다시 시작하는 윈도우 프로그래밍
핸들과 콜백 메커니즘

목차

소개

C언어는 객체 지향이란 말이 일상적으로 사용되기 이전에 설계된 언어다. 따라서 흔히 그런 언어에서 강조하는 다형성, 은닉성, 상속성 등의 개념을 표현하기 위한 언어적인 장치가 없다. 이러한 환경에서 개발자들이 그러한 것을 표현하기 위해서 자주 사용하는 방법이 핸들과 콜백이다. 윈도우의 대부분의 코드는 C로 작성되었기 때문에 수 많은 API가 C언어 기반으로 되어 있다. 따라서 이러한 메커니즘이 곳곳에서 드러난다. 이번 시간에는 핸들과 콜백의 개념과 그것들이 내부적으로 어떻게 구현되는지에 대해 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

프로그래밍이란 작업은 늘 버그를 동반한다. 어쩌면 개발이란 문제 해결의 연속인지도 모른다. 문제가 발생하지 않는 제품 개발은 없고, 늘 문제는 개발자를 그림자처럼 따라 다닌다. 그것들을 얼마나 빨리 얼마나 효율적으로 처리 하는가가 그 개발자의 능력이 되기도 한다. 버그 처리 시간을 줄이는 것 또한 개인의 생산성 향상에 높은 기여를 한다.

그렇다면 이러한 막히는 구간을 어떻게 하면 줄일 수 있을까? 개발자들을 곰곰 살펴보면 지나치게 지엽적인 문제에 과도하게 집착하는 것을 알 수 있다. 대부분의 개발자는 사소한 버그라도 발생하면 밤샘을 한다. 해결될 때까지 데스크탑 앞을 떠나지 않는 개발자도 있다. 과연 이러한 접근 방식이 도움이 될까?

몇 년간 개발자로 일하면서 내가 가지게 된 답안은 '아니오'다. 밤샘이나 집중은 일견 문제를 빨리 해결할 수 있게 만들어 주는 것처럼 보이지만 실제로 가장 골치 아픈 문제들은 저런 과정을 통해서 해결되지 않는 경우가 많았다. 특급 저질 버그들은 의외로 일상의 다른 활동을 – 샤워를 하거나, 화장실에 앉아 있거나, 설거지를 하는 등의 사소한 작업들 -- 하는 과정에서 생각난 아이디어가 단초가 되는 경우가 많았다. 이러한 것을 느낀 이후로 나는 가끔 진짜 골치 아픈 놈들을 만나면 의도적으로 설거지를 하거나 다른 활동으로 시선을 분산시키려는 노력을 하곤 한다.

아직까지 자신만의 버그 대처 법을 만들지 못한 개발자라면 오늘부턴 한번 골치 아픈 버그를 만날 때마다 설거지를 하는 걸 원칙으로 세워 보자. 의외로 자신의 버그 수정률이 높아지는 걸 느끼게 될지도 모른다. 물론 설거지는 한 가지 예일 뿐이다. 요는 집중된 흐름을 끊고, 주의를 환기시키는 작업이 필요하다는 것이다.

Introduction

여름이 다가온다. 여름 하면 난 할머니 댁에서 난생 처음 보았던 모기장이 떠오르곤 한다. 어렸을적 유달리 대청마루 모기장 속에서 노는 것을 좋아했다. 모기장은 이후 모기향으로, 모기향은 뿌리는 에프킬러로, 다시 뿌리는 에프킬러는 콘센트에 꽂아 만 두고 있어도 되는 홈키퍼로 발전했다. 홈키퍼가 제일 편한 건 사실이다. 하지만 아직도 여러 가지 이유로 모기장, 모기향, 에프킬러가 사용된다.

