

3장. 포인터, 배열, 구조체

□ 포인터, 배열, 구조체

- 기본적 데이터 타입
- 기본 데이터 타입 위에서 더욱 복잡한 자료구조가 형성됨
- 활성화 레코드는 재귀호출의 이해하기 위해 필요

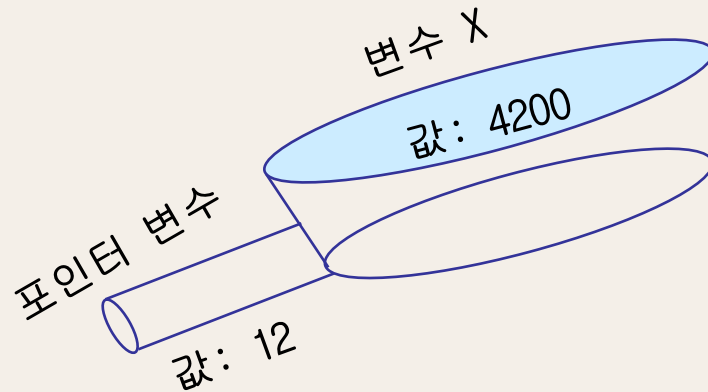
□ 학습목표

- 포인터, 배열, 구조체의 기본 정의를 이해한다.
- 동적 메모리의 할당 및 반납, 주소 연산자 등에 대해 이해한다.
- 참조호출과 값 호출의 차이점을 이해한다.
- 함수 호출시에 배열, 구조체의 전달 메커니즘을 이해한다.
- 복사생성자가 필요한 근본적인 이유를 이해한다.
- 활성화 레코드가 필요한 이유와 그 구성요소를 이해한다.
- 어서트 매크로를 사용하는 방법을 익힌다.

포인터

□ 포인터 변수

- 주소값만을 저장할 수 있는 변수
- 팬 핸들을 통한 변수 접근 cf. 변수명을 통한 접근



주소	값	변수
12	4200	X
16	8600	Y

48	9400	Z
49	12	p

포인터 선언

int Date, Month;

Date Month 변수는 정수 타입

int *p;

p가 가리키는 것은 정수 타입

float *q;

q가 가리키는 것은 부동소수 타입

□ int *p;

- “p가 가리키는 것은 정수 타입”
- “ ‘Something pointed by p is of type integer’ ”
- 애스터리스크(Asterisk, *)는 참조의 대상을 찾아내는(Dereference, Indirection) 연산자

주소	값	변수
200		Date
204		Month

404	?	p
408	?	q

- 현재 상태에서 *p = 15; 명령은 오류. *p가 정의되어 있지 않음.

동적 변수공간 할당

동적 변수공간 할당

- `p = (int *) malloc(sizeof(int));`
- `*p = 15;`
- *p는 힙 메모리 변수, p는 스택 메모리 변수

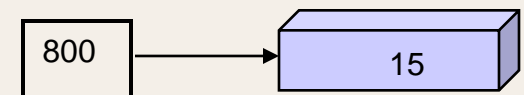
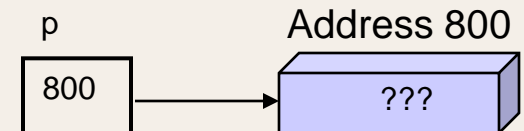
주소	값	변수
200		Date
204		Month

404	800	p
408		q

800	15	?
-----	----	---

`p=(int*)malloc(sizeof(int));`

`*p=15;`



C/C++의 동적 변수 할당

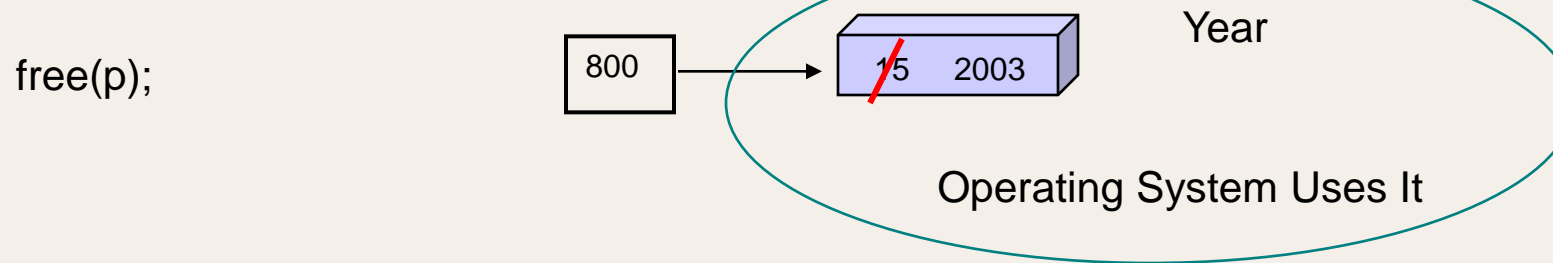
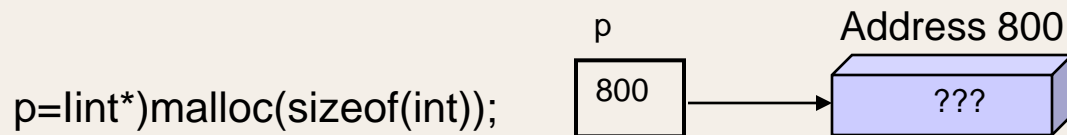
□ C/C++의 동적 변수 할당

	C	C++
동적변수 할당	<code>p = (int *)malloc(sizeof(int));</code>	<code>p = new int;</code>
동적변수 해제	<code>free p;</code>	<code>delete p;</code>

