15. 포인터 II

```
#include <stdio.h>

void f(int* ip) {
    (*ip)++;
}

void main() {
    int* ip;

    printf("Enter number:");
    scanf("%d",ip);
    f(ip);
    printf("%d\n",*ip);
}
```

• 수정된 소스는 아래와 같습니다.

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

void f(int* ip) {
    (*ip)++;
}//f

void main() {
    int* ip;

    ip=(int*)malloc(4);//(int*)의 형 변환은 반드시 필요하다.
    printf("Enter number:");
    scanf("%d",ip);
    f(ip);
    printf("%d\n",*ip);
    free(ip);
}
```

• 정수 3개를 할당하기 위한 아래의 문장이 항상 바르게 동작한다고 가정해서는 안 됩니다.

```
ip=(int*)malloc(12);
```

• 16비트 운영체제에서는 정수가 2바이트이지만, Windows의 Win32 환경에서는 정수는 4바이트이므로, 위의 문장은 16비트 운영체제에서는 정수를 6개 할당합니다.
• int가 처음 설계되었을 때, int는 운영체제의 워드크기에 맞추도록 정의되었으나, 64bit 운영체제에서는 코드의 하위 호환성을 위해서 int가 여전히 4바이트의 메모리를 차지합니다.
• 정수를 3개 할당하는 바른 문장은 다음과 같습니다.

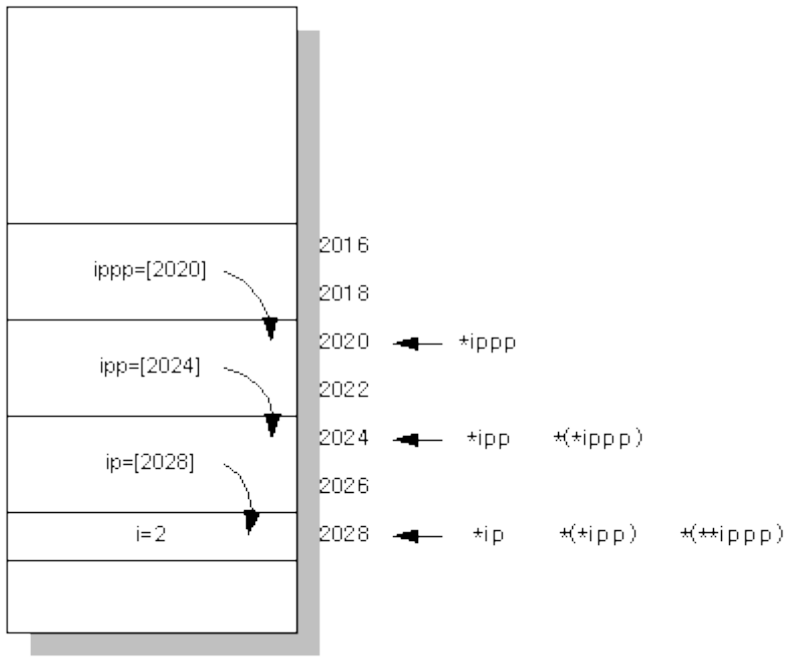
```
ip=(int*)malloc(sizeof(int)*3);
```

포인터의 포인터(pointer to pointer)

• 정수형 포인터의 포인터 ipp를 선언하기 위해, 다음과 같이 코드를 작성할 수 있습니다.

int ip;**

```
#include <stdio.h>

void main() {
    int i=2;
    int* ip;
    int** ipp;
    int*** ippp;

