한빛아카데미 Hanbit Academy, Inc.

C 언어 프로그래밍

Part 03. C 언어 심화 학습

Chapter 11. 포인터 심화

목차

학습목표

- **문자열을 포인터로 표현하는 방법을 살펴본다.**
- **배열 매개변수와 포인터 매개변수의 관계를 살펴본다.**
- **동적 메모리의 필요성과 할당, 해제 방법을 살펴본다.**

01

포인터와 배열

01. 포인터와 배열

I. 문자열 포인터

[코드 11-1] 문자열 포인터

```
01  #include <stdio.h>
02
03  int main( )
04  {
05      char name[16] = "Kim Do Hyung";
06      printf("%s\r\n", name);
07
08      char* pName = "Kim Na In";
09      printf("%s", pName);
10  }
```

```
Kim Do Hyung
Kim Na In
```

01. 포인터와 배열

I. 문자열 포인터

- **문자열 포인터와 배열의 관계**
  - C 언어에서는 큰따옴표로 감싸진 리터럴 문자열은 배열로 취급함
  - 배열이 값으로써 초기화나 대입에 사용될 때는 배열의 첫 번째 요소를 가리키는 포인터가 됨

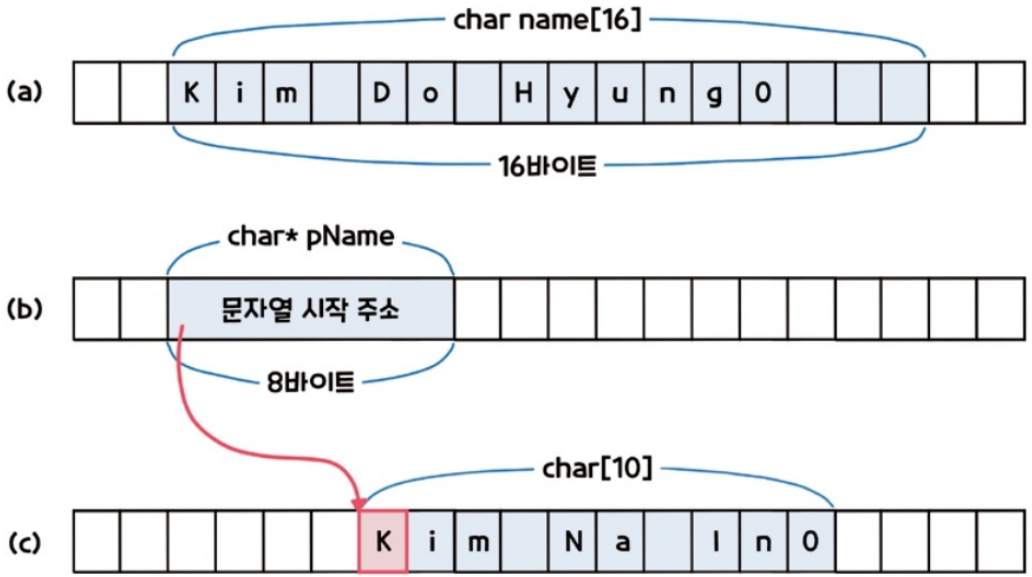


그림 11-1 문자열 포인터

01. 포인터와 배열

II. 배열을 가리키는 포인터

- **대상 타입이 배열 타입인 포인터**
  - 배열에 참조(&) 연산자를 적용할 경우
    - » 배열 자체를 가리키는 포인터를 구할 수 있음

