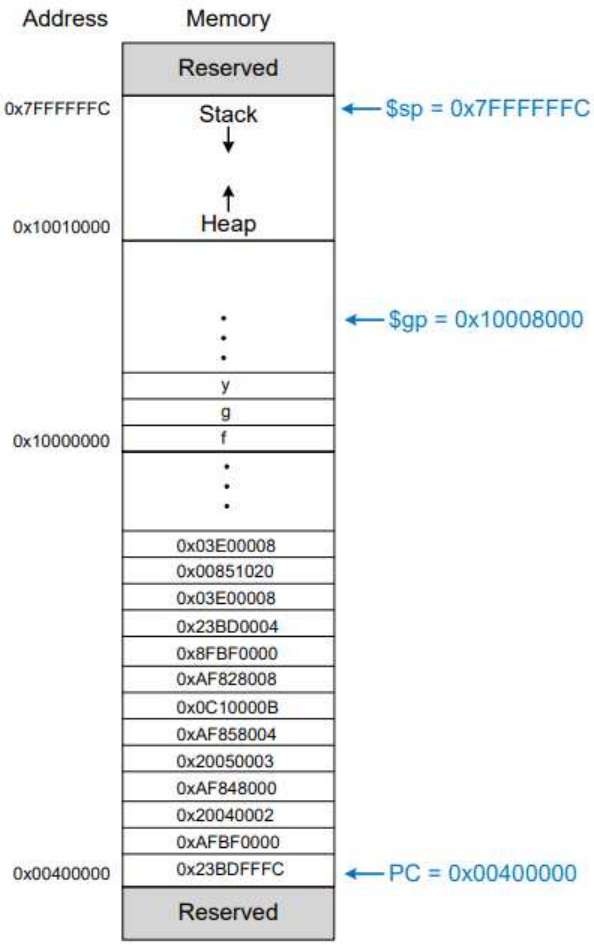
10 장 동적 메모리 할당

10.1 메모리 할당

학교에서 자신의 책이나 소지품을 보관하려면 사물함을 할당 받아야 하듯, C언어로 작성된 프로그램에서 데이터를 저장하기 위해서는 메모리를 할당 받아야 합니다. 모든 시스템에서 메모리는 무한히 제공되는 영역이 아닌 제한된 공간이기 때문에 효율적 사용을 위해 전체 메모리 영역을 용도에 맞게 분할해서 사용하게 됩니다.



[그림 10.1] Executable loaded in memory ("Digital Design and Computer

Architecture 2nd Edition", David Harris, Sarah Harris, Elsevier, 2012.)

[그림 10.1]은 메모리의 사용 잘 보여주는 간단한 예입니다. 그림의 아래에부터 메모리 주소 0x00400000의 Address 부분은 사용자가 작성한 프로그램이 놓이는 부분이고, 0x10000000의 f, g, y에 해당하는 부분은 글로벌 데이터 영역입니다. 0x10010000부터 아래쪽에서부터 쌓아 올라가는 부분이 힙(Heap) 영역, 위쪽에서부터 저장되는 부분이 스택(Stack) 영역입니다.

메모리를 할당 받는 방법은 크게 정적 메모리 할당(Static Memory Allocation) 방법과 동적 메모리 할당(Dynamic Memory Allocation) 방법이 있습니다. 프로그램에서 배열이나 함수 등 정적 메모리를 사용할 때에는 스택(Stack) 영역에 데이터들이 저장(push)되거나 꺼내(pop)지게 됩니다. 또한 프로그램 수행 도중 동적으로 메모리를 사용하는 경우에는 힙(Heap) 영역을 할당(allocation) 받거나 반납(free)하여 데이터를 사용하게 됩니다.

가. 정적 메모리 할당 (Static Memory Allocation)

정적 메모리 할당이란 사용하고자 하는 메모리의 크기를 미리 정해 놓고 데이터를 저장하는 방법입니다. 가장 간단하게는 저장하려는 자료의 유형에 알맞은 변수를 선언해서 그 변수에 데이터를 저장하는 것입니다. 변수를 선언한다는 것은 그 변수의 자료형 크기만큼 스택(Stack) 영역의 메모리를 할당 받는 것이기 때문입니다. 만약 같은 유형의 자료가 여러 개라면 배열을 사용할 수 있습니다. 이러한 방식은 사용량을 예측해서 미리 메모리에 자리를 잡아 두는 방식이므로 데이터의 양에 따라 메모리가 부족할 수도 남을 수도 있습니다.

나. 동적 메모리 할당 (Dynamic Memory Allocation)

정적 메모리 할당 방식을 사용할 경우 확보된 메모리 공간의 낭비 또는 부족이 발생할 수 있습니다. 문제가 생기면 좋은 방향으로 해결을 해 나가면 되겠죠? 정적 메모리 할당 방식의 근본적인 문제는 어느 정도의 메모리 공간이 필요한지 예측이 불가능한 상태에서 프로그램이 실행되기 전에 메모리 공간을 확보했기 때문입니다. 그러니 이 방식의 순서를 뒤집으면 되지 않을까요? 즉, 프로그램이 실행하면서 필요에 따라 메모리 공간을 확보할 수 있게 하는 것입니다. 이렇게 하면 원하는 시점에 알맞은 크기로 메모리를 사용할 수 있을 것입니다.

□ 생각해 볼까요?

