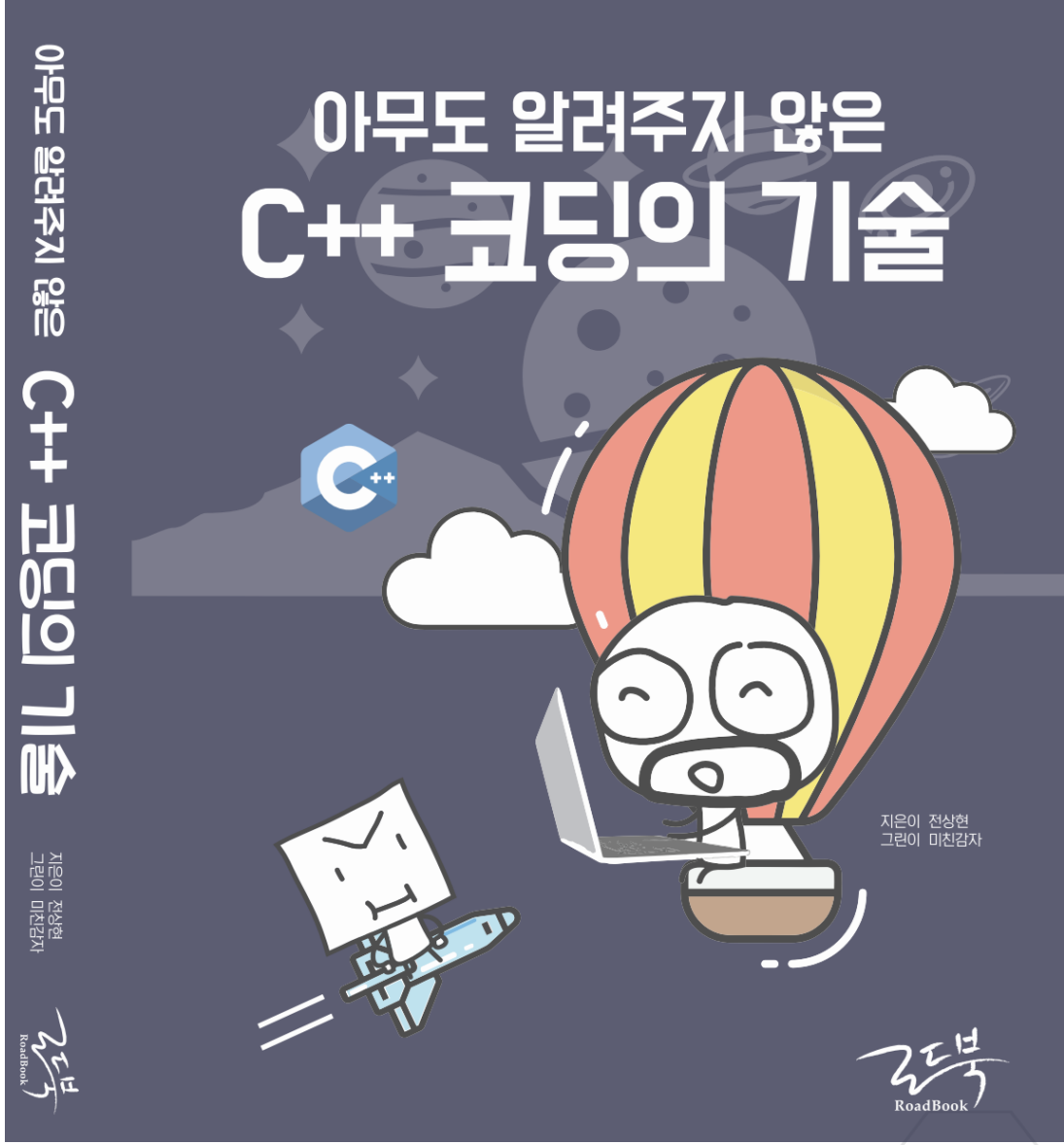
시스템 프로그래밍7

S-개발자 4기 2026-03-12(목)

서울 송파구 동남로 130, 2층 제 4강의실

악성코드검거단 MALWARE ARREST TEAM 대표이사 전상현

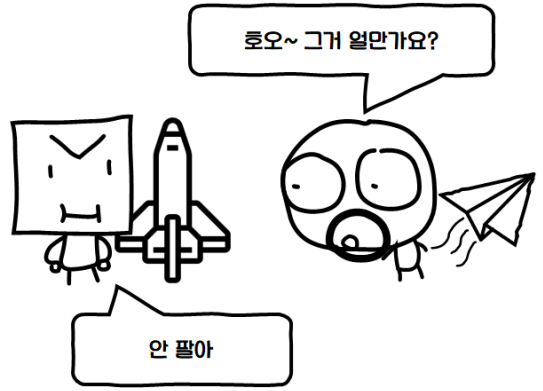


들어가기 전에 수업 개요

과정명	시스템 프로그래밍	강사명	전상현
강의시간	24시간 (4일 x 6시간)		
강의목표	컴퓨터 시스템을 이해한다. 나아가 다양한 시스템을 제어하는 C++ 프로그래밍 기술을 익힌다.		
평가방식	시험 50%, 실습 50%		

강의내용

시간(H)	제목	내용
1H	오리엔테이션	강사 소개 후 간단한 퀴즈와 함께 실행파일의 구조를 이해한다.
2H	변수형과 유니코드	시스템에 존재하는 모든 종류의 변수형을 알아본다. 다국어를 표현하는 문자체계인 유니코드를 이해한다.
3H	개발환경 구축하기	윈도우/리눅스(wsl)에 빌드할 수 있는 환경을 구성한다. 기본적인 개발툴 사용법을 익힌다.
2H	문자열 정복하기	C++과 친해지기 위한 코드를 작성해본다. VisualStudio 디버깅 기술을 배운다.
2H	C++ 컴파일러/링커/디버거 정복하기	유니코드 상호 변환하는 함수를 이해한다. 문자열을 자유자재로 다루는 방법을 배운다.
2H	파일시스템 정복하기	플랫폼별 파일/디렉토리 접근 함수를 이해한다. 파일 시스템을 자유자재로 다루는 방법을 배운다.
2H	데이터 정복하기	STL 자료구조를 이해하고, 실무 응용 기술을 배운다. XML/JSON/INI 등을 자유자재로 다루는 방법을 배운다.
2H	메모리 정복하기	C++의 꽃, 스택 메모리의 장단점을 알아본다. 4가지 영역의 메모리를 자유자재로 활용한다.
2H	소켓 정복하기	UDP/TCP 통신, DATAGRAM/STREAM의 차이를 이해한다. 소켓 통신 시스템을 자유자재로 다루는 방법을 배운다.
6H	**최종실습**	**앞에서 배운 지식과 기술을 활용하여 미니 프로젝트를 진행한다.**



코딩 실력이 곧 우주선이다.