[코드 11-2] &배열명

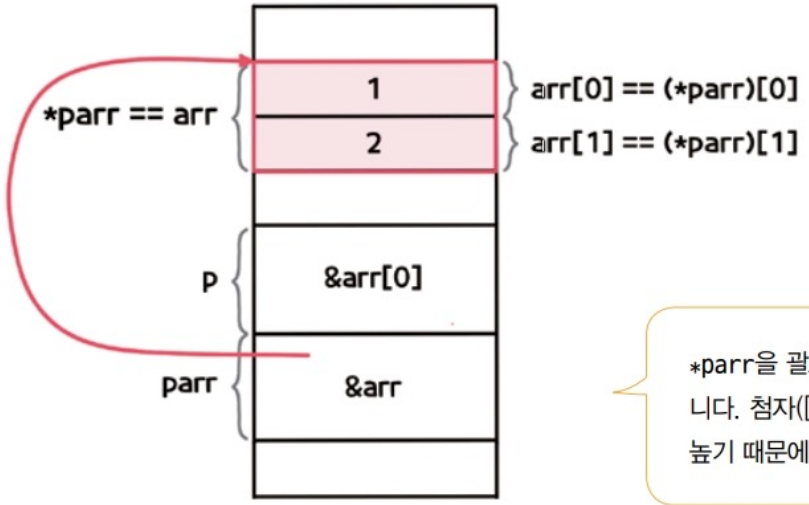
```
01  #include <stdio.h>
02
03  int main( )
04  {
05      int arr[2] = { 1, 2 };
06
07      int* p = &arr;                  // 경고
08      printf("%d, %d\r\n", p[0], p[1]);
09
10      int (*parr)[2] = &arr;          // 문제 없음
11      printf("%d, %d", (*parr)[0], (*parr)[1]);
12  }
```

```
1, 2
1, 2
```

01. 포인터와 배열

II. 배열을 가리키는 포인터

- 대상 타입이 배열 타입인 포인터



*parr을 괄호로 감싼 이유는 연산자의 우선순위 때문입니다. 첨자([ ]) 연산자의 우선순위가 간접(*) 연산자보다 높기 때문에 괄호를 이용하여 적용 순서를 바꿔줍니다.

그림 11-2 배열 전체를 가리키는 포인터

01. 포인터와 배열

II. 배열을 가리키는 포인터

- **대상 타입이 배열 타입인 포인터**

하나 더 알기 **배열을 가리키는 포인터의 정의 형식**

- 배열을 가리키는 포인터를 정의할 경우 먼저 배열을 정의한 후에 배열 이름 앞에 기호 *를 붙인 후 이름과 기호 *를 괄호로 감싸면 됨

```
int parr[2] → int *parr[2] → int (*parr)[2]
```

01. 포인터와 배열

확인문제1

1. 다음 빈칸에 들어갈 단어를 채우시오.

큰따옴표로 묶인 문자열 리터럴은 실제로 메모리 어느 곳에 문자 [ ]처럼 저장되어 있으므로, 문자열 리터럴이 값(초기값, 대입 값, 실인자)으로써 사용될 경우 첫 번째 요소(문자)를 가리키는 [ ]가 된다.

2. `int arr[4];`일 때, `arr`과 `&arr`의 차이는 무엇인지 설명하시오.

01. 포인터와 배열

LAB 11-1 문자열의 길이를 반환하는 함수 만들기

strlen 함수는 문자열을 받아서 길이를 반환합니다. 같은 기능을 하는 mystrlen 함수를 구현해봅시다. 단, 매개변수 str은 NULL 종료 문자열을 가리키는 포인터입니다.

```
int strlen(const char* str);
```

str은 NULL 종료 문자열을 가리키는 포인터로 대상 타입이 const char이므로 문자를 하나씩 가리킬 수가 있다. 따라서 대상 주소를 1씩 증가시키면서 문자 배열의 요소가 NULL인 것을 찾을 때 인덱스를 반환하면 길이가 된다.

