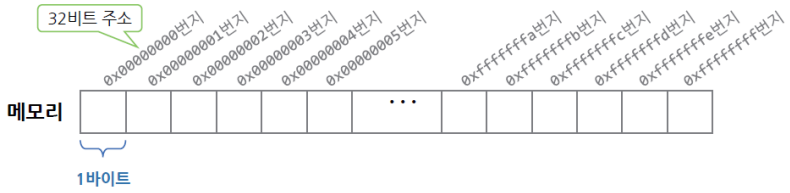
CHAPTER 8
포인터

목차

포인터의 개념 (1)

- 포인터(pointer) : 주소(address)를 저장하는 변수

- 메모리는 연속된 바이트의 모임이다.
  - 각각의 바이트를 구분하기 위해서 주소(번지)를 사용한다.
- 메모리 주소 공간(address space)
  - 메모리 주소가 가질 수 있는 범위
  - 0x00000000번지~0xffffffff번지 사이의 값 (4바이트 크기의 메모리 주소인 경우)



포인터의 개념 (2)

- 포인터는 다른 변수를 가리킨다.

- 포인터는 변수의 이름을 사용할 수 없더라도 주소로 변수에 접근할 수 있는 방법을 제공한다.

포인터의 선언 (1)

형식 데이터형 *변수명;
데이터형 *변수명 = 초기값;

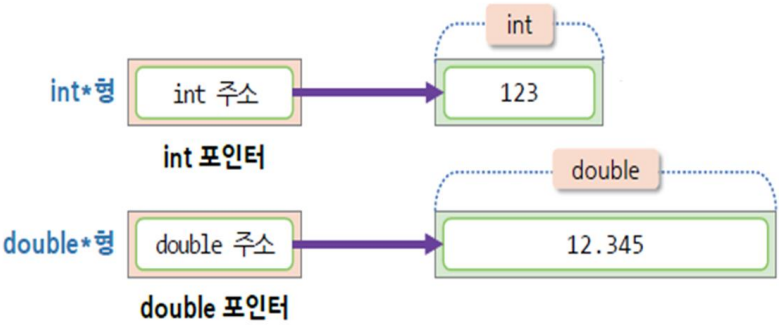
사용예

```
int *p;
double x;
double *pd = &x;
char *ptr = NULL;
```

- 어떤 형의 변수를 가리키는지에 따라 포인터의 데이터형이 결정된다.
  - int 포인터는 int*형의 포인터이며, int 변수를 가리킨다.

포인터의 선언 (2)

- 데이터형에 관계없이 포인터의 크기는 항상 같다.
  - 포인터(주소)의 크기는 플랫폼에 의해서 결정된다.
  - 32비트 플랫폼에서 포인터의 크기는 4바이트이다.



예제 8-1: 포인터의 바이트 크기 구하기

```
01  #include <stdio.h>
03  int main(void)
04  {
05      int *pi;
06      double *pd;
07      char *pc;
09      printf("sizeof(pi) = %d\n", sizeof(pi));
10      printf("sizeof(pd) = %d\n", sizeof(pd));
11      printf("sizeof(pc) = %d\n", sizeof(pc));
13      printf("sizeof(int*) = %d\n", sizeof(int*));
14      printf("sizeof(double*) = %d\n", sizeof(double*));
15      printf("sizeof(char*) = %d\n", sizeof(char*));
16  }
```

실행 결과

```
sizeof(pi) = 4
sizeof(pd) = 4
sizeof(pc) = 4
sizeof(int*) = 4
sizeof(double*) = 4
sizeof(char*) = 4
```


포인터의 초기화

- 주소 구하기(address-of) 연산자
  - 포인터를 초기화하려면 주소가 필요하다.
  - 변수의 주소를 구하려면 &를 이용한다.

```
int a = 10;
int *p = &a;
```

- 널 포인터(NULL)
  - 표준 C 라이브러리에 0으로 정의된 매크로 상수
  - 포인터를 어떤 변수의 주소로 초기화할지 알 수 없으면 NULL로 초기화한다.
  - NULL은 메모리 0번지가 아니라 포인터가 어떤 변수도 가리키지 않는다는 뜻이다.

```
int *q = NULL;
int *r = 0;
```

어떤 변수도 가리키지 않는다.

