Модуль №3

Занятие №4

Версия 1.0.1

План занятия:

1. Повторение пройденного материала.
2. Распаковщики сообщений.
3. Общие элементы управления.
4. Элемент управления «индикатор процесса» (Progress Control).
5. Сообщения индикатора процесса.
6. Практическая часть.
7. Подведение итогов.
8. Домашнее задание.

1. Повторение пройденного материала

Данное занятие необходимо начать с краткого повторения материала предыдущего занятия. При общении со слушателями можно использовать следующие контрольные вопросы:

1) Чем отличается обычный список от комбинированного списка?
2) Какими способами можно создать список на диалоге?
3) Какое сообщение придёт в диалоговую процедуру при выборе элемента списка?
4) Какое уведомление от списка приходит родительскому окну (диалогу) при изменении текущего выбора?
5) Какие сообщения необходимо отправить списку для добавления, вставки и удаления строки?
6) Какое сообщение необходимо отправить списку для поиска в нём указанной строки?

7) Какое сообщение необходимо отправить списку, чтобы определить индекс выбранного элемента списка?
8) Как программно выбрать элемент списка?
9) Какое сообщение необходимо отправить списку, чтобы определить количество элементов в нем?
10) Какое сообщение необходимо отправить списку, чтобы его очистить?
11) Какое сообщение необходимо отправить списку, чтобы получить текст указанного элемента?
12) Какая функция возвращает битовую маску логических дисков, которые доступны в данный момент?
13) Какая функция возвращает тип накопителя по заданному имени корневого пути?

2. Распаковщики сообщений

Повторив материал предыдущего занятия, ознакомить слушателей с макросами – распаковщиками сообщений (**Message crackers**). Отметить, что распаковщики сообщений упрощают написание оконной процедуры, в теле которой обычно один огромный оператор **switch**, содержащий большое число строк кода, что является образцом плохого стиля программирования. Распаковщики сообщений позволяют разбить оператор **switch** на небольшие функции – по одной на оконное сообщение. Это значительно улучшает внутреннюю структуру кода.

Как известно, с каждым сообщением в оконную процедуру приходит дополнительная информация о сообщении. Эта информация упакована в параметрах **WPARAM** и **LPARAM** и специфична для каждого сообщения. При написании приложений необходимо помнить эту дополнительную информацию или искать её в справочнике. Однако использование макросов упрощает разработку приложений, так как макросы распаковывают параметры сообщений.

Например, для обработки сообщения **WM_CLOSE** необходимо в операторе **switch** оконной процедуры указать макрос **HANDLE_MSG**:

```
BOOL CALLBACK DlgProc(HWND hwnd, UINT message, WPARAM wParam, LPARAM lParam)
{
	switch(message)
	{
		HANDLE_MSG(hwnd, WM_CLOSE, Cls_OnClose);
	}
	return FALSE;
}
```

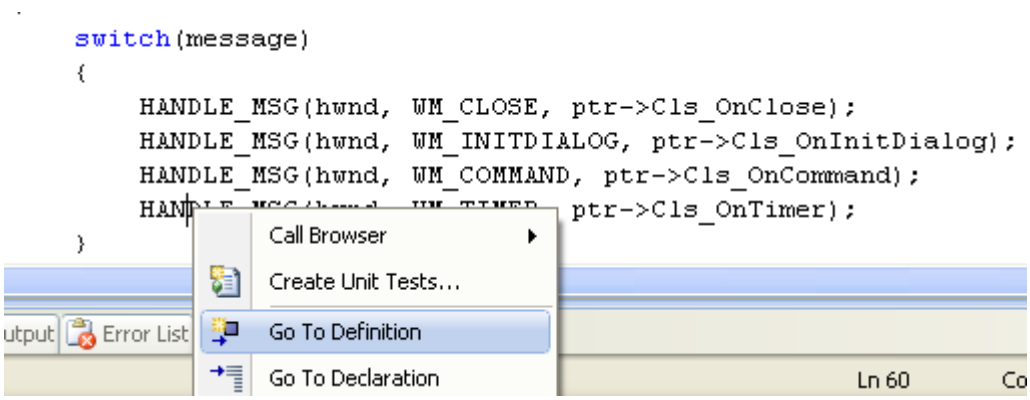
В коде приложения предусмотреть функцию–обработчик сообщения **WM_CLOSE**:

```
void Cls_OnClose(HWND hwnd)
{
	EndDialog(hwnd, 0);
}
```

