PART 04

실전프로젝트

CHAPTER 01 FormPad 프로젝트

PREVIEW

PART 04에서는 규모가 좀 더 큰 응용 프로그램을 개발하는 과정을 통해 본문에서 배운 다양한 기법을 총정리 해본다. 본문에서 미처 다루지 못한 의미 있는 내용도 실습을 통해 새로 배우고, 실전 응용력을 키울 수 있도록 구성했다.

CHAPTER 01

FormPad 프로젝트

학습 목표

MFC 프로그래밍 기술을 종합적으로 활용하는 응용 프로그램을,
초기 설계부터 개발까지 단계별로 구현함으로써
실전 개발 능력을 기른다.

01 프로젝트 개요

프로젝트 개발 방법론은 두 가지다. 설계를 완벽히 마친 다음 개발하는 방식(설계 후 개발 방식)과 해야 할 것을 목차로 정리한 후 설계와 개발을 병행하는 방식(개발 중 설계 방식)이다.

- **설계 후 개발 방식** : 수행 모델 설계 → 요소 배치 → 구체적 클래스 설계 → 개발 → 설계와 비교 → 완료
- **개발 중 설계 방식** : 목차 작성 → 개발 진행 → 설계 변경 → 수정 및 개발 → 완료

설계 후 개발 방식은 MS 같은 소프트웨어 전문 회사에서 사용하며, 이를 위해 UML Unified Markup Language 같은 설계 모델을 다루는 전문 교육 과정이 존재한다. 그러나 대부분의 개발은 개발 중 설계 방식을 사용한다. 이 방식은 프로그래머의 감각에 의존하기 때문에 개발 속도가 빠른 대신 복잡한 프로그램을 개발하기 어렵다. 대형 프로젝트의 경우, 설계 후 개발 방식으로 프로토타입을 만들고, 개발 중 설계 방식으로 재설계하는 복합 방식을 사용한다. 따라서 개발 중 설계 방식은 프로젝트 개발을 위한 기본 역량이라 할 수 있다.

이 장에서는 개발 중 설계 방식으로 프로젝트를 진행한다. 개발 중 설계 방식의 첫 번째 단계 산출물인 목차는 [표 16-1]에 정리해 두었다(다른 프로그램을 개발할 때도 이 목차를 참고하여 작성하면 된다). 목차를 작성할 때는 '먼저 구현할 것'과 '쉽게 구현할 수 있는 것'을 위주로 정렬한다. 그리고 개발 중간에 재설계가 필요한 부분이 있으면 바로 반영한다. 이 장의 프로젝트를 진행하면서 개발 중 설계 방식을 적용하는 방법도 함께 익히자.

1 FormPad 프로젝트 화면 소개

[그림 16-1]과 같이 대화상자 리소스 템플릿을 수정하는 화면을 기억할 것이다. 컨트롤 툴바에서 필요한 컨트롤을 배치하고 속성을 변경하여 대화상자 템플릿을 쉽게 구성할 수 있다.

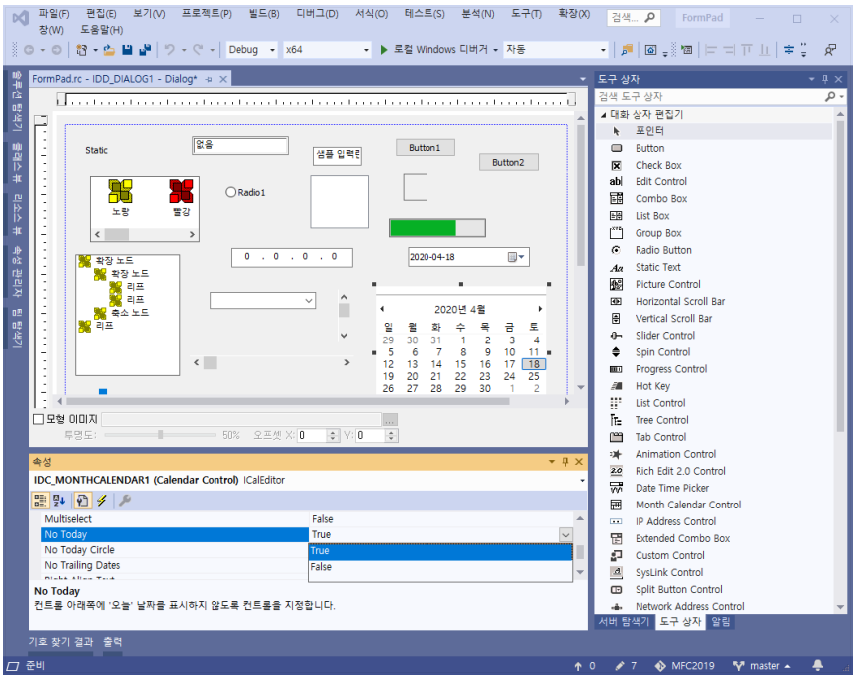
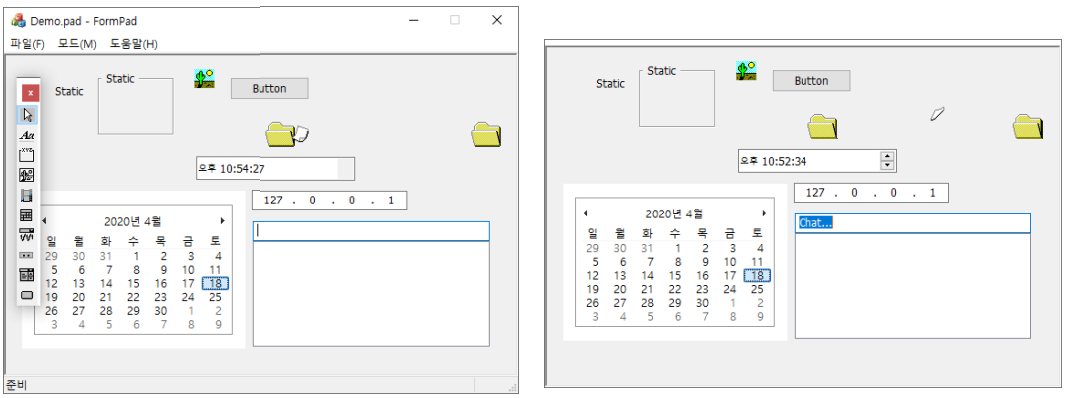


그림 16-1 비주얼 C++와 리소스 템플릿 편집

컨트롤을 배치하는 환경을 통틀어 폼이라고 한다. FormPad 프로젝트는 이런 폼 방식의 편집과 더불어 컴파일 없이 바로 실행할 수 있는 환경을 구축하는 내용이다. 따라서 FormPad 프로그램은 다음과 같이 편집 모드와 동작 모드라는 두 상태를 가지며, 다용도 도구 상자로 활용할 수 있다.



(a) 편집 모드 (b) 동작 모드

그림 16-2 FormPad 응용 프로그램 윈도우 모드

그림 16-3 FormPad 컨트롤 툴바

편집 모드에서는 비주얼 C++처럼 컨트롤 툴바를 제공하고, 원하는 컨트롤을 드래그하여 배치할 수 있다. 그리고 메뉴를 통해 필요한 작업을 지원한다. 반면 동작 모드는 툴바나 메뉴가 없고, 심지어 타이틀 바도 보이지 않는다. 단지 대화상자 내 컨트롤 동작에만 집중한다. FormPad 프로젝트에서 지원할 컨트롤은 왼쪽과 같다. 물론, 필요에 따라 컨트롤을 더 추가하거나 변경할 수 있다.

2 FormPad 프로젝트 개발 목차

[표 16-1]은 FormPad 프로젝트의 각 단계에서 익힐 수 있는 기술을 정리한 것으로, 개발 중 설계 방식의 첫 번째 단계에서 작성한 개발 목차이다. 많은 내용이 앞서 배운 내용에 기반하지만 소개되지 않은 응용 기술도 일부 포함된다.

표 16-1 FormPad 프로젝트 적용 기술

2. 응용 프로그램 제작	
(1) 폼뷰 SDI 응용 프로그램	
FormPad 프로젝트 생성	폼뷰 프로젝트 생성
최대화 버튼 비활성화	윈도우 생성 시 스타일 변경(CWnd::PreCreateWindow)
스크롤 바 없애기	폼뷰 스크롤바 숨기기
(2) 편집 모드와 실행 모드 전환 구성	
메뉴 편집과 단축키	메뉴 추가, 단축키 설정, 명령/갱신 핸들러 구현
편집 모드와 실행 모드	시스템 정보 알아내기(GetSystemMetrics( )), 타이틀바/메뉴/툴바 숨기기, 윈도우 크기 변경
실행 모드에서 이동 처리	마우스 클릭으로 윈도우 이동, 메시지 전송 함수
3. 컨트롤 툴바 제작	
(1) 리소스 준비	EXE나 DLL에서 리소스 추출, 툴바 비트맵 변경
(2) 툴바 만들기	

툴바 생성	툴바 리소스 변경, 툴바 명령/갱신 핸들러 구현, 범위 매크로를 사용한 핸들러 지정, 툴바 비트맵 배경 투명 처리
툴바 띄우기	툴바 공중에 띄우기, 툴바 크기 변경, 윈도우 생성 위치 알기
(3) 커서 설정	
커서 설정	툴바 클래스를 상속받아서 툴바 메시지 처리(서브클래싱), 커서 변경, 마우스 캡처, 마우스 메시지로 드래그&드롭 구현, 클릭된 툴바 버튼 인덱스 얻기
기타 정보 보관	리소스 크기 단위(DLU) 계산, RUNTIME_CLASS( ) 매크로 활용, 영역 이동(CRect::OffsetRect( )), 화면 좌표계 변환
메시지 전송	사용자 메시지 정의 메시지로 데이터 전송
4. 편집 모드 지원	
(1) 드래그&드롭 구성	사용자 메시지 수신(ON_MESSAGE( ) 매크로), 클라이언트 좌표계 변환, 마우스로 드래그하는 테두리 그리기(CDC::DrawDragRect( )), 객체 리스트 사용(CObList), 실행 시간 클래스로 객체 종류 확인(CObject::IsKindOf( )), 실행 시간 클래스로 동적 생성(CRuntimeClass::CreateObject( ))
(2) 크기 변경 및 이동 지원	
CMyButton 클래스 생성	버튼 클래스 상속받아 버튼 메시지 처리, DECLARE_SERIAL( ) 매크로 활용, 버튼 컨트롤 동적 생성, 윈도우 폰트 얻고 변경하기, 자식 윈도우 자동 소멸 처리(CWnd::PostNcDestroy( ))
이동 지원과 커서 설정	마우스로 클릭한 위치 반영, 커서 클리핑, 다른 곳에서 사용자 메시지 보내기
크기 변경 지원	위치를 기준으로 영역 검사(CRect::PtInRect( )), 영역 확장/축소(CRect::In/DeflateRect( ))
공통 클래스 구현	베이스 클래스 설계, 순수 가상 함수 활용, 클래스 멤버 변수에 레퍼런스 활용, 베이스 클래스로 공통 처리 수행
5. 속성 대화상자	
(1) 컨텍스트 메뉴	메뉴 동적 생성, 컨텍스트 메뉴 띄우기, 윈도우 사용자 공간 활용(Get/SetWindowLongPtr( ))

(2) 속성 대화상자 구현	대화상자 생성, 자식 윈도우 동적 생성과 배치, 베이스 클래스로 자식 윈도우 관리, 자식 윈도우 속성 활용(WS_CHILD), 대화상자 템플릿으로 클래스 생성
(3) 컨트롤 속성 구현	레퍼런스로 대화상자 변수 자동화 처리, 윈도우 타이틀 설정(CWnd::SetWindowText( ))
6. 실행 모드 지원	
(1) 편집 모드와 구분 동작	외부 프로그램 실행, 최소화 및 종료
(2) 실행 모드 라우팅	순수 가상 함수로 열거 후 라우팅 구현
7. 다른 컨트롤 구현	
(1) 컨트롤 클래스 구현	사용자 그리기 처리(CDC::DrawItem( )), 그룹 박스 그리기(CDC::DrawEdge( )), 비트맵 파일을 처리하는 루틴
애니메이션 컨트롤	외부에서 메시지 전달, 영역 확인으로 객체 선정, 실행 시간 클래스로 해당 객체 찾기
(2) 속성 대화상자	파일 대화상자와 경로 얻기, 윈도우 속성 변경(CWnd::ModifyStyle( )), 컨트롤 활성화/비활성화(CWnd::EnableWindow( ))
(3) 실행 모드 지원	웹 페이지 열기, 밑줄 있는 폰트 생성, 타이머, 소켓을 통한 네트워크 데이터 전송, 작업자 스레드 생성, 콤보 박스 목록에 데이터 저장, 콤보 박스 목록에 입력 데이터 추가

02 응용 프로그램 제작

FormPad 프로젝트를 위한 응용 프로그램을 생성한다. FormPad 프로젝트는 SDI 도큐먼트/뷰 구조에 폼뷰Form View를 사용한다. 물론 대화상자를 기반으로 개발할 수도 있지만, 폼뷰를 사용하면 대화상자 기반 환경과 함께 도큐먼트/뷰 구조의 장점을 동시에 누릴 수 있다.

1 폼뷰 SDI 응용 프로그램

FormPad 이름으로 폼뷰 SDI 응용 프로그램을 생성하고 기본 기능을 수정한다. FormPad 프로그램은 크기 변경을 허용하지만, 최대화 버튼은 제공하지 않는다. 또한 폼뷰의 크기를 줄일 때 나타나는 스크롤바도 없애도록 한다.

1.1 FormPad 프로젝트 생성

비주얼 스튜디오를 실행한 후 [파일]–[새로 만들기]–[프로젝트...] 메뉴를 선택하여 [새 프로젝트 만들기] 대화 상자를 연다. 프로젝트 종류 중 'MFC 앱'을 선택하고 〈다음〉을 클릭한다. 프로젝트 이름으로 'FormPad'를 입력하고 프로젝트가 생성될 폴더를 지정한다. 그리고 '솔루션 및 프로젝트를 같은 디렉터리에 배치' 옵션을 체크한 후 〈만들기〉를 클릭한다. 각 단계별로 변경할 사항은 [표 16-2]와 같다. 특별히 고급 기능 단계에서는 소켓 기능을 추가하고, 생성된 클래스 단계에서는 CFormPadView 클래스가 CFormView 클래스를 상속받도록 변경한다([그림 16-4]와 [그림 16-5] 참고).