프로그래밍의 세계는 어떨까? 사람 사는 세상과 별로 다르지 않다. 기계어는 어셈블리 언어로, 어셈블리 언어는 컴파일러 언어로, 컴파일러 언어는 다시 점점 더 편한 언어로 발전했다. 하지만 아

직도 여러 가지 이유로 과거의 어셈블리 언어와 컴파일러 언어에서도 구식 취급을 받는 C언어가 사용된다.

우리가 사용하는 윈도우도 그러한 언어를 사용해서 구현되었다. 애석하게도 윈도우 API가 기반하고 있는 C언어는 객체 지향이 소개되기 전에 설계된 언어다. 따라서 객체 지향에서 말하는 복잡한 개념들을 위한 언어적인 메커니즘이 없다. C언어 개발자들은 이러한 환경에서 보다 높은 추상화와 다형성등을 구현하기 위해서 핸들과 콜백이라는 프로그래밍 테크닉을 자주 사용한다. 윈도우 API에도 이러한 기법이 광범위하게 사용되고 있다. 이번 시간에는 이러한 두 가지 개념에 대해서 살펴보고 실제로 어떻게 구현이 되는 것인지에 대해서도 알아 보도록 하자.

핸들이 뭔가요?

윈도우 프로그래밍을 처음 시작하면 제일 먼저 만나게 되는 것이 핸들이다. 윈도우를 생성하면 핸들이 반환된다고 한다. 브러시를 만들어도 핸들이 반환된다. 폰트를 생성해도, 스레드를 만들어도 핸들이 반환된다. 뭐만 나오면 죄다 핸들인 것이다. 이러한 개념에 익숙하지 않은 개발자들은 이게 과연 무엇을 의미하는지, 왜 이렇게 만든 것인지 의문이 생길 법도 하다. 조금 똘똘한 신입 개발자들은 HANDLE의 정의를 가지고 있는 헤더 파일을 찾아보곤 한다. 결국 그들이 만나는 정의란 다음과 같은 황당한 한 줄이다.

```
typedef void *HANDLE;
```

핸들 하면 가장 먼저 뭐가 떠오르는가? 아마 대부분의 사람들은 자동차를 떠올릴 것이다. 일상생활에서는 자동차의 운전석에 있은 그것을 핸들이라 부르기 때문이다. 프로그래밍 세상에서 말하는 핸들 또한 그것과 비슷한 개념이다.

자동차를 조종하기 위해서는 핸들이 있어야 한다. 핸들이 없다면 차를 운전하는 것 자체가 불가능하다. 이와 마찬가지로 윈도우에서 말하는 핸들도 특정 객체를 조작하기 위해서 사용된다. 개발자가 윈도우에게 부탁해서 얻어 온 핸들을 통해서만 해당 객체를 조작할 수 있다.

핸들을 생성하는 것은 자동차를 사는 행위에 비유할 수 있다. 메모리라는 대가를 지불하고 자동차와 같이 뭔가 이용 가능한 객체를 생성하는 것이다. 그리곤 그 객체에 해당하는 핸들을 반환받는다. 핸들을 닫는 것은 자동차를 폐차 시키는 것과 같다. 현실 세계와 컴퓨터 세상이 다른 한 가지 차이는 현실 세계의 차는 감가상각이 되는 반면 컴퓨터 세계에서는 핸들을 생성할 때 지불했던 메모리를 그대로 돌려받는 다는 차이만 있을 뿐이다.

메모리 할당을 이용한 방법

핸들을 구현하는 가장 전통적인 방법은 메모리 할당을 이용하는 것이다. 이 경우에 핸들은 할당

된 메모리의 번지가 된다. 동일한 프로세스의 컨텍스트에서 메모리 번지는 고유하다는 특징을 이용한 것이다. 더욱이 메모리 번지를 인덱스로 이용하게 되면 부가적으로 핸들을 참조하기 위한 오버헤드가 없기 때문에 효율적인 구현이 가능하다는 장점도 가지고 있다.

