<8장> 포인터 1

학습 목표

- 포인터의 개념, 변수 선언 방법, 포인터 연산 등 사용법을 이해한다.
- 포인터 연산을 이용한 배열 사용 방법을 학습한다.
- 함수에 포인터를 전달하거나 포인터 매개 변수 활용, 포인터 반환, 상수 포인터 등을 학습한다.
- 함수 포인터를 이해한다.
- void 포인터를 학습한다.

목차

01

포인터와 필요성

1. 포인터와 필요성

- 메모리 주소를 표현

그림 8-1 포인터 변수를 이용해서 간접적으로 메모리 공간의 값을 사용하거나 변경 가능

1. 포인터와 필요성

- 포인터의 필요성

  - 동적할당받는 메모리 공간 사용
  - 자료구조 구현
  - 메모리를 직접 접근하고 사용하는 기능을 통한 하드웨어 조작
  - 배열 구현

02

포인터 자료형 변수 사용

2. 포인터 자료형 변수 사용

포인터 자료형 변수 선언 방법

- 포인터 변수를 선언하는 방법

```
TYPE* 변수_이름 = [초깃값];
```

- 포인터 변수를 선언하고 NULL 포인터로 초기화

코드 8-1 포인터 변수 선언 및 초기화

```
TYPE* ptr = NULL;
```

2. 포인터 자료형 변수 사용

NULL 포인터

- 주솟값이 없다는 의미의 상수(constant value)
- 주로 포인터 변수를 초기화할 때 사용

코드 8-2 NULL 포인터 사용을 위해 헤더 파일 포함

```
#include <stddef.h>

TYPE* ptr = NULL;
```

2. 포인터 자료형 변수 사용

포인터 자료형의 크기

- 메모리 주소의 크기는 하드웨어와 운영체제에 의해서 결정
- 때문에 시스템에 따라서 다를 수 있음
- 비주얼 스튜디오 컴파일러나 GCC는 32 비트/64 비트용 코드 생성을 선택 가능
- 어떤 종류의 코드를 생성하느냐에 따라 포인터 변수의 크기는 달라짐

2. 포인터 자료형 변수 사용

포인터 변수 사용과 간접 참조 연산자(*)

- 간접 참조(dereference)란 포인터 변수에 저장되어 있는 메모리 주소에 있는 값을 접근하는 것
- 간접 참조는 역참조라고도 부름
- 때로는 포인터 변수가 가리키는 메모리 공간의 값에 접근한다고 말하기도 함

포인터 변수로 간접 참조하는 방법

- 변수 이름 앞에 간접 참조 연산자인 '*'를 붙임

2. 포인터 자료형 변수 사용

포인터 변수 사용과 간접 참조 연산자(*)

- ptr 변수에 메모리 주소 1000을 저장
- ptr 변수를 이용해서 메모리 주소 1000에 값을 저장
- 읽어서 출력

코드 8-3

```
1    TYPE* ptr = (TYPE*) 1000; // 정수 값을 메모리 주소로 변환해서 포인터 변수에 저장
2    *ptr = VALUE; // ptr에 주어진 주소에 VALUE 값을 저장
3    TYPE v = *ptr; // ptr에 주어진 주소에 있는 값 (VALUE)을 간접 참조를 통해 v 변수에 저장
```

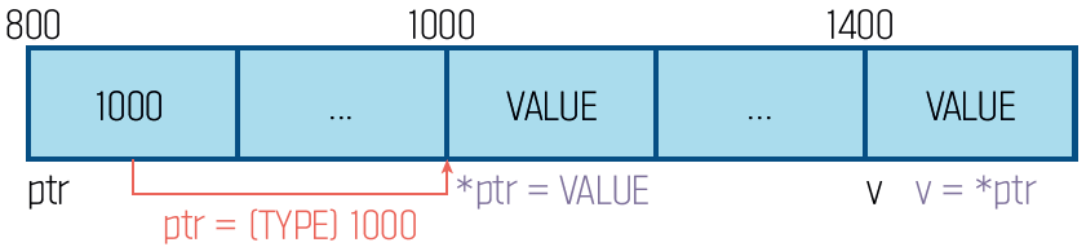

그림 8-2 포인터 변수를 이용해서 간접 참조

2. 포인터 자료형 변수 사용

포인터 변수의 자료형(TYPE) 지정

- 포인터 변수 선언

```
TYPE* ptr = NULL;
```

- 여러 가지 포인터 변수 선언

코드 8-4

```
1    int* nPtr = NULL;
2    float* fPtr = NULL;
3    unsigned int* uPtr = NULL;
4    char* cPtr = NULL;
```

2. 포인터 자료형 변수 사용

■ 포인터와 주소 연산자

- 주소 연산자를 이용해 기존 변수의 주소를 저장

코드 8-5 주소 연산자와 포인터 변수 사용

```
1   int* nPtr = NULL; // nPtr의 변수값을 NULL로 초기화
2   int n = 5;
3   nPtr = &n;          // n 변수의 주소를 nPtr에 저장
4   *nPtr = 3;
```

nPtr 변수를 선언하고 NULL로 초기화. nPtr은 n과는 다른 메모리 공간에 있는 포인터 변수(그림 8-3 참조)

주소 연산자를 이용해서 n의 주소 308을 nPtr에 저장

nPtr의 메모리 주소를 간접 참조해서 3을 저장. 즉, 변수 n의 메모리 공간에 3을 저장

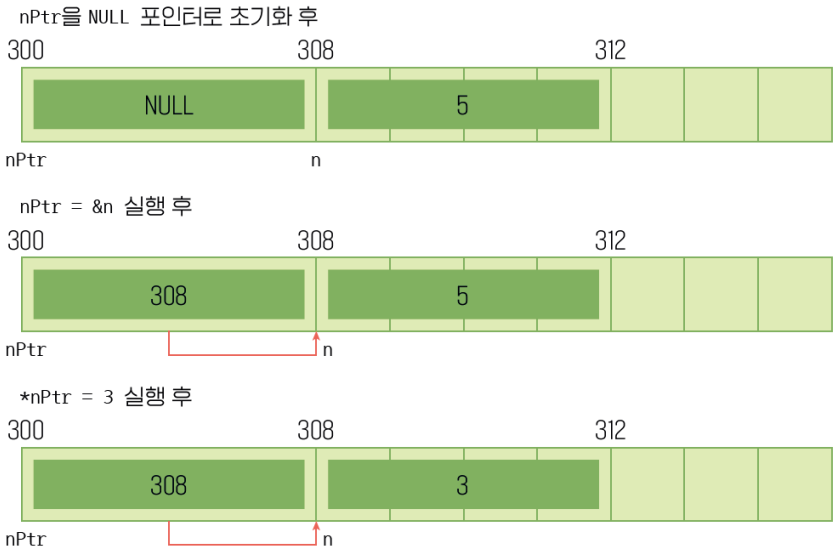

그림 8-3 코드 8-5가 실행되는 과정

2. 포인터 자료형 변수 사용

포인터 형 변환

- TYPE이 다른 포인터 변수는 다른 자료형으로 인식
- 다른 자료형 포인터 사이에 값을 저장하려면 강제 형 변환 사용

코드 8-6

```
1    TYPE1 var1;
2    TYPE1* ptr1 = &var1;
3    TYPE2* ptr2 = ptr1; // 컴파일 오류 (TYPE1과 TYPE2가 다른 자료형이라고 가정)
```

