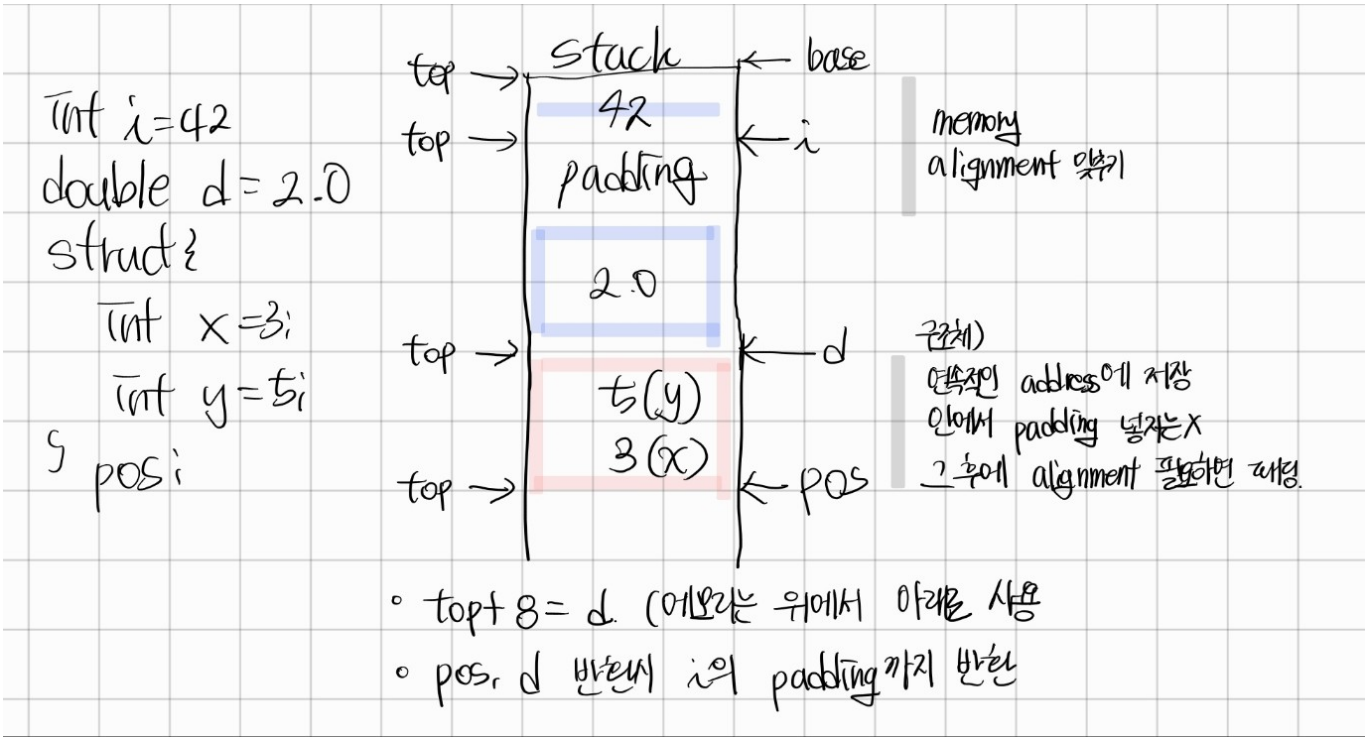
Stack

- stack에는 function call과 automatic variable을 저장한다.
- 메모리 주소의 위에서 아래로 할당한다. (top이 메모리 주소의 아래에 존재)
- LIFO

Stack 할당 과정



automatic variable

- 함수의 지역변수이며, 사용되기 전에 stack에 할당되어 있다. (함수 호출시 지역변수 모두 할당)
- 해당 function 끝날 때까지 stack에 유지한다.
- 프로그래머들은 해당 변수의 위치나 순서를 예측하지 못한다. (컴파일러의 최적화)
    - 어느 순서든 할당 가능하지만, 구조체의 경우는 순서가 고정되어있다.

function call

- 함수가 네스팅 가능한 경우, 한번에 하나의 함수만 호출할 수 있다.
- 과정

    1. 함수의 위치로 jump
    2. 해당 함수 실행
    3. 함수가 끝난 후 호출부로 jump
    - 여기서 지역변수를 할당하거나, 또다른 함수를 부르거나 등 여러 복잡성 존재

--> 이러한 경우를 처리하기 위해 함수는 stack frame이 필요하다.

