

Лабораторная работа № 1.

Создание оконного приложения *Windows*.

1. Цель работы

Ознакомиться со структурой оконного приложения *Win32*, реализацией обработки событий, простейшими способами отображения графических примитивов.

2. Описание лабораторного стенда

В качестве лабораторного стенда используется IBM-совместимый персональный компьютер с установленной интегрированной средой разработки программ *Dev-C++ 5.11*.

3. Подготовка к выполнению работы

- 3.1. Повторить по конспекту лекций и литературе следующие вопросы:
 - 3.1.1. основы программирования на языке C;
 - 3.1.2. использование указателей;
 - 3.1.3. основные приёмы работы в среде *Dev-C++*.
- 3.2. Подготовить электронный носитель для сохранения полученных результатов (файлов).

4. Программа работы

- 4.1. Создать папку для сохранения результатов работы. Создать проект *Dev-C++* и сохранить его в созданную папку.
- 4.2. Изучить структуру основной функции приложения *WinMain()*.
- 4.3. Изучить структуру оконной функции приложения *WndProc()*.
- 4.4. Изменить заголовок основного окна. Скомпилировать и запустить созданную заготовку приложения. Исследовать поведение окна, реализуемое операционной системой. Изменить цвет фона клиентской области окна, проверить результат.
- 4.5. Ознакомиться с общими принципами вывода графики в клиентской области окна. Написать, отладить и выполнить программу, выводящую в окно текст. Продемонстрировать результат преподавателю.
- 4.6. Ознакомиться с простейшими способами отображения графических примитивов в клиентской области окна. Написать, отладить и выполнить программу, выводящую в окно различные графические примитивы. Продемонстрировать результат преподавателю.
- 4.7. Модифицировать имеющуюся программу таким образом, чтобы она рисовала в окне фигуру в соответствии с индивидуальным заданием (см. п. 5). Продемонстрировать результат преподавателю.
- 4.8. Сохранить результаты работы (папку с проектом *Dev-C++*) на внешнем носителе, поскольку они потребуются для выполнения следующей работы.
- 4.9. Оформить отчет по работе.

5. Варианты заданий

Таблица 1.1. Варианты заданий

Вариант	1	2	3	4	5	6	7	8	9	10
Номер фигуры	1	2	3	4	5	6	1	2	3	4
Цвет 1	6	5	4	3	2	1	1	2	3	4
Цвет 2	1	2	3	4	5	6	4	3	2	1
Тип 1	1	2	3	1	2	3	2	1	3	2
Тип 2	2	3	1	2	3	2	1	3	2	1
Толщина 1	3	4	5	6	5	4	3	4	5	6
Толщина 2	6	5	4	3	4	5	6	5	4	3

Параметры «Тип 1», «Толщина 1», «Цвет 1» относятся к линиям, которыми нарисовано перекрестье, параметры «Тип 2», «Толщина 2» и «Цвет 2» - к линиям, образующим прямоугольник, окружность или эллипс.

Таблица 1.2. Виды фигур для рисования


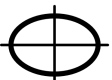




Номер	1	2	3	4	5	6
Вид						

Таблица 1.3. Номера цветов и типов линий

Номер	1	2	3	4	5	6
Цвет	красный	жёлтый	зелёный	синий	чёрный	серый
Тип	PS_SOLID (сплошная)	PS_DASH (пунктир)	PS_DOT (точки)			

6. Указания к выполнению работы.

6.1. К пункту 4.1

Поскольку среда разработки *Dev-C++* не всегда корректно работает с именами файлов, содержащими символы национальных алфавитов, рекомендуется использовать для имён файлов и папок **только цифры и английские буквы**. Кроме того, символы национальных алфавитов не должны присутствовать в путях к файлам.

Пример правильного имени папки: «D:\Work\MyProject».

Пример недопустимого имени папки: «C:\Users\Student\Рабочий стол\MyProject».

Для создания проекта необходимо запустить среду *Dev-C++* и выбрать в меню пункт «File / New / Project...». В появившемся окне «New Project» на вкладке «Basic» выбрать «**Windows Application**» и «**C++ Project**». В строке «Name» следует задать имя проекта, после чего нажать кнопку «Ok». В следующем окне нужно перейти в созданную ранее рабочую папку и нажать кнопку «Сохранить».

В результате *Dev-C++* сформирует заготовку проекта, реализующего простейшее оконное приложение *Win32*, состоящую из единственного файла «*main.cpp*». Рекомендуется сразу нажать кнопку «Save» и сохранить этот файл в рабочую папку.

В файле «*main.cpp*» находятся заготовки двух функций: основной функции приложения «*WinMain*» и обработчика событий «*WndProc*». В процессе разработки эти заготовки могут дополняться необходимыми элементами.

6.2. К пункту 4.2

Основная функция приложения *Win32*, вызываемая автоматически при запуске программы, называется не «*main*», как это было для консольных приложений, а «*WinMain*». Её подробное изучение выходит за рамки данной работы, рассмотрим лишь общий алгоритм этой функции (рис. 1.1).

Прежде всего описываются параметры нового класса окна, соответствующего основному окну создаваемого приложения. К этим параметрам, в частности, относятся вид курсора, иконка окна, цвет фона и некоторые другие. После того, как параметры заданы, предпринимается попытка зарегистрировать класс в системе (функция «*RegisterClassEx*»). В случае неудачи приложение завершает работу.

Далее на основе зарегистрированного класса создаётся экземпляр окна (функция «*CreateWindowEx*»). Поскольку ОС *Windows* позволяет создавать окна самого разнообразного вида и поведения, функция имеет множество параметров, отвечающих за это разнообразие. Для выполнения данной работы параметры, заданные в заготовке, изменять не следует (исключение составляют параметры *width* и *height*, задающие начальную ширину и высоту окна, а также *Caption*, определяющий заголовок окна, – их при желании можно изменить).

