다시 시작하는 윈도우 프로그래밍
메모리 이야기 (1)

목차

소개

프로그램이란 메모리에 있는 데이터에 적절한 연산을 해서 결과를 만들어 내는 것을 말한다. 따라서 프로그래밍이란 작업을 하면서 가장 많이 접하고, 가장 신중히 다루어야 할 것은 다름 아닌 메모리에 저장된 데이터다. 이번 연재에서는 이러한 메모리 중에서도 스택과 힙의 구조와 특징에 대해서 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

우스운 일이지만 최근에 타이핑 실수로 인한 디버깅을 제법 했다. 한 가지는 'ninfo.data =

msi.data;'라고 쳐야 하는 부분을 'info.data = msi.data;'라고 친 것이다. 물론 해당 스코프에 info라는 객체도 있었기 때문에 코드는 정상적으로 컴파일이 되었고, 이 간헐적으로 실행되는 코드의 결과는 전혀 엉뚱한 증상으로 나타났다. 그 덕에 몇 시간 동안 디버깅을 한 후에야 겨우 버그를 찾을 수 있었다. 또 한번은 'XReport = _XReport'라고 쳐야 하는 부분을 'XReport = XReport'라고 타이핑한 것이다. 금방 크래시가 나서 잡을 수 있는 버그였으나, 우아하게 처리된 예외 핸들러들이 크래시를 삼키면서 제법 고생한 끝에 잘못 타이핑한 것을 알 수 있었다.

자주는 아니지만 이러한 디버깅을 하면서 드는 생각은 쫓기면서 만든 코드, 급하게 작성한 코드에는 그렇지 않은 코드들보다 훨씬 많은 버그가 있다는 것이다. 그럴 때면 '급할수록 돌아가라'는 옛말이 틀린 게 하나도 없음을 깨닫곤 한다. 언젠가 텔레비전 다큐멘터리에서 팔만대장경에 글을 새기던 사람들은 한 글자 한 글자 새길 때마다 부처님께 절을 드렸다는 이야기를 들은 적이 있다. 그들의 정성, 신중함 내지는 여유가 조금은 아쉬운 요즘이다.

Introduction

"스택을 너무 많이 써 서 그래요. 스택 크기를 늘려 보세요."
"힙에 할당된 메모리 해제를 안 하셨네요. 그러니 메모리 누수가 생기는 거잖아요."
"프로그램 속도를 높이려면 가상 메모리를 한번에 크게 할당해서 사용해 보세요."
"데이터 영역에 있는 값을 변경하니 메모리 오류가 발생하는 거예요"
"해당 번지는 유저 모드 프로그램이 접근할 수 없는 메모리예요."

흔히들 이런 말들을 한다. 하지만 처음 윈도우 프로그래밍을 접하는 사람들을 어리둥절하기 마련이다. 스택, 힙, 가상 메모리, 유저 모드, 데이터 영역 따위의 말들에 익숙하지도 않을뿐더러 그것들이 무엇을 의미하는 것인지 모르기 때문이다.

이번 시간에는 이러한 것들 중에서도 C/C++을 개발하면서 비교적 쉽게 접해보았을 법한 스택과 힙의 구조와 그것들을 사용하면서 주의해야 할 점에 대해서 알아보도록 한다.

스택 (Stack)

스택은 C/C++ 개발자가 의식하지 않는 과정에서 만나게 되는 가장 기본적인 데이터 저장 공간이다. C/C++의 코드에서 사용하는 자동 변수가 모두 기본적으로 스택에 저장되기 때문이다. 자동 변수가 저장되는 공간을 스택이라고 부르는 이유는 그 공간이 동작하는 방식이 자료구조에서 배우는 스택의 메카니즘과 동일하기 때문이다 <리스트 1>을 살펴보자. 코드에는 arr, score, i라는 지역 변수가 사용되었다. 이들이 저장되는 공간이 스택이다.

