17. 구조체(structure), 공용체(union)

• 다른 형의 변수(필드 이름(field name))를 하나의 대표이름(구조체(structure) 변수 이름)으로 참조하는 방법에 대해서 배웁니다.

• 다소 어려워지는 포인터에 관한 주제를 다시 접할 것이며, 구조체를 파라미터로 전달하는 효과적인 방법을 배울 것입니다.

• 구조체가 함수의 파라미터로 전달되거나, 함수가 구조체를 리턴할 때 생기는 **복사 문제**에 대한 해결책에 대해서도 잠시 언급하게 될 것입니다.

• 구조체의 필드 혹은 멤버를 접근하기 위한 새로운 2개의 연산자 . 와 -> 에 대해서도 배우게 될 것입니다.

왜 이것이 필요한가?

• 회원 한사람의 정보를 관리하기 위해 세개의 변수를 사용한다고 가정해 봅시다.

```
char name[10];
short age;
long phone;
```

• name, age 와 phone은 각각 회원의 이름, 나이와 전화번호를 나타내기 위해 사용할 예정입니다. 만약 이러한 회원이 100여명 정도 예상된다면, 프로그램에서 아래처럼 변수 선언을 할 수 있습니다.

```
char name[100][10];
short age[100];
long phone[100];
```

• 다음과 같은 문제점이 존재합니다.

① name[0], age[0]과 phone[0]이 연관되어 있다는 문법적 지시가 필요합니다. 이것은 사소한 문제가 아닙니다. 왜 회원 한 사람을 나타내기 위한 세 개의 변수가 서로 관련이 없는 것처럼 선언되어야 하는가요!

② 100명 이상의 회원은 관리할 수 없습니다. 또한 회원수가 10명이라면, 배열의 나머지 공간은 낭비됩니다.

• 구조체(structure)는 이러한 2개의 문제점에 대해, 모두 해결책을 제시합니다. 즉 **구조체는 서로 다른 형을 가지는 여러 개의 변수들을 1개의 대표이름으로 참조할 수 있는 문법적 구조를 제공**합니다.

문법

struct [tag-name] { 필드선언;[...] } [변수리스트];

- 구조체는 키워드 struct로 시작합니다.
- struct 다음에 구조체의 **태그 이름tag name**을 명시합니다.
- 태그는 형을 정의하는 경우의 선택사항이므로, 생략해도 좋습니다(그래서 이것은 C에서 구조체 이름이 아니라 태그 이름입니다).
- 구조체를 이루는 필드의 선언은 struct 다음의 연속하는 대괄호 사이에 필드 수만큼 명시합니다.
- 필드 선언 후에 변수들의 리스트가 올 수도 있습니다.
- 마지막에 문장의 끝을 나타내는 **세미콜론(;)은 반드시 명시**되어야 합니다. 이것은 구조체가 블록이 아니라, 문장이기 때문입니다.

```
struct _PEOPLE { short age; char name[10]; long phone; } i;
```

• 위의 예는 age,name과 phone을 필드로 가지는 새로운 구조체 형 struct _PEOPLE을 정의(definition)한 것입니다.
• 또한 구조체 struct _PEOPLE 형의 변수 i를 선언(declaration)한 것입니다.

```
struct { short age; char name[10]; long phone; }   i;
```

①　　②

• 위의 예에서는 태그 이름을 생략하였습니다. 그러므로 새로운 형은 정의되지 않았으며, ①이 형 이름에 해당하고, ②가 변수 이름에 해당합니다(변수를 선언하기 위해서 ***형이름 변수이름;*** 형태의 문법을 사용한 것에 주목하세요. 이것은 정수형 변수를 선언하기 위해 ***int i;*** 라고 사용한 것과 같은 형태입니다).

```
struct _PEOPLE { short age; char name[10]; long phone; };
```

• 위의 예에서는 태그 이름은 생략하지 않았고, 변수 이름은 생략한 형태입니다. 이것은 struct _PEOPLE이라는 새로운 형을 정의(type definition)한 것입니다. 이제 형이 정의되었으므로, 변수를 선언하기 위해서 다음과 같이 사용할 수 있습니다.

```
struct _PEOPLE  i;
       ①        ②
```

• ①이 형 이름에 해당하고, ②가 변수 이름에 해당하는 타당한 변수 선언문장입니다.

• '형을 정의하는 것'과 '변수를 선언하는 것'을 분리해 주므로 좋은 습관입니다. 아래의 예를 참고하세요.

```
struct _PEOPLE {
    short age;
    char name[10];
    long phone;
};//struct _PEOPLE형을 정의한다. 보기 좋게 필드를 각각의 줄에 명시한다.
...
struct _PEOPLE i;//struct _PEOPLE형의 변수 i를 선언한다.
```

구조체 멤버 참조 연산자: . 과 ->

• . 과 ->은 각각 **점 연산자(dot operator)**, **화살표 연산자(arrow operator)**라 읽습니다.

• 구조체의 필드를 접근(access)하기 위해 구조체 **멤버 접근 연산자(member access operator) . 과 ->**를 사용합니다.

```
struct _PEOPLE {
    short age;
    char name[10];
    long phone;
};//struct _PEOPLE
struct _PEOPLE i;
```

• 위의 예에서 struct _PEOPLE형의 구조체 i의 멤버 age를 접근하기 위해 다음과 같이 사용할 수 있습니다.

i.age

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

struct _PEOPLE {
    short age;
    char name[10];
    long phone;
};//struct _PEOPLE

void main() {
    struct _PEOPLE i;

    i.age=48;//이런 어느새 50을 바라보네요.
    strcpy(i.name,"seojt");
    i.phone=9408690;
    printf("%d,%s,%ld\n",i.age,i.name,i.phone);
}//main()
```

구조체 포인터가 사용된 경우

