Лабораторная работа №1

Тема работы: классы и объекты, организация ввода/вывода, динамическое выделение памяти.

Цель работы: изучить организацию ввода/вывода и работу с динамической памятью при программировании алгоритмов в C++.

Теоретические сведения: рассмотрена в соответствующих разделах [1, 3−8].

Ввод/вывод. В C++ ввод и вывод данных производится потоками байт. Поток (последовательность байт) − это логическое устройство, которое выдает и принимает информацию от пользователя и связано с физическими устройствами ввода/вывода. При операциях ввода байты направляются от устройства в основную память. В операциях вывода – наоборот.

Имеется четыре потока (связанных с ними объекта), обеспечивающих ввод и вывод информации и определенных в заголовочном файле iostream.

В файле iostream перегружаются два оператора побитового сдвига

```
<<          // поместить в выходной поток
>>          // считать со входного потока
```

и объявляются три стандартных потока:

```
cout        // стандартный поток вывода (экран)
cin         // стандартный поток ввода (клавиатура)
cerr        // стандартный поток диагностики (ошибки)
```

Объект cin. Для ввода информации с клавиатуры используется объект cin. Формат записи cin имеет следующий вид:

cin [>>имя_переменной];

При вводе необходимо, чтобы данные вводились в соответствии с форматом переменных.

Объект cout. Объект cout позволяет выводить информацию на стандартное устройство вывода – экран. Формат записи cout имеет следующий вид:

cout << data [<< data];

где data − это переменные, константы, выражения или комбинации всех трех типов. Пример использования объектов cin и cout.

```
#include<iostream>
using namespace std;
int main()
{
    setlocale(LC_ALL,"Russian");
    int i;
    double x;
    cout << "Введите число с двойной точностью" << endl;;
    cin >> x;                       // ввод числа с плавающей точкой
    cout << "Введите положительное число" << endl;
    cin  >> i;                      // ввод целого числа
    cout << "i * x=" <<  i*x  << endl;    // вывод результата
```

```
    return 0;
}
```

Идентификатор endl называется манипулятором. Он очищает поток cerr и добавляет новую строку.
Для управления выводом информации в языке C++ используются манипуляторы. Для их использования необходим заголовочный файл iomanip. Манипуляторы hex и oct используются для вывода числовой информации в шестнадцатеричном или восьмеричном представлении. Применение их можно видеть на примере следующей простой программы:

```
#include<iostream>
using namespace std;
int main()
{
  setlocale(LC_ALL,"Russian");
    int a=0x11, b=4,     // целые числа: шестнадцатеричное и десятичное
        c=051, d=8,      //                восьмеричное и десятичное
    i, j;
    i=a+b;
    j=c+d;
    cout << i << ' ' << hex << i << ' ' << oct << i << ' ' << dec << i << endl;
    cout << hex << j << ' ' << j << ' ' << dec << j << ' ' << oct << j << endl;
    return 0;
}
```

Манипуляторы изменяют значение некоторых переменных в объекте cout. Эти переменные называются флагами состояния. Когда объект посылает данные на экран, он проверяет эти флаги.
Пример использования манипуляторов форматирования информации.

```
#include<iostream>
#include <iomanip>
using namespace std;
int main()
{
     int a=0x11;
     double d=12.362;
     cout << setw(4) << a << endl;
     cout << setw(10) << setfill('*') << a << endl;
     cout << setw(10 ) << setfill(' ') << setprecision(3) << d << endl;
     return 0;
}
```

Манипуляторы setw(), setfill(' ') и setprecision() позволяют изменять флаги состояния объекта cout. Синтаксис их показывает, что это функции, позволяющие изменять флаги состояния объекта cout. Функции имеют следующий формат:
setw(количество_позиций_для_вывода_числа)
setfil(символ_для_заполнения_пустых_позиций)
setprecision (точность_при_выводе_дробного_числа)
Наряду с перечисленными выше манипуляторами в С++ используются также манипуляторы **setiosflags**() и **resetiosflags**() для установки определенных глобальных флагов, используемых при вводе и выводе информации. На эти флаги ссылаются как на *переменные состояния*. Функция setiosflags() устанавливает указанные в ней флаги, а resetiosflags() сбрасывает (очищает) их. Для того чтобы установить или сбросить некоторый флаг, могут быть использованы функции **setf**() или **unsetf**(). Флаги формата объявлены в классе **ios**. Рассмотрим пример, демонстрирующий использование манипуляторов.

```
#include<iostream>
#include <iomanip>
using namespace std;
int main()
{
    setlocale(LC_ALL,"Russian");
    char s[]="Минск БГУИР ";
    cout << setw(30) << setiosflags(ios::right) << s << endl;
    cout << resetiosflags(ios::right);
    cout << setw(30) << setiosflags(ios::left) << s << endl;
    return 0;
}
```

Динамическое распределение памяти. Память под массивы можно выделять динамически, т. е. размещать в свободной памяти (free store). Свободная память – это предоставляемая системой область памяти для объектов, время жизни которых устанавливается программистом. В С++ для операций выделения и освобождения памяти используются встроенные операторы **new** и **delete.**
Оператор new имеет один операнд. Оператор имеет две формы записи:
[::] new [(список_аргументов)] имя_типа [(инициализирующее_значение)]
[::] new [(список_аргументов)] (имя_типа) [(инициализирующее_значение)]
В простейшем виде оператор new можно записать следующим образом:
new имя_типа;
new имя_типа(выражение);
new имя_типа[выражение];
Оператор new выделяет надлежащий объем свободной памяти для хранения указанного типа и возвращает базовый адрес объекта. Когда память

