제10장

포인터와 동적 배열

객체 배우기

- 포인터
  - 포인터 변수
  - 메모리 관리
- 동적 배열
  - 생성과 사용
  - 포인터의 산술연산
- 클래스, 포인터, 동적 배열
  - *This* 포인터
  - 소멸자, 복사 생성자

포인터 소개

- 포인터는 변수의 메모리 주소!

- 복습: 분할된 메모리
  - 숫자로 표현되는 메모리 장소

- 변수를 위한 이름으로 사용되는 주소
- 여러분은 이미 포인터를 사용한 적이 있다!
  - Call-by-reference 매개변수
    - 인자의 실제 주소가 전달됨

예제

```
int val1 = 8, val2 = 3 ;  // 정수형 변수
int *pInt1, *pInt2 ; // 정수형 포인터

double x1 = 1.7, *pDouble = &x1 ; // 실수형
char c1 = 'A', *pChar = &c1; // 문자형
```

```
pInt1 = &val1 ;
cout << &val1 << " " << val1 << endl ;
cout << pInt1 << " " << *pInt1 << " " << endl ;

pInt1 = &val2 ; pInt2 = pInt1 ;
cout << pInt1 << " " << *pInt1 << " " << endl ;
cout << pInt2 << " " << *pInt2 << " " << endl ;

*pInt1 = 77 ;
cout << pInt2 << " " << *pInt2 << " " << endl ;

cout << (void *)pChar << " " << *pChar << " " << endl ;

cout << sizeof(pChar) << " " << sizeof(pInt1) << " " << sizeof(pDouble) << endl ;
```

```
int *pIntArr = new int[20] ;
……
delete [] pIntArr ;

Money *pMArr = new Money[10] ;
……
delete [] pMArr ;
```

포인터 변수

- 자료형으로 포인터
  - 포인터 "형"
  - 포인터는 변수에 저장될 수 있다.
    - int, double 등의 변수는 아니다.
    - 대신 int, double 등을 가리키는 포인터 변수에 저장된다!

- 예: `double *p;`
  - p는 double을 가리키는 포인터 변수로 선언되었다.
  - 포인터는 double형의 변수를 가리킬 수 있다.
    - 그 외 타입의 변수는 가리킬 수 없다!
      (위험을 감수하고 typecast를 사용하지 않는 한)

포인터 변수의 선언

- 변수 선언 방식
  - 포인터는 다른 타입과 같이 선언된다.
  - 변수 이름 앞에 “*”을 붙인다.
  - 그 타입을 가리키는 포인터를 생성한다.
- "*" 은 각 변수 앞에 하나씩 있어야 한다.
-
```
int *p1, *p2, v1, v2;
```
  - p1, p2 은 int형을 가리키는 포인터이다.
  - v1, v2 은 평범한 int형 변수이다.

주소와 숫자

- 포인터는 주소이고, 주소는 정수이지만
  포인터는 정수가 아니다!
  - 추상화된 개념일 뿐!
- C++ 포인터가 주소로만 사용되도록 한다.
  - 비록 이것이 숫자라 할지라도 숫자로는 사용될 수 없다.

- Pointing!
  - “주소“가 아니라, “가리키는 것”에 대한 말
  - 포인터 변수는 다른 메모리 공간을 “가리킨다”
  - “주소”에 대한 말은 생략한다.

- 가독성 높이기
  - 화살표로 그리면 메모리 참조를 눈으로 볼 수 있다.

…을 가리키다

```
int *p1, *p2, v1, v2;
p1 = &v1;
```

- 포인터 변수 p1이 int형 변수 v1를 “가리키도록” 설정한다
  - 연산자 &는 변수의 주소를 나타냄
  - "p1은 v1의 주소와 같다 " 또는 "p1이 v1를 가리킨다 " 로 읽음.

- 지금 v1을 참조하는 두 가지 방법:
  - 변수 v1을 직접 참조: `cout << v1;`
  - 포인터 p1을 통해서: `cout *p1;`

- 역 참조 연산자, *
  - “역 참조된” 포인터 변수
  - 의미: “p1이 가리키는 데이터를 얻는다"

"Pointing to(가리키는 것)" 예시

- 생각해 보기:

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

```
42
42
```

  - p1과 v1은 동일한 변수를 참조한다.