리스트 1 스택을 사용하는 코드

```
int GetSeq(int i)
{
```

```
    int arr[5] = { 0, 13, 17, 21, 5 };
    return arr[i % 5];
}

int main()
{
    int score[10];

    for(int i=0; i<10; ++i)
        score[i] = GetSeq(i);

    return 0;
}
```

<리스트 1>의 코드가 스택과 연동해서 어떻게 동작하는지가 <그림 1>에 나와 있다. 그림에 보여지는 것처럼 GetSeq 함수가 호출되면 스택에 arr 변수 저장을 위한 공간이 할당되고 스택의 탑(top) 포인터가 변경된다. 함수가 리턴되면 탑 포인터만 조작해서 해제한다. 이러한 동작방식에서 알 수 있듯이, 스택은 생명 주기가 결정되어 있는 작은 데이터의 빈번한 할당 삭제에 최적화 되어 있다는 것을 알 수 있다.

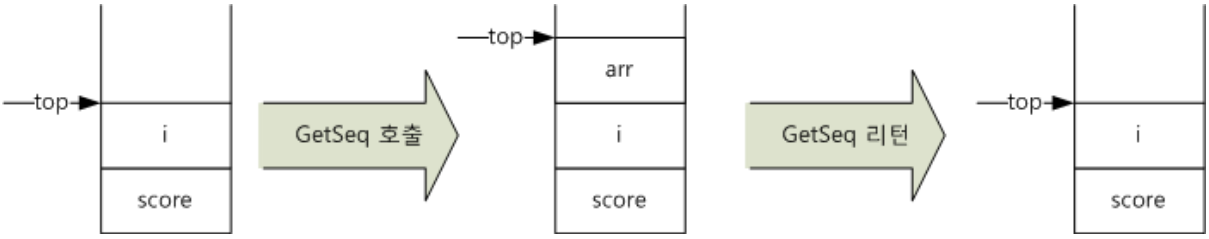
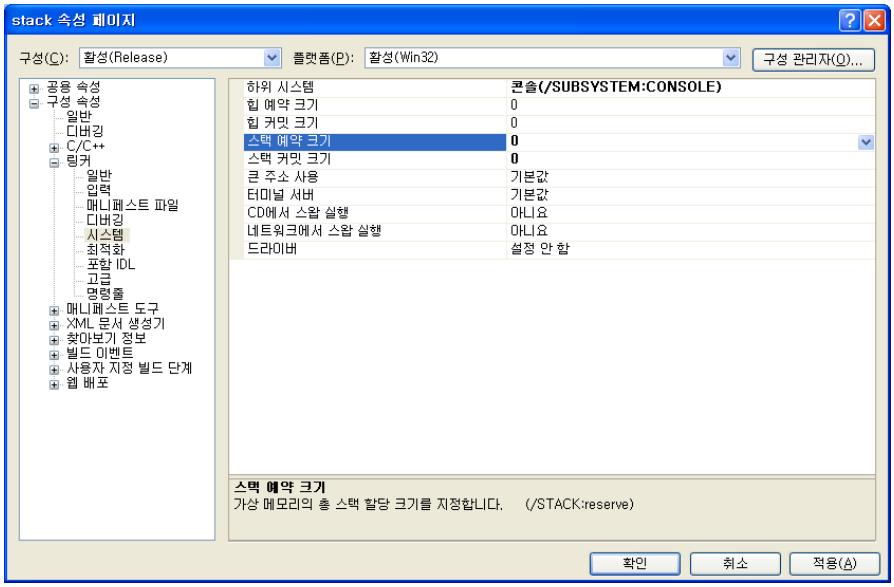


그림 1 함수 호출에 따른 스택 동작 방식

스택을 사용하면서 가장 주의해야 할 사실은 스택의 크기가 고정적이란 점이다. 기본적으로 실행 파일은 스택 크기로 1MB를 사용한다. 이 크기는 프로젝트의 링커 속성에서 조절할 수 있다. <화면 1>에 스택 크기를 조정하는 대화상자가 나와있다. 대화 상자의 스택 예약 크기와 스택 커밋 크기 부분을 조정해서 스택 크기를 늘릴 수 있다.



화면 1 스택 크기 설정 대화 상자

스택의 크기를 침범해서 사용하는 가장 흔한 실수는 잘못 작성된 재귀 호출 함수나 과도한 지역 변수의 사용이다. <리스트 2>에는 잘못된 재귀 호출 함수가 나와있다. infinite 재귀 호출 함수는 종료 지점 없이 무한대로 자신을 호출하면서 스택을 사용한다. 1MB의 스택 크기를 넘는 순간 프로그램은 비 정상적으로 동작한다.