В файле **WindowsX.h** макрос **HANDLE_MSG** определён так:

```
#define HANDLE_MSG(hwnd, message, fn)    \
    case (message): return HANDLE_##message((hwnd),
        (wParam), (lParam), (fn));
```

Стоит отметить, что для быстрого перехода к месту определения макроса **HANDLE_MSG** в файле **WindowsX.h** необходимо в коде приложения щелкнуть правой кнопкой мыши по макросу, и в появившемся контекстном меню выбрать **Go To Definition**.



Для сообщения **WM_CLOSE** эта строка после обработки препроцессором выглядит как:

```
    case (WM_CLOSE):
    return HANDLE_WM_CLOSE((hwnd), (wParam), (lParam), (Cls_OnClose));
```

Макросы **HANDLE_##message** (например, **HANDLE_WM_CLOSE**, **HANDLE_WM_COMMAND** и т.д.) представляют собой распаковщики сообщений. Они распаковывают содержимое параметров **wParam** и **lParam**, выполняют нужные преобразования типов и вызывают соответствующую функцию – обработчик сообщения (например, **Cls_OnClose**). Например, макрос **HANDLE_WM_CLOSE** в файле **WindowsX.h** определён следующим образом:

```
#define HANDLE_WM_CLOSE(hwnd, wParam, lParam, fn) ((fn)(hwnd), 0L)
```

Результат раскрытия препроцессором этого макроса – вызов функции **Cls_OnClose**, которой передаются распакованные части параметров **wParam** и **lParam**. При этом производятся соответствующие преобразования типов.

Чтобы использовать распаковщик для обработки сообщения, например, **WM_COMMAND**, следует найти в файле **WindowsX.h** следующий фрагмент кода:

```
/* void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) */
#define HANDLE_WM_COMMAND(hwnd, wParam, lParam, fn) \
            ((fn)((hwnd), (int)(LOWORD(wParam)), (HWND)(lParam),
            (UINT)HIWORD(wParam)), 0L)
#define FORWARD_WM_COMMAND(hwnd, id, hwndCtl, codeNotify, fn) \
          (void)(fn)((hwnd), WM_COMMAND,
          MAKEWPARAM((UINT)(id),(UINT)(codeNotify)), (LPARAM)(HWND)(hwndCtl))
```

Первая строка в этом коде - прототип функции – обработчика сообщения **WM_COMMAND**. Следующая строка – распаковщик сообщения. Последняя строка в этом фрагменте кода содержит предописатель сообщения (**message forwarder**), который используется в том случае, когда при обработке сообщения необходимо вызвать стандартный обработчик по умолчанию.

```
void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify)
{
	// выполняем обработку сообщения


	// обработка по умолчанию
	FORWARD_WM_COMMAND(hwnd, id, hwndCtl, codeNotify, DefWindowProc);
}
```