- & 연산자: "주소"연산자
  - call-by-reference매개변수를 명시하는데도 사용되었음
    - 우연의 일치가 아니다!
    - 복습: call-by-reference 매개변수는 인자의 실제 주소를 전달한다.
  - 역참조 연산자 * 와 대응

포인터의 할당

- 포인터 변수는 “할당”될 수 있다:

```
int *p1, *p2;
p2 = p1;
```

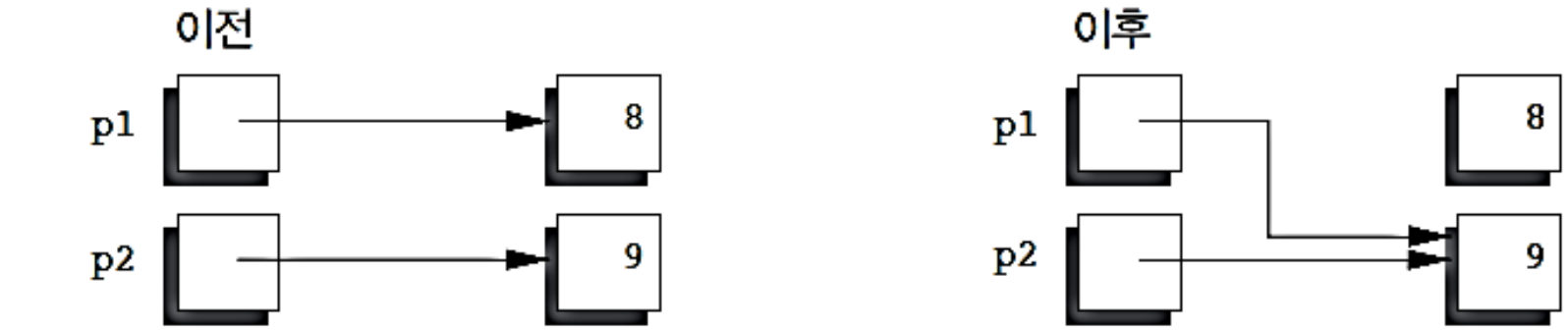
- 하나의 포인터를 또 다른 포인터에 할당한다
  - “p2가 p1이 가리키는 곳을 할당하도록 만든다”
- 다음의 것과 혼동해서는 안 된다:

  *p1 = *p2;
  - P1이 “가리키는 값”을, p2가 “가리키는 값”에 할당한다

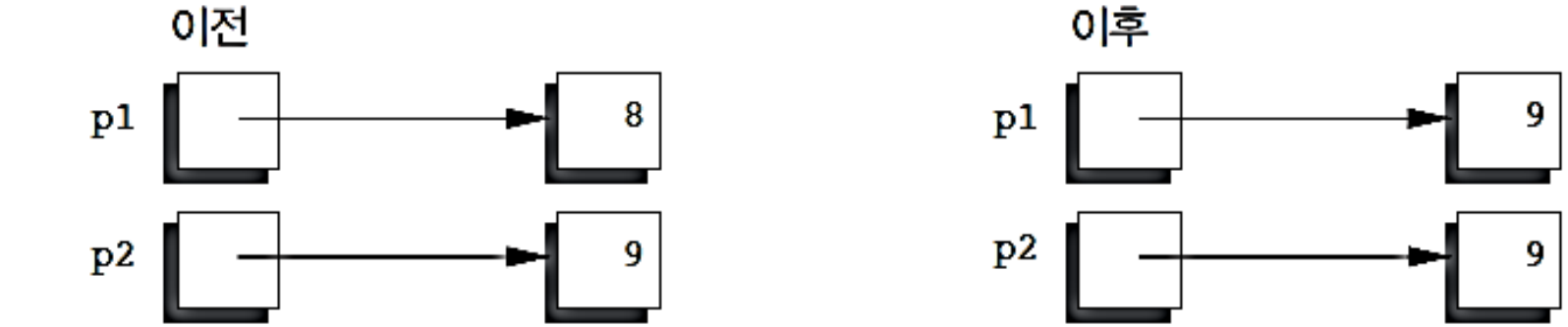
포인터 변수를 동반하는 할당 연산자의 사용

디스플레이 10.1 포인터 변수와 할당문의 사용

```
p1 = p2;
```




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



New 연산자

- 포인터가 변수를 참조할 수 있기 때문에…
  - 표준 식별자를 가질 필요는 없다.
- 변수를 동적으로 할당할 수 있다.
  - 연산자 new는 메모리 공간을 할당한다.
    - 이들을 가리키는 식별자는 없다.
    - 단지 하나의 포인터!
- p1 = new int;
  - 새로운 “이름없는” 변수를 생성하고,
    p1에 이것을 “가리키는 포인터”를 할당한다.
  - *p1에 접근하여, 일반 변수와 같이 사용한다.

