학번 : _____ 성명 : _____.

- 본 강의자료는 과학기술정보통신부 및 정보통신기획평가원에서 지원하는『소프트웨어중심대학』사업의 결과물입니다.
- 본 강의자료는 내용은 전재할 수 없으며, 인용할 때에는 반드시 과학기술정보통신부와 정보통신기획평가원의 '소프트웨어중심대학'의 결과물이라는 출처를 밝혀야 합니다.

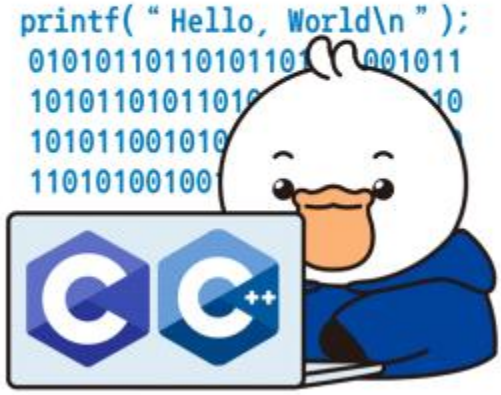
SW 대구가톨릭대학교 소프트웨어중심대학사업단

대구가톨릭대학교 컴퓨터소프트웨어학부 School of Computer Software, Daegu Catholic University

Part 9. 변수와 메모리

School of Computer Software at Daegu Catholic University @ 2021

목차



9.1 지역 변수와 전역 변수

- C 언어의 변수
  - ✓ [ ]과 동시에 초기화하여 사용하는 것이 바람직
    - ➢ 초기화를 하지 않는 경우 쓰레기 값 저장 (개발환경 및 컴파일러에 따라 다름)
      - 이전 프로그램에서 해당 [ ] 영역을 사용한 흔적이 쓰레기 값
        - 이 값을 사용하는 경우 의도와 동떨어진 결과의 원인이 될 가능성 존재
  - ✓ 지역 변수란?
    - ➢ 해당 함수 또는 블록 내에서 사용하는 변수
      - 지금까지 사용한 변수는 모두 [ ]동 변수(automatic variable) 또는 자동 지역 변수
    - ➢ `int a;`와 같이 선언해서 사용했으나 이는 `auto int a;`
      - 자주 사용하는 형태이기에 `auto` 키워드 생략하고 사용
      - 변수 `a`는 자동 변수이자 지역 변수이므로 자동 지역 변수
        - 자동 변수와 지역 변수, 자동 지역 변수는 같은 뜻
    - ➢ 메모리의 [ ] 영역에 저장

9.1 지역 변수와 전역 변수

소스 9-1

```
1  #include <stdio.h>
2  int main(void) {
3     auto int a = 10;
4     auto int b;
5     printf("%d %d", a, b);
6     return 0;
7  }
```

출력 예 1 (Visual Studio)	초기화되지 않은 메모리 'b'를 사용하고 있습니다. 초기화되지 않은 'b' 지역 변수를 사용했습니다.
출력 예 2 (gcc 계열)	10 0
출력 예 3 (clang 계열)	10 32767

Visul Studio – 경고 메시지 출력

gcc 계열 - 자동으로 0으로 초기화

clang 계열 – 쓰레기 값 저장

9.1 지역 변수와 전역 변수

소스 9-2

```
1  #include <stdio.h>
2  int add(int a, int b) {
3     return a+b;
4  }
5  int main(void) {
6     int a = 10;
7     int b = 20;
8     int result = add(a, b);
9     printf("%d", result);
10    return 0;
11 }
```

서로 다른 지역에서 선언된 변수이므로
값이 같더라도 서로 다른 변수

출력 예	30

9.1 지역 변수와 전역 변수

- C언어의 변수
  - ✓ 전역 변수란?
    - ➢ 지역 변수와 대비되는 개념의 변수
      - 함수나 블록 [ ] 부에 선언되어 프로그램 전체에서 사용 가능
    - ➢ 전역 변수는 `main()` 함수가 실행되기 전에 자동 생성되어 초기화
      - 프로그램 [ ] 시점에 생성, 프로그램 [ ] 시점에 소멸
      - 초기화는 각 데이터 타입의 `0`에 해당하는 기본값
        - 정수형: `0`
        - 문자형: `null`을 의미하는 `'\0'`
        - 실수형: `0.0`
        - 포인터: `nil`
    - ➢ 메모리 [ ] 영역에 저장

9.1 지역 변수와 전역 변수

소스 9-3

```
1  #include <stdio.h>
2  int g_int;
3  float g_float;
4  double g_double;
5  char g_char;
6  int *g_pointer;
7  int main(void) {
8     printf("%d, %f, %lf, ", g_int, g_float, g_double);
9     printf("%c, %p\n", g_char, g_pointer);
10    return 0;
11 }
```

출력 예 1 (Visual Studio)	0, 0.000000, 0.000000, , 00000000
출력 예 2	0, 0.000000, 0.000000, , (nil)

