다시 시작하는 윈도우 프로그래밍
기반 다지기

목차

목차 1
소개 1
연재 가이드 1
필자소개 1
필자 메모 1
Introduction 2
개발 환경 구성 3
테스트 프로젝트 4
미리 컴파일된 헤더 4
버전 매크로 7
타입 시스템 11
참고자료 12

소개

윈도우 프로그래밍을 하기 위해서 준비해야 할 사항들에 대해서 살펴본다. 컴파일러와 플랫폼 SDK를 설치하고 그것을 설정하는 방법. 미리 컴파일된 헤더, 버전 매크로, 타입 시스템과 같은 기본 적인 것들에 대해서 하나씩 살펴본다.

연재 가이드

운영체제: Windows XP
개발도구: Visual Studio 2005
기초지식: C/C++ 문법
응용분야: 윈도우 응용 프로그램

필자소개

신영진 pop@jiniya.net, http://www.jiniya.net
웰비아닷컴에서 보안 프로그래머로 일하고 있다. 시스템 프로그래밍에 관심이 많으며 다수의 PC 보안 프로그램 개발에 참여했다. 현재 데브피아 Visual C++ 섹션 시삽과 Microsoft Visual C++ MVP로 활동하고 있다. C와 C++, Programming에 관한 이야기를 좋아한다.

필자 메모

종종 신입 개발자들이 만든 코드를 검토하는 시간을 가져보곤 한다. 그럴 때면 묘한 현상 한 가지를 경험하게 되는데, 바로 개발자들이 자신이 만든 코드를 자기와 같은 존재로 취급한다는 점

이다. 코드의 문제점을 지적하면 마치 자신이 지적당한 것처럼 그 문제에 대해서 방어적이 된다. 또한 겉으로 표출되지 않는 문제에 대한 논리적인 결함에 대해서 이야기를 하면 그들은 항상 "원래 잘 되던 건데요.", 별 문제 없이 잘 동작하는데요." 같은 대답을 한다. 참으로 안타까운 현상이 아닐 수 없다.

현업에서 릴리즈 후에 겪게 되는 거의 대부분의 문제는 개발자가 테스트 하는 컴퓨터에서는 나타나지 않는다. 또한 대부분의 복잡한 동기화 문제 내지는 시스템 의존적인 문제는 테스트 컴퓨터 또한 정상적으로 통과한다. 진짜 잡기 어려운 문제들은 그러한 것들을 모두 통과하고 나서도 발생하는 예외들이다. 앞선 신입 개발자들과 같이 단지 돌아간다는 것에 만족해서는 그러한 문제를 해결할 수가 없다. 그들은 항상 자신들의 프로그램이 동작하지 않는 시스템을 가져다 주어야만 문제를 인정하고 해결한다. 그러면서 덧붙인다. "세상에 이런 시스템도 있네요." 그들에게 시스템의 그러한 부분을 보장한 사람은 아무도 없는데 말이다.

눈에 보이는 것만 가지고는 만족스러운 코드를 만들 수 없다. 안타깝게도 대부분의 개발자들이 싫어하는 이론적인 부분, 논리적인 부분을 꼼꼼히 생각해야만 만족스러운 코드를 만들 수 있다. 자신이 만든 코드를 보면서 '항상 이 케이스가 실패 한다면 어떻게 될까?' 내지는 '시스템의 이러한 부분은 항상 보장되는 것일까?'란 질문을 끊임없이 던져야 한다. 시스템이 보장을 해 준다고 선언한 것, 자신이 사용하는 라이브러리에서 보장해 준다고 선언한 것, 내지는 자신이 보장이 되어야 한다고 선언한 것 외에는 절대로 암묵적으로 그렇다고 생각하지 말아야 한다. 원래부터 그런 건 없기 때문이다. 인정받는 개발자가 되기 위해서는 눈에 보이는 것보다 더 많은 것을 볼 수 있는 차가운 두뇌는 필수 조건일 것이다.

Introduction

몇 년간 Visual C++ 커뮤니티 활동을 하면서 언젠가는 윈도우 프로그래밍이라는 주제로 글을 써보고 싶다는 생각을 했었다. 케케묵은 윈도우 프로그래밍이란 주제를 겁 없이 다시 꺼내는 이유는 너무도 많은 개발자들이 여전히 똑 같은 문제에 대해서 고민하고 있음에도, 그러한 것들을 설명하는 자료는 너무나 산재해 있기 때문이다. 늘 검색해 보라는 답변을 달지만 실상은 좁은 시야를 가진 초보적인 입장에서는 검색 또한 쉽지 않은 게 현실이지 않던가.