<리스트 1>에는 이러한 방법을 사용해서 LINE이라는 객체를 구현하는 방법이 나와 있다. 두 개의 함수 CreateLine과 CloseLine이 각각 LINE 핸들을 할당하고 해제하는 역할을 한다. 살펴 보면 알겠지만 new/delete가 해당 함수의 전부일 정도로 간단하다.

리스트 1 메모리 할당을 이용한 핸들 구현

```
typedef struct _LINE
{
    int sx; int sy;
    int dx; int dy;
    int width;
} LINE, *PLINE;

HANDLE CreateLine(int sx, int sy, int dx, int dy, int width)
{
    PLINE line = (PLINE) malloc(sizeof(LINE));
    if(!line)
        return NULL;

    line->sx = sx; line->sy = sy;
    line->dx = dx; line->dy = dy;
    line->width = width;
    return (HANDLE) line;
}

BOOL CloseLine(HANDLE h)
{
    __try
    {
        PLINE line = (PLINE) h;
        free(line);
        return TRUE;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
    }

    return FALSE;
}
```

앞서 살펴본 방법과 같은 핸들 구현의 가장 취약한 점은 핸들 값으로 잘못된 값이 넘어온 경우에 대한 예외 처리다. 앞선 코드의 CloseLine 함수에 잘못된 인자를 전달할 경우에 할당되지 않은 포

인터를 해제하려는 시도를 하기 때문에 치명적인 오류가 발생할 수 있다. 이러한 문제점을 해결하기 위한 코드가 <리스트 2>에 나와 있다. <리스트 2>의 코드는 포인터 인코딩이라는 방식을 통해서 핸들 값을 보호한다. 포인터 인코딩이란 주소 값을 특수한 값과 마스킹을 해서 변조 시키는 것을 말한다. 이러한 식으로 변경할 경우의 장점은 주소 값이 특이하기 때문에 일반적으로 프로그램에서 사용하는 주소 값과 구분이 쉽다는 장점이 있다. 물론 잘못된 값이 전달되는 경우에 취약한 것은 마찬가지 이지만 크래시가 발생한 경우에 디버깅이 좀 더 용이하다는 장점이 있다.

리스트 2 포인터 인코딩을 이용한 핸들 값 보호

```
#define LINE_XOR_VALUE 0x13942578

HANDLE XorPtr(PVOID ptr, ULONG_PTR sig)
{
    return (HANDLE)(((ULONG_PTR) ptr) ^ sig);
}

HANDLE CreateLine(int sx, int sy, int dx, int dy, int width)
{
    PLINE line = (PLINE) malloc(sizeof(LINE));
    if(!line)
        return NULL;

    line->sx = sx; line->sy = sy;
    line->dx = dx; line->dy = dy;
    line->width = width;
    return (HANDLE) XorPtr(line, LINE_XOR_VALUE);
}
```

<리스트 2>의 코드가 <리스트 1>의 코드보다 디버깅이 용이하긴 하지만 여전히 크래시가 발생할 가능성은 있다. <리스트 3>에는 이러한 방법을 줄이기 위해서 매직 넘버라는 기법을 사용한 코드가 나와 있다. 매직 넘버는 특수한 값을 설정해서 그 데이터가 맞는지를 검증하는 기법이다. LINE 구조체의 앞쪽에 특수한 값을 기록해 놓고 CloseLine에서는 그 기록된 값이 동일하지 않을 경우에는 잘못된 파라미터 전달로 간주하는 것이다. 물론 여전히 잘못된 포인터가 동일한 매직 넘버를 가지고 전달되는 경우에는 오류가 발생할 수 있다. 하지만 이러한 확률은 일반적으로 극히 낮기 때문에 제법 신뢰성 있다고 할 수 있다. 이 또한 프로그래밍 세계에서 흔히 사용되는 테크닉 중에 하나다.

