



## **Введение в C/C++ программирование консоли**

На начальном этапе обучения C/C++ обычно создаются консольные приложения. Затем, познакомившись с основными возможностями языка, но еще не имея достаточного опыта, учащимся хочется создавать сколько-нибудь привлекательные приложения. В этом пособии мы предлагаем познакомиться с продвинутыми возможностями программирования в консольном окне Windows. В первом разделе пособия мы покажем, как можно улучшить внешний вид программ, используя только «текстовый» режим консоли. Затем вы познакомитесь с графическими возможностями этого окна. Вероятно многие программисты даже не подозревают, что в этом окне можно рисовать. Однако консольное окно – это окно Windows и в нем, как и в любом другом окне, можно использовать графические функции. Во втором разделе пособия читатели познакомятся с основными графическими функциями, которые программисты C/C++ могут использовать в простейшем режиме консольного окна.

### **Оглавление**

1. Элементы C/C++ программирования в консольном окне.....	2
1.1 Управление консольным окном в «текстовом» режиме .....	2
1.2 Рисование в консольном окне.....	35
1.3 Заключение .....	64

## 1. Элементы C/C++ программирования в консольном окне.

Консольное окно Windows является интерфейсом, который предоставляет Windows для ввода и вывода данных приложениям, работающим в символьном режиме, т.е. таким, которые не имеют собственного графического интерфейса. Так программы, которые создаются на начальном этапе обучения, обычно выполняются в консольном окне. Они используют стандартные «текстовые» функции C ввода/вывода `printf(...)`, `scanf(...)`, `getchar(...)` и др. или стандартные потоки `cout` и `cin` ввода/вывода C++. Однако библиотека функций не ограничивается только ими. Имеется возможность использовать функции Windows API для управления консольным вводом – выводом. В этом параграфе мы познакомим читателя с возможностями этих функций. Чтобы использовать такие функции вы должны включить в программу заголовочный файл `windows.h`, благодаря которому можно получить доступ к большому количеству функций, составляющих библиотеку Windows API. Здесь мы кратко опишем некоторые из них.

### 1.1 Управление консольным окном в «текстовом» режиме

Функция `SetConsoleTextAttribute` устанавливает атрибуты цвета текста (цвет букв и цвет фона ячеек с буквами). Она использует следующий синтаксис `BOOL SetConsoleTextAttribute(HANDLE, WORD);`

и принимает два аргумента, обозначающие дескриптор окна (в нашем случае консоли), и число, биты которого определяют цвета. Напомним, что тип `WORD` представляет 16 битовое беззнаковое целое число. Принцип установки и использования атрибутов прост: все символы, выведенные после вызова этой функции будут иметь установленные атрибуты.

Для получения идентификатора/дескриптора консольного окна (`HANDLE`) можно использовать функцию `GetStdHandle`:

```
HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
```

Здесь константа `STD_OUTPUT_HANDLE` говорит, что вы хотите получить дескриптор стандартного устройства ввода/вывода `stdout`.

Вторым аргументом в функцию `SetConsoleTextAttribute` передается число, биты которого управляют цветами фона и текста. Например, команды

```
SetConsoleTextAttribute(hStdOut, 2);  
cout << "Green\n";
```

выводят текст зеленым цветом на черном фоне, где `hStdOut` должен быть получен, так как описано ранее.

Второй аргумент функции `SetConsoleTextAttribute` использует младший байт слова `WORD`, биты которого мы поясняем на следующем рисунке

$$I_b \quad R_b \quad G_b \quad B_b \quad I_f \quad R_f \quad G_f \quad B_f$$

Интенсивность фона и текста (яркость) задается битами  $I_b$  и  $I_f$ . Наличие красной, зеленой и синей составляющей цвета фона определяется битами

$R_b, G_b, B_b$ , а биты  $R_f, G_f, B_f$  определяют цвет текста. Т.о. цвета фона и текста вы можете задать, передав вторым аргументом число в диапазоне от 0 до 255. Его биты будут управлять наличием или отсутствием RGB составляющих в цвете фона и текста.

В примере ранее, передав вторым аргументом десятичное число 2, которое в двоичной записи имеет вид 00000010, мы задали только бит  $G_f = 1$ , который и определил зеленый цвет текста на черном фоне. Однако второй аргумент функции `SetConsoleTextAttribute` удобнее конструировать из набора флагов. Интересующие нас флаги включают `BACKGROUND_` и `FOREGROUND_`, которые обозначают цвет фона и текста соответственно. Например, команды

```
SetConsoleTextAttribute(hStdOut, FOREGROUND_RED |
                          FOREGROUND_GREEN | FOREGROUND_INTENSITY);
cout << "Yellow\n";
```

выводят текст ярко желтым цветом на черном фоне. Если преобразовать второй аргумент в число, то в двоичной записи оно будет иметь вид 00001110, т.е. включение флага `FOREGROUND_INTENSITY` соответствует биту  $I_f = 1$ , флаг `FOREGROUND_RED` соответствует биту  $R_f = 1$ , а флаг `FOREGROUND_GREEN` соответствует биту  $G_f = 1$ . Оставшиеся биты равны нулю.

Аналогично задаются цвета фона. Например, команды

```
SetConsoleTextAttribute(hStdOut,
                          BACKGROUND_BLUE | BACKGROUND_INTENSITY |
                          FOREGROUND_RED | FOREGROUND_GREEN |
                          FOREGROUND_BLUE | FOREGROUND_INTENSITY);
cout << "White on blue\n";
```

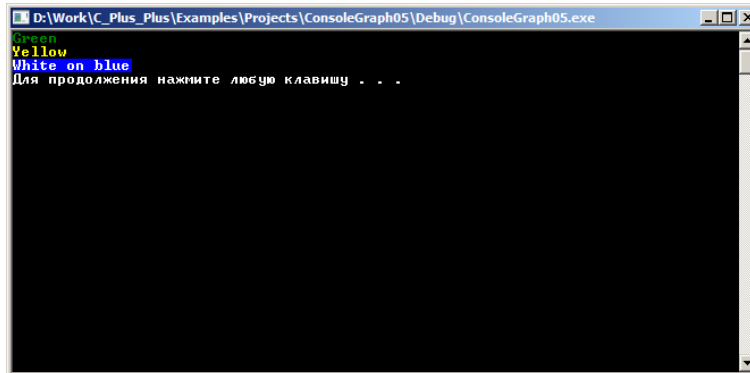
выводят белый текст на ярко синем фоне.

Приведенные выше команды, объединены в единый код в следующем примере.

### Пример 1.

```
#include <windows.h>
#include <iostream>
using namespace std;
void main()
{
    //получаем дескриптор
    HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
    //Выводим текст разными цветами
    SetConsoleTextAttribute(hStdOut, 2);
    cout << "Green\n"; //зеленый текст на черном фоне
    SetConsoleTextAttribute(hStdOut, FOREGROUND_RED |
                            FOREGROUND_GREEN | FOREGROUND_INTENSITY);
    cout << "Yellow\n"; //желтый текст на черном фоне
    SetConsoleTextAttribute(hStdOut,
                            BACKGROUND_BLUE | BACKGROUND_INTENSITY |
                            FOREGROUND_RED | FOREGROUND_GREEN |
                            FOREGROUND_BLUE | FOREGROUND_INTENSITY);
    cout << "White on blue\n"; //белый текст на синем фоне
```

```
// установка цветов для последующей печати
SetConsoleTextAttribute(hStdOut,
    FOREGROUND_RED | FOREGROUND_GREEN |
    FOREGROUND_BLUE | FOREGROUND_INTENSITY);
system("pause");
}
```



Последняя функция `system(...)` выполняет стандартные текстовые команды (команды DOS), которые понимает консоль. Например, ввод в качестве аргумента этой функции строки `"cls"` эквивалентно команде консоли `cls`, которая очищает экран.

В данной программе для простоты не был реализован контроль ошибок. О нем мы поговорим позже.

Таким образом, цветом текста управляет всего 4 бита, из которых мы можем получить 16 разных комбинаций. Раскраска фона выполняется аналогично. В частности команда

```
system("CLS");
```

очищает консольное окно, заполняя его фон цветом, который установлен в битах  $I_b, R_b, G_b, B_b$  последней команды `SetConsoleTextAttribute`.

В следующем примере мы отображаем 16 возможных цветов фона в виде таблицы, а также выводим двоичные значения соответствующих цветов.

## Пример 2.

```
#include <windows.h>
#include <string>
#include <iostream>
using namespace std;

string into4chars(int val1, int val2);

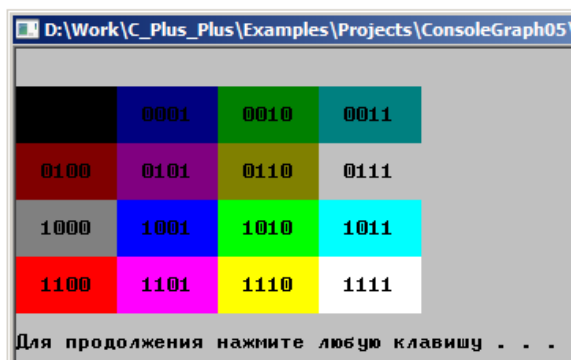
void main()
{
    HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(hStdOut, BACKGROUND_RED |
        BACKGROUND_GREEN | BACKGROUND_BLUE);
    system("CLS"); // заливка консольного окна серым цветом
    char strbox[]="          "; //восемь пробелов
    int k,IR,GB,rez;
    string strrez;
```

```

cout << "\n\n";
for(IR=0; IR<=3; IR++)
{
    for(k =0; k<3; k++)
    {
        for(GB=0; GB<=3; GB++)
        {
            strrez = into4chars(IR, GB);
            rez = (IR<<6) + (GB<<4);
            SetConsoleTextAttribute(hStdOut,rez);
            if (k==1)
                cout<<" " <<strrez<<"\t";
            else
                cout<<strbox ;
        }
        cout<<"\n";
    }
}
cout << "\n";
SetConsoleTextAttribute(hStdOut, 0x70);
system("pause");
}

// преобразование пары чисел 0<=val1<=3 и 0<=val2<=3 в
// четырехсимвольную «двоичную» строку
string into4chars(int val1, int val2)
{
    char chrez[20];
    int len = 4;
    int val = (val1 << 2) + val2;
    string sbfull(len, '0');
    string sbrez(_itoa(val, chrez, 2));
    sbfull.replace(len - sbrez.length(), sbrez.length(), sbrez);
    return sbfull;
}

```



На предыдущем рисунке показано консольное окно после выполнения программы. По вертикали таблицы меняются значения пары битов  $I_b R_b = \{00, 01, 10, 11\}$ , а по горизонтали – биты  $G_b B_b = \{00, 01, 10, 11\}$ . Соответственно этим значениям в программе введены переменные IR и GB, которые принимают значения от 0 до 3. Вспомогательная функция `into4chars` переводит пару чисел IR и GB в строковое представление

двоичного числа IRGB, которое затем выводится в соответствующей ячейке матрицы. Мы предполагаем, что с методами класса `string`, которые мы использовали в этой функции, вы знакомы.

■

Функция `SetConsoleTitle` устанавливает заголовок консольного окна. Например,

```
SetConsoleTitle(L"Заголовок консольного окна");
```

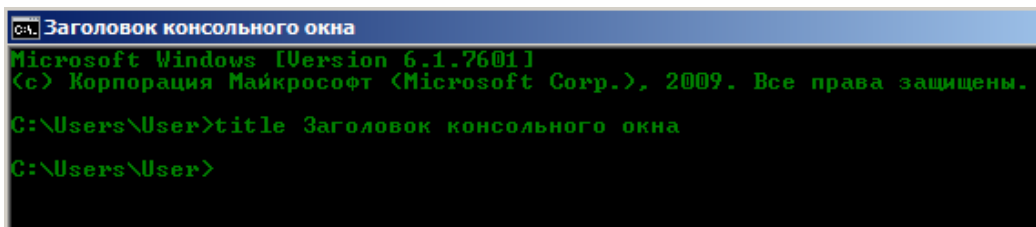
Обратите внимание на префикс `L`, который стоит перед строкой заголовка. О нем мы поговорим позже.

То же можно сделать командой

```
system("title Заголовок консольного окна");
```

Действительно, если бы вы захотели изменить заголовок консоли не из программы, а из самой консоли, то должны были бы подать команду

```
C:\...> title Заголовок консольного окна
```



Функция `SetConsoleCursorPosition` устанавливает положение текстового курсора в консольном окне. Например,

```
COORD cursorPos;  
cursorPos.X = 5;  
cursorPos.Y = 2;  
SetConsoleCursorPosition(hStdOut, cursorPos);
```

Первым аргументом она принимает дескриптор консольного окна, а вторым – переменную типа `COORD` (это структура с двумя полями `X` и `Y`), в которой должны находиться координаты текстового курсора.

Функция `GetConsoleScreenBufferInfo` извлекает информацию о буфере консольного окна. Например,

```
HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);  
CONSOLE_SCREEN_BUFFER_INFO csbiInfo;  
GetConsoleScreenBufferInfo(hStdOut, &csbiInfo);
```

Первый аргумент является дескриптором консольного окна (`HANDLE`), второй – структурой `CONSOLE_SCREEN_BUFFER_INFO`, предназначенной для хранения информации об окне консоли. В частности, ее поле `csbiInfo.wAttributes` типа `WORD` получит в своих битах информацию о текущих цветах текста и фона консольного окна. А поля

```
csbiInfo.srWindow.Left  
csbiInfo.srWindow.Top  
csbiInfo.srWindow.Right  
csbiInfo.srWindow.Bottom
```

содержат значения текстовых координат левого верхнего и правого нижнего углов консольного окна.

Функцию `GetConsoleScreenBufferInfo` рекомендуется использовать в начале программы, чтобы потом, если нужно, восстановить исходные значения.

В следующем примере мы демонстрируем работу описанных сейчас функций, дважды отображая «текстовый» прямоугольник разными цветами.

### Пример 3.

```
#include <windows.h>
#include <iostream>
using namespace std;

CONSOLE_SCREEN_BUFFER_INFO csbiInfo;
void drawBox(HANDLE hStdOut, COORD Pos);

void main()
{
    SetConsoleTitle(L"Hello World!"); //заголовок окна
    HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
    GetConsoleScreenBufferInfo(hStdOut, &csbiInfo);
    WORD wOldColorAttrs;
    wOldColorAttrs = csbiInfo.wAttributes; // старые атрибуты

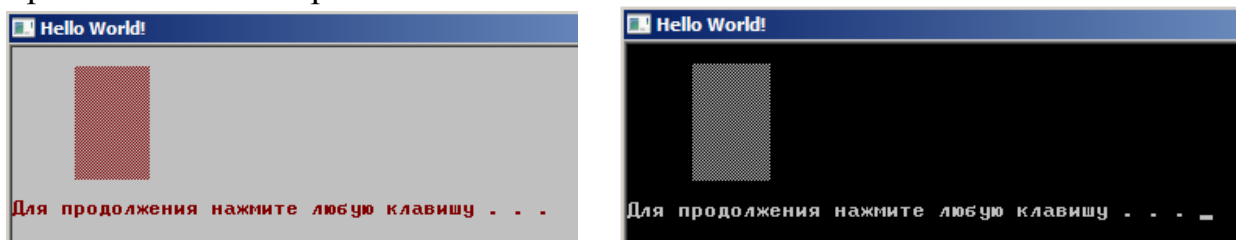
    SetConsoleTextAttribute(hStdOut, BACKGROUND_RED |
        BACKGROUND_GREEN | BACKGROUND_BLUE | FOREGROUND_RED);
    system("CLS"); // заливка консольного окна серым цветом

    COORD cursorPos = {5,1}; // к-ты левого верхнего угла
    drawBox(hStdOut, cursorPos); // Печать прямоугольника

    cursorPos = {0,8};
    SetConsoleCursorPosition(hStdOut, cursorPos);
    system("pause");
    // Возвращаем исходные цвета
    SetConsoleTextAttribute(hStdOut, wOldColorAttrs);
    system("CLS"); //очищаем/перекрашиваем все окно
    cursorPos = {5,1};
    drawBox(hStdOut, cursorPos); // Снова рисуем прямоугольник
    cursorPos = {0,8}; // Новое положение текстового курсора
    SetConsoleCursorPosition(hStdOut, cursorPos);
    system("pause");
}

void drawBox(HANDLE hStdOut, COORD Pos)
{
    for (int i = 0; i < 6; i++) {
        SetConsoleCursorPosition(hStdOut, Pos);
        for (int j = 0; j < 6; j++) cout << char(177);
        Pos.Y += 1;
    }
}
```

Первое состояние консольного окна показано на следующем рисунке слева. Справа показано второе состояние окна.



Функция `drawBox` рисует прямоугольник из 6 x 6 символов с кодом 177, используя второй аргумент для позиционирования его левого верхнего угла.

Когда открывается консольное окно, то для хранения символов, напечатанных в этом окне, заранее выделяется некоторая область памяти, которая называется буфером окна консоли (`console screen buffer`). Фактически буфер, это одномерный массив символов и их атрибутов. Каждый элемент этого массива имеет тип `CHAR_INFO`, являющийся структурой, хранящей код символа (ASCII или Unicode) и его атрибуты. Нулевой элемент этого массива соответствует символьной ячейке (0,0) в окне консоли, следующий – ячейке (0,1), и далее последовательно до конца текстовой строки окна. Последующий элемент массива соответствует точке/ячейке (1,0). Например, если в строке консоли 80 текстовых ячеек, то 80 – й элемент массива будет хранить код символа, отображаемого в этой ячейке (а также его атрибут). В 160 – ом элементе будет содержаться код символа (и атрибут), напечатанный в ячейке (2,0) и т.д. Имеются функции, которые печатают символы или их атрибуты в этот буфер.

