

# Computer Engineering Programming 2

- 제17장 동적 메모리와 연결 리스트
- 제14장 포인터 활용

Lecturer: JUNBEOM YOO  
jbyoo@konkuk.ac.kr

# 이번 장에서 학습할 내용



- 동적 할당 메모리
- 연결 리스트
  
- 이중 포인터
- 포인터 배열
- 다차원 배열과 포인터
- `main` 함수의 인수

이번 단원에서는 보다 advanced된 기법으로서 동적 메모리의 할당과 다양한 포인터의 활용을 학습하겠습니다.



# 동적 할당 메모리의 개념

- 프로그램이 메모리를 할당 받는 방법
  - 정적(static)
  - 동적(dynamic)
- 정적 메모리 할당
  - 프로그램이 시작되기 전에 미리 정해진 크기의 메모리를 할당받는 것
  - 메모리의 크기는 프로그램이 시작하기 전에 결정

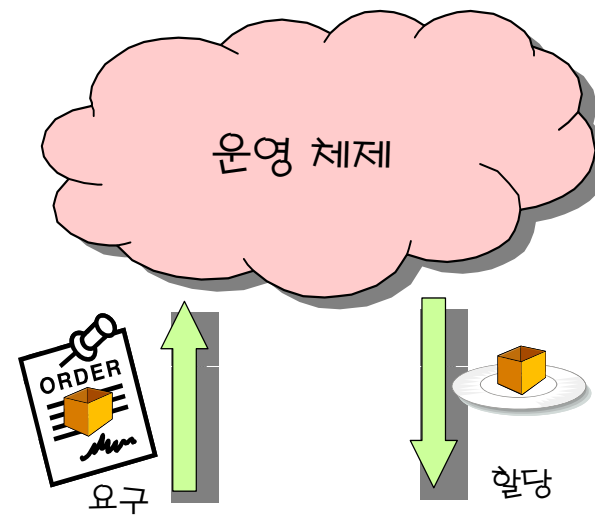
```
int i, j;  
int buffer[80];  
char name[] = "data structure";
```

- 처음에 결정된 크기보다 더 큰 입력이 들어온다면 처리하지 못함
- 더 작은 입력이 들어온다면 남은 메모리 공간은 낭비

# 동적 메모리

- 동적 메모리

- 실행 도중에 동적으로 메모리를 할당 받는 것
- 사용이 끝나면 시스템에 메모리를 반납
- 필요한 만큼만 할당을 받고 메모리를 매우 효율적으로 사용
- malloc() 계열의 라이브러리 함수를 사용



```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *p;
    p = (int *)malloc( sizeof(int) );
    ...
}
```

프로그램

# 동적 메모리 할당의 과정

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
    int *pi;    // 동적 메모리를 가리키는 포인터

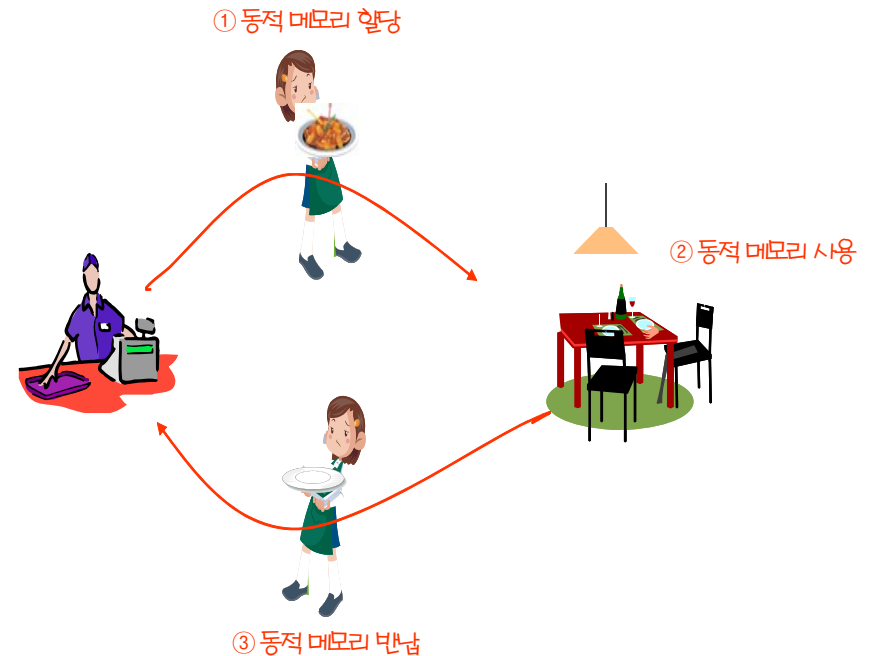
    pi = (int *)malloc(sizeof(int)); // ① 동적 메모리 할당

    if (pi == NULL )        // 반환값이 NULL인지 검사
    {
        printf("동적 메모리 할당 오류\n");
        exit(1);
    }

    *pi = 100; // ② 동적 메모리 사용
    printf("%d\n", *pi);

    free(pi); // ③ 동적 메모리 반납

    return 0;
}
```



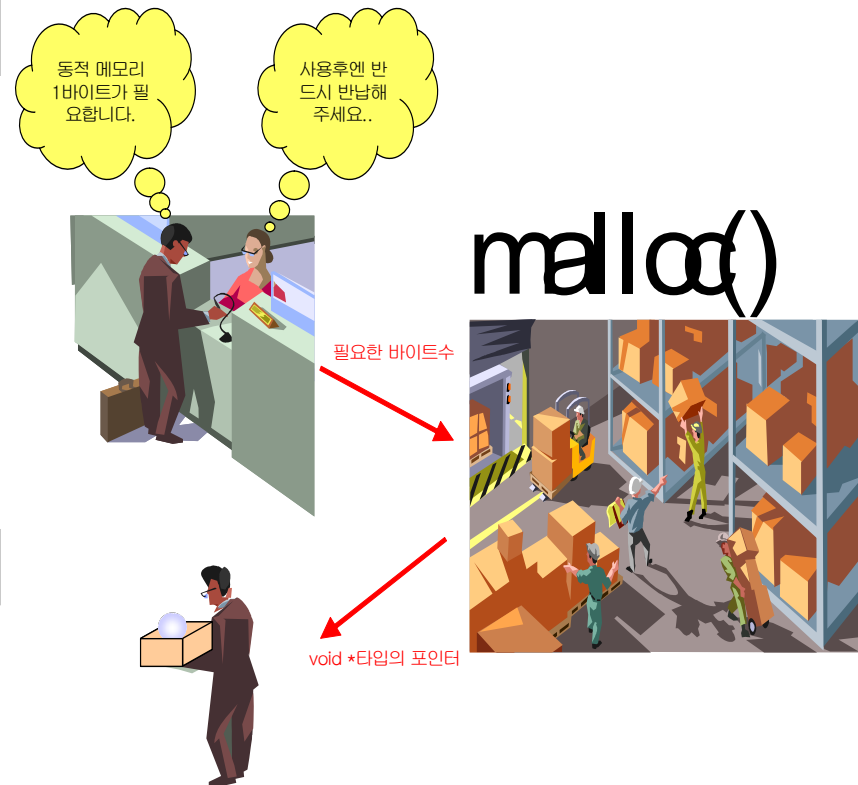
# malloc()과 free()

```
void *malloc(size_t size);
```

- malloc()은 바이트 단위로 메모리를 할당
- size는 바이트의 수
- malloc()함수는 메모리 블록의 첫 번째 바이트에 대한 주소를 반환
- 만약 요청한 메모리 공간을 할당할 수 없는 경우에는 NULL값을 반환

```
void free(void *ptr);
```

- free()는 동적으로 할당되었던 메모리 블록을 시스템에 반납
- ptr은 malloc()을 이용하여 동적 할당된 메모리를 가리키는 포인터



# malloc1.c



```
1. #include <stdio.h>
2. #include <stdlib.h>
3.
4. int main( void )
5. {
6.     char *pc = NULL;
7.
8.     pc = (char *)malloc( sizeof(char) );
9.     if( pc == NULL )
10.    {
11.        printf( "메모리 할당 오류\n" );
12.        exit(1);
13.    }
14.    *pc = 'm';
15.    printf( "*pc = %c\n", *pc );
16.    free( pc );
17.
18.    return 0;
19. }
```



1000 바이트가 할당되었습니다.  
메모리를 반납하였습니다.

# malloc2.c



