

객체지향언어



4장 복합 자료형

학습목표

- 배열의 개념, 배열의 메모리 구조를 설명할 수 있다.
- 포인터 변수와 값 변수 사이의 관계를 설명할 수 있다
- 포인터 변수를 사용할 수 있다.
- 참조의 개념을 설명하고, 참조를 사용할 수 있다.
- 메모리 할당을 동적으로 선언하고 해제하여 필요할 때만 메모리를 사용할 수 있다.

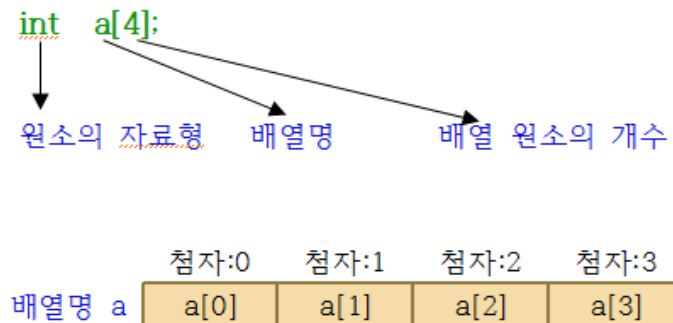
배열

■ 배열(array)

- 배열(array)은 동일한 자료형의 데이터를 연속적으로 저장하는 집합체
- 배열의 선언 : 데이터형 배열명[배열 원소의 개수];
- 배열 사용(접근) : 배열명[배열의 첨자(index)]
 - int arr[N];과 같이 배열 arr이 정의되어 있을 때 원소의 개수는 N이므로, 첨자(인덱스)는 0부터 N-1까지임. 따라서 arr[0]은 첫 번째 요소를 나타내고, arr[N-1]은 마지막 요소를 나타냄

■ 1차원 배열

- int a[4]; // 배열의 선언
- a[2] = 20; // 배열에 접근



```
void main()
{
    int a[2] = {0, 1};
    int b[2] = a;    // Error
}
```

■ 배열의 초기화

- 배열 선언시 초기값을 지정
- 원소의 값을 콤마로 구분하여 나열하고 전체를 { }로 묶어줌
- 원소의 개수보다 초기값이 적은 경우에는 나머지 원소의 값은 수이면 0, 문자 혹은 문자열이면 NULL 값을 가짐
- 배열 원소의 개수보다 더 많은 초기값을 주면 오류가 발생

■ [예]

- `int a[4] = {10, 20, 30, 40};` // 일반적인 초기화
- `int b[] = {10, 20, 30, 40, 50};` // 배열의 크기는 5가 된다.
- `int c[5] = {10, 20, 30};` // 마지막 두 개는 0이 된다.
- `int d[5] = { };` // 모든 배열 값을 0으로 초기화
- `int e[4] = {10, 20, 30, 40, 50, 60};` // 오류 발생
- `char f[5] = {'a', 'b', 'c'};` // 마지막 두 개는 NULL이 된다.

■ ex4_1.cpp (배열)

```
#include <iostream>
using namespace std;

void main()
{
    int score[5] = {82, 93, 91, 80, 73};
    int total = 0;
    double average;

    for (int i=0; i<5; i++)
        total += score[i];

    average = (double)total / 5.0;

    cout << "total = " << total << endl;
    cout << "average = " << average << endl;
}
```

```
total = 419
average = 83.8
계속하려면 아무 키나 누르십시오 . . .
```

배열

■ 배열 원소의 주소와 원소의 값

```
int a[5] = {10, 20, 30, 40, 50};
```

■ 배열의 주소

- 배열의 이름 a 는 배열의 첫 번째 원소의 주소
- 각 원소의 주소는 원소 앞에 주소 연산자(&)를 붙여서 구할 수 있음

■ 배열 원소의 값

- 배열 원소의 값은 색인을 써서 $a[0]$, $a[1]$, ... 으로 구할 수 있음
- 배열의 이름 앞에 간접 연산자(*)를 붙여 $*a$, $*(a+1)$, ... 구할 수 있음

첨자	0	1	2	3	4
배열명: a	10	20	30	40	50
원소 주소	a	$a+1$	$a+2$	$a+3$	$a+4$
원소 주소	$\&a[0]$	$\&a[1]$	$\&a[2]$	$\&a[3]$	$\&a[4]$
원소 값	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$
원소 값	$*a$	$*(a+1)$	$*(a+2)$	$*(a+3)$	$*(a+4)$

■ ex4_2.cpp (1) (1차원 배열의 주소와 원소의 값)

```
#include <iostream>
using namespace std;
void main()
{
    int a[3] = {10, 20, 30};
    int i;

    for (i=0; i<3; i++)
        cout << i << " 번째 원소의 주소: " << (a+i) << endl;
    cout << endl;

    for (i=0; i<3; i++)
        cout << i << " 번째 원소의 주소: " << &a[i] << endl;
    cout << endl;

    for (i=0; i<3; i++)
        cout << i << " 번째 원소의 값: " << a[i] << endl;
    cout << endl;

    for (i=0; i<3; i++)
        cout << i << " 번째 원소의 값: " << *(a+i) << endl;
    cout << endl;
}
```

```
0 번째 원소의 주소: 00A2FF00
1 번째 원소의 주소: 00A2FF04
2 번째 원소의 주소: 00A2FF08

0 번째 원소의 주소: 00A2FF00
1 번째 원소의 주소: 00A2FF04
2 번째 원소의 주소: 00A2FF08

0 번째 원소의 값: 10
1 번째 원소의 값: 20
2 번째 원소의 값: 30

0 번째 원소의 값: 10
1 번째 원소의 값: 20
2 번째 원소의 값: 30
계속하려면 아무 키나 누르십시오 . . .
```

배열

■ 문자 배열

- 문자열은 이중 따옴표로 나타내고, 마지막에 NULL 문자('\0')가 있다
- 마지막에 NULL 문자가 없으면 단순 문자들의 집합이고 문자열로 취급하지 않음

```
char str1[6] = {'k', 'o', 'r', 'e', 'a', '\0'};    // 문자열
char str2[6] = {"korea"};                        // 문자열로, 배열의 크기는 6
char str3[5] = {'k', 'o', 'r', 'e', 'a'};        // 문자들의 집합으로 문자열은 아님
char str4[5] = {"korea"};                        // 오류
```

배열명	str2[0]	str2[1]	str2[2]	str2[3]	str2[4]	str2[5]
str2	'k'	'o'	'r'	'e'	'a'	'\0'

(a) str2의 구조, str1의 구조도 같다.

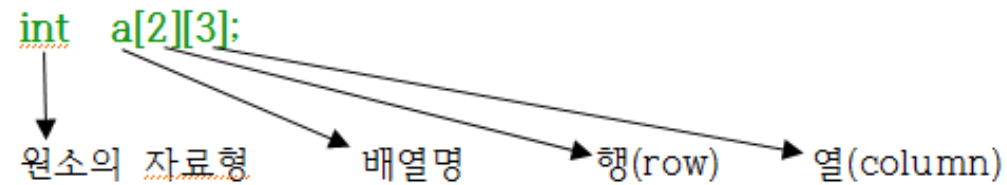
배열명	str3[0]	str3[1]	str3[2]	str3[3]	str3[4]
str3	'k'	'o'	'r'	'e'	'a'

(b) str3의 구조 - 마지막에 널 문자가 없다.