리스트 2 스택 오버플로를 유발시키는 코드

```
void infinite()
{
    infinite();
}

int main()
{
    infinite();
    return 0;
}
```

<리스트 3>에는 이러한 스택 오버플로가 왜 위험한지를 보여주는 코드가 나와있다. 스택에는 실제로 지역 변수 외에도 함수 호출과 관련된 파라미터와 리턴 주소도 보관된다. 따라서 자신이 할당한 스택을 침범해서 쓰는 것은 이러한 프로그램 동작과 관련된 메타 정보를 손상시킬 수 있고, 결국에는 잘못된 동작으로 이어진다. <리스트 3>의 코드는 의도적으로 리턴 주소 값을 Hell로 덮어 씌움으로써 Victim 호출 이후의 메인 코드가 실행이 되지 않고 Hell의 코드가 수행되고 종료돼 버린다. 이런 측면에서 볼 때 <리스트 4>에 나타나 있는 것과 같이 변수의 크기를 체크하지 않고 문자열을 복사하는 함수가 얼마나 위험한지를 알 수 있다.

리스트 3 스택 오버플로의 위험성을 보여주는 코드

```
void Hell()
{
    printf("In the hell.₩n");
    ExitProcess(0);
}

void Victim()
{
    char buf[8];
    *((DWORD *) buf) = (DWORD) Hell;
    memcpy(buf + 16, buf, 4);
}

int main()
{
    Victim();
    printf("Exit₩n");
    return 0;
}
```

리스트 4 문자열 복사 함수의 위험성

```
int main()
{
    char buf[8];
    strcpy(buf, some_input_value);
}
```

지금까지 우리는 스택에 정적인 데이터만 할당해서 사용하는 것을 살펴보았다. 하지만 스택이 항상 컴파일타임에 계산된 크기의 데이터만 저장할 수 있는 것은 아니다. 아래 두 함수를 사용하면 런타임에 스택에 가변 크기의 데이터를 할당하는 것이 가능하다.

void *_alloca(size_t size);
_alloca 함수는 스택에 size 크기만큼을 할당하는 함수다. 할당된 메모리 포인터는 리턴 값으로 반환된다. 할당에 실패하거나, 스택의 크기가 충분하지 않은 경우에는 STATUS_STACK_OVERFLOW 예외가 발생한다. 따라서 반드시 SEH 예외 처리기를 통해서 보호해서 처리해야 한다. 스택에 할당했기 때문에, 할당된 메모리는 함수 리턴 시에 같이 정리된다.

void *_malloca(size_t size);
void _freea(void *memblock);
_malloca 함수는 _alloca 함수를 보완한 버전이다. size의 크기에 따라서 힙 또는 스택에 메모리를 할당한다. 할당에 실패한 경우에는 NULL이 반환되며, 스택의 공간이 충분하지 않은 경우에는 _alloca 함수와 마찬가지로 STATUS_STACK_OVERFLOW 예외가 발생한다. 또한 _malloca 함수를 통

해 할당한 메모리는 반드시 _freea 함수를 통해서 반환해주어야 한다.

_alloca, _malloca 같은 함수를 쓰는 이유는 스택이 힙에 비해서 할당 및 해제 속도가 비약적으로 빠르기 때문이다. 문자열 변환과 같이 미리 결정된 크기는 아니지만 스택에 할당할 만큼 충분한 크기의 데이터를 빈번하게 할당, 해제해야 하는 경우에 사용하면 적합하다. 또한 반드시 기억해야 할 한 가지 사실은 스택에 할당한 데이터는 함수가 리턴되는 시점에 메모리가 정리된다는 점이다. 따라서 추후 참조하기 위한 데이터라면 절대로 스택에 할당해서는 안 된다. _malloca 함수 또한 이 원칙이 적용되기 때문에, _malloca 함수를 통해 할당한 메모리는 항상 함수 리턴 직전에 _freea를 통해서 해제해 주어야 한다.

