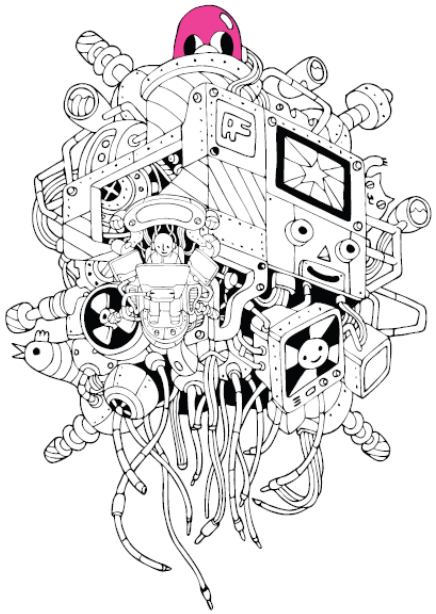
C 프로그래밍



포인터와 함수, 동적할당에 대한 이해

리포트 체크

```
int i = 10;
int *pi = &i;
int j = 3;
char k = 'A';
int z = 0;
char *p = (char*)&i;
int x = 0;

printf( "시작주소 : %p |", p);
printf(" z : %p | ", &z);
printf(" 시작주소 : %p ", (p-60));

for( z = 60,x=0; z>-60; z--,x++ )
{
    printf("%x|", *((unsigned char*)(p-z) ));
    if (x%16==0)
    {
        printf("₩n");
        printf(" 시작주소 : %p ", (p-z));
    }
}
printf("₩n ");
printf("&i: %p | &pi: %p | &j %p |  ₩n &k : %p | &z  %p | &p %p |₩n",
    &i, &pi, &j, &k, &z, &p );
```

```
시작주소 : 0038F814 | z : 0038F7E4 |  시작주소 : 0038F7D8 14|
 시작주소 : 0038F7D8 f8|38|0|cc|cc|cc|cc|cc|cc|cc|cc|30|0|0|0|cc|
 시작주소 : 0038F7E8 cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|41|cc|cc|cc|cc|cc|
 시작주소 : 0038F7F8 cc|cc|cc|3|0|0|0|cc|cc|cc|cc|cc|cc|cc|cc|14|
 시작주소 : 0038F808 f8|38|0|cc|cc|cc|cc|cc|cc|cc|cc|a|0|0|0|cc|
 시작주소 : 0038F818 cc|cc|cc|38|f9|38|0|be|19|34|1|0|0|0|0|0|
 시작주소 : 0038F828 0|0|0|0|e0|fd|7e|cc|cc|cc|cc|cc|cc|cc|cc|cc|
 시작주소 : 0038F838 cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|
 시작주소 : 0038F848 cc|cc|cc|cc|cc|cc|cc|cc|
 &i: 0038F814 | &pi: 0038F808 | &j 0038F7FC |
 &k : 0038F7F3 | &z  0038F7E4 | &p 0038F7D8 |
```

리포트 체크

```
void point2()
{
    char i = 10;
    int j = 3;
    char *pi = &i;
    char k = 'A';
    int z = 30;
    int  arr1[5] = { 1, 2, 3 };
    char arr2[10] = "Hello!";
    char *p2 = pi;
    int x = 0;
    char *p = (char*) &x;

    printf(" 시작주소 : %p ", (p-60));

    for( z = 30,x=0; z>-130; z--,x++ )
    {
        printf("%x|", *((unsigned char*)(p-z) ));
        if (x%16==0)
        {
            printf("₩n");
            printf(" 시작주소 : %p ", (p-z));
        }
    }
    printf("₩n ");
    printf("&i: %p | &pi: %p | &j %p |  ₩n &k : %p | &z  %p | &p %p |₩n &x %p",
        &i, &pi, &j, &k, &z, &p, &x );
}
```

```
시작주소 : 0017F6DE cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|cc|
시작주소 : 0017F6EE cc|fc|f6|17|0|cc|cc|cc|cc|cc|cc|cc|cc|1e|0|0|
시작주소 : 0017F6FE 0|cc|cc|cc|cc|cc|cc|cc|cc|77|f7|17|0|cc|cc|cc|
시작주소 : 0017F70E cc|cc|cc|cc|cc|48|65|6c|6c|6f|21|0|0|0|0|cc|
시작주소 : 0017F71E cc|cc|cc|cc|cc|cc|cc|cc|cc|1|0|0|0|2|0|0|
시작주소 : 0017F72E 0|3|0|0|0|0|0|0|0|0|0|0|0|0|cc|cc|cc|
시작주소 : 0017F73E cc|cc|cc|cc|cc|b8|ff|ff|ff|cc|cc|cc|cc|cc|cc|cc|
시작주소 : 0017F74E cc|cc|cc|cc|41|cc|cc|cc|cc|cc|cc|cc|cc|77|f7|17|
시작주소 : 0017F75E 0|cc|cc|cc|cc|cc|cc|cc|cc|3|0|0|0|cc|cc|cc|
시작주소 : 0017F76E cc|cc|cc|cc|cc|cc|cc|cc|a|cc|cc|cc|cc|23|c|
&i: 0017F777 | &pi: 0017F75C | &j 0017F768 |
&k : 0017F753 | &z  0017F744 | &p 0017F6F0 |
&x 0017F6FC
```

인자전달의 기본방식은 값의 복사이다!

```
int SimpleFunc(int num) { . . . . }
int main(void)
{
    int age=17;
    SimpleFunc(age);
    . . . .
}
```

age에 저장된 값이 매개변수 num에 복사가 된다.

실제 전달되는 것은 age가 아니라 age에 저장된 값이다.

배열을 함수의 매개변수에 전달하는 이유는 함수 내에서 배열에 저장된 값을 참조하도록 하기 위함이다. 그런데 배열을 통째로 전달하지 않아도 이러한 일이 가능하다.

위의 코드에서 보이는 바와 같이, 배열을 함수의 인자로 전달하려면 배열을 통째로 복사할 수 있도록 배열이 매개변수로 선언되어야 한다. 그러나 C언어는 매개변수로 배열의 선언을 허용하지 않는다. 결론! 배열을 통째로 복사하는 방법은 C언어에 존재하지 않는다.

따라서 배열을 통째로 복사해서 전달하는 방식 대신에, 배열의 주소 값을 전달하는 방식을 대신 취한다.

배열을 함수의 인자로 전달하는 방식

```
int main(void)
{
    int arr[3]={1, 2, 3};
    int * ptr=arr;
    . . . .
}
```
배열의 이름은 int형 포인터!

배열의 이름은 int형 포인터! 따라서 int형 포인터 변수에 배열의 이름이 지니는 주소 값을 저장할 수 있다.

위의 예제를 통해서 다음과 같은 코드의 구성이 가능함을 유추할 수 있다.

```
int main(void)
{
    int arr[3]={1, 2, 3};
    SimpleFunc(arr);
    . . . .
}
```
배열이름 arr이 지니는 주소 값의 전달

```
void SimpleFunc(int * param)
{
    printf("%d %d", param[0], param[1]);
}
```
배열이름 arr은 int형 포인터이므로 매개변수는 int형 포인터 변수!

