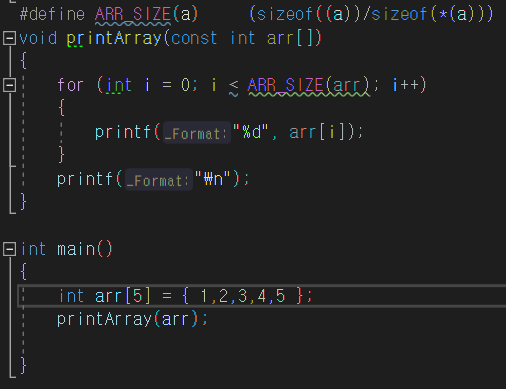
배열(Array)에 대한 여정

배열의 한계

1. 배열의 크기(길이) 정보가 없다.
2. 배열의 범위를 넘어설 경우를 알 수 없음
3. 배열의 크기를 조절할 수 없음

일단 기본적인 코드를 보자.

배열의 원소의 크기는 각각다를수있으므로 그것을 추상화하기위해 위와 같은 매크로를 사용한다.

#define ARR\_SIZE(a)

(sizeof((a))/sizeof(\*(a)))

배열전체의 크기를 구한뒤 역참조를 이용한 배열의 첫번째 자리의 원소의 크기로 나눠주면 배열의 길이가 나온다.

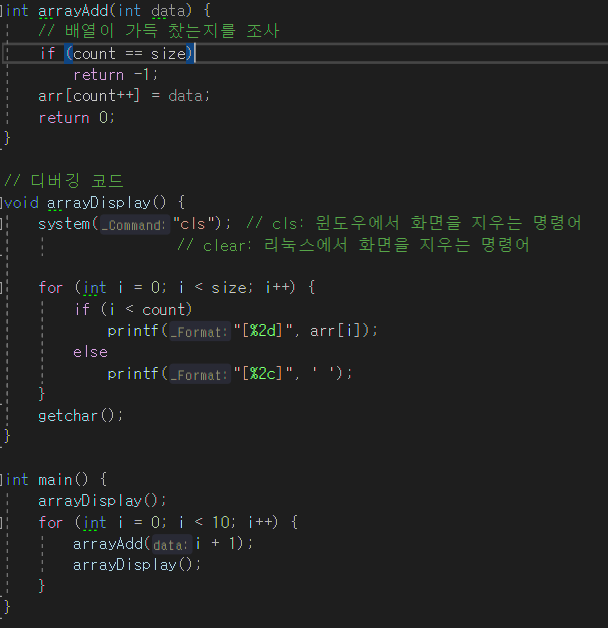
그럼 이제 정수를 저장하는 배열을 만들어보자

전역변수

int arr[5];

int size = 5; // 배열의 크기

int count; // 배열 안의 원소의 개수 & 저장할 다음 위치

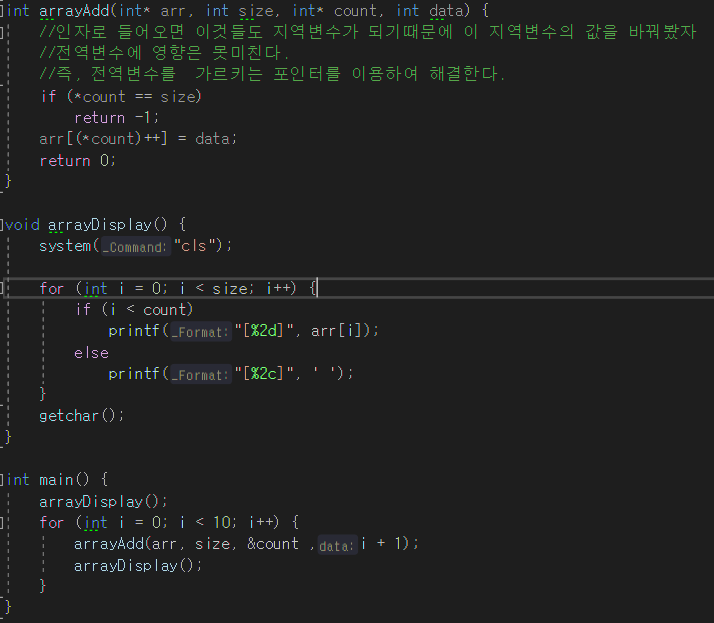


다른 라이브러리의 심볼과 충돌을 막기 위해 관례적으로 라이브러리 이름을 접두어로 사용한다.

여기에서 arrayAdd(int data)는 배열에 데이터를 넣을때사용하는 함수이다. 먼저 배열이 가득 차있는지 조사를 한후 (count == size면 가득 찬 상태) 그렇지 않으면 데이터를 넣어주며 이 과정에서 count는 개수를 늘려준다.

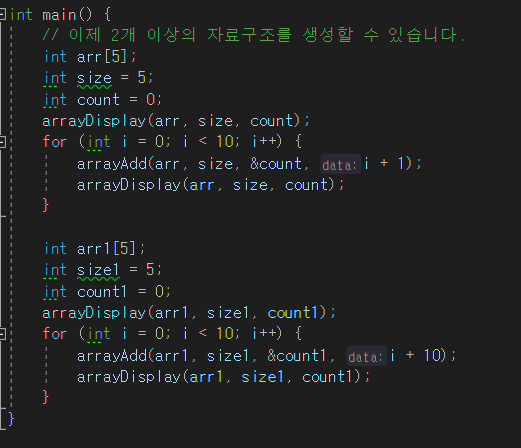
step2. 이전 자료구조는 추가가 잘 동작하나 2개 이상의 자료구조를 만들 수 없다는 문제가 있다.

이는 함수가 전역 심볼을 직접적으로 사용하고 있기 때문이다. 이제 이 문제를 해결하기 위해 전역 심볼을 지역 심볼로 변경하여 처리한다.

arrayAdd함수에서 arr를 즉 전역변수를 가리키는 포인터를 넣고 count또한 가리켜서 늘린다. size의 경우 값을 바꾸는 것이 아닌 고정된 값이므로 그냥 그대로 사용해도 괜찮다.