어쨌든 이러한 이유로 이 연재는 윈도우의 특정 컨트롤을 다룬다거나, 쉘의 특정 기능을 이용하는 방법에 대해서는 다루지 않을 것이다. 그것보다는 윈도우 시스템을 이루고 있는 근간에 대해서 다룰 예정이다. 따라서 내용의 상당수가 시스템 의존적이거나 컴파일러 지엽적인 문제에 대한 설명이 될 것이다. 이런 것들이 당장 뭔가를 만드는데 도움이 되진 않겠지만 어려운 문제에 봉착했을 때 근본적인 원인을 이해하는데 도움을 줄 수 있다고 생각한다.

이번 시간에는 그런 출발로 윈도우 프로그래밍 환경을 구성하고 기본적으로 윈도우 프로그래밍을 하기 전에 이해해야 하는 것들에 대해서 살펴보도록 하자.

개발 환경 구성

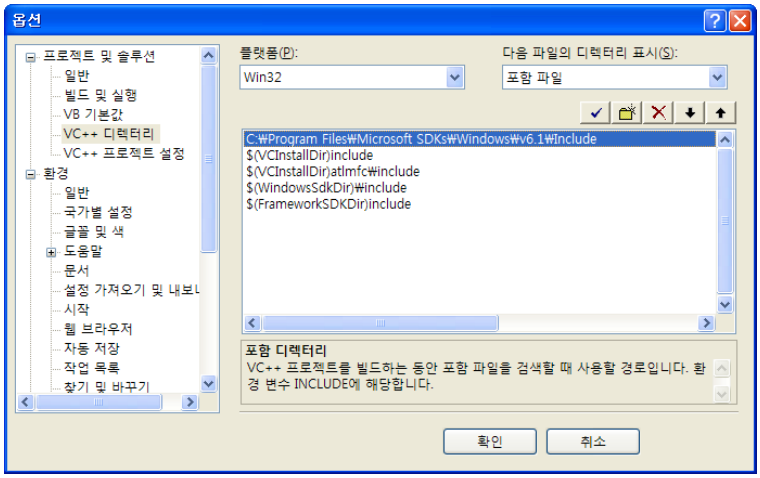
윈도우 프로그래밍을 배우기 위해서 가장 먼저 해야 할 작업은 개발 환경을 구성하는 일이다. 윈도우에서 C/C++을 사용해서 프로그래밍을 하기 위해서 필요한 것은 딱 두 가지다. 하나는 윈도우 실행 파일 포맷을 지원하는 컴파일러이고, 다른 하나는 윈도우 시스템 API들을 정의해둔 라이브러리다. 개발자를 장려하는 마이크로소프트의 정책 덕분에 이러한 것들은 별도의 비용을 지불하지 않고 구할 수 있다. 마이크로소프트의 최신 C/C++ 컴파일러가 탑재된 Visual C++ 2008 Express Edition과 최신 플랫폼 SDK를 설치해서 사용한다. 두 가지 프로그램을 아래 경로에서 다운로드 받아서 설치하도록 하자.

Visual Studio 2008 Express Edition
http://www.microsoft.com/express/vc/

플랫폼 SDK
http://www.microsoft.com/downloads/details.aspx?FamilyID=e6e1c3df-a74f-4207-8586-711ebe331cdc&displaylang=en

설치를 모두 마친 다음에는 Visual Studio에 플랫폼 SDK 경로를 등록해 주어야 한다. 최신 SDK의 경우 이 부분을 돕기 위한 별도의 등록 프로그램이 있긴 하나 종종 오류가 나는 경우도 있기 때문에 수동으로 경로를 설정하는 방법에 대해서 알아보도록 하자.

등록하는 방법은 간단하다. Visual Studio에서 도구 메뉴에서 옵션을 선택한다. 화면에 표시된 대화상자에서 프로젝트 및 솔루션 항목에 있는 VC++ 디렉터리 항목을 선택한다. 그러면 <화면 1>과 같은 대화상자가 표시된다. 여기서 새로운 항목을 만든 다음 플랫폼 SDK가 설치된 경로에 있는 include 폴더를 추가해준다. 마찬가지로 라이브러리 항목을 선택한 다음 lib 폴더도 추가해준다. 이 방법은 공용으로 사용하는 boost나 WTL같은 다른 라이브러리를 설치하는 경우에도 동일하게 적용된다.