Int *형, 출력이 다르지만 동일한 의미

9.1 지역 변수와 전역 변수

소스 9-4

```c
#include <stdio.h>
int gNo = 10;
int add(int lNo) {
   return lNo + gNo;
}
int main() {
   int lNo = 80;
   int gNo = 20;
   printf("%d\n", lNo + gNo);
   printf("%d\n", add(lNo));
   return 0;
}
```

출력 예	100 90

9.1 지역 변수와 전역 변수

- C언어의 변수
  - ✓ 외부 변수란?
    - ➢ 다른 소스코드 파일에 선언된 [ ] 변수를 사용하고자 하는 경우에 사용
      - extern 키워드 이용 다른 소스코드 파일에서 선언된 [ ] 변수를 사용

main.c

```c
#include <stdio.h>
extern double pi;
int main() {
   int radius = 5;
   double result = 5 * pi;
}
```

myMath.c

```c
#include <math.h>
double pi=3.14
```

9.1 지역 변수와 전역 변수

소스 9-5a (global.c)

```
1 #include <stdio.h>
2 int gNo1 = 10;
3 int gNo2 = 90;
```

소스 9-6b (main.c)

```
1  #include <stdio.h>
2  extern int gNo1;
3  extern int gNo2;
4  int add(int gNo1, int gNo2) {
5     return gNo1 + gNo2;
6  }
7  int main() {
8     printf("%d", add(gNo1, gNo2));
9     return 0;
10 }
```

출력 예	100

9.2 정적 변수와 레지스터 변수

- 지금까지 다룬 변수들
  - ✓ [ ] 일정 공간을 할당 받아 사용
  - ✓ 연산을 위해 CPU 내부 레지스터(register)로 복사되어 연산 수행, 저장이 필요한 경우 다시 레지스터 값을 메모리에 저장
  - ✓ 메모리를 읽어 레지스터로 복사하는 과정과 다시 레지스터에서 변경된 값을 메모리에 쓰는 과정 필요
    - ➢ 대부분의 CPU는 일반적으로 메모리 접근에 비해 보다 [ ] 속도로 동작
  - ✓ CPU와 메모리 사이 존재하는 메모리 버스(bus) 이용 데이터 [ ] 처리
    - ➢ 매우 빠른 속도로 처리되나 CPU 처리 속도, 버스 크기, 버스 속도 등 다양한 요인으로 인하여 [ ] 현상 발생

9.2 정적 변수와 레지스터 변수

- 레지스터 변수란?
  - ✓ 메모리가 아닌 레지스터 공간에 할당하여 사용하는 변수
  - ✓ [ ]접근이 이루어지지 않기 때문에[ ]내부에서 빠른 연산을 통해 프로그램의 성능에 도움
  - ✓ `register` 키워드를 이용하여 선언

<일반적인 변수> <레지스터 변수>

9.2 정적 변수와 레지스터 변수

소스 9-7

```
1  #include <stdio.h>
2  int main() {
3    register int sum = 0;
4    register int i = 0;
5    register int j = 0;
6    for(i = 0; i < 10000; i++)
7      for(j = 0; j < 10000; j++)
8        sum += i;
9    printf("%d", sum);
10   return 0;
11 }
```

출력 예	1733793664

9.2 정적 변수와 레지스터 변수

소스 9-8

```
1  #include <stdio.h>
2  #include <time.h>
3  int main() {
4     int sumInt = 0, i = 0, j = 0;
5     register int sumRegi = 0, ri = 0,rj = 0;
6     clock_t clk1 = clock(), clk2;
7     for(i = 0; i < 10000; i++)
8        for(j = 0; j < 10000; j++)
9           sumInt += i;
10    printf("int : %dms \n", clock() - clk1);
11    clk2 = clock();
12    for(ri = 0; ri < 10000; ri++)
13       for(rj = 0; rj < 10000; rj++)
14          sumRegi += ri;
15    printf("register int : %dms", clock() - clk2);
16    return 0;
17 }
```

출력 예 1 (local)	int : 132ms register int : 128ms
출력 예 2 (web)	int : 305338ms register int : 62978ms

9.2 정적 변수와 레지스터 변수

- 정적 변수란?
  - ✓ 변수 선언 시 [ ] 키워드를 앞에 추가하여 선언
  - ✓ 선언하는 [ ]에 따라 정적 전역 변수와 정적 지역 변수로 구분
    - ➢ 정적 전역 변수
      - 어떠한 함수나 블록에 속하지 않고 전역 변수로 선언한 경우
      - 일반 전역 변수와는 달리 다른 소스코드 파일에서 [ ] 키워드 이용 공유가 불가능한 전역 변수
    - ➢ 정적 지역 변수
      - 함수 내부 또는 블록 내부에 선언된 지역 변수
      - 다른 지역 변수와는 다르게 함수 또는 블록이 종료되어도 소멸하지 않고 유지
      - 최초 한 번만 초기화되어 두 번 다시 초기화 되지 않음 (반드시 상수를 이용 초기화해야 함)
      - 최초 호출 후 종료시 [ ] 하지 않고 유지되어 [ ] 값을 그대로 유지