배열

■ 2차원 배열

- `int a[2][3];`
- `int a[2][3] = {{1, 2, 3}, {4, 5, 6}};`
- `int a[2][3] = {1, 2, 3, 4, 5, 6};`
- 행우선 순위 방식으로 기억 공간을 차지
 - `a[0][0]`, `a[0][1]`, `a[0][2]`, `a[1][0]`, `a[1][1]`, `a[1][2]`의 순서로 배치



	0열	1열	2열
0행	a[0][0]	a[0][1]	a[0][2]
1행	a[1][0]	a[1][1]	a[1][2]

주소	메모리(원소값)
a[0]	a[0][0]
a[0]+1	a[0][1]
a[0]+2	a[0][2]
a[1]+3	a[1][0]
a[1]+4	a[1][1]
a[1]+5	a[1][2]

■ ex4_3.cpp (1) (2차원 배열의 주소와 원소의 값)

```
#include <iostream>
using namespace std;
void main()
{
```

```
    int a[2][3] = {1, 2, 3, 4, 5, 6};
    int i, j;
    for (i=0; i<2; i++)
    {
        for (j=0; j<3; j++)
        {
            cout << "&a[" << i << "][" << j << "] = " << &a[i][j] << ", " ;
            cout << "a[" << i << "][" << j << "] = " << a[i][j] << endl ;
        }
    }
```

```
    cout << endl;
    for (i=0; i<6; i++)
    {
```

```
        cout << "a[0]+" << i << " = " << a[0]+i << ", " ;
        cout << "*(a[0]+" << i << ") = " << *(a[0]+i) << endl ;
    }
```

```
    cout << endl;
}
```

```
&a[0][0] = 0012FF4C, a[0][0] = 1
&a[0][1] = 0012FF50, a[0][1] = 2
&a[0][2] = 0012FF54, a[0][2] = 3
&a[1][0] = 0012FF58, a[1][0] = 4
&a[1][1] = 0012FF5C, a[1][1] = 5
&a[1][2] = 0012FF60, a[1][2] = 6
```

```
a[0]+0 = 0012FF4C, *(a[0]+0) = 1
a[0]+1 = 0012FF50, *(a[0]+1) = 2
a[0]+2 = 0012FF54, *(a[0]+2) = 3
a[0]+3 = 0012FF58, *(a[0]+3) = 4
a[0]+4 = 0012FF5C, *(a[0]+4) = 5
a[0]+5 = 0012FF60, *(a[0]+5) = 6
```

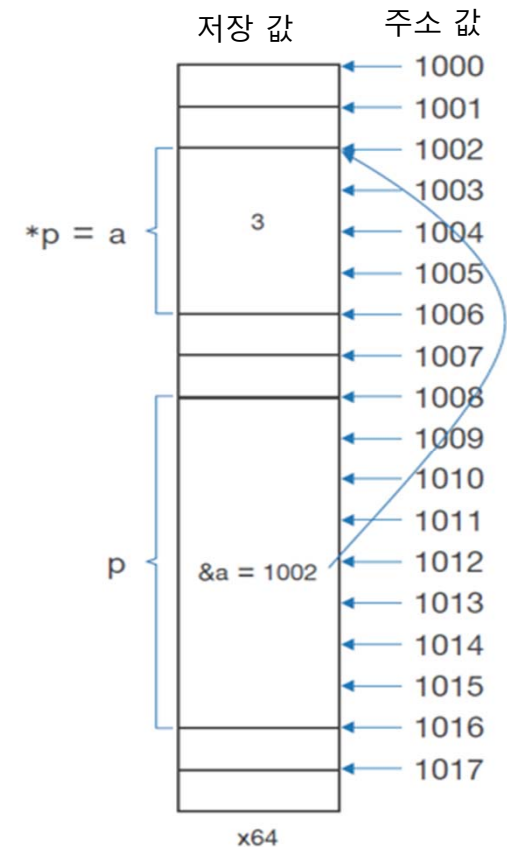
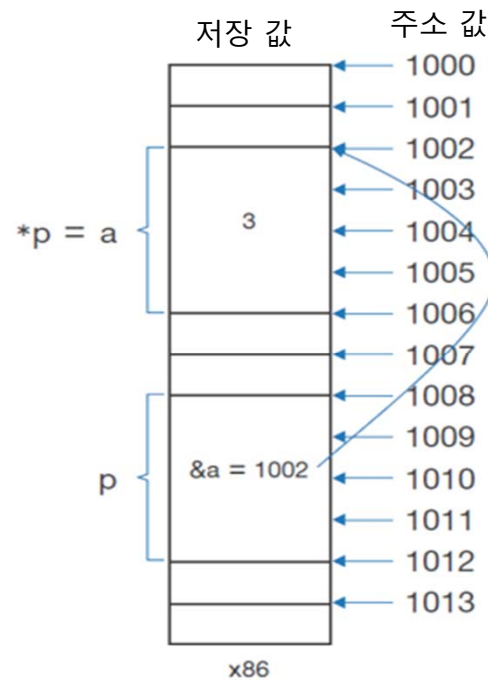
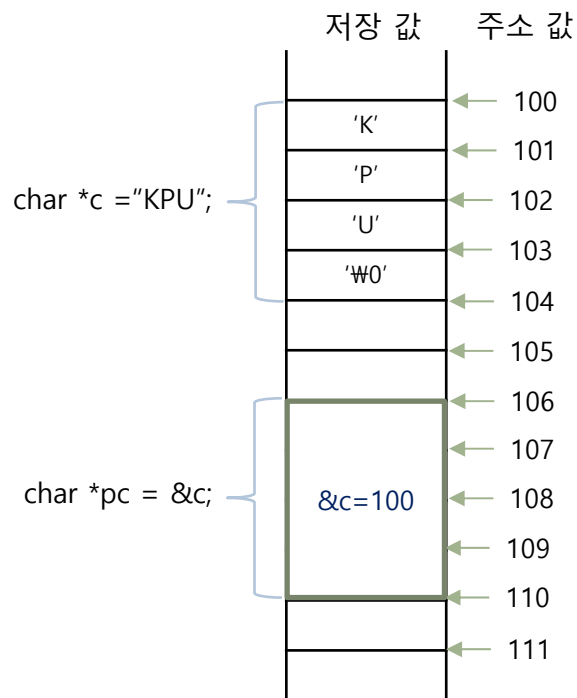
계속하려면 아무 키나 누르십시오 . . .