표 16-2 응용 프로그램 마법사 단계별 변경 사항

응용 프로그램 마법사 단계	변경 사항
애플리케이션 종류	'애플리케이션 종류'로 '단일 문서' 선택 – '문서/뷰 아키텍처 지원' 선택되어 있음 '프로젝트 스타일'로 'MFC standard' 선택
문서 템플릿 속성	'파일 확장명'으로 "pad"를 입력

사용자 인터페이스 기능	없음 – ‘초기 상태 표시줄’, ‘클래스 메뉴 사용’, ‘클래식 도킹 도구 모음 사용’ 선택되어 있음
고급 기능	‘ActiveX 컨트롤’ 옵션 해제 ‘인쇄 및 인쇄 미리 보기’와 ‘공용 컨트롤 매니페스트’를 제외한 모든 옵션 해제 ‘Windows 소켓’ 선택
생성된 클래스	뷰(View) 기본 클래스를 ‘CFormView’로 변경

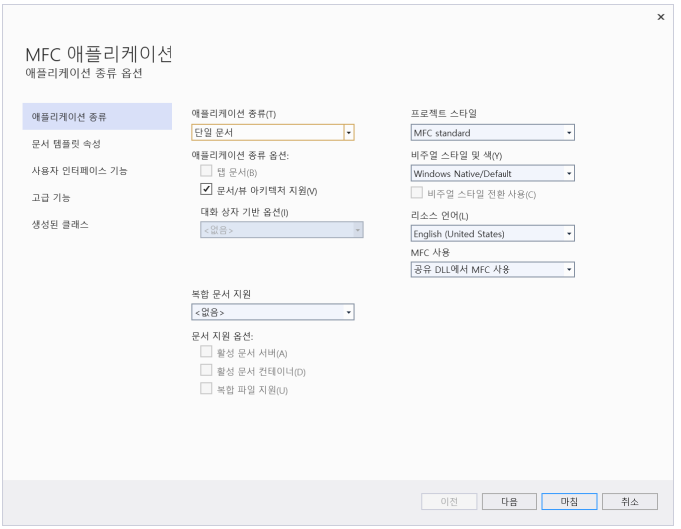


그림 16–4 응용 프로그램 마법사의 고급 기능 단계

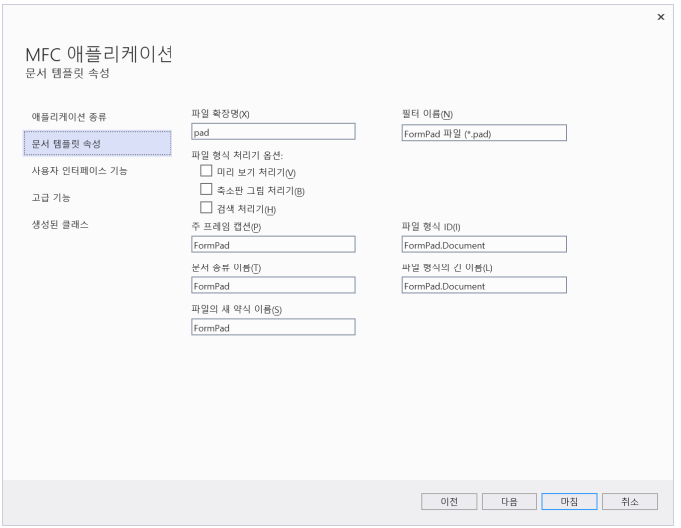


그림 16–5 응용 프로그램 마법사의 생성된 클래스 단계

1.2 최대화 버튼 비활성화

[그림 16-6]과 같이 최대화 버튼을 비활성화하려면 CMainFrame 클래스를 수정해야 한다. PreCreateWindow() 멤버 함수에 있는 CREATESTRUCT 구조체의 WS_MAXIMIZEBOX 속성을 제거한다.

```
BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
        ...
        cs.style &= ~WS_MAXIMIZEBOX;
        return TRUE;
}
```

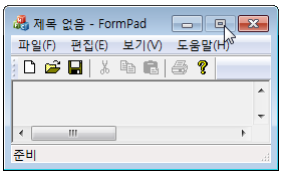


그림 16-6 최대화 버튼 비활성화

1.3 스크롤바 없애기

폼뷰 기반 응용 프로그램은 [그림 16-6]과 같이 크기를 줄이면 스크롤바가 생긴다. 그런데 FormPad 프로그램은 편집 모드에서 조절한 크기가 바로 실행 모드 크기이므로, 스크롤바가 적합하지 않다. 따라서 클래스 마법사(Ctrl+Shift+X)를 이용하여 CFormPadView 클래스에 WM_SIZE 메시지를 처리하는 OnSize() 함수를 생성하고 다음과 같이 수정하여 스크롤바를 없앤다.

```
void CFormPadView::OnSize(UINT nType, int cx, int cy)
{
        CFormView::OnSize(nType, cx, cy);

        CFormView::ShowScrollBar(SB_VERT, FALSE);
        CFormView::ShowScrollBar(SB_HORZ, FALSE);
}
```

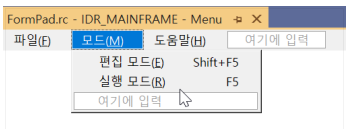
2 편집 모드와 실행 모드 전환 구성

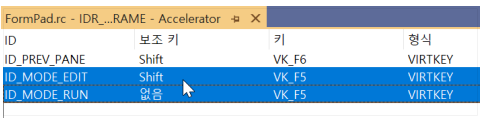
FormPad 프로그램의 편집 모드와 실행 모드를 위한 기본 응용 프로그램 기능을 구현한다. 두 모드 간 전환을 위한 메뉴와 단축키를 제공한다. 메뉴와 타이틀 바는 편집 모드에서만 보이게 해야 하므로 실행 모드에서 타이틀 바 없이 프로그램을 이동시키는 기능도 구현한다.

2.1 메뉴 편집과 단축키

리소스 뷰(Ctrl+Shift+E)에서 [Menu]-[IDR_MAINFRAME] 항목을 선택하여 [편집(E)]과 [보기(V)] 메뉴를 제거한다. 그리고 다음 표를 참고하여 [모드(M)] 메뉴와 하위 메뉴들을 추가하고, [Accelerator]-[IDR_MAINFRAME]에도 단축키를 추가한다. 단축키로 Shift를 입력할 경우에는 아래 그림과 같이 보조키 항목을 사용해야 한다.

참고 실행 모드와 편집 모드의 단축키는 각각 비주얼 C++의 디버그 실행과 디버그 중단 단축키와 같다. FormPad 프로젝트가 비주얼 C++에서 소스 코드를 편집하고 프로그램을 실행한다는 느낌을 주도록 설계했기 때문이다.





메뉴	ID	설정	
모드	(Pop-up 속성)	캡션	모드(&M)
편집 모드	ID_MODE_EDIT	캡션	편집 모드(&E)\tShift+F5
		단축키	Shift + VK_F5
실행 모드	ID_MODE_RUN	캡션	실행 모드(&R)\tF5
		단축키	VK_F5

그림 16-7 FormPad 메뉴 구성과 실행 모드 단축키 설정

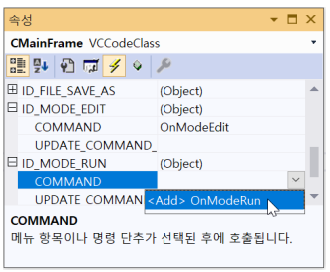


그림 16-8 모드 전환 명령 핸들러 함수 추가

FormPad 프로그램을 실행하면 [Mode]의 하위 메뉴가 모두 비활성화되어 있다. 이것을 해결하려면 명령(COMMAND) 핸들러를 만들어야 한다. 클래스 마법사를 이용하여 CMainFrame 클래스에 [편집 모드] 메뉴(ID_MODE_EDIT)에 대해 OnModeEdit() 함수를, [실행 모드] 메뉴(ID_MODE_RUN)에 대해 OnModeRun() 함수를 각각 만든다.

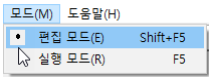


그림 16-9 현재 모드 표시

계속하여 클래스 마법사를 이용하여 [편집 모드] 메뉴(ID_MODE_EDIT)와 [실행 모드] 메뉴(ID_MODE_RUN) 각각에 대해 명령 갱신(UPDATE_COMMAND_UI) 핸들러 함수 OnUpdateModeEdit( )와 OnUpdateModeRun( )을 생성한다. 이들 핸들러 함수에서 CCmdUI::SetRadio( ) 함수를 호출하여 [그림 16-9]와 같이 현재 선택된 모드를 라디오 버튼 형태로 표시할 수 있다.

```
void CMainFrame::OnUpdateModeEdit(CCmdUI*pCmdUI)
{
        pCmdUI->SetRadio(m_bRunMode == FALSE);
}

void CMainFrame::OnUpdateModeRun(CCmdUI*pCmdUI)
{
        pCmdUI->SetRadio(m_bRunMode == TRUE);
}
```

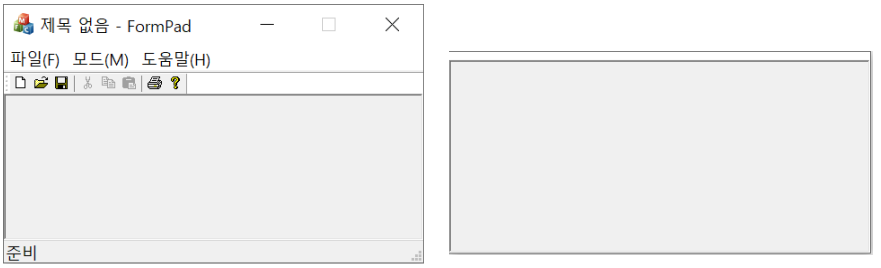
2.2 편집 모드와 실행 모드

FormPad 프로그램은 현재 모드를 보관하기 위한 멤버 변수가 필요하다. 그리고 편집 모드와 실행 모드 전환 시 윈도우 크기를 변경해야 하므로 크기 차이도 저장해야 한다. CMainFrame 클래스에 현재 모드를 저장하기 위한 m_bRunMode 변수와 윈도우의 크기 차이를 저장하기 위한 m_iDifference 멤버 변수를 추가한다.

```
class CMainFrame : public CFrameWnd
{
…
// 특성입니다.
public:
        BOOL &GetMode( )        {return m_bRunMode;}
        …
protected:
        BOOL            m_bRunMode;
        const int       m_iDifference;
};
```

```
CMainFrame::CMainFrame( )
        : m_bRunMode(FALSE)
        , m_iDifference(::GetSystemMetrics(SM_CYCAPTION)
                        + ::GetSystemMetrics(SM_CYMENU))
{
}
```

m_bRunMode 멤버 변수는 BOOL 형으로 선언되어 TRUE면 실행 모드, FALSE면 편집 모드이다. 현재 CMainFrame 생성자에서 m_bRunMode를 FALSE로 설정하였으므로 FormPad 프로그램은 편집 모드로 시작한다. 편집 모드와 실행 모드에서 윈도우 크기 차이는 타이틀 바와 메뉴 크기이므로 ::GetSystemMetrics( ) 함수를 사용하여 값을 계산하여 그 결과를 m_iDifference 멤버 변수에 저장한다. 또한 GetMode( ) 함수가 제공되어 현재 모드를 CMainFrame 클래스 외부에서 확인할 수 있다. 다음과 같이 편집 모드에서 실행 모드로 전환될 때, 타이틀 바, 메뉴, 툴바를 숨기고 크기를 조절해야 한다. 이 기능을 CMainFrame 클래스의 OnModeRun( ) 멤버 함수에 구현하자.



(a) 편집 모드 윈도우 (b) 실행 모드 윈도우

그림 16-10 윈도우 모드 전환

```
void CMainFrame::OnModeRun( )
{
        if(m_bRunMode == TRUE)
                return;
        m_bRunMode = TRUE;

        HWND hWnd = GetSafeHwnd( );
        LONG_PTR lStyle = ::GetWindowLongPtr(hWnd, GWL_STYLE);
        lStyle &= ~WS_CAPTION;
```

```
        :: SetWindowLongPtr(hWnd, GWL_STYLE, lStyle);

        CRect rect;
        GetWindowRect(rect);
        rect.top += m_iDifference;

        SetWindowPos(NULL, rect.left, rect.top,
                rect.Width( ), rect.Height( ),
                SWP_NOZORDER|SWP_FRAMECHANGED);
        RecalcLayout(TRUE);
}
```

타이틀 바를 숨기는 것은 윈도우 스타일에서 WS_CAPTION을 제거하는 것이다. 이러한 작업은 ::SetWindowLongPtr( ) 함수와 GWL_STYLE 플래그로 수행할 수 있다. 다른 스타일에 영향을 주지 않기 위해, ::GetWindowLongPtr( ) 함수로 먼저 윈도우 스타일을 받아오는 것도 잊지 말자. 그리고 윈도우 크기를 줄이는 것은 SetWindowPos( ) 함수를 사용한다. 이 함수는 관심 없는 부분을 제외시키는 플래그를 지원한다. 따라서 본래 첫 번째 인자에 윈도우 Z축 위치를 지정해야 하지만, 현재 NULL을 넣고 SWP_NOZORDER 플래그로 무시하도록 설정하였다. 그리고 ::SetWindowLongPtr( ) 함수로 수정된 내용을 반영하기 위해, 반드시 SetWindowPos( ) 함수와 SWP_FRAMECHANGED 플래그를 사용해야 한다. RecalcLayout( ) 함수는 크기 변경에 따라 프레임 내 윈도우가 재배치하도록 한다. m_iDifference 멤버 변수는 크기 계산을 위해, m_bRunMode 멤버 변수는 변경된 모드를 저장하기 위해 사용되었다.

OnModeEdit( ) 멤버 함수는 OnModeRun( ) 멤버 함수와 정반대 역할을 수행한다. 따라서 소스 코드도 이와 유사하게 구성하면 된다.

```
void CMainFrame::OnModeEdit( )
{
        if(m_bRunMode == FALSE)
                return;
        m_bRunMode = FALSE;

        HWND hWnd = GetSafeHwnd( );
        LONG_PTR lStyle = ::GetWindowLongPtr(hWnd, GWL_STYLE);
        lStyle |= WS_CAPTION;
```

```
        :: SetWindowLongPtr(hWnd, GWL_STYLE, lStyle);

        CRect rect;
        GetWindowRect(rect);
        rect.top -= m_iDifference;

        SetWindowPos(NULL, rect.left, rect.top,
                rect.Width( ), rect.Height( ),
                SWP_NOZORDER|SWP_FRAMECHANGED);
        RecalcLayout(TRUE);
}
```

앞에서 타이틀 바를 없애고 크기 변경을 수행했고, 이제 메뉴와 상태줄 그리고 툴바를 없애거나 나타내기 위해 각각 다음과 같은 코드를 추가한다.

```
void CMainFrame::OnModeRun( )
{
        …
❶       SetMenu(NULL);
        m_wndStatusBar.ShowWindow(SW_HIDE);
        if (m_wndToolBar.IsFloating( ) == TRUE)
                m_wndToolBar.GetDockingFrame( )->ShowWindow(SW_HIDE);
        else
                m_wndToolBar.ShowWindow(SW_HIDE);

        HWND hWnd = GetSafeHwnd( );
        …
}
void CMainFrame::OnModeEdit( )
{
        …
❷       SetMenu(CMenu::FromHandle(m_hMenuDefault));
❸       m_wndStatusBar.ShowWindow(SW_SHOW);
        if (m_wndToolBar.IsFloating( ) == TRUE)
❺               m_wndToolBar.GetDockingFrame( )->ShowWindow(SW_SHOW);
        else
```

```
❹                m_wndToolBar.ShowWindow(SW_SHOW);

        HWND hWnd = GetSafeHwnd( );
        …
}
```

❶ 메뉴 설정은 SetMenu( ) 함수를 사용한다. NULL을 지정하면 사라지고, 특정 메뉴를 지정하면 해당 메뉴가 나타난다. ❷ m_hMenuDefault는 CFrameWnd 클래스 멤버 변수로, 현재 프로그램의 기본 메뉴를 보관한다. 따라서 SetMenu( ) 함수로 지정하여 사라진 메뉴를 복원할 수 있다. 즉, ❸ 상태줄(Status Bar)은 m_wndStatusBar 멤버 변수에 ShowWindow( ) 함수를 사용하여 화면에서 보이거나 사라지게 처리할 수 있다. ❹ 툴바의 경우도 m_wndToolBar 멤버 변수에 ShowWindow( ) 함수를 사용할 수 있다. ❺ 그런데 툴바가 떠 있지 않고 프레임 윈도우에 붙어 있을 경우 GetDockingFrame( ) 함수가 CMainFrame 윈도우가 된다. 이 경우에 ShowWindow( ) 함수와 SW_HIDE 플래그를 사용하면 FormPad 프로그램 자체가 사라진다. 따라서 툴바는 IsFloating( ) 함수를 사용하여 두 가지 상황을 다르게 처리한다는 것에 주목하자.

2.3 실행 모드에서 이동 처리

실행 모드의 FormPad 프로그램은 타이틀 바가 없어 윈도우를 이동하기 불편하다. 그러므로 타이틀 바 대신 왼쪽 마우스 버튼을 눌러 이동할 수 있도록 대화상자를 구현하자. 클래스 마법사를 이용하여 CFormPadView 클래스에 WM_LBUTTONDOWN 메시지를 처리하는 OnLButtonDown( ) 함수를 생성하고, 다음과 같이 수정한다. 왼쪽 마우스 버튼이 눌리면 타이틀 바가 눌린 것처럼 프레임 윈도우에 메시지를 보내는 방식이다. WM_NCLBUTTONDOWN 메시지와 HTCAPTION 플래그를 사용한다. 메시지 전송 함수로 PostMessage( )와 SendMessage( ) 모두 사용할 수 있다. 그리고 실행 모드에서만 동작해야 하므로 메인 프레임 윈도우에게 모드를 확인하는 과정이 있다(CMainFrame 클래스를 참조하기 때문에 MainFrm.h 헤더를 포함시켜야 컴파일된다).

```
#include "FormPadView.h"
#include "MainFrm.h"
…
void CFormPadView::OnLButtonDown(UINT nFlags, CPoint point)
{
```

```
        CMainFrame *pFrame = (CMainFrame *)GetParentFrame( );
        if(pFrame->GetMode( ) == TRUE)
                pFrame->PostMessage(WM_NCLBUTTONDOWN,
                        HTCAPTION, MAKELPARAM(point.x, point.y));
        CFormView::OnLButtonDown(nFlags, point);
}
```

03 컨트롤 툴바 제작

응용 프로그램 마법사가 기본으로 제공하는 툴바를 FormPad 프로그램에서 사용할 컨트롤 툴바로 교체한다. 기본 툴바처럼 컨트롤을 마우스로 드래그&드롭하는 방식을 채택한다. 따라서 컨트롤 툴바는 컨트롤을 드래그할 때 해당 커서를 설정하고, 마우스 움직임을 메시지로 전송하는 역할을 수행한다. 필요한 리소스는 빠른 이해를 위해 비주얼 C++ 프로그램에서 빌려오자.