- stack에서 함수 호출시 해당 함수공간을 할당하기 위해 사용한다.

Stack Frames

A Stack frame holds information for a single function invocation. (단일 함수 호출)

- Saved processor registers (rsp, rbp..)
- 함수의 지역변수 저장
- 매개변수 저장
- 호출부의 주소 저장

Local Variables

함수 호출시 stack의 top부터 저장공간을 만들어, 지역변수를 할당한다.
스택 프레임의 사이즈는 고정된 사이즈이며, 해당 사이즈는 저장되지 않는다. 컴파일러가 따로 관리한다.

Function Argument

함수의 매개변수.
6개까지는 레지스터에 저장하며 6개 이상은 스택에 저장한다.
레지스터에 저장한 변수가 더 빠른 접근성을 가지기 때문에 가능하면 6개 이하로 지정하기..

-> 만약 함수 매개변수가 스택에 저장된다면, 반대 순서로 push된다.

Program Counter

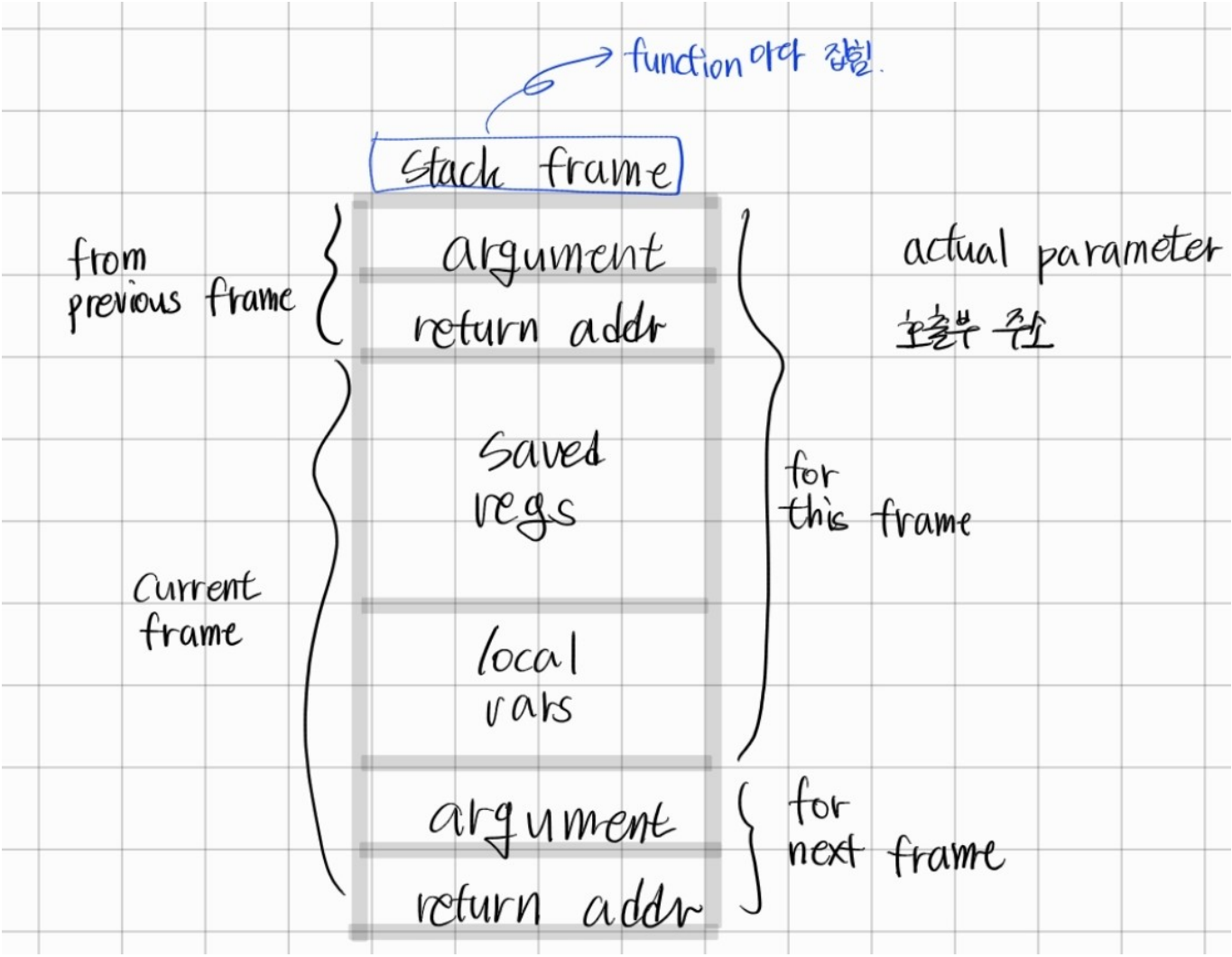
함수 호출시 해당 PC값을 저장한다. 함수를 처리한다. 함수 종료시 PC값을 통해 호출부로 리턴한다. **PC**는 현재 실행중인 프로세스의 machine instruction 주소이다.