포인터

■ 기억 공간

- 물리적인 기억공간을 바이트(byte, 8비트) 단위로 구분하기 위하여 주소(address)를 사용

■ 기억공간의 주소 예



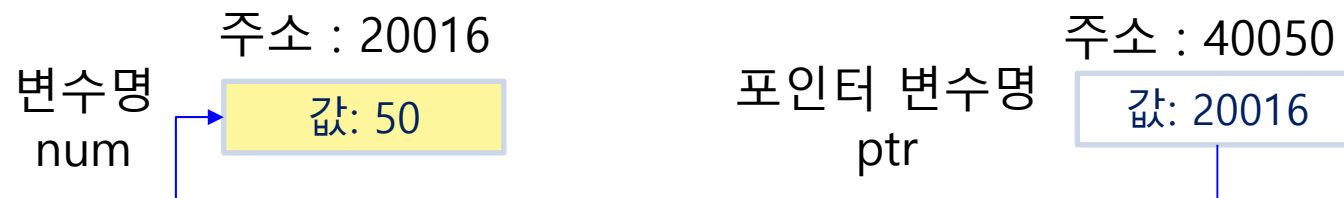
포인터

■ 포인터(포인터 변수)

- 변수의 주소를 가지고 있는 변수를 포인터(pointer)라고 함
- 포인터 변수는 주소를 가질 뿐이고, 이 주소가 가리키는 기억공간에 접근하여야 실제 값을 구할 수 있음

■ [예]

```
int num;           // num은 int형 자료를 가지는 변수
int *ptr;          // ptr은 int형 자료를 가리키는 주소를 가지는 변수
num = 50;          // num은 50이라는 int형의 값을 가진다.
ptr = &num;        // ptr은 num의 주소를 가진다.
cout << *ptr;      // *ptr은 ptr이 가리키는 기억공간의 값이다.
```



포인터

■ 포인터

- 포인터 변수를 선언할 때

- * 는 변수가 포인터라는 것을 나타냄

```
int * pnum;
```

- 포인터 변수가 가리키는 기억공간의 값을 나타낼 때

- * 는 포인터 변수가 참조하는 값이란 의미

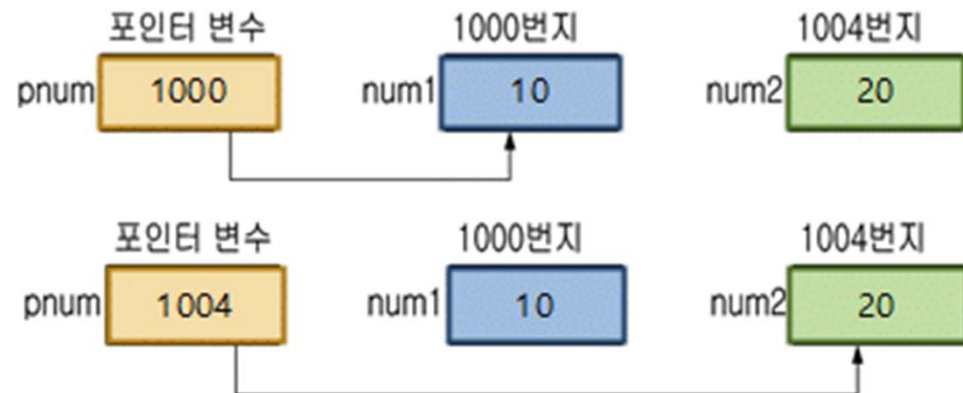
```
cout << *pnum;
```

- [예]

```
int num1 = 10, num2 = 20;
```

```
int* pnum = &num1;
```

```
pnum = &num2;
```



포인터

■ ex4_4.cpp (포인터 사용)

```
#include <iostream>
using namespace std;

void main()
{
    int num1 = 10, num2 = 20 ;
    int* pnum = &num1;

    cout << "*pnum = " << *pnum << endl;

    pnum = &num2;
    cout << "*pnum = " << *pnum << endl;
}
```

```
*pnum = 10
*pnum = 20
계속하려면 아무 키나 누르십시오 . . .
```

포인터

■ 포인터 선언

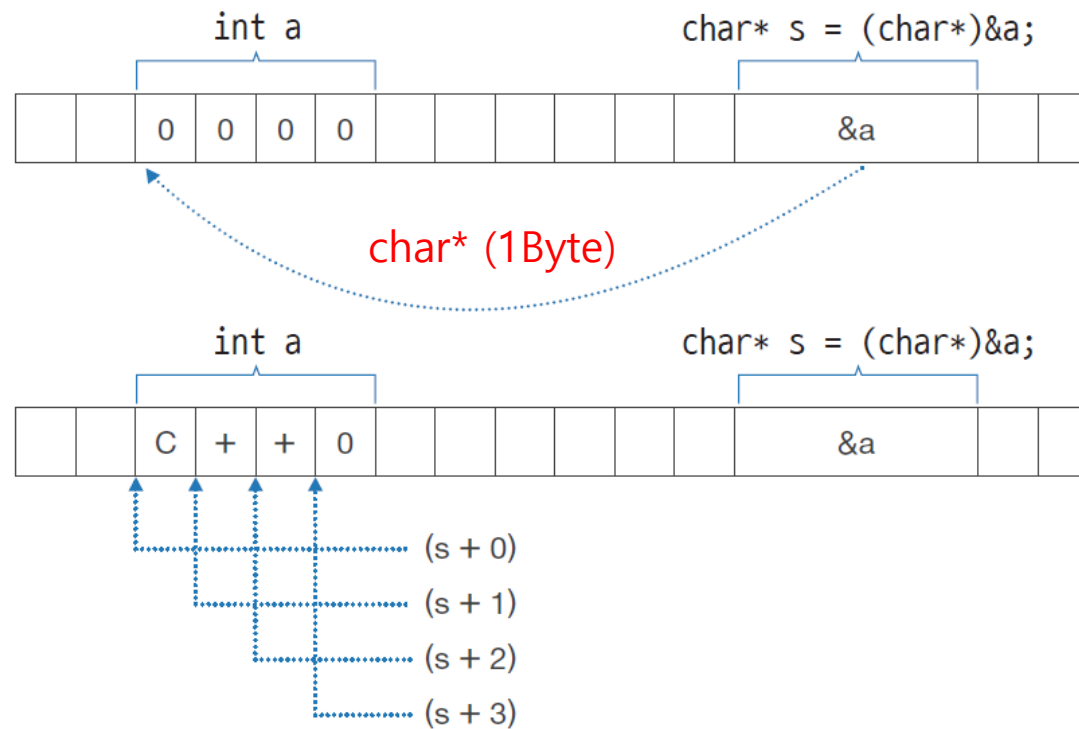
- 메모리에는 변수의 값만 저장될 뿐 변수의 자료형은 저장하지 않음
- 변수의 값만으로는 저장되어 있는 것이 char인지, int인지, float인지 알 수가 없다. 변수의 크기도 알 수 없음
- 포인터 형은 메모리의 주소만을 가지고 있음
- 주소만으로는 가리키는 곳에 무엇이 있는지 알 수 없음
- 포인터 변수는 선언할 때 어떤 자료형의 변수를 가리키는지 알려주어야 함

```
int a;  
int *pnum = &a;
```

```
void main()  
{  
    char c;  
    int i;  
  
    //char* pC = &i;  // Error  
    char* pC = &c;    // OK  
  
    //int* pI = &c;   // Error  
    int* pI = &i;    // OK  
}
```

