Chapter 2 Dynamic Memory Management in C

📌 C Program은 `runtime system`에 의해 실행된다 그리고 그 환경은 `OS`가 제공한다.
📌 C언어의 memory allocation과 deallocation은 프로그램의 `효율성`과 `유연성`을 제공한다.

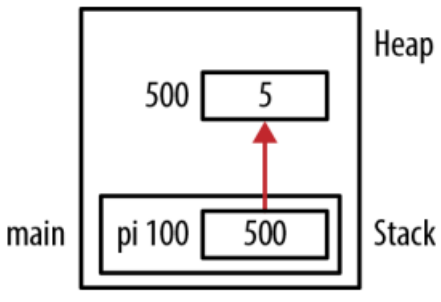
Dynamic Memory Allocation

Dynamic memory allocation의 basic concept

1. `malloc`을 사용해서 memory를 할당하는 행위
2. `malloc`을 사용한 메모리를 사용하는 행위
3. `free`를 사용해서 memory를 해제하는 행위

malloc && 작동원리

```c
int *ptr = (int*)malloc(sizeof(int));
*ptr = 5;
printf("*ptr : %d\n", *ptr);
free(ptr);
```



왜 sizeof연산자를 사용하나요?

- 자료형의 사이즈는 memory model에 따라 다를 수 있고, OS에 따라 다양하기 때문에 `sizeof` 연산자를 사용하는 것이 좋다.

👀흔히 하는 실수!!

```c
int *ptr;
*ptr = (int*)malloc(sizeof(int)); // Fail

ptr = (int*)malloc(sizeof(int)); // Good
```

Memory Leaks

개념 : Memory Leak은 프로그램이 동적으로 할당한 메모리를 해제하지 않고 사용하는 상황을 말한다.

원인

1. Memory address가 lost되는 경우
2. `free`가 실패하는 경우 (hidden leak)
   영향
3. heap메모리에 사용가능한 메모리가 줄어든다.

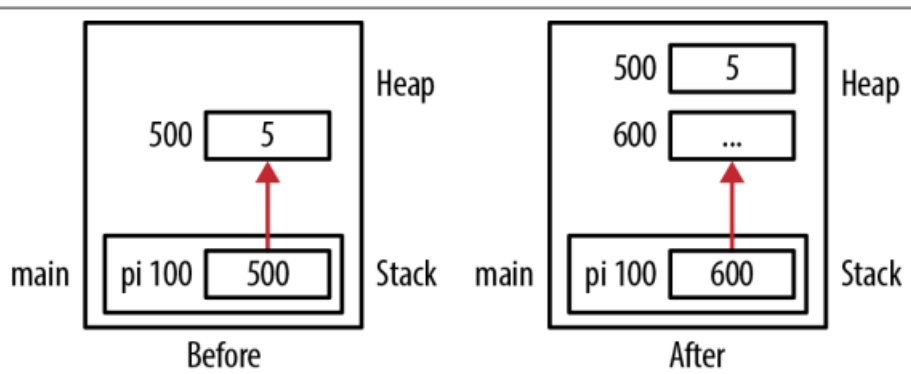
Losing address

Case1

- 첫번째 할당했던 동적 메모리에 다시 할당을 한 상황이 발생되어서 처음에 할당한 메모리를 추적하지 못하는 상황.

```c
#include <stdio.h>

#include <stdlib.h>



int main(void)
```

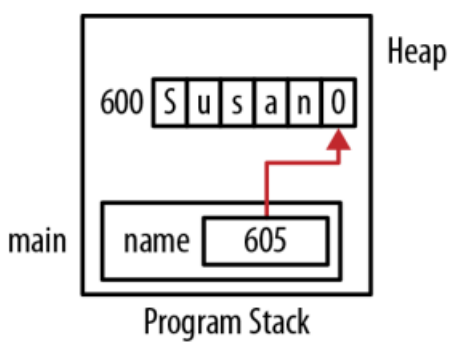
```
{

    int *ptr = (int*)malloc(sizeof(int));
    *ptr = 5;

    ptr = (int*)malloc(sizeof(int));
    *ptr = 4;
    return (0);

}
```



Case 2

- string을 출력하는 상황에서 memory 자체를 움직이면서 string을 출력하는 경우

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

int main(void)