```
struct _PEOPLE* i;
```

• 위의 예에서 처럼, 구조체 포인터가 선언된 경우에는 어떻게 멤버를 참조할까요?

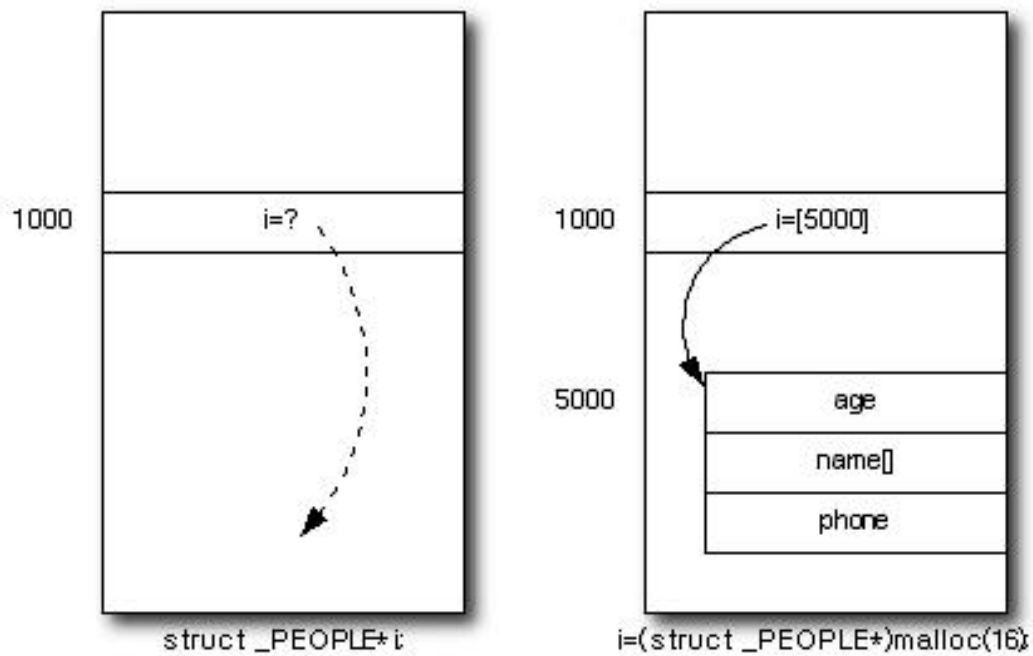
```
i.age=48;
```

• 위 문장에서처럼 .을 사용하는 것은 명백한 잘못입니다. i는 구조체가 아니라, 구조체를 가리키는 포인터이기 때문입니다.
• i에 대해서는 아래와 같이 사용해야 합니다.

```
(*i).age=48;
```

• 구조체 포인터 변수에 대해서 *를 사용할 때, **연산자의 우선순위 문제 때문에 괄호를 반드시 명시해야 합니다**.

```
i=(struct _PEOPLE*)malloc(sizeof(struct _PEOPLE));
(*i).age=48;
```



구조체의 메모리 할당: i는 포인터이므로, 4바이트 할당되었습니다. malloc()에 의해 16바이트(2+10+4)의 메모리가 할당됩니다. 그리고 이곳을 i가 가리키게 됩니다.

malloc()의 리턴 값을 왜 ***(struct _PEOPLE*)***로 형변환(type conversion)해야 하죠?

int* i와 char* i는 다릅니다. i가 같은 값 [1000]을 가지더라도, 덧셈 혹은 뺄셈에 의해 의미가 달라집니다. int* i 인 경우 i+1은 [1004]입니다. 이것은 i+1이 i에서 1만큼 떨어진 정수(interger)를 의미하므로, 컴파일러가 그렇게 해석하는 것입니다. char* i 인 경우 i+1은 [1001]입니다. 이것은 i에서 1만큼 떨어진 문자(character)를 의미합니다. 비록 포인터에 가감 연산이 취해지지 않더라도 컴파일러는 이러한 불법 형 변환에 대해 경고합니다. 만약 프로그램에서 포인터에 대해 가감 연산이 필요 없다면, 단순히 이런 경고를 무시해도 좋습니다. 하지만, 정확한 형 변환은 좋은 프로그램 습관입니다. 위의 예에서 i+1은 i값을 16증가시켜야 할 것입니다. 그러므로, (struct _PEOPLE*)의 형 변환은 반드시 필요합니다.

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

struct _PEOPLE {
    short age;
    char name[10];
    long phone;
};//struct _PEOPLE

void main() {
    struct _PEOPLE* i;
    i=(struct _PEOPLE*)malloc(sizeof(struct _PEOPLE));
    (*i).age=31;
    strcpy((*i).name,"seojt");
    (*i).phone=9408690L;
    printf("%d,%s,%ld\n",(*i).age,(*i).name,(*i).phone);
    free(i);
}//main()
```

• 구조체를 포인터로 사용하는 경우가 많습니다. 그렇다면, i의 멤버 age를 참조하기 위해 다음과 같이 입력하는 것은 번거로운 일입니다.

```
(*i).age
```

• 그래서 특별하게 구조체 포인터에 대해서는 위의 예를 다음과 같이 화살표 연산자를 사용해서 작성할 수 있습니다.

```
i->age
```

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

struct _PEOPLE {
    short age;
    char name[10];
    long phone;
};//struct _PEOPLE

void main() {
    struct _PEOPLE* i;
    i=(struct _PEOPLE*)malloc(sizeof(struct _PEOPLE));
    i->age=31;
    strcpy(i->name,"seojt");
    i->phone=9408690L;
    printf("%d,%s,%ld\n",i->age,i->name,i->phone);
    free(i);
}//main()
```

구조체의 필드를 바라보는 컴파일러의 입장: 상대 주소(offset address)

i.age

- 위 문장에서 표현식의 주소는 컴파일러에 의해 다음과 같이 해석됩니다.

[i의 시작주소]+0

- 이것은 age가 구조체의 첫 번째 멤버이기 때문에 그렇습니다.

i.name

- age가 차지하는 메모리 공간은 2바이트 이므로, 다음과 같이 해석됩니다.

[i의 시작주소]+2

• 컴파일러는 컴파일 시간에 필드의 형(type)을 정확하게 알고 있으므로, 이러한 상대 주소의 계산이 가능합니다.

i.phone

• phone을 접근하는 위 문장은 다음과 같이 주소를 결정할 것입니다.

[i의 시작주소]+12

• 그렇다면, i의 시작 주소는 어떻게 결정되는 것일까요? 만약 구조체가 전역 변수라면, 실행 파일이 메모리에 로드될 때 주소가 결정됩니다. 구조체가 지역 변수라서 스택에 할당된다면, [i의 시작주소] 역시 스택 포인터를 참조하는, 상대 주소로 표현됩니다.