Summary

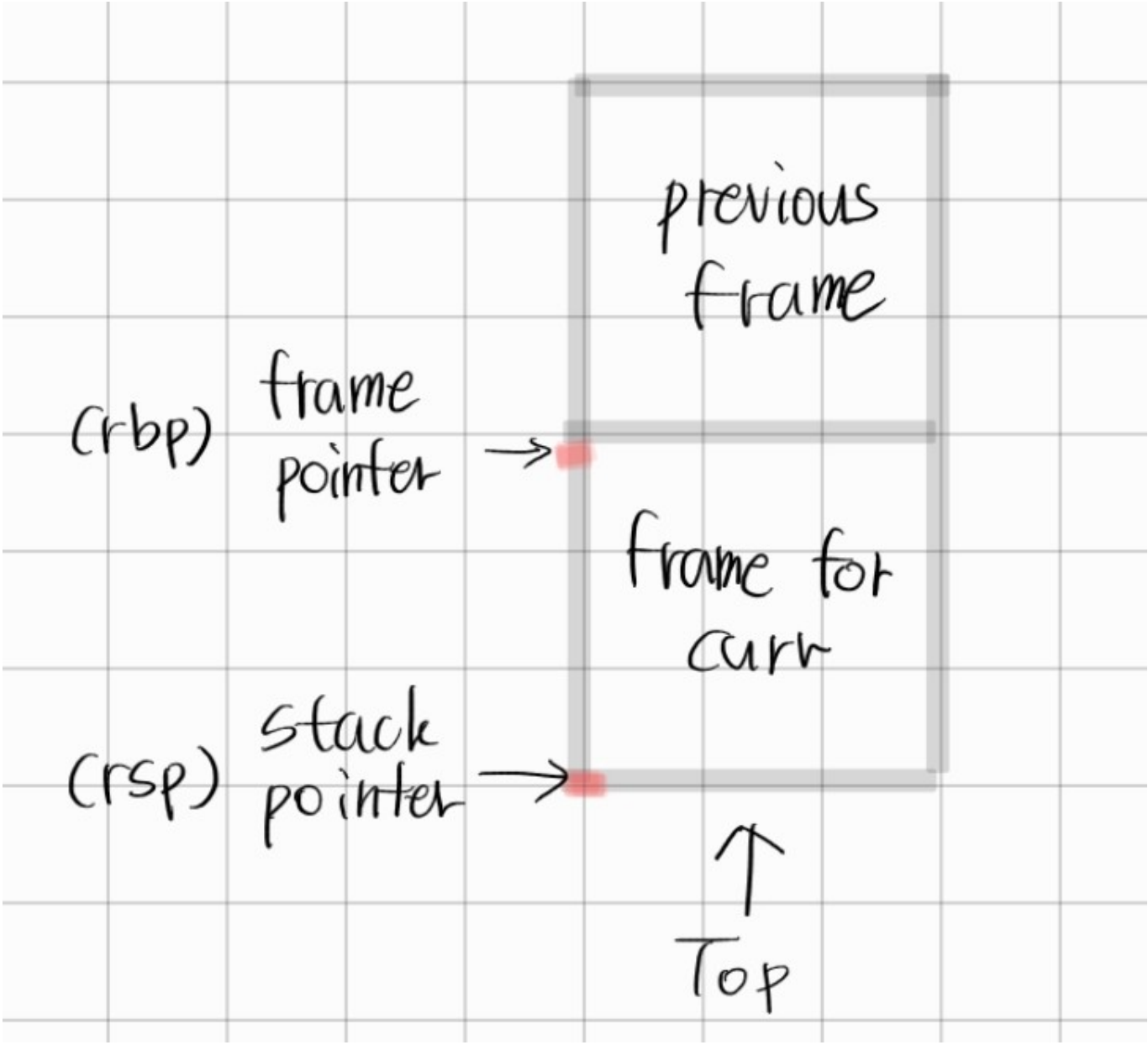
1. Automatic variables are allocated on the stack
2. The stack grows downward
3. Items removed from the stack are not cleared
4. Stack frames track function calls (함수 호출시 해당 함수의 stack frame 생성함)