- 포인터 형 변환은 (TYPE*)을 붙임

```
TYPE2* ptr2 = (TYPE2*) ptr1; // TYPE1*형 ptr1의 주소를 변환해서 ptr2에 저장
```

2. 포인터 자료형 변수 사용

포인터 형 변환

- int* 형을 unsigned int* 형으로 변환

코드 8-7 Convert1.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   int main(void)
5   {
6       int n = 3;
7       int* ptr1 = &n;
8       unsigned int* ptr2 = (unsigned int*) ptr1;
9
10      printf("*ptr2 = %u\n", *ptr2); // *ptr2 = 3 출력
11      return 0;
12  }
```

변수 n의 주소를 포인터 변수 ptr1에 저장

int 포인터 변수 ptr1을 unsigned int 포인터 변수 ptr2에 강제 형 변환해서 저장

ptr2를 간접 참조해서 ptr2의 주소에 있는 값(변수 n의 값)을 출력

<실행 결과>

```
*ptr2 = 3
```

2. 포인터 자료형 변수 사용

포인터 형 변환

- int* 형을 unsigned int* 형으로 변환

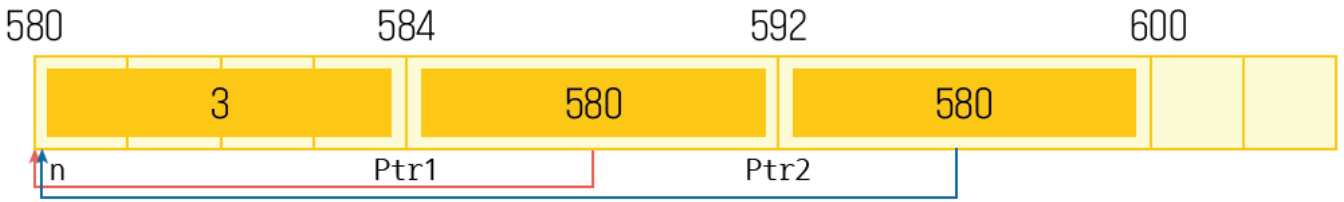

그림 8-4 int*를 unsigned int* 형으로 변환

2. 포인터 자료형 변수 사용

정수를 메모리에 저장한 후 바이트 단위로 출력하는 프로그램

- 정수형 변수 n을 선언하고 0A040E03 저장
- int* 포인터 변수에 n의 주소 저장

```
int n = 0x0A040E03; // n에 16진수로 0A040E03을 저장
int* ptr1 = &n;
```

- ptr1을 char* 형으로 변환하고 첫 번째 바이트를 정수 값 형태로 출력하는 코드

```
char* ptr2 = (char*) ptr1;
printf("%d\n", *ptr2);
```

2. 포인터 자료형 변수 사용

■ 정수를 메모리에 저장한 후 바이트 단위로 출력하는 프로그램

- 형 변환해서 첫 번째 바이트를 출력하는 내용

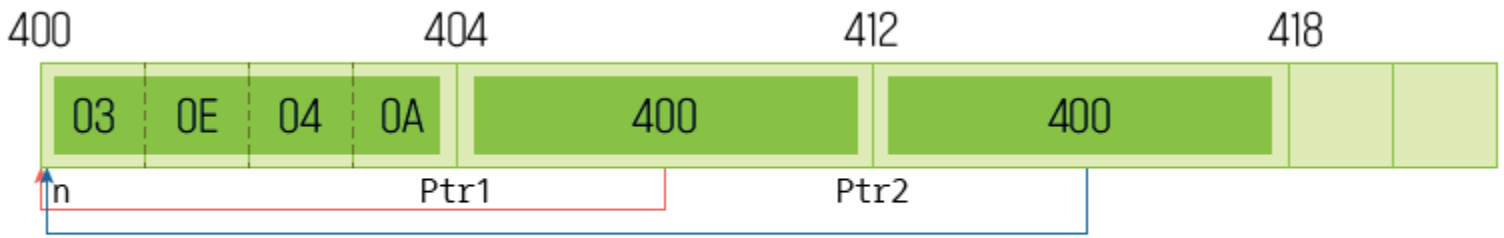

그림 8-5 int*를 char* 형으로 변환. 리틀 엔디안(little endian) 형태로 저장됨

2. 포인터 자료형 변수 사용

■ 정수를 메모리에 저장한 후 바이트 단위로 출력하는 프로그램

- 리틀 엔디안(little endian)을 사용하는 컴퓨터(윈도우 PC, 맥 등)에서 실행하면 0x0A040E03 역순으로 출력

코드 8-8 Convert2.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   int main(void)
5   {
6       int n = 0x0A040E03;
7       int* ptr1 = &n; // p는 n을 가리킴
8       char* ptr2 = (char*) ptr1;
9       for (int i = 0; i < sizeof(int); i++) {
10          printf("%02X", ptr2[i]);
11      }
12      return 0;
13  }
```

변수 n에 16진수로 0A040E03을 저장

ptr1이 n을 가리키도록 함

ptr2에 ptr1을 형 변환해서 저장
ptr2는 char*이므로 n에 바이트 단위로 접근

바이트 단위로 n을 출력하되 2 자리 16진수로 출력('A'~'F'는 대문자로 출력)하고 필요하면 앞에 0을 채워서 자릿수를 맞춤

<실행 결과>

```
03 0E 04 0A
```

2. 포인터 자료형 변수 사용

어려운 포인터 변수 선언

- 포인터 변수를 선언하는 다양한 방법

코드 8-9

```
1    TYPE *ptr2;                    // (1)
2    TYPE    *    ptr3              // (2)
```

- 같은 자료형의 포인터 변수 두 개 이상 선언할 때

코드 8-10

```
1    TYPE *ptr1, *ptr2;              // (3)
2    TYPE* ptr3, *ptr4;              // (4)
3    TYPE   * ptr5, *   ptr6, *ptr7;  // (5)
```

2. 포인터 자료형 변수 사용

어려운 포인터 변수 선언

- 일반 변수와 포인터 변수의 차이는 *로 구별

코드 8-11

```
TYPE *ptr1, var1, *   ptr2;
```

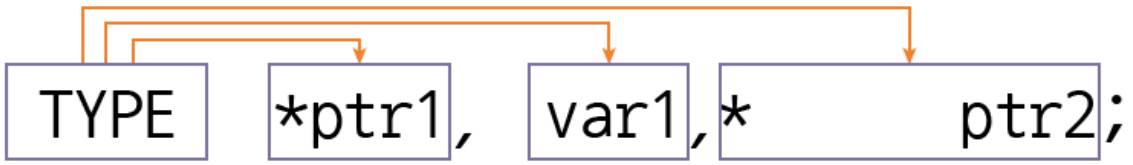

그림 8-6 TYPE이 변수에 적용되는 것을 나타냄

2. 포인터 자료형 변수 사용

■ 포인터 변수 선언 방법 정리

- 변수는 한 줄에 한 개씩 선언한다.
- 만약 포인터 변수를 동시에 여러 개를 선언하고 싶다면 typedef를 사용한다.
- 포인터 변수와 일반 변수를 함께 선언하지 않는다.