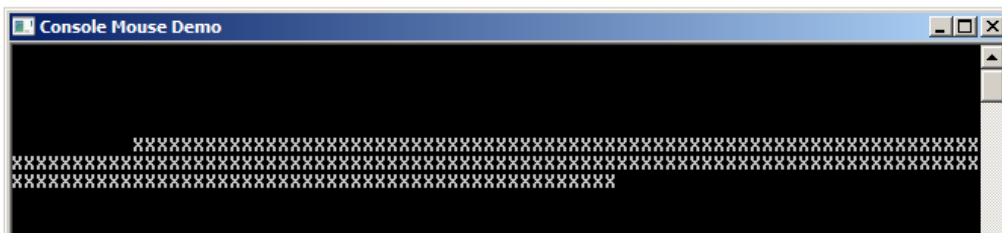
Функция `FillConsoleOutputCharacter` многократно печатает заданный символ в буфер консоли, начиная с указанной позиции

```
BOOL WINAPI FillConsoleOutputCharacter(  
    HANDLE hConsoleOutput, // идентификатор окна консоли  
    TCHAR cCharacter,      // символ, который будет печататься  
    DWORD nLength,        // количество символьных ячеек в  
                          // которые символ должен быть напечатан  
    COORD dwWriteCoord,   // COORD структура, которая  
                          // задает координаты первой ячейки  
    LPDWORD lpNumberOfCharsWritten // Указатель на  
                          // переменную, которая получит количество  
                          // символов, напечатанных в действительности  
);
```

Например, если переменная `wHnd` является идентификатором окна консоли, то следующий фрагмент кода выводит подряд 200 символов 'X', начиная с 10 – й ячейки 5 – й строки.

```
DWORD cWrittenChars;  
FillConsoleOutputCharacter(wHnd, (TCHAR) 'X',  
    200, {10, 5}, &cWrittenChars);
```





У функции `FillConsoleOutputCharacter` есть функция – двойник `FillConsoleOutputAttribute`, которая устанавливает атрибуты цвета для заданного количества символьных ячеек, начиная с указанного места в буфере консоли.

```

BOOL WINAPI FillConsoleOutputAttribute(
    HANDLE hConsoleOutput, // идентификатор окна консоли
    WORD wAttribute, // заполняющий атрибут
    DWORD nLength, // количество ячеек, которые
                    // должны быть заполнены атрибутом
    COORD dwWriteCoord, // координаты первой ячейки
    LPDWORD lpNumberOfAttrsWritten // Указатель на
                                    // переменную, которая получит количество
                                    // символьных ячеек, чьи атрибуты
                                    // установлены в действительности
);

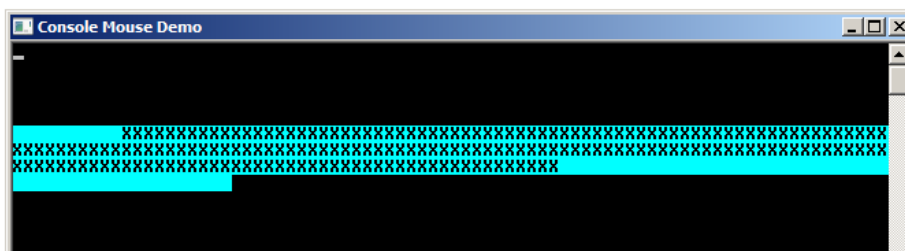
```

Например, следующий фрагмент кода устанавливает бирюзовый цвет ячеек, начиная с 5 – й по 7 – ю строки и еще 20 ячеек 8 – й строки.

```

DWORD cWrittenChars;
FillConsoleOutputCharacter(wHnd, (TCHAR) 'X', 200,
                           {10, 5}, &cWrittenChars);
WORD attr = BACKGROUND_BLUE |
            BACKGROUND_GREEN |
            BACKGROUND_INTENSITY;
FillConsoleOutputAttribute(wHnd, attr, 260,
                           {0, 5}, &cWrittenChars);

```



Используя эти функции, напечатав многократно пробел, можно написать собственную функцию очистки консольного окна. Следующий пример демонстрирует эту возможность.

#### Пример 4.

```

#include <conio.h>
#include <iostream>
#include <windows.h>
using namespace std;

```

```

void cls(HANDLE hConsole)
{
    COORD coordScreen = {0,0}; // начальное положение курсора
    CONSOLE_SCREEN_BUFFER_INFO csbi;
    DWORD dwConSize;
    DWORD cCharsWritten;
    GetConsoleScreenBufferInfo(hConsole, &csbi);
    dwConSize = csbi.dwSize.X * csbi.dwSize.Y; // количество
        // символьных ячеек в буфере консоли
    FillConsoleOutputCharacter(hConsole, (TCHAR) ' ',
        dwConSize, coordScreen, &cCharsWritten);
    FillConsoleOutputAttribute(hConsole, csbi.wAttributes,
        dwConSize, coordScreen, &cCharsWritten);
    SetConsoleCursorPosition(hConsole, coordScreen);
}

int main() {
    //получение дескриптора стандартного устройства вывода
    HANDLE wHnd = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTitle(L"Demo Console Cls"); // Заголовок консоли
    cout << "Hello\n";
    system("pause"); // Наблюдаем текст в окне
    SetConsoleTextAttribute(wHnd,
        BACKGROUND_RED |
        BACKGROUND_GREEN |
        BACKGROUND_BLUE |
        FOREGROUND_RED);
    cls(wHnd); // очищаем окно
    int iKey = 67;
    // Цикл до тех пор пока не нажата клавиша ESC
    while (iKey != 27) { if (_kbhit()) iKey = _getch(); }
}

```

Последняя строка приведенного кода выполняет задержку до тех пор, пока не будет нажата клавиша ESC, имеющая код 27. Функция `_kbhit()` возвращает истину, если нажата какая – либо клавиша на клавиатуре. В противном случае возвращается 0. При этом код нажатой клавиши не удаляется из входного буфера. Если клавиша была нажата, то функция `_getch()` читает ее код в переменную `iKey` без отображения в консоли. ■

В следующем примере мы управляем положением прямоугольника с помощью клавиатуры. Точнее, мы рисуем прямоугольник такой, как в примере 3, и сдвигаем его с помощью клавиш управления курсором.

### Пример 5.

```

#include <windows.h>
#include <iostream>
#include <conio.h>
using namespace std;

```

```

#define KEY_ARROW_UP          72
#define KEY_ARROW_RIGHT      77
#define KEY_ARROW_DOWN       80
#define KEY_ARROW_LEFT       75
enum direction {RIGHT,LEFT,UP,DOWN };

class textRect {
    enum style {HOR,VERT};
    HANDLE hStdOut;
    COORD leftTop;          // положение левого верхнего угла
    char c = char(177); //символ рисования
    char d = ' ';          //символ затирания
    int size = 8;          //ширина и высота прямоугольника
    void drawline(COORD cursorPos,style dir,char ch);
public:
    textRect(HANDLE hStdOut, COORD Pos)
        {this->hStdOut = hStdOut; leftTop = Pos; draw();}
    void draw();
    void move(direction dir);
    short right() { return leftTop.X+size; }
    short left() { return leftTop.X; }
    short top() { return leftTop.Y; }
    short bottom() { return leftTop.Y+size; }
};

void main()
{
    SetConsoleTitle(L"Box moving!"); // вывод заголовка окна
    HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
    // определение ширины и высоты консольного окна
    CONSOLE_SCREEN_BUFFER_INFO csbInfo;
    GetConsoleScreenBufferInfo(hStdOut, &csbInfo);
    COORD cursorPos = { 5, 2 }; // положение текстового курсора
    //создаем объект прямоугольника
    textRect rect(hStdOut, cursorPos);

    int iKey = 67;
    while (iKey != 27) // Выход по клавише ESC
    {
        if (_kbhit())
        {
            iKey = _getch();
            switch (iKey)
            {
                case KEY_ARROW_UP:
                    if (rect.top() <=csbInfo.srWindow.Top)
                        cout << "\a";
                    else rect.move(UP);
                    break;
                case KEY_ARROW_RIGHT:
                    if (rect.right() > csbInfo.srWindow.Right)
                        cout << "\a";
                    else rect.move(RIGHT);
                    break;
            }
        }
    }
}

```

```

    case KEY_ARROW_DOWN:
        if (rect.bottom() > csbInfo.srWindow.Bottom)
            cout << "\a";
        else rect.move(DOWN);
        break;
    case KEY_ARROW_LEFT:
        if (rect.left() <= csbInfo.srWindow.Left)
            cout << "\a";
        else rect.move(LEFT);
        break;
    case 120: //клавиша x
    case 88: //клавиша X
        exit(0); //завершение программы
    }
}
}

// Печать прямоугольника
void textRect::draw()
{
    COORD pos = leftTop;
    for (int j = 0; j < size; j++) {
        drawline(pos, VERT, c); //печать вертикальной линию
        pos.X += 1;
    }
    // Текстовый курсор в начало консоли
    SetConsoleCursorPosition(hStdOut, {0,0});
}

// перепечатаваем сдвинутый прямоугольник
void textRect::move(direction dir)
{
    COORD pos = leftTop;
    switch (dir)
    {
        case RIGHT:
            drawline(pos, VERT, d); //стереть верт. линию
            pos = leftTop;
            pos.X += size;
            drawline(pos, VERT, c); //напечатать верт. линию
            leftTop.X += 1;
            break;
        case LEFT:
            pos.X += size-1;
            drawline(pos, VERT, d);
            pos = leftTop;
            pos.X -= 1;
            drawline(pos, VERT, c);
            leftTop.X -= 1;
            break;
        case DOWN:
            drawline(pos, HOR, d); //стереть гор. линию
            pos = leftTop;

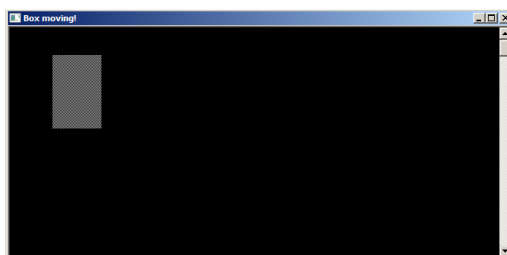
```

```

    pos.Y += size;
    drawline(pos, HOR, c); //напечатать гор. линию
    leftTop.Y += 1;
    break;
case UP:
    pos.Y += size - 1;
    drawline(pos, HOR, d); //стереть гор. линию
    pos = leftTop;
    pos.Y -= 1;
    drawline(pos, HOR, c); //напечатать гор. линию
    leftTop.Y -= 1;
    break;
default:
    break;
}
// Текстовый курсор в начало консоли
SetConsoleCursorPosition(hStdOut, {0,0});
}

// Если dir=VERT печатаем вертикальную линию символами ch
// иначе печатаем горизонтальную линию
void textRect::drawline(COORD cursorPos, style dir, char ch)
{
    for (int i = 0; i < size; i++) {
        SetConsoleCursorPosition(hStdOut, cursorPos);
        cout << ch;
        if (dir==VERT) cursorPos.Y+=1; else cursorPos.X+=1;
    }
}
}

```



Как и в предыдущем примере, вначале в функции `main()` устанавливается заголовок и определяется дескриптор консольного окна. Затем команда

```
GetConsoleScreenBufferInfo(hStdOut, &csbInfo);
```


извлекает информацию об окне. В программе мы используем поля структуры `csbInfo.srWindow`, которые содержат первоначальный размер консольного окна для контроля границ. Если прямоугольник будет находиться у границы окна, то команда

```
cout << "\a";
```

будет издаваться «гудок». Напомним, что если символ `'\a'` посылается на терминал, то ничего не печатается, а издается звуковой сигнал.

В программе мы создали класс – прямоугольник. Смысл данных, используемых в классе, пояснен в комментариях к ним. Конструктор принимает дескриптор окна, положение левого верхнего угла, и печатает/рисует прямоугольник символом `█`, код которого равен 177.

Затем мы входим в цикл, который завершается, когда будет нажата клавиша ESC. В цикле функция `_kbhit()` проверяет, была ли нажата какая-либо клавиша. Напомним, что функция `_kbhit()` возвращает не ноль, если нажата какая-либо клавиша на клавиатуре, и код клавиши не удаляется из входного буфера. Если клавиша была нажата, то функция `_getch()` читает ее код в переменную `iKey` без отображения символа в консоли. Прочитанный код `iKey` затем анализируется в операторе `switch`. В зависимости от кода клавиши (анализируется только код управляющих клавиш `→, ←, ↑, ↓`) вызывается метод `move` класса `textRect` с одним из четырех возможных аргументов, описанных в перечислении `direction`. Основную работу по перепечатаванию прямоугольника в новом положении выполняет функция `move()`.

Закрытая функция класса `drawline` печатает одну строку в текущем положении курсора `cursorPos` символом, заданным третьим аргументом. Направление строки вниз или вправо (т.е. строка вертикальная или горизонтальная) определяется вторым аргументом. Эта функция используется при отображении прямоугольника в методе `draw` и при его перепечатавании в методе `move()`. Например, при смещении прямоугольника вправо, стирается левый столбец символов прямоугольника (точнее печатается столбец из пробелов), а справа допечатывается новый столбец из символов . ■

В предыдущем примере мы управляли положением прямоугольника в окне консоли. Хотя мы «перепечатавали» только по одной его строке или столбцу, мерцание при сдвиге было заметно. Существует другой, более профессиональный способ «перепечатавания». У каждой консоли есть область памяти, в которой хранятся коды и атрибуты символов, показанные в ее окне. Мы уже говорили, что она называется буфером консоли (`console screen buffer`) и является массивом, элементы которого имеют тип `CHAR_INFO`. Имеется возможность быстро переносить данные в прямоугольный блок этого буфера. Для этого создается специальный одномерный массив, каждый элемент которого соответствует некоторой символьной ячейке в прямоугольной области окна. Этот массив состоит из элементов типа `CHAR_INFO`, содержащих информацию о коде символов в ячейках и их атрибуты. Вывод в консоль выполняется в два этапа. Вначале выполняется вывод информации в этот массив. Затем он отображается в окне консоли. Поскольку отображение блока (массива) выполняется очень быстро, то мерцание изображения практически незаметно.

Ниже мы приведем пример, использующий этот способ отображения. Но перед этим опишем несколько новых функций.

Размер консольного окна изменяется функцией

```
BOOL WINAPI SetConsoleWindowInfo(  
    HANDLE    hConsole, // идентификатор окна  
    BOOL      bAbsolute,  
    const SMALL_RECT *lpWindowSize  
);
```

Третий аргумент функции является указателем на переменную типа `SMALL_RECT`, в которой задаются координаты вершин консольного окна. Если второй аргумент равен `TRUE` то, если это возможно, окно консоли устанавливается точно заданного размера. Если второй параметр равен `FALSE`, то видимый размер консоли устанавливается по умолчанию, но появляются полосы прокрутки.

Как мы сказали выше, у каждого консольного окна есть свой буфер, который является одномерным массивом элементов типа `CHAR_INFO`. Размер этого массива не может быть меньше количества текстовых ячеек в окне консоли. Поэтому, после изменения размера консоли рекомендуется сразу устанавливать размер этого буфера. Это выполняется функцией

```
BOOL WINAPI SetConsoleScreenBufferSize(  
    HANDLE hConsole, // идентификатор окна  
    COORD dwSize  
);
```

Второй аргумент является переменной типа `COORD`, в которой задаются новые размеры буфера консоли (количество символьных столбцов и строк).

Для отображения прямоугольного блока данных в буфер консоли используется функция

```
BOOL WINAPI WriteConsoleOutput(  
    HANDLE hConsoleOutput, // идентификатор консоли  
    const CHAR_INFO *lpBuffer, // указатель на блок данных  
    COORD dwBufferSize, // размер блока данных {col,row}  
    COORD dwBufferCoord, // вершина в блока  
    PSMAALL_RECT lpWriteRegion // область буфера для перезаписи  
);
```

Второй аргумент `lpBuffer` обычно является именем одномерного массива, содержащего данные, переносимые в буфер консоли. Третий аргумент `dwBufferSize` является переменной типа `COORD`, содержащей количество столбцов и строк в переносимом блоке. При переносе прямоугольного блока из него можно выбрать не все данные, а только прямоугольный фрагмент, координаты левой верхней вершина которого задаются четвертым аргументом `dwBufferCoord`. В последнем аргументе `lpWriteRegion` передаются координаты левой верхней и правой нижней прямоугольной области буфера консоли, в которую переносится блок данных. Вот пример использования этой функции

### Пример 6.

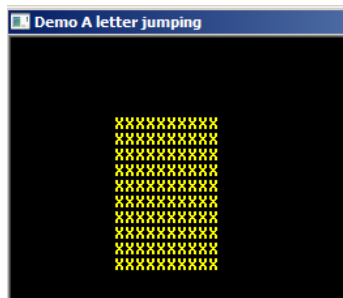
```
#include <conio.h>  
#include <iostream>  
#include <windows.h>  
using namespace std;  
  
int main() {  
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);  
    SetConsoleTitle(L"Draw block");  
    const int row = 10; // количество строк блока  
    const int col = 10; // количество столбцов блока
```

```

CHAR_INFO consBuffer[col * row]; // массив блока данных
WORD attrib = FOREGROUND_RED | FOREGROUND_GREEN |
              FOREGROUND_INTENSITY; //желтый текст на черном фоне
// заполнение массива блока данных
for (int i = 0; i<col*row; ++i) {
    consBuffer[i].Char.UnicodeChar = 'X';
    consBuffer[i].Attributes = attrib;
}
COORD charPosition={0,0}; // точка в блоке
SMALL_RECT writeArea={10,5,19,14};
COORD bufferSize = {col,row};
WriteConsoleOutput(hConsole, consBuffer, bufferSize,
                  charPosition, &writeArea);

getch();
}

```



В этом примере мы задали размеры блока (row и col) и создали массив consBuffer блока данных. Затем в символьные поля элементов этого массива занесли символ 'X' и установили атрибут цвета. Прямоугольный блок символов 'X' размером {col,row}, начиная с его точки {0,0}, мы поместили в область консоли {10,5,19,14}, где 10,5 – координаты левой верхней вершины, а 19,14 – правой нижней.

■

В следующем примере мы используем функцию WriteConsoleOutput для многократной печати «матрицы» изображения буквы А в случайно расположенных местах консоли. Задержка между кадрами выполняется функцией

```
VOID WINAPI Sleep(DWORD dwMilliseconds);
```

которая останавливает выполнение текущего потока на заданное количество миллисекунд.

В примере в разных случайно выбранных местах консоли 20 раз выводится «изображение» буквы А.

Перед отображением буквы вся консоль очищается. Для этого создается блок данных типа CHAR\_INFO, размер которого в точности равен количеству символьных ячеек в окне консоли. Он заполняется пробелами и соответствующими атрибутами и затем функцией WriteConsoleOutput переносится в буфер консоли. Тем самым, выполняется полная очистка окна консоли. Эту последовательность команд мы выделили в отдельную функцию cls2(...).



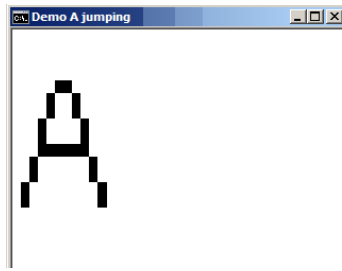
Матрица изображения буквы А задается массивом нулей и единиц. Для ее отображения создается блок данных, размер которого равен размеру матрицы. Этот блок мы заполняем пробелами, но для тех элементов, которым в матрице соответствует единица, мы меняем атрибут цвета. В результате отображения этого блока в случайной точке консоли, получается изображение буквы. Эту последовательность команд мы выделили в функцию drawA(...).

