문자열의 표현과 길이

2020년 12월 2일 수요일 오전 4:47

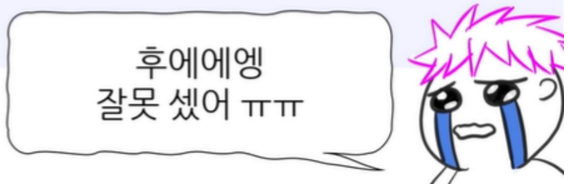
- 기본 자료형은 크기와 범위가 고정이 됐었음
- 하지만 문자열은 크기를 고정할 수 없다
  - pizza는 6바이트 Alle Vongole는 13바이트로 가지각색
- 그래서 컴퓨터에게 "문자열 하나 읽어" 하면 어쩔 줄 모름
  - 이게 문자열이 기본 자료형이 아닌 이유
- 문자열은 기본적으로 여러 개의 문자(char)가 모인 것
  - 그래서 문자열은 char 배열
- 문자열(배열)이 메모리에 저장될 때는 배열의 길이가 따로 저장이 안 됨
  - 그래서 프로그래머가 그 길이를 따로 기억해 둬야 함

문자열 관리 시 길이의 문제

- 길이를 저장하는 변수를 따로 만들때 생기는 실수

```
void print_string(void)
{
    char chars[] = "I want to be the best C programmer";
    const size_t NUM_CHARS = 33;

    size_t i;
    for (i = 0; i < NUM_CHARS; ++i) {
        printf("%c", chars[i]);
    }
}
```

```
I want to be the best C programme
```



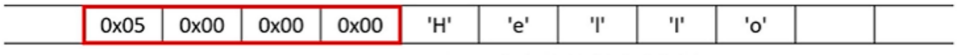
  - 변수에 잘못된 길이를 넣었을 때 생기는 문제

문자열 길이 문제 해결 방법1: 길이를 배열 첫 위치에 저장

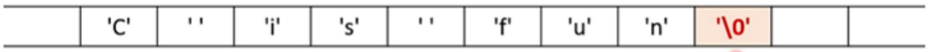
  - 첫 메모리 위치에 문자열 길이를 저장하고
  - 실제 문자열이 뒤 따라오게 함
  - 길이는 int로 저장하고 그 뒤에 char로 문자들을 저장
    - unsigned char은 최대 255글자로 길이를 저장하긴 너무 짧음

- 해결 방법 1의 장점
  - 첫 주소를 보는 것만으로도 총 글자 수가 몇인지 앎
  - 다른 언어에서 문자열의 크기를 바로 알 수 있던 이유이기도 한다
    - C#에서 실제로 이런 방식으로 문자열의 길이를 저장해 둠
- 해결 방법 1의 단점
  - 글자 1개가 들어간 문자열에 4바이트를 더 써야 함
    - 용량 낭비가 될 수 있다
  - 순수 C 코드로 이것을 어떻게 작성해야 할지 애매함
    - 일단 char 배열에 바이너리 패턴을 모두 때려 넣은 다음
      첫 데이터는 int*로 캐스팅해서 읽고 그 다음부터는 char*로 읽어야 함

```
char array[9]; /* 길이 5와 문자열 "Hello"가 적혀있음 */

int* len = (int*)&array[0];
char* str = &array[4];
```



- 문자열 길이 문제 해결 방법2: 문자열이 끝나는 위치를 표시
  - 그냥 char[] 만 쓰되 그 문자열이 끝나는 위치에 널 문자(null character)를 붙임
    - 배열에서 값을 찾을 수 없으면 존재할 수 없는 색인 -1을 반환하는 방식과 마찬가지
    - 널 문자는 '₩0'으로 아스키코드로 0이다.
    - C 스타일 문자열, null terminated string이라 부름



메모리(매우 단순화시킨 예)