포인터 변수를 이용해서도 배열의 형태로 접근가능!

배열을 함수의 인자로 전달하는 예제

```
void ShowArayElem(int * param, int len)
{
    int i;
    for(i=0; i<len; i++)
        printf("%d ", param[i]);
    printf("\n");
}

int main(void)
{
    int arr1[3]={1, 2, 3};
    int arr2[5]={4, 5, 6, 7, 8};
    ShowArayElem(arr1, sizeof(arr1) / sizeof(int));
    ShowArayElem(arr2, sizeof(arr2) / sizeof(int));
    return 0;
}
```

실행결과

```
1 2 3
4 5 6 7 8
```

```
void ShowArayElem(int * param, int len)
{
    int i;
    for(i=0; i<len; i++)
        printf("%d ", param[i]);
    printf("\n");
}
void AddArayElem(int * param, int len, int add)
{
    int i;
    for(i=0; i<len; i++)
        param[i] += add;
}

int main(void)
{
    int arr[3]={1, 2, 3};
    AddArayElem(arr, sizeof(arr) / sizeof(int), 1);
    ShowArayElem(arr, sizeof(arr) / sizeof(int));

    AddArayElem(arr, sizeof(arr) / sizeof(int), 2);
    ShowArayElem(arr, sizeof(arr) / sizeof(int));

    AddArayElem(arr, sizeof(arr) / sizeof(int), 3);
    ShowArayElem(arr, sizeof(arr) / sizeof(int));
    return 0;
}
```

실행결과

```
2 3 4
4 5 6
7 8 9
```

배열을 함수의 인자로 전달받는 함수의 또 다른 선언

```
void ShowArayElem (int * param, int len) { . . . . }
void AddArayElem (int * param, int len, int add) { . . . . }
```

동일한 선언

```
void ShowArayElem (int param[], int len) { . . . . }
void AddArayElem (int param[], int len, int add) { . . . . }
```

매개변수의 선언에서는 int * param과 int param[]이 동일한 선언이다. 따라서 배열을 인자로 전달받는 경우에는 int param[]이 더 의미있어 보이므로 주로 사용된다.

```
int main(void)
{
    int arr[3]={1, 2, 3};
    int * ptr=arr;    // int ptr[]=arr; 로 대체 불가능
    . . . .
}
```

하지만 그 이외의 영역에서는 int * ptr의 선언을 int ptr[]으로 대체할 수 없다.

C 프로그래밍

Call-by-value vs. Call-by-reference

값을 전달하는 형태의 함수호출: Call-by-value

함수를 호출할 때 단순히 값을 전달하는 형태의 함수호출을 가리켜 **Call-by-value**라 하고, 메모리의 접근에 사용되는 주소 값을 전달하는 형태의 함수호출을 가리켜 **Call-by-reference**라 한다. 즉, Call-by-value와 Call-by-reference를 구분하는 기준은 함수의 인자로 전달되는 대상에 있다.

```
void NoReturnType(int num)
{
    if(num<0)
        return;
    . . . .
}
```

call-by-value

```
void ShowArayElem(int * param, int len)
{
    int i;
    for(i=0; i<len; i++)
        printf("%d ", param[i]);
    printf("\n");
}
```

call-by-reference

call-by-value와 call-by-reference라는 용어를 기준으로 구분하는 것이 중요한 게 아니다.

중요한 것은 각 함수의 특징을 이해하고 적절한 형태의 함수를 정의하는 것이다.

call-by-value 형태의 함수에서는 **함수 외부에 선언된 변수에 접근이 불가능**하다. 그러나 call-by-reference 형태의 함수에서는 **외부에 선언된 변수에 접근이 가능**하다.

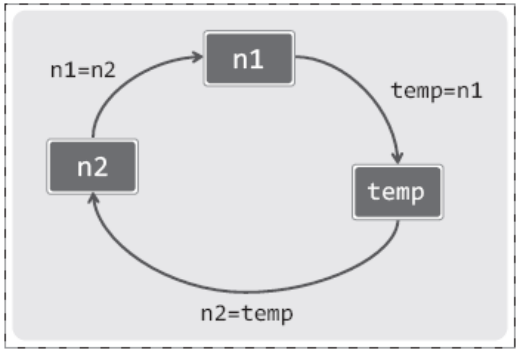
잘못 적용된 Call-by-value

```
void Swap(int n1, int n2)
{
    int temp=n1;
    n1=n2;
    n2=temp;
    printf("n1 n2: %d %d \n", n1, n2);
}

int main(void)
{
    int num1=10;
    int num2=20;
    printf("num1 num2: %d %d \n", num1, num2);

    Swap(num1, num2);   // num1과 num2에 저장된 값이 서로 바뀌길 기대!
    printf("num1 num2: %d %d \n", num1, num2);
    return 0;
}
```

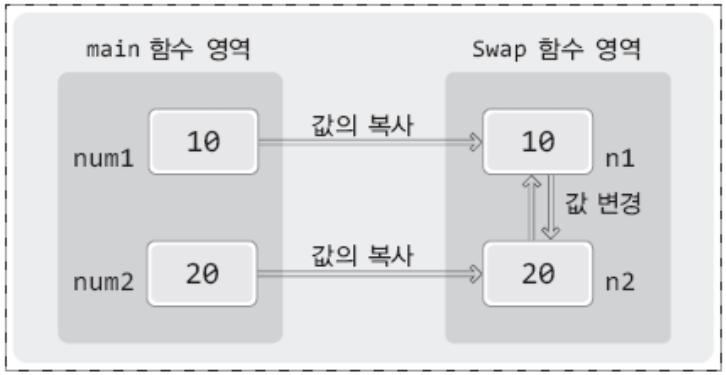
```
num1 num2: 10 20
n1 n2: 20 10
num1 num2: 10 20
```

실행결과

call-by-value가 적절치 않은 경우



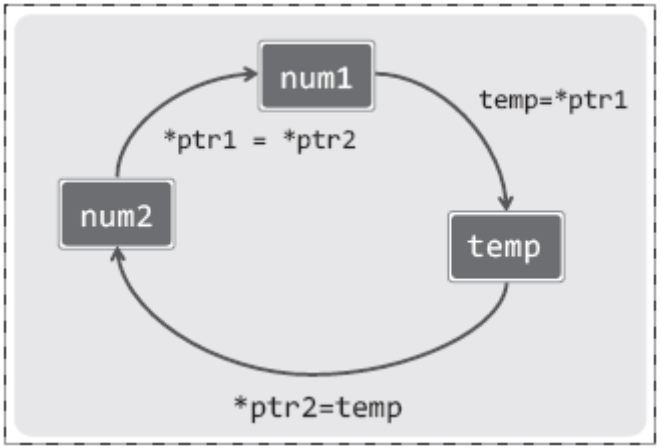
Swap 함수 내에서의 값의 교환



Swap 함수 내에서의 값의 교환은 외부에 영향을 주지 않는다.

