

4차시 - 포인터(4)

포인터 전달의 효과

◆ 파라미터 전달 효율이 좋다.

- 배열도 포인터 하나 전달하면 끝
- 함수에 다양한 기능을 부가할 수 있다.

◆ 단점 : 위험하다. 누구나 값을 바꿀 수 있기 때문에

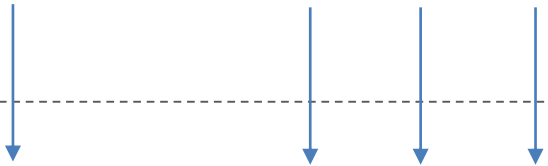
◆ 반환값에 포인터 활용

- 반환값은 하나지만, 포인터를 이용하면 여러 개를 전달 가능
- `void func(int *a, int *b, int *c)`
// a, b, c를 반환용으로 이용
// 각 변수는 caller에서 생성해야 한다.

포인터 전달의 효과

```
int func(int *q, int *w, int *e )  
{  
    *q = 1; *w = 2; *e = 3;  
    return array;  
}
```

```
int main( )  
{  
    int    a, b, c, d;  
  
    d = func(&a, &b, &c);  
}
```



기본 반환값 + 포인터를 이용한 반환값 = 총 4개의 반환값

잘못된 반환값의 활용

```
int *func( )
{
    int array[3] = {1, 3, 5};

    return array;
}

int main( )
{
    int *ptr;

    ptr = func( );
    for(int i = 0; i<3; i++)
        printf("%d\n", *(ptr+i) );
}
```

무엇이 잘못되었을까?

함수의 매개 변수에 붙이는 const

◆ 함수의 매개 변수에 포인터가 전달되면

- 포인터를 통하여 값을 마음대로 바꿀 수 있다.
- 함수가 값을 바꾸는 것을 원치 않는다면?

```
void func(const int *mydata)
// 인수로 넘어온 포인터를 사용하되, 포인터가 가리키는 곳의 내용을 바꿔서는 안 된다.
{
    printf("%d\n", mydata[0]);
    mydata[0] = 5; // 값을 바꾸려는 시도는 컴파일 오류이다.
}

int main(void)
{
    int data[3] = { 1, 2, 3 };

    func(data); // 배열의 값을 전달하기 위해 포인터 사용
}
```

typedef

◆ 새로운 자료형을 선언하는 용도

- 기존 자료형을 알아보기 쉽게 만드는 용도로 사용한다.
- 형식)

typedef (기존 형) (새로운 형);

- 예)

```
typedef    int    STUDENTCODE;  
  
STUDENTCODE std1, std2;    // 훨씬 코드가 명시적이다.
```

- 선언된 자료형은 어디에든 사용 가능하다.

```
void   calcScore(STUDENTCODE code, int score)
```

◆ 복잡한 자료형을 간단히 만드는 용도

- 포인터와 연관하여 보다 자료형을 알아보기 쉽게 만든다.
- 예)

```
typedef    int*    PCODE;  
PCODE     code;    // int *code보다 훨씬 이해가 쉽다  
  
typedef    int***   PPPCODE;  
PPPCODE    pers;   // 알아보기 쉽다.
```

void 포인터

◆ void는 아무것도 없다는 의미이다.

- 함수의 반환 자료형으로 void를 써왔다.
- 아무것도 없기 때문에 무엇이 될 수 있다. (형 변환을 통하여)
- void 포인터를 범용 포인터라고 한다.
- 주소 저장용으로 사용한다.

◆ 포인터 변수 선언 시 void * 형을 쓸 수 있다.

- 형식: void *ptr;
- void 포인터는 **주소만 저장**할 뿐, 주소를 역참조할 수 없다.

```
printf("%d\n", *ptr);           // error  
printf("%d\n", *(int *)ptr);    // OK
```


void 포인터

```
void    func( void );  
void    *func( void );
```

- ◆ 함수의 반환값이 void *(주소) 이다.
- ◆ 형 변환을 통하여 데이터를 사용할 수 있다.
- ◆ void 포인터의 불가능한 것
 - 포인터 연산
- ◆ void * 대신 char *도 많이 사용한다.

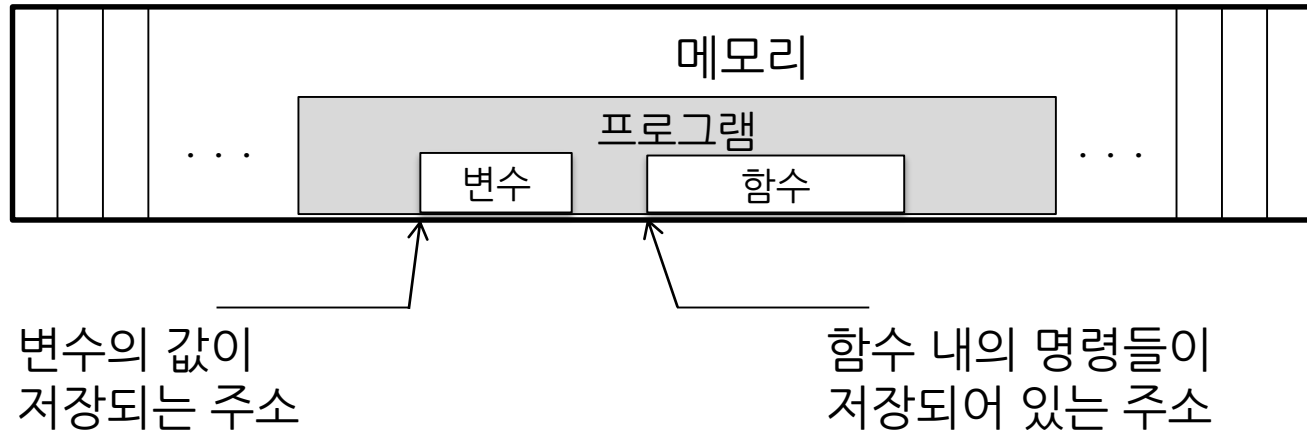
포인터를 쓸 때의 주의할 점

◆ 함수의 인수로 포인터를 쓸 때

- 필요로 하는 데이터와 자료형이 무엇인가
- “일단 보내고 알아서 쓰자”는 잘못이다.