박스 1 스택과 스코프

신입 개발자들은 스코프 시작 지점에 변수가 생성되어서, 스코프를 벗어나는 순간 삭제된다고 생각한다. 하지만 이는 잘못된 지식이다. <리스트 5>를 살펴보자. 스코프 별로 변수가 생긴다는 생각대로 라면 p1 시점에서 i가 생성되고, p2에 i는 파괴되고 새로운 i가 생성될 것이다. 또한 p2'이라는 루프는 돌 때마다 k라는 변수가 스택에 할당되었다 지워지기를 반복해야 한다. 하지만 실제로는 그렇지 않다. 컴파일러는 함수 스코프 내에 존재하는 지역 변수를 계산해서 함수가 시작되는 p1 시점에 모든 공간을 할당한다. 그리고 그 모든 공간은 p3 시점에서 삭제된다. 그렇다면 똑 같은 이름의 i는 어떻게 처리되는 것일까? 그것은 컴파일러가 내부적으로 이름을 변경해서 사용한다. 즉, 아래에서 i는 이름은 하나지만 내부적으로는 두 개로 만들어지는 것이다. 그렇다면 p3 시점에도 p2에서 사용한 i라는 변수를 참조할 수 있다는 의미일까? 그렇다. 단지 언어적인 장치가 그렇게 하지 못하도록 막을 뿐이다. 이러한 생각에 기반해서 p2'과 같은 코드를 쓰는 것이 성능에 나쁘다는 것은 전적으로 잘못된 상식이다. 루프 내부에 k를 선언하나, 루프 외부에 k를 선언하나 그 결과는 똑같기 때문이다.

리스트 5 자동 변수와 스택

```
int main()
{
    // ... p1
    {
        int i;
        i = 0;
    }

    // ... p2
    for(int i=0; i<10; ++i)
    {
        // ... p2'
        int k;
    }

    // ... p3
```

```
    return 0;
}
```

<리스트 6>의 코드는 <리스트 5>의 코드와 비슷해 보이지만 중요한 차이점이 있다. 여기서는 스택에 Animal 타입의 ani와 k라는 객체가 사용된다. 이것들에 대한 공간 또한 함수 시작 지점에 모두 할당되고, 함수가 끝나는 시점에 모두 수거된다는 점은 <리스트 5>와 같다. 하지만 각 객체의 생성자가 호출되는 시점은 함수 시작 지점이 아닌 해당 객체가 초기화 되는 시점에 호출된다. 따라서 p2'에 사용된 k같은 변수는 루프가 돌 때마다 생성자가 호출된다. 만약 이 변수의 초기화 작업이 매번 필요하지 않다면, 이는 루프 밖에 변수를 위치시키는 것과 비교해서 엄청난 오버헤드를 가지는 셈이 된다.

리스트 6 객체와 스택

```
int main()
{
    // ... p1
    {
        Animal ani;
    }

    // ... p2
    for(int i=0; i<10; ++i)
    {
        // ... p2'
        Animal k;
    }

    // ... p3
    return 0;
}
```

힙 (Heap)

C/C++에서 포인터를 배운 다음 접하는 용어가 힙이다. 힙은 프로그램에서 동적으로 할당한 내용을 저장하는 공간이다. C/C++에서는 malloc이나 new로 할당된 메모리가 이 공간에 저장된다. 또한 윈도우 API인 HeapAlloc을 통해서 할당된 메모리 또한 이 공간에 저장된다. 힙이라는 용어는 스택과는 달리 자료 구조의 힙과는 관련이 없다. 힙이란 말은 단순히 사용하지 않고 남아있는 메모리 덩어리를 나타내는 말이다. 우리는 그 덩어리에서 그때그때 필요한 만큼의 메모리를 할당받아 사용하는 것이다.

힙은 스택과 달리 내부적으로 복잡한 알고리즘을 사용해서 할당과 삭제 작업을 수행한다. 따라서 스택에 비해서 할당과 삭제에 드는 비용이 높다. 이렇게 비용을 지불해서 할당 해제를 하면서 얻을 수 있는 힙의 가장 큰 특징은 생명 주기가 불규칙한 가변 크기 데이터에 대한 할당과 삭제에

좋은 효율을 보인다는 점이다. 여기서 좋은 효율이란 효과적인 메모리 사용과 더불어, 적은 단편화를 가진다는 점이다.

