기억클래스

Auto / extern / register / static

Register : 레지스터를 변수 범용처럼 쓰길 희망한다. 이공간을 원하는대로 해봐라. 진짜 레지스터에 들어간다는 보장은 아니고 할수있으면 해봐 . 아무도 안쓴다.

Extern : 전역변수로 어딘가 선언했는데 어딘가에 있다. 이런게 어디에 있으니까 일단은 허용해 주세요 이런 의미

속성 수식자

Const / volatile

Volatile : 컴파일러 최적화 방지. 온전히 내가 짠 그대로 컴파일 하겠다. 캐시메모리를 쓰지않고 직접 메모리에 들어간다? 절대 아니다. 무조건 모든 변수는 캐시를 거쳐야 한다. 근데 여기서 캐시를 하지 않는다는 이야기는 뭐냐? 우리가 아는 cpu의 캐시가 아니라 실제로는 g\_a와 비교해야하지만 이 값을 ecx에 넣어두고 이것과 비교한다. 레지스터는 빠르니까 이게 더 낫다고 판단하는 것. Ecx 레지스터를 g\_a의 대용으로 쓴 것. 이 정도 수준의 캐시를 의미하는거지 cpu의 캐시메모리라는 말이 아니다. 레지스터를 캐시메모리처럼 사용한다는 것. 코드안에서 판단해서 재사용한다는 뜻. 결론은 그냥 최적화 안하는 역할밖에 없다. 레지스터에 있는거 쓰지않고 정말로 g\_a에 있는거 쓰겠다는 의미. 멀티스레드의 입장에서 하나씩 ++이 되길 바라는데 컴파일러가 최적화시켜서 한번에 코드를 보고 한방에 5를 더하는건 멀티스레드에서 의도한 결과가 아니니까.

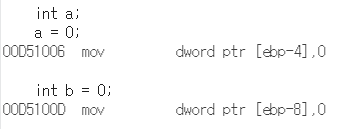
64비트라고 항상 int가 64비트가 되는게 아니다. 데이터모델의 4종류중에 어떤건지 파악해야한다. 윈도우는 LLP64를 쓰기 때문에 int가 32비트이다. 근데 어떤건 64비트인 모델도 있으니까 확인을 해야한다. 🡪 윈도우는 LLP64 모델을 쓴다고는 하는데 나머지 모델일땐 어떻게 알아보나? 직접 데이터 사이즈를 알아보던지 하면 된다.

보수 : 보충해주는 수. 자릿수를 올리는 수. 한 자리를 올리는데 필요한 수. 7의 보수는 3 . 이걸갖고 뭘 하냐? 4-3을 하는데 컴퓨터에는 뺄셈이 없다. 그래서 덧셈을 할건데 보수를 덧셈한다. 7의 보수는 3이니까 4 + 7을 해서 올라간수를 버리면 1이 나오는 것. 7 – 4 = 3 인데 7 + 6 = 13 해서 십의자리 올라간거 버리고 3. Int a에 -4가 들어가있다면 사실 보수인 6이들어가있는 것. 근데 그냥 6이면 구별이안가니까 최상위비트를 부호비트로 둔다. 이진수니까 2의 보수를 쓰고있음. Int a = -1이면 1111 1111 임. 왜일까? 1은 0000 0001 이것의 보수를 구하기 위해 일단 반전 1111 1110 여기에 + 1 이게 2의 보수. 그냥 반전이 보수가 아니라 우린 2의보수가필요하다. 1101 1010 이 있다면? 1바이트짜리 숫자. 이건 일단 음수. 그러니까 -1 하면 1101 1001. 이걸 반전하면 0010 0110 이 값은 38인데 사실 이 값은 -38이었던 것. 0xda = -38이 되는 것. Char x = 0x8c; 가 있다면

1000 1100 니까 음수. -1하면 1000 1011. 반전하면 0111 0100. 4 + 16 + 32 + 64 = -116

🡪디스어셈블리로 봤을 때 sub 하고 돌아가는것도 다 내부적으로는 2의 보수가 더해지는건가?? 맞다

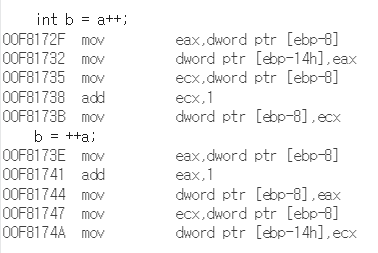
Int a; a=0; 과 int b =0; 뭐가 더 좋을까? 선언이라는 과정은 사실 존재하지 않으니 변수에 0을 넣는다는 의미밖에 없다. 그냥 편의대로 써라.