1 리소스 준비

FormPad 프로젝트는 비주얼 C++를 참고하므로 비주얼 C++ 프로그램의 리소스를 개발에 활용하자. 하지만 비주얼 C++ 같은 상용 프로그램은 라이선스가 있으므로 빌린 리소스로 상용 프로그램은 제작할 수 없다. 여기서는 단지 교육 목적으로 사용하는 것임을 밝혀둔다.

[비주얼 C++ 설치 폴더](예를 들어, C:₩Program Files (x86)₩Microsoft Visual Studio 12.0)의 하위 폴더(VC₩vcpackages₩1042 - 영문 버전: 1033, 한글 버전: 1042)에 reseditui.dll 파일이 있다. 비주얼 C++에서 reseditui.dll 파일을 리소스 형태로 열면, [그림 16-11]과 같이 reseditui.dll에 포함된 리소스가 열린다. 여기서 Bitmap과 Cursor 범주를 열어 리소스를 [표 16-3]과 같이 사용할 것이다.

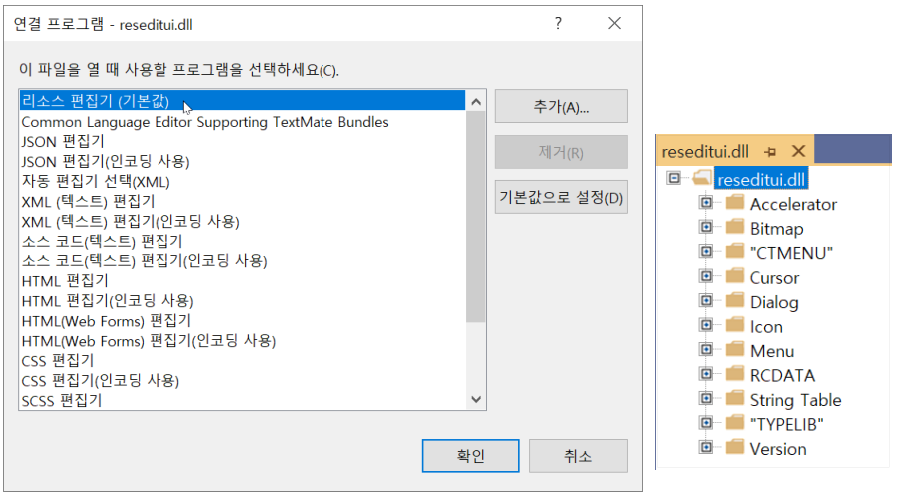


그림 16-11 reseditui.dll 리소스 파일 열기

참고 파일을 리소스 형태로 여는 방법 : [열기(O)] 옆 화살표 버튼을 클릭하여 [연결 프로그램(W)...] 버튼을 선택한 후 [리소스 편집기]를 선택한다.

표 16-3 컨트롤 툴바의 비트맵과 커서 ID

컨트롤	비트맵(ID)	커서(ID)
선택	(26189)	
정적 텍스트	(26197)	(26168)
그룹 박스	(26195)	(26166)
그림	(26198)	(26169)
애니메이션	(26208)	(26205)
월력	(26212)	(26303)
날짜 시간 선택기	(26211)	(26174)*
IP 주소	(26213)	(26299)
콤보 박스	(26193)	(26164)
버튼	(26190)	(26160)

참고 날짜 시간 선택기의 커서는 제공되지 않아서 유사한 것을 선택하였으므로, 필요에 한 경우 복사하여 수정하면 됨

리소스 뷰의 Toolbar 범주에 IDR_CONTROLS 툴바를 추가한다. 그리고 reseditui.dll의 툴바 비트맵을 [그림 16-12]와 같이 이 툴바에 차례로 복사하여 붙여 넣는다. 반면 커서는 리소스 뷰로 바로 드래그&드롭을 지원하지 않으므로, 커서 파일(*.cur)을 직접 추가하여 [그림 16-13]과 같이 완성한다.

참고 커서를 빼낸 reseditui.dll 파일을 저장하지 않도록 주의하자!

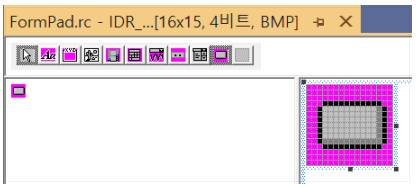


그림 16-12 컨트롤 툴바

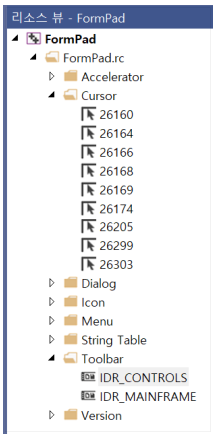


그림 16-13 컨트롤 툴바 및 커서 추가

여기서 잠깐! 리소스 편집기 내의 리소스를 리소스 스크립트(*.rc)에 추가하는 방법

리소스 편집기 내의 리소스(예: 커서 등)를 현재 프로젝트의 리소스 스크립트(*.rc)에 추가하는 방법으로 크게 두 가지 방법을 소개할 수 있다.

첫 번째 방법은 리소스 편집기를 이용하는 방법이다. 리소스 편집기와 리소스 뷰가 분리되면서부터 리소스 편집기에서 리소스 뷰로 드래그&드롭을 지원하지 않게 되었다. 대신 리소스 편집기와 리소스 편집기 간은 드래그&드롭을 여전히 지원한다. 따라서 현재 프로젝트의 리소스 스크립트(*.rc)를 리소스 뷰가 아닌 리소스 편집기에서 열어서 드래그&드롭을 하는 방식이다. 방법은 솔루션 편집기 등에서 현재 프로젝트의 리소스 스크립트(*.rc)를 선택한 후 마우스 오른쪽 버튼을 누르고 [연결 프로그램(N)...] 메뉴를 선택하여 [리소스 뷰]가 아닌 [리소스 편집기]를 선택하면 된다.

두 번째 방법은 리소스 편집기 내 리소스들을 리소스 스크립트(*.rc)로 저장하는 방법이다. [파일(F)] 메뉴의 [다른 이름으로 … 저장(A)...] 하위 메뉴를 선택하여 파일 형식(T)을 '리소스 스크립트 (*.rc)' 저장한다. 그러면 리소스 스크립트(*.rc) 뿐만 아니라 모든 리소스가 파일로 저장된다. 그러면 현재 프로젝트의 리소스 뷰에서 이들 리소스 파일을 선택하여 리소스로 가져오는 방식으로 추가하면 된다.

2 툴바 생성

앞서 준비한 툴바 비트맵으로 교체하고 비주얼 C++ 컨트롤 툴바처럼 프레임 윈도우에서 떨어지도록 배치한다.

2.1 툴바 만들기

CMainFrame 클래스의 OnCreate() 멤버 함수에 LoadToolBar() 함수를 호출하는 부분이

있다. 기존의 IDR_MAINFRAME을 다음과 같이 새로 준비한 IDR_CONTROLS로 변경하면 [그림 16-14]와 같이 다른 형태의 툴바가 나타난다.

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        …
        !m_wndToolBar.LoadToolBar(IDR_CONTROLS))
        …
}
```

[그림 16-14] 변경된 툴바 버튼

현재는 명령 핸들러가 없어 툴바 버튼이 모두 비활성화되어 있다. 명령 핸들러 10개를 일일이 추가할 수도 있지만, 다음 코드와 같이 범위(range) 매크로를 사용하여 하나의 명령 핸들러로 처리할 수도 있다(단, 범위 매크로를 사용할 때는 반드시 Resource.h 헤더 파일에서 정렬 여부를 확인하여 정렬되어 있지 않으면 직접 수정한다. 여기서는 32781~32790까지 10개가 정렬되어 있다고 가정한다). 하나의 명령 핸들러 함수, 즉 OnControl() 멤버 함수만 만들어 준다.

```
class CMainFrame : public CFrameWnd
{
        …
protected:
        afx_msg void CMainFrame::OnControl(UINT nID);
}
```

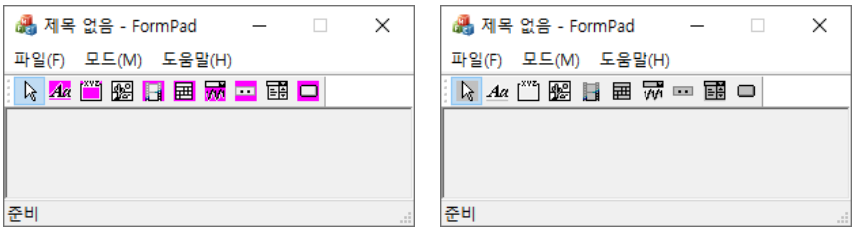
```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
…
        ON_COMMAND_RANGE(32781, 32790, &CMainFrame::OnControl)
END_MESSAGE_MAP( )
…
```

```
void CMainFrame::OnControl(UINT nID)
{
}
```

이 중 선택 버튼(32781)에 한해, 항상 선택된 효과(버튼이 눌려진 상태)를 주기 위해 다음과 같이 핸들러를 갱신한다. 그런 다음 실행하면 [그림 16-15](a)와 같이 활성화된 툴바를 볼 수 있다.

```
class CMainFrame : public CFrameWnd
{
        ...
protected:
        afx_msg void CMainFrame::OnUpdateButton32781(CCmdUI*pCmdUI);      ...
}
```

```
BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
...
        ON_UPDATE_COMMAND_UI(ID_BUTTON32781,
        &CMainFrame::OnUpdateButton32781)
END_MESSAGE_MAP( )
...
void CMainFrame::OnUpdateButton32781(CCmdUI*pCmdUI)
{
        pCmdUI->SetRadio(TRUE);
}
```



(a) (b)

그림 16-15 활성화된 툴바 버튼

그런데 툴바 버튼이 모두 활성화되자 툴바 비트맵의 배경색(R:255, G:0, B:255)이 투명 처리되지 않고 그대로 나타난 것을 알 수 있다. 직접 비트맵을 수정하여 배경색을 원하는 색으로 통일시킬 수 있지만, 여기서는 이미지 리스트를 이용하여 색상키 Color Key 를 지정하고 배경색을 투명 처리하는 방식을 사용한다.

CToolBar::LoadToolBar( ) 함수 호출이 성공하면 내부적으로 툴바 리소스를 분석하여 툴바 버튼 정보를 보관하고 CToolbarCtrl 객체 내에 이미지 리스트 객체를 생성한다. 다음 코드는 색상키를 지정하여 생성한 CImageList 객체를, CToolbar 객체 내 CToolbarCtrl 객체에 보관된 CImageList 객체와 바꾸는 과정(CToolbarCtrl::SetImageList( ) 함수)을 진행한다. 기존에 생성된 이미지 리스트 객체는 객체 포인터를 받아서 CImageList::DeleteImageList( ) 함수를 호출해 정상적으로 삭제한다. 아울러 지역변수로 선언된 CImageList 객체는 이제 툴바 내에서 사용하도록 설정되었으므로, 블록 밖으로 벗어날 때 삭제되지 않도록 CImageList::Detach( ) 함수를 호출하여 분리한다. 그런 다음 실행하면 [그림 16-15](b)와 같이 활성화된 툴바를 볼 수 있다.

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        …
        if (!m_wndToolBar.CreateEx(…) ||
                !m_wndToolBar.LoadToolBar(IDR_CONTROLS))
        {       …
        }

        // 툴바의 투명색 설정
        CImageList imgToolBar;
        BOOL bRtn = imgToolBar.Create(
                IDR_CONTROLS, 16, 15, RGB(255,0,255) );
        if ( bRtn != FALSE )
        {
                CImageList *pImageList = m_wndToolBar.GetToolBarCtrl( )
                        .SetImageList( &imgToolBar );
                if ( pImageList != NULL )
                        pImageList->DeleteImageList( );
                imgToolBar.Detach( );
        }
        …
}
```

2.2 툴바 띄우기

이제 툴바를 왼쪽에 수직으로 길게 배열하기 위해 CMainFrame 클래스의 OnCreate( ) 멤버 함수를 수정한다.

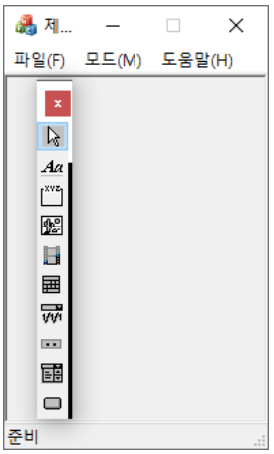


그림 16-16 컨트롤 툴바 띄우기

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
        ...
        CToolBarCtrl& tbctrl = m_wndToolBar.GetToolBarCtrl( );
        int nButtonCount = tbctrl.GetButtonCount( );
❹       CPoint pt( lpCreateStruct->x + 10, lpCreateStruct->y + 70 );
        CRect rect;

        FloatControlBar( &m_wndToolBar, CPoint(0, 0) ); // 먼저 띄운다!
❶       tbctrl.SetRows( nButtonCount, TRUE, rect );
❷       m_wndToolBar.CalcDynamicLayout( rect.Width( ), LM_HORZ|LM_COMMIT );
❸       FloatControlBar( &m_wndToolBar, pt,
                CBRS_ALIGN_TOP|CBRS_SIZE_DYNAMIC);

        return 0;
}
```

툴바를 띄우는 것은 FloatControlBar( ) 함수를 호출하는 것만으로 충분하지만 수직으로 길게 배열하려면 세 단계를 더 거쳐야 한다. ❶ 먼저 SetRows( ) 함수를 사용하여 한 줄로 배열할 때의 영역 정보를 얻는다. ❷ 그리고 CalcDynamicLayout( ) 함수를 사용하여 툴바가 크기를 다시 계산하도록 한다. ❸ 마지막으로 다시 FloatControlBar( ) 함수를 호출하여 계산된 크기를 최종 반영한다. 툴바가 떠 있는 위치는 프레임 윈도우에 제한되지 않고, 화면 내 어디라도 가능하다. 따라서 ❹ FormPad 프로그램을 처음 실행했을 때 프레임 윈도우 내에 있도록, lpCreateStruct 구조체의 좌표를 활용한다.

3 커서 설정

컨트롤 툴바를 마우스 왼쪽 버튼으로 드래그하는 작업을 수행하자. 이때 [그림 16-17]처럼 커서를 드래그하는 모습이 보이게 설정해야 한다. 그런데 지금까지 사용한 CToolBar 클래스로는 부족하다. 마우스 메시지를 받아 적절히 처리해야 하는데, CToolBar 클래스로는 불가능하기 때문이다. 이를 위해 MFC에서 권장하는 방법이 '상속' 또는 서브클래싱이다. FormPad 프로젝트에서는 CToolBar 클래스를 상속받는 CControlToolBar 클래스를 구현하자.

그림 16-17 컨트롤 툴바와 커서

참고 클래스 마법사 등을 이용하여 MFC 클래스를 생성할 때, 상속할 대상 클래스 목록에 CToolBar 클래스를 지정한다.

CControlToolBar 클래스를 생성한 후, 다음과 같이 CMainFrame 클래스를 수정한다. CToolBar로 되어 있던 내용을 CControlToolBar로 변경하는 것이다. 그러면 모든 메시지는 CControlToolBar 클래스에 자동 전달된다.

```
#include "ControlToolBar.h"
class CMainFrame : public CFrameWnd
{
        ...
        CControlToolBar m_wndToolBar;
        ...
};
```

3.1 커서 설정 마우스 처리

커서 설정을 위해 다음과 같이 CControlToolBar 클래스에 멤버 변수를 선언하고 초기화한다.

```
class CControlToolBar : public CToolBar
{
        ...
protected:
        int     m_iIndex;
        BOOL    m_bDragging;
        UINT    m_CursorArray[10];
};
```

```
CControlToolBar::CControlToolBar( )
        : m_iIndex(-1), m_bDragging(FALSE)
{
        m_CursorArray[0] = 0;
        m_CursorArray[1] = 26168;
        m_CursorArray[2] = 26166;
        m_CursorArray[3] = 26169;
        m_CursorArray[4] = 26205;
        m_CursorArray[5] = 26303;
        m_CursorArray[6] = 26174;
        m_CursorArray[7] = 26299;
        m_CursorArray[8] = 26164;
        m_CursorArray[9] = 26160;
}
```

m_iIndex 멤버 변수는 현재 클릭된 툴바 버튼 인덱스이고, m_bDragging 멤버 변수는 현재 마우스로 드래그 중임을 나타내는 플래그이다. 그리고 m_CursorArray 배열은 툴바 버튼의 커서 ID 배열이다. 생성자에서 이들 멤버 변수를 초기화한다. 특히 m_CursorArray의 경우 이전 [표 16-3]에 따라 초기화한다. 그리고 0번 인덱스를 비워둠으로써, 배열의 인덱스를 컨트롤 툴바 버튼의 인덱스와 동일하게 설정하였다.

마우스 버튼 드래그를 구현하려면 세 가지 마우스 메시지(WM_LBUTTONDOWN, WM_MOUSEMOVE, WM_LBUTTONUP)가 필요하다. 클래스 마법사를 이용하여 이들 각각에 대한 메시지 핸들러 함수(OnLButtonDown( ), OnMouseMove( ), OnLButtonUp( ))를 생성한 후 다음과 같이 수정한다.