일반적으로 힙은 스택과 달리 필요할 때에 동적으로 그 크기를 늘린다. 따라서 스택처럼 전체 크기에 대한 제약 사항은 적은 편이다. 하지만 스택의 지역 변수와 마찬가지로 힙에 할당한 크기를 넘어서 기록하는 것은 힙을 손상시킬 수 있다. 즉, 다음과 같은 코드는 스택과 똑같이 힙의 블록을 손상시켜서 잘못된 동작을 유발시킬 수 있다.

```
char *ptr = (char *) malloc(8);
strcpy(ptr, "Hello, World!!!!");
```

힙은 할당과 해제 작업을 개발자가 전적으로 통제해야 하기 때문에 주의해야 할 점이 많다. 그 중에서도 개발자들이 가장 많이 하는 실수는 잘못된 포인터를 사용해서 해제하려는 것과 메모리 해제 작업을 하지 않는 것이다. 이러한 실수를 보여주는 코드가 <리스트 7>과 <리스트 8>에 나와 있다. 이러한 실수를 하지 않기 위해서는 항상 할당 받은 메모리 포인터는 저장해두고 있어야 하며, 그것을 조작하는 작업이 필요한 경우에는 임시 변수를 만들어서 사용하는 것이 좋다. 그리고 힙을 통해서 할당한 메모리는 항상 해제하는 코드를 같이 만들도록 하는 것이 메모리 누수가 없는 코드를 만드는 지름길이다.

리스트 7 잘못된 포인터를 사용해서 해제하는 코드

```
char *ptr = (char *) malloc(8);
strcpy(ptr, "Hello");
for(; *ptr; ++ptr)
    *ptr++;
free(ptr);
```

리스트 8 해제 작업을 하지 않는 코드

```
char *ptr;
for(int i=0; i<10; ++i)
{
    ptr = (char *) malloc(80);
    // ...
}
```

이 글을 읽는 독자라면 C/C++에서 사용하는 new/delete, malloc/free 함수에 대해서는 충분히 알고 있을 것이다. 여기서는 힙 메모리를 사용하기 위한 윈도우 API에 대해서만 살펴본다.

HANDLE WINAPI GetProcessHeap(void);
GetProcessHeap은 프로세스의 기본 힙 핸들을 반환하는 함수다. 물론 여기서 반환되는 핸들은 HeapDestroy등을 통해서 파괴할 필요가 없다. 프로세스가 종료되는 시점에 자동적으로 파괴된다.

HANDLE WINAPI HeapCreate(DWORD flOptions, SIZE_T dwInitialSize, SIZE_T dwMaximumSize);
HeapCreate는 힙을 생성하는 함수다. 생성된 힙 핸들이 리턴 된다. flOptions에는 <표 1>에 나와 있는 값을 조합해서 사용할 수 있다. 지정할 옵션이 없으면 0을 지정하면 된다. dwInitialSize는 힙의 기본 크기를 나타낸다. 바이트 단위로 지정하면 된다. 끝으로 dwMaximumSize에는 힙의 최대 크기를 지정한다. 0을 지정하면 자동적으로 크기가 늘어난다.

표 1 flOptions 플래그 별 의미

옵션	설명
HEAP_CREATE_ENABLE_EXECUTE	힙에서 할당한 메모리 블록에서 코드를 실행할 수 있도록 만든다. 데이터 실행 방지 기능이 활성화 된 컴퓨터에서는 이 옵션 없이 생성된 힙의 코드에서 코드가 실행되면 오류가 발생한다. 이는 보안상의 이유로 힙 오버런을 이용해서 공격자가 임의의 코드를 실행시키는 것을 막기 위함이다. 따라서 특별하게 코드를 할당해서 사용하는 경우가 아니라면 이 옵션은 사용하지 않는 것이 좋다.
HEAP_GENERATE_EXCEPTION	힙에서 오류가 발생할 경우 예외를 발생시킨다. 대표적으로 메모리가 부족한데 HeapAlloc을 수행한 경우 NULL을 리턴하는 대신 메모리 부족 예외를 발생 시키는 것을 들 수 있다.
HEAP_NO_SEIRIALIZE	힙을 직렬화 시키지 않는다. 이 플래그를 설정하면 힙의 성능이 향상된다. 만약 이 플래그를 설정한 상태에서 멀티 스레드에서 힙에 접근하는 경우에는 힙이 손상될 수 있다. 따라서 다중 스레드에서 힙에 접근하는 경우에는 이 플래그를 지정하지 않아야 한다.

