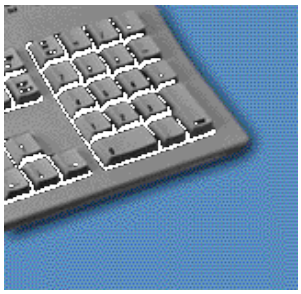


1. 윈도우즈 프로그래밍

<장도비라>
이 장에서는 윈도우즈 프로그래밍의 기본 구조와 동작 방식을 살펴본다. 윈도우 클래스를 등록하는 것부터 시작해서 윈도우 메시지를 처리해서 화면에 출력하는 것까지의 내부 동작을 자세히 살펴 볼 것이며, 아울러 동작 원리를 분석하고 이해한다. 이 장의 내용은 필자가 이 책의 모든 부분에 걸쳐서 설명할 MFC 코드 분석의 핵심이 되는 내용을 담고 있다. 책의 의도가 윈도우 프로그래밍을 처음부터 가리키려는 것이 아니라, MFC가 어떻게 윈도우 프로그래밍을 하는지 구조 분석을 하는 것이라는 점을 상기하자. 그러므로 13장까지 진행하면서 이 장에서 설명하는 내용 외에 추가적인 API함수를 사용하는 일은 거의 없을 것이다.
</장도비라>

간단한 윈도우즈 응용 프로그램

<절도비라>
이 절에서는 윈도우즈 본격적인 윈도우즈 응용 프로그램의 분석을 하기에 앞서, 메시지 박스(message box)를 출력하고 리턴하는 간단한 윈도우즈 응용 프로그램을 작성해 본다.
</절도비라>

독자들은 고전적인 Hello C 프로그램을 알고 있을 것이다. 리치(Ritchie)□가 그의 유명한 "C 언어 프로그래밍"에서 제시한 Hello 프로그램은 다음과 같다.

<여기서잠깐>
데니스 리치(Dennis M. Ritchie)는 C 언어의 창시자로서, 그의 저서 『C 언어 프로

그래밍(The C Programming Language)』은 프로그래밍 서적의 고전이 되었다.
</여기서잠깐>

```
#include <stdio.h>
int main()
{
    printf( "Hello World" );
    return 0;
}
```

위 프로그램은 표준 출력 장치(standard output device)인 콘솔(console)의 현재 커서(cursor) 위치에 Hello World를 출력한다. 우리는 이 프로그램의 윈도우즈 버전을 작성해 볼 수 있다.

먼저 #include <stdio.h>는 #include <windows.h>로 대치한다. stdio.h 파일은 printf()를 비롯한 표준 입출력 함수들의 선언을 담고 있다. 따라서 printf() 함수를 사용하기 위해서는 stdio.h를 포함시켜서 함수의 선언□을 해 주어야 한다.

<여기서잠깐>
함수나 변수들은 사용(호출)하기 전에 반드시 선언해 주어야 한다. 헤더 파일은 함수나 변수의 선언을 담고 있는데, 특정한 함수를 사용한 경우, 직접 선언을 적어주는 것보다 함수의 선언이 포함된 파일을 포함(include)시켜서 같은 효과를 거둔다. printf()의 선언은 stdio.h 파일에 들어 있고, MessageBox()의 선언은 windows.h에 들어 있다.
</여기서잠깐>

하지만, 윈도우즈 프로그래밍에서는 필요한 함수들의 선언은 windows.h에 들어 있다. 따라서 윈도우즈 프로그램은 windows.h를 포함시켜서 필요한 함수들의 선언을 해 주어야 한다. stdio.h에 선언된 함수들은 C 표준 함수들이지만, windows.h에 선언된 함수들은 오직 윈도우즈 운영체제에서만 사용할 수 있는 함수들이다. 이렇게 특정 플랫폼에서 지원하는 함수들을 **API(application programming interface)**□라고 한다. 우리가 사용할 첫 번

<여기서잠깐>
API는 윈도우즈 운영체제가 제공하는 함수들의 묶음이다. MessageBox()는 윈도우즈 프로그래밍의 API이고, printf()는 C API라고 할 수 있다. API함수들의 종류와 사용방법, 상호관계를 이해하는 것은 윈도우즈 프로그래밍의 넘어야 할 산이다. 우리가 C/C++을 모두 알고 있지만, 윈도우즈 응용 프로그램을 작성할 수 없는 이유는 윈도우즈의 API를 잘 모르기 때문이다.
</여기서잠깐>

째 API함수는 MessageBox()이다. 윈도우즈는 표준 출력 장치인 콘솔(console) 대신에 그래픽 장치에 출력을 하므로, 콘솔 출력 함수인 printf()를

이용해서 문자열을 출력할 수는 없다. 대신에 윈도우즈가 제공하는 대화상자(dialog box)를 이용해서 문자열을 출력해 보자. 대화상자 출력 함수가 바로 MessageBox()이다. MSDN□에서 MessageBox()를 찾아보면 다음과 같이 선

<여기서잠깐>
http://www.microsoft.com/msdownload/platformsdk/sdkupdate/에서 플랫폼 SDK를 설치할 수 있다. 플랫폼 SDK를 설치하면 도움말을 실행해서 API함수들의 도움말을 확인할 수 있다.
</여기서잠깐>

언 되어 있음을 알 수 있다.

```
int MessageBox(
  HWND hWnd,             // 소유자 윈도우에 대한 핸들
  LPCTSTR lpText,        // 메시지 박스에 출력할 텍스트
  LPCTSTR lpCaption,     // 메시지 박스의 제목
  UINT uType             // 메시지 박스의 스타일
);□
```

<여기서잠깐>
API 함수의 이름은 단어의 첫글자가 대문자로 시작하고, 연속된 단어 사이에 다른 특수문자 없이 구성하여, 함수의 이름을 읽기 좋게 만든다. 예를 들면, 메시지 박스를 표시하는 함수의 이름은 messagebox(), message_box()가 아니라 MessageBox()이다. 또한 호환을 위해 #define이나 typedef로 정의된 형이름이나 명칭들은 모두 대문자로 구성된다. 예를 들면 const char* 타입은 LPCTSTR인데 이것은 long pointer to const string의 약자이다. long pointer는 이전의 16비트 버전의 윈도우의 포인터에서 유래한 것으로, 지금은 사용되지 않지만 호환을 위해 남아 있다. 32비트 윈도우에서 포인터는 항상 32비트이다.
</여기서잠깐>

우리는 MessageBox()를 다음과 같이 호출할 것이다.

```
MessageBox( NULL, "Hello World", "Hello", MB_OK );
```

MessageBox()의 첫 번째 파라미터는 메시지 박스를 소유한 윈도우의 핸들(handle)□이다. 이 값을 0(NULL)으로 설정하여 이 메시지 박스의 소

<여기서잠깐>
핸들의 개념은 나중에 설명할 것이다. 간단히 말하면, 핸들이란 특정한 객체를 나타내는 숫자(number)이다. 이 숫자는 객체의 시작 주소와 같을 수도 있지만, 항상 그런 것은 아니다. 우리는 서로 다른 객체들을 이 숫자 즉 핸들을 통해 구분한다.
</여기서잠깐>

유자가 바탕화면(desktop window)□임을 알려준다.

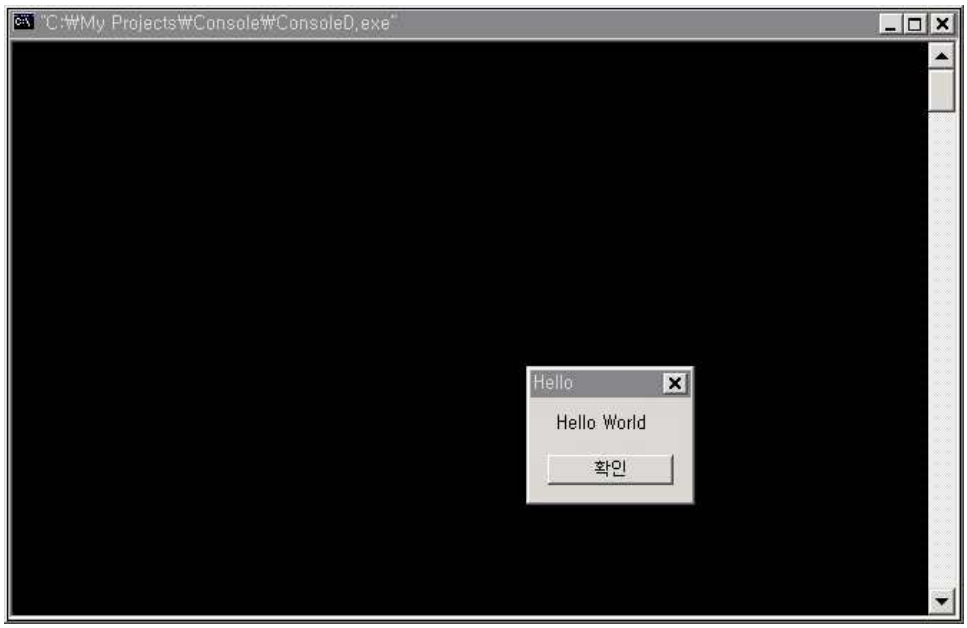
<여기서 잠깐>바탕화면은 아이콘 형태로 표시되는 리스트 뷰 컨트롤(list view control) 한 개를 포함하는 특별한 윈도우이며, 이 윈도우에 대한 핸들은 0이다.</여기서 잠깐>

두 번째 파라미터는 메시지 박스에 출력할 텍스트이며, 세 번째 파라미터는 메시지 박스의 타이틀에 출력할 제목이다. 네 번째 파라미터는 버튼과 모양의 조합을 의미하는 비트 플래그(bit flag)인데, MB_OK를 지정해 [확인]버튼을 가질 것을 지시한다. 이제 우리는 Hello 프로그램의 윈도우즈 버전을 다음 [예제 1.2]와 같이 작성해 볼 수 있다.