리스트 3 매직 넘버를 사용한 핸들 보호

```
#define LINE_MAGIC 'enil'
#define LINE_XOR_VALUE 0x13942578

typedef struct _LINE
{
    ULONG magic;
```

```
    int sx; int sy;
    int dx; int dy;
    int width;
} LINE, *PLINE;

HANDLE CreateLine(int sx, int sy, int dx, int dy, int width)
{
    PLINE line = (PLINE) malloc(sizeof(LINE));
    if(!line)
        return NULL;

    line->sx = sx; line->sy = sy;
    line->dx = dx; line->dy = dy;
    line->width = width;
    line->magic = LINE_MAGIC;
    return (HANDLE) XorPtr(line, LINE_XOR_VALUE);
}

BOOL CloseLine(HANDLE h)
{
    __try
    {
        PLINE line = (PLINE) XorPtr(h, LINE_XOR_VALUE);
        if(line->magic != LINE_MAGIC)
            return FALSE;

        free(line);
        return TRUE;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
    }

    return FALSE;
}
```

핸들 테이블을 이용한 방법

앞서 우리는 메모리 할당을 이용한 핸들의 구현 방법에 대해서 살펴보았다. 이 방법은 구현하기 쉽고 빠르다는 장점은 있으나 신뢰성은 떨어진다는 단점이 있었다. 잘못된 값을 함수로 전달하는 것에 취약한 것이다. 이러한 단점을 개량한 것이 핸들 테이블을 이용한 방법이다. 핸들 테이블을 이용한 방법은 테이블을 구성해서 생성한 목록을 관리한다. 이 경우에 핸들 값은 해당 테이블에서 고유한 핸들을 찾기 위한 키가 된다. 이 방법은 구현이 어렵고, 핸들 조회를 위한 추가적인 비용이 든다는 단점이 있지만 높은 신뢰성을 가진다는 점이 장점이다. 또한 핸들 테이블을 사용하게 되면 할당된 모든 핸들의 목록을 가지고 있기 때문에 한번에 모든 핸들을 제거한다거나 할당된 핸들을 추적할 수 있다는 장점도 있다.

실제로 핸들 테이블을 구현하는 코드를 살펴보도록 하자(<리스트 4> 참고). 핸들 테이블에는 테이블 구현을 위한 어떠한 방법을 사용해도 된다. 일반적으로는 해시 테이블이 많이 사용된다. 여기서는 구현을 간단히 하고 핵심 메커니즘을 파악하기 쉽게 하기 위해서 간단한 배열을 사용해서 구현했다. 고정 크기 배열을 사용했기 때문에 핸들의 할당 개수에 제한이 있다는 단점이 있다. 여기서 핸들은 배열의 인덱스 값이 된다.

리스트 4 핸들 테이블을 사용한 구현

```
#define LINE_MAX
LINE *g_lines[LINE_MAX] = { NULL, };
int g_lineCount = 0;

HANDLE CreateLine(int sx, int sy, int dx, int dy, int width)
{
    if(g_lineCount >= LINE_MAX)
        return NULL;

    PLINE line = (PLINE) malloc(sizeof(LINE));
    if(!line)
        return NULL;

    line->sx = sx; line->sy = sy;
    line->dx = dx; line->dy = dy;
    line->width = width;

    for(int i=0; i<LINE_MAX; ++i)
    {
        if(g_lines[i] == NULL)
        {
            ++g_lineCount;
            g_lines[i] = line;
            return (HANDLE) i;
        }
    }

    free(line);
    return NULL;
}

BOOL CloseLine(HANDLE h)
{
    __try
    {
        if(g_lineCount == 0)
            return FALSE;

        ULONG_PTR index = (ULONG_PTR) h;
        if(index >= LINE_MAX)
```

```
            return FALSE;

        if(g_lines[index] == NULL)
            return FALSE;

        free(g_lines[index]);
        g_lines[index] = NULL;
        --g_lineCount;
        return TRUE;
    }
    __except(EXCEPTION_EXECUTE_HANDLER)
    {
    }

    return FALSE;
}
```