```
1. // 메모리 동적 할당
2. #include <stdio.h>
3. #include <stdlib.h>

4. int main(void)
5. {
6.     char *pc = NULL;
7.     int i = 0;

8.     pc = (char *)malloc(100*sizeof(char));
9.     if( pc == NULL )
10.    {
11.        printf("메모리 할당 오류\n");
12.        exit(1);
13.    }
14.    for(i=0;i<26;i++)
15.    {
16.        *(pc+i) = 'a'+i;           // 알파벳 소문자를 순서대로 대입
17.    }
18.    *(pc+i) = 0; // NULL 문자 추가

19.    printf("%s\n", pc);
20.    free(pc);

21.    return 0;
22. }
```





# malloc3.c



```
1. #include <stdio.h>
2. #include <stdlib.h>

3. int main(void)
4. {
5.     int *pi;

6.     pi = (int *)malloc(5 * sizeof(int));

7.     if(pi == NULL){
8.         printf("메모리 할당 오류\n");
9.         exit(1);
10.    }

11.    pi[0] = 100;           // *(pi+0) = 100;와 같다.
12.    pi[1] = 200;           // *(pi+1) = 200;와 같다.
13.    pi[2] = 300;           // *(pi+2) = 300;와 같다.
14.    pi[3] = 400;           // *(pi+3) = 400;와 같다.
15.    pi[4] = 500;           // *(pi+4) = 500;와 같다.

16.    free(pi);
17.    return 0;
18. }
```

# malloc4.c



```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <string.h>

4. struct Book {
5.     int number;
6.     char title[10];
7. };

8. int main(void)
9. {
10.     struct Book *p;

11.     p = (struct Book *)malloc(2 * sizeof(struct Book));

12.     if(p == NULL){
13.         printf("메모리 할당 오류\n");
14.         exit(1);
15.     }

16.     p->number = 1;
17.     strcpy(p->title,"C Programming");

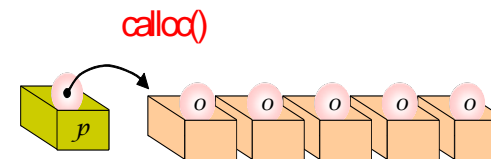
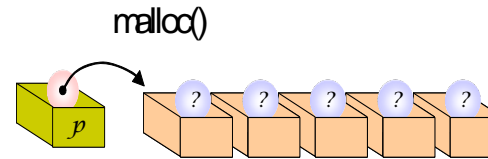
18.     (p+1)->number = 2;
19.     strcpy((p+1)->title,"Data Structure");

20.     free(p);
21.     return 0;
22. }
```

# calloc()과 realloc()

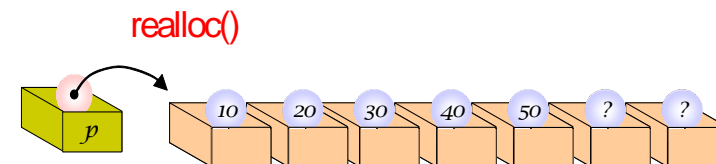
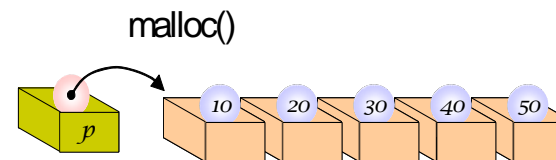
```
void *calloc(size_t n, size_t size);
```

- calloc()은 malloc()과는 다르게 0으로 초기화된 메모리 할당
- 항목 단위로 메모리를 할당
- (예)
- int \*p;
- p = (int \*)calloc(5, sizeof(int));



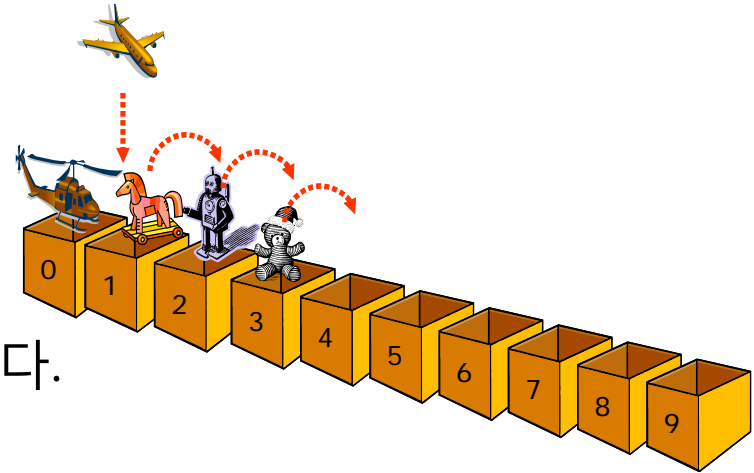
```
void *realloc(void *mемblock, size_t size);
```

- realloc() 함수는 할당하였던 메모리 블록의 크기를 변경
- (예)
- int \*p;
- p = (int \*)malloc(5 \* sizeof(int));
- p = realloc(p, 7 \* sizeof(int));

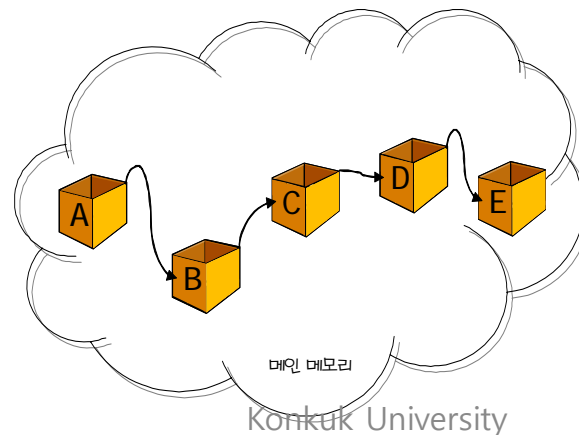


# 연결 리스트

- 배열(array)
  - 장점: 구현이 간단하고 빠르다
  - 단점: 크기가 고정된다.
  - 중간에서 삽입, 삭제가 어렵다.

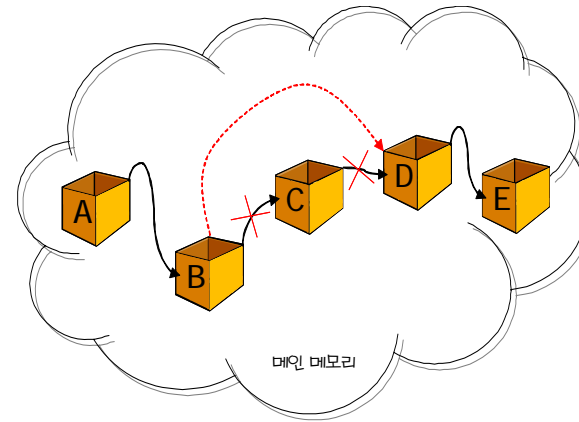
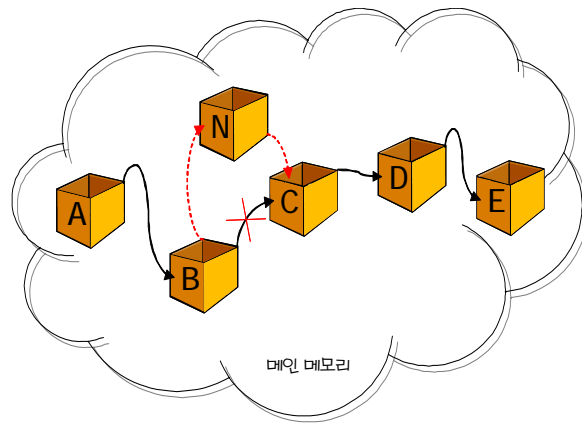


- 연결 리스트(linked list)
  - 각각의 원소가 포인터를 사용하여 다음 원소의 위치를 가리킨다.



# 연결 리스트의 장단점

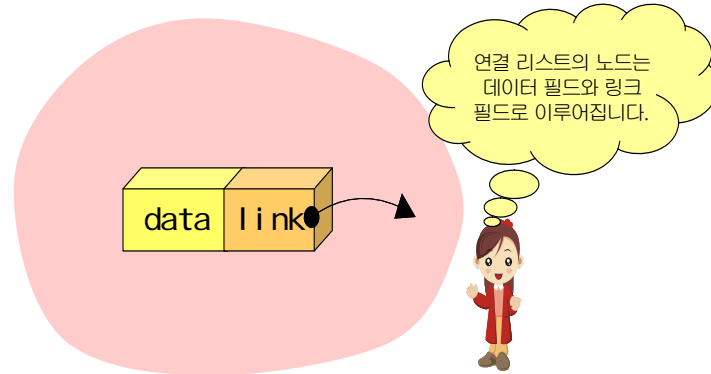
- 중간에 데이터를 삽입, 삭제하는 경우



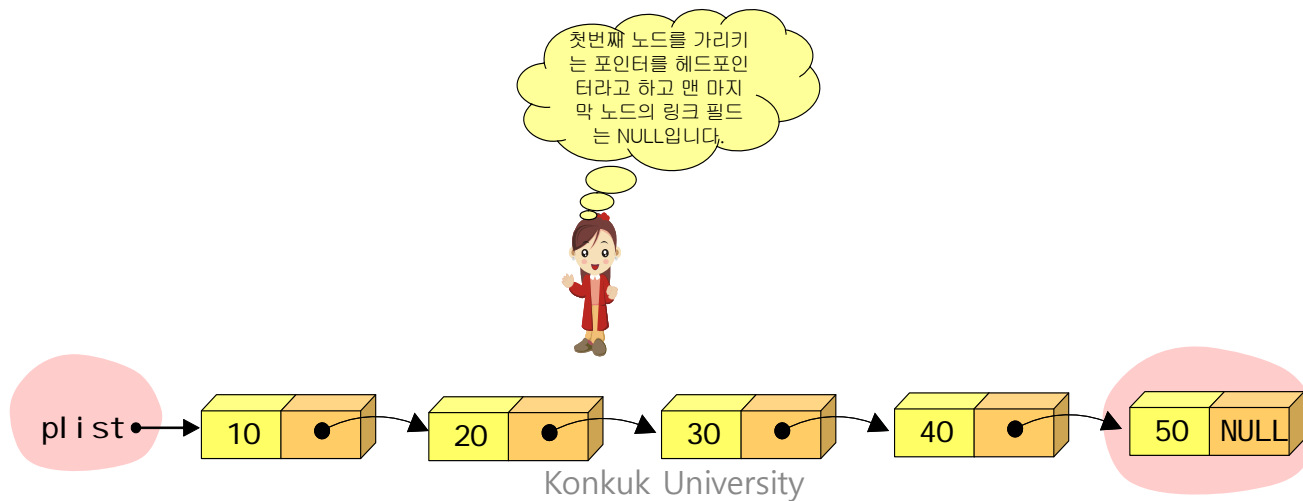
- 데이터를 저장할 공간이 필요할 때마다 동적으로 공간을 만들어서 쉽게 추가
- 구현이 어렵고 오류가 나기 쉽다.

# 연결 리스트의 구조

- 노드(node) = 데이터 필드(data field)+ 링크 필드(link field)



- 헤드 포인터(head pointer): 첫번째 노드를 가리키는 포인터



# 자기 참조 구조체

- 자기 참조 구조체(**self-referential structure**)는 특별한 구조체로서 구성 멤버 중에 같은 타입의 구조체를 가리키는 포인터가 존재하는 구조체

