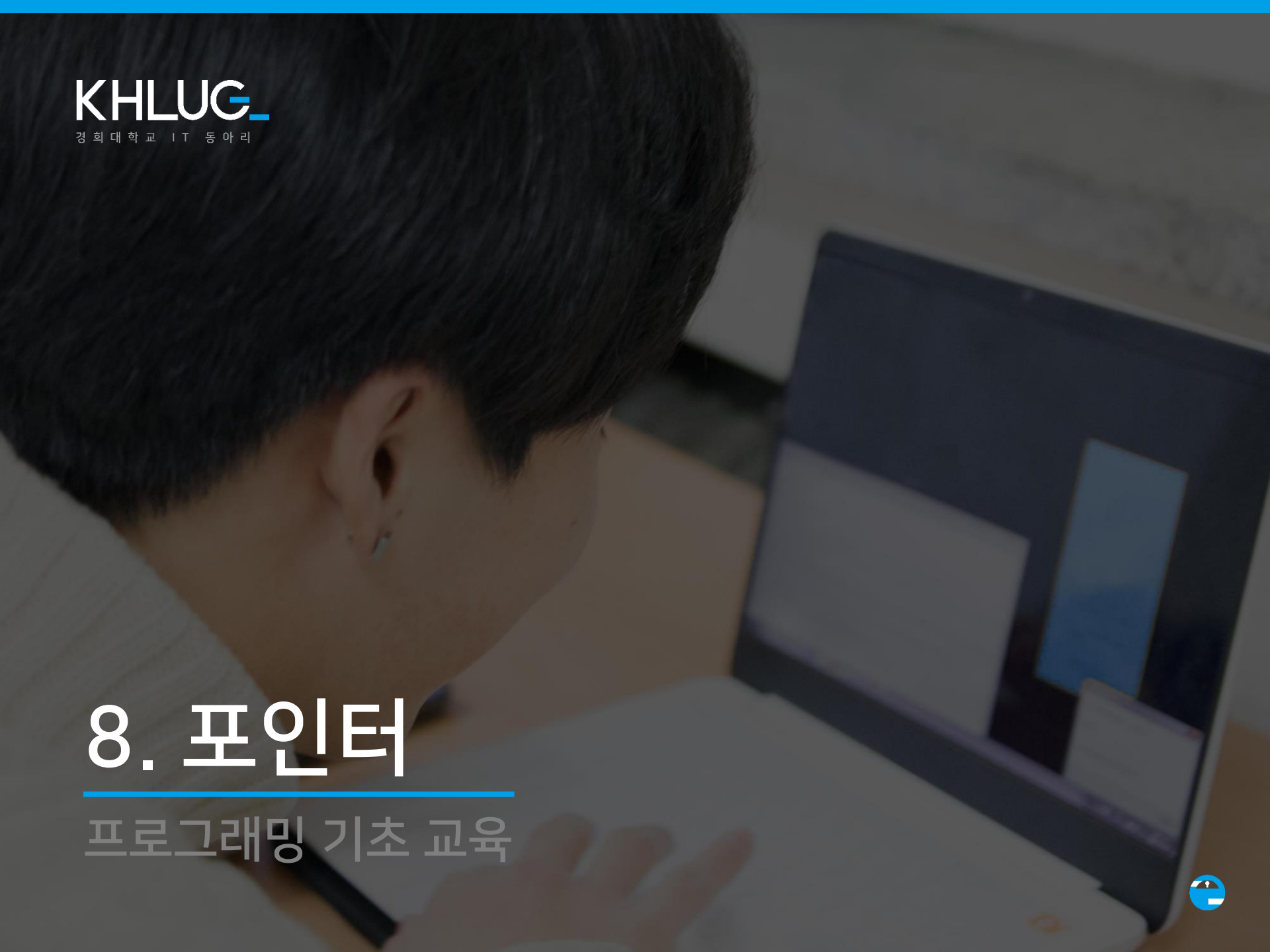
KHLUG

경희대학교 IT 동아리

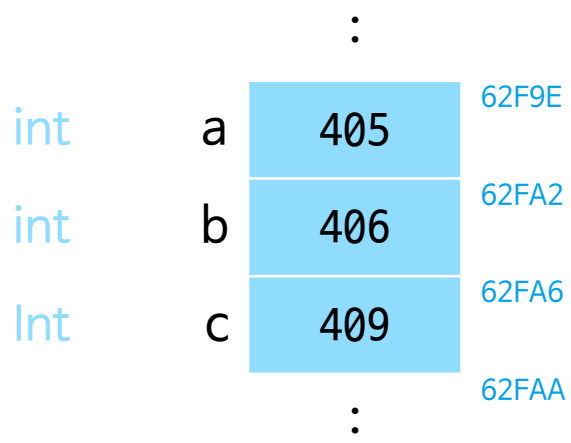


8. 포인터

프로그래밍 기초 교육

메모리

- 변수를 만들면 메모리의 어딘가에 저장되는데, 각 변수에는 주소(address)가 있다.