недоступна, оператор new возвращает NULL либо возбуждает соответствующее исключение, например:

```
#include<iostream>
#include <iomanip>
using namespace std;
int  main()
{
    setlocale(LC_ALL,"Russian");
    int *p, *q, size,i;
    p=new int(5);        // выделение памяти и инициализация
    cout << "Введите размер массива" << endl;
    cin >> size;
    q=new int[size];     // выделение памяти под массив
    cout << "Введите элементы массива" << endl;
    for(i=0; i < size; i++)
        cin >> q[i];
    cout << " Массив q" << endl;
    for(i=0; i < size; i++)
        cout << setw(4)<<  q[i];
    cout << endl;
    delete p;
      delete [] q;
    return 0;
}
```

Можно динамически выделить память под двухмерный массив, используя «указатель на указатель». В языке C++ допустимо объявлять переменные, имеющие тип «указатель на указатель». Объявляется «указатель на указатель» следующим образом:

```
int **mas;
```

Фактически «указатель на указатель» – это адрес ячейки памяти, хранящей адрес указателя. Например:

```
#include<iostream>
#include <iomanip>
using namespace std;
int main()
{
   setlocale(LC_ALL,"Russian");
   int **mas;                          // указатель на указатель на массив
   int n, m;                      // количество строк и столбцов массива
   cout << "Введите количество строк и столбцов" << endl;
   cin >> n >> m;                       // ввод количества строк и столбцов
```

```
    mas=new int*[n];                    // выделение памяти под массив указателей
    for(int j=0; j < n; j++)
        mas[j]=new int[m];
    cout << "Введите элементы в массив" << endl;
    for(int i=0; i < n; i++)
        for(int j=0; j < m; j++)
            cin >> mas[i][j];
    cout << "Двухмерный массив" << endl;
    for(int i=0; i < n; i++)
    {
         cout << endl;
         for(int j=0; j < m; j++)
         cout << setw(4) << mas[i][j];
      }
    cout << endl;
    for(int i=0; i < n; i++)
        delete [] mas[i];
     delete [] mas;
     return 0;
}
```

Оператор delete уничтожает объект, созданный с помощью new.
Оператор delete имеет две формы записи:
[::] delete переменная_указатель // для указателя на один элемент
[::] delete [] переменная_указатель // для указателя на массив
Первая форма используется, если соответствующее выражение new размещало не массив. Во второй форме присутствуют пустые квадратные скобки, показывающие, что изначально размещался массив объектов. Оператор delete не возвращает значения.

Основная идея введения классов заключается в том, чтобы предоставить программисту средства для создания новых типов данных, которые могут использоваться так же, как и встроенные типы. Класс – это абстракция, точнее, это тип, определяемый пользователем. Например, мы можем задать структуру с именем point, которая содержит координаты ***x*** и ***y*** точки на экране дисплея:

```
struct point
{
   int x,y;
};
```

Пусть нам нужны две функции, которые позволяют нарисовать точку set_pixel() и прочитать ее координаты get_pixel():

```
void set_pixel(int, int);
void get_pixel(int *, int *);
```

В нашем примере данные и функции, работающие с этими данными, отделены друг от друга. Связь между ними можно установить, если задать структуру в виде

```
struct point
{
   int x,y;
   void set_pixel(int, int);
   void get_pixel(int *, int *);
};
```

Данные x и y, объявленные в приведенной структуре, называются компонентами-данными, а функции – компонентами-функциями или методами. Объект объявляется следующим образом:
имя_класса имя_объекта;
Теперь мы можем обратиться к данным и вызвать функции, только указав имя объекта, к которому они принадлежат. Для этих целей можно использовать те же операции точка «.» и «->». Рассмотрим пример.

```
#include<iostream>
#include <iomanip>
using namespace std;
struct point                    // объявление структуры
{
    int x,y;
    void set_pixel(int, int);       // установить значения переменных x и y
    void get_pixel(int *, int *);   // получить значения x и y
};
int main()
{
    setlocale(LC_ALL,"Russian");
    int a,b;
    point my_point, *pointer;
    pointer = &my_point;
    my_point.set_pixel(50,100);
    pointer -> get_pixel(&a,&b);
    cout << "a= " << a << " b= " << b << endl;
    return 0;
}
void point::set_pixel(int a, int b)
{   x=a;
    y=b;
```

```
}
void point::get_pixel(int *a, int *b)
{
    *a=x;
    *b=y;
}
```

Поскольку различные структуры могут иметь функции с одинаковыми именами, при описании функции необходимо указывать, для какой структуры она описывается:

```
void point::set_pixel(int x, int y)
{
    тело функции
}
```

Синтаксис описания функции, принадлежащей структуре, имеет следующий вид (рис. 1).

