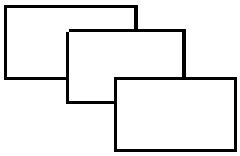
№ 11 Создание приложений и в Win32 API C++

Создайте пустой проект типа Win32 Application. Создайте главное окно приложения определенного размера с заголовком, определенного цвета и стиля. Напишите обработчики сообщений

Вариант	Задание
1,10,19	реакция на щелчок правой кнопки мыши - меняется заголовок
2,11,20	реакция на щелчок левой кнопки мыши - меняется заголовок
3,12,21	реакция на двойной щелчок левой кнопки мыши - меняется заголовок
4,13,22	реакция на ввод символа - меняется позиция окна
5,14,23	реакция на щелчок правой кнопки мыши - меняется размер окна
6,15,24	реакция на перемещение мыши - меняется размер и позиция окна
7,16,26	реакция на двойной щелчок правой кнопки мыши - меняется размер окна и заголовок
8,17,26	реакция на двойной щелчок левой кнопки мыши - меняется размер и позиция окна
9,18,27	реакция на щелчок левой кнопки мыши - меняется размер и позиция окна, заголовок окна

При закрытии приложения (сообщение WM_CLOSE) выведите окно сообщений MessageBox с двумя кнопками - OK и Cancel. Сделайте так, чтобы была активна вторая кнопка для чет. варианта и первая для нечет.

Добавьте реакцию на щелчок левой кнопки мыши: добавляется временное окно, в заголовке которого - порядковый номер окна. При повторном нажатии добавляется еще одно окно с соответствующим заголовком (максимум 5 окон). Окна должны быть расположены каскадом. (дополнительно, цвет фона дочерних окон случайный).



При нажатии любого символа окна должны закрываться в обратном порядке.

Методические указания

Краткие теоретические сведения

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

Функция окна

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

Цикл сообщений

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

Класс окна

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

Существует три основных типа окон - перекрывающиеся, всплывающие и дочерние, из которых можно создавать множество самых разнообразных объектов, комбинируя биты стиля. Перекрывающиеся (overlapped window) - основной, наиболее универсальный тип окон. Для их создания используется стиль WS_OVERLAPPEDWINDOW. Вспомогательный или всплывающие окна (popup window) Используется стиль WS_POPUP. Используются для диалоговых окон и окон сообщений. Дочерние окна (child window) Они связаны некоторыми характеристиками с главным окном, из которого они были созданы.

Типы данных в Windows

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

Программа для Windows

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

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

```
        Обработать полученное сообщение
        Возврат
}
```

Функция WinMain

Рассмотрим простейшее приложение Win32:

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

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

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

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

lpCmdLine - указатель на командную строку;

nCmdShow - код режима начального отображения главного окна приложения;

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

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

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

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

Регистрация класса окна

Для регистрации класса окна рекомендуется использовать функцию

RegisterClassEx или RegisterClass.

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

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

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

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

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

Для загрузки курсора используется функция LoadCursor:

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

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

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

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

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

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

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

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

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

Пример регистрации класса окна:

```
ATOM MyRegisterClass(HINSTANCE hInstanse)
{WNDCLASSEX wcex;
 wcex.cbSize         =sizeof(WNDCLASSEX);
 wcex.style                  =CS_HREDRAW|CSVREDRAW;
 wcex.lpfnWndProc                =(WNDPROC)WndProc;
 wcex.cbClsExtra             =0;
```

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

Сообщения и функция обратного вызова

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

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

```
LRESULT CALLBACK WndProc(
  HWND hwnd,      //дескриптор оконного объекта
  UINT uMsg,      //код сообщения
```

```
  WPARAM wParam,  //первый параметр сообщения
  LPARAM lParam   //второй параметр сообщения
);
```

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

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

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

Параметры функции:

HWND hwnd - декриптор объекта окна, для которого предназначено сообщение, если создано несколько объектов данного оконного класса, то по нему обработчик выбирает нужный; для главного окна приложения, которое существует в одном экземпляре его можно проигнорировать;

UINT uMsg - код сообщения, беззнаковое целое;

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

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

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

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

Теперь можно написать простейшую функцию обработки сообщений:

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

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

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

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

Создание главного окна приложения

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

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

```
                    //который создает окно
LPVOID lpParam      //указатель на дополнительные параметры
);
```

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

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

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

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

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

WS_CHILD - создает дочернее окно;

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

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

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

WS_SYSMENU - окно имеет системное меню;

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

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

WS_OVERLAPPEDWINDOW - рекомендуемый стиль перекрывающегося окна для главного окна приложения, определен как битовая комбинация

следующих стилей: WS_OVERLAPPED, WS_CAPTION, WS_SYSMENU, WS_THICKFRAME, WS_MINIMIZEBO, WS_MAXIMIZEBOX.