```
{
 int k;
 struct _PEOPLE i;
 ...
}
```

• 위의 경우 [i의 시작주소]는 [stack-pointer+4]로 해석될 것입니다. 왜냐하면, 이미 정수 k가 메모리를 4바이트 차지하고 있기 때문입니다. 물론 k의 주소는 [stack-pointer+0]으로 해석될 것입니다.

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

struct _PEOPLE {
    short age;
    char name[10];
    long phone;
};//struct _PEOPLE

void main() {
    struct _PEOPLE* i;

    i=(struct _PEOPLE*)malloc(sizeof(struct _PEOPLE));

    (*i).age=31;
    strcpy((*i).name,"seojt");
    (*i).phone=9408690L;
```

```
    char* cp=(char*)i;//i의 주소를 복사한다.
    printf("%c\n",*(cp+2));//구조체에서 3번째 문자를 가리킨다.

    free(i);
}//main()
```

- 출력 결과는 다음과 같습니다.

```
s
```

- 이 예에서는 우리가 직접 구조체의 상대 주소를 계산하는 방식을 보여줍니다.
- struct _PEOPLE 형의 p를 초기화하기 위해 다음과 같이 사용합니다.

```
struct _PEOPLE p={31,"seojt",9408690L};
```

- 이것은 p.age, p.name과 p.phone을 각각 주어진 값으로 초기화합니다.

구조체의 전달(passing), 리턴(return)

• 아래의 예에서 struct _PEOPLE을 파라미터로 전달받고, struct _PEOPLE을 리턴하는 함수 IncAge()를 볼 수 있습니다.

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

struct _PEOPLE {
    short age;
    char name[10];
    long phone;
};//struct _PEOPLE

struct _PEOPLE IncAge(struct _PEOPLE p) {
    p.age++;
    return p;
}
```

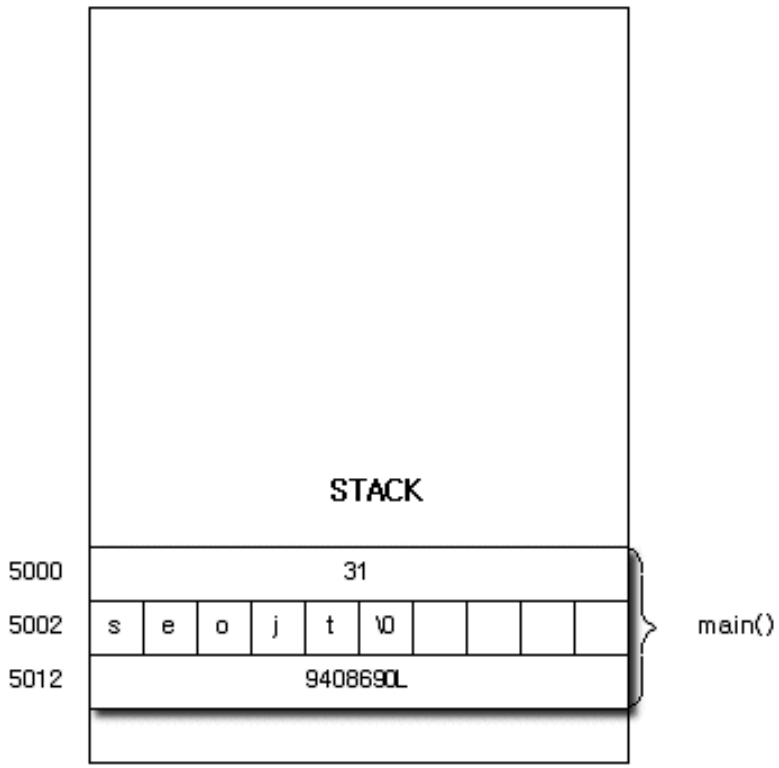
```
void main() {
    struct _PEOPLE p={31,"seojt",9408690L};

    IncAge(p);
    printf("%d\n",p.age);
}//main()
```

• 프로그램의 실행 결과는 다음과 같습니다.

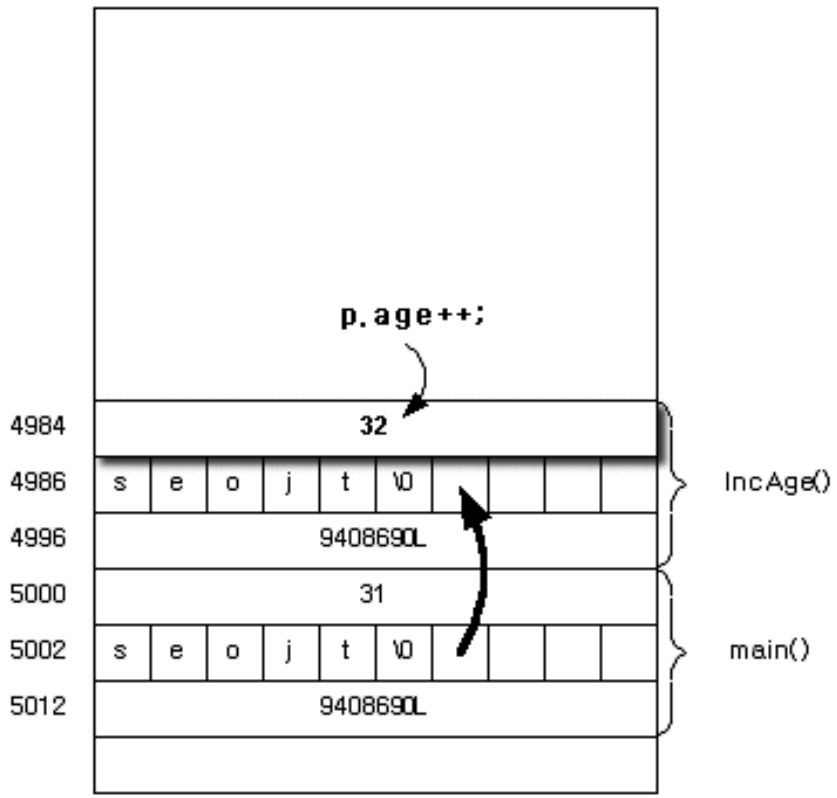
• 아래와 같이 소스를 수정해 봅시다.