Example Stack Frame

A Stack Frame



saved register
현재 함수의 상태를 저장하는 레지스터다.

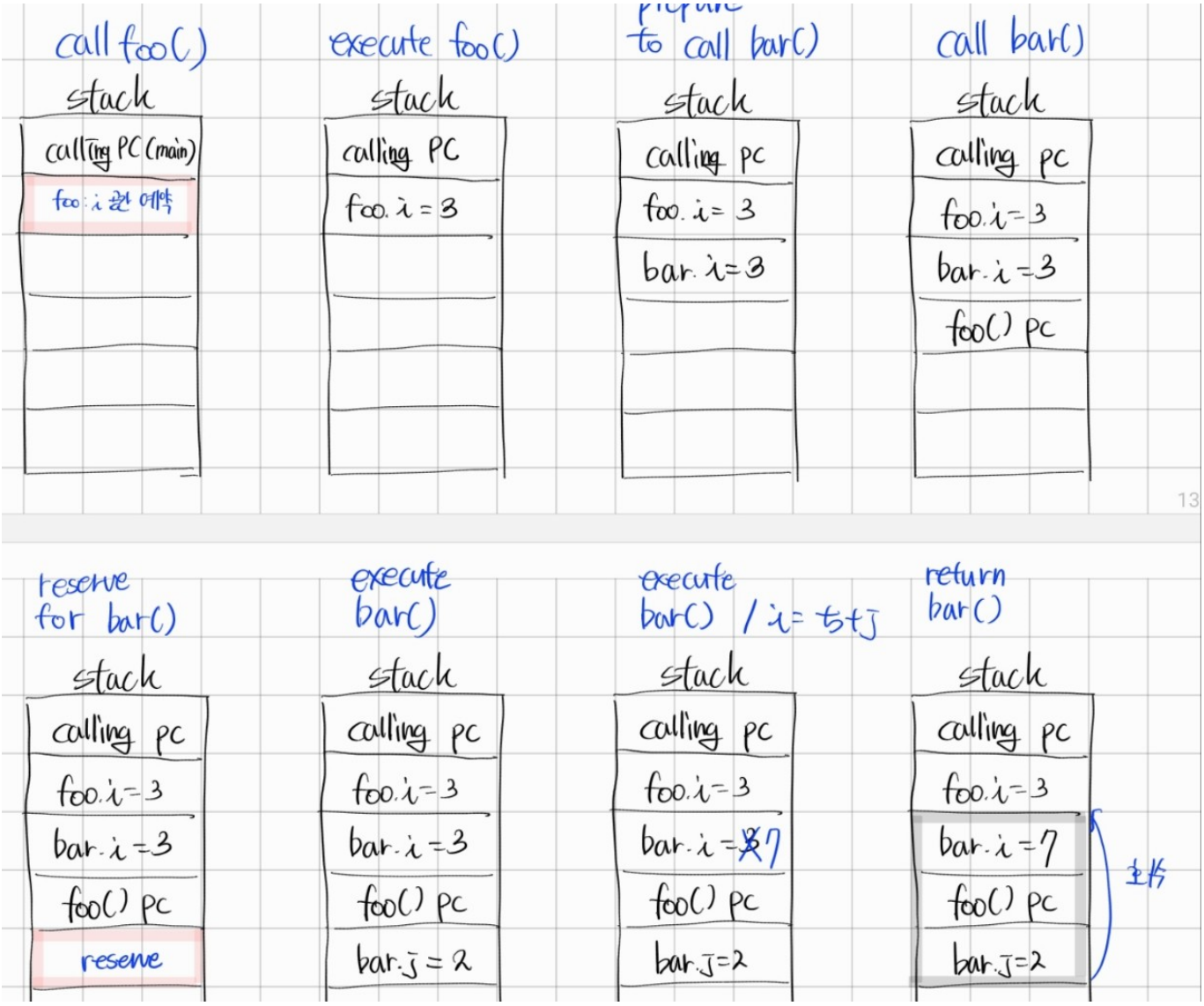


- rbp: 스택의 시작점 (base pointer)
- rsp: 스택의 꼭대기

A call stack

```
int main(){
    ...
    foo()
    ...
}

void foo(){
    int i=3;
    bar(i);
    ...
}

void bar(int i){
    int j=2;
```

```
    i=5+j;
}
```



1. 호출부의 PC값 저장
2. 지역변수를 위한 공간 예약
3. 변수 할당
4. 함수를 호출하기 위한 매개변수 할당
5. 해당 함수 호출 준비를 위해 호출부의 PC값 저장
6. 지역변수를 위한 공간 예약
7. 변수 할당
8. 함수 실행 (매개변수의 변경)
9. 함수 호출시 지역 + 매개변수 포함하여 리턴