    ip=&i;
    ipp=&ip;
    ippp=&ipp;
    printf("%d,%d,%d,%d\n",i,*ip,**ipp,***ippp);
    printf("%p,%p,%p,%p\n",&i,ip,ipp,ippp);
    printf("%p,%p,%p,%p\n",&i,ip,*ipp,**ippp);
}
```

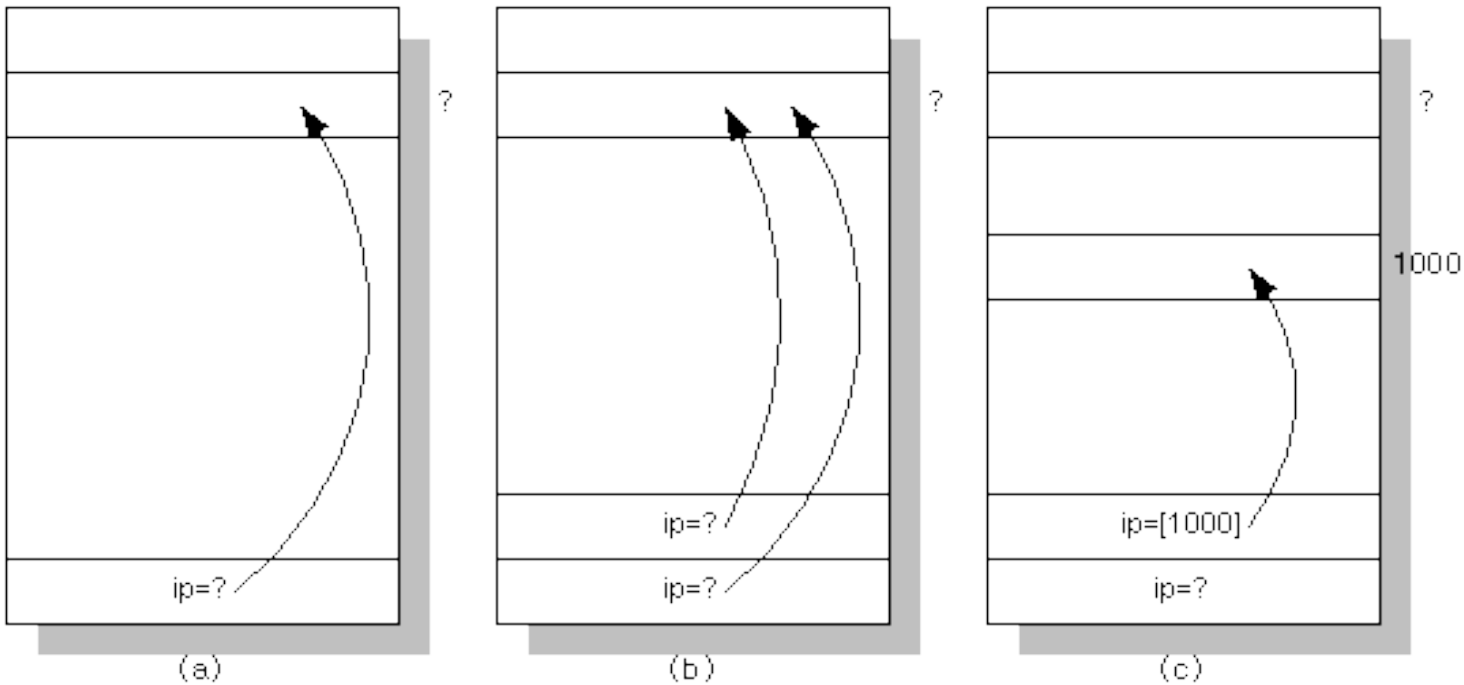


포인터의 포인터의 포인터: i=2이, &i=[2028]입니다. ip는 i의 주소를 가리키므로, ip=[2028]입니다. ip를 한 번 재참조(dereference)한 *ip는 [2028]이 가리키는 곳, 즉 2를 의미합니다. ipp=&ip이므로 [2024]입니다. *ipp는 ipp가 가리키는 곳 즉 [2028]을 의미합니다. **ipp는 ipp가 가리키는 곳이 가리키는 곳, 즉 정수 2를 의미합니다. ippp는 &ipp이므로 [2020]입니다. ***ippp는 ippp가 가리키는 곳이 가리키는 곳이 가리키는 곳, 즉 정수 2를 의미합니다.