Для создания главного окна приложения можно использовать следующую функцию:

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

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

Цикл обработки сообщений

Простейший цикл обработки сообщений выглядит следующим образом:

```
while(GetMessage(&msg,NULL,0,0))
        {       TranslateMessage(&msg);
                DispatchMessage(&msg)
        }
```

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

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

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

Пример функции WinMain()

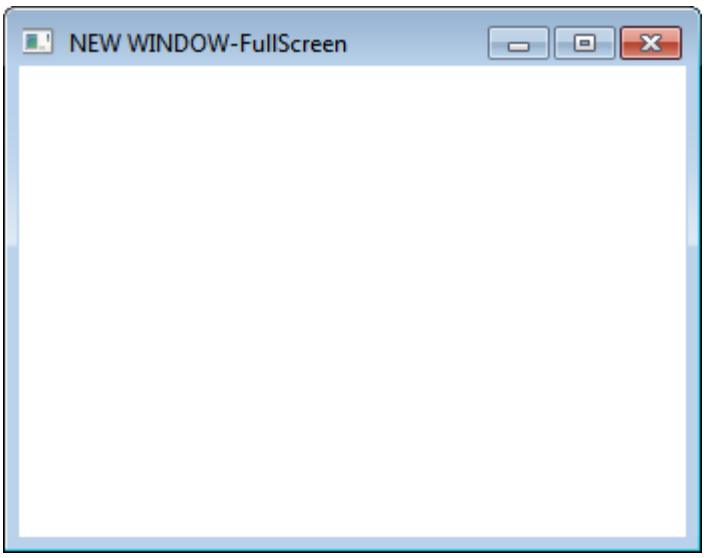
```
int WINAPI WinMain(HINSTANCE     hInstance,
                   HINSTANCE     hPrevInstance,
                   LPSTR         lpCmdLine,
                   int           nCmdShow)
                         {
MSG msg;
MyRegisterClass(hInstanse);
if(!InitInstance(hInstance,nCmdShow)) return FALSE;
while(GetMessage(&msg,NULL,0,0))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

Пример проекта Win32 API

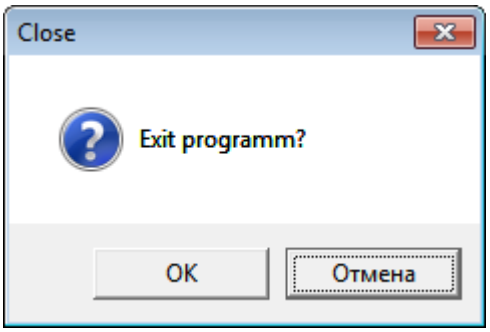
```
#include "stdafx.h"
#include "Task1.h"
#include <windows.h>
LONG WINAPI WndProc(HWND, UINT, WPARAM,LPARAM);
int WINAPI WinMain(      HINSTANCE hInstance,
                         HINSTANCE hPrevInstance,
                         LPSTR lpCmdLine,
                         int nCmdShow)
{   HWND hwnd;
   MSG msg;
   WNDCLASS w;
   memset(&w,0,sizeof(WNDCLASS));
   w.style = CS_HREDRAW|CS_VREDRAW;
   w.lpfnWndProc = WndProc;
   w.hInstance = hInstance;
   w.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
   w.lpszClassName = L"My Class";
```

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

        MB_OKCANCEL|MB_ICONQUESTION|MB_DEFBUTTON2))
              SendMessage(hwnd, WM_DESTROY, NULL, NULL);
              break;
    default:  return DefWindowProc(hwnd, Message, wparam, lparam);
  }
  return 0;
}
```

Результат выполнения программы.