```
// 데이터의 정의
typedef struct data {
    int id;
    char name[20];
    char phone[12];
} DATA;

// 노드의 정의
typedef struct NODE {
    DATA data;
    struct NODE *link;
} NODE;
```

# 간단한 연결 리스트 생성

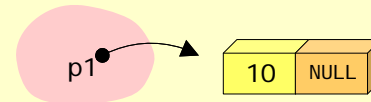
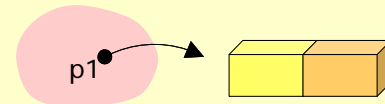
```
NODE *p1;  
p1 = (NODE *)malloc(sizeof(NODE));
```

```
p1->data = 10;  
p1->link = NULL;
```

```
NODE *p2;  
p2 = (NODE *)malloc(sizeof(NODE));  
p2->data = 20;
```

```
p2->link = NULL;  
p1->link = p2;
```

```
free(p1);  
free(p2);
```



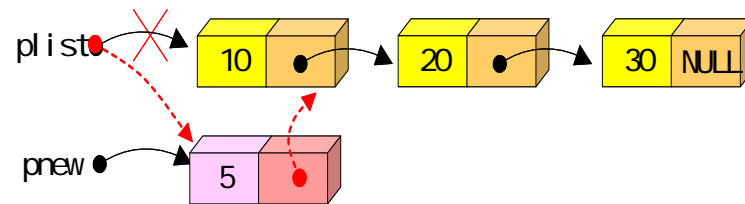


# 연결 리스트의 삽입 연산

```
NODE *insert_NODE(NODE *plist, NODE *pprev, DATA item);
```

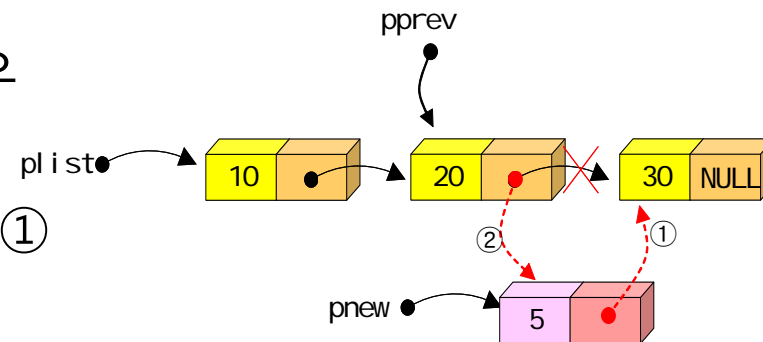
## 1. 리스트의 처음에 삽입하는 경우

```
pnew->link = plist;  
plist = pnew;
```



## 2. 리스트의 중간에 삽입하는 경우

```
pnew->link = pprev->link; // ①  
pprev->link = pnew; // ②
```



# 연결 리스트의 삽입 연산

```
NODE *insert_node(NODE *plist, NODE *pprev, DATA item)
{
    NODE *pnew = NULL;

    if( !(pnew = (NODE *)malloc(sizeof(NODE))) )
    {
        printf("메모리 동적 할당 오류\n");
        exit(1);
    }

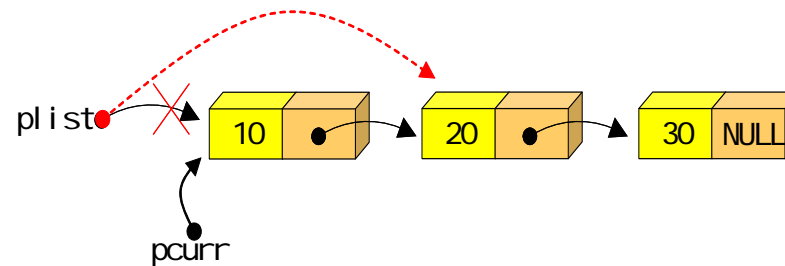
    pnew->data = data;
    if( pprev == NULL ) // 연결 리스트의 처음에 삽입
    {
        pnew->link = plist;
        plist = pnew;
    }
    else // 연결 리스트의 중간에 삽입
    {
        pnew->link = pprev->link;
        pprev->link = pnew;
    }
    return plist;
}
```

# 연결 리스트의 삭제 연산

```
NODE *delete_node(NODE *plist, NODE *pprev, NODE *pcurr);
```

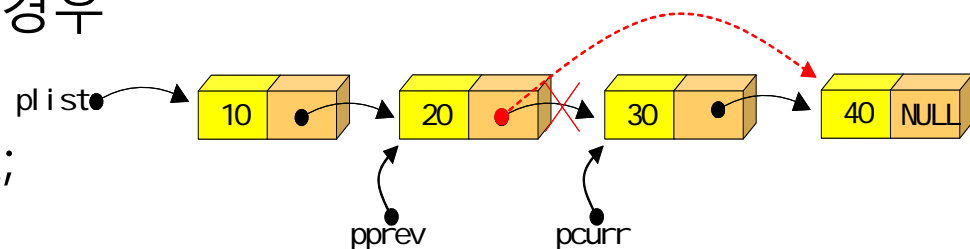
## 1. 리스트의 처음을 삭제하는 경우

