

빌드, 메모리, 포인터

C++ 1주차

목차

- 빌드
 - 빌드의 구성
 - 전처리 지시문
 - 조건부 컴파일 지시문
 - 미리 컴파일된 헤더
 - 링크 에러
- 메모리
 - 가상 메모리
 - 메모리 할당/해제
- 포인터
 - 포인터
 - 참조 타입
 - 함수 포인터

빌드 : 빌드의 구성

- 빌드의 단계 : 전처리 -> 컴파일 -> 링크
- 전처리
 - 전처리 지시문을 통해 컴파일 시점에 컴파일이 가능하도록 원래 소스 구문을 복구하는 과정

```
#include <stdio.h>

int main(void)
{
    printf("%s", "Hello World!");
    return 0;
}
```

빌드 : 빌드의 구성

- 컴파일
 - 사람이 이해할 수 있는 C++언어로 작성된 소스파일을 기계어 파일로 변환하는 작업
 - 컴파일의 대상 : 소스 파일, 즉 cpp 파일
 - > 헤더파일은 cpp파일 안의 #include구문에 의해 포함될 뿐 컴파일의 대상이 아님, 즉 헤더파일은 cpp파일과 같이 컴파일됨
 - 소스파일(cpp)을 컴파일하면 목적파일 생성

```
#include <stdio.h>

int main(void)
{
012A1810  push    ebp
012A1811  mov     ebp,esp
012A1813  sub     esp,000h
012A1819  push    ebx
012A181A  push    esi
012A181B  push    edi
012A181C  lea     edi,[ebp-000h]
012A1822  mov     ecx,30h
012A1827  mov     eax,00000000h
012A182C  rep stos dword ptr es:[edi]
012A182E  mov     ecx,offset _48B431D3_소스@cpp (012AC003h)
012A1833  call    @__CheckForDebuggerJustMyCode@4 (012A1212h)
    printf("%s", "Hello World!");
012A1838  push    offset string "Hello World!" (012A7B30h)
012A183D  push    offset string "%s" (012A7B40h)
012A1842  call    _printf (012A104Bh)
012A1847  add     esp,8
    return 0;
012A184A  xor     eax,eax
}
012A184C  pop     edi
012A184D  pop     esi
012A184E  pop     ebx
012A184F  add     esp,000h
012A1855  cmp     ebp,esp
012A1857  call    __RTC_CheckEsp (012A121Ch)
012A185C  mov     esp,ebp
012A185E  pop     ebp
012A185F  ret
```

빌드 : 빌드의 구성

- 링크

- 목적파일들을 적절히 연결하는 과정, 즉 목적 파일들이 프로세스 메모리 가상 공간에 적재되었을 때 서로 맞물릴 수 있도록 각각의 주소를 변경해 주는 것을 의미한다.

- 예시

```
#include <stdio.h>

int main(void)
{
    printf("%s", "Hello World!");
    return 0;
}
```

- printf를 사용하기 위해서는 CRT라이브러리(C Runtime Library)를 사용해야 한다.
- printf는 printf.c가 컴파일된 printf.obj에 존재한다.
- 예시 파일의 printf호출은 printf.obj파일이 프로세스 메모리에 올라간 주소로 변형시켜 주는 것이 링크이다. (정적 라이브러리 기준)

빌드 : 전처리 지시문

- #include : 여기에 사용되는 파일이 곧 헤더파일
 - "" : #include문이 포함된 디렉터리 -> /컴파일러 옵션(IDE의 추가 포함 경로) 지정 경로
-> INCLUDE환경 변수 지정 경로
 - <> : /컴파일러 옵션(IDE의 추가 포함 경로) 지정 경로 -> INCLUDE환경 변수 지정 경로
 - 일반적으로 구분을 위해 직접 작성한 것은 ""로 기존 라이브러리는 <>로 사용
- 확장자에 상관없이 헤더파일임, 문법적 제한 없음 -> 컴파일때 설명한대로 그냥 붙여넣기