주소 값을 전달하는 형태의 함수호출: Call-by-reference

```
void Swap(int * ptr1, int * ptr2)
{
    int temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}

int main(void)
{
    int num1=10;
    int num2=20;
    printf("num1 num2: %d %d \n", num1, num2);
    Swap(&num1, &num2);
    printf("num1 num2: %d %d \n", num1, num2);
    return 0;
}
```



Swap 함수 내에서 함수 외부에 있는 변수간 값의 교환

```
num1 num2: 10 20
num1 num2: 20 10
```

실행결과

Swap 함수 내에서의 *ptr1은 main 함수의 num1

Swap 함수 내에서의 *ptr2는 main 함수의 num2

를 의미하게 된다.

scanf 함수호출 시 & 연산자를 붙이는 이유는?

```
int main(void)
{
     int num;
     scanf("%d", &num);
     . . . .
}
```

변수 num 앞에 & 연산자를 붙이는 이유는?

scanf 함수 내에서 외부에 선언된 변수 num에 접근 하기 위해서는 num의 주소 값을 알아야 한다. 그래서 scanf 함수는 변수의 주소 값을 요구한다.

```
int main(void)
{
     char str[30];
     scanf("%s", str);
     . . . .
}
```

배열 이름 str 앞에 & 연산자를 붙이지 않는 이유는?

str은 배열의 이름이고 그 자체가 주소 값이기 때문에 & 연산자를 붙이지 않는다. str을 전달함은 scanf 함수 내부로 배열 str의 주소 값을 전달하는 것이다.

C 프로그래밍

포인터 대상의 const 선언

포인터 변수의 참조대상에 대한 const 선언

```
int main(void)
{
    int num=20;
    const int * ptr=&num;
    *ptr=30;    // 컴파일 에러!
    num=40;     // 컴파일 성공!
    . . . .
}
```

왼편의 const 선언이 갖는 의미

포인터 변수 ptr을 이용해서 ptr이 가리키는 변수에 저장된 값을 변경하는 것을 허용하지 않겠습니다!

그러나 변수 num에 저장된 값 자체의 변경이 불가능한 것은 아니다.
다만 ptr을 통한 변경을 허용하지 않을뿐이다.

포인터 변수의 상수화

```
int main(void)
{
    int num1=20;
    int num2=30;
    int * const ptr=&num1;
    ptr=&num2;      // 컴파일 에러!
    *ptr=40;        // 컴파일 성공!
    . . . .
}
```

왼편의 const 선언이 갖는 의미

포인터 변수 ptr에 저장된 값을 상수화 하겠다. 즉, ptr에 저장된 값은 변경이 불가능하다. ptr이 가리키는 대상의 변경을 허용하지 않는다.

```
const int * ptr=&num;
 int * const ptr=&num;
```

```
const int * const ptr=&num;
```

두 가지 const 선언을 동시에 할 수 있다.

const 선언이 갖는 의미

```
int main(void)
{
     double PI=3.1415;
     double rad;
     PI=3.07;   // 실수로 잘못 삽입된 문장, 컴파일 시 발견 안됨
     scanf("%lf", &rad);
     printf("circle area %f \n", rad*rad*PI);
     return 0;
}
```

안전성이 높아진 코드

```
int main(void)
{
     const double PI=3.1415;
     double rad;
     PI=3.07;   // 컴파일 시 발견되는 오류상황
     scanf("%lf", &rad);
     printf("circle area %f \n", rad*rad*PI);
     return 0;
}
```

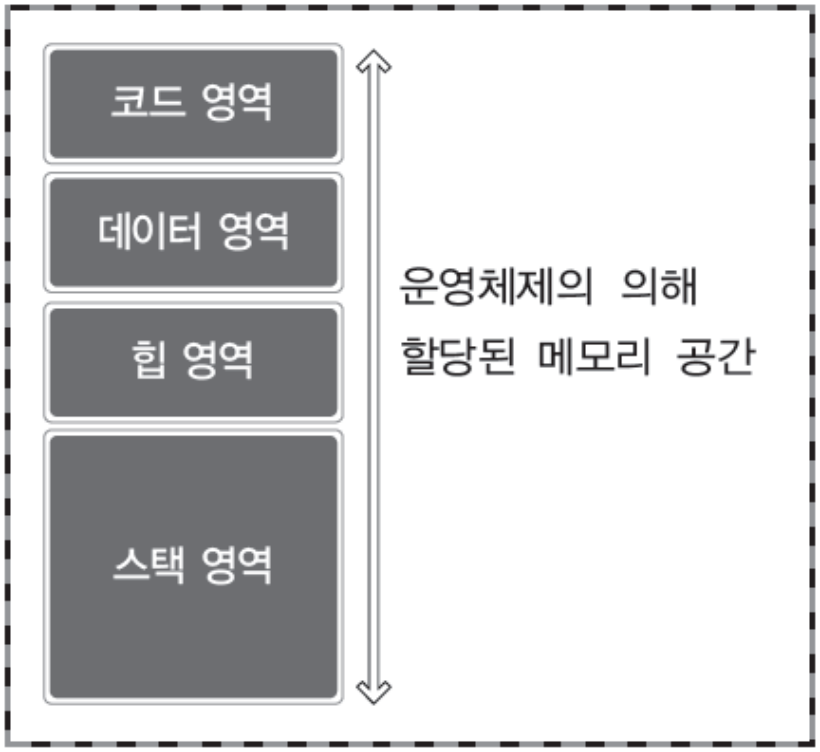
const 선언은 추가적인 기능을 제공하기 위한 것이 아니라, 코드의 안전성을 높이기 위한 것이다. 따라서 이러한 const의 선언을 소홀히하기 쉬운데, const의 선언과 같이 코드의 안전성을 높이는 선언은 가치가 매우 높은 선언이다.

C 프로그래밍

C언어의 메모리 구조

윤성우 저 열혈강의 C 프로그래밍 개정판

메모리의 구성



메모리 공간을 나눠놓은 이유는 커다란 서랍장의 수납공간이 나뉘어 있는 이유와 유사하다.

메모리 공간을 나눠서 유사한 성향의 데이터를 묶어서 저장을 하면, 관리가 용이해지고 메모리의 접근 속도가 향상된다.