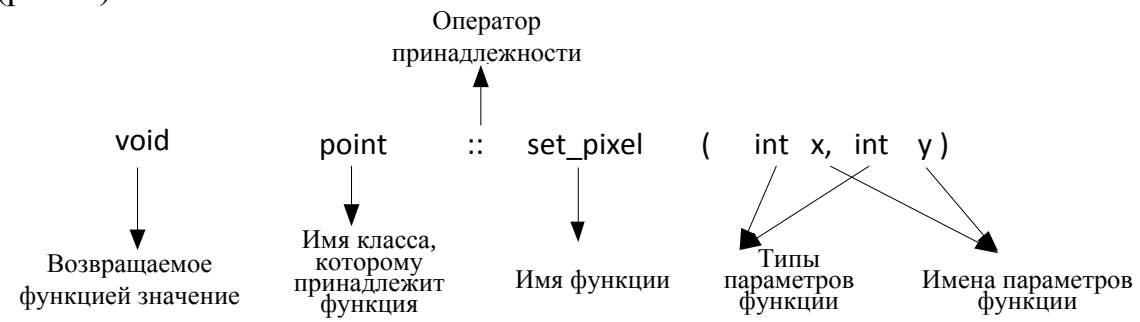

Рис.1. Синтаксис описания функции, принадлежащей структуре

Оператор принадлежности «::» иначе называется оператором разрешения области видимости. Этот оператор самого высокого приоритета.

```
::i          // унарный оператор :: указывает на внешнюю область видимости
point::i     //  бинарный оператор :: указывает на область видимости класса
```

В языке C++ структура – это тоже класс. С другой стороны, мы можем записать вместо ключевого слова struct ключевое слово class, например:

```
class point
{
    int x,y;
    void set_pixel(int, int);
    void get_pixel(int *, int *);
};
```

В языке C++ класс, определяемый посредством ключевых слов struct, class, union, включает в себя функции и данные, создавая новый тип объектов. Компоненты класса имеют ограничения на доступ. Эти ограничения определяются ключевыми словами private, protected, public. Для ключевого слова class по умолчанию все компоненты будут private. Это означает, что они

(их имена) будут недоступны для использования вне компонентов класса. Ограничения доступа для некоторого компонента можно изменить, записав перед ним атрибут модификации доступа – ключевое слово public или protected и двоеточие. Таким образом, упрощенную форму описания класса можно записать в виде:

```
class имя_класса
{
      данные и функции с атрибутом private (по умолчанию)
  protected:
      данные и функции с атрибутом protected
  public:
      данные и функции с атрибутом public
} объекты этого класса через запятую.
```

Обычно ограничения на уровень доступа касаются элементов данных: данные имеют атрибут private или protected, а методы – public.
Смысл атрибутов доступа следующий:

- **private** – член класса с атрибутом private может использоваться только методами собственного класса и функциями-«друзьями» этого же класса; по умолчанию все члены класса, объявленного с ключевым словом class, имеют атрибут доступа private;
- **protected** – то же, что и private, но дополнительно член класса может использоваться методами и функциями-«друзьями» производного класса, для которого данный класс является базовым;
- **public** – член класса может использоваться любой функцией программы, т. е. защита на доступ снимается.

Явно ограничения на доступ могут переопределяться записью атрибута перед компонентами класса. Элементы класса типа структуры (struct) и объединения (union) по умолчанию принимаются как public. Для ключевого слова struct атрибут можно явно переопределить на private или protected. Для ключевого слова union явное переопределение атрибута доступа невозможно. Класс или структура может содержать любое количество секций с заданными атрибутами. Секция начинается с ключевого слова и двоеточия после него. Секция заканчивается в конце описания структуры (класса) или началом другой секции.
Пример использования атрибутов доступа к элементам класса.

```
#include<iostream>
#include <iomanip>
using namespace std;
 class String
{
   char str[25];                          // атрибут доступа private
  public:
   void set_string(char *);   // функция инициализации строки str
```

```
    void display_string();                  // функция вывода строки str на экран
    char * return_string();             // функция возвращения строки
};
void String::set_string(char *s)
{
   strcpy(str,s);                          // копирование s в str
}
void String::display_string()
{
    cout << str << endl;
}
char * String::return_string()
{
    return str;
}
int main()
{
   setlocale(LC_ALL,"Russian");
   String str1;                        // объявление объекта
   str1.set_string("Минск");
   str1.display_string();
   cout << str1.return_string() << endl;
     return 0;
}
```

Использование функций для установки начальных значений данных объекта часто приводит к ошибкам. В связи с этим введена специальная функция, позволяющая инициализировать объект в процессе его декларирования (определения). Эта функция называется конструктором. Функция-конструктор имеет то же имя, что и соответствующий класс.

```
class String
{
    char str[25];                // атрибут доступа private
  public:
    String(char *s)        // конструктор
    {
       strcpy(str,s);
    }
};
```

Конструктор может иметь и не иметь аргументы и он никогда не возвращает значение (даже типа void). Класс может иметь несколько конструкторов, что позволяет использовать несколько различных способов инициализации соответствующих объектов. Иначе можно сказать – конструктор является функцией, а значит он может быть перегружен. Конструктор вызывается, когда связанный с ним тип используется в определении. Например:

```
#include<iostream>
using namespace std;
 class Over
{
     int i;
     char *str;
   public:
     Over()
     {
          str="Первый конструктор";
         i=0;
     }
     Over(char *s)           // Второй конструктор
    {
          str=s;
          i=50;
    }
     Over(char *s, int x)     // Третий конструктор
     {
          str=s;
          i=x;
     }
     Over(int *y)
     {
          str="Четвертый конструктор\n";
          i=*y;
     }
     void print();
};
void Over::print()
{
cout << "i= " << i << " str= " << str << endl;
}
int main()
{    setlocale(LC_ALL,"Russian");
int a=10, *b;
b=&a;
Over my_over;               // Активен конструктор Over()
Over my_over1("Для конструктора с одним параметром");
Over my_over2("Для конструктора с двумя параметрами", 100);
Over my_over3(b);          // Для четвертого конструктора
my_over.print();
my_over1.print();
my_over2.print();
my_over3.print();
return 0;
}
```