우주같이 광활한 컴퓨터 세계로 여행을 떠나자.

<참고서적>

크로스플랫폼 핵심모듈 설계의 기술(2018) – 전상현, 로드북

아무도 알려주지 않은 C++ 코딩의 기술 (2023) – 전상현, 로드북

싱글턴 패턴에 대해서 알고 계신 여러분,

다음의 코드가 세상에서 가장 아름다운 싱글턴이란 것을 느끼십니까?

```
class CGlobalDoc
{
        CGlobalDoc(void);
        ~CGlobalDoc(void);

public:
        static CGlobalDoc* GetInstance(void)
        {
                static CGlobalDoc instance;
                return &instance;
        }
};

inline CGlobalDoc* Doc(void)
{
        return CGlobalDoc::GetInstance();
}
```

스택 메모리

이미 잘 알고 있는 힙(HEAP) 메모리

C++의 장점 중 하나가 메모리를 다룰 수 있다는 것입니다.

수 기가에 이르는 넓은 범위를 **어떻게 활용할지 고민하는 즐거움**을 느껴봅시다.

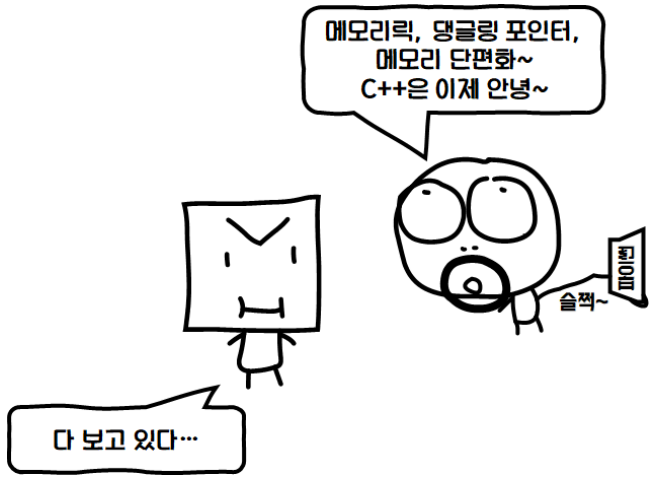
언어를 막론하고 아래와 같은 코드는 여러분들에게 친숙할 것입니다.

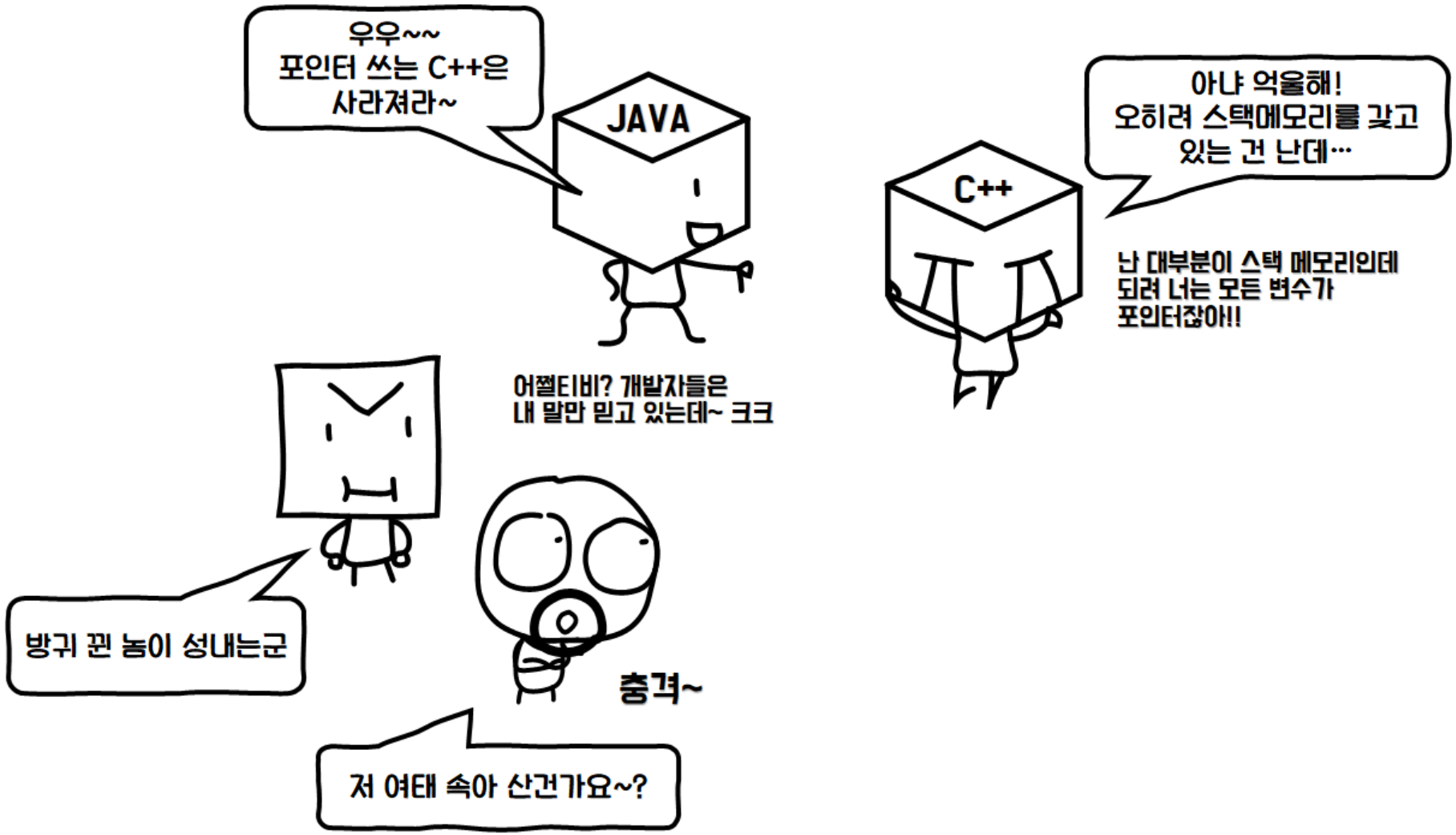
하지만...

```
Object obj = new SomeObject;
```

New로 할당한 객체는 잘못 다루면 다음과 같은 세 가지 문제에 직면합니다.

