

WEEK

1

# DAY 3

## 마우스와 키보드 입력의 처리

만들고 있는 애플리케이션 중에는, 사용자가 가지고 노는 마우스의 움직임을 잡아내어 처리할 필요성이 있는 부분이 분명히 있다. 언제 어디서 어떤 마우스 버튼이 클릭되었는지, 그리고 언제 마우스 버튼이 떼어졌는지 등을 알아내어야 한다. 심지어는 마우스 버튼이 눌린 동안 사용자가 어떤 동작을 했는가도 프로그래밍을 할 때는 꼭 필요한 정보이다. 프로그래밍을 하면서 필요한 또 한 가지는 키보드 이벤트를 읽는 것이다. 마우스와 마찬가지로, 언제 키가 눌렸고, 얼마나 오래 눌렸고, 또 언제 떼어졌는지를 알아내어야 한다.

이번 장에서는 바로 이것에 필요한 공부를 해보도록 하자.

- ❖ 어떤 마우스 이벤트를 사용할 수 있는지와 어떤 것이 애플리케이션의 필요에 적합한지 결정하는 방법
- ❖ 마우스 이벤트를 잡아 비주얼 C++ 애플리케이션에서 처리하는 방법
- ❖ 사용할 수 있는 키보드 이벤트와 각 이벤트를 발생시키는 동작
- ❖ 키보드 이벤트를 잡은 후 사용자가 누른 키에 대응하여 동작을 처리하기

## 마우스 이벤트에 대하여

전 장에서 언급했지만, 대부분의 컨트롤을 가지고 작업할 때 클래스위저드를 사용하면, 한정된 개수의 이벤트만을 선택할 수 있었다. 마우스 이벤트의 경우에는 어떨까? 역시 클릭과 더블클릭 이벤트 정도만 나와있을 뿐이다. 마우스를 가지고 놀며 잘 생각해 보자. 어디 발생할 수 있는 메시지가 어찌 이 두 개뿐이겠는가? 오른쪽 마우스 버튼은 또 어떻게 감지할 수 있을까? 그럼 그리기 프로그램을 만든다고 하면, 마우스를 드래그할 때 어떻게 제대로 따라오게 할 수 있을까?

여러분의 프로젝트에서 클래스위저드를 열고, Object IDs에서 다이얼로그를 선택한 다음 Messages 리스트 박스를 스크롤해 보면 여러 가지의 메시지들이 훑휙 지나가는데, 이 중에서 마우스에 관련된 메시지를 추가로 더 찾을 수 있다. [표 3.1]에 정리해 두었으며, 이들 메시지를 여러분의 애플리케이션에서 사용해서 어떤 특수한 동작을 할 수 있도록 꾸밀 수 있는 것이다.

[표 3.1] 마우스 이벤트 메시지

메시지	설명
WM_LBUTTONDOWN	왼쪽 마우스 버튼이 눌렸다
WM_LBUTTONUP	왼쪽 마우스 버튼이 떼였다.
WM_LBUTTONDOWNDBLCLK	왼쪽 마우스 버튼이 더블클릭되었다
WM_RBUTTONDOWN	오른쪽 마우스 버튼이 눌렸다
WM_RBUTTONUP	오른쪽 마우스 버튼이 떼였다
WM_RBUTTONDOWNDBLCLK	오른쪽 마우스 버튼이 더블클릭되었다
WM_MOUSEMOVE	마우스 커서가 애플리케이션 윈도우 공간 위를 이동하고 있다
WM_MOUSEWHEEL	마우스 휠이 움직이고 있다

## ■ 마우스로 그림 그리기

이번 장에서 만들어 볼 프로그램은 다이얼로그 윈도우에 마우스로 그림을 그릴 수 있는 정말 간결한 그리기 프로그램이다. 이 애플리케이션은 프로그램 처리의 대부분을 WM\_MOUSEMOVE 이벤트 메시지에 의존하는데, 즉 마우스가 이동하는 도중에 어떤 동작을 걸어 처리하겠다는 뜻이다. 이 프로그램을 만들어 보면서, 여러분은 이 이벤트가 발생할 때 호출되는 함수 안에서 어떻게 왼쪽 마우스 버튼이 눌렸는지와 떼였는지 구분할 수 있는지를 지켜보게 될 것이며, 마우스가 윈도우 위의 어느 위치에 떠있는지도 알아내는 방법도 배울 수 있을 것이다. 쇠뿔도 단김에 빼했다고 그럼 지금 시작해 보기로 하자.

Project Name : Mouse

1. Mouse란 이름의 프로젝트를 MFC AppWizard(exe) 모드로 시작하자.
2. 애플리케이션 위저드의 첫번째 단계에서 다이얼로그 기반 애플리케이션을 선택하자.
3. 두번째 단계에서는 다이얼로그의 타이틀을 Mouse and Keyboard로 바꾸어 주고 넘어가자.
4. 애플리케이션 골격이 만들어진 다음에는 다이얼로그 윈도우에 있는 모든 컨트롤을 없앤다. 전체 다이얼로그 윈도우를 그림을 그리는 화폭으로 만들기 위해서이다. 이 단계는 여러분의 애플리케이션에서 키보드 이벤트를 잡아내기 위해서도 매우 중요하다.

### Note

만약 다이얼로그 윈도우 위에 컨트롤이 하나라도 있을 경우에는 키보드 이벤트가 모두 입력 포커스를 가지고 있는 컨트롤(강조되었거나 커서가 위치해 있는 컨트롤)로 몰려가기 때문에 키보드 이벤트를 잡아내기가 힘들다. 따라서 컨트롤을 모두 없애 두어야 한다.