9.2 정적 변수와 레지스터 변수

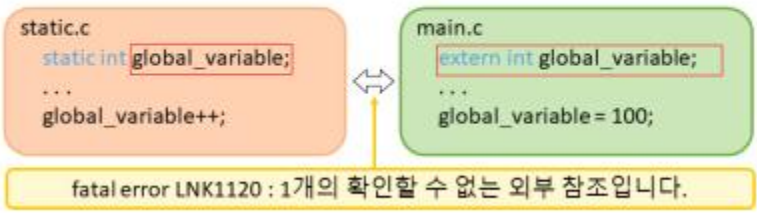
소스 9-9

```
 1 #include <stdio.h>
 2 int function(void) {
 3    int autoLocalVariable = 1;
 4    static int staticLocalVariable = 1;
 5    printf("Auto local variable : %1d\t", autoLocalVariable++);
 6    printf("Static local variable : %1d\n", staticLocalVariable++);
 7 }
 8 int main(void) {
 9    int i;
10    for(i=0; i<3; i++) function();
11    return 0;
12 }
```

출력 예

```
Auto local variable : 1      Static local variable : 1
Auto local variable : 1      Static local variable : 2
Auto local variable : 1      Static local variable : 3
```


9.2 정적 변수와 레지스터 변수



<정적 전역 변수의 외부 변수 선언시의 에러>

<기억 부류를 결정하는 키워드의 종류>

지정자	선언 위치	유효 범위	저장 위치	생성 시기	소멸 시기	초기화
auto	함수 내부	지역	스택 영역	함수 시작	함수 종료	X
extern	함수 외부	전역	데이터 영역	프로그램 시작	프로그램 종료	O
register	함수 내부	지역	레지스터	함수 시작	함수 종료	X
static	함수 내부, 외부	지역, 전역	데이터 영역	프로그램 시작	프로그램 종료	O

9.3 동적 메모리 할당과 해제

- 변수 메모리 할당 방식
  - ✓ 정적 메모리 할당(Static Memory Allocation)
    - ➢ [ ] 단계에서 프로그램에 필요한 메모리 계산, 프로그램 실행 전 필요한 메모리를 할당 받아 사용
    - ➢ 변수와 배열 등 메모리를 사용하는 모든 것들은 다 정적 메모리 할당 방식
    - ➢ 프로그램 시작 전 메모리 공간을 준비하므로 필요 메모리 공간 [ ]필요
      - 계산이 부정확한 겨우 여분은 낭비됨
      - 메모리 낭비 방지를 위해 메모리 [ ] 할당 프로그래밍
        - 메모리 공간 부족으로 데이터가 불가능한 상황 발생 가능성 존재
    - ➢ 안정적으로 필요한 메모리를 모두 할당 받을 수 있음

9.3 동적 메모리 할당과 해제

- 변수 메모리 할당 방식
  - ✓ 동적 메모리 할당 (Dynamic Memory Allocation)
    - ➢ 프로그램 실행 도중 필요한 만큼 메모리를 할당 받아 사용하는 방식
      - 필요한 만큼만 메모리를 사용하여 [ ]이 높음
      - 현재 [ ] 메모리 부족 시 필요한 메모리를 할당 받지 못하는 경우 발생
    - ➢ 메모리를 할당 받기 위한 부가적 프로그램 코드 필요
      - 처음 프로그래밍을 시작하는 사람들에게는 다소 어려울 수 있다.
    - ➢ 최근 프로그래밍 언어는 자동 동적 메모리 할당 기능 제공
      - c 언어는 프로그래머가 [ ] 메모리 관리