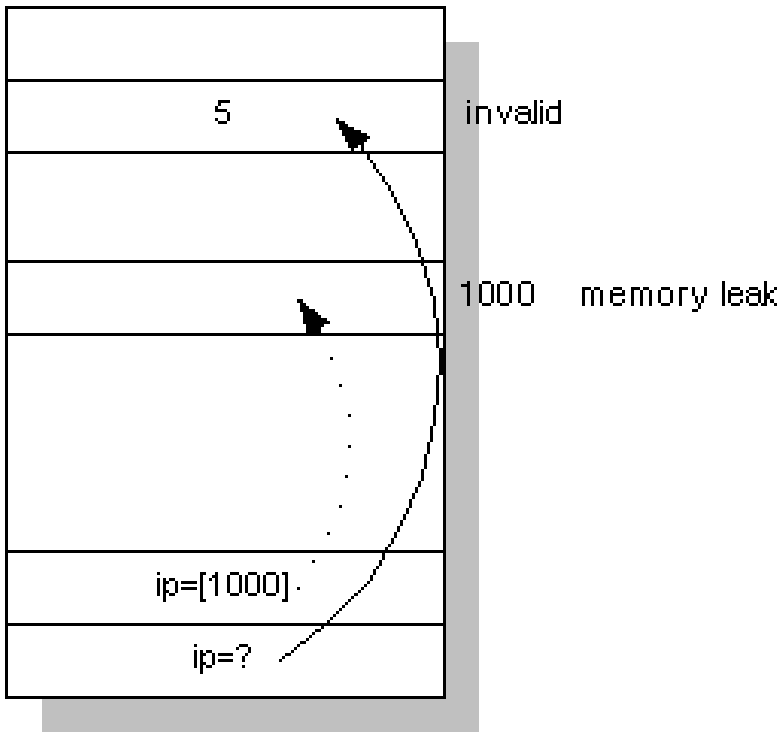
- 아래의 예에서 논리적 에러(semantic error)를 수정해 보세요.
- 이 프로그램은 IntAlloc()에서 정수 메모리를 할당합니다.
- 이것은 main()에서 사용되며, IntFree()에서 할당된 정수 메모리를 해제합니다.

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

void IntAlloc(int* ip) {
    ip=(int*)malloc(sizeof(int));
}//IntAlloc

void IntFree(int* ip) {
    free(ip);
}//IntFree

void main() {
    int* ip;
```

```
    IntAlloc(ip);
    *ip=5;
    printf("%d\n",*ip);
    IntFree(ip);
}
```