В качестве примера использования распаковщиков сообщений, привести модифицированный код приложения, рассмотренного на предыдущих занятиях.

```
// header.h

#pragma once

#include <windows.h>
#include <WindowsX.h>
#include <tchar.h>
#include "resource.h"

// CMessageCrackersDlg.h

#pragma once
#include "header.h"

class CMessageCrackersDlg
{
public:
	CMessageCrackersDlg(void);
public:
	static BOOL CALLBACK DlgProc(HWND hWnd, UINT mes, WPARAM wp, LPARAM lp);
	static CMessageCrackersDlg *ptr;
	BOOL Cls_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam);
	void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);
	void Cls_OnClose(HWND hwnd);
	void Cls_OnTimer(HWND hwnd, UINT id);
	HWND hDialog;
	HWND hStart, hStop, hPicture;
	HBITMAP hBmp[5];
};


// CMessageCrackersDlg.cpp

#include "CMessageCrackersDlg.h"

CMessageCrackersDlg* CMessageCrackersDlg::ptr = NULL;

CMessageCrackersDlg::CMessageCrackersDlg(void)
{
	ptr = this;
}
```

```
void CMessageCrackersDlg::Cls_OnClose(HWND hwnd)
{
	EndDialog(hwnd, 0);
}

BOOL CMessageCrackersDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus,
                                           LPARAM lParam)
{
	hDialog = hwnd;
	hStart = GetDlgItem(hDialog, IDC_START);
	hStop = GetDlgItem(hDialog, IDC_STOP);
	hPicture = GetDlgItem(hDialog, IDC_PICTURE);
	for(int i = 0; i < 5; i++)
		hBmp[i] = LoadBitmap(GetModuleHandle(NULL),
				MAKEINTRESOURCE(IDB_BITMAP1 + i));
	return TRUE;
}

void CMessageCrackersDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
                                        UINT codeNotify)
{
	if(id == IDC START)
	{
		SetTimer(hDialog, 1, 1000, NULL);
		EnableWindow(hStart, FALSE);
		EnableWindow(hStop, TRUE);
		SetFocus(hStop);
	}
	else if(id == IDC_STOP)
	{
		KillTimer(hDialog, 1);
		EnableWindow(hStart, TRUE);
		EnableWindow(hStop, FALSE);
		SetFocus(hStart);
	}
}

void CMessageCrackersDlg::Cls_OnTimer(HWND hwnd, UINT id)
{
	static int index = 0;
	index++;
	if(index > 4)
		index = 0;
	SendMessage(hPicture, STM_SETIMAGE, WPARAM(IMAGE_BITMAP),
			LPARAM(hBmp[index]));
}

BOOL CALLBACK CMessageCrackersDlg::DlgProc(HWND hwnd, UINT message,
                                   WPARAM wParam, LPARAM lParam)
{
	switch(message)
	{
		HANDLE_MSG(hwnd, WM_CLOSE, ptr->Cls_OnClose);
		HANDLE_MSG(hwnd, WM_INITDIALOG, ptr->Cls_OnInitDialog);
		HANDLE_MSG(hwnd, WM_COMMAND, ptr->Cls_OnCommand);
		HANDLE_MSG(hwnd, WM_TIMER, ptr->Cls_OnTimer);
	}
	return FALSE;
}
```

```
                                // CMessageCrackers.cpp

#include "CMessageCrackersDlg.h"


int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrev, LPTSTR lpszCmdLine,
                     int nCmdShow)
{
      CMessageCrackersDlg dlg;
      return DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
                          CMessageCrackersDlg::DlgProc);
}
```

В завершении рассмотрения данного вопроса ещё раз акцентировать внимание слушателей на следующих преимуществах распаковщиков сообщений:

- сокращение числа явных преобразований типов в коде приложения, а также возникающих при этом ошибок;
- читабельность кода;
- простота и удобство в использовании при разработке приложения.

3. Общие элементы управления

Напомнить слушателям, что помимо базовых элементов управления (Button, Edit Control, Static и т.д.), которые поддерживались самыми ранними версиями Windows, в системе используется библиотека элементов управления общего пользования (**common control library**). Общие элементы управления, включенные в эту библиотеку, дополняют базовые элементы управления и позволяют придать приложениям более совершенный вид. К общим элементам управления относятся панель инструментов (**Toolbar**), окно подсказки (**Tooltip**), индикатор (**Progress Bar**), счётчик (**Spin Control**), строка состояния (**Status Bar**) и другие. Библиотека элементов управления общего пользования реализована в виде динамически загружаемой библиотеки **comctl32.dll**.

Большинство элементов управления общего пользования реализовано в виде окна соответствующего предопределенного класса, и, следовательно, элементы управления могут быть созданы вызовом функции **CreateWindowEx**.