메모리 영역별로 저장되는 데이터의 유형

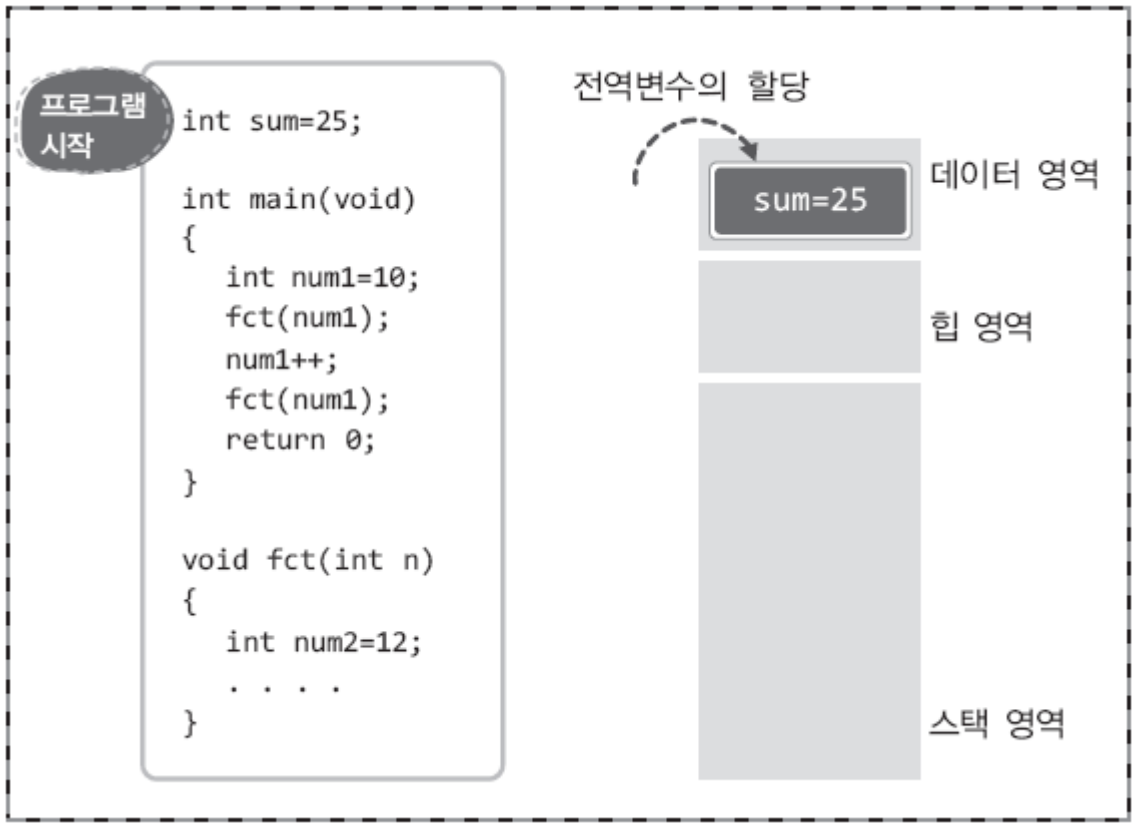
실행할 프로그램의 코드가 저장되는 메모리 공간.
CPU는 코드 영역에 저장된 명령문을 하나씩 가져다가 실행

전역변수와 static 변수가 할당되는 영역.
프로그램 시작과 동시에 할당되어 종료 시까지 남아있는 특징의 변수가 저장되는 영역

프로그래머가 원하는 시점에 메모리 공간에 할당 및 소멸을 하기 위한 영역

지역변수와 매개변수가 할당되는 영역
함수를 빠져나가면 소멸되는 변수를 저장하는 영역

프로그램의 실행에 따른 메모리의 상태 변화1

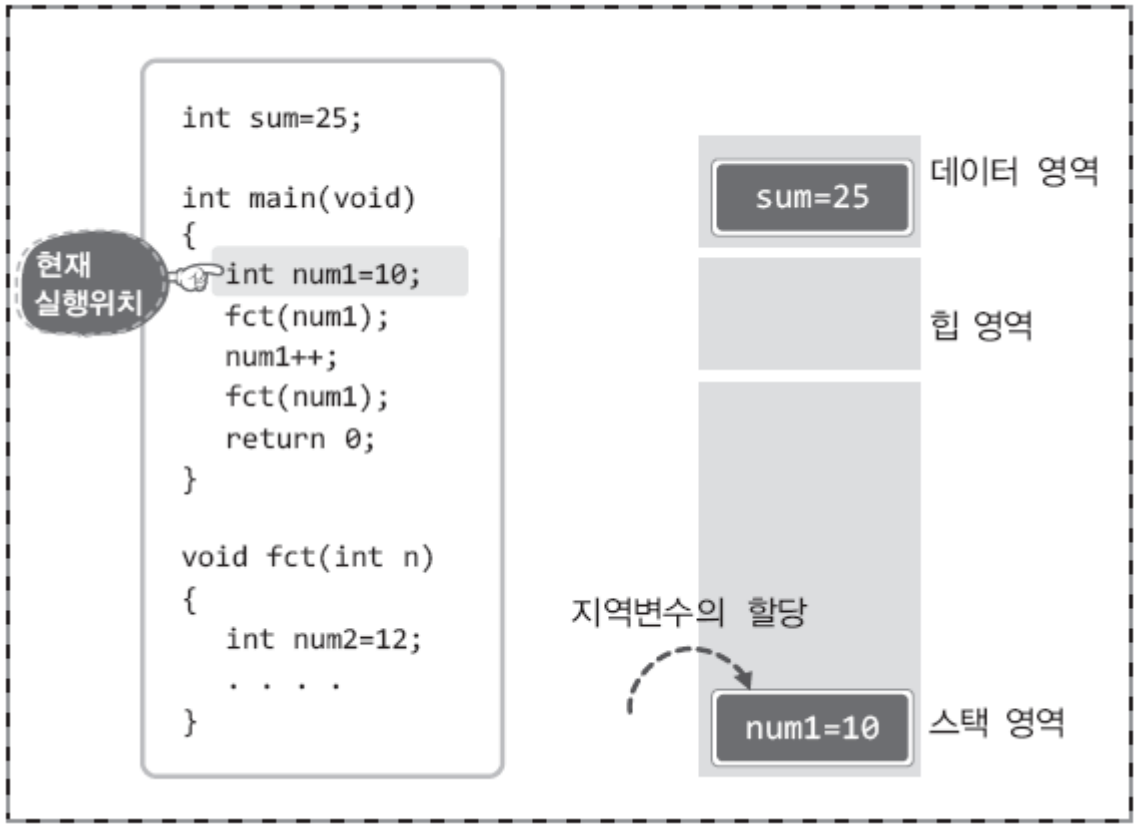


프로그램의 시작:

전역변수의 할당 및 초기화

실행의 흐름/

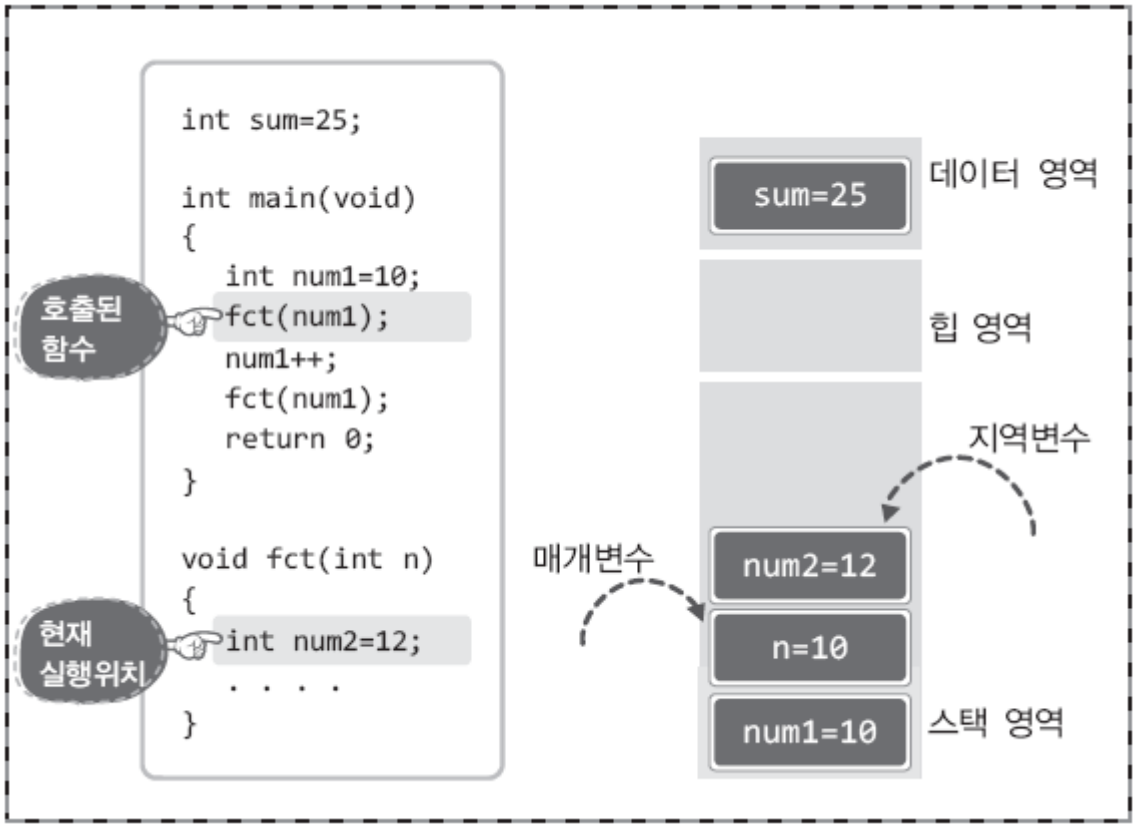
프로그램의 실행에 따른 메모리의 상태 변화2



main 함수의 호출 및 실행

실행의 흐름2

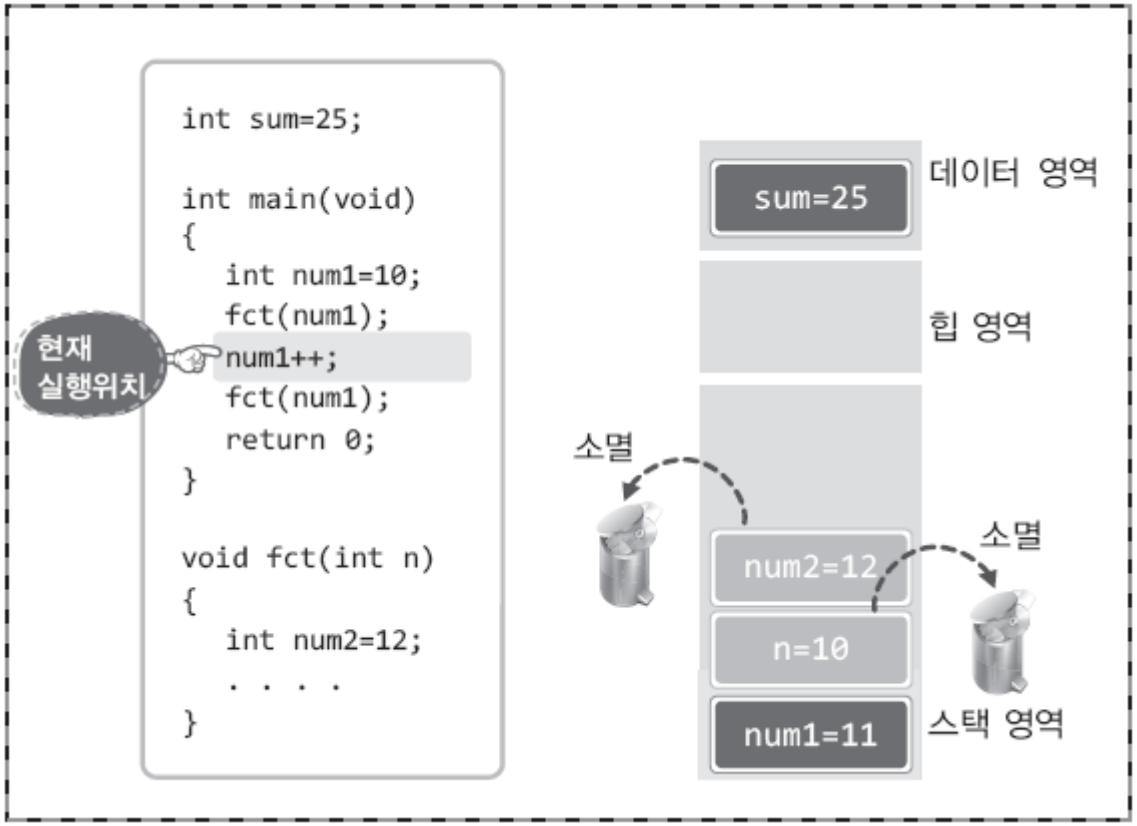
프로그램의 실행에 따른 메모리의 상태 변화3



fct 함수의 호출

실행의 흐름3

프로그램의 실행에 따른 메모리의 상태 변화4

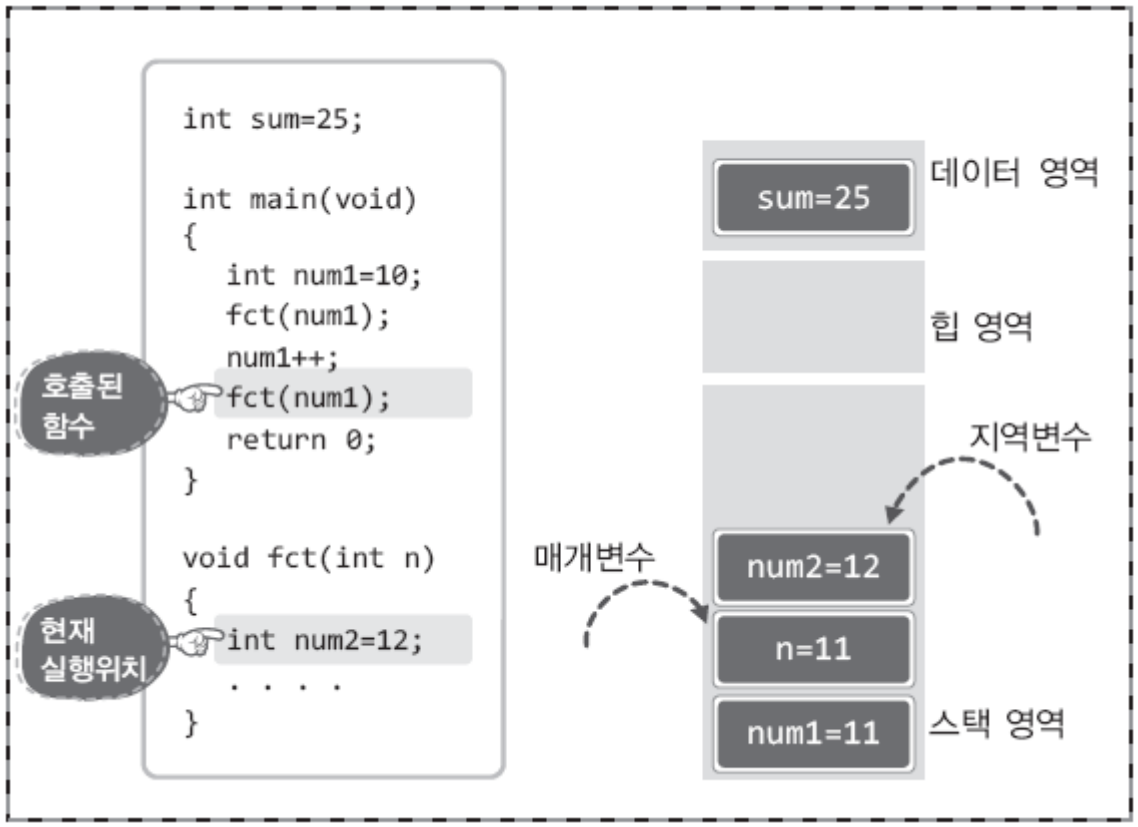


fct 함수의 반환

그리고 main 함수 이어서 실행

실행의 흐름4

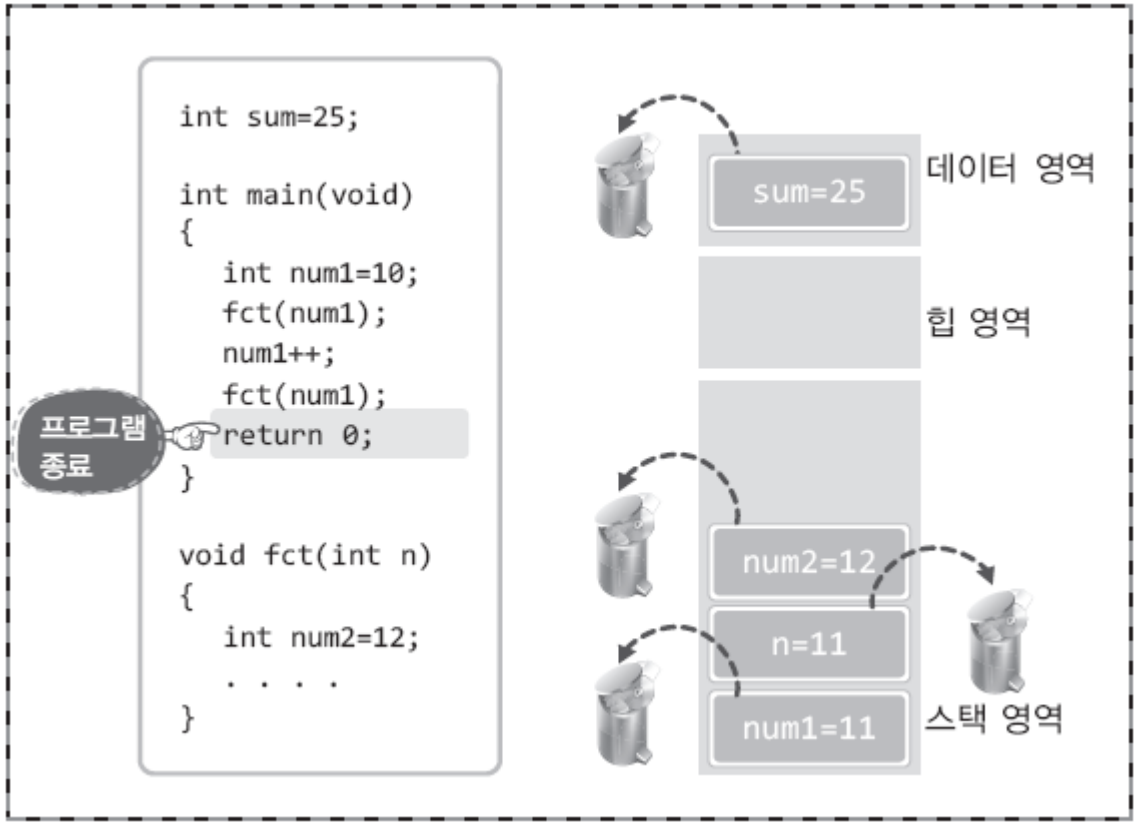
프로그램의 실행에 따른 메모리의 상태 변화5



fct 함수의 재호출 및 실행

실행의 흐름5

프로그램의 실행에 따른 메모리의 상태 변화6



fct 함수의 반환

및 main 함수의 반환

실행의 흐름6 (프로그램 종료)

함수의 호출순서가 main → fct1 → fct2이라면 스택의 반환은(지역변수의 소멸은) 그의 역순인 fct2 → fct1 → main으로 이루어진다는 특징을 기억하자!

윤성우의

열혈 C 프로그래밍

Chapter 25-2. 메모리의 동적 할당

윤성우 저 열혈강의 C 프로그래밍 개정판

전역변수와 지역변수로 해결이 되지 않는 상황

```
char * ReadUserName(void)
{
    char name[30];
    printf("What's your name? ");
    gets(name);
    return name;
}

int main(void)
{
    char * name1;
    char * name2;
    name1=ReadUserName();
    printf("name1: %s \n", name1);
    name2=ReadUserName();
    printf("name2: %s \n", name2);
    return 0;
}
```

무엇을 반환하는가?

변수 name은 ReadUserName 함수호출 시 할당이 되어야 하고,

ReadUserName 함수가 반환을 하더라도 계속해서 존재해야 한다.

그런데 전역변수도 지역변수도 이러한 유형에는 부합하지 않는다!

혹시 전역변수가 답이 된다고 생각하는가?