  - C 스타일 문자열 선언하기
    - `char str1[] = "abc";`
      - 스택에 "abc" 저장
      - 함수 안에서 바꿔야 될 문자열을 선언할 때
    - `char* str2 = "abc";`
      - 데이터 섹션에 "abc" 저장
      - 바꾸지 않은 문자열을 선언할때
      - 앞에 const를 붙여주자
    - `char str[] = { 'a', 'b', 'c' };`
      - 이 경우에는 '₩0'을 넣어주지 않음
  - C 스타일 문자열의 장단점
    - 장점
      - 가장 최소한의 메모리를 사용
      - 한 가지 데이터형으로 문자열과 길이를 다 표현
    - 단점
      - 어떤 문자열의 길이를 알려면 배열을 끝까지 훑어야 함
        - 처음부터 끝까지 한번 훑어야 하니까 O(N)

문자열 길이 구하기

- 문자열 길이 구하기 개념
  1. char 배열의 요소를 처음부터 차례대로 읽는다
  2. 널 문자를 만나면 멈춘다
  3. 여태까지 총 몇 개의 char를 방문했는지 그 카운터를 반환
- 문자열 길이 구하는 함수

```
size_t get_string_length(const char* str)
{
    size_t i;
    for (i = 0; str[i] != '\0'; ++i) {
    }

    return i;
}
```

  - *p++가 아니라 배열첨자를 사용하면
    반복문의 매 회차마다 시작주소 + i 만큼 점프
    - 조금은 비효율적이다
- 좀 더 효율적인 문자열 길이 구하는 함수

코드 1

```
size_t get_string_length(const char* str)
{
    const char* p = str;

    while (*p++ != '\0') {
    }

    return p - str - 1;
}
```

코드 2

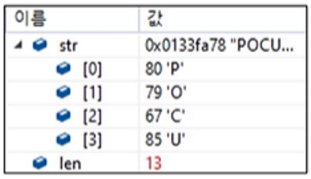
```
size_t get_string_length(const char* str)
{
    size_t count = 0;
    const char* p = str;

    while (*p++ != '\0') {
        ++count;
    }

    return count;
}
```

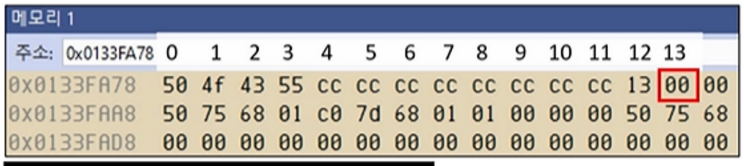
- 코드1
  - 포인터(주소)끼리 빼면 그 사이에 몇 개가 있는지 나옴
    - p가 널문자까지 이동하고 반복문을 탈출했으므로 1을 더 빼준다
- 코드2
  - count라는 변수를 따로 두고 p의 주소가 한칸씩 이동할 때마다 count를 증가시킴
    - p가 널문자까지 이동하고 count가 증감하기전에 반복문을 탈출함(1을 빼줄 필요 없음)
    - count 변수가 있어 좀더 명확함

문자열 길이를 구하는 함수 strlen()

- `size_t strlen(const char* str);`
  - 위에 작성한 코드와 거의 동일하다
  - <string.h>를 인클루드하면 사용 가능
  - 이 외에 다른 함수들도 있음
    - 모두 직접 작성할 줄 알아야 한다
- 가끔 하는 실수

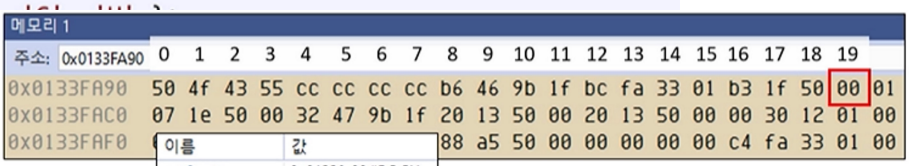
```
#define BUFFER_LENGTH (4)

char str[BUFFER_LENGTH];
size_t len;

str[0] = 'P';
str[1] = 'O';
str[2] = 'C';
str[3] = 'U';

len = strlen(str);
printf("%d\n", len);
```

