Chapter 10. 포인터와동적배열

박 종 혁 교수
UCS Lab
(http://www.parkjonghyuk.net)

Tel: 970-6702
Email: jhpark1@snut.ac.kr

Learning Objectives

- 포인터
  - 포인터 변수
  - 메모리 관리
- 동적 배열
  - 생성과 사용
  - 포인터 연산
- 클래스, 포인터, 동적 배열
  - this 포인터
  - 소멸자, 복사 생성자

포인터 개요

- 포인터 정의:
  - 변수의 메모리 주소
- 메모리 분할
  - 넘버링 된 메모리 위치
  - 변수를 위해 사용된 메모리 주소
- 이미 사용!
  - Call-by-reference 매개변수
    - 실 인자의 주소가 넘겨짐

포인터 변수

- 포인터는 자료형
  - 포인터는 변수에 저장
  - Int나 double형 변수에 저장할 수 없음
    - 포인터 형을 선언하고 저장해야 함!
- 예: **각 데이터 유형을 가리키기 위해서 그에 대응하는 포인터 형이 필요하다!**

```
double *p;
```

  - p는 double 형을 가리키는 포인터 변수
  - double 형 변수의 포인터를 저장
    - 다른 형이 아님 → 포인터 형!!

포인터 변수 선언

- 다른 형과 동일하게 선언
  - 변수 명 전에 "*" 추가
- "*" 는 각각의 변수 전에 추가
- int *p1, *p2, v1, v2;
  - p1, p2는 정수형 포인터 변수
  - v1, v2는 정수형 변수

주소와 숫자

- 포인터는 주소 → 주소는 정수
- 하지만, 포인터는 정수가 아님!
  - 괴변이 아님 → 추상화(abstraction)!
  - C++에서 포인터는 주소로 사용됨
    - 주소가 정수이기는 하지만, 정수형 변수로 사용될 수 없음

포인팅 (1/3)

- 용어
  - 포인팅, 주소가 아님
  - 일반 변수를 지적하는 포인터 변수
  - 주소의 개념을 배제
- 개념적 접근
  - 참조하는 메모리를 지적한다, 가리킨다
  - “→”

포인팅 (2/3)

- int *p1, *p2, v1, v2;
  p1 = &v1;
  - 포인터 변수 p1은 변수 v1을 가리킨다.
- & 연산자
  - 변수의 주소를 리턴
- 이해:
  - p1은 v1의 주소와 같다
  - p1은 v1을 가리킨다.

포인팅 (3/3)

- int *p1, *p2, v1, v2;
  p1 = &v1;
- v1을 참조하는 두 가지 방법:
  - v1 자신:
    cout << v1;
  - p1 이용:
    cout << *p1;
- 역참조(dereference) 연산자, *
  - 의미: p1이 지적하는 데이터를 얻어온다

포인팅 예제

- 예:

```
v1 = 0;
p1 = &v1;
*p1 = 42;
cout << v1 << endl;
cout << *p1 << endl;
```

- 결과:

```
42
42
```

- p1과 v1의 참조 결과는 같음

& 연산자

- 주소 연산자
- call-by-reference 매개변수 명시에 사용
  - 우연이 아님!
  - call-by-reference 매개변수는 실 인자의 주소가 전달됨
- 주소 연산과 call-by-reference 매개변수 명시의 두 가지 사용법은 밀접한 연관성이 있음

포인터 할당

- 포인터 변수도 할당이 가능:

  int *p1, *p2;

  p2 = p1;

  – p1이 지적하는 곳을 p2도 지적하도록 함
- 다음과 혼돈하지 말 것:

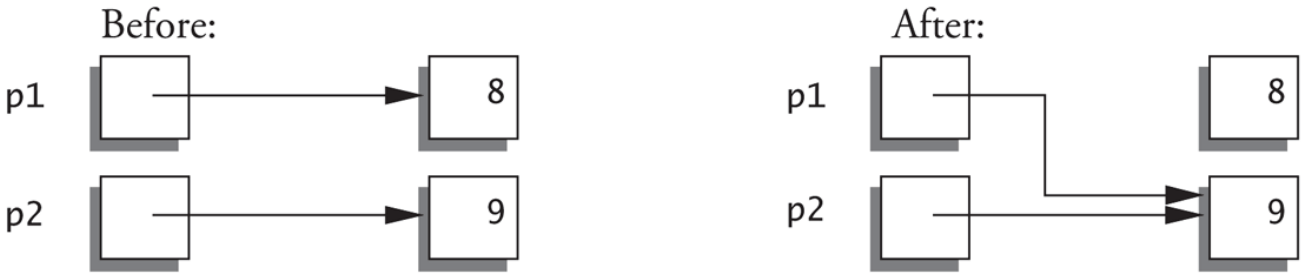
  *p1 = *p2;

  – p2가 지적하는 변수의 값을 p1이 지적하는 변수에 할당

디스플레이 10.1 포인터 변수와 할당문의 사용

Display 10.1 Uses of the Assignment Operator with Pointer Variables

```
p1 = p2;
```




```
*p1 = *p2;
```