화면 1 플랫폼 SDK 경로 등록

테스트 프로젝트

설정이 모두 제대로 되었는지를 검사하기 위한 간단한 테스트 프로젝트를 만들어 보자. firstapp란 이름으로 콘솔 프로젝트를 생성한다. 프로젝트를 만들고 나면 stdafx.cpp, firstapp.cpp, stdafx.h, targetver.h라는 네 개의 파일이 만들어진다. 각 파일의 기능은 <표 1>에 나와있는 것과 같다. stdafx.h와 firstapp.cpp 파일을 <리스트 1>과 <리스트 2>에 나타난 것과 같이 수정한 다음 컴파일해 보자. 컴파일과 실행이 정상적으로 이루어진다면 환경 설정이 제대로 이루어진 것이다.

표 1 각 파일의 기능

파일명	역할
stdafx.cpp	미리 컴파일된 헤더를 생성하는 역할을 한다.
firstapp.cpp	메인 프로그램을 구성한다.
stdafx.h	미리 컴파일된 헤더.
targetver.h	프로그램 구성에 사용되는 버전 매크로를 정의한다.

리스트 1 stdafx.h

```
#pragma once

#include "targetver.h"

#include <stdio.h>
#include <tchar.h>
#include <windows.h>
```

리스트 2 firstapp.cpp

```
#include "stdafx.h"

int main()
{
    printf("1초만 기다리세요...₩n");
    Sleep(1000);
    printf("완료.₩n");
    return 0;
}
```

미리 컴파일된 헤더

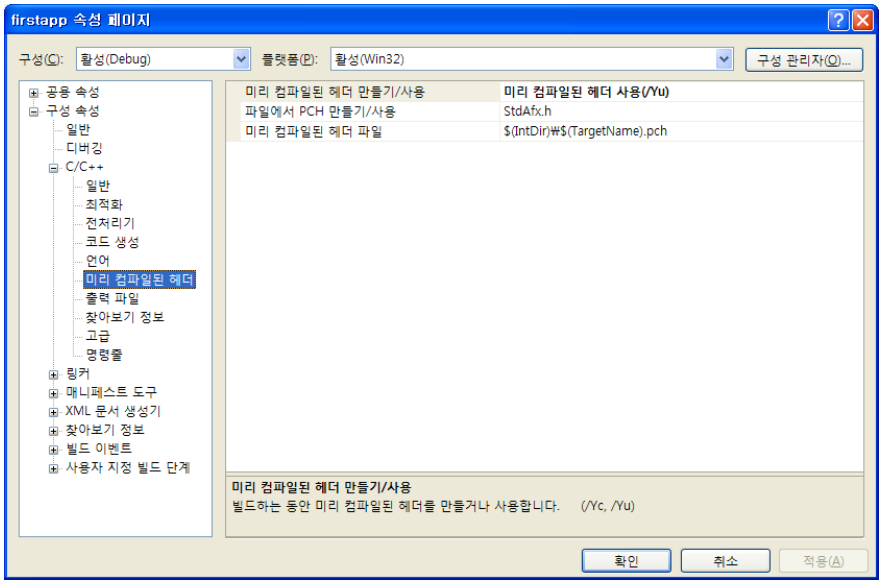
Visual Studio로 처음 프로젝트를 생성해서 프로그램을 만들다 보면 가장 먼저 생기는 의문은 #include “stdafx.h”에 있다. 처음에는 단순히 이 파일이 공통 헤더를 포함하기 위한 전초기지 정도로 생각한다. 하지만 새로운 소스 파일(.cpp)을 프로젝트에 추가하고 나면 그 파일엔 그것 이상의 의미가 있다는 것을 알게 된다. #include “stdafx.h”로 시작하지 않는 소스 파일은 컴파일이 되지 않기 때문이다.