[예제 1.2] Hello 프로그램의 윈도우즈 버전

```
#include <windows.h>
int main()
{
    MessageBox( NULL, "Hello World", "Hello", MB_OK );
    return 0;
}
```

위 프로그램은 에러 없이 컴파일되고 실행된다. 하지만, 실행해 보면 콘솔창이 먼저 실행되고 그 위에 메시지 박스가 출력되는 것을 볼 수 있다. 프로그램의 실행 결과는 [그림 1.1]과 같다.



[그림 1.1] 첫번째 윈도우즈 Hello 프로그램: 첫번째 윈도우즈 Hello 프로그램을 작성했다. 하지만, 콘솔창이 실행되는 불완전한 프로그램은 원하지 않는다.

일반적인 윈도우즈 응용 프로그램은 콘솔창을 가지지 않는다. 윈도우즈는 콘솔창을 실행하는 윈도우즈 프로그램인지 콘솔창이 없는 윈도우즈 프로그램인지를 구분하기 위해 시작함수로 판단한다. 시작함수가 고전적인 main()이라면 윈도우즈는 콘솔창을 만든다. 콘솔창을 만들지 않는 윈도우즈 프로그램의 시작 함수는 main()이 아닌 WinMain()이며, 일반적인 윈도우즈 응용 프로그램의 시작 함수가 이 WinMain()이다. 이제 콘솔창이 실행되지 않는 윈도우즈용 Hello 프로그램을 [예제 1.3]과 같이 작성해 볼 수 있다.

[예제 1.3] WinMain()으로 시작하는 Hello 프로그램

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
                   PSTR szCmdLine,int iCmdShow)
{
    MessageBox( NULL, "Hello World", "Hello", MB_OK );
    return 0;
}
```

위 프로그램은 바탕화면(desktop)의 중앙에 메시지 박스(message box)를 출력한다. 출력 결과는 [그림 1.2]와 같다. 콘솔창이 실행되지 않았다는 사실에 주목하기 바란다.

[그림 1.2] 간단한 Win32응용 프로그램: 화면의 중앙에 Hello World라는 메시지 박스를 보여준다.

main()과 달리 WinMain()은 함수 호출 관례(function calling convention)□

<여기서잠깐>
함수 호출 관례란 함수를 호출할 때, 파라미터를 처리하는 방식을 말한다. 고전적인 pascal 방법은 이제 더 이상 윈도우즈에서 사용하지 않는다. C 방법(cdecl)은 함수 호출시에 파라미터를 오른쪽에서 왼쪽으로 스택에 푸시하고, 파라미터의 처리를 호출하는 쪽에서 한다. stdcall 방법은 파라미터를 오른쪽에서 왼쪽으로 푸시하고 파라미터의 처리를 호출된 쪽에서 처리한다. 대부분의 윈도우즈 함수들은 모두 stdcall 방식의 호출 관례를 사용한다.
</여기서잠깐>

로 _stdcall□을 사용한다. [예제 1.2]의 main()에는 _cdecl이 생략되어 있는데

<여기서잠깐>
키워드 앞에 붙은 _(underscore)는 해당 키워드가 표준 키워드가 아님을 의미한다. _stdcall은 비주얼 C++에서만 사용 가능한 키워드이다.
</여기서잠깐>

사실 main()은 다음과 같이 적을 수 있다.

```
int _cdecl main()
```

하지만, 기본적으로 C의 함수들은 _cdecl 호출 관례를 사용하므로 main() 앞의 _cdecl을 생략한 것이다. 하지만 윈도우즈가 제공하는 대부분의 API 함수들은 _stdcall 방식의 호출 관례를 사용하므로 함수의 리턴 타입과 함수 이름 사이에 _stdcall을 반드시 적어주어야 한다.

```
int _stdcall WinMain(...)
```

하지만, [예제 1.3]에서 _stdcall 대신 WINAPI를 사용한 것을 볼 수 있는데, 사실 WINAPI는 _stdcall을 정의한 것이다. WINAPI에서 F12를 누르면 다음과 같은 WINAPI의 정의를 확인할 수 있다.