디스플레이 10.2 기본적인 포인터의 사용

디스플레이 10.2 기본 포인터 연산의 예

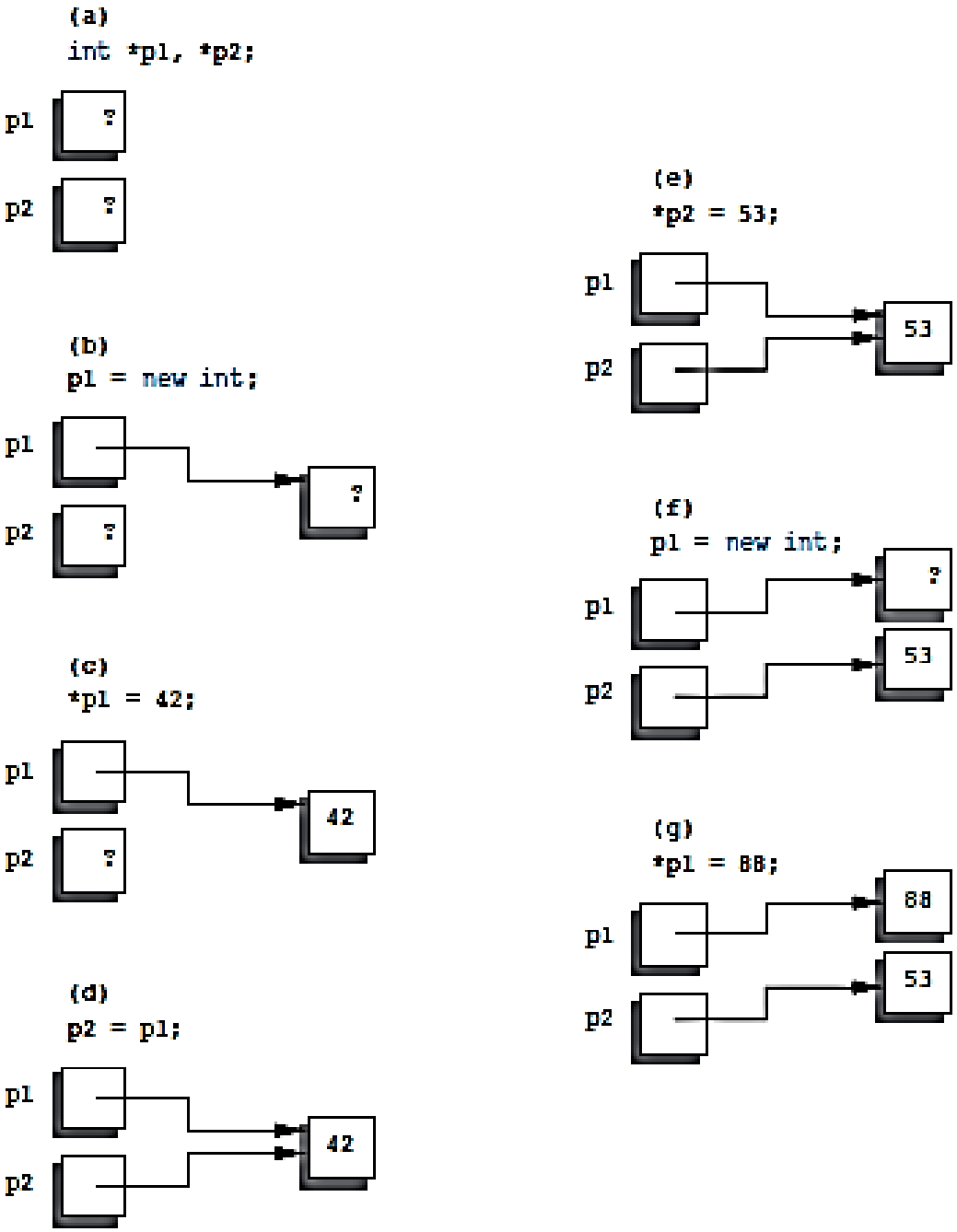
```
1   //포인터와 동적 변수에 관한 예제 프로그램.
2   #include <iostream>
3   using namespace std;

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

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

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

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

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

Sample Dialogue

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

기본적인 포인터 사용 그림: 디스플레이 10.3 디스플레이 10.2의 확장

디스플레이 10.3 디스플레이 10.2의 그림 설명



New연산자에 대해 더 알아보자

- 새로운 동적 변수를 생성하여 이에 대한 포인터를 반환.
- 데이터 형(type)이 클래스형인 경우:
  - 새 객체를 위해 생성자가 호출
  - 초기화 인자를 지닌 다른 생성자를 호출할 수도 있음:

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

  - 클래스가 아닌 데이터 형도 초기화 할 수 있다.

