Chapter 6) Pointers and structure

구조체의 특성

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

typedef struct person
{
    char *firstName;
    char *lastName;
    char *title;
    unsigned int age;
} Person;

int main()
{

    // using . operator
    Person p = {NULL, NULL, NULL, 0};
    p.age = 3;
    p.firstName = (char*)malloc(strlen("Emily") + 1);
    strcpy(p.firstName, "Emily");
    p.age = 23;

    // using -> operator
    Person *ptr_p = (Person*)malloc(sizeof(Person));
    ptr_p->age = 23;
    ptr_p->firstName = (char*)malloc(strlen("Daddy") + 1);
    strcpy(ptr_p->firstName, "Daddy");

    // using dereferencing
    Person  *ptr_p2 = (Person*)malloc(sizeof(Person));
    (*ptr_p).age = 23;
    (*ptr_p).firstName = (char*)malloc(strlen("asd") + 1);
    strcpt((*ptr_p).firstName, "Daddy");

    return (0);
}
```

1. 구조체는 instance operator `.`을 기본적으로 사용한다.
2. pointer로 선언을 하게 되면 point-to operator `->`를 사용한다.
3. pointer로 선언해도 dereference를 사용하게 된다면 `.`을 사용할 수 있다.

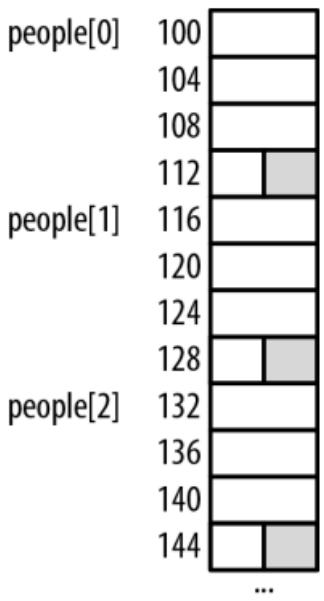
How memory is Allocated for a Structure

- 구조체가 메모리에 할당이 될 때, 개별 필드(요소)의 minimun size의 합이다.
- size가 개별 요소의 합보다 더 커지는 경우가 있다(padding)
- 패딩은 구조체 내 맴버들을 특정 바이트 경계에 정렬하기 위해 추가하는 `빈 공간`을 말한다.
- CPU는 메모리를 특정 크기의 단위로 읽고 쓰기 때문에 데이터를 정렬하면 더 빠르게 접근할 수 있다.
- Padding의 크기는 맴버 변수의 순서를 변경함으로 최소화 할 수 있다.

```c
typedef struct _alternatePerson{
    char *first;
    char *lastName;
    char *little;
    short age;
}Person;
```

```c
optimizing structure memory

#include <stdio.h>

typedef struct Example
{
    char a;
    int b;
    char c;
}Example;
```

```
typedef struct Optimize
{
    int a;
    char b;
    char c;
}Optimize;


int main()
{
    printf("%d\n", sizeof(Example));
    printf("%d", sizeof(Optimize));
    return (0);
}

result
12
8
```



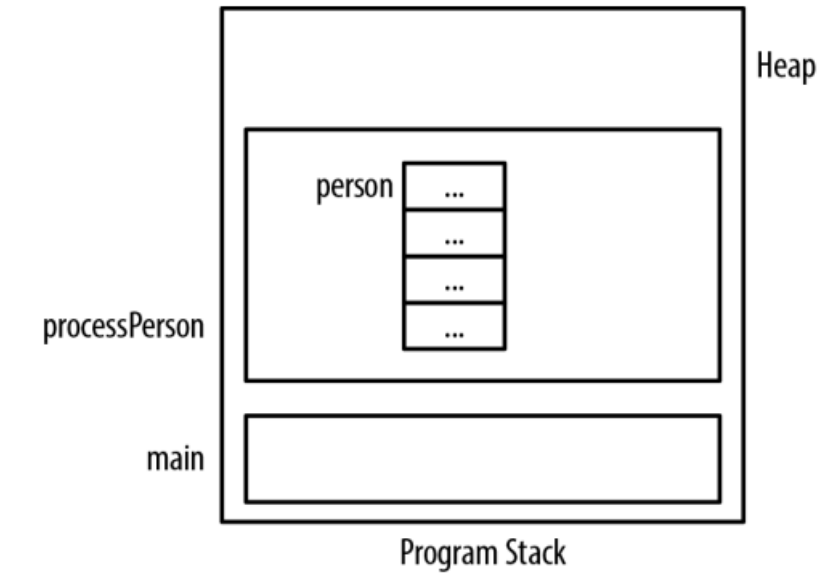
👀회색 부분이 Padding부분이다.

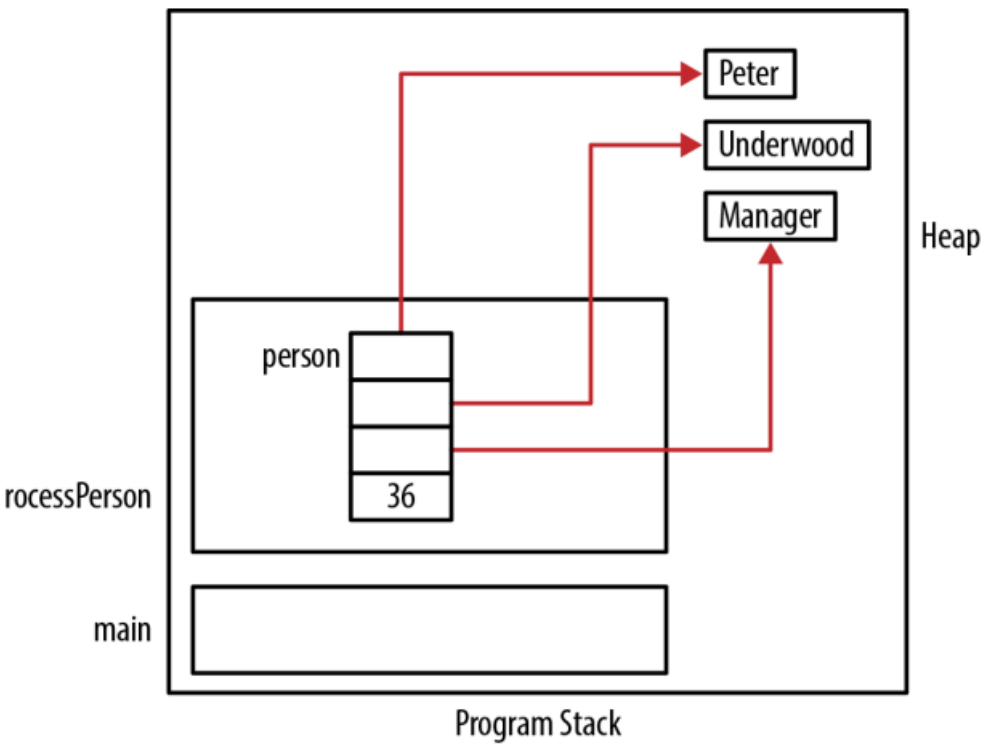
Structure And deallocating