Разница между базовыми элементами управления и общими элементами управления состоит в типе посылаемых уведомительных сообщений. Базовые элементы управления посылают сообщения **WM_COMMAND**, а общие элементы управления почти всегда посылают сообщения **WM_NOTIFY**.

Чтобы использовать в приложении какой-либо элемент управления общего пользования, сначала нужно вызвать функцию API **InitCommonControlsEx**, которая регистрирует оконные классы элементов управления.

```
BOOL InitCommonControlsEx(
    LPINITCOMMONCONTROLSEX lpInitCtrls
);
```

В единственном параметре **lpInitCtrls** передается адрес структурной переменной типа **INITCOMMONCONTROLSEX**, содержащей информацию о том, какие классы элементов управления должны быть зарегистрированы.

Структура **INITCOMMONCONTROLSEX** имеет следующее определение:

```
typedef struct tagINITCOMMONCONTROLSEX {
DWORD dwSize; // размер структуры в байтах
DWORD dwICC; // флаги загрузки классов из DLL
} INITCOMMONCONTROLSEX, *LPINITCOMMONCONTROLSEX;
```

Второй параметр может принимать одно или несколько значений, перечисленных в таблице.

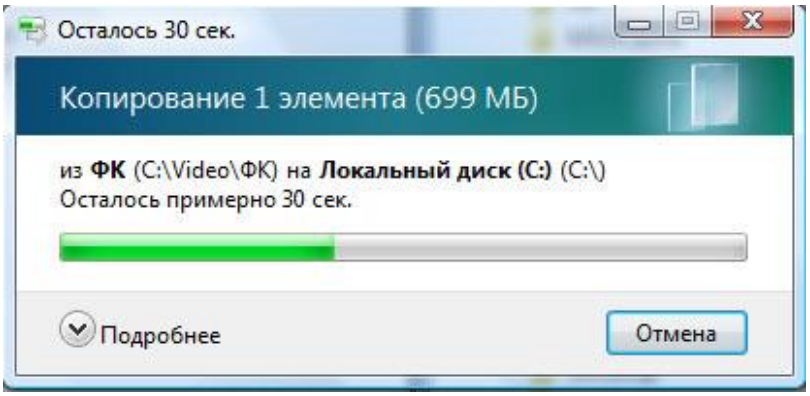
Флаг	**Оконные классы для элементов управления, которые будут загружены**
ICC_ANIMATE_CLASS	animate
ICC_BAR_CLASSES	toolbar, status bar, slider, tooltip
ICC_LISTVIEW_CLASSES	list view, header

Флаг	Оконные классы для элементов управления, которые будут загружены
ICC_PROGRESS_CLASS	progress bar
ICC_TAB_CLASSES	tab, tooltip
ICC_TREEVIEW_CLASSES	tree view, tooltip
ICC_UPDOWN_CLASS	up-down
ICC_WIN95_CLASSES	animate, header, hot key, list view, progress bar, status bar, tab, tooltip, toolbar, slider, tree view, up-down

Необходимо отметить, что библиотека элементов управления общего пользования реализована в виде динамически загружаемой библиотеки **comctl32.dll**. Для того чтобы использовать в приложении функцию **InitCommonControlsEx** необходимо подключить заголовочный файл **commctrl.h**, в котором она описана. Помимо этого следует указать компоновщику расположение библиотечного файла **comctl32.lib**, содержащего ссылку на **DLL** и перечень находящихся в ней функций.

```
#pragma comment(lib,"comctl32")
```

4. Элемент управления «индикатор процесса» (Progress Control)