LAB 11-1 정답

```
01  int mystrlen(const char* str)
02  {
03     for(int i = 0; ; i++)
04        if(str[i] == '\0')
05           return i;
06  }
```

02

포인터와 함수

02. 포인터와 함수

I. 포인터 매개변수

- 참조 전달과 비슷하게 인자를 전달하기

[코드 11-3] 포인터 매개변수

```
01  #include <stdio.h>
02
03  void func(int* p)
04  {
05      *p = 1;
06  }
07
08  int main( )
09  {
10      int a = 0;
11      func(&a);
12
13      printf("%d", a);
14  }
```

```
1
```

02. 포인터와 함수

I. 포인터 매개변수

- **INOUT 매개변수**
  - 변수가 입출력 모두에 이용될 수 있는 것

[코드 11-4] INOUT 매개변수

```
01  #include <stdio.h>
02
03  void rounds(int* p)
04  {
05      *p = (*p + 5) / 10 * 10;
06  }
07
08  int main( )
09  {
10      int a = 7;
11      printf("%d 반올림: ", a);
12
13      rounds(&a);
14      printf("%d", a);
15  }
```

```
7 반올림: 10
```

02. 포인터와 함수

II. 배열 매개변수와 포인터의 관계

- 참조 전달처럼 동작하는 배열 매개변수

[코드 11-5] 배열 실인자, 매개변수의 크기 출력

```
01  #include <stdio.h>
02
03  void func(int arg[16])
04  {
05      printf("arg size: %d", sizeof(arg));
06  }
07
08  int main( )
09  {
10      int arr[16];
11      printf("arr size: %d\r\n", sizeof(arr));
12      func(arr);
13  }
```

```
arr size: 64
arg size: 8
```

II. 배열 매개변수와 포인터의 관계

- **배열과 호환되는 포인터 매개변수**
  - Fake Techniques : 배열 매개변수가 가능한 것처럼 보이게 하는 눈속임

그림 11-3 배열 매개변수에서 포인터 매개변수로 변경

II. 배열 매개변수와 포인터의 관계

- **배열 매개변수의 주의할 점**
  - 배열 매개변수가 포인터 매개변수로 변경될 때 배열 요소의 개수는 무시됨

[코드 11-6] 다른 타입 배열 전달과 주의점

```
01  #include <stdio.h>
02
03  void func(int arg[16])
04  {
05      arg[7] = 1;     // 문제 없음
06      arg[8] = 2;     // 위험: 선언은 8개 요소를 했으나 매개변수가 16개!!
07  }
08
09  int main( )
10  {
11      int arr[8];
12      func(arr);
13  }
```

02. 포인터와 함수

II. 배열 매개변수와 포인터의 관계

- **배열 매개변수의 주의할 점**
  - 배열 매개변수를 표시할 때 잘못된 사용을 줄이기 위하여 배열 요소의 개수 자체를 생략하는 경우가 대부분임

```
void func(int arg[16]) → void func(int arg[])
```

02. 포인터와 함수

II. 배열 매개변수와 포인터의 관계

하나 더 알기 2차원 배열 매개변수

[코드 11-7] 2차원 배열 매개변수

```
01  void func(int arg[2][3])
02  {
03  }
04
05  int main( )
06  {
07      int arr[2][3] = { 0 };
08      func(arr);
09  }
```

- 2차원 배열 매개변수 또한 2차원 배열과 호환되는 포인터로 변경됨

```
int arg[2][3] → int (*arg)[3]
```

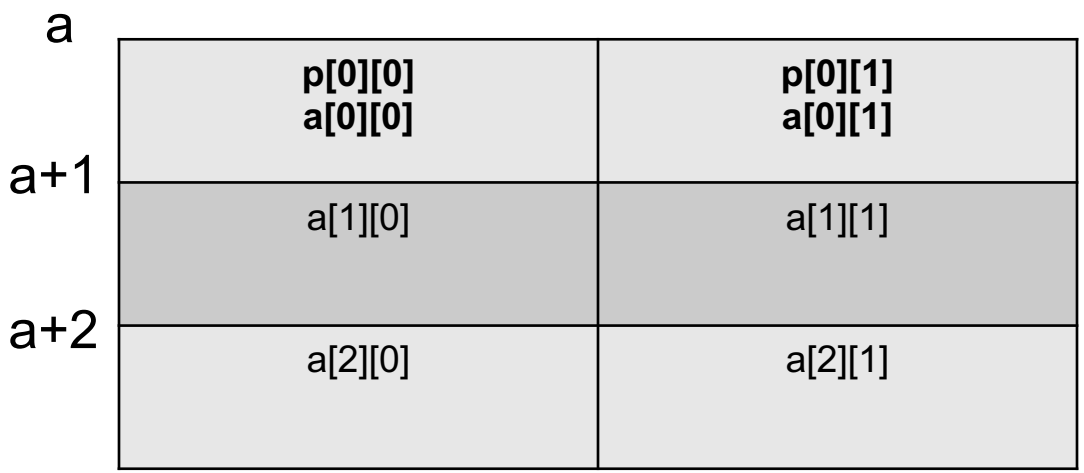
- N차원 배열 매개변수가 컴파일러에 의해서 자동으로 변경되는 호환 포인터 매개변수 형식

```
타입 arg[C1][C2]...[Cn] → 타입 (*arg)[C2]...[Cn]
```