동적 (Dynamic) - 실행 중 변화하는 값을 다루는 경우 주로 사용되는 용어

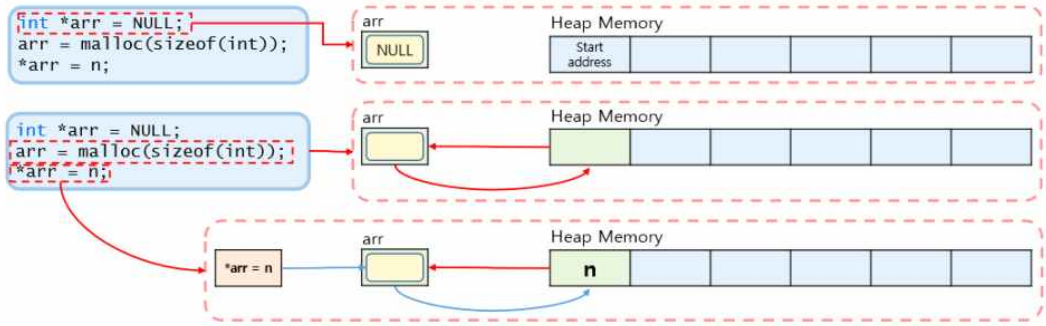
9.3 동적 메모리 할당과 해제

- 변수 메모리 할당 방식
  - ✓ 동적 메모리 할당(Dynamic Memory Allocation)
    - ➢ malloc() 함수 이용 [ ] 영역의 메모리 할당
      - 불필요시 free() 함수 이용 메모리 할당을 해제
        - 해제하지 않는 경우 프로그램이 종료될 때 까지 메모리 [ ]
    - ➢ malloc()과 free() 함수
      - 표준 C 라이브러리 함수로 stdlib.h 헤더 파일에 정의
        - stdlib는 standard library의 약자
    - ➢ 배열과 같이 다수의 데이터 저장 공간 할당 시 주로 사용

9.3 동적 메모리 할당과 해제

<동적 메모리 할당 관련 함수의 종류>

메모리	함수 원형	기능
메모리 할당(기본값 X)	`void * malloc(size_t)`	인자만큼의 메모리 할당 후 기본 주소 반환
메모리 할당(기본값 0)	`void * calloc(size_t, size_t)`	뒤 인자 크기로 앞 인자 수 만큼 할당 후 기본 주소 반환
기존 메모리 변경(이전 값)	`void * realloc(void*, size_t)`	앞 인자의 메모리를 뒤 인자 크기로 변경 후, 기본 주소 반환
메모리 해제	`void free(void*)`	인자를 기본 주소로 갖는 메모리 해제



<malloc() 이용 int형 크기만큼 동적 메모리 할당>

9.3 동적 메모리 할당과 해제

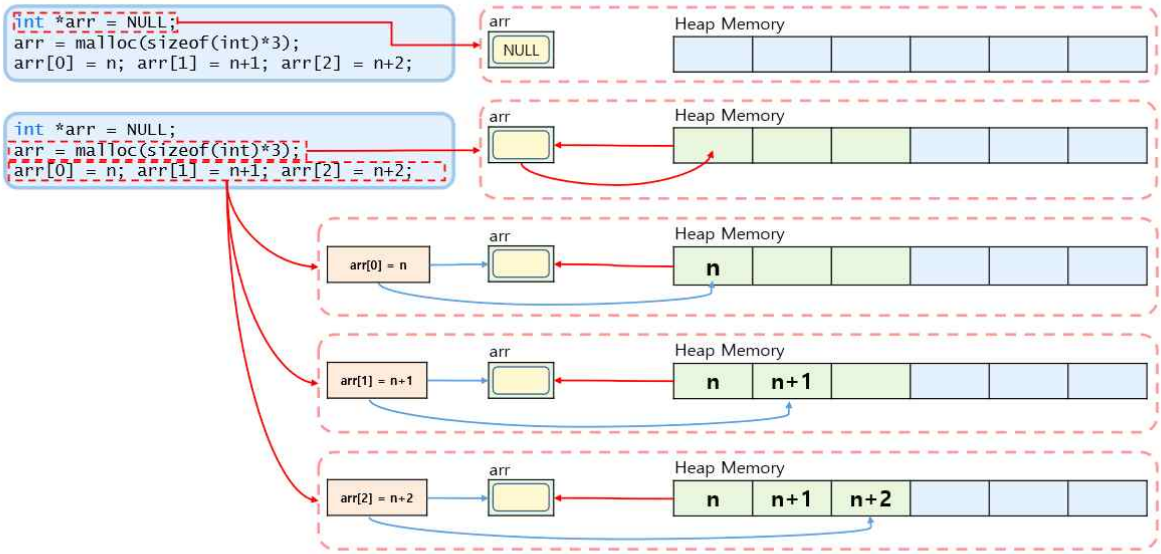
소스 9-10

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
   int *arr = NULL;
   arr = malloc(sizeof(int));
   if(arr == NULL) { //메모리 할당 실패시 동작
      printf("Memory allocation problem!\n");
      return -1;
   }
   *arr = 100;
   printf("Address : %p, Value : %d\n", arr, *arr);
   free(arr); //메모리 할당 해제
   arr = NULL; //포인터 변수는 null을 가리키게 변경
   return 0;
}
```

출력 예