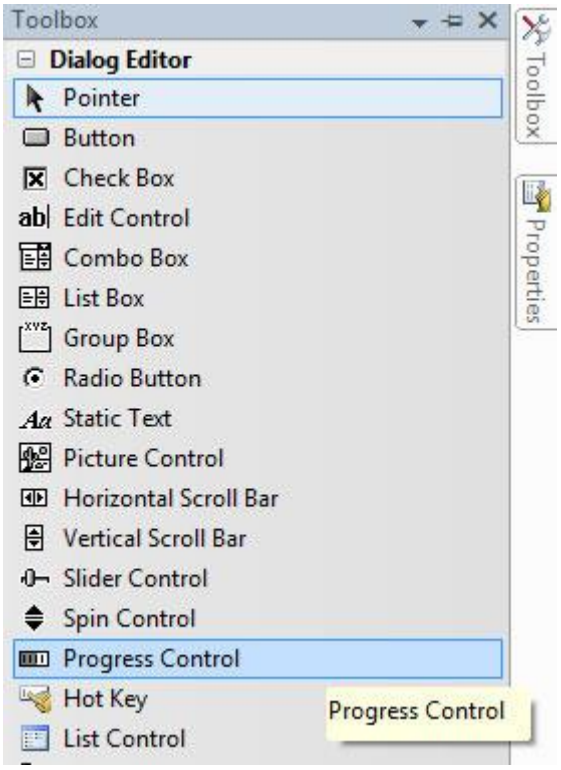


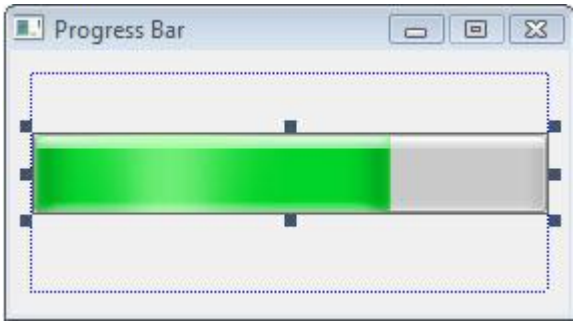
Элемент управления **Progress Control** обычно используется в приложениях для отображения процесса выполнения некоторой длительной операции.

Создать индикатор процесса на форме диалога можно двумя способами:

- с помощью средств интегрированной среды разработки **Microsoft Visual Studio**;
- посредством вызова функции **CreateWindowEx**.

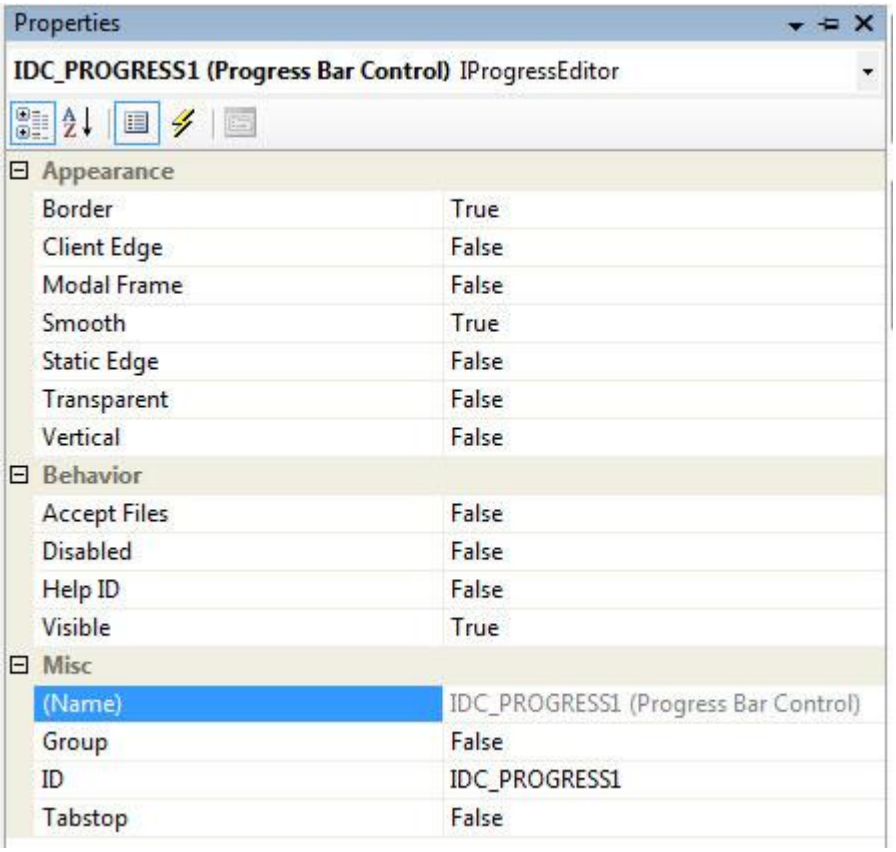
При первом способе необходимо определить **Progress Control** в шаблоне диалогового окна на языке описания шаблона диалога. Для этого следует активизировать окно **Toolbox** и «перетащить» индикатор на форму диалога.





