동적 메모리

2021년 2월 11일 목요일 오전 8:05

- 프로그램이 동적 메모리를 가져다 사용할 때는 총 세 가지 단계를 거침
  1. 메모리 할당
  2. 메모리 사용
  3. 메모리 해제
- ## 동적 메모리 사용 단계 1: 메모리 할당
  - 힘 관리자에게 메모리를 xxx 바이트만큼 달라고 요청
    - 관리자는 연속되는 그만큼의 메모리를 찾아서 반환
    - 반환된 메모리는 어떤 자료형에 저장 가능? 메모리 주소니 당연히 포인터!

- ## 동적 메모리 사용 단계 2: 메모리 사용
  - 그 메모리를 원하는대로 사용
    - 포인터 변수 사용하는 거와 똑같이 사용하면 된다
    - 예: int 배열에 성적을 저장한 뒤 평균을 구해서 float 변수에 딱 저장

- ## 동적 메모리 사용 단계 3: 메모리 반납/해체
  - 힘 관리자에게 그 메모리 주소를 돌려주면서 다 썼다고 알려줌

  - 관리자는 그 메모리 주소를 아무도 사용하지 않는 상태로 바꿈

- 그렇다면 c에서 이런거 하는 함수는 뭘까?

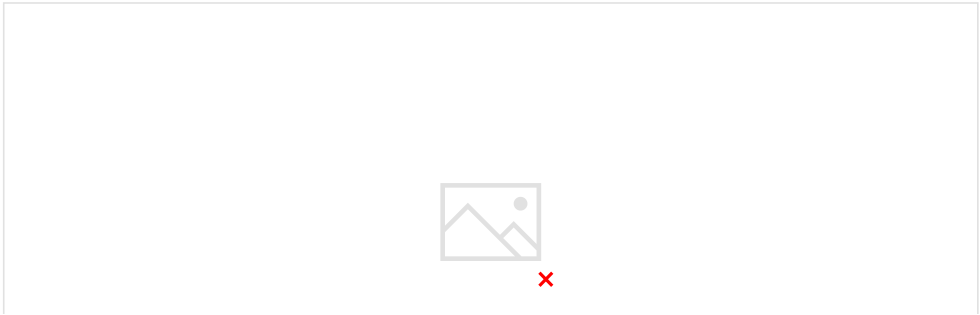
메모리 할당 및 해제 함수

malloc()

- void* malloc(size_t size);
- 메모리 할당(memory allocation)의 약자
- size 바이트 만큼의 메모리를 반환해줌
- void*형으로 반환하는 이유
  - 범용적이고 프로그래머가 알아서 캐스팅해서 사용할 거니까
- 포인터 외에 다른 자료형으로 반환될 수가 없음
  - 메모리 주소를 반환하니까..
- 반환된 메모리에 들어있는 값은 쓰레기 값
  - 즉, 초기화 안 해줌
- 메모리가 더 이상 없다거나 해서 실패하면 NULL반환
- malloc( )으로 할당받은 메모리를 사용하는 방법을 보기 전에 짝꿍 함수부터 배우자

malloc()의 짝꿍 함수 free()

- 동적 메모리는 프로그래머가 직접 빌리고 반납해야 한다고 했죠?
- 그럼 빌렸으면 뭘 해야 하지?
  - 메모리를 반납해야 함
  - 그것을 메모리에 건 속박을 풀어준다고 해서 free()
- 안 지우면 어떻게 된다고 했죠?
  - 메모리 누수가 발생
- **고로 malloc() 코드를 작성하면 곧바로 free() 코드도 추가하는 습관을 들이는 게 좋음**
- malloc()을 쓰는 예 (초간단 버전)



- 여러 줄의 입력을 받아 출력하기 예

  - free(lines) 이거는 왜 안풀어줄까?
    - lines는 malloc으로 만든게 아니라 스택 메모리에 들어간 것임
  - 위 예의 한계
    - 줄 수가 NUM_LINES로 고정

- 학생 정보 입력 받기 예

  - scanf에서 NAME_LEN보다 긴 문자열을 입력하면 문제가 생긴다

- 위 코드에서 while반복문 안에 생략된 코드가 많음
- current_index, max_students는 0으로 시작
- 탈출조건을 추가해줘야 함
    - 예: 학생 수가 5000명이상이거나, 탈출 코드로 exit을 입력하거나

- 파일에서 학생 정보 읽기 예

- free(students)를 해주자!