이렇게 가리키는 함수를 사용하게 되면 이제 전역변수로 또다른 배열을 선언하면 이 arrayAdd를 사용할 수있게 된다.

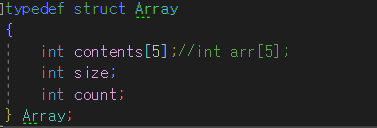
하지만 여기에도 단점이 있다. 그것은 arr, size, count는 항상 같이 세트로 있어야 의미가 있는 애들이기 때문이다. 따라서 전역변수보단 지역변수로 메인함수에서 같이 선언을 해서 만든다.

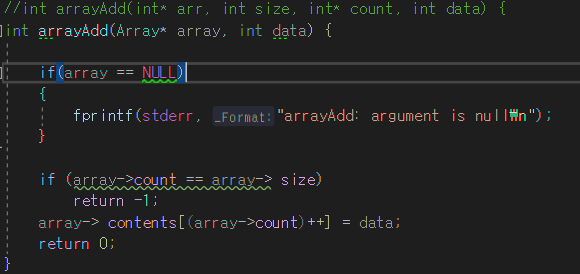
이렇게 하면 이제 메인함수 즉 사용자는 언제든 배열이라는 자료구조를 임의로 추가할 수 있게 되었다.

step3. 이전 자료구조는 데이터들이 분리되어 있어 사용하기 어렵다는 단점이 있다.

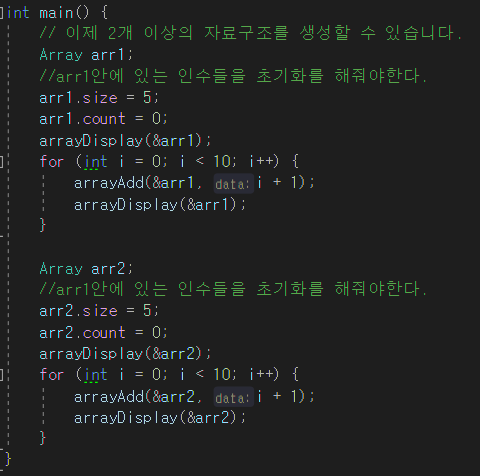
arr, size, count는 항상 같이 있어야 의미가 있다.

이를 해결하기 위해 데이터를 묶어 새로운 타입으로 정의(추상화)한다.

이렇게 의미있는 애들끼리 같이있어야 하므로 아예 구조체로 선언해서 만들어버린다. 이렇게 되면 뗄수없는 사이가 되었다.

위와 같이 구조체로 Array를 선언하고 정의를 하면 다른 함수에서도 넣어야할 인자들이 간단해진다.

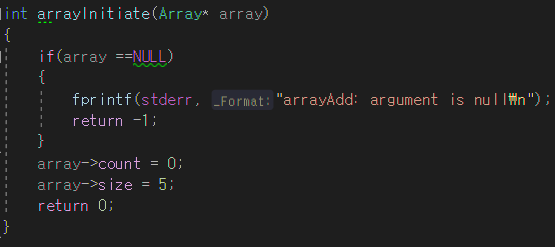
arr, count, size를 다 넣는 것이 아닌 Array 하나만 넣으면 되는것이다.

대신 Array를 만들 때 안에 있는 인수들을 초기화 해줘야 한다.

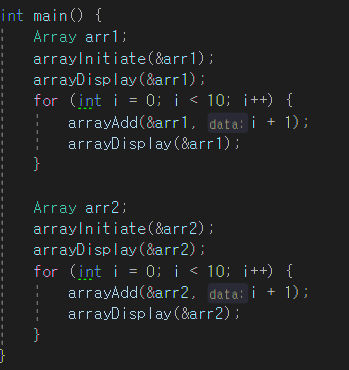
step4. 이전 자료구조는 자료구조 생성 시 ,내부 데이터가 초기화되지 않는다는 단점이 있다.

이를 해결하기 위해 자료구조를 초기화하는 함수를 추가한다.

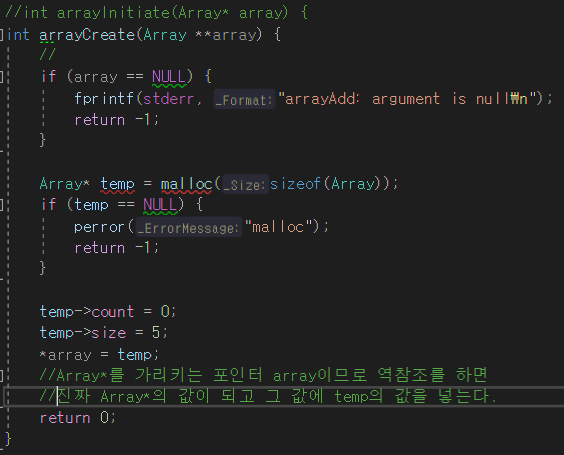
무조건 이 Array라는 구조체를 사용할때는 각각의 구조체의 인수에 값을 넣어줘서 완성을 해야하기 때문에 arrayInitiate()라는 함수를 만든다.



그냥 정말 단순하게 Array가 있는지 체크하고 있다면 값을 초기화하는 작업밖에 안한다.

arr1이라는 자료구조를 생성하고 바로 arrayInitiate()함수를 사용해서 초기화 해준다.

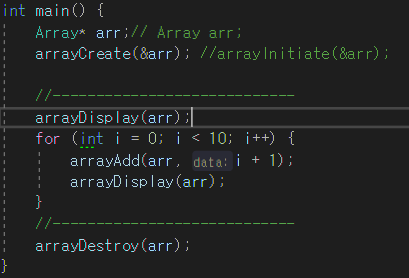
step5. 이전 자료구조는 자료구조의 컨텍스트를 저장하기 위해 스택 메모리를 사용하고 있다는 단점이 있다. 이를 해결하기 위해 정보 구조체를 힙에 생성한다.