프로그램이 실행하면서 필요에 따라 메모리 공간을 할당한다면 확보된 메모리 공간의 낭비 또는 부족 문제는 발생하지 않을 수 있습니다. 그러나 계속 할당만 한다면 전체 메모리 용량을 다 써버려 더 이상 메모리를 할당 할 수 없는 경우가 발생 할 수 있습니다. 그렇다면 이러한 문제는 어떻게 해결할 수 있을까요?

프로그램이 실행하면서 필요에 따라 메모리 공간을 확보하면서, 동시에 사용이 끝난 메모리 영역은 반납하는 형식을 취한다면 좀 더 효율적으로 메모리를 사용할 수 있게 됩니다. 이러한 방식으로 메모리를 할당 받는 체계를 동적 메모리 할당 방식이라고 하며 이 경우에는 메모리의 힙(Heap) 영역을 할당(allocation) 받거나 반납(free)하여 데이터를 사용하게 됩니다.

10.2 동적 메모리 할당에 관련된 함수

동적으로 메모리를 할당 받기 위해서는 관련 함수들을 사용해야 합니다. 이 함수들을 사용하려면 #include <stdlib.h>를 포함해야 합니다.

[표 10.1] 동적 메모리 할당에 관련된 함수

함수명	함수 원형	기능
malloc()	void * malloc(size_t)	size_t 크기에 해당하는 메모리를 힙(Heap) 영역에 할당하여 그 시작 주소를 반환한다. 메모리 할당을 할 수 없을 경우에는 NULL을 반환한다. 초기 값은 무의미한 값이다.
calloc()	void * calloc(size_ta, size_tb)	size_ta 개수만큼 size_tb 크기에 해당하는 메모리 할당하여 그 시작 주소를 반환한다. 메모리 할당을 할 수 없을 경우에는 NULL을 반환한다. 초기 값은 0이다.
realloc()	void * realloc(void *, size_t)	함수 매개변수 void * 에 해당하는 메모리 영역의 크기를 size_t 크기로 변경하여 메모리를 재 할당한다. 메모리 할당을 할 수 없을 경우에는 NULL을 반환한다. 이미 저장되어있는 값은 그대로 두고, 늘어난 크기에 해당하는 영역의 초기 값은 무의미한 값이다.
free()	free(void *)	함수 매개변수 void * 에 해당하는 메모리 영역의 사용을 해제한다.

[예제 10.1]는 malloc()을 사용하여 정수를 한 개를 저장하는 예입니다. malloc()을 사용해서 sizeof(int) *10 bytes 만큼의 메모리를 동적으로 할당 받아 그 주소를 포인터 변수 pi에 연결하고 그곳에 정수를 저장합니다. 실행결과에서 알 수 있듯이, 메모리를 할당 받은 후 값이 저장되지 않은 상태에서는 그 영역이 초기화 되어있지 않은 무의미한 값을 가지고 있습니다. 이후 메모리의 사용이 끝나면 free()를 사용하여 할당 받았던 메모리를 반납합니다.

[예제 10.1] malloc()을 사용하여 정수를 한 개를 저장

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        int *pi = NULL;

        if ((pi = (int *)malloc(sizeof(int))) == NULL) {
                printf("메모리 할당에 문제가 있습니다.");
                exit(1);
        }
        printf("Before : *pi=%d, pi=%u \n", *pi, pi);
        *pi = 3;
        printf("After : *pi=%d, pi=%u \n", *pi, pi);
        free(pi);
        return 0;
}
```

```
Before : *pi=8065896, pi=8073128
After : *pi=3, pi=8073128
```

[예제 10.2]는 malloc()을 사용하여 정수를 세 개를 저장하는 예입니다. 즉, 정수형 배열처럼 사용된 예입니다.

[예제 10.2] malloc()을 사용하여 정수 여러 개를 배열에 저장

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        int *piary = NULL, count = 3;

        if ((piary = (int *)malloc(sizeof(int)*count)) == NULL) {
                printf("메모리 할당에 문제가 있습니다.");
                exit(1);
        }
        for (int i = 0; i < count; i++) {
```

```
Before : *(piary+0)=7738216, piary=7745448
Before : *(piary+1)=7733440, piary=7745452
Before : *(piary+2)=100663302, piary=7745456
After : *(piary+0)=3, piary=7745448
After : *(piary+1)=4, piary=7745452
After : *(piary+2)=5, piary=7745456
```

```
                printf("Before : *(piary+%d)=%d, piary=%u \n", i, *(piary+i),
(piary+i));
        }
        for (int i = 0; i < count; i++) {
                *(piary + i) = i + count;
                printf("After : *(piary+%d)=%d, piary=%u \n", i, *(piary+i),
(piary+i));
        }
        free(piary);
        return 0;
}
```

[예제 10.3]은 calloc()을 사용하여 정수를 세 개 저장하는 [예제 10.2]와 같은 기능을 하는 예입니다.

□ 생각해 볼까요?

[예제 10.3]는 [예제 10.2]와 어떤 점이 달라졌나요?

실행결과의 Before 부분과 After 부분을 비교해 보세요. calloc()을 사용하여 메모리를 할당 받은 경우에는 영역이 지정됨과 동시에 0으로 초기화 된다는 것을 알 수 있습니다.