NULL 대신 0을 사용할 수도 있다.

예제 8-2: 포인터의 선언 및 초기화

```c
#include <stdio.h>

int main(void)
{
    int a = 10;
    int *p = &a;
    int *q = NULL;
    int *r = 0;

    printf("p = %p\n", p);
    printf("q = %p\n", q);
    printf("r = %p\n", r);
}
```

주소를 16진수로 출력한다.

실행 결과

```
p = 00FFF948
q = 00000000
r = 00000000
```


포인터의 표기

- '*type* 주소'는 *type*형 변수의 주소, *type* 포인터라는 뜻이다.
- 화살표로 어떤 변수를 가리키는지 나타낸다.

```c
int a = 10;
int *p = &a;
```

주소 구하기 연산자

- & 연산자는 반드시 변수와 함께 사용해야 하며, 상수나 수식에는 사용할 수 없다.

```c
p = &10;
p = &(a + 1);
p = &MAX;
```

리터럴 상수의 주소는 구할 수 없다.

수식의 주소는 구할 수 없다.

매크로 상수의 주소는 구할 수 없다.

역참조 연산자 (1)

- 포인터 앞에 *를 쓰면 포인터가 가리키는 변수의 값을 읽어오거나 변경할 수 있다.

```c
*p = 20;
printf("*p = %d\n", *p);
```

- 간접 참조(indirection) 연산자
  - * 연산의 결과는 포인터가 가리키는 변수이다.

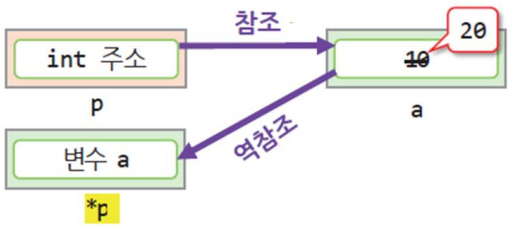
- 역참조 연산자를 이용하면 변수의 이름을 모르더라도 주소로 변수에 접근할 수 있다.

역참조 연산자 (2)

- 역참조 연산자를 사용하면 포인터가 가리키는 변수가 대신 사용된다.
  - p가 a를 가리킬때, *p 대신 a가 사용되는 것처럼 처리된다.

```
int a = 10;
int *p = &a;
```


```
*p = 20;
printf("%d", *p);
```



예제 8-3: 포인터의 사용

```
01  #include <stdio.h>
03  int main(void)
04  {
05      int a = 10;
06      int* p = &a;
08      printf(" a = %d\n", a);
09      printf("&a = %p\n", &a);
11      printf(" p = %p\n", p);
12      printf("*p = %d\n", *p);
13      printf("&p = %p\n", &p);
15      *p = 20;
16      printf("*p = %d\n", *p);
17  }
```

포인터에도 주소가 있다.

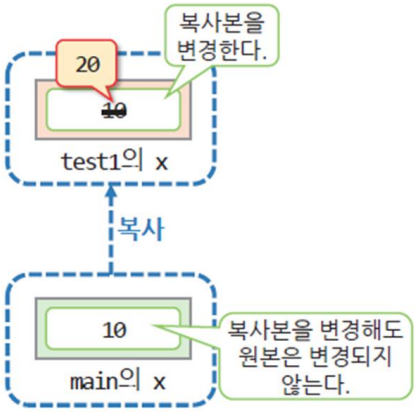
실행 결과

```
a = 10
&a = 004FF71C
 p = 004FF71C
*p = 10
&p = 004FF710
*p = 20
```


포인터의 용도 (1)

- 인자를 매개변수로 복사해서 전달하는 경우에는 매개변수를 변경해도 원본은 변경되지 않는다.