Code

▶ Stack.c

```c
#include <stdio.h>

// size is 8, 4 + 1, then round to multiple of 4 (int's size),
struct stu_a {
```

```
5       int i; // 4 bytes
6       char c; // 1 byte
7       // char pad[3]; // 3 bytes padding
8   };
9
10  // size is 16, 8 + 1, then round to multiple of 8 (long's size),
11  struct stu_b {
12      long l; // 8 bytes
13      char c; // 1 byte
14      // char pad[7]; // 7 bytes padding
15  };
16
17  // size is 24, l need padding by 4 before it, then round to multiple of 8 (long's s
18  struct stu_c { //가장 큰 long 기준으로 allignment (8byte) --> 변수 선언 위치가 중요하다
19      int i; // 4
20      // char pad[4]; // 4 bytes padding after int var
21      long l; // 8
22      char c; // 1
23      // char pad[7]; // 7 bytes padding
24  };
25
26  // size is 16, 8 + 4 + 1, then round to multiple of 8 (long's size),
27  struct stu_d {
28      long l; // 8
29      int i; // 4
30      char c; // 1
31      // char pad[3];
32  };
33
34  // size is 16, 8 + 4 + 1, then round to multiple of 8 (double's size),
35  struct stu_e {
36      double d; // 8
37      int i; // 4
38      char c; // 1
39      // char pad [3];
40  };
41
42  // size is 24, d need align to 8, then round to multiple of 8 (double's size),
43  struct stu_f {
44      int i; // 4
45      // char pad[4];
46      double d; // 8
47      char c; // 1
48      // char pad[7];
49  };
50
51  // size is 4,
52  struct stu_g {
53      int i; //4
54  };
55
56  // size is 8,
57  struct stu_h {
58      long l; // 8
59  };
60
61  // test - padding within a single struct, 실제 스택에 들어간 주소
```

```c
int test_struct_padding() {
    printf("구조체 size\n");
    printf("%s: %ld\n", "stu_a", sizeof(struct stu_a));
    printf("%s: %ld\n", "stu_b", sizeof(struct stu_b));
    printf("%s: %ld\n", "stu_c", sizeof(struct stu_c));
    printf("%s: %ld\n", "stu_d", sizeof(struct stu_d));
    printf("%s: %ld\n", "stu_e", sizeof(struct stu_e));
    printf("%s: %ld\n", "stu_f", sizeof(struct stu_f));