BOOL WINAPI HeapDestroy(HANDLE hHeap);
HeapDestroy 함수는 HeapCreate 함수를 통해서 생성된 힙을 파괴하는 역할을 한다. 별도로 생성한 힙은 반드시 프로그램 종료 전에 HeapDestroy를 통해서 파괴해 주어야 한다.

윈도우의 모든 프로세스는 생성과 동시에 한 개의 기본 힙을 가진다. 이 힙 핸들을 구하는 것이 GetProcessHeap이다. 물론 개발자는 이러한 기본 힙 외에도 HeapCreate란 함수를 통해서 추가적으로 필요한 만큼 힙을 생성할 수 있다. 이렇게 힙을 여러 개 만들 수 있도록 한 이유는 우리가 서랍을 통해 물건을 정리하는 것과 동일한 원리다. 큰 서랍이 하나 있는 것 보다는 작은 서랍을 여러 개 이용하는 것이 훨씬 더 물건을 깔끔하게 정리할 수 있기 때문이다. 별도의 힙을 만들어 사용하면 프로그램의 다른 컴포넌트의 코드가 발생시킨 메모리 오류로부터 자유로울 수 있다. 힙을 공유하지 않기 때문이다. 또한 별도의 힙에 같은 크기의 데이터만 할당함으로써 단편화를 만들지 않을 수도 있다. 더 나아가서는 별도의 힙에 대한 정보를 토대로 힙 직렬화를 하지 않거나,

일일이 해제하지 않고 HeapDestroy를 통해서 한 번에 해제하는 식의 최적화 작업을 할 수 있다.

LPVOID WINAPI HeapAlloc(HANDLE hHeap, DWORD dwFlags, SIZE_T dwBytes);
HeapAlloc 함수는 힙에 메모리를 할당하는 함수다. hHeap에는 할당할 힙의 핸들을 넣어준다. dwFlags에는 일반적으로 0을 지정하면 된다. dwBytes에는 할당할 메모리 크기를 지정한다.

BOOL WINAPI HeapFree(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem);
HeapFree 함수는 힙에 할당한 메모리를 삭제하는 함수다. hHeap에는 메모리가 할당된 힙의 핸들을 넣어준다. dwFlags에는 0을, lpMem에는 할당 받은 메모리 포인터를 넣어주면 된다. HeapDestroy를 사용하는 경우에는 해당 힙을 통해서 할당된 메모리가 동시에 삭제된다. 따라서 HeapDestroy 함수를 호출하는 경우에는 굳이 일일이 할당한 메모리에 대해서 HeapFree를 호출해줄 필요가 없다.

SIZE_T WINAPI HeapSize(HANDLE hHeap, DWORD dwFlags, LPCVOID lpMem);
HeapSize는 힙에 할당된 메모리의 크기를 구하는 함수다. hHeap에는 메모리가 할당된 힙의 핸들을 넣어준다. dwFlags에는 0을, lpMem에는 할당 받은 메모리 포인터를 넣어준다.

BOOL WINAPI HeapWalk(HANDLE hHeap, LPPROCESS_HEAP_ENTRY lpEntry);
HeapWalk는 특정 힙을 통해서 할당된 메모리 정보를 알아내는 함수다. hHeap은 정보를 알고 싶은 힙 핸들을 전달하면, lpEntry를 통해서 할당된 메모리 정보가 반환된다. LPPROCESS_HEAP_ENTRY에 대한 보다 자세한 내용은 MSDN을 참조하도록 하자.

<리스트 9>에는 힙을 생성하고 메모리를 할당한 다음 그것들의 정보를 살펴보는 프로그램의 소스 코드가 나와 있다. 지금까지 설명한 함수들이 실제로는 어떻게 사용되는지 살펴보도록 하자. 그리고 HeapWalk의 결과와 할당한 결과가 일치하는지에 대해서 알아보자.

