윈도우 프로그래머를 위한 PE 포맷 가이드
실행파일 생성기의 원리

목차

목차 1
소개 1
연재 가이드 1
연재 순서 1
필자소개 2
필자 메모 2
Introduction 2
컴파일러와 스텁(Stub) 프로그램 3
실행 파일에 데이터 합치기 4
텍스트 뷰어 스텁 프로그램 5
리소스에 데이터 추가하기 10
퍼즐 메이커 13
퍼즐 스텁 프로그램 16
도전 과제 19
참고자료 19

소개

요즘 대부분의 압축 프로그램은 자동 풀림 압축 파일을 지원한다. 자동 풀림 압축 파일이란 압축된 데이터와 함께 그것을 푸는 코드를 함께 가지고 있는 프로그램이다. 과거 DOS 시절에는 이런 형태의 대표적인 프로그램으로 파크 생성기가 있었다. 이번 시간에 우리는 실행 파일 생성기의 원리와 구조에 대해서 알아보고, 이 지식을 바탕으로 그림 파일로 퍼즐 게임을 생성해주는 퍼즐 메이커를 만들어본다.

연재 가이드

운영체제: 윈도우 2000/XP
개발도구: Visual Studio 2005
기초지식: C/C++, Win32 API
응용분야: 보안 프로그램

연재 순서

2007. 08. 실행파일 속으로
2007. 09. DLL 로딩하기

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

2007. 10. 실행 파일 생성기의 원리
2007. 11 코드 패칭

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

주연은 조연이 있기에 더욱 빛나고, 어둠이 있기에 빛이 있다는 것을 알 수 있다. 느린 것이 있기에 빠른 것의 가치를 알 수 있으며, 약한 것이 있기 때문에 강한 것의 존재를 알 수 있다. 전혀 다른 반대편이 있기에 자신의 존재가 부각되는 셈이다. 어둠이 없다면 빛도 없고, 약한 것이 없다면 강한 것도 존재할 수 없다.

딱딱한 가지는 바람결에 부서지지만 가녀린 갈대는 부서지지 않는다. 약하기 때문에 강한 것이다. 훌륭한 투수는 완급 조절을 잘 한다. 강속구도 느린 공 속에 숨어 있을 때 진정한 가치를 발휘하는 법이다. 무엇이든 한쪽에 치우치는 것은 좋지 않다.

이제 2007년도 몇 달 남지 않았다. 이맘때면 많은 사람들이 2007년의 계획을 돌아보고 성취하지 못한 계획들로 후회한다. 그리곤 바쁜 요즘 사람들은 남은 나머지 시간마저도 빼곡히 할 일로 채워놓는다. 여백이 있기에 채움이 있고, 여백이 있기 때문에 더 큰 것을 담을 기회가 있다는 것을 모르기 때문이다.

작은 일들에 일희일비하지 말고, 잠시나마 여유를 찾는 지혜를 가지도록 하자. 그 시간으로 인해 분명 더 큰 것을 담을 수 있는 기회를 찾을 수 있을 것이다. 쉬어가는 페이지 하나 없다면 삶이 너무 각박하지 않을까?

Introduction

예전에 DOS를 사용하던 시절에는 파크 프로그램이란 게 있었다. 파크 프로그램이 하는 일은 컴퓨터 전원을 끄기 전에 하드디스크의 헤드를 안전한 위치로 옮기는 것이다. 그 당시 굉장히 신기하게 생각했던 프로그램 중에 하나가 파크 생성기였다. 파크 생성기는 사용자가 넣은 이미지를 사용하는 파크 프로그램을 생성해 주는 유틸리티였다. 그 때 필자는 프린세스 메이커 그림으로 파크 프로그램을 만들어서 사용하곤 했다.

윈도우 시절로 바뀌면서 파크 프로그램은 아련한 추억이 돼버렸다. 그러나 아직도 파크 생성기의

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

원리는 여러 곳에서 사용되고 있다. 대표적인 예가 인스톨 프로그램과 자동 압축 풀림 파일이다. 인스톨 프로그램은 인스톨 데이터를 입력 받아서 그것을 설치하는 프로그램을 생성해내고, 자동 압축 풀림 파일은 데이터를 넣으면 압축하고 그것을 자동으로 풀 수 있게 exe 형태로 만들어준다. 결국 파크 생성기처럼 실행 파일을 생성해내는 것이다.

Visual C++ 게시판에는 종종 이런 실행 파일 생성기와 관련된 질문이 올라오곤 한다. 질문을 보면 많은 개발자들이 과거 필자가 생각했던 것과 같은 실수를 하는 것을 알 수 있었다. 이번 시간에는 이러한 실행 파일 생성기의 원리에 대해서 알아보고, 그 지식을 토대로 그림 파일로 퍼즐 게임을 생성해주는 퍼즐 메이커를 만들어본다.

컴파일러와 스텁(Stub) 프로그램

실행 파일을 생성하는 도구 중에 개발자들이 처음 접하는 것은 컴파일러다. 그래서 그런지 실행 파일 생성기를 만들어내는 방법에 관한 질문 속에는 컴파일러 책을 추천해달라는 이야기가 포함된 경우가 종종 있다. 이는 컴파일러와 실행 파일 생성기의 정확한 차이를 인식하지 못해서 생기는 오류다.