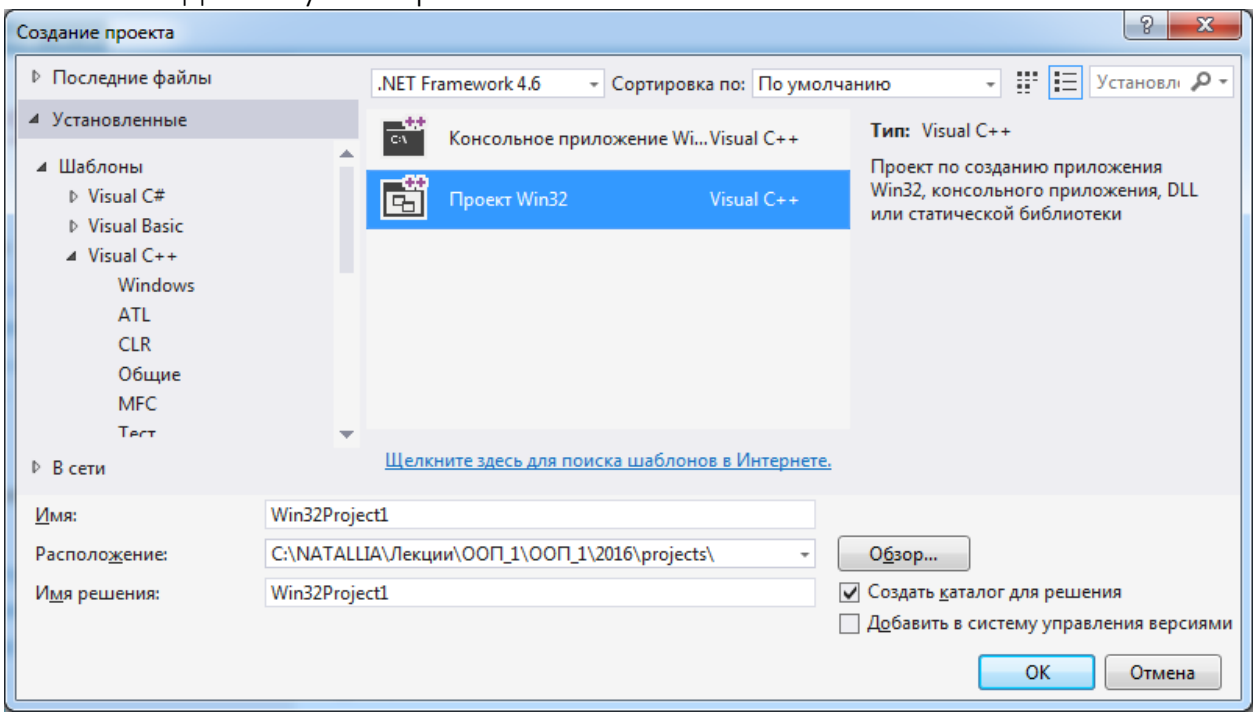


При закрытии программы появляется диалоговое окно, следующего вида:



Пример выполнения задания

1.Создайте пустой проект типа Win32.