```
Address : 0x6f4260, Value : 100
```


9.3 동적 메모리 할당과 해제



<동작 할당을 통해 배열처럼 사용하는 예>

9.3 동적 메모리 할당과 해제

소스 9-11

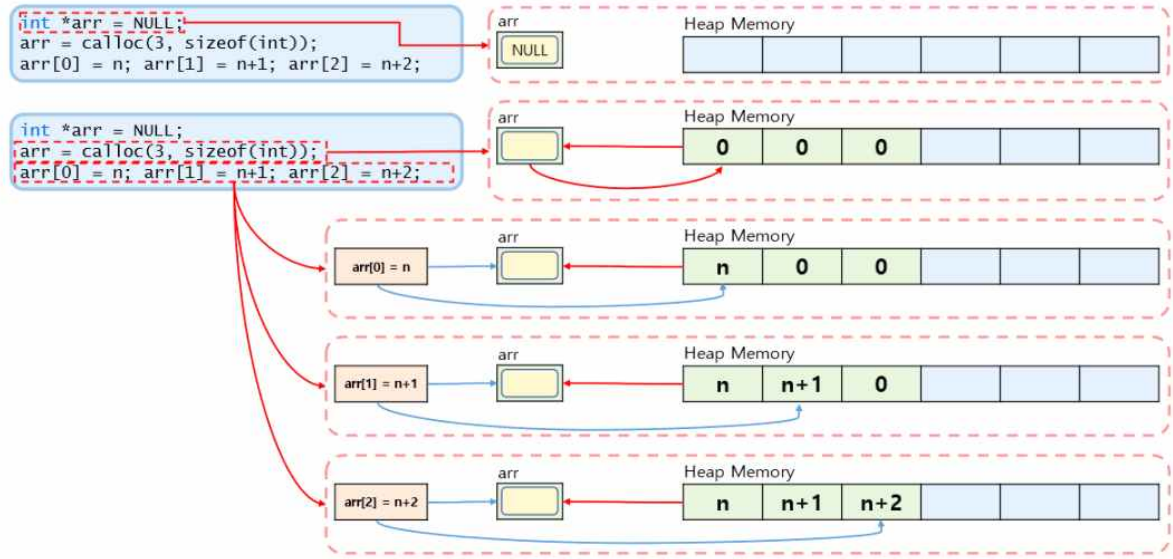
```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    int n=0, sum=0, i;
    int *arr =NULL;
    printf("Input number of score : ");
    scanf("%d", &n);
    if((arr = malloc(sizeof(int)*n)) ==NULL) {
        printf("Memory allocation problem!\n");
        return -1;
    }
    printf("Input %d numbers : ", n);
    for(i=0; i<n; i++){
        scanf("%d", (arr+i));
        sum +=*(arr+i);
    }
    printf("Inputed scores : ");
    for(i =0; i < n; i++){
        printf("%d ", *(arr+i));
    }
    printf("\nSum : %d, Average : %.1f\n", sum, (double)sum/n);
    free(arr);
    arr =NULL;
    return 0;
}
```

입력 예	3 15 89 45
출력 예	Input number of score : 3 Input 3 numbers : 15 89 45 Inputted scores : 15 89 45 Sum : 149, Average : 49.7

9.3 동적 메모리 할당과 해제

- malloc() 함수
  - ✓ 하나의 인자로 할당 메모리 [ ] 결정
  - ✓ 할당받은 메모리 공간에는 [ ] 값 존재
- calloc() 함수
  - ✓ 두 개의 인자로 할당 메모리의 크기 결정
    - ➢ 첫 번째 인자 – 원소의 수
    - ➢ 두 번째 인자 – 원소의 크기
      - 두 번째 인자 크기를 가지는 데이터 공간을 첫 번째 인자의 수만큼 할당
    - ➢ 첫 번째 원소와 두 번째 원소를 곱하면 전체 할당받는 메모리의 크기
  - ✓ [ ]등으로 할당받는 메모리 초기화
- 최근 많은 컴파일러들이 malloc()을 사용하더라도 메모리를 [ ] 해서 할당

9.3 동적 메모리 할당과 해제



<calloc() 함수 이용 동적 메모리 할당>

9.3 동적 메모리 할당과 해제

소스 9-12

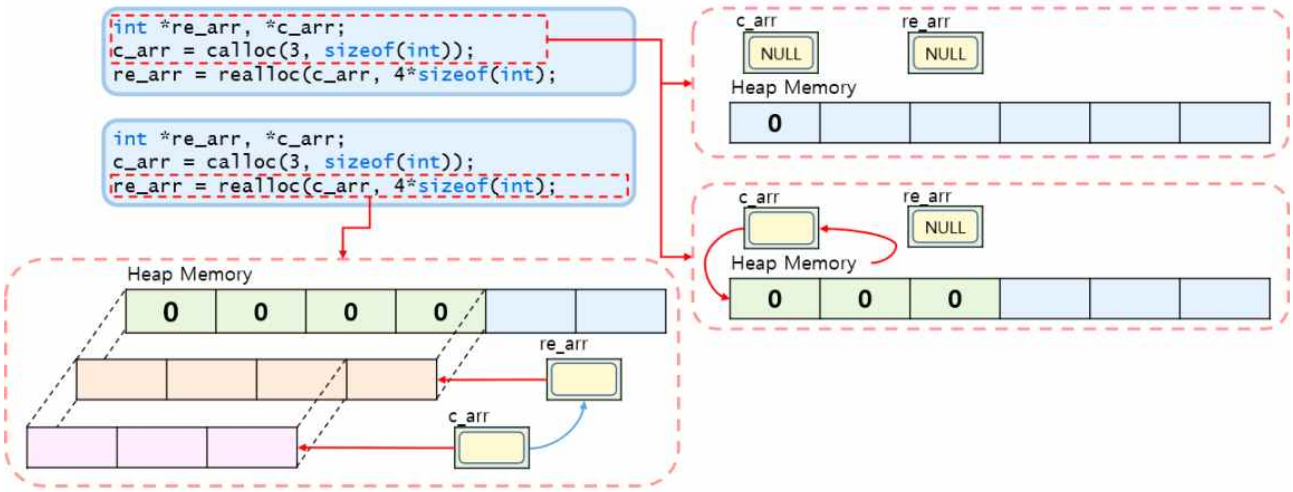
```
1  #include <stdio.h>
2  #include <stdlib.h>
3  void printMem(int *arr, int n){
4      int i;
5      for(i=0; i<n; i++){
6          printf("arr[%d] = %d\n", i, *(arr + i));
7      }
8  }
9  int main(void) {
10     int *arr =NULL;
11     if((arr=calloc(3, sizeof(int))) ==NULL) {
12         printf("Memory allocation problem!");
13         return -1;
14     }
15     printMem(arr, 3);
16     free(arr);
17     arr =NULL;
18     return 0;
19 }
```

출력 예	arr[0] = 0 arr[1] = 0 arr[2] = 0

9.3 동적 메모리 할당과 해제

- realloc() 함수
  - ✓ □적 할당된 메모리 공간에서 새로운 크기의 메모리 공간을 확보하고자 할 때 사용
  - ✓ 이전에 확보된 공간을 새로운 영역에 재할당하여 이전 값들을 복사하는 경우에도 사용 가능
  - ✓ 첫 번째 인자는 변경할 저장공간의 주소
    - ➢ 첫번째 인자가 메모리 주소가 아닌 □인 경우 malloc() 함수와 동일
  - ✓ 두 번째 인자는 변경하고 싶은 메모리 공간의 총 크기
    - ➢ 두 번째 인자의 크기만큼 공간 할당

9.3 동적 메모리 할당과 해제



<realloc() 함수 이용 동적 메모리 할당>

9.3 동적 메모리 할당과 해제

소스 9-13

