**※ What is the difference between memmove and memcpy?**

→ With memcpy, the destination cannot overlap the source at all. With memmove it can. This means that memmove might be very slightly slower than memcpy, as it cannot make the same assumptions.

For example, memcpy might always copy addresses from low to high. If the destination overlaps after the source, this means some addresses will be overwritten before copied. memmove would detect this and copy in the other direction - from high to low - in this case. However, checking this and switching to another (possibly less efficient) algorithm takes time.

→ memcpy에서는, 목적지에 원본이 완전히 복사되지 않는 상황이 발생할 수도 있다. memmove는 반드시 복사된다. 이것은 memmove는 memcpy보다 약간 느리다는 것을 의미한다. 예를 들어, memcpy는 아마 항상 낮은 주소에서 높은 주소로 복사를 진행할 것이다. 만약 목적지가 원본 뒤에서(메모리 주소상에서) 덮어씌워진다면, 이것은 어떤 주소는 복사되기전에 덮어씌워진다는 것을 의미한다. 이러한 경우에, memmove는 이것을 감지한 뒤 다른 방식으로 즉, 높은 주소에서 낮은 주소로 복사를 진행한다. 그러나 이것을 감지하고 다른 알고리즘으로 변경하여 처리해야하기 때문에 시간이 더 소요된다.

**※ When should I use perror(“…”) and fprintf(stderr, “…”)?**

→ Calling perror will give you the interpreted value of errno, which is a thread-local error value written to by POSIX syscalls (i.e., every thread has it's own value for errno). For instance, if you made a call to open(), and there was an error generated (i.e., it returned -1), you could then call perror immediately afterwards to see what the actual error was. Keep in mind that if you call other syscalls in the meantime, then the value in errno will be written over, and calling perror won't be of any use in diagnosing your issue if an error was generated by an earlier syscall.

fprintf(stderr, ...) on the other-hand can be used to print your own custom error messages. By printing to stderr, you avoid your error reporting output being mixed with "normal" output that should be going to stdout. Keep in mind that fprintf(stderr, "%s\n", strerror(errno)) is similar to perror(NULL) since a call to strerror(errno) will generate the printed string value for errno, and you can then combined that with any other custom error message via fprintf.

→ perror()를 호출하는 것은 당신에게 해석된 errno값을 전달한다 / errno변수는 POSIX 시스템콜에 의해 설정된 thread-local 에러값이다.(모든 thread는 고유의 errno값을 가지고 있다) / 예를 들어 / 당신이 open()을 호출했는데 / 어떤 에러가 발생한다면 / 당신은 즉시 perror()를 호출할 수 있다 / 나중에 실제로 어떤 에러가 발생했는지를 확인하기 위해 / 명심해라 / 만약 당신이 다른 시스템콜을 호출한다면 / 이럭 저럭 하는 사이에 / errno값은 덮어씌워질 것이고 / perror()를 호출하는 것은 쓸모없을 것이다 / 이슈를 진단하는데 / 이전의 시스템 콜에의해 발생된 문제에 대한

반면에 fprintf()는 사용될 수 있다 / 당신의 고유의 에러 메시지를 출력하기 위해 / stderr(표준에러)에 출력함으로서 / 당신은 에러 결과가 출력되는 것을 피할 수 있다 / “일반”출력창에 섞인채로 / stdout(표준출력)과 연결 되어있는 / 명심해라 / fprintf(stderr, “%s\n", stderror(errno)) 는 perror(NULL)과 유사하다는 것을 / 왜냐하면 stderror(errno)를 호출하면 errno값에 매핑되어있는 string값을 생성하기 때문이다 / 그리고 당신은 그것과 fprintf에 의해 생성되는 다른 커스텀 에러메시지를 결합할 수 있다.

**※ undefined reference error in inline function in header file in c99**

→ The inline model in C99 is a bit different than most people think, and in particular different from the one used by C++. Inline is only a hint such that the compiler doesn't complain about doubly defined symbols. It doesn't guarantee that a function is inlined, nor actually that a symbol is generated, if it is needed. To force the generation of a symbol you'd have to add a sort of instantiation *after* the inline definition: int func(int i);

Usually you'd have the inline definition in a header file, that is then included in several .c

files (compilation units). And you'd only have the above line in exactly one of the compilation

units. You probably only see the problem that you have because you are not using optimization for your compiler run.

So, your use case of having the inline in the .c file doesn't make much sense, better just use static for that, even an additional inline doesn't buy you much.

→ C99의 인라인 모델은 약간 다르다 / 대부분의 사람들이 생각하는 것과 / 그리고 특히 다르다 / C++에서 사용되는 인라인 모델과는 / 인라인 키워드는 (실제 코드에 영향을 주는 키워드 라기보다는 컴파일러에게 미리 알려주는) 일종의 힌트에 불과하다 / 그래서 컴파일러는 불평하지 않는다(신경쓰지 않는다) / 심볼의 중복정의에 대해서(인라인 함수의 심볼을 중복 정의해도) / 그것(인라인 키워드)은 보장하지 않는다 / 함수가 인라인 처리되는 것을, / 그리고 필요하다면 실제로 심볼이 생성되는 것을 / 심볼의 생성을 강제하기 위해 / 당신은 일련의 초기화(함수 선언) 문장들을 더해야한다 / 인라인 정의 뒤에

당신은 보통 헤더파일에 인라인 정의를 한다 / 그리고 그것(헤더파일)은 여러 C파일들(컴파일유닛들)에 삽입된다. 그리고 당신은 위의 문장을 선언한다 / 컴파일 유닛들 중 정확히 한 곳에서 / 당신은 아마 당신이 가지고 있는 문제를 발견할 것이다 / 오직 당신이 최적화를 사용하지 않고 있을 때만 / 당신의 컴파일러가 작동될 때

그래서 당신의 C파일에서의 인라인의 사용은 더욱 이해할 수 없다 / 차라리 static을 사용하는 것이 낫다