Результаты выполнения программы представляются в следующем виде:
i=0; str= Первый конструктор

i=50; str= Для конструктора с одним параметром
i=100; str= Для конструктора с двумя параметрами
i=10; str= Четвертый конструктор

Конструктор может содержать значения аргументов по умолчанию. Задание в конструкторе аргументов по умолчанию позволяет гарантировать, что объект будет находиться в непротиворечивом состоянии, даже если в вызове конструктора не указаны никакие значения. Созданный программистом конструктор, у которого все аргументы по умолчанию, называется конструктором с умолчанием, т. е. конструктором, который можно вызывать без указания каких-либо аргументов. Для каждого класса может существовать только один конструктор с умолчанием.

Пример использования конструктора по умолчанию.

```
#include<iostream>
using namespace std;
class Time
{
  public:
      Time(int = 0, int = 0, int = 0);  // конструктор с параметрами по умолчанию
                void  setTime(int, int, int);      // установка часов, минут, секунд
                void printStandart();              // печать времени в стандартном формате
            private:
       int hour;                          // 0–23
       int minute;                        // 0–59
       int second;                        // 0–59
}
// Конструктор Time инициализирует члены класса нулевыми значениями
         Time::Time(int hr, int min, int sec)
{
     setTime(hr, min, sec);
}
// Установка нового значения времени. Выполнение проверки корректности
// значений данных. Установка неправильных значений на 0.
void Time::setTime(int h, int m, int s)
{
     hour     = (h >= 0 && h < 24)  ?  h : 0;
     minute = (m >= 0 && m < 60)  ?  m : 0;
     second = (s >= 0 && s < 60) ?  s : 0;
}
 // Печать времени в стандартном формате
void Time::printStandart()
{
     cout << (( hour == 0 || hour == 12) ?  12 : hour % 12)
           << ":" << ( minute < 10 ?  "0" : "" ) << minute
           << ":" << ( second  < 10 ?  "0" : "" ) << second
           <<  ( hour < 12 ?  "AM" : "PM" );
}
int main()
```

```
{
    setlocale(LC_ALL,"Russian");
    Time t1,                    // все аргументы являются умалчиваемыми
   t2(2),                  // минуты и секунды являются умалчиваемыми
   t3(21, 34),             // секунды являются умалчиваемыми
   t4(12, 25, 42),         //  все значения указаны
   t5(27, 74, 99);         // все неправильные значения указаны
     cout << "\nВсе аргументы по умолчанию ";
     t1.printStandart();
     cout << "\nЧасы заданы; минуты и секунды по умолчанию ";
     t2.printStandart();
     cout << "\nЧасы и минуты заданы;  секунды по умолчанию ";
     t3.printStandart();
     cout << "\nЧасы, минуты и секунды заданы: ";
     t4.printStandart();
     cout << "\nВсе значения заданы неверно: ";
     t5.printStandart();
     cout << endl;
     return 0;
}
```

Конструктор имеет следующие отличительные особенности:

− всегда выполняется при создании нового объекта, т.е. когда под объект отводится память и когда он инициализируется;

− может определяться пользователем или создаваться по умолчанию;

− не может быть вызван явно из пределов программы (не может быть вызван как обычный метод). Он вызывается явно компилятором при создании объекта и неявно при выполнении оператора new для выделения памяти объекту;

− всегда имеет то же имя, что и класс, в котором он определен;

− никогда не должен возвращать значения;

− не наследуется.

Копирующий конструктор. Рассмотрим следующую программу, демонстрирующую использование копирующего конструктора.

```
#include<iostream>
#include<iomanip>
using namespace std;
class Massiv
{
    int *mas;              // указатель на массив
    int n;                 // количество элементов массива
 public:
    Massiv(int n1=0);      // конструктор с параметрами по умолчанию
    void vvod();           // функция ввода значений массива
    void display();        // функция вывода значений массива
    int fun(Massiv ob);    // функция вычисления суммы элементов массива
    ~Massiv();             // деструктор
```

```
 };
Massiv::Massiv(int n1)      // определение конструктора
{
      n=n1;
      mas=new int[n];
}
Massiv::~Massiv()           // определение деструктора
{
     delete [] mas;
}
void Massiv::vvod()         // определение функции ввода значений массива
{
     for(int i=0; i < n; i++)
          cin >> mas[i];
}
void Massiv::display()      // определение функции вывода значений массива
{
     for(int i=0; i < n; i++)
          cout << setw(4) << mas[i];
     cout << endl;
}
int Massiv:: fun(Massiv ob)   // функция вычисления суммы элементов массива
{
     int sum=0;
     for(int i=0; i < n; i++)
          sum+=ob.mas[i];
     return sum;
}
int  main()                    // головная функция
{
      setlocale(LC_ALL,"Russian");
      int summa;
      int n;                       // размер массива
      cout << "Введите размер массива" << endl;
      cin >> n;
      Massiv ob(n);                // объявление объекта
      cout << "Введите элементы массива" << endl;
      ob.vvod();                          // вызов функции vvod
      cout << "Исходный массив" << endl;
      ob.display();                // вызов функции display
      summa=ob.fun(ob);            // вызов функции fun
      cout << "сумма элементов массива" << setw(4) << summa << endl;
       ob.display();               // при вызове функции display() возникает ошибка
}
```

В функцию fun() передается значение объекта типа Massiv. Даже если вызванная функция ничего не будет выполнять, произойдет ошибка, связанная с динамическим выделением и освобождением памяти. Параметр в функции fun() является локальным (автоматическим объектом) в теле этой функции.