    printf("%s: %ld\n", "stu_g", sizeof(struct stu_g));
    printf("%s: %ld\n", "stu_h", sizeof(struct stu_h));

    return 0;
}

// test - address of struct,

int test_struct_address() {
    printf("\n구조체 주소\n");
    printf("%s: %ld\n", "stu_g", sizeof(struct stu_g));
    printf("%s: %ld\n", "stu_h", sizeof(struct stu_h));
    printf("%s: %ld\n", "stu_f", sizeof(struct stu_f));

    struct stu_g g;
    struct stu_h h;
    struct stu_f f1;
    struct stu_f f2;
    int x = 1;
    long y = 1;

    printf("address of %s: %p\n", "g", &g);
    printf("address of %s: %p\n", "h", &h);
    printf("address of %s: %p\n", "f1", &f1);
    printf("address of %s: %p\n", "f2", &f2);
    printf("address of %s: %p\n", "x", &x);
    printf("address of %s: %p\n", "y", &y);

    // g is only 4 bytes itself, but distance to next struct is 16 bytes(on 64 bit
    printf("space between %s and %s: %ld\n", "g", "h", (long)(&h) - (long)(&g));

    // h is only 8 bytes itself, but distance to next struct is 16 bytes(on 64 bit
    printf("space between %s and %s: %ld\n", "h", "f1", (long)(&f1) - (long)(&h));

    // f1 is only 24 bytes itself, but distance to next struct is 32 bytes(on 64 bi
    printf("space between %s and %s: %ld\n", "f1", "f2", (long)(&f2) - (long)(&f1))

    // x is not a struct, and it reuse those empty space between structs, which exi
    printf("space between %s and %s: %ld\n", "x", "f2", (long)(&x) - (long)(&f2));
    printf("space between %s and %s: %ld\n", "g", "x", (long)(&x) - (long)(&g));

    // y is not a struct, and it reuse those empty space between structs, which exi
    printf("space between %s and %s: %ld\n", "x", "y", (long)(&y) - (long)(&x));
    printf("space between %s and %s: %ld\n", "h", "y", (long)(&y) - (long)(&h));

    return 0;
}

```

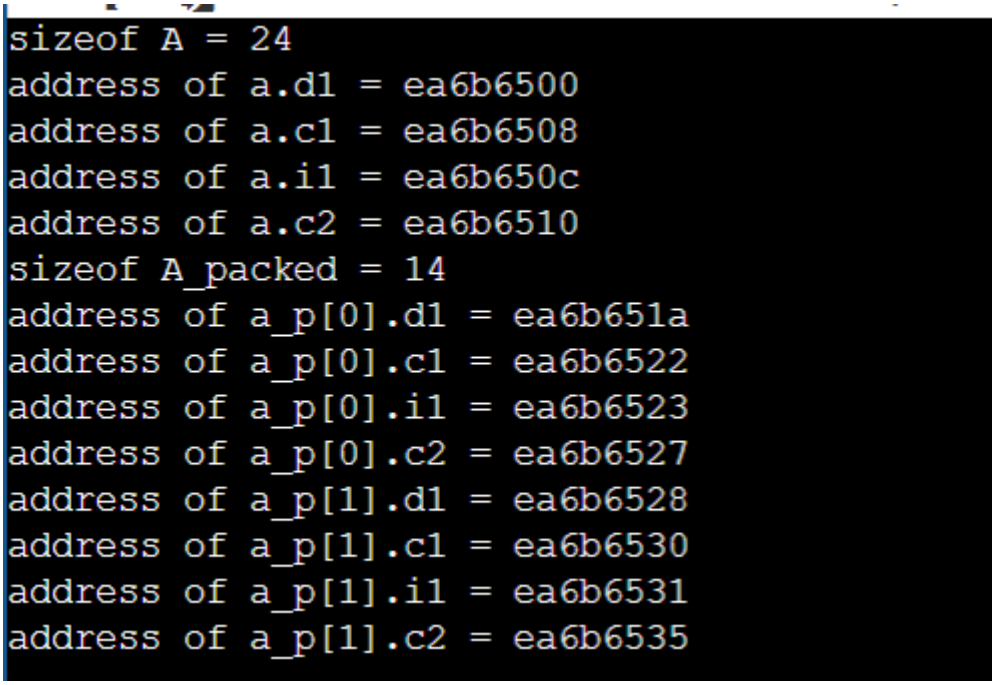
```c
int main(int argc, char * argv[]) {
    test_struct_padding();
    test_struct_address();

    return 0;
}