[예제 10.3] calloc()을 사용하여 정수 여러 개를 배열에 저장

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        int *piary = NULL, count = 3;

        if ((piary = (int *)calloc(count, sizeof(int))) == NULL) {
                printf("메모리 할당에 문제가 있습니다.");
                exit(1);
        }
```

```
Before : *(piary+0)=0, piary=14167976
Before : *(piary+1)=0, piary=14167980
Before : *(piary+2)=0, piary=14167984
After : *(piary+0)=3, piary=14167976
After : *(piary+1)=4, piary=14167980
After : *(piary+2)=5, piary=14167984
```

```
        for (int i = 0; i < count; i++) {
                printf("Before : *(piary+%d)=%d, piary=%u \n", i, *(piary+i),
(piary+i));
        }
        for (int i = 0; i < count; i++) {
                *(piary + i) = i + count;
                printf("After : *(piary+%d)=%d, piary=%u \n", i, *(piary+i),
(piary+i));
        }
        free(piary);
        return 0;
}
```

[예제 10.4]는 할당받은 메모리의 크기를 변경해야 하는 경우 realloc()을 사용하여, 함수의 매개변수인 piary가 지시하는 메모리 영역을 count+add 만큼의 크기로 변경하여 메모리를 다시 할당합니다. 실행결과에서 알 수 있듯이 이미 저장되어있는 값은 그대로 두고, 늘어난 크기에 해당하는 영역의 초기 값은 무의미한 값이 됩니다.

[예제 10.4] realloc()을 사용하여 정수 여러 개를 배열에 저장

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        int *piary = NULL, count = 3, add = 2;

        if ((piary = (int *)calloc(count, sizeof(int))) == NULL) {
                printf("메모리 할당에 문제가 있습니다.");
                exit(1);
        }
        if ((piary = (int *)realloc(piary, sizeof(int)*(count+add))) == NULL) {
                printf("메모리 할당에 문제가 있습니다.");
                exit(1);
        }
        for (int i = 0; i < count + add; i++) {
```

```
*(piary+0)=0, piary=7286696
*(piary+1)=0, piary=7286700
*(piary+2)=0, piary=7286704
*(piary+3)=0, piary=7286708
*(piary+4)=83886085, piary=7286712
```

```
            printf("*(piary+%d)=%d, piary=%d \n", i, *(piary+i), (piary+i));
        }
        free(piary);
        return 0;
}
```

[예제 10.2], [예제 10.3], [예제 10.4]의 실행결과에서 포인터 변수 piary가 나타내는 주소를 확인해 보시기 바랍니다. 세 예제에서 공통적으로 확인 할 수 있는 것은 메모리의 영역이 연속적으로 할당되었다는 것입니다.

10.3 정적 메모리 할당과 동적 메모리 할당의 비교

[예제 10.5]에서는 학생들의 성적을 저장하기 위해 sizeof(int)*10 bytes 만큼의 정적 메모리를 할당 받았습니다. 학생 수가 10명 이하로 입력되면 메모리에 잡아 놓은 자리에 여유가 생기지만, 10개 보다 많이 입력하게 되면 메모리에 데이터를 저장할 공간이 부족하게 됩니다.

[예제 10.5] 배열에 정수 10개 저장 (정적 메모리 할당 방식)

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int index = 0;
        int score[10]; //정적 메모리 할당

        printf("학생 수를 입력하세요 : ");
        scanf("%d", &index);

        for (int i = 0; i < index; i++) {
                printf("점수를 입력하세요(입력 종료 -1) : ");
                scanf("%d", score+i);

                if (*(score + i) == -1) break;
```

```
학생 수를 입력하세요 : 3
점수를 입력하세요(입력 종료 -1) : 100
점수를 입력하세요(입력 종료 -1) : 95
점수를 입력하세요(입력 종료 -1) : 98
score[0]=100
score[1]=95
score[2]=98
```

```
        }
        for (int i = 0; i < index; i++) {
                printf("score[%d]=%d\n", i, *(score + i));
        }
        return 0;
}
```

예를 들어 두 개 분반의 C 프로그래밍 수업이 있다고 가정합니다. 분반마다 학생 수가 다를 때, 정적 메모리 할당 방식을 사용한다면 두 분반 중 학생 수가 많은 분반을 기준으로 저장 영역을 확보해야 합니다. [예제 10.5]에서는 10명 학생의 정보를 저장할 수 있는 영역이 확보되었지만 세 개의 성적을 입력했기 때문에 sizeof(int)*7 bytes의 메모리 공간이 남았습니다.

□ 생각해 볼까요?

정적 메모리 할당 방식을 사용할 경우 "확보된 메모리 공간의 낭비 또는 부족" 문제가 발생 할 수 있습니다. 그렇다면 이러한 문제는 어떻게 해결할 수 있을까요?

[예제 10.5]와 [예제 10.6] 두 프로그램의 차이점을 찾아보면 어떻게 해결해야 하는지 알 수 있습니다.

[예제 10.6] 배열에 정수 10개 저장 (동적 메모리 할당 방식)

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
        int index = 0;
        int *score;

        printf("학생 수를 입력하세요 : ");
        scanf("%d", &index);
```

```
학생 수를 입력하세요 : 3
점수를 입력하세요 : 100
점수를 입력하세요 : 95
점수를 입력하세요 : 98
score[0]=100
score[1]=95
score[2]=98
```

```
        //동적 메모리 할당
        if ((score = (int *)calloc(index, sizeof(int))) == NULL) {
                printf("메모리 할당에 문제가 있습니다.");
                exit(1);
        }
        for (int i = 0; i < index; i++) {
                printf("점수를 입력하세요 : ");
                scanf("%d", score+i);
        }
        for (int i = 0; i < index; i++) {
                printf("score[%d]=%d\n", i, *(score + i));
        }
        free(score);
        return 0;
}
```

[예제 10.6]은 동적 메모리의 사용을 보여드리기 위한 가장 간단한 예제 입니다. 그러나 프로그램이 실행 된 후 학생 수에 해당하는 영역만큼 메모리를 동적으로 할당 받았기 때문에 프로그램이 동작하면서 필요에 따라 메모리 공간을 늘려가는 진정한 동적 메모리 할당의 진수는 보여주지 못한 예입니다.