```
void CControlToolBar::OnLButtonDown(UINT nFlags, CPoint point)
{
❶       m_iIndex = GetToolBarCtrl( ).HitTest(&point);
❷       if(m_iIndex > 0)
        {
❸               m_bDragging = TRUE;
❻               ::SetCursor(AfxGetApp( )->LoadCursor(m_CursorArray[m_iIndex]));
❽               SetCapture( );
        }
```

```
        CToolBar::OnLButtonDown(nFlags, point);
}

void CControlToolbar::OnMouseMove(UINT nFlags, CPoint point)
{
❹❺      if(m_bDragging == TRUE && nFlags & MK_LBUTTON)
        {
        }
        CToolBar::OnMouseMove(nFlags, point);
}

void CControlToolbar::OnLButtonUp(UINT nFlags, CPoint point)
{
❹       if(m_bDragging == TRUE)
        {
❽               ReleaseCapture( );
❼               ::SetCursor(::LoadCursor(NULL, IDC_ARROW));
                m_bDragging = FALSE;
        }
        CToolBar::OnLButtonUp(nFlags, point);
}
```

컨트롤 툴바에서 마우스 왼쪽 버튼을 누르면 OnLButtonDown( ) 멤버 함수가 호출된다. 그러면 ❶ GetToolBarCtrl( ) 함수로 CToolBarCtrl 클래스를 얻고, 다시 HitTest( ) 함수를 사용하여 마우스가 클릭된 툴바 버튼 인덱스를 얻어 결과 값을 m_iIndex에 보관한다. ❷ 결과 값이 0보다 큰 경우만 처리한다(0은 선택 버튼). ❸ 검사를 통과하면 m_bDragging을 TRUE로 변경하여 마우스 드래그 시작을 기록한다. ❹ OnMouseMove( )와 OnLButtonUp( ) 멤버 함수는 m_bDragging 멤버 변수를 통해 마우스 드래그로 호출되었는지 여부를 판단한다. ❺ 특히OnMouseMove( ) 멤버 함수는 nFlag 변수를 통해 마우스 왼쪽 버튼을 누른 상태로 이동했는지도 확인해야 한다.

❻ 커서는 ::SetCursor( ) API 함수로 설정하며, ::LoadCursor( ) API 함수로 불러온다. OnLButtonDown( ) 멤버 함수에서 현 툴바 버튼에 해당하는 마우스 커서를 설정하고, ❼ OnLButtonUp( ) 멤버 함수에서 기본 커서(화살표 모양, IDC_ARROW)로 복원한다. ❽ 그리고 컨트롤 툴바를 벗어나도 메시지를 계속 받아야 하므로 마우스 캡처를 수행한다. 역시

OnLButtonDown() 함수에서 설정한 후 OnLButtonDown() 함수가 호출되었을 때 해제한다. 이제 FormPad 프로그램을 실행하면, 이전 [그림 16-17]처럼 툴바 버튼을 드래그하는 효과를 경험할 수 있다.

3.2 기타 정보 보관하기

컨트롤을 폼에 드래그&드롭하기 위해서는 커서 정보 외에 크기와 실행 시간 클래스Run Time Class 정보가 필요하다. 이를 위해 다음과 같은 배열을 멤버 변수로 추가하자. 구조체로 포장할 수도 있으나 10개 인덱스로 통일하면 편하게 쓸 수 있다.

```
class CControlToolBar : public CToolBar
{
        ...
protected:
        CSize                   m_SizeArray[10];
        CRuntimeClass   *       m_ClassArray[10];
        ...
};

CControlToolBar::CControlToolBar()
{
        ...
        m_SizeArray[0] = CSize(0, 0);
        m_SizeArray[1] = CSize(20, 8);
        m_SizeArray[2] = CSize(48, 40);
        m_SizeArray[3] = CSize(20, 20);
        m_SizeArray[4] = CSize(20, 20);
        m_SizeArray[5] = CSize(140, 100);
        m_SizeArray[6] = CSize(100, 15);
        m_SizeArray[7] = CSize(100, 15);
        m_SizeArray[8] = CSize(48, 30);
        m_SizeArray[9] = CSize(50, 14);
        m_ClassArray[0] = RUNTIME_CLASS(CStatic);
        m_ClassArray[1] = RUNTIME_CLASS(CStatic);
        m_ClassArray[2] = RUNTIME_CLASS(CButton);
        m_ClassArray[3] = RUNTIME_CLASS(CStatic);
        m_ClassArray[4] = RUNTIME_CLASS(CAnimateCtrl);
```

```
        m_ClassArray[5] = RUNTIME_CLASS(CStatic);
        m_ClassArray[6] = RUNTIME_CLASS(CStatic);
        m_ClassArray[7] = RUNTIME_CLASS(CIPAddressCtrl);
        m_ClassArray[8] = RUNTIME_CLASS(CComboBox);
        m_ClassArray[9] = RUNTIME_CLASS(CButton);
}
```

여기서 지정한 크기는 비주얼 C++의 RC 파일에서 가져온 값이다. 그런데 리소스 스크립트(*.rc)에서는 글자 폭과 높이를 DLU dialog box unit 단위로 사용하기 때문에, 화면 단위, 즉 픽셀 pixel로 사용하기 위해서는 단위 변환이 필요하다. 이를 위해 ::MapDialogRect( ) API 함수를 사용할 수 있으며, 다음과 같이 컨트롤이 나타날 실제 영역을 계산한다. 그리고 [그림 16-18]처럼 드래그&드롭할 때 커서가 중앙에 오도록 CRect::OffsetRect( ) 함수를 이용하여 중앙 위치로 영역을 조정한다.

```
void CControlToolBar::OnLButtonDown(UINT nFlags, CPoint point)
{
        m_iIndex = GetToolBarCtrl( ).HitTest(&point);
        if(m_iIndex > 0)
        {
                ...
                CFrameWnd *pFrame = (CFrameWnd *)AfxGetMainWnd( );
                CView *pView = pFrame->GetActiveView( );

                CSize size = m_SizeArray[m_iIndex];
                CRect rectUnit(CPoint(0,0), size);
                ::MapDialogRect(pView->GetSafeHwnd( ), &rectUnit);
                size = rectUnit.Size( );

                CPoint pt(point);
                ClientToScreen(&pt);

                CRect rect(pt, size);
                rect.OffsetRect(-rect.Width( ) >> 1, -rect.Height( ) >> 1);
        }
        CToolBar::OnLButtonDown(nFlags, point);
}
```

그림 16-18 마우스로 드래그&드롭할 때 커서 위치

이 코드는 OnLButtonDown(), OnMouseMove() 멤버 함수 모두에 동일하게 쓰여 함수로 포장할 수도 있지만, 다른 곳에서 사용하지 않으므로 그대로 복사해 사용하자. OnLButtonUp() 함수는 OnMouseMove() 멤버 함수가 최종 호출된 후 바로 실행된다. 따라서 OnMouseMove() 멤버 함수가 처리한 것으로 충분하므로, OnLButtonUp() 멤버 함수가 따로 영역을 처리하지 않는다. 그리고 영역을 전달할 때 화면 좌표를 사용하는 것에 주목하자. 클라이언트 좌표를 사용하면 윈도우마다 위치 정보 해석이 달라지므로, 모든 윈도우에 공통인 화면 좌표를 기준으로 한다.

3.3 메시지 전송

이제 폼을 담당하는 CFormPadView 클래스로 계산된 영역과 실행 시간 클래스를 전송해야 한다. CFormPadView 클래스가 윈도우므로 메시지를 사용해 전송하자. 우선 ControlToolBar.h 헤더에 사용자 메시지를 정의한다.

```
#define WM_APP_CONTROL_DOWN  WM_APP + 0
#define WM_APP_CONTROL_DRAG  WM_APP + 1
#define WM_APP_CONTROL_DROP  WM_APP + 2
```

그리고 CControlToolBar 클래스에서 OnLButtonDown()과 OnMouseMove() 멤버 함수는 해당 컨트롤 영역을, OnLButtonUp() 멤버 함수는 해당 컨트롤 실행 시간 클래스를 전송하도록 수정한다. 특히 WPARAM에는 '생성하라'는 의미로 TRUE를 전송한다. 현재 컨트롤은 m_iIndex 멤버 변수로 지정한다.

폼에 데이터와 함께 메시지를 전송함에 따라 컨트롤 툴바가 할 일은 모두 수행했다. 이제 폼에서 메시지를 처리하여 컨트롤을 생성하는 일만 남았다.

```
void CControlToolBar::OnLButtonDown(UINT nFlags, CPoint point)
{
        m_iIndex = GetToolBarCtrl().HitTest( &point );
        if ( m_iIndex > 0 )
```

```
        {
                ...
                BOOL bSuccess = pView->SendMessage(
                        WM_APP_CONTROL_DOWN,
                        (WPARAM)&rect, NULL);
        }
        CToolBar::OnLButtonDown(nFlags, point);
}

void CControlToolBar::OnMouseMove(UINT nFlags, CPoint point)
{
        if ( m_bDragging == TRUE && nFlags & MK_LBUTTON )
        {
                ...
                BOOL bSuccess = pView->SendMessage(
                        WM_APP_CONTROL_DRAG,
                        (WPARAM)&rect, NULL);
        }
        CToolBar::OnMouseMove(nFlags, point);
}

void CControlToolBar::OnLButtonUp(UINT nFlags, CPoint point)
{
        if ( m_bDragging == TRUE )
        {
                ...
                CFrameWnd *pFrame = (CFrameWnd *)AfxGetMainWnd( );
                CView *pView = pFrame->GetActiveView( );
                BOOL bSuccess = pView->SendMessage(
                        WM_APP_CONTROL_DROP,
                        (WPARAM)TRUE, (LPARAM)m_ClassArray[m_iIndex]);
        }
        CToolBar::OnLButtonUp(nFlags, point);
}
```

04 편집 모드 지원

폼을 담당하는 CFormPadView 클래스는 컨트롤 툴바에서 드래그&드롭한 컨트롤을 생성하는 역할을 한다. 컨트롤은 각각 자신에게 전달된 메시지를 처리하므로 부모 윈도우에 컨트롤이 처리한 메시지가 전달되지 않는다. 따라서 컨트롤의 동작을 변경하기 위해서는 해당 컨트롤을 상속받아야 한다. 이를 통해 컨트롤의 이동 및 크기 변경을 구현하고, 해당 동작에 따라 커서도 변경한다. 여기서 컨트롤을 9개 구현하므로, 공통 부분을 추출하여 베이스 클래스를 도출한다.

1 드래그&드롭 구성

앞서 컨트롤 툴바에서 컨트롤을 드래그&드롭할 때 발생하는 사용자 메시지를 처리해 보자. CFormPadView 클래스에 세 가지 메시지(WM_APP_CONTROL_DOWN, WM_APP_CONTROL_DRAG, WM_APP_CONTROL_DROP)를 각각 처리하는 OnControlDown(), OnControlDrag(), OnControlDrop() 멤버 함수를 선언한다. 그리고 메시지 맵에서 ON_MESSAGE() 매크로를 사용하여 이들 관계를 명시한다. 그 밖에 이전 영역을 저장하기 위한 m_rectOld 멤버 변수도 선언한다.

```
class CFormPadView : public CFormView
{
        ...
public:
        afx_msg LRESULT OnControlDown(WPARAM wParam, LPARAM lParam);
        afx_msg LRESULT OnControlDrag(WPARAM wParam, LPARAM lParam);
        afx_msg LRESULT OnControlDrop(WPARAM wParam, LPARAM lParam);

protected:
        CRect    m_rectOld;
};
```

```
BEGIN_MESSAGE_MAP(CFormPadView, CFormView)
        …
        ON_MESSAGE(WM_APP_CONTROL_DOWN, &CFormPadView::OnControlDown)
        ON_MESSAGE(WM_APP_CONTROL_DRAG, &CFormPadView::OnControlDrag)
        ON_MESSAGE(WM_APP_CONTROL_DROP, &CFormPadView::OnControlDrop)
END_MESSAGE_MAP( )

CFormPadView::CFormPadView( ) : CFormView(CFormPadView::IDD)
{
        m_rectOld.SetRectEmpty( );
}
```

OnControlDown( ) 멤버 함수는 다음과 같이 구현한다. 우선 전달되는 영역 정보가 화면 좌표로 전송되므로, ScreenToClient( ) 함수를 사용하여 클라이언트 좌표로 변환한다. 그리고 [그림 16-19]와 같이 DrawDragRect( ) 함수로 마우스로 컨트롤을 드래그할 때 나타나는 테두리를 그린다. 그리고 OnControlDown( ) 함수는 드래그 테두리를 처음 출력하므로, DrawDragRect( ) 함수 세 번째 인자에 NULL을 사용한다. m_rectOld 멤버 변수에 현재 영역을 저장한다.

```
LRESULT CFormPadView::OnControlDown(WPARAM wParam, LPARAM lParam)
{
        if (wParam != NULL)
        {
                CClientDC dc(this);
                CRect *pRect = (CRect *)wParam;
                CRect rect(*pRect);
                ScreenToClient(rect);

                dc.DrawDragRect(rect, CSize(1, 1),
                        m_rectOld.IsRectEmpty( ) ? NULL : m_rectOld,
                        CSize(1, 1));
                if (m_rectOld.IsRectEmpty( ) == FALSE)
                        m_rectOld = rect;
        }
        Invalidate( );
        return 1;
}
```

OnControlDrag() 멤버 함수도 OnControlDown() 멤버 함수와 유사하다. 단, DrawDragRect() 함수 세 번째 인자에 m_rectOld 변수를 사용하여, DrawDragRect() 함수가 이전 기록을 지우도록 한다. 따라서 m_rectOld 변수에 현재 영역을 계속 저장해야 한다.

그림 16-19 컨트롤을 드래그할 때 나타나는 직사각형

```
LRESULT CFormPadView::OnControlDrag(WPARAM wParam, LPARAM lParam)
{
	if (wParam != NULL)
	{
		CClientDC dc(this);
		CRect *pRect = (CRect *)wParam;
		CRect rect(*pRect);
		ScreenToClient(rect);

		dc.DrawDragRect(rect, CSize(1, 1),
			m_rectOld, CSize(1, 1));
		m_rectOld = rect;
	}
	return 1;
}
```

OnControlDrop() 멤버 함수가 호출되면 컨트롤의 최종 위치가 결정되므로, CFormPadView 클래스는 해당 컨트롤을 생성한다. 단, 그 전에 먼저 DrawDragRect() 함수에서 크기를 (0, 0)으로 새로 주어 이전 드래그 테두리만 지우고 새 테두리도 출력하지 않도록 한다. 그리고 m_rectOld 멤버 변수를 초기화한다.

```
LRESULT CFormPadView::OnControlDrop(WPARAM wParam, LPARAM lParam)
{
	CClientDC dc(this);
	dc.DrawDragRect(m_rectOld, CSize(0, 0),
		m_rectOld, CSize(1, 1));
	CRect rect = m_rectOld;
	m_rectOld.SetRectEmpty( );

	Invalidate( );
	return 1;
}
```

컨트롤 객체를 생성하기 전에 생성한 객체를 관리할 리스트를 준비하자. 컨트롤 정보는 곧 FormPad 프로그램의 데이터이므로, 도큐먼트 클래스에서 관리한다(그러면 향후 직렬화를 구현할 때도 용이하다). 따라서 CFormPadDoc 클래스에 다음과 같이 멤버 변수와 멤버 함수를 추가한다.

```
class CFormPadDoc : public CDocument
{
public:
	CObList &GetControlList( ) {return m_listControl;}
	POSITION AddControl(CObject *pObj)
	{
		return m_listControl.AddTail(pObj);
	}
	void RemoveControl(CObject *pObj)
	{
		POSITION pos = m_listControl.Find(pObj);
		if (pos != NULL)
			m_listControl.RemoveAt(pos);
	}
	...
protected:
	CObList		m_listControl;
};
```

먼저 버튼 컨트롤을 구현하자. 우선 CRuntimeClass의 CreateObject( ) 함수로 CObject 객체를 생성하고, RUNTIME_CLASS( ) 매크로와 IsKindOf( ) 함수를 사용하여 CButton인지 확인한다. 확인이 끝나면 CButton 클래스로 캐스팅한 후, Create( ) 함수로 생성한다. 그리고 생성된 객체를 도큐먼트 클래스에 등록한다.