리스트 9 힙 함수를 사용하는 간단한 예제

```
int main()
{
    HANDLE heap = HeapCreate(0, 1024, 0);
    if(!heap)
        return 0;

    printf("Heap Create %08X₩n", heap);


    PVOID ptr[3];
    ptr[0] = HeapAlloc(heap, 0, 40960);
    printf("ALLOC %d => %08X₩n", 40960, ptr[0]);
    ptr[1] = HeapAlloc(heap, 0, 4096);
    printf("ALLOC %d => %08X₩n", 4096, ptr[1]);
```

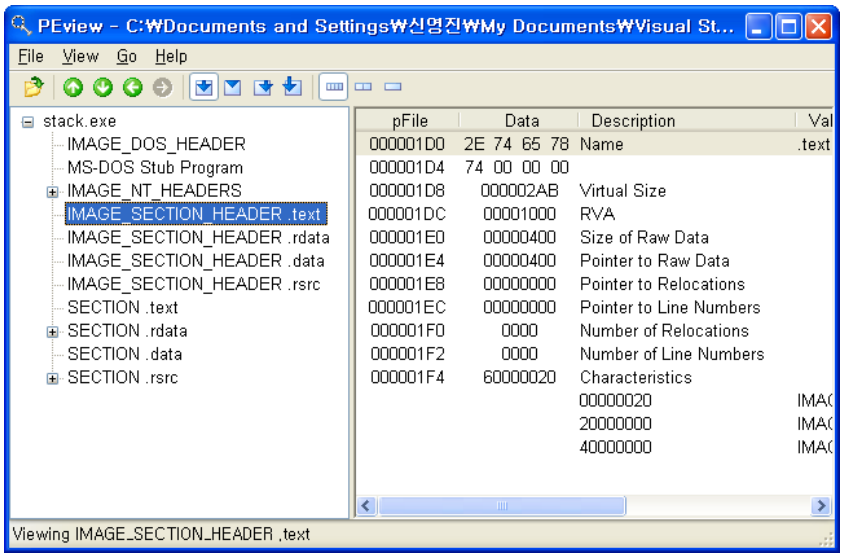
```
    ptr[2] = HeapAlloc(heap, 0, 111);
    printf("ALLOC %d => %08X₩n", 111, ptr[2]);

    printf("₩nHeap Info₩n");
    PROCESS_HEAP_ENTRY entry;
    entry.lpData = NULL;
    while(HeapWalk(heap, &entry))
    {
        if(entry.wFlags & PROCESS_HEAP_ENTRY_BUSY)
        {
            printf("%08X %d %d %d₩n"
                    , entry.lpData
                    , entry.cbData
                    , entry.cbOverhead
                    , entry.wFlags);
        }
    }

    HeapDestroy(heap);
}
```

박스 2 데이터 영역, 코드 영역이란 말은 무슨 의미일까?

데이터 영역, 코드 영역 이라는 용어는 실행 파일의 구조와 상관이 있는 용어다. 컴파일러는 일반적으로 추후 분석의 용의성과 보안성을 위해서 실행 파일을 다양한 파트로 나누어서 생성한다. 그러한 것을 나누는 기준의 하나가 코드와 데이터다. <화면 2>는 PE-View라는 프로그램을 사용해서 윈도우 실행 파일의 구조를 살펴보고 있는 화면이다. 화면을 보면 실행 파일에는 총 네 개의 섹션이 있음을 알 수 있다.



화면 2 실행 파일의 섹션 구조

.text 섹션은 실행 코드를 저장하는 섹션이다. .rdata 섹션은 방금 살펴보았던 것처럼 초기화된 데이터 영역으로 프로그램에서 사용된 문자열이나 숫자 상수 등이 저장되는 공간이다. .data 섹션은 초기화되지 않은 전역 데이터가 저장되는 공간이다. 끝으로 .rsrc 섹션은 프로그램에 사용된 리소스가(비트맵, 아이콘, 버전 정보 등) 저장되는 공간이다.