10.4 연결 리스트

[예제 10.6] 프로그램은 진행 도중 학생 수가 변경된다면 메모리를 재 할당해야 하는 문제점도 있습니다. 또한 [예제 10.2], [예제 10.3], [예제 10.4]의 실행결과에서 확인할 수 있듯이 메모리의 영역이 연속적으로 할당되기 때문에 방대한 양의 데이터를 저장하기 위해 메모리를 재 할당할 경우에는 기존에 저장되어있던 데이터와의 충돌이 발생할 가능성도 존재합니다.

이러한 이유 때문에 동적 메모리를 할당 받아 데이터를 저장하기 위한 가장 좋은 방법은 저장이 필요한 순간 마다 메모리를 할당 받아 저장하고 그

저장한 순서대로 “이것 다음에는 그것, 그것 다음에는 저것, 저것 다음은 없음” 이렇게 연결고리를 가지고 데이터를 저장 하도록 하는 것입니다. 이러한 방법, 즉 연결 리스트의 형태를 사용하면 저장하려고 하는 데이터의 개수에도 상관없고 저장되는 위치가 연속적일 필요도 없기 때문에 좀 더 메모리를 효율적으로 사용할 수 있게 됩니다.

연결 리스트를 사용하려면 두 가지의 필수적인 정보가 필요합니다. 그 하나는 현재의 데이터이고, 나머지 하나는 다음 데이터의 위치 정보입니다.

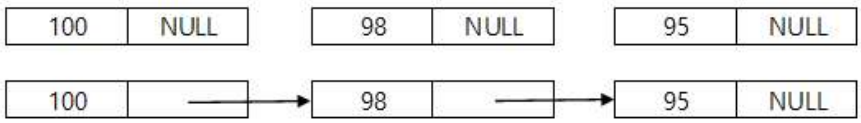
□ 생각해 볼까요?

C언어에서 서로 다른 형태의 여러 가지 정보를 저장하기 위해 사용하는 방법은 무엇인가요?

우리는 서로 다른 형태의 정보를 저장하기 위한 방법으로 구조체를 사용한다는 것을 알고 있습니다. 점수를 저장하기 위한 score와 다음 데이터를 지시하기 위한 next를 구성 성분으로 하는 단위 구조체 unit을 아래의 방법으로 정의했습니다.

```
//현재 점수와 다음 데이터를 지시하기 위한 단위 구조체 unit
struct unit {
        int score; //현재 unit의 점수를 저장
        struct unit *next; //다음 unit의 링크를 저장
};
typedef struct unit unit; //struct unit을 unit으로 재정의
```

이 구조체를 사용하여 연결리스트의 개념을 파악할 수 있도록 간단한 예를 작성해 보도록 하겠습니다. [예제 10.7]은 [그림 10.2]를 프로그램화 했으며, 가독성을 높이기 위해 불필요한 부분은 가능한 생략하여 아주 간단하게 작성했습니다.



[그림 10.2] 연결 리스트 개념

[예제 10.7] [그림 10.2]연결 리스트 구현

```
#include <stdio.h>
#include <stdlib.h>
```

```
first->score=100
first->next->score=95, second->score=95
first->next->next->score=98, third->score=98
```

```
//점수와 다음 node를 지시하기 위한 단위 구조체
struct unit {
        int score;
        struct unit *next;
};
typedef struct unit unit;

int main(int argc, char *argv[])
{
        unit *first = NULL, *second = NULL, *third=NULL;
        int val;

        first = (unit *)malloc(sizeof(unit));
        first->score = 100;
        first->next = NULL;

        second = (unit *)malloc(sizeof(unit)); //동적 메모리 할당
        second->score = 95; //데이터 저장
        second->next = NULL;

        third = (unit *)malloc(sizeof(unit));
        third->score = 98;
        third->next = NULL;

        first->next = second; //first에 second을 연결
        second->next = third; //second에 third을 연결

        printf("first->score=%d\n", first->score);
        printf("first->next->score=%d, second->score=%d\n", first->next->score,
```

```
second->score);
        printf("first->next->next->score=%d, third->score=%d\n",
first->next->next->score, third->score);

        free(first);
        free(second);
        free(third);
        return 0;
}
```

unit 구조체 타입의 포인터 변수 first, second, third는 동적메모리를 할당받아 필요한 데이터를 저장한 후 first 변수의 next에 second를, second 변수의 next에 third를 저장했습니다. 이렇게 되면 first는 second를 알고, second는 third를 알고, third의 next는 NULL 이므로 이 세 변수들의 연결의 끝은 third가 됩니다. 서로가 연결되어 있기 때문에 first->next->score는 second->score, first->next->next->score는 third->score가 되는 것입니다.

[예제 10.8]은 [예제 10.5]와 [예제 10.6]을 수정하여 연결 리스트를 사용한 동적 메모리 할당 방식으로 변경한 것입니다. 프로그램 코드에서 head는 연결 리스의 시작, node는 unit 단위 구조체로 작성된 하나의 데이터, list는 연결된 전체를 의미합니다.