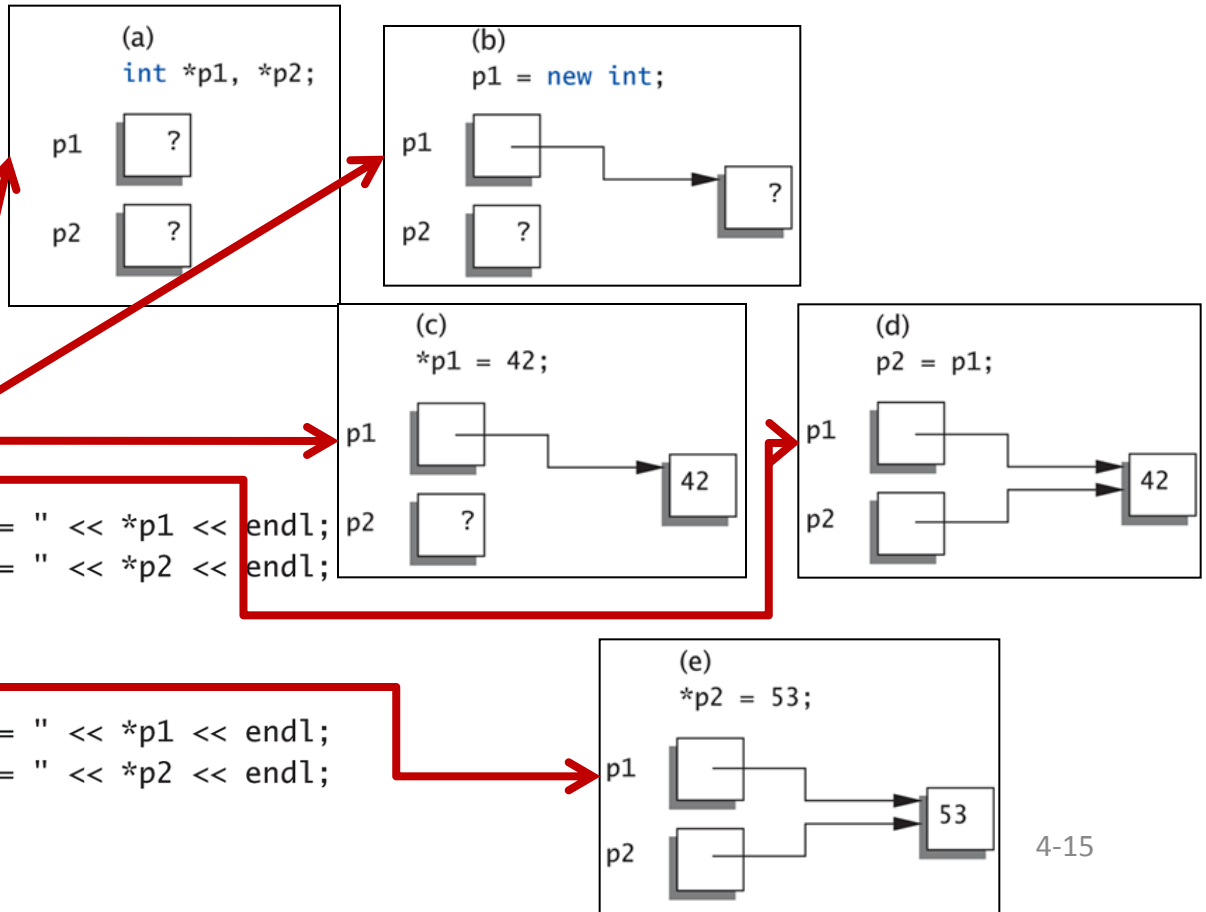
new 연산자

- 포인터가 변수를 참조할 수 있기 때문에…
  - 변수의 이름 없이 변수의 조작이 가능
- 변수의 동적 할당이 가능
  - new 연산자는 변수를 생성
    - 변수의 이름이 필요 없음
    - 포인터로 조작 가능!
- p1 = new int;
  - 이름 없는 변수를 생성하고 포인터 p1이 지적하게 함
  - *p1로 접근 가능
    - 일반 변수와 동일하게 사용 가능

디스플레이 10.2 기본 포인터 연산의 예 (1/2)

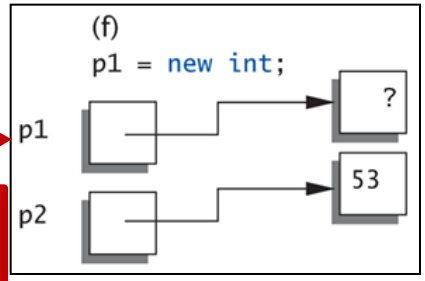
Display 10.2 Basic Pointer Manipulations