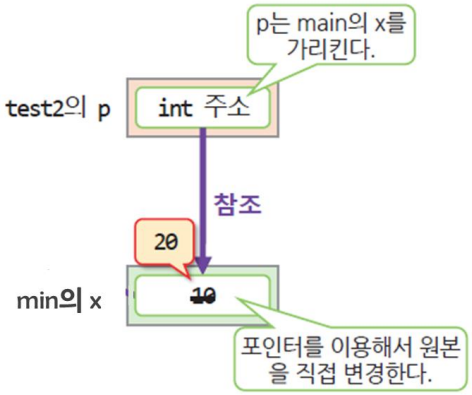
```
void test1(int x)
{
  x = 20;
}

int main(void)
{
  int x = 10;
  test1(x);
  printf("x = %d\n", x);
}
```



포인터의 용도 (2)

- 변수를 직접 사용할 수 없을 때 포인터를 이용해서 주소로 접근할 수 있다.

```
void test2(int *p)
{
  *p = 20;
}

int main(void)
{
  int x = 10;
  test2(&x);
  printf("x = %d\n", x);
}
```



예제 8-4: 포인터가 필요한 경우 (1)

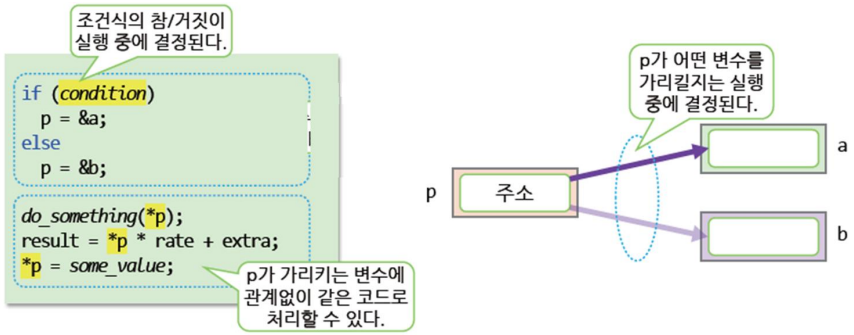
```c
#include <stdio.h>

void test1(int x)
{
    x = 20;
}

void test2(int *p)
{
    *p = 20;
}

int main(void)
{
    int x = 10;
    test1(x);
    printf("test1 호출 후 x = %d\n", x);

    test2(&x);
    printf("test2 호출 후 x = %d\n", x);
}
```

실행 결과

```
test1 호출 후 x = 10
test2 호출 후 x = 20
```


포인터의 용도 (3)

- 여러 변수에 대하여 공통의 코드를 작성할 수 있다.
  - 포인터가 어떤 변수를 가리킬지 미리 알 수 없는 경우에도 포인터가 가리키는 변수로 어떤 작업을 수행하도록 코드를 작성할 수 있다.



포인터 사용 시 주의 사항 (1)

- 어떤 변수도 가리키지 않는 포인터는 널 포인터로 만든다.
  - 포인터는 항상 특정 변수를 가리키거나 널 포인터로 만든다.

```c
int* q = NULL;
printf("%d", *q);
```

널 포인터로 역참조 연산을 수행하면 프로그램이 죽는다.

```c
if(q != NULL)
    printf("%d", *q);
```

q가 널 포인터가 아닌 경우에만 역참조 연산한다.

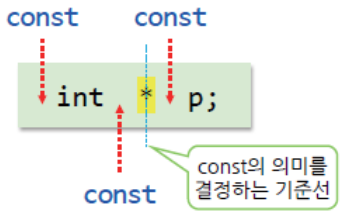
포인터 사용 시 주의 사항 (2)

- 포인터와 포인터가 가리키는 변수의 데이터형이 같아야 한다.
  - 포인터형이 일치하지 않으면 컴파일 경고가 발생하며, 컴파일 경고를 무시하고 실행하면 실행 에러가 발생한다.

```c
int x = 1;
double *pd = NULL;

pd = &x;      // 컴파일 경고
*pd = 12.34; // 실행 에러
```

const 포인터

- 포인터도 const 변수로 선언할 수 있다.
  - const의 위치에 따라 const 포인터의 의미가 달라진다.
  - 세 가지 const 포인터 중에서 읽기 전용 포인터만 주로 사용된다.




```
const int * p;
```
읽기 전용 포인터

```
int * const p;
```
특정 변수 전용 포인터

```
const int * const p;
```
읽기 전용이면서 특정 변수 전용 포인터

const *type* * *variable*;

- 포인터가 가리키는 변수의 값을 변경할 수 없다.
  - 포인터가 가리키는 변수를 const 변수인 것처럼 사용한다.
  - 읽기 전용 포인터

```
int a = 10, b = 20;
const int *p1 = &a;

printf("*p1 = %d\n", *p1);
*p1 = 100;
```
p1이 가리키는 변수의 값을 변경할 수는 없다.