```
plist = pcurr->link;  
free(pcurr);
```



## 2. 리스트의 중간에 삽입하는 경우

```
pprev->link = pcurr->link;  
free(pcurr);
```



# 연결 리스트의 삭제 연산

```
NODE *delete_node(NODE *plist, NODE *pprev, NODE *pcurr)
{
    if( pprev == NULL )
        plist = pcurr->link;
    else
        pprev->link = pcurr->link;

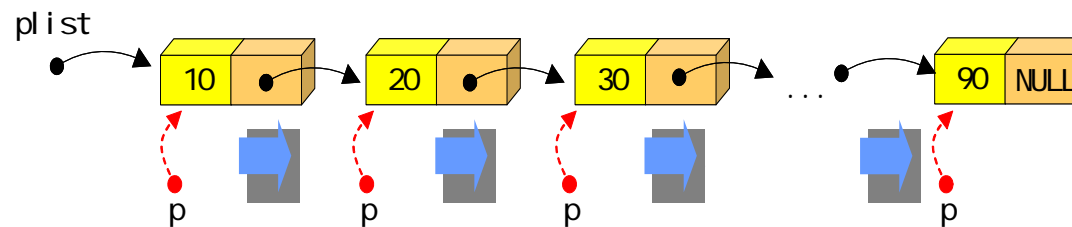
    free(pcurr);
    return plist;
}
```

# 연결 리스트의 순회 연산

```
void print_list(NODE *plist)
{
    NODE *p;

    p = plist;
    printf("( ");

    while( p )
    {
        printf("%d ", p->data);
        p = p->link;
    }
    printf("\n");
}
```



# 노드의 개수 세기

```
1. int get_length(NODE *plist)
2. {
3.     NODE *p;
4.     int length = 0;
5.
6.     p = plist;
7.
8.     while( p )
9.     {
10.         length++;
11.         p = p->link;
12.     }
13.     printf("리스트의 길이는 %d\n", length);
14.     return length;
15. }
```

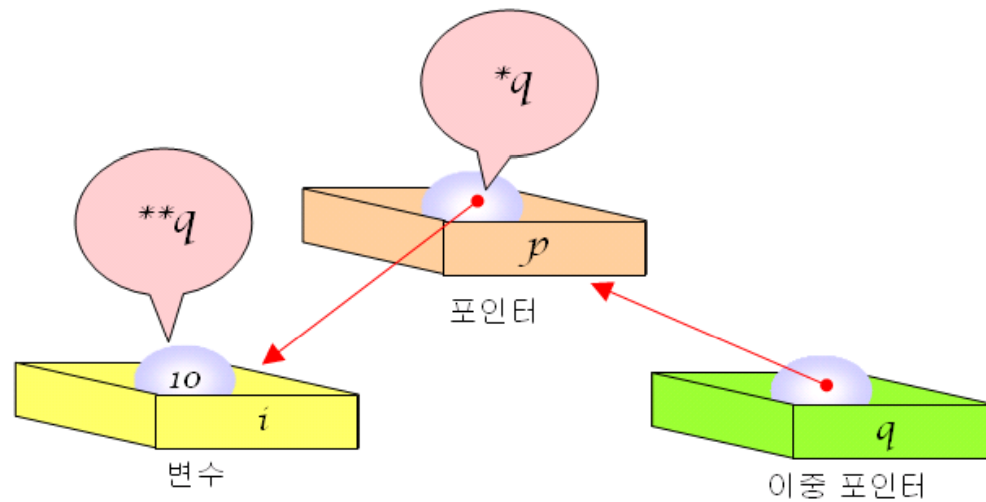
# 합계 구하기

```
1. int get_sum(NODE *plist)
2. {
3.     NODE *p;
4.     int sum = 0;
5.
6.     p = plist;
7.
8.     while( p )
9.     {
10.        sum += p->data;
11.        p = p->link;
12.    }
13.    printf("리스트의 합계는 %d\n", sum);
14.    return sum;
15. }
```

# 이중 포인터

- 이중 포인터(double pointer): 포인터를 가리키는 포인터

```
int i = 100;           // i는 int형 변수  
int *p = &i;          // p는 i를 가리키는 포인터  
int **q = &p;         // q는 포인터 p를 가리키는 이중 포인터
```





# 이중 포인터



```
// 이중 포인터 프로그램
#include <stdio.h>

int main(void)
{
    int i = 100;
    int *p = &i;
    int **q = &p;

    *p = 200;
    printf("i=%d *p=%d **q=%d \n", i, *p, **q);

    **q = 300;
    printf("i=%d *p=%d **q=%d \n", i, *p, **q);

    return 0;
}
```



```
i=200 *p=200 **q=200
i=300 *p=300 **q=300
```

# 예제 #2



// 이중 포인터 프로그램

```
#include <stdio.h>
```

```
void set_proverb(char **q);
```

```
int main(void)
```

```
{
```

```
    char *s = NULL;
```

```
    set_proverb(&s);
```

```
    printf("selected proverb = %s\n",s);
```

```
    return 0;
```

```
}
```

```
void set_proverb(char **q)
```

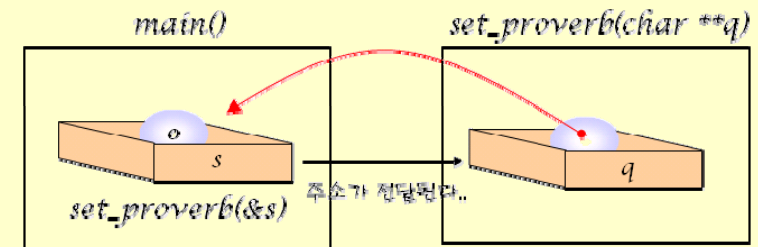
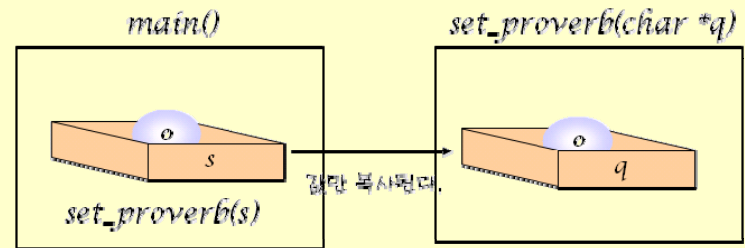
```
{
```

```
    static char *str1="A friend in need is a friend indeed";
```

```
    static char *str2="A little knowledge is a dangerous thing";
```

```
    *q = str1;
```

```
}
```



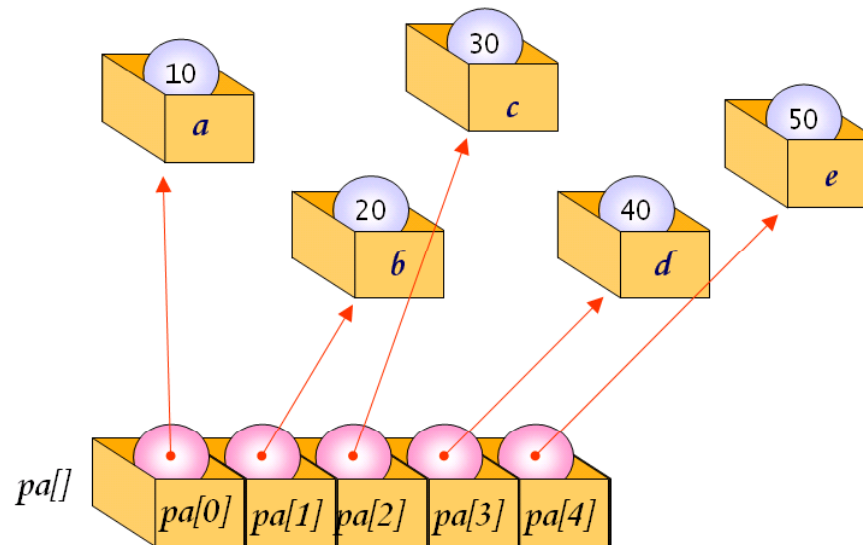
```
selected proverb = A friend in need is a friend indeed
```

# 포인터 배열

- *포인터 배열(array of pointers)*: 포인터를 모아서 배열로 만든 것