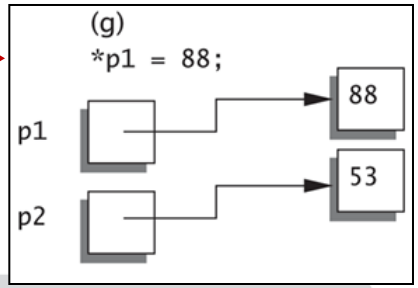
```
1   //Program to demonstrate pointers and dynamic variables.
2   #include <iostream>
3   using std::cout;
4   using std::endl;

5   int main( )
6   {
7       int *p1, *p2;

8       p1 = new int;
9       *p1 = 42;
10      p2 = p1;
11      cout << "*p1 == " << *p1 << endl;
12      cout << "*p2 == " << *p2 << endl;

13      *p2 = 53;
14      cout << "*p1 == " << *p1 << endl;
15      cout << "*p2 == " << *p2 << endl;
```



디스플레이 10.2 기본 포인터 연산의 예 (2/2)

```
16        p1 = new int;
17        *p1 = 88;
18        cout << "*p1 == " << *p1 << endl;
19        cout << "*p2 == " << *p2 << endl;


20        cout << "Hope you got the point of this example!\n";
21        return 0;
22    }
```





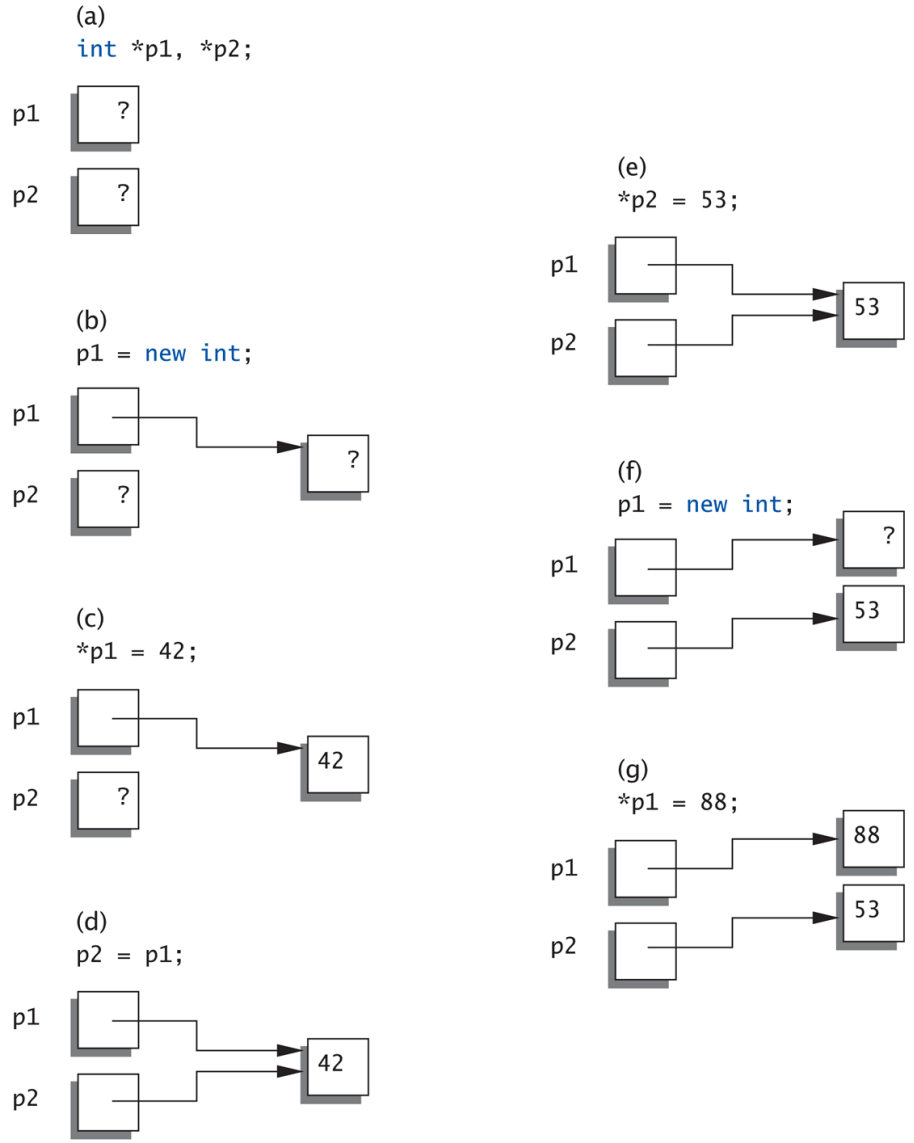
SAMPLE DIALOGUE

```
*p1 == 42
*p2 == 42
*p1 == 53
*p2 == 53
*p1 == 88
*p2 == 53
Hope you got the point of this example!
```

디스플레이 10.3

10.2 설명

Display 10.3 Explanation of Display 10.2



new 연산자 - 동적 변수 생성

- new 연산자는 동적 변수 생성하고, 그 변수를 가리키는 포인터를 반환
- 클래스 형:
  - 새로운 객체 생성을 위해 생성자 호출
  - 초기화를 위해 다른 생성자의 호출이 가능:

```
MyClass *mcPtr;
mcPtr = new MyClass(32.0, 17);
```

