문자열

소프트웨어학부
박영훈 교수

이 단원의 목표

- 문자열과 포인터의 관계
- 문자열 입출력 하기
- 문자열 관련 라이브러리 사용하기
- 문자열의 배열

문자열

- 문자열이란, 두 개의 따옴표 사이에 있는 문자들의 나열을 말한다. 예를 들어:

```
"When you come to a fork in the road, take it."
```

- 문자열에는 Escape sequence 를 포함이 될 수도 있다.

```
"I said him, \"Be quiet!\"\n"
```

- 또한, 역슬래시는 다음 줄로 넘어간 두 개의 문자열을 연결할 때도 사용된다. 예를 들어:

```
"When you come to a fork in the roa\
d, take it."
```

  은 맨 위의 문자열과 같은 문자열이다.

- 또는, 두 개의 문자열을 연달아 써도 하나의 문자열로 인식한다. 즉,

```
"When you come to a fork in the roa""d, take it."
```

  도 같은 문자열이다.

문자열이 저장되는 방법

- C에서 문자열의 길이가 n일 때, n+1바이트의 메모리가 사용된다.
- n+1바이트 중 처음 n바이트에는 문자열이 한 character씩 순서대로 저장된다.
- n+1바이트 중 마지막 한 바이트에는 `NULL` 문자가 들어간다.
- `NULL` 문자는 아스키코드값이 0인 문자로, 소스코드에서는 `NULL` 혹은 `'\0'`으로 표시한다.
- 예를 들어, `"ab\"d"`라는 문자열은 다음과 같이 5바이트가 메모리에 잡히고 저장된다.

주소	저장된 값
...	...
00400614	61
00400615	62
00400616	22
00400617	64
00400618	0
...	...

문자열이 저장되는 방법

- 문자열은 C에서 포인터로 취급 되며, 문자열이 저장된 장소의 첫 번째 주솟값을 나타낸다.
- `printf("abc");` 에서도, `"abc"`가 저장된 메모리의 첫 번째 주솟값이 `printf` 함수의 인자로 들어가는 것이다.

```
char *p;
p = "abc";
```

와 같은 소스코드에서 `p`는 `"abc"`의 첫 번째 문자인 `'a'`를 가리키고 있다. 즉, `p`에는 `'a'`가 저장된 메모리의 주솟값이 저장되어 있다.

- 포인터는 문자열처럼 취급될 수 있으므로 `p[1]` 또는 `*(p+1)`은 `'b'` 값을 나타낸다.
- 또한, 문자열 그 자체도 포인터이므로 `"abc"[1]`의 값도 `'b'`이다.

- 이 성질을 이용하여 다음과 같은 함수를 만들 수 있다.

```
char digit_to_hex_char(int digit){
    return "0123456789ABCDEF"[digit];
}
```

문자열이 저장되는 방법

- 문자열은 문자의 배열이기 때문에, 배열 형태로도 선언하여 저장할 수 있다. 예를 들어:

```
char ch[4] = "abc";
```

- **ch[0]**, **ch[1]**, **ch[2]**, **ch[3]**의 값은 각각 **'a'**, **'b'**, **'c'**, **'\0'**이 된다.

- 선언되는 문자열의 크기를 저장할 문자열의 길이보다 더 크게 잡을 수도 있다. 이 경우, 초기화되지 않은 공간은 모두 **NULL**문자로 초기화 된다. 즉,

```
char ch[6] = "abc";
```

  라고 선언하면, **ch[4]**, **ch[5]**의 값은 모두 **'\0'**이 된다.

- **char ch[3] = "abc";** 와 같이, 선언되는 문자열의 크기가 저장할 문자열과 같아도 되긴 하지만, 원하지 않는 결과를 초래할 수 있다.

- 이를 위하여 **char ch[] = "abc";** 와 같이 길이를 지정하지 않는 것도 한 방법이다.

문자열 사용 시 주의점

- 포인터로 선언된 문자열을 수정하려고 하면 에러가 발생한다. 예를 들어,

```
char* p = "abc";
*(p+1) = 'd';
```

  라고 하면 런타임 오류가 발생한다.