- 포인터 자신의 값(포인터에 저장된 주소)은 변경할 수 있다.

```
p1 = &b;
```
p1이 다른 변수를 가리킬 수 있다.

type * const *variable*;

- 포인터 자신의 값(포인터에 저장된 주소)을 변경할 수 없다.
  - 특정 변수의 전용 포인터
  - 선언 시 특정 변수의 주소로 초기화해야 하며, 초기화 후에 다른 변수를 가리킬 수 없다.

```
int *const p2 = &a;
p2 = &b;
```
p2는 a 전용 포인터

- 포인터가 가리키는 변수의 값은 변경할 수 있다.

```
*p2 = 100;
```
p2가 가리키는 변수의 값을 변경할 수 있다.

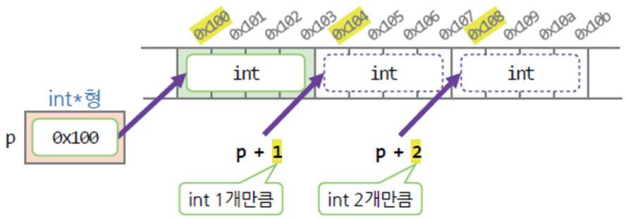
const *type* * const *variable*;

- 읽기 전용 포인터이면서 특정변수의 전용 포인터
  - 반드시 초기화해야 한다.
  - 포인터로는 가리키는 변수의 값을 변경할 수 없다.
  - 포인터 자신의 값(포인터에 저장된 주소)도 변경할 수 없다.

```
const int * const p3 = &a;
*p3 = 100;
p3 = &b;
```

포인터와 +, - 연산 (1)

- p+N 연산의 결과는 p가 가리키는 데이터형을 N개만큼 더한 주소이다.

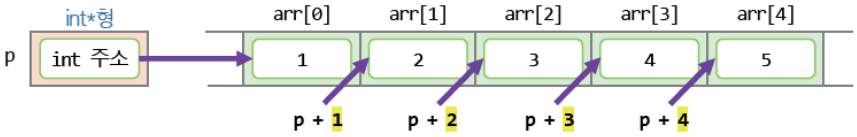


포인터와 +, - 연산 (2)

- 포인터 p가 배열 arr를 가리킬 때 *(p+i)는 arr[i]를 의미한다.

```
int arr[5] = { 1, 2, 3, 4, 5 };
int *p = &arr[0];
```

p는 arr[0]을 가리킨다.



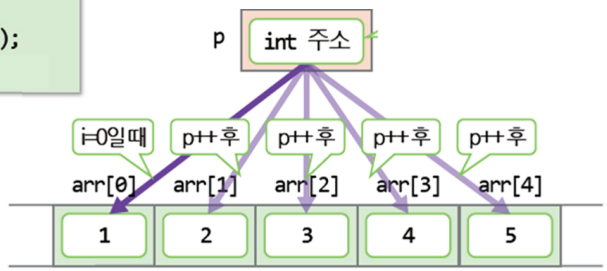
`p + i == &arr[i]`

`*(p + i) == arr[i]`

포인터와 ++, -- 연산

- p++이나 ++p는 p가 가리키는 데이터형 1개 크기만큼 주소를 증가시킨다.

```
int arr[5] = { 1, 2, 3, 4, 5 };
int *p = &arr[0];

for (i = 0; i < 5; i++, p++)
{
    printf("p= %p, ", p);
    printf("*p = %d\n", *p);
}
```



예제 8-7: 포인터와 증감 연산의 의미

```
01  #include <stdio.h>
02
03  int main(void)
04  {
05      int arr[5] = { 1, 2, 3, 4, 5 };
06      int *p = &arr[0];
07      int i;
08
09      for (i = 0; i < 5; i++, p++)
10      {
11          printf("p= %p, ", p);
12          printf("*p = %d\n", *p);
13      }
14  }
```