02. 포인터와 함수

II. 배열 매개변수와 포인터의 관계

Int a[3][2];
int (*p)[2] = a; // int 형의 주소를 2개씩 점프하는 pointer p를 선언!



02. 포인터와 함수

확인문제2

1. 다음 빈칸에 들어갈 단어를 채우시오.

참조 전달이란 실인자로 [ ]의 값을 넘기는 것이 아니라 [ ] 자체를 넘기는 방식을 말한다. C 언어는 참조 전달을 지원하지 않지만 [ ] 매개변수를 이용하면 참조 전달과 비슷한 효과를 낼 수 있다.

2. 다음 빈칸에 들어갈 단어를 채우시오.

C 언어에서 배열 매개변수는 지원되지 않으며 대신 배열과 호환되는 [ ]로 매개변수가 변경된다. 변경된 매개변수는 길이가 다른 배열도 받을 수 있으므로 호환성이 증가하지만 인덱스를 잘못 사용할 경우 위험할 수도 있다.

02. 포인터와 함수

LAB 11-2 **배열의 요소를 오름차순으로 정리하기**

요소 타입이 int인 배열과 요소의 개수를 인자로 받은 뒤 배열의 값들을 오름차순으로 정렬하는 함수를 구현해봅시다. 함수의 형식은 다음과 같습니다.

```
void Sort(int arr[], int count);
```

① 함수의 매개변수가 배열이므로 처리해야 할 요소의 개수도 매개변수로 받아야 한다.

② 첫 번째 요소부터 시작하여 오른쪽 끝까지 진행하면서 가장 작은 요소를 발견하면 두 요소를 교환한다. 이런 식으로 두 번째 요소, 세 번째 요소 순으로 차례차례 가장 작은 요소와 교환하는 작업을 반복하면 오름차순 정렬이 이루어진다.

LAB 11-2 정답

```
01  void Sort(int arr[], int count)
02  {
03     for(int i = 0; i < count - 1; i++)
04        for (int j = i + 1; j < count; j++)
05           if(arr[i] > arr[j])
06           {
07              int temp = arr[i];
08              arr[i] = arr[j];
09              arr[j] = temp;
10           }
11  }
```

03

동적 메모리 할당과 해제

03. 동적 메모리 할당과 해제

I. 힙 영역

- **힙(Heap) 영역의 개념**
  - 메모리의 크기가 필요에 따라서 변할 수 있는 새로운 형태의 메모리 영역

- **메모리 영역의 할당과 해제**
  - 프로그램이 시작되면 기본적으로 힙(Heap) 영역이 마련되어 필요한 만큼 메모리 영역을 할당받아 사용할 수 있음
  - 힙에서 할당받은 메모리 영역이 더 이상 사용되지 않을 경우 반드시 해제해야 함

03. 동적 메모리 할당과 해제

I. 힙 영역

- **힙 매니저**
  - 프로그램이 동적 메모리 할당과 해제를 쉽게 사용할 수 있도록 함
  - 요청 크기의 메모리를 할당해줌
  - 할당받은 메모리는 명시적으로 해제하기 전까지 안전하게 유지됨