**Пример 7.** Печать буквы А в буфер с последующим отображением в случайно выбранной точке консоли.

```
#include <conio.h>
#include <iostream>
#include <windows.h>
#include <time.h>
using namespace std;
const int col = 80; // количество текстовых ячеек в строке
const int row = 50; // количество строк
void drawA(HANDLE hWnd, int posX, int posY);
void cls2(HANDLE hConsole);

int main() {
    HANDLE hWnd = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTitle(L"Demo A jumping"); //Вывод заголовка консоли
    SMALL_RECT windowSize={0,0,col-1,row-1}; // Координаты вершин
                                                // консольного окна
    SetConsoleWindowInfo(hWnd, TRUE, &windowSize); //Задание
                                                // размеров окна консоли
    COORD bufferSize={col,row}; // размера буфера
    SetConsoleScreenBufferSize(hWnd, bufferSize); // Изменение
                                                // размера внутреннего буфера окна консоли
    // Установка атрибутов консоли (белый фон)
    SetConsoleTextAttribute(hWnd, BACKGROUND_BLUE |
                                BACKGROUND_GREEN |
                                BACKGROUND_RED |
                                BACKGROUND_INTENSITY);
    srand((unsigned)time(NULL)); // инициализация генератора
                                // случайных чисел
    CHAR_INFO consBuffer[col*row]; // Создание буфера консоли

    int rX, rY; // (x,y) координаты вершины блока в окне
    for (int i = 0; i<20; i++) {
        cls2(hWnd, consBuffer, bufferSize); // очистка окна
        rX = rand()%(col-10); //случайные значения координат
        rY = rand()%(row-10);
        drawA(hWnd, rX, rY); // рисуем букву А
        Sleep(400); // задержка на 0.4 секунда
    }
    int iKey = 67;
    cout << "Press ESC for exit\n";
    // задержка до тех пор, пока не нажата ESC
    while (iKey != 27) { if (_kbhit()) iKey = _getch(); }
}
```



Ниже приведен код функции рисования буквы А. Функция содержит двумерный массив из нулей и единиц – матрицу изображения. Нулю отвечает цвет фона, единице – цвет символов.

```
// Печатаем изображение буквы А размером 10 x 10 символьных ячеек
void drawA(HANDLE wHnd, int posx, int posy)
{
    const int rowA = 10;
    const int colA = 10;
    int pictLetterA[colA][rowA] = {
        { 0, 0, 0, 0, 1, 1, 0, 0, 0, 0 },
        { 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 },
        { 0, 0, 0, 1, 0, 0, 1, 0, 0, 0 },
        { 0, 0, 1, 0, 0, 0, 0, 1, 0, 0 },
        { 0, 0, 1, 0, 0, 0, 0, 1, 0, 0 },
        { 0, 0, 1, 1, 1, 1, 1, 1, 0, 0 },
        { 0, 1, 0, 0, 0, 0, 0, 0, 1, 0 },
        { 0, 1, 0, 0, 0, 0, 0, 0, 1, 0 },
        { 1, 0, 0, 0, 0, 0, 0, 0, 0, 1 },
        { 1, 0, 0, 0, 0, 0, 0, 0, 0, 1 },
    };
    CONSOLE_SCREEN_BUFFER_INFO csbiInfo;
    GetConsoleScreenBufferInfo(wHnd, &csbiInfo); // атрибуты консоли
    CHAR_INFO picBufferA[colA * rowA]; // Создаем символьный
    // буфер для печати буквы А
    for (int i = 0; i < colA * rowA; ++i) {
        picBufferA[i].Char.UnicodeChar = ' ';
        picBufferA[i].Attributes = csbiInfo.wAttributes; // в буфер
        // буквы А переносим атрибуты консоли
    }
    for (int i = 0; i < rowA; i++)
        for (int j = 0; j < colA; j++) {
            if (pictLetterA[i][j]==1)
                picBufferA[i*colA+j].Attributes = 0;
        }
    // прямоугольник для рисования буквы
    SMALL_RECT writeArea = { posx, posy, posx+colA, posy+rowA };
    // переносим блок данных в буфер консоли
    WriteConsoleOutput(wHnd, picBufferA,
        {colA,rowA}, {0,0}, &writeArea);
}
```

Функция `cls2(...)` очищает буфер консоли. Точнее она переписывает в него блок данных, состоящий из пробелов, атрибуты которых совпадают с атрибутами, взятыми из окна консоли. При этом размер блока в точности соответствует размеру буфера консоли.

```

void cls2(HANDLE hConsole)
{
    CHAR_INFO consBuffer[col * row];    // Создание блока для всего
                                        // буфера консоли
    CONSOLE_SCREEN_BUFFER_INFO csbiInfo;
    GetConsoleScreenBufferInfo(hConsole, &csbiInfo); //атриб. консоли

    for (int i = 0; i < col * row; ++i) {
        consBuffer[i].Char.AsciiChar = ' ';
        consBuffer[i].Attributes = csbiInfo.wAttributes;
    }
    COORD charPosition={0,0}; // точка в блоке
    SMALL_RECT writeArea={0,0,col-1,row-1};
    COORD bufferSize = {col,row};
    WriteConsoleOutputA(hConsole, consBuffer, bufferSize,
                        charPosition, &writeArea);
}

```

Обратите внимание на завершающий символ А в имени функции WriteConsoleOutputA. Многие функции Windows, принимающие в качестве одного из аргументов массив символов (строку), имеют две формы. Одна – заканчивается символом W, вторая – символом A. Например,

```

WriteConsoleOutputA(...);
WriteConsoleOutputW(...);

```

Они отличаются типом передаваемых аргументов – строк, которые могут быть однобайтовыми или двухбайтовыми. В документации описана одна функция WriteConsoleOutput(...). Если вызвать функцию без завершающего символа А или W, то обычно вызывается вторая «двухбайтовая» версия. В нашем случае это приведет к ошибке, поскольку в строке

```

consoleBuffer[offsetPos].Char.AsciiChar = ' ';

```

мы задаем однобайтовый ASCII код пробела и вызов функции WriteConsoleOutputW приведет к ошибке или некорректному отображению. Для сравнения посмотрите как мы записали пробелы в блок данных в теле функции cls2(...).

```

picBufferA[i].Char.UnicodeChar = ' ';
.....
WriteConsoleOutput(...)

```

У поля Char структуры CHAR\_INFO есть поля AsciiChar и UnicodeChar. Если вы заполнили поле UnicodeChar, то вызывайте функцию WriteConsoleOutput(.), иначе вызывайте WriteConsoleOutputA(.).

Функции Windows, которые заканчиваются символом A используют ANSI кодировку символов (т.е. однобайтовые коды, например, из таблицы cp1251). Функции Windows, которые заканчиваются символом W используют UNICODE стандарт, т.е. двухбайтовые строки символов. В справочной системе дается описание таких функций без окончания, а препроцессор определяет, которую из функций использовать. Например, для функции SetWindowText, которая

изменяет заголовок окон Windows, прототип в справочной системе дается в общем виде

```
BOOL SetWindowText(HWND hwnd, LPCTSTR lpText);
```

Но заголовочный файл определяет общее имя в виде макроса

```
#ifdef UNICODE
#define SetWindowText SetWindowTextW
#else
#define SetWindowText SetWindowTextA
#endif // !UNICODE
```

Препроцессор заменяет имя в соответствии с используемой кодировкой. Также в заголовочном файле приводятся два прототипа

```
BOOL SetWindowTextA( HWND hwnd, LPCSTR lpText);
BOOL SetWindowTextW( HWND hwnd, LPCWSTR lpText);
```

один для кодовых страниц Windows, второй – для Unicode. Различить такие функции вы также можете по типу аргумента. Тип LPCSTR используется для строковых указателей в кодовых страницах Windows и LPCWSTR – для Unicode. В общем прототипе функции такой аргумент имеет тип LPCTSTR. ■

В следующем примере мы создадим примитивный консольный графический редактор, который будет «рисовать» с помощью мыши в текстовом режиме консоли, используя три символа: ■, ■■■ и пробел. Алгоритм «графического редактора» состоит из следующей последовательности действий:

1. Задаем фиксированный размер окна консоли.
2. Создаем блок данных по размеру консоли, заполняем его пробелами с атрибутами – черные символы на белом фоне и переносим данные блока в буфер консоли. Потом в этот блок данных мы будем печатать символы «графики».
3. Запускаем бесконечный цикл

```
bool isRunning = true; // флаг продолжения работы
while(isRunning) {
    // Анализируем входные события консоли:
    - Если событие сгенерировано клавиатурой, то проверяем код
      нажатой клавиши и выполняем соответствующее действие;
    - Если событие сгенерировано мышью, то проверяем, какая из
      кнопок мыши нажата и печатаем в блок данных соответствующий
      символ; затем переносим блок в буфер консоли;
}
```

Но вначале мы должны понять, как программа будет обрабатывать сообщения, поступающие от мыши. Для этого нам понадобится использовать несколько функций Windows.

Функция `GetNumberOfConsoleInputEvents` определяет количество необработанных сообщений во входном буфере, включая сообщения клавиатуры, мыши и сообщения об изменении размеров окна (не путайте входной буфер и текстовый буфер консоли).

```
BOOL WINAPI GetNumberOfConsoleInputEvents(
    HANDLE hConsoleInput, // дескриптор входного буфера
    LPDWORD lpNumberOfEvents // указатель на количество
```

```
        // непрочитанных консольных сообщений
```

```
);
```

Дескриптор входного буфера получается аналогично дескриптору консоли. Например, следующий фрагмент определяет количество необработанных сообщений во входном буфере.

```
HANDLE rHnd = GetStdHandle(STD_INPUT_HANDLE);
DWORD numEvents = 0; // Кол-во непрочитанных сообщений
GetNumberOfConsoleInputEvents(rHnd, &numEvents);
```

Если есть необработанные сообщения, т.е. `numEvents!=0`, то их можно извлечь из входного буфера.

Функция `ReadConsole` выбирает только сообщения клавиатуры. Ее здесь мы не рассматриваем. Если нужны все сообщения (клавиатуры и мыши), то используется функция `ReadConsoleInput`, которая извлекает все сообщения из буфера ввода консоли. Она имеет следующий формат

```
BOOL WINAPI ReadConsoleInput(
    HANDLE          hConsoleInput, // дескриптор входного буфера
    PINPUT_RECORD  lpBuffer,
    DWORD          nLength,        // количество сообщений
    LPDWORD        lpNumberOfEventsRead
);
```

Второй аргумент `lpBuffer` является указателем на массив элементов типа `INPUT_RECORD`, в который извлекаются все данные из входного буфера. Третий аргумент `nLength` представляет количество сообщений во входном буфере, полученное ранее функцией `GetNumberOfConsoleInputEvents`. Аргумент `lpNumberOfEventsRead` является указателем на переменную, принимающую количество реально прочитанных сообщений.

После того, как в массив `lpBuffer` будут перенесены данные сообщений, их можно анализировать. Каждый элемент массива является структурой типа `INPUT_RECORD`. Одно из полей этой структуры `EventType` содержит тип сообщения: клавиатуры (значение `KEY_EVENT`) или мыши (значение `MOUSE_EVENT`). Другие возможные события мы здесь анализировать не будем.

Анализ сообщений консоли (3 –й пункт алгоритма) может выглядеть примерно так:

```
DWORD numEvents = 0; // Кол-во непрочитанных сообщений
bool isRunning = true; // флаг продолжения работы
DWORD numEventsRead = 0; // Кол-во прочитанных сообщений
while(isRunning) {
    GetNumberOfConsoleInputEvents(hWnd, &numEvents);
    if(numEvents!= 0){
        // выделение памяти под массив сообщений
        INPUT_RECORD *eventBuffer =new INPUT_RECORD[numEvents];
        ReadConsoleInput(hWnd, eventBuffer,
            numEvents, &numEventsRead);
        // перебор сообщений в массиве eventBuffer
        for (DWORD i = 0; i<numEventsRead; ++i) {
```

```

    if(eventBuffer[i].EventType==KEY_EVENT) {
        ..... обработка нажатий клавиш клавиатуры ....
    }
    else if(eventBuffer[i].EventType==MOUSE_EVENT) {
        ..... обработка нажатий кнопок мыши ....
    }
}
delete[] eventBuffer; // освобождение памяти
}
}

```

Покажем, как можно обрабатывать нажатие клавиш клавиатуры. Если нажата кнопка клавиатуры, то  $i$  – й элемент буфера ввода `eventBuffer[i]` в поле `EventType` содержит значение константы `KEY_EVENT`. Тогда можно обращаться к полю `Event` этого элемента и к полям этого поля. Мы можем узнать значение виртуального кода нажатой клавиши с помощью значения `eventBuffer[i].Event.KeyEvent.wVirtualKeyCode` или ее ASCII код с помощью значения

```
eventBuffer[i].Event.KeyEvent.uChar.AsciiChar
```

Виртуальные коды – это номера, которые использует система для идентификации клавиш. Таким образом, проверка того, что нажата клавиша ESC может выглядеть так

```

if(eventBuffer[i].Event.KeyEvent.wVirtualKeyCode==VK_ESCAPE) {
    ..... // сброс флага продолжения работы
}



```

Проверка того, что нажата клавиша с символом 'c' будет выглядеть так

```

if (eventBuffer[i].Event.KeyEvent.uChar.AsciiChar=='c') {
    ..... // очистка окна консоли
}



```

Покажем, как мы будем обрабатывать нажатие кнопок мыши. Если нажата кнопка мыши, то элемент массива `eventBuffer[i]` в поле `EventType` содержит значение константы `MOUSE_EVENT`. Тогда можно обращаться к полю `Event` элемента и к полю `MouseEvent` этого поля. Но прежде, чем определять, какая кнопка мыши нажата, следует узнать и использовать (если нужно) координаты мыши. В нашем «графическом редакторе» мы будем рисовать/печатать символ  (или символ ) в соответствующее место текстового буфера консоли. Команда, определяющая индекс `consPos` в буфере консоли по координатам места щелчка, может иметь следующий вид

```

int consPos=eventBuffer[i].Event.MouseEvent.dwMousePosition.X
    + 80*eventBuffer[i].Event.MouseEvent.dwMousePosition.Y;

```

Затем, в зависимости от нажатой кнопки мыши, в элемент `consBuffer[consPos]` буфера консоли мы будем записывать один из трех символов ,  или пробел (очистка). Соответствующий фрагмент кода может иметь следующий вид



```

if(eventBuffer[i].Event.MouseEvent.dwButtonState &
    FROM_LEFT_1ST_BUTTON_PRESSED) {
    consBuffer[consPos].Char.AsciiChar = (char)0xDB;
    WriteConsoleOutputA(wHnd, consBuffer, charBufSize,
        characterPos, &writeArea);
}
else if(eventBuffer[i].Event.MouseEvent.dwButtonState &
    RIGHTMOST_BUTTON_PRESSED) {
    consBuffer[consPos].Char.AsciiChar = (char)0xB1;
    WriteConsoleOutputA(wHnd, consBuffer, charBufSize,
        characterPos, &writeArea);
}
else if (eventBuffer[i].Event.MouseEvent.dwButtonState &
    FROM_LEFT_2ND_BUTTON_PRESSED) {
    consBuffer[consPos].Char.AsciiChar = ' ';
    WriteConsoleOutputA(wHnd, consBuffer, charBufSize,
        characterPos, &writeArea);
}

```

Теперь перейдем к примеру.

### Пример 8. Примитивный «графический редактор».

В программе щелчок левой кнопкой мыши печатает в консоль символ , щелчок правой – печатает символ , нажатие колесика (средней кнопки мыши) печатает пробел, т.е. стирает символ, стоявший в соответствующей текстовой ячейке. Выбор символа 'g' делает фон окна серым, выбор символа 'w' возвращает белый фон. Символа 'c' – удаляет рисунок (очищает консоль).

Вот полный текст примитивного «графического редактора».

```

#include <conio.h>
#include <Windows.h>

const int col = 80;    // ширина окна консоли
const int row = 50;   // высота консоли

int main()
{
    HANDLE wHnd = GetStdHandle(STD_OUTPUT_HANDLE);
    HANDLE rHnd = GetStdHandle(STD_INPUT_HANDLE);
    SetConsoleTitle(L"Console Mouse Demo");

    // --- 1. Задаем размер консоли и его внутреннего буфера ---
    SMALL_RECT windowSize={0,0,col-1,row-1};
    SetConsoleWindowInfo(wHnd,TRUE,&windowSize);
    COORD bufferSize={col,row};    // размеры внутреннего буфера
    // Изменение размера внутреннего буфера окна консоли
    SetConsoleScreenBufferSize(wHnd, bufferSize);

    // --- 2. Создаем и заполняем блок данных по размеру консоли
    CHAR_INFO consBuffer[col * row];    // Создание буфера консоли
    for (int i = 0; i < col * row; ++i) {
        // Заполнение буфера консоли пробелами
        consBuffer[i].Char.AsciiChar = ' ';
    }
}

```

```

    consBuffer[i].Attributes =
        BACKGROUND_BLUE | BACKGROUND_GREEN |
        BACKGROUND_RED | BACKGROUND_INTENSITY;
}
COORD charPosition = {0,0}; // положение курсора после
                            // каждого обновления буфера
SMALL_RECT writeArea = windowSize;
// Перенос данных буфера на дисплей
WriteConsoleOutputA(wHnd, consBuffer, bufferSize,
    charPosition, &writeArea);

// --- 3. Чтение и анализ буфера ввода. Реакция на события.
DWORD numEvents = 0; // Количество непрочитанных сообщений
DWORD numEventsRead = 0; // Количество прочитанных сообщений
bool isRunning = true; // флаг продолжения работы

//Ели isRunning=false, то программа завершается
while (isRunning) {
    // Определить количество событий, связанных с консолью
    GetNumberOfConsoleInputEvents(rHnd, &numEvents);
    if (numEvents != 0) {
        // выделение памяти для хранения данных о событиях
        INPUT_RECORD *eventBuffer=new INPUT_RECORD[numEvents];
        // Извлечение данных во временный буфер событий и
        // определение количества событий numEventsRead
        ReadConsoleInput(rHnd, eventBuffer,
            numEvents, &numEventsRead);
        // Цикл по всем извлеченным событиям
        for (DWORD i = 0; i < numEventsRead; ++i) {
            // нажата кнопка клавиатуры
            if (eventBuffer[i].EventType == KEY_EVENT) {
                // анализ виртуальных кодов клавиш
                switch (eventBuffer[i].Event.KeyEvent.wVirtualKeyCode)
                {
                    case VK_ESCAPE:
                        isRunning = false;
                        break;
                }
                // анализ кодов символов
                switch (eventBuffer[i].Event.KeyEvent.uChar.AsciiChar)
                {
                    case 'c': // полная очистка окна
                        for (int i=0; i<col*row; ++i)
                            consBuffer[i].Char.AsciiChar = ' ';
                        WriteConsoleOutputA(wHnd, consBuffer, bufferSize,
                            charPosition, &writeArea);
                        break;
                    case 'g': // серый фон
                        for (int i=0; i<col*row; ++i) {
                            consBuffer[i].Attributes = BACKGROUND_BLUE |
                                BACKGROUND_GREEN | BACKGROUND_RED;
                        }
                }
            }
        }
    }
}

```

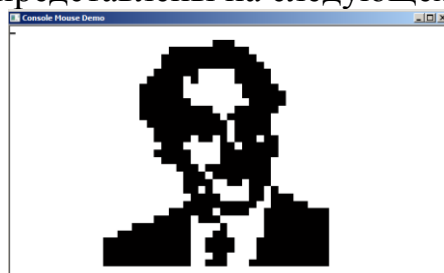


```

        WriteConsoleOutputA(wHnd, consBuffer, bufferSize,
                            charPosition, &writeArea);
    break;
case 'w': // белый фон
    for (int i=0; i<col*row; ++i) {
        consBuffer[i].Attributes = BACKGROUND_BLUE |
            BACKGROUND_GREEN | BACKGROUND_RED |
            BACKGROUND_INTENSITY;
    }
    WriteConsoleOutputA(wHnd, consBuffer, bufferSize,
                        charPosition, &writeArea);
    break;
}
}
else if (eventBuffer[i].EventType == MOUSE_EVENT) {
    // вычисление индекса в текстовом буфере консоли
    int consBuffPos =
        eventBuffer[i].Event.MouseEvent.dwMousePosition.X +
        col*eventBuffer[i].Event.MouseEvent.dwMousePosition.Y;
    if (eventBuffer[i].Event.MouseEvent.dwButtonState &
        FROM_LEFT_1ST_BUTTON_PRESSED) {
        consBuffer[consBuffPos].Char.AsciiChar=(char)0xDB;
        WriteConsoleOutputA(wHnd, consBuffer, bufferSize,
                            charPosition, &writeArea);
    }
    else if (eventBuffer[i].Event.MouseEvent.dwButtonState &
            RIGHTMOST_BUTTON_PRESSED) {
        consBuffer[consBuffPos].Char.AsciiChar=(char)0xB1;
        WriteConsoleOutputA(wHnd, consBuffer, bufferSize,
                            charPosition, &writeArea);
    }
    else if (eventBuffer[i].Event.MouseEvent.dwButtonState &
            FROM_LEFT_2ND_BUTTON_PRESSED) {
        consBuffer[consBuffPos].Char.AsciiChar = ' ';
        WriteConsoleOutputA(wHnd, consBuffer, bufferSize,
                            charPosition, &writeArea);
    }
}
}
}
delete[] eventBuffer; // освобождение памяти
}
}
}
}

```

Возможности программы представлены на следующем рисунке



*Замечание.* Здесь можно использовать способ рисования, который был нами показан в примере с «изображением» буквы А. Весь рисунок печатался пробелами, но для каждого из них устанавливался свой атрибут – белый или черный фон. Есть возможность выбора 16 цветов фона и, следовательно, у нас есть возможность преобразовать программу в «цветной графический редактор».