- 제대로 된 free() 설명
    - void free(void* ptr)
    - 할당받은 메모리를 해제하는 함수
    - 즉, 메모리 할당 함수들을 통해서 얻은 메모리만 해제 가능
    - 그 외의 주소를 매개변수로 전달할 경우 결과가 정의되지 않음

233. 동적 메모리 할당 시 문제

- 할당받아 온 주소를 그대로 연산에 사용하면
    - 메모리 할당 함수가 반환한 주소가 저장된 변수를 그대로 포인터 연산에 사용하면 메모리 해제할 때 문제가 발생할 수도 있음
    - 최초에 받아온 주소가 아니라 다른 위치를 가리킴 -> 그 주소로 메모리 해제 요청 -> 결과가 정의되지 않음 -> 망함!!

- 코딩 표준: 할당받은 포인터로 연산 금지
  - 메모리 할당 함수에서 받아온 포인터와 포인터 연산에 사용하는 포인터를 분리하자

- 해제한 메모리를 또 해제해도 문제
  - 해제한 메모리를 또 해제하려고 할 때도 결과가 정의되지 않음
    - 잘못되면 크래시가 날 수도

- 해제한 메모리를 사용해도 문제
  - 해제한 메모리를 또 사용하려 하면 역시 결과가 정의되지 않음
    - 역시 잘못하면 크래시가 날 수도
    - 소유하지 않은 메모리에 접근하는 것을 memory stomp라고 함
      - 남의 메모리를 꽝꽝 밟는 느낌

  - 이러한 문제를 줄일 수 있는 여러 방법(습관)이 있음
- 코딩 표준: 해제 후 널 포인터를 대입
  - free()한 뒤에 변수에 NULL을 대입해서 초기화
    - 안 그러면 해제된 놈인지 나중에 모르니
    - 널 포인터를 free()의 매개변수로 전달해도 안전

- 왜 free()설명을 두 번에 나눠했나요?
  - 메모리 누수 때문
  - malloc() 한 뒤 **free() 까먹으면 메모리 누수**
  - malloc() 으로 받아온 주소를 지역 변수에서 저장해놨는데 **해제 안 하고 함수에서 나가면** 주소가 사라져서 **지울 방법이 아예 없어짐**
  - 그래서 습관을 잘 들여놔야 함
- 베스트 프랙티스: malloc()과 free()는 한 몸
  - malloc()을 코드에 추가하자마자 다른 일 하기 전에 곧바로 free()도 추가 할 것
  - 누가 질문해도 "잠깐만요! 1분만요!"를 외치고 free()를 추가

234. free()는 몇바이트를 해제할지 어떻게 알지? calloc(), memset(), realloc()

- free()는 몇 바이트를 해제할지 어떻게 알까?

  넘겨주는 건 주소뿐인데..

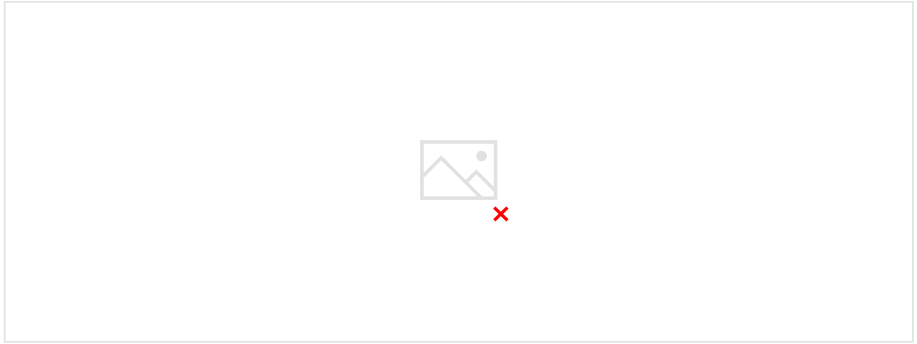
  1. 구현마다 다르지만 보통 malloc(32)하면 그것보다 조금 큰 메모리를 할당한 뒤, 제일 앞부분에 어떤 데이터들 채워 놓음

