00_Libft

1. ft_memset.c

```
void  *ft_memset(void *ptr, int value, size_t size)
{
  unsigned char *res;
  //unsigned char를 사용하는 이유
  //부호비트를 제외하고 byte 단위로 접근하여 값을 다루기 위해

  res = (unsigned char *)ptr;
  while (size--)
    *res++ = value;
  return (ptr);
}
```

시작 주소 ptr부터 연속된 size(byte)만큼 value로 초기화하는 함수.

value는 unsigned char로 형변환된다.

메모리에 접근할 때 **unsigned char** 형을 사용하는 이유

- unsigned char 외에 다른 type의 경우, 내부 비트의 일부를 값을 표현하는데 사용할 뿐만 아니라 부호비트로도 사용하지만 unsigned char는 그렇지 않음. → 부호를 고려하지 않음.
- 임의의 메모리에 byte단위로 접근하여 값을 다룰 때는 unsigned char를 사용해야 full portability(이식성)을 얻을 수 있다.
- 1byte씩 참조해서 값을 넣는 작업을 할 때 부호비트로 잘못 넣지 말라는 의미로 사용. → byte단위로 사용하는 이유는 포인터의 주소를 참조하기 위한 것.
  - 포인터의 크기는 32bit(4byte), 64bit(8byte)

2. ft_bzero.c

```
void  ft_bzero(void *ptr, size_t size)
{
  ft_memset(ptr, '\0', size);
  //ft_memset 함수를 이용하여 0으로 초기화.
}
```

시작 주소 ptr부터 size(byte)만큼 0으로 초기화하는 함수.

3. ft_memcpy.c

```c
void  *ft_memcpy(void *dest, const void *src, size_t size)
{
  unsigned char   *dp;
  const unsigned char *sp;
  size_t        i;
  //const를 사용하는 이유
  //넘어온 인자가 변수로 저장되어서 넘어온건지 "test"처럼 상수로 넘어온건지
  //모르기 때문에 참조해서 값을 덮어 쓰는 행위를 막기위해

  if (!dest && !src)
    return (0);
  dp = (unsigned char *)dest;
  sp = (const unsigned char *)src;
  i = 0;
  while (i < size)
  {
    dp[i] = sp[i];
    i++;
  }
  return (dest);
}
```

src가 가르키는 곳부터 size(byte)만큼을 dest에 복사하는 함수.

- 메모리 영역은 겹치지 않으며, 만약 겹쳐서 사용한다면 memmove를 사용한다.
- src의 null 문자를 검사하지 않는다. 언제나 size(byte)만큼을 복사.
- overflow 문제를 방지하기 위해 dest와 src가 가리키는 배열의 크기는 반드시 size(byte) 이상이어야 하며, 서로 겹치면 안된다.
    - 만일 두 메모리 블록이 겹쳐있다면 memmove 함수를 이용하는 것이 좋다.

4. ft_memccpy.c

```c
void  *ft_memccpy(void *dest, const void *src, int c, size_t size)
{
  unsigned char     *dp;
  const unsigned char   *sp;
  size_t          i;

  if (!dest && !src)
    return (0);
  dp = (unsigned char *)dest;
  sp = (const unsigned char *)src;
  i = 0;
```

```
  while (i < size)
  {
    dp[i] = sp[i];
    if (sp[i] == (unsigned char)c)
      return (&dp[i] + 1);
      //c를 만나면 복사를 중단하고 dest의 다음 주소를 반환.
    i++;
  }
  return (0);
}
```

src가 가르키는 곳부터 size(byte)만큼 c를 만날 때 까지 dest에 복사하는 함수.

- 복사 중 src에서 c를 만나면 c까지 복사하고 중단하고 복사가 끝난 **dest의 다음 주소를 반환**한다.
- c를 만나지 않으면 size(byte)만큼 복사 후 null를 반환.

5. ft_memmove.c

```
void  *ft_memmove(void *dest, const void *src, size_t size)
{
  unsigned char   *dp;
  const unsigned char *sp;

  if (!dest && !src)
    return (0);
  dp = (unsigned char *)dest;
  sp = (const unsigned char *)src;
  //overlap이 발생하지 않도록 조정.
  //dest가 뒤인 경우 주소값을 맨 뒤로 옮긴 뒤 뒤에서 부터 복사.
  if (dp > sp)
  {
    dp += size;
    sp += size;
    //주소값을 맨 뒤로 옮긴다.
    while (size--)
      *--dp = *--sp;
  }
  //dest가 앞인 경우 그대로 복사.
  else
  {
    while (size--)
      *dp++ = *sp++;
  }
  return (dest);
}
```

src가 가르키는 곳부터 size(byte)만큼 dest에 옮기는 함수.