Любой автоматический объект конструируется тогда, когда встречается его объявление, и разрушается, когда блок, в котором он описан, прекращает существование. После завершения функция прекращает существовать, в результате вызывается деструктор объекта ob.
В функции main() выполняются следующие действия:
− описание Massiv ob(n); задается конструирование нового объекта ob. Конструктор объекта ob выделяет (динамически) память под массив mas с помощью оператора new;
− вызывается функция fun(ob);
− значение объекта ob копируется из функции main в стек функции fun();
− копия объекта ob содержит указатель на ту же динамическую память (указатель на динамическую память в объекте-оригинале и объекте-копии имеет одинаковые значения);
− функция fun() завершается;
− вызывается деструктор для копии объекта ob, который разрушает динамически выделенную память под массив mas;
− указатель в оригинале объекта адресует несуществующую удаленную память.
Если в приведенном примере изменить заголовок функции fun(Massiv ob) на заголовок fun(Massiv& ob), то ошибка будет устранена. При необходимости можно оставить и исходное объявление функции. В этом случае надо устранить ошибку в самом классе Massiv. Когда объект ob копируется из функции main() в функцию fun, то должен вызываться конструктор для копирования. Общий вид конструктора копирования имеет следующий вид:

```
имя_класса (const имя_класса  & );
```

Так как в нашем классе такого конструктора нет, то вызывается конструктор, заданный по умолчанию. Этот конструктор строит точную копию всех данных объекта ob, что и приводит к ошибке. Если в классе Massiv задать явно конструктор для копирования, например,

```
Massiv(const Massiv& ob)
{
   n=ob.n;
   mas=new int[n];
   for(int i=0; i < n; i++)
        mas[i]=ob.mas[i];
}
```

то ошибка будет устранена. Таким образом, если в конструкторе некоторого класса Massiv осуществляется динамическое выделение памяти, такой класс должен иметь соответствующий конструктор для копирования, а также деструктор (для освобождения памяти).
Деструктор. Противоположные действия, по отношению к действиям конструктора, выполняют функции-деструкторы (destructor), или разрушители, которые уничтожают объект. Деструктор может вызываться явно или неявно.

Неявный вызов деструктора связан с прекращением существования объекта из-за завершения области его определения. Явное уничтожение объекта выполняет оператор delete. Деструктор имеет то же имя, как и класс, но перед именем записывается знак тильда «~». Кроме того, деструктор не может иметь аргументы, возвращать значение и наследоваться.
Пример использования деструктора.

```
#include<iostream>
using namespace std;
class String
{
    int i;
  public:
    String(int j);             // объявление конструктора
    ~String();                 // объявление деструктора
    void show_i(void);
};                              // конец объявления класса
String::String(int j)         // определение конструктора
{
    i=j;
   cout << "Работает конструктор" << endl;
}
void String::show_i(void)    // определение функции
{
   cout << "i= " << i << endl;
}
String::~String()                 // определение деструктора
{
   cout << "Работает деструктор" << endl;
}
int main()
{
   setlocale(LC_ALL,"Russian");
   String my_ob1(25);     // инициализация объекта my_ob1
   String my_ob2(36);     // инициализация объекта my_ob2
   my_ob1.show_i();       // вызов функции show_i() класса String для my_ob1
   my_ob2.show_i();       // вызов функции show_i() класса String для my_ob2
     return 0;
}
```

Результаты работы программы следующие:

```
Работает конструктор
Работает конструктор
i=25
i=36
Работает деструктор
Работает деструктор
```