기존의 arrayInitiate()는 Array자체를 스택에 만드므로 스택에 메모리가 쌓이게 되고 이건 좋지 않다. 따라서 힙에 동적할등을 통해 만들어서 사용하고 다 사용하면 해제하는 방식을 택한다.

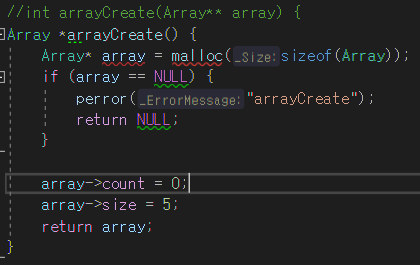
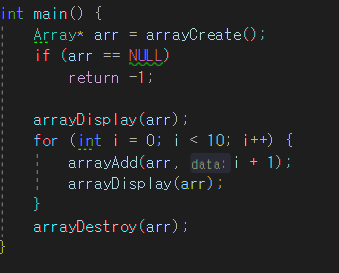
ArrayCreate()라는 함수에서는 Array\*\* array를 인수로 받는데 그 이유는 메인 함수에서 array가 아닌 array\*의 주소를 인수로 넣기 때문에 그 주소를 가리키는 포인터로 받는것이다. 따라서 array\* 의 포인터이므로 array\*\* 가 된다. 그러면 이제 create 에서는 array\* 인 temp를 만들고 동적할당으로 Array만큼의 사이즈를 만든다. 그아래의 식은 동적할당에 실패했을때이고 이제 성공을 했다면 temp가 가리키는 count 와 size를 초기화한수 인수로 받은 Array\*를 가리키는 포인터 array 를 역참조를 통해 array\*의 값에 temp를 넣는다. 이렇게 하면 임의의 포인터 temp로 동적할당을 한후 그가리키는 값을 온전히 인수에다가 넣어준다. 포인터이므로 주소만 알면 언제든 접근가능하므로 이 방식이 사용되었다.

주소로 받는 이유는 Array\* arr로 할당을 할것이기 때문인데 그냥 arr만 넘기게 되면 arrayCreate함수에서는 힙에 할당을 할순있겠지만 그 함수내에서 한것이므로 돌아왔을 때 메인의 arr는 여전히 쓰레기값을 가지고 있을 것이다. 따라서 arr의 주소를 보내게 되면 create함수에서는 temp로 할당을 한 후 메인의 arr를 가리키는 array의 포인터값에 temp를 넣는다.

따라서 메인에서는 이런식으로 보내는것이다.

step6. 이전 자료구조는 함수로 전달되는 배열에 대하여 그 방식이 일반화되지 않았다.

이는 자료구조의 사용을 어렵게 한다는 단점이 된다.



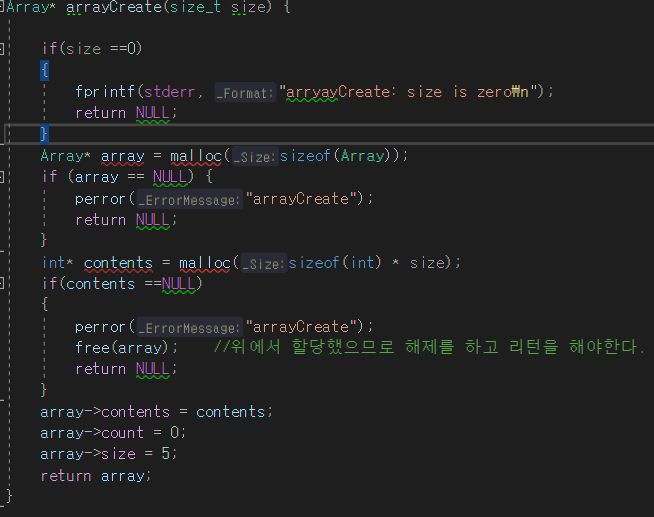
이전에는 메인에서 Array를 만들고 create함수를 통해서 힙에 할당하며 초기화했다면 이제는 Array\*를 선언하며 create함수를 바로 사용하고 힙에 할당된 배열포인터를 리턴받는다.

원래는 리턴을 지역변수로 주면 안좋은코드이나 이것은 힙에 할당한 주소를 넘겨주는것이므로 아무 문제가 없다.

step 7. 이전 자료구조는 배열의 크기가 고정되어 있다는 단점이 있다. 따라서 우리가 원하는 크기로 자유자재로 만들수있도록 한다.

create() 에서 크기에 대한 인수를 사용자로부터 받게 되면 그 크기를 size에 넣어서 동적할당한다.

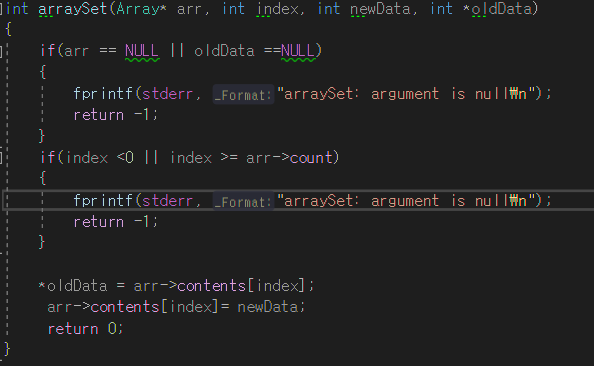
그 전에는 contents자체를 배열로 선언을 했었는데 이것도 포인터로 만들어버린다.그래야 이 포인터로 동적할당을 해서 원하는 크기의 배열을 만들수있기 때문이다. 이렇게 되면 Array\*로 동적할당해서 array를 만들지만 그 array안에서 또 포인터가 있어서 그 포인터로 동적할당을 해서 배열을 만들고 그것을 가리킨다.

여기에서 진짜 소심하지만 세세하게 신경쓸것이 있는데 처음에 array에 대한 동적할당이 성공했으나 contents에 대한 동적할당이 실패했다면 그 위에서 성공했었던 array를 해제해주고 실패했다고 리턴을 해야한다. 따라서 free(array)가 있는 것이다.

