойствам функции вставки.

#### Пример 20.13

```
#include "stdafx.h"
#include <iostream>
#include <conio.h>
#include <string>
using namespace std;

class PhotoModel
{ char name[10];           // имя
  char c_hair[10];          // цвет волос
  char c_eyes[10];          // цвет глаз
  double weight;            // вес
  int height;               // рост
public:
  PhotoModel(char *n, char *c_h, char *c_y, double w, int h)
  { strcpy(name, n);
    strcpy(c_hair, c_h);
    strcpy(c_eyes, c_y);
    weight = w;
    height = h;
  }
  friend ostream &operator << (ostream &stream, PhotoModel ob);
  friend istream &operator >> (istream &stream, PhotoModel &ob);
};

ostream &operator <<(ostream &stream, PhotoModel ob)
{ stream << ob.name << ' ' << ob.c_hair << ' ' << ob.c_eyes << ' '
  << ob.weight << ' ' << ob.height << endl;
  return stream;
}

istream &operator >>(istream &stream, PhotoModel &ob)
{ cout << "Input name ";

```

```
stream >> ob.name;
cout << "Input color of hair ";
stream >> ob.c_hair;
cout << "Input color of eyes ";
stream >> ob.c_eyes;
cout << "Input weight ";
stream >> ob.weight;
cout << "Input height ";
stream >> ob.height;
return stream;
}

int _tmain(int argc, _TCHAR* argv[])
{ PhotoModel Olga("Olga", "black", "black", 56.5, 170);
  PhotoModel Anna("", "", "", 0, 0);
  cout << Olga;
  cin >> Anna;
  cout << Anna;
  _getch();
  return 0;
}
```

Формально функции вставки и извлечения могут выполнять любые действия, в том числе и не связанные с вводом-выводом. Но хороший стиль программирования не рекомендует использовать любые перегруженные операции не по их прямому назначению.

#### 20.9. Перегрузка оператора вызова функции

Перегрузка оператора вызова функции `operator()` делает объект класса похожим на функцию, которую можно вызвать. Перегружаемый оператор вызова функции при необходимости может обеспечить возвращение значения заданного типа. Кроме того, в этой функции могут объявляться параметры. Перегружаемая операция вызова функции не может быть статическим членом класса.

#### Пример 20.14

```
class MyCls { int x;
public:
  int operator()(void);
  MyCls(int n) { x = n; }
};
```

В классе `MyCls` перегружается оператор вызова функции с помощью объявления

```
int operator()(void);
```

Можно заменить `(void)` списком необходимых параметров. Перегруженная функция-элемент реализуется как обычно:

```
int Mycls :: operator() (void) { return x; }
```

Здесь функция operator() возвращает целочисленное значение x объекта MyCls:

```
int _tmain(int argc, _TCHAR* argv[])
{ MyCls object =100;
  int q = object();           // выглядит как вызов функции,
  object.operator()           // но на самом деле это оператор
  cout << q;
  return 0;
}
```

### Контрольные вопросы

1. Как осуществляется перегрузка операций?
2. В чем состоят различия между перегрузкой операций с использованием дружественных функций и без них?
3. Какую роль играет указатель this при перегрузке операторов?

### Практические задания *20*

1. Определить класс-строку. В класс включить два конструктора: для определения класса строки строкой символов и путем копирования другой строки (объекта класса строки). Определить операции над строками:

>> переворачивание строки (запись символов в обратном порядке);  
++ нахождение наименьшего слова в строке.

2. Определить класс-строку. В класс включить два конструктора: для определения класса строки строкой символов и путем копирования другой строки (объекта класса строки).

Определить операции над строками:

+ конкатенация двух строк;  
++ преобразование символов строки в строчные символы.

3. Определить класс-матрицу. В класс включить два конструктора для определения матрицы по количеству элементов и путем копирования другой матрицы. При задании матрицы предусмотреть ее заполнение случайными числами.

Определить операции над матрицей:

++ нахождение наибольшего значения матрицы;  
+ получение новой матрицы, каждый элемент которой равен сумме элементов двух других матриц.

4. Определить класс-вектор. В класс включить два конструктора для определения вектора по его размеру и путем копирования другого вектора. При задании вектора по его размеру предусмотреть его заполнение случайными числами.

Определить операции над векторами:

& формирование нового вектора так, что каждый элемент нового вектора определяется следующим образом:  $c[i] = (a[i] > b[i]) ? a[i] : b[i];$

5. Определить класс-вектор. В класс включить два конструктора для определения вектора по его размеру и путем копирования другого вектора. При задании вектора по его размеру предусмотреть его заполнение случайными числами.

Определить операции над векторами:

[ ] нахождение значения элемента вектора по заданному номеру;  
-- сортировка элементов вектора по невозрастанию.

6. Определить класс список элементов. В определение класса включить два конструктора для определения списка по его размеру и путем копирования другого списка.

Определить операции над списком:

& формирование нового списка из двух списков так, что каждый элемент информационного поля нового списка удовлетворяет условию

$c=(a < b) ? a : b$

Определить функцию-элемент класса для вставки нового элемента в список на определенное место.

7. Определить класс-список элементов. В определение класса включить два конструктора для определения списка по его размеру и путем копирования другого списка.

8. Определить операции над списком: ++ сортировка списка по возрастанию; -- расположение элементов списка в обратном порядке.