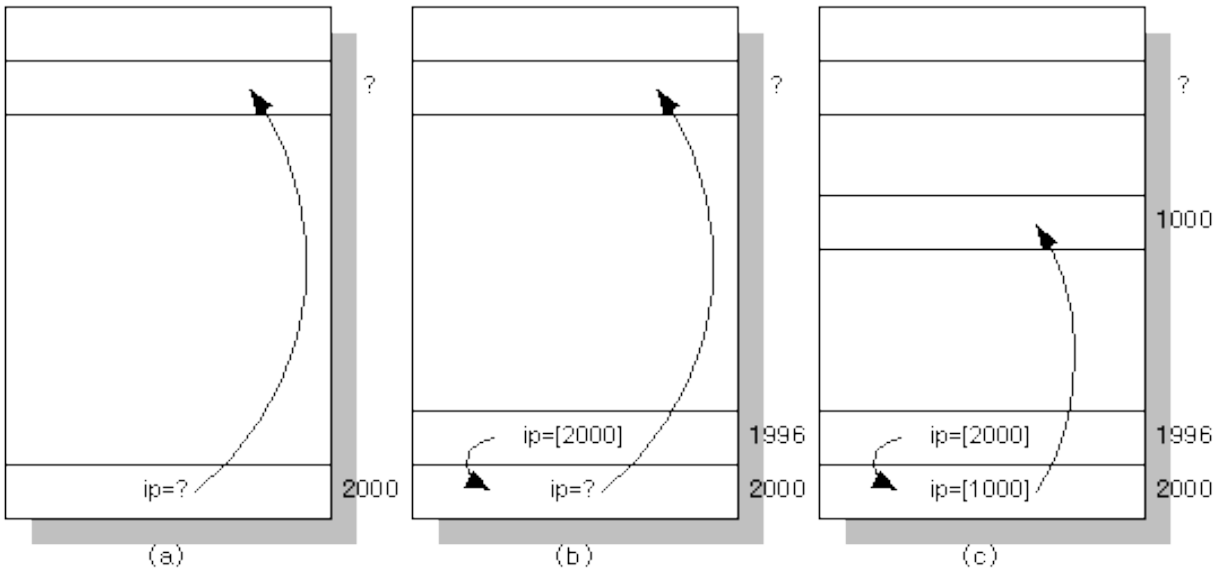


(a) main()에서 정수형 포인터 변수 ip의 공간을 스택에 할당합니다. 이 변수는 초기화되어 있지 않으므로 메모리의 ?곳을 가리키게 됩니다. (b) 함수 IntAlloc()을 호출합니다. ip의 값이 복사되어 전달되므로, IntAlloc()의 ip역시 처음에는 ?의 곳을 가리키게 됩니다. (c) malloc()의 호출에 의해 힙에 4바이트의 메모리 공간이 할당됩니다. 이 곳을 [1000]이라고 합시다. 이제 ip는 [1000]을 가리키게 됩니다.

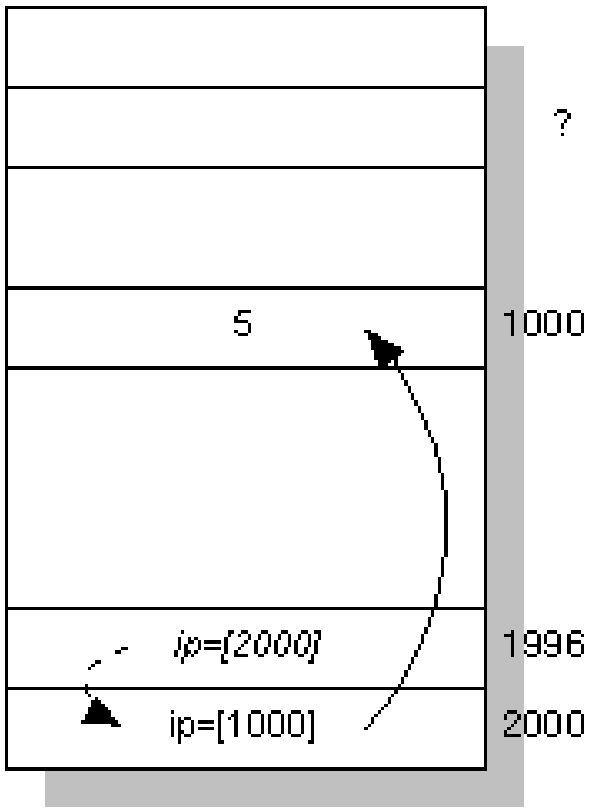


IntAlloc()을 빠져 나오면, [1000]을 가리키던 ip가 메모리에서 해제(팝pop)됩니다. 문제점은 다음과 같습니다. 할당된 [1000]을 가리키는 포인터가 없으므로, 메모리 릭(memory leak)이 발생합니다. main()의 ip는 여전히 할당되지 않은 메모리의 곳 ?를 가리키므로, *ip를 사용하는 것은 불법입니다.

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

void IntAlloc(int** ip) {
    *ip=(int*)malloc(sizeof(int));
}//IntAlloc

void IntFree(int* ip) {
    free(ip);
}//IntFree

void main() {
    int* ip;

    IntAlloc(&ip);
    *ip=5;
    printf("%d\n",*ip);
    IntFree(ip);
}
```



(a) main()의 ip가 스택에 할당되었을 때 ip는 쓰레기(garbage)로 초기화됩니다. (b) main()에서 함수 IntAlloc()을 호출할 때 ip의 주소를 전달합니다. 그러므로, IntAlloc()에는 [2000]이 전달되며 파라미터는 int**로 선언되어야 합니다. 이 변수는 1996번지에 할당된다고 가정합니다. IntAlloc()에서 ip는 main()의 ip, 즉 [2000]을 가리킵니다. (c) IntAlloc()에서 동적으로 메모리를 할당했을 때 1000번지가 할당되었다고 합시다. ip가 가리키는 곳(*ip)에 [1000]을 대입합니다. [2000]의 ip가 [1000]을 가리키게 됩니다.



main()에서 ip가 가리키는 [1000]의 곳(*ip)에 5를 대입합니다. 이것은 타당한 메모리 접근이므로 에러가 발생하지 않습니다.

참조(reference)

• C++의 새로운 참조(reference) 기능에 의해서도 해결이 가능합니다. 참조를 사용한 소스는 아래와 같습니다.

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

void IntAlloc(int*& ip) {
    ip=(int*)malloc(sizeof(int));
}

void main()
{
    int* ip;
    IntAlloc(ip);
    *ip=5;
    printf("%d\n",*ip);
    free(ip);
}
```