5. 클래스위저드를 연다. 메시지 리스트에서 WM\_MOUSEMOVE를 선택한 다음, Add Function 버튼을 클릭해서 이 메시지가 발생할 때 호출될 함수를 추가한다. OK 버튼을 클릭해서 클래스위저드가 정성껏 만들어준 이름을 고이 받아들이기로 하자.
6. Edit Code 버튼을 클릭해서 OnMouseMove() 함수로 이동한 다음, [리스트 3.1]에 나온 코드를 입력하도록 하자.

### 리스트 3.1 OnMouseMove() 함수

```

1: void CMouseDlg::OnMouseMove(UINT nFlags, CPoint point)
2: {
3:     // TODO: Add your message handler code here and/or call default
4:
5:     /////////////////
6:     // 새로 넣을 코드가 여기서부터 시작된다
7:     /////////////////
8:
9:     //왼쪽 마우스 버튼이 눌렸나 체크한다
10:    if ((nFlags & MK_LBUTTON) == MK_LBUTTON)
11:    {
12:        // 다이얼로그의 디바이스 컨텍스트를 잡아낸다
13:        CClientDC dc(this);
14:
15:        // 점을 찍는다
16:        dc.SetPixel(point.x, point.y, RGB(0, 0, 0));
17:    }
18:
19:    /////////////////
20:    // 새로 넣을 코드는 여기서 끝난다
21:    /////////////////
22:
23:    CDialog::OnMouseMove(nFlags, point);
24: }
```

우선 이 리스트의 처음에 나와있는 함수 정의 부분을 보자. 두 개의 인수가 넘겨지고 있음을 알 수 있는데, 이 중 첫번째는 마우스 버튼이 눌렸는지 등을 알아내는데 사용하는 플래그 값의 집합이다. 아니나 다를까, 마우스 버튼의 눌림을 점검하는 코드가 이 리스트의 처음을 장식하고 있다.

```
if ((nFlags & MK_LBUTTON) == MK_LBUTTON)
```

위의 수식에서 첫 부분은 플래그 집합이 필터링되어 왼쪽 마우스 버튼이 눌려진 상태인지를 가리키는 플래그 값만 남게 되는 부분이며, 다음 부분에서 필터링된 플래그 값이 왼쪽 마우스 버튼이 눌렸는지를 가리키는 값과 비교되고 있다. 만약 두 개의 값이 같으면 왼쪽 마우스 버튼이 눌린 것이다.

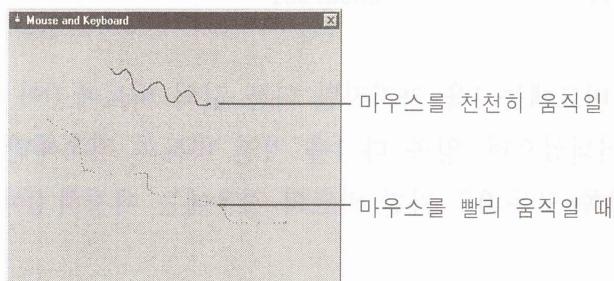
이제 두번째 인수를 살펴보기로 하자. 이 두번째 인수는 마우스 커서의 위치를 나타내는 값이다. 더 정확히 말하면 윈도우의 왼쪽 상단을 기준으로 마우스 커서가 위치한 좌표이며, 이 정보를 사용해서 다이얼로그 윈도우에 점을 찍을 수 있다.

다이얼로그 윈도우에 그림을 그리기 전에 꼭 필요한 것이 이 윈도우에 대한 디바이스 컨텍스트를 잡아내는 것인데, **CCClientDC** 클래스의 인스턴스를 만들므로써 자동으로 수행되었다. 이 클래스는 윈도우의 클라이언트 영역(프레임 윈도우를 제외한 부분)에 대한 디바이스 컨텍스트와 이와 관련된 대부분의 연산 동작을 감싸고 있기 때문에, 당연히 화면에 그림을 그리는 함수도 이 클래스에서 찾을 수 있다. 어떤 의미에서는 디바이스 컨텍스트 자체를 화폭으로 생각해도 무방하다. 이렇게 생각해 보면, 그림을 그리기 위해서는 반드시 화폭이 필요하다는 사실도 감이 올 것이다. 디바이스 컨텍스트를 획득한 다음에, 이 디바이스 컨텍스트의 멤버 함수인 **SetPixel()**이란 함수를 사용해서 점을 찍고 있다. 이 함수는 좌표를 나타내는 두 개의 인수와 색깔을 나타내는 인수를 받아들인다.

이 상태에서 프로그램을 컴파일하고 실행시킨 다음 나타나는 윈도우에 마우스 커서를 대고 왼쪽 마우스 버튼을 누른 채 움직여보자. [그림 3.1]과 같이 마우스가 지나가는 곳마다 점이 찍힐 것이다.

**그림 3.1** →

마우스를 사용해서 윈도우에 그림을 그리고 있다.



윈도우 운영체제에서의 색깔 지정은 세 개의 숫자를 조합한 값으로 설정된다. 이 세 개의 숫자는 각각 빨강, 녹색, 파랑의 강도를 나타내며, 앞에 쓰인 RGB 함수는 이 세 개의 숫자값을 하나의 값으로 조합해 주는 매크로이다. 각 숫자는 0부터 255 사이의 값을 가질 수 있다.

### AND와 OR 연산자를 사용하기

C++의 왕초보라면 일단 여기서 AND와 OR 논리 연산자에 대해 확실히 이해하고 갈 필요가 있다. AND와 OR는 이진(binary) 논리(logical) 연산자라고 불리며, 논리 수식 또는 if나 while 같은 조건 수식에 쓰인다. 또한, 두 개의 피 연산자를 이진수 레벨(비트 대 비트)에서 조합할 때도 사용한다.