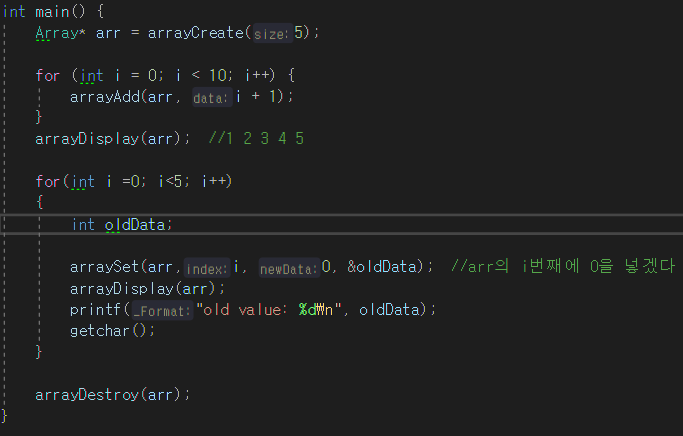
step8. 데이터가 존재하는 위치에 데이터를 설정하는 함수를 구현해 보자.

여기에서 주의할 점은 반드시 기존 데이터는 임의로 삭제하면 안된다는 것이다.

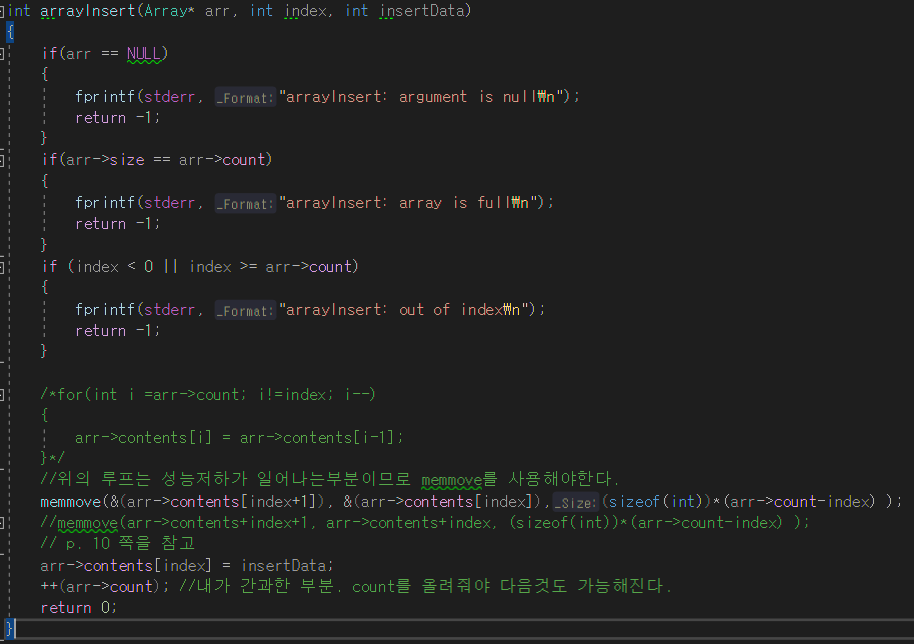
따라서 기존의 데이터는 사용자가 처리하도록 해줘야한다.

옆의 코드를 보면 oldData를 가리키는 포인터를 받았고 역참조를 통해 contents에서 기존에 있던 데이터를 oldData에다가 넣는다. 이렇게 해서 사용자가 처리할 수 있도록 했다. 그렇게 하고 나서 이제 새로운 데이터를 넣는다.

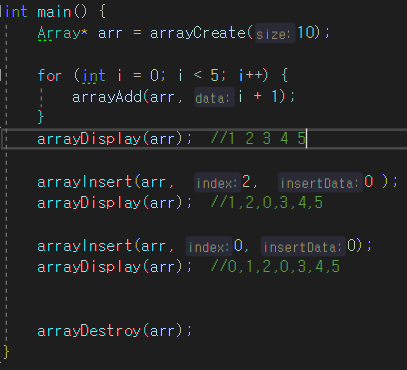
index를 통해 배열의 몇번째에다가 새로운데이터를 넣고싶은지를 정한다.



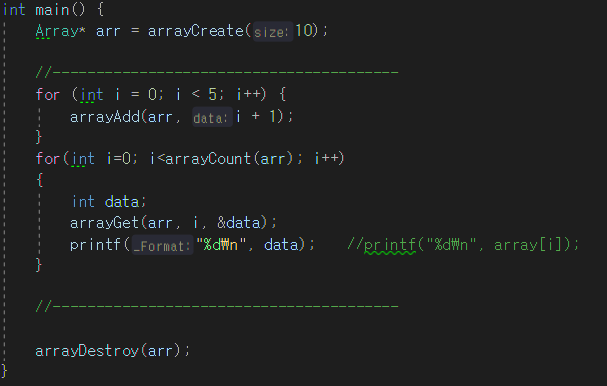
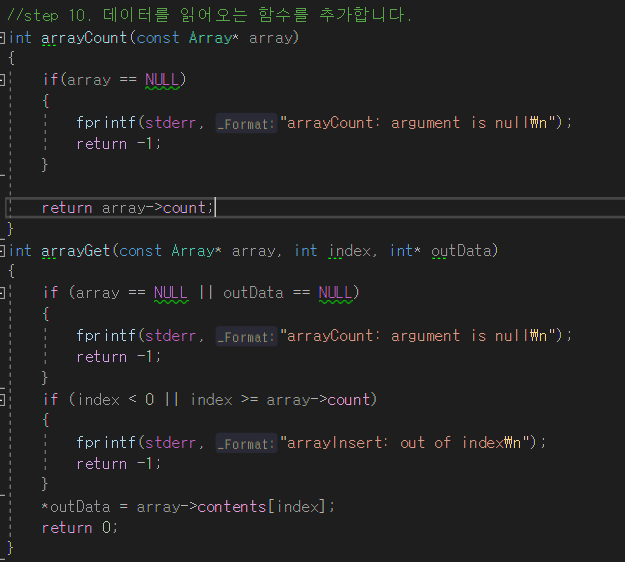
step 9. 원래 있던 배열에다가 데이터를 넣는 것은 add함수가 있다. 이제는 배열의 원하는 index에 데이터를 추가해보자.