먼저 컴파일러에 대해서 살펴보자. 컴파일러란 사람이 이해하기 편하도록 만들어진 구문을 컴퓨터가 이해할 수 있는 바이너리 코드로 변환하는 프로그램을 말한다. 여기서 우리가 집중해야 하는 단어는 변환이다. 컴파일러는 소스 코드가 달라지면 그에 맞는 새로운 실행 파일을 만들어 낸다. 실행 파일은 단순히 소스 코드의 변환 결과물인 셈이다.

반면 실행 파일 생성기는 어떨까? 자동 풀림 압축 파일을 생각해 보자. 자동 풀림 압축 파일을 만들 때 마다 새로운 코드가 생성되는 것일까? 압축 파일에 맞는 코드가 변환되어서 생기는 것일까? 그렇지 않다. 헥사 에디터로 살펴보면 알겠지만 동일한 압축 프로그램으로 생성해낸 자동 풀림 압축 파일의 코드 부분은 모두 동일하다. 다른 여타의 실행 파일 생성기도 동일한 원리로 만들어진다. 이미 만들어진 코드에 데이터만 바꾸어가면서 생성해 내는 것이다. 이렇게 실행 파일 생성기에서 추후에 사용할 수 있도록 미리 만들어둔 프로그램을 스텁 프로그램이라고 부른다. 지금까지 복잡하게 설명한 내용을 그림으로 표시해 보면 <그림 1>과 같이 나타낼 수 있다.

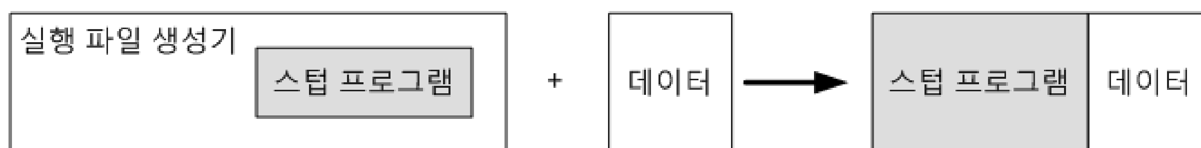


그림 1 실행 파일 생성기 원리

<그림 1>과 같은 구조의 프로그램을 만들기 위해서는 실행 파일 생성기 속에 어떻게 스텁 프로그램을 포함시키는지, 스텁 프로그램과 데이터는 어떻게 결합 시키는지, 스텁 프로그램은 자신의 데이터를 어떻게 찾아서 읽어 들이는 지에 관해서 알아야 한다. 각각에 대해서 좀 더 자세하게

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

살펴보도록 하자.

실행 파일에 데이터 합치기

실행 파일 속에 스텁 프로그램을 집어 넣는 것이나, 스텁 프로그램과 데이터를 합치는 일이나 모두 결국은 실행 파일에 데이터를 추가하는 작업이다. 이렇게 실행 파일과 데이터를 합치는 데에는 여러 가지 방법이 있다. 여기에서는 대표적인 세 가지 방식에 대해서만 살펴본다. 각각 특징과 장단점이 확실하기 때문에 각 방법을 어떤 경우에 사용하면 될지 생각하면서 살펴보도록 하자.

첫 번째 방법은 데이터를 리소스에 추가하는 하는 방법이다. 이 방법은 보통 컴파일 타임에 사용되는 방법으로 스텁 프로그램을 생성해내는 실행 파일 생성기에서 많이 사용된다. 실행 파일 자체를 바이너리 리소스로 포함시킨 다음 그것을 로드해서 사용하는 형태다. 윈도우에서 제공하는 리소스 API를 사용하면 이러한 작업을 손쉽게 할 수 있다. 리소스는 실행 파일과 함께 메모리에 같이 로딩되기 때문에 리소스에 추가된 데이터가 크고, 한번에 로드할 필요가 없는 경우에는 메모리 낭비가 발생한다는 단점이 있다.

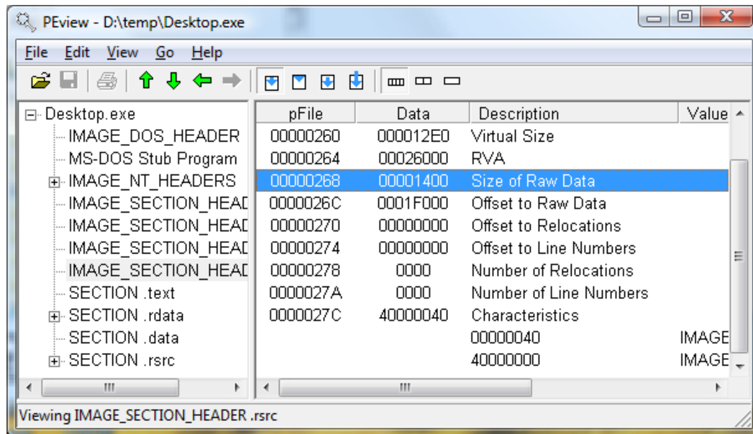
두 번째 방법은 새로운 섹션을 실행 파일에 추가해서 거기에 데이터를 추가하는 방법이다. 섹션 추가 방식은 컴파일 타임과 컴파일 이후에 모두 사용할 수 있다. 컴파일 타임에 섹션을 추가하기 위해서는 #pragma data_seg를 사용하면 된다. 완성된 EXE에 섹션을 추가하기 위해서는 별도의 유틸리티를 제작해야 한다. 이 방식의 가장 큰 장점은 실행 가능한 코드를 추가할 수 있다는 점이다. 반면 지원하는 API가 없기 때문에 이미 만들어진 EXE에 섹션을 추가하는 것은 직접 구현해야 한다는 단점이 있다. 섹션 또한 리소스와 마찬가지로 메모리에 같이 올라가기 때문에 메모리 낭비에 대한 문제는 그대로 남는다. 섹션 추가는 이미 만들어진 프로그램을 다시 컴파일하지 않고 추가적인 코드를 삽입하기 위해서 많이 사용된다.