```
int *n;
n = new int(17);    //*n을 17로 초기화
```

메모리 관리

- 힙(heap) : “자유저장공간”
  - 동적 할당변수를 위해 준비된 공간.
  - 모든 새로운 동적 변수들은 자유저장공간의 메모리를 사용.
    - 너무 많을 경우→ 모든 자유저장공간의 메모리를 소모할 수도 있음.
    - 자유저장공간이 포화상태인 경우, "new" 연산자의 호출에 실패할 수도 있음.

- 예전 컴파일러:

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

- 새로운 컴파일러:
  - New연산자가 실패하면, 에러메시지 출력 후 프로그램은 자동 종료
  - 허나, NULL을 이용해 점검하는 것은 좋은 프로그래밍 습관이다.

자유저장공간의 크기

- 프로그램이 모든 메모리를 사용하는 경우는 거의 없음
- 메모리 관리
  - 여전히 좋은 습관
  - 견고한 소프트웨어 프로그래밍의 원칙
  - 메모리는 제한되어 있다.
    - 얼마나 많이 있느냐에 상관없이!

delete 연산자

- 동적 메모리의 할당을 해제한다.
  - 더 이상 필요 없을 때
  - 메모리를 자유저장공간에 반납한다.
  - 예시:

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

  - “p가 가리키는” 동적 메모리의 할당을 해제한다.
    - 문자 그대로 메모리를 “파괴한다”

허상 포인터(dangling pointer)

- delete p; 를 수행한 뒤에 p를 사용하는 경우
  - 반환돤 동적 메모리를 p가 여전히 가리키고 있는 상태
  - 이후에 p가 역참조될 경우 (*p)
    - 때때로 재앙적 수준의 예측할 수 없는 결과를 얻음!

- 허상 포인터 피하기
  - delete사용 후 NULL을 포인터에 할당한다:

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

동적 변수 와 포인터 타입의 정의

- 동적 변수 v.s. 자동 변수
  - 동적변수: new연산자로 생성되어, 프로그램 실행 중에 생성되고 파괴.
  - 지역 변수 : 함수 정의 안에서 선언되는 정적인 변수
    - 영역이 시작될 때/함수가 호출될 때 생성되고, 끝나면 파괴됨
    - 종종 “자동”변수라고 불린다.

- 포인터 타입의 정의
  - 포인터 타입의 명명이 가능
  - 포인터를 다른 변수들처럼 선언하기 위해
    - 포인터 선언에서 "*" 없이 할 수 있는 방법을 제공

```
typedef int* IntPtr;

IntPtr p;
//int *p;와 동일
```

포인터와 함수

- 포인터는 데이터 형으로서의 자격을 모두 갖췄다.
  - 다른 데이터 형이 사용되는 방식과 똑같이 사용될 수 있다.

- 함수의 매개변수가 될 수 있고, 함수로부터 반환될 수 있음
- 예: `int* findOtherPointer(int* p);`
  - 이 함수의 선언:
    - "int를 가리키는 포인터"를 매개변수로 갖고,
    - "int를 가리키는 포인터" 변수를 반환.

- 포인터에 대한 call-by-value
  - 역참조에 대해서는 call-by-reference처럼 동작
  - C언어에서 call-by-reference를 구현하는 방식

```
void swap(int *val1, int *val2)
{
    int temp = *val1 ;
    *val1 = *val2 ;
    *val2 = temp ;
}
```

A Call-by-Value 포인터 parameter

디스플레이 10.4 call-by-value 포인터 인자

```
1   //call-by-value 인자의 동작 과정을
2   //설명하는 예제 프로그램.
3   #include <iostream>
4   using namespace std;

5   typedef int* IntPointer;

6   void sneaky(IntPointer temp);

7   int main( )
8   {
9       IntPointer p;

10      p = new int;
11      *p = 77;
12      cout << "Before call to function *p == "
13           << *p << endl;

14      sneaky(p);

15      cout << "After call to function *p == "
16           << *p << endl;