메모리 릭	new로 할당한 메모리를 해제하지 않을 때, 쓰이지 않는 메모리가 지속적으로 쌓이면서 시스템 메모리를 고갈시키는 문제를 말합니다. 누구나 한 번쯤은 겪어보는 이 문제는 시간이 지날수록 프로그램이 느려지고 나중에는 강제종료 되기도 합니다.
댕글링 포인터	메모리 릭을 경험한 개발자들은 메모리 해제하는 것이 중요하다는 것을 인식합니다. 그러다 너무 빨리 반환하거나 마감 처리를 제대로 하지 못해서 이미 해제한 메모리에 접근하다 발생하는 접근 오류입니다. 이 또한 치명적인 오류로 프로그램이 강제 종료될 수 있습니다.
메모리 단편화	할당과 해제를 문제없이 개발했어도 발생할 수 있는 문제입니다. 넓어 보이는 메모리 공간이지만 잦은 할당과 해제로 만들어진 메모리 파편은 새롭게 할당할 때 방해가 됩니다. 극단적으로 파편화된 메모리는 새로운 메모리 공간을 찾는데 CPU 소모량이 높아지거나 결국 할당에 실패하게 만들 수도 있습니다.




우우~~
포인터 쓰는 C++은
사라져라~
JAVA
어쩔티비? 개발자들은
내 말만 믿고 있는데~ 크크
방귀 뀐 놈이 성내는군
충격~
저 여태 속아 산건가요~?
아냐 억울해!
오히려 스택메모리를 갖고
있는 건 난데…
C++
난 대부분이 스택 메모리인데
되려 너는 모든 변수가
포인터잖아!!

C++은 객체를 다음과 같이 생성해서 사용할 수 있습니다.

이른바 **스택 메모리**를 활용한 **지역변수**이죠.

```
SomeObject obj;

Obj.DoSomething();
```

지역변수는 일시적으로 생성되었다가 블록이 해제되면 자동으로 소멸되는 변수입니다. 다음과 같은 장점이 있습니다.

1. 메모리 할당이 아주 빠르다

2. 자동으로 소멸된다.

세상에 완벽한 것은 없듯이 스택 메모리에도 단점이 있습니다.

바로, **메모리 공간이 적다(통상적으로 1MB)**는 점입니다.

chkstk.asm

```
; Handle allocation size that results in wraparound.
; Wraparound will result in StackOverflow exception.

        sbb     eax, eax                ; 0 if CF==0, ~0 if CF==1
        not     eax                     ; ~0 if TOS did not wrapped around, 0 otherwise
        and     ecx, eax                ; set to 0 if wraparound

        mov     eax, esp                ; current TOS
        and     eax, not ( _PAGESIZE_ - 1) ; Round down to current page boundary

cs10:
        cmp     ecx, eax                ; Is new TOS
    bnd jb      short cs20              ; in probed page?
        mov     eax, ecx                ; yes.
        pop     ecx
        xchg    esp, eax                ; update esp
        mov     eax, dword ptr [eax]    ; get return address
        mov     dword ptr [esp], eax    ; and put it at new TOS
    bnd ret

; Find next lower page and probe
cs20:
        sub     eax, _PAGESIZE_         ; decrease by PAGESIZE
        test    dword ptr [eax],eax     ; probe page.
        jmp     short cs10

_chkstk endp

        end
```

예외가 처리되지 않음

0x00F71809에(test.exe의) 처리되지 않은 예외가 있습니다.
0xC00000FD: Stack overflow(매개 변수: 0x00000000, 0x00702000).

세부 정보 복사

▸ 예외 설정

예외 발생(0x00F71809, test.exe): 0xC00000FD: Stack overflow(매개 변수: 0x00000000, 0x00702000).

0x00F71809에(test.exe의) 처리되지 않은 예외가 있습니다. 0xC00000FD: Stack overflow(매개 변수: 0x00000000, 0x00702000).