```
#define WINAPI      __stdcall
```

직접 _stdcall을 적지 않고, WINAPI를 적은 이유는 호환성을 위해서이다. 후에 변경되거나, 윈도우즈가 아닌 다른 플랫폼에서 호출 관례가 바뀌는 경우, WINAPI의 정의만 바꾸면 되기 때문이다.

WinMain()의 파라미터에 대한 자세한 설명은 본격적인 윈도우즈 프로그램을 작성하면서 살펴보기로 하자.

지금 작성해 본 예제 프로그램은 확실히 윈도우즈 프로그램이다. 하지만, 완벽하지는 않다. 왜냐하면 시스템 메뉴(system menu)를 가지고, 제목줄(title bar)에 최대화(maximize), 최소화(minimize) 버튼을 가지는 윈도우가 없기 때문이다.

우리는 다음 절에서 일반적인 윈도우즈 응용 프로그램의 구조를 살펴 본 다음, 그 다음 절에서 본격적인 윈도우즈 프로그램을 단계별로 작성해 나갈 것이다.

윈도우 프로그램의 구조

<절도비라>

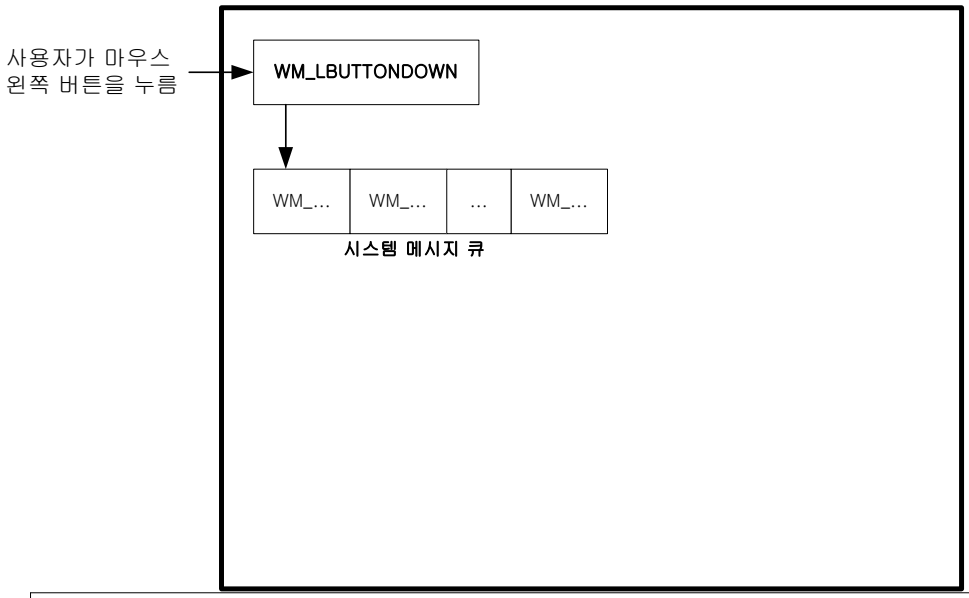
이 절에서는 일반적인 윈도우 프로그램의 구조와 동작 방식을 살펴본다. 윈도우즈가 이벤트를 메시지로 표현하므로, 응용 프로그램은 이러한 메시지를 가져오는 루프를 가져야 하고, 또한 메시지를 처리하는 특별한 함수를 가져야 함을 살펴볼 것이다.

</절도비라>

일반적인 윈도우즈 프로그램의 구조를 알아보기 위해 사용자가 마우스 왼쪽 버튼을 누른 경우 윈도우즈의 내부에서 어떤 일이 일어나는지 살펴보자.

윈도우즈 운영체제는 처리해야할 이벤트(event)가 발생한 경우, 이것은 윈도

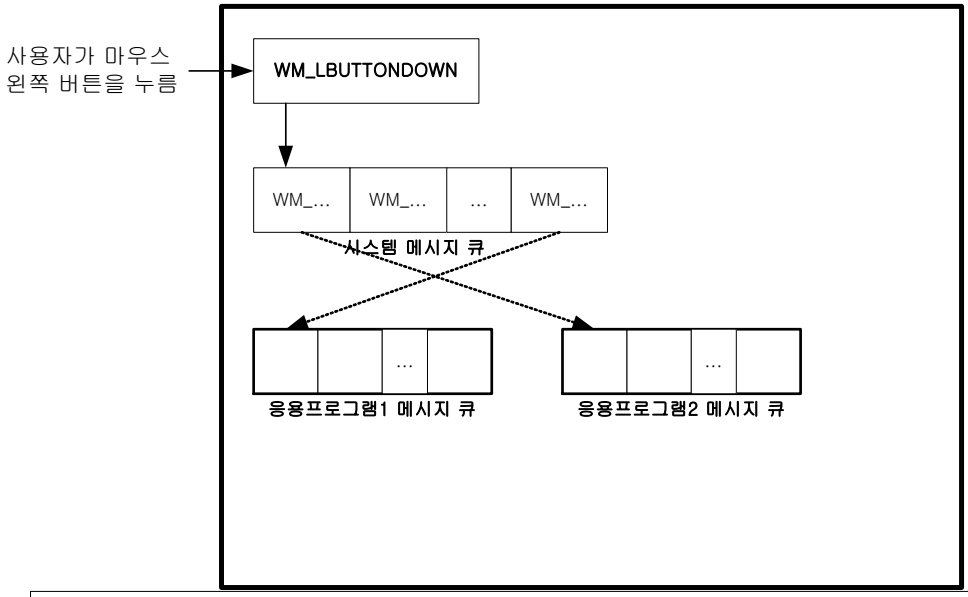
우즈 내부에서 **메시지로(message)** 표현된다. 마우스 왼쪽 버튼을 누른 이벤트의 경우에 해당하는 메시지를 WM_LBUTTONDOWN이라 하자. 이러한 일련의 메시지들은 운영체제가 관리하는 큐에 저장되는데, 이 큐를 **시스템 메시지 큐(system message queue)**라 한다. 아래 [그림 1.4]를 보자.



[그림 1.4] 시스템 메시지 큐: 이벤트가 발생한 경우, 이벤트는 메시지로 표현되어 큐에 저장된다.

이제 윈도우즈에서 실행되고 있는 두개의 응용 프로그램이 있다고 하자. 그리고 WM_LBUTTONDOWN을 처리해야 할 응용 프로그램은 두 번째 응용 프로그램이라고 가정하자. 두 번째 응용 프로그램은 어떻게 WM_LBUTTONDOWN을 처리할 수 있을까? 그것은 다음과 같다.

윈도우즈는 메시지를 처리할 응용 프로그램을 결정하기 위해, 응용 프로그램마다 별도의 **응용 프로그램 메시지 큐(application message queue)**를 유지하고, 시스템 큐에 있는 각 메시지를 해당하는 응용 프로그램 메시지 큐에 저장한다. 시스템 메시지 큐와 응용 프로그램 메시지 큐는 문맥상에서 구분되어야 하는데, 별다른 언급이 없다면 **메시지 큐**는 응용 프로그램 메시지 큐를 의미한다. 아래 [그림 1.5]를 보자.



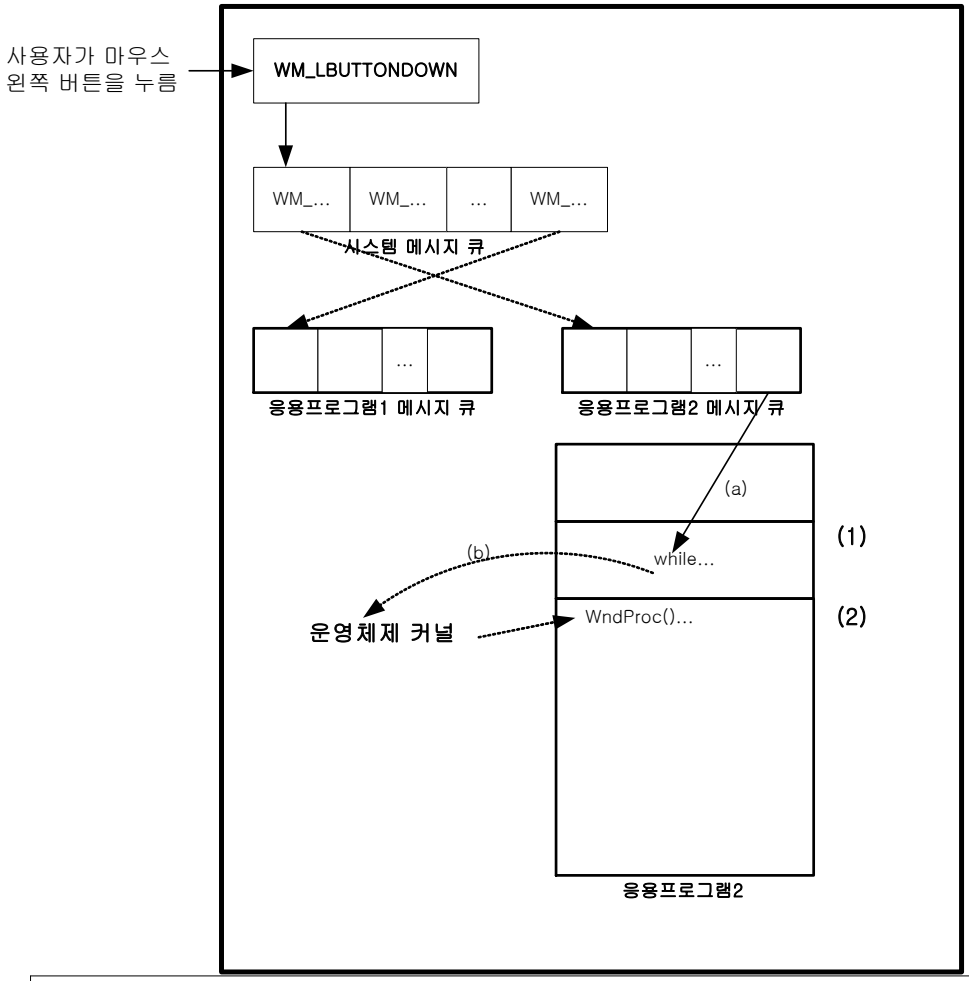
[그림 1.5] 응용 프로그램 메시지 큐: 이벤트가 발생하면 메시지는 먼저 시스템 메시지 큐에 저장된다. 그리고 이 메시지는 메시지를 처리해야 할 응용 프로그램 메시지 큐로 다시 저장된다.

이제 응용 프로그램은 메시지 큐에서 메시지를 꺼내 오고, 꺼내온 메시지를 적절하게 처리해야 한다. 그러므로 윈도우 프로그램에는 메시지 큐에서 메시지를 꺼내오는 부분과, 꺼내온 메시지를 처리하는 부분이 반드시 있어야 함을 알 수 있다.

각 부분은 운영체제가 제공하는 하나의 API 함수로 구현할 수 없다. 그 이유는 다음과 같다.

먼저 메시지 큐에서 메시지를 꺼내오는 부분을 보면, 메시지를 꺼내오고 꺼내온 메시지를 번역하고, 메시지 큐에 메시지가 없을 때 처리하는 부분을 어떻게 동작하게 할지는 응용 프로그램이 결정해야 한다. 응용 프로그램의 이러한 부분을 **메시지 루프(message loop)**라 한다.

다음으로 꺼내온 메시지를 처리하는 부분을 보면, 각 응용 프로그램마다 메시지를 처리하는 방식과 처리할 메시지를 결정하는 것이 다르므로 이 부분 역시 응용 프로그램이 결정해야 한다. 꺼내온 메시지를 처리하는 부분은 하나의 함수로 작성되어 윈도우 클래스를 등록할 때, 지정되어야 하는데 이 함수를 **윈도우 프로시저(window procedure)**라 한다. 아래 [그림 1.6]을 보자.



[그림 1.6] 메시지 루프와 윈도우 프로시저: 각 응용 프로그램은 메시지 큐에서 메시지를 꺼내오고(a), 꺼내온 메시지를 처리할 것이므로, 나를 불러달라고 요청한다(b). 그러면 운영체제는 윈도우 클래스에 등록된 메시지를 처리하는 함수를 호출한다. 전자(a,b)에 해당하는 프로그램 부분을 메시지 루프(1)라 하고, 후자에 해당하는 부분, 즉 메시지를 처리하는 함수를 윈도우 프로시저라 한다.

그래서 일반적인 윈도우 프로그램은 메시지 루프와 윈도우 프로시저를 반드시 가져야 한다. 메시지 루프는 대개 WinMain()안에 존재하고, 윈도우 프로시저는 별도의 함수로 존재한다.

일반적인 윈도우 프로그램의 구조를 살펴보았으므로, 다음 절에서 이 구조를 따르는 윈도우즈 응용 프로그램을 단계별로 작성해 보자.

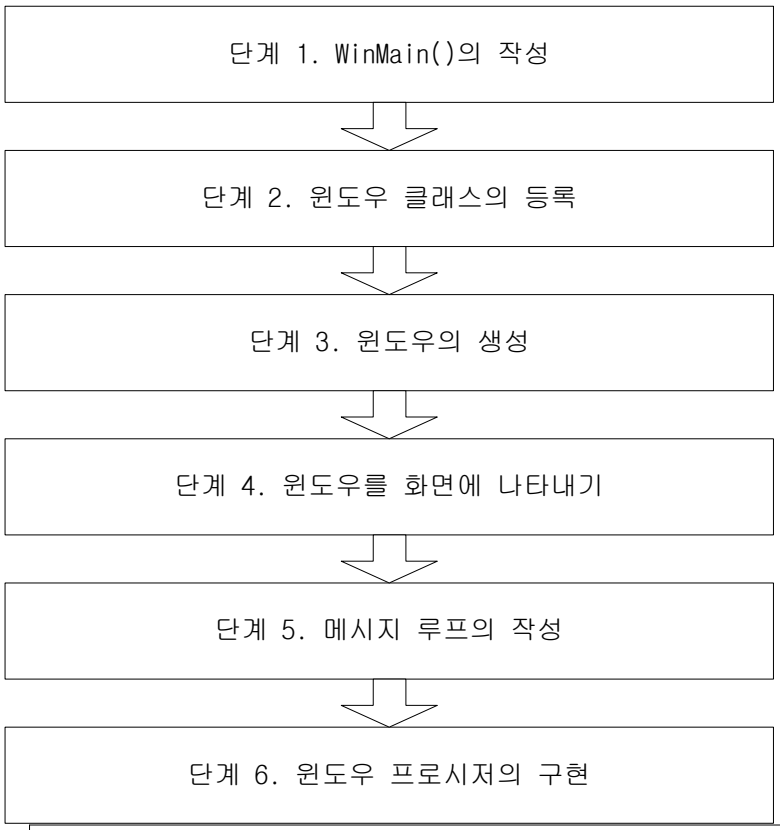
본격적인 윈도우즈 프로그램

<절도비라>

일반적인 윈도우즈 프로그램은 메시지 루프(message loop)와 윈도우 프로시저(window procedure)를 포함한다. 이 절에서는 이러한 본격적인 윈도우즈 응용 프로그램을 작성하고 내부 동작을 살펴본다. 이 절에서 작성하는 프로그램은 이 책의 모든 곳에서 사용할 기본적인 프로그램이 된다. 필자는 이 프로그램을 MFC 구조에 맞게 차례대로 수정해 나갈 것이므로, 이 절의 내용을 완벽하게 이해해야 한다.

</절도비라>

이제 윈도우를 가지는 본격적인 윈도우즈 응용 프로그램을 작성해 보자. 우리는 필요한 개념들을 하나하나씩 살펴보며, 코드를 완성해 갈 것이다. 우리는 아래 [그림 1.2]처럼 응용 프로그램을 단계별로 작성할 것이다.



[그림 1.2] 본격적인 윈도우즈 프로그램의 작성 단계

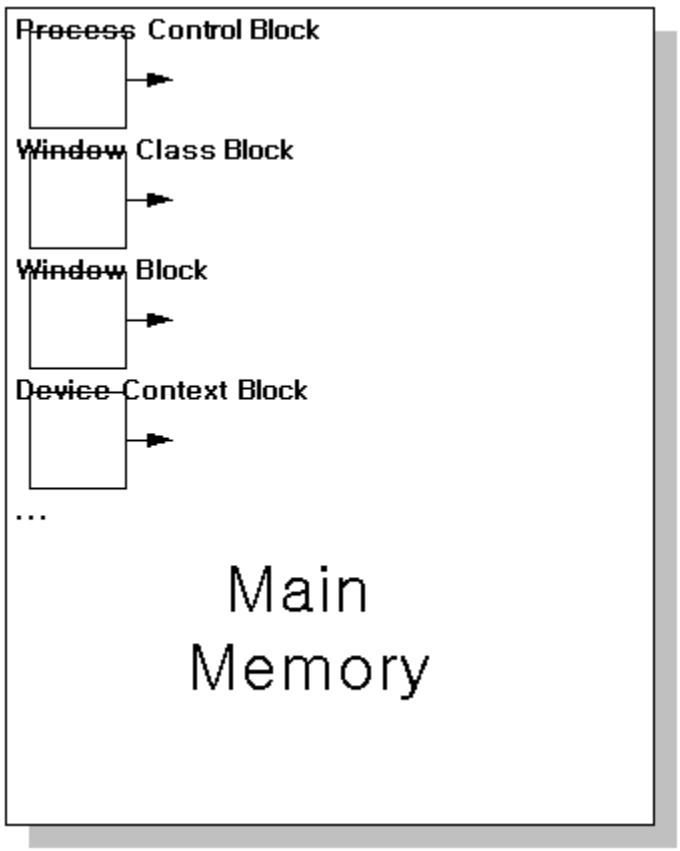
단계 1을 시작하기 전에 운영체제가 관리하는 구조체와 핸들의 개념을 먼저 살펴보자.

윈도우즈 운영체제는 관리하는 모든 객체(object)의 정보를 담고 있는 구조체(structure)를 리스트(list)로 유지한다. 객체들은 가시적일 수도 아닐 수도 있다. 객체의 예로 프로세스, 윈도우, 디바이스 컨텍스트(device context)□를 들 수 있다.

<여기서잠깐>
실행중인 프로그램을 프로세스라 한다. 디바이스 컨텍스트란 윈도우즈의 표면에 그리기 작업을 수행할 때 필요한 정보를 담고 있는 구조체를 말하며, 모든 그리기 함수는 디바이스 컨텍스트를 필요로 한다.
</여기서잠깐>

핸들은 윈도우즈 운영체제가 유지하는 객체를 표현하는 구조체와 관련된 것이다. 예를 들면, 윈도우즈는 프로세스(process)가 실행될 때마다 프로세스를 관리하기 위해 구조체를 만든다. 그리고 각 프로세스를 구분하기 위해 각각의

구조체에 다른 프로세스의 구조체와 구분되는 유일한 ID를 할당한다. 이 ID를 **핸들(handle)**이라고 하는데, 여기서는 **프로세스 핸들(process handle)**이 된다. 핸들은 의미 있는 값일 수도 있고, 단지 구조체가 할당된 메모리의 시작 주소일 수도 있다. [그림 1.3]은 운영체제가 메모리에 유지하는 몇 가지 구조체를 보여준다.



[그림 1.3] 운영체제의 몇 가지 구조체들: 운영체제는 객체들의 정보를 구조체의 리스트로 유지한다. 각각의 구조체를 구분하는 유일한 값을 핸들이라고 한다.

우리가 고려할 첫 번째 핸들은 **인스턴스 핸들(instance handle)**□이다. 인

<여기서잠깐>
C++에서 클래스의 한 예, 즉 객체를 의미하는 인스턴스(instance)와 헷갈리지 말자. 인스턴스 핸들의 인스턴스는 프로세스의 한 예(instance)를 의미한다.
</여기서잠깐>

스턴스 핸들이란 윈도우즈가 프로세스 핸들을 가리키는 말인데, 이 값은

WinMain()의 첫 번째 파라미터로 전달된다. 이제 본격적인 윈도우즈 프로그램 작성을 시작해 보자.

단계 1. WinMain()의 작성

우리는 다음과 같이 윈도우즈 프로그램 작성을 시작할 수 있다.