■

Если вы попытаетесь в поток `cout` вывести текст на национальном языке, то в результате увидите абракадабру. Это связано с тем, что в редакторе Visual Studio для русского языка используется стандартная Windows кодировка `cp1251` и во встроенном редакторе строки представлены символами с кодами из этой кодовой таблицы. Но консоль понимает только кодировку `cp866`. В итоге получается, что программа передаёт коды символов кодовой таблицы `cp1251`, а в консоли в кодовой таблице `cp866` им соответствуют совсем другие символы.

Одно из решений проблемы состоит в том, чтобы перед передачей текста в консоль выполнялась его перекодировка. Для этого можно использовать функцию `setlocale()`, которая выполняет перекодировку символов в соответствии с требуемым языком. Эта функция определена в заголовочном файле `<locale>`. Например,

```
.....
#include <locale>
.....
int main {
    setlocale(LC_TYPE, "rus");// вызов функции локализации
    ....
}
```

Функция `setlocale()` имеет два параметра, первый параметр – тип перекодировки. В нашем случае `LC_TYPE` говорит, что будет перекодироваться набор символов. Второй параметр – строка, указывающая язык. В нашем случае это `"rus"`. Можно также писать `"Russian"`, или оставлять пустые двойные кавычки, тогда набор символов будет такой же как и в ОС. После этого можно использовать кириллицу, для вывода сообщений в консоль.

Заметим, что при вводе кириллицы проблемы остаются, но здесь мы не будем касаться решения этого вопроса.

Ниже мы используем еще одну функцию `SetConsoleCursorInfo`. Она устанавливает размер и «видимость» текстового курсора. Первым аргументом функция принимает дескриптор окна, а вторым – указатель на структуру `CONSOLE_CURSOR_INFO`, которая имеет вид

```
typedef struct _CONSOLE_CURSOR_INFO {
    DWORD dwSize;
    BOOL bVisible;
} CONSOLE_CURSOR_INFO;
```

Поле `dwSize` задает размер курсора (в процентах от размера ячейки одного символа; изменяется от 1 до 100). Поле `bVisible` включает (`true`) или выключает (`false`) видимость текстового курсора.

В следующем примере мы создаем программу с однострочным текстовым меню. Пункты меню расположены в верхней строке консоли. Перемещение по ним выполняется клавишами управления курсора →, ←. Выбор выделенного пункта меню осуществляется клавишей ENTER. При выборе меню происходит вызов соответствующей функции, в которой ввод и вывод данных в рабочую область консоли выполняется обычными «текстовыми» функциями C/C++. Пример окна программы показан на следующем рисунке.



Код программы menudemo и ее описание приведены ниже.

### Пример 9. Работа с текстовым меню.

```

/* ---- файл menudemo.h --- */
// описание функций, которые подключаются к пунктам меню
void File(void);
void Do(void);
void Clear(void);
void Exit(void);

/* ---- файл drawmenu.h --- */
// описание функций, которые управляют работой меню
void DrawMenu(); //создание меню
void gotoxy(int x, int y); // перевод курсора в точку x,y
void itemMenu(int sel, bool activate); // выделить пункт меню
void cls(int it=0); // очистка консоли; при it==0 оставляем
// строку меню иначе очищаем всю консоль
void getCursorPosition(void); // запомнить положение курсора
//в глобальную переменную curspos
void showCursor(bool visible); // скрыть/показать курсор

/* ---- файл menudemo.cpp --- */
#include <windows.h>
#include <conio.h>
#include <iostream>
#include <locale>
#include "drawmenu.h"
#include "menudemo.h"
using namespace std;
extern HANDLE hStdOut; //дескриптор консольного окна
extern CONSOLE_SCREEN_BUFFER_INFO csbInfo; //информация о
// консольном окне в структуре csbInfo
extern SMALL_RECT consolRect; //координаты углов консоли
extern WORD wokrWindowAttributes; //атрибуты рабочей
//области консоли

```

```

void main()
{
    setlocale(LC_STYPE, "rus"); // вызов функции настройки
                               // национальных параметров
    SetConsoleTitle(L"Пример создания строки меню");
    hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
    GetConsoleScreenBufferInfo(hStdOut, &csbInfo);
    consolRect = csbInfo.srWindow; //координаты углов консоли
    SetConsoleTextAttribute(hStdOut, wokrWindowAttributes);
    system("CLS"); // установка атрибутов цвета рабочей области
    DrawMenu();    // рисуем меню в верхней строке консоли
}

// ===== Функции меню =====
//Функция меню <Файл>. Заполняется кодом пользователя
void File() {
    cout << "Вы выбрали меню 'Файл'\n";
}

//Функция меню <Действие>. Заполняется кодом пользователя
void Do() {
    long val = 0;
    cout << "Введите целое число: ";
    cout << "Квадрат " << val << " равен " << val*val << "\n";
}

//Функция меню <Выход> - завершение программы
void Exit() {
    int resp;
    cout << "Вы уверены, что хотите выйти из программы? (y/n)?";
    resp = getchar();
    if (resp == 'y' || resp == 'Y') { cls(1); exit(0); }
}

// Функция меню <Очистить>
void Clear(void)
{
    cls();
}

```

Файл `menudemo.cpp` прост. Функция `main()` вызывает функцию `setlocale`, позволяющую выводить в консоль шрифт национального алфавита. Потом в глобальную переменную запоминается дескриптор консольного окна и определяются координаты углов окна. После этого устанавливается цвет рабочей области консоли и в конце вызывается функция `DrawMenu()`, которая создает меню.

После функции `main` содержатся определения функций, которые будут вызываться при выборе пунктов меню. Пользователь может добавить свои функции, если пунктов будет больше.

Основной код, который создает меню и управляет его работой, находится в следующем модуле.

```

/* ----- модуль drawmenu.cpp -----*/
// Код функций для управлением строчным меню
#include <windows.h>
#include <conio.h>
#include <iostream>
#include "drawmenu.h"
#include "menudemo.h"
using namespace std;

#define KEY_ARROW_RIGHT 77
#define KEY_ARROW_LEFT 75
#define KEY_ENTER 13
typedef void(*FUN) (void); //Указатель на функцию void f(void)
//они будут выполнять пункты меню

typedef struct { //Структура для элемента меню
    int x, y; //Столбец и строка консоли
    char *str; //Наименование пункта меню
    FUN f; //Функция, привязанная к пункту меню
} ITEM;

// Глобальные переменные, используемые в функциях меню
HANDLE hStdOut; //дескриптор консольного окна
CONSOLE_SCREEN_BUFFER_INFO csbInfo; //информация о консольном окне
SMALL_RECT consRect; //координату углов консольного окна
COORD curspos={0,1}; //координаты текстового курсора
WORD wokkWindowAttributes = 158; //атрибуты рабочей области
WORD inactiveItemAttributes = 31; //атрибуты цвета неактивного
// пункта меню
WORD activeItemAttributes = 160; // атрибуты цвета активного
// пункта меню

// Изменяемые элементы меню
enum menuitems { MNUFILE, MNUDO, MNUCLEAR, MNUEXIT };
extern const int numMenu = 4; //количество пунктов меню
ITEM menu[numMenu] = { //положение (x,y), заголовок,
// указатель на функцию
    { 1, 0, " Файл ", File },
    { 11, 0, " Действие ", Do },
    { 21, 0, " Очистить ", Clear },
    { 31, 0, " Выход ", Exit }
};
// Длина строк заголовков " Файл ", " Действие ", " Очистить ",
// " Выход " должна быть подобрана в соответствии с их
// X - координатами в массиве menu[]

void DrawMenu() { //Управление меню
    menuitems sel = MNUFILE; // Номер текущего пункта меню
    SetConsoleTextAttribute(hStdOut, inactiveItemAttributes);
    string s(80, ' '); cout << s.c_str(); //залить фон строки меню
    for (int i = 0; i < numMenu; i++) { //Напечатать заголовки
        //пунктов меню
        gotoxy(menu[i].x, menu[i].y);
        cout << menu[i].str;
    }
}

```

```

itemMenu(sel, true);           // выделить пункт меню
fflush(stdin);                //очистить буфер клавиатуры
int iKey = 67;
while (1)
{
    if (_kbhit())
    {
        iKey = _getch();
        switch (iKey)
        {
            case KEY_ARROW_RIGHT:
                if (sel < numMenu - 1) {
                    itemMenu(sel, false); // сделать неактивным пункт меню
                    sel = (menuitems)(sel + 1);
                    itemMenu(sel, true); // выделить активный пункт меню
                }
                else {
                    itemMenu(sel, false); // сделать неактивным пункт меню
                    sel = MNUFILE; // прокрутка влево
                    itemMenu(sel, true); // выделить активный пункт меню
                }
                showCursor(false);
                break;
            case KEY_ARROW_LEFT:
                if (sel > 0) {
                    itemMenu(sel, false);
                    sel = (menuitems)(sel - 1);
                    itemMenu(sel, true);
                }
                else {
                    itemMenu(sel, false);
                    sel = MNUEXIT; // прокрутка влево
                    itemMenu(sel, true);
                }
                showCursor(false);
                break;
            case KEY_ENTER:
                gotoxy(curspos.X, curspos.Y); //возвращаем курсор из строки
                                                // меню в прежнюю позицию
                SetConsoleTextAttribute(hStdOut,
                    wokrWindowAttributes); // Установить цвет
                                                // рабочих сообщений

                showCursor(true);
                switch (sel)
                {
                    case MNUFILE:
                        File();
                        getCursorPosition(); // запомнить положение курсора
                        break;
                    case MNUDO:
                        Do();
                        getCursorPosition(); // запомнить положение курсора
                        break;
                }
            }
        }
    }
}

```

```

case MNUCLEAR:
    Clear();
    curspos = { 0, 1 }; // после очистки курсор
                        //в левый верхний угол консоли

    break;
case MNUEXIT:
    int resp;
    cout << "Вы уверены, что хотите выйти
            из программы? (y/n)?"

    resp = getchar();
    if (resp == 'y' || resp == 'Y')
        { gotoxy(0, 0); cls(1); exit(0); }
    getCursorPosition(); // запомнить положение курсора,
                        // если отменили выход

    break;
}
fflush(stdin); //очистить буфер клавиатуры
gotoxy(menu[sel].x, menu[sel].y); // курсор в
                                // текущий пункт меню

showCursor(false);
break;
case 120: // выход по клавише x
case 88: // выход по клавише X
case 27: // выход по клавише ESC
    gotoxy(0, 0);
    cls(1);
    exit(0); //завершение программы
}
}
}

// Текстовый курсор в точку x,y
void gotoxy(int x, int y)
{
    COORD cursorPos = { x, y };
    SetConsoleCursorPosition(hStdOut, cursorPos);
    //SetConsoleCursorPosition(hStdOut, {x,y});
}

// запись текущего положения текстового курсора в глобальную
// переменную curspos
void getCursorPosition(void)
{
    GetConsoleScreenBufferInfo(hStdOut, &csbInfo);
    curspos = csbInfo.dwCursorPosition; // положение курсора
}

// очистка тестовой области консоли. Если it==0, то очистка со
// строки следующей за строкой меню, иначе очистка с левого
// верхнего угла консоли

```

```

void cls(int it)
{
    int i;
    string s(80, ' ');
    SetConsoleTextAttribute(hStdOut, workWindowAttributes);
    if (it == 0) gotoxy(0, consolRect.Top + 1);
    else gotoxy(0, consolRect.Top);
    for (i = consolRect.Top; i<curpos.Y+1; i++) // очистка от
                                                // первой строки до строки с курсором
        cout << s.c_str(); // залить фон строки меню
    gotoxy(0, 0);
}

// выделить пункт меню с номером sel
void itemMenu(int sel, bool activate)
{
    WORD itemAttributes;
    if (activate) itemAttributes = activeItemAttributes;
    else itemAttributes = inactiveItemAttributes;
    gotoxy(menu[sel].x, menu[sel].y);
    SetConsoleTextAttribute(hStdOut, itemAttributes);
    cout << menu[sel].str;
}

// скрыть/показать текстовый курсор в консоли
void showCursor(bool visible)
{
    CONSOLE_CURSOR_INFO ccInfo;
    ccInfo.bVisible = visible;
    ccInfo.dwSize = 20;
    SetConsoleCursorInfo(hStdOut, &ccInfo);
}

```

Большую часть кода этого модуля составляет определение функции DrawMenu(). В ней «спрятана» вся функциональность меню. Функция рисует/печатает меню в верхней строке консольного окна, а затем запускает бесконечный цикл обработки событий нажатий клавиш. Комментарии в коде поясняют основные команды. Затем в модуле определяются некоторые вспомогательные функции.

Функция gotoxy(int x, int y) устанавливает курсор в заданную позицию.

Функция getCursorPosition(void) сохраняет текущее положение курсора в глобальную переменную curpos.

Функция cls(int it) выполняют очистку консоли (всей или без строки меню).

Функция itemMenu(int sel, bool activate) выделяет или снимает выделение пункта меню.

Функция showCursor(bool visible) делает невидимым или показывает текстовый курсор.



*Замечание.* Нашему меню далеко до функциональности, которую вы наблюдаете в меню программ Windows. Однако оно дает представление о том, как можно создать меню в текстовом режиме консоли. Вы можете использовать нашу программу как шаблон для ваших консольных программ. Все, что вам надо изменить находится в цикле обработки событий нажатий клавиш функции DrawMenu(), а также небольшом фрагменте кода, который еще раз приведен ниже.

```
// Изменяемые элементы меню
enum menuitems { MNUFILE, MNUDO, MNUCLEAR, MNUEXIT };
extern const int numMenu = 4; //количество пунктов меню
ITEM menu[numMenu] = { //положение (x,y), заголовок,
    // указатель на функцию
    { 1, 0, "  Файл  ", File },
    { 11, 0, " Действие ", Do },
    { 21, 0, " Очистить ", Clear },
    { 31, 0, "  Выход  ", Exit }
};
```

Изменения в программе, которые должен выполнить пользователь состоят в следующем:

- изменить количество пунктов меню, задав значение константы numMenu;
- Определить функции, которые будут вызываться при выборах пунктов меню (у нас функции File, Do, Clear, Exit определены в модуле menudemo.cpp);
- добавить описание функций меню в модуль menudemo.h;
- изменить количество пунктов и их заголовки в массиве menu[]; при этом длины строк заголовков (в нашем примере " Файл ", " Действие ", " Очистить ", " Выход ") должна быть подобрана в соответствии с их первой координатой в этом массиве; у нас длина строк заголовков меню равна 10 символам);
- изменить/добавить имена констант в перечислении menuitems; элементов в перечислении должно быть столько, сколько элементов меню;
- в функции DrawMenu() в цикл обработки нажатий клавиш в раздел case KEY\_ENTER внутри тела оператора switch(sel) нужно добавить код обработки выбора соответствующих пунктов меню. Например, если вы добавите в перечисление menuitems константу MYMENU, а функция, которая будет обрабатывать соответствующий ей пункт меню, будет называться mymenu(), то вы должны будете добавить следующий код:

```
case MYMENU:
    mymenu();
    getCursorPosition();
    break;
```

У нашего строчного меню есть недостаток – оно привязано к первой текстовой строке консоли. При прокрутке текста в консоли оно будет также прокручиваться. Вы можете попробовать самостоятельно доработать программу, чтобы это неудобство устранить.

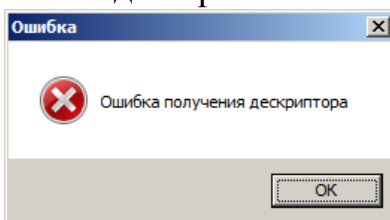



Сделаем еще одно замечание. Большинство функций, которые определяют дескриптор окна, определяют или устанавливают его размер, атрибуты и т.д. могут не справиться со своей задачей. В этом случае они возвращают соответствующие значения, которые нужно (и правильно) проверять. Для простоты контроль ошибок в наших примерах не был реализован. Однако в реально функционирующем приложении такой контроль обязателен.