디버깅 해본 사람들은 알겠지만 콜스택은 우리가 버그를 추적함에 있어 아주 중요한 정보입니다. 그런데 가끔 어떤 상황에서는 콜스택이 제대로 나오지 않는 경우가 있습니다. 바로 메모리가 다른 변수에 의해 침범되어 덮어씌워진 경우이죠.

```c
#include <stdio.h>


void Test(void)
{
    int a = 10;
    int b = 20;
    char c[10] = { 0, };
    strcpy(c, "123456789 123456789 123456789 123456789 123456789 ");
}


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

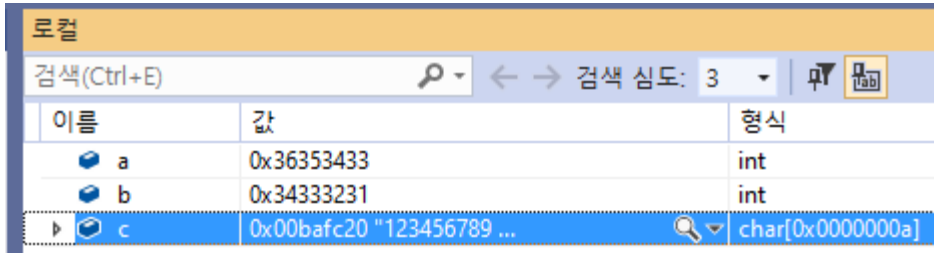
어떤 결과가 나올지 예측해보세요

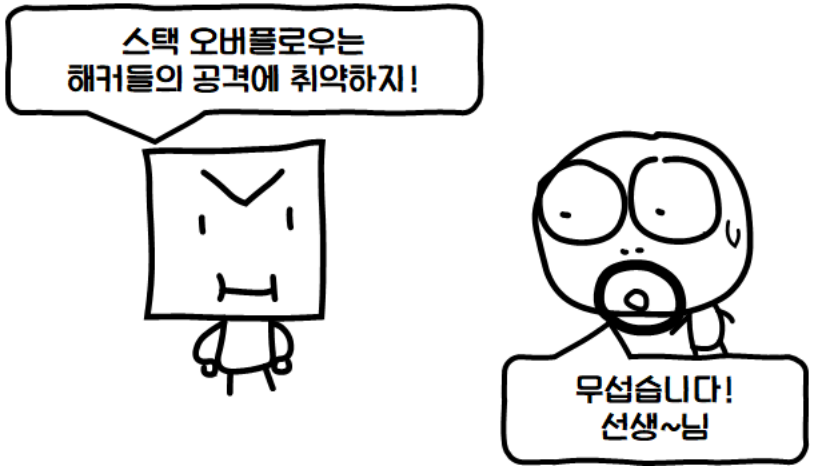
디버깅 모드로 실행하면 다음과 같은 결과가 나옵니다.

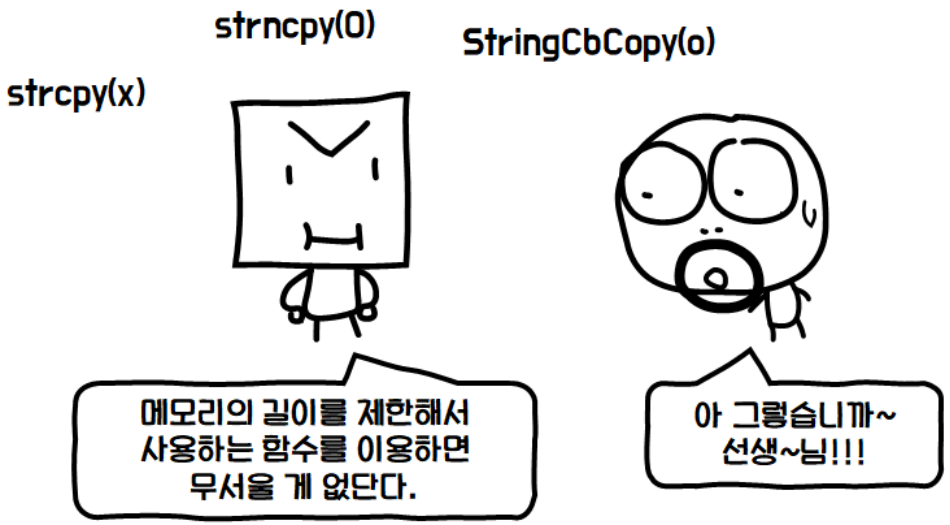
```
strcpy(c, "123456789 123456789 123456789 123456789 123456789 ");
```

그리고 그 때의 콜스택과 변수 상태는 다음과 같은 모습입니다.

문제의 코드 실행 직전	문제의 코드 실행 직후
호출 스택 이름 test.exe!Test() 줄 9 test.exe!main() 줄 15 [외부 코드]	호출 스택 이름 test.exe!Test() 줄 10 [외부 코드] [아래 프레임이 누락 및/또는 올바르지 않음]




스택 오버플로우는
해커들의 공격에 취약하지!
무섭습니다!
선생~님

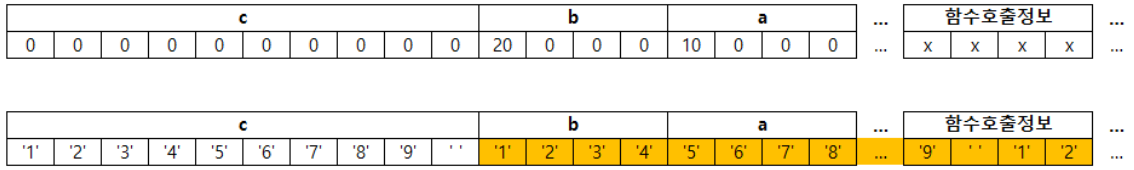

strncpy(O)
StringCbCopy(o)
strcpy(x)
메모리의 길이를 제한해서
사용하는 함수를 이용하면
무서울 게 없단다.
아 그렇습니까~
선생~님!!!

전체 메모리 공간에 실행 코드도 있고, 전역 변수나 정적 변수 공간도 있습니다. 각종 DLL 공간도 따로 있습니다. 스레드마다 스택 메모리가 존재하는데 그 외에는 모두 힙 메모리 영역이라고 생각하면 됩니다. 사실 **스택 메모리도 큰 범위에서 보면 힙 메모리의 일부**라고 보면 됩니다.

영역		내용	생성시점	특징
code(text)		- 기계어로 번역된 프로그램 실행코드가 저장되는 영역	Compile Time	고정 길이
data	data	- 초기값이 있는 전역변수와 정적변수가 저장되는 영역		
	bss	- Block Started by Symbol 의 줄임말 - 초기값이 없는 전역변수와 정적변수가 저장되는 영역		
heap		- 동적 할당 메모리 영역 - 낮은 메모리에서 높은 메모리로 할당	Run Time	가변 길이
...		- 빈공간 - heap 과 stack 이 추가로 할당될 수 있음		
stack		- 지역변수, 함수인자, 리턴 값등 임시로 사용하는 데이터를 저장하는 영역 - 높은 메모리에서 낮은 메모리로 할당		

스택 메모리의 특성

1) 메모리의 시작점과 끝점, 마지막 할당 지점이 있음
2) 메모리를 할당할 때는 마지막 할당 지점 이후로 순차적으로 할당
3) 메모리를 해제할 때는 마지막 할당한 지점부터 순차적으로 해제
4) 메모리는 높은 주소에서 낮은 주소로 할당됨
5) 메모리의 시작점과 끝점 뒤에는 가드 영역이 존재해서 접근하면 프로그램이 강제 종료됨
6) 가드 영역은 디버그 모드와 릴리즈 모드에 따라 크기가 다름



스택 메모리는 왜 거꾸로 사용하도록 설계되었을까요?
메모리를 순차적으로 할당한다면 이렇게 이전에 선언된 변수나 콜 스택이 손상될 가능성이 줄어들텐데요.

디버그 모드와 릴리즈 모드는 우리가 잘 모르는 차이가 있습니다. 그 중 하나가 **메모리 가드**라는 것인데 스택 메모리를 사용하는 사이사이마다 임의의 공간을 비워 두는 것입니다.

그림으로 나타내 보면 다음과 같습니다.

b												a			
20	0	0	0	?	?	?	?	?	?	?	?	10	0	0	0

콜스택 정보
빈공간
함수 인자1
빈공간
...
빈공간
함수 인자N
빈공간
지역변수1
빈공간
지역변수2

그럼 왜 릴리즈 모드와 디버그 모드를 다르게 분리해서 운영할까요? 만일 메모리 가드가 없다면 예상치 못한 버퍼 오버플로우를 잡아내기가 어려울 겁니다.
프로그램이 커질수록 시간낭비가 더 커지는 것이죠.
덕분에 그런 시간낭비를 덜 수 있게 됩니다.

릴리즈 모드는 다릅니다. 아까운 메모리를 조금이라도 더 효율적으로 사용하기 위해서는 없애는 것이 낫습니다. 더구나 메모리 가드 연산을 위한 CPU 사용량도 줄일 수 있겠죠. 이런 차이를 알고 있는 개발자들은 디버그 모드와 릴리즈 모드를 적절히 사용할 수 있어야 합니다.

이 차이를 모르면 **시간에 쫓겨 디버그 모드로 배포하는 경우**도 발생하는데 **그런 경우는 피해야** 합니다. 별거 아닌 버그로 프로그램이 크래시돼버릴 수도 있으니까요.

1) 크기가 큰 데이터를 만들지 말자.

어떤 클래스나 구조체를 지역 변수로 사용할지 모릅니다. 가급적 모든 것을 작게 만드는 습관을 들입시다. 보통 덩치가 커지는 이유가 문자열이나 배열 때문입니다. STL의 string과 vector는 내부에서 동적 메모리인 힙을 이용하므로 크기를 줄일 수 있습니다.

2) 크기가 큰 데이터는 정적 변수나 전역 변수를 쓰자.

그럼에도 어떤 경우에는 반드시 크기가 큰 데이터를 만들어야 하는 경우가 있습니다. 프로그램에서도 그런 데이터가 흔하지는 않을 것이라서 전역 변수나 정적 변수로 선언하는 것을 추천합니다.

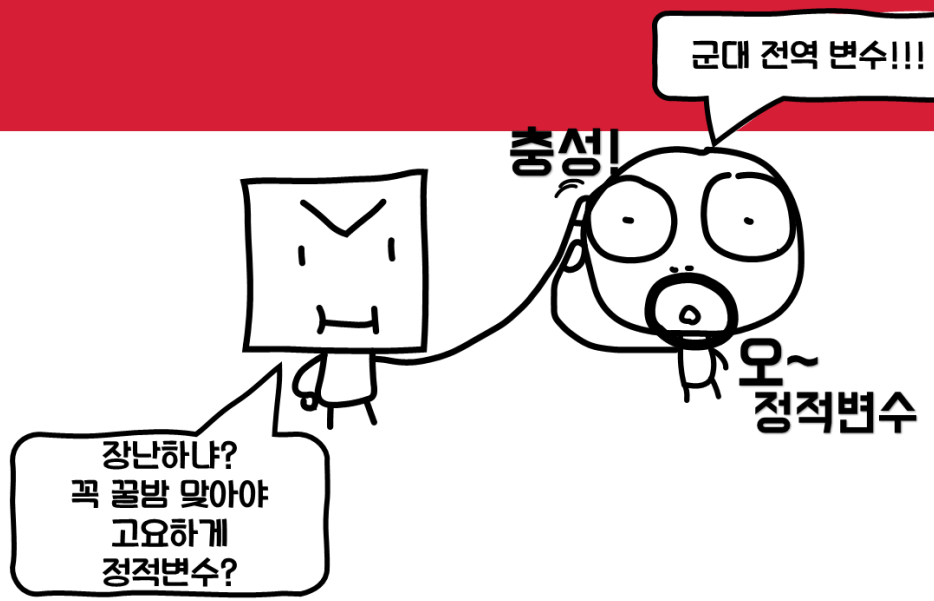
3) 가급적 재귀함수를 만들지 말자.

프로그래밍을 공부하다 한두 번은 짜보는게 재귀 함수입니다. 경우에 따라 복잡한 로직을 쉽게 풀어낼 수 있는 기막힌 묘수를 제공합니다. 하지만 실무에서는 사용하는 것은 바람직하지 않습니다. 스택 메모리가 언제든 터질 수 있어요.

4) 어쩔 수 없이 재귀함수를 써야 한다면? 반복함수로 바꾸자.

이미 수학적으로 모든 재귀함수는 반복 함수로 바꿀 수 있다고 증명되었습니다. 반복 함수란 for문을 이용해 N번 반복하는 구간을 포함한 함수를 말합니다.

전역 변수와 정적 변수



데이터 영역에 기록되는 변수

다음 프로그램 실행 결과는?