포인터

- 대상 타입과 실제 타입의 불일치
 - 대상 타입은 char, 실제 타입은 int



포인터

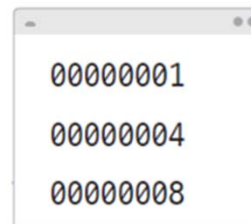
■ 포인터 연산

- 포인터에 N을 더하는 것은 주소가 N*sizeof(대상 데이터형) 만큼 증가한 것과 같음

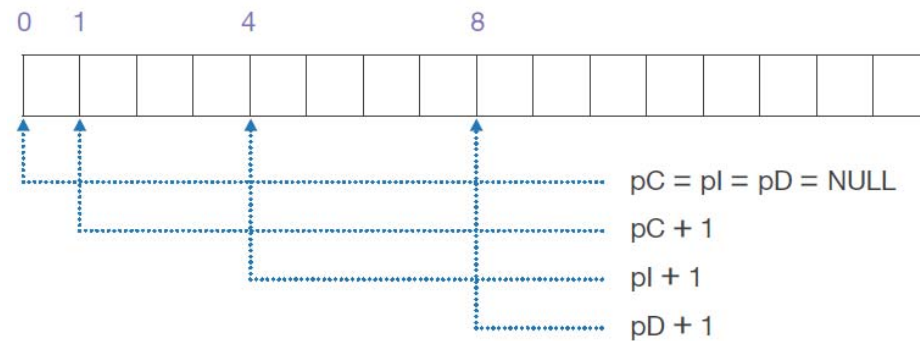
```
#include <iostream>
using namespace std;
```

```
void main()
{
    char* pC = NULL;
    int* pI = NULL;
    double* pD = NULL;

    cout << (void*)(pC + 1) << endl;
    cout << (void*)(pI + 1) << endl;
    cout << (void*)(pD + 1) << endl;
}
```



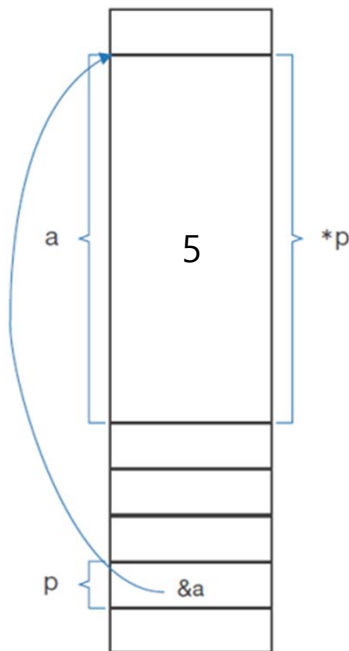
```
00000001
00000004
00000008
```



포인터

■ 간접(*) 연산자

```
int a = 5;  
int* p = &a;  
*p = a;
```

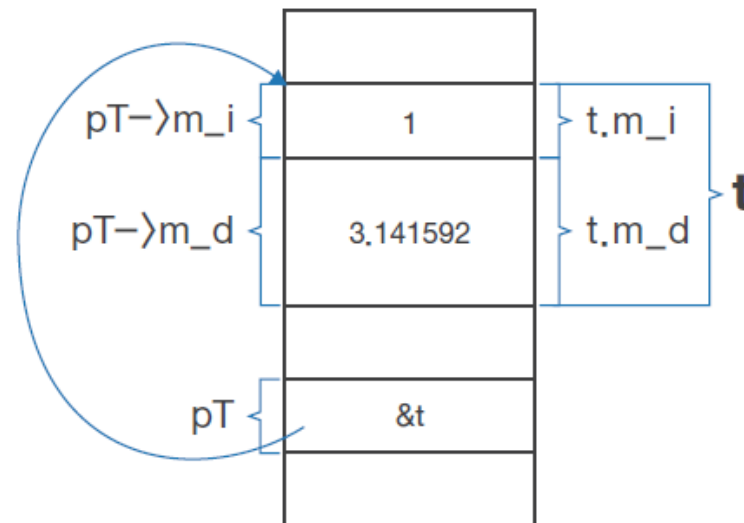


■ 간접 멤버(->) 연산자

```
struct Ctest{  
    int m_i;  
    double m_d;  
};
```

```
struct Ctest t;  
Ctest* pT = &t;
```

```
pT->m_i = 1;  
pT->m_d = 3.141592;
```



포인터

■ void 자료형은 '자료형이 없다'는 의미

- 함수의 전달인자로 void를 사용하면 전달인자가 없는 것이고, 함수가 값을 반환하지 않으면 void를 반환형으로 사용
- C++에서 void라는 자료형은 있지만 void 자료형의 변수는 존재할 수 없음

■ void 포인터 (1)

`void* p;` // p는 모든 자료형을 가리킬 수 있음

- void 포인터는 모든 자료형을 가리킬 수 있는 포인터로 char, int, float 등 모두 다 가리킬 수 있음
- void 포인터에는 지금 가리키고 있는 변수가 어떤 자료형 인지 에 관한 정보가 없기 때문에 할 수 있는 일이 없음
- void 포인터는 아무 자료형이나 가리킬 수 있기 때문에 주소를 저장하는 용도로만 사용
- 다음과 같은 코드는 C, C++에서 모두 사용할 수 있음

`int i = 100;`

`void* pv = &i;` // void * 자료형은 모든 자료형을 가리킬 수 있으므로,
// 형 변환 없이도 &i의 값을 보관할 수 있다.

포인터

■ void 포인터 (2)

- void 포인터 형은 특정 자료형의 포인터 형으로 변환할 수 없음
- 보관된 주소를 사용하기 위해서는 특정 자료형의 포인터로 형 변환되어야 함
- C에서는 void 포인터가 묵시적으로 다른 자료형의 포인터 형으로 형 변환을 용인하지만, C++에서는 엄격한 형 검사(type checking)를 적용하기 때문에 묵시적인 형 변환은 허용하지 않고 **명시적으로 형 변환만 허용**

```
int i = 100;
```

```
void* pv = &i;    // pv는 i의 주소를 보관
```

```
int* pi = (int*)pv; // 보관하고 있는 주소를 int를 가리키는  
                  // 주소로 사용하기 위해 void* 자료형인 pv의 값을  
                  // int* 자료형으로 형 변환하여야 한다.
```

포인터

■ ex4_5.cpp (void 포인터의 사용)

```
#include <iostream>
using namespace std;

void main()
{
    int i = 100;
    void* pv = &i; // void* 자료형은 모든 자료형을 가리킬 수 있으므로,
                  // 형 변환 없이도 &i의 값을 보관할 수 있다.

    // int* pi = pv;    // 오류

    int* pi = (int*)pv; // 보관하고 있는 주소를 int를 가리키는 주소로
                      // 사용하기 위해 명시적으로 형 변환하여야 한다.

    cout << *pi << endl;
}
```

100
계속하려면 아무 키나 누르십시오 . . .