[예제 10.8] 여러 개의 정수를 연결 리스트 형태로 저장

```
#include <stdio.h>
#include <stdlib.h>

struct unit {
        int score;
        struct unit *next;
};
typedef struct unit unit;
int main(int argc, char *argv[])
{
        unit *head = NULL, *node = NULL, *list = NULL;
```

```
점수를 입력하세요(입력 종료 -1) : 100
점수를 입력하세요(입력 종료 -1) : 95
점수를 입력하세요(입력 종료 -1) : 98
점수를 입력하세요(입력 종료 -1) : -1

list sequence = 100 -> 95 -> 98 -> NULL
free sequence = free 100 -> free 95 -> free 98 -> NULL
```

```
int val;

do {
        printf("점수를 입력하세요(입력 종료 -1) : ");
        scanf("%d", &val);

        if (val == -1) break;

        //단위 구조체를 이용하여 하나의 노드를 위한 메모리 할당
        if ((node = (unit *)malloc(sizeof(unit))) == NULL) {
                printf("메모리 할당에 문제가 있습니다.");
                exit(1);
        }
        //메모리가 할당된 노드에 입력 받은 점수를 저장
        node->score = val;
        node->next = NULL;

        head = list; //head에 전체 list의 시작 정보를 저장

        if (list == NULL) {
                //list에 어떠한 node도 연결되지 않은 시작 시점
                list = node;
        } else {
                //list에 연결된 node가 존재한다면 list의 마지막까지
진행
                while (list->next != NULL) {
                        list = list->next;
                }
                //list의 마지막에 현재 만들어진 노드를 연결
                list->next = node;
        }
} while (1);

printf("\nlist sequence = ");
list = head;
while (list != NULL) {
        printf("%d -> ", list->score);
        list = list->next;
}
puts("NULL");
```

```
        //list의 마지막까지 진행하며 시작 노드부터 메모리 해제
        printf("free sequence = ");
        while (head != NULL) { //while (head->next != NULL)
                node = head;   //head의 시작 node를 node에 저장
                head = head->next; //head의 다음 node를 시작 node로 변경
                printf("free %d -> ", node->score);
                free(node);       //현재 node를 free()
                node = NULL;
        }
        puts("NULL");
        return 0;
}
```

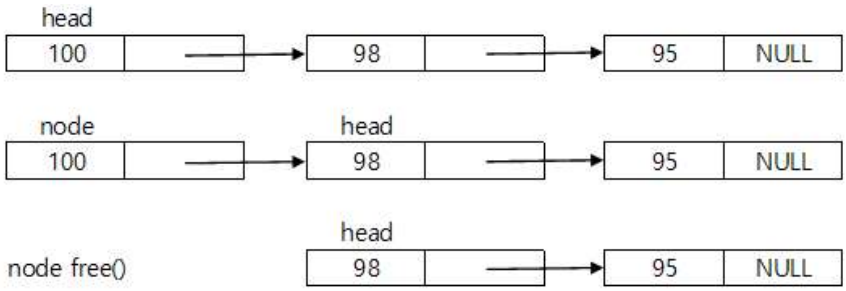
새로운 노드 연결을 위한 프로그램 코드를 세부적으로 살펴보면, 리스트의 마지막은 NULL로 저장되어 있으므로 while (list->next != NULL)의 조건을 이용해 연결 리스트의 마지막까지 찾아갑니다. 이 후 NULL이 저장되어 있던 list->next에 현재 만들어진 node를 연결하는 것입니다.

```
while (list->next != NULL) {
        list = list->next;
}
//list의 마지막에 현재 만들어진 노드를 연결
list->next = node;
```

메모리 해제를 위한 프로그램 코드의 동작 방식은 [그림 10.3]과 같습니다. 그림에서처럼 head가 가리키고 있는 첫 노드부터 하나씩 free()를 해나가다 보면 마지막 노드까지 할당 받았던 모든 메모리를 해제 할 수 있게 됩니다.

```
        //list의 마지막까지 진행하며 시작 노드부터 메모리 해제
        while (head != NULL) { //while (head->next != NULL)
                node = head;   //head의 시작 node를 node에 저장
                head = head->next; //head의 다음 node를 시작 node로 변경
                free(node);       //현재 node를 free()
```

```
        node = NULL;
}
```



[그림 10.3] free() 동작 방식

10.5 연결 리스트 응용

연결 리스트를 사용 하는 프로그램에서는 메모리 할당, 리스트 연결, 리스트 출력, 메모리 해제 등 각각의 기능을 구분하여 함수로 처리해 사용하는 것이 편리합니다. 함수를 사용할 때 중요한 것은 아래의 [표 10.2]에서 설명했듯이 동적 메모리 영역을 사용하는 함수의 호출에서는 반드시 메모리 영역의 주소를 리턴을 받아 사용해야 한다는 것입니다. 왜냐하면 프로그램이 수행되는 영역은 스택(stack) 영역이고, 동적 메모리를 할당하는 곳은 힙(heap) 영역이라서 call by reference 형태로 함수를 사용하는 것은 불가능하기 때문입니다.