- 하지만, 배열로 선언된 문자열은 수정이 가능하다. 즉, 다음은 가능하다.

```
char ch[4] = "abc";
ch[1] = 'd';
```

- 앞의 단원에서 다뤘던 바와 같이, 포인터와 배열은 사용 방법이 바뀌어도 된다. 즉, `p[1]`이나 `*(ch + 1)`과 같이 사용해도 된다.

- 포인터로 선언 되었을 때와 배열로 선언 되었을 때의 차이점은 다음과 같다.
  - 포인터로 초기화된 문자열은 내용을 수정할 수 없지만, 배열로 초기화된 문자열은 수정이 가능하다.
  - 포인터로 선언되면 그 포인터 변수는 다른 곳을 가리키는 것이 가능하다. 즉, `p = p + 1;`과 같은 것이 가능하지만, `ch = ch + 1;`은 불가능하다.

문자열 사용 시 주의점

- ""와 ''는 완전히 다른 의미이다. 즉,

  "a" 은 포인터로 취급 되고,

  'a' 는 값으로 취급 된다.

- 따라서 **printf("a");**는 맞지만, **printf('a');**는 컴파일 에러를 발생시킨다.

- 문자열 포인터를 초기화하지 않고 그냥 사용하면 런타임 에러가 발생한다. 즉,

```
char* p;
*p = 'a';  *(p+1) = 'b';    *(p+2) = 'c';    *(p+3) = '\0';
```

  는 메모리 오류를 일으킨다. 이를 위해서는 다음 두 방법 중 하나를 쓰면 된다.

```
char a[4] = "abc";
char* p = a;
```

또는

```
char* p;
p = (char*)malloc(4);
*p = 'a';  *(p+1) = 'b';    *(p+2) = 'c';    *(p+3) = '\0';
```

문자열의 입출력

- 문자열을 화면에 출력할 때는 **printf**나 **puts** 함수를 사용한다.

문자: putchar

- 문자열을 키보드로 입력받아 저장할 때는 **scanf**나 **gets** 함수를 사용한다.

문자: getchar

- 문자열을 입력 받을 때 키보드로 입력한 문자의 개수를 검사하지 않는다. 따라서 선언한 문자열 길이보다 더 많은 문자를 입력하면 문제가 발생한다.

- 문자열을 입출력할 때 사용되는 변수 표시자는 **%s**이다.

문자열 출력 함수: printf()

- **printf** 함수를 이용하여 문자열을 출력하기 위해서는 다음과 같이 한다.

  포인터로 선언 되었을 때:

```
char* p = "abcde";
printf("%s", p);
```

  배열로 선언 되었을 때:

```
char a[6] = "abcde";
printf("%s", a);
```

  문자열 자체를 출력:

```
printf("%s", "abcde");
```
또는 `printf("abcde");`

- %m.ps

  m ⊕ 오른쪽정렬, m개칸
  m ⊖ 왼쪽정렬, m개칸

  출력되는 문자열의 길이가 m보다 짧을 때, m개 칸을 차지하고, 오른쪽 정렬로 출력한다.
  왼쪽 정렬로 출력하고 싶으면 m 대신 -m이라 쓰면 된다.

  출력되는 문자열의 갈이가 p보다 길 때, 처음 p개의 문자만을 출력한다.

(교재된 문자열만 출력해줌

문자열 출력 함수: puts()

- **puts** 함수를 이용하여 문자열을 출력하기 위해서는 다음과 같이 한다.

  포인터로 선언 되었을 때:

```
char* p = "abcde";
puts(p);
```

  배열로 선언 되었을 때:

```
char a[6] = "abcde";
puts(a);
```

  문자열 자체를 출력:

```
puts("abcde");
```

- **puts** 함수로 문자열을 출력하면 출력 후, 자동으로 한 줄 바뀐다.

문자열 출력 함수: printf(), puts()

- printf와 puts함수는 문자열에서 '\0'이 등장하기 직전까지 출력한다.

(10) NULL: 문자열의 끝을 나타냄

```
char* p = "abcdef\0ghij";
printf("%s", p); // 또는 puts(p);
```

(총 10바이트 할당되었을것)

를 실행하면 **abcdef**만 출력된다.