Если основное окно приложения создано удачно, функция *CreateWindowEx* возвращает так называемый дескриптор (*handle*) окна – уникальный идентификатор, позволяющий впоследствии обращаться к этому окну. Дескриптор сохраняется в переменной *hwnd*. Если же окно создать не удалось, функция возвращает *NULL*, в результате чего приложение завершает работу.

Далее начинает работать цикл обработки сообщений, о которых следует сказать более подробно. Приложения *Win32* относятся к классу программ, управляемых событиями. Это значит, что любое действие в программе выполняется лишь в ответ на некоторое событие, о котором программе стало известно. В качестве примеров событий можно привести нажатие клавиши, выбор пункта меню, перемещение мыши, срабатывание таймера, сворачивание, развёртывание, перемещение окна и многие-многие другие.

Приложения информируются о возникающих событиях с помощью сообщений. Источниками сообщений могут быть другие приложения, операционная система; приложение также может генерировать сообщения само для себя. В *Windows* определено несколько сотен стандартных сообщений, коды и назначение которых документированы; кроме того, разработчики приложений могут создавать собственные сообщения.

Цикл обработки сообщений реализован следующим образом. Прежде всего вызывается функция *GetMessage()*, запрашивающая у ОС код очередного сообщения. Функция не возвращает управление до тех пор, пока в системе не появится сообщение, предназначенное для нашей программы. Если же таких сообщений нет, то ОС предоставляет вычислительные ресурсы другим приложениям.

Если функция *GetMessage()* получила сообщение *WM_QUIT*, информирующее о необходимости завершить работу приложения, она возвращает нулевой результат, что приводит к завершению цикла обработки событий *while*. Для любого другого сообщения *GetMessage()* возвращает ненулевой результат.



Рис. 1.1. Алгоритм функции *WinMain*

Полученный код сообщения передаётся функции *TranslateMessage()*, которая выполняет дополнительную обработку сообщений, связанных с виртуальными клавишами. В нашем случае эта обработка интереса не представляет.

Наконец, функция *DispatchMessage()* отправляет полученное сообщение на обработку оконной функции *WndProc()*.

6.3. К пункту 4.3

Реакция приложения на события реализуется в так называемой оконной функции *WndProc()*. Функция получает четыре параметра: дескриптор окна, которому предназначено сообщение (*hwnd*), код сообщения *Message* и два дополнительных параметра сообщения: *wParam* и *lParam*. Смысл дополнительных параметров зависит от конкретного кода сообщения. Например, для сообщения *WM_MOUSEMOVE* (перемещение мыши) *wParam* содержит признаки нажатых кнопок, а *lParam* – координаты курсора. Для сообщения *WM_SIZE* (изменение размера окна) *wParam* содержит дополнительные признаки операции, а *lParam* – новую ширину и высоту окна.

В сложных приложениях реализация функции *WndProc* может быть очень громоздкой, хотя структура её проста: обычно она содержит единственный оператор *switch*, сравнивающий код полученного сообщения с кодами, на которые программа должна реагировать:

LRESULT CALLBACK

```
WndProc(HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam) {
    switch (Message) {

        case WM_MOUSEMOVE:
            // действия в ответ на перемещение мыши
            break;
        case WM_PAINT:
            // действия в ответ на перерисовку содержимого окна
            break;
        case WM_DESTROY:
            // действия в ответ на уничтожение окна
            break;
        // и так далее

        default:
            return DefWindowProc(hwnd, Message, wParam, lParam);
    }
    return 0;
}
```

Сообщения, не обработанные явным образом оконной функцией (а их может быть много), передаются в обработчик по умолчанию (*DefWindowProc*). Именно он позволяет нам не перерисовывать всевозможные меню, не заботиться о поведении стандартных элементов окна – всё это по умолчанию делает ОС. Если же какие-либо стандартные действия в конкретном приложении нужно выполнять иначе, то можно явным образом обрабатывать соответствующие сообщения, не передавая их в *DefWindowProc*.

6.4. К пункту 4.4

В созданной заготовке проекта основное окно приложения имеет заголовок «*Caption*». В разрабатываемом приложении имеет смысл заменить его на более информативный текст, например, «Моё первое приложение Win32». Это можно сделать путём изменения фактических параметров функции *CreateWindowEx* (см. п. 6.2).

Компиляция и запуск приложений Win32 ничем не отличается от аналогичных действий для консольных приложений: в меню среды разработки следует выбрать пункт «*Execute* /

Compile & Run». При отсутствии синтаксических ошибок программа запустится и на экране появится основное окно приложения с заданным заголовком.

Несмотря на то, что заготовка приложения не содержит описания каких-либо сложных действий, поведение основного окна достаточно осмысленно: его можно перемещать, изменять размер, оно может переходить из активного состояния в неактивное и обратно, имеет системное меню и выполняет команды, определённые в нём. Эти действия по умолчанию выполняются операционной системой стандартным способом, общим для всех окон.

При этом клиентская область основного окна представляет собой пустое поле, закрашенное цветом, заданным при описании класса окна (см. п. 6.2). Содержимое этой области должно определяться приложением.

Примечание. Попробуйте задать другой цвет фона клиентской области путём замены в функции *WinMain* строки

```
wc.hbrBackground = (HBRUSH) (COLOR_WINDOW+1);
```

на строку

```
wc.hbrBackground = (HBRUSH) (COLOR_MENUBAR);
```

Можно попробовать и другие константы цвета, определённые в файле «*winuser.h*» (чтобы быстро их найти, нажмите клавишу *Ctrl* и, не отпуская её, щелкните мышкой по тексту «*COLOR_MENUBAR*» или «*COLOR_WINDOW*»).