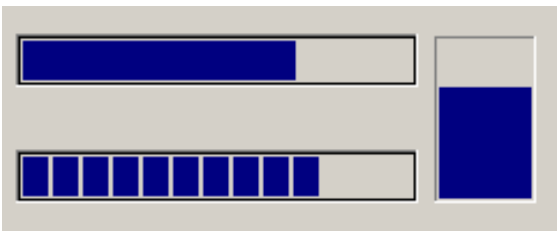
После размещения индикатора на форме диалога ему назначается идентификатор (например, **IDC_PROGRESS1**), который впоследствии можно изменить на идентификатор, отражающий семантику ресурса.

Необходимо ознакомить слушателей с некоторыми свойствами элемента управления **Progress Control**.



По умолчанию **Progress Control** заполняется по горизонтали, т.е. слева направо. Для вертикального заполнения индикатора (снизу вверх) необходимо свойству **Vertical** указать значение **True**.

При ложном значении свойства **Smooth** (по умолчанию) индикатор процесса заполняется отдельными маленькими прямоугольниками (нижний индикатор, представленный на изображении).



Если же необходимо использовать сплошное заполнение окна индикатора процесса, то свойству **Smooth** нужно указать значение **True** (на изображении - верхний индикатор и индикатор справа).

5. Сообщения индикатора процесса

Для управления индикатором процесса используются сообщения, приведенные в таблице.

Код сообщения	wParam	lParam	Описание
PBM_SETRANGE	0	MAKELPARAM (wMin, wMax)	Установка интервала для индикатора
PBM_SETPOS	nNewPos	0	Установка текущей позиции индикатора
PBM_DELTAPOS	nInc	0	Изменение текущей позиции прибавлением смещения nInc
PBM_SETSTEP	nStepInc	0	Установка шага приращения для индикатора
PBM STEPIT	0	0	Изменение текущей позиции прибавлением шага nStepInc
PBM_SETBARCOLOR	0	(COLORREF) clrBar	Установка цвета заполняемых прямоугольников
PBM_SETBKCOLOR	0	(COLORREF) clrBk	Установка цвета фона индикатора

Как видно из таблицы, для установки интервала индикатора следует отправить ему сообщение **PBM_SETRANGE**, указав в параметре **lParam** границы интервала. Для того чтобы упаковать дополнительную информацию в параметры **wParam** и **lParam** удобно использовать следующие макросы:

```
WPARAM MAKEWPARAM( WORD wLow, WORD wHigh );
LPARAM MAKELPARAM( WORD wLow, WORD wHigh );
```

Следует отметить, что текущее состояние индикатора можно изменять тремя альтернативными способами. Для этого могут использоваться сообщения **PBM_SETPOS**, **PBM_DELTAPOS** или **PBM_STEPIT**. В третьем способе текущая позиция изменяется прибавлением шага приращения **nStepInc**, который можно

предварительно установить, отправив сообщение **PBM_SETSTEP** (по умолчанию **nStepInc** = 10).

Если при обработке сообщения **PBM_STEPIT** индикатор достигнет значения **wMax** или превысит его, то его текущая позиция сбрасывается в значение **wMin** и индикатор процесса стартует сначала.

Два последних сообщения из вышеприведенной таблицы позволяют изменить цветовые атрибуты элемента **Progress Control**.