앰퍼샌드 문자(&)는 AND 연산자를 표기할 때 쓴다. 이 문자를 하나만 쓰면(&) 이진 AND 연산자이며, 두 개의 문자를 연달아 쓰면(&&) 논리 AND 연산자이다. 논리 AND 연산은 파워빌더나 비주얼 베이직에서의 AND라는 말과 똑같다. if 문장에 쓰여서 “이 조건이 맞고(AND) 저 조건도 맞는가”를 알아보는데, 두 개의 조건이 모두 참이어야 전체 문장이 참이 된다. 이진 AND는 특정 비트를 설정하거나 클리어할 때 사용한다. 두 개의 값이 이진 AND 연산자에 사용되면, 값을 구성하고 있는 각 대응 비트들이 모두 1일 때만 1로 남게 되며, 나머지는 모두 0이 된다. 잘 이해가 안되는 부분을 위해 아래에 8비트 숫자의 예를 들어보았다.

첫번째 값

01011001

두번째 값

00101001

위의 두 값을 이진 AND 연산자에 사용하면 다음의 값이 결과로 나오게 된다.

AND된 값

00001001

한쪽 값의 비트에는 1을 가지지만 다른 값의 비트에 0이 있는 경우는 해당 비트가 0으로 클리어되었으며, 양쪽 다 1을 가진 비트의 경우에만 해당 비트가 1로 설정되었다. 또한 양쪽 모두 0을 가진 비트의 경우에는 여전히 0이 되었다.

OR는 파이프 문자(|)로 표기하며 AND와 마찬가지로 하나만 쓸 경우는(|) 이진 OR 연산자, 두 개를 연달아 쓸 경우(||)는 논리 OR 연산자이다. 역시 AND와 마찬가지로 논리 OR은 if나 while같은 조건수식에 쓰이고 파워빌더나 비주얼 베이직에서의 OR와 동일한 의미이다. 즉, 두 개의 조건 중 하나만 참이면 전체 문장이 참이 되는 것이다. 이진 OR 연산자는 두 개의 값을 이진수 레벨에서 조합할 때 사용한다. 두 개의 값을 구성하는 비트 중 한쪽만 1이 있으면 결과는 1로 되며, 양쪽 모두 0이 있는 경우의 결과만 0이다. 이진 AND를 설명할 때 사용했던 값을 그대로 이진 OR 연산자에 넣어보도록 하자.

첫번째 값

01011001

두번째 값

00101001

위의 두 값을 이진 OR 연산자에 넣으면 다음의 값이 결과로 나온다.

OR된 값

01111001

결과값

이 경우, 한쪽에만 1이 있으면 OR 연산의 결과는 1이 되었고, 양쪽 비트 모두 0일 경우의 결과값만 0이 되었다.

### 이진 속성 플래그

이진 AND와 이진 OR는 C++에서 속성 플래그 값을 설정하고 읽을 때 사용된다. 속성 플래그는 특정한 옵션의 활성/비활성 여부를 결정짓는 각각의 비트로 구성된 어떤 값을 말한다. 따라서 프로그래머는 이 옵션을 읽어내기 위해 정의된 플래그를 사용할 수 있다. 정의된 플래그(defined flag)란 단지 한 개의 비트만 1로 설정되어 있거나 다른 값과 조합되어 여러 개의 옵션이 설정되어 있는(즉, 여러 개의 비트가 1로 설정된) 값을 말한다. 방금 말한 그 여러 가지 옵션을 조절하는 플래그들은 OR되어, 어떤 옵션은 활성화되고 어떤 옵션은 비활성화되어 있는 혼합 플래그 값을 만들 수 있다.

만일 특정한 조건을 나타내는 두 개의 플래그가 한 바이트 내에서 다른 비트로 설정되어 있다면, 다음과 같이 OR시킬 수 있다.

플래그 1

00001000

플래그 2

00100000

조합 결과

00101000

이렇게 함으로써, 제한된 메모리 공간을 가지고도 상당히 다양한 옵션을 복합적으로 설정할 수 있다. 사실 윈도우 시스템에서 운영하는 컨트롤 대부분이 이러한 플래그로 구동되는데, 다이얼로그 에디터에서 불러내는 프로퍼티 다이얼로그 박스가 가지고 있는 체크 박스들이 그 플래그들인 것이다. 체크 박스를 체크하거나 해제한 결과는 모두 OR되어 한 개 혹은 두 개의 플래그 값 집합을 형성하고, 이것은 윈도우 운영체제에 의해 검사되어 해당되는 컨트롤이나 윈도우의 외형과 동작 방식이 결정된다.

이 과정 중에, 특정한 플래그가 플래그 조합값에 포함되었는지를 알아보려면 AND 연산을 시도하면 된다. 바로 다음처럼 말이다.

플래그 조합값	00101000
플래그 1	00001000
결과	00001000

앞의 연산 결과는 플래그 조합값을 필터링(AND 연산) 하기 위해 사용한 플래그와 같은지, 다른지 비교할 수 있다. 만약 비교 결과가 같은 것으로 나오면, 그 플래그는 플래그 조합값에 포함된 것이다.

또 다른 방법은 필터링된 플래그 조합값이 0이 아닌지를 체크하는 것으로, 플래그 조합값을 필터링하는데 사용된 플래그가 포함되어 있지 않았다면 필터링된 플래그는 0이 될 것이며, 결과적으로 앞선 방법에서 쓰인 if 문장 내부의 비교 수식은 없어져도 무방하다. 즉, 다음과 같이 써도 된다는 뜻이다.

```
if (nFlags & MK_LBUTTON)
```

어떤 플래그가 플래그 조합값에 포함되어 있지 않은가를 알아보려면 아래와 같이 고친다.

```
if (!(nFlags & MK_LBUTTON))
```

여기서 제시된 방법 중에 이해하기 쉬운 것부터 사용해 보길 바란다. 어차피 둘 다 빈번하게 사용되는 것이다.