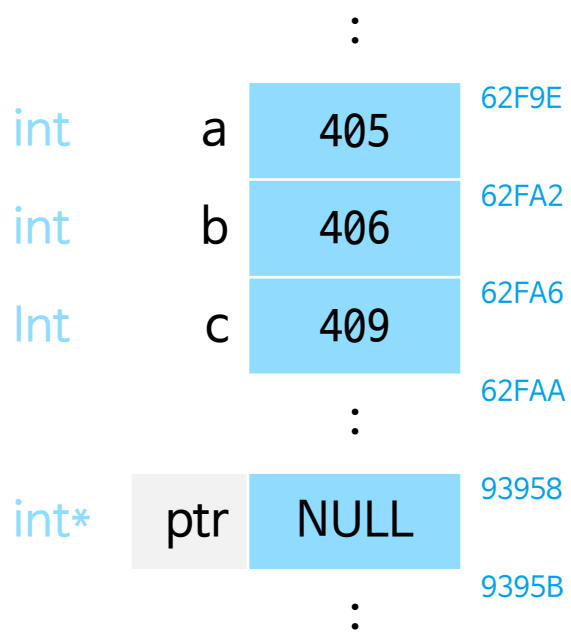

```
int a = 405;
int b = 406;
int c = 409;
```

1바이트마다 1의 주소를 갖기 때문에
4바이트인 int형 변수의 주소와
바로 다음 번지의 주소는 4 차이 난다.

연속적으로 선언했더라도
배열이 아닌 이상 연속될
거라는 보장은 없다.

포인터

- 그리고 이 주소를 담는 변수를 '포인터(pointer)'라고 한다.




```
int a = 405;
int b = 406;
int c = 409;

int* ptr;
```

int*형 포인터는 int형 변수의 주소를 담는다.

다른 자료형, 구조체 등의 포인터도 만들 수 있다.

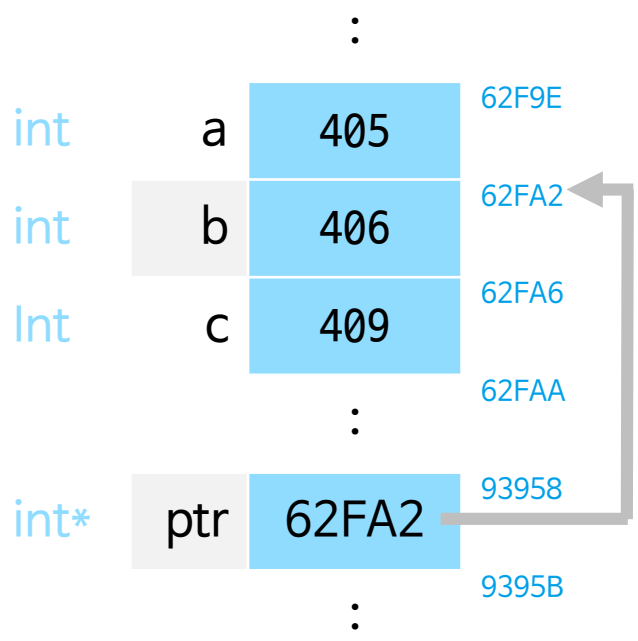
포인터는 주소를 저장하고, 주소는 32비트 운영체제에서는 32비트=4바이트의 크기를 가진다.

여기선 편의상 5자리만 표시했다.

초기화를 하지 않았을 땐 NULL(빈) 값이 들어간다.

변수의 주소

- 변수의 주소는 &을 붙여 가져올 수 있다.



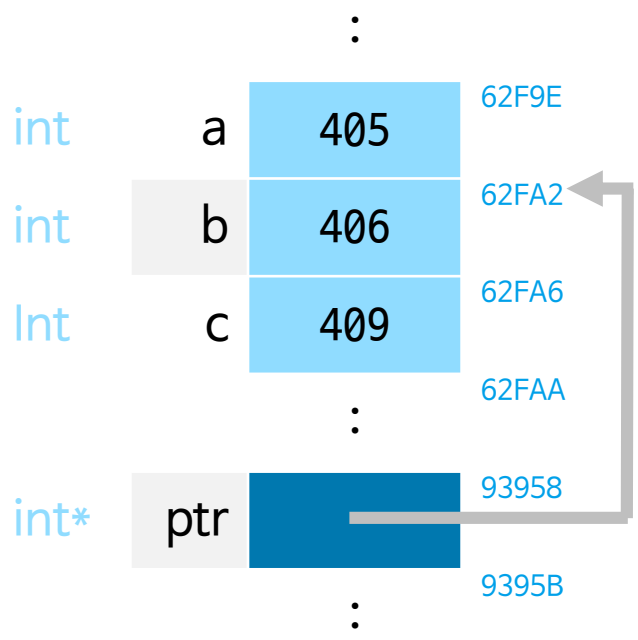

```
int a = 405;
int b = 406;
int c = 409;

int* ptr;
ptr = &b;
```

int*형 포인터는 int형 변수만 가리킬(pointing) 수 있다.

&는 앰퍼샌드(ampersand)가 공식 명칭이다.

포인터의 값

- 포인터 변수에 *를 붙여
  포인터가 가리키는 값을 사용한다.




```
주소: 62FA2
값  : 406
```

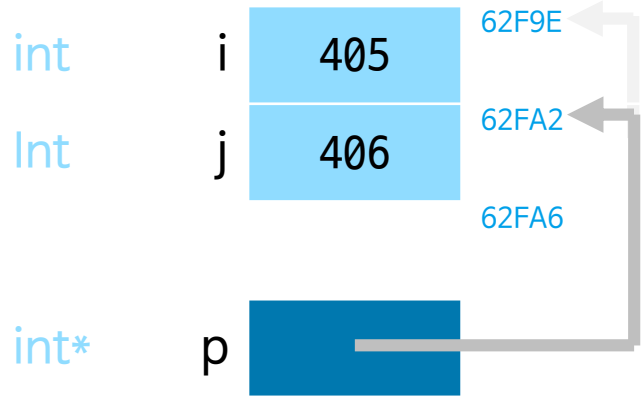
```
int a = 405;
int b = 406;
int c = 409;

int* ptr;
ptr = &b;

cout << "주소: " << ptr << endl;
cout << "값  : " << *ptr << endl;
```

곱하기를 할 때의 *와
포인터를 선언할 때의 *와
포인터를 사용할 때의 *를
헷갈리지 말자.