```
...
void main() {
    struct _PEOPLE p={31,"seojt",9408690L};

    p=IncAge(p);//임시 구조체의 복사가 일어난다. 비효율적이다.
    printf("%d\n",p.age);
}//main()
```

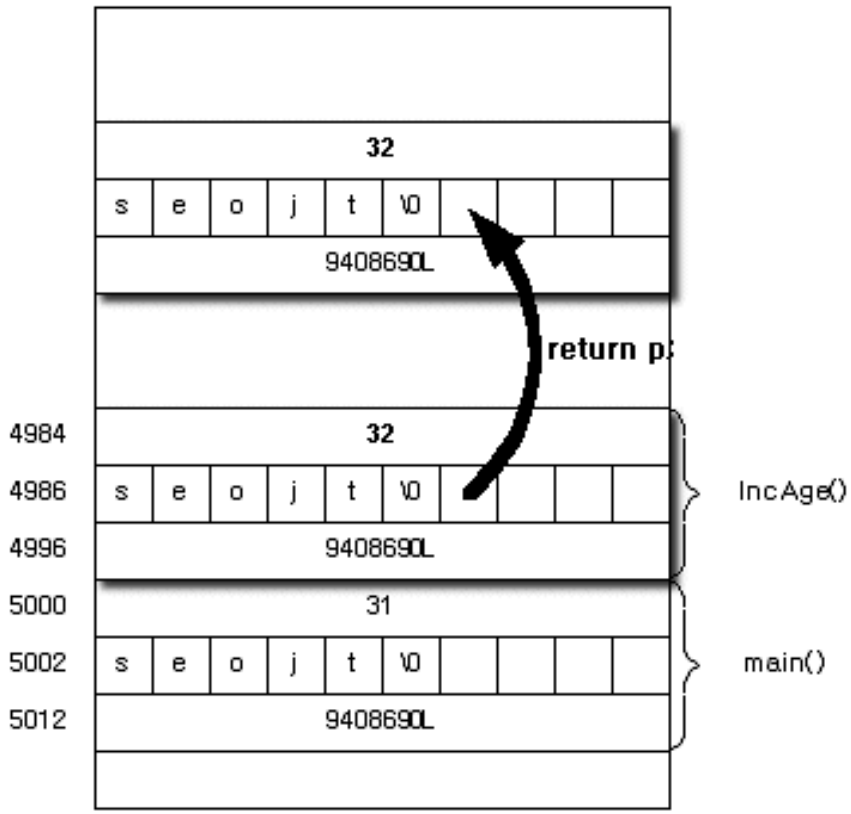
• 문제가 해결된 것처럼 보이지만, 증가된 **메모리 복사(memory copy)의 오버헤드**가 있습니다.



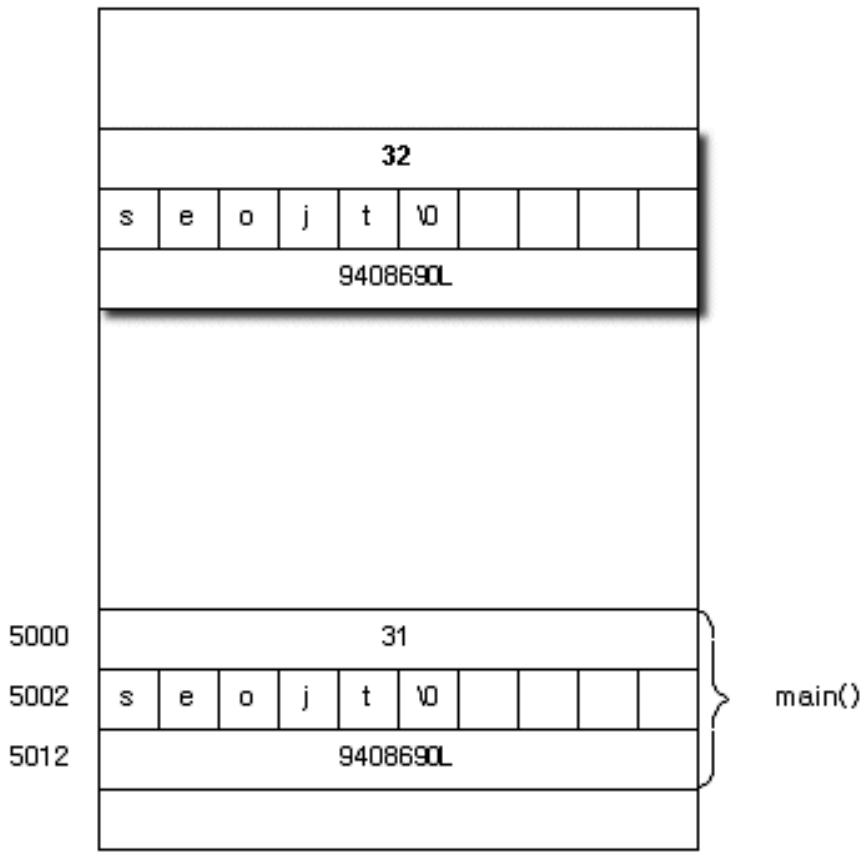
main()에 struct _PEOPLE의 변수 p의 할당: struct _PEOPLE p={31,"seojt",9408690L}; 에 의해 스택에 p가 할당됩니다. name과 phone의 상대 주소가 +2, +12로 결정되었음을 주목하세요.



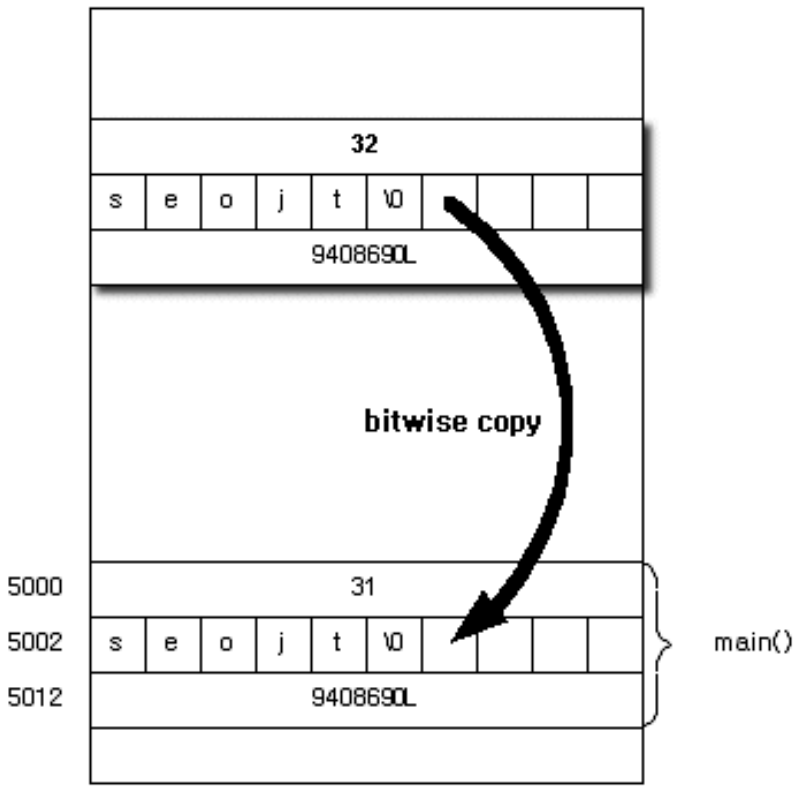
main()의 p가 IncAge()를 위해 스택에 복사됩니다: IncAge()의 호출은 main()의 p를 값에 의한 전달(call by value)로 스택에 복사합니다. IncAge()에서 p.age++; 는 복사된(main()의 p가 아닌) 구조체의 멤버 age를 갱신합니다.



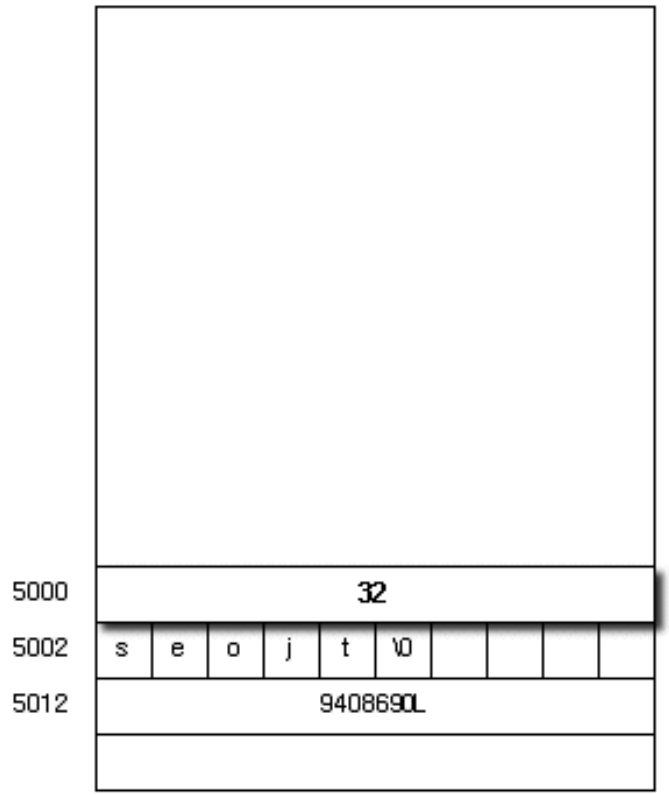
return p; 의 실행: return p; 문장은 구조체 리턴을 위해 임시 구조체를 메모리에 할당합니다.



IncAge()가 리턴했을 때의 메모리 상태: IncAge()가 종료한 후에도 임시 메모리에 할당된 임시 구조체(temporary structure)는 메모리에 남아 있습니다.



대입문에 의해 임시 구조체가 main()의 p에 복사됩니다: p=IncAge(p); 에 사용된 대입문은 IncAge()가 생성한 임시 객체를 main()의 p에 1비트도 틀림없이(bitwise) 복사합니다.



이제 main()의 p가 변경되었습니다: main()의 p.age는 32로 갱신된 것이 확실하지만, 구조체가 여러번 복사되는 오버헤드 때문에 수행능력(performance)가 떨어진 것은 확실합니다.

• 다른 해결방법은 무엇일까요? 이것은 앞 장에서의 포인터를 사용해서 해결한 방법입니다.

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

struct _PEOPLE {
    int age;
    char name[10];
    long phone;
};//struct _PEOPLE

void IncAge(struct _PEOPLE* p) {
    p->age++;
}

void main() {
    struct _PEOPLE p={31,"seojt",9408690L};
```

```
    IncAge(&p);
    printf("%d\n",p.age);
}//main()
```