핸들 사용의 모든 것

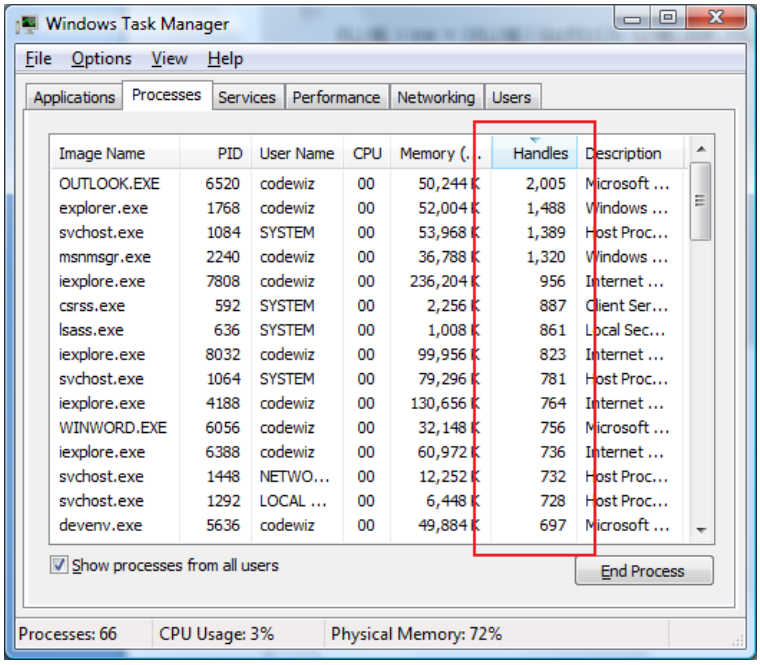
핸들의 사용은 크게 생성, 소멸, 조작 함수로 구성된다. C++이란 언어를 알고 있다면 이것들은 각각 생성자, 소멸자, 메소드에 해당하는 것들이다. 단지 C++은 이러한 것들을 자동적으로 처리해 주지만 C언어에서는 개발자가 일일이 모두 제어해야 한다는 것만 다를 뿐이다.

따라서 이러한 핸들 메커니즘에 접근하는 방법은 C언어 적인 함수 기반 틀보다는 객체 기반 틀로 접근하는 것이 용이하다. 즉, CreateThread, CreateEvent, CloseHandle 등과 같이 개별 함수의 사용법을 기준으로 접근하는 것보다는 프로세스 핸들의 생성 함수는 CreateProcess, OpenProcess 이고, 소멸 함수는 CloseHandle, 그리고 조작 함수는 VirtualAllocEx, VirtualQueryEx 등이 있다, 와 같은 방식으로 접근하는 것이 더 좋다는 의미다.

핸들을 사용할 때에 한 가지 기억해야 할 점은 핸들은 리소스라는 점이다. 따라서 핸들을 생성할 때에는 반드시 언제 핸들을 파괴할지를 생각해야 한다. 또한 적절한 핸들 파괴 함수로 파괴할 수 있도록 해야 한다. 초보 윈도우 개발자들이 가장 많이 저지르는 실수는 적절하지 않은 파괴 함수에 핸들을 전달하는 것이다. CreateProcess로 생성한 핸들은 CloseHandle을 통해 파괴한다. CreateMutext로 생성한 뮤텍스 핸들 또한 CloseHandle로 파괴한다. 이러한 메커니즘에 익숙하면 HeapCreate로 생성한 힙 핸들을 CloseHandle에 집어 넣는 실수를 하게 된다. HeapCreate로 생성한 핸들을 파괴하는 함수는 HeapDestory이다. 따라서 한 가지 핸들을 획득할 때에는 반드시 그것을 파괴하는 함수를 기억해 두어야 한다.