*는 애스터리스크(asterisk)가 공식 명칭이다.

포인터 변수

- 포인터가 가리키는 변수를 바꿀 수 있다.

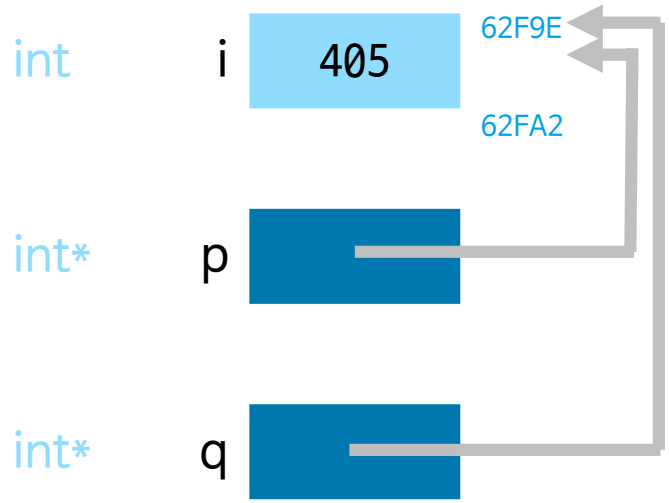



```
int i = 405;
int j = 406;
int* p;

p = &i;

cout << " p: " << p << endl;
cout << "*p: " << *p << endl;

p = &j;

cout << " p: " << p << endl;
cout << "*p: " << *p << endl;
```

```
 p: 62F9E
*p: 405
 q: 62FA2
*q: 406
```

포인터 변수

- 여러 개의 포인터가 한 변수를 가리킬 수 있다.

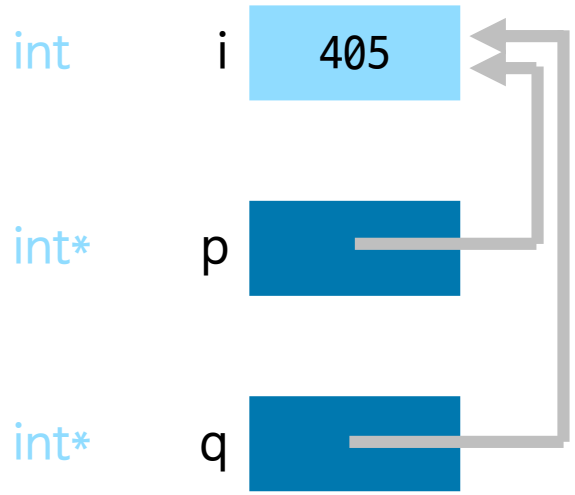



```
int i = 405;
int* p = &i;
int* q = &i;

cout << " p: " << p << endl;
cout << "*p: " << *p << endl;

cout << " q: " << p << endl;
cout << "*q: " << *p << endl;
```

```
 p: 62F9E
*p: 405
 q: 62F9E
*q: 405
```

포인터로 접근하기

- * 연산자로 접근해서 변수를 사용할 수 있다.




```
int i = 405;
int* p = &i;
int* q = &i;

cout << "[i = 405]       i: " << i
    << endl;

i = i + 1;              // i = i + 1
cout << "[i = i + 1]     i: " << i
    << endl;

*p = 409;               // i = 409
cout << "[*p]            i: " << i
    << endl;


*&i = *p + *q;          // i = i + i
cout << "[*&i = *p + *q] i: " << i
    << endl;

i = *p * *q;            // i = i * i
cout << "[i = *p * *q]   i: " << i
    << endl;
```

```
[i = 405]       i: 405
[i = i + 1]     i: 406
[*p]            i: 409
[*&i = *p + *q] i: 818
[i = *p * *q]   i: 669124
```

i의 주소(&)에 있는 값(*)이므로 그냥 i다.

포인터 연산자 *와 &는 반대되는 개념이다.

포인터 연산자와 곱셈 연산자를 헷갈리지 말자.

매개변수로 포인터 넘기기

- 포인터로 매개변수를 전달하면 포인터 연산자를 통해 변수를 사용할 때는 주소를 통해 그 변수를 직접 찾아간다.
- 따라서 함수 안에서 값이 바뀌면 함수가 끝나도 바뀐 값이 유지된다. (참조 값으로 매개변수를 전달할 때와 유사하다.)

참조 값으로 전달

```
void exchange_ref(int &x, int &y) {
    int hand = x;
    x = y;
    y = hand;
    return;
}
```

포인터로 전달

```
void exchange_ptr(int* px, int* py) {
    int hand = *px;
    *px = *py;
    *py = hand;
    return;
}
```

호출 예시

```
exchang_ref(x, y);
exchange_ptr(&x, &y);
```

둘 다 함수 밖에서도 바뀐 값이 적용되는지 확인해보다.