03. 동적 메모리 할당과 해제

II. malloc 함수와 free 함수

- **동적 메모리 할당 및 해제 함수**
  - malloc 함수 : 매개변수 size 바이트만큼 메모리 영역을 힙에서 할당받는 함수

```
void* malloc(size_t size);
```

하나 더 알기 **malloc 매개변수 타입 size_t**

- size_t는 typedef로 새롭게 정의된 타입임

```
typedef unsigned __int64 size_t;
→ size_t는 unsigned __int64로서 8바이트 부호 없는 정수 타입입니다.
```

03. 동적 메모리 할당과 해제

II. malloc 함수와 free 함수

- **동적 메모리 할당 및 해제 함수**
  - free 함수 : 힙에서 할당받은 메모리 영역을 해제하는 함수

```
void free(void* ptr);
```

[코드 11-8] malloc & free

```
01  #include <stdio.h>
02  #include <malloc.h>
03
04  int main( )
05  {
06      int* p = malloc(4);
07      *p = 1;
08      printf("%d", *p);
09      free(p);
10  }
```

```
1
```

03. 동적 메모리 할당과 해제

II. malloc 함수와 free 함수

- **메모리 할당과 해제 과정**
  - 힙 매니저는 내부적으로 할당된 메모리 영역의 정보를 관리하여 할당받은 메모리 영역의 크기를 기억함

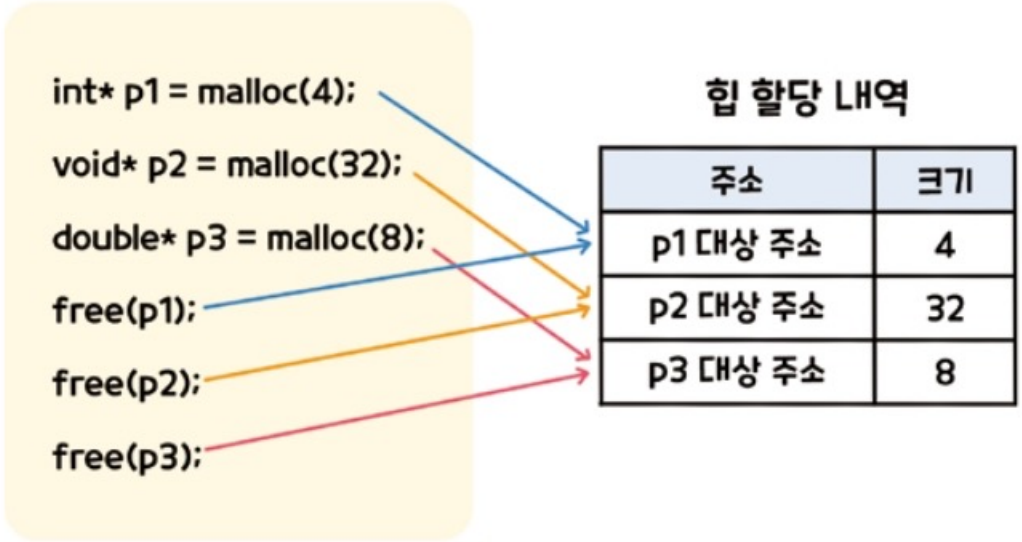


그림 11-4 힙 매니저 할당 내역

03. 동적 메모리 할당과 해제

II. malloc 함수와 free 함수

- **동적 메모리 할당의 장점**
  - 할당된 메모리 영역을 마치 전역, 정적 객체처럼 함수 경계를 넘어서 사용 가능함

[코드 11-9] 함수 경계를 벗어난 사용

```
01  #include <stdio.h>
02  #include <malloc.h>
03
04  int* GetStack( )
05  {
06      int a = 1;
07      return &a;
08  }
09
10  int* GetHeap( )
11  {
12
13      int* p = malloc(sizeof(int));
14      *p = 2;
15      return p;
16  }
17
18  int main( )
19  {
20      int* p1 = GetStack();
21      int* p2 = GetHeap();
22      printf("%d, %d", *p1, *p2);
23      free(p2);
24  }
```