핸들이 적절한 시점에 파괴되지 않고 누수가 일어나는 경우에는 프로세스가 사용하는 리소스가 계속 늘어난다. 이것은 메모리 누수와 마찬가지로 프로그램에 있어서는 큰 문제다. 자신의 프로그램에서 사용하는 핸들의 개수에 대해서 알고 싶으면 윈도우의 작업 관리자를 실행하면 된다. 보기의 열 선택을 하면 핸들 항목이 있다. 해당 항목을 체크하면 <화면 1>과 같이 각 프로세스에

서 사용하고 있는 핸들 개수를 보여 준다.



화면 1 작업 관리자에 나타난 핸들 개수

콜백이 뭔가요?

콜백(call back)이란 영어를 그대로 직역하면 '역으로 호출한다'라는 의미다. 실제로 콜백은 그대로의 의미를 나타낸다. <그림 1>에 콜백 함수의 일반적인 호출 흐름이 나와 있다. funcA가 funcB를 호출하면서 callback1을 전달한다. 그러면 funcB는 필요한 시점에 callback1 함수를 호출해서 작업을 완료한다.

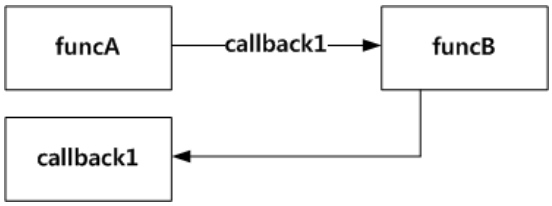


그림 1 콜백 함수의 호출 흐름

그렇다면 이러한 콜백 함수는 왜 사용하는 것일까? 여기는 크게 두 가지 이유가 있다. 한 가지 이유는 일반화 프로그래밍을 하기 위함이다. <그림 2>를 살펴보자. 콜백 메커니즘을 사용하는 sort 함수의 구조가 나와 있다. sortName과 sortYear 함수는 각각 이름과 년도를 기반으로 자료를 정렬하려는 함수다. sortName과 sortYear 함수는 각각 비교 함수만을 분리해서 sort 함수를 호출함으로써 목적을 달성한다.

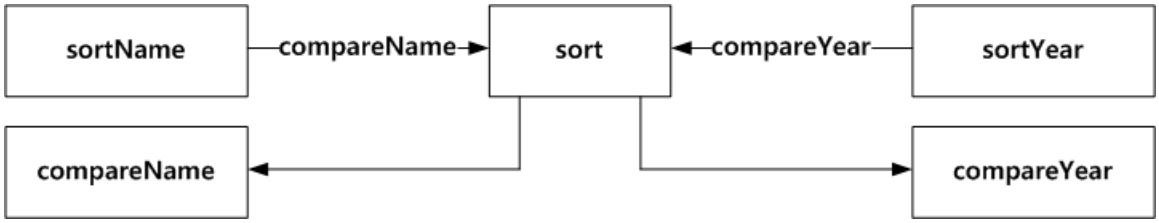


그림 2 sort 콜백 함수의 구조

이제 콜백 구조를 사용하지 않는 경우의 sortName과 sortYear 함수를 살펴보자. <그림 3>에 그 구조가 나와 있다. 이 경우에는 그림에 나와 있듯이 각각의 함수가 정렬 알고리즘 코드를 직접 포함하고 있어야 한다.

정렬 알고리즘을 교체하는 상황을 가정해 보자. <그림 3>과 같은 구조를 가진 경우에는 sortName과 sortYear에 포함된 함수를 모두 수정해야 한다. 반면에 콜백과 같은 구조에서는 실제로 정렬을 수행하는 sort 함수의 코드만 바꿔 주면 다른 부분을 수정하지 않고도 알고리즘을 바꿀 수가 있다. 구조가 좀 더 분명하고, 중복이 없기 때문에 수정이 용이하다.