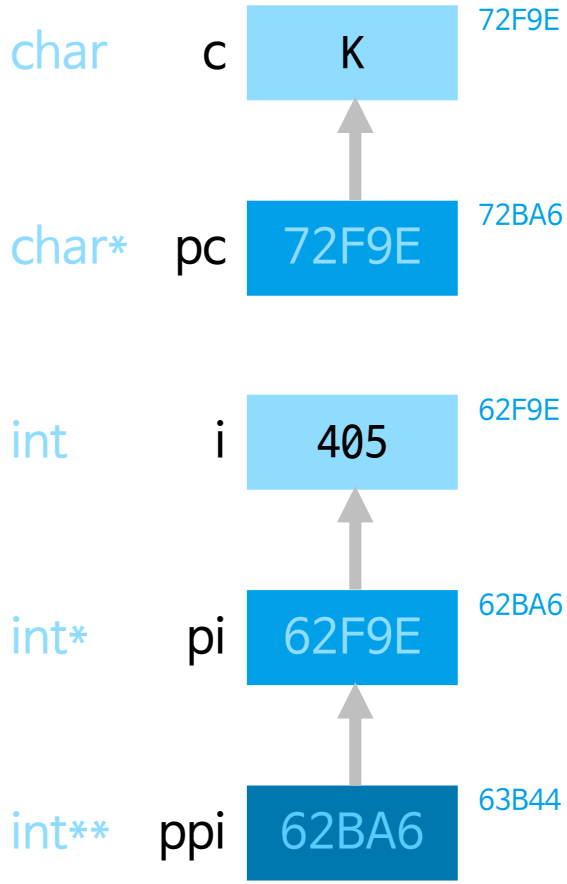
포인터를 반환하기

- 포인터도 자료형의 일종이므로 함수의 매개변수로 들어갈 수도, 함수의 반환형이 될 수도 있다.

```
double* larger(double*, double*);

int main() {
    double a;
    double b;
    double* p;

    cin >> a >> b;
    p = larger(&a, &b);
    cout << "더 큰: " << *p << endl;
}

double* larger
    (double* px, double* py) {
    return (*px > *py ? px : py);
}
```

다중 포인터

- 포인터는 다른 포인터를 가리킬 수 있다.




```
char c = 'K';
char* pc;

int i = 58;
int* px;
int** ppa;

pc = &c;
pi = &a;
ppi = &pi;

pc = &i;
ppi = &i;
```

포인터가 가리키고자 하는 자료형이 변수와 일치한다.

int**형은 int*형을 가리킨다.

char*형은 int형을 가리키지 못 한다.

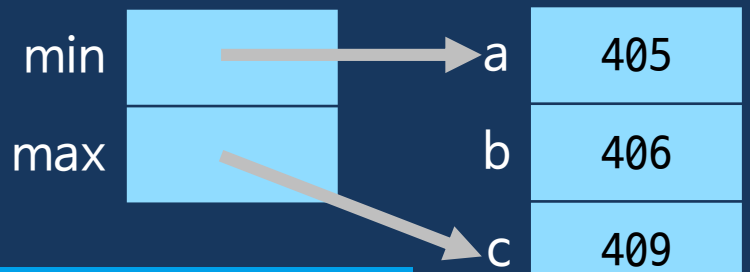
int**형은 int형을 가리키지 못 한다. (int*형을 가리켜야 한다.

이중 포인터(ppi)가 최종적으로 가리키는 곳(i)에 접근하려면 **ppi처럼 포인터 연산자를 두 번 써주면 된다.

포인터 사용해보기

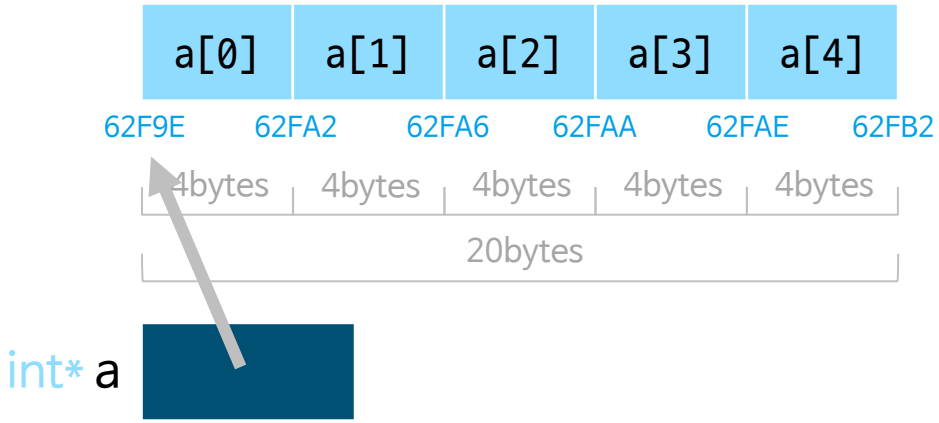
5개의 문자를 입력 받고, 입력 받은 문자의 주소와 값을 출력하는 프로그램을 작성하시오. 입력 받거나 출력할 때는 포인터 변수만 사용해야 하며, 입력 받을 때에는 공백 문자를 무시하지 않도록 하시오.

숫자 3개를 입력 받아 가장 큰 수와 가장 작은 수를 동시에 구하는 함수를 작성하시오. 이때 가장 큰 수와 가장 작은 수는 main()에서 출력하되, 함수에서 값을 전달받을 때 기존에 변수를 가리키는 포인터로 받도록 않도록 하시오. (아래의 구조를 참고)



값을 새로 만들지 말고 기존의 값을 가져와서 사용할 수 있도록 구성한다.