동적 변수공간의 반납

□ 공간반납

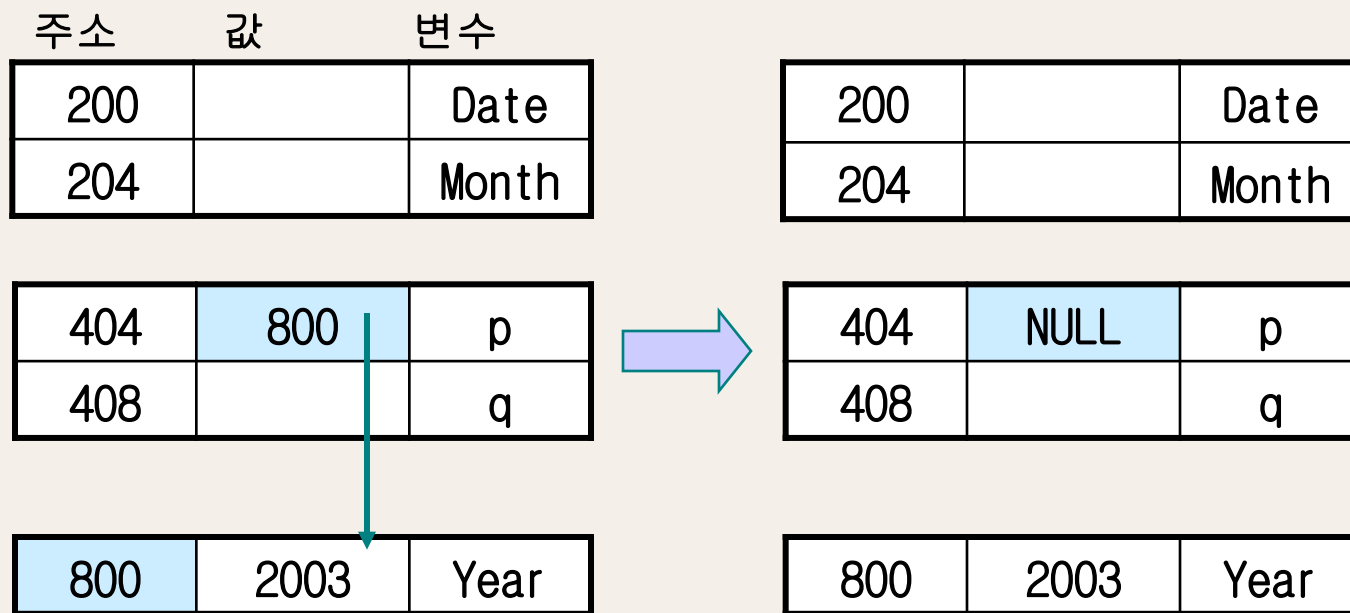
- *p 공간은 반납
- p 공간은 그대로 유지. 값도 그대로 유지
- p를 따라가는 것은 문제가 됨.



동적 변수공간의 반납

□ 널 포인터

- ‘현재 아무 것도 가리키지 않고 있다’ 는 의미
- 값이 정의가 안 된(Undefined) 포인터와는 의미가 다름
- `free (p);` `p`가 가리키는 변수 공간을 반납
- `p = NULL;` `p`를 따라가지 않도록

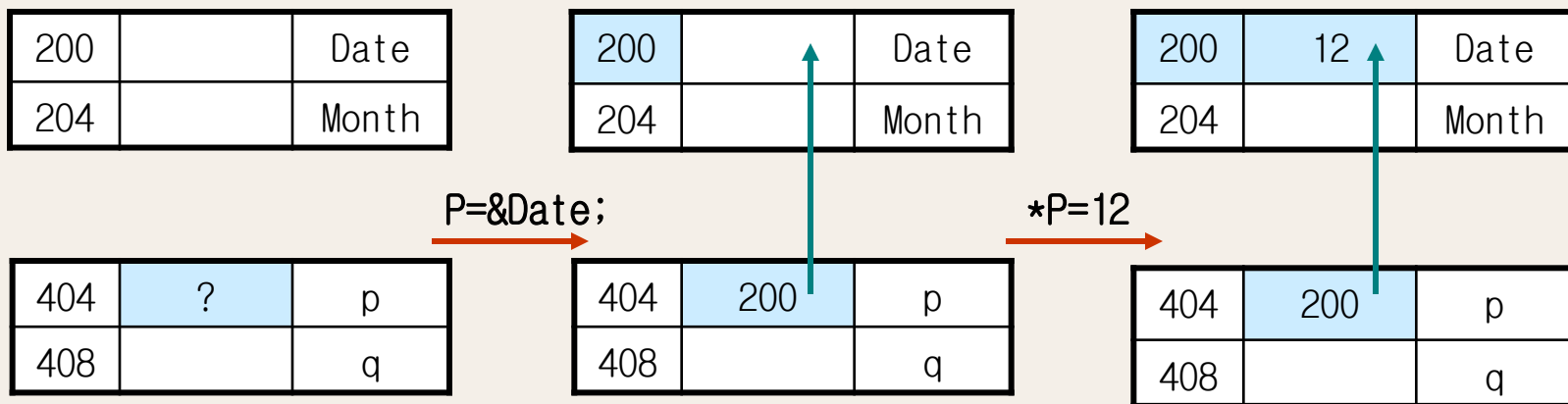


주소 연산자

```
int date, month;  
int *p;  
p = &date;
```

주소 연산자 앰퍼샌드 (Ampersand, &)

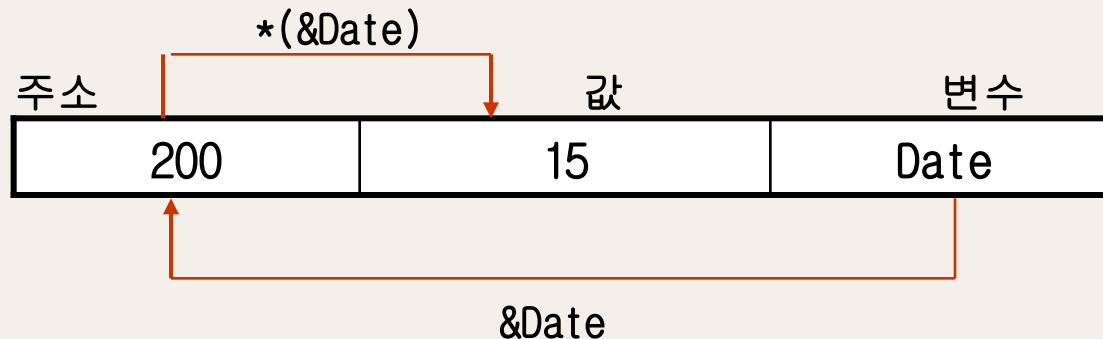
- 'Address of'
- 변수 Date의 주소 값을 포인터 변수 p에 할당
- 변수에 대한 두가지 접근방법: Date = 12; 또는 *p = 12;