가용 공간 리스트(available list): 진보된 주제

- malloc() 혹은 new의 동작을 자세히 살펴보면 신기한 부분을 포함하고 있습니다.

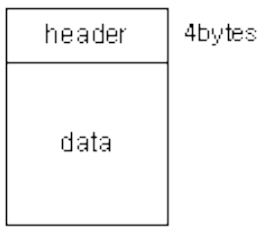
```
#include <stdio.h>
#include <stdlib.h>

void main()
{
    short* ip;
    short* ip2;

    ip=(int*)malloc(sizeof(short)*2);
    ip2=(int*)malloc(sizeof(short)*4);
    ip[0]=1;
    ip[1]=2;
    ip2[0]=3;
    ip2[1]=4;
```

```
    ip2[2]=5;
    ip2[3]=6;
    printf("%d,%d\n",ip[1],ip2[2]);//2와 5가 출력된다.
    free(ip);//ip만으로는 ip가 가리키는 곳이 4바이트인지 알 수 없다.
    free(ip2);
}
```

• **free()함수가 ip와 ip2를 해제할 때 ip는 4바이트의 곳을 가리키며, ip2는 8바이트의 곳을 가리키는지 어떻게 알 수 있을까요?**

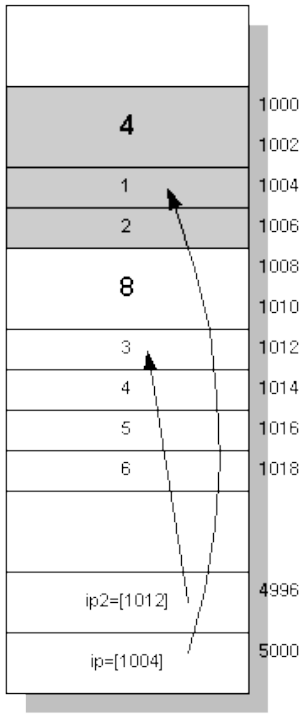
• 이것은 C컴파일러가 **사용 가능한 메모리 블록(available memory block)**에서 메모리를 할당하고 유지하는 방식 때문에 가능한 것입니다.

• 사용자가 10바이트의 메모리 블록을 할당하기 위해 malloc(10)을 호출하면, CRT 힙 관리자(heap manager)는 크기 정보 4바이트를 포함한 14바이트의 메모리 블록을 할당한 후(이 블록의 주소를 1000이라고 합시다) 4바이트를 건너뛴(skip) 곳의 주소를 리턴합니다(1004가 리턴됩니다).



memory block

malloc()이 할당하는 메모리 블록: 크기 정보 4바이트를 포함한 크기를 메모리에서 할당합니다. 사용자는 이 포인터를 이용해서 연산을 할 수 있어야 하므로, 리턴되는 주소는 헤더의 주소가 아니라, 헤더+4의 주소가 리턴됩니다. free()에 주소를 넘겨주면 free()함수는 '파라미터 주소-4'를 계산하여 실제 할당된 메모리의 크기를 구합니다. 그런 다음 이것을 해제합니다. 메모리 블록은 컴파일러에 의존적입니다. 다른 C컴파일러는 다른 구조의 메모리 블록을 사용할 수도 있습니다.



ip가 [5000]에 ip2가 [4996]에 할당되었다고 합시다. 첫 번째 malloc()호출은 4바이트를 할당하려 시도합니다. 사용 가능한 힙의 주소가 [1000]이었다면, 헤더 4바이트가 블록의 크기 4로 초기화되면서, 모두 8바이트(헤더 4바이트+데이터 4바이트)가 할당됩니다. 그리고 malloc()은 [1000]이 아닌 [1004]를 리턴합니다. 두 번째 malloc() 호출도 마찬가지입니다.