- 메모리주소가 겹칠 경우 memcpy 대신 사용.
- man page
  - The memmove() function copies len bytes from string src to string dst. The two strings may overlap; the copy is always done in a non-destructive manner.

6. ft_memchr.c

```
void  *ft_memchr(const void *ptr, int value, size_t size)
{
  const unsigned char *s;

  s = (const unsigned char *)ptr;
  while (size--)
  {
    if (*s == (unsigned char)value)
      return (void*)s;
    s++;
  }
  return (0);
}
```

ptr의 데이터 중 size(byte)만큼 검사하여 value를 찾아 그 주소를 반환하는 함수.

7. ft_memcmp.c

```
int   ft_memcmp(const void *s1, const void *s2, size_t size)
{
  const unsigned char *p1;
  const unsigned char *p2;

  p1 = (const unsigned char *)s1;
  p2 = (const unsigned char *)s2;
  while (size--)
  {
    if (*p1 != *p2)
      return (*p1 - *p2);
    p1++;
    p2++;
  }
  //같으면 while문을 전부 돌기 때문에 0을 반환.
  return (0);
}
```

s1이 가르키는 size(byte)의 데이터와 s2가 가르키는 size(byte)의 데이터를 비교하는 함수.

- 같으면 0 반환.
- 다르면 0이 아닌 다른 값 반환.

8. ft_strlen.c

```
size_t  ft_strlen(const char *str)
{
  size_t  i;

  i = 0;
  while (str[i])
    i++;
  return (i);
}
```

str 이 가르키는 문자열의 길이를 반환한다.

- size_t
  - C언어에서의 임의의 객체가 가질 수 있는 최대 크기를 나타낸다.

9. ft_strlcpy.c

```
size_t    ft_strlcpy(char *dest, const char *src, size_t size)
{
  size_t  src_len;
  size_t  i;

  src_len = ft_strlen(src);
  //src의 길이
  if (size == 0)
    return (src_len);
  i = 0;
  while (src[i] != '\0' && i < (size - 1))
  {
    dest[i] = src[i];
    i++;
    //dest에 src를 복사한다.
  }
  dest[i] = '\0';
  return (src_len);
}

==========================ft_memcpy이용==========================
```

```c
size_t  ft_strlcpy(char *dest, const char *src, size_t size)
{
  size_t  src_len;

  src_len = ft_strlen(src);
  //src_len + 1 < size일 경우 src의 길이만큼 복사한다.
  //이 조건을 size != 0 보다 먼저 해주는 이유
  //size != 0의 조건을 먼저해주면 이 조건이 실행이 되지 않는다.
  if (src_len + 1 < size)
  {
    ft_memcpy(dest, src, src_len);
    dest[src_len] = '\0';
  }
  //src에서 dest로 size-1개의 문자열을 복사하고 size번째에 null을 붙인다.
  else if (size != 0)
  {
    ft_memcpy(dest, src, size - 1);
    dest[size - 1] = '\0';
  }
  return (src_len);
}
```

src에서 dest로 size-1개의 문자열을 복사하고 size-1번째에는 null을 붙인다.

- **반환값은 src의 길이다.** (null을 제외한 문자의 길이)
- src+1의 길이가 size보다 작을 경우 src의 길이만큼 복사하고 null를 붙인다.

10. ft_strlcat.c

```c
size_t  ft_strlcat(char *dest, const char *src, size_t size)
{
  size_t  s_len;
  size_t  d_len;

  s_len = ft_strlen(src);
  d_len = ft_strlen(dest);
  if (d_len > size)
    d_len = size;
  if (d_len == size)
    return (size + s_len);
  if (s_len + d_len < size)
  {
    ft_memcpy(dest + d_len, src, s_len);
    dest[d_len + s_len] = '\0';
  }
  else
  {
    ft_memcpy(dest + d_len, src, size - d_len - 1);
    dest[size - 1] = '\0';
  }
  return (d_len + s_len);
  //만들고자하는 문자열의 길이 반환.
```

```
  //dest > size -> d_len = size 이므로 s_len + size 반환
}
```

dest뒤에 src를 붙이는 함수.