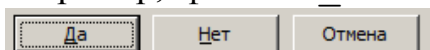
Например, чтение дескриптора консольного окна может выполняться следующим образом

```
hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
if (hStdout == INVALID_HANDLE_VALUE) {
    MessageBox(NULL, L"Ошибка получения дескриптора",
        L"Ошибка", MB_OK | MB_ICONERROR);
    return 1;
}
```

Здесь функция `MessageBox` выводит простое окно сообщений



Первый ее аргумент является идентификатором родительского окна или `NULL`, если такового нет. Второй аргумент является указателем на двухбайтовую строку, которая выводится в теле окна сообщения. Обычная строка превращается в массив символов типа `wchar_t` (каждый символ занимает в памяти два байта), если перед строкой указать префикс `L`. Окно `MessageBox` является диалоговым окном Windows и, поэтому, использует двухбайтовые коды символов в стандарте Unicode, в отличие от однобайтовых строк консоли. Третий аргумент является указателем на двухбайтовую строку, которая выводится в заголовке окна. Последний параметр типа `UINT` определяет какие в окне сообщений отображаются кнопки и иконки. Установка битов этого беззнакового целого числа говорит о том будет ли включен тот или иной элемент интерфейса окна сообщений. Чаще всего его задают комбинацией флагов. В приведенном выше фрагменте кода мы задали этот аргумент в виде комбинации двух флагов `MB_OK | MB_ICONERROR`. Первый из них говорит о необходимости отобразить кнопку `ОК`, второй выводит иконку . А например, флаг `MB_YESNOCANCEL` выводит в окне сообщений три кнопки:



Функция `MessageBox` возвращает целое значение, которое зависит от того, какая кнопка была нажата в окне сообщений.

Существует много других, не затронутых нами функций, которые предназначены для управления консольным окном. Познакомиться с ними вы можете по справочной системе в разделе «Console Functions».

## 1.2 Рисование в консольном окне

В этом параграфе мы покажем, как можно применять графические функции Windows для рисования в консольном окне. Однако помните, что графические возможности этого окна недостаточно функциональны и основным режимом работы консоли является текстовый. Тем не менее, все, что здесь будет изложено, без всяких изменений, но с некоторыми существенными улучшениями функциональности, используется в «оконных» приложениях Windows.

Все графические функции C/C++ в Windows используют специальную структуру, которая называется контекстом окна. Мы здесь не будем описывать ее подробно. Вы должны знать, что она хранит текущие параметры области, в которой мы рисуем: цвет линий, их толщину и стиль (пунктир, сплошной,...), цвет заливки областей, размеры окна и т.д. В программах на C/C++ контекст окна хранится в переменной типа HDC, которая должна быть связана с окном, в котором происходит рисование. Простейшая команда, которая создает такую переменную, имеет вид

```
HDC hdc = GetDC(GetConsoleWindow());
```

Сейчас вы можете не задумываться о ее смысле – просто включайте эту строку в начало функции `main()`, а возвращенный ее дескриптор используйте во всех графических функциях. Любая функция WINDOWS, которая что-то рисует, первым параметром будет принимать переменную `hdc`. Перед завершением программы дескриптор контекста консольного окна рекомендуется освободить командой

```
ReleaseDC(NULL, hdc);
```

Первый параметр представляет идентификатор окна `HWND`, который может использоваться разными функциями программы. Обычно его получают командой

```
HWND hwnd = GetConsoleWindow();
```

Если он использовался в программе, то его идентификатор `hwnd` должен был бы быть передан первым аргументом в функцию `ReleaseDC(hwnd, hdc)`.

В случае ошибки функции `GetConsoleWindow()` и `GetDC()` возвращают специальные значения, которые рекомендуется проверять, например, следующим образом:

```
HWND hwnd = GetConsoleWindow();
HDC hdc;
if (hwnd!=NULL)
{
    hdc=GetWindowDC(hwnd);
    if(hdc==0) cout<<"Error DC Window"<<endl;
}
```

```
else
    cout << "Error Find Window" << endl;
```

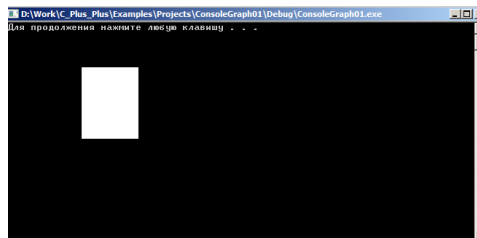
Для упрощения кода в примерах этого параграфа мы не будем выполнять множество проверок значений, возвращаемых функциями, которые получают или создают различные объекты Windows (или их идентификаторы). Однако в реально функционирующих приложениях такой контроль обязателен.

При работе с графикой используется система координат при которой по умолчанию левый верхний угол окна имеет координаты (0,0), оси направлены вправо и вниз, а логическими единицами являются пиксели.

**Пример 1a.** В этом примере мы рисуем прямоугольник в консольном окне.

```
#include <windows.h>

void main()
{
    HDC hdc = GetDC(GetConsoleWindow());
    Rectangle(hdc, 100, 60, 180, 160);
    system("pause");
}
```



Здесь переменная *hdc*, содержащая дескриптор контекста консольного окна, передается первым аргументом функции `Rectangle(hdc, ...)`, которая рисует прямоугольник. Другие аргументы этой функции – координаты левого верхнего и правого нижнего углов  $x_{left}$ ,  $y_{left}$ ,  $x_{right}$ ,  $y_{right}$ .

Начало координат находится в левом верхнем углу рабочей области окна. Горизонтальная ось *X* направлена вправо, вертикальная *Y* – вниз. По умолчанию единицами измерения являются пиксели.

Обратите внимание на поведение рисунка после сворачивания консольного окна, перекрытия его другим окном, изменении размера или прокручивании его внутренней области. Графика исчезает! Этого недостатка обычно нет у стандартных окон Windows потому, что они «умеют» автоматически перерисовывать свою рабочую область после событий сворачивания, перекрытия или изменения размера. В учебных консольных программах перерисовку можно выполнять вручную. Для этого любое рисование следует помещать внутрь создаваемой вами функции `draw(.)`, и вызывать ее повторно в случае, если графика окна исчезла.

**Пример 1b.** Более гибкая версия консольной программы с графикой.

```
#include <windows.h>
#include <iostream>
#include <conio.h>
using namespace std;
void draw(HWND hwnd, HDC hdc);
void setConsoleSize();
```

```

void main()
{
    SetConsoleTitle(L"Simple Rectangle Drawing");
    HWND hwnd = GetConsoleWindow();
    HDC hdc = GetDC(hwnd);
    setConsoleSize(); // задание размеров консоли
    Sleep(100);
    draw(hwnd, hdc); // функция рисования в консоли

    int iKey = 1;
    while (iKey != 27) { // Задержка и выход по клавише ESC
        if (_kbhit()) {
            iKey = _getch();
            switch (iKey)
            {
                case 112: case 80: case 167: case 135:
                    draw(hwnd, hdc); // перерисовка консоли по клавише 'р'
                    break;
            }
        }
    }
    ReleaseDC(hwnd, hdc); //освобождаем дескрипторы консоли
}

// Функция рисования. Помещайте сюда всю графику
void draw(HWND hwnd, HDC hdc)
{
    Rectangle(hdc, 100, 60, 180, 160); // Рисуем прямоугольник
}

void setConsoleSize() // Задание размеров окна консоли
{
    const int colConsole = 80;
    const int rowConsole = 30;
    HANDLE hNdl = GetStdHandle(STD_OUTPUT_HANDLE);
    SMALL_RECT windowSize = {0,0,colConsole-1,rowConsole-1};
    SetConsoleWindowInfo(hNdl, TRUE, &windowSize);
    COORD bufferSize = {colConsole, rowConsole }; // размеры буфера
    SetConsoleScreenBufferSize(hNdl, bufferSize);
}

```

Во второй версии программы мы создали функцию `setConsoleSize(.)`, которая задает размеры консольного окна без полос прокрутки (пока мы не уменьшили размеры окна). Мы также вынесли рисование в отдельную функцию `draw(.)`, которую можем вызывать по нажатию клавиши 'р', например для перерисовки графики, если она пропала после перекрытия консоли другим окном. Перед завершением программы мы также освободили дескрипторы консоли.

Как вы заметили, графика в консоли видна тогда, когда рисование выполняется в видимое окно. После сворачивания или перекрытия окна консоли графика исчезает. На самом деле приложения Windows получают сообщение от ОС о том, что они должны перерисовать свое окно (или его часть) и

программисты помещают вызов графических функций в обработчик этого сообщения (специальную функцию). В программе, написанной для консоли, мы не можем перехватить и обработать такое сообщение. Но перерисовка консоли после восстановления видимости ее окна все равно происходит. Это приводит к рисованию «пустоты», т.е. к очистке окна. Поэтому нам приходится «выкручиваться». После сворачивания или перекрытия окна консоли рисование нужно повторить. Для этого в предыдущей программе мы создали функцию `draw(.)`, которую могли вызвать клавишей 'p'.

Обратите также внимание на функцию `Sleep(.)`, которую мы вызываем перед вызовом функции `draw(.)`. Она приостанавливает выполнение потока инструкций программы на заданное количество миллисекунд. На современных компьютерах, если в коде предыдущей программы не использовать функцию `Sleep(.)`, функция `draw(.)` успевает выполниться еще до того, как консоль станет видимой на экране. После появления консоли на экране, окно получает сообщение о необходимости перерисовки, и в результате происходит очистка окна. Вставка функции `Sleep()` перед вызовом функции `draw` приводит к тому, что окно консоли станет видимым раньше того, как мы что – то нарисуем. Время задержки будет зависеть от быстродействия компьютера. Возможно, вставка функции `Sleep()` вам не понадобится.

В консольном приложении у нас есть возможность обрабатывать сообщение получения окном фокуса. Это сообщение помещается в буфер событий, который заполняется функцией `ReadConsoleInput`. Мы использовали эту функцию в примере 8 предыдущего параграфа. Она помещает в буфер событий информацию о мыши, клавиатуре, а также о получении окном фокуса. Если, например, буфер/массив событий называется `eventBuffer`, то проверку того, что консоль получила фокус, можно выполнить следующим образом

```
if (eventBuffer[i].EventType == FOCUS_EVENT) {
    Sleep(300);
    draw(hwnd, hdc);
}
```

Здесь опять требуется небольшая задержка, поскольку рисование может завершиться до того, как окно консоли появится поверх остальных окон. Еще одна версия консольной программы с графикой может выглядеть следующим образом.

**Пример 1с.** Пример консольной программы с перерисовкой графики при получении окном фокуса.

```
#include <windows.h>
void draw(HWND hwnd, HDC hdc);
void setConsoleSize();
```

```

void main()
{
    HWND hwnd = GetConsoleWindow();
    HDC hdc = GetDC(hwnd);
    HANDLE rHnd = GetStdHandle(STD_INPUT_HANDLE);
    SetConsoleTitle(L"Simple Rectangle Drawing");
    setConsoleSize();
    Sleep(100); // задержка
    draw(hwnd, hdc); // рисование любого множества фигур

    DWORD numEvents = 0; // Количество непрочитанных сообщений
    DWORD numEventsRead = 0; // Количество прочитанных сообщений
    bool isRunning = true; // флаг продолжения работы
    //Если isRunning=false, то программа завершается
    while (isRunning) {
        // Определить количество событий консоли
        GetNumberOfConsoleInputEvents(rHnd, &numEvents);
        if (numEvents != 0) {
            // выделение памяти для хранения данных о событиях
            INPUT_RECORD *eventBuffer = new INPUT_RECORD[numEvents];
            // Извлечение данных во временный буфер событий eventBuffer[]
            ReadConsoleInput(rHnd, eventBuffer, numEvents,
                            &numEventsRead);

            // Цикл по всем извлеченным событиям
            for (DWORD i = 0; i < numEventsRead; ++i) {
                if (eventBuffer[i].EventType == KEY_EVENT) {
                    if(eventBuffer[i].Event.KeyEvent.wVirtualKeyCode ==
                        VK_ESCAPE)
                        isRunning=false; // выход, если нажата клавиша ESC
                    else if(eventBuffer[i].Event.KeyEvent.uChar.AsciiChar=='d')
                        draw(hwnd, hdc); // перерисовка по клавише 'd'
                }
                if (eventBuffer[i].EventType == FOCUS_EVENT) {
                    Sleep(300);
                    draw(hwnd, hdc); // перерисовка при получении фокуса
                }
                else if (eventBuffer[i].EventType == MOUSE_EVENT) {
                    // обработка событий мыши
                }
            }
            delete[] eventBuffer;
        }
    }
    ReleaseDC(hwnd, hdc); //освобождаем дескрипторы
}

```

Здесь мы используем функцию `setConsoleSize()`, которую создали в примере 1b, и код которой остался без изменений, а также функцию `draw(.)`, в которую вы можете помещать любые функции рисования.

Изменение размеров окна консоли не приводит к событию `FOCUS_EVENT`, но вызывает очистку окна. Поэтому мы оставили команду принудительной перерисовки – вызов функции `draw` при нажатии клавиши `'d'`.

В программах, которые приведены далее в этом параграфе, для простоты не будет реализован контроль ошибок, не будут освобождаться ресурсы, и графические функции не будут собираться в отдельную функцию (например, draw). Однако в реально функционирующих приложениях соответствующие фрагменты кода обязательны.

■

Линия может быть жирной и тонкой, прерывистой и штрих – пунктирной, иметь определенный цвет. Это все вместе называется стилем линии (или просто пером). Текущее перо является частью структуры контекста окна. Чтобы линию нарисовать требуемым пером, вначале вы должны создать объект типа HPEN

```
HPEN Pen = CreatePen(PS_SOLID, 3, RGB(255,0,0));
```

Первый аргумент функции CreatePen определяет будет ли линия сплошной (PS\_SOLID), пунктирной, штрих – пунктирной и т.д. Можно использовать значения PS\_DOT, PS\_DASH, PS\_DASHDOT и некоторые другие. Вторым аргументом определяется толщина линии, третий – задает цвет. Аргументы макроса RGB являются целыми числами со значениями от 0 до 255 и задают цвет, составленный из оттенков/яркостей трех базовых цветов: красного (Red), зеленого (Green) и синего (Blue). Нулевое значение исключает соответствующий базовый цвет, максимальное допустимое значение яркости базовых цветов 255.

По умолчанию установлено черное сплошное перо толщиной в 1 пиксель.

После создания пера (HPEN) его нужно загрузить в контекст (HDC) функцией SelectObject(HDC,HPEN). После этого все линии будут рисоваться созданным пером до тех пор, пока в контекст не будет загружено новое перо.

Есть еще один момент. В контексте хранится положение так называемого текущего указателя, который содержит координаты точки от которой следующая графическая функция начинает рисовать свою фигуру. При запуске программы этот указатель устанавливается в точку (0,0). Его местоположение на экране невидимо, он означает лишь позицию в окне для некоторых функций. Не все графические функции используют этот указатель. Функция

```
MoveToEx (HDC, X, Y, NULL);
```

переносит текущую позицию указателя в точку (X,Y) и не запоминает старую позицию (если последний аргумент равен NULL).

В следующем примере мы используем функцию

```
LineTo (HDC, X, Y)
```

которая рисует отрезок прямой от текущего положения указателя до точки (X,Y) и перемещает указатель в эту точку.

## Пример 2.

```
#include <windows.h>
void main()
{
    // получаем дескриптор контекста консольного окна
    HDC hdc = GetDC(GetConsoleWindow());
```

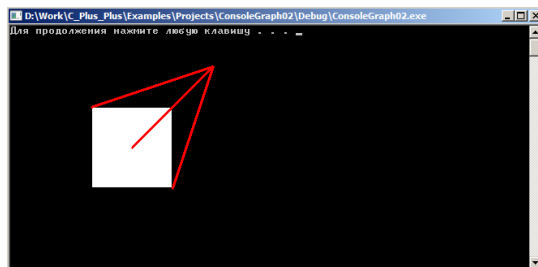


```

Rectangle(hdc,100,100,200,200); // Рисуем квадрат
HPEN Pen = CreatePen(PS_SOLID,3,RGB(255,0,0)); // создаем перо
SelectObject(hdc, Pen); // загружаем созданное перо в контекст
// перемещаем указатель в центр квадрата
MoveToEx(hdc,150,150,NULL);
//рисуем линию от текущего положения указателя
LineTo(hdc, 250, 50);
//рисуем линию из конца предыдущего отрезка
LineTo(hdc, 200, 200);
//перемещаем указатель в левую верхнюю вершину квадрата
MoveToEx(hdc, 100, 100, NULL);
//рисуем линию от текущего положения курсора до точки (250,50)
LineTo(hdc, 250, 50);

system("pause");
DeleteObject(Pen);
ReleaseDC(NULL, hdc);
}

```



■

Графическая область окна представляет собой массив пикселей. Каждый пиксель соответствует одной точке на экране и может иметь свой цвет. Установить цвет пикселя в точке экрана с координатами (X,Y) можно с помощью функции

```
SetPixel(HDC dc, int X, int Y, COLORREF Color);
```

Здесь COLORREF представляет специальный тип (структуру), которую создает макрос RGB. Функция SetPixel отображает точку заданного цвета по указанным координатам и возвращает цвет, который был установлен в действительности (иногда дисплей не в состоянии точно воспроизвести заданный цвет).

### Пример 3. Рисование синусоиды.

В этом примере мы демонстрируем возможность рисования графика функции в консольном окне двумя способами – поточечно и сплошной линией.

```

#include <windows.h>
#include <math.h>
const double Pi = 3.141592;
void main()
{
    HDC hdc = GetDC(GetConsoleWindow());
    double x, y;
    COLORREF penColor = RGB(255, 255, 0);

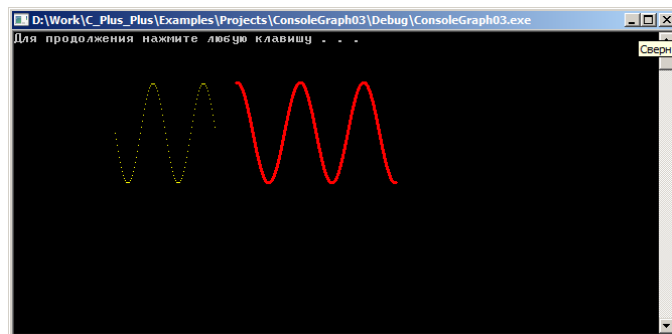
```

```

// Рисуем график синусоиды по точкам
for(x =0;x<100;x++){
    y = sin(Pi*x/25);
    SetPixel(hdc,100+int(x),100+int(50*y),penColor);
}
// создаем перо для рисования
HPEN Pen = CreatePen(PS_SOLID,3,RGB(255,0,0));
SelectObject(hdc, Pen);
// рисуем синусоиду по отрезкам
int xScale=10,yScale=50; // масштаб по осям X и Y
int x0=300,y0=100; // положение начала координат
// диапазон изменения абсциссы
float xBeg = -8.0f, xEnd = 8.0f;
// указатель в начальную точку кривой
MoveToEx(hdc,x0+xScale*xBeg, y0+yScale*sin(xBeg), NULL);
//рисуем отрезки от точки к точке
for (x=xBeg;x<=xEnd;x+=0.01f)
    LineTo(hdc,x0+xScale*x,y0+yScale*sin(x));

system("pause");
DeleteObject(Pen);
ReleaseDC(NULL, hdc);
}

```



Обратите внимание, как используется указатель текущей позиции при рисовании сплошной синусоиды. Вначале он устанавливается в начальную точку функцией `MoveToEx`. Затем в теле цикла функция `LineTo(hdc, Xx, Yx)` рисует отрезок от положения указателя до точки  $X_x, Y_x$  и устанавливает указатель в эту точку. На следующем шаге цикла функция `LineTo` соединяет отрезком текущее положение указателя (т.е. старую точку  $X_x, Y_x$ ) с новой точкой кривой  $X_x, Y_x$  и т.д.