• 출력 결과는 다음과 같습니다.

```
32
```

• 구조체의 주소를 나타내기 위해 &p를 사용하였습니다. 이것은 그림에서 보듯이 [5000]을 의미합니다. IncAge()의 선언 또한 포인터를 위해 적절히 변경되었습니다.

비트 필드 구조체(bit-field structure)

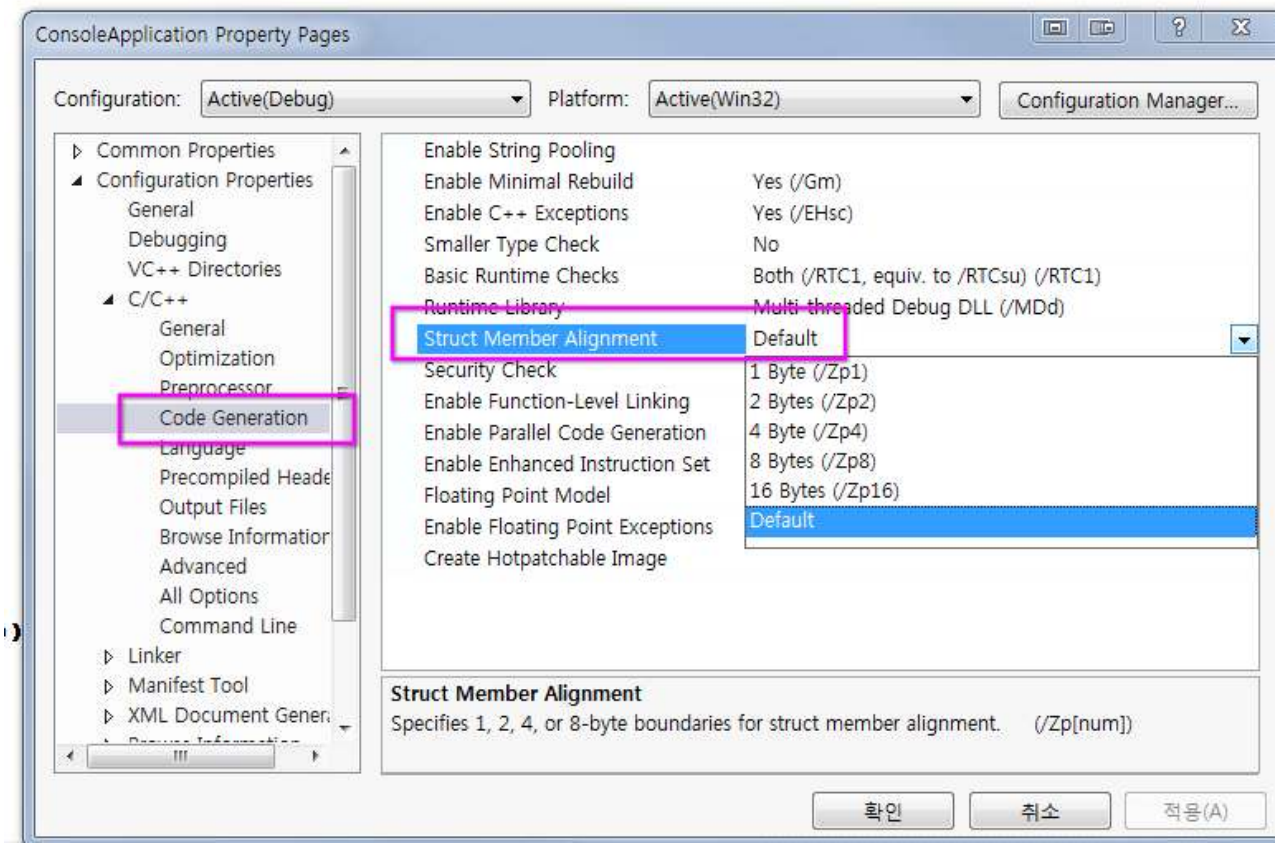
- 구조체의 필드가 사용자에 의해 정의된 임의 크기의 비트로 설정될 수 있습니다.
- 이러한 구조체를 **비트 필드 구조체(bit-field structure)**라 합니다.

```
type-specifier [bitfield-id] : width;
```

- 위의 문법에서 type-specifier 위치에는 정수 호환형인 char, unsigned char, int와 unsigned int만이 올 수 있습니다.
- 콜론(:) 다음에 비트의 크기를 명시합니다.
- bitfield-id에는 비트 필드 변수의 이름을 적습니다. 공간만 확보할 목적이라면, id는 생략해도 좋습니다.

```
struct SBitfield {
    unsigned i : 1;
    unsigned j : 1;
    unsigned k : 2;
};
```

• 위의 예는 필드로 i,j와 k를 가지는 구조체 struct SBitfield를 정의한 것입니다. i,j와 k를 위해 메모리는 각각 1비트,1비트와 2비트가 할당됩니다.



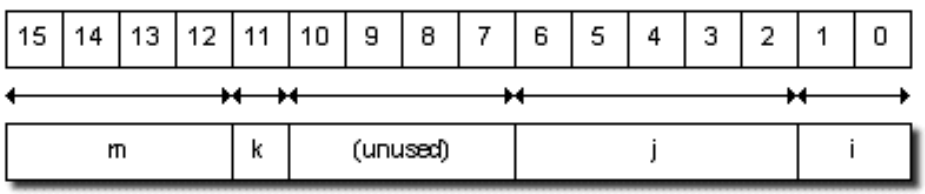
비주얼 스튜디오 2013의 구조체 멤버 정렬 설정: 구조체 멤버 간의 정렬 방식은 컴파일러의 환경설정에서 할 수 있습니다.

• 아래의 예는 비트 필드 구조체가 워드 정렬될 때의 적당한 사용을 보여 줍니다.

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

struct SBitField {
    int         i : 2;
    unsigned    j : 5;
    int           : 4;
    int         k : 1;
    unsigned    m : 4;
};

void main() {
    struct SBitField s;

    s.i=-1;
    s.j=2;
    s.k=0;
```

```
    s.m=15;
    printf("%d,%d,%d,%d\n",s.i,s.j,s.k,s.m);
}//main()
```

- 위의 예에서 s는 다음과 같이 메모리 할당됩니다.




```
struct SBitfield {
   int          i : 2;
   unsigned     j : 5;
   int            : 4;
   int          k : 1;
   unsigned     m : 4;
};
```

비트 필드 구조체 SBitField의 메모리 구조: 전체의 크기가 계산되어 워드 정렬된 후, 하위 비트(LSB)에서 상위 비트(MSB)로 비트 필드가 채워집니다.

비트 필드와 일반 형을 썩어서 구조체를 만들 수 있는가요?

만들 수 있습니다. 공용체와 결합하면, 필드의 일부를 꺼내오기 위해서 비트 필드를 사용할 수도 있습니다. 실수의 표현법을 2진수로 표현하기 위해서, '**4. 데이터형**'에서 이러한 기교를 사용한 적이 있습니다. 그러므로, 아래의 문장은 문법적 에러가 아닙니다.

```
struct SWow {
    unsigned i : 4;
    int j;
};
```

진보된 주제: 구조체 필드, 자기 참조 구조체와 구조체 배열

• 구조체 자체가 형을 정의하는 문장이므로, 구조체 안에서 구조체를 정의하면서, 필드를 선언하는 것도 가능합니다.