- size = strlen(dest) + strlen(src) + nulll
- 반환값
  - 만들고자 하는 문자열의 길이(d_len + s_len)
  - size == 0 → src의 길이 return.
  - dest < size → src + dest의 길이 return.
  - dest ≥ size → src + size의 길이 return.

11. ft_strchr.c

```
char  *ft_strchr(const char *str, int c)
{
  while (*str != c)
  {
    if (*str == '\0')
      return (0);
    str++;
  }
  //c를 만날때까지 반복문을 돌고 c를 찾지못하고 문자열이 끝나면 null 반환.
  return (char *)str;
}
```

str 문자열에서 특정문자 c를 찾는 함수.

c를 찾으면 해당 문자의 포인터를 반환한다.

12. ft_strrchr.c

```
char  *ft_strrchr(const char *str, int c)
{
  size_t  len;

  len = ft_strlen(str);
  while (len != 0 && *(str + len) != c)
    len--;
  //문자열의 뒤에서부터 str[len]의 값이 c와 같은지 비교.
  //*(str + len) == str[len].
  if (*(str + len) == c)
```

```
    return (char *)(str + len);
  return (0);
}
```

str 문자열에서 마지막으로 있는 문자 c를 찾는 함수.

마지막으로 있는 c를 찾는 것이기 때문에 문자열의 뒤에서부터 비교를 한다.

13. ft_strnstr.c

```
char  *ft_strnstr(const char *big, const char *little, size_t size)
{
  size_t i;
  size_t j;

  i = 0;
  if (little[i] == '\0')
    return ((char *)big);
  //little이 비어있을 경우 big을 반환.
  while (big[i] && i < size)
  {
    //big이 문자열이 끝나지 않는 동아 little 문자열이 big에 존재하는지 확인.
    j = 0;
    while (little[j] && big[i] == little[j] && i < size)
    {
      i++;
      j++;
    }
    //little이 존재하면 각각 index를 증가시키며 while문을 통해 비교
    if (!little[j])
      return ((char *)&big[i - j]);
    //index를 증가시켜서 little문자열이 끝난 경우
    //big안에서 little을 찾았다는 의미 -> big에서 little의 시작 부분 주소 return.
    i = i - j + 1;
    //big과 little을 비교하다가 중간에 다른 문자가 있을 경우
    //big에서 little이 중간에 달라지기 때문에 비교했던 little이 의미가 없어짐.
    //비교했던만큼 다시 돌아가서 index를 다시 증가.
    //big의 index를 증가시키고 little의 index를 다시 초기화한다.
  }
  return (0);
}
```

big 문자열의 size 길이 중에서 little 문자열을 찾는 함수.

- 반환값
  - little 문자열이 비었을 경우, big return.
  - big에서 little을 찾지못할 경우, null return.
  - little 문자열을 찾은 경우, big에서 little의 시작 부분 주소를 return.

14. ft_strncmp.c

```c
int   ft_strncmp(const char *s1, const char *s2, size_t n)
{
  unsigned char *str1;
  unsigned char *str2;
  size_t      i;

  i = 0;
  str1 = (const unsigned char *)s1++;
  str2 = (const unsigned char *)s2++;
  while (i < n && (str1[i] || str2[i]))
  {
    if (str1[i] != str2[i])
      return (str1[i] - str2[i]);
    //두 문자열이 중간에 같지 않으면 해당 ascii의 값의 차이를 반환.
    //s1 > s2 -> 0보다 큰 수
    //s1 < s2 -> 0보다 작은 수
    i++;
  }
  return (0);
}
```

s1 과 s2 문자열을 n만큼 비교하는 함수.

두 문자열의 비교는 ascii 값을 비교한다.

- 반환값
  - 문자열이 같은 경우, 0 return.
  - s1이 작은 경우, 0보다 작은 수 return.
  - s1이 큰 경우, 0보다 큰 수 return.