## 그림 그리기 프로그램을 더 좋게 만들자

조금 아둔한 사람이라도, 그리기 프로그램에 작은 문제 하나가 있음을 눈치챘을 것이다. 큰 맘 먹고 직선을 하나 그리려면 마우스를 깜빡이 소다에 나오는 달팽이처럼 느리게 움직여야 한다. 다른 멋진 그리기 프로그램들은 이러지 않는데, 무슨 해결책을 쓰고 있는 것이 분명하다. 그런데 조금만 생각해 보면 단순한 문제이다. 즉, 마우스로 찍힌 점 사이를 선으로 이어주면 되는 것이다. 속임수인 것처럼 보일 테지만, 컴퓨터는 원래 이렇게 굴러먹는 것을 어찌하랴.

마우스를 화면상에서 움직일 때 여러분의 컴퓨터는 시간 간격을 두고 마우스의 위치를 점검한다. 즉, 계속해서 마우스가 어디에 박혀 있는지 감시하고 있지 않기 때문에

몇 가지 가정을 세워야 하는데, 컴퓨터가 현재 알고 있는 점들을 잡아서 그 사이를 직선으로 이어주는 것이다. 그림판 프로그램으로 자유 그리기를 해보면 바로 이런 방식으로 그려진다.

이러한 방식은 대부분의 상용 프로그램이 취하고 있는데, 어떻게 이런 위대한 기술을 우리의 프로그램에도 적용시킬 수 있을까? 우선 마우스의 바로 이전의 위치를 기억해두는 것이 필요하다. 즉, X와 Y 좌표를 담을 두 개의 변수를 다이얼로그 윈도우에 추가해야 한다는 뜻이다. 자, 그럼 다음의 단계를 차근히 봅아보자.

1. 프로젝트 워크스페이스에서 클래스뷰 탭을 선택한다.
2. 다이얼로그 클래스를 선택해야 하는데, 이 경우 CMouseDlg 클래스를 선택하면 된다.
3. 마우스의 오른쪽 버튼을 클릭한 다음, 문맥 메뉴에서 Add Member Variable을 선택한다.
4. [그림 3.2]와 같이 Variable Type에 int를 입력하고, Variable Name에는 m\_iPrevY를 입력한다. 액세스 지정자는 Private를 선택하자.

그림 3.2 ➔

Add Member Variable  
다이얼로그



5. OK를 클릭해서 변수를 추가한다.
6. 3번 단계부터 5번 단계까지 반복해서 m\_iPrevX라는 이름의 int 타입의 변수를 하나 더 추가하자.

마우스 위치를 기억해 두기 위한 변수를 모두 추가했으면, [리스트 3.2]와 같이 OnMouseMove() 함수를 고치도록 하자.

### 리스트 3.2 OnMouseMove() 함수를 고친 상태

```

1: void CMouseDlg::OnMouseMove(UINT nFlags, CPoint point)
2: {
3:     // TODO: Add your message handler code here and/or call default
4:
5:     /////////////////
6:     // 새로 넣을 코드가 여기서부터 시작된다
7:     ///////////////
8:
9:     // 왼쪽 마우스 버튼이 눌렸나 본다
10:    if ((nFlags & MK_LBUTTON) == MK_LBUTTON)
11:    {
12:        // 디바이스 컨텍스트를 잡아 낸다
13:        CCClientDC dc(this);
14:
15:        // 바로 이전의 점과 현재 점 사이를 선으로 잇는다
16:        dc.MoveTo(m_iPrevX, m_iPrevY);
17:        dc.LineTo(point.x, point.y);
18:
19:        // 현재 점을 (이후의) 이전 점으로 저장한다
20:        m_iPrevX = point.x;
21:        m_iPrevY = point.y;
22:    }
23:
24:    /////////////////
25:    // 새로 넣을 코드는 여기서 끝난다
26:    ///////////////
27:
28:    CDialog::OnMouseMove(nFlags, point);
29: }
```

이전 점에서 현재 점까지 선을 그리는 코드를 잘 보도록 하자.

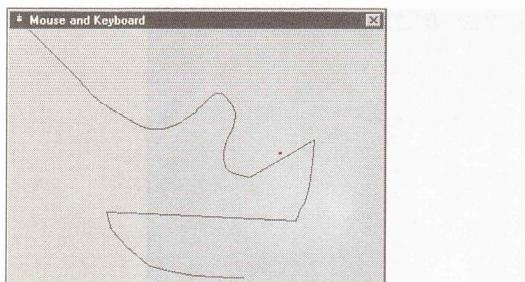
```

dc.MoveTo(m_iPrevX, m_iPrevY);
dc.LineTo(point.x, point.y);
```

첫번째 점으로 옮겨간 다음, 두번째 점으로 선을 그어주어야겠다는 생각이 들지 않느가? 이 첫번째 단계가 매우 중요하다. 왜냐하면, 이것을 빼놓았다면 시작점이 어디 있는지를 알 도리가 없기 때문이다. 애플리케이션을 컴파일하고 실행시키면 이전보다는 많이 좋아진 것 같다. 하지만 이제는 이상한 동작을 하고 있다. 무언가를 그리기 위해 왼쪽 마우스 버튼을 누를 때마다 방금 전에 그리기가 끝난 그 점에서부터 선이 그려지는 것이다. 마치 [그림 3.3]처럼 말이다.

**그림 3.3 →**

아직 기괴한 행동을 하는  
그리기 프로그램

**잔손질이 조금 남았다**

현재 여러분의 애플리케이션은 왼쪽 버튼이 눌린 상태에서 마우스 이동 이벤트가 일어날 때마다 선을 그려대고 있다. 따라서 이것을 고치기 위해서는 왼쪽 버튼이 눌렸을 때, 이전 좌표를 담는 변수를 마우스의 현재 위치로 초기화해 주어야 한다. 조금만 참고 다음 단계를 따르자.