```c
#include <stdio.h>
#include <stdlib.h>
void printMem(int *arr, int n){
    int i;
    for(i=0; i<n; i++){
        printf("arr[%d] = %d", i, *(arr + i));
    }
    printf("\n");
}
int main(void) {
    int *re_arr, *c_arr;
    if((c_arr=calloc(3, sizeof(int))) ==NULL) {
        printf("Memory allocation problem!");
        return -1;
    }
    printMem(c_arr, 3);
    if((re_arr=realloc(c_arr, sizeof(int)*4)) ==NULL) {
        printf("Memory allocation problem!");
        return -1;
    }
    printMem(re_arr, 4);
    free(re_arr);
    re_arr =NULL;
    return 0;
}
```

출력 예

```
arr[0] = 0 arr[1] = 0 arr[2] = 0
arr[0] = 0 arr[1] = 0 arr[2] = 0 arr[3] = 0
```


9.4 메모리 영역과 변수의 이용

<메모리 영역 그림>

- C 언어의 메모리 영역
  - ✓ 각각 메모리 영역은 변수 유효 [ ](scope)와 [ ](life time)에 결정적인 역할, 기억 부류 (storage class)에 따라 할당되는 메모리 공간의 차이
  - ✓ 코드 영역(code area) : 프로그램을 실행을 위한 실행 코드가 저장되는 영역, [ ] 코드 저장
  - ✓ 데이터 영역(data area) : 전역 변수와 정적 변수가 저정되는 영역
    - ➢ 3 가지 영역으로 세분화
      - rodata(읽기 전용 데이터 영역) : const 상수, 전역 변수, 문자열 상수가 저장
      - data(일반 데이터 영역) : 0이 아닌 값으로 초기화 된 전역 변수 또는 정적 변수를 저장하는 영역, 읽기/쓰기 가능
      - BSS(Block Started Symbol) : 0으로 초기화 되었거나, 초기화되지 않은 전역 변수나 정적 변수 저장
    - ➢ [ ]은 주소에서 [ ]은 주소의 순서로 할당, [ ]적으로 할당되기 때문에 사전에 크기가 결정

9.4 메모리 영역과 변수의 이용

<메모리 영역 그림>

- C 언어의 메모리 영역
  - ✓ 힙 영역 (Heap area)
    - ➢ 동적 할당 메모리 영역, 데이터 영역과 스택 영역 중간에 위치
    - ➢ [ ]은 주소에서 [ ]은 주소 방향으로 공간 사용
  - ✓ 스택 영역 (Stack area)
    - ➢ 함수 호출시 함수에 전달하는 [ ]변수와 함수 내 지역 변수 저장
    - ➢ 함수 호출하는 곳의 프로그램 카운터(PC : Program Counter)와 현재 지역 변수 상태 저장
      - PC는 함수를 호출하고 되돌아올 [ ]영역의 주소
    - ➢ FILO (First In Last Out) : 함수 호출을 순차적으로 저장하여 역순으로 빠져 나올 수 있도록 하는 역할
    - ➢ [ ]은 주소에서 [ ]은 주소 방향으로 데이터 할당
  - ✓ 힙 영역과 스택 영역에는 사용하지 않는 메모리 영역 존재
    - ➢ 해당 공간에서 힙 영역과 스택 영역 충돌 시 메모리 오류 발생 가능성
    - ➢ 최근 [ ]는 이러한 문제를 대부분 해결하고 있어 실제 오류 가능성 희박

9.4 메모리 영역과 변수의 이용

<메모리 영역 그림>

- C 언어의 메모리
  - ✓ 프로그래밍을 할 때 각 기억 부류의 특징을 잘 이해하고 변수를 선언해야 효율적으로 메모리를 관리