Reference, Dereference

□ `*&date = 15;`

- `*(&date)`는 ‘변수 `date`의 주소 값이 가리키는 것’
- 변수 `date`의 주소 값이 가리키는 것은 변수 자체
- 변수를 주소 값으로 매핑(Mapping)시키는 함수가 앰퍼샌드(&)
- 주소 값을 그 주소가 가리키는 변수로 매핑시키는 함수가 애스터리스크(*).
- 역함수이므로 `*&date = date`와 동일한 의미



□ Lvalue, Rvalue

- `*p = 15;` `*p`는 등식의 좌변 값(Lvalue)으로 사용가능
- `Data = *p;` `*p`는 등식의 우변 값(Rvalue)으로도 사용가능

저네릭 포인터

❑ Generic Pointer

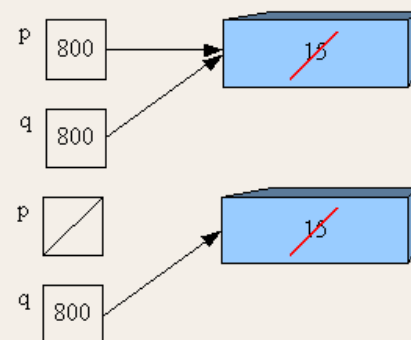
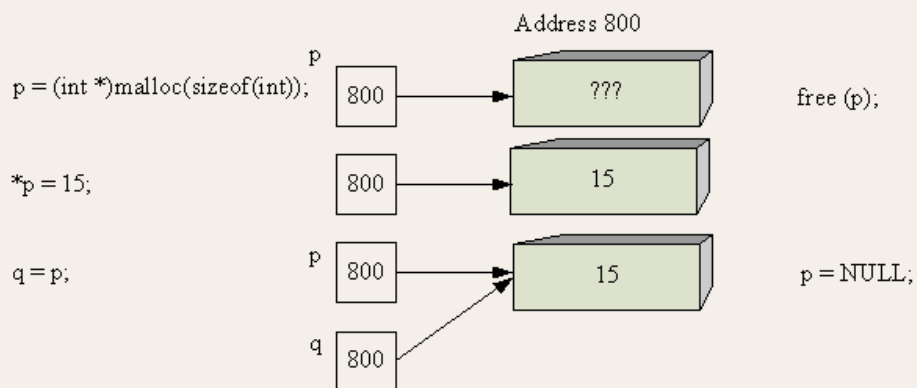
- 포인터는 그것이 가리키는 변수의 타입에 따라 분류
- 가리키는 변수에 따라 읽어와야 할 데이터의 분량이 달라짐.
- `Void *Ptr`; 선언시에는 가리키는 대상을 명시하지 않음

❑ `Ptr = (float *)malloc(sizeof(float));`

- 타입변환 연산자에 의해 실행시 포인터의 타입을 할당
- `float *Ptr`; 로 선언되었다면 `(float *)`는 없어도 된다
- 단, 커멘트 효과를 기대하기 위해 사용할 수는 있다.

Dangling Pointer

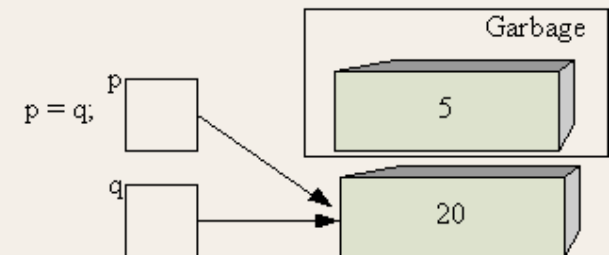
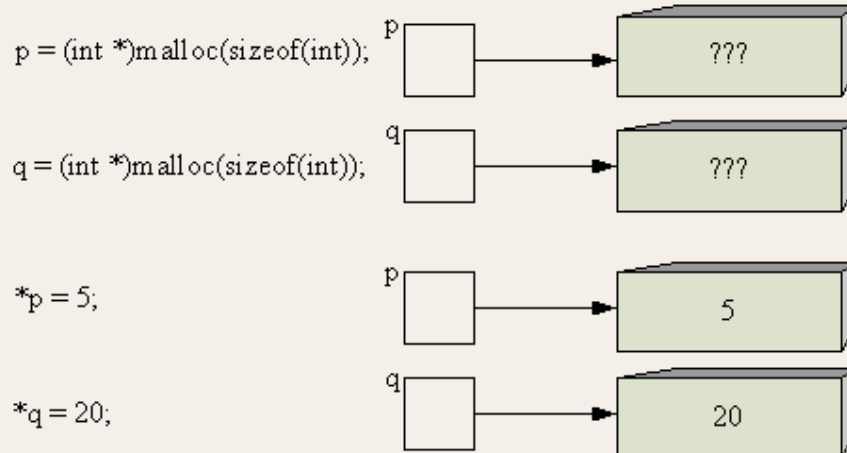
```
int *p, *q;  
p = (int *) malloc(sizeof(int));  
*p = 15;  
q = p;  
free (p);  
p = NULL;  
*q = 30;
```



Garbage

❑ 반납도 접근도 할 수 없는 공간

```
int *p; *q;  
p = (int *)malloc(sizeof(int));  
q = (int *)malloc(sizeof(int));  
*p = 5; *q = 20;  
p = q;
```