- 또한, 포인터 변수가 가리키는 곳으로부터 출력한다.

```
char* p = "abcdef\0ghij";
printf("%s", p + 2);    // 또는 puts(p + 2);
```

를 실행하면 **cdef**만 출력된다.

p → a
p+1 → b
p+2 → c

```
printf("%9.6s\n", p);
```

9칸할당, 6칸출력
··· ABCDEF
cf) %.9.3s → ····· ABC

문자열 입력 함수: scanf()

- scanf 함수를 이용하여 문자열을 입력 받을 때는 다음과 같이 한다.

```
char a[10];
scanf("%s", a);  // 9바이트 이내의 문자열을 입력해야 함.
```

- 또는

```
char a[10];
char *p = a;
scanf("%s", p);  // 9바이트 이내의 문자열을 입력해야 함.
```

- 또는 동적 할당을 이용하여 메모리 공간을 확보할 수 있다.

```
char *p;
p = (char*)malloc(N + 1);
scanf("%s", p);  // N바이트 이내의 문자열을 입력해야 함.
...
free(p);
```

문자열 입력 함수: gets()

- **scanf** 함수는 키보드로 입력한 문자열에서 white space를 만나면, 그 전까지만 읽어들인다. 즉, **scanf("%s", a);** 에서 **vw xyz**를 입력하면, **a[0]**, **a[1]**, **a[2]**에는 각각 **'v'**, **'w'**, **'\0'**이 저장되고, **' '**, **'x'**, **'y'**, **'z'**는 배열 **a[]**에 들어가지 않는다.
- 하지만, **gets**함수는 줄바꿈을 제외한 모든 white space를 읽어들일 수 있다. 즉, **gets** 함수는 엔터 누르기 직전까지의 모든 문자열을 입력 받는다.

```
char a[10];
char* p = a;
gets(a);   //gets(p); 도 가능, 9바이트까지 입력 가능
```

또는 동적할당으로 다음과 같이 사용할 수 있다.

```
char* p;
p = (char*)malloc(N + 1);  // N바이트까지 입력 가능
gets(p);
...
free(p);
```

- 이 때, **vw xyz**를 입력하면, **a[0]**, **a[1]**, **a[2]**, **a[3]**, **a[4]**, **a[5]**, **a[6]**에는 각각 **'v'**, **'w'**, **' '**, **'x'**, **'y'**, **'z'**, **'\0'**이 들어간다.

문자열 입력 함수: scanf(), gets()

- 만일 어떤 길이가 N인 문자열 a[]가 선언되었을 때, scanf나 gets함수를 부른 상황에서 N바이트 이상의 문자를 입력하면 에러는 발생하지 않지만 의도하지 않은 결과가 나올 수 있다.

- 이를 방지하기 위하여, scanf에서는 %s 대신 %ns를 사용할 수 있다. (단, n에는 자연수 값이 들어가야 한다.) 예를 들어,

```
scanf("%9s", a);
```

- 라고 하면 10바이트 이상 입력해도 처음 9바이트까지만 들어간다.

- 따라서 `char a[10];`이라 선언되어 있으면 `scanf("%9s", a);`와 같이 하는 것이 좋다.

- gets() 함수도 비슷한 이유로 fgets() 함수를 사용하는 것이 더 안전하다. 사용법은 gets() 함수와 같다.

문자열 안에 있는 문자 사용하기.

- ```
  char a[13] = "Hello World!";
  char *p = "My name is Foo.";
  ```
  수정가능 / 수정불가능

  로 선언되어 있을 때, 각각의 경우 `i`번째 문자를 엑세스 하는 방법은 다음과 같다:

  `a[i]` 와 `p[i]`, 또는 `*(a + i)` 와 `*(p + i)`

- 앞서 언급했듯이, 위의 경우 `p[i]` 와 `*(p + i)` 의 값은 수정할 수 없다.

- 하지만

  ```
  char* p = a;
  ```
  또는
  ```
  char* p = (char*)malloc(13);
  ```
  와 같이 초기화 되어있을 경우, `*(p + i)` 에 값을 대입하거나 수정할 수 있다.

- 실습: 키보드로 입력한 문자열에서 스페이스 (`' '`) 의 개수를 반환하는 함수를 만들어보자.
  ```

```c
#include <stdio.h>
int space(char *str){
        int i, n;
        for(i=0, n=0; str[i] ; i++){
                if(str[i] == ' ')       n++;
        }
        return n;
}
int main(void){
        char a[32];
        printf("Enter a sentence: ");
        gets(a);
        printf("%d spaces\n", space(a));
        return 0;
}
```

문자열 관련 라이브러리 사용하기

- C에서는 문자열과 관련된 다양한 라이브러리들을 제공하고 있다.

- 이 라이브러리들은 **string.h** 에 선언되어 있다. 따라서 문자열 관련 함수를 사용하려면 프로그램 처음에 다음과 같이 써 줘야 한다.