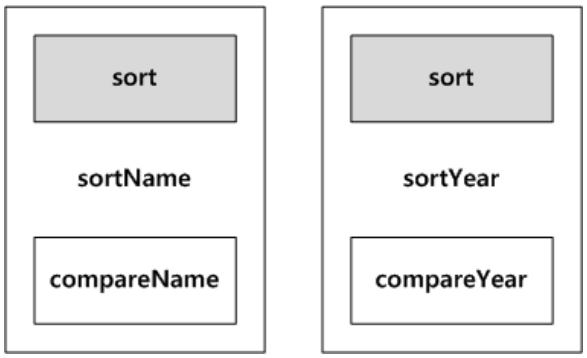


그림 3 콜백을 구현하지 않은 경우의 구조

콜백을 사용하는 두 번째 중요한 이유는 바로 함수의 동작이 완료되는 시점이 불분명하거나 미래인 경우다. 타이머가 대표적인 예다. 미래의 시점이나 불분명한 시점에 호출되는 함수는 개발자가 미리 해당 이벤트가 발생했을 때 할 일을 지정해 주어야 한다. 미리 할 일을 지정해 주지 않는다면 해당 이벤트가 발생할 때까지 무작정 대기할 수 밖에 없고, 그렇게 되면 정상적인 프로그램 처리를 할 수 없다. 이렇게 미리 지정해 주는 할 일이 콜백 함수가 되는 것이다.

콜백 함수 사용시 주의 해야 할 점

콜백 함수는 우리가 직접 제어할 수 없는 상황에서 호출되기 때문에 콜백 함수를 작성할 때에는 여러 가지 사항에 주의해야 한다. 가장 기초적인 사항은 콜백 함수의 원형을 정확하게 이해하는 것이다. 함수 원형이 다를 경우에 콜백을 호출하는 부분에서 오류가 발생할 수 있다. 윈도우 개발자들 사이에서 가장 많은 실수는 아무래도 호출 규약을 잘못 지정하는 것이다.

두 번째로 주의해야 할 점은 콜백 함수로 넘어오는 파리미터의 사용이다. 가장 먼저 파악해야 할 것은 콜백 함수로 넘어온 파라미터의 스코프다. 일반적으로 콜백 함수의 파라미터는 콜백 함수 내에서만 유용한 경우가 대부분이다. 따라서 그 파라미터가 추후에 필요하다면 반드시 별도로 복사해 두어야 한다. 특수한 콜백 함수의 경우에는 할당된 메모리를 파라미터로 전달하는 경우가 있다. 이런 경우에는 해당 메모리를 해제하는 것이 콜백 함수의 몫인 경우가 대부분이다..

세 번째로 주의해야 할 점은 콜백 함수의 리턴 값이다. 열거하는 용도로 사용되는 대부분의 콜백 함수들은 리턴 값을 통해서 콜백을 호출하는 루프를 간접 제어하는 형태를 취한다.

끝으로 가장 주의해야 하며, 개발자들이 가장 많이 하는 실수 중의 하나는 콜백 함수의 호출 컨텍스트에 관한 것이다. 콜백 함수는 우리가 직접 호출하는 함수가 아니다. 따라서 콜백이 어떤 스레드 컨텍스에서 호출되는지는 알 수 없다. 콜백을 전달한 스레드에서 호출될 수도 있고, 다른 스레드에서 호출될 수도 있다. 항상 같은 스레드에서 호출되는 것을 보장하는 콜백이 아니라면 임의의 스레드 컨텍스트에서 호출된다고 생각하는 편이 좋다. 따라서 콜백 함수에서 공유 접근하는 데이터가 있다면 반드시 락을 통해서 보호해야 한다.