- 볼런드 C의 함수 coreleft()를 사용해 봅시다.

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

void main()
{
    short* ip;
    short* ip2;
    unsigned long size0,size1,size2;

    size0=coreleft();
    ip=(short*)malloc(sizeof(short)*30);
    size1=coreleft();
    ip2=(short*)malloc(sizeof(short)*8);
    size2=coreleft();
    printf("%d\n",sizeof(short));
    printf("%ld,%ld,%ld\n",size0,size0-size1,size1-size2);
```

```
    free(ip);
    free(ip2);
}
```

• 결과는 다음과 같습니다.

```
2
561280,64,32
```

• 이 프로그램을 실행한 컴퓨터의 메모리가 64메가 바이트임에도 불구하고, 사용 가능한 메모리가 561,280으로 찍힌 이유는 coreleft()가 도스 전용 함수이기 때문입니다.
• short 30개(60바이트)를 할당하기 전의 메모리에서 60바이트를 할당한 후의 메모리 차이는 64입니다. 여분의 4바이트가 더 할당된 것을 알 수 있습니다.

new와 delete

- new/delete는 다음과 같은 점이 malloc() 부류의 함수보다 좋습니다.

  - ■ 명시적인 형 변환이 필요없습니다.
  - ■ malloc()은 함수지만, new는 연산자(operator)입니다.
  - ■ [연산자이기 때문에 유연함을 제공합니다.]

- new의 문법은 다음과 같습니다.

  [::] new [new-args] type-name [initializer]
  [::] new [new-args] (type-name) [initializer]

• delete의 문법은 다음과 같습니다.

[::] delete pointer
[::] delete [ ] pointer

• 위의 예에서 ip가 가리키는 할당된 메모리를 해제하기 위해서는 다음과 같이 사용합니다.

```
delete ip;
```

- 메모리 할당 후 값을 초기화하는 new문장을 사용할 수 있습니다.

```
ip=new  int;
*ip=3;
```

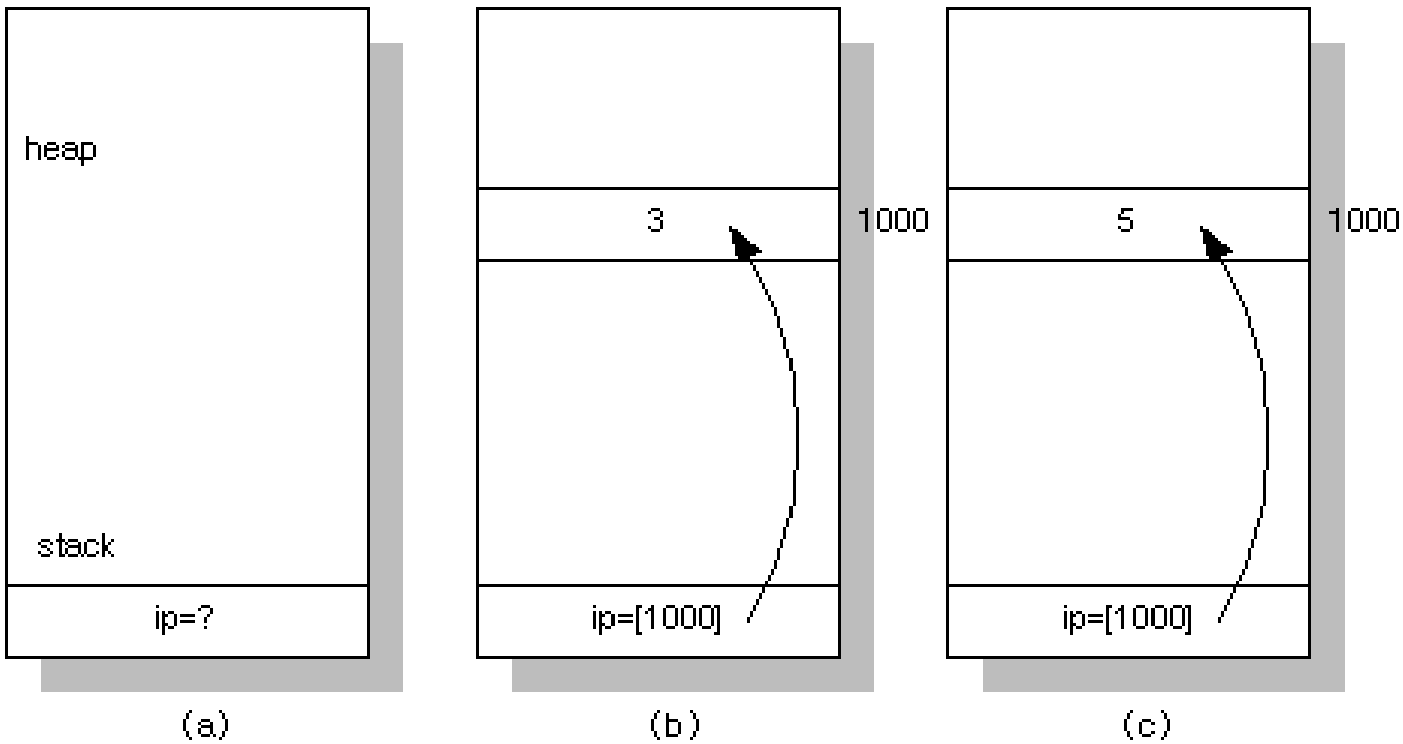
- 위의 문장 대신에 다음과 같이 사용할 수 있습니다.