```
#include <string.h>
```

- 본 수업에서 다룰 문자열 관련 함수는 다음과 같다.

strcpy() copy
strlen() length
strcat() concatenation : 이어붙이는것
strcmp() compare

strcpy() 함수 String copy

- **strcpy** 함수는 다음과 같이 선언되어 있다:

```
char* strcpy(char* s1, const char* s2);
```

타겟 / 소스

constant : s2 내용을 변화시키지 않겠다

- **strcpy**의 역할은 문자열 **s2**를 문자열 **s1**으로 복사하는 것이다.
- 이 때, **s2**에서 **'\0'**이 등장하기 직전까지의 문자열을 복사한다.
- **strcpy**는 **s1**값을 반환한다.

- 사용 예:

```
char st1[4], st2[5];
strcpy(st1, "abc");  // st1[0], st1[1], st1[2], st1[3]
become 'a', 'b', 'c', '\0', respectively
strcpy(st2, st1);    // st2[0], st2[1], st2[2], st2[3]
become 'a', 'b', 'c', '\0', respectively
st2[2] = 'd';
printf("%s, %s\n", st1, st2);
```

abc, abd 가 출력된다.

strcpy() 함수

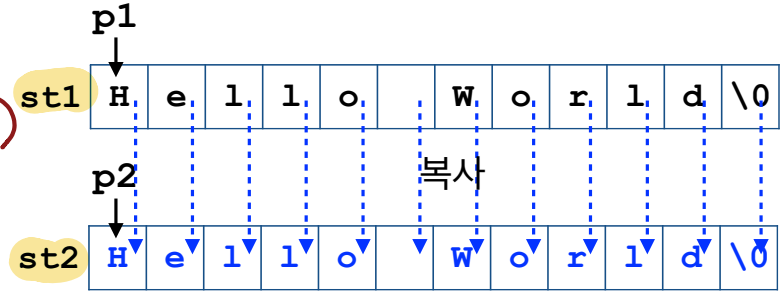
- 다음과 같은 소스코드를 생각하자.

```
char st1[12] = "Hello World", st2[12];
char *p1 = st1, *p2 = st2;
```

- 위의 상황에서 다음 소스코드를 실행하면,

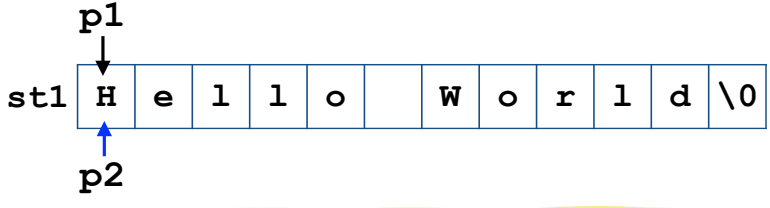
```
strcpy(p2, p1);
printf("%s", p2);
```

Hello World 가 출력된다.



- 하지만, 다시 맨 위의 상황에서 다음 소스코드를 실행해도, 같은 결과가 나온다.

```
p2 = p1;
printf("%s", p2);
```



- 위 두 상황은 같은 현상처럼 보여도, 첫 번째 코드는 배열 **st1**에 있는 문자들이 배열 **st2**에 복사된 것이고, 두 번째 코드는 **p1**과 **p2**가 둘 다 **st1[0]**을 가리키게 된 것이므로 메모리에서는 다른 일이 일어난 것이다.

strcpy(), strncpy() 함수

- 다음 소스코드를 보자:

```
char str[3];
strcpy(str, "abcdef");
```

- **strcpy** 함수는 문자열의 선언된 길이를 측정하지 않기 때문에 위와 같이 코딩해도 컴파일 에러가 나지는 않는다. 하지만, 심각한 오류가 발생할 수 있다.

stack smashing

- 위의 문제를 해결하기 위하여 **strcpy** 함수 대신 **strncpy** 함수를 사용하면 좀더 안전하다.

- 사용 방법은 다음과 같다:

```
strncpy(st1, st2, [복사할 문자열 길이]);
```

- 예를 들어 다음과 같이 사용하면 된다.