빌드 : 전처리 지시문

- #include 주의사항
 - 전역변수나 전역함수를 정의해서는 안된다. 오직 선언만 할 것.
 - 전역변수의 경우 extern 키워드 붙여줄 것.
 - > 전역변수 자체가 아닌 외부에 있다는 것을 알려줌(선언).

```
//common.h
int val;           //중복정의 - 링크에러
static int sval;   //정상 - static은 오브젝트 파일 내에서만 유효
void func() {...}  //중복정의 - 링크에러

//header1.h
#include "common.h"
...

//header2.h
#include "common.h"
...

//main.cpp
#include "header1.h"
#include "header2.h"

int main() {...}
```

빌드 : 전처리 지시문

- `#define` : `#define identifier token-string -> identifier`를 `token-string`으로 대체
`#define identifier -> identifier`를 정의만 한다. 만약 소스에 `identifier`가 나타나면 컴파일 에러가 발생한다. 조건부 컴파일(`#if defined` 혹은 `#ifdef`)시 함께 테스트하는데 사용된다.
- `identifier`를 매크로라고 부르기도 함.
- `#define identifier(identifier, ... , identifier) token-string` 처럼 함수 매크로로 사용할 수 있음
-> 권장하지 않음, 인라인 함수 쓸 것(이것도 보통은 컴파일러가 알아서 변환해 줌)

빌드 : 조건부 컴파일 지시문

- 조건부 컴파일 지시문 : #if, #ifdef, #ifndef, #else, #elif, #endif

```
#include <string>

#ifdef UNICODE                //UNICODE는 IDE에서 지정한 매크로
    typedef wstring TSTR;     //유니코드 string 클래스
    #define TCSTR(str) L##str //L"str"
#else
    typedef string TSTR;      //멀티바이트 string클래스
    #define TCSTR(str) str
#endif

using namespace std;

int main()
{
    TSTR str = TCSTR("Hello World!");
}
```

빌드 : 조건부 컴파일 지시문

- #pragma once : 헤더파일을 한번만 포함시키라는 의미. 표준 아님, VC++과 GCC지원.
- 조건부 컴파일 지시문을 통해 헤더파일 한번만 포함시키기

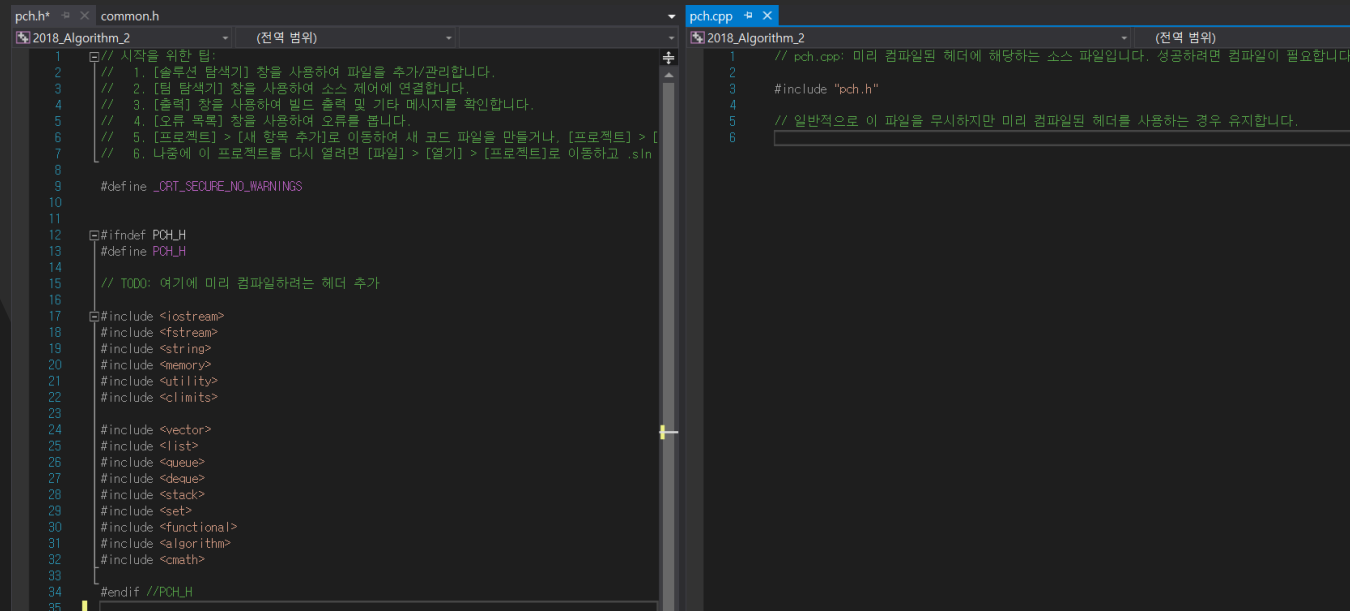
```
//common.h
#ifndef _common_H
#define _common_H

...

#endif
```