Обратите также внимание на то, что для отображения точки функцией `SetPixel` нужно задавать только цвет пикселя, в то время как для рисования отрезка функцией `LineTo` нужно создавать перо и загружать его в контекст окна. Если в этом примере вы не создадите перо, то второй график вы не увидите, поскольку перо по умолчанию является черным (как цвет фона).

Константу  $\pi$  мы создали самостоятельно

```
const double Pi = 3.141592;
```

При компиляции вы должны получить два предупреждения (warning) «conversion from 'float' to 'int', possible loss of data». Они означают, что при построении второго графика выполнялось

автоматическое приведение типа из `float` в `int` в аргументах функции `LineTo`. Чтобы такого сообщения не было вы должны самостоятельно выполнить преобразование типа так, как мы это сделали в функции `SetPixel` при построении графика по точкам.

■

Существует много функций, которые рисуют разные графические объекты. Сейчас рассмотрим те, которые рисуют кривые. Функция

```
BOOL LineTo(hDC, int x, int y);
```

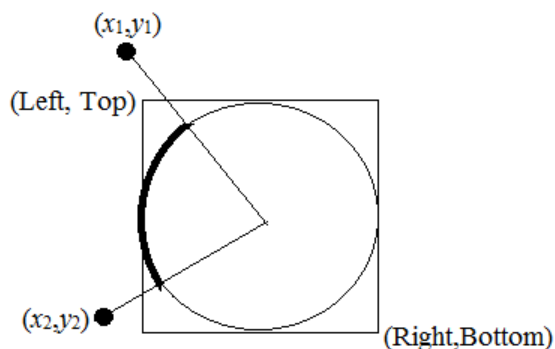
рисует линию от текущей позиции до точки, указанной в аргументах, и переводит указатель позиции в эту точку. Поскольку в GDI нет функции рисования линии от одной указанной точки до другой, то её можно создать самому. Она будет рисовать отрезок из точки  $x_1, y_1$  до точки  $x_2, y_2$ .

```
BOOL Line(HDC hdc, int x1, int y1, int x2, int y2)
{
    MoveToEx(hdc, x1, y1, NULL);
    return LineTo(hdc, x2, y2);
}
```

**Функция**

```
BOOL Arc(hDC, int left, int top, int right, int bottom,
         int x1, int y1, int x2, int y2);
```

рисует дугу эллипса/окружности. Аргументы `left, top` и `right, bottom` определяют левый верхний и правый нижний углы прямоугольника, в который вписан эллипс. Последующие значения – координаты точек, от которых будут проведены прямые к центру эллипса. В местах их пересечения с эллипсом, начинается и кончается дуга (сами прямые не рисуются). Дуга рисуется в направлении против часовой стрелки. Используемые обозначения пояснены на следующем рисунке.



Чтобы нарисовать контур всего эллипса/окружности достаточно в качестве последних 4 – х параметров передать нули. Например,

```
Arc(hdc, 220, 100, 280, 160, 0, 0, 0, 0);
```

Функция `Arc` указатель текущей позиции не использует. Ее аналог функция

```
BOOL ArcTo(hDC, int left, int top, int right, int bottom,
           int x1, int y1, int x2, int y2);
```

использует и обновляет положение текущего указателя. Вначале от текущего указателя до начала дуги функция рисует отрезок, затем рисует дугу и устанавливает текущий указатель в ее конечную точку.

## Функция

```
BOOL AngleArc(HDC hdc, int X, int Y, DWORD R,  
              FLOAT startAngle, FLOAT sweepAngle);
```

рисует дугу окружности и отрезок от текущей позиции до начала этой дуги. Дуга является частью окружности радиуса R с центром в точке X,Y. Дуга задается двумя углами, измеряемыми в градусах и отсчитываемыми против часовой стрелки от положительного направления оси X. Параметр startAngle задает начало дуги, а sweepAngle ее угловую меру. Функция AngleArc начинает рисовать от положения текущего указателя и устанавливает его в конечную точку своей дуги. Дуга рисуется текущим пером.

## Функция

```
BOOL Polyline(hdc, POINT *pt, int cPoints);
```

рисует ломаную, проходящую через точки, координаты которых находятся в массиве pt. Структура POINT определена в библиотеке следующим образом.

```
typedef struct tagPOINT {  
    LONG x;  
    LONG y;  
} POINT;
```

В аргументе cPoints функции Polyline передается количество точек массива (>=2). Например, следующие команды рисуют треугольник

```
POINT ptr[4]={{350,90},{400,140},{300,140},{350,90}};  
Polyline(hdc,ptr,4);
```

Функция Polyline не использует и не обновляет указатель текущей позиции. Ее аналог функция

```
BOOL PolylineTo(hdc, POINT *pt, int cPoints);
```

использует и обновляет указатель текущей позиции. Ломаная начинает рисоваться от текущего положения указателя до первой точки массива pt, и далее проходит через другие точки этого массива. Функция переводит указатель текущей позиции в конечную точку ломаной.

Функция PolyPolyline рисует сразу несколько ломаных

```
BOOL PolyPolyline(HDC hdc, constPOINT *lppt,  
                  constDWORD *lpdwPolyPoints, DWORD cCount);
```

Второй аргумент принимает массив координат вершин всех кусков ломаной, третий аргумент lpdwPolyPoints – это массив количеств вершин в каждом куске, четвертый передает количество кусков.

Функция TextOut отображает текстовую строку в заданном месте окна, используя текущие шрифт, цвет фона и символов.

```
BOOL TextOut(HDC hdc, int xStart, int yStart,  
             LPCTSTR lpString, int countChars);
```

Аргументы xStart и yStart определяют точку привязки текстовой строки lpString. Строка не обязана заканчиваться нулевым символом, поскольку аргумент countChars содержит количество символов, которое следует

напечатать. Учитывая, что мы говорили о завершающем символе `\A` функций Windows, фрагмент вызова этой функции может иметь следующий вид

```
char str[] = "Три ломаных";
TextOutA(hdc, 360, 40, str, strlen(str));
```

Функция `TextOutA` не должна быть последней в цепочке графических функций (иначе текст не отображается).

Чтобы изменить цвет текста и цвет фона, на котором он отображен, используют функции `SetBkColor` и `SetTextColor`. Функция `SetTextColor` «рисует» текст заданным цветом. Функция `SetBkColor` закрашивает фон между буквами заданным цветом, а также промежутки пунктирных линий пера, создаваемого функцией `CreatePen`.

В следующем фрагменте кода показано как можно их использовать

```
COLORREF bk = SetBkColor(hdc, RGB(255, 255, 0));
if (bk == CLR_INVALID) cout << "Color error\n";
SetTextColor(hdc, RGB(255, 0, 0));
TextOutA(hdc, 260, 20, "Красный текст на желтом фоне", 28);
```

#### Пример 4. Демонстрация функций рисования кривых.

В следующем примере мы используем описанные выше функции для рисования нескольких простых контуров.

```
#include <windows.h>
#include <iostream>
using namespace std;
BOOL Line(HDC hdc,int x1,int y1,int x2,int y2);

void main()
{
    SetConsoleTitle(L"Example of drawing");
    HDC hdc=GetDC(GetConsoleWindow());
    HPEN Pen = CreatePen(PS_SOLID, 3, RGB(255, 0, 0)); // перо 1
    SelectObject(hdc, Pen); // помещаем перо 1 в контекст
    Arc(hdc, 40,40,140,140,200,90,0,90); //Дуга полуокружности
    Line(hdc,40,90,140,90); // диаметр полуокружности

    HPEN Pen2 = CreatePen(PS_SOLID,3,RGB(0,255,255)); // перо 2
    SelectObject(hdc, Pen2); // помещаем перо 2 в контекст
    LineTo(hdc,190,90); // горизонтальный отрезок
    // вертикальный радиус и дуга четверти окружности
    ArcTo(hdc,140,40,240,140,190,0,0,90);

    HPEN Pen3=CreatePen(PS_SOLID,3,RGB(255,255,0)); // перо 3
    SelectObject(hdc, Pen3); // помещаем перо 3 в контекст
    AngleArc(hdc, 190,90,50,180,180); // желтая полуокружность

    HPEN Pen4=CreatePen(PS_SOLID,3,RGB(0,0,200)); // перо 4
    SelectObject(hdc, Pen4); // помещаем перо 4 в контекст
    POINT ptr[3]={{340,90},{290,40},{240,90}};
    PolylineTo(hdc, ptr, 3); // рисуем треугольник
```

```

POINT tArrow[7]={ {20,150}, {180,150}, {180,120}, {230,170},
                  {180,220}, {180,190}, {20,190} };
Polyline(hdc,ptArrow,7);      // рисуем стрелку

char str[] = "Три ломаных";
TextOutA(hdc,360,40,str,strlen(str)); // выводим текст

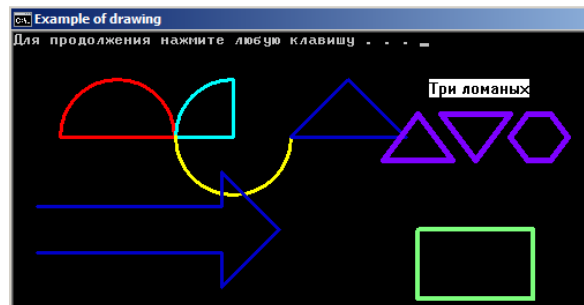
SelectObject(hdc,CreatePen(PS_SOLID,5,RGB(125,0,255)));
POINT ptFigs[15]={ {350,70}, {320,110}, {380,110}, {350,70},
                  {370,70}, {400,110}, {430,70}, {370,70}, {445,70}, {430,90},
                  {445,110}, {465,110}, {480,90}, {465,70}, {445,70} };
DWORD PolyPoints[3]={4,4,7};      // рисуем сразу
PolyPolyline(hdc,ptFigs,PolyPoints,3); // три ломаных

SelectObject(hdc,CreatePen(PS_SOLID,4,RGB(125,255,125)));
POINT ptRect[5]=
    { {350,170}, {450,170}, {450,230}, {350,230}, {350,170} };
Polyline(hdc, ptRect, 5);      // рисуем прямоугольник

system("pause");
}

BOOL Line(HDC hdc, int x1, int y1, int x2, int y2)
{
    MoveToEx(hdc, x1, y1, NULL);
    return LineTo(hdc, x2, y2);
}

```



Есть много функций, которые рисуют геометрические фигуры (закрашенные плоские области). Но также, как для линий нужно сперва назначить перо, для фигур надо установить кисть, которая будет закрашивать их внутренность. Есть два способа объявить кисть. Первый – задать кисть со сплошной заливкой, второй – указать кисть с заданным стилем заливки. Для этого существуют две функции: `CreateSolidBrush()` и `CreateHatchBrush()`. После создания кисть должна быть загружена в контекст устройства функцией `SelectObject`. Например, в следующем фрагменте мы создаем сплошную красную кисть и делаем ее активной.

```

HBRUSH hBrush;      //создаём объект-кисть
hBrush =CreateSolidBrush( RGB(255,0,0) ); //создаем сплошную кисть
SelectObject(hdc, hBrush);      //делаем кисть активной

```

После этого графические функции, которые рисуют фигуры, будут закрашивать их сплошной красной кистью. Эта кисть активна до тех пор, пока в контекст не будет загружена другая кисть.

Функция объявления несплошной кисти имеет вид

```
HBRUSH CreateHatchBrush(int fnStyle, RGB(r,g,b));
```

Аргумент `fnStyle` может принимать следующие значения:

```
HS_DIAGONAL - штрихует по диагонали
HS_CROSS - клеточка
HS_DIAGCROSS - диагональная сетка
HS_FDIAGONAL - по диагонали в другую сторону
HS_HORIZONTAL - горизонтальная "тельняшка"
HS_VERTICAL - вертикальный "забор"
```

Вот пример установки такой кисти

```
HBRUSH hBrush;
hBrush = CreateHatchBrush(HS_CROSS, RGB(255,0,120));
SelectObject(hdc, hBrush); //делаем кисть активной
```

При завершении работы программы желательно в контексте консоли восстановить исходную кисть. Следующий фрагмент кода показывает правильный способ такого восстановления

```
.....
HBRUSH hBrush = CreateSolidBrush(RGB(0, 160, 160));
HBRUSH hBrush2=(HBRUSH)SelectObject(hdc, hBrush);
.....
// Вызов графических функций
.....
DeleteObject(SelectObject(hdc, hBrush2));
```

Функция `SelectObject` обычно возвращает объект, который был ею заменен. В нашем фрагменте кода кисть `hBrush2` – это кисть, которая находилась в контексте до замены. Последняя строка фрагмента возвращает кисть `hBrush2` в контекст, функция `SelectObject` возвращает прежнюю кисть `hBrush`, и функция `DeleteObject` освобождает ресурсы от этой кисти.

При старте программы кистью по умолчанию является `WHITE_BRUSH`.

Функция `Rectangle` рисует закрашенный прямоугольник, тип заливки которого определяется текущей кистью.

```
BOOL Rectangle(hdc, left, top, right, bottom);
```

Аргументами этой функции являются контекст консоли и координаты левого верхнего и правого нижнего углов прямоугольника. Контур этого прямоугольника и других фигур рисуется текущим пером.

Функция `RoundRect` рисует скругленный прямоугольник.

```
BOOL RoundRect(hdc, left, top, right, bottom, width, height);
```

Первые пять параметров совпадают с параметрами предыдущей функции. Для скругления углов используются дуги эллипса, ширина и высота которого задаются в аргументах `width` и `height`.

Функция `FillRect` закрашивает прямоугольную область окна заданной кистью (без контура). Прямоугольник определяется вторым аргументом через

указатель на структуру RECT. Кисть, передаваемая третьим аргументом, может задаваться дескриптором кисти или константой, идентифицирующей системный цвет.

```
int FillRect(HDC hdc, const RECT *lprc, HBRUSH hbr);
```

Эту функцию можно использовать для задания фона консольного окна, если передать ей во втором аргументе размеры консоли.

В нашей следующей программе мы будем закрашивать все окно. Для получения размеров его прямоугольника можно использовать функцию GetClientRect. Она имеет следующий синтаксис

```
BOOL GetClientRect(HWND hWnd, LPRECT lpRect);
```

Здесь первым аргументом является идентификатор окна HWND, а вторым – указатель на переменную типа RECT, которая получит координаты левого верхнего и правого нижнего углов клиентской области окна. Например,

```
RECT rectConsole;  
GetClientRect(hWnd, &rectConsole);  
cout<<rectConsole.left<<" "<<rectConsole.top<<" "<<  
    rectConsole.right<<" "<<rectConsole.bottom<<"\n";
```

Функция FrameRect(HDC, RECT \*, HBRUSH) рисует контур вокруг прямоугольной области, заданной вторым аргументом, используя кисть, передаваемую в третьем аргументе. Толщина контура всегда равна одной логической единице (обычно один пиксель). Например,

```
RECT rct={300,40,400,180};  
FrameRect(hdc, &rct, CreateSolidBrush(RGB(255,0,0)));
```

Функция InvertRect(HDC, RECT \*) инвертирует цвета пикселей внутри прямоугольной области, заданной вторым аргументом. Обращение состоит в применении операции NOT к битам значения цвета каждого пикселя. Например,

```
RECT rc3={40,40,120,140};  
InvertRect(hdc, &rc3); // инвертирование цветов
```

Функция Ellipse рисует эллипс или окружность.

```
BOOL Ellipse(HDC hdc, int left, int top, int right, int bottom);
```

Ее аргументы задают координаты левого верхнего и правого нижнего углов прямоугольника, в который вписан эллипс.

Функция Chord рисует сегмент (область между дугой эллипса и ее хордой). Контур сегмента рисуется текущим пером, а его внутренность закрашивается текущей кистью.

```
BOOL Chord(HDC hdc, int left, int top, int right, int bottom,  
           int x1, int y1, int x2, int y2);
```

Аргументы left, top и right, bottom определяют левый верхний и правый нижний углы прямоугольника, в который вписан эллипс. Последующие значения – координаты точек, от которых будут проведены прямые к центру эллипса. В местах их пересечения с эллипсом, начинается и заканчивается дуга.



Дуга рисуется в направлении против часовой стрелки, а ее концы соединяются хордой. Смысл аргументов  $x_1, y_1, x_2, y_2$  такой же, как у функции `Arc`.

Функция `Pie` рисует сектор (область между дугой эллипса и двумя ее радиальными отрезками). Контур сектора рисуется текущим пером, а его внутренность закрашивается текущей кистью.

```
BOOL Pie(hdc, int left, int top, int right, int bottom,  
         int x1, int y1, int x2, int y2);
```

Аргументы функции `Pie` имеют такой же смысл, как у функции `Chord`.

**Функция**

```
BOOL Polygon(hdc, POINT *pt, int cPoints);
```

рисует многоугольник, координаты вершин которого расположены в массиве, имя которого передается во втором аргументе. Элементы массива имеют тип `POINT`. Контур многоугольника рисуется текущим пером, а его внутренность закрашивается текущей кистью. Количество вершин многоугольника передается в третьем аргументе. Функция `Polygon` не использует и не обновляет указатель текущей позиции. Многоугольник замыкается автоматически. Вот фрагмент кода построения треугольника.

```
POINT ptTriangle[3]={{310,200},{280,240},{340,240}};  
Polygon(hdc, ptTriangle, 3);
```

Обратите внимание, что аргументы функции `Polygon` такие же, как и у функции `Polyline`.

Функция `PolyPolygon` рисует сразу несколько многоугольников, аналогично тому, что делает функция `PolyPolyline`.

```
BOOL PolyPolygon(HDC hdc, const POINT *lppt,  
                const DWORD *lpdwPolyPoints, DWORD cCount)
```

Второй аргумент принимает массив координат вершин всех многоугольников, третий аргумент `lpdwPolyPoints` – это массив количеств вершин в каждом многоугольнике, четвертый передает количество многоугольников. Как видите, аргументы функции `PolyPolygon` такие же, как и у функции `PolyPolyline`.

В следующем примере мы используем описанные выше функции для рисования нескольких областей.