```
#include <stdio.h>

typedef unsigned int uint;

typedef struct _person {
 char* firstName;
 char* lastName;
 char* title;
 uint age;
} Person;


void initializePerson(Person *person, const char* fn,
 const char* ln, const char* title, uint age)
 {
 person->firstName = (char*) malloc(strlen(fn) + 1);
 strcpy(person->firstName, fn);
 person->lastName = (char*) malloc(strlen(ln) + 1);
 strcpy(person->lastName, ln);
 person->title = (char*) malloc(strlen(title) + 1);
 strcpy(person->title, title);
 person->age = age;
}



void deallocatePerson(Person *person) {
 free(person->firstName);
 free(person->lastName);
 free(person->title);
}



void processPerson()
```

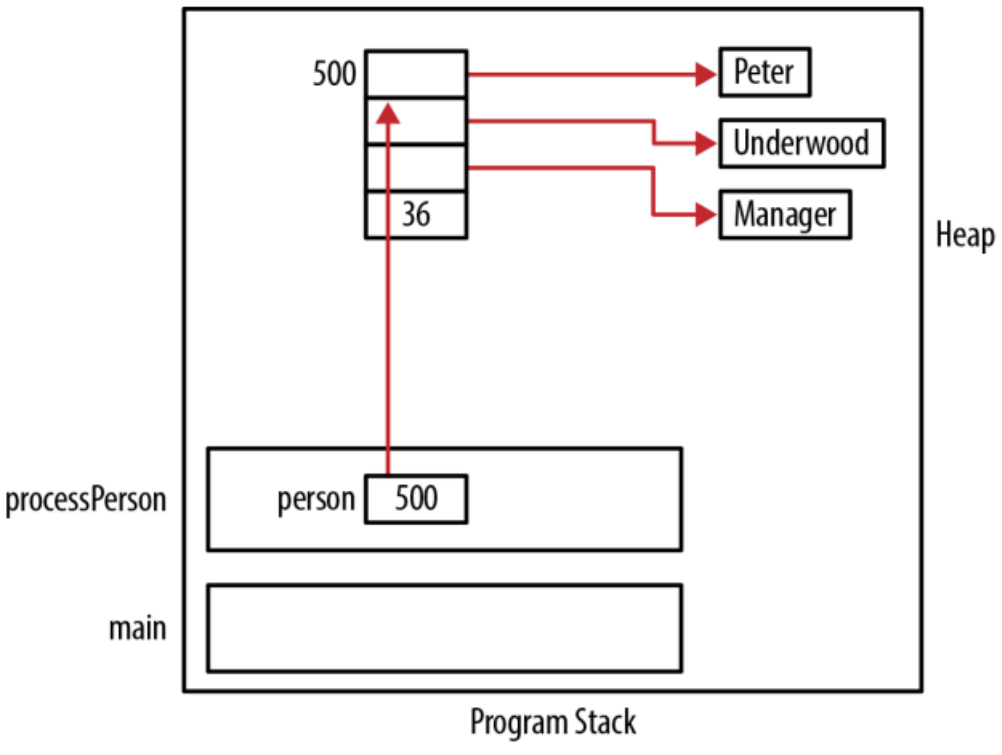
```
{
    /* 선언을 stack으로 하는 경우
    Person person;
    initializePerson(&person, "Peter", "Underwood", "Manager", 36 );
    deallocatePerson(&person);
    */

   Person *ptrPerson;
   ptrPerson = (Person*) malloc(sizeof(Person));
   initializePerson(ptrPerson, "Peter", "Underwood", "Manager", 36);
   deallocatePerson(ptrPerson);
   free(ptrPerson);
}


int main()
{
    processPerson();
    return (0);

}
```

Stack으로 선언한 경우





Pointer로 선언한 경우



- C Compiler가 자동적으로 memory를 free하지 않는 것 처럼, 구조체에 있는 영역도 직접 free를 해줘야한다.
- 직접 Initialize함수도 만들어줘야하고, deallocate함수도 만들어줘야한다. 대부분의 Object-Orient programming들은 (C++, Python) 자동적으로 이런 행위들을 해준다.
- 이런 구조체의 반복적인 allocation과 deallocation은 `malloc/free Overhead`를 야기한다.