1. 클래스워저드를 사용해서 다이얼로그 개체에 대해 WM\_LBUTTONDOWN 메시지의 함수를 추가한다.
2. 방금 추가한 OnLButtonDown() 함수를 [리스트 3.3]과 같이 편집하자.

**리스트 3.3 OnLButtonDown() 함수**

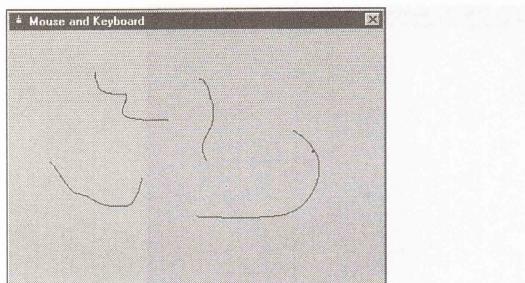
```

1: void CMouseDlg::OnLButtonDown(UINT nFlags, CPoint point)
2: {
3:     // TODO: Add your message handler code here and/or call default
4:
5:     /////////////////
6:     // 새로 넣을 코드가 여기서부터 시작된다
7:     /////////////////
8:
9:     // 현재 점을 시작점으로 설정한다
10:    m_iPrevX = point.x;
11:    m_iPrevY = point.y;
12:
13:    /////////////////
14:    // 새로 넣을 코드는 여기서 끝난다
15:    ///////////////
16:
17:    CDialog::OnLButtonDown(nFlags, point);
18: }
```

이제 애플리케이션을 컴파일하고 실행시키면, 이제 “그림 그리기”다운 그리기를 해낼 것이다. [그림 3.4]에 나온 그림도 얼마든지 그릴 수 있게 된다.

**그림 3.4 →**

이제 그리기 프로그램이 완성되었다.

**키보드 이벤트를 잡아내자**

키보드 이벤트를 읽어내는 일은 마우스 이벤트를 읽어내는 일과 거의 비슷하다. 마우스와 마찬가지로, 키가 눌렸을 때와 떼였을 때의 메시지가 따로 존재한다. [표 3.2]를 보도록 하자.

[표 3.2] 키보드 이벤트 메시지

메시지	설명
WM_KEYDOWN	키가 눌렸다.
WM_KEYUP	키가 떼였다.

확실히 키보드 이벤트는 마우스보다는 수가 적은 것 같다. 키보드로 해줄 일 또한 이만큼 뿐이다. 이들 이벤트 메시지는 다이얼로그 윈도우 개체에서 사용이 가능하며, 윈도우에 활성화된 컨트롤이 하나도 없을 때에만(중요하다!) 제대로 잡아낼 수 있다. 활성화된 컨트롤은 입력 포커스를 가지고 있기 때문에 모든 키보드 이벤트가 그려로 쏠리고 만다. 필자가 이전부터 계속 메인 다이얼로그 윈도우에서 컨트롤을 모두 지우라고 길길이 뛴 이유를 이해했으면 좋겠다.

**그리는 도중에 커서를 바꾸자**

키보드와 관련된 이벤트 메시지를 사용할 수 있는 방법에 대해 좋은 생각이 없을까? 어떤 키를 누르면, 마우스 커서가 바뀌도록 하면 재미있을 것 같다. A 키를 누르면 디폴트 화살표(arrow)로, B 키를 누르면 I자 막대(bean)로, C 키를 누르면 모래시계로 커서를 바꾸도록 해보자. 자, 그럼 시작하도록 하자.

1. 클래스위저드를 사용해서 다이얼로그 개체에 대해 WM\_KEYDOWN 메시지의 핸수를 추가한다.
2. [리스트 3.4]와 같이 방금 추가한 OnKeyDown() 함수를 편집하자.

#### 리스트 3.4 OnKeyDown() 함수

```

1: void CMouseDlg::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
2: {
3:     // TODO: Add your message handler code here and/or call default
4:
5:     ///////////////////////
6:     // 새로 넣을 코드가 여기서부터 시작된다
7:     /////////////////////
8:
9:     char lsChar;           // 눌린 문자 키 코드
10:    HCURSOR lhCursor;    // 표시될 커서의 핸들
11:
12:    // 키코드를 문자로 바꾼다
13:    lsChar = char(nChar);
14:
15:    // 'A' 문자인가?
16:    if (lsChar == 'A')
17:    {
18:        // 화살표 커서를 로드한다
19:        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_ARROW);
20:        // 화면 커서를 설정한다
21:        SetCursor(lhCursor);
22:    }
23:
24:    // 'B' 문자인가?
25:    if (lsChar == 'B')
26:    {
27:        // I자 막대 커서를 로드한다.
28:        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_IBEAM);
29:        // 화면 커서를 설정한다
30:        SetCursor(lhCursor);
31:    }
32:
33:    // 'C' 문자인가?
34:    if (lsChar == 'C')
35:    {
36:        // ahfotlzP 커서를 로드한다.
37:        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_WAIT);
38:        // 화면 커서를 설정한다
39:        SetCursor(lhCursor);
40:    }
41:
42:    // 'X' 문자인가?
43:    if (lsChar == 'X')
44:    {

```

```

45: // 화살표 커서를 로드한다.
46: lhCursor = AfxGetApp()->LoadStandardCursor(IDC_ARROW);
47: // 화면 커서를 설정한다
48: SetCursor(lhCursor);
49: // 애플리케이션을 종료한다
50: OnOK();
51: }
52:
53: /////////////
54: // 새로 넣을 코드는 여기서 끝난다
55: /////////////
56:
57: CDIalog::OnKeyDown(nChar, nRepCnt, nFlags);
58: }

```

OnKeyDown() 함수는 세 개의 인수를 받는다. 첫번째는 눌린 키의 코드로서, 리스트의 첫 부분에서 문자타입으로 변환되고 있다. 이렇게 한 다음에는 어떤 키가 눌렸는지 직접 비교할 수 있게 된다.

```
void CMouseDlg::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
```

두번째 인수는 키가 눌린 횟수이다. 대개 키가 눌린 다음 떼이면 이 값이 1이지만, 이 키가 눌리고 나서 계속 눌린 채로 있으면 반복 횟수가 증가한다. 결국 이 값은 윈도우 운영체제 마음대로 결정하는 반복횟수이지, 절대적인 횟수가 아니란 점을 알아두기 바란다.