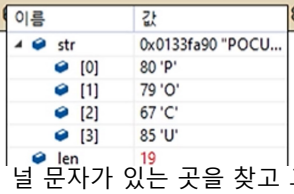


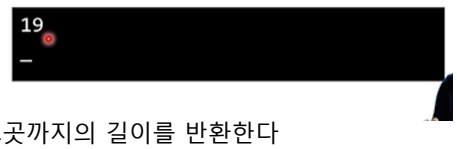


  - 배열의 길이는 문자열의 길이 +1 이여야 함
  - 위같은 실수를 막기 위해 배열 선언할 때 무조건 원소의 수를 +1 하기도 한다
- 가끔 하는 실수 2

```
char str[] = { 'P', 'O',
size_t len;

len = strlen(str);
printf("%d\n", len);
```



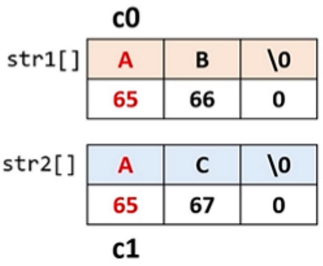




  - 널 문자가 없는 char 배열에 문자열 함수를 사용해서 나타나는 문제
  - 문자열 다음에 남의 메모리가 있는 곳에서 널 문자가 있는 곳을 찾고 그곳까지의 길이를 반환한다
    - 만약에 메모리에 널 문자가 없는거면 길이도 없는거다..
- 안전하지 않으므로 C11의 strlen_s() 함수를 사용하기도 한다
  - strlen(str)은 그냥 읽는거니 안전하다?
    - 나중에라도 널문자가 나오지 않을 수도 있다
    - 하드웨어가 보호하는 메모리에 접근한다면 아예 뻑이 남
    - 읽는 것이어도 소유하지않은 메모리에 접근하는 건 위험한 일임

두 문자열 비교 함수

- 사전식 순서(lexicographical order)로 어떤 문자의 아스키코드가 더 작냐/같냐/크냐를 판별
  - 사전식 순서 예
    - ABCD는 ABCE보다 작음
      - 사전에서 더 먼저 나온다는 거 (아스키 코드가 작다는 것)
    - abcd는 ABCD보다 큼
      - 소문자가 대문자보다 아스키 코드가 큼
    - ABC는 ABCEFG보다 작음
      - "ABC"에 숨겨진 널 문자(₩0)와 E를 비교
    - abcd는 abcd와 같음
    - 좌항 > 우항, 좌항 < 우항, 좌항 = 우항
      이렇게 3가지 경우가 있다
      - 그래서 두 문자열을 비교하는 함수의 반환형은 음수, 양수, 0이 나올 수 있게 int로
- 문자열 비교 함수 알고리듬
  1. 두 문자열에서 문자를 하나씩(c0, c1) 읽음
  2. 두 문자를 비교
     i. c0이 c1보다 작으면 음수를 반환
     ii. c0이 c1보다 크면 양수를 반환
     iii. (c0과 c1이 같고) 널 문자면 0을 반환
  3. 다음 문자로 이동 후 1번 단계로 돌아감



- 두 문자열을 비교하는 함수
  - 방법1

방법 1

```
int compare_string(const char* str0, const char* str1)
{
    while (*str0 != '\0' && *str0 == *str1) {
        ++str0;
        ++str1;
    }

    return *str0 - *str1;
}
```

    - 첫번째 문자가 널이 아니고, 두 문자가 같으면 계속 진행(다음 문자를 읽음)
      - 반복문을 나올때는 첫번째 문자가 널문자거나, 두 문자가 달라서 나옴
        - 나와서 두 문자 차가 양수이면 첫 번째 문자열이 크고
          차가 음수라면 두 번째 문자열이 크다
          차가 0이면 둘다 널문자라는 것, 두 문자열이 같다는 것
          (두 문자가 널일때만 조건을 불만족해서 나올 수 있음)
  - 방법2

방법 2

```
int compare_string(const char* str0, const char* str1)
{
    /* 방법 1의 while 문과 같은 코드 생략 */

    if (*str0 == *str1) {
        return 0;
    }

    return *str0 > *str1 ? 1 : -1;
}
```