    - ➢ 최근 컴퓨터는 충분한 메모리를 가지고 있어 메모리에 대한 고려를 적게 해도 됨
      - 적은 메모리를 가지는 초소형 장치에 프로그램을 동작 시켜야 하는 경우도 존재하므로 효율적 메모리 관리 습관을 가지는 것이 좋음
    - ➢ 가능하면 전역 변수 보다 지역 변수 사용
    - ➢ 많은 횟수의 반복을 빠르게 처리할 필요가 있을 때 [ ] 변수 고려
    - ➢ 프로그램 전체에서 **반드시** 공유되어야 할 값이 있다면 [ ]역 변수 이용
  - ✓ 일반적인 변수와 포인터 변수 등은 동적 할당 방식이 아니라면 [ ] 영역 또는 [ ]영역에 할당
  - ✓ 동적으로 메모리를 할당 받아 사용하는 경우 [ ]영역 사용

9.4 메모리 영역과 변수의 이용

<메모리 영역 그림>

- C 언어의 메모리
  - ✓ 동적 할당 받은 메모리 접근은 정적으로 선언한 [ ] 변수를 이용
    - ➢ 포인터 변수는 주솟값을 가지며 주솟값을 복사하면 다른 포인터에서도 동적 할당 받은 메모리 공간 사용 가능
    - ➢ 하나의 포인터로만 동적 할당 받은 메모리 공간에 접근하고자 한다면 [ ]를 이용
    - ➢ 한정자 : [ ] 키워드를 포인터 변수 선언 시 작성
      - 해당 포인터가 가리키는 주소 공간을 다른 포인터가 접근하지 못하도록 차단
    - ➢ volatile 변수
      - 메모리 접근 [ ] 를 수행하지 않는 변수를 선언하는 것
        - 변수가 레지스터에 복사되어 연산을 통해 값이 변경되어도 메모리에 존재하는 변수 값은 변경되지 않고 레지스터에서만 값이 계속 변경되는 경우를 막는 역할
        - 자주 사용하지는 않지만 상식으로 알아 두도록 하자.

9.5 변수의 유효 범위와 생명 주기

종류	키워드	유효범위	선언위치	기억장소	생존기간
전역 변수	global extern	프로그램 전역	전역	데이터 영역	프로그램의 시작과 종료와 동일
정적 전역 변수	static	파일 내부			
정적 지역 변수	static	함수나 블록 내부	지역		
레지스터 변수	register			레지스터	함수 또는 블록의 시작과 종료와 동일
자동 지역 변수	auto (생략가능)			스택 영역	

<변수의 생명주기와 유효 범위>

9.5 변수의 유효 범위와 생명 주기

<변수의 종류에 따른 메모리 할당 시기와 외부 변수 선언 가능성>

구분	종류	메모리 할당 시기	동일 파일 기준 외부 함수에서 이용	다른 파일 기준 외부 함수에서 이용
전역	전역 변수	프로그램 시작	O	O
	정적 전역 변수	프로그램 시작	O	X
지역	정적 지역 변수	프로그램 시작	X	X
	레지스터 변수	함수 또는 블록 시작	X	X
	자동 지역 변수	함수 또는 블록 시작	X	X

<변수의 종류에 따른 초깃값과 초깃값 할당 시점>

구분	종류	기본 초기값	초기값 할당 시점
전역	전역 변수	정수형(int) = 0 실수형(float,double) = 0.0000000 문자형(char) = '\0' or NULL	프로그램 시작 시
	정적 전역 변수		
지역	정적 지역 변수		
	레지스터 변수	쓰레기 값	함수나 블록 실행 시
	자동 지역 변수		

9.6 Q&A

Q 30. static 함수나 변수를 선언할 때, 항상 static이라고 써 주어야 하나요?

A . 언어 표준에서는 항상 써 줄 것을 요구하지는 않는다. (가장 중요한 것은 첫 선언에 static을 써 주는 것이다.) 함수인가 변수인가에 따라 조금씩 다르다. (게다가 이 문제에 있어서 현존하는 코드는 너무 다양하다.) 따라서 가장 안전한 방법은, 모든 정의와 선언에서 항상 static을 써 주는 것이다.

Q 31. malloc() 함수의 리턴 값이 void*형인데 포인터 변수에 저장할 때 형 변환이 꼭 필요한가?

A . C에서는 void*형은 다른 포인터형으로 언제든지 형 변환이 가능하다. 따라서 아래의 코드는 에러 없이 컴파일된다