```
#include <stdio.h>


int g_nGlobalVar = 0;
int main()
{
    static int s_nStaticVar = 0;
    int nLocalVar = 0;
    int* pHeapVar = new int;
    printf("GlobalVar address: 0x%08X₩n", &g_nGlobalVar);
    printf("StaticVar address: 0x%08X₩n", &s_nStaticVar);
    printf("LocalVar  address: 0x%08X₩n", &nLocalVar);
    printf("HeapVar   address: 0x%08X₩n", pHeapVar);
    return 0;
}
```

데이터 영역에 기록되는 변수

```
D:₩GIT₩test₩Debug₩test.exe
GlobalVar address: 0x00A7B3D8
StaticVar address: 0x00A7B3DC
LocalVar  address: 0x00A2FB0C
HeapVar   address: 0x00EA4038
```

test.exe - PID: 66744 - Thread: 주 스레드 81728 - x32dbg

파일(F) 보기(V) 디버그(D) Tracing 플러그인(P) 즐겨찾기(I) 설정(O) 도움말(H) Mar 26 2022 (TitanEngine)

CPU | 로그 | 메모 | 중단점 | 메모리 맵 | 호출 스택 | SEH | 스크립트 | 기호 | 소스

주소	크기	정보	내용	유형	보호	초기
008A0000	00010000			MAP	-RW--	-RW--
008B0000	00001000			MAP	-R---	-R---
008C0000	00001000			MAP	-R---	-R---
008D0000	0001D000			MAP	-R---	-R---
008F0000	00035000	예약됨		PRV		-RW--
00925000	0000B000			PRV	-RW-G	-RW--
00930000	000FA000	예약됨		PRV		-RW--
00A2A000	00006000	스레드 13F40 스택		PRV	-RW-G	-RW--
00A30000	00004000			MAP	-R---	-R---
00A40000	00001000			MAP	-R---	-R---
00A50000	00002000			PRV	-RW--	-RW--
00A60000	00001000	test.exe		IMG	-R---	ERWC-
00A61000	00010000	".textbss"	실행 가능한 코드	IMG	ERWC-	ERWC-
00A71000	00007000	".text"	실행 가능한 코드	IMG	ER---	ERWC-
00A78000	00003000	".rdata"	읽기 전용 초기화 데이터	IMG	-R---	ERWC-
00A7B000	00001000	".data"	초기화 된 데이터	IMG	-RW--	ERWC-
00A7C000	00001000	".idata"	데이터 가져오기	IMG	-R---	ERWC-
00A7D000	00001000	".msvcjmc"		IMG	-RWC-	ERWC-
00A7E000	00001000	".00cfg"		IMG	-R---	ERWC-
00A7F000	00001000	".rsrc"	리소스	IMG	-R---	ERWC-
00A80000	00001000	".reloc"	기준 재배치	IMG	-R---	ERWC-
00A90000	000C9000	\Device\HarddiskVolume3\Windows\		MAP	-R---	-R---
00B60000	00001000			MAP	-R---	-R---
00B70000	00035000	예약됨		PRV		-RW--
00BA5000	0000B000			PRV	-RW-G	-RW--
00BB0000	00004000			MAP	-R---	-R---
00BB4000	00004000	예약됨 (00BB0000)		MAP		-R---
00BD0000	00007000			PRV	-RW--	-RW--
00BD7000	00009000	예약됨 (00BD0000)		PRV		-RW--
00C00000	00194000	예약됨		PRV		-RW--
00D94000	0000E000			PRV	-RW--	-RW--
00DA2000	0005E000	예약됨 (00C00000)		PRV		-RW--
00E00000	00035000	예약됨		PRV		-RW--
00E35000	0000B000			PRV	-RW-G	-RW--
00E40000	00035000	예약됨		PRV		-RW--
00E75000	0000B000			PRV	-RW-G	-RW--
00E90000	0001D000			PRV	-RW--	-RW--
00EAD000	000E3000	예약됨 (00E90000)		PRV		-RW--
00F90000	000FC000	예약됨		PRV		-RW--
0108C000	00004000	스레드 4D84 스택		PRV	-RW-G	-RW--
01090000	000FD000	예약됨		PRV		-RW--
0118D000	00003000	스레드 6B94 스택		PRV	-RW-G	-RW--
01190000	00001000			PRV	-RW--	-RW--
01191000	00069000	예약됨 (01190000)		PRV		-RW--
01200000	000FD000	예약됨		PRV		-RW--
012FD000	00003000	스레드 55C0 스택		PRV	-RW-G	-RW--
01300000	00045000			MAP	-R---	-R---
01345000	001BB000	예약됨 (01300000)		MAP		-R---
01500000	00181000			MAP	-R---	-R---
01690000	0026D000			MAP	-R---	-R---
018FD000	01194000	예약됨 (01690000)		MAP		-R---

데이터 영역에 기록되는 변수

지역 변수는 함수 안에서만 선언하고 사용할 수 있습니다. 만일 함수 밖에서 선언하면 바로 **전역 변수**가 됩니다. 특징이라면 같은 이름을 갖는 변수가 둘 이상 존재할 때 링크 에러가 발생한다는 점입니다.

정말 그런지 확인 차원에서 서로 다른 cpp 파일에 동일한 이름의 전역 변수를 선언해보겠습니다. 이름과 형식은 int g_nGlobalVar입니다.

main.cpp	test.cpp
int g_ nGlobalVar = 0;	int g_ nGlobalVar = 1;

빌드하면 여러 번 정의된 기호가 있다고 오류가 발생합니다. **전역 변수는 같은 이름으로 하나만 선언할 수 있기 때문**입니다.