실행 결과

```
p= 0102FBD8, *p = 1
p= 0102FBDC, *p = 2
p= 0102FBE0, *p = 3
p= 0102FBE4, *p = 4
p= 0102FBE8, *p = 5
```

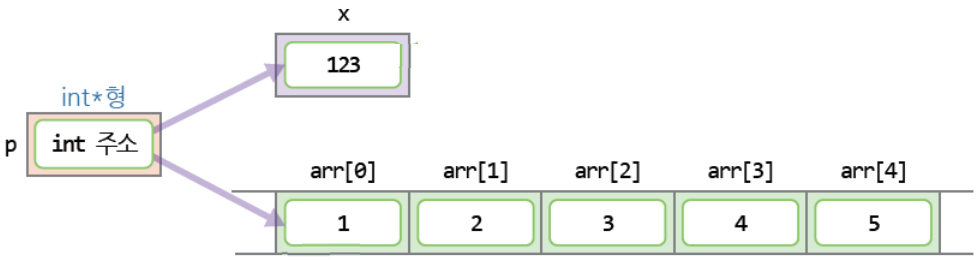
배열처럼 사용되는 포인터 (1)

- 배열 원소를 가리키는 포인터
  - *type**형의 포인터는 *type*형의 변수 또는 *type*형 배열의 원소를 가리킬 수 있다.

```
int x = 123;
int *p = &x;
```

```
int arr[5] = {1, 2, 3, 4, 5};
int *p = &arr[0];
```

int 배열의 원소를 가리킬 수 있다.



배열처럼 사용되는 포인터 (2)

```
int* p = arr;
```

p에 arr 배열의 시작 주소 (&arr[0])를 저장한다.

- p[i]는 항상 *(p+i)으로 처리된다.

```
*(p + i)  ==  p[i]
```


예제 8-8: 포인터를 배열처럼 사용하는 경우

```
01  #include <stdio.h>
02
03  int main(void)
04  {
05      int arr[5] = { 1, 2, 3, 4, 5 };
06      int *p = arr;
07      int i;
08
09      for (i = 0; i < 5; i++)
10          printf("p[%d] = %d\n", i, p[i]);
11  }
```

실행 결과

```
p[0] = 1
p[1] = 2
p[2] = 3
p[3] = 4
p[4] = 5
```


포인터처럼 사용되는 배열

- 배열 이름은 배열의 시작 주소이므로 배열 이름을 포인터처럼 사용할 수 있다.
  - arr[i] 대신 *(arr+i)를 사용할 수 있다.

```
arr[i]  ==  *(arr + i)
```

int 변수

배열과 포인터의 비교 (1)

- 배열 이름은 특정 변수 전용 포인터로 볼 수 있다.
  - 배열 이름(배열의 시작 주소)에 다른 주소를 대입하거나 배열 이름으로 증감 연산을 할 수 없다.
  - 포인터는 값을 변경할 수 있다.

```
int x[5] = { 1, 2, 3, 4, 5 };
int y[5] = { 0 };

x = y;   //컴파일 에러
x++;     //컴파일 에러
```

```
int x[5] = { 1, 2, 3, 4, 5 };
int y[5] = { 0 };
int *p = x;

p = y;   // OK
p++;     // OK
```


배열과 포인터의 비교 (2)

- sizeof(배열명)는 배열 전체의 바이트 크기를 구하지만, sizeof(포인터명)은 포인터 변수의 크기를 구한다.

```
int x[5];
int *p = x;
printf("x의 크기 = %d\n", sizeof(x));
printf("p의 크기 = %d\n", sizeof(p));
```

x 배열 전체의 크기 이므로 20바이트

포인터 p의 크기 이므로 4바이트

함수의 인자 전달 방법

구분	특징
값에 의한 호출 (값 호출)	• 인자를 매개변수로 복사해서 전달한다. • 함수 안에서 매개변수(복사본)를 변경해도 인자(원본)은 변경되지 않는다.
참조에 의한 호출 (참조 호출)	• 인자의 주소를 포인터형의 매개변수로 전달한다. • 함수 안에서 매개변수(참조)가 가리키는 인자(원본)을 변경할 수 있다.