```
int a = 10, b = 20, c = 30, d = 40, e = 50;
```

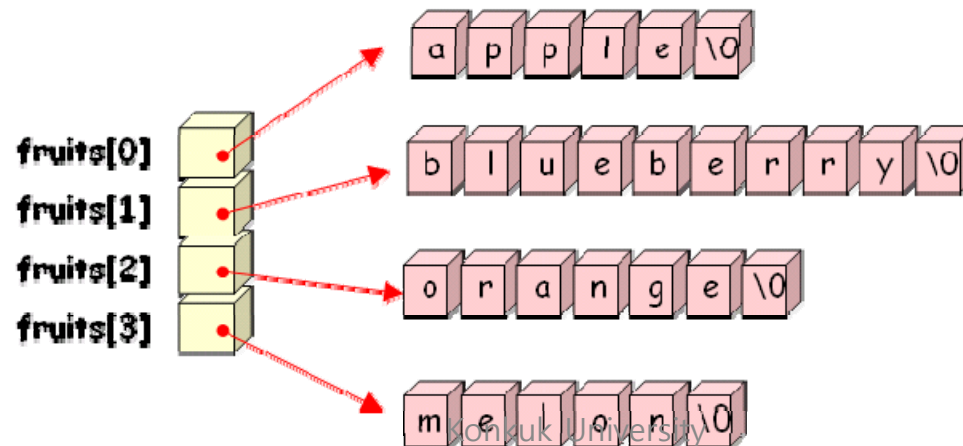
```
int *pa[5] = { &a, &b, &c, &d, &e };
```



# 문자열 배열

- 문자열 배열
  - 가장 많이 사용되는 포인터 배열
  - 문자열들을 효율적으로 저장할 수 있다.

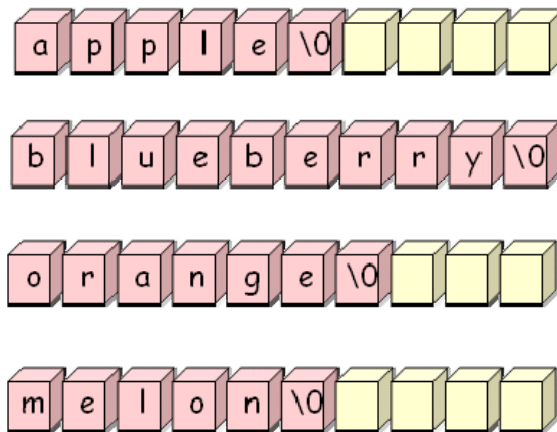
```
char *fruits[ ] = {  
    "apple",  
    "blueberry",  
    "orange",  
    "melon"  
};
```



# 문자열 배열 vs 2차원 배열

- 문자열들을 저장하는 2차원 배열
  - 공간의 낭비가 발생할 수 있다.

```
char fname[ ][10] = {  
    "apple",  
    "blueberry",  
    "orange",  
    "melon"  
};
```



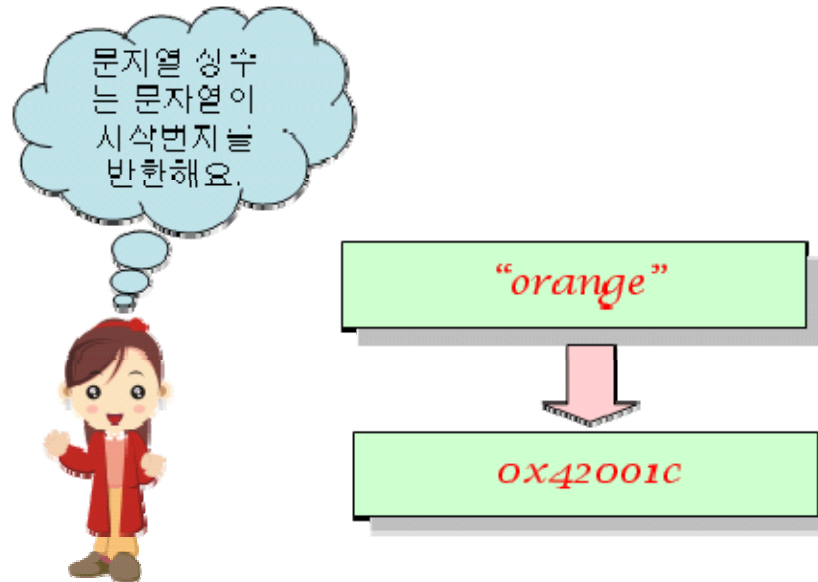
낭비되는  
공간!

2차원 배열을 사  
용하면 낭비되는  
공간이 생성되죠.



# 문자열 상수의 의미

- 문자열 상수는 문자열의 시작 번지를 반환한다.
- 포인터 배열의 각 원소들은 이 시작 번지로 초기화된다.



# stringarray.c



```
// 문자열 배열
#include <stdio.h>

int main(void)
{
    int i, n;
    char *fruits[ ] = {
        "apple",
        "blueberry",
        "orange",
        "melon"
    };

    n = sizeof(fruits)/sizeof(fruits[0]);    // 배열 원소 개수 계산

    for(i = 0; i < n; i++)
        printf("%s \n", fruits[i]);

    return 0;
}
```

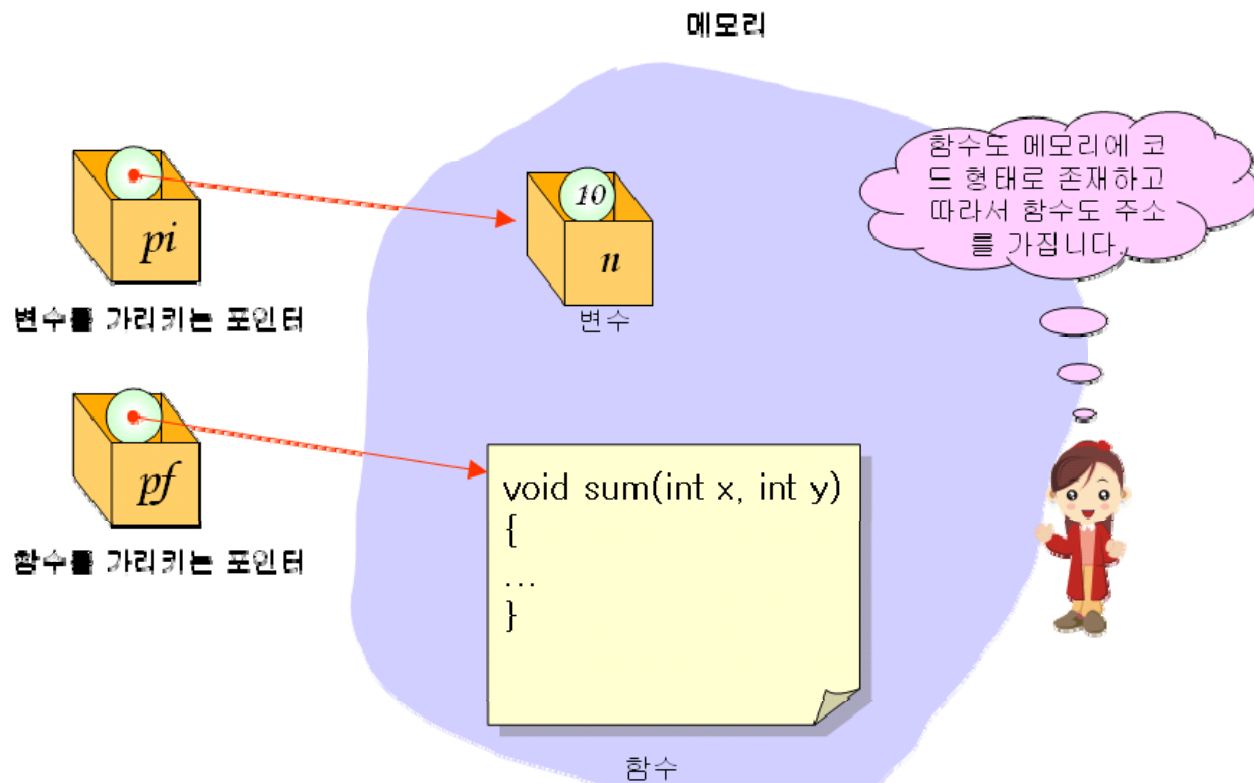


```
apple
blueberry
orange
melon
```

# 함수 포인터

- **함수 포인터(function pointer):** 함수를 가리키는 포인터

반환형 (\*함수포인터이름)(매개변수1, 매개변수2, ...);





# fp1.c