6.5. К пункту 4.5

6.5.1. Отрисовка содержимого клиентской области должна выполняться приложением в ответ на сообщение *WM_PAINT*. Это сообщение система направляет приложению при изменении размера окна, при его восстановлении из свёрнутого состояния, а также в других случаях, когда клиентская область или её часть становится видимой. Следовательно, приложение должно быть готово в любой момент перерисовать клиентскую область в ответ на сообщение *WM_PAINT*.

Если самому приложению необходимо в определённый момент времени перерисовать клиентскую область (например, для изменения отображаемой информации), то оно вызывает системную функцию *InvalidateRect* или *InvalidateRgn*, помечая всю клиентскую область или её часть, как требующую перерисовки. ОС, обнаружив наличие таких помеченных областей, посылает приложению сообщение *WM_PAINT* (это может произойти не сразу, а после выполнения системой более приоритетных задач).

Таким образом, для вывода изображений в клиентскую область в функции *WndProc* необходимо выполнить обработку сообщения *WM_PAINT*. Это можно сделать следующим образом:

```
LRESULT CALLBACK
WndProc(HWND hwnd, UINT Message, WPARAM wParam, LPARAM lParam) {
    switch(Message) {

        case WM_PAINT:
            ClientDraw(hwnd, wParam, lParam);
            break;
        // ..... - обработка других сообщений

    default:
        return DefWindowProc(hwnd, Message, wParam, lParam);
    }
    return 0;
}
```

В приведённом примере при получении сообщения *WM_PAINT* вызывается функция *ClientDraw*, в которой и должно выполняться рисование (такой подход позволяет не загромождать функцию *WndProc* деталями обработки каждого сообщения).

6.5.2. Функция *ClientDraw* может иметь следующий вид:

```
void ClientDraw(HWND hwnd, WPARAM wParam, LPARAM lParam) {
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hwnd, &ps);
    // .....
    // Операции вывода графики в клиентскую область
    // .....
    EndPaint(hwnd, &ps);
}
```

В качестве параметров она получает дескриптор окна, в котором будет выполняться рисование, а также параметры *wParam* и *lParam* сообщения *WM_PAINT*.

Для вывода графики в ОС *Windows* используется так называемый контекст графического устройства (*DC – device context*). Его можно сравнить с листом бумаги, на котором выполняется рисование. Свой контекст есть у каждого окна, и любой функции, выполняющей рисование, в качестве одного из параметров передаётся дескриптор (идентификатор) контекста.

Функция *ClientDraw* прежде всего вызывает системную функцию *BeginPaint*, которая готовит графический контекст окна к использованию. *BeginPaint* по окончании своей работы возвращает дескриптор контекста *hdc* – его мы будем использовать в дальнейшем при вызове графических функций.

После того, как контекст подготовлен, можно использовать его для вывода графики (примеры будут рассмотрены далее).

По окончании рисования вызывается функция *EndPaint* для освобождения системных ресурсов, занятых при вызове *BeginPaint*.

6.5.3. Рассмотрим несложный пример вывода текста в клиентскую область окна. Для этого дополним функцию *ClientDraw* следующим образом (нижеприведённый текст следует вставить до функции *WndProc*):

```
void ClientDraw(HWND hwnd, WPARAM wParam, LPARAM lParam) {
    char str[]="Ура! Мы сделали это!!!";
    PAINTSTRUCT ps;
    HDC hdc = BeginPaint(hwnd, &ps);
    TextOut(hdc, 50, 20, str, strlen(str));
    EndPaint(hwnd, &ps);
}
```

Функция *TextOut* выводит строку из массива *Str*, начиная с позиции (50, 20) в локальной системе координат окна (вообще все графические функции в приведённых примерах работают с использованием локальной системы координат). Следует отметить, что ось *Y* локальной системы координат направлена вниз, то есть при увеличении координаты «у» точка смещается не вверх, а вниз.

После компиляции и запуска программы на экране появится окно, содержащее заданный текст.

6.6. К пункту 4.6

Рассмотрим другие функции, позволяющие отображать в окне различные графические примитивы. Работу всех примеров, приведённых далее, следует проверять экспериментально путём последовательного дополнения ими функцию *ClientDraw* с

последующей компиляцией и выполнением программы. После включения всех примеров в указанную функцию следует продемонстрировать результат преподавателю.

6.6.1. Рисование линий

Для рисования линий можно использовать следующие функции (приведено упрощенное описание; полную документацию можно найти в [9.1]):

BOOL MoveToEx(HDC hdc, int X, int Y, LPPOINT lpPoint) – помещает текущую позицию (начальную точку рисования) в координаты (X, Y);

BOOL LineTo(HDC hdc, int nXEnd, int nYEnd) – рисует линию из текущей позиции в точку (nXEnd, nYEnd); перемещает текущую позицию в эту точку.

Пример:

```
MoveToEx(hdc, 50, 60, NULL);
LineTo(hdc, 80, 80);
LineTo(hdc, 40, 20);
MoveToEx(hdc, 30, 30, NULL);
LineTo(hdc, 50, 100);
```