[표 10.2] 동적 메모리 할당 시 함수의 사용

가능	불가능

//동적 메모리 영역을 사용하는 함수의 호출에서는 반드시 메모리 영역의 주소를 리턴을 받아 사용해야 함
 	
//가능한 함수 원형
	//call by reference 형태로 함수를 사용하는 것은 불가능
	
	//불가능한 함수 원형
	void allocation(int val, unit *item);

unit * allocation(int val);
 	
//가능한 함수 호출
unit *node = allocation(val);
	
	//불가능한 함수 호출
	allocation(val, node);

[예제 10.9] 함수 호출을 이용한 연결 리스트 작성

```
#include <stdio.h>
#include <stdlib.h>

struct unit {
        int score;
        struct unit *next;
};
typedef struct unit unit;

unit *allocation(int val) //void allocation(int val, unit *item) 형태는 불가능
{
        unit *item;

        if ((item = (unit *)malloc(sizeof(unit))) == NULL) {
                printf("메모리 할당에 문제가 있습니다.");
                exit(1);
        }
        item->score = val;
        item->next = NULL;
        return item;
}

unit *connect(unit *list, unit *item)
{
        unit *head = list; //list의 처음 시작을 저장

        if (head == NULL) {
                head = item;
                return head;
        }
        while (list->next != NULL) {
                list = list->next;
```

```
점수를 입력하세요(입력 종료 -1) : 100
점수를 입력하세요(입력 종료 -1) : 95
점수를 입력하세요(입력 종료 -1) : 98
점수를 입력하세요(입력 종료 -1) : -1

list sequence = 100 -> 95 -> 98 -> NULL
free sequence = free 100 -> free 95 -> free 98 -> NULL
```

```
        }
        list->next = item;
        return head;
}

void print_list(unit *list)
{
        unit *node = list;

        printf("\nlist sequence = ");
        while (list != NULL) {
                printf("%d -> ", list->score);
                list = list->next;
        }
        puts("NULL");
}

unit *free_list(unit *head)
{
        unit *node;

        printf("free sequence = ");
        while (head != NULL) {
                node = head;
                head = head->next;
                printf("free %d -> ", node->score);
                free(node);
                node = NULL;
        }
        puts("NULL");
        return head;
}

int main(int argc, char *argv[])
{
        unit *head = NULL, *node = NULL, *list = NULL;
        int val;

        do {
                printf("점수를 입력하세요(입력 종료 -1) : ");
```

```
            scanf("%d", &val);

            if (val == -1) break;

            node = allocation(val); //allocation(val, node); 형태는 불가능
            head = connect(head, node);
        } while (1);

        if (head != NULL) print_list(head);

        head = free_list(head);
        return 0;
}
```

단위 구조체의 멤버를 다양하게 선언하면 어떤 하나의 정보가 아닌 여러 개의 정보를 포함하는 연결 리스트를 만들 수 있습니다. [예제 10.10]에서는 학생의 이름과 성적을 입력 받아 리스트를 구현하는 예를 수행해 보겠습니다.

이 예제를 수행하기 위해서는 단위 구조체의 멤버에 학생이름을 저장할 수 있는 항목을 추가하고 학생 이름 즉, 문자열을 처리하는 부분을 넣어주면 됩니다.

```
struct unit {
        char *name; //포인터로 선언하여 동적 메모리 할당을 받아 사용
        int score;
        struct unit *next;
};
typedef struct unit unit;

unit *item->name = (char *)malloc(sizeof(char) * (strlen(name)+1));

strcpy_s(item->name, strlen(name)+1, name);
```

이 때 학생 이름을 저장하는 변수 또한 포인터로 선언하여 동적 메모리

할당을 받아 사용합니다. 문자열은 마지막에 널을 포함해야 하므로 “문자열의 길이+1” 만큼의 동적 메모리 공간을 확보합니다. 또한 학생 이름이 저장되는 변수 name은 문자열이기 때문에 <string.h>에서 제공하는 strcpy(), strlen(), strcmp() 등 문자열 관련 함수를 사용합니다.

[예제 10.10] 연결 리스트 응용

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

//단위 구조체
struct unit {
        char *name;
        int score;
        struct unit *next;
};
typedef struct unit unit;

unit *allocation(char *name, int val)
{
        unit *item;

        if ((item = (unit *)malloc(sizeof(unit))) == NULL) {
                printf("메모리 할당에 문제가 있습니다.");
                exit(1);
        }
        item->name = (char *)malloc(sizeof(char) * (strlen(name) + 1));
        strcpy(item->name, name);
        item->score = val;
        item->next = NULL;
        return item;
}

unit *connect(unit *head, unit *item)
{
        unit *savehead = head; //list의 처음 시작을 저장
```

```
이름을 입력하세요(입력 종료 enter) : benny
점수를 입력하세요 : 100
이름을 입력하세요(입력 종료 enter) : daniel
점수를 입력하세요 : 95
이름을 입력하세요(입력 종료 enter) : joon
점수를 입력하세요 : 98
이름을 입력하세요(입력 종료 enter) :

list sequence = benny(100) -> daniel(95) -> joon(98) -> NULL
free sequence = benny(100) -> daniel(95) -> joon(98) -> NULL
```

```
        if (head == NULL) {
                head = item;
                return head;
        }
        while (head->next != NULL) {
                head = head->next;
        }
        head->next = item;
        return savehead;
}

void print_list(unit *head)
{
        printf("\nlist sequence = ");
        while (head != NULL) {
                printf("%s(%d) -> ", head->name, head->score);
                head = head->next;
        }
        puts("NULL");
}

unit *free_list(unit *head)
{
        unit *node;

        printf("free sequence = ");
        while (head != NULL) {
                node = head;
                head = head->next;
                printf("%s(%d) -> ", node->name, node->score);
                free(node->name);
                free(node);
                node = NULL;
        }
        puts("NULL");
        return head;
}

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

```
        unit *head = NULL, *node = NULL, *list = NULL;
        char name[20], score[10];

        do {
                printf("이름을 입력하세요(입력 종료 enter) : ");
                gets(name);

                if (*name == '\0') break;

                printf("점수를 입력하세요 : ");
                gets(score);

                node = allocation(name, atoi(score)); //allocation(val, node);
불가능
                head = connect(head, node);
        } while (1);

        if (head != NULL) print_list(head);

        head = free_list(head);
        return 0;
}
```

10장에서는 동적 메모리 할당과 관련된 내용을 학습 했습니다. 개념 확인 학습, 적용 확인 학습, 응용 프로그래밍을 통해 지금까지 학습한 내용을 점검해 보시기 바랍니다.