```
1>Test.obj : error LNK2005: "int g_nGlobalVar" (?g_nGlobalVar@@3HA)이(가) main.obj에 이미 정의되어 있습니다.
1>D:\GIT\test\Debug\test.exe : fatal error LNK1169: 여러 번 정의된 기호가 있습니다.
```

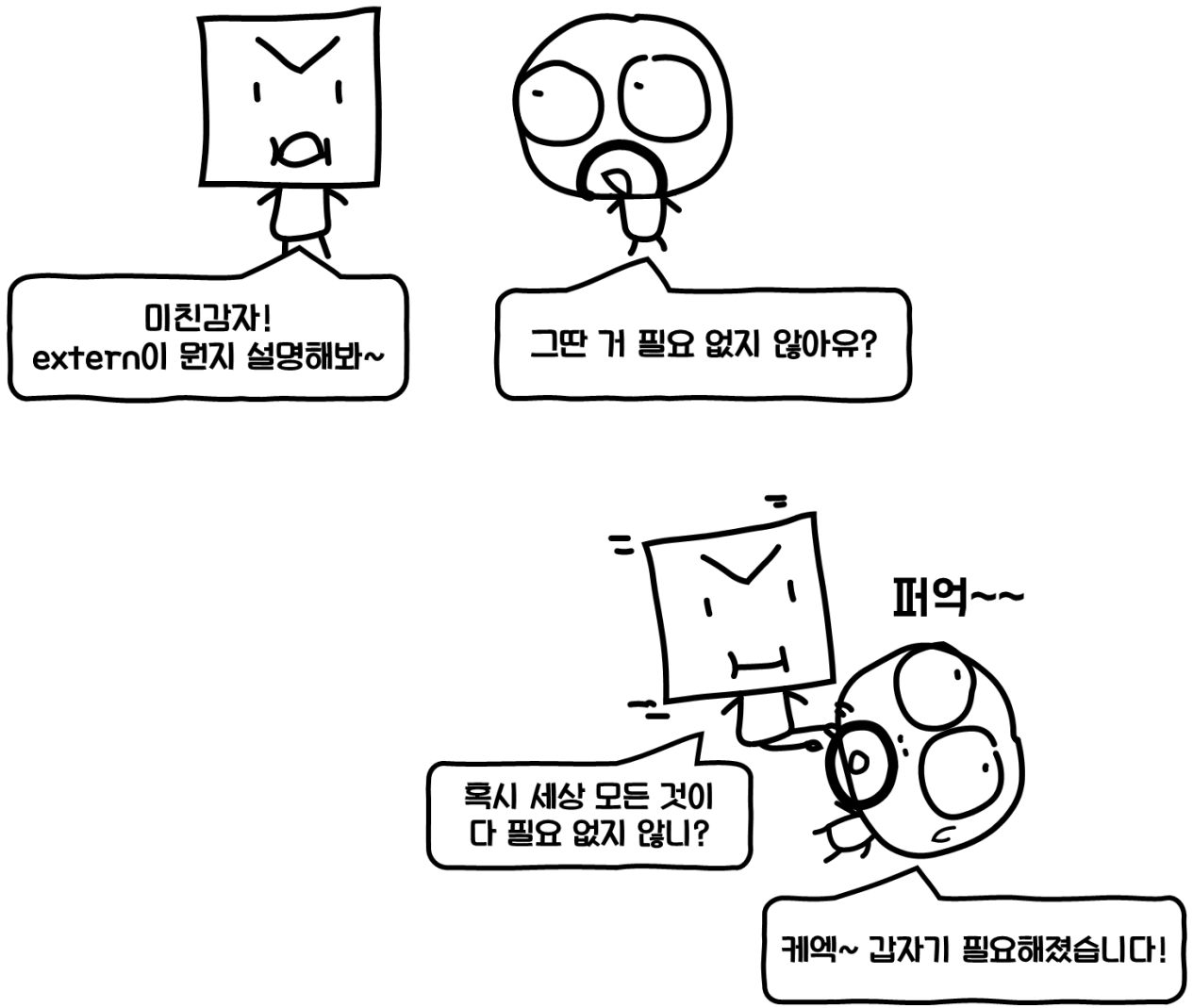
전역 변수와 정적 변수 그리고 extern 키워드

정적 변수는 전역 변수와 유사하지만 **사용 범위에 제한이 있다는 점**이 다릅니다. 앞에서 전역 변수는 동일한 이름으로 선언하면 안 된다고 했었는데 정적 변수는 가능합니다.

a.cpp	b.cpp
static int g_ nStaticVar = 0;	static int g_ nStaticVar = 1;

이렇게 서로 다른 cpp 파일에 선언된 같은 이름의 정적 변수지만 링크 에러가 발생하지 않습니다. 전역 변수는 안되는데 정적 변수는 되는 것은 해당 변수가 코드 전체에서 사용할 수 있던 전역 변수와는 달리 **정적 변수는 그 범위가 선언된 파일 안으로만 한정되기 때문**입니다.

전역 변수와 정적 변수 그리고 extern 키워드



둘 중에 실행 가능한 경우는?

a.cpp

```
int g_nGlobalValue = 0;
```

b.cpp

```
#include <stdio.h>

extern int g_nGlobalValue;

int main(void)
{
        printf("%d\n", g_nGlobalValue);
        return 0;
}
```

a.cpp