빌드 : 미리 컴파일된 헤더

- 미리 컴파일된 헤더 : 헤더를 미리 컴파일해 두는 것
-> 컴파일 시간을 그만큼 단축시킬 수 있음.
컴파일시 (헤더 + 소스)를 합쳐 컴파일함.
중복되는 헤더가 많다면 그만큼 컴파일 시간이 늘어남.



```
pch.h* common.h
1 // 시작을 위한 팁:
2 // 1. [솔루션 탐색기] 창을 사용하여 파일을 추가/관리합니다.
3 // 2. [탐색기] 창을 사용하여 소스 제어에 연결합니다.
4 // 3. [솔루션] 창을 사용하여 빌드 속성 및 기타 메시지를 확인합니다.
5 // 4. [오류 목록] 창을 사용하여 오류를 봅니다.
6 // 5. [프로젝트] > [새 항목 추가]로 이동하여 새 코드 파일을 만들거나, [프로젝트] > [
7 // 6. 나중에 이 프로젝트를 다시 열려면 [파일] > [열기] > [프로젝트]로 이동하고 .sln
8
9 #define _CRT_SECURE_NO_WARNINGS
10
11
12 #ifndef PCH_H
13 #define PCH_H
14
15 // TODO: 여기에 미리 컴파일하려는 헤더 추가
16
17 #include <iostream>
18 #include <fstream>
19 #include <string>
20 #include <memory>
21 #include <utility>
22 #include <limits>
23
24 #include <vector>
25 #include <list>
26 #include <queue>
27 #include <deque>
28 #include <stack>
29 #include <set>
30 #include <functional>
31 #include <algorithm>
32 #include <cmath>
33
34 #endif //PCH_H
35

pch.cpp
1 // pch.cpp: 미리 컴파일된 헤더에 해당하는 소스 파일입니다. 성공하려면 컴파일이 필요합니다.
2
3 #include "pch.h"
4
5 // 일반적으로 이 파일을 무시하지만 미리 컴파일된 헤더를 사용하는 경우 유지합니다.
6
```