Стиль, цвет и толщина выводимых линий определяются текущим пером (*Pen*). По умолчанию для контекста окна уже выбрано какое-то перо, до сих пор мы им и рисовали. Чтобы изменить параметры линий, следует создать и выбрать соответствующее перо:

```
LOGBRUSH lb; // структура, описывающая заливку линии
lb.lbStyle = BS_SOLID; // стиль заливки - сплошной
lb.lbColor = RGB(255, 0, 0); // цвет - красный
lb.lbHatch = 0; // штриховки нет

// создаём перо1: сплошная линия (PS_SOLID), ширина 5
HGDIOBJ hPen1 = ExtCreatePen(PS_GEOMETRIC | PS_SOLID, 5, &lb, 0, NULL);
// создаём перо2: пунктирная линия (PS_DASH), ширина 8
lb.lbColor = RGB(0, 255, 0); // цвет пера 2 - зелёный
HGDIOBJ hPen2 = ExtCreatePen(PS_GEOMETRIC | PS_DASH, 8, &lb, 0, NULL);

// выбираем новое перо (1), сохраняя старое в hPenOld
HGDIOBJ hPenOld = SelectObject(hdc, hPen1);
// Продолжаем рисовать пером 1
LineTo(hdc, 200, 100);
// выбираем перо 2 (старое можно не сохранять, мы его и так знаем)
SelectObject(hdc, hPen2);
// Продолжаем рисовать пером 2
LineTo(hdc, 50, 150);
// выбираем перо 1
SelectObject(hdc, hPen1);
// Продолжаем рисовать пером 1
LineTo(hdc, 200, 150);

// восстанавливаем перо по умолчанию
SelectObject(hdc, hPenOld);
// Уничтожаем созданные нами перья (освобождаем ресурсы)
DeleteObject(hPen1);
DeleteObject(hPen2);
```

Разумеется, уничтожать созданные перья следует лишь тогда, когда они больше не нужны. Перо, выбранное в настоящий момент, уничтожать не следует, предварительно нужно выбрать другое перо (то, которое использовалось по умолчанию).

Если в процессе рисования ломаной линии смена пера не требуется, можно использовать функцию *Polyline* или *PolylineTo*. Они отличаются лишь тем, что первая не изменяет текущую позицию, а вторая перемещает её в конец нарисованной линии (аналогично функции *LineTo*).

```
BOOL Polyline(HDC hdc, const POINT *lppt, int cPoints)
BOOL PolylineTo(HDC hdc, const POINT *lppt, int cPoints)
```

Второй параметр – указатель на массив координат точек (x, y), через которые пройдет ломаная линия, третий параметр – количество точек.

Дополним функцию *ClientDraw* следующими строками перед строкой *SelectObject(hdc, hPenOld)* и проверим результат работы программы:

```
// массив координат точек (x,y)
POINT Points[]={300,300, 350,300, 350,350, 200,300};
// рисуем линию через заданные четыре точки
Polyline(hdc, Points, 4);
```

6.6.2. Рисование прямоугольников

Прямоугольник, как, впрочем, и любой другой многоугольник, можно нарисовать линиями с помощью функций *LineTo*, *Polyline*, *PolylineTo*, однако, для этого есть более простой способ – использование функции *Rectangle*:

```
BOOL Rectangle(HDC hdc, int nLeftRect, int nTopRect, int nRightRect,
int nBottomRect);
```

Параметры со второго по пятый задают координаты верхней левой и нижней правой точек прямоугольника. Фигура рисуется с использованием текущего пера и закрашивается текущей кистью (по умолчанию – сплошная белая).

Изобразим прямоугольник, дополнив функцию *ClientDraw*:

```
SelectObject(hdc, hPen1);
Rectangle(hdc, 200, 20, 300, 50);
```

Нарисуем ещё один прямоугольник, выбрав другое перо и кисть. При этом используем одно из созданных перьев, а кисть возьмём стандартную с помощью функции *GetStockObject*:

```
SelectObject(hdc, hPen2); // выбираем перо
SelectObject(hdc, GetStockObject(GRAY_BRUSH)); // выбираем кисть
Rectangle(hdc, 50, 200, 100, 300);
```

Если закрашивать прямоугольник вообще не нужно, вместо *GRAY_BRUSH* следует использовать *NULL_BRUSH*.

6.6.3. Рисование эллипсов и окружностей

```
BOOL Ellipse(HDC hdc, int nLeftRect, int nTopRect, int nRightRect, int
nBottomRect);
```

Для рисования эллипса задаются те же параметры, что и для прямоугольника, однако функция выводит не прямоугольник, а вписанный в него эллипс. Если вместо прямоугольника задать квадрат, то вместо эллипса получится окружность:

```
SelectObject(hdc, hPen1); // выбираем перо 1
// Выбираем серую кисть
SelectObject(hdc, GetStockObject(GRAY_BRUSH));
// Рисуем окружность
Ellipse(hdc, 50, 350, 150, 450);
```



```
// Выбираем «пустую» кисть (нет заливки)
SelectObject(hdc, GetStockObject(NULL_BRUSH)) ;
// Рисуем эллипс
Ellipse(hdc, 200, 350, 500, 450) ;
```

Кроме функций, рассмотренных выше, Windows предоставляет множество других возможностей для вывода графики. Более подробную информацию об этом можно получить в [9.1].

6.7. К пункту 4.7

Для выполнения индивидуального задания необходимо привести функцию *ClientDraw* к виду, показанному в п. 6.5.2. Затем её нужно дополнить вызовами функций, рисующими в окне заданную фигуру. При этом в качестве образца следует использовать примеры, рассмотренные в п. 6.6.

7. Содержание отчёта.

Общие положения.

Отчёт должен содержать только сведения, описанные далее в этом разделе. Копирование иных материалов, приведённых в методических указаниях, **не допускается**.

Использование в отчёте экранных копий допускается **только** для окон, содержащих результат работы программы (т.е. окончательные результаты выполнения пунктов 4.6, 4.7). В текст вставляется только изображение окна разработанной программы, остальные части копии экрана должны быть удалены.

Исходные тексты программ, включенные в отчёт, должны быть отформатированы в соответствии с правилами, принятыми для языка программирования C++, аналогично тому, как это сделано в методических указаниях (отступы вложенных блоков, согласованное расположение парных открывающих и закрывающих скобок, одинаковые отступы для комментариев и т.п.). С целью упрощения форматирования и восприятия для исходных текстов следует использовать полужирный моноширинный шрифт (например «**Courier New**»). Фрагменты исходных текстов должны сопровождаться комментариями.