□ 개념 확인 학습

1. 정적 메모리 할당 방식과 동적 메모리 할당 방식의 의미는 무엇인가요?

2. 동적 메모리를 할당 받아 메모리를 사용할 때의 장점은 무엇인가요?

3. 정적 메모리 할당 방식과 동적 메모리 할당 방식을 사용할 때 사용되는 메모리 영역의 차이점은 무엇인가요?

4. 동적 메모리 할당에 관련된 함수들은 어떤 것들이 있나요?

5. 할당 받은 동적 메모리의 사용이 완료되었으면 메모리를 반납해야 하는 이유는 무엇인가요?

6. 동적 메모리의 사용할 때 연결 리스트가 사용되는 이유는 무엇인가요?

7. 동적 할당을 받아 메모리를 사용하는 연결 리스트에서 단위 구조체가 하는 역할은 무엇인가요?

8. 동적 할당을 받아 메모리를 사용하는 연결 리스트의 단위 구조체에서 자기 참조 구조체를 사용하는 이유는 무엇인가요?

9. 동적 할당을 받아 메모리를 사용하는 연결 리스트의 단위 구조체에서 정수 배열이나 문자 배열을 포인터로 사용하려면 어떻게 해야 할까요?

10. 동적 할당을 받은 메모리를 사용하는 선형 연결 리스트에서 마지막 데이터를 알 수 있는 방법은 무엇입니까?

11. 동적 할당을 받은 메모리를 사용하는 선형 연결 리스트에서 메모리를 해제 하는 방법을 설명하세요.

12. 동적 할당을 받은 메모리를 사용하는 선형 연결 리스트를 원형 연결 리스트로 변경하는 방법을 설명하세요.

□ 적용 확인 학습

1. 다음 서술 내용이 바르면 O, 그렇지 않으면 X를 표시 하세요.

A. 동적 할당된 메모리는 스택(stack) 영역의 메모리를 할당 받는다. ( )

B. 함수 malloc()과 calloc()은 반환형이 요구한 자료의 포인터 형이다. ( )

C. 함수 realloc()은 이미 확보한 저장 공간의 크기에 늘리고자 하는 크기를

합한 만큼의 메모리 영역을 할당한다. ( )

D. 한번 동적으로 할당된 메모리는 함수 realloc()으로도 다시 확장할 수 없다. ( )

E. 함수 free()의 인자는 해제할 메모리 공간의 주소 값을 갖는 포인터이다. ( )

F. 함수 free()를 사용하여 할당 받은 메모리를 해제 할 때에는 할당 받는 모든 메모리 공간을 순차적으로 모두 해제 해 주어야 한다. ( )

G. 자료를 순차적으로 저장하기 위한 방법으로 배열 또는 연결 리스트를 사용할 수 있다. ( )

H. 구조체는 자기 자신의 포인터를 멤버로 하는 항목을 가질 수 있다. ( )

I. 연결 리스트로 이용하고자 하는 구조체는 여러 자료들과 자기 자신의 포인터가 멤버로 구성된다. ( )

J. 동적으로 메모리가 할당된 연결 리스트의 노드들은 연속된 저장 공간에 저장된다. ( )

K. 연결 리스트는 원소인 단위 구조체의 노드(node)가 순차적으로 연결된 자료구조이다. ( )

L. 연결리스트에서 헤드(head)는 항상 첫 번째 노드를 가리키는 포인터로 이용하는 것이 편리하다. ( )

M. 연결 리스트의 마지막 노드가 가지고 있는 링크에 NULL이 저장되면 그

연결 리스트는 하나의 선처럼 연결된다. ( )

N. 연결 리스트의 마지막 노드가 가지고 있는 링크에 첫 번째 노드가 저장되면 그 연결 리스트는 하나의 원처럼 연결된다. ( )

2. 다음에 서술된 문장에 어울리는 프로그램 코드를 완성해 보세요.

A. 한 개의 정수형 자료를 저장할 수 있도록 동적 메모리를 할당 받아 그곳에 값을 저장합니다.

---

*num = 9;

B. 세 개의 정수형 자료를 저장할 수 있도록 동적 메모리를 할당 받아 그곳에 값을 저장합니다.

---

*(nums+0) = 9;
*(nums+1) = 8;
*(nums+2) = 7;

C. 위 B에서 할당받은 메모리를 다섯 개의 정수형 자료를 저장할 수 있도록 동적 메모리 공간을 확장합니다.

D. 한 개의 문자열(최대 20문자) 자료를 저장할 수 있도록 동적 메모리를 할당 받아 그곳에 “apple”을 저장합니다.

char *str;

_____

_____

E. 위 D에서 저장된 문자열을 아래에 선언된 fruit 변수에 복사합니다.

char *fruit;

_____

_____

F. 아래와 같이 문자열 한 개와 실수 두 개를 요소로 하는 데이터 단위로 연결 리스트를 만들려고 합니다. 단위 구조체를 완성하세요.

```
struct unit {
    char *name;
    double *nums;
    ____________________________
};
typedef struct unit unit;
```

G. 위 F번의 단위 구조체를 이용하여 하나의 노드를 만들어 메모리를 할당받는 문장을 완성하세요. name에는 최대 20문자가 저장될 수 있습니다.

unit *node;