Для демонстрации применения индикатора процесса привести следующий код:

```
                                    // header.h

#pragma once
#include <windows.h>
#include <windowsX.h>
#include <ctime>
#include <tchar.h>
#include <commctrl.h>
#include "resource.h"

#pragma comment(lib, "comctl32")


                                // ProgressControlDlg.h

#pragma once
#include "header.h"

class ProgressControlDlg
{
public:
      ProgressControlDlg(void);
public:
      static BOOL CALLBACK DlgProc(HWND hWnd, UINT mes, WPARAM wp, LPARAM lp);
      static ProgressControlDlg* ptr;
      BOOL Cls_OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam);
      void Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify);
      void Cls_OnTimer(HWND hwnd, UINT id);
      void Cls_OnClose(HWND hwnd);
      HWND hDialog, hProgress1, hProgress2;
};


                               // ProgressControlDlg.cpp

#include "ProgressControlDlg.h"

ProgressControlDlg* ProgressControlDlg::ptr = NULL;

ProgressControlDlg::ProgressControlDlg(void)
{
```

```
	ptr = this;
}



void ProgressControlDlg::Cls_OnClose(HWND hwnd)
{
	EndDialog(hwnd, 0);
}

BOOL ProgressControlDlg::Cls_OnInitDialog(HWND hwnd, HWND hwndFocus,
                                          LPARAM lParam)
{
	srand(time(0));
	hDialog = hwnd;
	hProgress1 = GetDlgItem(hDialog, IDC_PROGRESS1);

	// Установка интервала для индикатора
	SendMessage(hProgress1, PBM_SETRANGE, 0, MAKELPARAM(0, 60));

	// Установка шага приращения  индикатора
	SendMessage(hProgress1, PBM_SETSTEP, 1, 0);

	// Установка текущей позиции индикатора
	SendMessage(hProgress1, PBM_SETPOS, 0, 0);

	// Установка цвета фона индикатора
	SendMessage(hProgress1, PBM_SETBKCOLOR, 0, LPARAM(RGB(0, 0, 255)));

	// Установка цвета заполняемых прямоугольников
	SendMessage(hProgress1, PBM_SETBARCOLOR, 0, LPARAM(RGB(255, 255, 0)));

	hProgress2 = GetDlgItem(hDialog, IDC_PROGRESS2);
	SendMessage(hProgress2, PBM_SETRANGE, 0, MAKELPARAM(0, 60));
	SendMessage(hProgress2, PBM_SETSTEP, 1, 0);
	SendMessage(hProgress2, PBM_SETPOS, 0, 0);
	SendMessage(hProgress2, PBM_SETBKCOLOR, 0, LPARAM(RGB(0, 255, 0)));
	SendMessage(hProgress2, PBM_SETBARCOLOR, 0, LPARAM(RGB(255, 0, 255)));
	return TRUE;
}

void ProgressControlDlg::Cls_OnCommand(HWND hwnd, int id, HWND hwndCtl,
                                       UINT codeNotify)
{
	if(id == IDC_BUTTON1)
		SetTimer(hwnd, 1, 100, NULL);
}

void ProgressControlDlg::Cls_OnTimer(HWND hwnd, UINT id)
{
	// Изменение текущей позиции индикатора путём прибавления шага
	SendMessage(hProgress1, PBM_STEPIT, 0, 0);
	int n = rand()%60;

	// Установка текущей позиции индикатора
	SendMessage(hProgress2, PBM_SETPOS, WPARAM(n), 0);
}

BOOL CALLBACK ProgressControlDlg::DlgProc(HWND hwnd, UINT message,
                                          WPARAM wParam, LPARAM lParam)
```

```
{
	switch(message)
	{


		HANDLE_MSG(hwnd, WM_CLOSE, ptr->Cls_OnClose);
		HANDLE_MSG(hwnd, WM_INITDIALOG, ptr->Cls_OnInitDialog);
		HANDLE_MSG(hwnd, WM_COMMAND, ptr->Cls_OnCommand);
		HANDLE_MSG(hwnd, WM_TIMER, ptr->Cls_OnTimer);
	}
	return FALSE;
}
```

```
// ProgressControl.cpp

#include "ProgressControlDlg.h"


int WINAPI _tWinMain(HINSTANCE hInst, HINSTANCE hPrev, LPTSTR lpszCmdLine,
		int nCmdShow)
{
	INITCOMMONCONTROLSEX icc = {sizeof(INITCOMMONCONTROLSEX)};
	icc.dwICC = ICC_WIN95_CLASSES;
	InitCommonControlsEx(&icc);
	ProgressControlDlg dlg;
	return DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), NULL,
			ProgressControlDlg::DlgProc);
}
```