Деструкторы выполняются в обратной последовательности по отношению к конструкторам. Первым разрушается объект, созданный последним.
Пример использования конструкторов и деструктора. Разработать класс вектор (одномерный динамический массив). Методы класса: конструкторы, деструктор и несколько методов, выполняющих преобразование в массиве (например, нахождения максимального значения и сортировки).

```
#include <iostream>
#include <new>
#include<iomanip>
using namespace std;
// объявление класса
class vect
{
    int *v;              //  вектор (одномерный массив)
    int n;               //  размерность вектора
  public:
    vect();              //  конструктор без параметров
    vect(int, int*);     //  конструктор с двумя параметрами
    ~ vect();            //  деструктор
    void set();          //  инициализация вектора
    void print();        //  вывод вектора на экран
    int funk1();         //  нахождения максимума в массиве
};
// реализация методов класса
vect :: vect() : n(0),v(0){}    //  конструктор без параметров
vect :: vect(int nn, int *vv)   //  конструктор с двумя параметрами
{
    n=nn;                          // инициализация размерности
    v=new int[n];                  // выделение памяти под вектор
    for(int i=0; i<n; i++)
        *(v+i)=*(vv+i);
}
vect :: ~ vect()                //  деструктор
{ delete [] v; }
void vect :: set()              //  инициализация вектора
{
    if(!v)
    {
        cout << "не выделена память под вектор" << endl;
        cout << "введите размерность вектора" << endl;
        cin >> n;
        v=new int[n];
   }
   cout << "Введите элементы в вектор" << endl;
   for(int i=0; i<n; i++)
       cin >> *(v+i);
}
void vect :: print()          //  функция вывода вектора на экран
```

```
{
    if(!v)
    {
       cout << "вектор пустой" << endl;
       return ;
   }
   for(int i=0; i<n; i++)
      cout << setw(4) << *(v+i);
   cout << endl;
}
int vect :: funk1()              // функция  нахождения максимума в массиве
{
    int max=*v;                    // инициализация максимального элемента
    for(int i=0; i < n; i++)
        if(max < *(v+i))
          max=*(v+i);
    return max;
}
int main()
{
     setlocale(LC_ALL,"Russian");
     int rez;                // максимальное число объекта
     vect v1;               // объявление объекта, вызывается конструктор
                             // без параметров

     int ms[]={2,1,6,4,5};                      // объявление и инициализация массива
     vect v2((sizeof(ms)/sizeof(int)),ms);  // объявление объекта, вызывается
                                              // конструктор с параметрами
     v1.set();                                // функция инициализирует объект v1
     cout << "Вектор v1" << endl;
     v1.print();                                // функция выводит на экран объект v1
     cout << "Вектор v2" << endl;
     v2.print();                                // функция выводит на экран объект v2
     rez=v2.funk1();                    // функция возвращает максимальное
                                  // число объекта
     cout << "Максимальное число = " << rez << endl;
     return 0;
}
```

Указатель this. Каждый новый объект имеет скрытый от пользователя свой указатель. Иначе это можно объяснить так. Когда объявляется объект, под него выделяется память. В памяти есть специальное поле, содержащее скрытый указатель, который адресует начало выделенной под объект памяти. Получить значение указателя в компонентах-функциях объекта можно с помощью ключевого слова this (рис. 2). Для любой функции, принадлежащей классу my_class, указатель this неявно объявлен так:

```
my_class *const this;
```

Если объявлен класс и объекты, то размещение объектов в памяти будет выглядеть следующим образом:

```
class string
{ . . .} str1, str2;
```

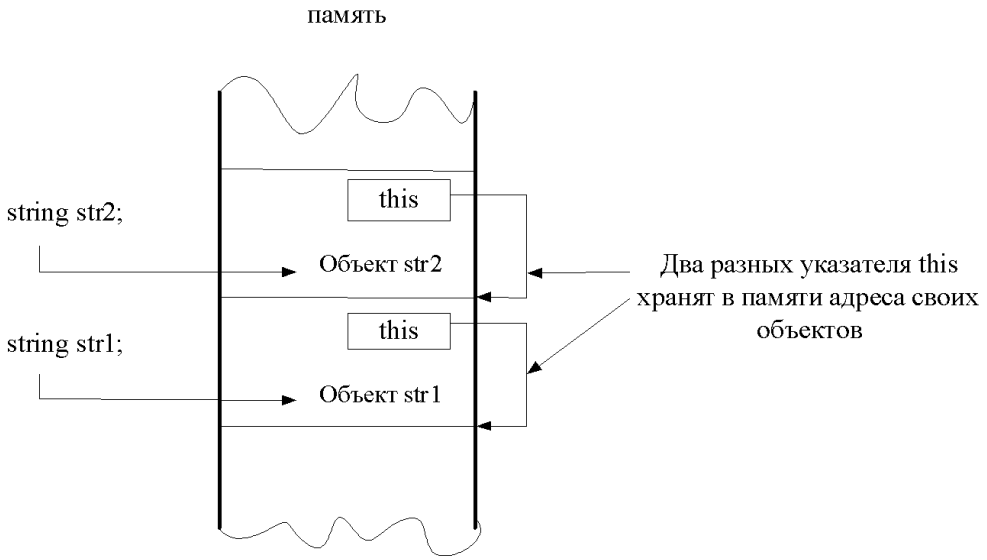

Рис. 2. Указатель this

Основные свойства и правила использования указателя this:

- каждый новый объект имеет свой скрытый указатель this;
- указывает на начало своего объекта в памяти компьютера;
- не надо дополнительно объявлять;
- передается как скрытый аргумент во все нестатические (т. е. не имеющие спецификатора static) компоненты-функции своего объекта;
- является локальной переменной, которая недоступна за пределами объекта (она доступна только во всех нестатических компонентах-функциях своего объекта);
- разрешается обращаться к указателю this непосредственно в виде this или *this.

Пример использования указателя this.

```
#include<iostream>
using namespace std;
#include<string.h>
class String                        // объявление класса
{
     char str[100];
     int k;
     int *r;
 public:
    String() {k=123; r=&k;}     // конструктор
    void read(){ gets(str);}              // функция чтения строки
    void print() { puts(str); }    // функция печати строки
    void read_print();
```

```
};
void String::read_print()
{
    char c;
    cout <<  this->k << endl;
    cout << "Введите строку" << endl;
    // ключевое слово this содержит скрытый указатель на класс String,
    // поэтому конструкция this->read() выбирает через указатель
    // функцию read() этого класса
    this->read();
    cout << "Введенная строка" << endl;
    this->print();
     // ниже, в цикле for, одинаково удаленные от середины строки
     // символы меняются местами
    for(int i=0, j=strlen(str)-1; i<j; i++,j--)
    {
        c=str[i];
      str[i]=str[j];
      str[j]=c;
    }
    cout << "Измененная строка" << endl;
    (*this).print();
}
int main()
{
    setlocale(LC_ALL,"Russian");
    String S;                    // объявление объекта
    S.read_print();              // вызов функции класса
}
```

static-члены (данные) класса. Компоненты-данные могут быть объявлены с модификатором класса памяти static. Класс, содержащий static компоненты-данные, объявляется как глобальный (локальные классы не могут иметь статических членов). static-компонента совместно используется всеми объектами этого класса и хранится в одном месте. Статическая компонента глобального класса должна быть явно определена в контексте файла. Основные правила использования статических компонент:

− статические компоненты будут одними для всех объектов данного класса, т. е. ими используется одна область памяти;

− статические компоненты не являются частью объектов класса;

− объявление статических компонент-данных в классе не является их описанием. Они должны быть явно описаны в контексте файла;

− локальный класс не может иметь статических компонент;

− к статической компоненте st класса cls можно обращаться cls::st независимо от объектов этого класса, а также при помощи операций «.» и «->» при использовании объектов этого класса;

− статическая компонента существует даже при отсутствии объектов этого класса;
− статические компоненты можно инициализировать, как и другие глобальные объекты, только в файле, в котором они объявлены.

Компоненты-функции static и const. В C++ компоненты-функции могут использоваться с модификатором static и const. Обычная компонента-функция имеет явный список параметров и неявный список параметров. Неявные параметры можно представить как список параметров, доступных через указатель this. Статическая (static) компонента-функция не может обращаться к любой из компонент посредством указателя this. Компонента-функция const не может изменять неявные параметры.

Основные свойства и правила использования static- и const-функций:
− статические компоненты-функции не имеют указателя this, поэтому обращаться к нестатическим компонентам класса можно только с использованием «.» или «->»;
− не могут быть объявлены две одинаковые функции с одинаковыми именами и типами аргументов, чтобы при этом одна была статической, а другая нет;
− статические компоненты-функции не могут быть виртуальными.

Пример использования static, const данных и методов.

```
#include <iostream>
using namespace std;
class cls
{
    int kl;                  // количество изделий
    double zp;               // зарплата на производство одного изделия
    double nl1,nl2;          // два налога на зарплату
    double sr;               // количество сырья на производство одного изделия
    static double cs;        // цена сырья на одно изделие
  public:
    cls(){}                  // конструктор по умолчанию
    ~cls(){}                 // деструктор
    void inpt(int);
    static void vvod_cn(double);
    double seb() const;
};
double cls::cs;              // явное определение static-члена в контексте файла
void cls::inpt(int k)
{
     kl=k;
     cin >> nl1 >> nl2 >> zp;
}
void cls::vvod_cn(double c)
{
     cs=c;           // можно обращаться в функции только к static-компонентам
}
```

```
double cls::seb() const
{
     return kl*(zp+zp*nl1+zp*nl2+sr*cs); // в функции нельзя изменить ни один
}                                                    // неявный параметр (kl zp nl1 nl2 sr)
int main()
{
    setlocale(LC_ALL,"Russian");
    cls c1,c2;
    cout << "Введите зарплату и два налога для объекта c1" << endl;
    c1.inpt(100);                          //  инициализация первого объекта
    cout << "Введите зарплату и два налога для объекта c2" << endl;
    c2.inpt(200);                          //  инициализация второго объекта
    cls::vvod_cn(500.);
    cout << "\nc1 = " << c1.seb() << "\nc2 = " << c2.seb() << endl;
    return 0;
}
```

Контрольные вопросы

1. Как осуществляется ввод/вывод в C++?
2. Что такое манипулятор ввода/вывода?
3. Для чего необходимы операторы new и delete? В чем их отличие от функций malloc() и free()?
4. Как создать и удалить массив объектов?
5. Как можно выделить память под двухмерный массив?
6. В чем разница между struct, class и union?
7. Что такое указатель this? Приведите пример использования этого указателя.
8. Какова основная форма конструктора копирования и когда он вызывается?
9. Когда вызывается деструктор?
10. Приведите пример использования константных и статических данных и методов.

Порядок выполнения работы

1. Изучить краткие теоретические сведения.
2. Ознакомиться с материалами литературных источников.
3. Ответить на контрольные вопросы
4. Разработать алгоритм программы.
5. Написать, отладить и выполнить программу.

Варианты заданий

1. Создать класс, в котором реализовать функции для работы с матрицами:

а) функция производит перемножение матриц;

б) функция производит сложение двух матриц.

Память под матрицы отводить динамически. Использовать конструктор с параметрами, конструктор копирования. Деструктор должен освобождать память, выделенную под матрицы.

2. Создать класс, в котором реализовать функции для работы с одномерными массивами:

а) получить пересечение элементов массивов;

б) получить объединение элементов массивов.

Память под массивы отводить динамически. Использовать конструктор с параметрами, конструктор копирования. Деструктор должен освобождать память, выделенную под массивы.

3. Создать класс, в котором реализовать функции для работы с двухмерными массивами:

а) получить пересечение элементов массивов;

б) получить объединение элементов массивов.

Память под массивы отводить динамически. Использовать конструктор с параметрами, конструктор копирования. Деструктор должен освобождать память, выделенную под массивы.