마지막 방법은 실행 파일의 끝에다 임의의 데이터를 추가하는 방법이다. 이 방법은 무식해 보이지만 가장 쉽고, 많이 사용된다. 뒤쪽에 더미 데이터가 붙어 있더라고 PE 포맷은 섹션 단위로 데이터를 관리하기 때문에 추가된 데이터는 메모리로 로딩되지 않는다. 이런 특징 때문에 압축 프로그램과 설치 프로그램을 포함한 대부분의 실행파일 생성기의 스텁 프로그램이 이 방식을 사용한다.

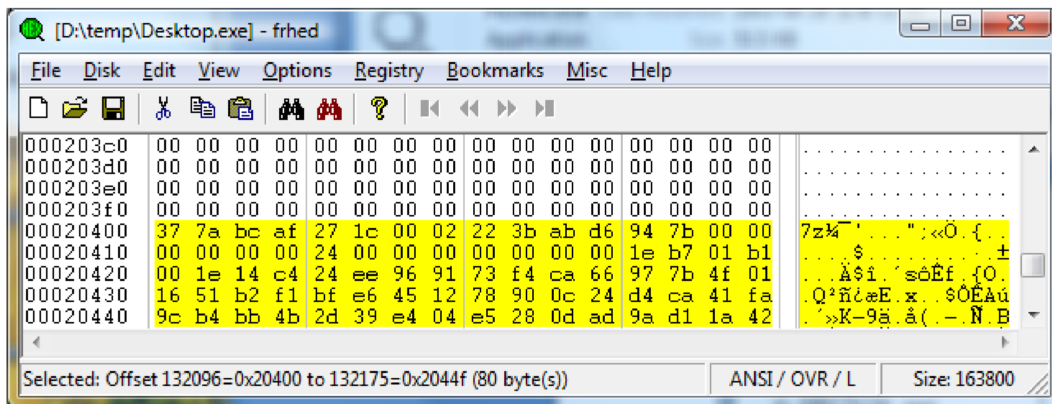
실제로 뒤쪽에 데이터를 붙이는 방법이 사용된 예를 한번 살펴보도록 하자. <화면 1>과 <화면 2>에는 7zip으로 생성한 자동 풀림 압축 파일에 대한 PE 헤더와 데이터가 나와있다. PE 헤더를 살펴보면 마지막 섹션인 rsrc 섹션의 파일 오프셋은 0x1f000이고, 크기는 0x1400이다. 따라서 실제 원래 실행 파일의 끝은 0x20400이다. 실행 파일에서 0x20400 부분을 살펴보면 압축 파일의 헤더가 시작되고 있는 것을 볼 수 있다. {0x37, 0x7a, 0bc, 0xaf, 0x,27, 0x1c} 는 7zip 압축 포맷의 매직 넘버다.

Copyright © 2011, 신영진

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/



화면 1 7zip으로 생성한 자동 풀림 압축 파일의 PE 헤더



화면 2 7zip으로 생성한 자동 풀림 압축 파일의 데이터

텍스트 뷰어 스텁 프로그램

텍스트 파일을 보여주는 간단한 형태의 스텁 프로그램을 만들어보자. 텍스트 파일 데이터는 앞서 소개했던 대로 파일 끝에 붙이는 방식을 사용한다. <리스트 1>에 텍스트 뷰어의 스텁 프로그램 코드가 나와있다. 스텁 데이터를 읽어 들이는데 사용하는 CStubDataReader에 대해서만 간단히 살펴보도록 하자.

CStubDataReader의 생성자는 스텁 데이터를 찾아서 읽어 들이는 일을 한다. 파일 뒤에 추가된 스텁 데이터를 읽어 들이는 데에는 크게 두 가지 방법이 있다. 하나는 DOS 헤더의 사용하지 않는 필드 값을 오프셋으로 사용하는 방법이고, 다른 하나는 PE 포맷 헤더를 분석해서 파일의 끝을 직접 계산해서 구하는 방식이다. CStubDataReader는 PE 포맷을 전혀 왜곡시키지 않는 두 번째 방법을 사용했다.