배열과 포인터



- 배열의 이름은 배열의 첫 번째 원소에 대한 포인터이다.

```
a[0] : 1
*a   : 1
&a[0]: 62F9E
a    : 62F9E
```

```
int a[5] = {1, 2, 3, 4, 5};
```

메모리의 저장 단위는 1바이트이다.
따라서 4바이트인 정수가 연속되어 저장되는
이 배열에서 각 원소의 주소는 4씩 차이 난다.

```
cout << "a[0] : " << a[0] << endl;
cout << "*a   : " << *a << endl;
```

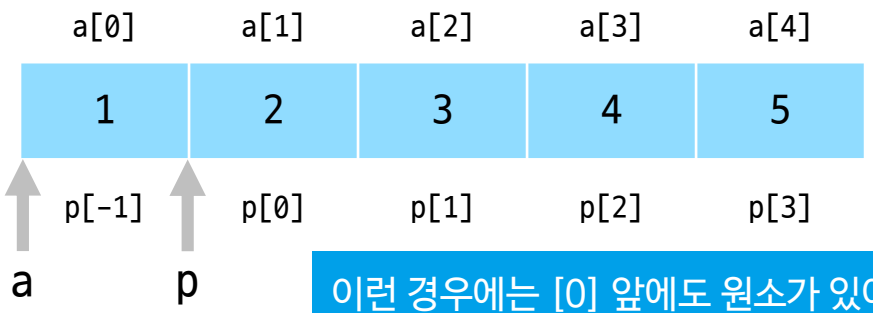
둘은 같은 값을 가진다.

```
cout << "&a[0]: " << &a[0] << endl;
cout << "a    : " << a << endl;
```

둘은 같은 값(주소)를 가진다.

포인터의 배열 연산

- 포인터는 배열의 원소도 가리킬 수 있다.
- 그리고 배열 이름이 포인터이듯이 배열 포인터 변수는 배열처럼 사용할 수 있다.



이런 경우에는 [0] 앞에도 원소가 있어 음수 인덱스가 가능하다.

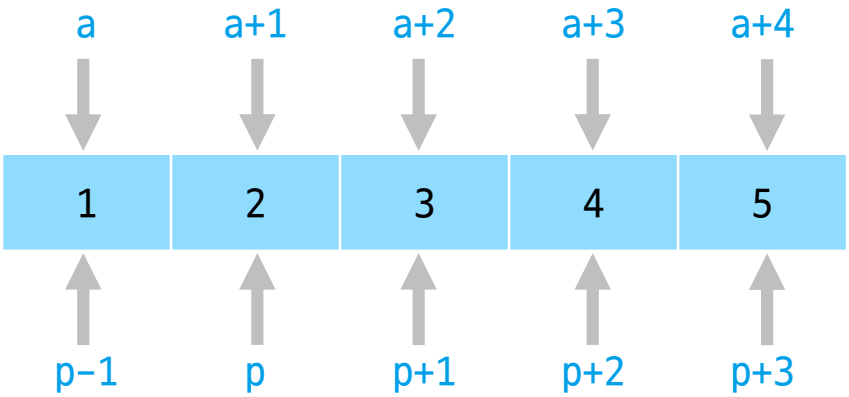
```
a[ 0]: 1
p[-1]: 1
a[ 1]: 2
p[ 0]: 2
```

```
int a[5] = {1, 2, 3, 4, 5};
int* p;


p = &a[1];

cout << "a[ 0]: " << a[0] << endl;
cout << "p[-1]: " << p[-1] << endl;

cout << "a[ 1]: " << a[1] << endl;
cout << "p[ 0]: " << p[0] << endl;
```

포인터 연산

- 배열 포인터에 +- 연산을 해 배열 원소에 접근할 수도 있다.
- *(a+n)은 a[n]과 같다.

++, -- 연산도 가능하다.




```
*(a+n): 1 2 3 4 5
*(p+n): 1 2 3 4 5
```

```
int a[5] = {1, 2, 3, 4, 5};
int* p;

p = &a[1];

cout << "*(a+n): ";
for (int i = 0; i <= 4; i++)
    cout << *(a + i) << " ";
cout << endl;

cout << "*(p+n): ";
for (int i = -1; i <= 3; i++)
    cout << *(p + i) << " ";
cout << endl;
```