빌드 : 링크 에러

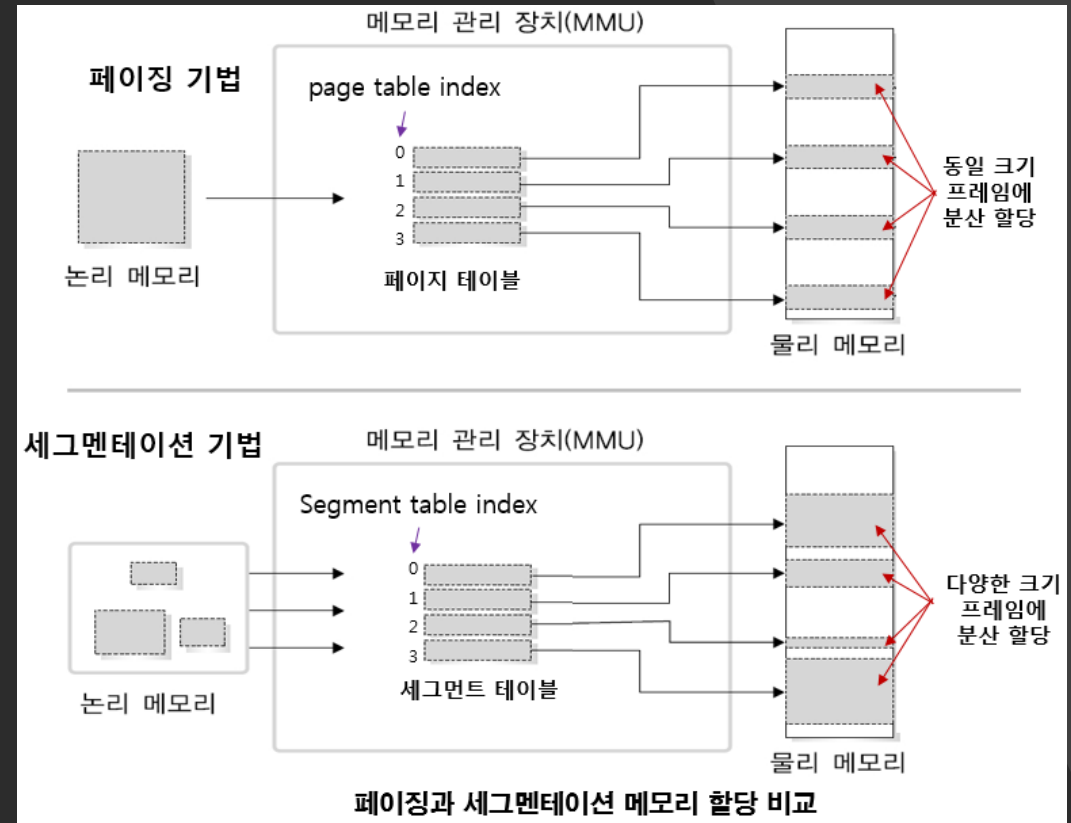
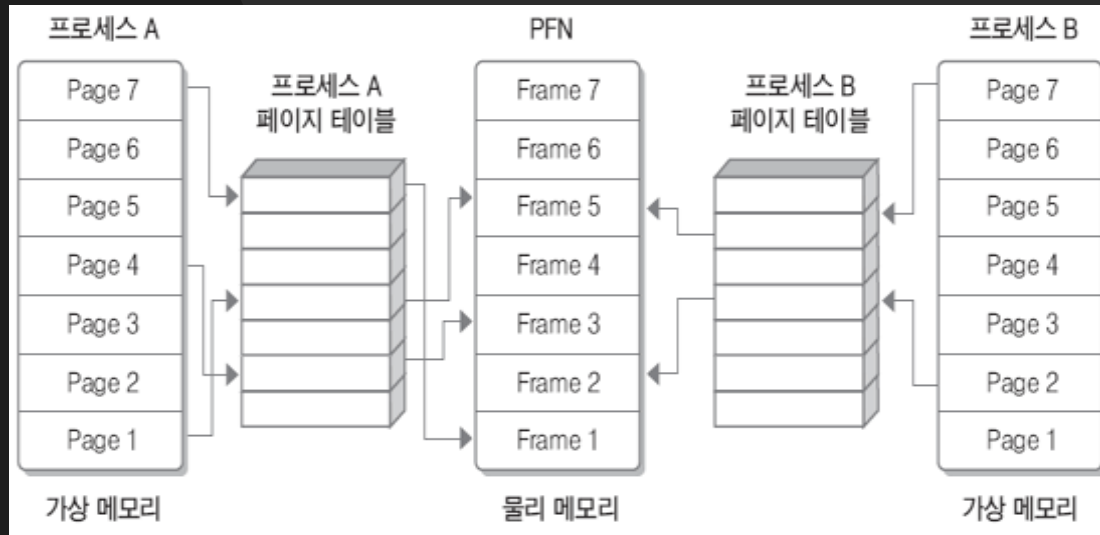
- 여러 번 정의된 기호
 - > 정의가 중복된 경우
 - Ex) 다른 파일에 시그니처까지 일치하는 함수가 여러 번 정의됨.
- 확인할 수 없는 외부 참조
 - > 정의를 하지 않았거나 찾을 수 없을 경우 발생
 - Ex) 다른 파일에 함수를 선언만 하고 정의하지 않음