2. 포인터 자료형 변수 사용

포인터 변수 선언 방법 정리

- 코드 8-11 다시 선언

코드 8-12

```
TYPE* ptr1;
TYPE var1;
TYPE* ptr2;
```

- typedef로 포인터 변수 선언

코드 8-13

```
typedef TYPE* TYPE_POINTER;

TYPE_POINTER ptr1, ptr2;
TYPE_POINTER ptr3, ptr4;
TYPE_POINTER ptr5, ptr6, ptr7;
```

2. 포인터 자료형 변수 사용

포인터 변수를 이용해서 연산하는 프로그램

- 포인터 변수 만들기

```
double d = 2.3;
double* ptrd = &d;
```

- ptrd에 저장하는 코드

```
*ptrd = *ptrd * 2; // 또는 *ptrd *= 2;
```

- 화면에 d 출력하기

```
printf("d = %f\n", d);
```

※ 포인터 변수를 이용해서 연산하는 프로그램 완성

➔ 코드 8-14
➔ 실행 결과

03

포인터 연산

3. 포인터 연산

- 포인터 변수에 사용할 수 있는 연산, 결괏값의 자료형, 연산의 결괏값

표 8-1 TYPE 포인터 변수 p0과 p1에 사용할 수 있는 산술 연산자와 자료형

연산	결과 자료형	연산 결과
포인터 p0 + 정수 n	포인터(주소)	p0 + (n * sizeof(TYPE))
포인터 p0 − 정수 n	포인터(주소)	p0 − (n * sizeof(TYPE))
포인터 p1 − 포인터 p0	ptrdiff_t(signed 정수)	(p1 − p0) / sizeof(TYPE)

3. 포인터 연산

- 예제 코드
  - int 형 배열을 만들고 요소 중 한 개의 주소를 포인터 변수에 저장

코드 8-15

```
int arr[4] = { 0, 1, 2, 3 };
int* p = &arr[2];
```

  - 코드 8-15에서 선언한 arr 배열

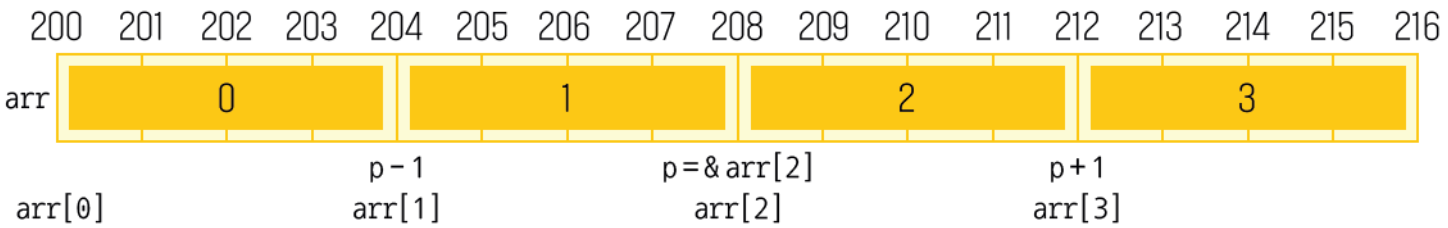

그림 8-7 정수형 포인터와 정수의 덧셈/뺄셈 연산

3. 포인터 연산

- 예제 코드
  - int와 double 형 배열을 선언
  - 세 번째 요소의 주소를 포인터 변수에 저장

코드 8-16 PtrArith.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   int main()
5   {
6       int arr[4] = { 0, 1, 2, 3 };
7       int* p1 = &arr[2];
8       int* p2 = p1 + 1;
9       int* p0 = p1 - 1;
10      printf("p1 = %p\n", p1);
11      printf("p2 = p + 1 = %p\n", p2);
12      printf("p0 = p - 1 = %p\n", p0);
13      printf("p2 - p1 = %td, p2 - p0 = %td\n", p2 - p1, p2 - p0);
14
15      double darr[4] = { 0.0, 1.0, 2.0, 3.0 };
16      double* dp1 = &darr[2];
17      double* dp2 = dp1 + 1;
18      double* dp0 = dp1 - 1;
19      printf("dp1 = %p\n", dp1);
20      printf("dp2 = dp + 1 = %p\n", dp2);
21      printf("dp0 = dp - 1 = %p\n", dp0);
22      printf("dp2 - dp1 = %td, p2 - dp0 = %td\n", dp2 - dp1, dp2 - dp0);
23      return 0;
24  }
```

- 7행: 세 번째 배열 요소의 주소를 p1에 저장
- 8행: p1 + 1은 int의 크기인 4바이트 증가
- 9행: p1 - 1은 int의 크기인 4바이트만큼 감소
- 13행: 포인터의 차(ptrdiff_t 자료형)를 출력할 때 %td를 사용함
- 16행: 세 번째 배열 요소의 주소를 dp1에 저장
- 17행: dp1 + 1은 double의 크기인 8바이트 증가
- 18행: dp1 - 1은 double의 크기인 8바이트만큼 감소

<실행 결과>

```
p1 = 000000054435ff888
p2 = p + 1 = 000000054435ff88c
p0 = p - 1 = 000000054435ff884
p2 - p1 = 1, p2 - p0 = 2
dp1 = 000000054435ff870
dp2 = dp + 1 = 000000054435ff878
dp0 = dp - 1 = 000000054435ff868
dp2 - dp1 = 1, dp2 - dp0 = 2
```

※ int와 double 형 포인터 변수에 증감 연산자를 사용하고 메모리 주소를 출력하는 코드

➔ 코드 8-17
➔ 실행 결과

04

포인터와 배열

4. 포인터와 배열

- C 언어의 포인터는 배열과 밀접하게 관련

■ 배열 이름은 배열의 시작 주소

- 배열의 변수 이름은 배열의 시작 주소를 나타내는 포인터 변수
- 하지만 일반 포인터 변수와는 달리 배열 변수는 다른 주소 저장 불가

포인터 변수와 배열

- 배열 요소의 주소를 포인터 변수에 저장해서 값을 변경하는 코드

코드 8-18 ArrPtr2.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   int main()
5   {
6       int arr[] = { 0, 1, 2, 3 };
7
8       printf("arr[0] = %d, arr[1] = %d\n", arr[0], arr[1]);
9       int* p0 = &arr[0];
10      *p0 = 3;
11      int* p1 = &arr[1]; // arr의 두 번째 요소의 주소 저장
12      *p1 = 5; // 포인터 변수로 arr의 두 번째 요소를 변경
13      printf("arr[0] = %d, arr[1] = %d\n", arr[0], arr[1]);
14      return 0;
15  }
```

arr[0]의 주소를 p0에 저장하고 간접 참조로 3을 저장 arr[0] = 3과 동일한 코드

arr[1]의 주소를 p1에 저장하고 간접 참조로 5를 저장 arr[1] = 5와 동일한 코드

9~12줄에서 포인터 변수를 이용해서 변경한 arr[0]과 arr[1]을 다시 출력

<실행 결과>

```
arr[0] = 0, arr[1] = 1
arr[0] = 3, arr[1] = 5
```

4. 포인터와 배열

포인터 변수와 배열

- 포인터 변수를 선언하고 배열의 시작 주소 저장

```
int* parr = arr; // parr 포인터 변수에 arr 배열의 시작 주소 저장
```