```
#include <windows.h>
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
{
    return 0;
}
```

프로그램이 실행되면 운영체제는 프로세스 인스턴스 블록(process instance block)을 만들고, WinMain()을 호출한다. 그리고 WinMain()의 첫 번째 파라미터로 인스턴스 핸들을 전달한다.

WinMain()의 두 번째 파라미터는 이미 존재하는 이전 프로세스에 대한 인스턴스 핸들이다.

<여기서잠깐>
윈도우즈 3.1시절까지는 해도 같은 프로그램이 두 번 실행되더라도 내용이 같은 코드 블록(code block)이 별도로 유지되었다. 윈도우즈 95시절부터 이러한 비효율적인 구조는 없어졌으나 호환성을 위해 두 번째 파라미터는 남아있지만, 사용되지 않는다.
</여기서잠깐>

세 번째 파라미터는 명령행 인자(command line argument)이며, 네번째 파라미터는 윈도우의 초기 상태를 나타내는 값으로

<여기서 잠깐>Win+R을 눌러 command(혹은 cmd)을 입력하면 명령행 창을 띄울 수 있다. 여기서 notepad.exe readme.txt를 입력한 경우, readme.txt가 WinMain()의 세번째 파라미터가 된다.</여기서 잠깐>

후에 윈도우를 표시하기 위해 호출할 ShowWindow()의 두 번째 파라미터와 역할이 같다.

WinMain()에서 첫 번째로 할 일은 윈도우 클래스를 등록하는 일이다.

단계 2. 윈도우 클래스의 등록

윈도우즈가 생성하는 모든 윈도우들은 윈도우의 동작과 모양을 정의하는 **윈도우 클래스(window class)**□으로부터 만들어진다. 그러므로 응용 프로그

<여기서잠깐>
윈도우 클래스를 C++에서 사용하는 클래스와 혼동하지 않도록 하자. 윈도우 클래스에 사용된 클래스는 분류(class)를 나타내는 일반적인 단어이다.
</여기서잠깐>

램에서 윈도우를 만들기 위해서는 윈도우를 만드는 API 함수에 이미 존재하는 윈도우 클래스 구조체의 핸들을 파라미터로 전달해 주어야 한다. 그러므로, WinMain()에서 첫 번째로 할 일은 윈도우를 생성하기 위해 윈도우 클래스를 등록하는 작업이다. 이제 WinMain()을 아래와 같이 작성할 수 있다.

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
{
    static char szAppName[] = "HelloWin";
    WNDCLASSEX  wndclass;

    wndclass.cbSize         = sizeof( wndclass ); // (1)
    wndclass.style          = CS_HREDRAW|CS_VREDRAW;
    wndclass.lpfnWndProc    = DefWindowProc; // WndProc // (2)
    wndclass.cbClsExtra     = 0;
    wndclass.cbWndExtra     = 0;
    wndclass.hInstance      = hInstance; // (3)
    wndclass.hIcon          = LoadIcon( NULL,IDI_APPLICATION );
    wndclass.hCursor        = LoadCursor( NULL,IDC_ARROW );
    wndclass.hbrBackground  = (HBRUSH)GetStockObject( WHITE_BRUSH
);
    wndclass.lpszMenuName   = NULL;
    wndclass.lpszClassName  = szAppName; // (4)
    wndclass.hIconSm        = LoadIcon( NULL, IDI_APPLICATION );

    RegisterClassEx( &wndclass ); // (5)

    return 0;
}//WinMain()
```

윈도우 클래스를 표현하는 구조체의 이름은 WNDCLASS인데, 윈도우즈 95가 등장하면서 확장된 윈도우 클래스를 나타내기 위해 이 구조체가 확장되었다. 확장된 구조체의 이름은 WNDCLASS**EX**이다. 사실 윈도우즈 프로그램을 하면서 제일 끝에 EX가 붙은 많은 구조체와 API 함수를 접할 수 있는데, 이들은 윈도우즈 3.1 시절에 사용되던 것들이 확장(extension)된 것들이다.

많은 윈도우즈 구조체 사용의 관례는 구조체의 첫 번째 4바이트를 구조체의 크기로 설정하는 것이다(1). 이것은 구조체의 포인터를 전달할 때, 접근 가능한 메모리의 범위를 알 수 있고, 모든 크기의 구조체를 전달할 수 있는 효율적인 방법으로 통신에서 패킷(packet)을 보낼 때도 흔히 사용되는 방법이다.

두 번째 중요한 멤버는 **윈도우 프로시저(window procedure)**□의 시작 주소이다(2). 윈도우 프로시저는 반드시 윈도우 클래스에 등록되어야 한다. 윈도우 프로시저는 후에 구현할 것인데 지금은 코드를 간단하게 하기 위해 DefWindowProc()으로 설정해 두자. 이 함수는 윈도우즈가 제공하는 디폴트 윈도우 프로시저이다.

<여기서잠깐>
윈도우즈 프로그램은 메시지로 동작한다. 윈도우에 특정한 이벤트가 발생하면, 윈도우즈는 이벤트가 발생한 윈도우에 등록된 이벤트 처리함수, 즉 윈도우 프로시저를 호출한다. 그러므로 윈도우 클래스를 등록할 때 윈도우 프로시저를 등록하는 것은 반드시 필요하다.
</여기서잠깐>

세 번째 중요한 멤버는 윈도우 클래스를 등록하는 프로세스의 인스턴스 핸들이다(3). 이것은 프로세스가 종료할 때 연관된 윈도우 클래스 구조체를 해제하기 위해 사용한다.

네 번째 중요한 멤버는 윈도우 클래스의 이름이다(4). 이것은 CreateWindow()를 호출하여 윈도우를 만들 때, 윈도우 클래스를 참조하기 위해 사용한다. 즉 모든 윈도우는 윈도우 클래스를 기반으로 하여 만들 수 있다. 구조체의 다른 멤버들의 의미는 MSDN에서 찾아보기 바란다.