메모리 : 가상 메모리

- 각각의 프로세스에게 독립적으로 부여되는 가상의 메모리 공간
- 32bit x86 시스템에서 개별 프로세스는 4GB의 메모리 공간을 부여 받음, 이 영역은 해당 프로세스만 접근할 수 있게 보호됨.
 - > IA-32는 주소를 가리킬 수 있는 레지스터 크기가 32bit임, 즉 2^{32} 개(약 42억)의 주소를 가짐, 각각의 주소가 1Byte씩 가리키므로 4GB의 메모리를 부여 받게 됨.

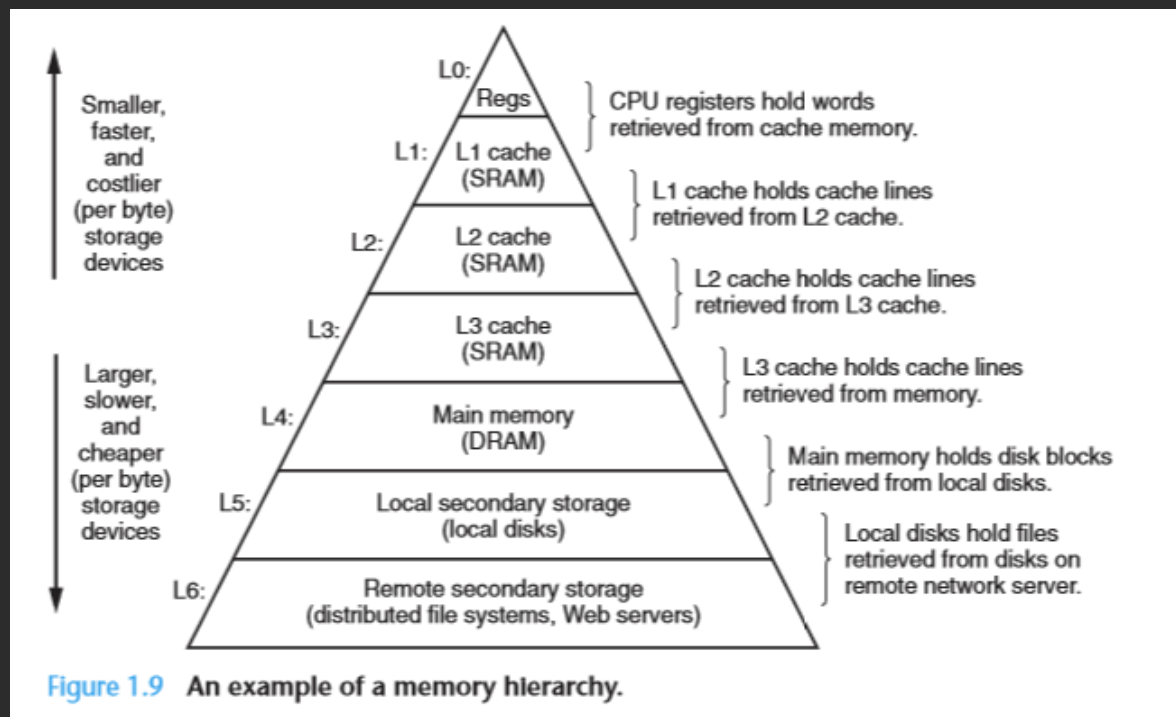
메모리 : 가상 메모리

- 메모리 페이징
- 메모리 세그먼트 테이션



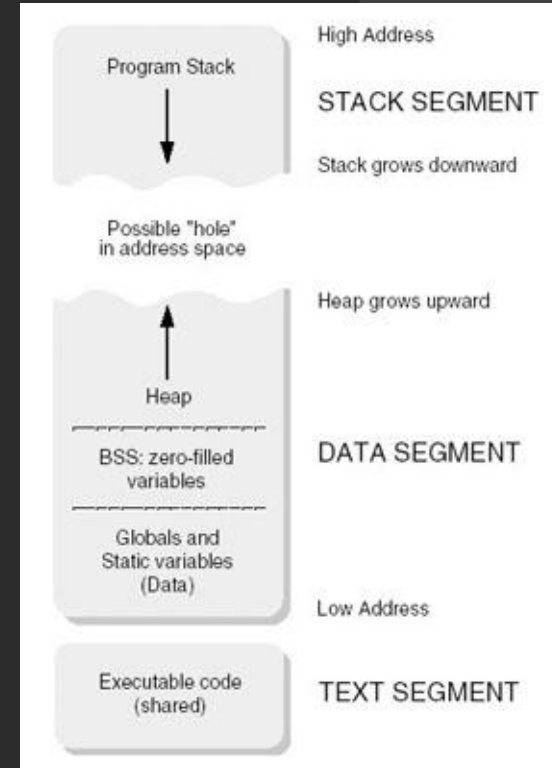
메모리 : 가상 메모리

- 메모리 계층구조



메모리 : 가상 메모리

- 가상 메모리 구조
 - TEXT : 이 영역은 읽기만 가능
 - DATA : static변수도 여기에 저장
 - BSS : 초기화되지 않은 static변수 혹은 전역변수 저장
 - HEAP : 힙 관리자에 의해 할당
 - STACK : 스레드당 하나씩 생성
- 현대 가상메모리 구조는 사진 같지 않음



메모리 : 메모리 할당/해제

- 앞서 살펴본 대로 가상 메모리는 페이지 단위로 물리메모리에 할당됨
- C/C++의 CRT가 제공하는 malloc, new의 근원은 OS의 메모리 할당 API나 시스템콜임
 - > 힙 관리자에 의해 처리됨
 - 프로세스 시작시 힙 관리자는 가상메모리를 큰 덩어리로 할당 -> malloc/new 요청 시 할당받은 메모리 적절히 분할해 돌려줌
- malloc : `void* malloc(size_t size);` -> size만큼 메모리 블록 할당 후 포인터 반환