```
strncpy(st1, st2, sizeof(st1));
```

strlen() 함수

- 다음과 같이 선언되어 있다.

return type : int라고생각

```
size_t strlen(const char *s);
```

- 문자열 s의 길이를 반환하는 함수. 이 때, 최초로 '\0'이 등장하기 직전까지 문자열의 길이를 나타낸다.

```
int len;
char st1[10], st2[10];
strcpy(st1, "abcdefg");
strcpy(st2, st1);
len = strlen(st1);  // len becomes 7
len = strlen(st2);  // len becomes 7
st1[4] = '\0';
len = strlen(st1);  // len becomes 4
len = strlen(st2);  // len becomes 7
```

strcat() 함수 : 일렬로 연결

- **strcat()** 함수는 다음과 같이 선언되어 있다.

```
char* strcat(char* s1, const char* s2);
```

- 이 함수는 기존의 문자열 **s1**에 문자열 **s2**의 내용을 연결시킨다.
- 이 때, **s2**의 내용 중 최초로 '\0'이 등장하기 직전까지의 문자열 내용이 연결된다.
- 이 함수는 **s1**을 반환한다.

```
char st1[10], st2[10];
strcpy(st1, "abc");     // st1 becomes "abc"
strcat(st1, "def");     // st1 becomes "abcdef"
strcpy(st1, "abc");     // st1 becomes "abc"
strcpy(st2, "def");     // st2 becomes "def"
strcat(st1, st2);       // st1 becomes "abcdef"

strcpy(st1, "abc");
strcpy(st2, "def");
strcat(st1, strcat(st2, "ghi")); // st2 becomes "defghi",
st1 becomes "abcdefghi"
```

```c
#include <stdio.h>
#include <string.h>
int main(void){
        char st1[10], st2[10];
        strcpy(st1, "abc");
        printf("st1: %s\n", st1);
        strcat(st1, "def");
        printf("st1: %s\n", st1);
        strcpy(st1, "abc");
        printf("st1: %s\n", st1);
        strcpy(st2, "def");
        strcat(st1, st2);
        printf("st1: %s\n", st1);
        return 0;
}
```

```
s1234567@whistle:~/lecture$ ./str
st1: abc
st1: abcdef
st1: abc
st1: abcdef
```

```c
strcpy(st1, "abc");
strcpy(st2, "def");
strcat(st1, strcat(st2, "ghi"));
return 0;
```

이걸추가하면

```
st1: abcdefghi
st2: defghi
```

strcmp() 함수

- **strcmp()** 함수는 다음과 같이 선언되어 있다.

```
int strcmp(const char* s1, const char* s2);
```

- 이 함수는 두 문자열 **s1**, **s2**에서 최초로 등장하는 '\0' 이전의 문자열들을 비교하여, 완전히 같으면 0을 반환, 다르면 1 또는 -1을 반환한다.

사전순서상 뒤에까지먼저는1, 앞이먼저는-1

- 사전식으로 배열했을 때, 문자열 **s2**가 문자열 **s1**보다 먼저 오면 1을, **s1**이 **s2**보다 먼저 오면 -1을 반환한다.

- 문자의 사전 배열 순서는 아스키 코드 순서를 따른다.

```
int a;
a = strcmp("ABC", "ABC");      // a is now 0
a = strcmp("XY", "XYZ");       // a is now -1
a = strcmp("abd", "abcd")      // a is now 1
```

```c
#include <stdio.h>
#include <string.h>
int main(void){
        char s1[10], s2[10];
        strcpy(s1, "abcdefg");
        strcpy(s2, "abcdEFG");
        printf("%d\n", strcmp(s1, s2));
        return 0;
}
```

사전도서상 뒤에꺼가 먼저 -1이 출력

근데 32가 출력됨

gcc에서는 둘의 차를 계산하는듯

```c
if(strcmp(s1, s2) == 0){
        printf("Same sentences!\n");
}
else{
        printf("Different!\n");
}
```

이렇게많이씀

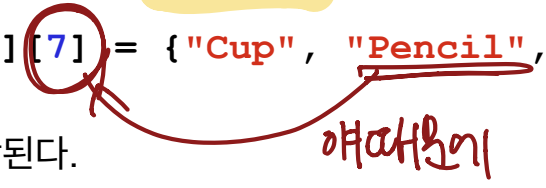
```c
if(!strcmp(s1, s2)){
        printf("Same sentences!\n");
}
else{
        printf("Different!\n");
}
```

(축은이렇게)

문자열의 배열 → 이차원

- 문자열의 배열을 저장하기 위해서는 이차원 배열을 사용한다. 다음과 같은 선언을 보자.