     
  2. 그리고 그만큼 오프셋을 더한 값을 주소로 돌려줌
     - 우린 돌려받은 주소로부터 원래 요청했던 32바이트를 사용

     
  3. 나중에 그 주소 해제를 요청하면 free()가 다시 오프셋만큼 빼서 그 앞 주소를 본 뒤, 실제 몇 바이트가 할당됐었는지 확인 후 해제

     ○ 실제로 한번 보면

     
     - 앞에 메모리의 meta data가 있고 뒤에 실제 데이터가 있음
       - □ 앞에 메모리가 4바이트의 크기를 갖는다는 meta data가 있고 뒤에 밑에줄에 실제 데이터가 있음

- (책) calloc()
  - void* calloc(size_t num, size_t size);
  - 의미는 아무도 모름 의견이 분분함
    - counted alloc
    - clear alloc
    - C alloc
  - 메모리를 할당할 때 자료형의 크기(Size)와 수(Num)를 따로 지정
  - 모든 바이트를 0으로 초기화 해 줌
  - 잘 안 씀
- calloc() = malloc() + memset()
  - 보통 calloc() 대신 malloc()와 memset()을 조합해서 씀
    - memset()을 쓰면 0외의 값으로도 초기화 가능
  - 두 방식은 거의 비슷하다!

- memset()
  - void* memset(void* dest, int ch, size_t count);
  - <string.h>에 있음
  - char로 초기화(1바이트씩)됨
  - 그 외의 자료형으로 초기화하려면 직접 for 문을 써야함
  - 다음과 같은 경우 결과가 정의되지 않음
    - count가 dest의 영역을 넘어설 경우 (소유하지 않은 메모리에 쓰기)
    - dest가 널 포인터일 경우 (널 포인터 역참조)
  - char로만 초기화해주는 memset() 예시

  - int로 초기화하고 싶다면?

- realloc()
  - void* realloc(void* ptr, size_t new_size);
  - 이미 존재하는 메모리(ptr)의 크기를 new_size 바이트로 변경

- 새로운 크기가 허용하는 한 기존 데이터를 그대로 유지
- 아까 봤던 학생 정보 입력 받기 예에서 동적 배열 크기를 늘리던 코드를 이 함수로 대체 가능
- 반환값은 중요하니 조금 뒤에 따로 설명

- 크기가 커져야할 때, 두 가지 경우가 있음

1. 지금 갖고 있는 메모리 뒤에 충분한 공간이 없으면 새로운 메모리를 할당한 뒤
   기존 내용을 복사하고 새 주소 반환

2. 지금 갖고 있는 메모리 뒤에 공간이 충분하다면 그냥 기존 주소를 반환(보장은 없음)
   그리고 추가된 공간을 쓸 수 있게 됨

- 크기가 작아질 때도 비슷함
  - 기존 주소가 그대로 반환되거나
  - 다른 곳에 메모리를 새로 할당 후, 새 주소를 반환해 줄 수도

235. realloc()의 메모리 누수 문제, memcpy()

- realloc()에서 메모리 누수가 날 수 있다
  - void* realloc(void* ptr, size_t new_size);
  - 반환값
    - 성공 시, 새롭게 할당된 메모리의 시작 주소를 반환하며 기존 메모리는 해제됨
    - 실패 시, NULL을 반환하지만 기존 메모리는 해제되지 않음
  - 실패 시 메모리 누수가 발생할 수 있음!!!!