4. Создать класс employee. Класс должен включать поле int для хранения номера сотрудника и поле float для хранения величины его оклада. Расширить содержание класса employee, включив в него класс date и перечисление etype. Объект класса date использовать для хранения даты приема сотрудника на работу. Перечисление использовать для хранения статуса сотрудника: лаборант, секретарь, менеджер и т. д. Разработать методы getemploy() и putemploy(), предназначенные соответственно для ввода и отображения информации о сотруднике. Написать программу, которая будет выдавать сведения о сотрудниках.

5. Создать класс, одно из полей которого хранит «порядковый номер» объекта, т. е. для первого созданного объекта значение этого поля равно 1, для второго − 2 и т. д. Для того чтобы создать такое поле, необходимо иметь еще одно поле, в которое будет записываться количество созданных объектов. Каждый раз при создании нового объекта, конструктор может получить значение этого поля и в соответствии с ним назначить объекту индивидуальный порядковый номер. В классе разработать метод, который будет выводить на экран свой порядковый номер, например: «Мой порядковый номер: 2».

6. Создать класс «Время» с полями: час (0−23), минуты (0−59), секунды (0−59). В классе реализовать функции конструктора, деструктора, установки времени, получения часа, минуты и секунды, а также две функции-члены печати: печать по шаблону «16 часов 18 минут 3 секунды» и «4:18:3 p.m.». Функции-члены установки полей класса должны проверять корректность задаваемых параметров.

7. Создать класс «Квадрат». Поле класса – длина стороны квадрата. Функции-члены вычисляют площадь, периметр, устанавливают поля и возвращают значения. Функции-члены установки полей класса должны проверять корректность задаваемых параметров. Разработать функцию вывода на экран результата.

8. Создать класс «Окружность». Данные класса: радиус окружности. Функции-члены класса: вычисление площади, вычисление длины окружности, установление радиуса окружности и возвращение значения радиуса окружности, вывод данных на экран. Функции-члены установки полей класса должны проверять корректность задаваемых параметров.

9. Создать класс «Прямоугольник». Данные класса: высота и ширина прямоугольника. Функции-члены класса: вычисление площади, периметра, установление данных класса и возвращение их значений, вывод данных на экран. Функции-члены установки полей класса должны проверять корректность задаваемых параметров.

10. Создать класс типа «Односвязный список». Функции-члены добавляют элемент к списку, удаляют элемент из списка, печатают элементы с начала списка. Найти элемент в списке.

11. Создать класс типа «Стек». Функции-члены вставляют элемент в стек, удаляют элемент из стека, печатают элементы списка.

12. Создать класс, в котором реализуются функции для работы с одномерными массивами:

а) функция должна найти максимальное число в одном массиве;

б) функция должна найти минимальное число во втором массиве;

в) функция должна поменять местами минимальное и максимальное значения в массивах.

Память под массивы отводить динамически. Использовать конструктор с параметрами. Деструктор должен освобождать память, выделенную под массивы.

13. Создать класс, в котором реализовать функции для работы с двухмерными массивами:

а) функция должна обнулить строку и столбец с минимальным элементом.

б) функция должна в каждой строке массива первый найденный нечетный элемент поменять местами с первым элементом этой строки;

Память под массивы отводить динамически. Использовать конструктор с параметрами. Деструктор должен освобождать память, выделенную под массивы.

14. Создать класс, в котором надо реализовать функции для работы с двухмерными массивами:

а) функция находит минимальный элемент ниже главной диагонали;

б) функция находит максимальный элемент выше главной диагонали.

Память под массивы отводить динамически. Использовать конструктор с параметрами. Использовать константные и статические данные и методы. Деструктор должен освобождать память, выделенную под массивы.

15. Создать класс, в котором требуется реализовать функции для работы с матрицами:

а) функция находит минимальное число в матрице, максимальное число и меняет их местами.

б) функция выполняет обмен элементов первой и второй, третьей и четвертой и т. д. строк матрицы.

Память под матрицы отводить динамически. Использовать конструктор с параметрами по умолчанию. Использовать константные и статические данные и методы. Деструктор должен освобождать память, выделенную под матрицы.

Литература

1. Шилдт, Г. Искусство программирования на C++ / пер. с англ. – СПб. : БХВ-Петербург, 2005. – 928 с.
2. Дейтел, Х. Как программировать на C++ / Х. Дейтел, П. Дейтел ; пер. с англ. – М. : Бином-Пресс, 2009. – 1037 с.
3. Страуструп, Б. Программирование. Принципы и практика использования C++ / Б. Страуструп ; пер. с англ. – М. : Вильямс, 2011. – 1246 с.
4. Страуструп, Б. Язык программирования C++. Специальное издание / Б. Страуструп ; пер. с англ. – М. : Вильямс, 2012. – 1136 с.
5. Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений / Г. Буч ; пер. с англ. – М. : Вильямс, 2010 – 720 с.
6. Джамса, К. Учимся программировать на языке C++ / К. Джамса ; пер. с англ. – М. : Мир, 1997. – 320 с.
7. Павловская, Т. C/C++. Программирование на языке высокого уровня /

Т. Павловская. – СПб. : Питер, 2013. – 464 с.
8. Луцик Ю.А. Объектно-ориентированное программирование на языке С++ : Учеб. Пособие / Ю.А. Луцик, В.Н. Комличенко. – Минск : БГУИР, 2008. – 266 с.