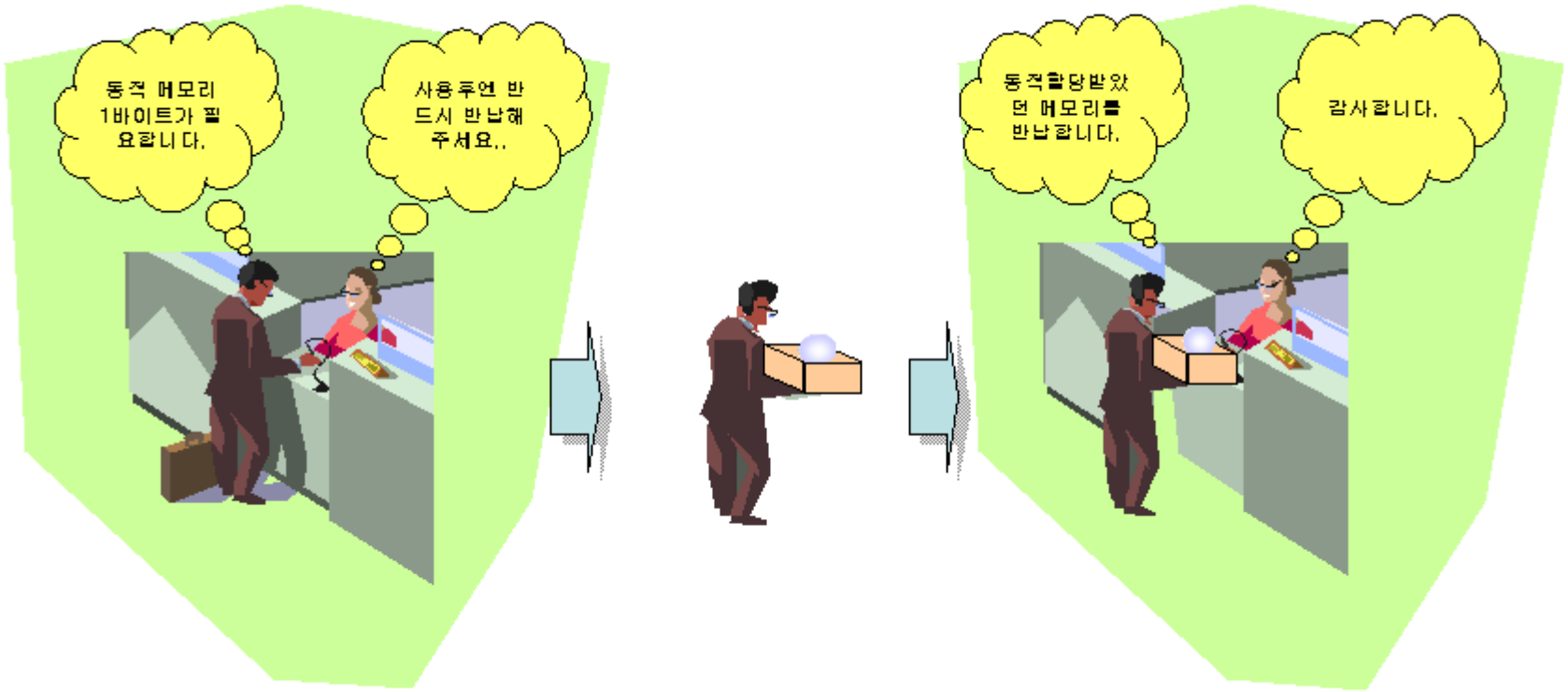
```
char name[30];
char * ReadUserName(void)
{
    printf("What's your name? ");
    gets(name);
    return name;
}
int main(void)
{
    char * name1;
    char * name2;
    name1=ReadUserName();
    printf("name1: %s \n", name1);
    name2=ReadUserName();
    printf("name2: %s \n", name2);
    printf("name1: %s \n", name1);
    printf("name2: %s \n", name2);
    return 0;
}
```

전역변수는 답이 될 수 없음을 보이는 예제 및 실행결과

실행결과

```
What's your name? Yoon sung woo
name1: Yoon sung woo
What's your name? Choi jun kyung
name2: Choi jun kyung
name1: Choi jun kyung
name2: Choi jun kyung
```

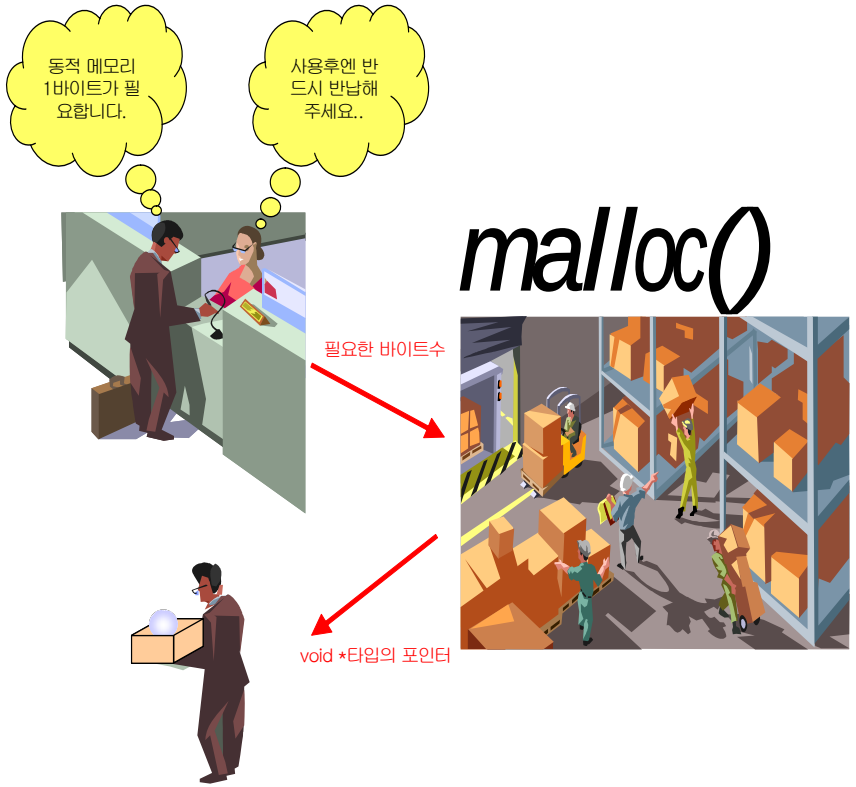
동적 메모리 할당 절차


동적 메모리 1바이트가 필요합니다.
사용후엔 반드시 반납해 주세요..
동적할당받았던 메모리를 반납합니다.
감사합니다.
동적 메모리 할당
동적 메모리 사용
동적 메모리 반납

malloc()과 free()

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

- malloc()은 바이트 단위로 메모리를 할당
- size는 바이트의 수
- malloc()함수는 메모리 블럭의 첫 번째 바이트에 대한 주소를 반환
- 만약 요청한 메모리 공간을 할당할 수 없는 경우에는 NULL값을 반환




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

- **free()**는 동적으로 할당되었던 메모리 블록을 시스템에 반납
- **ptr**은 **malloc()**을 이용하여 동적 할당된 메모리를 가리키는 포인터

힙 영역의 메모리 공간 할당과 해제

```
#include <stdlib.h>
void * malloc(size_t size);       // 힙 영역으로의 메모리 공간 할당
void free(void * ptr);     // 힙 영역에 할당된 메모리 공간 해제
```

➡ malloc 함수는 성공 시 할당된 메모리의 주소 값, 실패 시 NULL 반환

```
int main(void)
{
    void * ptr1 = malloc(4);     // 4바이트가 힙 영역에 할당
    void * ptr2 = malloc(12);    // 12바이트가 힙 영역에 할당
    . . . .
    free(ptr1);    // ptr1이 가리키는 4바이트 메모리 공간 해제
    free(ptr2);    // ptr2가 가리키는 12바이트 메모리 공간 해제
    . . . .
}
```

반환형이 void형 포인터임에 주목!