```
arr = malloc(sizeof(int)*size);  // malloc의 리턴 값은 arr의 포인터형으로 자동 형 변환된다.
```

위의 코드는 C++에서 컴파일 시 컴파일 에러를 발생시키지만, C에서는 그렇지 않다. C++에서는 포인터형 사이의 형 변환을 C 보다 엄격히 처리하기 때문이다.

9.6 Q&A

Q 32. 해제된 동적 메모리인 허상 포인터의 종류에는 또 어떤 것이 있나?

A. 초기화를 하지 않고 선언하는 포인터 또한 허상 포인터, 허상 포인터는 잘못 사용하면 문제를 발생시킬 수 있기 때문에 반드시 NULL로 초기화

```
int *p; // 초기화 생략에 따른 쓰레기 값 할당
```

다른 경우로는 함수가 지역 변수의 주소를 리턴하면 이 주소는 허상 포인터가 되는 경우이다. 포인터를 반환 함수가 지역 변수의 주소를 반환하면 함수 반환 시 지역 변수가 소멸되므로 함수 호출 쪽에서는 소멸된 변수 주소가 반환 따라서, 포인터 반환하는 함수 작성시 지역 변수 주소를 리턴하지 않도록

9.6 Q&A

Q 33. 전역(global) 함수와 변수를 선언 또는 정의하는 가장 좋은 방법이 무엇일까?

A. 먼저 한 전역 변수는 여러 개의 선언을 가질 수 있지만, 반드시 하나의 정의를 가져야 한다. 전역 변수에서 정의는 공간을 할당하고, 필요하다면 초깃값을 지정하는 일종의 선언이다. 함수에서 선언은 함수 몸체를 제공하는 일종의 선언이다.

```
extern int i;
extern int f();
// extern keyword는 함수 선언에서 option, 아래는 정의의 예이다:)
int i = 0;
int f()
{
return 1;
}
```

9.6 Q&A

Q 33. 전역(global) 함수와 변수를 선언 또는 정의하는 가장 좋은 방법이 무엇일까?

A. 여러 소스 파일에서 변수나 함수를 공유할 필요가 있다면, 당연히 모든 변수와 함수를 일관되게(consistent) 만들어야 한다. 가장 좋은 방법은 각 정의를 관련된 .c 파일에 저장하고, external 선언을 헤더 파일(".h")에 두는 것
선언이 필요한 곳에서는 #include를 써서 포함 시키면 된다. 정의를 포함하는 .c 파일에도 같은 헤더 파일을 포함시켜야 컴파일러가 선언과 정의가 일치하는지 검사해 준다.

9.6 Q&A

Q 33. 전역(global) 함수와 변수를 선언 또는 정의하는 가장 좋은 방법이 무엇일까?

A. 한 헤더 파일에 하나의 선언만 나오도록 하기 위해 다음과 같은 전처리기 트릭을 쓸 수도 있다:

DEFINE(int, i);

그리고 어떤 매크로의 설정에 따라 이 줄이 선언이나 정의가 되도록 할 수 있지만 이는 문제를 유발할 가능성이 많으므로 추천하지 않는다.

컴파일러가 선언이 불일치하는지 검사하기 위해서는 반드시 전역 선언을 헤더 파일에 넣는 것이 중요하다. 특히, external 함수의 prototype을 .c 파일에 넣지 않도록 하기 바란다. 이는 정의와 일치하는지 검사해 주지도 않으며, 만약 정의와 일치하지 않는다면 오히려 쓰지 않는 것보다 못하다.

9.6 Q&A

Q 34. 함수 선언에서 extern이 무엇을 의미하는가?

A. 함수의 정의가 다른 소스 파일에 있을 수 있다는 것을 알려주는 단순한 스타일적인 문제이다. 따라서 다음 두 줄의 차이는 없다:

```
extern int f();
int f();
```


9.7 실습 및 과제

- 실습 및 과제 진행
  - DCU Code : http://code.cu.ac.kr
  - 9 주차 실습
    - 코딩9-1, 코딩9-2, 코딩9-3 (p355-365)
  - 9 주차 과제
    - 9장 프로그래밍 연습 1-6번 (p376-377)

9.8 참고 문헌

- 지역 변수
  - https://ko.wikipedia.org/wiki/지역_변수
- 전역 변수
  - https://ko.wikipedia.org/wiki/전역_변수

E N D