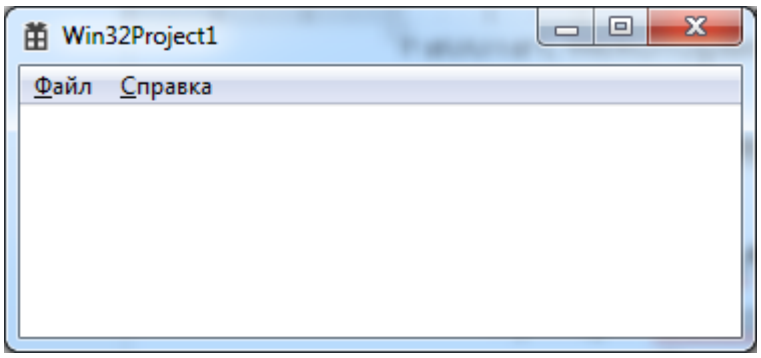
2. Ознакомьтесь с автоматически сгенерированным текстом приложения.

```
// Win32Project1.cpp: определяет точку входа для приложения.
//

#include "stdafx.h"
#include "Win32Project1.h"

#define MAX_LOADSTRING 100

// Глобальные переменные:
HINSTANCE hInst;                                // текущий экземпляр
WCHAR szTitle[MAX_LOADSTRING];                  // Текст строки заголовка
WCHAR szWindowClass[MAX_LOADSTRING];            // имя класса главного окна

// Отправить объявления функций, включенных в этот модуль кода:
ATOM                MyRegisterClass(HINSTANCE hInstance);
BOOL                InitInstance(HINSTANCE, int);
LRESULT CALLBACK    WndProc(HWND, UINT, WPARAM, LPARAM);
INT_PTR CALLBACK    About(HWND, UINT, WPARAM, LPARAM);

int APIENTRY wWinMain(_In_ HINSTANCE hInstance,
                      _In_opt_ HINSTANCE hPrevInstance,
                      _In_ LPWSTR    lpCmdLine,
                      _In_ int       nCmdShow)
{
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);

    // TODO: разместите код здесь.

    // Инициализация глобальных строк
    LoadStringW(hInstance, IDS_APP_TITLE, szTitle, MAX_LOADSTRING);
    LoadStringW(hInstance, IDC_WIN32PROJECT1, szWindowClass, MAX_LOADSTRING);
    MyRegisterClass(hInstance);

    // Выполнить инициализацию приложения:
    if (!InitInstance (hInstance, nCmdShow))
    {
        return FALSE;
    }

    HACCEL hAccelTable = LoadAccelerators(hInstance, MAKEINTRESOURCE(IDC_WIN32PROJECT1));

    MSG msg;

    // Цикл основного сообщения:
    while (GetMessage(&msg, nullptr, 0, 0))
    {
        if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }

    return (int) msg.wParam;
}



//
```

```
//  ФУНКЦИЯ: MyRegisterClass()
//
//  НАЗНАЧЕНИЕ: регистрирует класс окна.
//
ATOM MyRegisterClass(HINSTANCE hInstance)
{
    WNDCLASSEXW wcex;

    wcex.cbSize = sizeof(WNDCLASSEX);

    wcex.style          = CS_HREDRAW | CS_VREDRAW;
    wcex.lpfnWndProc    = WndProc;
    wcex.cbClsExtra     = 0;
    wcex.cbWndExtra     = 0;
    wcex.hInstance      = hInstance;
    wcex.hIcon          = LoadIcon(hInstance, MAKEINTRESOURCE(IDI_WIN32PROJECT1));
    wcex.hCursor        = LoadCursor(nullptr, IDC_ARROW);
    wcex.hbrBackground  = (HBRUSH)(COLOR_WINDOW+1);
    wcex.lpszMenuName   = MAKEINTRESOURCEW(IDC_WIN32PROJECT1);
    wcex.lpszClassName  = szWindowClass;
    wcex.hIconSm        = LoadIcon(wcex.hInstance, MAKEINTRESOURCE(IDI_SMALL));

    return RegisterClassExW(&wcex);
}

//
//   ФУНКЦИЯ: InitInstance(HINSTANCE, int)
//
//   НАЗНАЧЕНИЕ: сохраняет обработку экземпляра и создает главное окно.
//
//   КОММЕНТАРИИ:
//
//        В данной функции дескриптор экземпляра сохраняется в глобальной переменной, а
также
//        создается и выводится на экран главное окно программы.
//
BOOL InitInstance(HINSTANCE hInstance, int nCmdShow)
{
   hInst = hInstance; // Сохранить дескриптор экземпляра в глобальной переменной

   HWND hWnd = CreateWindowW(szWindowClass, szTitle, WS_OVERLAPPEDWINDOW,
      CW_USEDEFAULT, 0, CW_USEDEFAULT, 0, nullptr, nullptr, hInstance, nullptr);

   if (!hWnd)
   {
      return FALSE;
   }

   ShowWindow(hWnd, nCmdShow);
   UpdateWindow(hWnd);

   return TRUE;
}

//
//  ФУНКЦИЯ: WndProc(HWND, UINT, WPARAM, LPARAM)
//
//  НАЗНАЧЕНИЕ:  обрабатывает сообщения в главном окне.
//
//  WM_COMMAND — обработать меню приложения
//  WM_PAINT — отрисовать главное окно
//  WM_DESTROY — отправить сообщение о выходе и вернуться
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
```

```
{
    switch (message)
    {
    case WM_COMMAND:
        {
            int wmId = LOWORD(wParam);
            // Разобрать выбор в меню:
            switch (wmId)
            {
            case IDM_ABOUT:
                DialogBox(hInst, MAKEINTRESOURCE(IDD_ABOUTBOX), hWnd, About);
                break;
            case IDM_EXIT:
                DestroyWindow(hWnd);
                break;
            default:
                return DefWindowProc(hWnd, message, wParam, lParam);
            }
        }
        break;
    case WM_PAINT:
        {
            PAINTSTRUCT ps;
            HDC hdc = BeginPaint(hWnd, &ps);
            // TODO: Добавьте сюда любой код прорисовки, использующий HDC...
            EndPaint(hWnd, &ps);
        }
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        return DefWindowProc(hWnd, message, wParam, lParam);
    }
    return 0;
}

// Обработчик сообщений для окна "О программе".
INT_PTR CALLBACK About(HWND hDlg, UINT message, WPARAM wParam, LPARAM lParam)
{
    UNREFERENCED_PARAMETER(lParam);
    switch (message)
    {
    case WM_INITDIALOG:
        return (INT_PTR)TRUE;

    case WM_COMMAND:
        if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
        {
            EndDialog(hDlg, LOWORD(wParam));
            return (INT_PTR)TRUE;
        }
        break;
    }
    return (INT_PTR)FALSE;
}
```

3. Откомпилируйте проект. Получите результат.