Stdafx.h는 미리 컴파일된 헤더라는 Visual C++ 컴파일러 기능과 관련된 파일이다. 미리 컴파일된

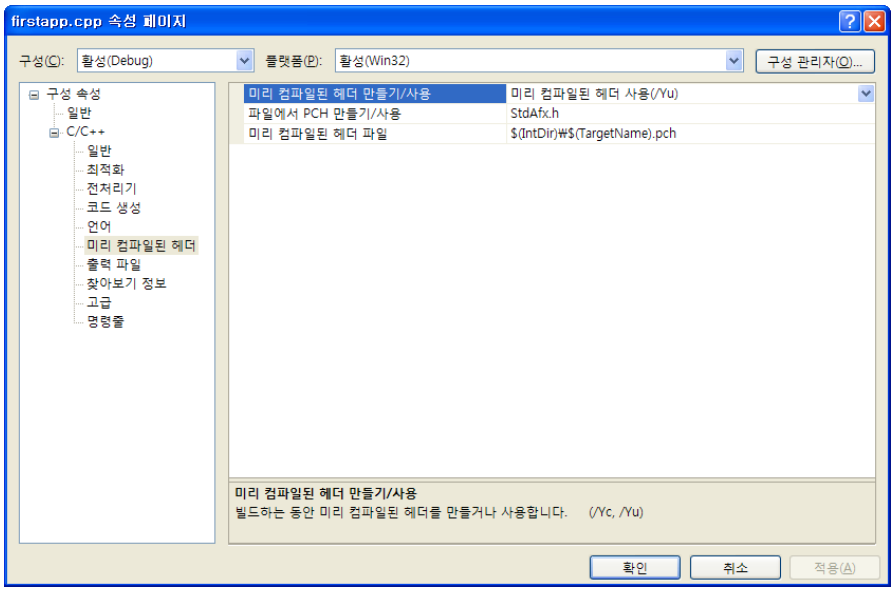
헤더란 자주 변경되지 않는 시스템 헤더를 미리 컴파일 해둠으로써 속도를 향상시키는 것을 말한다. Windows.h와 같은 헤더 파일은 크기도 크고 헤더 자체적으로 새롭게 포함하는 파일도 굉장히 많다. 따라서 헤더를 컴파일 하는데도 상당한 시간이 소요된다. Visual C++은 이러한 단점을 극복하기 위해서 크고 자주 변경되지 않는 시스템 헤더 파일을 미리 컴파일해두고 사용하는 방법을 제공한다. 이 기능의 이름이 미리 컴파일된 헤더다.

C/C++ 항목의 미리 컴파일된 헤더를 선택하면 <화면 2>와 같은 대화 상자가 나타난다. 미리 컴파일된 헤더 사용으로 되어 있기 때문에 이 프로젝트는 미리 컴파일된 헤더를 사용한다는 것을 알 수 있다. 이렇게 미리 컴파일된 헤더를 사용하는 프로젝트의 모든 소스 파일은(*.cpp) #include "stdafx.h"로 시작해야 한다. #include "stdafx.h"로 시작하지 않는다면 오류 메시지가 표시된다. 일반적인 경우에는 이러한 것은 문제가 되지 않지만 다른 사람이 만든 소스를 가져다 사용하는 경우에는 일일이 변경해주어야 하는 불편함이 따른다. 이런 경우를 피하기 위해서는 해당 파일의 속성을 열어서(<화면 3> 참고) 미리 컴파일된 헤더를 사용하지 않음으로 설정해 주면 된다.

끝으로 한 가지 더 주의해야 할 점은 stdafx.cpp의 미리 컴파일된 헤더 속성은 변경하지 않도록 해야 한다는 점이다. 미리 컴파일된 헤더를 사용하기 위해서는 특정 파일해서 사용하기 위한 pch 파일을 생성해야 한다. 기본적으로 이러한 역할을 담당하도록 되어 있는 파일이 stdafx.cpp다. 따라서 해당 파일의 옵션을 사용이나 사용하지 않음 등으로 변경할 경우에는 프로젝트가 정상적으로 컴파일되지 않는 문제가 발생한다.