- free : `void free(void* memblock);` -> memblock 미할당영역으로 변경
 - free의 인자로 할당된적 없는 주소를 넘긴다면 : 다음 할당 요청에 영향 줄 수 있음

메모리 : 메모리 할당/해제

- new/delete : 내부적으로 malloc/free 호출
 - 차이점 : 생성자와 소멸자의 호출 -> 클래스 객체는 생성되면서 생성자 호출, 소멸되면서 소멸자 호출되게 만들어짐, malloc/free로는 생성자와 소멸자를 호출할 수 없음
- new []/delete [] : 배열타입 객체 할당/해제 시 사용
 - new/delete처럼 malloc/free를 그대로 호출하지 않음, 배열 요소 저장을 위해 4Byte추가 할당
 - new []로 할당받은 객체 delete로 해제하면 할당 정보를 찾을 수 없어 정상적으로 해제되지 않고 힙 충돌 발생, 메모리 해제 주의할 것

포인터 : 포인터

- 포인터는 메모리 주소를 가리키는 객체임
- 객체의 타입과 포인터의 원천타입이 반드시 같을 필요는 없음
 - > int타입을 가리키기 위해 꼭 int*타입 포인터를 쓸 필요는 없음
- `TYPE* p` 가 나타내는 의미는 p가 가리키는 주소를 기준으로 `sizeof(TYPE)`크기의 블록의 비트 상태를 TYPE에 의해 부여하여 대응시키겠다는 의미

포인터 : 참조 타입

- 참조 타입 : C++에 도입된 객체를 가리키기 위한 타입
 - 그저 별칭정도가 아님, 메모리 공간이 따로 마련됨
 - 참조는 초기화시 주소(&)연산자가 생략되고 직접 사용시 간접(*)연산자가 생략됨