이 경우는 데이터가 이미 꽉차있어도 안되고 원하는 index가 0보다 작거나 count즉 개수보다 크면 안된다. 사실 이경우 index가 있는곳까지 for문을 돌려서 갈수는 있으나 이렇게 되면 루프를 돌게되고 성능저하가 일어난다. 따라서 이미 있는 라이브러리의 메소드중에 하나인 memmove를 사용하기로 하자. 여기에서 memcpy를 사용하지 않는 이유는 memcopy는 별도의 버퍼를 사용하기 않기 때문에 두개의 포인터가 하나의 대상을 가리키게 되면 overlab이 될수있기 때문이다. 따라서 memmove를 사용한다.

memmove는 첫인자가 dst 이고 두번째가 src, 그리고 세번째는 이동할 크기 이다. 즉 src부터 dst까지 원하는 크기만큼을 통째로 이동시킨다.이렇게 해서 배열의 원하는 index를 비우게하고 다 한칸씩 밀어내고 그 index에 새로운 값을 넣는것이다. 물론 넣고 나서 배열에 값이 추가되었으므로 count도 올려줘야한다.

이렇게 원하는 값을 사이에 낄수있게 되었다.

step 10. 데이터를 읽어오는 함수를 추가한다.

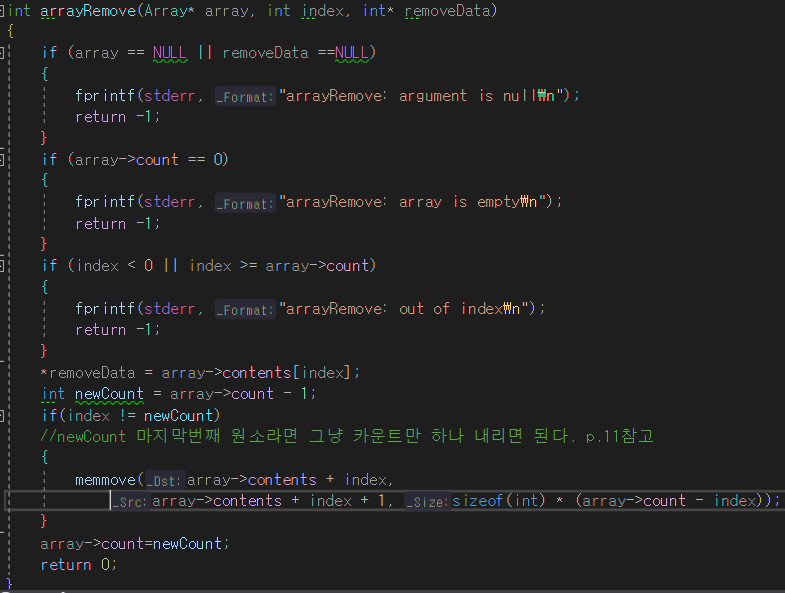
 count함수는 밑의 메인에서 모든 값들을 읽어오기위해 사용한것이다.

이 함수는 딱히 설명할것들이 없다.

step 11. 임의의 위치에 있는 데이터를 삭제하는 함수를 구현해 보자.

이때, 자료구조에서의 삭제는물리적으로 지우는 것이 아니라 자료구조 내에 그 데이터를 더 이상 유지하지 않는다는 개념이다. 삭제된데이터는사용자에게 전달하여 사용자가 처리하도록 해야한다.

제거 하고 나서 생기는 빈부분은 메꿔져야 한다.

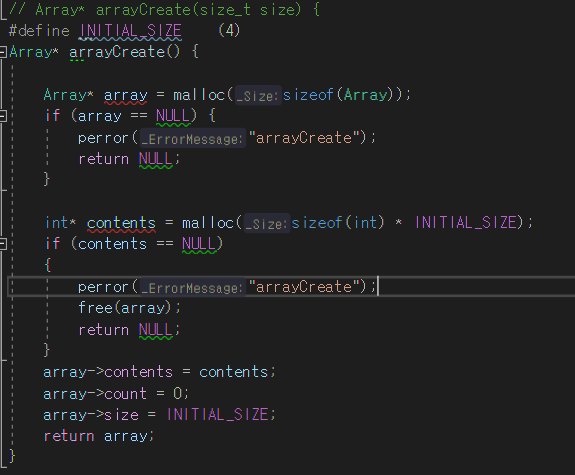


일단 removeData에 지금 배열이 가지고 있는 contents의 데이터를 넣는다.

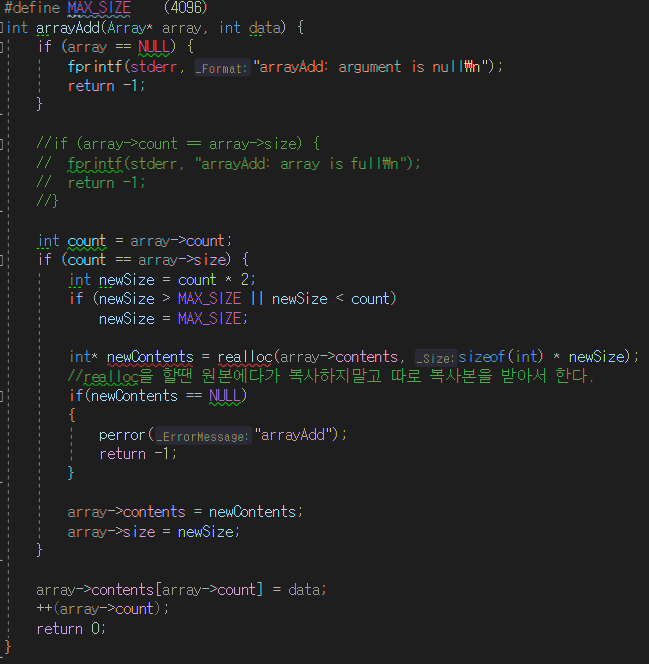
그러고 나서 하나의 데이터가 제거된다면 count값도 1이 줄어야한다. newCount = countn-1을 해준다. 그리고 만약 newCount가 마지막번째 라면 그냥 카운트만 하나 내리면 끝나고 그게 아니라면 memmove를 통해 없앤부분을 기준으로 하나씩 땡기면 된다.