Отчёт по работе должен содержать:

- Название работы.
- Цель работы.
- Номер варианта и соответствующее ему индивидуальное задание из п. 5 (с расшифровкой обозначений цветов и типов линий). Копирование таблиц п. 5 в отчёт полностью **не допускается**.

Для отдельных пунктов программы работы в отчёте необходимо привести следующее.

Для пункта 4.2

- Схему алгоритма функции *WinMain()*.
- Исходный текст функции *WinMain()* с пояснением основных операций в виде комментариев.

Для пункта 4.3

- Схему алгоритма функции *WndProc()*, учитывающую изменения, добавленные в п. 4.5.
- Исходный текст функции *WndProc()*, соответствующий приведённой схеме алгоритма, с пояснением основных операций в виде комментариев.

Для пункта 4.6

- Окончательный вариант исходного текста функции *ClientDraw()*, полученного в данном пункте, с пояснением каждого действия в виде комментариев.

- Изображение основного окна разработанной программы.

Для пункта 4.7

- Исходный текст функции *ClientDraw()*, полученный в данном пункте, с пояснением каждого действия в виде комментариев.
- Изображение основного окна разработанной программы.

Заключительная часть

В заключительной части отчёта следует указать факультет, курс, группу, фамилию и подпись студента, выполнившего работу, а также дату составления отчёта. Кроме того, необходимо указать фамилию преподавателя, принимающего отчёт, и предусмотреть поля для оценки, подписи и даты защиты отчёта.

8. Контрольные вопросы.

- 8.1. Каково назначение основной функции приложения *WinMain*? Какие задачи она выполняет в разработанной программе?
- 8.2. Что такое «программа, управляемая событиями»? Как приложение *Windows* узнаёт о том или ином событии?
- 8.3. Что может служить источником сообщений, поступающих в программу? Приведите примеры сообщений.
- 8.4. Каким образом описываются сообщения в ОС *Windows*?
- 8.5. С какой целью в приложении *Windows* реализуется обработка сообщений? Все ли поступающие сообщения необходимо обрабатывать явным образом?
- 8.6. Каково назначение оконной функции *WndProc*? Опишите её структуру.
- 8.7. В каких случаях приложение *Windows* должно выполнять перерисовку клиентской области окна? Как приложение узнаёт о том, что необходимо выполнить перерисовку?
- 8.8. С помощью какой функции выполняется вывод текста в окно? Опишите параметры этой функции.
- 8.9. С помощью каких функций выполняется рисование линий, окружностей, эллипсов, прямоугольников? Опишите параметры какой-либо из этих функций.
- 8.10. Какими способами можно нарисовать в окне прямоугольник? Опишите особенности каждого из способов.
- 8.11. Какими способами можно нарисовать в окне ломаную линию? Опишите особенности каждого из способов.

9. Список литературы.

- 9.1. Каталог API (Microsoft) и справочных материалов [Электронный ресурс] URL: <http://msdn.microsoft.com/library>.

Лабораторная работа № 2.

Программирование реакции графических элементов интерфейса на действия пользователя.

1. Цель работы

Ознакомиться со способами реализации реакции графических объектов оконного приложения *Windows* на действия пользователя и внутренние события приложения.

2. Описание лабораторного стенда

В качестве лабораторного стенда используется IBM-совместимый персональный компьютер с установленной интегрированной средой разработки программ *Dev-C++ 5.11*.

3. Подготовка к выполнению работы

- 3.1. Подготовить работоспособный проект, полученный в результате выполнения лабораторной работы №1.
- 3.2. Подготовить электронный носитель для сохранения полученных результатов (файлов).

4. Программа работы

- 4.1. На рабочем компьютере создать папку для сохранения результатов работы. Скопировать в неё проект *Dev-C++* с результатами выполнения индивидуального задания ЛР №1.
- 4.2. Изменить функцию *ClientDraw()* таким образом, чтобы она выполняла рисование заданной фигуры относительно базовой точки, координаты которой хранятся в отдельной переменной.
- 4.3. Реализовать перемещение изображаемой фигуры в пределах клиентской области окна в ответ на нажатие клавиш управления курсором. Продемонстрировать результат преподавателю.
- 4.4. Реализовать перемещение изображаемой фигуры в пределах клиентской области окна в ответ на перемещение мыши. Продемонстрировать результат преподавателю.
- 4.5. Реализовать автоматическое перемещение изображаемой фигуры в пределах клиентской области окна по заданной траектории в соответствии с индивидуальным заданием. Продемонстрировать результат преподавателю.
- 4.6. Сохранить результаты работы (папку с проектом *Dev-C++*) на внешнем носителе, поскольку они потребуются для выполнения следующей работы.
- 4.7. Оформить отчет по работе.

5. Варианты заданий

Основные параметры индивидуального задания следует взять из п.5 ЛР №1. Дополнительно к ним в данной работе задаётся траектория перемещения фигуры. Виды

траекторий приведены в таблице 2.2, соответствие номера траектории варианту индивидуального задания – в таблице 2.1.

Таблица 2.1. Варианты заданий

Вариант	1	2	3	4	5	6	7	8	9	10
Траектория перемещения	1	2	3	4	5	6	5	4	3	2

Таблица 2.2. Траектории перемещения фигуры

Номер	1	2	3	4	5	6
Вид						

6. Указания к выполнению работы.

6.1. К пункту 4.2

6.1.1. В предыдущей работе координаты всех графических примитивов при рисовании задавались относительно начала координат клиентской области – точки (0, 0). Это приемлемо в случае вывода статических изображений, но неудобно, если изображение необходимо перемещать в пределах окна: при таком способе задания координат для перемещения фигуры придётся изменить координаты *каждой* ключевой точки *каждого* графического примитива, составляющего фигуру (т.е. по две точки для линий эллипсов и прямоугольников). Чем сложнее фигура, тем большее количество ключевых точек она имеет.