화면 2 미리 컴파일된 헤더

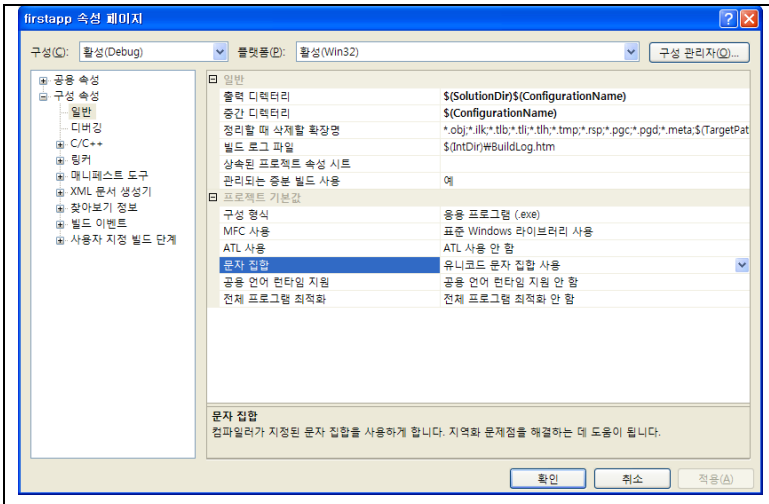


화면 3 firstapp.cpp의 설정

박스 1 컴파일 옵션

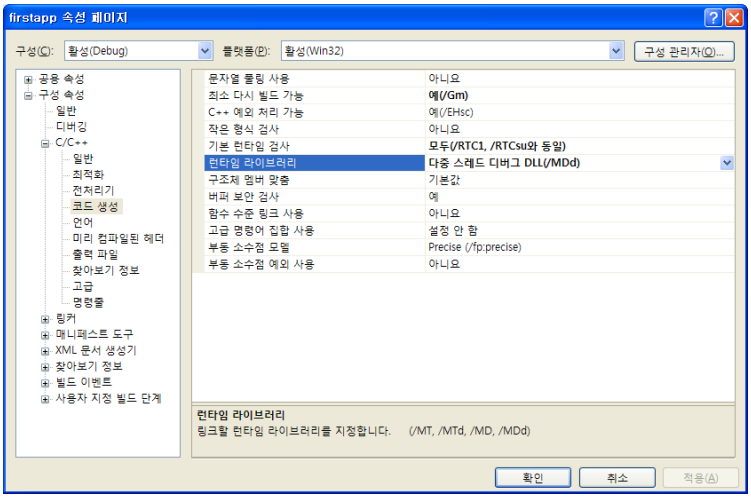
자주 질문하는 Visual Studio의 몇 가지 컴파일 옵션을 살펴보도록 하자. 프로젝트의 컴파일 옵션을 조정하기 위해서는 프로젝트를 선택한 다음 속성 메뉴를 클릭하면 된다. 그러면 <화면 4>와 같은 대화 상자가 나온다.

일반 탭(<화면 4> 참고)에서 가장 많이 문제가 되는 옵션은 MFC 사용과 문자 집합 부분이다. MFC 사용의 경우 MFC 프로그램을 만드는 경우에 MFC 라이브러리를 어떤 식으로 연결할지를 설정하는 부분이다. firstapp의 경우 MFC를 사용하지 않기 때문에 표준 Windows 라이브러리 사용으로 되어있다. MFC를 사용하는 경우 정적과 동적 중에 하나를 선택할 수 있다. 정적으로 연결하면 MFC DLL이 없는 컴퓨터에서도 잘 실행 할 수 있는 반면, 동적으로 연결하면 MFC DLL이 없는 컴퓨터에서는 MFC DLL을 설치한 다음에 실행을 해야 한다는 점이 차이다. 문자 집합의 경우에는 프로젝트에서 사용하는 문자 집합을 설정하는 옵션이다. 유니코드를 선택한 경우에는 기본적으로 모든 함수가 유니코드 버전으로 호출된다는 것을 의미한다. 멀티바이트를 선택하면 반대로 기본적으로 모든 함수가 멀티바이트 버전으로 호출된다는 것을 의미한다.



화면 4 일반 속성 탭

다음으로 CRT를 설정하는 부분에 대해서 알아보자. C/C++ 항목의 코드 생성 부분을 선택하면 <화면 5>과 같은 대화 상자가 나타난다. 여기서 살펴볼 부분은 런타임 라이브러리 항목이다. 이 또한 앞서 설명한 MFC와 마찬가지로 C/C++ 런타임 라이브러리를 어떤 식으로 연결할지를 결정하는 부분이다. 여기에도 크게 두 가지 옵션이 있다. 정적과 동적으로, MFC와 마찬가지로 정적을 선택하면 실행 파일에 런타임 라이브러리 함수가 모두 포함되므로 실행 파일의 크기가 커지고 런타임 라이브러리가 설치되지 않은 컴퓨터에서도 실행할 수 있다. 반면 동적을 선택할 경우에는 런타임 라이브러리가 설치되지 않은 컴퓨터에서는 정상적으로 실행되지 않는다.



화면 5 런타임 라이브러리

버전 매크로

targetver.h 파일은 윈도우 헤더에 컴파일된 프로그램이 동작할 윈도우 버전을 알려주는 역할을 한다. 이 파일에 기록된 내용은 윈도우 헤더에서 사용되도록 미리 약속된 버전 매크로다. firstapp에 기본적으로 만들어진 targetver.h 파일은 <리스트 3>과 같이 되어 있다.