윈도우 클래스 구조체가 준비되면 RegisterClassEx()를 사용하여 운영체제에 윈도우 클래스를 등록한다(5)□.

<여기서잠깐>
모든 프로그램에서 윈도우 클래스를 등록할 필요는 없다. 윈도우즈 운영체제가 이미 등록해 놓은 윈도우 클래스만을 사용한다면 윈도우 클래스를 등록하는 과정은 필요 없다. 하지만, 대부분의 경우 특정한 메시지를 처리하는 자신만의 윈도우 프로시저를 가지는 윈도우 클래스를 사용하므로 윈도우를 만드는 함수를 호출하기 전에 윈도우 클래스 구조체를 만들고 이를 운영체제에 등록하는 일을 먼저 해 주어야 한다.
</여기서잠깐>

단계 3. 윈도우의 생성

윈도우 클래스가 등록되었으므로 이제 윈도우 클래스를 기반으로 윈도우를 만들 수 있다. 윈도우즈 운영체제는 윈도우 클래스에 기반하여 만들어진 각각의 윈도우에 대해서 **윈도우 구조체(window structure)**를 유지하는데, 윈도우 구조체에 대한 핸들을 **윈도우 핸들(window handle)**이라고 한다.

윈도우를 만드는 함수는 CreateWindow()로, 이 함수는 윈도우 핸들을 리턴한다. 윈도우와 관련된 API 함수를 호출할 때는 항상 이 윈도우 핸들을 파라미터로 전달해 주어야한다. 이제 소스에 아래와 같이 윈도우 생성 코드를 추가한다.

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
{
    static char szAppName[] = "HelloWin";
    HWND        hwnd;
    WNDCLASSEX  wndclass;

    wndclass.cbSize         = sizeof( wndclass );
    wndclass.style          = CS_HREDRAW|CS_VREDRAW;
    wndclass.lpfnWndProc    = DefWindowProc; // WndProc
    wndclass.cbClsExtra     = 0;
    wndclass.cbWndExtra     = 0;
    wndclass.hInstance      = hInstance;
    wndclass.hIcon          = LoadIcon( NULL,IDI_APPLICATION );
    wndclass.hCursor        = LoadCursor( NULL,IDC_ARROW );
    wndclass.hbrBackground  = (HBRUSH)GetStockObject( WHITE_BRUSH );
    wndclass.lpszMenuName   = NULL;
    wndclass.lpszClassName  = szAppName;
    wndclass.hIconSm        = LoadIcon( NULL, IDI_APPLICATION );

    RegisterClassEx( &wndclass );

    hwnd = CreateWindow( szAppName, // window class name //(1)
        "The Hello Program",        // window caption
        WS_OVERLAPPEDWINDOW,        // window style
        CW_USEDEFAULT,              // initial x position
        CW_USEDEFAULT,              // initial y position
```

```
        CW_USEDEFAULT,                  // initial x size
        CW_USEDEFAULT,                  // initial y size
        NULL,                           // parent window handle
        NULL,                           // window menu handle
        hInstance,                      // program instance handle(2)
        NULL);                          // creation parameters

    return 0;
}//WinMain()
```

<여기서잠깐>
윈도우 구조체를 만든다는 것은 화면에 윈도우를 그리기 위한 정보를 메모리에 만든다는 것을 의미한다.
</여기서잠깐>

CreateWindow()의 첫 번째 파라미터는 윈도우 클래스의 이름이다(1). 모든 윈도우는 윈도우 클래스에 기반하여 만들어진다. 윈도우 클래스를 지정하는 것은 반드시 필요한데, 왜냐하면 윈도우 클래스에 윈도우의 메시지를 처리할 윈도우 프로시저가 지정되어 있기 때문이다(지금 윈도우 프로시저의 값은 DefWindowProc이다).

CreateWindow()의 또 다른 중요한 파라미터는 인스턴스 핸들이다(2). 이것은 이 윈도우를 소유한 프로세스를 지정하여, 프로세스가 죽을 때 프로세스가 소유한 모든 윈도우를 종료하도록 한다.

이제 윈도우 클래스를 등록하고, 윈도우를 만드는 작업까지 했으므로, 프로그램을 디버그 실행(F5)해 보자.

단계 4. 윈도우를 화면에 나타내기

프로그램은 실행되지만, 아무것도 보이지 않는다. 왜냐하면, 윈도우를 만들기만 했을 뿐, 윈도우를 화면에 나타내는 작업은 하지 않았기 때문이다. 이제 윈도우를 화면에 나타내는 코드를 추가한다.

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
{
    static char szAppName[] = "HelloWin";
    HWND        hwnd;
```

```
    WNDCLASSEX  wndclass;

    wndclass.cbSize         = sizeof( wndclass );
    wndclass.style          = CS_HREDRAW|CS_VREDRAW;
    wndclass.lpfnWndProc    = DefWindowProc;
    wndclass.cbClsExtra     = 0;
    wndclass.cbWndExtra     = 0;
    wndclass.hInstance      = hInstance;
    wndclass.hIcon          = LoadIcon( NULL,IDI_APPLICATION );
    wndclass.hCursor        = LoadCursor( NULL,IDC_ARROW );
    wndclass.hbrBackground  = (HBRUSH)GetStockObject( WHITE_BRUSH );
    wndclass.lpszMenuName   = NULL;
    wndclass.lpszClassName  = szAppName;
    wndclass.hIconSm        = LoadIcon( NULL, IDI_APPLICATION );

    RegisterClassEx( &wndclass );

    hwnd = CreateWindow( szAppName,      // window class name
        "The Hello Program",             // window caption
        WS_OVERLAPPEDWINDOW,             // window style
        CW_USEDEFAULT,                   // initial x position
        CW_USEDEFAULT,                   // initial y position
        CW_USEDEFAULT,                   // initial x size
        CW_USEDEFAULT,                   // initial y size
        NULL,                            // parent window handle
        NULL,                            // window menu handle
        hInstance,                       // program instance handle
        NULL);                           // creation parameters

    ShowWindow( hwnd, iCmdShow ); // (1)

    return 0;
}//WinMain()
```

윈도우를 그리는 API함수의 이름은 ShowWindow()이다(1). 이 함수는 당연히 윈도우 핸들을 필요로 한다.

이제 프로그램을 실행해 보면, 윈도우가 표시되는 것을 알 수 있다. 하지만, 윈도우는 나타나는 즉시 없어진다. 왜냐하면, ShowWindow()를 호출한 이후 프로그램이 즉시 종료하기 때문이다. 우리는 이러한 현상을 막기 위해 ShowWindow() 다음에 무한 루프(infinite loop)를 추가할 수 있다.

```
    ...
    hwnd = CreateWindow( szAppName,      // window class name
        "The Hello Program",             // window caption
        WS_OVERLAPPEDWINDOW,             // window style
        CW_USEDEFAULT,                   // initial x position
        CW_USEDEFAULT,                   // initial y position
        CW_USEDEFAULT,                   // initial x size
        CW_USEDEFAULT,                   // initial y size
        NULL,                            // parent window handle
        NULL,                            // window menu handle
        hInstance,                       // program instance handle
        NULL);                           // creation parameters

    ShowWindow( hwnd, iCmdShow );

    while ( 1 )
    {
    }//while

    return 0;
}//WinMain()
```

하지만, 이렇게 무한 루프를 추가하면, 윈도우가 화면에 나타나고 나서 아무런 동작도 하지 않는다. 왜냐하면 윈도우에 필요한 동작들을 하지 않고 그저 무한히 대기하기만 하기 때문이다. 그래서 우리는 이 루프를 앞서 살펴본 대로, 윈도우에 필요한 작업을 하면서 필요한 경우 종료하도록 작성해 주어야 한다. 이렇게 특별하게 만들어진 루프를 **메시지 루프(message loop)**라 한다.

단계 5. 메시지 루프(Message Loop)의 작성

WinMain()의 마지막에는 생성된 윈도우에 전달될 메시지를 처리하는 루프가 존재한다. 이 루프는 **응용 프로그램 메시지 큐(application message queue)**에서 메시지를 가져와서, 윈도우에게 메시지를 불러 달라는 요청(dispatch)을 하는 비교적 짧은 문장들이다. 그래서 이 루프를 **메시지 루프**라 한다. 이 루프를 종료하는 것은 프로그램을 종료하는 것을 의미한다. 우리는 메시지 루프를 아래와 같이 작성할 수 있다.

```
    while (GetMessage(&msg,NULL,0,0)) // (1)
    {
        DispatchMessage(&msg); // (2)
    }//while
```

GetMessage()는 메시지 큐에서 메시지(message queue)를 꺼내온다(1). 메시지를 성공적으로 꺼낸 대부분의 경우 이 함수는 1을 리턴한다. 메시지는 함수의 첫 번째 파라미터로 전달된 메시지 구조체에 파라미터로 나온다(out parameter).

GetMessage()가 1을 리턴하면 DispatchMessage()를 호출하여 윈도우즈에게 msg가 지시하는 메시지를 처리해달라고 요청한다(2). 그러면 윈도우즈 운영체제는 윈도우 클래스에 등록된 윈도우 프로시저를 호출한다.

지금까지 작성된 소스를 아래에 리스트하였다. 이제 프로그램은 완벽하게 실행된다.