상수 변수, 상수 포인터

□ 상수변수

- `int a = 24; const int* Ptr = &a;`

□ 상수 포인터

- 포인터가 항상 일정한 메모리 주소를 가리킴
- `int* const Ptr = new int; *Ptr = 25;`
- `int a; *Ptr = &a; (이것은 오류)`

□ 배열

- 배열 이름은 배열의 시작주소 값을 지님. 즉 포인터에 해당
- 배열은 프로그램 시작 시에 메모리 위치가 고정
- 따라서 배열변수는 일종의 상수 포인터에 해당

□ 참조호출

- Call by Reference
- Call by Variable
- 원본전달 호출

□ 값 호출

- Call by Value
- 사본전달 호출

□ 실제 파라미터, 형식 파라미터

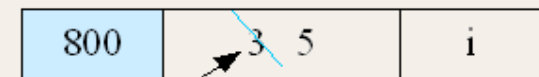
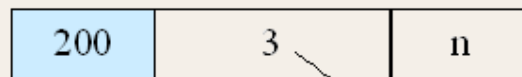
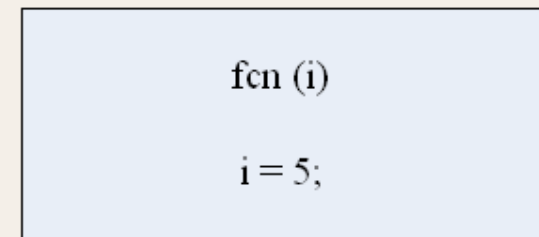
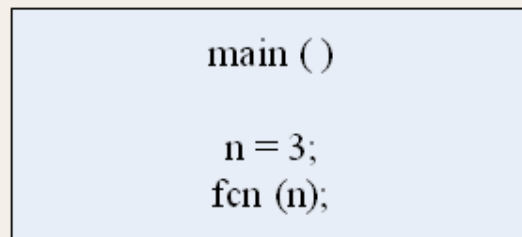
- CallMe(int a);가 있을 때, 호출 함수가 CallMe(n);으로 호출
- 호출 함수의 n = 실제 파라미터(Actual Parameter)
- 피호출 함수의 a = 형식 파라미터(Formal Parameter)
- 참조호출에서는 실제 파라미터와 형식 파라미터가 완전히 일치
- 값 호출에서는 실제 파라미터와 형식 파라미터가 서로 다른 변수

값 호출

- C/C++은 기본적으로 값 호출만을 지원

```
void fcn(int i)
{ i = 5;
}
```

```
void main( )
{ int n = 3;
  fcn(n);
  printf("%d", n);
}
```



COPY


```
int fcn(int& a)
{ int b = a;
  ...
  return b;
}
```

□ 함수실행 결과 리턴 값

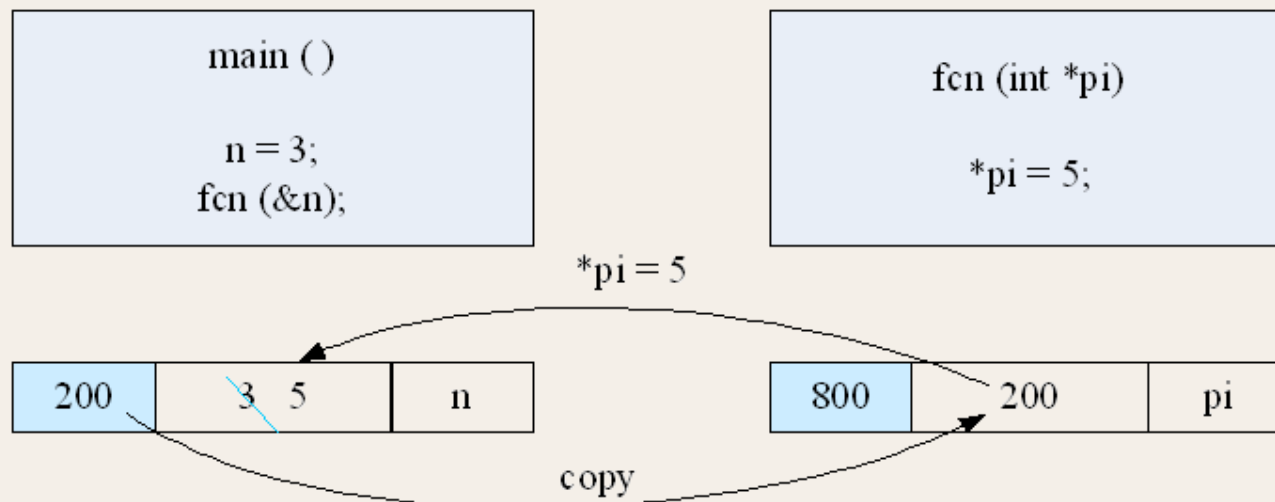
- 복사에 의해 전달 됨
- 호출함수에게 리턴 되는 것은 변수 b 가 아니고, 변수 b를 복사한 값
- 지역변수는 함수 실행 종료와 함께 그 수명이 끝남.
- 호출함수로 되돌아 온 순간에는 더 이상 존재하지 않음
- 따라서 지역변수를 리턴 할 수는 없음

참조 호출

```
void fcn(int *pi)
{
    *pi = 5;
}

void main( )
{
    int n = 3;
    fcn(&n);
    printf("%d",n);
}
```

- ❑ 포인터 변수의 값 호출에 의해 참조 호출 효과를 이룸
- ❑ C/C++은 기본적으로 참조호출을 지원하지 않음



C++ 객체의 전달

□ 값 호출

```
void fcn(bookclass b)
{  cout << b.Number;
}
```

책의 일련번호를 프린트하기

C++의 cout은 C의 printf에 해당

```
void main( )
```

```
{  bookclass MyBook;
    fcn(MyBook);
}
```

bookclass에 속하는 MyBook 객체선언

□ 참조호출 (실제로는 포인터 변수의 값 호출)

```
void fcn(bookclass *b)
{  cout << (*b).Number;
  Number를 프린트
}
```

b는 객체의 시작주소를 가리키는 포인터

b가 가리키는 객체의 상태변수 중

```
void main( )
```

```
{  bookclass MyBook;
    fcn(&MyBook);
}
```