- 배열 변수 arr 대신 parr을 이용해서 배열의 요소를 화면에 출력하는 코드

코드 8-19

```
1    for (int i = 0; i < 4; i++) {
2        printf("arr[%d] = %d\t", i, parr[i]);
3    }
```

arr[i] 대신 parr[i]를 사용

〈실행 결과〉

```
arr[0] = 0      arr[1] = 1      arr[2] = 2      arr[3] = 3
```

포인터 연산을 이용한 배열 접근

- 배열의 요소 위치를 포인터 연산 형태로 표현

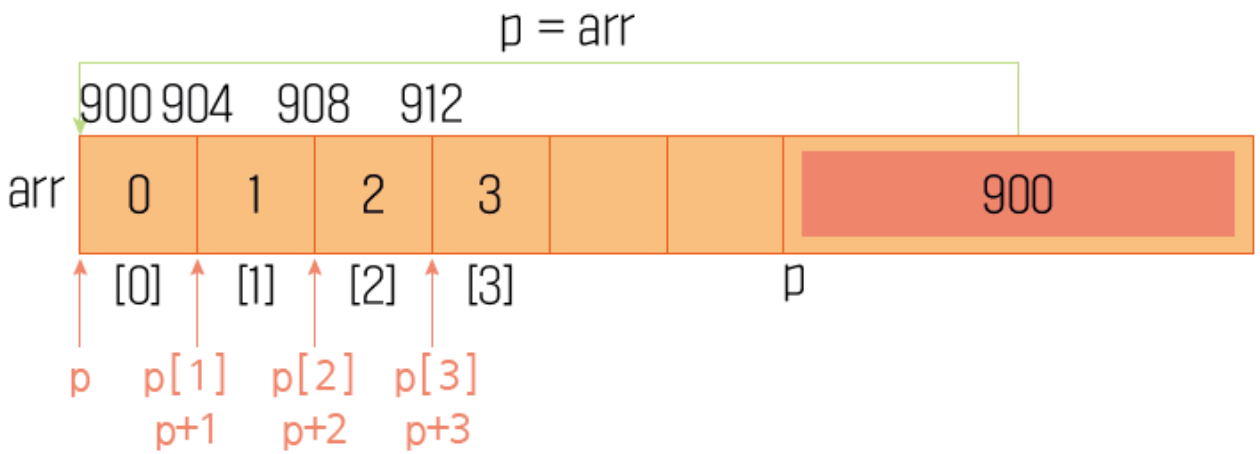

그림 8-8 배열의 인덱스와 포인터 연산

포인터 연산을 이용한 배열 접근

- 배열의 요소들에 접근해서 화면에 값을 출력하는 코드

코드 8-21 ArrPtr4.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   int main()
5   {
6       int arr[] = { 0, 1, 2, 3 };
7       int* p = arr;
8       for (int i = 0; i < 4; i++) {
9           printf("arr[%d] = %d, p[%d] = %d, *(p + %d) = %d\n", i, arr[i], i,
p[i], i, *(p + i));
10      }
11      return 0;
12  }
```

<실행 결과>

```
arr[0] = 0, p[0] = 0, *(p + 0) = 0
arr[1] = 1, p[1] = 1, *(p + 1) = 1
arr[2] = 2, p[2] = 2, *(p + 2) = 2
arr[3] = 3, p[3] = 3, *(p + 3) = 3
```

포인터 연산을 이용한 배열 접근

- 포인터 변수에 중간 인덱스의 주소 저장

그림 8-9 포인터 변수 p가 arr[1]의 주소를 가리키도록 함

4. 포인터와 배열

■ 포인터 연산을 이용한 배열 접근

- 포인터 변수에 arr 배열의 두 번째 요소의 주소를 저장해 마치 새로운 배열인 것처럼 접근

코드 8-22 ArrPtr5.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   int main()
5   {
6       int arr[] = { 0, 1, 2, 3 };
7       int* p = &arr[1];
8       printf("포인터 변수 p는 arr[1]의 주소로 저장됨\n");
9       for (int i = 0; i < 3; i++) {
10          printf("arr[%d] = %d, p[%d] = %d, *(p + %d) = %d\n", i + 1, arr[i +
1], i, p[i], i, *(p + i));
11      }
12      return 0;
13  }
```

<실행 결과>

```
포인터 변수 p는 arr[1]의 주소로 저장됨
arr[1] = 1, p[0] = 1, *(p + 0) = 1
arr[2] = 2, p[1] = 2, *(p + 1) = 2
arr[3] = 3, p[2] = 3, *(p + 2) = 3
```

4. 포인터와 배열

포인터 변수와 증감 또는 복합 연산자를 이용한 배열 접근

- arr에서부터 한 개 요소만큼씩 증가시키면서 다음 요소 가리키기

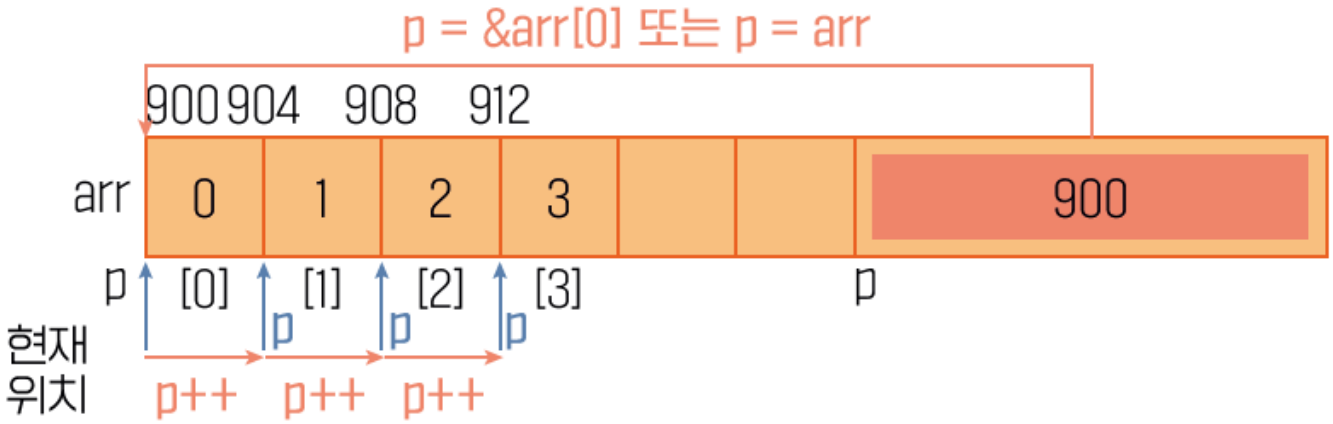

그림 8-10 p를 증가시키면서 다음 요소를 가리킴

4. 포인터와 배열

■ 포인터 변수와 증감 또는 복합 연산자를 이용한 배열 접근

- 메모리 주소와 값을 출력하는 코드

코드 8-23 ArrPtr6.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   int main()
5   {
6       int arr[] = { 0, 1, 2, 3 };
7       int* p = arr;   ← 포인터 변수 p를 arr[0]의 주소를 가리키게 함
8       printf("포인터 변수 p는 arr[0]의 주소로 저장됨\n");
9       printf("&arr[0] = %p, p = %p, arr[0] = %d, *p = %d\n", &arr[0], p,
arr[0], *p);   ← arr[0]과 p의 주소와 값을 출력해서 같은 내용을 가지고 있는 것을 확인
10      p++;   ← p를 1만큼 증가시킴. p에 저장된 주소가 sizeof(int) 바이트만큼 증가됨. p가 arr[1]을 가리킴
11      printf("&arr[1] = %p, p = %p, arr[1] = %d, *p = %d\n", &arr[1], p,
arr[1], *p);   ← arr[1]과 p의 주소가 같은 주소를 보이는 것을 확인
12      return 0;
13  }
```

<실행 결과>