이차원 배열과 포인터

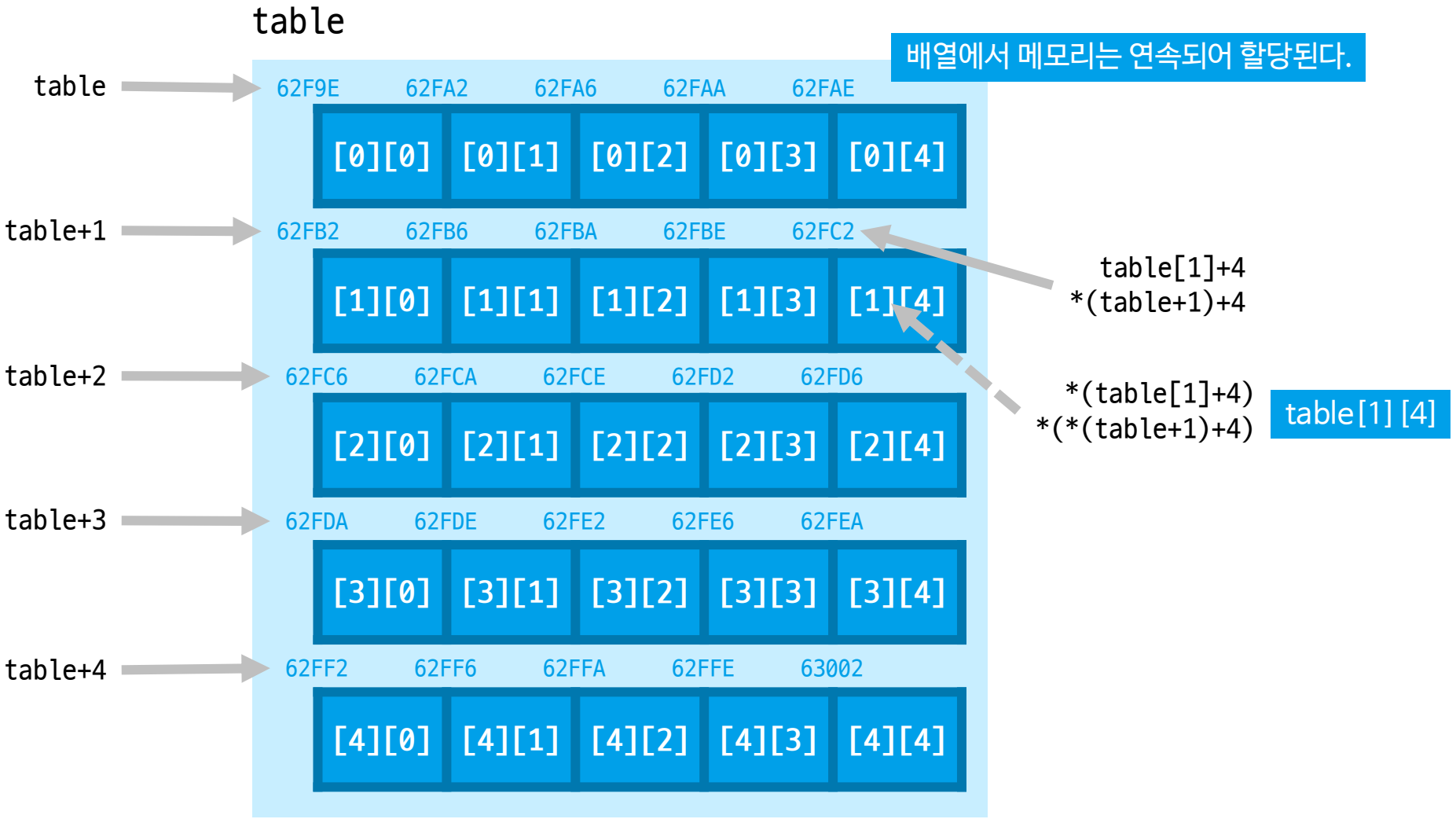
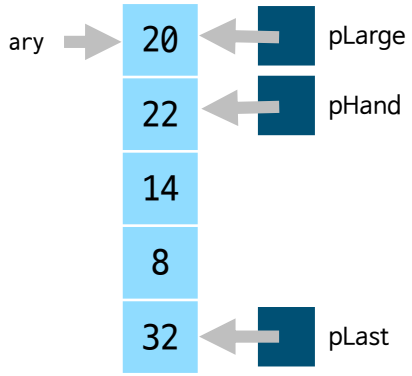


table
table
62F9E 62FA2 62FA6 62FAA 62FAE
[0][0] [0][1] [0][2] [0][3] [0][4]
table+1
62FB2 62FB6 62FBA 62FBE 62FC2
[1][0] [1][1] [1][2] [1][3] [1][4]
table+2
62FC6 62FCA 62FCE 62FD2 62FD6
[2][0] [2][1] [2][2] [2][3] [2][4]
table+3
62FDA 62FDE 62FE2 62FE6 62FEA
[3][0] [3][1] [3][2] [3][3] [3][4]
table+4
62FF2 62FF6 62FFA 62FFE 63002
[4][0] [4][1] [4][2] [4][3] [4][4]
배열에서 메모리는 연속되어 할당된다.
table[1]+4
*(table+1)+4
*(table[1]+4)
((table+1)+4)
table[1][4]

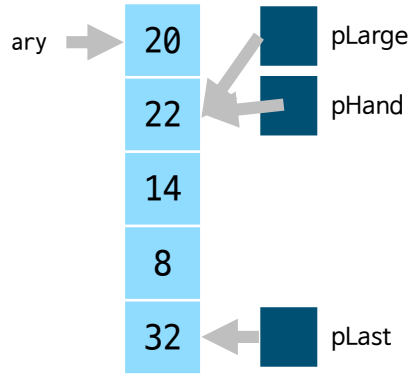
배열 포인터 사용해보기

포인터 연산을 사용하여 배열 안에서 가장 큰 값을 찾는 함수를 작성하시오. (아래: 예시)

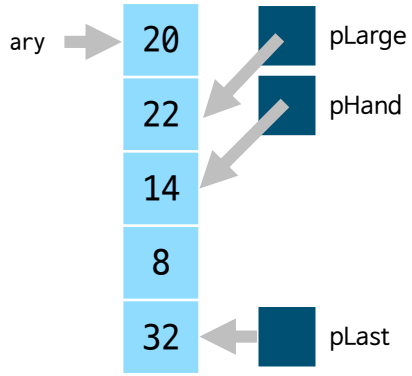
초기화 직후



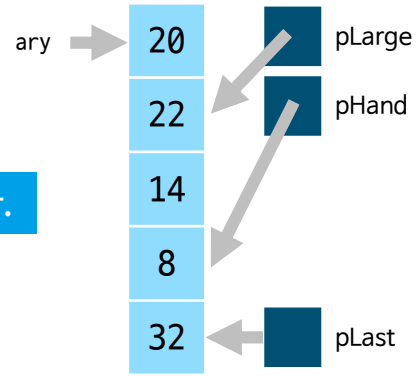
1차 반복



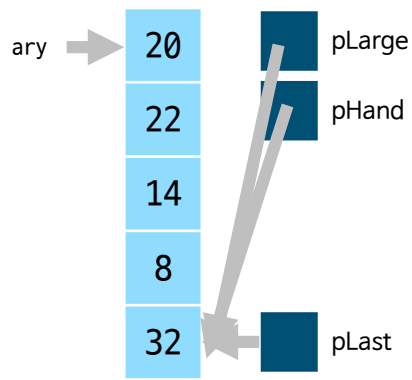
2차 반복



3차 반복



4차 반복



정렬을 배울 때 사용했던 방법과 유사하다.