```
LRESULT CFormPadView::OnControlDrop(WPARAM wParam, LPARAM lParam)
{
        ...
        if (wParam == TRUE)
        {
                CRuntimeClass *pClass = (CRuntimeClass *)lParam;
                CObject *pObj = pClass->CreateObject( );
                if(pObj != NULL)
                {
                        if (pObj->IsKindOf(RUNTIME_CLASS(CButton)) == TRUE)
                        {
                                CButton *pButton = (CButton *)pObj;
                                pButton->Create(_T("Button"),
                                    WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
                                    rect, this, 101);
                        }
                        GetDocument( )->AddControl(pObj);
                }
        }
        Invalidate( );
        return 1;
}
```

이 코드는 별 문제가 없어 보이지만 실행하면 동작하지 않는다. 그 이유는 CButton 클래스가 다음과 같이 DECLARE_DYNAMIC( ) 매크로로 선언되어 있기 때문이다. 동적 생성 기능을 사용하려면 DECLARE_DYNCREATE( )이나 DECLARE_SERIAL( ) 매크로가 선언되어야 한다. 찾아보면 다른 컨트롤도 마찬가지임을 알 수 있다. 이 문제를 해결하려면 클래스를 새로 정의해야 한다.

```
class CButton : public CWnd
{
        DECLARE_DYNAMIC(CButton)
        ...
};
```

컨트롤마다 중복되는 부분이 많아 이후 버튼 컨트롤을 중심으로 설명하도록 한다. 다른 컨트롤들은 각자 구현해 보도록 하자. 대신 7절에서 컨트롤마다 다른 부분에 대해 컨트롤 클래스, 속성 대화상자, 실행 모드로 나누어 제시한다.

2 크기 변경과 이동 지원

동적 생성이 가능한 컨트롤 클래스를 생성하고, 생성 후 메모리 관련 문제 등을 해결한다. 그리고 컨트롤 클래스를 상속받으므로 컨트롤에 전달되는 메시지를 처리할 수 있다. 따라서 마우스를 이용한 크기 변경과 이동 기능을 구현한다. 그리고 이를 통해 다른 컨트롤을 위한 공통 부분을 파악하고 베이스 클래스를 이끌어낸다.

2.1 CMyButton 클래스 생성

CButton을 상속받는 CMyButton 클래스를 생성하고, DECLARE_SERIAL() 매크로를 정의한다. DECLARE_DYNCREATE() 매크로를 사용할 수도 있지만, 향후 저장을 위해 직렬화 기능을 지원하는 것이 좋다. DECLARE_SERIAL() 매크로를 사용하려면 Serialize() 함수를 반드시 구현해야 한다. 그리고 컨트롤을 생성하는 Create() 함수도 추가한다. Create() 멤버 함수에는 버튼 컨트롤 생성 후 부모 폰트를 현재 윈도우에 설정하는 부분이 있다. 폰트가 일치해야 다른 컨트롤과 통일성을 이룰 수 있기 때문이다.

```
class CMyButton : public CButton
{
        DECLARE_SERIAL(CMyButton)
public:
        BOOL Create(CRect &rect, CWnd *pParentWnd);
        ...
protected:
        void Serialize(CArchive& ar);
};
```

```
IMPLEMENT_SERIAL(CMyButton, CButton, 1)
…
void CMyButton::Serialize(CArchive& ar)
{
}

BOOL CMyButton::Create(CRect &rect, CWnd *pParentWnd)
{
        BOOL bRtn = CButton::Create(_T("Button"),
                WS_CHILD|WS_VISIBLE|BS_PUSHBUTTON,
                rect, pParentWnd, 101);
        SetFont(pParentWnd->GetFont( ));
        return bRtn;
}
```

이제 CControlToolBar의 생성자에서 실행 시간 클래스를 CButton에서 CMyButton으로 변경한다. 그리고 ControlToolBar.h 헤더 파일에 MyButton.h를 포함시킨다.

```
…
#include "ControlToolBar.h"
#include "MyButton.h"
…
CControlToolBar::CControlToolBar( )
{
        …
        m_ClassArray[9] = RUNTIME_CLASS(CMyButton);
        …
}
```

이제 CFormPadView 클래스의 OnControlDrop( ) 함수를 다음과 같이 수정하여, 컨트롤을 동적으로 생성할 수 있다. 그러면 [그림 16-20]처럼 컨트롤 툴바의 버튼 컨트롤을 마음대로 폼에 드래그&드롭할 수 있다.

```
...
#include "MainFrm.h"
#include "MyButton.h"
...
LRESULT CFormPadView::OnControlDrop(WPARAM wParam, LPARAM lParam)
{
        ...
        if (wParam == TRUE)
        {
                CRuntimeClass *pClass = (CRuntimeClass *)lParam;
                CObject *pObj = pClass->CreateObject( );
                if(pObj != NULL)
                {
                        if(pObj->IsKindOf(RUNTIME_CLASS(CMyButton)) == TRUE)
                        {
                                CMyButton *pButton = (CMyButton *)pObj;
                                pButton->Create(rect, this);
                        }
                        GetDocument( )->AddControl(pObj);
                }
        }
        Invalidate( );
        return 1;
}
```

그림 16-20 폼뷰에 컨트롤 드래그&드롭

아직 해결되지 않은 것이 있다. 바로 메모리 문제다. 디버그로 FormPad 프로그램을 실행한 후 종료하면, 출력 창에 메모리가 누수되었다는 메시지가 나타난다. 버튼 객체를 생성만 하고 소멸시키지 않았기 때문이다. 이 문제는 CMyButton 클래스에 PostNcDestroy( ) 멤버 함수

를 재정의하여 해결할 수 있다(멤버 함수 재정의를 추가하는 것은 클래스 위저드 등을 활용할 수 있다).

```
void CMyButton::PostNcDestroy( )
{
        delete this;
        CButton::PostNcDestroy( );
}
```

자식 윈도우는 부모 윈도우가 소멸될 때 함께 소멸하므로, 버튼 컨트롤의 윈도우 객체는 소멸시킬 필요 없다. 대신 CMyButton 윈도우가 소멸되면서 PostNcDestroy( ) 멤버 함수가 호출되고, 이때 'delete this;' 문장이 실행되면서 CMyButton 객체가 소멸된다. 따라서 메모리 문제는 자동으로 해결된다.

2.2 이동 지원과 커서 설정

컨트롤 툴바에서 컨트롤을 드래그&드롭할 때처럼, 폼에 올린 버튼도 옮기고 커서도 변경해야 한다. 따라서 CMyButton 클래스에도 마우스 드래그 기능을 수행할 OnLButtonDown( ), OnMouseMove( ), OnLButtonUp( ) 메시지 핸들러 함수 세 개를 구현해야 한다. 우선 CControlToolBar 클래스에 정의되어 있는 커서 식별자를 받아오기 위해 CMainFrame 클래스와 CControlToolBar 클래스에 각각 멤버 함수를 구현한다.

```
class CMainFrame : public CFrameWnd
{
        ...
public:
        CControlToolBar &GetToolBar( )      {return m_wndToolBar;}
        ...
};
```

```
class CControlToolBar : public CToolBar
{
        ...
public:
        UINT GetImageAt(int iIndex)         {return m_CursorArray[iIndex];}
        ...
};
```

또한 CMyButton 클래스도 CControlToolBar 클래스처럼 자신의 인덱스를 저장하는 멤버변수가 필요하다. 다음과 같이 멤버 변수를 선언하고 초기화한다. 그리고 버튼 컨트롤은 컨트롤 툴바에서 10번째에 위치하므로, 9번 인덱스로 정의한다.

```
class CMyButton : public CButton
{
        ...
protected:
        int     m_iIndex;
        BOOL    m_bDragging;
        CPoint  m_ptHotspot;
};

CMyButton::CMyButton( )
        : m_iIndex(9), m_bDragging(FALSE)
{
}
```

m_ptHotspot은 마우스로 클릭한 위치를 저장하는 멤버 변수이다. 컨트롤 툴바는 무조건 중앙에 맞추면 되었지만, 컨트롤에서 이동할 때는 [그림 16-21]처럼 커서 위치가 중심이기 때문이다.

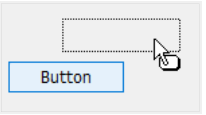


그림 16-21 컨트롤 위치 이동

다음과 같이 OnLButtonDown( ) 멤버 함수를 구현한다. CControlToolBar 클래스에서 구현한 것처럼 커서를 설정하고 마우스 캡처를 실행한다. 그리고 이동할 때 폼 영역 밖으로 벗어나지 않도록 폼 영역을 얻어 커서 이동 범위를 제한(Clipping)한다. CControlToolBar 클래스에서는 위치 정보로 크기를 만들어 보냈다. 그러나 현 컨트롤은 이미 생성된 상태이므로 현재 영역을 그대로 화면 좌표로 변경하여 전송하면 된다. 메시지를 보낼 대상을 CControlToolBar 클래스와 똑같이 CFormPadView 클래스로 맞추면 코드를 재활용할 수 있다. CFormPadView 클래스는 GetParent( ) 함수로 간단히 얻을 수 있다. CMainFrame 클래스와 CControlToolBar 클래스가 사용되므로 MainFrm.h 헤더 파일과 ControlToolBar.h 헤더 파일을 포함시킨다.

```
…
#include "MyButton.h"
#include "MainFrm.h"
#include "ControlToolBar.h"
…
void CMyButton::OnLButtonDown(UINT nFlags, CPoint point)
{
        m_bDragging = TRUE;

        CControlToolBar &wndToolBar
                = ((CMainFrame *)AfxGetMainWnd( ))->GetToolBar( );
        ::SetCursor(AfxGetApp( )->LoadCursor(
                wndToolBar.GetImageAt(m_iIndex)));
        SetCapture( );

        CRect rect;
        GetParent( )->GetClientRect(rect);
        GetParent( )->ClientToScreen(rect);
        ::ClipCursor(rect);

        GetClientRect(rect);
        ClientToScreen(rect);
        m_ptHotspot = point;

        BOOL bSuccess = GetParent( )->SendMessage(
                WM_APP_CONTROL_DOWN,
                (WPARAM)&rect, NULL);
        CButton::OnLButtonDown(nFlags, point);
}
```

OnMouseMove( ) 멤버 함수는 OnLButtonDown( ) 멤버 함수에서 저장한 m_ptHotspot 위치로 영역을 계산하고, 그 값을 CFormPadView 클래스에 메시지로 전송한다.

```
void CMyButton::OnMouseMove(UINT nFlags, CPoint point)
{
	if (m_bDragging == TRUE && nFlags & MK_LBUTTON)
	{
		CRect rect;
		GetClientRect(rect);

		CSize size(point.x - m_ptHotspot.x, point.y - m_ptHotspot.y);
		ClientToScreen(rect);
		rect.OffsetRect(size);

		BOOL bSuccess = GetParent( )->SendMessage(
			WM_APP_CONTROL_DRAG,
			(WPARAM)&rect, NULL);
	}
	CButton::OnMouseMove(nFlags, point);
}
```

OnMButtonUp( ) 멤버 함수는 OnLButtonDown( ) 멤버 함수와 반대로 커서 클리핑, 마우스 캡처, 커서 해제를 수행한다. 그리고 CFormPadView 클래스에 WM_APP_CONTROL_DROP 메시지를 전송한다. 이때 WPARAM에 FALSE를 보내 컨트롤을 생성하는 것이 아님을 명시한다.

```
void CMyButton::OnLButtonUp(UINT nFlags, CPoint point)
{
        if(m_bDragging == TRUE)
        {
                BOOL bSuccess = GetParent( )->SendMessage(
                        WM_APP_CONTROL_DROP,
                        (WPARAM)FALSE, (LPARAM)this);

                ::ClipCursor(NULL);
                ReleaseCapture( );
                ::SetCursor(::LoadCursor(NULL, IDC_ARROW));

                m_bDragging = FALSE;
        }
        CButton::OnLButton(nFlags, point);
}
```

CFormPadView 클래스의 OnControlDrop( ) 멤버 함수에 FALSE 처리가 되어 있지 않으므로 다음과 같이 수정한다. 향후 크기 변경을 고려해 영역의 위치와 크기가 모두 반영하도록 SetWindowPos( ) 함수를 처리한다. 이제 폼 위의 버튼도 자유롭게 이동할 수 있다.

```
LRESULT CFormPadView::OnControlDrop(WPARAM wParam, LPARAM lParam)
{
        ...
        if ( wParam == TRUE)
        {
                ...
        }
        else
        {
                if(rect.IsRectEmpty( ) == FALSE)
                {
                        CWnd *pWnd = (CWnd *)lParam;
                        pWnd->SetWindowPos(NULL, rect.left, rect.top,
                                rect.Width( ), rect.Height( ),
                SWP_NOREPOSITION|SWP_NOZORDER|SWP_SHOWWINDOW);
                }
        }
        Invalidate( );
        return 1;
}
```

2.3 크기 변경 지원

크기 변경 지원도 마우스 왼쪽 버튼이 눌렸을 때 판단하자. 즉, [그림 16-22]와 같이 마우스 왼쪽 버튼이 눌리면 해당 위치가 크기 변경 위치인지, 높이 변경인지, 폭 변경인지 판단하자.

그림 16-22 크기 변경 위치 파악

다음과 같이 CMyButton 클래스에 열거형 변수 m_Resize를 추가한다. NONE은 이동 시, HORZ는 폭 변경, VERT는 높이 변경을 나타낸다. RESIZE_POINT는 테두리부터 크기 변경으로 인정할 폭을 나타낸다.

```
...
#define RESIZE_POINT 5

class CMyButton : public CButton
{
	...
protected:
	enum { NONE, HORZ, VERT, } m_Resize;
};
```

OnLButtonDown() 멤버 함수에서 전달된 위치를 기준으로 영역을 검사하고, 각각 HORZ, VERT, NONE으로 m_Resize 멤버 변수를 초기화한다. 그리고 HORZ와 VERT일 경우 각각 좌우(IDC_SIZEWE), 상하(IDC_SIZENS) 커서로 변경한다.

```
void CMyButton::OnLButtonDown(UINT nFlags, CPoint point)
{
	...
	::ClipCursor(rect);

	GetClientRect(rect);
	CRect rectHORZ(rect), rectVERT(rect);
	rectHORZ.DeflateRect(RESIZE_POINT, 0);
	rectVERT.DeflateRect(0, RESIZE_POINT);
	if (rectHORZ.PtInRect(point) == FALSE)
	{
		::SetCursor(::LoadCursor(NULL, IDC_SIZEWE));
		m_Resize = HORZ;
	}
	else if (rectVERT.PtInRect(point) == FALSE)
	{
		::SetCursor(::LoadCursor(NULL, IDC_SIZENS));
		m_Resize = VERT;
	}
	else
	{
		m_Resize = NONE;
```

```
        }

        GetClientRect(rect);
        ...
}
```

OnLButtonDown() 멤버 함수에서 판단한 자료를 통해 OnMouseMove() 멤버 함수에서 이동과 크기 변경을 반영하여 영역을 설정한다. 기존의 CRect::OffsetRect() 함수를 호출한 부분은 m_Resize가 NONE인 경우로 이동하면 된다.