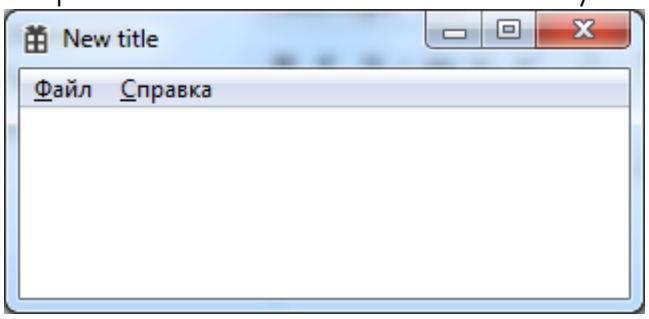


4. Напишите обработчики следующих сообщений:

а) реакция на щелчок правой кнопки мыши - меняется заголовок

```
case WM_LBUTTONUP:
      SetWindowText(hWnd, L"New title");
      break;
```

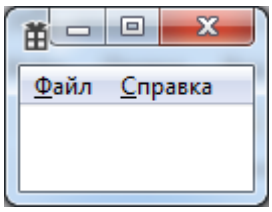
Здесь мы вызываем API-функцию SetWindowText. Она меняет заголовок окна. У этой функции два параметра. Первый указывает на то, для какого окна мы будем это делать. Второй - что за новый заголовок мы установим.



б) реакция на ввод символа - меняется размер и позиция окна

```
case   WM_KEYDOWN:
       {

            int new_x=10;
        int new_y =10;
        int Width =100;
        int Height =100;

        MoveWindow(hWnd,  new_x, new_y, Width, Height,TRUE);
       }
         break;
```

Функция MoveWindow изменяет позицию и размеры окна



5. Можно пользоваться вспомогательными функциями.

- Изменить координаты окна hwnd на экране и его расположение по отношению к другим окнам.

```
BOOL SetWindowPos( HWND hwnd,
                   int y, // новая координата верхнего края
                   int cx, // новая ширина
                   int cy, // новая высота
                   UINT uFlags); // флажок позиционирования
```

- Двойной щелчок мыши WM_LBUTTONDBLCLK. Сообщение относится только к окнам созданным со стилем CS_DBLCLKS. Указывается при регистрации класса окна.

Работа с некоторыми Win API функциями(информация о системе)

Некоторые Win API функции:

1) GetLogicalDrives

Функция GetLogicalDrives возвращает число-битовую маску в которой храняться все

доступные диски.

DWORD GetLogicalDrives(VOID);

Параметры:

Эта функция не имеет параметров.

Возвращаемое значение:

Если функция вызвана правильно, то она возвращает число-битовую маску в которой хранiться все доступные диски (если 0 бит равен 1, то диск "A:" присутствует, и т.д.)

Если функция вызвана не правильно, то она возвращает 0.

Пример:

```
int n;
char dd[4];
DWORD dr = GetLogicalDrives();
for( int i = 0; i < 26; i++ )
{
    n = ((dr>>i)&0x00000001);
```

```
            if( n == 1 )
            {
                  dd[0] =  char(65+i); dd[1] = ':'; dd[2] = '\\'; dd[3] = 0;
                  cout << "Available disk drives : " << dd << endl;
            }
      }
```

2) GetDriveType

Функция GetDriveType возвращает тип диска (removable, fixed, CD-ROM, RAM disk, или network drive).

```
 UINT GetDriveType(LPCTSTR lpRootPathName);
```

Параметры:

lpRootPathName

[in] Указатель на не нулевую стоку в которой хранится имя главной директории на диске. Обратный слэш должен присутствовать!

Если lpRootPathName равно NULL, то функция использует текущую директорию.

Возвращаемое значение:

Функция возвращает тип диска. Могут быть следующие значения:

Значение	Описание
DRIVE_UNKNOWN	Неизвестный тип.
DRIVE_NO_ROOT_DIR	Не правильный путь.
DRIVE_REMOVABLE	Съёмный диск.
DRIVE_FIXED	Фиксированный диск.
DRIVE_REMOTE	Удалённый или network диск.
DRIVE_CDROM	CD-ROM диск.
DRIVE_RAMDISK	RAM диск.

Пример:

```
      int d;

      d = GetDriveType( "c:\\" );
      if( d == DRIVE_UNKNOWN ) cout << " UNKNOWN" << endl;
      if( d == DRIVE_NO_ROOT_DIR ) cout << " DRIVE NO ROOT DIR" << endl;
      if( d == DRIVE_REMOVABLE ) cout << " REMOVABLE" << endl;
      if( d == DRIVE_FIXED ) cout << " FIXED" << endl;
      if( d == DRIVE_REMOTE ) cout << " REMOTE" << endl;
      if( d == DRIVE_CDROM ) cout << " CDROM" << endl;
      if( d == DRIVE_RAMDISK ) cout << " RAMDISK" << endl;
```

3) GetVolumeInformation

Функция GetVolumeInformation возвращает информацию о файловой системе и дисках(директориях).

```
      BOOL GetVolumeInformation(
```

```
LPCTSTR lpRootPathName,          // имя диска(директории)          [in]
LPTSTR lpVolumeNameBuffer,       // название диска                 [out]
DWORD nVolumeNameSize,           // длина буфера названия диска    [in]
LPDWORD lpVolumeSerialNumber,    // сериальный номер диска         [out]
LPDWORD lpMaximumComponentLength, // максимальная длина фыйла
[out]
LPDWORD lpFileSystemFlags,       // опции файловой системы         [out]
LPTSTR lpFileSystemNameBuffer,   // имя файловой системы           [out]
DWORD nFileSystemNameSize        // длина буфера имени файл. сист. [in]
);
```

Возвращаемое значение:

Если функция вызвана правильно, то она возвращает не нулевое значение(TRUE).