```

```
구조체 size
stu_a: 8
stu_b: 16
stu_c: 24
stu_d: 16
stu_e: 16
stu_f: 24
stu_g: 4
stu_h: 8

구조체 주소
stu_g: 4
stu_h: 8
stu_f: 24
address of g: 0x7ffd86749f88
address of h: 0x7ffd86749f90
address of f1: 0x7ffd86749fa0
address of f2: 0x7ffd86749fc0
address of x: 0x7ffd86749f8c
address of y: 0x7ffd86749f98
space between g and h: 8
space between h and f1: 16
space between f1 and f2: 32
space between x and f2: -52
space between g and x: 4
space between x and y: 12
space between h and y: 8
```

▶ packking.c

```c
#include <stdio.h>
//#pragma pack(1)

struct A { //8byte 맞춤
    double d1;
    char c1; //1, pad 3 //pad 7
    int i1;// 4 //pad 4 --> 도 가능하나 컴파일러는 c1, i1패킹한듯 / 워드단위로 불러오니까
    char c2;//1, pad 7

```

```
    // double, char, char, int 순서로 선언하면 16byte. char+char+2+int => 8
};


struct __attribute__((packed)) A_packed { //A_packed 속성으로 만들면 메모리 아끼기
    double d1;
    char c1;
    int i1;
    char c2;
};

int main()
{
    struct A a;
    struct A_packed a_p[1];
    //struct A_packed a_p[2]; //두개가 완전히 붙어있음(packing) - c compiler default =


    printf("sizeof A = %ld\n", sizeof(a));
    printf("address of a.d1 = %08x\n", &a.d1);
    printf("address of a.c1 = %08x\n", &a.c1);
    printf("address of a.i1 = %08x\n", &a.i1);
    printf("address of a.c2 = %08x\n", &a.c2);

    printf("sizeof A_packed = %ld\n", sizeof(a_p));

    printf("address of a_p[0].d1 = %08x\n", &(a_p[0].d1));
    printf("address of a_p[0].c1 = %08x\n", &(a_p[0].c1));
    printf("address of a_p[0].i1 = %08x\n", &(a_p[0].i1));
    printf("address of a_p[0].c2 = %08x\n", &(a_p[0].c2));

    printf("address of a_p[1].d1 = %08x\n", &(a_p[1].d1));
    printf("address of a_p[1].c1 = %08x\n", &(a_p[1].c1));
    printf("address of a_p[1].i1 = %08x\n", &(a_p[1].i1));
    printf("address of a_p[1].c2 = %08x\n", &(a_p[1].c2));

    return 0;
}


```



구조체

구조체로 선언하면 패딩 없이 연속적인 메모리로 할당된다. (변수 타입 간 패딩은 존재함)
int와 char 사이 패딩이 없는 이유
-> 워드단위로 불러오기 때문에 컴파일러가 패킹하듯.. (시스템 상이)

```
struct A{
    double d1; //8byte
    char c1; //1byte + padded 3
    int i1; //4byte
    char c2; //1byte + padded 7
}
//24byte

struct A{
    double d1; //8byte
    char c1; //1byte
    char c2; //1byte + padded 2
    int i1; //4byte
}
//16byte

//구조체 변수 선언 순서에 따라 메모리 차이 존재
```

Packed

메모리를 아끼기 위해 패딩 없이 연속으로 붙이기

```
struct __attribute__((packed)) A_packed { //A_packed 속성으로 만들면 메모리 아끼기
    double d1; //8
    char c1; //1
    int i1; //4
    char c2; //1
};
//8+1+4+1 = 14byte
```