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   PSTR szCmdLine, int iCmdShow)
{
    static char szAppName[] = "HelloWin";
    HWND        hwnd;
    MSG         msg;
    WNDCLASSEX  wndclass;

    wndclass.cbSize         = sizeof( wndclass );
    wndclass.style          = CS_HREDRAW|CS_VREDRAW;
    wndclass.lpfnWndProc    = DefWindowProc;
    wndclass.cbClsExtra     = 0;
    wndclass.cbWndExtra     = 0;
    wndclass.hInstance      = hInstance;
    wndclass.hIcon          = LoadIcon( NULL,IDI_APPLICATION );
    wndclass.hCursor        = LoadCursor( NULL,IDC_ARROW );
    wndclass.hbrBackground  = (HBRUSH)GetStockObject( WHITE_BRUSH );
    wndclass.lpszMenuName   = NULL;
    wndclass.lpszClassName  = szAppName;
    wndclass.hIconSm        = LoadIcon( NULL, IDI_APPLICATION );

    RegisterClassEx( &wndclass );
```

```
    hwnd = CreateWindow( szAppName,      // window class name
        "The Hello Program",             // window caption
        WS_OVERLAPPEDWINDOW,             // window style
        CW_USEDEFAULT,                   // initial x position
        CW_USEDEFAULT,                   // initial y position
        CW_USEDEFAULT,                   // initial x size
        CW_USEDEFAULT,                   // initial y size
        NULL,                            // parent window handle
        NULL,                            // window menu handle
        hInstance,                       // program instance handle
        NULL);                           // creation parameters

    ShowWindow( hwnd, iCmdShow );

    while ( GetMessage(&msg,NULL,0,0) )
    {
        DispatchMessage( &msg );
    }//while
    return 0;
}//WinMain()
```

프로그램의 실행 결과는 아래 그림과 같다. 이제 프로그램은 기본적인 윈도우 기능을 모두 지원한다.

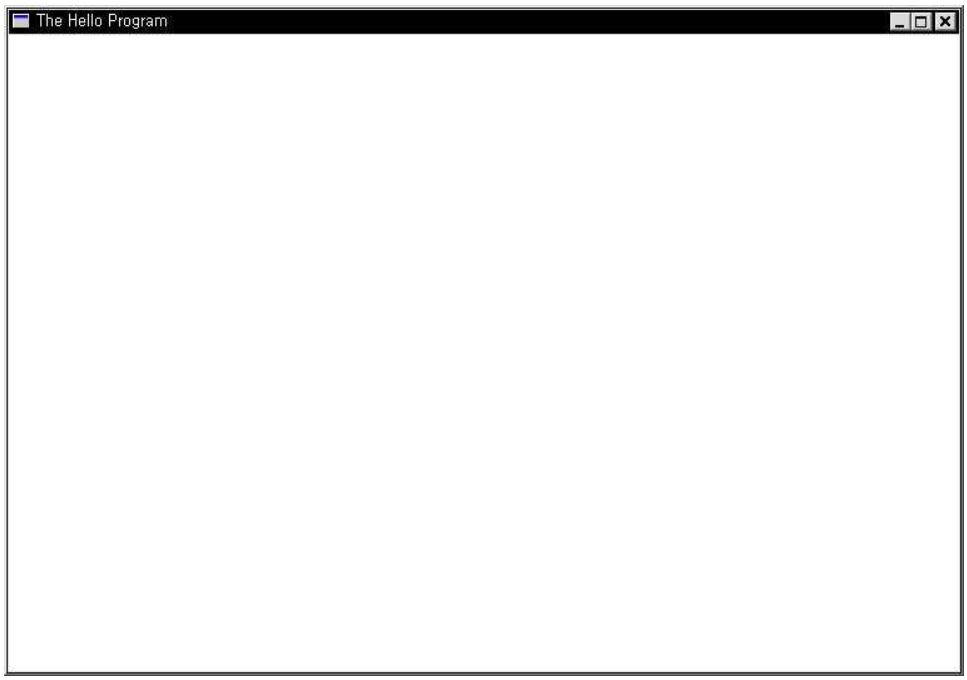


그림. Hello 프로그램: 이제 Hello 프로그램은 기본적인 윈도우 기능을 모두 처리한다.

이제 우리는 윈도우의 기본적인 기능만을 처리하는 것이 아니라, 이 기능을 확장하려고 한다. 그러기 위해서는 사용자가 윈도우 프로시저를 구현해 주어야 한다.

단계 6. 윈도우 프로시저의 구현

먼저 윈도우 프로시저의 몸체(body)를 구현해 보자. 일반적으로 윈도우 프로시저의 이름은 WndPoc()이다. 윈도우 프로시저 WndProc()은 윈도우즈에 의해 호출되는 함수이므로 호출관례로 _stdcall을 명시해야 한다. CALLBACK은 _stdcall과 같지만, 호환성을 위해서 _stdcall보다는 CALLBACK□으로 적는다.

<여기서잠깐>
WinMain()에서 처럼 WINAPI를 적어도 무방하다. 하지만, CALLBACK으로 적은 이유는 윈도우 프로시저는 윈도우즈에 의해 호출되는 함수이기 때문이다.
</여기서잠깐>

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam,
                         LPARAM lParam) // (1)
{
    return DefWindowProc(); // (2)
}
```

(1) 윈도우 프로시저는 네 개의 파라미터를 받는다. 첫 번째 파라미터는 윈도우에 대한 핸들이다. 두 번째 파라미터는 메시지 ID이다. 세 번째와 네 번째 파라미터는 각 메시지 ID에 종속적인 값으로 각 메시지 ID마다 해석하는 방법이 다르다.

(2) 지금 윈도우 프로시저의 내부에서 아무 일도 하지 않는다. 다만 윈도우 프로시저에 기본으로 해 주어야 하는 일을 처리하기 위해 DefWindowProc()□

<여기서잠깐>
DefWindowProc()을 호출하는 것은 매우 중요하다. 윈도우즈에서 우리가 처리하지 않는 많은 메시지들은 디폴트 처리를 해 주어야 한다. 예를 들면 타이틀 바를 드래그하는 경우, 대부분의 경우 우리는 이 처리를 DefWindowProc()에 맡긴다. DefWindowProc()호출이 없다면 타이틀 바를 드래그 할 수 없을 것이다.
</여기서잠깐>

을 호출한다.

윈도우 프로시저가 제일 처음 처리하는 메시지는 1(WM_CREATE)이다. 이 메시지는 윈도우 구조체를 메모리에 할당하고 즉 CreateWindow() 호출의 내부에서 발생하는 메시지이다. 윈도우 구조체만 할당되었고, 윈도우가 화면에 나타난 것은 아니므로, 이 메시지에서 그리기 작업을 할 수는 없다. 다만 유효한 윈도우 핸들과 연관된 데이터 초기화 작업을 하는 것은 가능하다. 이제 윈도우 프로시저를 아래와 같이 작성할 수 있다. WM_CREATE에서 지금은 아무 일도 하지 않고, 메시지를 처리했다는 것을 윈도우즈에 알려주기 위해 DefWindowProc()의 리턴값이 아닌 0을 리턴한다. 이렇게 사용자가 메시지를 처리한 경우는 0을 리턴해야 한다.

```
LRESULT CALLBACK WndProc(HWND hwnd, UINT iMsg, WPARAM wParam,
                         LPARAM lParam)
{
    switch (iMsg)
    {
    case WM_CREATE:
```

```
        return 0;
    }//switch
    return DefWindowProc( hwnd, iMsg, wParam, lParam );
}//WndProc()
```

윈도우즈는 윈도우의 내부를 그려야 할 때 WM_PAINT(16)□ 메시지를 발

<여기서잠깐>
WM_PAINT 값은 F12를 눌러서 확인할 수 있다.
</여기서잠깐>

생시킨다. 이러한 상황은 프로그램을 처음 시작할 때, 윈도우의 가려졌던 부분이 다시 나타날 때 등 다양하다.

이제 iMsg가 WM_PAINT라고 가정해 보자. 윈도우의 중앙에 Hello, Windows를 출력하기 위해 WM_PAINT의 처리 루틴을 아래와 같이 작성할 수 있다.

```
    case WM_PAINT:
        hdc=BeginPaint(hwnd,&ps); // (1)
        GetClientRect(hwnd,&rect);
        DrawText(hdc,"Hello, Windows",-1,&rect,
            DT_SINGLELINE|DT_CENTER|DT_VCENTER); // (2)
        EndPaint(hwnd,&ps);
        return 0;
```

모든 그리는 작업을 하기 전에는 먼저 **디바이스 컨텍스트에 대한 핸들(HDC, handle to device context)**을 얻어야 한다. 그래서 BeginPaint()를 호출하여 hdc를 얻는다. DC는 윈도우에 종속적이므로 BeginPaint()의 첫번째 파라미터로 윈도우 핸들을 전달한다(1).

DC를 얻는 함수는 몇 개가 존재하지만 WM_PAINT 메시지를 처리하는 경우에는 반드시 BeginPaint()□를 사용해야 하는데, 그 이유는 뒤에서 살

<여기서 잠깐>BeginPaint()는 메모리에 DC 구조체를 만든다. 그리기 작업이 끝난 후 EndPaint()를 호출하여 메모리에서 DC 구조체를 해제해 주어야 한다.</여기서 잠깐>

펴보도록 하자. 윈도우 프로그래밍에서 이러한 이유를 이해하는 것은 반드시 넘어야 할 산이다.

DC를 얻어냈으므로 이제 그리는 작업을 할 수 있다. 윈도우즈는 설치된 그

래픽 카드와 색 깊이 등과 상관없이 일관된 그리기 함수를 제공하는데 이것을 **GDI 함수(Graphic Device Interface Functions)**라고 한다. 모든 GDI 함수는 DC를 파라미터로 요구한다. 여기서는 DC에 텍스트를 출력하는 함수로 DrawText()를 사용한다(2).

윈도우 프로시저에서 세 번째로 처리할 메시지는 윈도우가 파괴될 때 발생하는 메시지이다. Alt+F4를 누르거나 종료 버튼을 선택하면, WM_DESTROY 메시지가 발생한다. 주의할 점은 윈도우 구조체가 파괴되고, 이 메시지가 발생한다는 것이다. 따라서서 이 메시지에서 DC를 얻어서 그리기를 시도하면 실패한다.

만약 마지막으로 그리기 작업이 필요하다면, WM_CLOSE에서 수행해야 한다. 이 메시지는 윈도우 구조체를 파괴하기 바로 전에 발생하는 메시지이다. 구조체가 파괴되기 전이므로 DC를 얻는 것은 유효하다.

```
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
```

WM_DESTROY 메시지의 처리는 그리기와 연관되지 않은 데이터 정리 작업을 할 수 있는 좋은 곳이다. 여기서는 단지 PostQuitMessage(0)를 호출하여 응용 프로그램 메시지 큐에 WM_QUIT를 집어 넣는다.