03. 동적 메모리 할당과 해제

II. malloc 함수와 free 함수

- **동적 메모리 할당의 장점**
  - 동적 메모리 할당을 통해서 대용량 메모리를 확보함

[코드 11-10] 대용량 메모리 할당

```
01  #include <malloc.h>
02
03  int main( )
04  {
05   //int arr[1024 * 1024 * 4] = { 0 };    // 런타임 오류
06
07      int size = 1024 * 1024 * 4 * sizeof(int);
08      int* parr = malloc(size);            // 문제 없음
09      free(parr);
10  }
```

03. 동적 메모리 할당과 해제

III. calloc 함수와 realloc 함수

- **calloc 함수**
  - 힙에서 메모리 영역을 할당해주는 함수
  - malloc에 비해서 매개변수가 하나 더 있음
  - 할당된 영역의 초기화가 있음
  - 메모리 할당과 0 초기화를 한 번에 해결해주는 함수

```
void* calloc(size_t num, size_t size);
```

03. 동적 메모리 할당과 해제

III. calloc 함수와 realloc 함수

- calloc 함수

[코드 11-11] malloc과 calloc의 비교

```
01  #include <stdio.h>
02  #include <malloc.h>
03
04  int main( )
05  {
06      int* p1 = malloc(sizeof(int));
07      int* p2 = calloc(1, sizeof(int));
08
09      printf("%d, %d", *p1, *p2);
10
11      free(p1);
12      free(p2);
13  }
```

```
67, 0
```

03. 동적 메모리 할당과 해제

III. calloc 함수와 realloc 함수

- **realloc 함수**
  - 이미 할당된 메모리 영역을 재할당하기 위하여 사용함
  - 주로 기존에 할당된 메모리 영역의 크기를 변경할 경우 사용함
  - realloc 호출 후에는 기존 할당된 메모리 영역을 해제해서는 안 됨

03. 동적 메모리 할당과 해제

III. calloc 함수와 realloc 함수

- realloc 함수

[코드 11-12] realloc

```
01  #include <stdio.h>
02  #include <malloc.h>
03
04  int main( )
05  {
06     int* p1 = malloc(sizeof(int));
07     *p1 = 3;
08
09     int* p2 = realloc(p1, sizeof(int) * 4);
10
11     printf("p1: %p, %d\r\n", p1, *p1);
12     printf("p2: %p, %d", p2, *p2);
13
14   //free(p1); //런타임 오류
15     free(p2);
16  }
```

```
p1: 000001E45BDF4CE0, 3
p2: 000001E45BDF8BA0, 3
```

03. 동적 메모리 할당과 해제

확인문제3

1. 다음 빈칸에 들어갈 단어를 채우시오.

   [ ]는 할당 요청을 받을 경우 힙 영역에서 적당한 메모리 영역을 마련하여 해당 영역을 가리키는 [ ]를 돌려준다.

2. 다음 빈칸에 들어갈 단어를 채우시오.

   malloc은 요청된 크기의 메모리 영역을 가리키는 void [ ]를 반환한다. 따라서 반환된 포인터가 가리키는 메모리 영역을 [ ]처럼 사용하기 위해서는 적절한 대상 타입의 포인터 변수에 저장해야 한다.

3. 다음 빈칸에 들어갈 단어를 채우시오.

   calloc은 할당된 메모리 영역을 모두 [ ]으로 초기화하고, realloc은 기존 할당된 메모리 영역을 해제하고 새로운 크기의 메모리 영역을 재할당받으면서 기존 메모리 영역의 값들을 그대로 [ ]한다.