15. ft_atoi.c

```c
int   ft_atoi(const char *str)
{
  int   res;
  int   sign;

  res = 0;
  sign = 1;
  while ((*str >= 9 && *str <= 13) || *str == ' ')
    str++;
  if (*str == '+' || *str == '-')
    if (*str++ == '-')
      sign *= -1;
```

```
  while (ft_isdigit(*str))
  {
    res *= 10;
    res += *str++ - '0';
  }
  return (res * sign);
}
```

str 문자열을 정수로 변환해주는 함수.

- white space를 만나면 문자열을 증가.
- '-' 부호를 만났을 때, 음수가 출력되게 끔 한다.
- 문자가 정수인지 판단하여 결과값을 저장하는 res에 담는다.

16. ft_isalpha.c

```
int   ft_isalpha(int c)
{
  return (('a' <= c && c <= 'z') || ('A' <= c && c <= 'Z'));
}
```

c의 ascii값이 영문자이면 1을 반환하는 함수. 영문자가 아니라면 0을 반환.

17. ft_isdigit.c

```
int   ft_isdigit(int c)
{
  return (c >= '0' && c <= '9');
}
```

c의 ascii값이 0~9에 해당하는지 판별하는 함수.

18. ft_isalnum.c

```
int   ft_isalnum(int c)
{
  if (ft_isalpha(c) || ft_isdigit(c))
    return (1);
  return (0);
}
```

c가 숫자 또는 영문자인지 판별하는 함수.

19. ft_isascii.c

```
int   ft_isascii(int c)
{
  return (c >= 0 && c < 128);
}
```

c가 ascii 문자인지 검사하는 함수.

20. ft_isprint.c

```
int   ft_isprint(int c)
{
  return (c >= ' ' && c <= '~');
}
```

c가 출력가능한 문자인지 판별하는 함수.

21. ft_toupper.c

```
int   ft_toupper(int c)
{
  if (c >= 'a' && c <= 'z')
    return (c - 32);
  return (c);
}
```

c가 소문자라면 대문자로 변환하는 함수.

22. ft_tolower.c

```
int   ft_tolower(int c)
{
  if (c >= 'A' && c <= 'Z')
    return (c + 32);
  return (c);
}
```

c가 대문자라면 소문자로 변환하는 함수.

23. ft_calloc.c

```
void  *ft_calloc(size_t nelem, size_t size)
{
  void *res;

  if (!(res = malloc(nelem * size)))
    return (0);
  ft_bzero(res, nelem * size);
  return (res);
}
```

size크기의 변수를 nelem개 저장할 수 있는 공간을 할당하는 함수.

- 할당된 공간의 값을 모두 0으로 초기화 한다.

24. ft_strdup.c

```
char  *ft_strdup(const char *src)
{
  size_t  len;
  char  *dest;

  len = ft_strlen(src);
  if (!(dest = (char*)malloc(sizeof(char) * (len + 1))))
    return (0);
  len = 0;
  while (src[len])
  {
    dest[len] = src[len];
    len++;
  }
  dest[len] = '\0';
  return (dest);
}
```

문자열 str 길이 + 1의 크기를 malloc으로 할당 후 문자열 str를 복사하는 함수.

- strcpy의 동작에 malloc을 추가.

PART 2

1. ft_substr.c

```
char  *ft_substr(char const *s, unsigned int start, size_t len)
{
  char *res;

  if (ft_strlen(s) < start)
    return (ft_strdup(""));
  //malloc을 이용하여 배열을 할당해준다.
  if (!(res = (char *)malloc(sizeof(char) * (len + 1))))
    return (0);
  //start index부터 len만큼 복사
  ft_memcpy(res, s + start, len);
  res[len] = '\0';
  return (res);
}
```

s문자열의 start index부터 len만큼 복제하여 부분 문자열을 가져오는 함수.

- s문자열의 길이가 start보다 작을 경우 빈 문자열을 반환한다.
  - 시작 인덱스가 10인데 문자열이 5이면 복사를 하지못하므로 빈 문자열 반환.

2. ft_strjoin.c

```
char  *ft_strjoin(char const *s1, char const *s2)
{
  int   len1;
  int   len2;
  char  *res;

  if (!s1 || !s2)
    return (0);
  len1 = ft_strlen(s1);
  len2 = ft_strlen(s2);
  if (!(res = (char *)malloc(sizeof(char) + (len1 + len2 + 1))))
    return (0);
  ft_memcpy(res, s1, len1);
  ft_memcpy(res + len1, s2, len2);
  res[len1 + len2] = '\0';
  return (res);
}
```

문자열 s1에 문자열 s2를 붙여 새로운 문자열을 만드는 함수.