```
// 함수 포인터
#include <stdio.h>

// 함수 원형 정의
int add(int, int);
int sub(int, int);

int main(void)
{
    int result;
    int (*pf)(int, int);           // 함수 포인터 정의

    pf = add;                     // 함수 포인터에 함수 add()의 주소 대입
    result = pf(10, 20);          // 함수 포인터를 통한 함수 add() 호출
    printf("10+20은 %d\n", result);

    pf = sub;                     // 함수 포인터에 함수 sub()의 주소 대입
    result = pf(10, 20);          // 함수 포인터를 통한 함수 sub() 호출
    printf("10-20은 %d\n", result);

    return 0;
}
```

# fp1.c

```
int add(int x, int y)
{
    return x+y;
}

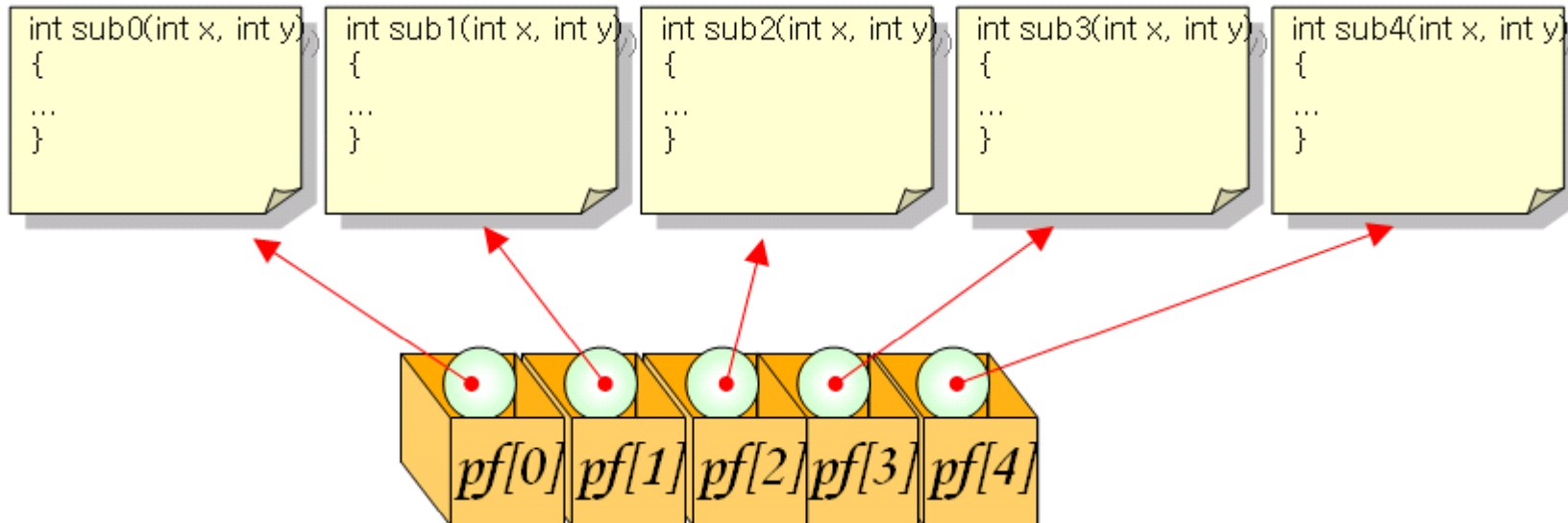
int sub(int x, int y)
{
    return x-y;
}
```



```
10+20은 30
10-20은 -10
```

# 함수 포인터의 배열

- 반환형 (\*배열이름[배열의\_크기])(매개변수목록);
  - `int (*pf[5])(int, int);`



# fp2.c



```
// 함수 포인터 배열
#include <stdio.h>

// 함수 원형 정의
void menu(void);
int add(int x, int y);
int sub(int x, int y);
int mul(int x, int y);
int div(int x, int y);

void menu(void)
{
    printf("=====\n");
    printf("0. 덧셈\n");
    printf("1. 뺄셈\n");
    printf("2. 곱셈\n");
    printf("3. 나눗셈\n");
    printf("4. 종료\n");
    printf("=====\n");
}
```

# fp2.c

```
int main(void)
{
    int choice, result, x, y;
    // 함수 포인터 배열을 선언하고 초기화한다.
    int (*pf[4])(int, int) = { add, sub, mul, div };
    while(1)
    {
        menu();

        printf("메뉴를 선택하시오:");
        scanf("%d", &choice);

        if( choice < 0 || choice >=4 )
            break;

        printf("2개의 정수를 입력하시오:");
        scanf("%d %d", &x, &y);

        result = pf[choice](x, y); // 함수 포인터를 이용한 함수 호출

        printf("연산 결과 = %d\n",result);
    }
    return 0;
}
```

함수 포인터 배열 선언

# fp2.c

```
int add(int x, int y)
{
    return x + y;
}

int sub(int x, int y)
{
    return x - y;
}

int mul(int x, int y)
{
    return x * y;
}

int div(int x, int y)
{
    return x / y;
}
```



- ```
=====
0. 덧셈
1. 뺄셈
2. 곱셈
3. 나눗셈
4. 종료
```

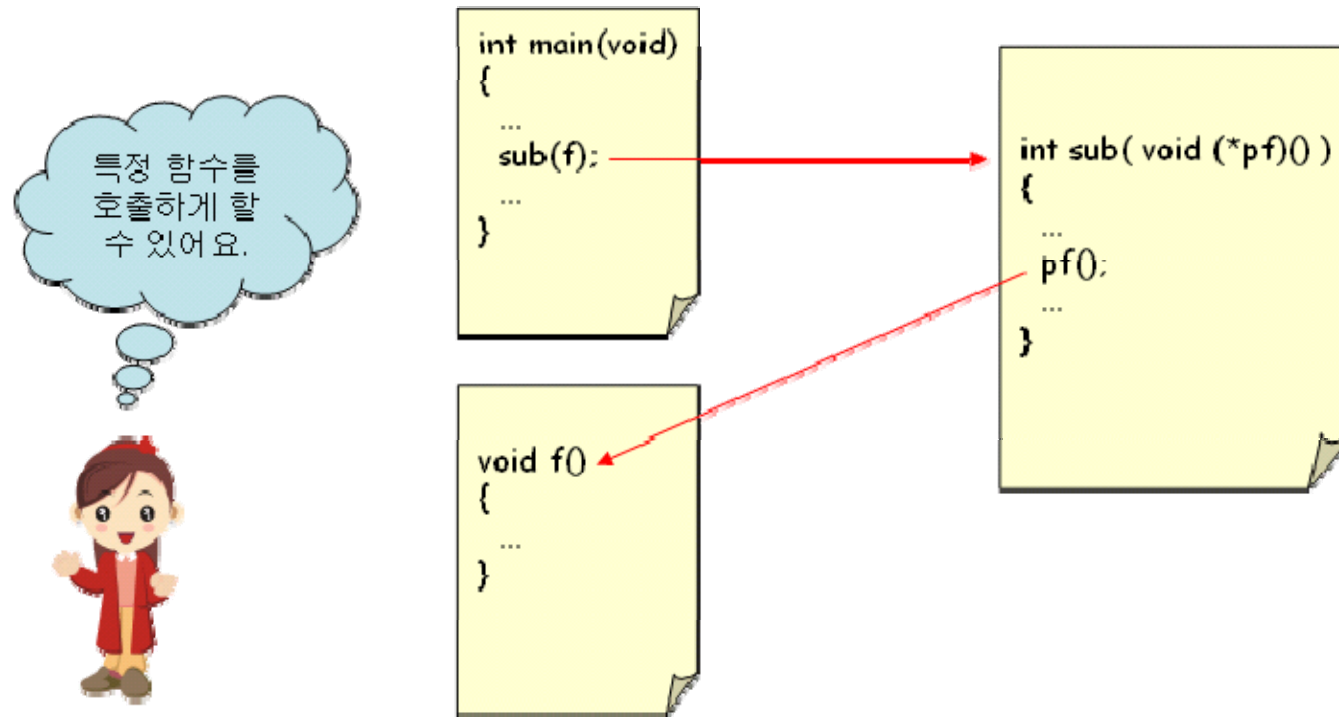
```
=====
메뉴를 선택하시오:2
2개의 정수를 입력하시오:10 20
연산 결과 = 200
```