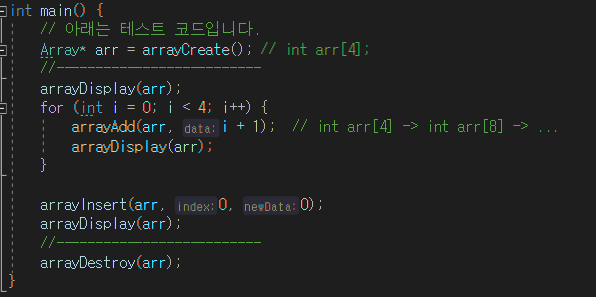
따라서 이번엔 아까 insert와는 반대로 dst가 src보다 한자리가 더 적게 설정한다. 사이즈는 총원소개수에서 index만큼을 빼준것의 int값을 곱한만큼을 이동한다. 즉 index부터 마지막까지를 하나씩 땡기는것이다.

step 12. 배열의 크기가 자동으로 증가될 경우, 배열의 크기를 사용자로부터 입력받을 필요가 없다. 따라서 기존의 크기를 INITIAL\_SIZE라고 하고 4를 받고 그 이상으로 데이터를 넣으려 한다면 크기를 자동으로 올린다.

 따라서 사용자로부터 크기를 받는 것이 아니게 설정한다.

step 13. 위의 자료구조는 배열의 크기가 고정되어 있다는단점이 있다. 이를 해결해보자

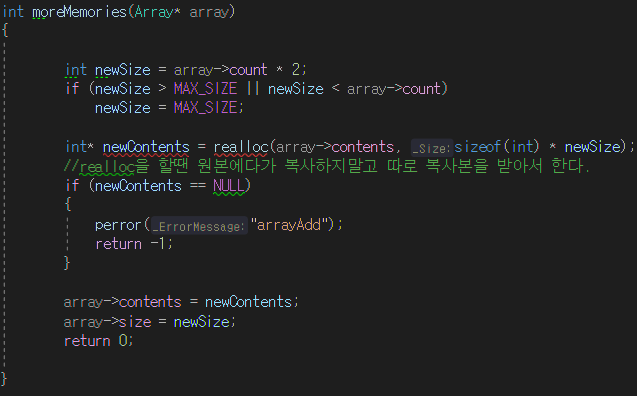
일단 크기가 무한정 늘수는없으므로 최대사이즈를 설정해놓고 count == size가 될 때 새로운사이즈를 두배로 올린다. array 자체를 새로 바꿀필욘없고 원래있던 contents만 바꾸면 되기 때문에 realloc을 통해 newContents를 만든다. realloc은 원본을 아예바꿔버리므로 복사본을 받아서 하는것이 안전하다. realloc을 통해 기존의 데이터와 새로운 사이트로 새롭게 할당해서 만든다. 여기에서 포인터로realloc을 통해 크기가 커진 contents를 가리키므로 contents가 없어지는 것이 아니다. 애초에 contents 도 포인터였으니 말이다. 다만 두 포인터의 차이는 가리키는 곳부터 얼마만큼의 사이즈를 읽을것인지의 차이이다. 이와 같이 insert함수에도 새롭게 크기를 만들어서 가리키는 과정을 한다.

테스트코드이며 넘을때마다 2배씩 커진다.

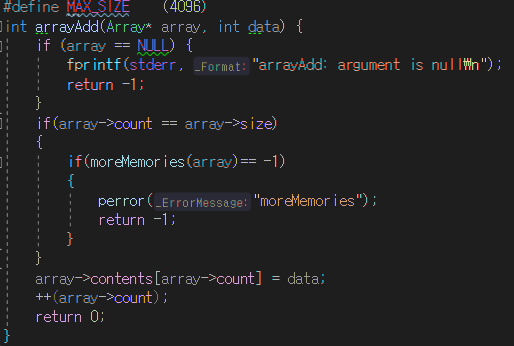
step 14. 이전 자료구조는 자원 할당 정책이 중복되어있다는 단점이 있다.( insert 와 add함수처럼)

이는 유지보수를어렵게 하고 버그가 발생할 확률이 높아진다는 문제가 있다.

이를 해결하기 위해 자원 할당 정책을 한 곳으로 모은다.