리스트 3 targetver.h

```
#pragma once

#ifndef _WIN32_WINNT
#define _WIN32_WINNT 0x0600
#endif
```

현재 시중에 출시되어 사용되는 윈도우의 종류는 Windows 95, Windows 95 OSR2, Windows 98, Windows 98 SE, Windows Me, Windows NT 4.0, Windows 2000, Windows XP, Windows Vista, Windows Server 2008등 과 같이 그 수를 헤아리기 힘들 만큼 다양한 종류가 있다. 따라서 윈도우 프로그램을 작성할 때에는 항상 자신이 제작하고자 하는 프로그램이 어떤 환경에서 동작할지를 먼저 결정해야 한다. 윈도우의 경우에는 하위 호환성이 대부분 지켜지도록 발전해 왔기 때문에 이 경우에는 자신의 프로그램이 제대로 동작하기 위한 최소한의 환경을 설정하면 된다. Windows 2000을 최소 사양으로 잡았다면, 해당 프로그램은 Windows 95, 98, Me, NT 4.0에서는 동작하지 않을 수도 있지만, 2000 이후에 출시된 시스템에서는 모두 동작한다는 것을 의미한다.

이렇게 설정된 프로그램을 구동하기 위해서 요구되는 윈도우 시스템의 버전을 windows.h에 알려주는 역할을 하는 것은 버전 매크로다. <표 2>에는 이렇게 요구되는 시스템에 따라서 정의해 주어야 하는 버전 매크로의 종류가 나와있다. 앞서 소개한 기본 적으로 생성된 targetver.h는 Windows Vista를 최소 요구 시스템으로 설정하고 있다는 것을 알 수 있다.

표 2 요구 시스템에 따른 버전 매크로

최소 요구 시스템	매크로
Windows Server 2008	_WIN32_WINNT>=0x0600 WINVER>=0x0600
Windows Vista	_WIN32_WINNT>=0x0600 WINVER>=0x0600
Windows Server 2003	_WIN32_WINNT>=0x0502 WINVER>=0x0502
Windows XP	_WIN32_WINNT>=0x0501 WINVER>=0x0501
Windows 2000	_WIN32_WINNT>=0x0500 WINVER>=0x0500
Windows NT 4.0	_WIN32_WINNT>=0x0400 WINVER>=0x0400
Windows Me	_WIN32_WINDOWS=0x0500 WINVER>=0x0500
Windows 98	_WIN32_WINDOWS>=0x0410 WINVER>=0x0410

Windows 95	_WIN32_WINDOWS>=0x0400 WINVER>=0x0400
Internet Explorer 7.0	_WIN32_IE>=0x0700
Internet Explorer 6.0 SP2	_WIN32_IE>=0x0603
Internet Explorer 6.0 SP1	_WIN32_IE>=0x0601
Internet Explorer 6.0	_WIN32_IE>=0x0600
Internet Explorer 5.5	_WIN32_IE>=0x0550
Internet Explorer 5.01	_WIN32_IE>=0x0501
Internet Explorer 5.0, 5.0a, 5.0b	_WIN32_IE>=0x0500
Internet Explorer 4.01	_WIN32_IE>=0x0401
Internet Explorer 4.0	_WIN32_IE>=0x0400
Internet Explorer 3.0, 3.01, 3.02	_WIN32_IE>=0x0300

이러한 버전 매크로가 정확하게 어떻게 컴파일된 프로그램에 영향을 주는지를 알려주는 간단한 예를 살펴보도록 하자. <리스트 4>에는 REBARBANDINFOW라는 구조체가 나와있다. <리스트 5>에는 이 구조체를 사용하는 전형적인 예제 코드가 나와있다. 버전 매크로가 비스타로 설정된 상황에서 이 코드를 컴파일 했다고 생각해보자. 그러면 rbi.cbSize에는 비스타에서만 포함되는 두 필드 rcChevronLocation, uChevronState가 포함된 크기가 들어갈 것이다. 이렇게 컴파일된 프로그램을 Windows XP에서 실행한다면 cbSize 필드의 크기가 XP에서는 지원하지 않는 크기이기 때문에 SendMessage 호출은 실패한다. 따라서 프로그램이 XP에서는 정상적으로 동작하지 않는 것이다. 반면 버전 매크로가 XP로 설정된 상태에서 컴파일 되었다면 cbSize는 두 필드가 빠진 크기로 컴파일이 될 것이다. 이것은 XP에서도 정상적으로 동작할 뿐더라, 비스타에서 정상적으로 동작한다. 왜냐하면 하위호환성을 유지하기 위해서 비스타는 기본적으로 XP의 구조체 크기를 이해하기 때문이다.