Другой подход заключается в том, чтобы рисовать сложную фигуру относительно некоторой базовой точки, координаты которой хранятся в специальной переменной. Тогда для изменения положения фигуры в пределах окна потребуется изменить лишь координаты базовой точки, остальные параметры изображения и алгоритм его вывода в окно останутся неизменными.

6.1.2. В качестве примера рассмотрим фигуру, приведённую на рис. 2.1 и состоящую из двух пересекающихся линий. Чтобы изобразить её тем способом, который использовался в ЛР №1, в функции *ClientDraw()* потребуется выполнить следующее:

```
// Первая линия
// Начинаем с точки (4,1)
MoveToEx(hdc, 4, 1, NULL);
// Линия в точку (8,5)
LineTo(hdc, 8, 5);
// Вторая линия
// Начинаем с точки (2,5)
MoveToEx(hdc, 2, 5, NULL);
// Линия в точку (8,2)
LineTo(hdc, 8, 2);
```

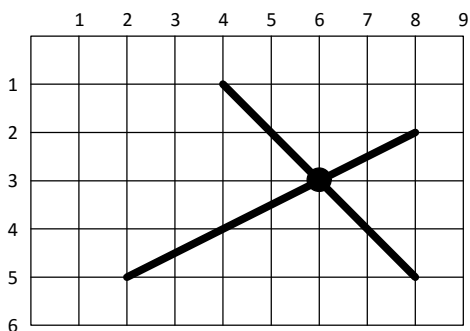


Рис. 2.1. Рисование линий

Если теперь потребуется переместить эту фигуру в другую позицию (например, на две единицы право и одну единицу вниз), то придётся в каждой строке приведённого фрагмента программы изменить значения координат, предварительно вычислив их.

6.1.3. Теперь рассмотрим другой способ рисования фигуры. Выберем некоторую точку в качестве базовой. В общем случае выбор может быть произвольным, но обычно бывает удобно выбирать в качестве базовой какую-либо характерную точку фигуры. Пусть это будет точка пересечения линий с координатами (6, 3). Для хранения координат точки в *Windows API* используется специальная структура *POINT*, имеющая два поля: *x* и *y*. Опишем глобальную переменную для хранения координат базовой точки и присвоим ей начальное значение, соответствующее текущим координатам точки пересечения линий:

```
POINT Base = {6,3};
```

Далее в функции *ClientDraw()* изобразим линии, задавая их координаты относительно базовой точки:

```
// Первая линия
// Начинаем с точки, отстоящей от базовой на -2 по X и -2 по Y
MoveToEx(hdc, Base.x-2, Base.y-2, NULL);
// Линия в точку, отстоящую от базовой на +2 по X и +2 по Y
LineTo(hdc, Base.x+2, Base.y+2);
// Вторая линия
// Начинаем с точки, отстоящей от базовой на -4 по X и +2 по Y
MoveToEx(hdc, Base.x-4, Base.y+2, NULL);
// Линия в точку, отстоящую от базовой на +2 по X и -1 по Y
LineTo(hdc, Base.x+2, Base.y-1);
```

Как видно из приведённого текста, он содержит вызовы тех же функций, что и ранее, но использует иной, на первый взгляд более сложный, способ задания координат. Однако такой подход позволит нам выполнять перемещение фигуры существенно проще: для этого потребуется всего лишь изменить координаты базовой точки и повторить без изменений основную часть программы рисования. Например, чтобы переместить фигуру на две единицы вправо и одну единицу вниз нужно скорректировать координаты базовой точки:

```
Base.x+=2; Base.y++;
```

Затем вызывается функция рисования *ClientDraw()*, которая изобразит фигуру уже в новой позиции.

6.1.4. Для выполнения п.4.2 программы работы следует изменить исходный текст программы, описав глобальную переменную *Base* как показано выше. Затем необходимо изменить функцию *ClientDraw()* таким образом, чтобы она выполняла рисование фигуры из индивидуального задания способом, описанным в п. 6.1.3, т.е. относительно базовой точки.

Для проверки программы нужно скомпилировать и выполнить её несколько раз, задавая различные координаты базовой точки и наблюдая изменение положения фигуры (при этом форма фигуры не должна искажаться).

6.2. К пункту 4.3

При нажатии клавиш на клавиатуре *Windows* посылает активному окну сообщение *WM_KEYDOWN*. Следовательно, чтобы реагировать на нажатия клавиш, программа должна обрабатывать это сообщение. Добавим обработку сообщения *WM_KEYDOWN* в оконную функцию *WndProc()*:

```
case WM_KEYDOWN:
    ProcessKey(hwnd, wParam, lParam);
    break;
```

В ответ на сообщение вызывается функция *ProcessKey()*, которой передаётся дескриптор окна и параметры сообщения, содержащие код клавиши (*wParam*) и дополнительную информацию (*lParam*).

Функция **ProcessKey()** должна быть описана до *WndProc()* и может выглядеть следующим образом:

```
void ProcessKey(HWND hwnd, WPARAM wParam, LPARAM lParam) {
    // проверяем код нажатой клавиши
    switch (wParam) {
        // VK_UP – код клавиши «Стрелка вверх»
        case VK_UP:    Base.y-=Step.y;    break;
        // VK_DOWN – код клавиши «Стрелка вниз»
        case VK_DOWN:  Base.y+=Step.y;    break;
        // ...
        // реакция на другие клавиши
        // ...
    }
    // помечаем всю клиентскую область окна, как требующую перерисовки
    InvalidateRect(hwnd, NULL, 1);
}
```

В ответ на нажатие клавиш «Вверх» и «Вниз» функция изменяет координату «у» базовой точки на величину *Step.y*. В конце вызывается функция *InvalidateRect()*, требующая перерисовки клиентской области окна при появлении такой возможности.