전위 연산자와 후위 연산자 둘중에 최적화 차원에서 뭐가 더 나을까?

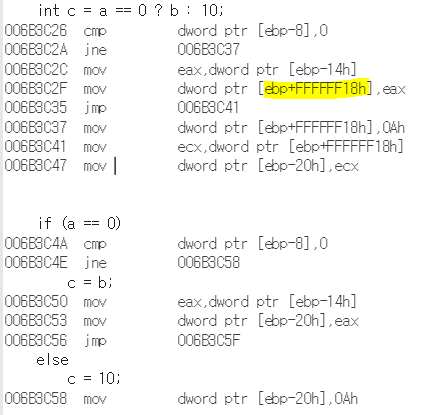
++a 는 나자신을 올리고 대입.

A++는 값을 던지고 나를 증가. 이건 사본을 만들어야함. 이미 리턴을 하고 ++을 한다는건 말이 안됨. 사본을 리턴하는 코드가 될 수밖에 없음. 과거의 데이터를 리턴해야하니까. 그래서보통 후위연산자가 성능이 떨어진다고 하는데 이건 객체일때의 이야기. A가 무슨 class 였어서 뭐 다음 노드를 넘어가는거라던가 그런경우엔 해당이 됨. iterator가 ++로 넘어가니까. 후위는 항상 내부에서 사본을 만든다. 근데 일반 변수의 경우에는 상관이 없다. **순서의 차이만 있을 뿐**.

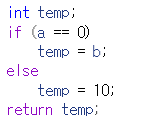


근데 좀 더 고차원적으로 접근하면? Cpu의 명령어 재배치가 이루어지는 상황에 따라서 후위증가가 조금 더 좋을 수 있다. 전위는 반드시 a를 증가시키고 전달해야한다. 증가가 늦어지면 값 전달도 늦어짐. 근데 후위는 값 전달과 후에 증가하는건 관련이 없음.

삼항연산자. 길이의 문제 외에 가끔가다 이게 더 성능이 좋아서 이걸 쓴다는 사람이 있다. 절대 아니다.

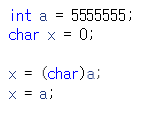


삼항 연산자는 오히려 내가 선언하지 않은 변수 하나를 더 만들어서 거기에 값을 넣고 마지막에 리턴한다. 직관적이지도 않고 오히려 저런 작업이 한번 더 들어가 있다.

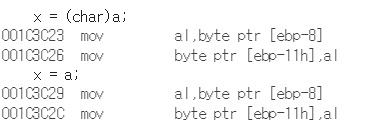


삼항연산자 대로 코드를 짠다면 이런 형태가 나올 것이다.

캐스팅이라는건 뭘까? 캐스팅은 존재하지 않는다 결국에는.



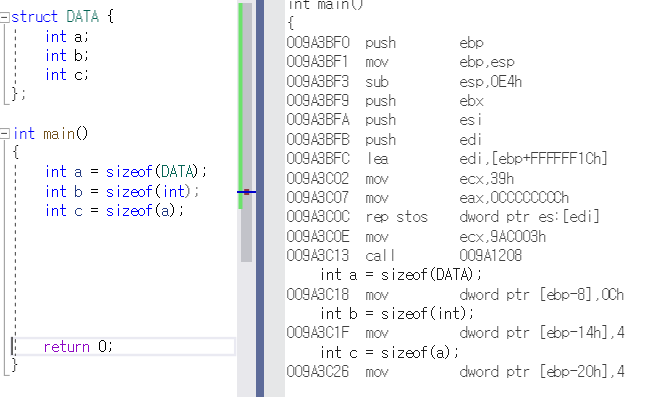
이런건 그냥 사실 컴파일러에게 알려주는거다. 이건 내가 실수한게 아니라 내가 의도한거다. 라고.



디스어셈블리 창을 보면 그냥 al에 넣는다. 값을 잘라서 1바이트만큼 뽑아서 char에 넣을 뿐.

그럼 이건 언제나 괜찮을까? 만약 포인터를 넣어주는 행위를 했다면? 64비트 환경에 대해서는 잘못 작동할 수 있다. Int는 4바이트인데 포인터는 그때 8바이트가 되니까.

Sizeof가 연산이 되는 시점은 컴파일 시점.



연산자 우선순위 - 산술연산자가 제일 높다. 쉬프트는 곱하기 덧셈보다 우선순위가 낮다.

조건문에서 &&로 묶이는 것들이 많으면 false가 나오는 애를 앞쪽에. ||로 묶이는 것들이 많으면 true가 나오는 애들을 앞쪽에. 내부적으로는 앞부터 if문을 수행하게 되니까.