- 기본 자료 형 초기화:

```
int *n;
n = new int(17); //Initializes *n to 17
```

포인터와 함수

- 포인터는 모든 것이 가능한 자료형
  - 다른 형과 동일하게 사용 가능
- 함수의 매개변수로 사용할 수 있음
- 함수의 리턴형으로도 사용 가능
- 예:

```
int* findOtherPointer(int* p);
```

메모리 관리

- 힙 (Heap)
  - 자유저장공간 (freestore)
  - 동적으로 할당되는 변수를 위해 예약된 메모리 영역
  - 모든 동적 변수는 자유저장공간을 사용 → 유한함
- 자유저장공간이 모두 사용되면 new 오퍼레이션은 실패함

동적 변수 생성 체크

- 구 컴파일러:

```
int *p;
p = new int;
if (p == NULL)
{
    cout << "Error: Insufficient memory.\n";
    exit(1);
}
```

  - new 오퍼레이션이 성공해야 프로그램이 계속 수행됨

동적 변수 생성 체크 – 새로운 컴파일러

- 신 컴파일러:
  - new 오퍼레이션이 실패하면:
    - 프로그램은 자동 종료
    - 에러 메시지 출력
- 아직도 null 체크는 좋은 프로그래밍 습관

자유저장공간 크기

- 컴파일러의 구현에 따라 다름
- 일반적으로 큼
  - 대부분의 프로그램이 모든 영역을 사용하지 못함
- 메모리 관리
  - 좋은 습관
  - 개발자의 중요한 규칙
  - 메모리는 유한함

delete 연산자

- 동적 메모리를 반환
  - 동적 변수가 사용하던 공간이 더 이상 필요 없을 때, 자유저장공간에 메모리를 반환
  - 예:

```
int *p;
p = new int(5);
… //Some processing…
delete p;
```

  - 포인터 p가 지적하고 있던 메모리를 반환

허상 포인터 (Dangling Pointer)

- delete p;
  - 동적 메모리를 반환
  - 하지만 포인터 p는 그곳을 지적하고 있음!
    - 허상 포인터(dangling pointer)라 지칭
  - *p 를 사용
    - 예기치 못한 결과 → 재앙!
- 허상 포인터의 회피
  - delete 이후에 포인터 변수에 null 할당:

```
delete p;
p = NULL;
```

동적 변수와 자동 변수

- 동적 변수(Dynamic variable)
  - new에 의해 생성
  - 프로그램 실행 중 생성되고 반환됨
- 지역 변수(Local variable)
  - 함수 내에 정의된 변수
  - 동적이 아님
    - 함수 호출 시 생성
    - 함수 호출이 완료되면 회수됨
  - 자동 변수라 지칭

포인터 형 정의

- 포인터 형의 명명이 가능
- typedef int* IntPtr;
  - 새로운 형의 별칭

```
IntPtr p;
int *p;
```

    - 두 가지 선언은 동일

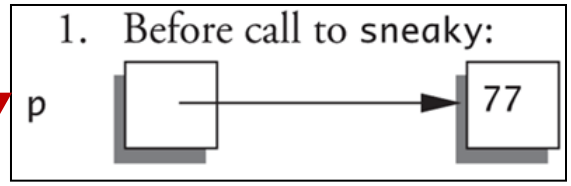
함정: Call-by-value 포인터

- 동작 과정이 미묘하고 까다로움
  - 만약 함수가 포인터 매개변수 자체를 변경 → 지역의 복사본 포인터의 값만 변화
  - 하지만, 포인터가 지적하는 변수의 값은???
- 디스플레이 10.4와 10.5

디스플레이 10.4 Call-by-Value 포인터 매개변수 (1/2)