    - 두 문자열이 같다면 0 반환
    - 좌항이크다면 1 반환, 우항이크다면 -1 반환
- ABC가 ABCDE보다 작은 이유를 모르는 경우
  - 널문자와 D를 비교한다는걸 모를 때 함수를 이렇게 작성할 수 밖에 없음

```c
int compare_string(const char* str0, const char* str1) {
{
    size_t i;
    size_t len0 = strlen(str0);
    size_t len1 = strlen(str1);

    if (len0 != len1) {
        for (i = 0; i < min(len0, len1); ++i) {
            /* 달라지면 str0[i] - str1[i] 반환 */
        }
        /* len0, len1에 따라 -1 또는 1 반환 */
    }

    for (i = 0; i < len0; ++i) {
        /* 달라지면 str0[i] - str1[i] 반환 */
    }
    return 0;
}
```

  - strlen은 for문을 이용해서 작성이 됨
  - for문은 4번이나 쓰는 비효율적인 코드가 됨
- strcmp( )
  - `int strcmp(const char* ihs, const char* rhs);`
  - 문자열 비교 함수
  - <string.h>에 있음
- strncmp( )
  - `int strncmp(const char* ihs, const char* rhs, size_t count);`
  - 최대 n 문자까지만 비교
  - 위에 코드에서 종료조건이 하나 추가될 뿐

문자열 복사

- 코드 보기

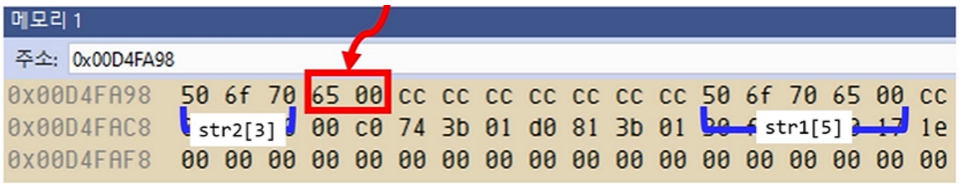
```c
void copy_string(char* dest, const char* src)
{
    while (*src != '\0') {
        *dest++ = *src++;
    }

    *dest = '\0';
}

/* 다른 함수 */
const char* str1 = "Pope";
char str2[5];

copy_string(str2, str1);
```

  - dest는 const가 아님
  - 문자열 길이를 전달할 필요 없음
    - 널 문자까지 복사하면 됨
- 문자열 복사: strcpy( )
  - `char* strcpy(char* dest, const char* src);`
  - <string.h>에 있음
  - 반환값 char*는 dest를 반환
    - 왜 그런지는 모르겠다고 함
    - 그래서 실제로 아무도 안 씀
  - C11에서 나온 strcpy_s( )는 errno_t를 반환
    - `errno_t strcpy_s(char *restrict dest, rsize_t destsz, const char *restrict stc);`
      - errno_t는 에러넘버 타입
        - 정수형임
        - strcpy_s()함수는 내부적으로 문제가 있으면 에러 코드를 반환

- strcpy( )를 사용하는데 dest가 src보다 짧으면? (dest < src)

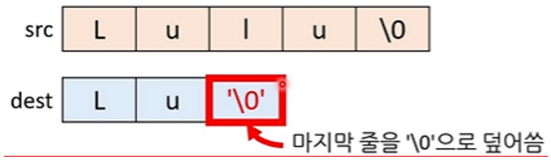
```
const char* str1 = "Pope";
char str2[3];

string_copy(str2, str1);
```

  - str2[3]의 범위를 넘어서서 계속 복사를 함



    - 이렇게 남의 메모리에 쓰는 건 문제
    - 그래서 외부로부터 문자열이 들어오는 경우에는 strcpy( )를 쓰지 않는게 좋음
      - src와 dest의 크기를 확실하게 통제 가능하다면 안전함
    - C11에서는 이보다 안전한 strcpy_s( )라는 함수가 나옴
    - C89에서는 비교적 안전한 strncpy( )를 사용함