세번째 인수는 동시에 Alt 키가 눌렸는지, 눌린 키가 확장키인지를 결정하기 위해 사용되는 플래그 조합값인데, 쉬프트나 컨트롤 키가 눌렸는지는 알아볼 수 없는 현실이 안타깝다.

어떤 키가 눌렸는지 확인되는 시점이 바로 그 키에 해당하는 커서로 바꿔줄 때이다. 두 단계의 과정을 거치는데, 일단 메모리에 커서를 로드해야 한다. 여기서는 애플리케이션 클래스(CWinApp)의 LoadStandardCursor() 함수를 사용하였는데, 이 함수는 표준 윈도우 커서 중에 하나를 로드해서 그 커서에 대한 핸들을 반환한다.

### Note

이 함수의 자매격인 LoadCursor()는 윈도우 운영체제에 내장된 표준 커서가 아니라, 여러분이 직접 만든 커서의 파일 이름 또는 리소스 이름을 인수로 받는다. 만약에 비주얼 C++의 리소스 에디터로 직접 여러분만의 커서를 만들었다면 이 커서의 이름을 LoadCursor()에 넘기면 된다. 예를 들어, 여러분이 만든 커서의 이름이 IDC\_MYCURSOR라면 다음과 같이 하자.

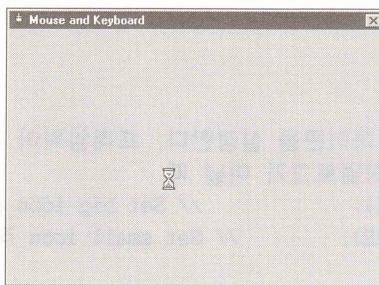
```
hCursor = AfxGetApp()->LoadCursor(IDC_MYCURSOR);
```

커서를 성공적으로 로드한 다음에는, 현재의 마우스 커서를 로드한 커서로 바꿔주기 위해 SetCursor() 함수를 사용하면 된다.

커서가 메모리에 로드된 후에 얻어낸 커서의 핸들이 이 함수에 넘겨지며, 핸들이 나타내는 커서가 현재 커서로 바뀐다. 이 상태에서 여러분의 애플리케이션을 컴파일하고 실행시키면 A, B, C 키를 누를 때마다 커서가 바뀌는 것을 확인할 수 있다([그림 3.5] 참조). 하지만, 그림을 그리기 위해 마우스를 조금만 움직여도 커서가 바로 원래의 화살표 커서로 돌아와 버리는데, 이에 대한 뒷처리는 다음 절에서 해주도록 하자.

그림 3.5 →

특정한 키를 눌렀을 때  
커서가 바뀐다.



### 커서를 바뀐 채로 유지하자

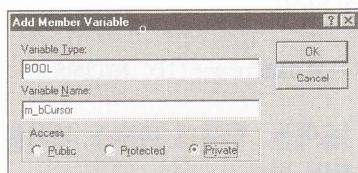
여러분의 그리기 프로그램이 가진 문제의 포인트는 마우스를 이동시킬 때마다 커서가 다시 그려진다는 점에 있다. 이러한 동작을 하지 못하게 할 방법이 있어야 한다.

마우스 커서가 다시 그려져야 할 때는 1) 마우스가 이동할 때, 2) 여러분의 애플리케이션 위에 있던 다른 윈도우가 치워졌을 때인데, 직접 WM\_SETCURSOR 이벤트 메시지가 여러분의 애플리케이션으로 보내질 때도 다시 그려져야 한다. 따라서 이 메시지의 원래 동작을 재정의하면 여러분이 바꾸어 준 커서가 계속 유지될 수 있는 것이다. 이제 시작하자.

1. 이전 좌표를 담는 변수를 추가할 때 해준 것처럼 CMouseDlg 클래스에 새 변수를 추가하자. 이번에는 BOOL 타입의 m\_bCursor란 이름으로([그림 3.6] 참조) 추가하면 된다.

그림 3.6 →

클래스 멤버 변수를 정의한다.



2. [리스트 3.5]와 같이, OnInitDialog() 함수 안에서 m\_bCursor 변수를 초기화하도록 하자.

### 리스트 3.5 OnInitDialog() 함수

```

1: BOOL CMouseDlg::OnInitDialog()
2: {
3:     CDialog::OnInitDialog();
4:
5:
6:
7:
8:     // 디아일로그에 대한 아이콘을 설정한다. 프레임워크가 자동으로 해주는 부분
9:     // 메인 윈도우가 디아일로그가 아닐 때
10:    SetIcon(m_hIcon, TRUE);           // Set big icon 큰 아이콘을 설정
11:    SetIcon(m_hIcon, FALSE);         // Set small icon 작은 아이콘을 설정
12:
13:    // TODO: Add extra initialization here
14:
15:    ///////////////////////////////
16:    // 새로 넣을 코드가 여기서부터 시작된다
17:    //////////////////////////////
18:
19:    // 커서를 화살표로 초기화한다
20:    m_bCursor = FALSE;
21:
22:    //////////////////////////////
23:    // 새로 넣을 코드는 여기서 끝난다
24:    //////////////////////////////
25:
26:    return TRUE; // return TRUE unless you set the focus to a control
27: }
```

3. [리스트 3.6]과 같이 OnKeyDown() 함수를 고쳐서 커서를 바꿀 때 m\_bCursor 플래그를 TRUE로 설정해 주자.