```
void CMyButton::OnMouseMove(UINT nFlags, CPoint point)
{
        if(m_bDragging == TRUE && nFlags & MK_LBUTTON)
        {
                ...
                CSize size(point.x - m_ptHotspot.x, point.y - m_ptHotspot.y);
                ClientToScreen(rect);
                switch(m_Resize)
                {
                case NONE:
                        rect.OffsetRect(size);
                        break;
                case HORZ:
                        rect.InflateRect(size.cx, 0);
                        break;
                case VERT:
                        rect.InflateRect(0, size.cy);
                        break;
                }
                rect.NormalizeRect( );

                BOOL bSuccess = GetParent( )->SendMessage(
                        WM_APP_CONTROL_DRAG,
                        (WPARAM)&rect, NULL);
        }
        CButton::OnMouseMove(nFlags, point);
}
```

이제 [그림 16-23]과 같이 다양한 크기의 버튼 컨트롤을 생성할 수 있다. 여기서는 크기 변경을 간단히 구현했으나 상하좌우를 분리하여 대각선 크기도 변경하면 좋다.

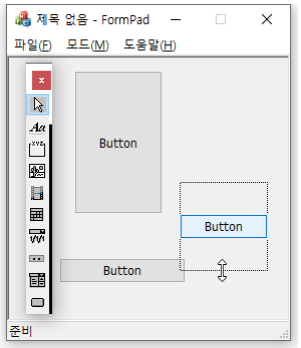


그림 16-23 크기 변경이 가능한 버튼 컨트롤

2.4 공통 클래스 구현

버튼 컨트롤 외에 앞으로 컨트롤을 8개 더 구현해야 하는데, 중복 구현이 상당히 많다. 이럴 때는 루틴을 함수로 제작하는 것처럼, 공통 클래스를 구현해 두면 구현 시간을 절약할 수 있다. 또한 인터페이스 개념을 사용하여 공통 클래스를 구현하면 중요한 사항을 구현하지 않고 잊는 것을 방지할 수 있다. 인터페이스, 즉 순수 가상 함수가 정의되어 있다면 이를 구현하지 않고서는 컴파일되지 않기 때문이다. 따라서 클래스를 상속받을 때 반드시 구현해야 할 함수를 지정함으로써 클래스 설계자의 의도를 그대로 관철시킬 수 있다. 9개 컨트롤 구현에서 공통 내용을 정리하면, 다음 CControlBase 클래스와 같다.

자세히 살펴보면 CMyButton 구현에 쓰인 멤버 변수와 멤버 함수를 모두 포함하고 있다. 따라서 CMyButton 클래스에 구현할 내용뿐만 아니라, 이후 컨트롤 클래스 구현의 부담이 줄어든다. 컨트롤마다 다른 m_iIndex 멤버 변수 값은 생성자를 통해 전달한다. 그리고 m_bRunMode와 m_wndToolBar를 레퍼런스 변수로 선언하고 CMainFrame 내 해당 멤버 변수(m_bRunMode와 m_wndToolBar)와 자동 연결하기 때문에, 이들 멤버 변수를 편리하게 사용할 수 있다.

참고 #pragma once 전처리기 구문은 현재 헤더 파일을 한 번만 포함시키라는 의미로 예전에 #ifndef와 #define을 섞어 쓰던 방식을 개선한 방식이다.

```
#pragma once

class CControlToolBar;

class CControlBase
```

```
{
public:
	CControlBase(int iIndex);
	virtual ~CControlBase( );

public:
	virtual BOOL Create(CRect &rect, CWnd *pParentWnd ) = 0;

protected:
	int		m_iIndex;
	BOOL	m_bDragging;
	CPoint	m_ptHotspot;
	enum { NONE, HORZ, VERT, } m_Resize;

protected:
	BOOL		&	m_bRunMode;
	CControlToolBar	&	m_wndToolBar;
};
```

```
#include "StdAfx.h"
#include "MainFrm.h"
#include "ControlToolBar.h"
#include "ControlBase.h"

CControlBase::CControlBase(int iIndex)
	: m_bRunMode(((CMainFrame *)AfxGetMainWnd( ))->GetMode( )),
	m_wndToolBar(((CMainFrame *)AfxGetMainWnd( ))->GetToolBar( )),
	m_iIndex(iIndex), m_bDragging(FALSE), m_Resize(NONE)
{
}

CControlBase::~CControlBase( )
{
}
```

이제 CMyButton 클래스가 CButton 클래스와 함께 CControlBase 클래스를 상속받도록 몇 가지 사항을 변경한다. 자세한 내용은 소스 파일을 참고하자.

```
#include  "ControlBash.h"
...
class CMyButton : public CButton, public CControlBase
{
        ...
};
```

```
CMyButton::CMyButton( )
        : CControlBase(9)
{
}
```

이렇게 모든 컨트롤을 CControlBase 클래스로 통합하면 모든 컨트롤을 공통 루틴으로 처리할 수 있다는 장점이 있다. 다음을 보자.

```
LRESULT CFormPadView::OnControlDrop(WPARAM wParam, LPARAM lParam)
{
        ...
        CRuntimeClass *pClass = (CRuntimeClass *)lParam;
        CObject *pObj = pClass->CreateObject( );
        if(pObj != NULL)
        {
                if (pObj->IsKindOf(RUNTIME_CLASS(CMyButton)) == TRUE)
                {
                        CMyButton *pButton = (CMyButton *)pObj;
                        CControlBase *pControl = (CControlBase *)pButton;
                        pControl->Create(rect, this);
                }
                GetDocument( )->AddControl(pObj);
        }
        ...
}
```

달라진 부분을 느낄 수 있는가? CMyButton 포인터를 다시 CControlBase 클래스로 받아 Create( ) 함수를 호출하였다. 즉, CObject 포인터와 런타임 클래스만 넘겨주면, CControlBase로 처리를 단순화할 수 있다. ※ 〈여기서 잠깐〉을 참고한다.

여기서 잠깐! **CControlBase 클래스에 의한 코드 간소화 효과**

OnControlDrop( ) 멤버 함수는 컨트롤을 더 구현하면 if 문을 반복해야 하지만, 이 방식을 쓰면 다음과 같이 간단해진다(아직 함수를 구현하지 않았으므로 동작하지 않음).

```
LRESULT CFormPadView::OnControlDrop(WPARAM wParam, LPARAM lParam)
{
        ...
        CRuntimeClass *pClass = (CRuntimeClass *)lParam;
        CObject *pObj = pClass->CreateObject( );
        if(pObj != NULL)
        {
                CControlBase *pControl = GetControlBase(pObj);
                pControl->Create(rect, this);
                GetDocument( )->AddControl(pObj);
        }
        ...
}
```

실제로 GetControlBase( ) 함수를 구현해 보자. CFormPadView 클래스에 구현할 수도 있지만, 컨트롤 관련 정보를 관리하는 CControlToolBar 클래스에 구현하는 것이 더 직관적이다. 현재는 CControlBase를 상속받은 클래스가 CMyButton 밖에 없지만, 이후 다른 컨트롤을 추가할 때 이 함수에 if 문으로 추가 분기하여 해당 컨트롤을 추가하면 된다. 그러면 더 이상 CFormPadView 클래스의 OnControlDrop( ) 멤버 함수는 수정할 필요가 없다.

```
...
class CControlBase;

class CControlToolBar : public CToolBar
{
        ...
public:
        CControlBase *GetControlBase(CObject *pObj );
        ...
};
```

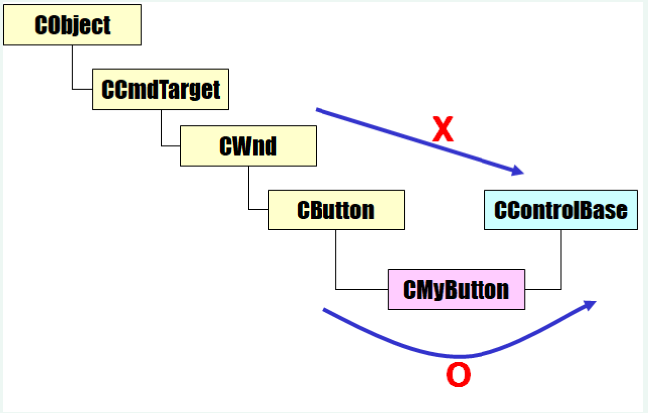
```
...
CControlBase *CControlToolBar::GetControlBase(CObject *pObj)
{
	if (pObj->IsKindOf(m_ClassArray[9]) == TRUE)
	{
		CMyButton *pButton = (CMyButton *)pObj;
		return pButton;
	}
	return NULL;
}
```

```
LRESULT CFormPadView::OnControlDrop(WPARAM wParam, LPARAM lParam)
{
	...
	CRuntimeClass *pClass = (CRuntimeClass *)lParam;
	CObject *pObj = pClass->CreateObject( );
	if (pObj != NULL)
	{
		CMainFrame *pFrame = (CMainFrame *)GetParentFrame( );
		CControlToolBar &wndToolBar = pFrame->GetToolBar( );
		CControlBase *pControl = wndToolBar.GetControlBase(pObj);
		pControl->Create(rect, this);
		GetDocument( )->AddControl(pObj);
	}
	...
}
```

여기서 잠깐! **CMyButton 클래스 거치기**

CObject 객체를 CControlBase 클래스로 직접 캐스팅하지 않고, CMyButton 클래스를 거쳐야 하는 이유가 궁금할 수 있다. 이유는 이들 클래스가 다음과 같은 상속 관계를 가지기 때문이다.

CMyButton 클래스는 다중 상속을 사용하기 때문에 양쪽 줄기는 CMyButton 클래스를 거치지 않고 서로에 접근할 수 없다. 반드시 CMyButton 클래스라는 '연결 고리'를 거쳐야 한다. 다중 상속 문제를 염려하는 독자가 있을지도 모른다. 하지만 CControlBase 클래스는 순수 가상 함수로 이루어진 인터페이스이므로 다중 상속 문제와 관련이 없다.



▲ **클래스 상속 관계**

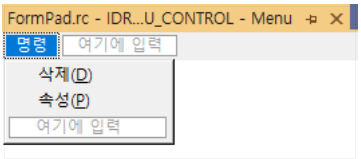
05 속성 대화상자

비주얼 C++에서 대화상자 리소스 템플릿을 편집할 때는 주로 속성 대화상자를 사용한다. FormPad 프로젝트도 편집 모드에서 폼 위의 컨트롤을 속성 대화상자로 관리한다. 그리고 컨트롤은 속성 대화상자를 열 수 있는 컨텍스트 메뉴를 제공한다. 속성 대화상자는 공통이며, 컨트롤 속성 대화상자는 자식 윈도우로 제공된다.

1 컨텍스트 메뉴

컨텍스트 메뉴를 사용해 속성 대화상자를 띄우게 하자. 윈도우 운영체제는 속성 대화상자를 띄워야 할 때 해당 윈도우에 WM_CONTEXTMENU 메시지를 전달한다. 따라서 이 메시지를 처리하여 컨텍스트 메뉴를 띄우면 된다.

먼저 비주얼 C++의 리소스 뷰에서 IDR_MENU_CONTROL 메뉴 리소스를 추가하자. 그리고 [그림16-24]와 같이 메뉴를 구성하고 표를 참고하여 ID를 지정한다.



메뉴	ID	캡션
(0번 메뉴)	(Pop-up 메뉴)	명령
삭제	ID_CONTROL_DELETE	삭제(&D)
속성	ID_CONTROL_PROPERTY	속성(&P)

그림 16-24 컨트롤 컨텍스트 메뉴 구성

CMyButton에서 WM_CONTEXTMENU 메시지를 처리하는 핸들러 함수 OnContextMenu()를 생성한다. 컨텍스트 메뉴에서 선택한 명령은 명령 라우팅에 의해 CFormPadView 클래스에 전달할 수 있다. 그러나 명령 핸들러는 인자가 없는 단순 함수이기 때문에 해당 컨트롤

객체를 전달하기 어렵다. 따라서 객체 정보를 CFormPadView 윈도우의 사용자 데이터 공간에 저장하자. ::SetWindowLongPtr( ) 함수를 GWLP_USERDATA 플래그와 함께 사용하면 된다. 컨텍스트 메뉴는 앞서 작성한 IDR_MENU_CONTROL 메뉴에서 하위 메뉴만 뽑아 사용한다.

```
...
void CMyButton::OnContextMenu(CWnd* pWnd, CPoint point)
{
        CMenu menu;
        menu.LoadMenu(IDR_MENU_CONTROL);
        CMenu *pMenu = menu.GetSubMenu(0);

        ::SetWindowLongPtr(GetParent( )->GetSafeHwnd( ),
                GWLP_USERDATA, (LONG_PTR)this);
        pMenu->TrackPopupMenu(TPM_LEFTALIGN | TPM_RIGHTBUTTON,
                point.x, point.y, GetParent( ));
}
```

버튼 컨트롤을 폼에 드래그&드롭한 후 마우스 오른쪽 버튼을 누르면, [그림 16-25]와 같이 컨텍스트 메뉴를 볼 수 있다. 하지만 아직 명령 핸들러를 구현하지 않아 아무 일도 하지 않는다.

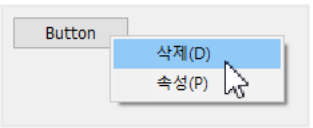


그림 16-25 버튼 컨트롤 컨텍스트 메뉴

그러면 CFormPadView 클래스에 컨텍스트 메뉴의 명령 핸들러를 구현하자. MFC 도큐먼트/뷰 구조가 제공하는 명령 라우팅으로 어디서나 명령 핸들러를 구현할 수 있다. 클래스 마법사를 사용하여 ID_CONTROL_DELETE와 ID_CONTROL_PROPERTY를 처리하는 OnControlDelete( )와 OnControlProperty( ) 명령 핸들러 함수를 생성한다. 그리고 OnControlDelete( ) 멤버 함수에 컨트롤을 삭제하고 CFormPadDoc 도큐먼트의 객체 리스트에서 현재 객체를 제거하는 기능을 구현한다. 다음 코드를 보자.

```
...
void CFormPadView::OnControlDelete( )
{
        CObject *pObj =
                (CObject *)::GetWindowLongPtr(GetSafeHwnd( ),GWLP_USERDATA);
        if (pObj != NULL)
        {
                CWnd *pWnd = (CWnd *)pObj;
                pWnd->ShowWindow(SW_HIDE);
                pWnd->DestroyWindow( );

                GetDocument( )->RemoveControl(pObj);

                ::SetWindowLongPtr(GetParent( )->GetSafeHwnd( ),
                        GWLP_USERDATA, NULL);
        }
}
```

삭제 대상 컨트롤, 즉 컨텍스트 메뉴가 발생한 컨트롤은 ::GetWindowLongPtr( ) 함수로 얻을 수 있다. 앞서 CMyButton 클래스에서 ::SetWindowLongPtr( ) 함수로 저장해 둔 것이다. 컨트롤은 모두 CWnd를 상속받으므로, 해당 객체를 CWnd 클래스로 캐스팅한 후 DestroyWindow( ) 함수를 호출하여 윈도우를 소멸시킬 수 있다. 삭제 전에 ShowWindow( ) 함수로 화면에서 감추면 안정된 효과(종료 직전 화면에 잔상이 남는 현상 제거)를 얻을 수 있다. CFormPadDoc 도큐먼트에 RemoveControl( ) 함수를 실행하여 객체 리스트에서 해당 컨트롤을 제거한다. 그리고 다시 ::SetWindowLongPtr( ) 함수를 호출하여 CFormPadView 윈도우에 저장된 값을 초기화한다.

이렇게 컨트롤의 삭제도 구현하였다. 실제로 속성 대화상자를 띄우는 OnControlProperty( ) 멤버 함수는 속성 대화상자를 생성한 후 구현하자.

2 속성 대화상자 구현

비주얼 C++ 리소스 뷰에서 IDD_CONTROL_PROPERTY 대화상자 템플릿을 생성한다. 그리고 [그림 16-26]과 같은 상태로 변경(Caption 속성을 "속성 대화상자", Border 속성을 'Thin', 그리고 확인, 취소 버튼을 삭제함)한다. 공통으로 사용하는 대화상자일 뿐이므로 다

른 모양이나 스타일로 설정해도 상관없다.

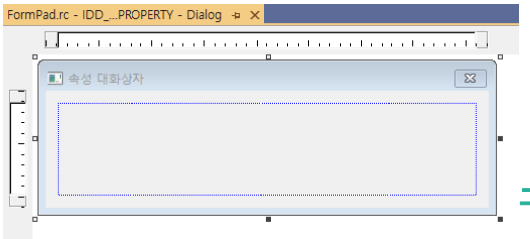


그림 16-26 속성 대화 상자 템플릿

그리고 IDD_CONTROL_PROPERTY 대화상자 템플릿으로 CPropertyDlg 클래스를 생성한다. 속성 대화상자는 클라이언트 영역에 자식 대화상자를 받으므로 다음과 같이 생성자와 멤버 변수로 자식 윈도우를 받도록 설정한다.

```
...
#include "PropertyBase.h"

class CPropertyDlg : public CDialog
{
public:
	CPropertyDlg(CPropertyBase *pDlg, CWnd* pParent = NULL);
	...
protected:
	CPropertyBase	*	m_pChildDlg;
};
```

```
CPropertyDlg::CPropertyDlg(CPropertyBase *pDlg, CWnd* pParent /*=NULL*/)
 : CDialog(CPropertyDlg::IDD, pParent), m_pChildDlg(pDlg)
{
}
```

그런데 아직 구현하지 않은 CPropertyBase 클래스를 언급하고 있다. 이 클래스는 컨트롤 속성 대화상자의 베이스 클래스이다. 앞서 구현한 CControlBase 클래스를 감안하여, 속성 대화상자는 처음부터 여러 컨트롤을 감안하여 설계하자. 일단 CPropertyBase 클래스가 있다고 가정하고 진행한다. 클래스 마법사를 사용하여 WM_CREATE 메시지를 처리하는 OnCreate( ) 멤버 함수를 생성하고, OnInitDialog( ) 가상 멤버 함수를 재정의하여 다음과 같이 구현한다.