- 메모리 누수가 나는 경우

  - 원래 nums에 저장되어있던 주소가 사라짐
  - NULL이 반환됐다는 이야기는 재할당에 실패했다는 의미
  - 따라서 기존 메모리는 해제되지 않았음
  - 그러나 그 주소를 잃어버려서 해제할 수 없다!! 메모리 누수 발생!
- 올바른 재할당 방법

- realloc() = malloc() + memcpy() + free()
  - realloc()은 malloc() + memcpy() + free()과 유사함

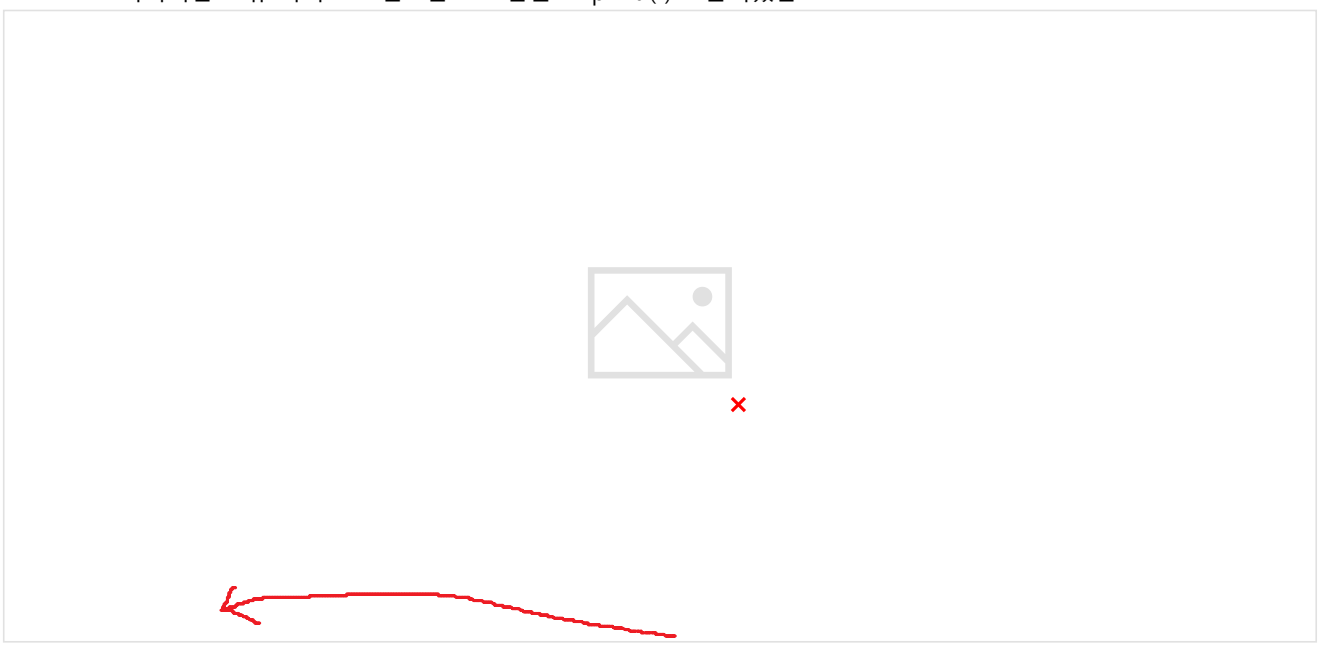
- memcpy()
  - void* memcpy(void* dest, const void* src, size_t count);
  - <string.h>에 있음
  - src의 데이터를 count 바이트 만큼 dest에 복사
  - 다음과 같은 경우 결과가 정의되지 않음
    - dest의 영역 뒤에 데이터를 복사할 경우 (소유하지 않은 메모리에 쓰기)
    - src나 dest가 널 포인터일 경우 (널 포인터 역참조)
- 메모리 누수 안 나게 코드를 작성할 것!
  - realloc()을 쓸 때는 정말 정말 조심해야 함
  - 그래서 차라리 malloc() + memcpy() + free()로 좀 더 명시적으로 드러나게 코딩하는게 나을지도
  - 그냥 신경 안 쓰고 realloc()을 쓰는 경우도 많음
    - 메모리 시작 주소가 변하지 않는 경우 데이터 복사를 하지 않아 성능상 이득
    - 그리고 메모리가 없어서 널 포인터를 반환하는 상황에 어떻게 대처할 건데?
      - 그냥 실패할 때 메모리 누수로 크래쉬를 내는게 맞다고 생각하고 코딩
      - 상황에 따라 성능상 이득을 위해 크래쉬를 감안하여 realloc()을 쓸 수 있는 거다
    - malloc()에서 실패하는 일이 없다고 가정하고 코딩을 하는 경우가 많은 이유도 마찬가지
- (책) realloc()의 특수한 경우
  - nums = realloc(NULL, LENGTH);