```
포인터 변수 p는 arr[0]의 주소로 저장됨
&arr[0] = 0000005043dff9e0, p = 0000005043dff9e0, arr[0] = 0, *p = 0
&arr[1] = 0000005043dff9e4, p = 0000005043dff9e4, arr[1] = 1, *p = 1
```

4. 포인터와 배열

■ 포인터 변수와 증감 또는 복합 연산자를 이용한 배열 접근

- p를 arr[0]의 주소부터 한 개씩 증가시키면서 배열 전체를 출력하는 코드

코드 8-24 ArrPtr7.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   int main()
5   {
6       int arr[] = { 0, 1, 2, 3 };
7       int* p = &arr[0]; // p = arr; 과 같음
8       for (int i = 0; i < 4; i++, p++) {
9           printf("arr[%d] = %d, *p = %d\n", i, arr[i], *p);
10      }
11      return 0;
12  }
```

p가 배열의 시작 주소를 가리키도록 함

배열 요소를 0부터 3까지 증가시키면서 arr[i]를 출력
동시에 p도 1씩 증가시키면서 *p를 출력함

<실행 결과>

```
arr[0] = 0, *p = 0
arr[1] = 1, *p = 1
arr[2] = 2, *p = 2
arr[3] = 3, *p = 3
```

※ p를 arr[3]의 주소부터 한 개씩 감소시키면서
배열 전체를 출력하는 코드

➔ 코드 8-25
➔ 실행 결과

4. 포인터와 배열

포인터 변수와 증감 또는 복합 연산자를 이용한 배열 접근

- 포인터 변수와 복합 연산자를 사용하는 것도 가능
- p를 2만큼 증가시켜 arr[2]의 주소를 가리키는 코드

```
int arr[] = { 0, 1, 2, 3 };
int* p = arr;
p += 2;
```

4. 포인터와 배열

배열에서 인덱스를 표현하는 []는 연산자임

- 두 개의 피연산자 사용

```
op1[op2]
```

배열 인덱스 연산자 [ ]

- op1과 op2를 바꿔 사용한 코드

코드 8-26 IndexOp.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   int main(void)
5   {
6       int arr[] = { 0, 1, 2, 3 };
7       printf("arr[1] = %d\n", arr[1]);
8       printf("1[arr] = %d\n", 1[arr]);
9       return 0;
10  }
```

일반적인 사용처럼 배열_이름[인덱스] 순서로 사용

인덱스[배열_이름] 형태로 사용해도 동일함

<실행 결과>

```
arr[1] = 1
1[arr] = 1
```

4. 포인터와 배열

배열 인덱스 연산자 [ ]

- 배열에서는 안되지만, 포인터와 사용할 때 음수 인덱스도 사용 가능

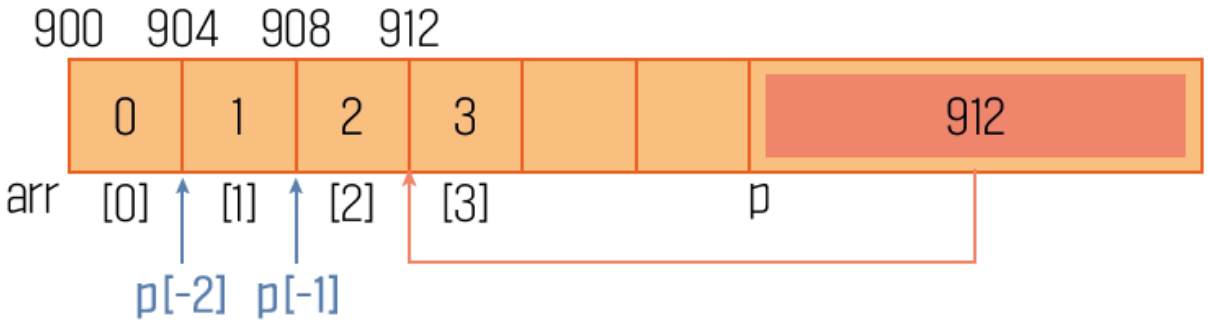

그림 8-11 포인터는 음수로 된 인덱스로 접근 가능

4. 포인터와 배열

배열 인덱스 연산자 [ ]

- 음수 인덱스

코드 8-27 IndexOp2.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   int main()
5   {
6       int arr[] = { 0, 1, 2, 3 };
7       int* p = &arr[3];
8       printf("포인터 변수 p는 arr[3]의 주소로 저장됨\n");
9       printf("arr[%d] = %d, p[%d] = %d, *(p + %d) = %d\n", 2, arr[2], -1, p[-1],
-1, *(p - 1));
10      printf("arr[%d] = %d, p[%d] = %d, *(p + %d) = %d\n", 1, arr[1], -2, p[-2],
-2, *(p - 2));
11      return 0;
12  }
```

변수 arr[3]을 가리키도록 함

p 변수에 음수 인덱스를 이용해 arr[2]와 arr[1]을 확인하는 코드 작성

<실행 결과>

```
포인터 변수 p는 arr[3]의 주소로 저장됨
arr[2] = 2, p[-1] = 2, *(p + -1) = 2
arr[1] = 1, p[-2] = 1, *(p + -2) = 1
```

포인터 연산자의 우선순위

- 배열을 선언하고 포인터 변수로 배열 가리키기

코드 8-28

```
int arr[] = { 0, 1, 2, 3 };
int* p = arr;
```

- 연산의 차이

표 8-2 *(p + 1)과 *p + 1의 차이

연산	설명
*(p + 1)	(p + 1)을 계산하고 그 메모리 주소를 이용해서 간접 참조를 처리한다.
*p + 1	p의 메모리 주소를 간접 참조한 후 그 값에 1을 더한다. *의 우선순위가 +보다 높아 *p가 먼저 처리된다. 즉, p[0] + 1이 된다.

포인터 연산자의 우선순위

- 예제 학습 코드

코드 8-29 PtrRef1.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   int main(void)
5   {
6       int arr[] = { 0, 2, 4, 6 };
7       int* p = arr;
8       printf("*(p + 1) = %d\n", *(p + 1));
9       printf("*p + 1 = %d\n", *p + 1);
10      return 0;
11  }
```

<실행 결과>

```
*(p + 1) = 2
*p + 1 = 1
```

포인터 연산자의 우선순위

- 괄호의 유무에 따른 동작

표 8-3 포인터와 증감 연산자가 사용될 때의 동작 방법 설명

자주 사용함

연산	설명
*p++	*보다 ++(후위 연산 : 변수 뒤에 있을 때)의 우선순위가 높다. 따라서 *(p++)를 계산한다. p++는 p 변수의 값을 사용하고, 나중에 1 증가시키므로 *p를 사용하고 p++가 실행되는 것과 동일하다. 콤마 연산자를 이용해서 표현하면 *p, p++와 같다.
(*p)++	*p를 계산하고 그 값을 1만큼 증가시킨다. p가 가리키는 메모리 공간의 값을 1 증가시킨다.
*++p	*와 ++(전위 연산 : 변수 앞에 있을 때)의 우선순위는 같다. 순서에 따라 *를 먼저 처리하는데 *은 오른쪽부터 왼쪽 방향으로 처리된다. 따라서 *++p는 *(++p)와 같다. p를 1 증가시킨 후 *p를 처리한다. 콤마 연산자를 이용해서 표현하면 p++, *p다.
++*p	순서에 따라 ++를 먼저 처리해야 하는데, 이것도 역시 오른쪽에서 왼쪽으로 처리된다. 따라서 ++(*p)와 같다. p가 가리키는 메모리 공간의 값을 1만큼 증가시킨다.

포인터 연산자의 우선순위

- *p++ 연산