Если функция вызвана не правильно, то она возвращает 0(FALSE).

Пример:

```
char VolumeNameBuffer[100];
char FileSystemNameBuffer[100];
unsigned long VolumeSerialNumber;
BOOL GetVolumeInformationFlag = GetVolumeInformationA(
"c:\\",
VolumeNameBuffer,
100,
&VolumeSerialNumber,
NULL, //&MaximumComponentLength,
NULL, //&FileSystemFlags,
FileSystemNameBuffer,
100
);
if(GetVolumeInformationFlag != 0)
{
    cout << "     Volume Name is " << VolumeNameBuffer << endl;
    cout << "     Volume Serial Number is " << VolumeSerialNumber << endl;
    cout << "     File System is " << FileSystemNameBuffer << endl;
}
else cout << " Not Present (GetVolumeInformation)" << endl;
```

4) GetDiskFreeSpaceEx

Функция GetDiskFreeSpaceEx выдаёт информацию о доступном месте на диске.

```
BOOL GetDiskFreeSpaceEx(
 LPCTSTR lpDirectoryName,              // имя диска(директории)          [in]
```

```
    PULARGE_INTEGER lpFreeBytesAvailable,    // доступно для
использования(байт) [out]
    PULARGE_INTEGER lpTotalNumberOfBytes,    // максимальный объём( в
байтах )   [out]
    PULARGE_INTEGER lpTotalNumberOfFreeBytes // свободно на диске( в
байтах )     [out]
    );
```

Возвращаемое значение:

Если функция вызвана правильно, то она возвращает не нулевое значение(TRUE).

Если функция вызвана не правильно, то она возвращает 0(FALSE).

Пример:

```
    DWORD FreeBytesAvailable;
    DWORD TotalNumberOfBytes;
    DWORD TotalNumberOfFreeBytes;

    BOOL GetDiskFreeSpaceFlag = GetDiskFreeSpaceEx(
    "c:\\",                          // directory name
    (PULARGE_INTEGER)&FreeBytesAvailable,     // bytes available to caller
    (PULARGE_INTEGER)&TotalNumberOfBytes,     // bytes on disk
    (PULARGE_INTEGER)&TotalNumberOfFreeBytes  // free bytes on disk
    );

    if(GetDiskFreeSpaceFlag != 0)
    {
        cout << " Total Number Of Free Bytes = " << (unsigned
long)TotalNumberOfFreeBytes
          << "( " <<  double(unsigned long(TotalNumberOfFreeBytes))/1024/1000
          << " Mb )" << endl;
        cout << " Total Number Of Bytes = " << (unsigned long)TotalNumberOfBytes
          << "( " << double(unsigned long(TotalNumberOfBytes))/1024/1000
          << " Mb )" << endl;
    }
    else    cout << "      Not Present (GetDiskFreeSpace)" << endl;
```

5) GlobalMemoryStatus

Функция GlobalMemoryStatus возвращает информацию о используемой системой памяти.

```
    VOID GlobalMemoryStatus(
     LPMEMORYSTATUS lpBuffer   // указатель на структуру MEMORYSTATUS
    );
```

```
typedef struct _MEMORYSTATUS {
DWORD dwLength;                 // длина структуры в байтах
DWORD dwMemoryLoad;             // загрузка памяти в процентах
SIZE_T dwTotalPhys;             // максимальное количество физической
памяти в байтах
SIZE_T dwAvailPhys;             // свободное количество физической памяти в
байтах
SIZE_T dwTotalPageFile;         // макс. кол. памяти для программ в байтах
SIZE_T dwAvailPageFile;         // свободное кол. памяти для программ в
байтах
SIZE_T dwTotalVirtual;          // максимальное количество виртуальной
памяти в байтах
SIZE_T dwAvailVirtual;              // свободное количество виртуальной
памяти в байтах
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

Возвращаемое значение:

Эта функция не возвращает параметров

Пример:

```
//  The MemoryStatus structure is 32 bytes long.
//  It should be 32.
//  78 percent of memory is in use.
//  There are   65076 total Kbytes of physical memory.
//  There are   13756 free Kbytes of physical memory.
//  There are  150960 total Kbytes of paging file.
//  There are   87816 free Kbytes of paging file.
//  There are  1fff80 total Kbytes of virtual memory.
//  There are  1fe770 free Kbytes of virtual memory.