```
ip=new  int(3);
```

```
#include <stdio.h>

void main() {
    int* ip;

    ip=new int(3);
    printf("%d\n",*ip);
    *ip=5;
    printf("%d\n",*ip);
    delete ip;
}
```

• 결과는 다음과 같습니다.

```
3
5
```



(a) int* ip; (b) ip=new int(3); (c) *ip=5;

• malloc()을 이용해서 10개의 정수를 위한 메모리를 할당하기 위해 다음과 같이 사용할 수 있습니다.

```
ip=(int*)malloc(sizeof(int)*10);
```

• 이것을 new로는 다음과 같이 합니다.

```
ip=new int[10];
```

• 어떤 형(type)의 변수 n개만큼의 메모리를 확보하기 위해서는 다음과 같이 사용할 수 있습니다.

```
new type[n];
```

• 이렇게 **기본형의 배열로 초기화**되었을 경우, 메모리를 해제하기 위해서는 다음과 같이 delete를 사용해야 합니다.

```
delete[] ip;
```

• delete ip; 처럼 사용하는 것은 오류라는 사실을 주의하세요.

```
#include <stdio.h>

void main() {
    int* ip;
    int* ip2;
    int* ip3[2];

    ip=new int;
    ip2=new int[5];
    ip3[0]=new int[5];
    ip3[1]=new int;
    *ip=5;
    printf("%d\n",*ip);
    delete ip;
    delete[] ip2;
    delete[] ip3[0];
    delete ip3[1];
}
```

2차원 배열의 할당

- new를 사용하여 2차원 이상의 배열을 할당하는 경우는 어떻게 할까요?

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

void main() {
    int dim=4;

    char (*pchar)[10];//배열이 선언된 것이 아니라, pchar이 선언된 것이
                      //다.

    pchar=new char[dim][10];
    strcpy(pchar[0],"gold");
    strcpy(pchar[1],"silver");
    strcpy(pchar[2],"copper");
    strcpy(pchar[3],"neck");
```

```
    printf("%s\n",pchar[0]);
    delete[] pchar;
}
```

• 출력 결과는 다음과 같습니다.

```
gold
```

실습문제

1. 포인터의 포인터의 포인터를 사용하는 적절한 예를 설명하세요.

2. delete[] i와 delete i 는 어떻게 다른지 설명하세요.

3. 크기가 알려지지 않은 자료를 관리하려고 합니다. 이 자료에는 새로운 자료를 입력하거나 삭제가 가능해야 하며, 관리되는 자료를 모두 출력할 수 있어야 합니다. 이러한 자료를 관리하기 위한 구조를 설계하고 구현하세요?(힌트: linked list)

@