```
*p
p++
```

※ 포인터 연산자의 우선순위 코드

➔ 코드 8-30
➔ 실행 결과

4. 포인터와 배열

배열 == 포인터?

- 배열이 포인터 연산을 이용해서 구현되기는 하지만, 배열과 포인터가 같지는 않음
- 배열 이름은 배열의 첫 번째 요소의 시작 주소
- 배열 이름을 sizeof 연산자에 전달하면 배열의 크기 반환
- 포인터는 주소만 저장하는 변수
- 함수의 반환_자료형에 포인터는 사용 가능/배열은 사용하지 못하는 것도 차이점

```
int* f() { ... };  // 가능
int[] f() { ... }  // 불가능
```

05

함수와 포인터

5. 함수와 포인터

- 함수에 포인터를 전달하면 얻을 수 있는 효과

  - 함수 내부에서 외부 변숫값 변경 가능
  - 크기가 큰 데이터를 함수에 전달할 때 효율적

5. 함수와 포인터

함수에 포인터 인자 전달

- 포인터를 매개변수로 선언한 함수와 사용하는 코드

코드 8-31

```
1   void func(int* p)
2   {
3       printf("p의 메모리 주소: %p, *p = %d\n", p, *p);
4   }
5
6   // 변수 선언 후 함수 호출
7   int a = 10;
8   int* pa = &a;
9   func(pa);           // func(&a)로 호출해도 됨
```

int* 형 매개변수를 포함하는 func() 함수를 정의

변수 p에 저장되어 있는 주소와 그 주소를 간접 참조하는 값을 화면에 출력

변수 a의 주소를 포인터 변수 pa에 저장

func() 함수를 호출하면서 pa에 저장된 주소의 값을 전달

함수에 포인터 인자 전달

- 코드 8-31이 실행되는 과정

그림 8-12 함수를 호출하면서 포인터 값을 전달하는 과정

포인터 매개변수로 함수 외부 변숫값 변경

- 사용자로부터 입력받은 정수 두 개를 반환하는 함수를 구현하고 사용하는 코드

코드 8-32 Scanf2.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   void readTwoInts(int* px, int* py)
5   {
6       int a;
7       int b;
8       printf("정수 두 개를 입력하세요: ");
9       scanf("%d %d", &a, &b);   ← 사용자로부터 정수 2개를 입력받아 변수 a와 b에 저장
10      *px = a;
11      *py = b;                  ← a와 b의 값을 px와 py가 가리키는 메모리 공간에 저장
12  }
13
14  int main( )
15  {
16      int x = 0;
17      int y = 0;
18      readTwoInts(&x, &y);      ← readTwoInts() 함수를 호출할 때 변수 x와 y의 주소를 전달
19      printf("x = %d y = %d\n", x, y);
20      return 0;
21  }
```

<실행 결과>

```
정수 두 개를 입력하세요: 10 20
x = 10 y = 20
```

포인터 매개변수로 함수 외부 변숫값 변경

- 코드 8-32가 실행되는 과정

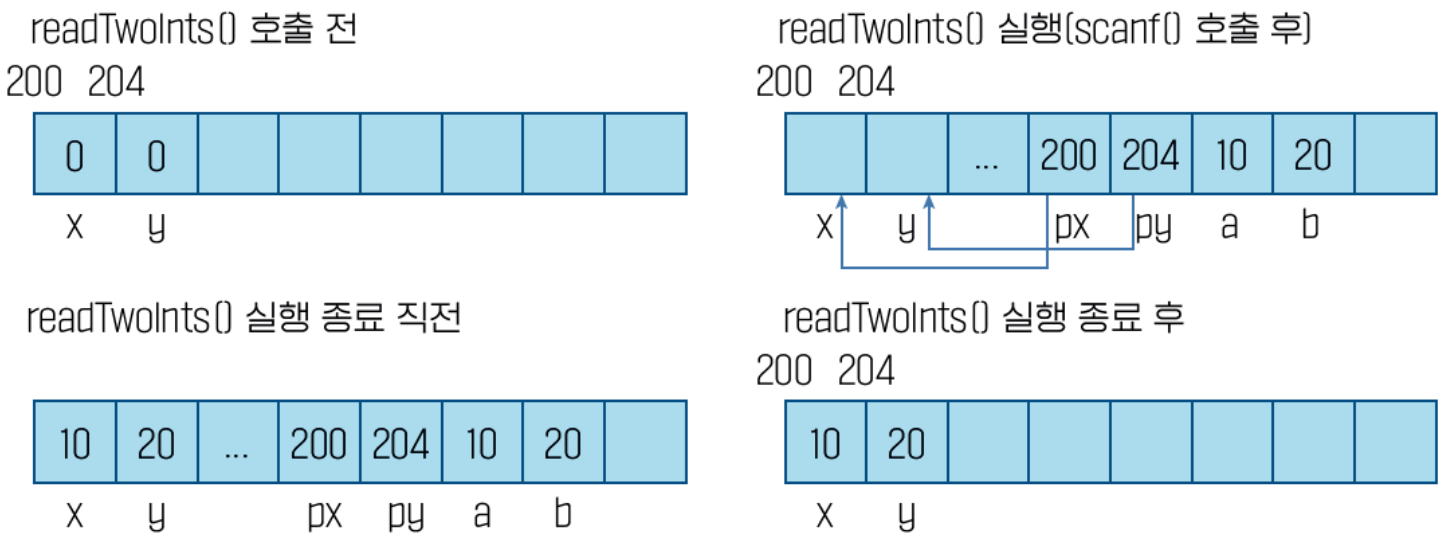

그림 8-13 포인터 매개변수를 이용해서 외부 변숫값을 변경

5. 함수와 포인터

포인터 매개변수로 함수 외부 변숫값 변경

- px와 py의 값을 전달해서 코드를 간단하게 작성

코드 8-33

```
1    void readTwoInts(int* px, int* py)
2    {
3        printf("정수 두 개를 입력하세요: ");
4        scanf("%d %d", px, py);
5    }
```

5. 함수와 포인터

■ 포인터 매개변수로 함수에 큰 데이터 효율적으로 전달하기

- 대용량 데이터를 복사하는 경우와 포인터로 전달하는 코드

코드 8-34 StructPtr.c

```
1   typedef struct {
2       int size;
3       double arr[1000000]; // double형 백만개 데이터
4   } LARGE_DATA;
5
6   void func(LARGE_DATA d) { }    // 대용량 데이터를 값 복사 형태로 전달받음
7   void func2(LARGE_DATA* pd) {} // 대용량 데이터의 시작 주소만 전달받음
8
9   int main(void)
10  {
11      LARGE_DATA data;
12
13      func(data);     // 값 복사
14      func2(&data);   // 시작 주소만 전달
15  }
```

double 백만 개를 포함하는 대용량 구조체 선언
구조체 안에 배열이 포함되어 있으므로 LARGE_DATA는 8 * 1000000 + 4바이트 공간을 사용