03. 동적 메모리 할당과 해제

LAB 11-3 숫자를 누적시켜 저장하고 보여주기

다음 출력처럼 수가 입력될 때마다 기존에 저장된 수를 모두 보여주는 프로그램을 작성해봅시다. 단, 동적 메모리를 이용하고 필요한 경우 재할당받습니다. 기본 할당 크기는 int 2개이며 메모리가 부족한 경우 int를 2개씩 늘립니다. 입력하는 수가 0인 경우 프로그램을 종료합니다.

```
저장할 수를 입력하세요 > 1
1
저장할 수를 입력하세요 > 3
1 3
저장할 수를 입력하세요 > 5
1 3 5
저장할 수를 입력하세요 > 0
```

1. 동적 메모리 할당 및 해제를 위해서 헤더 파일 malloc.h를 포함한다.
2. malloc을 호출하여 기본 크기의 메모리 영역을 할당받는다.
3. 저장할 공간이 부족할 경우 realloc을 통하여 재할당받는다.
4. 수가 입력될 때마다 할당된 메모리 공간에 저장하고 기존 저장된 수를 출력한다.
5. 더 이상 동적 메모리가 사용되지 않을 경우 free를 호출하여 해제한다.

03. 동적 메모리 할당과 해제

LAB 11-3 정답

```
01  #define _CRT_SECURE_NO_WARNINGS
02  #include <stdio.h>
03  #include <malloc.h>
04
05  int main( )
06  {
07     int index = 0;
08     int size = 2;
09     int* p = malloc(size * sizeof(int));
10
11     while(1)
12     {
13        if(index >= size)
14        {
15           size += 2;
16           p = realloc(p, size * sizeof(int));
17        }
18
```

03. 동적 메모리 할당과 해제

LAB 11-3 정답

```
19              int n;
20              printf("저장할 수를 입력하세요 > ");
21              scanf("%d", &n);
22
23              if(n)
24              {
25                  p[index++] = n;
26                  for(int i = 0; i < index; i++)
27                      printf("%d ", p[i]);
28
29                  printf("\r\n");
30              }
31              else
32                  break;
33          }
34
35      free(p);
36  }
```

[실전예제]

두 문자열을 하나의 문자열로 연결하기

[실전예제] 두 문자열을 하나의 문자열로 연결하기

[문제]

인자로 입력받은 두 문자열을 연결하여 하나의 문자열로 만들어서 반환하는 함수를 작성해봅시다. 함수 형식은 다음과 같습니다. 다음 실행 결과는 두 문자열로 '난생처음'과 'C 프로그래밍'이 전달된 경우입니다.

```
char* CatenateString(const char* str1, const char* str2);
```

실행 결과

```
난생처음 C 프로그래밍
```

[실전예제] 두 문자열을 하나의 문자열로 연결하기

[해결]

1. CatenateString의 두 매개변수 str1과 str2에 대해서 먼저 문자열의 길이를 구한다.

2. '두 길이의 합+1'만큼 malloc을 호출하여 동적 메모리를 할당받는다.

3. 반복문을 통하여 str1과 str2의 문자를 하나씩 동적 메모리 영역에 복사한다.

4. 마지막은 NULL 종료 문자열을 완성하기 위하여 '₩0'을 복사하고 포인터를 반환한다.

[실전예제] 두 문자열을 하나의 문자열로 연결하기

[해결]

```
01  #include <stdio.h>
02  #include <malloc.h>
03
04  char* CatenateString(const char* str1, const char* str2)
05  {
06      int len1 = 0;
07      while(str1[len1] != '\0')
08          len1++;
09
10      int len2 = 0;
11      while(str2[len2] != '\0')
12          len2++;
13
14      int index = 0;
15      char* pStr = malloc(len1 + len2 + 1);
16
17      for(int i = 0; i < len1; i++)
18      {
19          pStr[index++] = str1[i];
```

[실전예제] 두 문자열을 하나의 문자열로 연결하기

[해결]

```
20      }
21
22      for(int i = 0; i < len2; i++)
23      {
24          pStr[index++] = str2[i];
25      }
26
27      pStr[index] = '\0';
28
29      return pStr;
30  }
31
32  int main( )
33  {
34      char* str = CatenateString("난생처음", " C 프로그래밍");
35      printf("%s", str);
36
37      free(str);
38  }
```

Thank you!

한빛아카데미 Hanbit Academy, Inc.