```
char items[3][7] = {"Cup", "Pencil", "Desk"};
```



- 다음과 같이 메모리에 저장된다.

주소	저장된 값
...	...
DB4C3200	'C'
DB4C3201	'u'
DB4C3202	'p'
DB4C3203	'\0'
DB4C3204	'\0'
DB4C3205	'\0'
DB4C3206	'\0'

주소	저장된 값
DB4C3207	'P'
DB4C3208	'e'
DB4C3209	'n'
DB4C320A	'c'
DB4C320B	'i'
DB4C320C	'l'
DB4C320D	'\0'
DB4C320E	'D'

주소	저장된 값
DB4C320F	'e'
DB4C3210	's'
DB4C3211	'k'
DB4C3212	'\0'
DB4C3213	'\0'
DB4C3214	'\0'
...	...

- 이렇게 메모리에 저장되는 것은 저장공간 낭비이다.

문자열의 배열

이차원배열인지 명시X

- 다음과 같이 선언되었을 때를 보자.

```
char *item[] = {"Cup", "Pencil", "Desk"};
```

- 그러면 다음과 같이 메모리에 저장된다.

주소	저장된 값
...	...
DB4C3200	'C'
DB4C3201	'u'
DB4C3202	'p'
DB4C3203	'\0'
DB4C3204	'P'
DB4C3205	'e'
DB4C3206	'n'

주소	저장된 값
DB4C3207	'c'
DB4C3208	'i'
DB4C3209	'l'
DB4C320A	'\0'
DB4C320B	'D'
DB4C320C	'e'
DB4C320D	's'
DB4C320E	'k'

주소	저장된 값
DB4C320F	'\0'
...	...

- 이렇게 선언되면 메모리 공간은 절약된다.
- 변수 사용 방법은 **item[i][j]** 이다.

실행할 때 인자 넣기

- gcc 에서 **src.c**라는 소스코드를 다음과 같이 컴파일하면 **run** 이라는 실행파일이 만들어진다.

```
gcc -o run src.c
```

- C Programming에서 **main**함수를 다음과 같이 정의할 수 있다.

```
int main(int argc, char *argv[]){
...
}
```

argument

- 실행할 때,

```
./run abcd ef ghijk
```

- 와 같이 입력하면 **argc**에는 **4**가, **argv[0]**, **argv[1]**, **argv[2]**, **argv[3]**는 각각 **"./run"**, **"abcd"**, **"ef"**, **"ghijk"**를 가리킨다.

- 일반적으로 **argc**에는 입력한 요소의 개수가 저장되고, ***argv[]**에는 입력한 문자들이 저장된다.
- 메모리에 문자열이 저장되는 방식은 바로 앞 페이지와 같다.

```c
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[]){
		int i, n;

		while(1){
			if (argv[1]==NULL){
				printf("No argument.\n");
				break;
			}
			else{
				for (i=1; i<argc ; i++){
					n=strlen(argv[i]);
					printf("arg%d: %s, %d\n", i, argv[i], n);
				}
				break;
			}
		}
		return 0;
}
```

```
s2016133@whistle:~/exer11$ ./run
No argument.
s2016133@whistle:~/exer11$ ./run hi
arg1: hi, 2
s2016133@whistle:~/exer11$ ./run hi hello
arg1: hi, 2
arg2: hello, 5
s2016133@whistle:~/exer11$ ./run Howdy, my name is Jae.
arg1: Howdy,, 6
arg2: my, 2
arg3: name, 4
arg4: is, 2
arg5: Jae., 4
s2016133@whistle:~/exer11$
```

실행할 때 인자 넣기

- Visual studio에서도 인자를 넣을 수 있다.

- 프로젝트 -> 속성 -> 디버깅 -> 명령 인수 에서 인자를 실행파일 이름을 빼고 white space 로 구분하여 넣으면 된다. 즉,

`abcd ef ghijk`

과 같이 입력하면 된다.