배열 포인터 사용해보기

아래는 6차시 정렬 시간에 배웠던 알고리즘이다.

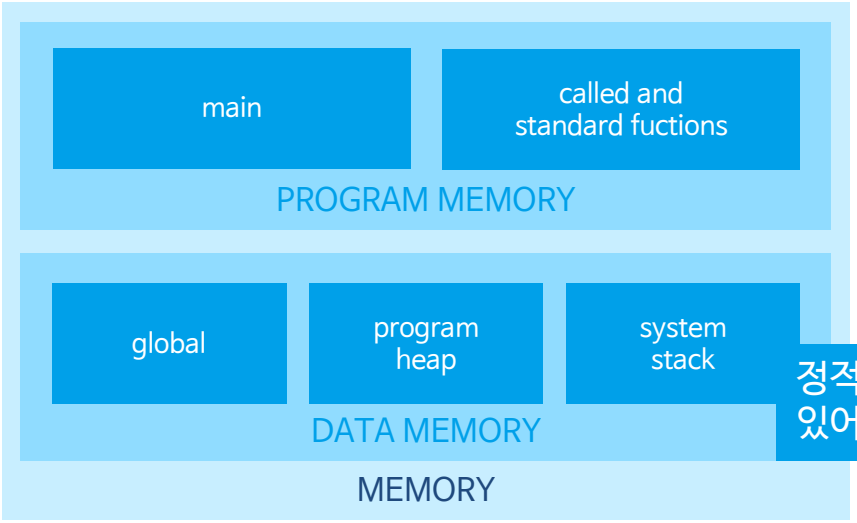
- 거품 정렬
- 선택 정렬
- 삽입 정렬
- 선형 탐색
- 이진 탐색

코드는 자료에 주어졌는데,
이 코드를 모두 배열이 아닌 포인터를 받는 함수로 수정하여라.

메모리 할당

- 여태까지 선언과 초기화를 통해 변수를 만들어 왔다.
- 이 데이터들은 프로그램을 로딩할 때 메모리가 정해져 할당되어 프로그램이 끝날 때까지 남아있다.

```
int i;
double d;
char c;
```



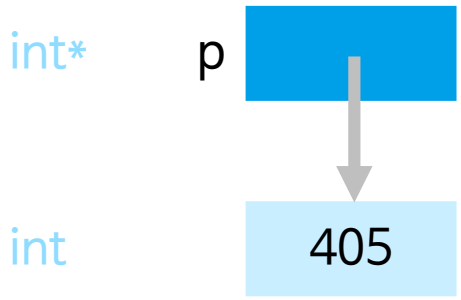
프로그램을 실행하면 이런 느낌으로 메모리가 구분된다.

정적 할당된 데이터는 스택에 할당되어 있어 최대 크기가 비교적제한된다.

동적 할당

- 반면 동적 할당은 사용할 메모리 공간을 정해놓지 않고 프로그램이 실행되어가면서 메모리를 할당 받게 할 수 있다.

정적 할당된 데이터는 힙에 할당된다.




```
int* p = new int;
```

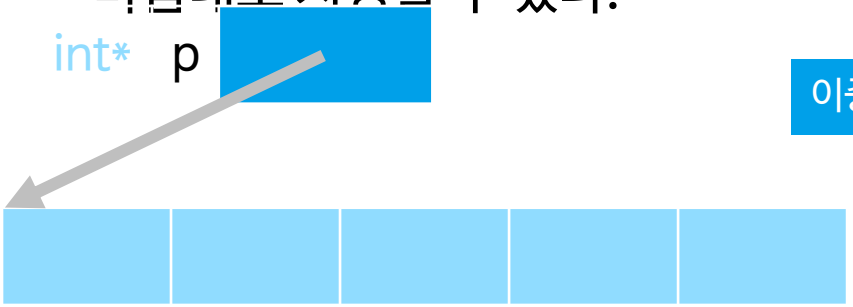
이 코드가 실행되는 순간 메모리가 할당된다.
이전까지 이 정수를 위한 공간은 없었다.

동적 할당은 포인터에만 할 수 있다.

```
*p = 405;
```

배열 동적 할당

- 배열 또한 동적 할당할 수 있는데, 처음부터 메모리 공간을 지정해놓을
  필요가 없기 때문에 배열의 크기를 마음대로 지정할 수 있다.




```
int* p = new int[5];
for(int i = 0; i < 5; i++)
    p[i] = i;

for(int i = 0; i < 5; i++)
    cout << *(p+i) << endl;
```

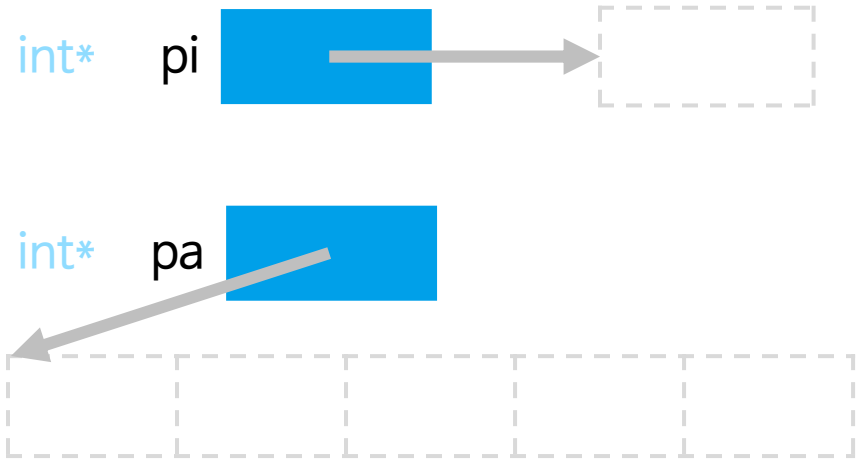
상수 대신 변수를 넣을 수 있다.
이제 사용자가 입력한 값만큼
배열의 크기를 정할 수 있다!