값에 의한 호출

- 인자를 매개변수로 복사해서 전달한다.
  - 함수의 매개변수는 함수가 호출될 때 생성되는 지역 변수로, 인자의 값으로 초기화된다.

값에 의한 호출

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

int main(void)
{
    int a = 1, b = 2;
    swap(a, b);
}
```

int y = b;

int x = a;

swap 함수는 값에 의한 호출로는 구현할 수 없다.

참조에 의한 호출 (1)

- 변수에 대한 참조를 전달한다.
  - 함수 안에서 직접 변수를 변경할 수 있다.
  - C에서는 포인터를 이용해서 참조에 의한 호출을 처리한다.

- 매개변수는 인자를 가리키는 포인터로 선언한다.

```
void swap(int* px, int* py);
```

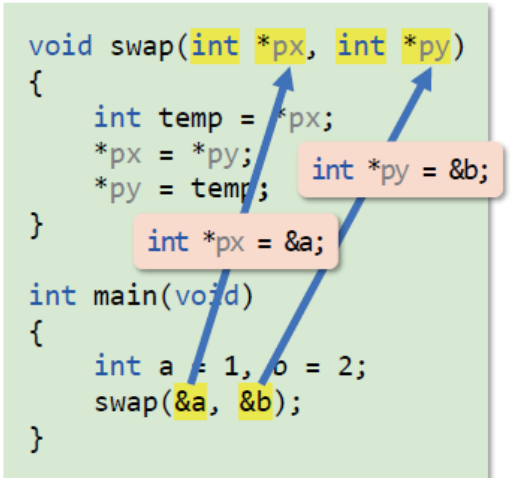
- 함수를 호출할 때는 인자로 전달하려는 변수의 주소를 전달한다.

```
swap(&a, &b);
```


참조에 의한 호출 (2)

- 함수를 정의할 때는 포인터형의 매개변수를 역참조해서 매개변수가 가리키는 변수에 접근한다.

참조에 의한 호출



예제 8-9: swap 함수의 구현 (1)

```
01  #include <stdio.h>
02  void swap(int* px, int* py);
03
04  int main(void)
05  {
06      int a = 1, b = 2;
07
08      printf("a = %d, b = %d\n", a, b);
09      swap(&a, &b);
10      printf("a = %d, b = %d\n", a, b);
11  }
12
13  void swap(int* px, int* py)
14  {
15      int temp = *px;
16      *px = *py;
17      *py = temp;
18  }
```

실행 결과

```
a = 1, b = 2
a = 2, b = 1
```


매개변수의 역할에 따른 인자 전달 방법 결정

구분	역할	인자 전달 방법
입력 매개변수	• 함수 안에서 값이 사용될 뿐 변경되지 않음. int add(int x, int y);	값에 의한 호출
출력 매개변수	• 함수 안에서 값이 사용되지는 않고 함수 리턴 전에 변경됨 void get_sum_product(int x, int y, int *sum, int *product);	참조에 의한 호출
입출력 매개변수	• 함수 안에서 값이 사용도 되고, 함수 리턴 전에 변경도 됨 void swap(int* px, int* py);	

- 함수의 처리 결과가 2개 이상인 경우에 출력 매개변수를 사용한다.

함수의 처리 결과를 출력 매개변수로 전달하는 방법 (1)

① 함수의 원형을 정할 때는, 출력 매개변수를 포인터로 선언한다.
- 처리 결과를 저장할 변수를 가리키는 포인터형으로 선언한다.

```
void get_sum_product(int x, int y, int *psum, int *pproduct);
```

② 함수를 호출할 때는, 처리 결과를 받아올 변수의 주소를 전달한다.

```
int sum, product;
get_sum_product(123, 456, &sum, &product);
```

처리 결과를 받아올 변수를 준비한다.

함수의 처리 결과를 출력 매개변수로 전달하는 방법 (2)

③ 함수를 정의할 때는, 포인터형의 매개변수가 가리키는 곳에 처리 결과를 저장한다.