- strlcat과 비슷해 보일 수 있지만, strlcat은 dest 뒤에 이어붙이는 함수이고 strjoin은 새로운 문자열 배열에 저장하는 함수.

3. ft_strtrim.c

```
char  *ft_strtrim(char const *s1, char const *set)
{
  char  *res;
  size_t  len;

  //예외처리
  if (!s1 || !set)
    return (0);
  //특정문자를 찾는 ft_strchr함수
  //앞에서 set을 잘라내고 문자열을 증가
  while (*s1 && ft_strchr(set, *s1) != 0)
    s1++;
  //문자열의 길이는 앞의 set을 잘라낸 후의 문자열 길이
  len = ft_strlen(s1);
  //뒤에서 set을 잘라내고 문자열의 길이를 감소시켜 뒤에서 부터 비교
  while (len && s1[len - 1] && ft_strchr(set, s1[len - 1]) != 0)
    len--;
  if (!(res = (char*)malloc(sizeof(char) * (len + 1))))
    return (0);
  ft_memcpy(res, s1, len);
  res[len] = '\0';
  return (res);
}
```

문자열 s1에서 앞, 뒤에 있는 set을 잘라낸 새로운 문자열을 반환하는 함수.

- 중간에 있는건 잘라내지 않는다.
- 보통 white space 를 넣어주어 공백제거에 사용함.
- 문자열 s1이 중간에서 멈추지않고 전부 다 잘리는 경우 null이 아닌 빈 문자열을 반환.

4. ft_split.c

```
//각 배열에 저장할 문자열의 길이를 구한다.
static int    cnt_size(char const *s, char c)
{
  int size;

  size = 0;
  while (*s && *s != c)
  {
    size++;
    s++;
  }
  return (size);
}
```

```c
//구분자로 나누고 저장해야할 문자열의 총 개수를 구한다.
static int    cnt_word(char const *s, char c)
{
  int cnt;

  cnt = 0;
  //처음 시작이 구분자이면 문자열 인덱스 증가
  while (*s && *s == c)
    s++;
  //문자열을 계속 반복
  while (*s)
  {
    cnt++;
    //cnt를 증가시키고 문자열을 증가시킨다.
    //구분자를 기준으로 둘 중 하나만 실행된다.
    while (*s && *s != c)
      s++;
    while (*s && *s == c)
      s++;
  }
  return (cnt);
}

/*
static int    cnt_word(char const *s, char c)
{
  int i;
  int cnt;

  i = 0;
  cnt = 0;
  //문자열 전체 while
  while (s[i])
  {
    //구분자를 만나면 그 전까지의 문자를 저장해야한다.
    //구분자가 아니라면 cnt를 증가시키고 다시 while을 통해 구분자를 만날때까지 증가시킨다.
    if (s[i] != c)
    {
      cnt++;
      while (s[i] && s[i] != c)
        i++;
    }
    //구분자를 만나면 인덱스 증가.
    else
      i++;
  }
  return (cnt);
}
*/

//이차원배열이 할당되지 않으면 전부 free를 시켜주기 위한 함수.
static char   **arr_free(char **arr, int i)
{
  while (i--)
    free(arr[i]);
  free(arr);
  return (0);
```

```
}

//구분자 c를 기준으로 함수를 나눈다.
char      **ft_split(const char *s, char c)
{
  char  **res;
  int   size;
  int   i;

  //구분자 c를 기준으로 나눈 문자열을 저장하기 위해 이차원배열을 할당한다.
  i = 0;
  //cnt_word를 통해 이차원배열의 행의 크기를 먼저 할당받는다.
  if (!(res = (char **)malloc(sizeof(char *) * (cnt_word(s, c) + 1))))
    return (0);
  //문자열이 끝날 때까지 반복문을 돌며 구분자 c를 만나는지 조건문을 이용해 분리한다.
  while (*s)
  {
    if (*s != c)
    {
      size = cnt_size(s, c);
      //저장할 문자열의 길이를 구하고 첫번째 행부터 문자열을 저장한다.
      //만약 메모리가 제대로 할당되지않으면 만들었던 이차원배열을 해제하고 반환한다.
      //내부에서 터지는지 확인하여 메모리의 해제를 해준다.
      if (!(res[i] = malloc(size + 1)))
        return (arr_free(res, i));
      //메모리가 제대로 할당됐다면 해당 행에 문자열을 저장하고 인덱스를 증가시켜 다음 행에 저장하게끔 한다.
      ft_strlcpy(res[i++], s, size + 1);
      while (*s && *s != c)
        s++;
    }
    else
      s++;
  }
  res[i] = 0;
  return (res);
}
```

문자열 s를 구분자 c를 기준으로 나누는 함수.