{

    char *name = (char*)malloc(sizeof("Susan") + 1);
    strcpy(name, "Susan");
    printf("I am memory safe print : %s\n",name);
    while (*name)
    {
        printf("%c", *name);
        name++;
    }
    return (0);
}
```



Hidden memory Leak

정의 : 사용자가 heap memory에 더 이상 필요하지 않은 object를 free하지 않고 memory에 남겨놓는 행위 (가장 많이 하는 실수)

Dynamic Memory Allocation Functions

Function	Description
`malloc`	Allocates memory from the heap
`realloc`	Reallocates memory to a larger or smaller amount based on a previously allocated block
`calloc`	Allocates and zeros out memory from the heap
`free`	Returns a block of memory to the heap

```
int *pi = (int*)malloc(sizeof(int));
```

1. size에 맞는 메모리가 heap에 할당이 된다.

2. Memory는 free되지 않는다면 변경되지 못한다.
3. 주소의 첫번째 byte를 return한다.

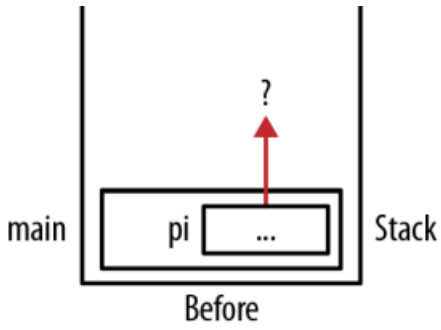
```
int *pi = (int*) malloc(sizeof(int));
if(pi != NULL) {
// Pointer should be good
} else {
// Bad pointer
}
```

- Dynamic memory은 `heap`에 `연속적인 메모리`로 할당된다. allocation function은 가장 낮은 주소를 return한다.

Falling to allocate memory

- pointer를 선언을 하고나서 할당은 하지 않는 경우 address가 포인트 한 값은 garbage가 들어간다.

```
int *pi;

printf("%d", *pi);
```



---

malloc && static, global pointer ⭐

```
static int *ptr = malloc(sizeof(int)); // Warning
```

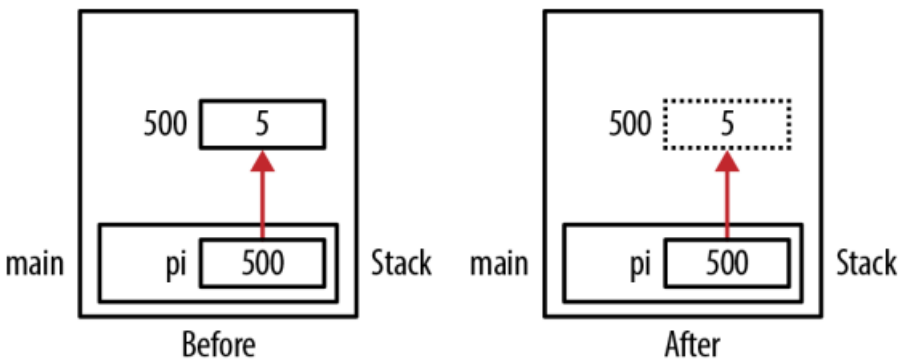
```
#include <stdio.h>
#include <stdlib.h>

int *ptr; // Good!

int main(void)
{

    ptr = (int*)malloc(sizeof(int));
    *ptr = 4;
    printf("%d", *ptr);
    return (0);
}
```

- static && global variable은 컴파일시 초기화가 된다 (Data영역에 존재함)
- `malloc`은 execute되면서 실행되는 function이다. 즉 런타임에 동적으로 메모리를 할당하는 함수이다. 이에 전역변수 컴파일 시점에 malloc으로 초기화를 하면 안된다. (컴파일 Error)
- Compiler 시점에서 initialize에 사용되는 `=`연산자와 assignment에 사용되는 `=`는 다르게 본다.

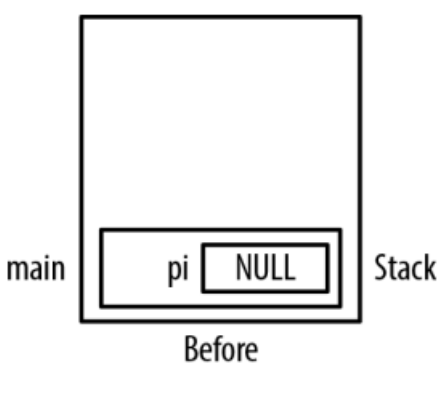
Deallocation Memory Using Free Function

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