스텁 데이터의 오프셋은 앞서 살펴본 것과 같이 마지막 섹션의 PointerToRawData, SizeOfRawData 필드의 값을 더해서 구했다. 대부분의 경우 이 방식은 정상적으로 동작한다. 하지만 특수하게 최적화된 일부 PE 포맷에서는 잘못된 결과를 구하기도 한다. 원인은 링커가 최적화 과정에서 마지

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

막 섹션 다음에 데이터를 추가했기 때문이다. 이런 경우까지 처리하기 위해서는 DOS 헤더의 필드를 사용하거나 아니면 PE 포맷의 모든 데이터 오프셋을 뒤져서 가장 큰 번지를 사용하면 된다.

성공적으로 스텁 데이터를 읽어 들인 경우엔 CStubDataReader 클래스의 Offset, Size, Get 메소드를 사용해서 스텁 데이터의 오프셋, 크기, 실제 저장된 내용에 접근할 수 있다.

리스트 1 textviewer 소스 코드

```
#include <tchar.h>
#include <windows.h>
#include <vector>
#include <iostream>
using namespace std;

#ifdef _UNICODE
#define tcout wcout
#else
#define tcout cout
#endif

typedef std::vector<BYTE> ByteVec;
typedef std::vector<BYTE>::iterator ByteVIt;

#define ERROR_CODE(n) (n | 0x20000000)
const DWORD ERROR_NOSTUBDATA = ERROR_CODE(1);
const DWORD ERROR_CANTREADSTUBDATA = ERROR_CODE(2);
const DWORD ERROR_INCORRECTSTUBDATA = ERROR_CODE(3);

class EWin32
{
private:
	DWORD m_err;

public:
	EWin32(DWORD err) : m_err(err) {}
	DWORD Code() { return m_err; }
	operator DWORD() const { return m_err; }
};
```

Copyright © 2011, 신영진
이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
PVOID GetPtr(PVOID base, SIZE_T offset)
{
    return (PVOID)((DWORD_PTR) base + offset);
}

class CHandleCloser
{
private:
    HANDLE m_handle;

public:
    CHandleCloser(HANDLE handle) : m_handle(handle) {}
    ~CHandleCloser() { CloseHandle(m_handle); }
};

class CStubDataReader
{
private:
    ByteVec m_stub;
    DWORD m_size;
    DWORD m_offset;

public:
    CStubDataReader()
    {
        // 로드된 EXE의
        PIMAGE_DOS_HEADER dos;
        PIMAGE_NT_HEADERS nt;
        PIMAGE_SECTION_HEADER sec;

        dos = (PIMAGE_DOS_HEADER) GetModuleHandle(NULL);
        nt = (PIMAGE_NT_HEADERS) GetPtr(dos, dos->e_lfanew);
        sec = (PIMAGE_SECTION_HEADER) GetPtr(nt, sizeof(*nt));

        // PE 포맷의 끝을 찾는다.
        sec += nt->FileHeader.NumberOfSections-1;
```


이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
		m_offset = (*sec).PointerToRawData;
		m_offset += (*sec).SizeOfRawData;

		TCHAR path[MAX_PATH + 1];
		GetModuleFileName(NULL, path, MAX_PATH);

		HANDLE file;
		file = CreateFile(path
						, GENERIC_READ
						, 0, NULL
						, OPEN_EXISTING
						, 0, NULL);

		if(file == INVALID_HANDLE_VALUE)
			throw EWin32(GetLastError());

		CHandleCloser fileCloser(file);

		// 스텁 크기를 계산한다.
		DWORD readed = 0;
		DWORD filesize = GetFileSize(file, NULL);
		m_size = filesize - m_offset;
		if(m_size == 0)
			throw EWin32(ERROR_NOSTUBDATA);

		// 스텁 데이터를 읽어 들인다.
		m_stub.resize(m_size);
		SetFilePointer(file, m_offset, NULL, FILE_BEGIN);
		if(!ReadFile(file, &m_stub[0], m_size, &readed, NULL))
			throw EWin32(GetLastError());

		if(readed != m_size)
			throw EWin32(ERROR_CANTREADSTUBDATA);
	}

	// 스텁 데이터를 구한다.
	PVOID Get() const { return (PVOID) &m_stub[0]; }
```


이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
    // 스텁 데이터 크기를 구한다.
    DWORD Size() const { return m_size; }
    // 스텁 데이터 오프셋을 구한다.
    DWORD Offset() const { return m_offset; }
};

int _tmain(int argc, _TCHAR* argv[])
{
    try
    {
        tcout.imbue(locale("korean"));
        CStubDataReader stub;
        tcout.write((TCHAR *) stub.Get(), stub.Size());
    }
    catch(EWin32 &e)
    {
        switch(e.Code())
        {
        case ERROR_NOSTUBDATA:
        case ERROR_CANTREADSTUBDATA:
            tcout << _T("스텁 데이터가 손상되었습니다.₩n");
            break;

        default:
            tcout << _T("에러 코드: ") << e.Code() << endl;
            break;
        }
    }

    return 0;
}
```

프로그램을 실행시키면 "스텁 데이터가 손상되었습니다."라는 메시지가 표시된다. 스텁 데이터를 추가하지 않았기 때문이다. 프로그램에 스텁 데이터를 추가하기 위해서는 바이너리에 파일에 데이터를 추가시킬 수 있는 유틸리티가 필요하다. 여기서는 cygwin의 cat을 사용해서 바이너리 데이터를 추가해 보았다. <화면 3>은 cat을 사용해서 텍스트 데이터를 실행 파일에 추가하는 방법을 보여준다.

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
$ ls
hello.txt  textviewer.exe*  textviewer.ilk*  textviewer.pdb*
$ ./textviewer.exe
스텁 데이터가 손상되었습니다.
$ cat textviewer.exe hello.txt > hview.exe
$ chmod +x hview.exe
$ ./hview.exe
Hello World!!!

$
```