■ const 제한자

- const 제한자는 메모리가 일단 초기화된 후에는 프로그램이 그 메모리를 변경할 수 없음
- const를 사용해서 변수를 정의할 때는 반드시 초기화해 주어야 함
- const 속성을 부여하는 경우
 - 처음 정의할 때부터 const로 정의하는 경우로 상수를 대신해서 사용할 때
 - 원래는 const가 아니지만 다른 곳에서 넘겨받을 때 const로 하는 경우로 다른 곳의 변수 값을 제공받기는 하지만 그곳에서는 읽기만 할 수 있을 뿐이고 변수의 값을 바꿀 수 없게 하고 싶을 때

포인터

■ const 제한자

```
int ia = 100;  
int ib = 200;
```

```
const int* p1 = &ia;    // p1은 const int 형을 가리키는 포인터이다.  
p1 = &ib;              // p1은 변경할 수 있다.  
*p1 = 300;             // 오류: p1이 가리키는 변수의 값은 변경할 수 없다.
```

```
int* const p2 = &ia;    // int 형을 가리키는 p2는 const로 제한된다.  
p2 = &ib;              // 오류: p2는 변경할 수 없다.  
*p2 = 300;            // p2가 가리키는 변수의 값은 변경할 수 있다.
```


```
const int* const p3 = &ia;  
// 포인터 p3과 p3이 가리키는 변수 모두 const로 제  
한된다.  
p3 = &ib;              // 오류: p3은 변경할 수 없다.  
*p3 = 300;            // 오류: p3이 가리키는 변수의 값은 변경할 수 없다.
```

포인터

■ 포인터로서의 배열명

```
#include <iostream>
using namespace std;

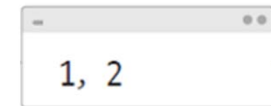
void main()
{
    int arr[2] = {1, 2};
    int* p = arr;
    cout << *p << ", " << *arr << endl;
}
```



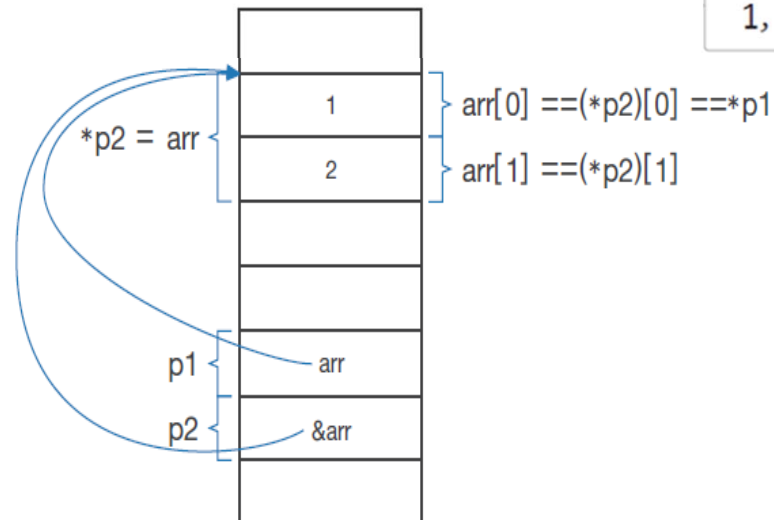
1, 1

```
#include <iostream>
using namespace std;

void main()
{
    int arr[2] = {1, 2};
    //int* p = &arr; // Error
    int (*p)[2] = &arr;
    cout << (*p)[0] << ", " << (*p)[1] << endl;
}
```



1, 2



포인터

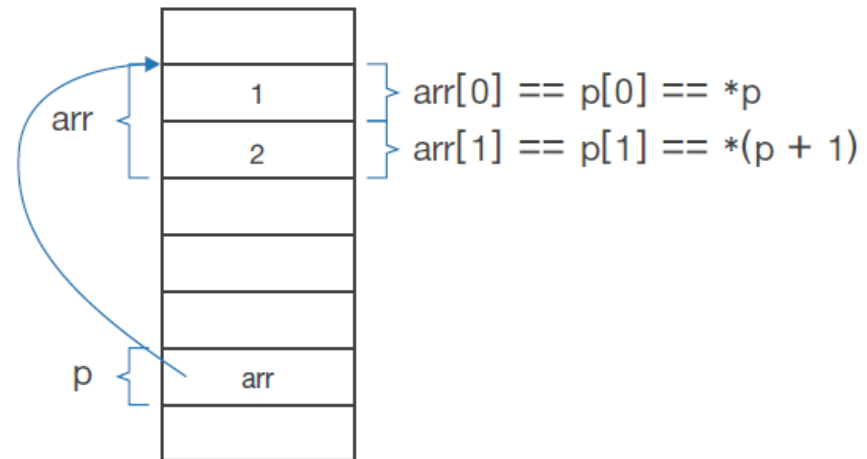
■ 1차원 배열과 포인터

```
#include <iostream>
using namespace std;

void main()
{
    int arr[2] = { 1, 2 };
    int* p = arr;
    cout << p[0] << ", " << p[1] << endl;
}
```



1, 2



포인터

■ 2차원 배열과 포인터

```
#include <iostream>
using namespace std;

void main()
{
    int arr[2][3] = { { 11, 12, 13 }, { 21, 22, 23 } };
    int (*pp)[3] = arr;
    cout << pp[1][2] << endl;
}
```

A small window with a title bar and three control buttons (minimize, maximize, close) in the top right corner. The window contains the text "23" in a monospaced font.

포인터

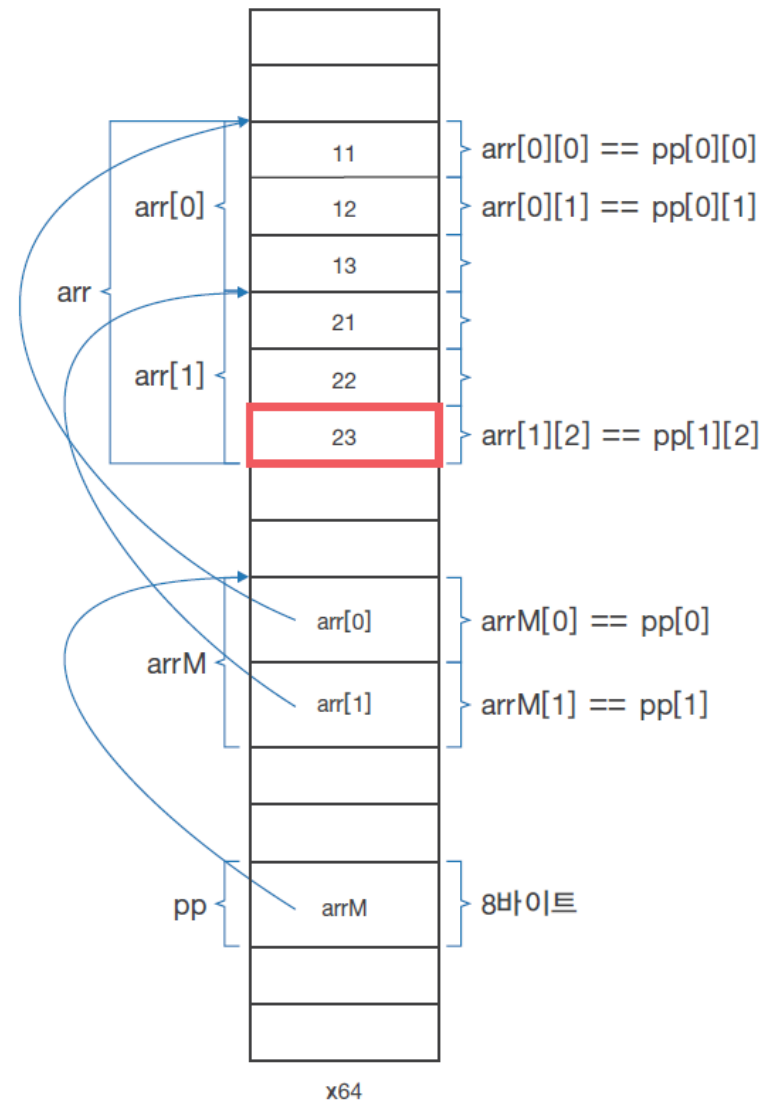
■ 2차원 배열과 포인터의 포인터

```
#include <iostream>
using namespace std;

void main()
{
    int arr[2][3] = { { 11, 12, 13 }, { 21, 22, 23 } };

    int* arrM[2] = { arr[0], arr[1] };
    int** pp = arrM;
    cout << pp[1][2] << endl;
}
```