```
void get_sum_product(int x, int y, int *psum, int *pproduct)
{
    *psum = x + y;
    *pproduct = x * y;
}
```

psum이 가리키는 변수에 합을 저장한다.

pproduct가 가리키는 변수에 곱을 저장한다.

예제 8-10: 출력 매개변수를 갖는 함수

```
01  #include <stdio.h>
03  void get_sum_product(int x, int y, int *psum, int *pproduct);
05  int main(void)
06  {
07      int sum, product;
10      get_sum_product(123, 456, &sum, &product);
11      printf("sum = %d, product = %d\n", sum, product);
12  }
14  void get_sum_product(int x, int y, int *psum, int *pproduct)
15  {
17      *psum = x + y;
18      *pproduct = x * y;
19  }
```

실행 결과

```
sum = 579, product = 56088
```


배열의 전달 (1)

- 배열은 항상 포인터로 전달한다.
  - 배열을 값에 의한 호출로 전달하면 배열 전체를 복사해야 하므로 시간적, 공간적 성능 저하가 발생한다.

```
void print_array(int arr[], int size);
void print_array(int *arr, int size);
```

두 문장은 항상 같은 뜻이다.

- 배열의 크기는 별도의 매개변수로 받아와야 한다.
  - *arr*는 포인터이므로 *arr*로 배열의 크기를 구할 수 없기 때문
- 함수 호출 시 배열을 전달하려면, 배열 이름, 즉 배열의 시작 주소를 전달한다.

```
int x[SIZE] = { 10, 20, 30, 40, 50 };
print_array(x, SIZE);
```

배열의 시작 주소 x를 arr로 전달한다.

배열의 전달 (2)

- 함수 안에서는 포인터형의 매개변수를 배열처럼 사용한다.

```
void print_array(int *arr, int size)
{
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
}
```

arr[i]는 *(arr+i)로 처리된다.

- 배열이 입력 매개변수일 때는 읽기 전용 포인터로 선언하는 것이 좋다.
  - const 포인터로 선언하면 함수 안에서 arr 가 가리키는 배열의 원소를 변경할 수 없다.

```
void copy_array(const int *source, int *target, int size)
```

입력 매개변수

출력 매개변수

예제 8-11: 배열을 매개변수로 갖는 함수 (1)

```
01  #include <stdio.h>
02  #define SIZE 5
03
04  void copy_array(const int *source, int *target, int size);
05  void print_array(const int *arr, int size);
06
07  int main(void)
08  {
09      int x[SIZE] = { 10, 20, 30, 40, 50 };
10      int y[SIZE] = { 0 };
11
12      printf("x = ");
13      print_array(x, SIZE);
```


예제 8-11: 배열을 매개변수로 갖는 함수 (2)

```
15      copy_array(x, y, SIZE);
16      printf("y = ");
17      print_array(y, SIZE);
18  }
19
```

```
21  void copy_array(const int *source, int *target, int size)
22  {
23      int i;
24      for (i = 0; i < size; i++)
25          target[i] = source[i];
26  }
```


예제 8-11: 배열을 매개변수로 갖는 함수 (3)

```
28  void print_array(const int *arr, int size)
29  {
30      int  i;
31      for (i = 0; i < size; i++)
32          printf("%d ", arr[i]);
33      printf("\n");
34  }
```

arr는 포인터지만 배열처럼 사용한다.

실행 결과

```
x = 10 20 30 40 50
y = 10 20 30 40 50
```

배열을 함수의 인자로 전달하는 방법 (1)

① 함수 원형을 정할 때, 함수의 매개변수는 배열의 크기를 생략하고 선언하거나 배열 원소에 대한 포인터형으로 선언한다.

```
void print_array(int arr[]);
void print_array(int *arr);
```

② 함수 안에서 배열의 크기가 필요하면 배열의 크기도 매개변수로 받아와야 한다.

```
void print_array(int *arr, int size);
```

③ 배열이 입력 매개변수일 때는 const를 지정한다.

```
void print_array(const int *arr, int size);
```


배열을 함수의 인자로 전달하는 방법 (2)

④ 함수를 호출할 때는 배열 이름을 인자로 전달한다.

```
int x[5] = { 10, 20, 30, 40, 50 };
print_array(x, 5);
```

⑤ 함수를 정의할 때는 매개변수인 포인터를 배열처럼 사용한다.