**리스트 3.6** OnKeyDown() 함수

```
1: void CMouseDlg::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
2: {
3:     // TODO: Add your message handler code here and/or call default
4:
5:     /////////////////
6:     // 새로 넣을 코드가 여기서부터 시작된다
7:     /////////////////
8:
9:     char lsChar;      // 현재 눌린 문자키
10:    HCURSOR lhCursor; // 표시될 커서의 핸들
11:
12:    // 눌린키를 문자로 바꾼다.
13:    lsChar = char(nChar);
14:
15:    // 'A' 문자인가?
16:    if (lsChar == 'A')
17:        // 화살표 커서를 로드한다.
18:        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_ARROW);
19:
20:    // 'B' 문자인가?
21:    if (lsChar == 'B')
22:        // I자 막대 커서를 로드한다
23:        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_IBEAM);
24:
25:    // 'C' 문자인가?
26:    if (lsChar == 'C')
27:        // 모래시계 커서를 로드한다
28:        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_WAIT);
29:
30:    // 'X' 문자인가?
31:    if (lsChar == 'X')
32:    {
33:        // 커서를 로드한다
34:        lhCursor = AfxGetApp()->LoadStandardCursor(IDC_ARROW);
35:        // 커서 플래그를 설정한다
36:        m_bCursor = TRUE;
37:        // 화면 커서로 설정한다
38:        SetCursor(lhCursor);
39:        // 애플리케이션을 종료한다
40:        OnOK();
41:    }
42:    else
43:    {
44:        // 커서 플래그를 설정한다
45:        m_bCursor = TRUE;
46:        // 화면 커서로 설정한다
47:        SetCursor(lhCursor);
48:    }
49:
50:    /////////////////
51:    // 새로 넣을 코드는 여기서 끝난다
```

```

52:   /////////////////
53:
54:   CDialog::OnKeyDown(nChar, nRepCnt, nFlags);
55: }
```

4. 클래스위저드를 사용해서 WM\_SETCURSOR 메시지에 대한 함수를 다이얼로그 개체에 추가한다.

5. [리스트 3.7]과 같이 OnSetCursor() 함수를 편집한다.

### 리스트 3.7 OnSetCursor() 함수

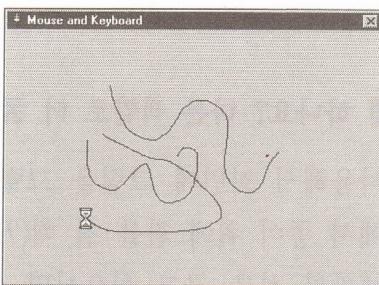
```

1: BOOL CMouseDlg::OnSetCursor(CWnd* pWnd, UINT nHitTest, UINT message)
2: {
3:     // TODO: Add your message handler code here and/or call default
4:
5:     // 새로 넣을 코드가 여기서부터 시작된다
6:     /////////////////
7:
8:     // TRUE 커서가 설정된 상태이면 TRUE를 반환한다
9:     if (m_bCursor)
10:         return TRUE;
11:     else
12:
13:     /////////////////
14:     // 새로 넣을 코드는 여기서 끝난다
15:     /////////////////
16:
17:
18:     return CDialog::OnSetCursor(pWnd, nHitTest, message);
19: }
```

OnSetCursor() 함수는 항상 TRUE를 반환하거나 그렇지 않으면 기본 클래스의 OnSetCursor()를 호출해야 한다. 기본 클래스의 함수가 하는 일은 커서를 리셋하는 것이며, 애플리케이션이 처음 시작될 때는 반드시 호출되어야 한다. 이 때문에 여러분의 변수를 FALSE로 초기화하여 사용자가 어떤 키를 눌러서 커서를 바꿀 때까지 기본 클래스의 디폴트 OnSetCursor() 처리가 실행되도록 내버려 두어야 하는 것이다. 사용자가 커서를 바꿀 때에는 디폴트 처리를 하지 않게 하고 바로 TRUE를 반환시키면, 디폴트 처리에 의해 커서가 리셋되지 않으므로 계속 바뀐 커서를 사용할 수 있는 것이다. [그림 3.7]은 모래시계 커서를 사용하고 있는 도중의 모습이다.

## 그림 3.7 →

모래시계 커서로 그림을  
그리고 있는 모습



## Note

여러분의 프로그램에서 가장 빈번하게 사용하게 될 커서 중 하나는, 조금 시간이 많이 걸리는 작업을 하는 도중에 사용자에게 기다리라는 뜻을 알리는 모래시계 커서이다. MFC를 사용해서 모래시계 커서를 현재 커서로 설정하기 위해서는 두 개의 함수를 사용한다. 첫번째는 사용자에게 모래시계 커서를 보이는 `BeginWaitCursor()` 함수이며, 두번 째는 디폴트 커서로 되돌리는 `EndWaitCursor()` 함수이다. 이들은 `CCmdTarget` 클래스의 멤버 함수이지만, 모든 MFC 컨트롤 클래스가 이 클래스에서 파생된 것이기 때문에 컨트롤에서도 사용할 수 있다.

모래시계를 표시해야 할 때와 모래시계를 더 이상 표시할 필요가 없는 상황의 모든 처리를 조절해 주는 하나의 함수를 만들고 싶으면, `CWaitCursor` 클래스 타입의 변수를 이 함수 안에 선언해 두자. 모래시계 커서가 자동으로 표시되며, 이 함수가 종료될 때 원래 커서로 되돌려지므로 매우 편리하다.