이제 다시 메시지 루프를 보자.

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

GetMessage()는 WM_QUIT 메시지를 꺼내면 0을 리턴한다. 그러므로 메시지 큐에 WM_QUIT메시지가 있으면, 메시지 루프는 종료된다.

윈도우 프로시저의 마지막에는 DefWindowProc()을 호출한다.

```
return DefWindowProc(hwnd,iMsg,wParam,lParam);
```

윈도우즈는 윈도우 프로시저에서 처리한 세 개의 메시지 외에도 많은 수의 메시지를 발생한다. 이렇게 윈도우 프로시저가 처리하지 못한 메시지는 기본

처리를 수행하도록 반드시 DefWindowProc()을 호출해 주어야 한다.

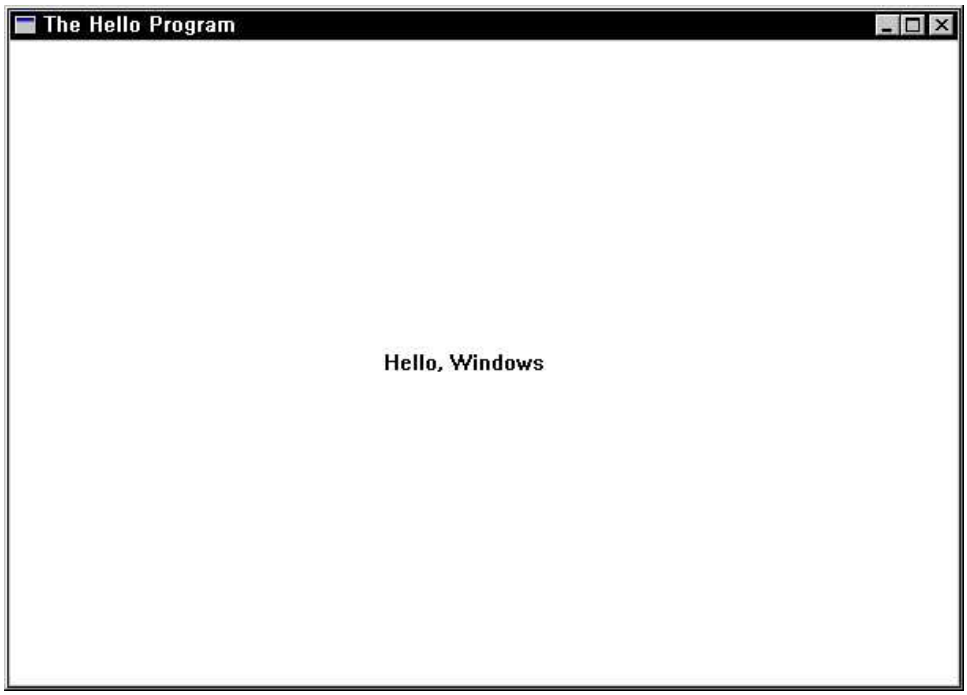
윈도우 프로시저의 세 번째 파라미터와 네번째 파라미터는 메시지에 종속적인 정보를 담고있다. 예를 들어, 어떤 메시지가 크기가 4바이트 이상인 구조체 정보를 필요로 한다고 하자. 어떻게 이것을 4바이트로 전달할 수 있을까? 바로 구조체의 포인터를 unsigned long 타입의 lParam으로 변환해서 네번째 파라미터로 전달한 다음, 이를 전달받은 쪽에서 다시 구조체 포인터로 변환하는 것이다(이는 윈도우즈가 파라미터를 전달하는 일반적인 방법이다). 이제 최종적으로 완성된 소스는 [예제 1.3]과 같다.

[예제 1.3] 윈도우 프로시저를 가지는 Hello 프로그램의 최종 소스

```
#include <windows.h>

LRESULT CALLBACK WndProc(HWND hwnd,UINT iMsg,WPARAM wParam,
                         LPARAM lParam)
{
    HDC         hdc;
    PAINTSTRUCT ps;
    RECT        rect;

    switch (iMsg)
    {
    case WM_CREATE:
        return 0;
    case WM_PAINT:
        hdc=BeginPaint(hwnd,&ps);
        GetClientRect(hwnd,&rect);
        DrawText(hdc,"Hello, Windows",-1,&rect,
            DT_SINGLELINE|DT_CENTER|DT_VCENTER);
        EndPaint(hwnd,&ps);
        return 0;
    case WM_DESTROY:
        PostQuitMessage(0);
        return 0;
    }//switch
    return DefWindowProc(hwnd,iMsg,wParam,lParam);
}//WndProc()

int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,
                   PSTR szCmdLine,int iCmdShow)
```

```
{
    static char szAppName[]="HelloWin";
    HWND        hwnd;
    MSG         msg;
    WNDCLASSEX  wndclass;

    wndclass.cbSize         =sizeof(wndclass);
    wndclass.style          =CS_HREDRAW|CS_VREDRAW;
    wndclass.lpfnWndProc    =WndProc;
    wndclass.cbClsExtra     =0;
    wndclass.cbWndExtra     =0;
    wndclass.hInstance      =hInstance;
    wndclass.hIcon          =LoadIcon(NULL,IDI_APPLICATION);
    wndclass.hCursor        =LoadCursor(NULL,IDC_ARROW);
    wndclass.hbrBackground  =(HBRUSH)GetStockObject(WHITE_BRUSH);
    wndclass.lpszMenuName   =NULL;
    wndclass.lpszClassName  =szAppName;
    wndclass.hIconSm        =LoadIcon(NULL,IDI_APPLICATION);

    RegisterClassEx(&wndclass);

    hwnd=CreateWindow(szAppName,    //window class name
        "The Hello Program",        //window caption
        WS_OVERLAPPEDWINDOW,        //window style
        CW_USEDEFAULT,              //initial x position
        CW_USEDEFAULT,              //initial y position
        CW_USEDEFAULT,              //initial x size
        CW_USEDEFAULT,              //initial y size
        NULL,                       //parent window handle
        NULL,                       //window menu handle
        hInstance,                  //program instance handle
        NULL);                      //creation parameters

    ShowWindow(hwnd,iCmdShow);
    //UpdateWindow(hwnd);

    while (GetMessage(&msg,NULL,0,0))
    {
        //TranslateMessage(&msg);
        DispatchMessage(&msg);
    }//while
    return msg.wParam;
}//WinMain()
```

출력 결과는 [그림 1.2]와 같다.



[그림 1.2] Hello 프로그램: 클라이언트 영역의 중앙에 Hello, Windows를 출력한다.

우리는 마침내 본격적인 윈도우 프로그램을 작성했다. 필자는 이 프로그램을 변경하면서 MFC 버전으로 작성해 볼 것인데, 최종 버전은 10장에 가서야 완성될 것이다.

자세히 살펴보기

<절도비라>

이 절에서는 앞 절에서 작성한 Hello 프로그램을 좀 더 자세히 살펴보고, 혹은 소스를 변경해 본다. 더 자세히 살펴볼 것들은 큐에 저장되는 메시지와 저장되지 않는 메시지에 관해서, DC를 얻는 함수의 종류에 대해서 그리고 MFC

의 메시지 루프의 형태에 관한 것이다.
</절도비라>

큐에 저장되는 메시지와 저장되지 않는 메시지

기민한 독자들은 이제까지 살펴본 소스에서 메시지 루프에 들어가기 전에 이미 윈도우 프로시저가 호출된 사실을 기억하고 있을 것이다. CreateWindow()를 호출했을 때 내부에서 윈도우 프로시저를 호출했다. 그러므로 윈도우 프로시저가 항상 메시지 루프에서만 호출된다는 가정을 해서는 안된다. 즉 메시지 큐에 저장되지 않는 메시지도 있다는 것이며 이러한 대표적인 예로 CreateWindow()가 발생시키는 WM_CREATE 메시지가 있다. 사실 WM_CREATE 메시지를 메시지 큐에 넣어서 처리하는 것은 불가능하다. CreateWindow()호출에서 직접 윈도우 프로시저를 호출하지 않고, 단지 메시지 큐에 WM_CREATE 메시지를 넣기만 한다면, WM_CREATE는 윈도우가 생성된 시점이 아닌 이상함 시점에 처리될 것이기 때문이다.

윈도우즈의 이러한 메시지 큐 모델은 일반적인 응용 프로그램을 설계하는데에도 많이 사용하므로 이러한 구조에 익숙해지도록 하자.

BeginPaint() vs. GetDC()

필자는 앞서 WM_PAINT 메시지를 처리하는 경우 DC를 얻기 위해 반드시 BeginPaint()를 호출해야 한다고 언급한 바 있다. 그 이유를 살펴보기 위해 응용 프로그램에 기능을 추가하자.

마우스 왼쪽 버튼을 누르면 클라이언트 영역의 좌측 상단인 (0,0)에서 (100,100)으로 두께가 1픽셀인 검은색 실선을 그리는 작업을 추가해 보자. 마우스 왼쪽 버튼을 누르는 이벤트에 대응하는 메시지는 WM_LBUTTONDOWN이다. [그림 1.14]를 보자.

아래 [예제 1.4]를 WndProc()의 switch문에 추가하면 된다.