```
int CPropertyDlg::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CDialog::OnCreate(lpCreateStruct) == -1)
		return -1;

	m_pChildDlg->Create(this);
	m_pChildDlg->ShowWindow(SW_SHOW);

	return 0;
}

BOOL CPropertyDlg::OnInitDialog( )
{
	CDialog::OnInitDialog( );

	CRect rect;
	GetClientRect(rect);
	m_pChildDlg->SetWindowPos(NULL, rect.left, rect.top,
		rect.Width( ), rect.Height( ),
		SWP_NOZORDER | SWP_SHOWWINDOW);

	return TRUE;
}
```

OnCreate( ) 멤버 함수에서 자식 윈도우를 생성하고, OnInitDialog( ) 멤버 함수에서 크기와 위치를 재조정한다. 아직 속성 대화상자가 화면에 나타나지 않았기 때문에 OnCreate( ) 멤버 함수에서는 정확한 크기나 위치 정보를 얻을 수 없다. 따라서 OnInitDialog( ) 멤버 함수에서 필요한 조정을 한다. 같은 방식으로 자식 대화상자를 동적으로 생성하여 붙이는 기술을 구현한다. OnCreate( ) 멤버 함수를 보면 CPropertyBase 클래스가 Create( ) 함수를 제공한다고 가정하였다. 따라서 다음과 같이 CPropertyBase 클래스를 구성한다.

```
class CPropertyBase : public CDialog
{
public:
	CPropertyBase(UINT nIDTemplate, CWnd* pParent = NULL);
	virtual ~CPropertyBase(void);
```

```
public:
	BOOL Create(CWnd *pParentWnd);

protected:
	virtual void OnOK( );
	virtual void OnCancel( );

private:
	UINT m_nIDTemplate;
};
```

```
#include "StdAfx.h"
#include "PropertyBase.h"

CPropertyBase::CPropertyBase(UINT nIDTemplate, CWnd* pParent /*=NULL*/)
	: CDialog(nIDTemplate, pParent), m_nIDTemplate(nIDTemplate)
{
}

CPropertyBase::~CPropertyBase(void)
{
}

BOOL CPropertyBase::Create(CWnd *pParentWnd)
{
	return CDialog::Create(m_nIDTemplate, pParentWnd);
}

void CPropertyBase::OnOK( )
{
	CDialog *pDlg = (CDialog *)GetParent( );
	pDlg->EndDialog(IDOK);
	CDialog::OnOK( );
}

void CPropertyBase::OnCancel( )
{
```

```
        CDialog *pDlg = (CDialog *)GetParent( );
        pDlg->EndDialog(IDCANCEL);
        CDialog::OnCancel( );
}
```

CPropertyBase 클래스는 m_nIDTemplate 멤버 변수를 비롯해 생성자로 전달된 자식 대화상자의 리소스 템플릿 ID를 보관한다. 그리고 Create( ) 멤버 함수에서 이 값을 이용해 자식 윈도우를 생성한다. 자식 윈도우에서 [Enter]나 [Esc] 키를 입력하면 기본 대화상자 구현에 따라 자식 대화상자가 사라져 버린다. 이 문제를 해결하기 위해 OnOK( )와 OnCancel( ) 가상 함수를 재정의한다. 그리고 키 입력을 받아 자식 대화상자가 사라질 때 부모 윈도우도 함께 사라지도록 EndDialog( ) 함수를 호출한다.

이제 버튼 컨트롤의 속성 대화상자를 제작하여 속성 대화상자를 띄워 보자. 비주얼 C++의 리소스 뷰에서 [그림 16-27]과 같이 IDD_PROPERTY_BUTTON 대화상자 템플릿을 생성한다. 편집 컨트롤로 캡션과 파일 경로를 입력하며, 라디오 버튼 컨트롤로 동작을 지정하도록 되어 있다. 속성 대화상자는 필요에 따라 구현할 수 있으므로 구현 내용은 소스 파일을 참고하자. 중요한 것은 대화상자 스타일을 반드시 자식(Child)으로 선언한다는 것이다. 그래야 속성 대화상자 클라이언트 영역에 깔끔하게 올라간다. 그리고 Border 속성을 없음(None)으로 설정하고, 속성 적용을 위해 ID가 IDOK인 버튼 컨트롤도 넣어준다.

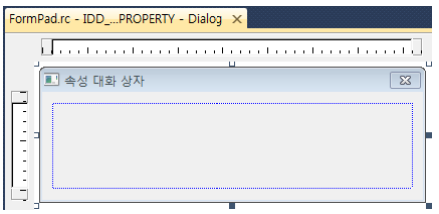


그림 16-27 버튼 컨트롤 속성 대화상자의 대화상자 템플릿

클래스 마법사를 사용하여 IDD_PROPERTY_BUTTON 대화상자 템플릿을 CPropertyButton 클래스로 구현한다. 상속할 클래스 목록에 CPropertyBase 클래스가 없으므로 CDialog 클래스를 선택한 후 CPropertyBase 클래스로 변경하면 된다. 구현은 일반 대화상자 응용 프로그램과 동일하므로 이것도 구현 내용은 소스 파일을 참고하자. 단, 통일할 것은 DDX를 사용하여 컨트롤 변수를 사용하는 것이다. 뒤에서 CMyButton 클래스가 이들 변수와 연결하게 된다.

```
#pragma once
#include "PropertyBase.h"

class CPropertyButton : public CPropertyBase
{
        ...
public:
        CString   m_strCaption;
        CString   m_strPath;
        int       m_iAction;
};
```

3 컨트롤 속성 구현

CControlBase 클래스에 순수 가상 함수 두 개를 추가한다. 하나는 GetProperyPage( ) 함수로 CPropertyBase 포인터를 건네주고, 다른 하나는 ReflectPropery( ) 함수로 컨트롤에 변경된 속성을 적용한다.

```
#pragma once
#include "PropertyBase.h"
...
class CControlBase
{
        ...
public:
        virtual BOOL Create(CRect &rect, CWnd *pParentWnd) = 0;
        virtual CPropertyBase *GetProperyPage( ) = 0;
        virtual void ReflectPropery( ) = 0;
        ...
}
```

CControlBase 클래스에 추가된 함수는 순수 가상 함수므로 이를 상속받는 CMyButton 클래스에서 반드시 구현해야 한다. 구현은 다음과 같다.

```
#pragma once
#include "ControlBase.h"
#include "PropertyButton.h"
...
class CMyButton : public CButton, public CControlBase
{
        ...
protected:
        CPropertyBase *GetProperyPage( ) { return &m_Property; }
        void ReflectPropery( );

protected:
        CPropertyButton   m_Property;
        CString   &       m_strCaption;
        CString   &       m_strPath;
        int       &       m_iAction;
};
```

```
...
CMyButton::CMyButton( )
        : CControlBase(9)
        , m_strCaption(m_Property.m_strCaption)
        , m_strPath(m_Property.m_strPath)
        , m_iAction(m_Property.m_iAction)
{
        m_strCaption = _T("Button");
}
...
void CMyButton::ReflectPropery( )
{
        SetWindowText(m_strCaption);
}
```

먼저 CMyButton 클래스는 버튼 클래스의 속성 대화상자인 CPropertyButton 객체 m_Property를 멤버로 포함하고 있다. 따라서 GetProperyPage( ) 함수가 호출되면 이 객체의 포인터를 전송한다. 또한 CPropertyButton 클래스에 있는 속성 변수를 CMyButton 클래스의 멤버 변수로 그대로 포함한다. 다만 다른 점은 CMyButton 클래스의 멤버는 레퍼런

스로 선언하고, 생성자에서 CPropertyButton 클래스에 있는 속성 변수로 초기화한다는 것이다. 따라서 이 들은 자동으로 연결되며, 생성자에서 m_strCaption 값을 설정하면 속성 대화상자에도 반영된다. 그리고 ReflectPropery() 함수가 호출되면 속성 멤버 변수를 참고하여 필요한 설정을 수행한다. CMyButton 클래스는 버튼 위 글자를 변경하는 작업을 한다. 속성 대화상자에서 DDX를 사용하면 기본 구현에서 〈확인(OK)〉 버튼을 누를 때만 속성 변수에 반영된다. 따라서 ReflectPropery() 멤버 함수는 어떤 확인 없이 속성 멤버 변수를 그대로 참고할 수 있다.

이제 CFormPadView 클래스의 OnControlProperty() 멤버 함수에서 속성 대화상자를 띄울 수 있다. 그런데 CPropertyBase 베이스 클래스를 구현함에 따라 속성 대화상자를 다음처럼 if 문 하나로만 간단히 구현할 수 있다. 이전에 작성한 GetControlBase() 함수도 큰 역할을 하고 있다. 이에 따라 CFormPadView 클래스 입장에서는 CControlBase와 CPropertyBase 클래스만으로 속성 대화상자를 띄우는 것과 같다.

참고 속성 대화상자를 직접 보면 자식대화상자가 클라이언트 영역에 동적으로 생성되어 놓여 있지만 전혀 눈치챌 수 없다. 이렇게 윈도우 사용자 인터페이스는 눈속임(?)에 불과하다. 수많은 자식 대화상자가 겹쳐진 윈도우를 봐도 아무도 이 사실을 눈치채지 못하는 경우가 많다.

```
...
#include "MyButton.h"
#include "PropertyDlg.h"
...
void CFormPadView::OnControlProperty( )
{
        CObject *pObj
                = (CObject *)::GetWindowLongPtr(GetSafeHwnd( ),GWLP_USERDATA);
        if(pObj != NULL)
        {
                CMainFrame *pFrame = (CMainFrame *)GetParentFrame( );
                CControlToolBar &wndToolBar = pFrame->GetToolBar( );
                CControlBase *pControl = wndToolBar.GetControlBase(pObj);

                CPropertyDlg dlg(pControl->GetProperyPage( ));
                int nResponse = dlg.DoModal( );
                if (nResponse == IDOK)
                        pControl->ReflectPropery( );
```

```
                ::SetWindowLongPtr(GetParent()->GetSafeHwnd(),
                GWLP_USERDATA, NULL);
        }
}
```

앞서 살펴본 OnControlDelete() 멤버 함수처럼 OnControlProperty() 멤버 함수도 ::GetWindowLongPtr() 함수를 사용하여 컨트롤 객체를 얻는다. 그리고 CPropertyDlg 클래스는 생성자로 전달받은 CPropertyBase 포인터를 사용하여 [그림 16-28]과 같이 버튼 컨트롤의 속성 대화상자를 보여준다.

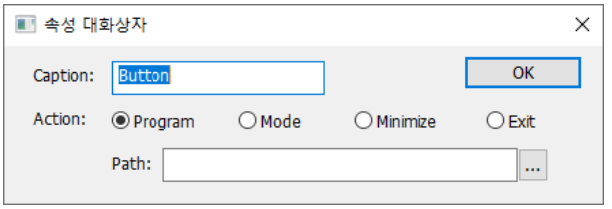


그림 16-28 버튼 컨트롤 속성 대화상자

06 실행 모드 지원

편집 모드와 실행 모드가 다르게 동작해야 하는 경우가 많다. 예를 들어 버튼 컨트롤을 마우스로 클릭하는 경우를 생각할 수 있다. 편집 모드에서는 이동 또는 크기 변경으로 연결되지만, 실행 모드에서는 버튼 고유의 클릭 기능이 수행되어야 한다. 따라서 같은 함수라도 두 가지 모드에 대해 다르게 처리하는 경우가 있다. 그 외 실행 모드로 진입하면 그 시점부터 동작을 실행해야 하는 경우가 있다. 예를 들어 날짜 시간 선택기 컨트롤이 실행 모드에서 시계로 동작한다고 하자. 그러면 편집 모드에서는 정지해 있다가 실행 모드가 되면 시계로 작동해야 한다. 따라서 이렇게 실행 모드로 진입한다고 알려주는 인터페이스가 필요하다.

1 편집 모드와 구분 동작

편집 모드와 실행 모드는 이미 CMainFrame 클래스의 m_bRunMode 멤버 변수를 통해 구분되고 있다. 다만 고려하지 않은 것은, 지금까지 편집 모드에 집중하였기 때문에, 편집 모드에서 실행 모드로 변경되었을 때가 문제다. 따라서 편집 모드와 실행 모드가 같이 사용하는 함수는 분기 처리를 해야 한다. 그런데 이미 CControlBase 클래스가 m_bRunMode 멤버 변수를 제공한다는 사실을 기억하는가? 이 멤버 변수는, 컨트롤이 CMainFrame 클래스의 m_bRunMode 멤버 변수를 참고하기 쉽도록, CControlBase 클래스가 제공하는 것이다. 이에 따라 컨트롤 클래스는 편집 모드와 실행 모드가 함께, 또는 개별적으로 사용하는 함수에 대해 다음과 같은 방식으로 처리한다.

```
void CMyButton::OnLButtonDown(UINT nFlags, CPoint point)
{
        if(m_bRunMode == TRUE)
        {
                … // 실행 모드 처리 루틴
        }
        else
        {
```

```
            … // 편집 모드 처리 루틴
        }
}
```

CMyButton 클래스에서 편집 모드와 실행 모드에서 모두 사용하는 함수는 OnLButtonDown(), OnMouseMove(), OnLButtonUp() 멤버 함수가 있고, 편집 모드에서만 실행되어야 하는 OnContextMenu() 함수가 있다. 특히 버튼 컨트롤의 경우, 별도의 실행보다 실행 모드에서 버튼이 눌렸을 때 동작하므로, 다음과 같이 OnLButtonDown() 멤버 함수에 동작 코드를 넣을 수 있다.

```
void CMyButton::OnLButtonDown(UINT nFlags, CPoint point)
{
        if(m_bRunMode == TRUE)
        {
                switch(m_iAction)
                {
                case 0: // Program
                        ::ShellExecute( NULL, _T("open"), m_strPath,
                        NULL, NULL, SW_SHOWNORMAL );
                        break;
                case 1: // Mode
                        ::AfxGetMainWnd( )->PostMessage( WM_COMMAND,
                        MAKEWPARAM(ID_MODE_EDIT, 0), NULL );
                        break;
                case 2: // Minimize
                        ::AfxGetMainWnd( )->CloseWindow( );
                        break;
                case 3: // Exit
                        ::AfxGetMainWnd( )->PostMessage( WM_COMMAND,
                                MAKEWPARAM(ID_APP_EXIT, 0), NULL );
                        break;
                }
        }
        else
        {
                m_bDragging = TRUE;
```

```
                ...
        }
        CButton::OnLButtonDown(nFlags, point);
}
```

[그림 16-29]는 프로그램을 실행하도록 편집된 버튼 컨트롤이, 실제로 실행 모드에서 메모장을 실행한 모습을 보여주고 있다. 이러한 방식으로 FormPad 프로그램은 아이디어에 따라 다양한 경우를 만들어 내고 조합할 수 있다.

참고 WM_LBUTTON DOWN 메시지 대신 BN_CLICKED 버튼 컨트롤 통지 메시지를 처리해도 된다.

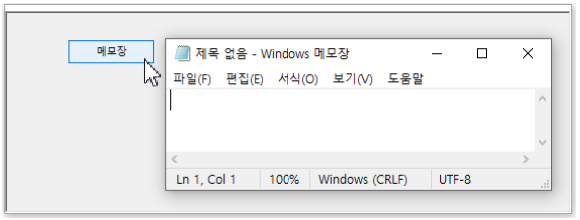


그림 16-29 실행 모드에서 메모장을 실행하는 버튼 컨트롤

2 실행 모드 라우팅

편집 모드와 실행 모드 분기만으로 실행 모드를 완성할 수 없는 경우, 실제로 실행 모드가 되었다고 알려주는 인터페이스가 필요하다. 이를 위해 CControlBase 클래스에, 다음과 같이 순수 가상 함수를 추가하자.

```
class CControlBase
{
        ...
public:
        virtual BOOL Create(CRect &rect, CWnd *pParentWnd) = 0;
        virtual CPropertyBase *GetProperyPage( ) = 0;
        virtual void ReflectPropery( ) = 0;
        virtual void Action( ) = 0;
        ...
};
```

```
class CMyButton : public CButton, public CControlBase
{
        ...
protected:
        void Action( ) {};
};
```

CControlBase 클래스에 순수 가상 함수가 추가되었으므로, 이를 상속받는 CMyButton 클래스는 반드시 이 함수를 구현해야 한다.

이제 실행 모드가 될 때, 컨트롤마다 Action( ) 함수를 호출해 줄 곳을 찾아보자. 실행 모드로 변경하는 장소가 CMainFrame 클래스의 OnModeRun( ) 멤버 함수이므로, 여기서 등록되어 있는 모든 컨트롤에 Action( )을 라우팅(Routing)하는 것이 가장 좋다. 다음과 같은 구문을 추가한다. CFormPadDoc 클래스를 사용하므로 FormPadDoc.h 헤더를 포함시켜야 한다.

이로써 실행 모드를 위한 준비가 끝났다.