리스트 4 REBARBANDINFOW 구조체

```
typedef struct tagREBARBANDINFOW
{
    UINT        cbSize;
    UINT        fMask;
    UINT        fStyle;
    COLORREF    clrFore;
    COLORREF    clrBack;
    LPWSTR      lpText;
    UINT        cch;
    int         iImage;
    HWND        hwndChild;
    UINT        cxMinChild;
    UINT        cyMinChild;
    UINT        cx;
    HBITMAP     hbmBack;
    UINT        wID;
#if (_WIN32_IE >= 0x0400)
```

```
    UINT        cyChild;
    UINT        cyMaxChild;
    UINT        cyIntegral;
    UINT        cxIdeal;
    LPARAM      lParam;
    UINT        cxHeader;
#endif
#if (_WIN32_WINNT >= 0x0600)
    RECT        rcChevronLocation;    // the rect is in client co-ord wrt hwndChild
    UINT        uChevronState; // STATE_SYSTEM_*
#endif
}   REBARBANDINFOW, *LPREBARBANDINFOW;
```

리스트 5 REBARBANDINFOW 구조체를 사용하는 예

```
REBARBANDINFOW rbi;

rbi.cbSize = sizeof(rbi);
//rbi.fMast = ...;
//rbi.fStyle = ...;

SendMessage(hwnd, RB_SETBARINFO, 0, (LPARAM) &rbi);
```

새로운 운영체제는 사용하기 편리한 API와 다양한 기능을 제공한다. 그러한 기능을 사용한다는 것은 이전 운영체제에서는 해당 프로그램이 정상적으로 동작하지 않는다는 것을 의미한다. 따라서 항상 윈도우 프로그램을 개발할 때에는 프로그램이 동작하는 최소한의 윈도우 버전을 정한 다음에 개발을 하는 것이 좋다.

박스 2 WIN32_LEAN_AND_MEAN

윈도우 헤더와 소켓 헤더를 동시에 포함하면 에러가 발생한다. 이러한 질문을 게시판에 올리면 윈도우 헤더를 포함하기 전에 다음과 같이 WIN32_LEAN_AND_MEAN을 선언해 주라는 답변이 올라온다.

```
#define WIN32_LEAN_AND_MEAN
#include "windows.h"
```

그렇다면 WIN32_LEAN_AND_MEAN은 무엇을 하는 것일까? WIN32_LEAN_AND_MEAN이 정의되어 있으면 windows.h는 암호화, DDE, RPC, 쉘, 윈도우 소켓 라이브러리를 포함하지 않는다. 소켓 헤더와 충돌이 발생했던 것은 windows.h가 winsock2 헤더가 아닌 예전 버전의 헤더를 먼저 포함했기 때문에 충돌이 발생했던 것이다. 그러면 왜 똑똑한 마이크로소프트 개발자들이 windows.h를 고치면 되는 문제를 고치지 않고 저런 요상한 정의를 하도록 만들었을까? 그것은 하위호환성을 유지하기 위해서다. windows.h를 고쳐버리면 기존에 멀쩡하게 잘 컴파일되던 소스 코드를 변경해야 할 수도 있기 때문이다. 결론적으로 컴파일 속도나 충돌 문제를 생각했을 때 windows.h를 포함하기 전에는 항상 WIN32_LEAN_AND_MEAN을 선언해 주는 것이 좋다고 할 수 있다.

타입 시스템

C/C++ 문법을 공부하고 처음 윈도우 프로그래밍을 접하는 사람들을 가장 놀래 키는 것 중의 하나는 방대한 타입 시스템이다. 더욱 재미있는 것은 다년간 윈도우 프로그래밍을 한 사람들 중에서도 그 타입 시스템이 정확하게 무엇인지를 모른다는 것이다. WPARAM이 무엇인지, LPARAM이 무엇인지, 왜 그런 타입을 정의해 놓은 것인지 정확하게 이해할 필요가 있다.