17      return 0;
18  }
19  void sneaky(IntPointer temp)
20  {
21      *temp = 99;
22      cout << "Inside function call *temp == "
23           << *temp << endl;
24  }
```

Sample Dialogue

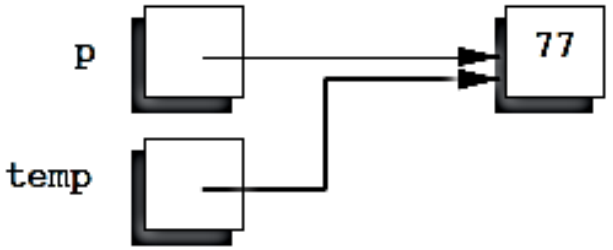
```
Before call to function *p == 77
Inside function call *temp == 99
After call to function *p == 99
```

디스플레이 10.5 함수 sneaky(p) 호출;

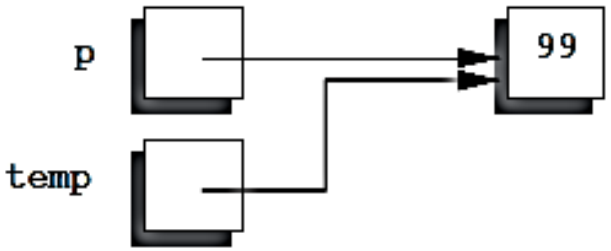
디스플레이 10.5 함수의 호출 `sneaky(p);`

1. sneaky 호출 전

2. temp가 p의 값에 접속됨



3. *temp의 값이 변경됨



4. sneaky 호출 후

동적 배열

- 포인터 변수로 크기를 조정할 수 있는 배열을 만들 수 있음
  - 표준 배열을 정적으로 고정된 크기로만 정의하고, 변경할 수 없음
    - 배열은 메모리 주소에 연속적으로 저장
    - 배열 변수는 첫 번째 인덱스의 변수를 가리킴
    - 따라서 배열 변수는 포인터 상수!

```
int a[10];
int * p;
```

- a 와 p 는 포인터 변수
  - 포인터는 할당 가능

```
p = a;     // 가능.
```

    - p 는 지금부터 가리켜야 할 곳을 가리킴
      - 배열 a변수의 첫 번째 인덱스로
  - 배열은 할당 가능

```
a = p;     // 불가능!
```

    - 배열 포인터는 상수 포인터!

```
int a[10];
typedef int* IntPtr;
IntPtr p;
```

배열 변수→ 포인터

- 배열 변수 `int a[10];`
  - "const int *" 타입
  - 배열은 이미 메모리에 할당되어서 변수 a 는 반드시 할당 메모리를 항상 가리켜야 하며, 절대 바뀔 수 없음!
  - 반드시 크기를 처음에 정해야 하지만, 프로그램 구동전 항상 알 수 있나?
    - 항상 최대 크기로 할당해야 하므로 메모리가 낭비되고, 혹시 더 커지만 구동 불가

- 포인터는 필요에 따라 크기를 바꿀 수 있음
  - new 연산자를 사용하여 포인터 변수에 메모리를 동적으로 할당
  - 표준 배열처럼 사용할 수 있음

```
typedef double * DoublePtr;
DoublePtr d;
d = new double[10];    //대괄호로 크기 정하기
```

    - 동적으로 할당되는 배열 변수 d를 10개의 요소와 기본 타입 double 로서 생성

동적 배열의 삭제와 함수에서의 반환

- 프로그램이 실행될 때 동적으로 할당된 메모리는, 반드시 프로그램이 실행되고 있을 때 직접 제거해야 함
- 간단. 예시 복습:

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

- 동적 배열을 위한 모든 메모리의 할당을 제거
  - 대괄호는 배열이 있다는 것을 의미
  - d는 여전히 포인터이므로 d = NULL; 해주기

- 배열 타입은 함수의 반환타입으로 허가되지 않지만, 동적 배열은 함수의 반환이 될 수 있음

```
int [] someFunction();   // 불가능!
int* someFunction();  // 가능!
```

자료구조의 다항식의 예(1/3)

- 다항식 표현의 첫번째 방법 : 정적 배열
  - 각 계수들을 지수 인덱스 위치에 저장