```
struct SInner {
    int i;
    char j;
};

struct SOuter {
    struct SInner i;
    int j;
};
```

• 위의 예에서는 이미 정의된 구조체 SInner를 struct SOuter에서 필드로 포함하고 있습니다.

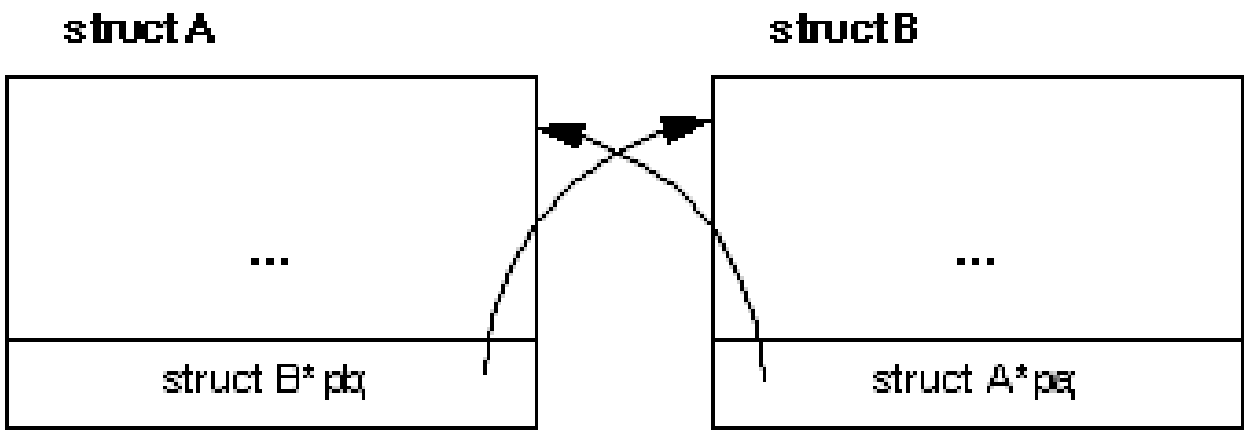
• struct SOuter의 변수는 (4+1)+4바이트의 메모리가 할당될 것입니다.

• **가시성(visibility)**을 설명하기 위해 struct SInner와 struct SOuter에서 같은 i,j를 사용한 것에 주목하세요.
• struct SInner와 struct SOuter는 다른 블록 범위를 가지므로, 같은 필드 이름을 사용하는 것이 가능합니다.
• 위의 예는 다음처럼 적어도 에러가 아닙니다.

```
struct SOuter {
    struct SInner {
        int i;
        char j;
    } i;
    int j;
};
```

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

struct SOuter {
    struct SInner {
        int i;
        char j;
    } i;
    int j;
};

void main() {
    struct SOuter i;

    i.i.i=1;
    i.i.j='A';
    i.j=2;
    printf("%d,%c,%d\n",i.i.i, i.i.j, i.j);
}//main()
```

불완전 선언(Incomplete Declaration)



상호참조 구조체: 구조체가 다른 구조체의 시작 주소를 필드로 가집니다. 그러므로, struct A를 정의할 때 struct B는 정의되어 있어야 합니다. 반대로 struct B를 정의할 때 struct A는 정의되어 있어야 합니다. 이러한 모순을 어떻게 해결할 수 있을까요?

- struct A는 struct B*를 필드로 가지고, struct B는 struct A*를 필드로 가집니다.
- 변수는 쓰기 전에 선언되어야하므로, struct A의 struct B* pb;를 필드로 선언하기 위해서는 struct B는 이미 정의되어 있어야 합니다.
- struct B입장에서는 struct A가 정의되어 있어야 합니다.
- 이러한 상호 참조의 모순을 해결하기 위해서, 구조체나 클래스는 **불완전 선언(incomplete declaration)** 및 **자기 참조(self referencing)**를 허락합니다.

```
struct A;//불완전 선언
struct B {
    int i;
    struct A* pa;
};
struct A {
    char c;
    struct B* pb;//struct B는 이미 정의되어 있다.
};
```

• 위의 코드 조각(program segment)에서 struct A; 를 불완전 선언이라고 합니다.
• 만약 이 문장이 없으면, struct B에서 struct A를 접근하는 아래의 문장은 컴파일 시간 에러가 발생할 것입니다.

struct A* pa;

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

struct A;
struct B {
    int i;
    struct A* pa;
};
struct A {
    char c;
    struct B* pb;
};

void main() {
    struct A a;
    struct B b;
    a.c='A';
    a.pb=&b;
    b.i=65;
```

```
    b.pa=&a;
    printf("%c,%c\n",a.c, b.pa->c);
}//main()
```

- 실행 결과는 다음과 같습니다.

```
A,A
```

• 구조체나 클래스는 **자기 참조(self referencing)**도 허락합니다. 자기 참조란 자신이 완전히 정의되기 전에 자신의 포인터를 필드로 가지는 기능을 의미합니다.

```
struct A {
    char c;
    struct A* pa;
};
```

• 위의 예에서 struct A는 자신의 완전한 정의가 끝나기 전에, 자신에 대한 포인터를 멤버로 선언하고 있습니다.

struct A*

• 이것은 에러가 아니며, 연결 리스트linked list 등을 구현하기 위해, 자주 사용하는 일반적인 방법입니다.

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

struct A {
    char c;
    struct A* pa;//struct A pa;처럼 사용하는 것은 불법이다.
};

void main() {
    struct A a1,a2;
    a1.c='A';
    a2.c='B';
    a1.pa=&a2;
    a2.pa=&a1;
    printf("%c,%c\n",a1.c,a2.pa->c);
}//main()
```

• 이제 구조체에 대해서 살펴볼 마지막 사항은 **구조체 배열(structure array)**에 관한 것입니다.

```
struct A {
    int age;
    char name[10];
};//struct A
```

• 위와 같은 구조체 형이 정의된 경우 struct A가 형이므로, struct A형의 크기 3인 배열을 선언하기 위해서는, 다음과 같이 할 수 있습니다.

struct A a[3];

• 여기서 a는 구조체 배열이 시작하는 곳의 시작 주소이며, a[0], a[1]과 a[2]가 구조체입니다.

```
#include <stdio.h>

struct A {
    int age;
    char name[10];
};//struct A

void main() {
    struct A a[3]={{48,"seojt"},{48,"jangwg"},{48,"parkmg"}};
    int i;

    for (i=0;i<3;++i) {
        printf("%s, %d\n",a[i].name,a[i].age);
    }//for
}//main()
```

공용체(union)

• **공용체(union)**는 필드들의 합집합(union) 만큼만 메모리에 할당되는 것을 제외하고, 구조체와 문법과 사용법이 모두 같습니다.

```
union A {
    char c;
    short i;
    long l;
};
```

• union A의 필드는 각각 1바이트, 2바이트 그리고 4바이트를 차지합니다. 만약 이것이 구조체라면, 메모리는 7바이트가 할당되겠지만, 공용체는 자신의 필드 중 가장 큰 필드만큼만 메모리에 할당됩니다. 즉, union A는 4바이트의 메모리가 할당됩니다.