- 이차원배열을 할당하여 각 열마다 문자열을 저장한다.
- 행(저장할 문자열의 개수)의 개수를 세는 것이 cnt_word.
- 열(저장할 문자열의 길이)의 개수를 세는 것이 cnt_size.

5. ft_itoa.c

```
//문자열의 길이를 구한다.
//10으로 나누면서 count를 하면 문자열의 길이를 구할수있다.
static int    cnt_num(int n)
{
  int cnt;

  cnt = 0;
```

```
  while (n)
  {
    n /= 10;
    cnt++;
  }
  return (cnt);
}

//정수에 ascii값을 더해 문자열로 변환하는 함수
//숫자가 10보다 작으면 바로 바꿀 수 있다.
//10보다 크면 10으로 나는 나머지부터 변환한다. 나머지부터 변환하기 때문에 뒤에서 부터 저장한다.
static void   write_num(char *dest, unsigned int n)
{
  if (n < 10)
    *dest = n + '0';
  else
  {
    *dest = n % 10 + '0';
    //뒤에서 부터 저장
    write_num(dest - 1, n / 10);
  }
}

char      *ft_itoa(int n)
{
  char      *res;
  unsigned int  nbr;
  int       len;

  nbr = n;
  if (n == 0)
    return (ft_strdup("0"));
  else
  {
    len = n < 0 ? cnt_num(n) + 1 : cnt_num(n);
    if (!(res = (char *)malloc(sizeof(char) * (len + 1))))
      return (0);
    //음수면 문자열 처음에 '-'부호를 넣고 뒤에서 부터 채운다.
    if (n < 0)
    {
      res[0] = '-';
      write_num((res + len - 1), -nbr);
    }
    //양수면 부호없이 뒤에서 부터 채운다.
    else
      write_num((res + len - 1), nbr);
    res[len] = '\0';
  }
  return (res);
}
```

정수를 문자열로 변환하는 함수

- 변환하고자하는 숫자가 0이면 "0"을 반환.
- 문자열로 변환하기 위해서는 문자열의 길이를 알아야한다.

        - 숫자가 음수라면 '-'부호가 들어가야 하므로 구한 길이에 + 1을 더해준다.
    - 10으로 나눈 몫을 저장하면 1의 자리를 저장하지 못하기 때문에 나머지부터 저장한다.
        - 나머지부터 저장할 때 문자열의 뒤에서부터 저장한다.

6. ft_strmapi.c

```
char  *ft_strmapi(char const *s, char (*f)(unsigned int, char))
{
  size_t  i;
  char  *res;

  if (!s)
    return (0);
  if (!(res = (char *)malloc(sizeof(char) * (ft_strlen(s) + 1))))
    return (0);
  i = 0;
  //해당 인덱스와 그 인덱스에 해당하는 문자를 f함수 처리하여 새로운 문자를 만든다.
  //반복문을 통해 함수포인터를 이용하고 새로운 문자열을 만든다.
  while (s[i])
  {
    res[i] = f(i, s[i]);
    i++;
  }
  res[i] = '\0';
  return (res);
}
```

문자열 s의 인덱스와 그 인덱스에 해당하는 문자를 f함수에 보내 변환하고 그 문자를 반환받아 새로운 문자열을 만드는 함수.

7. ft_putchar_fd.c

```
void  ft_putchar_fd(char c, int fd)
{
  write(fd, &c, 1);
}
```

문자 c를 fd 파일 디스크립터에 출력하는 함수.

8. ft_putstr_fd.c

```c
void  ft_putstr_fd(char *s, int fd)
{
  write(fd, s, ft_strlen(s));
}
```

문자열 s를 fd 파일 디스크립터에 출력하는 함수.