실행 파일이 메모리에 로딩되면 <그림 2>에 나타난 것처럼 섹션 별로 배치가 된다. 각각의 섹션은 각 섹션의 특성에 맞는 접근 속성을 가지고 있다. 예를 들면, .text 섹션은 코드가 모여있는 부분이기 때문에 읽기, 실행이 가능한 것이다. 이 영역에 데이터를 기록하는 작업을 하면 잘못된 접근 오류가 발생한다.

그림 2 섹션의 메모리 배치

흔히 사용하는 아래와 같은 코드를 생각해보자. str은 배열이고, pstr은 포인터다. 이 둘의 차이점은 무엇일까? 과연 둘 다 안전한 코드일까?

```
char str[] = "Hello.."; str[1] = 'a';
char *pstr = "Hello.."; pstr[1] = 'a';
```

str과 pstr은 비슷하게 보이지만 틀리게 동작한다. <그림 3>에는 그러한 차이를 나타내는 메모리 구조가 나와있다. 살펴보면 str은 데이터 .rdata 영역에 있는 "Hello.. " 문자열을 스택의 공간으로 복사한 것이고, pstr은 단지 데이터 영역에 대한 참조로 이루어졌다는 것을 알 수 있다. 이러한 복사와 같은 작업은 컴파일러가 자동적으로 생성하는 코드에 의해서 이루어진다.

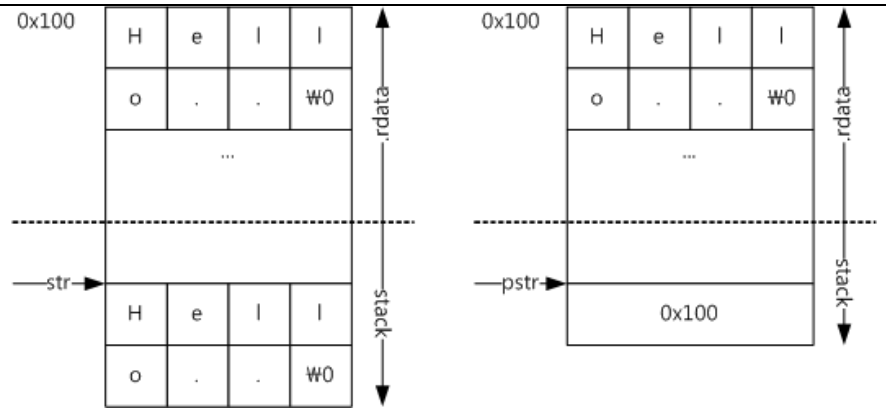


그림 3 str과 pstr 메모리 구조

이제 각각의 변수의 1번 원소에 'a'를 대입하는 코드를 살펴보자. 이 경우에 str의 값을 변경하는 것은 안전한 동작이지만, pstr의 값을 변경하는 것은 문제가 될 소지가 있다. 왜냐하면 pstr이 가리키고 있는 문자열이 저장되어 있는 메모리 영역이 데이터 영역이기 때문이다. 위에 표기된 .rdata 영역은 일반적으로 초기화된 상수가 저장되는 공간으로 읽기만 가능한 영역이기 때문이다. 그곳의 값을 변경하려 했기 때문에 최악의 경우에는 잘못된 접근 오류가 발생할 수 있다.

도전 과제

이번 시간에 우리는 스택과 힙의 구조와 원리에 대해서 살펴보았다. 또한 그것들을 다루는 윈도우 API의 사용 방법에 대해서도 간략하게나마 살펴보았다. 한 가지 명심해야 할 것은 이러한 힙이나 스택이란 영역이 물리적인 메모리에서 구분 지어져 있는 것이 아니란 점이다. 이것은 단지 개발자들이 편리하게 사용하기 위해서 용도별로 분리해둔 논리적인 기준점일 뿐이란 사실을 꼭 기억하자.

이번 시간의 도전 과제는 <리스트 5>의 코드와 관련이 있다. 해당 코드는 스코프가 끝나도 변수에 접근 가능하다는 사실을 보여주진 못하고 있다. 그것을 직접적으로 보여줄 수 있도록 예제 코드를 수정해보자. 더 나아가 _alloca 함수를 사용해서 함수의 진입 시점에 이미 세 변수를 위한 공간이 할당되었음을 보여주도록 만들어보자.

참고자료

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

Windows API 정복
김상형, 가남사