객체 MyBook의 시작주소를 넘김

대용량 데이터: 참조호출에 의해 복사에 소요되는 시간을 줄임

에일리어스

```
void fcn(int &pi)
{ pi = 5;
}
void main( )
{ int n = 3;
  fcn(n);
  cout << n;
}
```

❑ 앰퍼샌드(&)

- Address 연산자가 아님.
- 에일리어스(Alias, Reference, 별명)을 의미
- 호출함수에서 던져준 변수 n을 ‘나로서는 일명 pi’로 부르겠다.
- 참조호출의 효과(본명이나 별명이나 동일한 대상)

❑ 왜 에일리어스를 사용하는가

- 포인터 기호 없이 간단하게 참조 호출
- 호출함수에서는 무심코 변수 자체를 전달
- 피호출 함수에서 에일리어스로 받으면 참조 호출, 그냥 받으면 값 호출

잘못 된 스왑

```
void swap (int x, int y)
{
    int temp;
    temp = x; x = y; y = temp;
}
```

```
void main( )
{
    int a = 20;
    int b = 40;
    swap(a, b);
    cout << a; cout << b;
}
```

두 가지 스왑

```
void swap (int *x, int *y)
```

```
{   int temp;
    temp = *x; *x = *y; *y = temp;
}
```

```
void main( )
```

```
{   int a = 20;
    int b = 40;
        swap(&a, &b);
    cout << a; cout << b;
}
```

변수 a, b의 주소 값 전달

```
void swap (int& x, int& y)
```

```
{   int temp;
    temp = x; x = y; y = temp;
}
```

int& x와 int &x는 동일한 기호임

```
void main( )
```

```
{   int a = 20;
    int b = 40;
    swap(a, b);
    cout << a; cout << b;
}
```

변수 a, b를 그대로 전달

C++의 복사 생성자

□ 세 가지 경우 호출됨

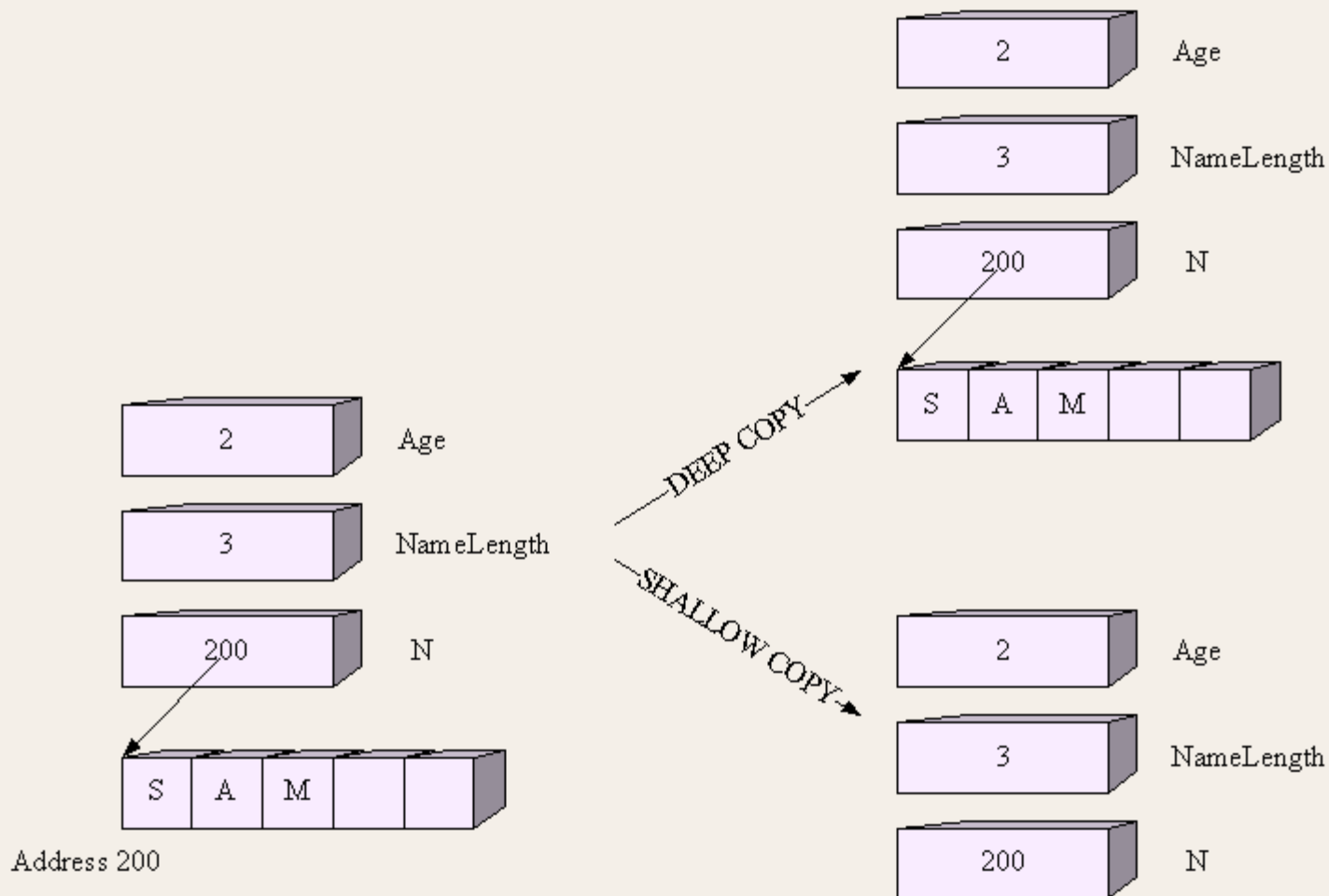
- 함수 호출시 객체가 파라미터로 전달될 때
 - `triangleClass T1; Call(T1);`
- 객체를 선언하면서 막바로 다른 객체 값으로 초기화 했을 때
 - `triangleClass T1; T1.Base = 10; T1.Height = 20;`
`triangleClass T2 = T1;`
- 피호출함수가 호출함수에게 객체를 리턴 값으로 전달할 때
 - `triangleClass T1; ... , return T1;`

□ 기본 복사 생성자

- 시스템이 자동으로 적용하는 디폴트 복사 생성자
- 프로그래머가 직접관리하는 사용자 복사 생성자를 사용하는 것이 좋음.