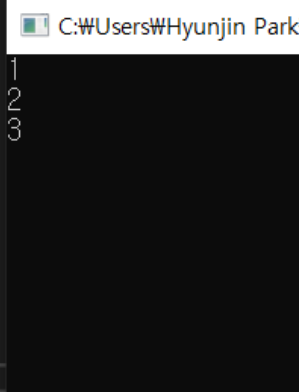
```
#include <iostream>

using namespace std;

int main()
{
    int a = 1;
    cout << a << endl;

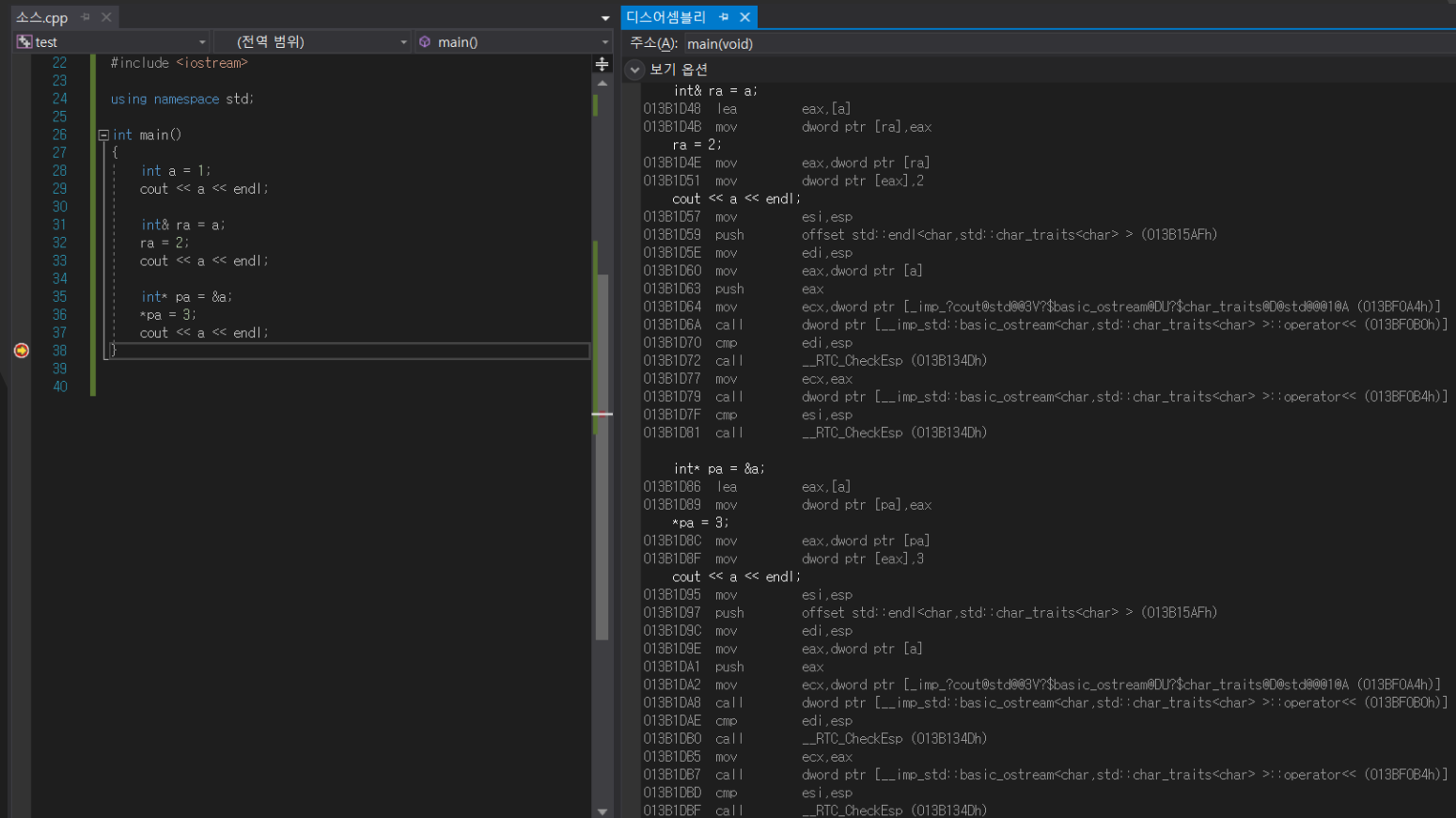
    int& ra = a;
    ra = 2;
    cout << a << endl;

    int* pa = &a;
    *pa = 3;
    cout << a << endl;
}
```



포인터 : 포인터

- 포인터와 참조의 차이 : 완전히 같은 방식으로 동작함



The screenshot displays a C++ IDE with two panels. The left panel shows the source code of a program named 'test.cpp'. The right panel shows the disassembled assembly code for the 'main' function.