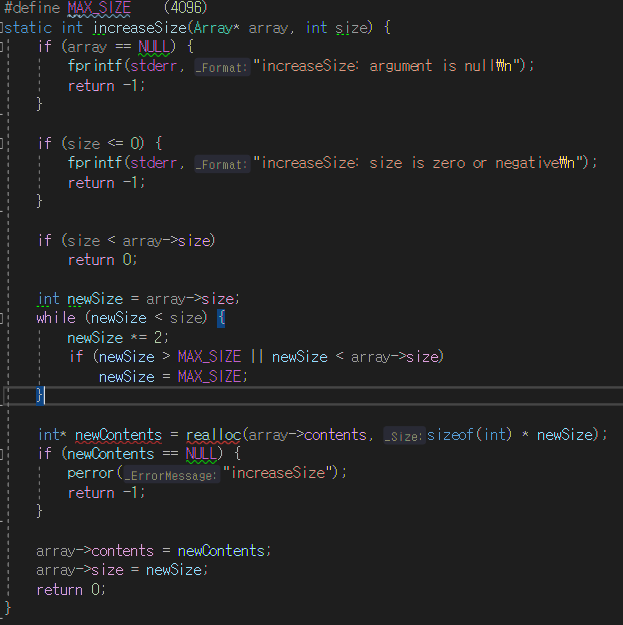


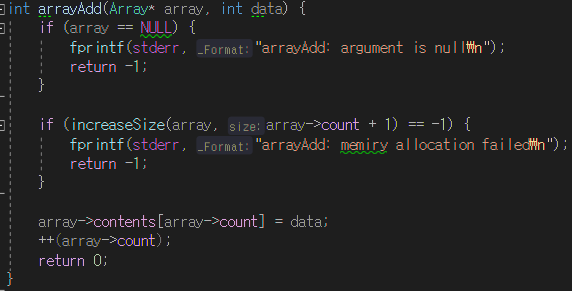
위에서 넣었던 코드와 정확하게 같지만 반복되는경우가 발생하기에 이렇게 따로 함수로 뺐다. 이러면 add 나 insert함수도 간략해지고 해석이 수월해진다.



엄청 짧아진 것을 알 수 있다.

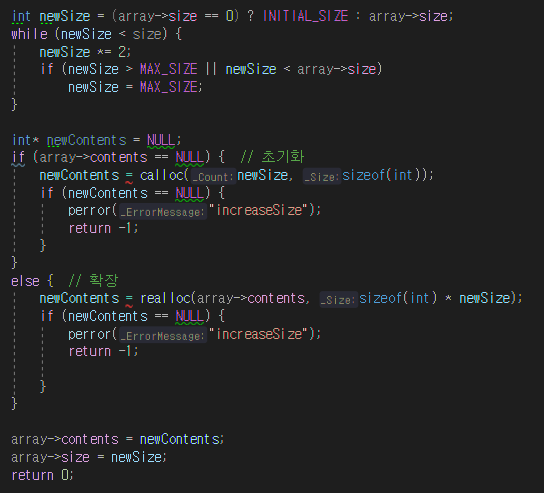
moreMemories 함수를 increasSize함수로 바꿔본다. increase함수에서는 임의의 크기도 같이 받아서 size만큼의 크기 이상으로는 안올라가도록한다.





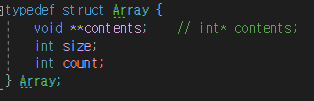
step 15. 이전의 자료구조는 언제 사용될지 모르는 배열에 대하여 미리 할당한다는 단점이 있다. 이는 자칫 메모리 낭비가 될 수 있다. 이를 해결하기 위해 지연된 초기화 (lazy initialization)을 도입한다.

increase함수에서 위의 검사하는 부분을 빼고 보면

array의 사이즈를 봤을 때 0이라면 초기의 사이즈인 INITIAL\_SIZE로 해주고 아니라면 array의 size를 newSize로 한다. newContents를 선언하여 만약 array->contents가 NULL이라면 즉 아직 아무 값도 추가를 안해서 데이터가 없을시에는 동적할당으로 초기화를 해주고 그게 아니라면 realloc을 통해 확장을 한다. 이렇게 하면 확인을 하고 확장을 하므로 사용하지도 않은 것을 미리 할당해놓지 않고, 지연된 초기화가 일어난다.

step 16. 이전 자료구조는 정수만 저장 가능하다는 단점이 있다.

이제 모든 타입에 대하여 저장 가능한 자료구조로 바꿔보자.

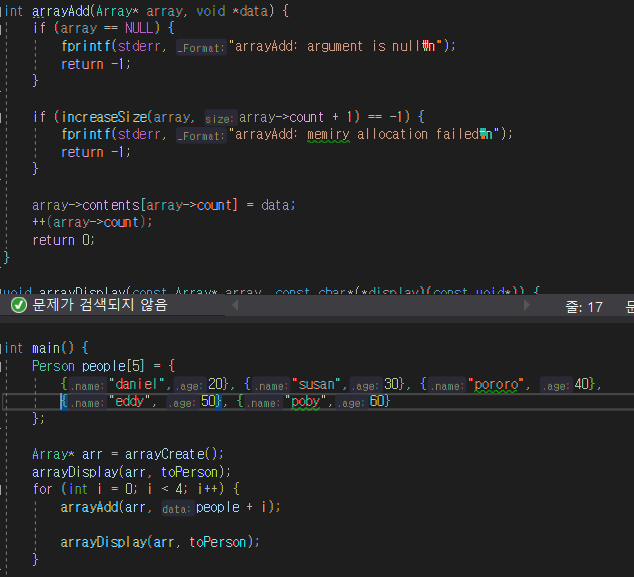


우리는 계속 int\* contents로 사용을 했지만 이젠 어떤값도 받을 수 있게 하기 위해서 void \*\* 형으로 만들었다.

void\* 가 아닌 void\*\*로 만든 이유:

void\*였으면 우리가 원래하던 int\*같이 그냥 void 형으로 Person 타입을 받으면 되는 것 아닌가?

답: void 는 매개변수로 사용될수없기때문이다. void 는 타입으로 치지 않기 때문에 모든 형의 타입을 받기 위해서 void\*를 이용하는것이다. 예를 들어 int Arrayadd() 이렇게 인수가 없으면 사실상 Arrayadd(void)와 같은 것이므로 void는 타입이 아닌것이다. 따라서 이렇게 void형으로 받을순 없으니 void\*로 모든 타입들의 포인터를 받는것이다.