- ```
=====
0. 덧셈
1. 뺄셈
2. 곱셈
3. 나눗셈
4. 종료
```

```
=====
메뉴를 선택하시오:
Konkuk University
```

# 함수 인수로서의 함수 포인터

- 함수 포인터도 인수로 전달이 가능하다.



# fp2.c



```
#include <stdio.h>
#include <math.h>

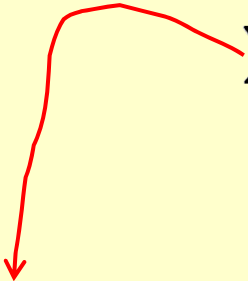
double f1(double k);
double f2(double k);
double formula(double (*pf)(double), int n);
```

```
int main(void)
{
    printf("%f\n", formula(f1, 10));
    printf("%f\n", formula(f2, 10));
}
```

```
double formula(double (*pf)(double), int n)
{
    int i;
    double sum = 0.0;

    for(i = 1; i < n; i++)
        sum += pf(i) * pf(i) + pf(i) + 1;
    return sum;
}
```

$$\sum_{k=1}^n (f^2(k) + f(k) + 1)$$





# fp2.c

```
double f1(double k)
{
    return 1.0 / k;
}

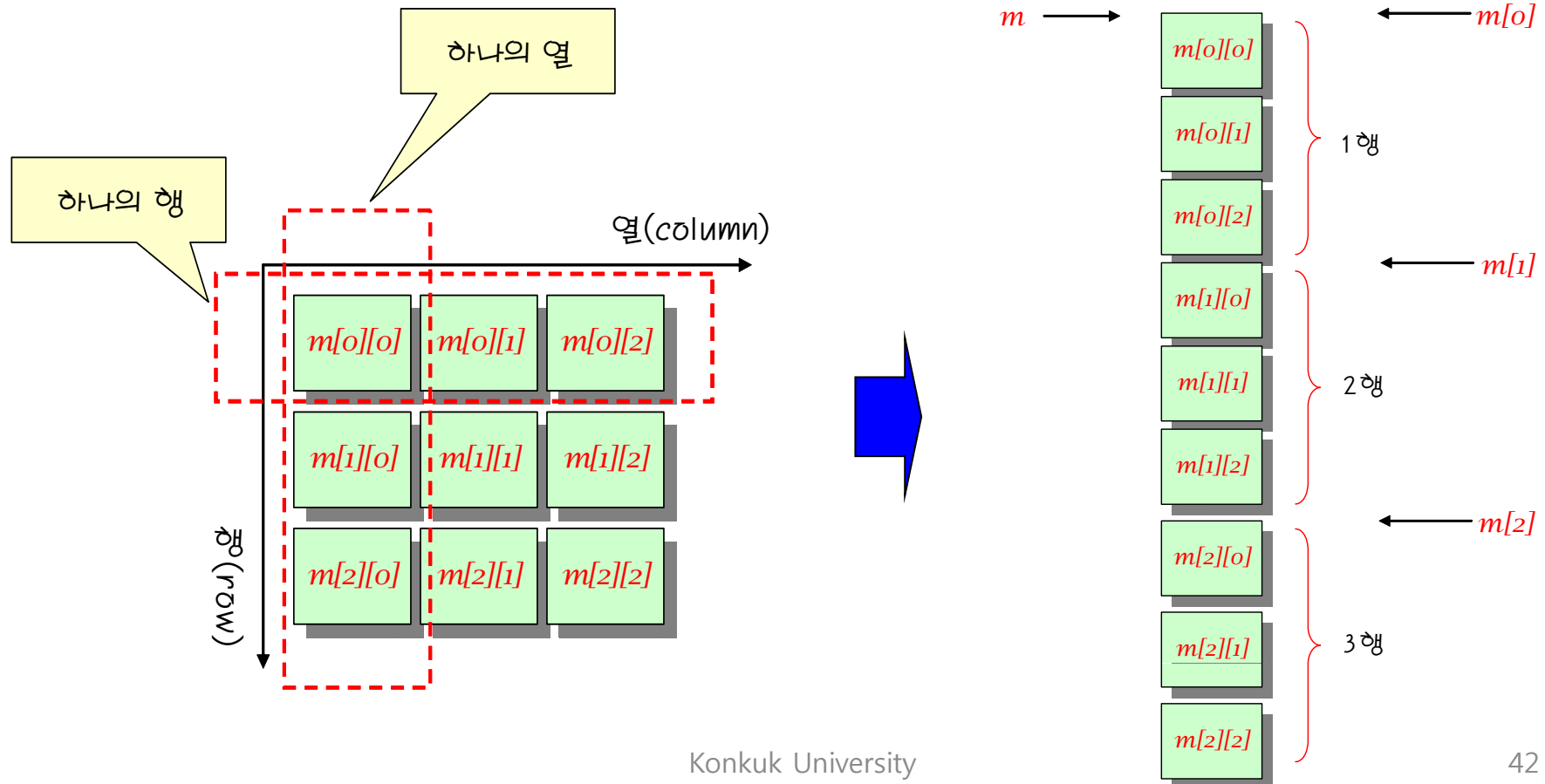
double f2(double k)
{
    return cos(k);
}
```



```
13.368736
12.716152
```

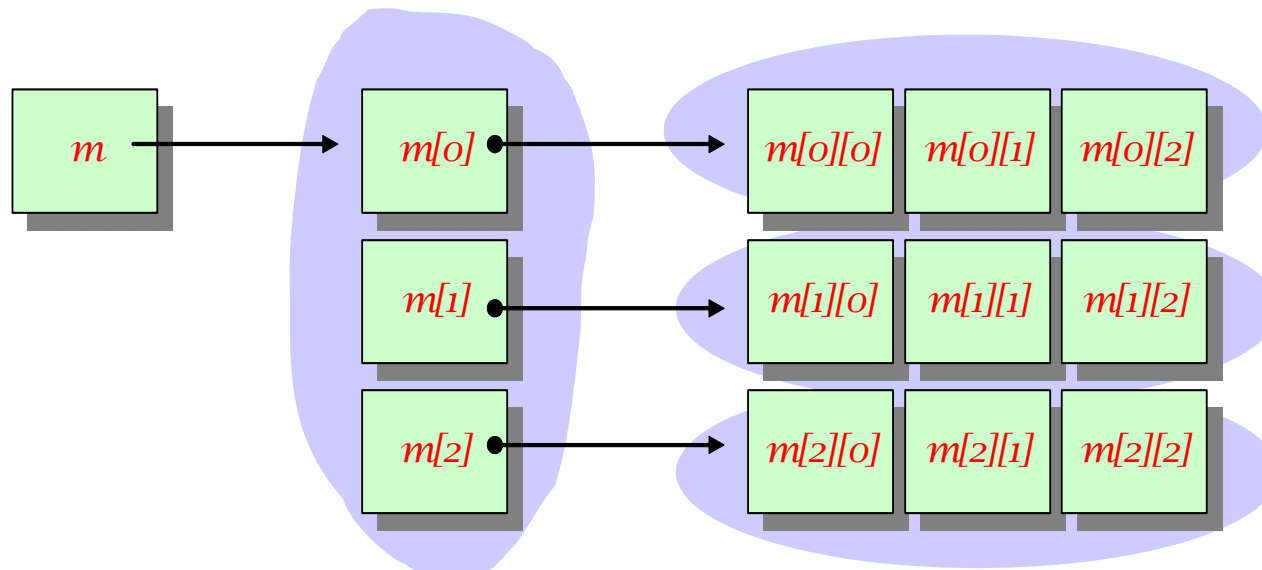
# 다차원 배열과 포인터

- 2차원 배열 `int m[3][3]`
- 1행->2행->3행->...순으로 메모리에 저장(행우선 방법)



# 2차원 배열과 포인터

- 배열 이름  $m$ 은  $\&m[0][0]$
- $m[0]$ 는 1행의 시작 주소
- $m[1]$ 은 2행의 시작 주소
- ...



# multi\_array.c



```
// 다차원 배열과 포인터
#include <stdio.h>

int main(void)
{
    int m[3][3] = { 10, 20, 30, 40, 50, 60, 70, 80, 90 };

    printf("m = %p\n", m);
    printf("m[0] = %p\n", m[0]);
    printf("m[1] = %p\n", m[1]);
    printf("m[2] = %p\n", m[2]);
    printf("&m[0][0] = %p\n", &m[0][0]);
    printf("&m[1][0] = %p\n", &m[1][0]);
    printf("&m[2][0] = %p\n", &m[2][0]);