## 요약

이 장에서는 마우스 이벤트 메시지를 잡는 방법을 공부했으며, 간단한 처리의 실습도 겸했다. 마우스 이벤트를 사용해서 간단한 그림 그리기 프로그램을 만들었는데, 이 프로그램은 다이얼로그 윈도우에 마우스를 통한 자유 그리기를 구현한 프로그램이었다.

그 다음에 공부한 것은 키보드 이벤트이다. 어떤 키가 눌렸는지를 알아내어서, 이 정보를 마우스 커서를 바꾸는데 사용하였다. 이 작업을 위해 MFC 애플리케이션에서 디폴트 마우스 커서 처리에 대한 사전 지식을 머리에 넣었고, 이것을 토대로 해서 여러분이 원하는 대로 애플리케이션이 제대로 작동하기 위한 코드를 작성하였다.

다음 장에서 배울 것은 윈도우 운영체제의 타이머로서, 일정 시간 간격으로 어떤 이벤트가 발생되도록 만드는 방법이다. 또한, 다이얼로그 윈도우를 하나 더 사용해서 여러분의 애플리케이션이 동작하는데 필요한 사용자로부터의 입력을 받아낼 수 있도록 해볼 것이다. 이 공부를 마치고 나서는 메뉴를 만드는 방법까지 공부하도록 하자.

## Q&A

① 선의 종류를 바꾸려면 어떻게 하나요? 다른 색깔로 더 굵은 선을 긋고 싶어요.

① 표준 디바이스 컨텍스트를 사용해서 화면에 그림을 그릴 때는 펜(pen)이란 것을 사용하는데, 이 펜은 일상생활에서 종이 위에 환을 칠 때(?) 쓰는 그 펜과 유사한 개념입니다. 굵은 선이나 다른 색깔의 선을 긋고 싶으시면, 새 펜을 선택해서 디바이스 컨텍스트에 꽂아야(!) 합니다. 여러분이 만든 프로그램에서는 디바이스 컨텍스트를 잡아내는 장소인 OnMouseMove() 함수를 고쳐주는 것이 합당하겠죠. 다음 코드는 빨간 색의 굵은 선을 사용할 수 있도록 합니다.

```
// 디바이스 컨텍스트를 잡아낸다.  
CCClientDC dc(this);  
  
// 새 펜을 생성한다  
CPen lpen(PS_SOLID, 16, RGB(255, 0, 0));  
  
// 새 펜을 사용한다  
dc.SelectObject(&lpen);  
  
// 이전 점에서 현재 점까지 선을 긋는다  
dc.MoveTo(m_iPrevX, m_iPrevY);  
dc.LineTo(point.x, point.y);
```

① WM\_KEYDOWN 메시지를 받았을 때 Shift 키나 Ctrl 키가 눌렸는지 알아내려면 어떻게 할까요?

① ::GetKeyState()라는 Win32 함수를 사용합시다. 이 함수에 특정한 키 코드를 넣어서 호출한 결과값으로 해당 키가 눌렸는지, 눌려지지 않았는지 알 수 있습니다. 만약 결과값이 음수이면 키가 눌린 것이고, 음수가 아니면 눌리지 않은 것입니다. 그럼, Shift 키가 눌렸는지를 알아보는 코드를 써 볼까요?

```
if (::GetKeyState(VK_SHIFT) < 0)  
    MessageBox("Shift Key is down!");
```

윈도우 운영체제에는 모든 특수키에 대해서 가상 키코드(virtual key code)가 정의되어 있습니다. 이 코드를 사용하면 OEM 키보드의 스캔 코드나 다른 종류의 키 시퀀스 따위를 걱정하지 않고 한결같이 특수키를 찾을 수 있습니다. 이 가상 키 코드는 ::GetKeyState() 함수의 인수나 OnKeyDown() 함수의 nChar 인수로 사용할 수 있습니다. 가상 키 코드에 대해서는 비주얼 C++의 도움말을 참고하세요.

## 실습해 보기

“실습해 보기” 절에서는 배운 것을 확인하는 퀴즈와 이를 활용해서 응용력을 높이기 위한 연습문제를 풀어볼 기회를 가질 수 있게 될 것이다. 퀴즈와 연습문제의 답은 부록 B “퀴즈 및 연습문제 해답”에 있지만 이제 무슨 말을 하려는지 감이 잡히겠지요?

### ■ 퀴즈

1. 함수를 추가할 수 있는 마우스 메시지에는 무엇이 있는가?
2. WM\_MOUSEMOVE 이벤트 메시지 안에서 왼쪽 마우스 버튼이 눌린 것을 어떻게 알아낼 수 있는가?
3. 마우스 커서를 바꾼 다음, 원래의 커서로 되돌아가지 못하게 하려면 어떻게 하는가?

### ■ 연습문제

1. 여러분의 그림 그리기 프로그램을 고쳐서 왼쪽 마우스 버튼으로는 빨간색선, 오른쪽 마우스 버튼으로는 파랑색선을 그릴 수 있도록 해보자.
2. OnKeyDown() 함수를 더 확장해서 다음의 표준 커서 중 몇 개를 더 사용할 수 있도록 해보자.

- ❖ IDC\_CROSS
- ❖ IDC\_UPARROW
- ❖ IDC\_SIZEALL
- ❖ IDC\_SIZENWSE
- ❖ IDC\_SIZENESW
- ❖ IDC\_SIZEWE
- ❖ IDC\_SIZENS
- ❖ IDC\_NO
- ❖ IDC\_APPSTARTING
- ❖ IDC\_HELP