```
class Polynomial
{
private:
    int degree;              //degree ≤  MaxDegree
    float coef [MaxDegree + 1];     // 계수 배열
```

  - Polynomial instance끼리의 다항식 연산을 매우 손쉽게 구현할 수 있음
  - 최대 차수가 명확하게 결정되는 경우 매우 효율적
  - 최대 차수보다 큰 크기의 배열을 “처리 불가 " 한 치명적인 문제점

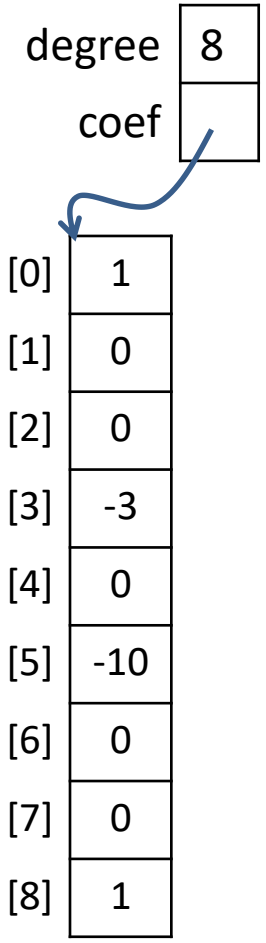
$a(x)=3x^2+2x-4$

$b(x)=x^8-10x^5-3x^3+1$

[10]	0	0
[9]	0	0
[8]	0	1
[7]	0	0
[6]	0	0
[5]	0	-10
[4]	0	0
[3]	0	-3
[2]	3	0
[1]	2	0
[0]	-4	1

자료구조의 다항식의 예(2/3)

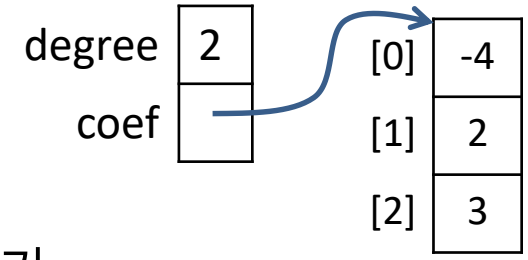
$b(x)=x^8-10x^5-3x^3+1$



- 다항식 표현의 두번째 방법 : 동적 배열

```
class Polynomial {
private:
      int degree;
      float *coef;
```

$a(x)=3x^2+2x-4$



  - 생성자와 파괴자를 Polynomial에 추가

```
Polynomial::Polynomial(int d) {
    degree=d;
    coef=new float[degree+1];
}
```

```
Polynomial::~Polynomial() {
    delete [] coef ;
}
```

  - 희소 다항식에서 기억 공간 낭비
    - (예) 다항식 $x^{1000}+1$ → coef에서 999개의 엔트리는 0

자료구조의 다항식의 예(3/3)

- 다항식 표현의 세번째 방법 : 희소다항식 표현
  - 계수가 0이 아닌 항만 저장.
  - termArray의 각 원소는 term 타입
  - Polynomial의 정적 클래스 데이터 멤버

```
class Polynomial
{
private:
    Term *termArray;   // 0이 아닌 항의 배열
    int capacity;      // termArray의 크기
    int terms;         // 0이 아닌 항의 수
```

```
class Polynomial;     //전방 선언
class Term
{
friend Polynomial;
private:
    float coef;   // 계수
    int exp;      // 지수
};
```

다항식 배열의 크기를 2배로 증가시키는 함수

Program 2.9:Adding a new term, doubling array size when necessary

```
==============================================
void Polynomial::NewTerm(const float theCoeff, const int theExp)
{// Add a new term to the end of termArray.
     if (terms == capacity)
     {// double capacity of termArray
          capacity *= 2;
          term *temp = new term[capacity];    // new array
          copy(termArray, termArray + terms, temp);
          delete [] termArray;        // deallocate old memory
          termArray = temp;
      }
     termArray[terms].coef = theCoeff;
     termArray[terms++].exp = theExp;
}
==============================================
```

```
1const Polynomial Polynomial::operator +(const Polynomial b) const
2{// Return the sum of of the polynomials *this and b.
3     Polynomial c;
4     int aPos = 0, bPos = 0;
5     while ((aPos < terms) && (bPos < b.terms))
6        if ((termArray[aPos].exp == b.termArray[bPos].exp) {
7                  float t = termArray[aPos].coef + b.termArray[bPos].coef;
8                  if (t) c.NewTerm(t, termArray[aPos].exp);
9                   aPos++; bPos++;
10       }
11       else if ((termArray[aPos].exp < b.termArray[bPos].exp) {
12                  c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
13                  bPos++;
14       }
15       else {
16                  c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
17                  aPos++;
18       }
19    // add in remaining terms of *this
20    for (; aPos < terms ; aPos++)
21         c.NewTerm(termArray[aPos].coef, termArray[aPos].exp);
22    // add in remaining terms of b(x)
23    for (; bPos < b.terms; b++)
24         c.NewTerm(b.termArray[bPos].coef, b.termArray[bPos].exp);
25    return c;
26}
```

참조형 변수

- 저장 위치의 이름으로 포인터와 유사하지만, 별명으로 봐야 함
- 독립 참조의 예제:
  - bob은 robert 의 저장 위치를 참조
  - 어떠한 Bob의 변화도 robert를 변하게 함

```
int robert;
int& bob = robert;
```

- 가독성이나 유지보수성에 유해하지만 몇가지 경우 매우 유용함
  - call by reference
  - 참조의 리턴
    - 연산자의 오버로딩을 자연스럽게 함