 #define DIV 1024
#define WIDTH 7
char *divisor = "K";

  MEMORYSTATUS stat;
  GlobalMemoryStatus (&stat);
printf ("The MemoryStatus structure is %ld bytes long.\n",
   stat.dwLength);
printf ("It should be %d.\n", sizeof (stat));
printf ("%ld percent of memory is in use.\n",
   stat.dwMemoryLoad);
printf ("There are %*ld total %sbytes of physical memory.\n",
```

```
    WIDTH, stat.dwTotalPhys/DIV, divisor);
  printf ("There are %*ld free %sbytes of physical memory.\n",
    WIDTH, stat.dwAvailPhys/DIV, divisor);
  printf ("There are %*ld total %sbytes of paging file.\n",
    WIDTH, stat.dwTotalPageFile/DIV, divisor);
  printf ("There are %*ld free %sbytes of paging file.\n",
    WIDTH, stat.dwAvailPageFile/DIV, divisor);
  printf ("There are %*lx total %sbytes of virtual memory.\n",
    WIDTH, stat.dwTotalVirtual/DIV, divisor);
  printf ("There are %*lx free %sbytes of virtual memory.\n",
    WIDTH, stat.dwAvailVirtual/DIV, divisor);
```

6) GetComputerName, GetUserNameA

Функция GetComputerName возвращает NetBIOS имя локального компьютера.

```
BOOL GetComputerName(
LPTSTR lpBuffer,
// имя локального компьютера( длина буфера равна MAX_COMPUTERNAME_LENGTH + 1 ) [out]
LPDWORD lpnSize
// размер буфера ( лучше поставить MAX_COMPUTERNAME_LENGTH + 1 ) [out/in]
);
```

Функция GetUserName возвращает имя текущего узера.

```
BOOL GetUserName(
LPTSTR lpBuffer,  // имя юзера( длина буфера равна UNLEN + 1 )   [out]
LPDWORD nSize     // размер буфера ( лучше поставить UNLEN + 1 ) [out/in]
);
```

Возвращаемые значения:

Если функции вызваны правильно, то они возвращают не нулевое значение(TRUE).

Если функции вызваны не правильно, то они возвращают 0(FALSE).

Пример:

```
char ComputerName[MAX_COMPUTERNAME_LENGTH + 1];
unsigned long len_ComputerName = MAX_COMPUTERNAME_LENGTH + 1;
char UserName[UNLEN + 1];
unsigned long len_UserName = UNLEN + 1;

BOOL comp = GetComputerName(
ComputerName,
&len_ComputerName
```

```
);

if( comp != 0 ) { cout << "Computer Name is " << ComputerName << endl; }
else cout << "Computer Name is NOT FOUND !!! " << endl;
comp = GetUserNameA (
UserName,
&len_UserName
);
if( comp != 0 ) { cout << "User Name is " << UserName << endl; }
else cout << "User Name is NOT FOUND !!! " << endl;
```

7) GetSystemDirectory, GetTempPath, GetWindowsDirectory, GetCurrentDirectory

Функция GetSystemDirectory возвращает путь к системной директории.

```
UINT GetSystemDirectory(
 LPTSTR lpBuffer,  // буфер для системной директории  [out]
 UINT uSize       // размер буфера                  [in]
);
```

Возвращаемое значение:

Эта функция возвращает размер буфера для системной директории не включая нулевого
значения в конце, если она вызвана правильно.
Если функция вызвана не правильно, то она возвращает 0.

Функция GetTempPath возвращает путь к директории, отведённой для временных файлов.

```
DWORD GetTempPath(
 DWORD nBufferLength,  // размер буфера                  [in]
 LPTSTR lpBuffer      // буфер для временной директории [out]
);
```

Возвращаемое значение:

Эта функция возвращает размер буфера для системной директории не включая нулевого
значения в конце, если она вызвана правильно.
Если функция вызвана не правильно, то она возвращает 0.

Функция GetWindowsDirectory возвращает путь к Windows директории.

```
UINT GetWindowsDirectory(
 LPTSTR lpBuffer,  // буфер для Windows директории [out]
 UINT uSize       // размер буфера            [in]
);
```

Возвращаемое значение:

Эта функция возвращает размер буфера для системной директории не включая нулевого

значения в конце, если она вызвана правильно.

Если функция вызвана не правильно, то она возвращает 0.

Функция GetCurrentDirectory возвращает путь к текущей директории.