타입을 재정의 하는 방법은 간단하다. C/C++에서 제공하는 typedef라는 키워드를 사용하면 된다. 다음과 같이 정의한 다음부터는 int를 사용하는 대신 MYINT를 사용할 수 있게 된다.

typedef int MYINT;

그렇다면 굳이 int를 사용하면 되는데 왜 이렇게 타입을 새롭게 정의하는 것일까? 여기에는 두 가지 중요한 이유가 있다. 첫 번째 이유는 타이핑 스트레스를 줄이기 위함이다. 매번 unsigned long이라고 치는 것보단 ULONG이라고 치는 것이 짧고 편하기 때문이다. 두 번째 이유는 타입을 추상화 시키기 위해서다. 타입을 사용하는 개발자에게 그 타입이 무엇인지 신경 쓸 필요가 없도록 만들어주기 때문이다. MYINT가 추후에 시스템의 요구에 의해서 float로 변경된다고 하더라도 외부 개발자들은 여전히 MYINT를 사용하기만 하면 되기 때문이다. 실제로 MYINT가 무슨 타입인지에 대해서 신경 쓸 필요가 없다는 말이다. 이러한 연유로 윈도우는 엄청나게 많은 타입을 재정의 해두었다. 그 중에 자주 사용되는 타입의 형태에 대해서 살펴보도록 하자. <표 3>에는 윈도우에서 사용하는 타입 이름에 나타나는 각 문자의 의미가, <표 4>에는 자주 사용하는 타입의 원래 의미가 나와 있다. 보다 많은 타입에 대한 더욱 상세한 설명은 MSDN 페이지를 (http://msdn2.microsoft.com/en-us/library/aa505945.aspx) 참고하도록 하자

표 3 타입 이름에 사용되는 문자의 의미

글자	의미
T	문자를 나타내는 타입에 주로 등장하는 글자다. T가 의미하는 바는 유니코드와 ANSI의 컴파일 환경에 따라 적절히 변경된다는 것을 나타낸다. 유니코드 빌드라면 wchar_t로 변환되고, ANSI 빌드라면 char로 변환됨을 나타낸다.
L	long을 나타낸다. 주로 포인터 타입에 많이 등장한다. 16비트 시절은 포인터가 near, far등으로 구분해서 사용했다. 그 시절 관습에 따라 붙여진 것이다. 32비트 환경에서는 그러한 포인터 사이에 구분이 없기 때문이 L이 붙은 것과 붙지 않은 것 모두 같은 의미를 가진다.
C	const를 나타낸다. C가 들어간 자료형은 값을 수정할 수 없는 const로 선언된 것이라 이해하면 된다.
D	double을 나타낸다. 크기가 두 배라는 의미다.
U	unsigned를 나타낸다.

H	핸들을 나타낸다. 대부분 이 타입은 void *로 변환된다.

표 4 윈도우 프로그래밍에 자주 사용되는 타입의 의미

타입 명	설명
TCHAR	char형을 표현하는 타입이다. T가 붙었기 때문에 유니코드에선 wchar_t로, ANSI에서는 char로 변환된다.
PTSTR, LPSTR	문자열에 대한 포인터를 나타낸다. 유니코드에선 wchar_t *로, ANSI에서는 char *로 변환된다.
PCTSTR, LPCTSTR	수정이 불가능한 문자열 포인터를 나타낸다. 유니코드에선 const wchar_t *로, ANSI에서는 const char *로 변환된다.
PVOID	void 포인터를 나타낸다. void *로 변환된다.
PCVOID	상수 void 포인터를 나타낸다. const void *로 변환된다.
BOOL	TRUE/FALSE를 나타내는데 사용됨을 의미한다. 실제로는 int로 변환된다.
BYTE	8비트 무부호 정수를 나타낸다. unsigned char로 변환된다.
WORD	16비트 무부호 정수를 나타낸다. unsigned short로 변환된다.
DWORD	32비트 무부호 정수를 나타낸다. unsigned long으로 변환된다.
UINT	unsigned int로 변환된다.
USHORT	unsigned short로 변환된다.
ULONG	unsigned long으로 변환된다.
PWORD, LPWORD, PUSHORT, LPUSHORT	unsigned short * 변환된다.
HANDLE	파일, 뮤텍스, 세마포어, 이벤트등의 핸들을 나타낸다. void *로 변환된다.
HPEN	GDI PEN 핸들이다. void *로 변환된다.
HBITMAP	GDI BITMAP 핸들이다. void *로 변환된다.

참고자료

찰스 페졸드의 Programming Windows, 5th Edition
Charles Petzold, 한빛미디어

Windows API 정복
김상형, 가남사