구조체를 값 복사 형태로 전달받는 함수를 구현

구조체의 주소를 전달받는 함수를 구현

구조체 변수 선언
백만 개 double 자료형 값을 저장할 수 있는 배열을 생성

전역변수 data를 값으로 복사해서 전달
func() 함수에서는 LARGE_DATA 크기의 공간을 새로 생성하고 값을 복사해야 함

전역변수 data의 주소를 전달
func2() 함수에서는 주소에 해당되는 4~8바이트 공간을 생성하고 값을 복사

함수에 배열 전달

- getSum() 함수를 사용하는 코드

코드 8-35 GetSum.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   int getSum(int* arr, int size) // int getSum(int arr[], int size)로 쓸 수 있음
5   {
6       int sum = 0;
7       for (int i = 0; i < size; i++) {
8           sum += arr[i];
9       }
10      return sum;
11  }
12
13  int main()
14  {
15      int arr[4] = { 1, 2, 3, 4 };
16      int sum = getSum(arr, 4);
17      printf("sum = %d\n", sum);
18      return 0;
19  }
```

정수 배열과 크기를 전달받고 배열에 있는 요소의 합을 계산해서 반환하는 함수

getSum() 함수를 호출해서 배열 요소의 합을 계산한 후 sum에 저장

〈실행 결과〉

```
sum = 10
```

함수에 배열 전달

- getSumOfChars() 함수를 사용하는 코드

코드 8-36 GetSumOfChars.c

```c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

int getSumOfChars(char* str)
{
    int sum = 0;
    char* p = str;

    while (*p != '\0') {
        sum += *p; // sum에 문자 코드(*p)를 더함
        p++;       // p를 다음 문자 위치로 이동시킴
    }
    return sum;
}

int main()
{
    char s[] = { 'h', 'e', 'l', 'l', 'o', '\0' };
    int sum = getSumOfChars(s);
    printf("sum = %d\n", sum);
    return 0;
}
```

문자열을 전달받고 널 문자를 만날 때까지 각 글자를 sum에 더해 합을 구한 다음 반환하는 함수를 구현

hello 문자열을 getSumOfChars() 함수에 전달해 hello 문자열 각 글자 코드의 합을 구함

<실행 결과>

```
sum = 532
```

함수에서 포인터 반환

- getFirstElemGreaterThan() 함수를 사용하는 코드

코드 8-37

```
1   #define SIZE 4
2
3   int arr[] = { 1, 2, 3, 4};
4
5   int* getFirstElementGreaterThan(int n)
6   {
7       for (int i = 0; i < SIZE; i++) {
8           if (arr[i] > n) { return &arr[i]; }
9       }
10      return NULL;
11  }
```

※ 함수에서 포인터 반환하는 코드

➔ 코드 8-38
➔ 실행 결과

2차 방정식의 해를 구하는 함수를 구현하고 사용하는 프로그램

- 근의 공식

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

표 8-4 quadraticEqn( ) 함수의 매개변수와 결괏값에 대한 설명

매개변수 또는 결괏값	설명
a, b, c	2차 방정식을 나타내는 계수
x1, x2	2차 방정식의 근을 반환하는 데 사용될 포인터 변수
결괏값	0 : 오류가 발생한다. b^2-4ac의 값이 음수인 경우 0을 반환한다. x1과 x2로 반환되는 변숫값은 정해지지 않는다. 1 : 계산이 완료된다. b^2-4ac의 값이 0 또는 양수인 경우 1을 반환한다. x1과 x2로 반환되는 변숫값에는 해가 있다.

2차 방정식의 해를 구하는 함수를 구현하고 사용하는 프로그램

- 2차 방정식에 대해 해를 구하거나 구할 수 없다는 내용을 화면에 출력하는 프로그램 작성

- $x^2 - 2x + 1$
- $4x^2 + 6x + 1$
- $5x^2 - 2x + 1$

- 함수는 a, b, c에 해당되는 세 개의 인자를 전달받고
- 함수 외부에서 선언된 두 개 변수에 해를 저장
- 세 개의 매개변수와 두 개의 포인터 변수를 사용해서 함수 정의
- 2차 방정식의 계수는 a, b, c로 이름 설정
- 해를 전달할 변수는 x1과 x2로 지정
- 모든 변수는 double 형으로 지정

2차 방정식의 해를 구하는 함수를 구현하고 사용하는 프로그램

- 표 8-4를 구현하고 함수를 호출하는 코드

코드 8-39 2차 방정식의 해를 구하는 함수와 사용 코드

```
1   #include <math.h>  // sqrt() 함수 사용
2
3   int quadraticEqn(double a, double b, double c, double* px1, double* x2)
4   {
5       double d = b * b - 4 * a * c;
6       if (d < 0) { return 0; }
7       *x1 = (-b + sqrt(d)) / (2 * a);
8       *x2 = (-b - sqrt(d)) / (2 * a);
9       return 1;
10  }
```

- 제곱근을 계산하는 sqrt() 함수를 사용하기 위해 math.h 헤더 파일을 포함시킴
- quadraticEqn() 함수의 헤더. 계수를 전달받을 a, b, c와 해를 반환할 포인터 변수 px1과 px2를 사용
- b^2-4ac를 계산한 후 변수 d에 저장
- d 값이 음수면 오류 반환
- 해를 구해서 px1과 px2의 메모리 주소가 가리키는 메모리 공간에 저장
- 해를 구했으므로 1을 반환

2차 방정식의 해를 구하는 함수를 구현하고 사용하는 프로그램

- quadraticEqn() 함수를 사용하는 코드

```
double s1;
double s2;
int r = quadraticEqn(1, -2, 1, &s1, &s2);
if (r > 0) { printf("s1 = %f, s2 = %f\n", s1, s2); }
else { printf("quadraticEqn( ) 함수는 허수인 해를 구할 수 없습니다.\n"); }
```

함수에 전달해서 해를 받아올 변수 선언

함수를 호출하면서 해를 받아올 변수의 주소 전달

함수의 결과가 0 이상이면 해를 구했다는 뜻이므로 값을 출력

※ 2차 방정식 해를 구하는 함수 구현하고 사용하는 프로그램 완성

➔ 코드 8-40
➔ 실행 결과

06

상수 포인터

6. 상수 포인터

- 1~4로 초기화된 배열을 함수에 전달해서 다른 값으로 변경하는 코드

코드 8-41 ConstPtr1.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   void printArr(int* parr, int size)
5   {
6       for (int i = 0; i < size; i++) {
7           printf("%d ", parr[i]);
8       }
9       printf("\n");
10  }
11
12  void  changeArr(int* parr, int idx, int newValue)
13  {
14      parr[idx] = newValue;
15  }
16
17  int main()
18  {
19      int arr[] = { 1, 2, 3, 4 };
20      changeArr(arr, 2, 3 * 3);
21      changeArr(arr, 3, 4 * 4);
22      printArr(arr, 4);
23      return 0;
24  }
```