23



포인터

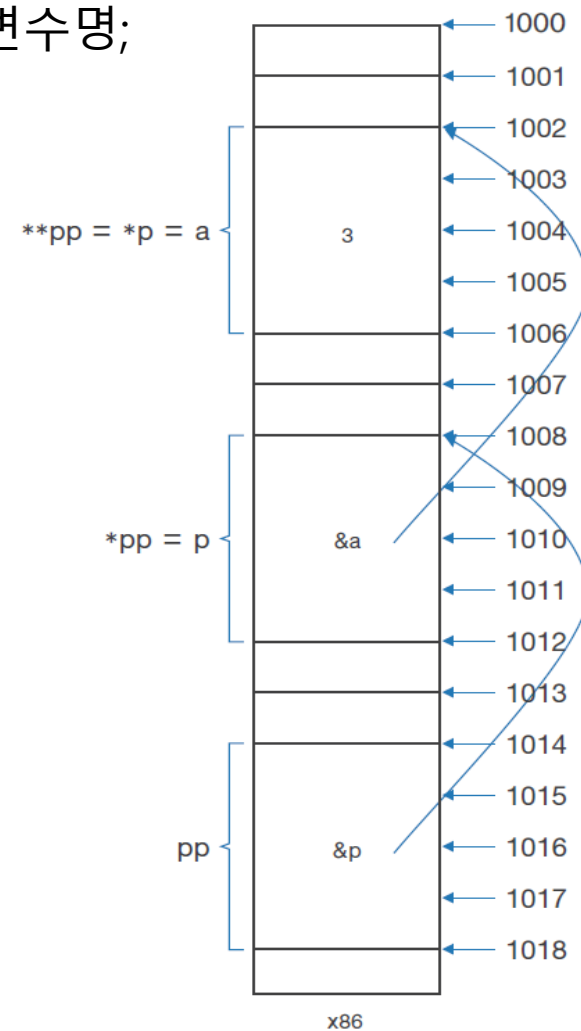
■ 이중 포인터

- 포인터 변수의 주소를 저장하는 포인터 변수
- 선언 형식 : 데이터 타입 ** 포인터 변수명;

```
#include <iostream>
using namespace std;

void main()
{
    int a;
    int* p = &a;
    int** pp = &p;

    **pp = 3;
    cout << a << endl;
}
```



포인터

■ ex4_6.cpp (2중 포인터의 사용)

```
#include <iostream>
using namespace std;

void main()
{
    int num = 100;
    int *pnum, **ppnum;

    pnum = &num;
    ppnum = &pnum;

    cout << "num : " << num << endl;
    cout << "*pnum : " << *pnum << endl;
    cout << "**ppnum : " << **ppnum << endl << endl;

    cout << "ppnum : " << ppnum << endl;
    cout << "pnum의 주소: " << &pnum << endl;
    cout << "pnum : " << pnum << endl;
    cout << "num의 주소: " << &num << endl;
}
```

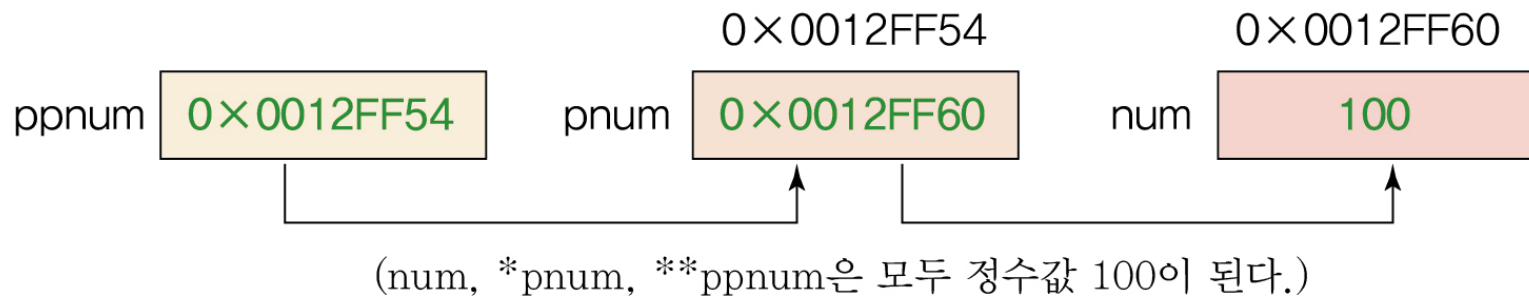
```
num : 100
*pnum : 100
**ppnum : 100

ppnum : 00BFFA48
pnum의 주소: 00BFFA48
pnum : 00BFFA54
num의 주소: 00BFFA54
계속하려면 아무 키나 누르십시오 . . .
```

포인터

■ 2중 포인터의 사용 예

```
int  num;           // num은 int형 자료가 들어가는 변수이다.  
int * pnum;         // pnum는 포인터 변수로, 주소가 들어가고  
                   // pnum가 가리키는 곳에는 int형 자료가 들어간다.  
int **ppnum;        // ppnum는 포인터 변수로  
                   // ppnum가 가리키는 곳에는 또한 주소가 들어가고,  
                   // 그 주소가 가리키는 곳에는 int형 자료가 들어간다.  
  
num = 100;  
pnum = &num;  
ppnum = &pnum;
```



포인터

■ 구조체의 포인터

- 구조체 변수의 멤버는 점 연산자(.)를 사용하여 구할 수 있음
- 구조체 포인터 변수는 간접 연산자(*)를 사용하여 포인터가 가리키는 값을 구할 수 있음
- 구조체 변수를 직접 사용하여 포인터가 가리키는 값을 구하려면 화살표 연산자(->)를 사용해야함

```
struct student
{
    int id;
    char name[20];
};
student stu = {1234, "Hong Gildong"};
student *pstu;
pstu = &stu;

cout << stu.id;
cout << (*pstu).id;
cout << pstu->id;
```