```
double& sampleFunction(double& variable){ return variable; }

int main()
{
      double m = 99 ;
      cout << sampleFunction(m) << endl ;
      sampleFunction(m) = 42 ;
      cout << m << endl ;
```

디스플레이 8.5 << 와 >>의 오버로딩

```
29  int main( )
30  {
31      Money yourAmount, myAmount(10, 9);
32      cout << "Enter an amount of money: ";
33      cin >> yourAmount;
34      cout << "Your amount is " << yourAmount << endl;
35      cout << "My amount is " << myAmount << endl;
36
37      if (y
38
39      else
40
41      Money
42      cout
43
44      Money
45      cout
46
47      retu
48  }
```

```
66  //iostream과 cstdlib 사용:
67  istream& operator >>(istream& inputStream, Money& amount)
68  {
69      char dollarSign;
70      inputStream >> dollarSign; //희망을 가지고
71      if (dollarSign != '$')
72      {
73          cout << "No dollar sign in Money input.\n";
74          exit(1);
75      }
76      double amountAsDouble;
77      inputStream >> amountAsDouble;
78      amount.dollars = amount.dollarsPart(amountAsDouble);
79      amount.cents = amount.centsPart(amountAsDouble);

80      return inputStream;
81  }
```

main 함수에서 cin이 inputStream에 대해 플러그인되었다.

멤버 연산자가 아니기 때문에 Money의 멤버 함수에 대해 객체를 호출하도록 기술해야 한다.

참조 리턴

포인터 연산

- 포인터로 주소에 대한 연산 수행이 가능
  - 주소에 대해 +와 -, ++, -- 연산 가능 (*, /는 불가능)

- 예시:

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

```
d == &d[0]
d+1==&d[1] d+2==&d[2]
```

- 포인터 연산으로 배열 조작하기

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

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

다차원 동적 배열

- “배열의 배열”로 구현
-
```
typedef int* IntArrayPtr;
IntArrayPtr *m = new IntArrayPtr[3];

for (int i = 0; i < 3; i++)
     m[i] = new int[4];
```
  - 세 개의 포인터를 갖는 배열 생성
  - 각 포인터당 4개의 int 할당
  - 결과적으로 3x4 동적 배열이 생성!

클래스에 대한 포인터!

- -> 연산자
  - 역 참조 연산자 *, 점 연산자를 조합
  - 주어진 포인터로 클래스의 멤버를 명시화

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

- this 포인터
  - 멤버 함수에서 호출 객체를 참조할 때 사용

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

```
void Simple::showStuff() const
{
        cout << stuff ;
}
```

```
void Simple::showStuff() const
{
        cout << this->stuff ;
}
```

할당 연산자 오버로딩

- 할당 연산자는 참조를 반환 `a = b = c;`
- 연산자는 반드시 그것의 왼쪽 항과 같은 타입을 반환해야 함
  - 사슬이 동작하게 끔 해야 함
  - This 포인터가 이것을 도와주게 됨!
- 할당 연산자는 반드시 클래스의 멤버여야만 함
  - 하나의 parameter를 가지며, 왼쪽 피 연산자가 호출 객체

```
s1 = s2;   // s1.=(s2) 와 동일
```

- s1 = s2 = s3;
  - s1 = (s2 = s3) 를 요구;
  - 따라서 (s2 = s3) 는 반드시 s2 타입의 객체를 반환 해야 함
    - 그리고 " s1 = " 으로 패스;

= 연산자 정의 오버로드

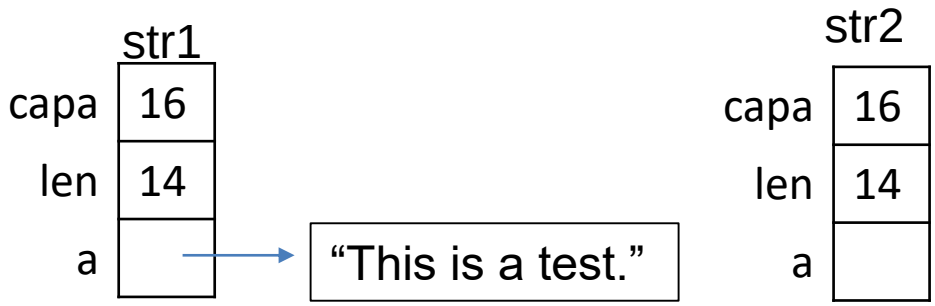
- string Class 예시 사용:

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

얕은 복사와 깊은 복사

- 얕은 복사
  - 할당 복사는 오직 멤버 변수만
  - 기본 할당과 복사 생성자