Source Code (test.cpp):

```
22 #include <iostream>
23
24 using namespace std;
25
26 int main()
27 {
28     int a = 1;
29     cout << a << endl;
30
31     int& ra = a;
32     ra = 2;
33     cout << a << endl;
34
35     int* pa = &a;
36     *pa = 3;
37     cout << a << endl;
38 }
39
40
```

Disassembly (디스어셈블리):

주소(A): main(void)

보기 옵션

```
int& ra = a;
013B1D48 lea     eax,[a]
013B1D4B mov     dword ptr [ra],eax
ra = 2;
013B1D4E mov     eax,dword ptr [ra]
013B1D51 mov     dword ptr [eax],2
cout << a << endl;
013B1D57 mov     esi,esp
013B1D59 push    offset std::endl<char,std::char_traits<char> > (013B15AFh)
013B1D5E mov     edi,esp
013B1D60 mov     eax,dword ptr [a]
013B1D63 push    eax
013B1D64 mov     ecx,dword ptr [__imp_?cout@std@@@3V?@basic_ostream@DU?@char_traits@D@std@@@10A (013BF0A4h)]
013B1D6A call    dword ptr [__imp_std::basic_ostream<char,std::char_traits<char> >::operator<< (013BF0B0h)]
013B1D70 cmp     edi,esp
013B1D72 call    __RTC_CheckEsp (013B134Dh)
013B1D77 mov     ecx,eax
013B1D79 call    dword ptr [__imp_std::basic_ostream<char,std::char_traits<char> >::operator<< (013BF0B4h)]
013B1D7F cmp     esi,esp
013B1D81 call    __RTC_CheckEsp (013B134Dh)

int* pa = &a;
013B1D86 lea     eax,[a]
013B1D89 mov     dword ptr [pa],eax
*pa = 3;
013B1D8C mov     eax,dword ptr [pa]
013B1D8F mov     dword ptr [eax],3
cout << a << endl;
013B1D95 mov     esi,esp
013B1D97 push    offset std::endl<char,std::char_traits<char> > (013B15AFh)
013B1D9C mov     edi,esp
013B1D9E mov     eax,dword ptr [a]
013B1DA1 push    eax
013B1DA2 mov     ecx,dword ptr [__imp_?cout@std@@@3V?@basic_ostream@DU?@char_traits@D@std@@@10A (013BF0A4h)]
013B1DA8 call    dword ptr [__imp_std::basic_ostream<char,std::char_traits<char> >::operator<< (013BF0B0h)]
013B1DAE cmp     edi,esp
013B1DB0 call    __RTC_CheckEsp (013B134Dh)
013B1DB5 mov     ecx,eax
013B1DB7 call    dword ptr [__imp_std::basic_ostream<char,std::char_traits<char> >::operator<< (013BF0B4h)]
013B1DBD cmp     esi,esp
013B1DBF call    __RTC_CheckEsp (013B134Dh)
```

포인터 : 함수 포인터

- 포인터가 가리키는 대상은 한계가 없음, 메모리를 점유하는 모든 것을 가리킬 수 있음
- 전역함수 포인터 작동 : 함수의 주소를 담고있는 포인터, 사용시 코드영역에서 함수 call
- 멤버함수 포인터와 가상 상속 클래스 멤버함수 포인터는 전역함수 포인터와 동작 다름
 - 멤버함수 포인터에는 시그니처에 클래스타입 정보가 추가됨
 - 컴파일러 마다 차이 보임

코드 리뷰

0주차 과제