얕은 복사, 깊은 복사

- 사용자 복사 생성자: 깊은 복사(Deep Copy)
- 디폴트 복사 생성자: 얕은 복사(Shallow Copy)



복사 생성자와 할당 연산자

□ 복사 생성자와 할당 연산자

복사 생성자	할당 연산자
<pre>triangleClass T1; T1.Base = 10; T1.Height = 20; triangleClass T2 = T1;</pre>	<pre>triangleClass T1, T2; T1.Base = 10; T1.Height = 20; T2 = T1;</pre>

□ 직접 접근

- 인덱스 계산에 의해 해당 요소의 위치를 바로 알 수 있음
- $\&A[i] = \&A[0] + (i - 1) * \text{Sizeof}(\text{Element Type})$

A[]

A = &A[0] = 200

200	98	0
204	92	1
208	88	2
212		3
216		4

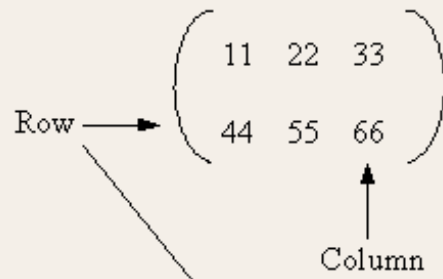
address

array element

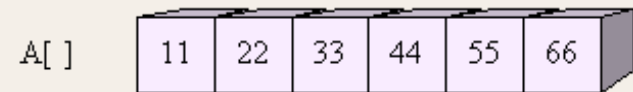
index

배열

□ 2차원 배열: 행 우선(Row-major Order)



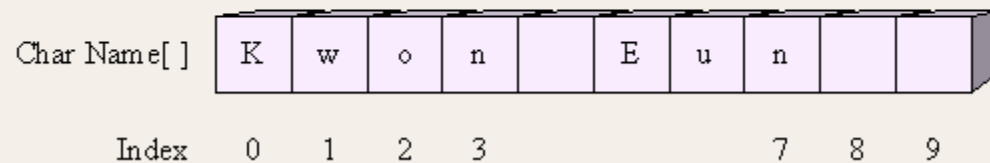
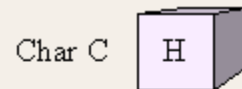
int A[2][3] = {{11, 22, 33}, {44, 55, 66}};



Index [0][0][0][1][0][2][1][0][1][1][1][2]

□ 문자열 배열

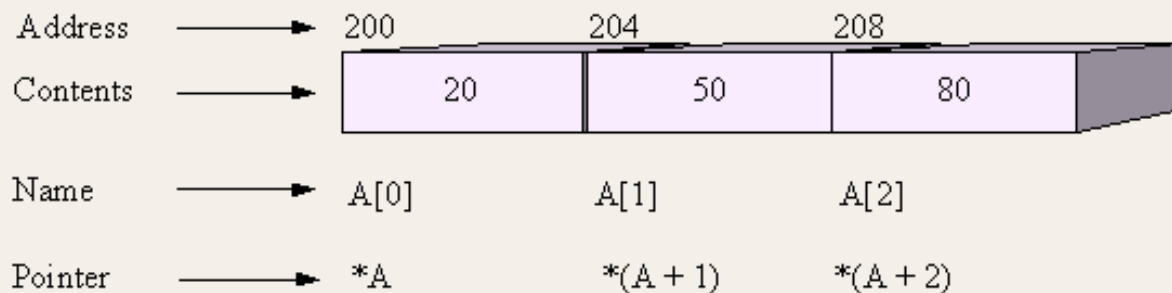
● char C = 'H'; char Name[10] = "Kwon Eun";



배열과 포인터

□ 배열의 정체는 다름아닌 포인터

- 배열 요소를 포인터로 표현할 수 있음
- 포인터 산술연산(Pointer Arithmetic)
 - $A + 1 = A + 1 * \text{Sizeof}(\text{Array Element})$



배열과 포인터

□ 배열 인덱스와 포인터 산술 연산

배열 인덱스
<pre>int Buffer[1024]; for (int i = 0; i < 1024; i++) Buffer[i] = 5;</pre>
포인터 산술연산
<pre>int Buffer[1024]; for (int* p = Buffer; p < &Buffer[1024]; p++) *p = 5;</pre>