Для нормальной работы функции *ProcessKey()* нужно описать глобальную переменную *Step* типа *POINT* для хранения величины шага перемещения фигуры по вертикали (*Step.y*) и по горизонтали (*Step.x*):

```
POINT Step = {4,4};
```

После компиляции программы и проверки перемещения фигуры вверх и вниз функцию *ProcessKey()* нужно дополнить обработкой кодов клавиш *VK_LEFT* «Стрелка влево» и *VK_RIGHT* «Стрелка вправо», обеспечив соответствующее перемещение фигуры. Добавьте также обработку клавиши «Home» (код *VK_HOME*), обеспечив возвращение фигуры в некоторую исходную позицию. Измените значения полей переменной *Step*, наблюдайте и объясните результат.

6.3. К пункту 4.4

При перемещении мыши в границах клиентской области окна *Windows* посылает ему сообщение *WM_MOUSEMOVE*. Следовательно, чтобы реагировать на движение мыши, программа должна обрабатывать это сообщение. Обработка сообщения *WM_MOUSEMOVE* выполняется аналогично обработке *WM_KEYDOWN*. Дополним функцию *WndProc()* следующими строками:

```
case WM_MOUSEMOVE:
    ProcessMouse(hwnd, wParam, lParam);
    break;
```

Параметр *wParam* содержит признаки кнопок мыши и клавиатуры, нажатых во время перемещения, а *lParam* – координаты курсора (x – младшая пара байтов, y – старшая).

Функция **ProcessMouse()** должна быть описана до *WndProc()* и может выглядеть следующим образом:

```
void ProcessMouse(HWND hwnd, WPARAM wParam, LPARAM lParam) {
    // Проверяем, установлен ли флаг нажатия левой кнопки мыши
    // Реагируем на мышшь только при нажатой левой кнопке
    if (wParam & MK_LBUTTON) {
        // Получаем координаты x и y указателя мыши
        // и перемещаем туда базовую точку
        Base.x = lParam % 0x10000;
        Base.y = lParam / 0x10000;
        // Информлируем ОС о необходимости перерисовки окна
        InvalidateRect(hwnd, NULL, 1);
    }
}
```

После доработки программы и проверки её работоспособности измените функцию *ProcessMouse()* так, чтобы фигура двигалась за мышью:

- только при нажатой правой кнопке мыши (флаг «MK_RBUTTON»);
- только при нажатой клавише «Shift» на клавиатуре (флаг «MK_SHIFT»);
- в любом случае независимо от нажатия кнопок.

6.4. К пункту 4.5

6.4.1. Чтобы фигура автоматически перемещалась в пределах окна, требуется с некоторой периодичностью формировать определённое сообщение, а в ответ на это сообщение корректировать координаты базовой точки, после чего перерисовывать клиентскую область. Во многом это напоминает обработку нажатий клавиш, рассмотренную в п. 6.2.

Для периодического формирования сообщений удобно использовать специальный объект *Windows* – таймер. Для создания таймера используется функция *SetTimer()*, описанная следующим образом:

```
UINT_PTR SetTimer(
    HWND      hwnd, // дескриптор окна, которому направляются сообщения
    UINT_PTR  nIDEvent, // уникальный идентификатор таймера
    UINT      uElapse, // период таймера, мс
    TIMERPROC lpTimerFunc // функция, вызываемая при срабатывании
);
```

После создания таймер начинает работать и с периодичностью *uElapse* направляет заданному окну сообщение *WM_TIMER* с параметром *wParam*, равным *nIDEvent*. Идентификатор *nIDEvent* должен быть уникальным для каждого таймера (если их несколько). После того, как таймер станет не нужен, его следует уничтожить при помощи функции *KillTimer()*.

Так как в нашей программе будет всего один таймер, в качестве *nIDEvent* можно указать любое число. Однако более правильно в начале программы определить константу с понятным именем и произвольным значением, а затем использовать в качестве идентификатора её:

```
#define TIMER_ID 1234
```

Изменим программу, добавив в неё создание и уничтожение таймера, формирующего сообщения *WM_TIMER* каждые 30 мс:

```
// Перед основным циклом обработки сообщений создаём таймер
SetTimer(hwnd, TIMER_ID, 30, NULL);
while(GetMessage(&msg, NULL, 0, 0) > 0) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
}
// По окончании работы программы уничтожаем таймер
KillTimer(hwnd, TIMER_ID);
return msg.wParam;
```

6.4.2. Для обработки сообщения *WM_TIMER* дополним функцию *WndProc()*:

```
case WM_TIMER:
    if (wParam==TIMER_ID) // если сообщение от «нужного» таймера,
        NextMoveStep(hwnd); // то выполняем очередной шаг перемещения
    break;
```

Функция *NextMoveStep()* должна быть описана до *WndProc()* и может выглядеть следующим образом:

```
void NextMoveStep(HWND hwnd) {
    // смещаем базовую точку
    Base.x+=Step.x;
    Base.y+=Step.y;
    // помечаем всю клиентскую область окна, как требующую перерисовки
    InvalidateRect(hwnd, NULL, 1);
}
```

После внесения в программу описанных изменений её необходимо скомпилировать и добиться работоспособности.

6.4.3. Рассмотренный в п. 6.4.2 обработчик сообщений от таймера обеспечивает лишь прямолинейное движение фигуры и со временем выводит её за пределы видимой клиентской области окна. Чтобы обеспечить движение фигуры по заданной траектории, необходимо контролировать её текущие координаты и своевременно изменять направление движения.