_____

_____

_____

□ 응용 프로그래밍

1. 데이터의 개수를 사용자로부터 입력 받아, 그 숫자만큼 double형 데이터를 입력 받아 동적 메모리를 할당하여 저장하고 출력하는 프로그램 작성하세요. (힌트 : malloc() 또는 calloc()을 이용합니다.)

```
double형 데이터의 개수 입력 : 3
double형 데이터 입력 : 1.1
double형 데이터 입력 : 2.2
double형 데이터 입력 : 3.3

동적 메모리에 저장된 값은 =   1.1   2.2   3.3
```

2. 위 1번의 프로그램을 연결 리스트 형태로 수정하고 내용을 출력하세요.

```
double형 데이터의 개수 입력 : 3
double형 데이터 입력 : 1.1
double형 데이터 입력 : 2.2
double형 데이터 입력 : 3.3

연결 리스트의 값은 =   1.1   2.2   3.3
```

3. 이름(문자열)과 전화번호(문자열)를 저장하여 연결 리스트로 사용할 수 있는 단위 구조체를 정의하세요.

4. 위 3번의 정의를 이용하여 입력 받은 이름과 전화번호를 하나의 단위로

하는 연결 리스트를 만들려고 합니다. 아래의 함수를 완성하고 이용하여 다음 그림과 같이 동작하도록 프로그램을 구현해 보세요.

A. unit *create_node(char *name, char *tel) //노드를 생성하는 함수
B. unit *append(unit *head, unit *item) //리스트의 마지막에 노드를 추가하는 함수
C. unit *print_list(unit *head) //연결 리스트의 모든 노드 출력 함수
D. unit *free_list(unit *head) //리스트 전체 메모리를 해제하는 함수

```
이름을 입력하세요(입력 종료 enter) : benny
전화번호를 입력하세요 : 010-111-1111
이름을 입력하세요(입력 종료 enter) : daniel
전화번호를 입력하세요 : 010-222-2222
이름을 입력하세요(입력 종료 enter) : jhon
전화번호를 입력하세요 : 010-333-3333
이름을 입력하세요(입력 종료 enter) :

list sequence = benny(010-111-1111) -> daniel(010-222-2222) -> jhon(010-333-3333) -> NULL

free sequence = benny(010-111-1111) -> daniel(010-222-2222) -> jhon(010-333-3333) -> NULL
```

5. 위 4번의 main() 함수에서 이름을 입력하면 연결 리스트의 내용을 검색하여 전화번호를 알려주는 함수 get_tel()을 추가하여 다음 그림과 같이 동작하도록 프로그램을 구현해 보세요.

E. void get_tel(unit *head, char *question, char *answer) //question에 해당하는 name의 전화번호 tel를 answer에 반환하는 함수

```
이름을 입력하세요(입력 종료 enter) : benny
전화번호를 입력하세요 : 010-111-1111
이름을 입력하세요(입력 종료 enter) : daniel
전화번호를 입력하세요 : 010-222-2222
이름을 입력하세요(입력 종료 enter) : joon
전화번호를 입력하세요 : 010-333-3333
이름을 입력하세요(입력 종료 enter) :

list sequence = benny(010-111-1111) -> daniel(010-222-2222) -> joon(010-333-3333) -> NULL

이름을 입력하시면 전화번호를 드립니다(입력 종료 enter) : daneil
찾으시는 전화번호 = 존재하지 않습니다.

이름을 입력하시면 전화번호를 드립니다(입력 종료 enter) : daniel
찾으시는 전화번호 = 010-222-2222

이름을 입력하시면 전화번호를 드립니다(입력 종료 enter) :

free sequence = benny(010-111-1111) -> daniel(010-222-2222) -> joon(010-333-3333) -> NULL
```

6. 위 5번에 사용자에게 입력받은 파일명에 리스트의 모든 내용을 출력하는 함수 save_file()을 추가하여 다음 그림과 같이 동작하도록 프로그램을 구현해 보세요.

F. int save_file(unit *head, char *filename) //filename에 해당하는 파일에 리스트의 모든 내용을 저장하는 함수, 파일 저장이 성공하면 1을 그렇지 않으면 0을 반환

```
이름을 입력하세요(입력 종료 enter) : benny
전화번호를 입력하세요 : 010-111-1111
이름을 입력하세요(입력 종료 enter) : daniel
전화번호를 입력하세요 : 010-222-2222
이름을 입력하세요(입력 종료 enter) : joon
전화번호를 입력하세요 : 010-333-3333
이름을 입력하세요(입력 종료 enter) :

list sequence = benny(010-111-1111) -> daniel(010-222-2222) -> joon(010-333-3333) -> NULL

이름을 입력하시면 전화번호를 드립니다(입력 종료 enter) : daniel
찾으시는 전화번호 = 010-222-2222

이름을 입력하시면 전화번호를 드립니다(입력 종료 enter) : raina
찾으시는 전화번호 = 존재하지 않습니다.

이름을 입력하시면 전화번호를 드립니다(입력 종료 enter) : joon
찾으시는 전화번호 = 010-333-3333

이름을 입력하시면 전화번호를 드립니다(입력 종료 enter) :

리스트 전체 내용을 저장할 파일이름을 입력하세요 : tel.txt
tel.txt 파일을 열어보세요.

free sequence = benny(010-111-1111) -> daniel(010-222-2222) -> joon(010-333-3333) -> NULL
```

□ 본 보고서/교재는 과학기술정보통신부 및 정보통신기획평가원에서 주관하여 진행한 결과물입니다. (과제번호:20180002160031001)