화면 3 textviewer 실행 화면

박스 1 실행 파일 크기를 줄이는 법

요즘은 메모리 크기나 파일 크기에 구애 받는 시대는 아니다. 수 기가에 달하는 파일을 인터넷으로 전송 받고, 과거 하드디스크 용량과 맞먹는 수 기가의 메모리가 실제 물리 메모리로 탑재되고 있기 때문이다. 하지만 아직도 몇몇 특수한 분야에 있어서는 파일 크기와 메모리 점유율은 여전히 큰 이슈가 되고 있다.

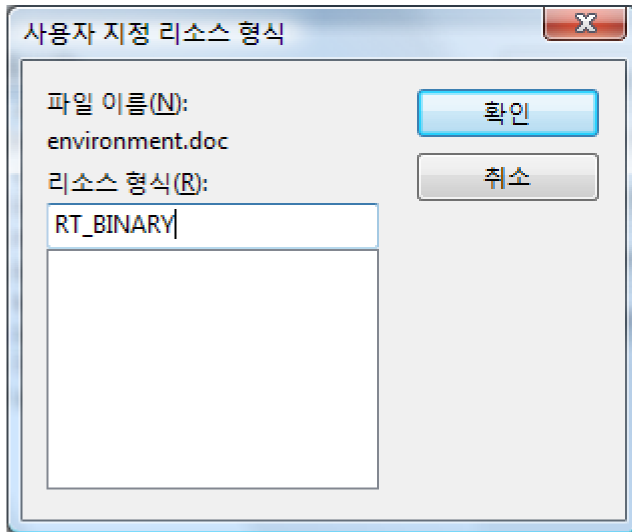
한번이라도 실행 파일 크기에 목숨을 걸어본 사람이라면 "실행 파일 크기가 왜 이렇게 클까?"라는 생각을 해본 적이 있을 것이다. 실행 파일 크기가 4K 이하로는 잘 내려가지 않기 때문이다. 여기에는 크게 두 가지 정도의 이유가 있다. 하나는 CRT 코드가 공간을 차지하는 것이고, 다른 하나는 파일 오프셋의 정렬 때문이다. CRT 코드가 차지하는 공간을 제거하기 위해서는 CRT 라이브러리를 링크 대상에서 제외시키면 된다. 대신 이 경우에 사용하는 CRT 함수는 모두 직접 새로 구현해 주어야 한다. 파일 오프셋의 정렬 문제를 해결하기 위해서는 링커 옵션인 /ALIGN을 사용해서 정렬 단위를 작게 만들어 주면 된다.

리소스에 데이터 추가하기

리소스에 데이터를 추가하는 방법은 간단하다. 리소스 에디터에서 추가를 선택한 다음 가져오기 메뉴를 선택한다. 그리고 추가할 바이너리 파일을 선택하고 확인을 누른다. 리소스가 인식할 수 없는 타입인 경우에는 리소스 형식을 물어본다(<화면 4> 참고). RT_BINARY와 같이 리소스에 맞는 적당한 이름을 넣어준다. 중요한 내용은 아니기 때문에 적당히 넣어주면 된다. 이렇게 추가된 바이너리 파일은 실행파일과 함께 컴파일되어서 실행 파일 내부의 리소스 섹션에 포함된다.

Copyright © 2011, 신영진

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/



화면 4 리소스 형식을 물어보는 대화상자

실행 파일에 들어있는 바이너리 리소스 데이터에 접근하기 위해서는 FindResource, LoadResource 함수를 사용한다. <리스트 2>에 이 함수들을 사용해서 리소스를 추출해내는 클래스인 CResourceExtractor의 소스가 나와있다. 가장 중요한 부분은 생성자로 넘어가는 인자다. 첫 번째 인자인 module은 리소스가 포함되어 있는 모듈의 핸들이다. 두 번째 인자인 id는 로드하려고 하는 리소스의 ID다. IDR_MAINFRAME과 같이 지정해 둔 리소스의 ID를 넘겨주면 된다. 마지막으로 넘어가는 type는 앞서 리소스 형식으로 지정했던 문자열을 넘겨주면 된다. 로드가 성공한 경우에는 Get, Size 메소드를 사용해서 리소스 데이터와 크기에 접근할 수 있다.