9. ft_putendl_fd.c

```c
void  ft_putendl_fd(char *s, int fd)
{
  ft_putstr_fd(s, fd);
  ft_putchar_fd('\n', fd);
}
```

문자열 s를 fd 파일 디스크립터에 출력한 후 줄바꿈하는 함수.

10. ft_putnbr_fd.c

```c
void  ft_putnbr_fd(int n, int fd)
{
  unsigned int nbr;

  nbr = n;
  if (n < 0)
  {
    ft_putchar_fd('-', fd);
    nbr = -n;
  }
  if (nbr >= 10)
    ft_putnbr_fd(nbr / 10, fd);
  ft_putchar_fd(nbr % 10 + '0', fd);
}
```

정수 n을 fd 파일 디스크립터에 출력하는 함수.

BONUS PART

struct

```c
typedef struct  s_list
{
```

```
  void        *content;
  struct s_list   *next;
}       t_list;
```

s_list 구조체를 선언후 t_list의 이름을 부여.

1. ft_lstnew

```
t_list  *ft_lstnew(void *content)
{
  t_list  *res;

  if (!(res = (t_list *)malloc(sizeof(t_list))))
    return (0);
  res->content = content;
  res->next = 0;
  return (res);
}
```

t_list구조체 멤버 *content에 새로만들고자 하는 res 리스트의 *content를 매칭하는 함수.

2. ft_lstadd_front

```
void  ft_lstadd_front(t_list **lst, t_list *new)
{
  new->next = *lst;
  *lst = new;
}
```

기존 list의 맨 앞에 새로운 new 리스트를 추가하는 함수.

3. ft_lstsize

```
int   ft_lstsize(t_list *lst)
{
  int cnt;

  cnt = 0;
  while (lst)
  {
    //리스트가 다음 리스트를 향하게 한다. 증감연산이라 생각하면 된다.
    lst = lst->next;
```

```
    cnt++;
  }
  return (cnt);
}
```

리스트의 개수를 구하는 함수.

4. ft_lstlast

```
t_list  *ft_lstlast(t_list *lst)
{
  if (!lst)
    return (0);
  while (lst->next)
    lst = lst->next;
  return (lst);
}
```

리스트 마지막 요소의 주소를 반환한다.

5. ft_lstadd_back

```
void  ft_lstadd_back(t_list **lst, t_list *new)
{
  if (*lst == 0)
    *lst = new;
  else
    (ft_lstlast(*lst))->next = new;
}
```

기존 list의 맨 뒤에 새로운 new 리스트를 추가하는 함수.

- 리스트가 비어있다면 바로 new 리스트를 생성.
- 마지막 리스트 뒤에 새로운 리스트를 추가해야하므로 ft_lstlast 함수를 이용해 마지막 리스트를 찾는다.

6. ft_lstdelone

```
void  ft_lstdelone(t_list *lst, void (*del)(void *))
{
  del(lst->content);
```

```
  free(lst);
}
```

리스트의 content를 del로 보내 내용을 삭제한 후 리스트를 free해주는 함수.

7. ft_ lstclear

```
void  ft_lstclear(t_list **lst, void (*del)(void *))
{
  t_list *tmp;

  if (lst == NULL)
    return ;
  //리스트 전체를 free해준다.
  //tmp에 다음 리스트의 주소를 저장해 현재 lst를 삭제한다.
  //현재 lst를 삭제해야했는데 다음 lst를 삭제해버렸다.
  while (*lst)
  {
    tmp = (*lst)->next;
    del((*lst)->content);
    free(*lst);
    *lst = tmp;
  }
  *lst = 0;
}
```

리스트의 content를 전부 del한 후 리스트 전체를 free해주는 함수.

- **ko**
  - 수정 전 free(tmp) → 수정 후 free(*lst);
  - 수정 전에 현재 리스트가 아닌 다음 리스트를 저장하고 있는 tmp를 free를 해주었다.
  - 테스터에서 통과가 된 이유는 바로 null값을 가르키고 프로그램이 종료되어 통과로 인식한 것 같다.

8. ft_lstiter

```
void  ft_lstiter(t_list *lst, void (*f)(void *))
{
  while (lst)
  {
    f(lst->content);
    lst = lst->next;
```

```
  }
}
```

리스트의 모든 content에 대해 f함수 처리를 해주는 함수.