■ ex4_7.cpp (구조체의 포인터 사용)

```
#include <iostream>
using namespace std;
struct student {
    int id;
    char name[20];
};
int main()
{
    student stu1 = {1234, "Hong Gildong"} ;
    student stu2 = {2345, "Lee Soonsin"} ;
    student *pstu;
    pstu = &stu2;

    cout << "stu1.id : " << stu1.id << ", stu1.name : " << stu1.name << endl;
    cout << "(*pstu).id : " << (*pstu).id << ", (*pstu).name : " << (*pstu).name << endl;
    cout << "pstu->id : " << pstu->id << ", pstu->name : " << pstu->name << endl;

    return 0;
}
```

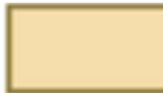
```
stu1.id : 1234, stu1.name : Hong Gildong
(*pstu).id : 2345, (*pstu).name : Lee Soonsin
pstu->id : 2345, pstu->name : Lee Soonsin
계속하려면 아무 키나 누르십시오 . . .
```


참조

■ 참조(reference)

- 참조는 변수를 참조하는 것으로 변수의 다른 이름(별명)이라고도 함
- 참조를 선언하려면 참조하는 변수가 먼저 선언되어 있어야 하고, 그 다음에 참조하는 이름 앞에 &(참조 변수 수식어)를 붙여서 참조 변수를 만들 수 있음
- 참조를 선언할 때 반드시 참조하는 변수로 초기화가 동시에 이루어 져야 함
- 참조를 선언할 때 &는 주소를 뜻하는 것이 아니고 참조형 변수라는 의미임
- 올바른 참조변수 선언 방법

```
int  number1;           // 참조하는 변수
int  &no1 = number1;    // 참조 변수 선언과 동시에
                        // 참조하는 변수로 초기화하여 함
```

number1 
no1

- 잘못된 참조변수 선언 방법

```
int  number1;
int  &no1;           // 잘못된 예
no1 = number1;       // 참조변수 선언 후 참조하는 변수로 초기화하면 안 됨
```

■ ex4_8.cpp (참조의 사용)

```
#include <iostream>
using namespace std;
```

```
void main()
{
    int num = 10;
    int &rnum = num;

    cout << "num = " << num << ", rnum = " << rnum << endl;

    rnum += 20;
    cout << "num = " << num << ", rnum = " << rnum << endl;
}
```

```
num = 10,  rnum = 10
num = 30,  rnum = 30
계속하려면 아무 키나 누르십시오 . . . _
```

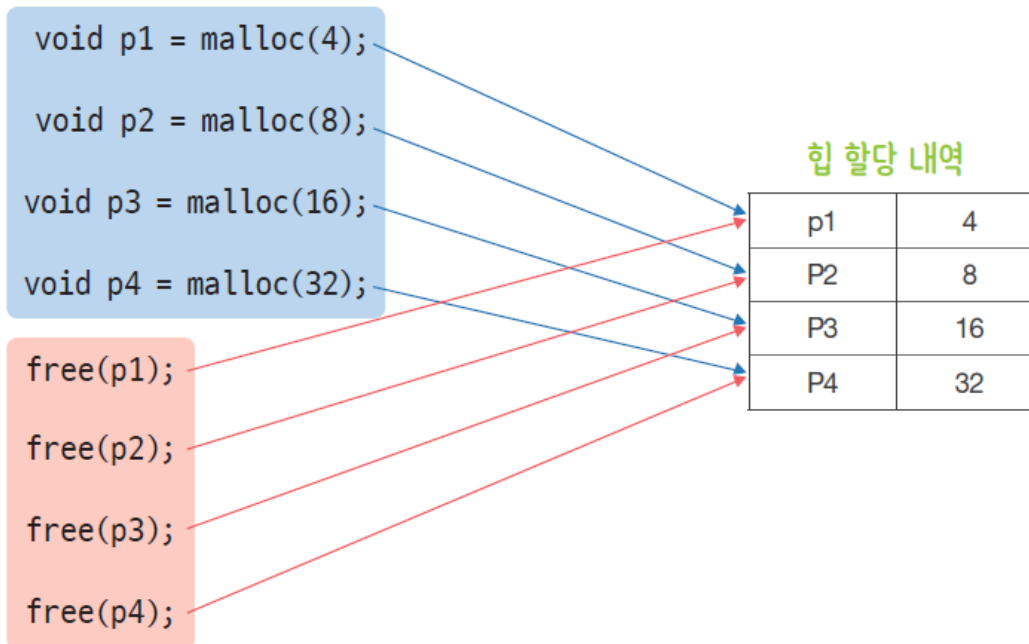
동적 메모리 관리

■ 동적 메모리 관리

- 필요한 시점에만 메모리를 할당하고, 필요가 없을 때는 메모리를 해제
- C 언어에서는 malloc(), calloc(), realloc(), free()와 같은 함수를 사용

```
void* malloc(size_t size);  
void free(void* memblock);
```

```
void* realloc(void* memblock, size_t size);
```



동적 메모리 관리

■ 동적 메모리 관리

■ malloc & free

```
#include <iostream>
using namespace std;

#include <stdlib.h>

void main()
{
    int a = 1;
    int* p = (int*)malloc(sizeof(int));
    *p = 2;
    cout << a << endl;    // 1
    cout << *p << endl;   // 2

    free(p);
}
```



```
1
2
```

calloc & free

```
#include <iostream>
using namespace std;

void main()
{
    int* p1 = (int*)malloc(sizeof(int));
    int* p2 = (int*)calloc(1, sizeof(int));

    cout << *p1 << endl;
    cout << *p2 << endl;

    free(p1);
    free(p2);
}
```



```
4390988
0
```

동적 메모리 관리

■ 동적 메모리 관리

■ realloc & free

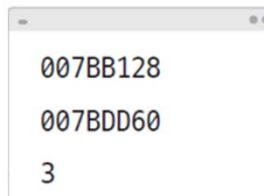
```
#include <iostream>
using namespace std;

void main()
{
    int* p1 = (int*)malloc(sizeof(int));
    *p1 = 3;

    int* p2 = (int*)realloc(p1, 1024 * sizeof(int));

    cout << p1 << endl;
    cout << p2 << endl;
    cout << p2[0] << endl;

    free(p1);    // Error
    free(p2);    // OK
}
```



```
007BB128
007BDD60
3
```

동적 메모리 관리

■ 동적 메모리 관리

- C++에서는 new, delete라는 연산자를 사용해서 메모리를 동적으로 관리

■ new, delete라는 연산자를 활용한 동적 메모리 활용 순서

(1) 기억공간의 주소를 저장할 포인터를 선언

- `int *pn;` // int형 자료를 저장할 때
- `char *pc;` // char형 자료를 저장할 때

(2) new 연산자로 기억공간을 확보하고 주소를 포인터에 저장

- `pn = new int;` // int형 데이터를 저장할 기억공간의 주소를 pn에 기억
- `pc = new char[10];` // char형 변수 10개의 기억공간의 주소를 pc에 기억

(3) 포인터를 이용하여 기억공간을 사용

- `*pn = 100;` // 동적 기억공간에 100을 넣어준다.