```
DWORD GetCurrentDirectory(
 DWORD nBufferLength,  // размер буфера                  [in]
 LPTSTR lpBuffer       // буфер для текущей директории [out]
);
```

Возвращаемое значение:

Эта функция возвращает размер буфера для системной директории не включая нулевого

значения в конце, если она вызвана правильно.

Если функция вызвана не правильно, то она возвращает 0.

Пример:

```
char path[100];

GetSystemDirectory( path, 100 );
cout << "System Directory is " << path << endl;
GetTempPath( 100, path );
cout << "Temp path is " << path << endl;
GetWindowsDirectory( path, 100 );
cout << "Windows directory is " << path << endl;
GetCurrentDirectory( 100, path );
cout << "Current directory is " << path << endl;
```

Контекст устройства

Для того, чтобы что-то нарисовать (вывести) необходимо получить контекст устройства (DC). Во всех случаях, кроме обработки сообщения WM_PAINT, для этой цели используется функция GetDC().

```
HDC GetDC(HWND hWnd);
```

где *hWnd* - handle окна.

Для получения контекста устройства для обработчика сообщения WM_PAINT используется функция BeginPaint()

```
HDC BeginPaint(
  HWND hwnd,          // дескриптор окна
  LPPAINTSTRUCT lpPaint // информация
);
```

За исключением самого первого сообщения WM_PAINT, посылаемого окну при вызове UpdateWindow(), эти сообщения будут посылаться в следующих случаях:

- при изменении размеров окнаж
- если рабочая область была скрыта меню или окном диалога, которое в данный момент закрывается;
- при использовании функции ScrollWindow();
- при принудительной генерации сообщения WM_PAINT вызовом функций InvalidateRect() или InvalidateRgn().

Вывод текста

Для вывода текста в области окна можно использовать функции TextOut() и DrawText().

```
BOOL TextOut(
  HDC hdc,          // дескриптор  DC
  int nXStart,      // x-координата начальной точки
  int nYStart,      // y- координата начальной точки
  LPCTSTR lpString,  // строка символов
  int cbString      // число символов
);

int DrawText(
  HDC hDC,          // handle to DC
  LPCTSTR lpString, // text to draw
  int nCount,       // text length
  LPRECT lpRect,    // formatting dimensions
  UINT uFormat      // text-drawing options
);
```

Для того, чтобы установить шрифт надо:

1)Создать структуру типа LOGFONT.

```
typedef struct tagLOGFONT {
  LONG lfHeight;
  LONG lfWidth;
  LONG lfEscapement;
  LONG lfOrientation;
  LONG lfWeight;
  BYTE lfItalic;
  BYTE lfUnderline;
  BYTE lfStrikeOut;
  BYTE lfCharSet;
  BYTE lfOutPrecision;
  BYTE lfClipPrecision;
  BYTE lfQuality;
  BYTE lfPitchAndFamily;
  TCHAR lfFaceName[LF_FACESIZE];
} LOGFONT;
```

Например,

```
LOGFONT  logfont;
logfont.lfHeight =50;
logfont.lfWidth =10;
//….
strcpy(logfont.lfFaceName, "Arial");
//…
```

2) Создать шрифт вызовом функции **CreateFontIndirect():**

HFONT CreateFontIndirect(CONST LOGFONT ****lplf*);**

Например,

```
HFONT newfont=CreateFontIndirect(&logfont);
```

3)Установить шрифт в контексте устройства вызовом функции **SelectObject():**

```
HGDIOBJ SelectObject(
  HDC hdc,        // handle to DC
  HGDIOBJ hgdiobj   // handle to object
);
```

Например,

SelectObject(hdc,(HGDIOBJ) newfont);

Если нужно установить цвет букв, отличный от по умолчанию, то вызываем функцию SetTextColor():

```
COLORREF SetTextColor(
  HDC hdc,          // handle to DC
  COLORREF crColor   // text color
);
```

Например,

SetTextColor(hdc, RGB(255, 0,0));

Рисование линий и фигур

Для рисования линий и геометрических фигур используются различные функции API.

Например, эллипс:

```
BOOL Ellipse(
  HDC hdc,        // handle to DC
  int nLeftRect,// x-coord of upper-left corner of rectangle
  int nTopRect,  // y-coord of upper-left corner of rectangle
  int nRightRect,//x-coord of lower-right corner ofrectangle
  int nBottomRect//y-coord of lower-right corner of rectangle
);
```

Прямоугольник:

```
BOOL Rectangle(
  HDC hdc,       // handle to DC
  int nLeftRect, // x-coord of upper-left corner of rectangle
  int nTopRect,  // y-coord of upper-left corner of rectangle
  int nRightRect,// x-coord of lower-right corner ofrectangle
  int nBottomRect// y-coord of lower-right corner ofrectangle
);
```

Цвет заливки замкнутой фигуры устанавливается следующим образом:

1)Создается кисть нужного цвета:

```
long color=RGB(125,125,225);
HBRUSH hNew;
hNew=CreateSolidBrush(color);
```

2)Устанавливается объект-кисть в контексте устройства:

```
SelectObject(hdc,hNew);
```