리스트 2 CResourceExtractor 클래스

```
class CResourceExtractor
{
private:
    DWORD m_size;
    HGLOBAL m_srcMem;
    PVOID m_ptr;

    CResourceExtractor(const CResourceExtractor &);
    CResourceExtractor &operator =(const CResourceExtractor &);

public:
    CResourceExtractor(HMODULE module, DWORD id, LPCTSTR type)
    {
        HRSRC src = NULL;
```

Copyright © 2011, 신영진
이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

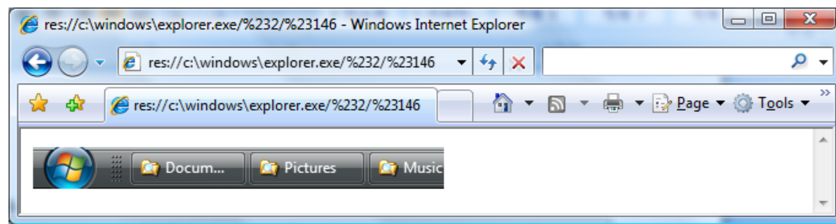
```
        src = FindResource(module, MAKEINTRESOURCE(id), type);
        if(!src)
            throw EWin32(GetLastError());

        m_size = SizeofResource(module, src);
        if(!m_size)
            throw EWin32(GetLastError());

        m_srcMem = LoadResource(module, src);
        if(!m_srcMem)
            throw EWin32(GetLastError());

        m_ptr = LockResource(m_srcMem);
        if(!m_ptr)
        {
            FreeResource(m_srcMem);
            throw EWin32(GetLastError());
        }
    }

    ~CResourceExtractor()
    {
        if(m_srcMem)
            FreeResource(m_srcMem);
    }

    // 리소스 크기를 구한다.
    DWORD Size() const { return m_size; }
    // 리소스 데이터를 구한다.
    PVOID Get() const { return m_ptr; }
    operator PVOID() const { return m_ptr; }
};
```

박스 2 res 프로토콜

Internet Explorer가 지원하는 프로토콜 중에 res 프로토콜이 있다. 이는 PE 포맷에 포함된 리소스에 접근해서 해당 리소스를 표시해주는 프로토콜이다. <화면 5>은 res 프로토콜을 사용해서 shell32.dll에 포함된 이미지를 브라우저를 사용해서 표시하는 것을 보여주고 있다.

Copyright © 2011, 신영진

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/



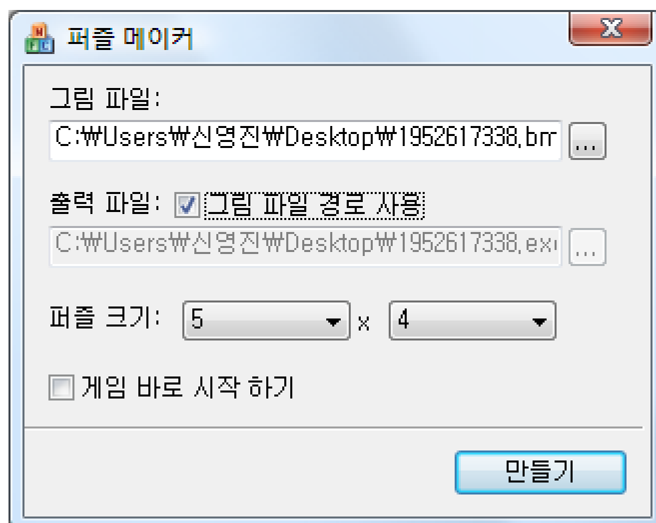
화면 5 res 프로토콜을 사용해서 explorer.exe에 포함된 이미지를 표시한 화면

res 프로토콜을 사용하는 방법은 간단하다. “res://파일명/타입/리소스ID”와 같이 적으면 된다. 여기서 타입과 리소스ID는 CResourceExtractor의 생성자 인자로 전달했던 것과 같은 의미다. 숫자로 표현하기 위해서는 #뒤에 아이디를 붙여주면 된다. 즉, “res://shell32.dll/#2/#130”은 shell32.dll에 포함된 리소스에서 타입이 2번이고, 아이디가 130번인 리소스를 찾아서 표시하라는 것을 지시한다. #은 브라우저에서 특수한 의미로 사용되기 때문에 %23으로 변환해서 넣어주어야 한다.

res 프로토콜을 리소스에 포함된 html 파일을 보여주기 위해서 많이 사용된다. res 프로토콜을 사용하지 않으면 같은 일을 하기 위해서 별도의 파일을 생성하거나, 웹브라우저 컨트롤에 직접 데이터를 기록하는 복잡한 코드를 작성해야 한다.

퍼즐 메이커

퍼즐 메이커는 그림 파일을 입력 받아서 퍼즐 게임을 생성해주는 프로그램이다. <화면 6>에 퍼즐 메이커의 실행 화면이 나와 있다. 입력으로 사용할 그림 파일 경로와 생성할 실행 파일 경로, 그리고 퍼즐에 사용할 조각 개수를 입력한 다음 만들기 버튼을 누르면 실행 파일이 생성된다.