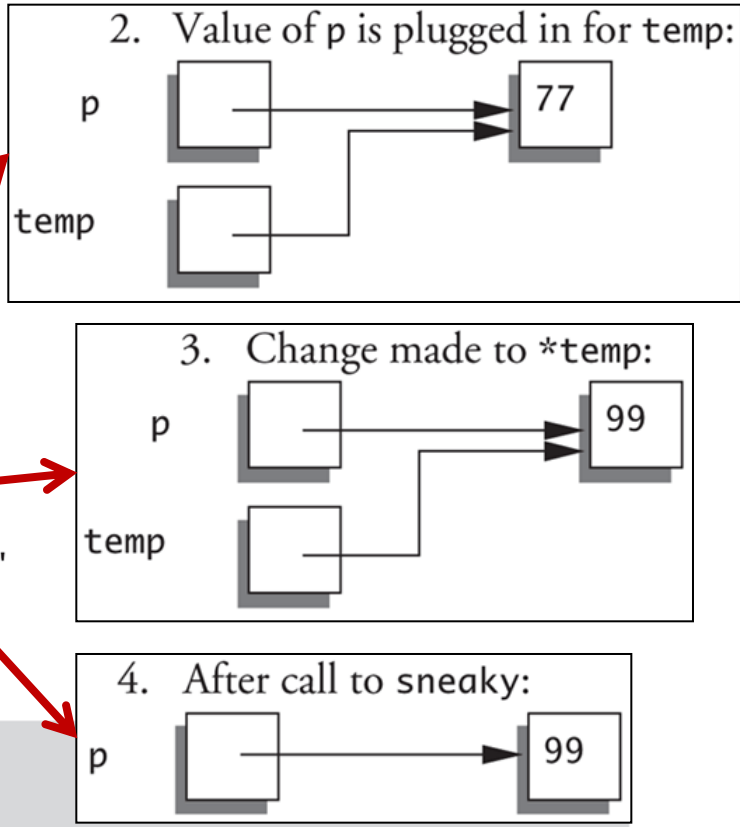
Display 10.4 A Call-by-Value Pointer Parameter

```
1   //Program to demonstrate the way call-by-value parameters
2   //behave with pointer arguments.
3   #include <iostream>
4   using std::cout;
5   using std::cin;
6   using std::endl;

7   typedef int* IntPointer;

8   void sneaky(IntPointer temp);

9   int main( )
10  {
11      IntPointer p;

12      p = new int;
13      *p = 77;
14      cout << "Before call to function *p == "
15           << *p << endl;
```



디스플레이 10.4 Call-by-Value 포인터 매개변수 (2/2)

```
16        sneaky(p);

17        cout << "After call to function *p == "
18             << *p << endl;

19        return 0;
20   }
21   void sneaky(IntPointer temp)
22   {
23        *temp = 99;
24        cout << "Inside function call *temp == "
25             << *temp << endl;
26   }
```



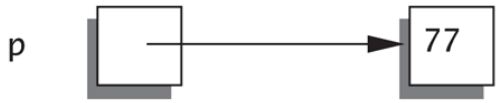
SAMPLE DIALOGUE

```
Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99
```

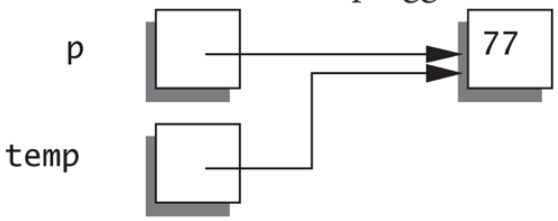
디스플레이 10.5 함수 호출 sneaky(p);

Display 10.5 The Function Call `sneaky(p);`

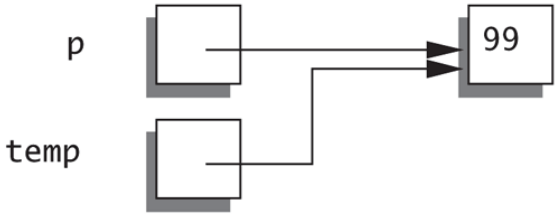
1. Before call to `sneaky`:



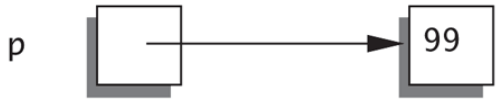
2. Value of `p` is plugged in for `temp`:



3. Change made to `*temp`:



4. After call to `sneaky`:



동적 배열

- 배열 변수
  - 실제 포인터 변수!
- 일반적인 배열
  - 고정된 크기
- 동적 배열
  - 크기가 프로그램 런타임에 결정

배열 변수

- 배열은 메모리에 연속적인 주소로 저장
  - 배열 변수는 첫 번째 인덱스 변수를 참조
  - 따라서 배열 변수는 포인터 변수의 일종!
- Example:
  int a[10];
  int * p;
  - a와 p 모두 포인터 변수!

배열 변수 → 포인터 (1/2)

- int a[10];
  typedef int* IntPtr;
  IntPtr p;
- a와 p는 포인터 변수 are pointer variables
  - 할당 가능:
    p = a; // Legal.
    - p는 a가 지적하는 곳을 지적 now points where a points
      - 배열 a의 첫 번째 인덱스 변수를 지적
  - a = p; // ILLEGAL!
    - 배열 포인터는 상수 포인터!

배열 변수 → 포인터 (2/2)

- 배열 변수
  int a[10];
- a 포인터 변수
  - "int * const" type
  - 배열은 메모리에 할당됨
  - 변수 a는 항상 배열을 지적해야 함!
    - 변경될 수 없음!
- 일반 포인터는 지적하는 곳의 변경이 가능

동적 배열

- 배열의 제약
  - 크기를 미리 지정해야 함
- 최대 크기의 예측이 필요
  - 작은 배열 크기 : 공간 부족 발생 → 재앙
  - 큰 배열 크기 : 메모리의 낭비
- 동적 배열
  - 필요 시 크기의 변경이 가능

동적 배열 생성

- 매우 간단하다!
- new 연산자를 사용
  - 포인터 변수와 함께 생성
  - 일반 변수와 동일하게 취급됨
- 예:

```
typedef double * DoublePtr;
DoublePtr d;
d = new double[10];    //Size in brackets
```