주어진 배열을 출력

주어진 배열 parr에서 인덱스 idx에 해당되는 요소에 newValue를 저장

arr[2]를 9로 변경

arr[3]을 16으로 변경

<실행 결과>

1 2 9 16

6. 상수 포인터

- printArr()의 매개변수를 상수 포인터 형태로 다시 수정한 코드

코드 8-42 ConstPtr2.c

```
1   /* printArr()을 제외한 나머지 코드는 Constptr1.c와 동일함 */
2   void printArr(const int* parr, int size)
3   {
4       parr[0] = 0;
5       for (int i = 0; i < size; i++) {
6           printf("%d ", parr[i]);
7       }
8       printf("\n");
9   }
```

- 비주얼 스튜디오 2022에서 코드 8-42를 컴파일한 결과

```
D:\Books\CBook\Code\08>cl ConstPtr2.c
Microsoft (R) C/C++ Optimizing Compiler Version 19.36.32535 for x86
Copyright (C) Microsoft Corporation.  All rights reserved.

ConstPtr2.c
ConstPtr2.c(5): error C2166: l-value specifies const object
```

07

함수 포인터

7. 함수 포인터

함수 포인터 자료형

함수 이름의 의미와 함수 자료형

- 함수 이름 : 함수 코드가 있는 메모리 주소
- 함수를 구별하는 방법 : 함수의 반환 자료형 및 매개변수를 확인
- 함수나 매개변수의 이름은 중요하지 않음

코드 8-45

```
1    int add(int a, int b) { return a + b; }
2    int subtract(int x, int y) { return x - y; }
3    int increase(int a) { return a + 1; }
```

7. 함수 포인터

함수 포인터 자료형

함수 포인터 자료형(함수 포인터)

- 함수 포인터 변수 선언

```
반환_자료형 (*함수_포인터_변수_이름)(매개_변수_리스트);
```

- int 자료형 인자로 포인터 변수 선언

```
int (*calculate)(int a, int b);
```

- 함수 포인터 변수 이름 주위에 괄호가 없을 때

```
int *calculate(int a, int b); // int*를 반환하는 함수 원형
```

7. 함수 포인터

함수 포인터 자료형

함수 포인터 자료형(함수 포인터)

- calculate 변수에 add()와 subtract() 함수 저장

```
calculate = add;        // calculate( ) 변수에 add( ) 함수의 주소를 저장
calculate = subtract;   // calculate( ) 변수에 subtract( ) 함수의 주소를 저장
```

- subtract( ) 함수를 호출하는 코드

```
int result = calculate(5, 3); // 5 - 3의 결과를 2에 저장
```

※ 함수 포인터 자료형 코드

➔ 코드 8-46
➔ 실행 결과

7. 함수 포인터

함수 포인터 자료형

함수 포인터 자료형 선언

- 함수 포인터 변수를 선언에 typedef 붙이기

```
typedef int (*CalculateFuncType)(int a, int b);
```

- 매개변수 이름 생략

```
typedef int (*CalculateFuncType)(int, int);
```

- 함수를 매개변수로 선언하는 함수

```
CalculateFuncType calc1; // Calculate 함수 포인터 자료형으로 변수 calc1을 선언
```

※ 함수 포인터 자료형 선언 코드

➔ 코드 8-47
➔ 실행 결과

7. 함수 포인터

함수 포인터 자료형

- 함수를 다른 함수에 인자로 전달
  - 함수를 매개변수로 선언하는 함수

코드 8-48

```
1    typedef int (*CalculateFuncType)(int a, int b);
2
3    void calculator(CalculateFuncType calc, int x, int y)
4    {
5        int result = calc(x, y);
6        printf("result = %d\n", result);
7    }
```

7. 함수 포인터

■ 함수 포인터 자료형

▪ 함수를 다른 함수에 인자로 전달

- calculator() 함수에 add()와 subtract() 함수를 전달 가능

```
calculator(add, 2, 3);      // result = 5 출력
calculator(subtract, 5, 2); // result = 3 출력
```

- 직접 매개변수를 함수 포인터로 지정하는 것도 가능

```
void calculator(int (*calc)(int a, int b), int x, int y) { // 코드 }
```

08

void 포인터 자료형

8. void 포인터 자료형

- 자료형이 정해지지 않은 특별한 포인터 자료형

- 세 가지 특징
  - void 포인터로 선언된 변수는 어떤 자료형의 포인터 값이라도 저장 가능
  - 매개변수를 void 포인터로 선언하면, 함수를 호출할 때 어떤 자료형의 포인터라도 전달 가능
  - void 포인터로 선언된 변수는 간접 참조를 통해 값을 저장하거나 사용 불가

8. void 포인터 자료형

- void 포인터 변수를 사용하는 코드

코드 8-49 VoidPtr1.c

```
1   #define _CRT_SECURE_NO_WARNINGS
2   #include <stdio.h>
3
4   void printAddress(void* p)
5   {
6       printf("주소 출력: %p\n", p);
7   }
8
9   int main()
10  {
11      int n = 3; // 정수형 변수 선언
12      double x = 4.3; // 실수 변수 선언
13
14      int* pn = &n;
15      double* px = &x;
16
17      void* p = pn;  // (void*) pn으로 쓰지 않음
18      pn = (int*) p;
19      // *p = 3;      // void*는 자료형이 없기 때문에 오류가 발생함
20
21      printAddress(pn); // printAddress(&n);
22      printAddress(px); // printAddress(&x);
23  }
```

포인터 값인 메모리 주소를 전달받고 화면에 출력하는 함수 구현

int 형 포인터 변수를 선언하고 int 형 변수 주소 저장

double 형 포인터 변수를 선언하고 double 형 변수 주소 저장

void 포인터 변수를 선언하고 정수형 포인터 값을 저장(형 변환 안 해도 됨)

void 포인터 값을 int 형 포인터 변수에 저장
이때 반드시 강제 형 변환을 통해 저장하는 포인터 변수 자료형으로 변경해야 함

void 포인터는 간접 참조가 되지 않으므로 주석 처리, 주석을 풀면 오류가 발생

각각 pn과 px에 저장되어 있는 메모리 주소를 printAddress() 함수에 전달해서 화면에 출력
&n과 &x를 printAddress() 함수에 전달한 것과 동일

<실행 결과>

```
주소 출력: 000000ae6efffaf4
주소 출력: 000000ae6efffae8
```

8. void 포인터 자료형

void 포인터와 함수 포인터를 사용하는 프로그램

- void 포인터 한 개를 인자로 전달받기
- 결괏값은 반환하지 않는 함수 포인터를 자료형으로 선언

```
typedef void (*FuncPtr)(void* p);
```

- FuncPtr 함수와 void 포인터를 매개변수로 가지는 함수 구현

코드 8-50

```
1    void doSomething(FuncPtr func, void* ptr)
2    {
3        func(ptr);
4    }
```

8. void 포인터 자료형

void 포인터와 함수 포인터를 사용하는 프로그램

- FuncPtr 형 함수 구현

코드 8-51

```
1   void printDouble(void* p)
2   {
3       double* ptr = (double*) p;
4       printf("%lf\n", *ptr);
5   }
6
7   void printInt(void* p)
8   {
9       int* ptr = (int*) p;
10      printf("%d\n", *ptr);
11  }
```

- doSomething() 함수에 printDouble() 함수와 printInt() 함수 전달

```
double d = 2.3;
int n = 3;

doSomething(printDouble, &d); // (1)
doSomething(printInt, &n);    // (2)
```

※ main( ) 함수를 구현해서 실행한 코드

➔ 코드 8-52
➔ 실행 결과