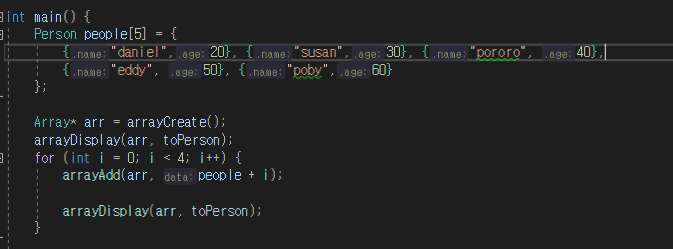


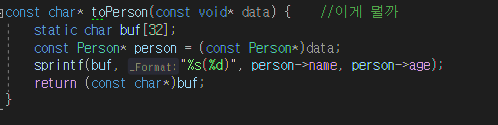
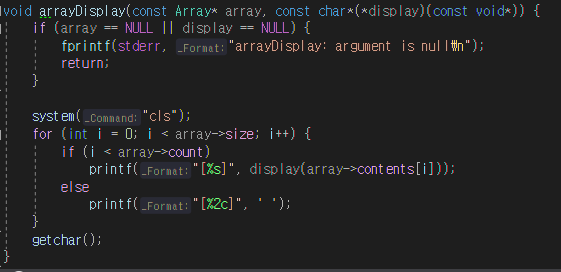
이렇게 해서 모든타입을 받을수 있게 만들면



위와 같이 int로 리턴을 받던것도 oldData가 필요가 없게 되고 리턴값으로 void\*를 준다.

사용자가 대신 display함수를 사용할 때 어느정도 크기인지를 알아야 그에 맞게 출력을해줄수있으므로 toString()이라는 함수로 만들어서 사용해야 한다.



위의 코드를 보면 메인함수에서 arrayDisplay함수에 인자로 toPerson함수 자체를 넣어준다.

이런 코드는 좀 생소했는데 함수자체를 넘길때는 ( )없이 그냥 변수이름만 넘기며 되는것이다.

어떻게 이렇게 해도 되는지: 함수를 넘기는 것이 아니라 함수의 주소를 넘기는것이고 따라서 display에서는 포인터로 그 함수의 주소를 가리키는 것이다. 따라서 display포인터에다가( )를 붙임으로서 함수가 완성된다. display(array-> contents[i])를 보면 array가 가리키는 contents의 i번째 값은 void\*가 되고 toPerson에서는 void\*를 data라는 변수로 받는다. 함수를 벗어나도 사라지지 않는 static형태로 buf[32]를 버퍼로 사용하기위해 만들어놓고 data를 Person\*로 형변환시켜놓은후 sprintf를 사용해서 buf에 person이 가지고 있는 name과 age를 넣는다. 그후 buf배열의 주소를 char\*형으로 리턴한다. printf에서는 주소를 보고 가서 읽음으로써 완벽하게 출력이 된다.

이제 모든게 거의 완벽히 완성되었다. 실무코드로 써도 괜찮을정도가 되었다.

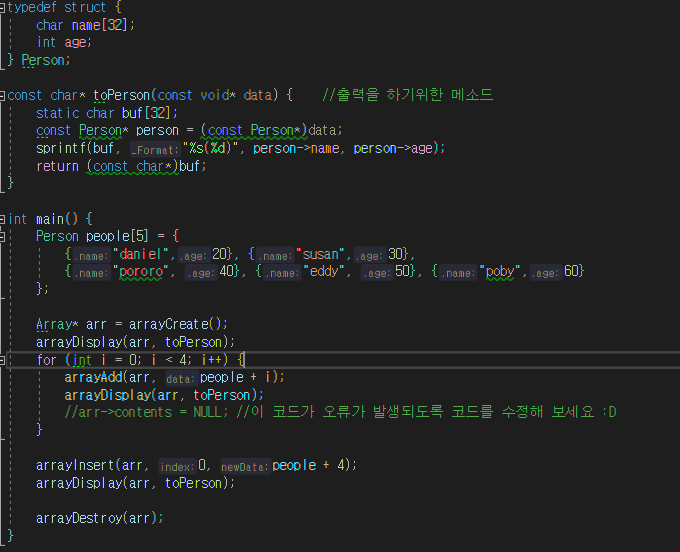
하지만 딱 한가지 문제가 있는데

last step. 현재 자료구조는 내부 구조가 은닉되어 있지 않아 코드가 올바르게 동작하지 않을 수 있다는 문제가 있다. 이를 해결하기 위해 구조체 멤버을 은폐한다.

헤더파일에다가 함수들의 선언만 넣어서 사용자와 공유를 할 수 있게 한다.

사용자가 몰라도 되는 내용은 libarray.c에다가 넣어서 정의부분을 은닉한다.

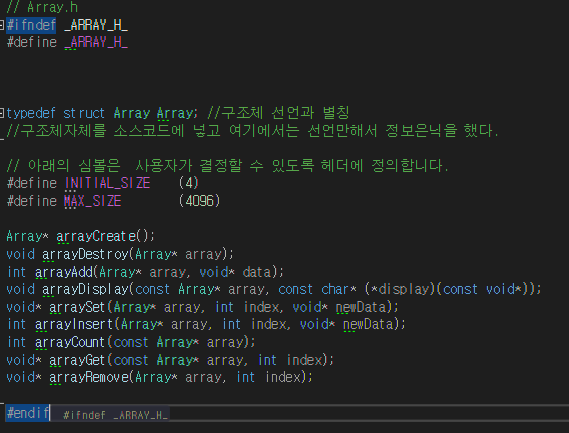
모듈화는 헤더파일과 함수구현파일(libarra.c)와 사용자파일 3.c로 만든다.



사용자가 함수의 의미를 임의로 바꿀수 없게 해야하고 그렇기 때문에 다른곳에 정의를 하고 헤더를이용해 공유한다. 사용자는 위의 코드정도만을 사용하면 된다.

나머지 거의 정의부분은 libarray.c에 들어가게 된다.

그리고 마지막 헤더파일에는



위와 같이함수 대부분의 선언과 공통되는 매크로 정도가 여기에 들어간다.

Array도 은닉을 했으므로 사용자는 array에 대한 내부는 알지 못 한다.

이로써 배열에 대한 것은 끝난다.

배열의 장점:

구현이쉽다.

임의접근이 가능하여 해당원소 접근이 빠르다.

배열의 단점:

메모리 낭비 발생 가능

삽입, 삭제에 대한 오버헤드가 크다.

따라서 이단점을 커버하기 위해 연결리스트를 사용한다.