• 왜 공용체를 사용하는 것일까요? 이유는 다음과 같습니다.

① 서로 배타적인 변수를 사용할 때
② 어떤 이유로 메모리에 존재하는 같은 변수의 다른 해석이 필요할 때

• 위의 예에서 보듯이, 문자형(char), 정수형(short) 그리고 긴 정수형(long integer) 변수가 필요한데, 이들 변수가 동시에 사용될 일은 없다고 합시다. 다음과 같이 변수 선언을 할 수 있습니다.

```
char c;
short i;
long l;
```

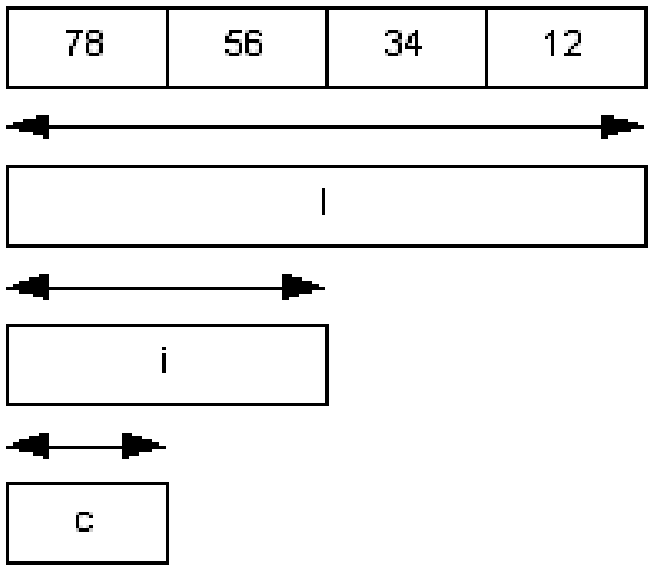
• 위의 선언이 틀린 것은 아니지만, 메모리의 효율 측면에서 아주 약간 비효율적인 것은 사실입니다. 즉, c, i와 l이 동시에 사용될 일은 없음에도 불구하고, 모두 메모리를 차지하고 있는 것이지요.

• 그러므로, 가장 긴 l(4바이트)만큼만 메모리를 확보하고, c와 i는 이 4바이트 영역을 공유하도록 하는 것입니다.
• 공용체를 정의한 다음, 다음과 같이 공용체 타입의 변수 a를 선언합니다.

```
union A a;
```

• 그러면, a를 위해서 4바이트의 메모리가 할당됩니다. 그리고, 문자형을 할당하여 쓸려면, a.c를 긴 정수형을 할당하여 쓸려면, a.l을 사용하면 되는 것입니다.

```
a.l=0x12345678L;
```



공용체의 메모리 할당: 각 필드가 메모리를 공유하여 할당됩니다. i80x 계열의 CPU는 역워드를 사용하므로, 이 점을 주의하세요.

• 위의 그림에서 보듯이 필드 l을 초기화했지만, i와 c의 값도 영향을 받는 것을 주목하세요.

```
#include <stdio.h>

union A {
    char c;
    int i;
    long l;
};

void main() {
    union A a;

    a.l=0x12345678L;
    printf("%x,%x\n",a.c,a.i);
}//main()
```

• 출력 결과는 다음과 같습니다.

```
78,5678
```

• 공용체를 사용하여 부동 소수의 내부 표현을 2진수로 출력하는 소스 코드를 다시 리스트합니다.

```
#include <iostream>

struct BITFIELD {//이것은 구조체 중에서도 비트 필드 구조체이다.
    unsigned m0:4;//아주 특별해 보이는 이 선언은 m0가 4비트를 차지하는
                  //것을 의미한다.
    unsigned m1:4;
    unsigned m2:4;
    unsigned m3:4;
    unsigned m4:4;
    unsigned m5:4;
    unsigned m6:4;
    unsigned m7:4;
};//struct BITFIELD

union FLOAT {//공용체는 필드(field)를 공유한다. f와 b는 같은 메모리를 공
             //유하며, f의 크기가 32비트, b의 크기 역시 32비트이므로 실
```

```
            //제 메모리는 64비트가 아닌 32비트가 할당된다.
    float f;
    BITFIELD b;
};//union FLOAT

void main() {
    FLOAT f;

    f.f=-14.24;
    //cout은 ostream 클래스의 전역 객체이다.
    cout << f.f << endl << hex;//endl과 hex 조작자(manupulator)는 출력
                               //스트림의 형식(format)을 조작한다. 이것
                               //은 '함수 포인터'에서 자세히 다룬다.
    cout << f.b.m0 << f.b.m1 << f.b.m2 << f.b.m3
        << f.b.m4 << f.b.m5 << f.b.m6 << f.b.m7 << endl;
    //output: c163d70a■
}
```

무명 공용체(anonymous union)

```
union A a;
a.l=0x12345678L;
```

• 위 코드를 다시 고려해 봅시다. 사용자의 목적은 l과 c와 i를 사용할 목적이었지만, 더미(dummy)로 a를 사용한 것이라고 가정해 봅시다.
• a를 사용하지 않고 c,i와 l을 바로 사용할 수는 없을까요?
• C++ 표준은 이러한 공용체를 지원하는데, 이것을 **무명 공용체(anonymous union)**라고합니다.
• 무명 공용체는 태그 이름을 가질 수 없으며, 변수 이름도 가질 수 없습니다. 아래의 예

```
#include <stdio.h>

void main() {
    union {//무명 공용체
        char c;
        int i;
        long l;
    };

    l=0x12345678L;//이제 l 및 i,c를 그냥 사용할 수 있다.
    printf("%x,%x\n",c,i);
}//main()
```

• 출력 결과는 다음과 같습니다.

78,5678

실습문제

1. 구조체가 값으로 전달될 때의 그림을 참고하여, 구조체 포인터가 전달될 때의 메모리 상태를 차례대로 그려보세요.

2. 자기 참조 구조체를 이용하여, 단일 연결 리스트(singly linked list)를 구현하세요.

3. 자기 참조 구조체를 이용하여, 이중 연결 리스트(doubly linked list)를 구현하세요.

@