malloc & free 함수 호출의 기본 모델

malloc 함수의 반환형이 void형 포인터인 이유

```
void * ptr1 = malloc(sizeof(int));
void * ptr2 = malloc(sizeof(double));
void * ptr3 = malloc(sizeof(int)*7);
void * ptr4 = malloc(sizeof(double)*9);
```

```
void * ptr1 = malloc(4);
void * ptr2 = malloc(8);
void * ptr3 = malloc(28);
void * ptr4 = malloc(72);
```

malloc 함수의 일반적인 호출형태

sizeof 연산 이후 실질적인 malloc의 호출

malloc 함수는 인자로 숫자만 하나 전달받을 뿐이니 할당하는 메모리의 용도를 알지 못한다. 따라서 메모리의 포인터 형을 결정짓지 못한다. 따라서 다음과 같이 형 변환의 과정을 거쳐서 할당된 메모리의 주소 값을 저장해야 한다.

```
int * ptr1 = (int *)malloc(sizeof(int));
double * ptr2 = (double *)malloc(sizeof(double));
int * ptr3 = (int *)malloc(sizeof(int)*7);
double * ptr4 = (double *)malloc(sizeof(double)*9);
```

malloc 함수의
가장 모범적인 호출형태

힙 영역으로의 접근

```
int main(void)
{
    int * ptr1 = (int *)malloc(sizeof(int));
    int * ptr2 = (int *)malloc(sizeof(int)*7);
    int i;

    *ptr1 = 20;
    for(i=0; i<7; i++)
        ptr2[i]=i+1;

    printf("%d \n", *ptr1);
    for(i=0; i<7; i++)
        printf("%d ", ptr2[i]);

    free(ptr1);
    free(ptr2);
    return 0;
}
```

```
int * ptr = (int *)malloc(sizeof(int));
if(ptr==NULL)
{
    // 메모리 할당 실패에 따른 오류의 처리
}
```

메모리 할당 실패 시 malloc 함수는 NULL을 반환

이렇듯 힙 영역으로의 접근은 포인터를 통해서만 이뤄진다.

실행결과

```
20
1 2 3 4 5 6 7
```

'동적 할당'이라 하는 이유!

컴파일 시 할당에 필요한 메모리 공간이 계산되지 않고, 실행 시 할당에 필요한 메모리 공간이 계산되므로!

free 함수를 호출하지 않으면?

- free 함수를 호출하지 않으면?

  할당된 메모리 공간은 메모리라는 중요한 리소스를 계속 차지하게 된다.

- free 함수를 호출하지 않으면 프로그램 종료 후에도 메모리를 차지하는가?

  프로그램이 종료되면 프로그램 실행 시 할당된 모든 자원이 반환된다.

- 꼭 free 함수를 호출해야 하는 이유는 무엇인가?

  fopen 함수와 쌍을 이루어 fclose 함수를 호출하는 것과 유사하다.

- 예제에서 조차 늘 free 함수를 호출하는 이유는 습관을 들이기 위해서인가?

  맞다! fopen, fclose가 늘 쌍을 이루듯 malloc, free도 쌍을 이루게 하자!

문자열 반환하는 함수를 정의하는 문제의 해결

```
char * ReadUserName(void)
{
    char * name = (char *)malloc(sizeof(char)*30);
    printf("What's your name? ");
    gets(name);
    return name;
}
```

할당!

ReadUserName 함수가 호출될 때마다 새로운 메모리 공간이 할당이 되고 이 메모리 공간은 함수를 빠져나간 후에도 소멸되지 않는다!

```
int main(void)
{
    char * name1;
    char * name2;
    name1=ReadUserName();
    printf("name1: %s \n", name1);
    name2=ReadUserName();
    printf("name2: %s \n", name2);

    printf("again name1: %s \n", name1);
    printf("again name2: %s \n", name2);
    free(name1);
    free(name2);
    return 0;
}
```

소멸!

```
What's your name? Yoon Sung Woo
name1: Yoon Sung Woo
What's your name? Hong Sook Jin
name2: Hong Sook Jin
again name1: Yoon Sung Woo
again name2: Hong Sook Jin
```

실행결과

calloc & realloc

```
#include <stdlib.h>
void * calloc(size_t elt_count, size_t elt_size);
```

➜ 성공 시 할당된 메모리의 주소 값, 실패 시 NULL 반환

malloc 함수와의 가장 큰 차이점은 메모리 할당을 위한 인자의 전달방식

elt_count × elt_size 크기의 바이트를 동적 할당한다. 즉, elt_size 크기의 블록을 elt_count 의 수만큼 동적할당! 그리고 malloc 함수와 달리 모든 비트를 0으로 초기화!

```
#include <stdlib.h>
void * realloc(void * ptr, size_t size);
```

➜ 성공 시 새로 할당된 메모리의 주소 값, 실패 시 NULL 반환

ptr이 가리키는 힙의 메모리 공간을 size의 크기로 늘리거나 줄인다!

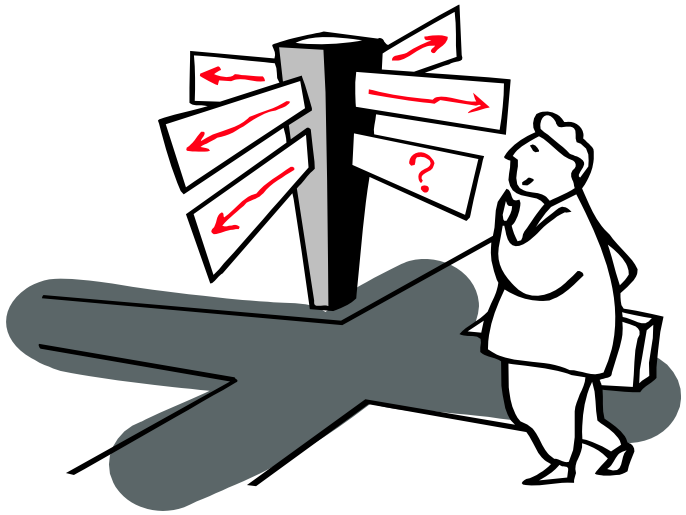
malloc, calloc, realloc 함수호출을 통해서 할당된 메모리 공간은 모두 free 함수호출을 통해서 해제한다.

realloc 함수의 보충설명

```
int main(void)
{
    int * arr = (int *)malloc(sizeof(int)*3);   // 길이가 3인 int형 배열 할당
    . . . .
    arr = (int *)realloc(arr, sizeof(int)*5);   // 길이가 5인 int형 배열로 확장
    . . . .
}
```

- malloc 함수! 그리고 realloc 함수가 반환한 주소 값이 같은 경우
  - ✈ 기존에 할당된 메모리 공간을 이어서 확장할 여력이 되는 경우
- malloc 함수! 그리고 realloc 함수가 반환한 주소 값이 다른 경우
  - ✈ 기존에 할당된 메모리 공간을 이을 여력이 없어서 새로운 공간을 마련하는 경우

새로운 공간을 마련해야 하는 경우에는 메모리의 복사과정이 추가됨에 주목!



Chapter 14가 끝났습니다. 질문 있으신지요?