```
int *ptr = (int*)malloc(sizeof(int));
...
free(pi);
```




```
int *ptr = (int*)malloc(sizeof(int));
...
free(ptr);
pi = NULL;
```



- free를 호출하게 된다면 memory는 heap에 return된다.
- pointer변수는 할당된 주소를 계속 pointing하게 된다. (나중에 reallocate가능하다)
- 이는 `Dangling Pointer`의 문제가 생길 수 있다.
- free를 한 포인터 변수는 주소를 계속 point를 하고 있기 때문에 unexpected behavior를 야기할 수 있다. 그래서 NULL으로 다시 할당을 해야한다!

Example Code

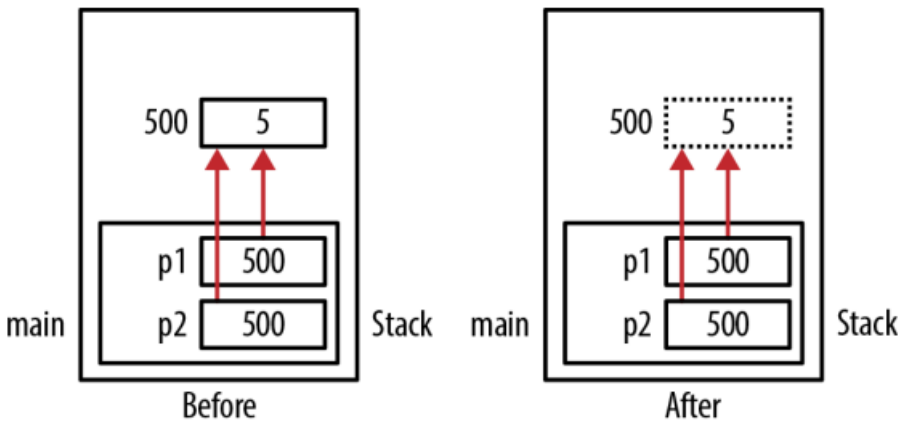
```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *ptr = (int*)malloc(sizeof(int));
    printf("%p\n", ptr);
    free(ptr);
    printf("%p", ptr);
    return (0);
}

Result
000001d5e1a613f0
000001d5e1a613f0
```

Double free

정의 : 이미 heap 영역에서 `free`가 된 memory블록을 다시 `free`하려고 하는 행위를 의미함.

- 이미 메모리 해제가 된 영역에 `free`를 다시 하려고 하면 `runtime-error`가 발생한다.

```
int *ptr = (int*)malloc(sizeof(int));
*ptr = 5;
free(ptr);