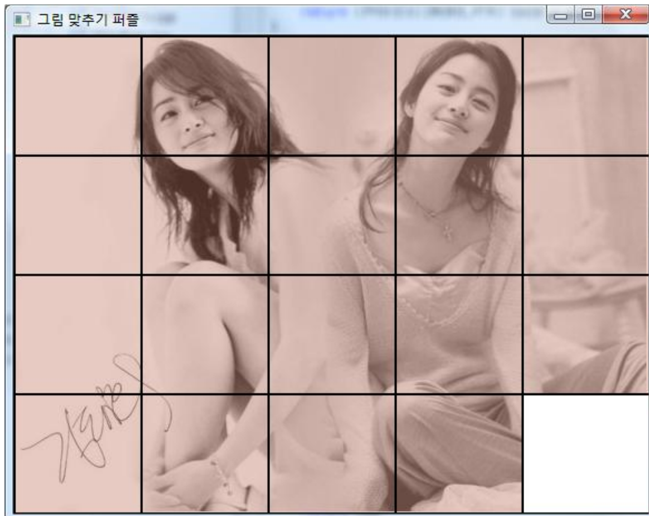


화면 6 퍼즐 메이커 실행 화면

생성된 파일을 실행 시키면 <화면 7>에 나타난 것과 같은 퍼즐 게임이 나온다. 어렸을 때 누구

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

나 한번쯤 해보았던 그림 맞추기 퍼즐이다. 방향키를 사용해서 조각들을 이리저리 움직여서 섞여 있는 것을 원위치로 옮기면 된다. 오른쪽 버튼을 누르면 메뉴가 나온다. 새 게임을 선택하면 새로운 게임을 시작할 수 있다. 하얀색으로 표시된 부분이 빈칸이다. 조각은 방향키를 사용해서 조작하면 된다.



화면 7 그림 맞추기 퍼즐 실행 화면

<리스트 3>에 스텁 데이터에 사용된 구조체가 나와 있다. 스텁 데이터는 결국 PUZINFOHEADER 형태로 이루어진다. 앞쪽에 헤더 정보가 있고, 뒤쪽에는 비트맵 파일이 그대로 붙여서 저장하도록 되어 있다. PUZINFOHEADER같은 구조체는 파일 I/O나 네트워크 패킷 처리할 때 빈번하게 사용된다. 처음 본다면 반드시 이 테크닉을 익혀 두도록 하자. 이 방법에 대한 자세한 설명은 C FAQ 2.6에 잘 나와있다(http://www.cinsk.org/cfaqs/html/node4.html#2.6 참고).

리스트 3 스텁 헤더 구조체

```
// 스텁 헤더 매직 넘버
const DWORD PUZ_MAGIC = 'abQt';

// 게임 바로 시작 플래그
const DWORD PUZFLAG_STARTNEWGAME = 1;

// 조각 개수 하한, 상한
const DWORD PIECE_MIN = 3;
const DWORD PIECE_MAX = 30;

// 스텁 헤더
typedef struct _PUZFILEHEADER
```


Copyright © 2011, 신영진

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
{
	DWORD magic;
	DWORD flag;
	SIZE piece;
} PUZFILEHEADER, *PPUZFILEHEADER;

// 스텝 정보
typedef struct _PUZINFOHEADER
{
	PUZFILEHEADER fh;
	BYTE bitmap[1];
} PUZINFOHEADER, *PPUZINFOHEADER;
```

만들기 버튼을 눌렀을 때 실행되는 코드가 <리스트 4>에 나와 있다. 이제껏 설명했던 내용대로 파일을 생성하는 것이 전부인 간단한 코드다. 파일 헤더를 만들고 쓰는 과정은 파일 처리를 할 때 빈번하게 사용되는 부분이기 때문에 익숙하지 않다면 잘 살펴보도록 하자.

리스트 4 퍼즐 게임 생성 부분

```
void CpuzmakDlg::OnBnClickedOk()
{
	try
	{
		// 그림 파일이 있는지 체크 한다.
		CString inPath;
		m_edtPath.GetWindowText(inPath);
		if(GetFileAttributes(inPath)==INVALID_FILE_ATTRIBUTES)
		{
			AfxMessageBox(IDS_ERRNOFILE);
			return;
		}

		// 그림 파일을 연다.
		ifstream in(inPath, ios::binary | ios::in);

		// 출력 파일을 생성 한다.
		CString outPath;
		m_edtOutPath.GetWindowText(outPath);
```


이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
		ofstream out(outPath
						, ios::binary | ios::out | ios::trunc);

		// 스텁 프로그램을 기록한다.
		CResourceExtractor re(AfxGetResourceHandle()
								, IDR_STUB
								, _T("RT_BINARY"));
		out.write((char *) re.Get(), re.Size());

		// 스텁 헤더를 만들어서 기록한다.
		PUZFILEHEADER fh;
		fh.piece.cx = m_cmbXPiece.GetCurSel() + PIECE_MIN;
		fh.piece.cy = m_cmbYPiece.GetCurSel() + PIECE_MIN;
		fh.magic = PUZ_MAGIC;
		fh.flag = 0;
		if(m_chkStartGame.GetCheck() & BST_CHECKED)
			fh.flag |= PUZFLAG_STARTNEWGAME;

		out.write((char *) &fh, sizeof(fh));

		// 이미지를 읽어서 기록한다.
		char buf[1024];
		for(;;)
		{
			in.read(buf, sizeof(buf));
			if(!in.gcount())
				break;

			out.write(buf, in.gcount());
		}

		AfxMessageBox(IDS_MAKECOMPLETE);
	}
}
```

퍼즐 스텁 프로그램

끝으로 퍼즐 스텁 프로그램의 핵심 코드를 간단하게 살펴보자. <리스트 5>에 퍼즐 스텁 프로그램

의 시작 부분 코드가 나와있다. 앞서 소개했던 CStubDataReader를 사용해서 스텁 데이터를 읽어들이는 것 외에는 별로 특별한 점이 없다. 함수 마지막 부분을 보면 AdjustWindow 함수를 사용해서 윈도우 크기를 조정하는 부분이 있다. 이는 윈도우 클라이언트 영역의 크기를 기준으로 전체 윈도우 영역을 계산해주는 함수다. 모르면 고생하기 쉬운 부분이기 때문에 꼭 익혀두도록 하자.