        - 새로운 메모리 할당
        - malloc(LENGTH)와 동일함

236. memcmp(), 정적 vs 동적 메모리

- 여러 줄을 입력받아 출력하는 코드도 이렇게 개선 가능

    - realloc() 함수는 오류가 났을 시 errornum을 설정해주지 않는다
        - realloc()문서를 읽어보면 알 수 있음
        - 그래서 perror()해봐야 오류문이 출력이 안 된다
        - 여기서는 오류 처리 코드를 if문으로 만들고 fprintf( )로 출력했음



    - 포인터를 담는 배열인 lines도 동적 할당을 하기 때문에 free 해줌
- memcmp()
    - int memcmp(const void* lhs, const void* rhs, size_t count);
    - 첫 count 바이트 만큼의 메모리를 비교하는 함수
    - strcmp()와 매우 비슷
    - 단, 널 문자를 만나도 계속 진행
    - 다음의 경우 결과가 정의되지 않음
        - lhs과 rhs의 크기를 넘어서서 비교할 경우 (소유하지 않은 메모리에 쓰기)
        - lhs이나 rhs이 널 포인터일 경우 (널 포인터 역참조)
- 구조체 두 개를 비교할 때도 유용

    - if 문 3개로 각 멤버를 비교해도 되고 이렇게 메모리 통째로 비교해도 된다
    - 문자열 비교 시 문자열 뒤 쓰레기 값 때문에 다른 결과가 나올 수 있다
        - 첫 번째 Lulu 뒤에 쓰레기 값과 두 번째 Lulu 쓰레기 값이 달라서 memcmp시 같지 않다고 나올 수 있다
            - firstname 배열의 크기는 64인데 Lulu 뒤에 59 바이트의 공간이 있다
        - 이를 예방하려면 문자열 뒤 쓰레기 값을 널 문자로 채워줘야 함
- 단, 구조체가 포인터 변수를 가질 경우에는..
    - 주소를 가진 구조체면 값이 같아도 주소가 달라 다를 수 있음

        - 구조체 멤버가 포인터면 메모리 주소를 비교한다
            - 포인터가 가리키는 값이 아니라
- 동적 메모리 할당을 이용한 깊은 복사
    - 이제 이런 것도 할 수 있음

        - 이전에 구조체끼리 깊은 복사가 안 되는 경우를 동적 메모리 할당으로 해결할 수 있다
            - 구조체에 문자열이 있으면 크기가 가변적이므로 복사하기 힘들었고 주소만 복사하는 약한 복사를 함
        - memcpy()말고 strcpy()로도 똑같이 할 수 있다
- 구조체 멤버 변수 - 배열 vs 포인터

    - 동적 메모리를 사용하느 포인터
        - 포인터 구조체 멤버는 주소를 반환함.
        - 그래서, 대입할 때 주소를 대입해버리고,
          파일에 저장할 때 주소를 저장해버린다.
    - 크기 제한이 되는 경우 스택 메모리 사용
        - 예: 이름
            - 10 글자 정도로 제한 가능
    - 크기 제한이 안 되는 경우 동적 메모리 사용
        - 예: 문장
            - 5글자일 지 100글자일 지 알 수 없음
- 베스트 프랙티스: 정적 vs 동적 메모리
    - 정적 메모리를 우선적으로 사용할 것
        - 훌륭한 C 프로그래머들은 최대한 정적 메모리를 쓰려고 함
    - 안 될 때만 동적 메모리
        - 동적 메모리는 실수할 여지가 많고 느리기 때문에 잘 안 쓰려고 함