이중 포인터일 필요는 없다.

그냥 배열처럼 쓸 수 있다.

할당 해제

- 할당 받은 메모리는 프로그램이 끝날 때까지 남아있기 때문에 사용이 끝나면 해제를 해서 메모리 공간을 절약해야 한다.

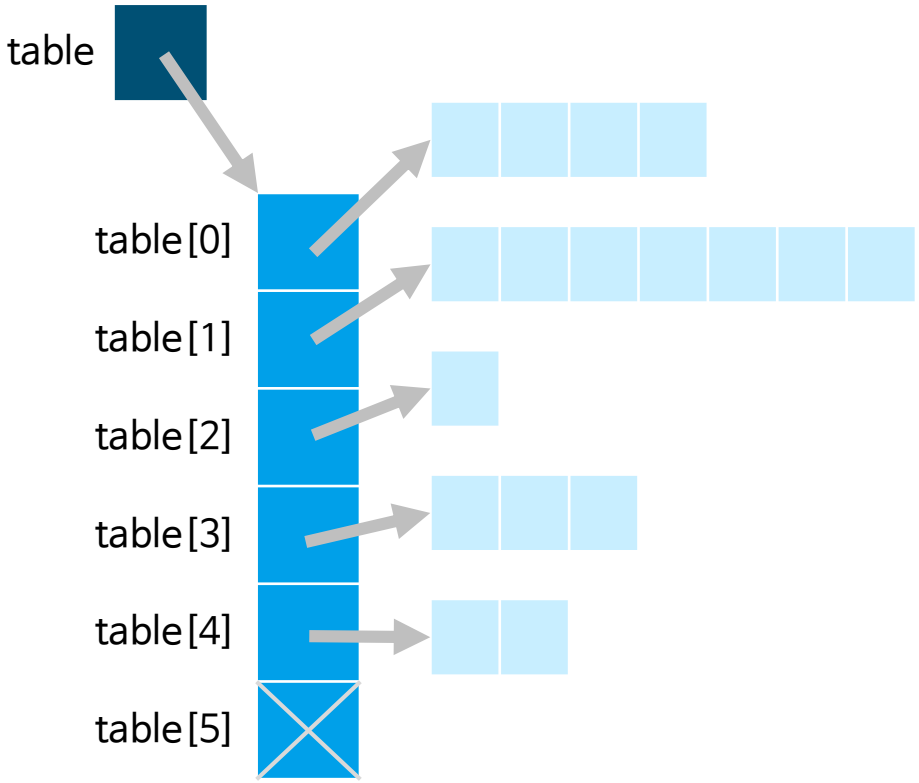



```
int* pi = new int;

delete ptr;


int* pa = new int[5];

delete[] ptr;
```

배열은 []를 붙여 해제하지 않으면 맨 앞의 것만 해제되는데,
그렇게 되면 뒤에 있는 원소는 접근도 못하고 메모리만 차지하게 된다.

포인터 배열

- 포인터를 원소로 하는 배열을 통해 자유롭게 표를 만들 수 있다.




```
int** table;
table = new int*[6];

table[0] = new int[4];
table[1] = new int[7];
table[2] = new int[1];
table[3] = new int[3];
table[4] = new int[2];
table[5] = NULL;
```

배열의 정적 할당과 동적 할당 비교

	정적 할당	동적 할당
선언	int arrayVar[50];	int* ptrVar;
메모리 할당	필요 없다 (로딩될 때 할당)	ptrVar = new int[50];
메모리 해제	불가능	delete ptrVar;
너무 많거나 적은 데이터	메모리 크기를 조절할 수 없다.	가볍게 메모리를 더 할당해준다.
장점	인덱스만을 이용해서 쉽게 사용할 수 있다.	메모리 공간을 효율적으로 관리할 수 있다.
단점	메모리 크기가 고정되어 있어 규모가 큰 프로그램은 너무 무거워진다. 예기치 않은 메모리 요구에 취약하다.	포인터를 관리하는 것이 까다롭다.

동적 할당 사용해보기

사용자가 자유롭게 2차원 배열의
행과 각 열의 원소 개수를 입력하고,
원소를 입력할 수 있도록 하는
프로그램을 작성하시오.
결과 값은 파일로 저장한다.

결과 예시

```
단순한 데이터베이스를 만듭니다.
> 몇 행입니까? 5

> 0행은 몇 개의 원소가 들어있습니까? 4
> 1행은 몇 개의 원소가 들어있습니까? 7
> 2행은 몇 개의 원소가 들어있습니까? 1
> 3행은 몇 개의 원소가 들어있습니까? 3
> 4행은 몇 개의 원소가 들어있습니까? 2

데이터를 입력하시오.
> 0행: 32 18 12 24
> 1행: 13 11 16 12 42 19 14
> 2행: 22
> 3행: 13 13 14
> 4행: 11 18

저장되었습니다.
```

위에서 저장된 파일을 이용하여
데이터를 불러오는 프로그램을
작성하시오.

결과 예시

```
데이터를 불러옵니다...

0행: 32 18 12 24
1행: 13 11 16 12 42 19 14
2행: 22
3행: 13 13 14
4행: 11 18
```

2017.05.04. 프로그래밍 기초 (2017-1)
with D.com