9. ft_lstmap

```
t_list  *ft_lstmap(t_list *lst, void *(*f)(void *), void (*del)(void *))
{
  t_list *curr;
  t_list *res;

  if (!(res = ft_lstnew(f(lst->content))))
    return (0);
  //res를 직접적으로 건들지 않고 사용하기 위해 curr가 res를 가르키게 한다.
  curr = res;
  lst = lst->next;
  while (lst)
  {
    if (!(curr->next = ft_lstnew(f(lst->content))))
    {
      ft_lstclear(&res, del);
      return (0);
    }
    //list를 생성해나가는 과정이다.
    //이 과정이 실패하면 if문으로 들어가 만들었던 모든 리스트를 삭제하고 함수를 종료한다.
    curr = curr->next;
    lst = lst->next;
  }
  //만약 중간에 리스트가 생성되지않고 끝나는 일이 없다면 계속해서 리스트가 만들어졌을 것이다.
  //이는 curr을 이용해 res를 만들었으며, res를 반환한다.
  return (res);
}
```

f함수를 적용한 새로운 리스트를 만드는 함수.

- 새롭게 만든 res를 반환하고, curr을 이용하여 res를 만든다.
- 리스트를 만들다 실패하면 지금까지 만들었던 리스트 전부를 삭제한 후 0을 반한다.

MAKEFILE

```
NAME = libft.a
//만들어질 이름

AR = ar
ARFLAGS = crs
//아카이브 라이브러리
```

```
//gcc를 통해 오브젝트 파일을 다른데서 사용할 수 있게 라이브러리화 해주는 옵션.
//create relink s(lib index 설정)

CC = gcc
CFLAGS = -Wall -Wextra -Werror
//컴파일러 옵션

RM = rm
RMFLAGS = -f
//삭제 옵션

INCLUDES = ./libfh.h
//헤더

SRCS = ft_memset.c\
      ft_bzero.c\
      ft_memcpy.c\
      ft_memccpy.c\
      ft_memmove.c\
      ft_memchr.c\
      ft_memcmp.c\
      ft_strlen.c\
      ft_strlcpy.c\
      ft_strlcat.c\
      ft_strchr.c\
      ft_strrchr.c\
      ft_strnstr.c\
      ft_strncmp.c\
      ft_atoi.c\
      ft_isalpha.c\
      ft_isdigit.c\
      ft_isalnum.c\
      ft_isascii.c\
      ft_isprint.c\
      ft_toupper.c\
      ft_tolower.c\
      ft_calloc.c\
      ft_strdup.c\
      ft_substr.c\
      ft_strjoin.c\
      ft_strtrim.c\
      ft_split.c\
      ft_itoa.c\
      ft_strmapi.c\
      ft_putchar_fd.c\
      ft_putstr_fd.c\
      ft_putendl_fd.c\
      ft_putnbr_fd.c

SRCS_BONUS = ft_lstnew.c\
      ft_lstadd_front.c\
      ft_lstsize.c\
      ft_lstlast.c\
      ft_lstadd_back.c\
      ft_lstdelone.c\
      ft_lstclear.c\
      ft_lstiter.c\
      ft_lstmap.c
```

```
//src file

OBJS = $(SRCS:.c=.o)
OBJS_BONUS = $(SRCS_BONUS:.c=.o)
//object file 생성

all : $(NAME)
//make 기본
//all -> NAME -> OBJS -> .c.o 여기서 컴파일을 해준다.

$(NAME) : $(OBJS)
	$(AR) $(ARFLAGS) $@ $^

bonus : $(OBJS_BONUS) $(OBJS)
	$(AR) $(ARFLAGS) $(NAME) $^

.c.o : $(SRCS) $(INCLUDES)
	$(CC) $(CFLAGS) -c $< -o $@

clean :
	$(RM) $(RMFLAGS) $(OBJS) $(OBJS_BONUS)

fclean : clean
	$(RM) $(RMFLAGS) $(NAME)
//clean에서 objs 삭제 -> fclean 실행파일 삭제

re : fclean all

.PHONY: all bonus clean fclean re
//실제로 똑같은 파일이 존재하면 우선순위에 뒤쳐져서 실행할 이유가 없어 작동하지 않음.
//이를 막기위해서 명령어임을 선언해주는 변수.
```