```
static int g_nGlobalValue = 0;
```

b.cpp

```
#include <stdio.h>

extern int g_nGlobalValue;

int main(void)
{
        printf("%d\n", g_nGlobalValue);
        return 0;
}
```

함수 안으로 범위를 한정하는 정적 변수

다음 코드들의 실행 결과는 무엇일까? 왼쪽과 오른쪽 각각

Main.cpp

```
#include <stdio.h>


int IssueNewId(void)
{
	static int nLastId = 0;
	return nLastId++;
}


int main(void)
{
	printf("%d,", IssueNewId());
	printf("%d,", IssueNewId());
	printf("%d₩n", IssueNewId());
	return 0;
}
```

a.cpp

```
#include <stdio.h>


static void TestFunc(void)
{
		printf("hello??\n");
}
```

main.cpp

```
void TestFunc(void);


int main(void)
{
		TestFunc();
		return 0;
}
```

함수 안으로 범위를 한정하는 정적 변수

참고로 다음 두 코드의 의미는 동일합니다.

하지만 어떤 코드가 더 잘 짠 코드일까요?

Main.cpp

```
#include <stdio.h>

int IssueNewId(void)
{
	static int nLastId = 0;
	return nLastId++;
}

int main(void)
{
	printf("%d,", IssueNewId());
	printf("%d,", IssueNewId());
	printf("%d₩n", IssueNewId());
	return 0;
}
```

Main.cpp

```
#include <stdio.h>

static int nLastId = 0;
int IssueNewId(void)
{
	return nLastId++;
}

int main(void)
{
	printf("%d,", IssueNewId());
	printf("%d,", IssueNewId());
	printf("%d₩n", IssueNewId());
	return 0;
}
```

정적 멤버 함수

멤버 함수와 정적 멤버 함수의 차이는 다음과 같습니다.

멤버 함수

```
#include <stdio.h>

class CTest
{
public:
        void Func(void)
        {
                printf("hello??₩n");
        }
};

int main(void)
{
        CTest test;
        test.Func();
        return 0;
}
```

정적 멤버 함수

```
#include <stdio.h>

class CTest
{
public:
        static void Func(void)
        {
                printf("hello??₩n");
        }
};

int main(void)
{
        CTest::Func();
        return 0;
}
```

함수 종류	유효 범위	정의 위치
전역 함수	전역	네임스페이스나 바깥 영역
정적 함수	해당 파일 한정	네임스페이스나 바깥 영역
멤버 함수	인스턴스 한정	클래스나 구조체
정적 멤버 함수	전역(클래스 한정자 사용)	클래스나 구조체

정적 멤버 변수

다음과 같이 두 가지 정적 변수 중 어떤 것이 훌륭할까요?

클래스 안의 정적 변수	정적 멤버 함수 안의 정적 변수

class CTest
{
static int m_nIncreaseValue = 0;
 	
public:
static void Func(void)
{
m_nIncreaseValue++;
printf("called count:%d\n",
m_nIncreaseValue);
}
};
	class CTest
	{
	public:
	static void Func(void)
	{
	static int m_nIncreaseValue = 0;
	m_nIncreaseValue++;
	printf("called count:%d\n",
	m_nIncreaseValue);
	}
	};

정적 변수의 끝판 왕, 싱글턴 패턴

싱글턴 패턴은 다음과 같은 특징을 갖고 있습니다.

- 클래스의 객체가 오직 한 개만 생성되도록 제한함
- 생성과 소멸 시점을 정할 수 있음

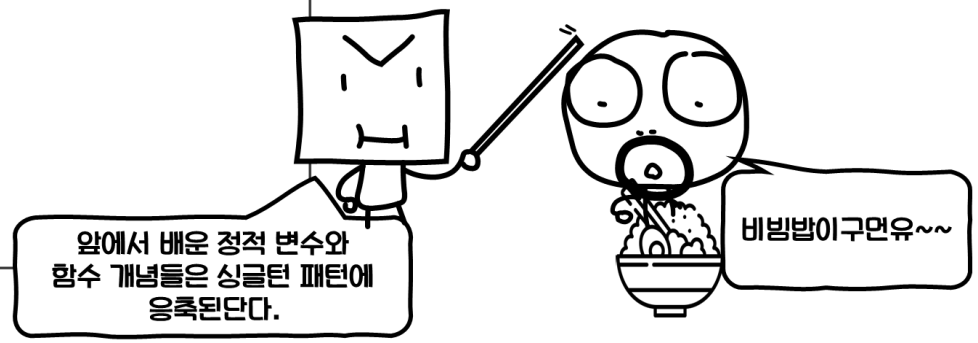
  1) 생성 시점은 싱글턴이 처음 사용될 때이다.

  ➔ 정적 변수는 그것을 품고 있는 함수가 최초로 불릴 때 생성된다.

  2) 소멸 시점은 프로그램 종료 직전이다.

  ➔ 정적 변수와 전역 변수는 main함수 탈출 이후 프로그램 종료 직전에 소멸된다.

```
class CGlobalDoc
{
        CGlobalDoc(void);
        ~CGlobalDoc(void);

public:
        static CGlobalDoc* GetInstance(void)
        {
                static CGlobalDoc instance;
                return &instance;
        }
};