**Пример 5.** Демонстрация графических функций рисования областей.

```
#include <windows.h>  
#include <iostream>  
#include <conio.h>  
using namespace std;  
  
void main()  
{  
    HWND hwnd = GetConsoleWindow(); // идентификатор окна консоли  
    HDC hdc = GetDC(hwnd); // дескриптор контекста консольного окна  
  
    RECT clientRect; // координаты прямоугольника  
    GetClientRect(hwnd, &clientRect); // получаем размеры окна
```

```

// заполняем фон окна
FillRect(hdc, &clientRect, CreateSolidBrush(RGB(0,160,160)));
Sleep(100);

COLORREF bk=SetBkColor(hdc,RGB(255,255,0)); //цвет фона текста
if (bk == CLR_INVALID) cout << "color error\n";
SetTextColor(hdc,RGB(255,0,0)); // Установка цвета текста
TextOutA(hdc, 260, 20, "Красный текст на желтом фоне", 28);

COLORREF penColor=RGB(0,0,255); //синее перо
SelectObject(hdc,CreatePen(PS_SOLID,3,penColor));
Rectangle(hdc,100,160,180,260); //кисть по умолчанию WHITE_BRUSH

POINT ptArrow[7]={{20,50},{180,50},{180,20},{230,70},
                  {180,120},{180,90},{20,90}};
Polygon(hdc, ptArrow, 7); //использование цвет фона по умолчанию

// Коричневое перо
SelectObject(hdc, CreatePen(PS_SOLID,3,RGB(120,0,120)));
// Желтая кисть
SelectObject(hdc, CreateSolidBrush(RGB(255,255,0)));

Ellipse(hdc,250,150,350,250);
Chord(hdc,250,20,350,120,150,20,350,120); // сегмент

SetBkColor(hdc,RGB(160,160,160)); // меняем задний фон кисти
SelectObject(hdc, CreateHatchBrush(HS_CROSS,RGB(0,0,255)));
POINT romb[4]={{450,70},{550,140},{450,210},{350,140}};
Polygon(hdc, romb, 4); // рисуем ромб

_getch(); // ----- Первый экран программы -----

//Меняем фон рабочего окна. Стираем предыдущие фигуры
FillRect(hdc, &clientRect, CreateSolidBrush(RGB(200, 200,200)));

Polygon(hdc, ptArrow, 7); // Полигональная область стрелки

RECT rct={300,40,400,180};
// заполненный прямоугольник без контура границы
FillRect(hdc, &rct, CreateSolidBrush(RGB(255,255,0)));
// контур прямоугольника рисуем кистью толщиной 1 пиксель
FrameRect(hdc, &rct, CreateSolidBrush(RGB(255, 0, 0)));

RECT rc3={40,40,120,140};
InvertRect(hdc, &rc3); // обращение цветов внутри прямоугольника

HBRUSH brsh = CreateSolidBrush(RGB(0,255,140));
SelectObject(hdc, brsh);
RoundRect(hdc,40,160,240,280,40,40); //скругленный прямоугольник

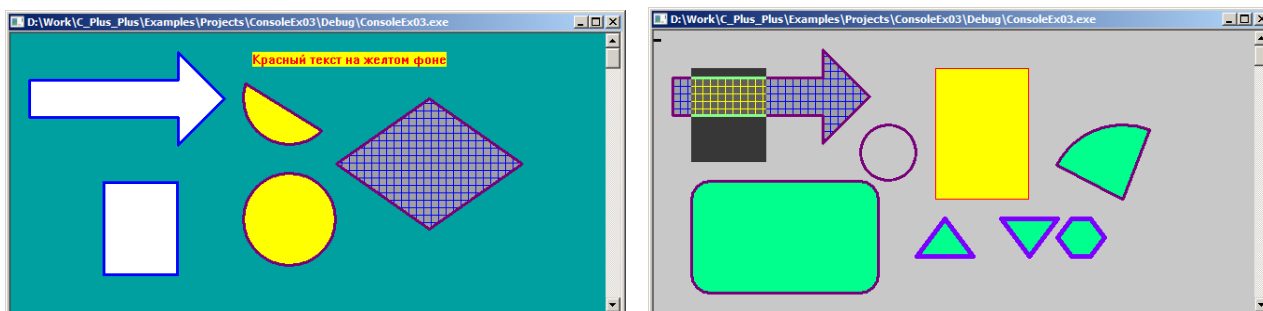
Arc(hdc,220,100,280,160,0,0,0,0); // коричневая окружность
Pie(hdc, 420, 100, 580, 260, 550, 50, 250, 50); // сектор

```

```
SelectObject(hdc, CreatePen(PS_SOLID, 5, RGB(125, 0, 255)));
POINT ptFigs[15]={{310, 200}, {280, 240}, {340, 240}, {310, 200},
{370, 200}, {400, 240}, {430, 200}, {370, 200}, {445, 200}, {430, 220},
{445, 240}, {465, 240}, {480, 220}, {465, 200}, {445, 200}};
INT PolyPoints[3]={4, 4, 7}; // рисуем сразу 3 многоугольника
PolyPolygon(hdc, ptFigs, PolyPoints, 3);
```

```
_getch(); // ----- Второй экран программы -----
}
```

На рисунке слева показан первый кадр программы, справа – второй.



В Windows API имеется много функций работы с графическими областями. Область – это прямоугольная, многоугольная, эллиптическая фигура или комбинация нескольких таких фигур. Над ее пикселями можно выполнять разные операции. Она может быть закрашена, обрaмлена контуром, ее цвета могут быть инвертированы, ее можно использовать для проверки выполнения щелчка мыши внутри нее.

Чтобы работать с областью, вначале она должна быть создана. Для этого имеется целый ряд функций, например, `CreateRectRgn`, `CreateRoundRectRgn`, `CreateEllipticRgn`, `CreatePolygonRgn` и еще несколько других. Они используют свои параметры для описания геометрии области и возвращают дескриптор области – объект типа `Hrgn`. После создания области с ней можно выполнять различные операции.

Для создания прямоугольной области используются функции `CreateRectRgn` и `CreateRectRgnIndirect`.

Функция `CreateRectRgn` создает прямоугольную область, используя координаты левого верхнего и правого нижнего углов прямоугольника.

```
Hrgn CreateRectRgn( int left, int top, int right, int bottom);
```

Функция `CreateRectRgnIndirect` создает прямоугольную область, используя один аргумент – указатель на переменную типа `RECT`, которая содержит координаты тех же вершин.

```
Hrgn CreateRectRgnIndirect( const RECT *lprc);
```

Функция `CreateRoundRectRgn` создает область «скругленного» прямоугольника.

```
Hrgn CreateRoundRectRgn( int left, int top, int right,
int bottom, int widthEllipse, int heightEllipse);
```

Первые четыре параметра совпадают с параметрами функции `CreateRectRgn`. Для скругления углов используются дуги эллипса, ширина и высота которого задаются в аргументах `widthEllipse` и `heightEllipse`.

Функция `CreateEllipticRgn` создает эллиптическую область.

```
HRGN CreateEllipticRgn(int left,int top,int right,int bottom);
```

Ее аргументы задают координаты левого верхнего и правого нижнего углов прямоугольника, в который вписан эллипс.

Функция `CreateEllipticRgnIndirect` также создает эллиптическую область, но использует один аргумент – указатель на переменную типа `RECT`, которая содержит координаты тех же вершин.

```
HRGN CreateEllipticRgnIndirect( const RECT *lprc);
```

Функция `CreatePolygonRgn` создает многоугольную область

```
HRGN CreatePolygonRgn(const POINT *lppt, int cPoints,  
int fnPolyFillMode);
```

Координаты вершин многоугольника расположены в массиве, имя которого передается в первом аргументе. Элементы массива имеют тип `POINT`. Количество вершин многоугольника передается во втором аргументе. Многоугольник замыкается автоматически.

Третий аргумент задает способ заполнения фона многоугольника. Он может принимать два значения `ALTERNATE` и `WINDING`. В режиме `ALTERNATE` включаются внутренние области, находящиеся между первой и второй, третьей и четвертой т.д. сторонами контура. В режиме `WINDING` включаются все внутренние области. Следующий рисунок поясняет различие между этими способами.



Функция `CombineRgn` позволяет выполнять операции теории множеств над двумя областями, как множествами точек на плоскости.

```
int CombineRgn(HRGN hrgnDest,HRGN hrgnSrc1,  
HRGN hrgnSrc2,int fnCombineMode);
```

Области `hrgnSrc1` и `hrgnSrc2` комбинируются, а результат помещается в область `hrgnDest`. Операция комбинирования задается 4 – м аргументом `fnCombineMode`. Он может принимать следующие значения:

- `RGN_AND` – пересечение областей;
- `RGN_DIFF` – разность областей;
- `RGN_OR` – объединение областей;
- `RGN_XOR` – симметрическая разность областей;
- `RGN_COPY` – создается копия области, преданной в аргументе `hrgnSrc1`;

Функция возвращает значение, которое определяет тип результирующей области. Это могут быть значения: NULLREGION (результатирующая область пуста), SIMPLEREGION (результат является прямоугольником), COMPLEXREGION (результат является более сложной фигурой, чем прямоугольник), ERROR (область не создана).

При вызове функции CombineRgn область результата hrgnDest уже должна быть создана. Три области, используемые в аргументах, не обязаны быть разными. Например, область hrgnSrc1 может совпадать с hrgnDest.

**Пример 6.** В примере мы используем многие из описанных выше функций работы с областями.

```
#include <windows.h>
#include <iostream>
#include <conio.h>
using namespace std;

void main()
{
    HWND hwnd = GetConsoleWindow();
    HDC hdc = GetDC(hwnd);          // контекст консоли
    RECT clientRect;
    GetClientRect(hwnd, &clientRect); // получаем размеры окна
    Sleep(100);
    HBRUSH hBrush = CreateSolidBrush(RGB(0, 80, 0));
    HRGN bgRgn = CreateRectRgnIndirect(&clientRect);
    FillRgn(hdc, bgRgn, hBrush);    // красим фон рабочего окна

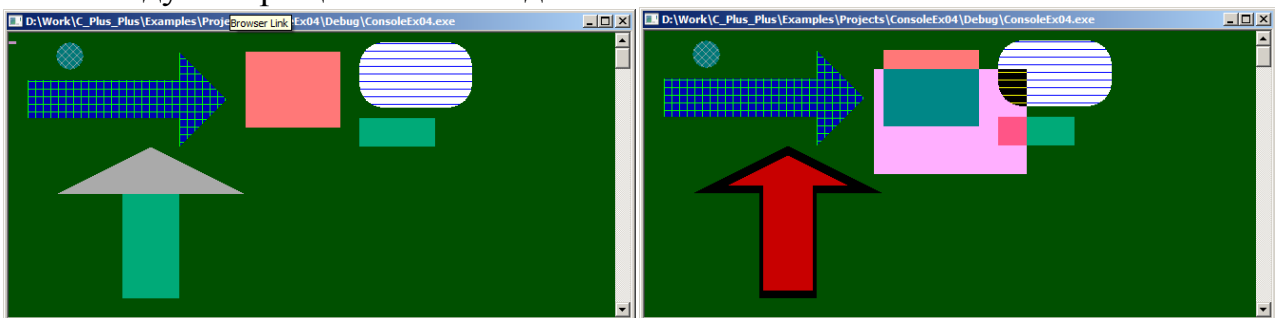
    //Прямоугольная область
    RECT rect1 = { 250, 20, 350, 100 };
    HRGN rgnRect1 = CreateRectRgnIndirect(&rect1);
    FillRgn(hdc, rgnRect1, CreateSolidBrush(RGB(255,120,120)));
    //Прямоугольная область
    HRGN rgnRect2 = CreateRectRgn(370,90,450,120);
    FillRgn(hdc, rgnRect2, CreateSolidBrush(RGB(0, 170, 120)));
    //Скругленная прямоугольная область.
    HRGN rgnRect3=CreateRoundRectRgn(370, 10, 490, 80, 50, 50);
    FillRgn(hdc, rgnRect3,
        CreateHatchBrush(HS_HORIZONTAL, RGB(0,0,255)));
    // эллиптическая область
    SetBkColor(hdc, RGB(0, 120, 120));
    HRGN rgnEll=CreateEllipticRgn(50, 10, 80, 40);
    FillRgn(hdc, rgnEll,
        CreateHatchBrush(HS_DIAGCROSS, RGB(120, 160, 160)));
    // Полигональная область
    SetBkColor(hdc, RGB(0, 0, 180));
    POINT ptArrow[7]={{20,50},{180,50},{180,20},{230,70},
        {180,120},{180,90},{20,90}};
    HBRUSH brsh1=CreateHatchBrush(HS_CROSS,RGB(0,255,0));
    HRGN bgRgn1 = CreatePolygonRgn(ptArrow, 7, WINDING);
    FillRgn(hdc, bgRgn1, brsh1);
```

```

// Треугольник
POINT ptTriangle[3]={{50,170},{150,120},{250,170}};
HRGN rgnTr = CreatePolygonRgn(ptTriangle, 3, WINDING);
FillRgn(hdc, rgnTr, CreateSolidBrush(RGB(170, 170, 170)));
// Прямоугольник под треугольником
HRGN rgnRect4=CreateRectRgn(120,170,180,280);
FillRgn(hdc, rgnRect4, CreateSolidBrush(RGB(0,170,120)));
_getch(); // ----- Первый экран программы -----
// Объединение областей. Результ. область должна существовать
int rez = CombineRgn(rgnTr, rgnTr, rgnRect4, RGN_OR);
// красный грибок
FillRgn(hdc, rgnTr, CreateSolidBrush(RGB(200, 0, 0)));
// Рамка вокруг грибка разной толщины
FrameRgn(hdc, rgnTr, CreateSolidBrush(RGB(0, 0, 0)), 4, 8);
// обращение цветов внутри прямоугольной области
HRGN rgnRect5 = CreateRectRgn(240, 40, 400, 150);
InvertRgn(hdc, rgnRect5);
_getch(); // ----- Второй экран программы -----
}

```

Ниже слева показан первый экран программы, справа – второй. На втором экране красная вертикальная стрелка (или грибок) являются единой областью и с ней следует обращаться как с одним объектом.



Есть еще одна полезная функция `OffsetRgn`. Она «сдвигает» область.

```
int OffsetRgn(HRGN hrgn, int nXOffset, int nYOffset);
```

Ее первый аргумент – дескриптор сдвигаемой области, второй и третий задают величину смещения по горизонтали и вертикали. Обе эти величины могут быть отрицательными. Заметим, что смещение области не приводит к ее автоматической перерисовке. Если вы желаете увидеть область в новом положении, то в начале ее надо стереть (например, закрасить цветом фона), затем сместить функцией `OffsetRgn`, а потом закрасить в новом положении. Следующий фрагмент поясняет эту последовательность команд.

```
FillRgn(hdc, rgnBall, bkBrsh);
OffsetRgn(rgnBall, rght, dwn);
FillRgn(hdc, rgnBall, fBrsh);
```

В этом фрагменте `hdc` является дескриптором консоли, `rgnBall` – сдвигаемой областью, а `bkBrsh` и `fBrsh` являются кистями фона и заливки области соответственно.

## Пример 7. Смещение области с помощью клавиш управления курсором.

```
#include <windows.h>
#include <conio.h>

#define KEY_ARROW_UP      72
#define KEY_ARROW_RIGHT  77
#define KEY_ARROW_DOWN   80
#define KEY_ARROW_LEFT   75

// Смещение круга (область rgnBall) на rght единиц вправо и
// dwn единиц вниз. Красим область цветом фона, смещаем область,
// заливаем заданным цветом
void drawBall(HDC hdc, HRGN rgnBall, HBRUSH bkBrsh,
              HBRUSH fBrsh, int rght, int dwn)
{
    FillRgn(hdc, rgnBall, bkBrsh);
    OffsetRgn(rgnBall, rght, dwn);
    FillRgn(hdc, rgnBall, fBrsh);
}

void main()
{
    HWND hwnd = GetConsoleWindow();
    HDC hdc=GetDC(hwnd);           // дескриптор консоли
    RECT clientRect;
    GetClientRect(hwnd, &clientRect); // получаем размеры окна
    Sleep(100);
    HBRUSH bkBrush = CreateSolidBrush(RGB(0, 80, 0));
    HRGN bgRgn = CreateRectRgnIndirect(&clientRect);
    FillRgn(hdc, bgRgn, bkBrush);    // заливаем фон консоли.

    intx1=50,y1=50,r1=30; // координаты центра шара и его радиус
    HBRUSH ballBrush1=CreateSolidBrush(RGB(200,0,0));
    HRGN rgnBall11=CreateEllipticRgn(x1-r1,y1-r1,x1+r1,y1+r1);
    FillRgn(hdc, rgnBall11, ballBrush1);

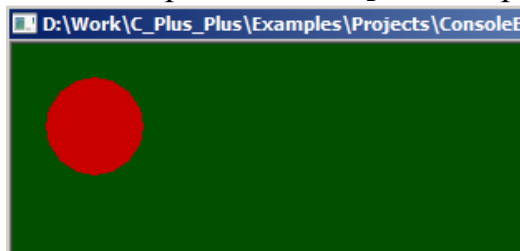
    int iKey = 67;
    while (iKey != 27) // Выход по клавише ESC
    {
        if (_kbhit())
        {
            iKey = _getch();
            switch (iKey)
            {
                case KEY_ARROW_UP:
                    drawBall(hdc, rgnBall11, bkBrush, ballBrush1, 0, -10);
                    break;
                case KEY_ARROW_RIGHT:
                    drawBall(hdc, rgnBall11, bkBrush, ballBrush1, 10, 0);
                    break;
                case KEY_ARROW_DOWN:
                    drawBall(hdc, rgnBall11, bkBrush, ballBrush1, 0, 10);
                    break;
            }
        }
    }
}
```

```

    case KEY_ARROW_LEFT:
        drawBall(hdc, rgnBall1, bkBrush, ballBrush1, -10, 0);
        break;
    }
}
}
}

```

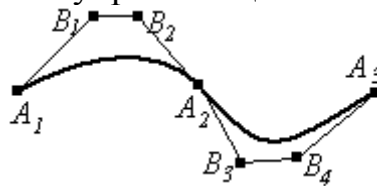
В этом примере мы перемещаем эллиптическую область внутри консоли, используя клавиши управления курсором. Для этого мы создали функцию `drawBall`, которая кроме дескриптора консоли `hdc` и области `rgnBall`, в качестве аргументов принимает кисти заливки фона `bkBrsh` и области `fBrsh`, а также величины смещения по горизонтали `right` и вертикали `dwn`.



Попробуйте перекрыть сверху окно консоли другим окном. Затем активизируйте консоль и понажимайте клавиши управления курсором. Что вы наблюдаете?

■

В библиотеке Windows API есть функция, которая рисует кубические кривые Безье. Это составные кривые, каждый сегмент которых является кубической параметрической кривой форма которой определяется вершинами многоугольника Безье. На следующем рисунке показана кривая Безье, составленная из двух сегментов. Точки  $A_1, A_2, A_3$  принадлежат кривой. Точки  $A_1, B_1, B_2, A_2$  и  $A_2, B_3, B_4, A_3$  являются вершинами двух многоугольников Безье. Точки  $B_1, B_2, B_3, B_4$  называются управляющими точками кривой.