- 비교적 안전한 문자열 복사: strncpy( )
  - `char* strncpy(char* dest, const char* src, size_t count);`
    - 최대 count만큼 복사
    - 널 문자를 먼저 만나면 그전에 끝냄
    - src가 count보다 짧거나 같으면
      - 남는 걸 다 0(₩0)으로 채워줌
    - src가 count보다 길다면
      - count만큼 복사함
      - **널 문자를 안 붙여줌**
  - strncpy( )의 일반적인 사용법
    - `strncpy(dest, src, DEST_SIZE);`
      `dest[DEST_SIZE - 1] = '\0';`
      1. dest의 최대길이를 지정해줌(DEST_SIZE)
      2. 복사할 때 DEST_SIZE만큼만 복사를 함
      3. **그리고 dest 마지막에 널문자를 습관처럼 넣어줌**
      - strcpy( )에서도 위처럼 dest마지막에 널 문자를 습관처럼 넣어주자
      - strlen( )을 써서 dest에 널 문자가 있고 없고를 판단할 바에 그냥 습관처럼 붙여주는게 낫다



- 정리

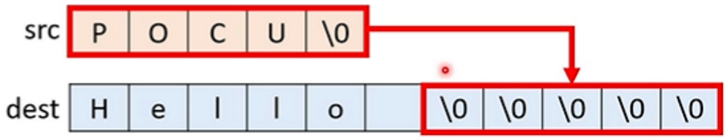
strcpy()	strncpy()

• 위험할 수 있는 함수
• dest 크기 < src 크기
  ○ 잘못된 메모리 쓰기 발생
• 두 크기를 확실히 통제 가능하다면 안전
	• strcpy()보다 안전
	• 덜 빠름
	  ○ dest의 남은 요소를 0으로 채우기 때문
	• 여전히 위험한 경우가 있음!
	  ○ count보다 src가 길 경우
	  ▪ 다 복사하고 널 문자가 없음
	  ▪ 프로그래머가 널 문자를 붙여줘야 함

문자열 합치기

- strcat( )
  - `char* strcat(char* dest, const char* src);`

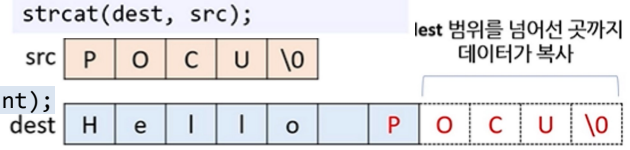
```
const char* src = "POCU";
char dest[11] = "Hello ";

strcat(dest, src);
```



    - <string.h> 안에 있음
    - src의 문자열을 dest 뒤에 덧붙이는 함수
      - dest의 널 문자가 들어있는 위치부터 src의 문자열 추가

        - src의 문자열을 dest 뒤에 덧붙이는 함수
            - dest의 널 문자가 들어있는 위치부터 src의 문자열 추가
                - dest의 널 문자가 src[0]으로 교체
        - dest의 길이가 충분해야 함
            - 이 길이를 넘어 쓸 경우 undefined behavior 발생

```
const char* src = "POCU";
char dest[7] = "Hello ";

strcat(dest, src);
```



- strncat( )
    - `char* strncat(char* dest, const char* src, size_t count);`
    - <string.h> 안에 있음
    - 최대 count개 만큼 src 문자열을 dest 뒤에 덧붙이는 함수
    - dest의 널 문자가 들어있는 위치부터 src의 문자열 추가
    - count 개의 문자를 복사한 뒤, **널 문자를 가장 마지막에 붙여줌**
        - 따라서, 최대 count + 1 개의 문자를 덮어씀

    - dest의 길이보다 길게 쓰면 마찬가지로 undefined behavior 발생
        - 그러나 count로 조금은 방지 가능

        - dest의 최대길이 - (현재 들어가 있는 숫자 + 널 문자)
            - 근데 strlen( )은 for문이니까 이렇게 하는 것 보다
              dest안에 몇 개가 들어갔는지 기억하는 size_t형 변수를 하나 더 만드는게 좋을 듯
- 정리

문자열 찾기

- `char* strstr(const char* str, const char* substr);`
    - <string.h> 안에 있음
    - 반환값: char 포인터
        - substr이 str에 있다면: 해당 substr이 시작하는 주소를 반환
        - substr이 str에 없다면: 널 포인터(NULL)를 반환