동적 배열 삭제

- 런타임 시에 동적으로 할당
  - 따라서, 런타임에 회수가 가능
- 예:

```
d = new double[10];
… //Processing
delete [ ] d;
```

  - 동적 배열에 할당된 공간을 회수
  - [ ]는 그곳에 있는 배열을 의미
  - *d는 여전히 지적하고 있음*!
    - d = NULL;

배열을 리턴하는 함수

- 배열은 함수의 리턴 형으로 지정될 수 없음
- 예:
  int [ ] someFunction(); // ILLEGAL!
- 배열의 기본 형 포인터의 리턴이 가능:
  int* someFunction(); // LEGAL!

포인터 연산

- 포인터의 연산
  - 주소 연산
- 예:

```
typedef double* DoublePtr;
DoublePtr d;
d = new double[10];
```

  - d는 d[0] 의 주소 값을 가짐
  - d + 1은 d[1]의 주소
  - d + 2는 d[2]의 주소

선택적 배열 조작

- 포인터 연산을 이용!
- 인덱싱을 사용하지 않음:

```
for (int i = 0; i < arraySize; i++)
        cout << *(d + I) << " " ;
```

- 위와 동일:

```
for (int i = 0; i < arraySize; i++)
        cout << d[I] << " " ;
```

- +/- 연산만이 가능
  - *, / 허용되지 않음
- ++나 --도 가능

다차원 동적 배열

- 배열의 배열
- typedef int* IntArrayPtr;
  IntArrayPtr *m = new IntArrayPtr[3];
  - 세 개의 포인터 변수 배열
  - 각각의 포인터에 4개의 정수형 배열 할당
- for (int i = 0; i < 3; i++)
  m[i] = new int[4];
  - 3*4 동적 배열!

클래스와 포인터

- -> 연산자
- *와 dot(.) 연산자를 함께 사용
- 특정 멤버의 지적이 가능 (구조체와 동일)
- 예:

```
MyClass *p;
p = new MyClass;
p->grade = "A";   Equivalent to:
(*p).grade = "A";
```

this 포인터

- 멤버 함수의 정의 시, 호출 객체의 참조가 필요
- *this* 포인터 사용
  - 자동으로 호출 객체를 지적:

```
Class Simple
{
public:
   void showStuff() const;
private:
   int stuff;
};
```

- 멤버 함수에서의 멤버 변수 접근:

```
cout << stuff;
cout << this->stuff;
```

- 호출 객체의 리턴:

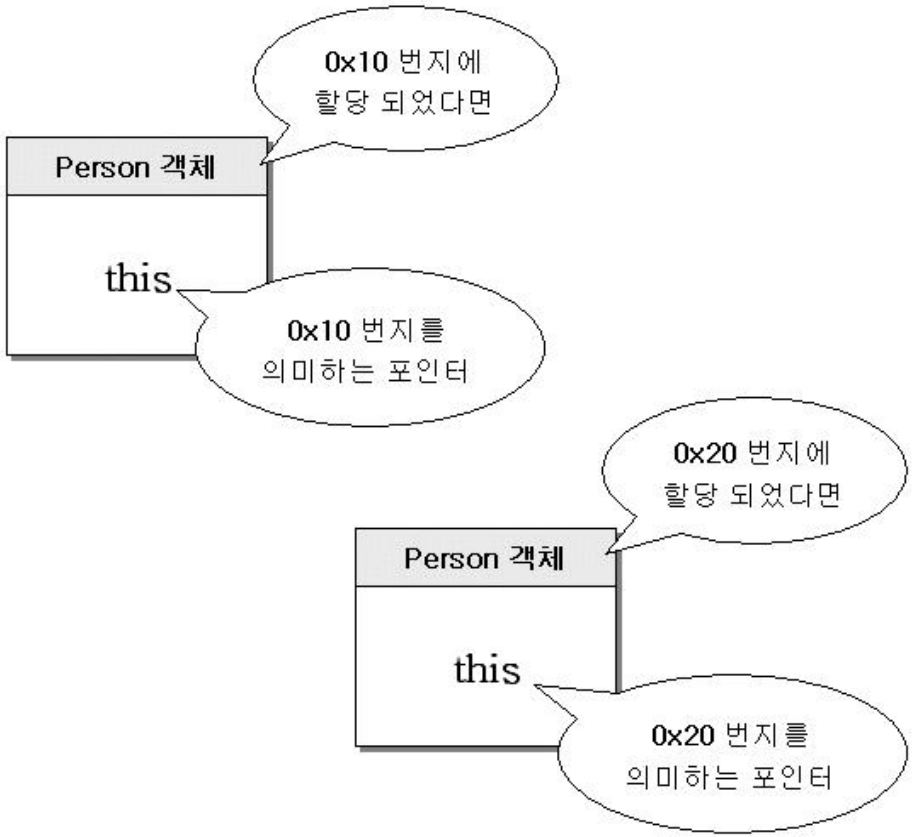
```
return *this
```

자기참조(self-reference)