콜백 함수 설계원칙

콜백 함수를 설계할 때에는 앞서 우리가 살펴보았던 주의 사항을 역으로 주입시키면 된다. 콜백 함수의 원형을 분명하게 지정하고, 파라미터의 사용 범위에 대해서 다른 개발자가 정확하게 이해할 수 있도록 코멘트를 한다. 리턴 값을 통해서 흐름을 제어하는 목적이 아니라면 리턴 값은 없는 형태를 사용한다. 끝으로 콜백 함수가 호출되는 컨텍스트를 분명히 지정해 준다.

이런 기본적인 설계 원칙에 더불어 한 가지 꼭 명심해야 할 것은 콜백 함수는 반드시 사용자 컨텍스트를 지정할 수 있도록 해야 한다는 점이다. 아래와 인터페이를 생각해보자. WalkDirectory 함수는 dir 폴더 안에 있는 파일을 열거하는 함수다. 각 개별 파일에 대해서 OnWalkDirectory 함수를 호출해서 path로 파일의 이름을 넘겨준다.

typedef BOOL (CALLBACK *OnWalkDirectory)(LPCTSTR path);
BOOL WalkDirectory(LPCTSTR dir, OnWalkDirectory callback);

위 함수를 사용해서 특정 디렉터리 안에서 abc로 시작하는 파일이 몇 개인지를 검사하는 코드를 작성한다고 생각해 보자. 자연스럽게 <리스트 5>와 같은 코드가 만들어 질 것이다. 이 코드에서 지적하고 싶은 문제점은 컨텍스트가 없기 때문에 어쩔 수 없이 전역 변수를 사용해야 한다는 점이다.

리스트 5 컨텍스트 변수를 지원하지 않는 콜백 함수를 사용하는 경우

```
int cnt;
```

```
BOOL CALLBACK FindAbcCountCallback(LPCTSTR path)
{
    // path가 abc로 시작하는 경우에 cnt 증가
}

int FindAbcCount(LPCTSTR dir)
{
    WalkDirectory(dir, FindAbcCountCallback);
    return cnt;
}
```

이러한 문제를 없애기 위해서는 반드시 콜백 함수로 컨텍스트 변수를 공급해 주어야 한다. 즉, WalkDirectory 함수를 다음과 같이 설계해야 한다는 이야기다. context 변수는 그냥 넘어온 값을 그대로 콜백으로 전달해 주는 것이다. 콜백 함수는 이를 용도에 맞게 변환해서 사용하면 된다. <리스트 6>은 이러한 구조를 사용하면 전역 변수를 사용하지 않아도 됨을 보여 준다.

typedef BOOL (CALLBACK *OnWalkDirectory)(LPCTSTR path, PVOID context);
BOOL WalkDirectory(LPCTSTR dir, OnWalkDirectory callback, PVOID context);

리스트 6 컨텍스트 변수를 지원하는 콜백 함수를 사용하는 경우

```
BOOL CALLBACK FindAbcCountCallback(LPCTSTR path, PVOID context)
{
    int *cnt = (int *) context;
    // path가 abc로 시작하는 경우에 cnt 증가
}

int FindAbcCount(LPCTSTR dir)
{
    int cnt;
    WalkDirectory(dir, FindAbcCountCallback, &cnt);
    return cnt;
}
```

도전 과제

<리스트 4>의 코드를 해시 테이블을 사용해서 구현해 보도록 하자. 실제로 윈도우 핸들은 여러 타입을 한 가지 통로를 제공하고 있는 것을 볼 수 있다. 이러한 것과 마찬가지로 LINE, RECT, TRIANGLE과 같은 객체를 생성하는 함수를 만들어 보자. 동시에 해당 객체들을 CloseShape를 통해서 할 수 있도록 만들어 보자. 또한 <리스트 6>에 사용된 WalkDirectory 함수를 직접 구현해 보도록 하자. FindFirstFile/FindNextFile을 사용하면 어렵지 않게 구현할 수 있다.

참고자료

찰스 페졸드의 Programming Windows, 5th Edition

Charles Petzold, 한빛미디어

Windows API 정복
김상형, 가남사