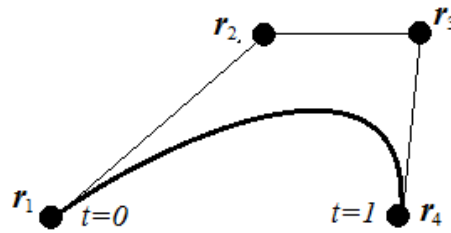


В параметрической форме сегмент кубической кривой Безье описывается уравнением

$$\mathbf{r}(t) = (1-t)^3 \mathbf{r}_1 + 3t(1-t)^2 \mathbf{r}_2 + 3t^2(1-t) \mathbf{r}_3 + t^3 \mathbf{r}_4,$$

где  $\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3, \mathbf{r}_4$  радиусы – векторы вершин многоугольника Безье, а параметр  $t$  на концах сегмента Безье принимает значения 0 и 1 (см. следующий рисунок). Точки  $\mathbf{r}_1, \mathbf{r}_4$  лежат на концах сегмента (кривая проходит через эти точки), а разности векторов  $\mathbf{r}_2 - \mathbf{r}_1$  и  $\mathbf{r}_4 - \mathbf{r}_3$  пропорциональны производным векторам  $\mathbf{r}'(0) = 3(\mathbf{r}_2 - \mathbf{r}_1)$  и  $\mathbf{r}'(1) = 3(\mathbf{r}_4 - \mathbf{r}_3)$  в концевых точках сегмента. Перемещая точки  $\mathbf{r}_2, \mathbf{r}_3$ , можно управлять направлением касательных на концах сегмента и, тем самым, управлять формой криволинейного сегмента.





Функция `PolyBezier` рисует кривую Безье, составленную из одного или более кубических сегментов.

```
BOOL PolyBezier(HDC hdc, const POINT *lppt, DWORD cPoints);
```

Ее второй аргумент является указателем на массив элементов типа `POINT`, который содержит координаты вершин многоугольников Безье. Третий аргумент задает количество точек в этом массиве. Например, следующий фрагмент кода строит один сегмент кривой Безье.

```
POINT arr[4]={{50,100},{100,10},{350,50},{200,100}};
PolyBezier(hdc, arr, 4);
```

Первый сегмент составной кривой проходит через первую и четвертую точки массива, используя вторую и третью точки в качестве управляющих. Каждый новый сегмент использует три следующие точки массива. При этом последняя точка предыдущего сегмента является начальной точкой следующего. Первые две точки из этих трех являются управляющими точками, а третья – конечной точкой текущего сегмента.

Функция `PolyBezier` указатель текущей позиции не использует и не обновляет. Кривая рисуется текущим пером.

Функция `PolyBezierTo` также рисует кривую Безье. При этом в качестве начальной точки первого сегмента используется текущее положение графического указателя.

```
BOOL PolyBezierTo(HDC hdc, const POINT *lppt, DWORD cPoints);
```

В результате для каждого сегмента кривой Безье требуется задать три точки. Функция обновляет положение текущего указателя, устанавливая его в последнюю вершину последнего сегмента Безье. Кривая рисуется текущим пером.

Работу функции `PolyBezier` мы продемонстрируем в следующем примере. Для этого нам понадобятся еще три функции Windows API.

Функция `PtInRegion` проверяет, попадает ли точка внутрь области.

```
BOOL PtInRegion(HRGN hrgn, int X, int Y);
```

Ее первый аргумент является дескриптором области, а `X` и `Y` представляют координаты точки. Если исследуемая точка принадлежит области, то функция возвращает ненулевое значение, иначе – ноль.

Функция `GetRgnBox` извлекает в свой второй аргумент охватывающий прямоугольник области, переданной ей в качестве первого аргумента.

```
int GetRgnBox(HRGN hrgn, LPRECT lprc);
```

Функция `GetCurrentConsoleFont` извлекает информацию о текущем шрифте консоли.

```
BOOL GetCurrentConsoleFont(HANDLE wHnd, BOOL bMaxWindow,  
    PCONSOLE_FONT_INFO lpConsoleCurrentFont);
```

Первый аргумент является идентификатором окна консоли. Если второй аргумент равен TRUE, то извлекается информация о шрифте при максимальном размере консоли. Если этот аргумент равен FALSE, то извлекается информация для окна текущего размера. Третий аргумент является указателем на переменную типа CONSOLE\_FONT\_INFO, которая принимает данные. Эта структура определена следующим образом

```
typedef struct _CONSOLE_FONT_INFO {  
    DWORD nFont;  
    COORD dwFontSize;  
} CONSOLE_FONT_INFO;
```

Ее первое поле nFont является индексом шрифта в таблице системных шрифтов консоли. Второе поле типа COORD содержит ширину и высоту символов (в пикселях).

В следующей программе мы используем функцию PolyBezier для построения одного сегмента кривой Безье. При этом мы будем строить также многоугольник Безье. Особенность программы состоит в том, что положение управляющих точек кривой мы будем контролировать с помощью мыши. Если в окрестности управляющей точки сделать щелчек левой кнопкой мыши с одновременным удерживанием нажатой левую клавишу CTRL, то соответствующая точка (точнее круг небольшого радиуса) будет выделяться цветом, идентифицируя ту из управляющих точек, которая будет перемещаться. Последующий однократный щелчек мыши в окне консоли сместит управляющую точку в место щелчка. Одновременно с этим будет перерисована кривая и ее многоугольник Безье. Если клавиша CTRL нажата, но щелчек мыши произошел вдали от управляющих точек, то выделение точки не происходит (или отменяется). Если не выбрана ни одна из двух управляющих точек, кривая Безье не перерисовывается.

В коде программы имеется одна особенность. В консольном окне мы можем определить место щелчка мыши с помощью функции ReadConsoleInput, которая помещает информацию о событиях в массив элементов типа INPUT\_RECORD. Эта структура имеет поле MouseEvent типа MOUSE\_EVENT\_RECORD. В ее поле dwMousePosition типа COORD будут находиться значения координат «текстового» курсора мыши. Чтобы определить «пиксельные» координаты точки щелчка мы используем размеры ячейки символов шрифта, которые получаем функцией GetCurrentConsoleFont. Например, если pozX, pozY являются текстовыми координатами места щелчка, а fontW и fontH задают ширину и высоту шрифта, то «пиксельные» координаты x, y мы вычисляем по формуле

```
x=pozX*fontW - 2;  
y=pozY*fontH - 4;
```

Небольшие добавки мы подобрали экспериментально.

## Пример 8. Демонстрация функции PolyBezier рисования кривой Безье.

```
#include <windows.h>

const int col=80;    // ширина окна консоли
const int row=25;    // высота консоли

void bigPoint(HDC hdc, int x, int y, int r, HBRUSH brush);
void drawBezier(HDC hdc, HPEN penBez, HPEN penLine, POINT *arr,
                HRGN rn1ControlPoint, HRGN rn2ControlPoint);

void main()
{
    // ----- 1. Оформление окна консоли (идентификаторы, шрифт, размеры, фон) -----
    HANDLE wHnd = GetStdHandle(STD_OUTPUT_HANDLE);
    HANDLE rHnd = GetStdHandle(STD_INPUT_HANDLE);
    HWND hwnd = GetConsoleWindow();
    HDC hdc = GetDC(hwnd);          // получаем дескриптор контекста консольного окна
    SetConsoleTitle(L"Bezier Curve Demo");

    // Получение информации о размере шрифта
    CONSOLE_FONT_INFO console_info;
    GetCurrentConsoleFont(wHnd, false, &console_info);
    int fontW, fontH;
    fontW = console_info.dwFontSize.X;          // ширина шрифта в пикселях
    fontH = console_info.dwFontSize.Y;          // высота шрифта в пикселях

    SMALL_RECT windowSize={0,0,col-1,row-1};   // Координаты вершин консольного окна
    SetConsoleWindowInfo(wHnd, TRUE, &windowSize); // Задание размеров окна консоли
    COORD bufferSize={col,row};                 // размера буфера
    SetConsoleScreenBufferSize(wHnd, bufferSize); // Изменение буфера
    Sleep(100);

    RECT clientRect;    //объявляем экземпляр структуры RECT - координаты прямоугольника
    GetClientRect(hwnd, &clientRect);           // получаем размеры окна
    HBRUSH bkBrush=CreateSolidBrush(RGB(0,120,120));
    FillRect(hdc, &clientRect, bkBrush);        // заливка фона окна
}
```

```

// ----- Рисование начального положения кривой Безье -----
HPEN penBez=CreatePen(PS_SOLID,3,RGB(255,0,0)); // перо для кривой Безье
HPEN penLine=CreatePen(PS_SOLID,1,RGB(0,255,0)); // перо для многоугольника Безье
HBRUSH inactiveBrush=CreateSolidBrush(RGB(200,200,200)); // кисть для неактивной управляющей точки
HBRUSH activeBrush=CreateSolidBrush(RGB(200,0,0)); // кисть для активной управляющей точки
int rgR = 10; // Радиус круга управляющей точки
POINT arrBez[4]={{250,200},{300,110},{550,150},{400,200}};
int rglx = arrBez[1].x, rgly = arrBez[1].y, rg2x = arrBez[2].x, rg2y = arrBez[2].y;
HRGN rn1ControlPoint=CreateEllipticRgn(rglx-rgR,rgly-rgR,rglx+rgR,rgly+rgR);
HRGN rn2ControlPoint=CreateEllipticRgn(rg2x-rgR,rg2y-rgR,rg2x+rgR,rg2y+rgR);
// рисуем сегмент и многоугольник Безье
drawBezier(hdc, penBez, penLine, arrBez, rn1ControlPoint, rn2ControlPoint);
HRGN rgnTek; // область выделенной точки

// ----- 3. Чтение и анализ буфера ввода. Реакция на события. -----
DWORD numEvents = 0; // Количество непрочитанных сообщений
DWORD numEventsRead = 0; // Количество прочитанных сообщений
bool isRunning = true; // флаг продолжения работы
int x, y, dx, dy;
RECT rgnRect;
BOOL bInPoint[2]={false,false}; // признаки попадания в области первой и второй точек
int rgnIndex;

while (isRunning) { //Ели isRunning=false, то программа завершается
    GetNumberOfConsoleInputEvents(rHnd, &numEvents); // Определить количество событий консоли
    if (numEvents != 0) {
        // выделение памяти для хранения данных о событиях
        INPUT_RECORD *eventBuffer = new INPUT_RECORD[numEvents];
        // Извлечение данных во временный буфер событий eventBuffer[]
        ReadConsoleInput(rHnd, eventBuffer, numEvents, &numEventsRead);
        // Цикл по всем извлеченным событиям
        for (DWORD i = 0; i < numEventsRead; ++i) {
            // нажата кнопка клавиатуры
            if (eventBuffer[i].EventType == KEY_EVENT) {

```

```

    if (eventBuffer[i].Event.KeyEvent.wVirtualKeyCode == VK_ESCAPE)
        isRunning = false;        // выход, если нажата клавиша ESC
}
else if (eventBuffer[i].EventType == MOUSE_EVENT) {
    // Событие левой кнопки мыши
    if (eventBuffer[i].Event.MouseEvent.dwButtonState & FROM_LEFT_1ST_BUTTON_PRESSED) {
        if (eventBuffer[i].Event.MouseEvent.dwEventFlags == 0) {
            // Выполнен однократный щелчек мыши
            if (eventBuffer[i].Event.MouseEvent.dwControlKeyState == 0) {
                // Управляющие клавиши не нажаты. Вычисляем «пиксельные» координаты точки щелчка
                x = eventBuffer[i].Event.MouseEvent.dwMousePosition.X*fontW-2;
                y = eventBuffer[i].Event.MouseEvent.dwMousePosition.Y*fontH-4;
                if (bInPoint[0]){rgnTek=rn1ControlPoint;rgnIndex=1;} // выделена 1-я управляющая точка
                else if (bInPoint[1]){rgnTek=rn2ControlPoint;rgnIndex=2;} //выдел. 2-я управляющая точка
                if (bInPoint[0] || bInPoint[1]) {
                    // получить старые координаты выделенной области (точки)
                    GetRgnBox(rgnTek, &rgnRect);
                    dx = x - rgnRect.left;                // величина сдвига области по горизонтали
                    dy = y - rgnRect.top;                // величина сдвига области по вертикали
                    FillRect(hdc, &clientRect, bkBrush); // полная очистка окна
                    OffsetRgn(rgnTek, dx, dy);           // смещение области
                    // получить новые координаты выделенной области (точки)
                    GetRgnBox(rgnTek, &rgnRect);
                    arrBez[rgnIndex].x = rgnRect.left + rgR;
                    arrBez[rgnIndex].y = rgnRect.top + rgR;
                    bInPoint[rgnIndex - 1] = false;
                    drawBezier(hdc, penBez, penLine, arrBez, rn1ControlPoint, rn2ControlPoint);
                }
            }
        }
    }
    else if (eventBuffer[i].Event.MouseEvent.dwControlKeyState==LEFT_CTRL_PRESSED){
        // Нажата левая CTRL и выполнен щелчек мыши.
        x = eventBuffer[i].Event.MouseEvent.dwMousePosition.X*fontW+3;
        y = eventBuffer[i].Event.MouseEvent.dwMousePosition.Y*fontH+5;
        // Проверка, что щелчек в окрестности одной из управляющих точек
        bInPoint[0]=PtInRegion(rn1ControlPoint,x,y);
    }
}

```

```

        bInPoint[1]=PtInRegion(rn2ControlPoint,x,y);
        if (bInPoint[0]) {
            FillRgn(hdc, rn1ControlPoint, activeBrush); // попали в область, меняем цвет
            FillRgn(hdc, rn2ControlPoint, inactiveBrush); // снять выделение 2-й точки
        }
        else if (bInPoint[1]) {
            FillRgn(hdc, rn2ControlPoint, activeBrush); // попали в область, меняем цвет
            FillRgn(hdc, rn1ControlPoint, inactiveBrush); // снять выделение 1-й точки
        }
        else {
            FillRgn(hdc, rn1ControlPoint, inactiveBrush); // снять выделение 1-й точки
            FillRgn(hdc, rn2ControlPoint, inactiveBrush); // снять выделение 2-й точки
        }
    }
}
}
}
}
}
delete[] eventBuffer;
}
}
}

// Функция рисования одного сегмента Безье и его многоугольника.
// Кривая рисуется пером HPEN penBez, а многоугольник пером HPEN penLine.
// POINT *arr -массив координат точек, точек всегда 4.
// HRGN rn1ControlPoint, HRGN rn2ControlPoint - дескрипторы областей управляющих точек.
void drawBezier(HDC hdc, HPEN penBez, HPEN penLine, POINT *arr,
                HRGN rn1ControlPoint, HRGN rn2ControlPoint)
{
    const int cnt = 4; // для кубического сегмента Безье нужны 4 точки
    SelectObject(hdc, penBez); // помещаем перо кривой в контекст
    PolyBezier(hdc, arr, cnt); // рисуем один сегмент кривой Безье
    HBRUSH pntBrush=CreateSolidBrush(RGB(0,0,255)); // кисть для концевых точек кривой (не управляющих)
    SelectObject(hdc, penLine); // помещаем перо ломаной в контекст
}

```

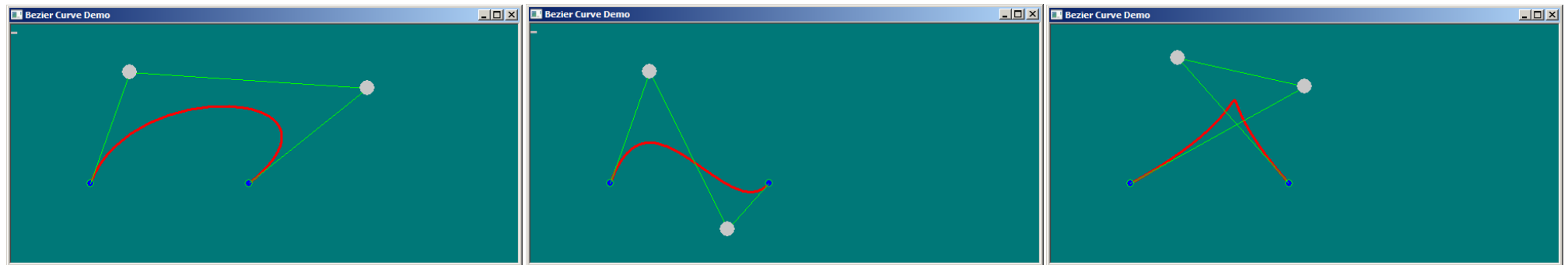
```

bigPoint(hdc, arr[0].x, arr[0].y, 5, pntBrush);           // начальная точка многоугольника
bigPoint(hdc, arr[cnt - 1].x, arr[cnt - 1].y, 5, pntBrush); // конечная точка многоугольника
Polyline(hdc, arr, cnt);                                 // рисуем многоугольник Безье
HBRUSH inactiveBrush = CreateSolidBrush(RGB(200, 200, 200));
FillRgn(hdc, rn1ControlPoint, inactiveBrush);           // заливаем область 1-й управляющей точки
FillRgn(hdc, rn2ControlPoint, inactiveBrush);           // заливаем область 2-й управляющей точки
}

// Рисуем круг с центром в точке (x,y) радиуса r кистью brush
void bigPoint(HDC hdc, int x, int y, int r, HBRUSH brush)
{
    SelectObject(hdc, brush);
    Ellipse(hdc, x - r, y - r, x + r, y + r);
}

```

На следующих рисунках показаны три различных сегмента Безье, имеющие одинаковые конечные точки и разные управляющие.



В программе созданы две вспомогательные функции. Функция `bigPoint` используется для рисования кругов, идентифицирующих концевые точки сегмента Безье. Функция `drawBezier` используется для рисования одного сегмента Безье и его многоугольника. Она принимает аргументы, необходимые для рисования сегмента и многоугольника Безье, назначение которых описано в комментарии к коду этой функции.

Попробуйте написать аналогичную программу для рисования кривой, составленной из двух сегментов Безье.

### 1.3 Заключение

В этом пособии мы привели описание небольшого количества функций Windows API, которые можно использовать в консольных приложениях. Значительно больше функций осталось за пределами нашего рассмотрения.

В первом параграфе мы описали «текстовые» функции работы с консолью. Во втором параграфе описали некоторые графические функции, которые, в принципе, не предназначены для работы с консолью. Помните, что основным режимом работы консольного окна является текстовый и графические возможности этого окна недостаточно функциональны. Тем не менее, все, что нами было изложено, без всяких изменений, но с некоторыми существенными улучшениями функциональности, используется в «оконных» приложениях Windows.

Надеемся, что наше небольшое введение в графические возможности консольных приложений послужит для вас стимулом к дальнейшему знакомству с графическими возможностями «настоящих» приложений Windows.