- 자기참조
  - 클래스의 멤버함수는 자신을 호출한 객체를 가리키는 포인터를 명시
  - this라는 키워드가 자기 자신 객체의 포인터를 가리킴

자기참조(self-reference)

- this 포인터의 의미
  - this_ur

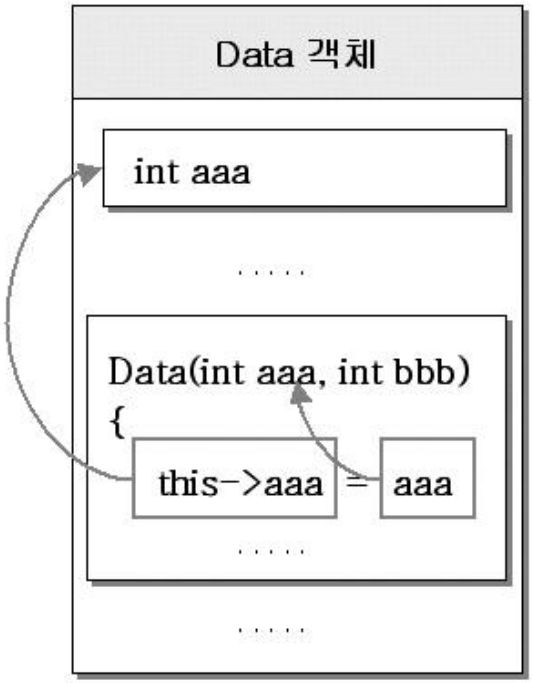


자기참조 예 (1)

```
#include <iostream>
using std::cout;
using std::endl;

class Person {
public:
	Person* GetThis(){
		return this; //this 포인터를 리턴.
	}
};

int main()
{
	cout<<"***** p1의 정보 ******"<<endl;
	Person *p1 = new Person();
	cout<<"포인터 p1: "<<p1<<endl;
	cout<<"p1의 this: "<<p1->GetThis()<<endl;

	cout<<"***** p2의 정보 ******"<<endl;
	Person *p2 = new Person();
	cout<<"포인터 p2: "<<p2<<endl;
	cout<<"p2의 this: "<<p2->GetThis()<<endl;

	return 0;
}
```

자기참조 예 (2)

```
#include <iostream>
using std::cout; using std::endl;
class Data {
	int aaa;
	int bbb;
public :
	Data(int aaa, int bbb) {
		//aaa=aaa;
		this->aaa=aaa;
		//bbb=bbb;
		this->bbb=bbb;
	}
	void printAll() {	cout<<aaa<<" "<<bbb<<endl;
	}
};
int main(void){
	Data d(100, 200);
	d.printAll();
	return 0;
}
```



자기참조 예 (3)

```
#include <iostream>
using std::cout;
using std::endl;

class Dog {
        Dog   *body;
        char *name;
        Dog   *tail;
public:
        Dog(char *s);
        char *dog_name();
        void make_tail(char *s);
        char *tail_name();
};

Dog::Dog(char *s)
{
   name = new char[strlen(s) + 1];
   strcpy(name, s);
   tail = body = 0;
}
char *Dog::dog_name()
{   return name;  }
```

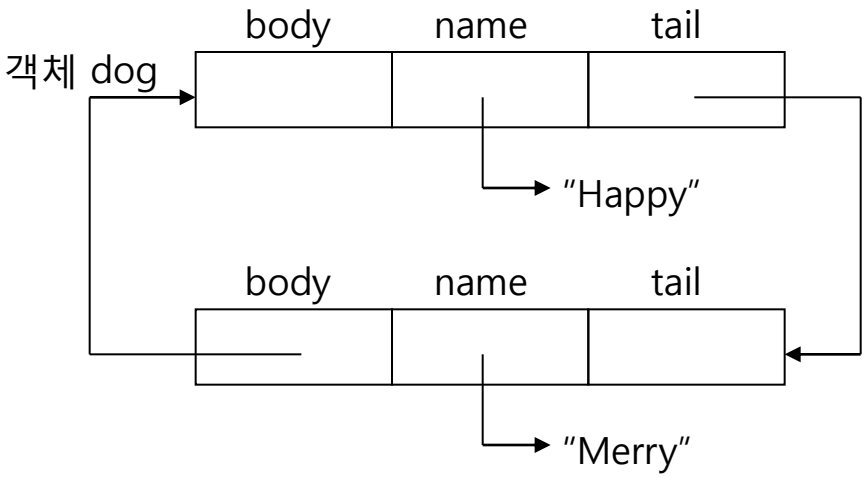
```
void Dog::make_tail(char *s)
{
   tail = new Dog(s);   // tail 객체 생성
   tail->body = this;
                        // 자기참조 포인터 지정
}

char *Dog::tail_name()
{
   return tail->dog_name();
}

int main()
{
   Dog  dog("Happy");

   dog.make_tail("Merry");
   cout << "dog name  : " << dog.dog_name()
      << endl;
   cout << "tail name : " << dog.tail_name()
      << endl;

   return 0;
}
```

자기참조 예 (3)

dog name : Happy

tail name : Merry

전용부분에서 같은 클래스의 포인터로서 body, tail을 선언하였다. make_tail() 함수에서 새로 생성한 객체 tail의 body가 호출한 객체를 가리키도록 자기참조 포인터 this를 지정하였다.