inline CGlobalDoc* Doc(void)
{
        return CGlobalDoc::GetInstance();
}
```

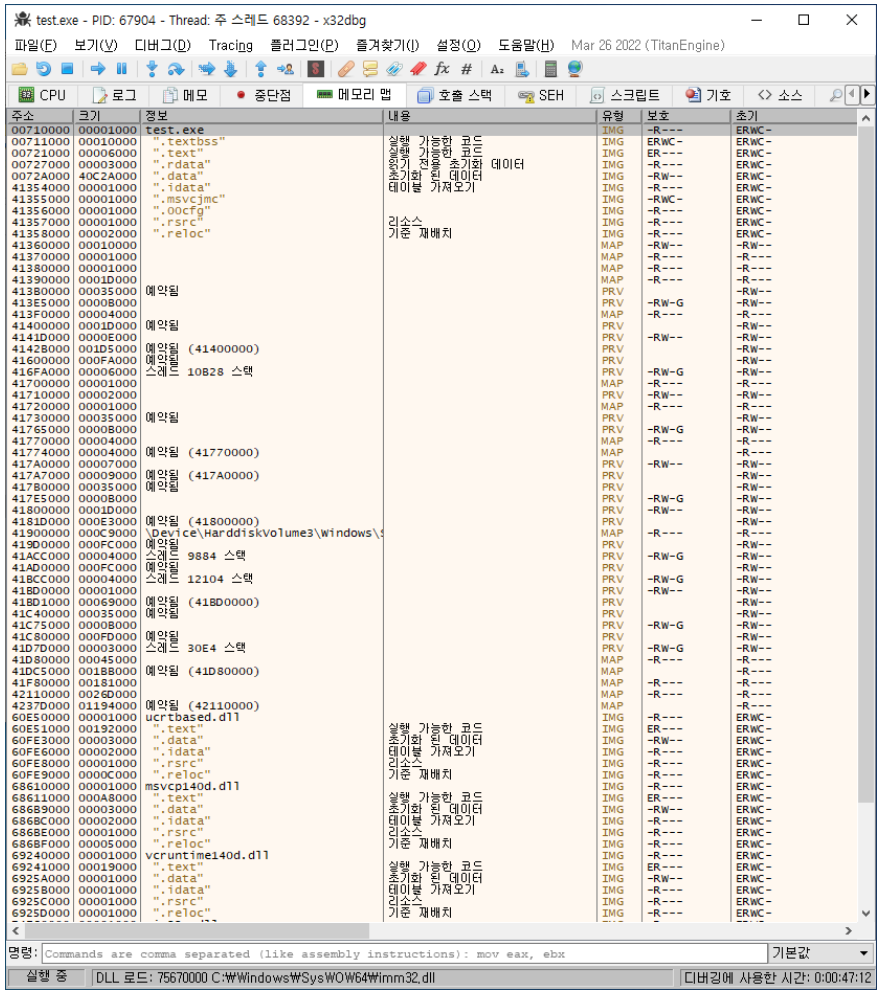


링 메모리

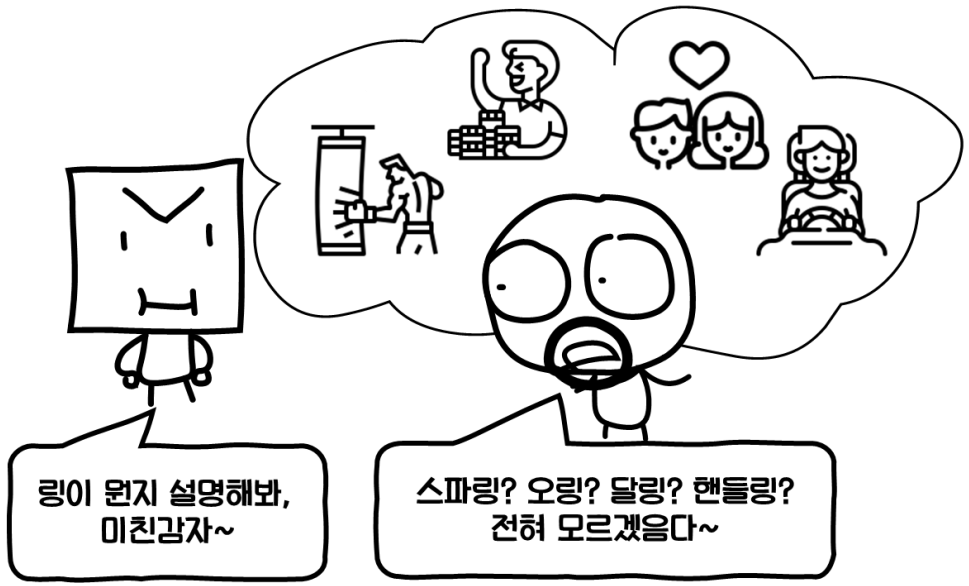
링 메모리 정복하기: 전역변수나 정적변수는 왜 크게 선언해도 될까?

무려 1GB나 되는 메모리 버퍼인데, 전역변수로 선언하면 잘 동작합니다. (정적 변수도 마찬가지)
메모리 상태를 조회해보면 전역변수로 인해 뒤에 쓰이는 메모리 오프셋이 밀릴 뿐, 동작에는 문제가 없습니다.
저장되는 공간이 다르기 때문이죠.

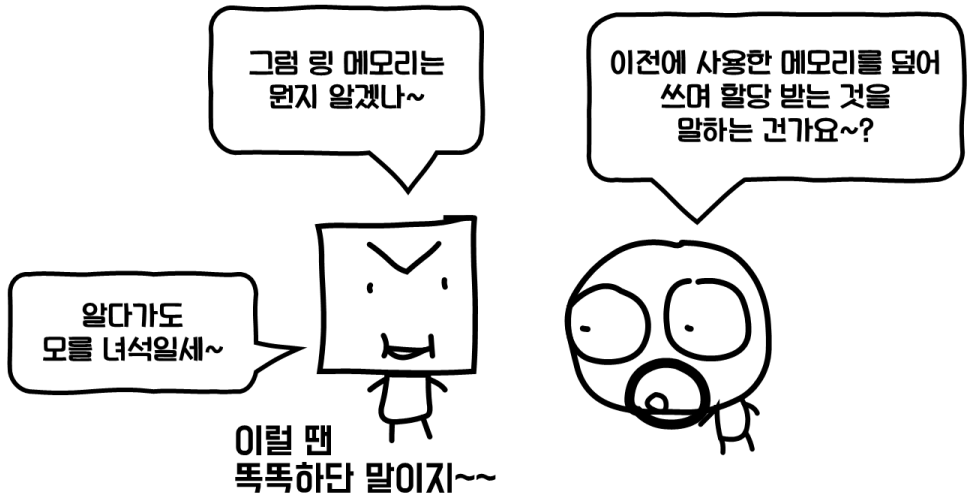
```
#include "pch.h"


char szBuffer[1000 * 1024 * 1024] = { 0, };


int main()

{

    printf("%08X", szBuffer);

    Sleep(1000 * 1000);

    return 0;

}
```



링 메모리 정복하기 : 메모리의 끝판 왕, 링 메모리



링 메모리 정복하기 : 메모리의 끝판 왕, 링 메모리



옆 그림에서는 같은 크기의 메모리를 10개로 잘라서 분할한 것을 나타냈는데 이렇게 동일한 크기를 가져다 쓸 수도 있고 공간의 효율성을 위해 필요한 만큼씩 잘라 쓸 수도 있습니다.

단 하나의 사용 예제가 모든 것을 설명합니다.

```
int main()
{
    CRingBuffer memory;
    memory.Create(1000);

    for (int I = 0; I < 10; i++)
    {
        void* pMemoryToken = memory.Alloc(128);
        ...
    }

    memory.Destroy();

    return 0;
}
```

질문, 여기서 1000을 넘어서는 순간 어떻게 동작할까요?

다음 코드의 실행 결과는?

```
#include "pch.h"

int main()
{
    CRingBuffer RingBuffer;
    RingBuffer.Create(5);

    std::list<char*> listTest;
    for (char i = 'A'; i <= 'Z'; i++)
    {
        char* pBuffer = (char*)RingBuffer.Alloc(1);
        (*pBuffer) = i;
        listTest.push_back(pBuffer);
    }

    for (char* pValue : listTest)
        printf("%c", *pValue);

    RingBuffer.Destroy();
    return 0;
}
```

모든 메모리의 장점을 아우르는 링 메모리를 이해하신 여러분은

이제 새로운 사람이 되었습니다!!

고생 많으셨습니다!