(4) 사용이 끝난 지점에서 바로 기억공간을 해제

- `delete pn;` // 배열 할당이 아닌 경우에 대한 해제
- `delete [] pc;` // 배열 할당한 메모리의 해제

동적 메모리 관리

■ ex4_9.cpp (동적 메모리 관리)

```
#include <iostream>
using namespace std;

void main()
{
    int *id;                // 포인터 선언
    id = new int ;          // 기억공간 할당
    cout << "번호를 입력하세요: " ;
    cin >> *id;

    char *name = new char[20] ; // 포인터 선언과 기억공간 할당
    cout << "이름을 입력하세요: " ;
    cin >> name;

    cout << "id : " << *id << ", name : " << name << endl;

    delete id;              // 기억공간 해제
    delete [] name;         // 배열 기억공간 해제
}
```

```
번호를 입력하세요 : 1234
이름을 입력하세요 : Hong
id : 1234, name : Hong
계속하려면 아무 키나 누르십시오 . . .
```

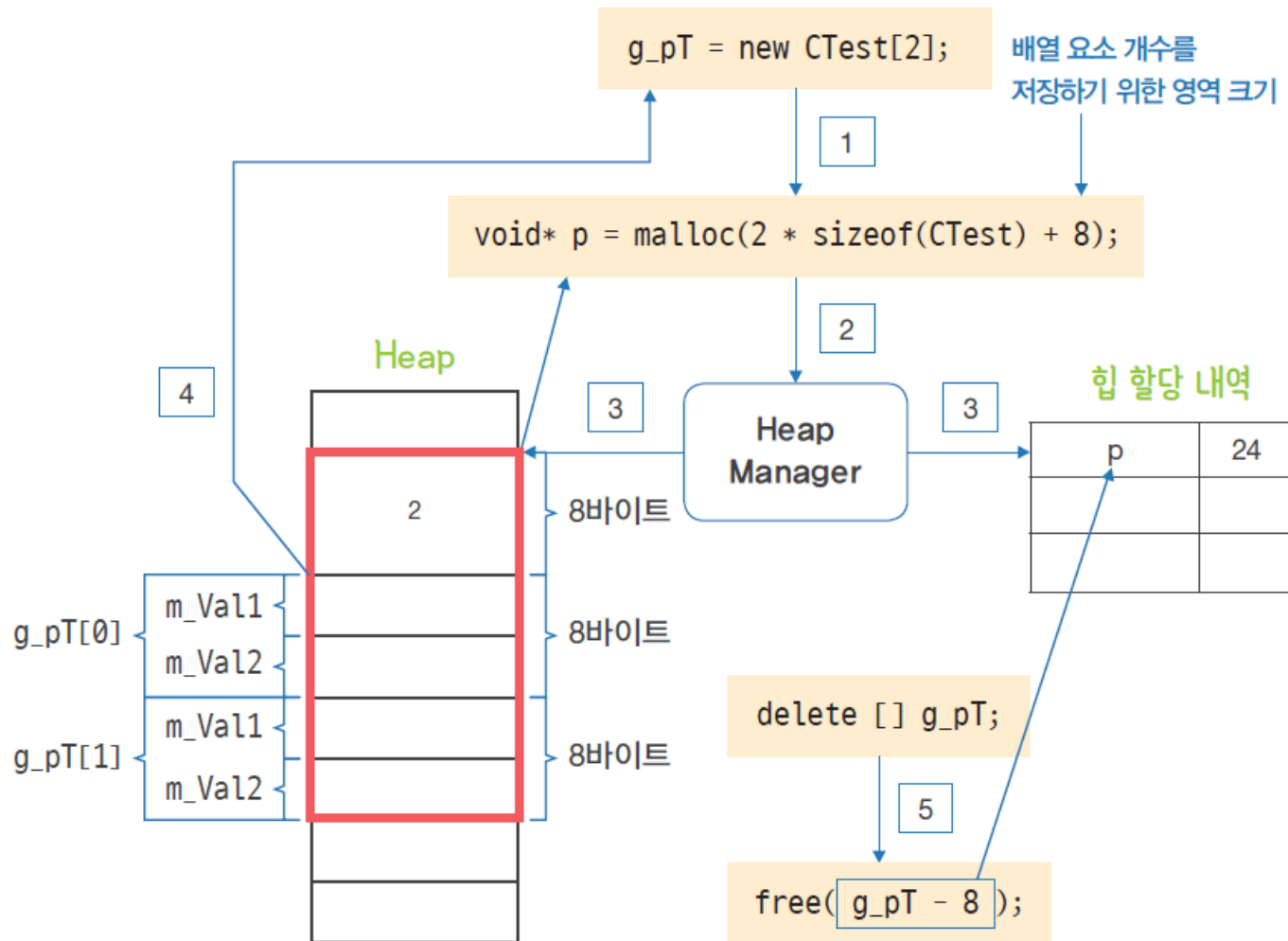
동적 메모리 관리

■ 동적 메모리 관리에서 지켜야 할 사항

- new로 메모리를 할당한 경우에는 delete로 해제
- new와 delete를 사용할 때 메모리 블록의 크기가 맞아야 함
 - 크기가 맞지 않으면 실행 시간 오류가 발생
- new []로 배열로 할당한 경우에는 delete []로 해제하여야 하고, new로 배열이 아니게 할당한 경우에는 그냥 delete로 해제
- new로 할당하지 않은 메모리는 delete로 해제하지 않음
- 같은 메모리 블록을 두 번 연달아 delete로 해제하지 않음
- NULL 포인터는 delete로 해제해도 아무 일도 일어나지 않음

동적 메모리 관리

delete와 delete [] 차이점



동적 메모리 관리

■ ex4_10.cpp (구조체의 동적 메모리 관리)

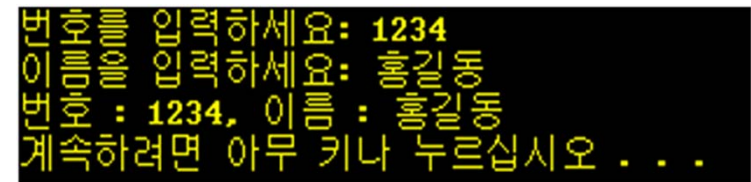
```
#include <iostream>
using namespace std;
struct Student {
    int id;
    char name[20];
};
void main()
{
    Student *pst;

    pst = new Student;                // 기억공간을 동적으로 할당

    cout << "번호를 입력하세요: " ;
    cin >> pst->id;

    cout << "이름을 입력하세요: " ;
    cin >> pst->name;

    cout << "번호 : " << pst->id << ", 이름 : " << pst->name << endl;
    delete pst;                      // 기억공간 해제
}
```



```
번호를 입력하세요: 1234
이름을 입력하세요: 홍길동
번호 : 1234, 이름 : 홍길동
계속하려면 아무 키나 누르십시오 . . .
```



Thank You