```
StringClass str1("This is a test."), str2 ;
str2 = str1 ; // 어떻게 동작할까?
        // 어떻게 동작하기를 원하나?
```



- 깊은 복사
  - 포인터, 동적 메모리 포함
  - 반드시 데이터 복사를 위해 역 참조 포인터 변수를 사용
  - 당신만의 할당 오버로드와 복사 생성자를 작성!

클래스 소멸자

- 동적 할당 변수는 삭제 될 때까지 사라지지 않음
- private 멤버 데이터에 동적 할당 받은 포인터가 있다면 객체가 사라질 때 반드시 "할당제거" 되어야
- 해답: 소멸자!

- 생성자의 반대
  - 객체가 범위를 벗어날 때 자동으로 호출
  - 기본 버전은 동적 변수가 아닌 오직 기본 변수만 제거
- 생성자처럼 정의, ~ 만 추가

```
MyClass::~MyClass()
{
      //삭제 정리 작업을 수행
    }
```

복사 생성자

- 자동으로 호출되는 경우:
  - 클래스의 객체가 선언되고 다른 객체로 초기화되는 경우
  - 함수가 클래스 타입의 값을 반환하는 경우
  - 클래스 타입의 인자가 call-by-value 인자인 “plugged in” 인 경우
- 객체의 “일시적인 복사” 가 요구되어 짐
  - 복사 생성자가 이것을 생성
- 기본 복사 생성자
  - 기본 “=“ 처럼, 멤버 복사 수행
- 포인터 → 자기만의 복사 생성자 작성!

```
#ifndef PFARRAYD_H
#define PFARRAYD_H

//Objects of this class are partially filled arrays of doubles.
class PFArrayD
{
public:
    PFArrayD( );
    //Initializes with a capacity of 50.

    PFArrayD(int capacityValue);

    PFArrayD(const PFArrayD& pfaObject);

    void addElement(double element);
    //Precondition: The array is not full.
    //Postcondition: The element has been added.

    bool full( ) const { return (capacity == used); }
    //Returns true if the array is full, false otherwise.

    int getCapacity( ) const { return capacity; }

    int getNumberUsed( ) const { return used; }

    void emptyArray( ){ used = 0; }
    //Empties the array.

    double& operator[](int index);
    //Read and change access to elements 0 through numberUsed - 1.

    PFArrayD& operator =(const PFArrayD& rightSide);

    ~PFArrayD( );
private:
    double *a; //for an array of doubles.
    int capacity; //for the size of the array.
    int used; //for the number of array positions currently in use.
};
#endif //PFARRAYD_H
```

```
#include "pfarrayd.h"
#include <iostream>
using namespace std;

PFArrayD::PFArrayD( ) :capacity(50), used(0)
{
    a = new double[capacity];
}

PFArrayD::PFArrayD(int size) :capacity(size), used(0)
{
    a = new double[capacity];
}

PFArrayD::PFArrayD(const PFArrayD& pfaObject)
  :capacity(pfaObject.getCapacity( )),
            used(pfaObject.getNumberUsed( ))
{
    a = new double[capacity];
    for (int i =0; i < used; i++)
        a[i] = pfaObject.a[i];
}

void PFArrayD::addElement(double element)
{
    if (used >= capacity)
    {
        cout << "Attempt to exceed capacity in PFArrayD.₩n";
        exit(0);
    }
    a[used] = element;
    used++;
}
```

```
double& PFArrayD::operator[](int index)
{
    if (index >= used)
    {
        cout << "Illegal index in PFArrayD.₩n";
        exit(0);
    }

    return a[index];
}

PFArrayD& PFArrayD::operator =(const PFArrayD& rightSide)
{
    if (capacity != rightSide.capacity)
    {
        delete [] a;
        a = new double[rightSide.capacity];
    }

    capacity = rightSide.capacity;
    used = rightSide.used;
    for (int i = 0; i < used; i++)
        a[i] = rightSide.a[i];

    return *this;
}

PFArrayD::~PFArrayD( )
{
    delete [] a;
}
```

요약

- 포인터는 메모리 주소로 간접적으로 변수에 접근하도록 제공
- 동적 변수로 프로그램이 실행되는 동안 생성되고 소멸
- heap은 동적 변수를 위한 메모리 저장 공간
- 동적 할당 배열은 프로그램이 실행되는 동안 크기가 결정
- 클래스 소멸자는 특수 멤버 함수로 자동으로 객체를 소멸
- 복사 생성자는 단일 인자 멤버 함수로
  내 클래스 타입이 call-by-value 인자로 전달될 때 때 자동 호출
- 할당 연산자는 멤버 함수로서 반드시 오버로드 되어야 하며,
  체인을 위해 참조를 반환