배열의 전달

```
❑ int SumUp(int A[ ], size)
{ int Sum = 0;
  for (int i = 0; i < size; i++)
    Sum += A[i];
  return Sum;
}

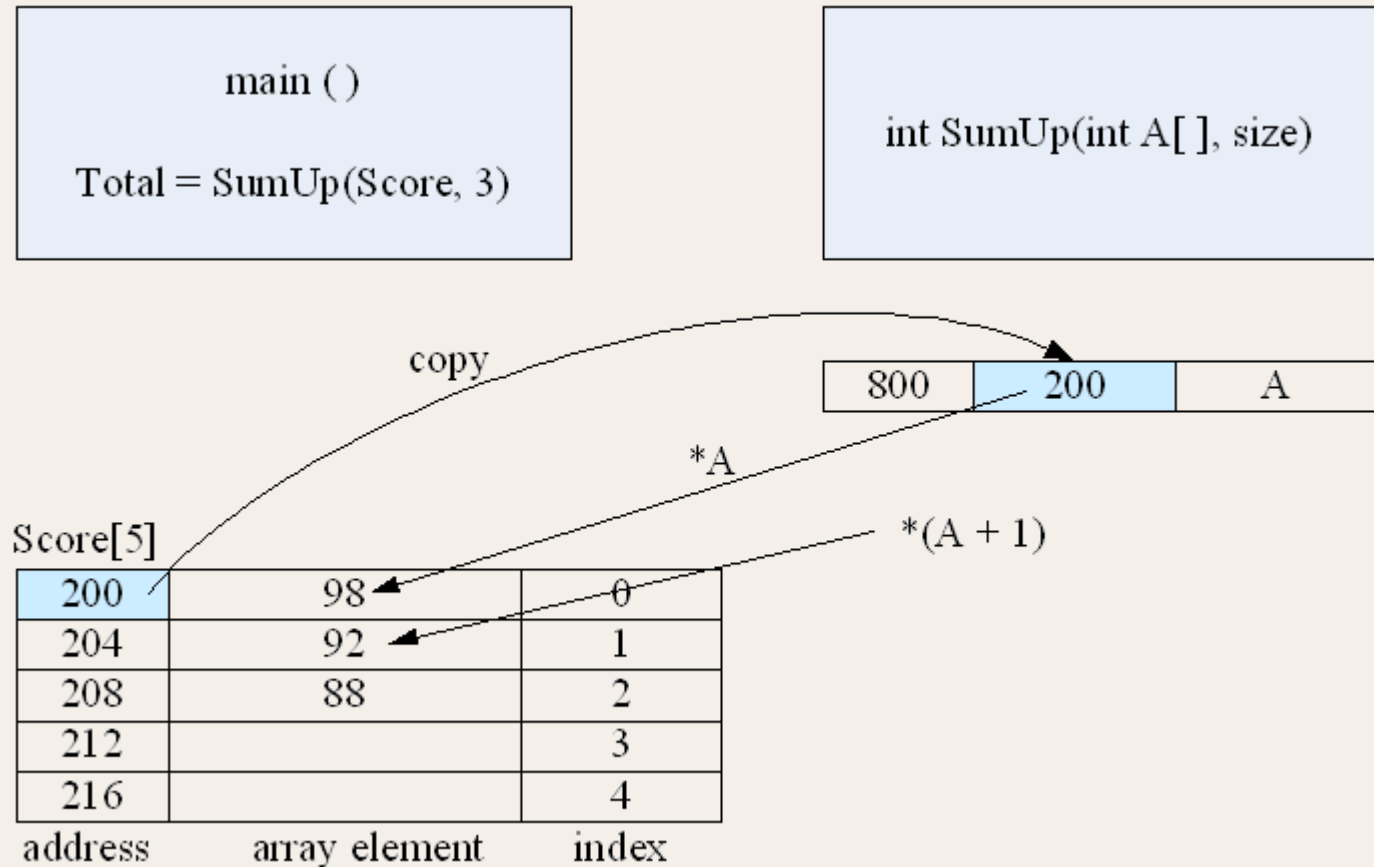
void main( )
{ int Total; int Score[5];
  Score[0] = 98; Score[1] = 92; Score[2] = 88;
  Total = SumUp(Score, 3);
}
```

❑ 배열의 전달

- 단순타입과 달리 값 호출로 배열요소가 복사되지는 않음
- 배열변수 이름, 즉 포인터 값만 값 호출로 복사되어 넘어감.
- 따라서 피 호출함수에서 작업을 가하면 이는 원본을 건드리는 참조호출 효과

배열의 전달

□ 배열의 전달



```
❑ int SumUp(int *A, size)
{   int Sum = 0;
    for (int i = 0; i < size; i++)
    {   Sum += *A;
        ++A;
    }
    return Sum;
}
```

❑ 포인터 산술연산

- $*(A+1) = A[1]$, $*(A+2) = A[2]$
- 값 호출이므로 A를 변화시켜도 무방함

❑ 상수 배열

- 읽기를 위한 접근만 허용
- `int SumUp(const int A[], size)`
- `int SumUp(const int *A, size)`

□ 정적배열과 동적배열

```
void Function1(int Size)
```

```
{ int MyArray[Size];
```

정적 배열 선언

```
...
```

```
}
```

```
void Function2(int Size)
```

```
{ int *MyArrayPtr = new int[Size]; 동적 배열 선언
```

```
...
```

```
delete[ ] MyArrayPtr;
```

```
}
```

□ 정적배열과 동적배열

- 정적 배열은 스택에(변수 선언에 의함)
- 동적배열은 힙에 (malloc, new 등 함수 실행에 의함)
- 정적 배열 크기는 컴파일 시에, 동적 배열 크기는 실행 중에 결정
- 동적 배열도 실행 중에 그 크기를 늘릴 수는 없음

필드, 레코드, 파일

□ 필드, 레코드, 파일

정지희	여	19	011-388-3031
박하영	여	19	02-445-5059
김무성	남	19	031-330-6432
정건호	남	19	019-301-3001

구조체 타입선언

❑ typedef struct

```
{ char Title[12];
```

```
  int Year;
```

```
  int Price;
```

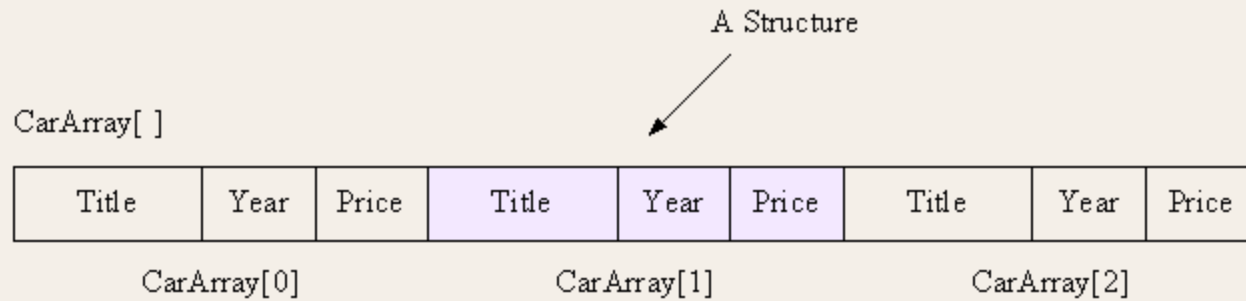
```
} carType;
```

- MyCar.Year = 2001;
- carType MyCar {"RedSportCar", 2005, 2000};

필드 이름	값	시작 주소
Title	Reds	200
	port	204
	Car	208
Year	2005	212
Price	1200	216

구조체 배열

❑ `carType CarArray[100];`



❑ `CarArray[2].Year` ?