```
void print_array(int *arr, int size)
{
    int  i;
    for (i = 0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```


2차원 배열을 포인터로

- 배열 포인터는 배열 전체를 가리키는 포인터다
- 2차원 배열을 처리하는 함수의 매개변수로는 배열포인터를 쓴다.

```
int  data[3][4] = { {1, 2, 3, 4}, {11, 12, 13, 14}, {21, 22, 23, 24} };

int   (*p)[4] ;
p = data;

for(i=0; i<3; i++) {
        for(j = 0; j<4; j++)
                printf("%3d", *(p[i]+ j));
        printf("₩n");
}
for(i=0; i<3; i++) {
        for(j = 0; j<4; j++)
                printf("%3d", *(*(p+i)+ j));
        printf("₩n");
}
```


예제1: 포인터

```
/*
   &연산자 : 피연산자의 주소값을 반환하는 연산자
   *연산자 : 포인터가 가리키는 메모리 공간에 접근할 때 사용하는 연산자
*/
 #include <stdio.h>

 int main()
 {
    int num1 = 100, num2 = 100;
    int *p;

    p = &num1;
    *p += 30;

    p = &num2;
    *p -= 20;

    printf("num1: %3d, num2: %3d \n", num1, num2);

    return 0;
  }
```

예제2: 포인터

```c
#include <stdio.h>

int main()
{
    int num = 100;
    int *nP;

    nP = &num;

    printf("변 수  num의  값  : %3d\n", num);
    printf("변 수  num의  주 소  : %#x\n", &num);
    printf("변 수  num의  주 소  : %#x\n", nP);  //%x는  16진 수

    *nP = 50;
    printf("변 수  num의  값  : %3d\n", num);
    return 0;
 }
```


예제3: 포인터3

- 포인터 변수를 이용하여 두 수를 바꾸는 프로그램

```c
#include <stdio.h>

int main()
{
    int num1 = 100, num2 = 200;
    int *ptr1 = &num1;
    int *ptr2 = &num2;
    int *temp;

    *ptr1 += 10;
    *ptr2 -= 10;

    temp = ptr1;
    ptr1 = ptr2;
    ptr2 = temp;

    printf("%3d %3d \n", *ptr1, *ptr2);
    return 0;

 }
```

예제4 : 포인터와 함수

- 포인터 변수를 이용하여 세 수를 바꾸는 프로그램

```c
#include <stdio.h>
void swap(int *p1, int *p2, int *p3)
{
    int temp = *p3;
    *p3 = *p2;
    *p2 = *p1;
    *p1 = temp;
}
int main()
{
    int num1 = 100, num2=200, num3=300;

    //swap(num1, num2, num3);
    swap(&num1, &num2, &num3);
    printf("%3d %3d %3d\n", num1, num2, num3);
    return 0;

 }
```


예제5 : 다차원 배열

0	1	2	3
3	4	5	6
6	7	8	9
9	10	11	12

```c
#include <stdio.h>

int main()
{
    int arr[4][4];
    int i, j;

    for(i=0; i<4; i++)
        for(j =0; j <4; j++)
            arr[i][j] = (i * 3) + j;

    for(i=0; i<4; i++){
        for(j =0; j <4; j++)
            printf("%4d", arr[i][j]);
        printf("\n");
    }
    return 0;
}
```

예제5 : 다차원 배열_포인터

0	1	2	3	4
3	4	5	6	7
6	7	8	9	10
9	10	11	12	13

```c
#include <stdio.h>

int main()
{
    int arr[4][5];
    int i, j;
    int (*p)[5] = arr;

    for(i=0; i<4; i++)
        for(j =0; j <5; j++)
            arr[i][j] = (i * 3) + j;


    for(i=0; i<4; i++){
        for(j =0; j <5; j++)
            printf("%4d", *(p[i]+j));
        printf("\n");
    }

    return 0;
}
```


예제6: call-by-value / call-by-reference

```c
#include <stdio.h>
int sumByValue(int n)
{
   return n + n;
}

void sumByReference(int *p)
{
   *p = *p + *p;
}
int main()
{
   int num= 10;

   printf("sum = %3d\n", sumByValue(num));

   sumByReference(&num );
   printf("sum = %3d\n", num);
   return 0;

 }
```