◆ 가능하면 단순화

- 배열을 최대한 활용
- 꼭 그 포인터를 써야 하는가



- ◆ 프로그램이 메모리에 탑재되면, 함수도 주소를 갖는다.
- ◆ 함수 포인터는 잘 사용되지는 않는다.
 - 그러나 사용될 때에는 그만큼 강력하다.

함수 포인터의 선언

변수 포인터	함수 포인터
<pre>int i=0, j, *k; // 포인터 변수 k 선언 k = &i; // 변수의 주소를 k에 할당 j = *k; // 포인터 변수를 이용하여 // i 변수의 값을 가져옴</pre>	<pre><u>int (*fp) (void);</u> // 함수 포인터 fp 선언 fp = printvalue; // 함수 주소를 fp에 할당 fp(); // 포인터로부터 함수 printvalue() 실행 // *fp() 라고 실행하지 않는다.</pre>

```
int printvalue(void)
{
    printf("3\n");
    return 3;
}
```

배열의 이름만 사용하면 배열 포인터 상수이고,
함수의 이름만 사용하면 함수의 포인터 상수이다.

함수 포인터와 원시 함수는 **같은 원형**을 가져야 한다.

함수 포인터의 예

```
int main ( )
{
    int    i = 1 ;
    switch( i ) {
        1 : printone( ); break;
        2 : printtwo( ); break;
        3 : printthree( ); break;
    }
}
```

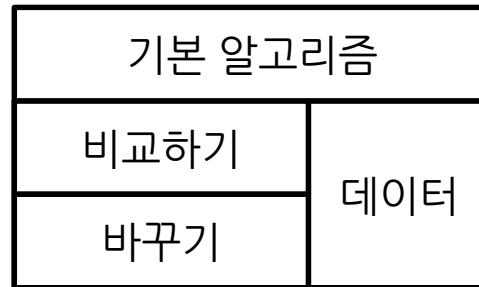
```
int main ( )
{
    int    i = 1;
    void    (*func)( );
    switch( i ) {
        1 : func = printone ; break;
        2 : func = printtwo ; break;
        3 : func = printthree ; break;
    }
    func( );
}
```

함수 포인터의 예 : qsort

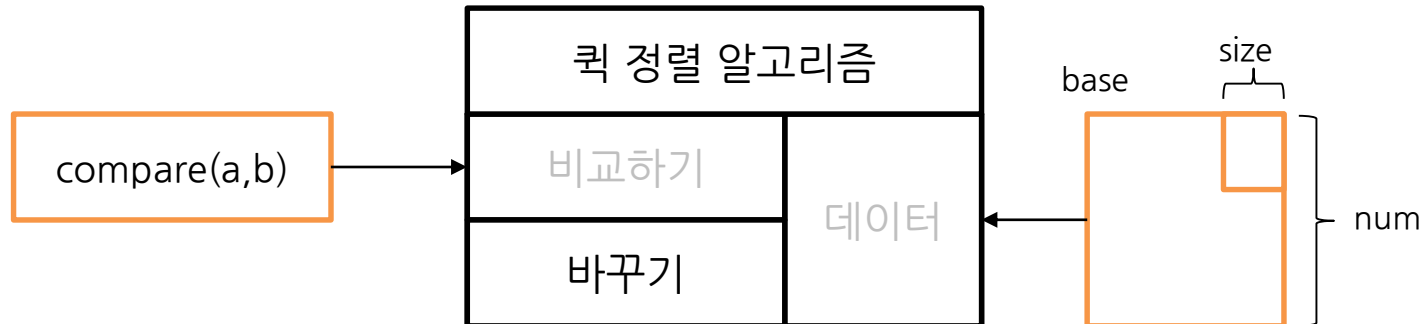
◆ 내장된 정렬용 함수

– 고성능

◆ 정렬이란



◆ qsort() 는



함수 포인터의 예 : qsort

◆ 함수 포인터를 이용해 범용적으로 사용 가능

◆ `void qsort (void* base, size_t num, size_t size,
int (*compar)(const void*,const void*));`

```
// 비교하기 함수
int compare(const void *a, const void *b) // 함수 원형은 그대로 유지
{
    if ( *(int*)a > *(int*)b ) return 1; // 매개변수가 void * 형이므로 형 변환
    else if ( *(int*)a == *(int*)b ) return 0;
    else return -1;
}
```

함수 포인터의 예 : qsort

```
int main( )
{
    int    array[10];    // 초기화는 생략함

    qsort(array, 10 , sizeof(int), compare );
    for (int i = 0; i < 10; i++)
        printf("%d ", array[i]);
}

int compare(const void  *a, const void  *b)
{
    return ( *(int*)a - *(int*)b );
}
```


Quiz)

◆ 10개의 double형 값을 정렬하여 출력하라.

```
int main( )
{
    double    fval[10];    // 초기화는 생략함

    qsort(fval, 10 , sizeof(double), compdouble );
    for (int i = 0; i < 10; i++)
        printf("%lf ", fval[i]);
}

_____ compdouble(          )
{
}
}
```