free(ptr);
```



OS와 Free

1. 운영체제의 역할

- **메모리 관리**: 프로그램이 실행되는 동안, 운영체제는 프로그램의 메모리를 관리합니다. 프로그램이 종료되면, 운영체제는 이 메모리를 다른 프로그램에서 사용할 수 있도록 재할당합니다.
- **메모리 상태**: 프로그램이 종료되었을 때, 그 메모리의 상태(정상 또는 손상)는 문제가 되지 않습니다. 프로그램이 비정상적으로 종료되었을 때 메모리가 손상되었더라도, 운영체제가 이를 회복하고 다른 프로그램에 재사용할 수 있습니다.

2. 메모리 해제의 필요성

- **프로그래머의 책임**: 비록 운영체제가 프로그램 종료 시 메모리를 자동으로 회수하지만, 메모리를 명시적으로 해제하는 것이 좋은 프로그래밍 습관입니다.
- **품질 보증**: 메모리 누수 검사 도구를 사용할 때, 메모리를 해제하면 도구의 출력 결과가 깔끔해질 수 있습니다. 즉, 프로그램 종료 전에 메모리를 해제하면 메모리 누수 문제를 미리 발견하고 해결할 수 있습니다.

3. 기타 고려사항

- **운영체제의 차이**: 일부 간단한 운영체제에서는 메모리를 자동으로 회수하지 않을 수 있습니다. 이런 경우, 프로그램이 종료 전에 메모리를 해제하는 것이 중요할 수 있습니다.
- **미래의 코드 추가**: 프로그램 종료 시점에 메모리를 해제하지 않으면, 나중에 프로그램에 새로운 코드가 추가될 때 이전 메모리 문제로 인해 문제를 일으킬 수 있습니다.

4. 문제와 고려 사항

- **비용과 복잡성**: 종료 전에 모든 메모리를 해제하는 작업은 시간이 많이 걸리고 복잡할 수 있으며, 추가적인 코드가 필요할 수 있습니다.
- **프로그램의 크기와 실행 시간**: 메모리 해제를 위한 추가 코드는 프로그램의 크기를 증가시키고 실행 시간을 늘릴 수 있습니다.
- **프로그래밍 오류**: 메모리 해제를 제대로 하지 않으면 더 많은 프로그래밍 오류가 발생할 수 있습니다.

결론

- **애플리케이션 별**: 메모리를 종료 전에 해제해야 하는지는 애플리케이션의 특성과 요구 사항에 따라 달라질 수 있습니다. 간단한 프로그램에서는 종료 시 메모리 해제가 필수가 아닐 수 있지만, 복잡한 애플리케이션에서는 명시적으로 메모리를 해제하는 것이 바람직할 수 있습니다.

결국, 메모리를 종료 시 해제하는 것이 항상 필요하지는 않지만, 메모리 관리의 좋은 습관으로 고려할 수 있으며, 각 애플리케이션의 요구 사항과 환경에 따라 적절히 결정해야 합니다.

Dangling Pointer

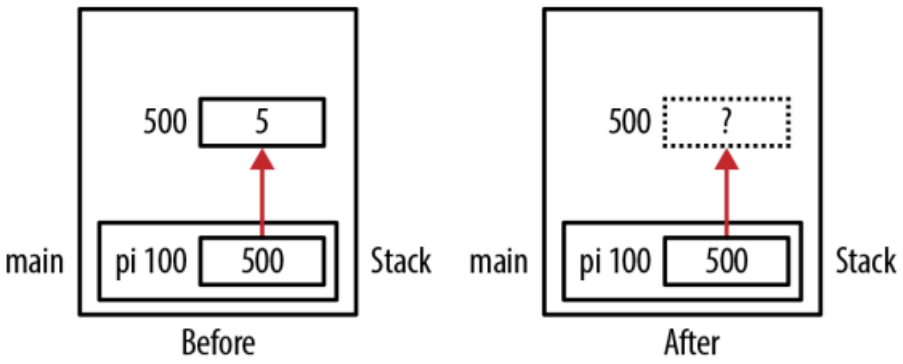
정의 : original memory가 free가 되었지만 pointer가 여전히 그 메모리 주소를 point를 하는 경우

문제점

1. memory에 접근할 때 Unpredictable behavior가 일어날 수 있다.
2. security risk가 발생한다.
3. Segment Fault가 발생할 수 있다.

Case 1

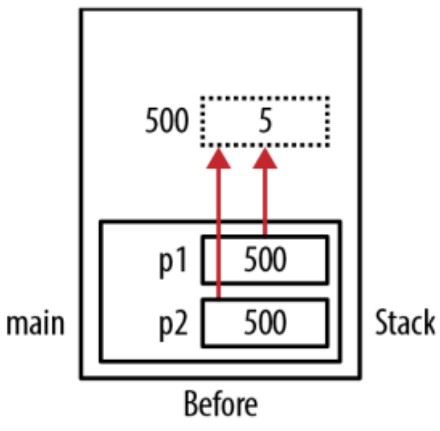
```c
#include <stdio.h>
#include <stdlib.h>

int main(void)
{

    int *ptr = (int*)malloc(sizeof(int));
    printf("%p\n", ptr);
    free(ptr);
    printf("%p", ptr);
    return (0);
}
```



- 포인터 변수 ptr는 여전히 free된 address를 가지고 있다.

Case 2

```
int *p1 = (int*)malloc(sizeof(int));
*p1 = 5;

int *p2;
p2 = p1;

free(p1);
...
*p2 = 10; Dangling Pointer !!
```



- p1과 p2모두 같은 메모리를 가르키고 있다.
- `aliasing` 이라고 지칭한다.

⭐Aliasing이란??

- 2개 이상의 포인터 또는 변수 이름이 동일한 메모리 주소를 참조하는 상태를 의미한다.