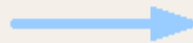
구조체 포인터

- ❑ $(*p).Year$ 와 $*p.Year$ 는 다르다.
- ❑ $(*p).Year$ 와 $P \rightarrow Year$ 는 같다.

MyCar

Title	RedS	200
	port	204
	Car	208
Year	2005	212
Price	1200	216

p		1622
---	--	------

 $p = \&MyCar$

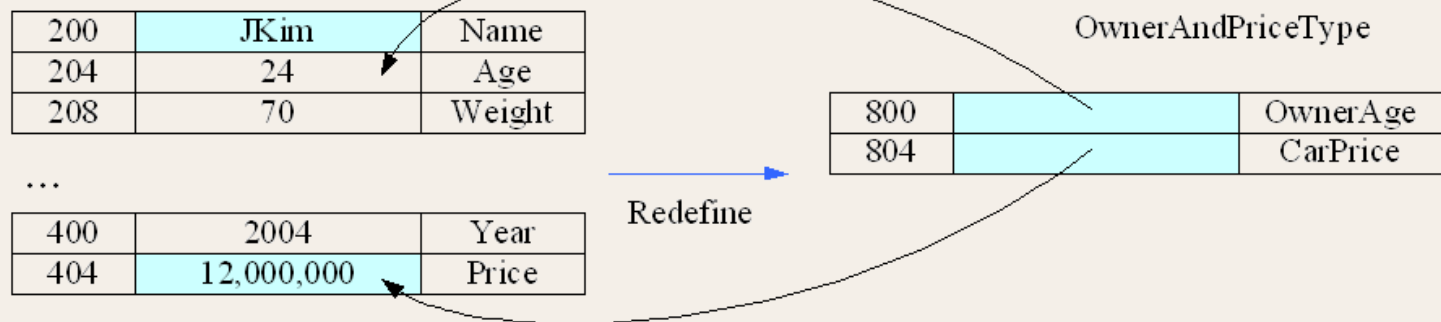
MyCar

Title	RedS	200
	port	204
	Car	208
Year	2005	212
Price	1200	216

p	200	1622
---	-----	------

포인터에 의한 개념적 재정의

```
□ typedef struct
{ int *OwnerAge;
  int *CarPrice;
} OwnerAndPriceType;
OwnerAndPriceType P;
P->OwnerAge = &Age;
P->CarPrice = &Price
```



□ 복사본을 만들지 않음

- 원본을 그대로 유지
- 원본을 가리키는 포인터만 재 구성
- 원본이 바뀌면 자동으로 내용이 바뀜

❑ FunctionCall(carType ThisCar);

```
{ ...  
}
```

```
main( )
```

```
{ carType MyCar;  
  FunctionCall(MyCar);  
}
```

❑ 필드 단위의 값 호출

- 필드가 배열이라면 배열 시작주소만 복사

❑ 참조호출

- 호출함수: FunctionCall(&MyCar);
- 피 호출함수: FunctionCall (carType *ThisCar)

❑ 함수 리턴값으로서의 구조체

- 함수 실행결과 구조체를 가리키는 포인터를 리턴
- 지역변수는 함수 실행이 끝남과 동시에 메모리 공간 소멸
- 소멸된 공간을 가리키는 포인터 를 호출함수에게 전달 -> 오류

메모리 구성

□ 메모리 구성

High Address(높은 번지)	힙(Heap) □
	미사용 공간(Available)
	스택(Stack) □
	전역변수(Global Variables)
Low Address(낮은 번지)	기계코드(Machine Code)

메인함수 호출에 따른 활성화 레코드

```
int a;           전역 변수
void main( )
{
    int b; char *s;   지역변수(스택)
    s = malloc(10);   10 바이트짜리 동적변수(힙)
    b = fcn(5);
    free (s);
}
```

힙		*S
미사용 공간		
스택 (main 함수의 활성화 레코드)	Local Variables	b, s
	Return Address	204
	Value Parameters	
	Return Value	
전역변수		a
기계코드		main() fcn1() fcn2()

Fcn 호출에 따른 활성화 레코드

```
int fcn(int p)      파라미터(스택)
{  int b = 2;
   b = b + p;
   return (b);
}
```

힙		*S
미사용 공간		
스택 (fcn 함수의 활성화 레코드)	Local Variables	b
	Return Address	5번
	Value Parameters	p: 5
	Return Value	7
스택 (main 함수의 활성화 레코드)	Local Variables	b, s
	Return Address	204
	Value Parameters	
	Return Value	
전역변수		a
기계코드		main() fcn1()

□ 함수 호출

- 컨텍스트 스위칭을 수반
- 새로운 활성화 레코드 생성
- 연속적인 함수 호출에 따라 활성화 레코드 스택이 생성
- 함수 종료시 제일 위의 활성화 레코드가 사라짐으로써 직전의 상황을 복원

□ 무한 루프

- 스택 오버플로우

❑ 디버깅시

```
#include <assert.h>
float SquareRoot (float t)
{
    assert (t >= 0);
    return sqrt(t);
}
```

❑ 디버깅이 끝난 후

```
#define NDEBUG
#include <assert.h>
float SquareRoot (float t)
{
    assert (t >= 0);
    return sqrt(t);
}
```


C/C++ 명령 비교

□ C/C++ 명령 비교

	C	C++
동적변수 할당	<code>p = (int *)malloc(sizeof(int));</code>	<code>p = new int;</code>
동적변수 해제	<code>free p;</code>	<code>delete p;</code>
화면출력 명령	<code>printf("%d\n", x);</code>	<code>cout << x << endl;</code>