할당 연산자 오버로딩 (1/2)

- 할당 연산자는 참조를 리턴
  - 따라서, 연속적인 중복 할당이 가능
  - 예) a = b = c;
- 연산자는 연산자의 좌측과 동일한 형을 리턴 해야 함
  - 연속적인 할당
  - *this* 포인터 사용!

할당 연산자 오버로딩 (2/2)

- 할당 연산자는 클래스의 멤버로 오버로딩 되어야 함
  - 하나의 매개변수를 가짐
  - 좌측 피연산자는 호출 객체

    s1 = s2;
    - s1.=(s2);
- s1 = s2 = s3;
  - s1 = (s2 = s3);
  - 리턴된 객체는 다음 연산자의 인자로 넘겨짐

= 연산자 오버로딩 정의 (1/3)

- 스트링 클래스 예:

```
class StringClass
{
 public:
        …
        void someProcessing();
        …
        StringClass& operator=(const StringClass& rtSide);
        …
 private:
        char *a; // 문자열을 위한 동적 배열
        int capacity; // 동적 배열 a의 크기
        int length; // a에 포함된 문자의 수
};
```

= 연산자 오버로딩 정의 (2/3)

- 스트링 클래스 예(오류):

```
StringClass& StringClass::operator=(const StringClass& rtSide)
{
        capacity = rtSide.length;
        length = rtSide.length;
        delete [] a;
        a = new char[capacity];
        for (int I = 0; I < length; I++)
                a[I] = rtSide.a[I];

        return *this;
}
```

```
s = s;
delete [] a;
delete [] s.a;
```

= 연산자 오버로딩 정의 (3/3)

- 스트링 클래스 예(버그수정):

```
StringClass& StringClass::operator=(const StringClass& rtSide)
{
        if (this == &rtSide)            // if right side same as left side
                return *this;
        else
        {
                capacity = rtSide.length;
                length
                length = rtSide.length;
                delete [] a;
                a = new char[capacity];
                for (int I = 0; I < length; I++)
                        a[I] = rtSide.a[I];
                return *this;
        }
}
```

얕은 복사(Shallow Copy)와 깊은 복사(Deep Copy) → 다시 작성

- 얕은 복사 (Shallow copy)
  - 멤버 변수간의 할당 시
  - 디폴트 할당과 복사 연산자
- 깊은 복사 (Deep copy)
  - 포인터, 동적 메모리와 연관될 때
  - 동적으로 할당된 변수들의 참조를 가져 오는 것이 아니고 새로운 복사본을 생성해주어야 함
  - 이 경우 사용자 정의 복사 생성자를 정의해 주어야 함!

소멸자의 필요성

- 동적 할당 변수
  - delete를 실행하기 전까지 소멸되지 않음
- 포인터가 private 멤버 데이터 이면
  - 생성자에서 동적으로 할당
  - 객체가 소멸될 때 할당된 메모리를 회수 해야 함
- 정답: 소멸자!

소멸자

- 생성자와 반대
  - 객체가 소멸될 때 자동으로 호출
  - 디폴트 버전은 일반적인 변수만 제거, 동적 변수는 제거할 수 없음
- 생성자의 정의와 유사, ~ 가 추가됨
  - MyClass::~MyClass()
    {
      //Perform delete clean-up duties
    }

복사 생성자 (Copy Constructor)

- 자동 호출의 경우:
  1. 다른 객체에서 클래스가 선언되고 초기화된 경우
  2. 함수가 클래스 형의 객체를 리턴 할 때
  3. 함수 인자가 객체이며, call-by-value로 전달되는 경우
- 객체의 임시 복사본이 필요
  - 복사 생성자가 생성시킴
- 디폴트 복사 생성자
  - "=", member-wise copy
- 포인터 → 사용자 복사 생성자의 정의가 필요!

요약 1

- 포인터는 메모리 주소
  - 변수의 간접 참조를 제공
- 동적 변수
  - 프로그램 실행 중 생성과 반환
- 자유저장공간 (Freestore)
  - 동적 변수를 위한 메모리 공간
- 동적 할당 배열
  - 프로그램 실행 중 크기가 결정

요약 2

- 클래스 소멸자
  - 특별한 멤버 함수
  - 자동으로 객체를 소멸 시킴
- 복사 생성자
  - 하나의 매개변수를 가지는 멤버 함수
  - 일시적인 복사가 필요할 때 자동으로 호출
- 할당 연산자
  - 멤버 함수로 구현 되어야 함
  - 중복 할당을 위해 참조를 리턴

Q&A