리스트 5 퍼즐 스텁 프로그램의 WM_CREATE 메시지 핸들러

```
int CChildView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    try
    {
        CStubDataExtractor stub;

        PPUZINFOHEADER hdr = (PPUZINFOHEADER) stub.Get();

        // 제대로 된 스텁 헤더인지 체크한다.
        if(hdr->fh.magic != PUZ_MAGIC)
            throw EWin32(ERROR_INCORRECTSTUBDATA);

        HBITMAP bmp = LoadBitmapFromBuffer(hdr->bitmap);
        if(!bmp)
            throw EWin32(GetLastError());

        m_bmp.Attach(bmp);

        // 퍼즐 데이터를 검증한다.
        if(hdr->fh.piece.cx < PIECE_MIN
            || hdr->fh.piece.cx > PIECE_MAX)
            hdr->fh.piece.cx = PIECE_MIN;

        if(hdr->fh.piece.cy < PIECE_MIN
            || hdr->fh.piece.cy > PIECE_MAX)
            hdr->fh.piece.cy = PIECE_MIN;
```

Copyright © 2011, 신영진

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
		// 퍼즐을 생성한다.
		m_puzzle.MakePuzzle(m_bmp, hdr->fh.piece);

		// 윈도우 크기를 조정한다.
		CSize windowSize = m_puzzle.Size();
		CRect rc(0, 0, windowSize.cx, windowSize.cy);

		CWnd *main = AfxGetMainWnd();
		AdjustWindowRect(&rc, main->GetStyle(), FALSE);
		main->SetWindowPos(&wndNoTopMost
						, 0
						, 0
						, rc.Width()
						, rc.Height()
						, SWP_NOMOVE | SWP_NOZORDER);

		// 새 게임 시작 플래그가 있는 경우 게임을 섞는다.
		if(hdr->fh.flag & PUZFLAG_STARTNEWGAME)
			OnStartgame();
	}
	catch(EWin32 &e)
	{
		switch(e.Code())
		{
		case ERROR_NOSTUBDATA:
			AfxMessageBox(IDS_ERRNOSTUBDATA);
			return -1;

		case ERROR_INCORRECTSTUBDATA:
			AfxMessageBox(IDS_ERRINCORRECTSTUBDATA);
			return -1;

		default:
			CString msg;
			msg.FormatMessage(IDS_ERRWIN32, e.Code());
			AfxMessageBox(msg);
			return -1;
```

Copyright © 2011, 신영진

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/

```
        }
    }

    return 0;
}
```

도전 과제

게임의 가장 큰 요소는 재미다. 우리가 만든 퍼즐 게임엔 재미라는 요소가 너무 없다. 게임성을 향상 시켜 보도록 하자. 타임 어택 모드나 움직임 횟수를 제한하는 모드를 만들어 보자. 사운드도 들어간다면 훨씬 재미있을 것이다. 친구 사진으로 게임을 만들어서 보내주는 것도 재미있는 이벤트가 될 것이다.

실행 파일 생성기에 대해서 좀 더 깊이 있게 연구해 보고 싶은 독자라면 스텁 프레임워크를 한번 제작해 보도록 하자. 우리가 만든 CStubDataReader 자체로도 스텁 처리에 범용적으로 사용할 수 있지만 기능이 많이 부족하다. 데이터를 검증할 수 있는 루틴을 추가하고, 사용자가 데이터를 편리하게 포맷팅해서 읽고 쓸 수 있도록 만들어보자.

다음 시간에는 이번 시간에는 이미 만들어진 프로그램의 코드를 변경시키고 새로운 코드를 추가하는 방법에 대해서 소개한다.

참고자료

What Goes On Inside Windows 2000: Solving the Mysteries of the Loader
http://msdn.microsoft.com/msdnmag/issues/02/03/Loader/

"Windows 시스템 실행 파일의 구조와 원리"
이호동저, 한빛미디어

An In-Depth Look into the Win32 Portable Executable File Format
http://msdn.microsoft.com/msdnmag/issues/02/02/PE/

An In-Depth Look into the Win32 Portable Executable File Format, Part 2
http://msdn.microsoft.com/msdnmag/issues/02/03/PE2/

Peering Inside the PE: A Tour of the Win32 Portable Executable File Format
http://msdn2.microsoft.com/en-us/library/ms809762.aspx

Copyright © 2011, 신영진

이 문서는 Creative Commons 라이센스를 따릅니다.
http://creativecommons.org/licenses/by-nc-nd/2.0/kr/