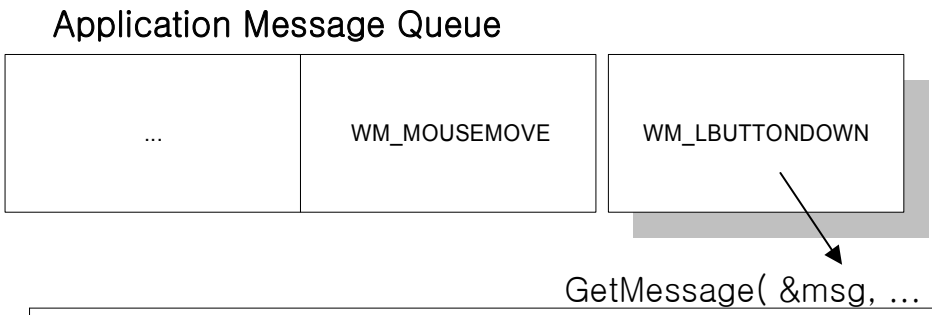
[예제 1.4] WM_LBUTTONDOWN의 처리

```
    case WM_LBUTTONDOWN:
        hdc = GetDC(hwnd);
        MoveToEx(hdc,0,0,NULL);
        LineTo(hdc,100,100);
```

```
        ReleaseDC(hwnd, hdc);
        return 0;
```

GDI 함수의 기본 좌표계는 - 윈도우즈는 **매핑 모드(mapping mode)**라 한다 - 클라이언트의 영역의 좌측 상단이 (0,0)이고 아래로 증가하는 y축을 가진다. MoveToEx()는 GDI의 **CP(current point)**를 옮긴다. LineTo()는 CP에서 전달된 파라미터의 위치까지 선을 그린다.

WM_PAINT가 아닌 메시지에서 DC를 얻는 API 함수는 GetDC()이다. GetDC()가 할당한 DC 구조체를 해제하는 함수는 ReleaseDC()이다. 이곳에서 DC를 얻기 위해 BeginPaint()를 사용하면 안 되는가? 안 된다. 그렇다면 WM_PAINT메시지를 처리할 때 GetDC()를 사용하면 안 되는가? 안 된다. 그 이유를 이제부터 알아보자.

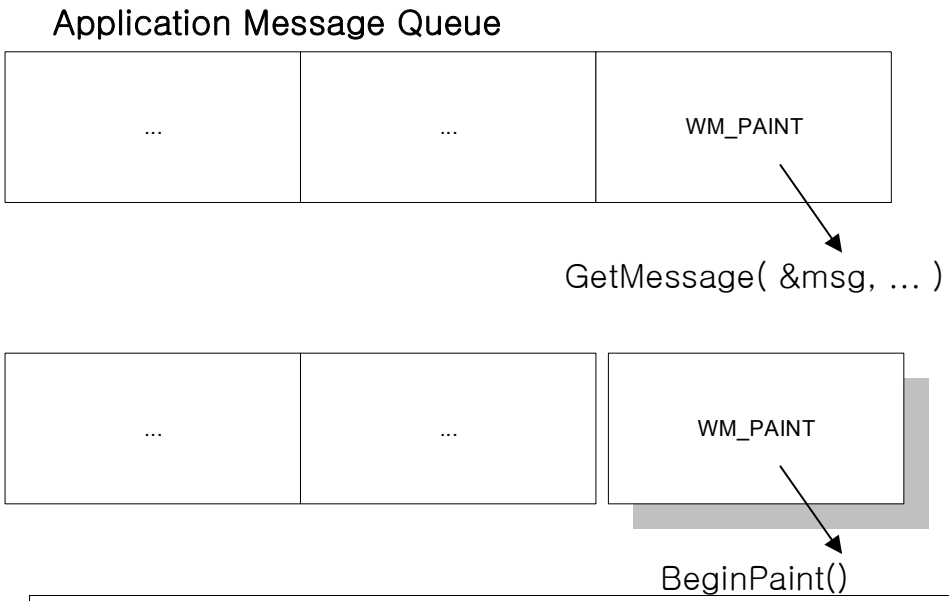


[그림 1.14] 메시지의 삭제: GetMessage()는 메시지 큐에서 메시지를 제거한다. 제거된 메시지는 GetMessage()의 첫번째 파라미터에 전달되어 넘어온다.

GetMessage()는 대부분의 경우 메시지 큐에서 메시지를 가져오고 큐에서 메시지를 지운다. 하지만, 메시지가 WM_PAINT인 경우 GetMessage()는 메시지를 큐에서 지우지 않는다. 큐에서 WM_PAINT메시지를 지우는 일은 그리기가 시작되는 시점 즉, BeginPaint()에서 한다. 이것은 WM_PAINT메시지를 처리할 때, DC를 얻기 위해 반드시 BeginPaint()를 호출해 주어야 하는 이유이다. 비록 그리기 작업이 필요 없더라도, 아래의 코드는 WM_PAINT 메시지에서 반드시 호출해 주어야 한다([그림 1.15]).

```
case WM_LBUTTONDOWN:
    BeginPaint();//(1)
    EndPaint();
    break;
```

윈도우 프로시저의 WM_PAINT에서 BeginPaint()를 호출하면 메시지 큐의 WM_PAINT메시지는 삭제된다(1). 만약 BeginPaint()를 사용하지 않고, GetDC()를 사용하면, DC는 제대로 얻어진다. 하지만, GetDC()는 메시지 큐에서 WM_PAINT메시지를 지우지 않는다. 그러므로, 메시지 루프는 계속해서 메시지 큐에 남아 있는 WM_PAINT메시지를 꺼내 올 것이고, 윈도우 프로시저는 계속해서 WM_PAINT메시지를 처리할 것이므로 클라이언트 영역을 무한히 계속 그리게 될 것이다. 아래 [그림 1.15]를 보자.



[그림 1.15] WM_PAINT메시지의 특별한 동작: WM_PAINT메시지가 메시지 큐에 있을 때, GetMessage()는 메시지를 제거하지 않는다. WM_PAINT메시지를 제거하는 곳은 BeginPaint()호출에서이다. 그러므로 WM_PAINT메시지를 처리하는 곳에서는 반드시 BeginPaint()를 호출해 주어야 한다. 그렇지 않으면 윈도우 프로시저는 제거되지 않는 WM_PAINT메시지를 계속 받게 된다. 또한, WM_PAINT메시지가 아닌 곳에서 DC를 얻기 위해 BeginPaint()를 사용하는 것은 메시지 큐에서 WM_PAINT를 지울 수 없으므로 타당한 DC를 얻지 못하게 된다.

반드시 기억하여 두자. WM_PAINT 메시지에서는 BeginPaint()로 DC를 얻고, 다른 메시지에서는 GetDC()를 사용해야 한다. 이것은 MFC(Microsoft Foundation Class)에서 BeginPaint()를 사용하는 DC와 GetDC()를 사용하는 DC 클래스가 다르게 구현되어야 하는 이유이다.

GetMessage() vs. PeekMessage()

메시지 루프에서 사용한 GetMessage()는 처리해야할 메시지가 없는 경우, 블록 상태□가 된다. 하지만, 큐에 메시지가 없는 경우에 즉시 리턴하고 계속

> <여기서잠깐>
> 블록 상태란 운영체제가 처리할 일이 없으므로 이벤트를 기다리는 상태를 말한다. 블록 상태의 CPU 점유률은 0%가 된다. 한 프로그램에서 처리할 메시지가 없는 경우 블록 상태가 되는 것은 멀티 태스킹 환경에서 다른 프로그램을 위해 필요한 일이다.
> </여기서잠깐>

해서 다른 일을 해야 하는 경우가 생긴다. 이 때는 PeekMessage()를 사용한다. PeekMessage()는 메시지를 꺼내 올 수도, 보기만 할 수도 있는데 두 경우 모두 즉시 리턴된다. 이제 메시지 루프를 다음 [예제 1.5]와 같이 바꾸어 보자.

[예제 1.5] Idle을 처리하는 메시지 루프

```
while ( 1 )
{
    if ( PeekMessage( &msg, NULL, 0, 0, PM_REMOVE ) )
    {
        if ( msg.message == WM_QUIT ) break;
        DispatchMessage(&msg);
    }//if
    static int  i = 0;
    char        buffer[8];
    wsprintf( buffer, "%i\n", ++i );
    OutputDebugString( buffer ); // (1)
}//while
```

F5를 눌러서 프로그램을 실행하면, 디버거 창에 증가하는 숫자가 계속 출력되는 것을 확인할 수 있다. 프로그램은 이제 Sleep()등을 써서 명시적으로 프로그램을 블록 상태로 만들지 않는 한, CPU를 점유한 상태에서는 블록 상태가 되지 않는다(윈도우즈 2000을 사용하는 경우, 태스크 매니저를 통해 보면, 이 프로그램이 CPU를 90% 이상 점유하고 있는 것을 확인할 수 있다).

OutputDebugString()은 디버거의 창에 문자열을 출력하는 API 함수로 비주얼 C++의 디버거 창에 문자열을 출력한다(1).

이 메시지 루프를 기억하여 두자. 왜냐하면 MFC의 메시지 루프가 이와 비슷한 구조로 되어 있기 때문이다.

요약

이 책에서는 MFC의 골격 코드를 만드는 데 이 장에서 설명한 코드를 기본으로 사용할 것이다. 따라서 이 장에서 제시한 윈도우 응용 프로그램은 이해하는 것만으로 부족하다. 책을 덮고 프로그램과 운영체제의 동작, 메모리에 구조체가 할당되고 해제되는 과정을 머리 속에서 따라갈 수 있을 정도로 외웠다면 다음 장으로 진행하도록 하자.

- Win32 프로그래밍에서 시작함수는 **WinMain()**이다. **Win32 API**란 운영체제가 DLL형태로 제공하는 함수들에 대한 인터페이스이다.
- **메시지 루프(message loop)**는 일반적으로 WinMain()에서 while문으로 작성되며, 응용 프로그램 메시지 큐에서 메시지를 가져와서(GetMessage()) 윈도우 클래스에 등록된 윈도우 프로시저를 호출해 달라는 요구(DispatchMessage())를 하는 논리로 구성된다.
- **핸들(handle)**이란 운영체제가 관리하는 제어 블록들을 유일하게 구분하는 ID로 일반적으로 정수(integer)이다.
- **윈도우 프로시저(window procedure)**는 윈도우 클래스에 등록된 콜백함수(callback function)로 메시지를 처리하는 첫 번째 함수이다.
- WM_PAINT 메시지를 처리할 때는 반드시 **BeginPaint()**를 호출해 주어야 한다.
- **PeekMessage()**는 GetMessage()와는 달리 큐에 메시지가 없는 경우 블록 상태가 되지 않는다.

[문서의끝]