Альтернативный способ создания индикатора - использование функции **CreateWindowEx**. В этом случае во втором параметре функции передается имя предопределенного оконного класса – **PROGRESS_CLASS**.

В качестве примера, демонстрирующего программный способ создания индикатора, привести следующий фрагмент кода:

```
HWND hProgress1 = CreateWindowEx(0, PROGRESS_CLASS, NULL,
				WS_CHILD | WS_VISIBLE | PBS_SMOOTH,
				LEFT, TOP, WIDTH, HEIGHT,
				hwndParent, NULL, GetModuleHandle(NULL), NULL);
```

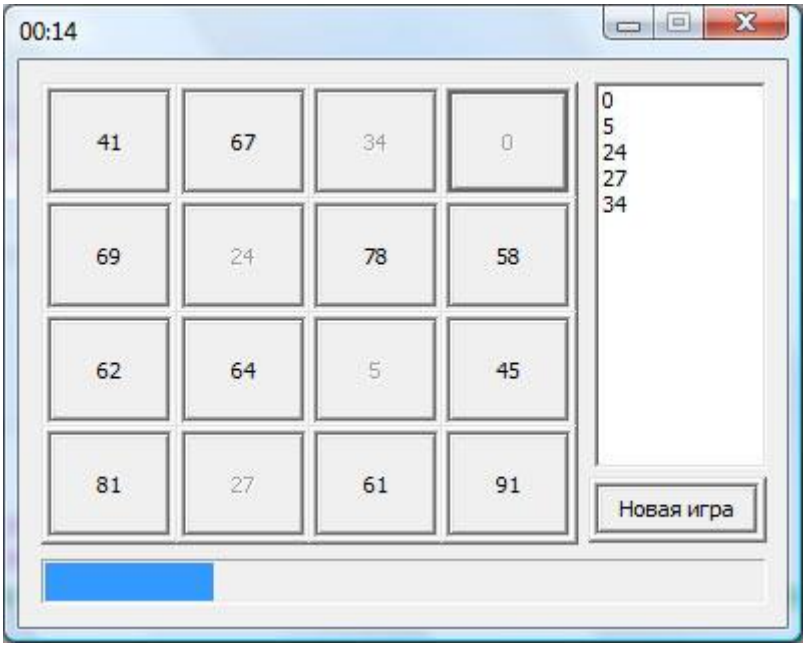
6. Практическая часть

Дополнить игру «пятнашки» индикатором процесса, который будет отображать процесс собирания «пятнашек», т. е. отображать, какое количество кнопок в процентном отношении находится на своих местах.

7. Подведение итогов

Подвести общие итоги занятия. Отметить удобство использования распаковщиков сообщений, перечислив их основные преимущества. Подчеркнуть основные способы создания индикаторов – с помощью средств IDE, а также посредством функции CreateWindowEx. Перечислить сообщения, используемые для управления индикатором. Акцентировать внимание слушателей на наиболее тонких моментах изложенной темы.

8. Домашнее задание



Написать игру, смысл которой состоит в следующем. На игровом поле имеются 16 кнопок, список и индикатор. При запуске игры (кнопка «Новая игра») на кнопки помещаются 16 случайных чисел из диапазона от 0 до 100.

Задача состоит в том, чтобы за указанное время успеть (пока не заполнится весь индикатор) щелкнуть по всем кнопкам в порядке возрастания чисел. При нажатии на кнопку число должно добавляться в список только в том случае, если это число является следующим по возрастанию.