Подобные задачи удобно решать путём создания так называемой машины состояний. Для неё характерно наличие конечного числа различных состояний процесса и определённое поведение в каждом из состояний.

Пусть, например, необходимо обеспечить циклическое движение фигуры по замкнутой траектории в форме прямоугольного треугольника: сначала вниз, затем вправо, затем влево-вверх (рис. 2.2). Очевидно, что поведение фигуры имеет три состояния: движение вниз, движение вправо и движение влево-вверх. Перечисленные состояния нужно каким-либо образом обозначить, в простейшем случае пронумеровать. Но поскольку номера состояний сложно запоминать, введём несколько констант с уникальными значениями и с понятными именами, соответствующими различным состояниям:

```
#define MS_DOWN      0 // движение вниз
#define MS_RIGHT     1 // движение вправо
#define MS_TOPLEFT   2 // движение влево-вверх
```

Ещё более правильно было бы для представления состояний определить перечислимый тип (*enum*), но для упрощения текста программы используем обычные целочисленные константы.

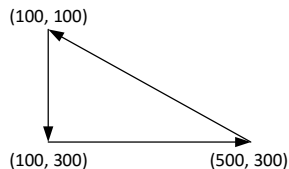


Рис. 2.2. Пример траектории движения

Затем необходимо описать глобальную целочисленную переменную, в которой будет храниться текущее состояние движения:

```
int MoveState = MS_DOWN; // начальное состояние – движение вниз
```

Функция *NextMoveStep()* при каждом вызове должна проверять, в каком состоянии сейчас находится фигура, и корректировать её координаты в соответствии с этим состоянием. Кроме того, в каждом состоянии нужно проверять текущие координаты и при достижении очередной точки излома траектории переходить в следующее состояние.

```
void NextMoveStep(HWND hwnd) {
    // Проверяем текущее состояние
    switch (MoveState) {
        case MS_DOWN: // движение вниз
            // Если координата y достигла значения 300, переходим
            // в состояние «движение вправо»
            if (Base.y>300) MoveState = MS_RIGHT;
            // а иначе продолжаем движение вниз
            else Base.y+=Step.y;
            break;
        case MS_RIGHT: // движение вправо
            // Если координата x достигла значения 500, переходим
            // в состояние «движение влево-вверх»
            if (Base.x>500) MoveState = MS_TOPLEFT;
            // а иначе продолжаем движение вправо
            else Base.x+=Step.x;
            break;
        case MS_TOPLEFT: // движение влево-вверх
            // Если координата x стала меньше 100, переходим
            // в исходное состояние «движение вниз»
            if (Base.x<100) MoveState = MS_DOWN;
            // а иначе продолжаем движение влево-вверх
            else {
                Base.x-=Step.x;
                Base.y-=Step.y/2;
            }
            break;
        // Если по какой-то причине переменная состояния имеет иное
        // значение, возвращаем её в исходное состояние
        default: MoveState = MS_DOWN;
    }
    // помечаем всю клиентскую область окна, как требующую перерисовки
    InvalidateRect(hwnd, NULL, 1);
}
```

После внесения в программу описанных изменений её необходимо скомпилировать и добиться работоспособности.

6.4.4. Для выполнения индивидуального задания необходимо по аналогии с примером, рассмотренным в п. 6.4.3, определить перечень состояний для заданной траектории движения, описать соответствующие константы состояний и модифицировать текст функции *NextMoveStep()* для корректного перемещения фигуры в каждом состоянии и своевременного переключения состояний. Координаты точек излома траектории можно выбирать произвольно с сохранением формы траектории (предполагается, что линии траектории направлены вертикально, горизонтально или под углом 45°).

7. Содержание отчёта.

Общие положения.

См. п. 7 лабораторной работы №1.

Отчёт по работе должен содержать:

- Название работы.
- Цель работы.
- Номер варианта и соответствующее ему индивидуальное задание из п. 5 (с расшифровкой обозначений цветов и типов линий). Копирование таблиц п. 5 в отчёт полностью **не допускается**.

Для отдельных пунктов программы работы в отчёте необходимо привести следующее.

Для пункта 4.2

- Исходный текст функции *ClientDraw()* с пояснением основных операций в виде комментариев.

Для пункта 4.3

- Исходный текст функции *ProcessKey()* с пояснением основных операций в виде комментариев.

Для пункта 4.4

- Исходные тексты всех самостоятельно полученных вариантов функции *ProcessMouse()* с пояснением основных операций в виде комментариев.

Для пункта 4.5

- Описание констант, соответствующих состояниям движения фигуры.
- Исходные тексты функций *NextMoveStep()* и *WndProc()*, соответствующие индивидуальному заданию, с пояснением основных операций в виде комментариев.

Заключительная часть

В заключительной части отчёта следует указать факультет, курс, группу, фамилию и подпись студента, выполнившего работу, а также дату составления отчёта. Кроме того, необходимо указать фамилию преподавателя, принимающего отчёт, и предусмотреть поля для оценки, подписи и даты защиты отчёта.

8. Контрольные вопросы.

- 8.1. Как ОС *Windows* уведомляет приложение о нажатии клавиши? Опишите параметры соответствующего сообщения.
- 8.2. Как ОС *Windows* уведомляет приложение о перемещении мыши? Опишите параметры соответствующего сообщения.
- 8.3. Для чего в ОС *Windows* используются таймеры? Опишите поведение таймера, функции для работы с таймером, используемые в программе, и их параметры.
- 8.4. В чём преимущества задания координат графических примитивов относительно базовой точки при рисовании сложных фигур?
- 8.5. Почему при перемещении фигуры с помощью клавиатуры она может полностью выходить за пределы клиентской области окна, а с помощью мыши – нет?
- 8.6. Для чего используются состояния при программировании движения по сложной траектории?