```
...
₩#include "MainFrm.h"
#include "FormPadDoc.h"
#include "ControlBase.h"
...
void CMainFrame::OnModeRun( )
{
        ...
        CFormPadDoc *pDoc = (CFormPadDoc *)GetActiveDocument( );
        CObList &list = pDoc->GetControlList( );
        POSITION pos = list.GetHeadPosition( );
        while (pos != NULL)
        {
                CObject *pObj = list.GetNext(pos);
                CControlBase *pControl = m_wndToolBar.GetControlBase(pObj);
                pControl->Action( );
        }
}
```

참고 예를 들면 애니메이션 컨트롤은 실행 모드로 진입하면 재생을 시작해야 하는데, 그 재생 명령을 내릴 곳이 필요한 것이다.

참고 버튼 컨트롤의 경우, Action( ) 함수에 따라 수행할 일이 없으므로 그냥 껍데기 함수로 구현하면 된다. 그러나 애니메이션이나 날짜 시간 선택기 컨트롤은 이 함수를 요긴하게 사용할 것이다.

여기서 잠깐! Action( ) 함수의 반대 역할을 하는 Stop( ) 함수 추가

Action( ) 함수와 반대로, 실행을 정지시키는 Stop( ) 함수가 필요하다. 이 함수도 Action( ) 함수와 같이 정의하고, OnModeEdit( ) 멤버 함수에 추가한다. 또한 CMainFrame의 DestroyWindow( ) 멤버 함수에서 OnModeEdit( ) 함수를 호출하여 종료 전 정지 처리를 수행했다.

```
class CControlBase
{
public:
        ...
        virtual void Action( ) = 0;
        virtual void Stop( ) = 0;
        ...
};
```

```
class CMyButton : public CButton, public CControlBase
{
        ...
protected:
        void Action( )      { }
        void Stop( )        { };
        ...
};
```

```
class CMainFrame : public CFrameWnd
{
        ...
public:
        virtual BOOL DestroyWindow( );
};
```

```
BOOL CMainFrame::DestroyWindow( )
{
	// 종료 전 정지모드 라우팅
	m_bRunMode = TRUE;
	OnModeEdit( );
	return CFrameWnd::DestroyWindow( );
}
```

```
void CMainFrame::OnModeEdit( )
{
	...
	CFormPadDoc *pDoc = (CFormPadDoc *)GetActiveDocument( );
	CObList &list = pDoc->GetControlList( );
	POSITION pos = list.GetHeadPosition( );
	while(pos != NULL)
	{
		CObject *pObj = list.GetNext(pos);
		CControlBase *pControl = m_wndToolBar.GetControlBase(pObj);
		pControl->Stop( );
	}
}
```

07 기타 컨트롤 구현

FormPad 프로젝트는 컨트롤 툴바를 통해 컨트롤을 9개 제공한다. 이전까지 버튼 컨트롤을 중심으로 필요한 기능을 구현하였다. 이제 버튼 컨트롤에 구현한 기능을 다른 컨트롤에 구현하는 작업을 한다. 컨트롤마다 특징적인 것만 언급하니 자세한 사항은 소스 파일을 참고하자.

1 컨트롤 클래스 구현

CControlToolBar 클래스의 생성자에서 실행 시간 클래스를 재정의하고 열거된 클래스를 모두 구현한다(단, 0번은 선택 버튼이므로 항상 눌린 상태로 존재하므로 구현할 필요가 없다). CMyButton 클래스처럼 CControlBase 클래스를 상속받고 공통된 부분을 구현하는데, CMyButton 클래스와 다르게 구현할 사항은 [표 16-4]에 정리하였다.

```
CControlToolBar::CControlToolBar( )
{
        ...
        m_ClassArray[0] = RUNTIME_CLASS(CStatic);
        m_ClassArray[1] = RUNTIME_CLASS(CMyStaticText);
        m_ClassArray[2] = RUNTIME_CLASS(CMyGroupBox);
        m_ClassArray[3] = RUNTIME_CLASS(CMyPicture);
        m_ClassArray[4] = RUNTIME_CLASS(CMyAnimate);
        m_ClassArray[5] = RUNTIME_CLASS(CMyMonthCal);
        m_ClassArray[6] = RUNTIME_CLASS(CMyDateTime);
        m_ClassArray[7] = RUNTIME_CLASS(CMyIPAddress);
        m_ClassArray[8] = RUNTIME_CLASS(CMyComboBox);
        m_ClassArray[9] = RUNTIME_CLASS(CMyButton);
}
```

표 16-4 컨트롤 클래스 특징

컨트롤	인덱스	중요 스타일	기타
정적 텍스트	1	SS_NOTIFY	–
그룹 박스	2	BS_OWNERDRAW BS_NOTIFY	DrawItem( ) 함수 재정의
그림	3	SS_BITMAP SS_ENHMETAFILE SS_NOTIFY	기본 비트맵 제공 비트맵 파일 읽기 루틴
애니메이션	4	WS_BORDER	CFormPadView에서 OnLButtonDown( ) 처리
월력	5	MCS_NOTODAY	–
날짜 시간 선택기	6	DTS_TIMEFORMAT	–
IP 주소	7	–	–
콤보 박스	8	CBS_SIMPLE	–
버튼	9	BS_PUSHBUTTON	편집 모드 때 OnLButtonDblClk( ) 막음

편집 모드에서 이동과 크기 변경에 중요한 마우스 메시지를 주지 않는 컨트롤이 있다. 이 경우 다음과 같이 특별히 처리해야 한다.

- **정적 컨트롤** : SS_NOTIFY 스타일을 준다.
- **그룹 박스 컨트롤** : BS_NOTIFY 스타일을 주어도 해결되지 않는다. 반드시 BS_OWNERDRAW 스타일과 함께 사용해야 한다. 이 경우 DrawItem( ) 함수를 재정의하여 직접 그룹 박스를 그려야 한다.
- **그림 컨트롤** : SS_NOTIFY 스타일을 준다.
- **애니메이션 컨트롤** : 아무 메시지도 주지 않는다. 마우스 메시지가 폼에 그대로 전달된다. 따라서 CFormPadView의 OnLButtonDown( )와 OnContextMenu( ) 멤버 함수에서 처리한다.

그림 컨트롤은 비트맵 파일을 읽는 루틴을 구현해야 한다. 코드구루(www.codeguru.com)나 코드프로젝트(www.codeproject.com) 웹사이트에서 이를 위한 소스 코드를 구할 수 있다. 소스 파일로 제공하는 것은 코드구루 사이트에서 받은 함수를 그대로 사용한 것인데, 이를 통해 소스 코드를 가져다 쓰는 방법에도 익숙해지자.

버튼 컨트롤은 편집 모드에서 마우스 왼쪽 버튼 더블 클릭을 막지 않으면 버튼이 눌리는 현상이 발생한다. 따라서 다음과 같이 편집 모드에서 더블 클릭 실행을 제한해야 한다.

```
void CMyButton::OnLButtonDblClk(UINT nFlags, CPoint point)
{
        if (m_bRunMode == TRUE)
                CButton::OnLButtonDblClk(nFlags, point);
}
```

특별히 애니메이션 컨트롤은 CFormPadView 클래스가 WM_LBUTTONDOWN 메시지를 받아 애니메이션 컨트롤에 알려주어야 하므로, 다음과 같이 OnLButtonDown() 함수를 구현한다.

```
void CFormPadView::OnLButtonDown(UINT nFlags, CPoint point)
{
        CMainFrame *pFrame = (CMainFrame *)GetParentFrame();
        if(pFrame->GetMode() == TRUE)
        {
                ...
        }
        else
        {
                CObList list;
                GetDocument()->LookUpControl(
                        list, RUNTIME_CLASS(CMyAnimate));

                POSITION pos = list.GetHeadPosition();
                while(pos != NULL)
                {
                        CMyAnimate *pAnimate
                                = (CMyAnimate *)list.GetNext(pos);
                        CRect rect;
                        pAnimate->GetClientRect(rect);
                        pAnimate->ClientToScreen(rect);
                        ScreenToClient(rect);
                        if (rect.PtInRect(point) == TRUE)
```

```
                        {
                                ClientToScreen(&point);
                                pAnimate->OnLButtonDown(nFlags, point);
                                return;
                        }
                }
        }
        CFormView::OnLButtonDown(nFlags, point);
}
```

CFormPadDoc 도큐먼트 클래스에 등록된 객체 중 CMyAnimate 클래스를 찾은 후, 현 위치가 애니메이션 컨트롤에 해당되는지 검사한다. 그리고 위치 정보를 해당 애니메이션 컨트롤에 전달한다. 애니메이션 컨트롤의 OnLButtonDown() 멤버 함수는 변경할 필요 없다. 또한 OnLButtonDown() 멤버 함수가 마우스 캡처 방식을 사용하므로 WM_MOUSEMOVE나 WM_LBUTTONUP 메시지는 자동으로 애니메이션 컨트롤에 전달된다. CFormPadDoc 클래스에서 해당 객체를 찾는 LookUpControl() 함수는 다음과 같다.

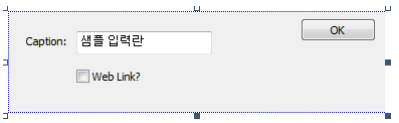
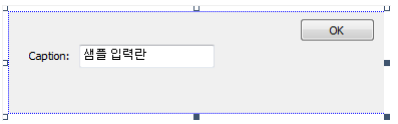
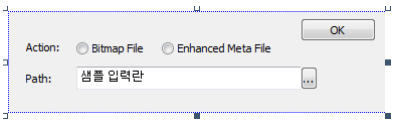
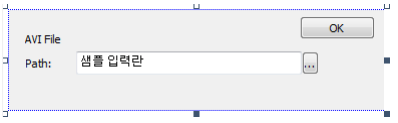
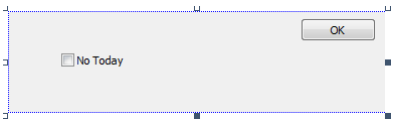
참고 OnContextMenu( ) 멤버 함수도 OnLButtonDown( ) 함수와 유사한 방식으로 구현한다.

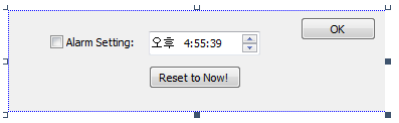
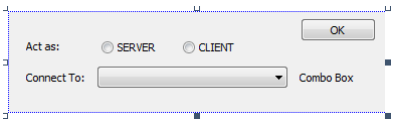
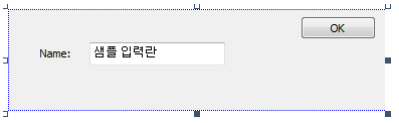
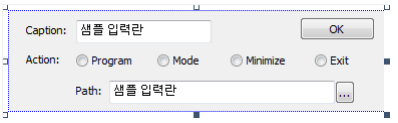
```
class CFormPadDoc : public CDocument
{
public:
        void LookUpControl(CObList &list, CRuntimeClass *pClass )
        {
                list.RemoveAll( );
                POSITION pos = m_listControl.GetHeadPosition( );
                while(pos != NULL)
                {
                        CObject *pObj = m_listControl.GetNext(pos);
                        if(pObj->IsKindOf(pClass) == TRUE)
                                list.AddTail(pObj);
                }
        };
        ...
};
```

2 속성 대화상자

각 컨트롤의 속성 대화상자가 포함하는 요소와 레이아웃은 [표 16-5]와 같다. 이와 똑같이 속성을 구현할 필요는 없고 소스 코드가 이 내용으로 구성되어 있다. 대부분 일반 대화상자 프로그래밍과 동일하다. 단, IP 주소 컨트롤은 현재 객체 리스트에 있는 콤보 박스 목록을 얻고, 이들 이름(Name) 속성으로 콤보 박스 목록을 채워야 한다. 이때 앞서 살펴본 CFormPadDoc 클래스의 LookUpControl( ) 함수를 활용한다.

표 16-5 속성 대화상자 템플릿과 구성요소

컨트롤	속성 대화상자	
정적 텍스트	템플릿	
	구성요소	캡션(편집 컨트롤) 링크(체크 박스)
그룹 박스	템플릿	
	구성요소	캡션(편집 컨트롤)
그림	템플릿	
	구성요소	비트맵/EMF 선택(라디오 버튼) 경로 지정(편집 컨트롤, 버튼)
애니메이션	템플릿	
	구성요소	경로 지정(편집 컨트롤, 버튼)
월력	템플릿	
	구성요소	오늘 없음(체크 박스)

날짜 시간 선택기	템플릿	
	구성요소	알람 설정(체크 박스, 날짜 시간 선택기) 리셋(버튼)
IP 주소	템플릿	
	구성요소	서버/클라이언트 선택(라디오 버튼) 연결 대상(콤보 박스)
콤보 박스	템플릿	
	구성요소	이름(편집 컨트롤)
버튼	템플릿	
	구성요소	캡션(편집 컨트롤) 프로그램/모드/최소화/종료(라디오 버튼) 경로 지정(편집 컨트롤, 버튼)

3 실행 모드 지원

각 컨트롤의 실행 모드 동작은 [표 16-6]과 같다. 물론 열거된 동작 외에 속성 대화상자와 컨트롤 클래스 구현에 따라 더 많고 다양해질 수 있다. 아울러 [표 16-6]을 보면 컨트롤의 시작점을 확인할 수 있다. 앞서 언급한 대로 크게 사용자 이벤트로 작동하는 경우와 Action() 함수로 동작하는 경우로 나눌 수 있다. 사용자 입력을 처리하는 컨트롤은 해당 순간에 동작할 수 있지만, 그 외 컨트롤은 다른 시작점이 필요하기 때문이다. 필요에 따라 방식을 선택하여 활용할 수 있다.

표 16-6 컨트롤 실행 모드의 동작

컨트롤	동작	시작점
정적 텍스트	레이블 역할 하이퍼링크 동작	마우스 왼쪽 버튼 클릭
그룹 박스	범주 레이블 역할	
그림	그림(비트맵, 메타 파일) 출력	
애니메이션	AVI 파일 재생 (무한 반복)	Action( ) 함수
월력	달력	
날짜 시간 선택기	시계, 알람 시계	Action( ) 함수
IP 주소	소켓 통신(서버, 클라이언트)	Action( ) 함수
콤보 박스	채팅(대화가 목록에 추가)	키보드 이벤트
버튼	프로그램 실행, 모드 전환, 최소화, 종료	마우스 왼쪽 버튼 클릭

08 결과 토의

윈도우 프로그래밍 분야는 매우 다양하지만 기본 윈도우 처리는 공통적인 주제이다. 기사나 도서에서는 부분적인 윈도우 처리를 다루고 있어, 독자 스스로 전체를 집대성한 예제를 찾아야 한다. FormPad 프로젝트는 이러한 부족함을 보충하는 하나의 기회가 될 것이다. 더욱이 FormPad 프로젝트를 바탕으로 더 다양한 것을 실험할 수 있다. 지면 제약으로 많은 부분을 생략했는데, 그 중 하나가 직렬화 처리이다. 컨트롤에 DECLARE_SERIAL() 매크로와 Serialize() 함수까지 선언해 두고 설명하지 못했다. 직렬화 기능을 구현해보면, MFC 도큐먼트/뷰 구조의 큰 장점을 느낄 수 있다. 따라서 직렬화를 비롯해 다음에 필자가 제시하는 몇 가지 과제뿐만 아니라 다양한 것을 발굴하여 구현해 보자. 설명을 열 번 듣는 것보다 한 번 해보는 것이 이해도 빠르고 오래 남는다. 이 책에서 다루는 것만으로 구현할 수 있는 프로그램은 수 없이 많다. 관심을 갖고 노력하기 바란다.

참고 소스 파일로 제공된다.

- IP 주소 컨트롤의 양방향 통신 지원(부록* : 버전 1.0)
- 직렬화를 통한 파일 열기와 저장 기능(부록* : 버전 1.1)
- 클립보드를 이용한 편집 기능(Undo, Redo, Copy, Cut, Paste 등)
- 트레이 아이콘으로 실행되는 기능
- 인쇄 기능
- 사용자 컨트롤 제작과 활용
- 버튼 컨트롤에 비트맵 배경 지원
- 마우스 드래그로 다수 컨트롤 선택과 이동
- 콤보 박스 컨트롤의 파일 전송 지원
- IP 주소 컨트롤의 일 대 다 통신 지원