    - 문자열 msg은 const char가 아닌 char로 선언해줬다
        - 어차피 char로 반환되니깐
    - 직접 함수 작성해보기 strlen쓰고서
- 왜 메모리 주소를 돌려주느냐?
    - (반환값) - (문자열 주소) 해서 문자열의 색인을 만들 수도 있고
    - 메모리 관리 측면에서 효율적임
        - 새로 문자열을 만들어 반환하지 않음
    - 만약에 반환값으로 새 문자열을 반환하려면?
        - 메모리 '어딘가'에 그 문자열을 복사해야 함
        - 복사하는 위치가 스택이면 함수가 끝나면 사라짐
            - 유효하지 않은 주소가 됨

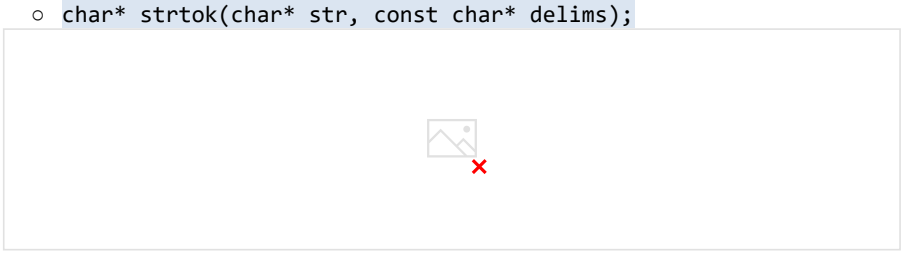
        - 복사하는 위치가 힙이면(동적 메모리 할당)
            - 메모리 할당을 운영체제에게 부탁해야 하므로 느림
            - 실수로 프로그래머가 메모리 해제 함수를 호출 안할 수 있음
                - free( )

                - 매니지드 언어에서는 OS가 메모리 해제를 해준다
    - 그래서 그냥 원본에서 찾고자 하는 문자열이 시작하는 주소를 반환하는게 가장 나음
        - 추가적으로 메모리를 쓰지도 않고, 실수도 줄일 수 있음

문자열 토큰화

- C#에서는 문자열을 토큰화 하면 char배열로 줬었음
- C에서는 기존 문자열을 수정함
    - 그래서 스택 메모리에 복사된 문자열을 사용
- 구분 문자를 널 문자로 바꿔줌

- 정리
  - `char* strtok(char* str, const char* delims);`



  - 토큰화를 시작하려면 문자열(msg)를 strtok( )에 넣음
  - 그 msg의 다음 토큰을 구하려면 대신 NULL
  - 더 이상 토큰이 없다면 strtok( )은 NULL을 반환
  - msg는 "Hi₩0 there₩0 Hello₩0 Bye₩0" 이렇게 바뀜
- 토큰화 함수에서 두 가지 알 수 있는 점
  1. 토큰화하는 문자열은 const가 아니라 원본이 바뀜
     - 따라서 스택 메모리에 복사된 문자열을 넣어야 겠지?
  2. 함수 매개변수로 NULL이 들어올 때 그 전에 받았던 msg를 사용하니
     이건 어딘가에 저장되어 있어야 함
     - 함수 내 정적(static)변수가 제일 적합
- 토큰화 함수 시간이 걸리더라도 혼자 작성해보기!

C 문자열 함수들의 특징

- <string.h>에 있는 문자열 함수들
  - strlen()
  - strcmp() / strncmp()
  - strcpy() / strncpy()
  - strstr()
  - strcat() / strncat()
  - strtok()
  - 그외 다수
  - , strcmp, strcpy들은 많이 쓴다
- C 문자열 함수들의 특징
  1. 꽤 많은 함수들이 문자열을 절대 변경하지 않는다
     - 그래서 매개변수에 const char* 가 붙어 있음
  2. 문자열을 변경하더라도 원본은 변경 안하려 함
     - 사본만 변경
       - 사본, 타겟, dest
     - 예외: strtok()
       - 원본을 지키려면 호출하는 함수에서 사본을 만든 뒤 strtok()을 호출해야 함
  3. 절대 새로운 문자열을 만들어 주지 않는다