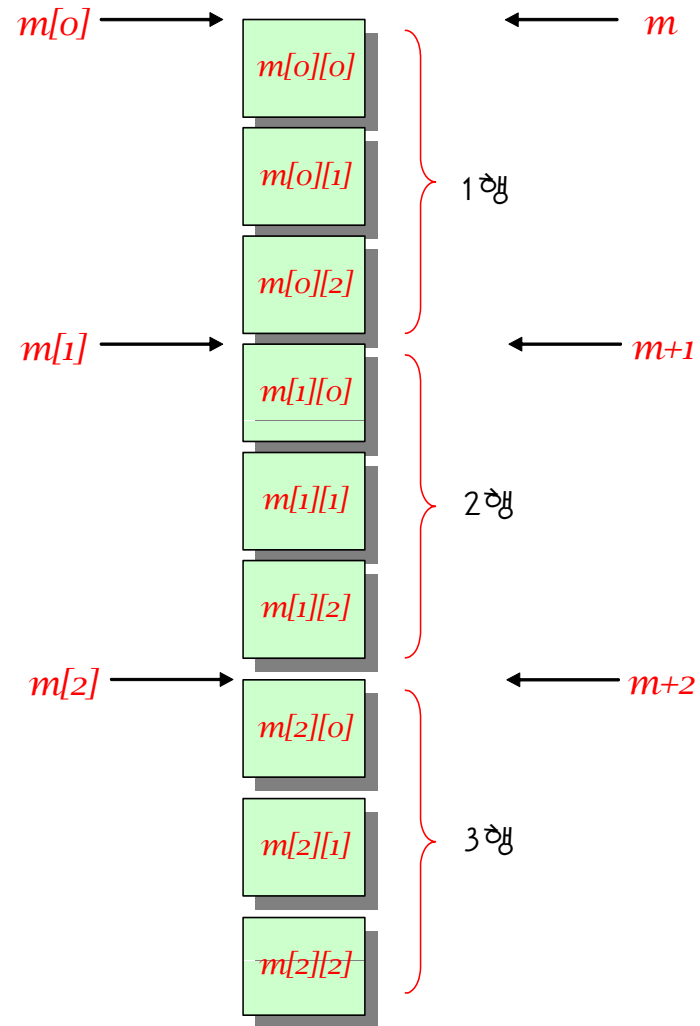
    return 0;
}
```



```
m = 1245020
m[0] = 1245020
m[1] = 1245032
m[2] = 1245044
&m[0][0] = 1245020
&m[1][0] = 1245032
&m[2][0] = 1245044
```

# 2차원 배열과 포인터 연산

- $m$ 에 대한 연산의 의미
- $m$ 은  $\&m[0][0]$
- $m+1$ 은  $m[1]$
- $m+2$ 은  $m[2]$



# two\_dim\_array.c



```
#include <stdio.h>

int main(void)
{
    int m[3][3] = { 10, 20, 30, 40, 50, 60, 70, 80, 90 };

    printf("m = %p\n", m);
    printf("m+1 = %p\n", m+1);
    printf("m+2 = %p\n", m+2);
    printf("m[0] = %p\n", m[0]);
    printf("m[1] = %p\n", m[1]);
    printf("m[2] = %p\n", m[2]);

    return 0;
}
```



```
m = 1245020
m+1 = 1245032
m+2 = 1245044
m[0] = 1245020
m[1] = 1245032
m[2] = 1245044
```

# 포인터를 이용한 배열 원소 방문

- 행의 평균을 구하는 경우

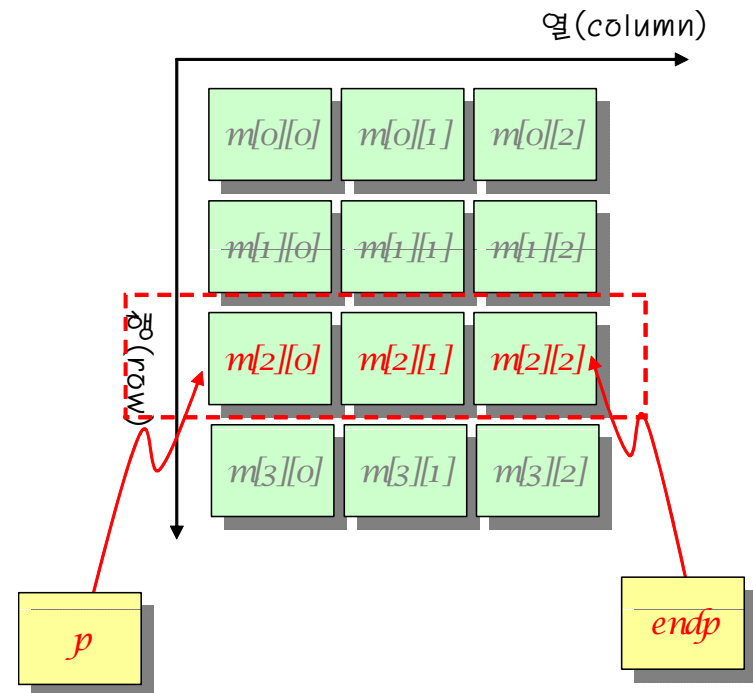
```
double get_row_avg(int m[][COLS], int r)
{
    int *p, *endp;
    double sum = 0.0;

    p = &m[r][0];
    endp = &m[r][COLS];

    while( p < endp )
        sum += *p++;

    sum /= COLS;

    return sum;
}
```



# 포인터를 이용한 배열 원소 방문

- 전체 원소의 평균을 구하는 경우

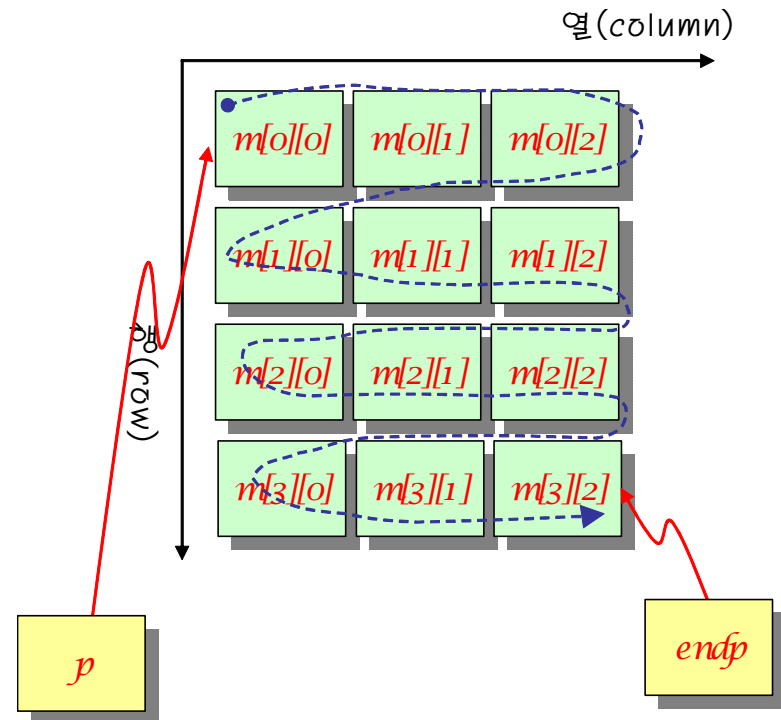
```
double get_total_avg(int m[][COLS])
{
    int *p, *endp;
    double sum = 0.0;

    p = &m[0][0];
    endp = &m[ROWS-1][COLS];

    while( p < endp )
        sum += *p++;

    sum /= ROWS * COLS;

    return sum;
}
```





# void 포인터

- 순수하게 메모리의 주소만 가지고 있는 포인터
- 가리키는 대상물은 아직 정해지지 않음  
(예) `void *vp;`
- 다음과 같은 연산은 모두 오류이다.

```
*vp;           // 오류  
*(int *)vp;    // void형 포인터를 int형 포인터로 변환한다.  
vp++;          // 오류  
vp--;          // 오류
```



# vp.c

```
#include <stdio.h>

int main(void)
{
    int a[] = { 10, 20, 30, 40, 50 };
    void *vp;

    vp = a; // 가능
    vp = &a[2]; // 가능

    *vp = 35; // 오류
    vp++; // 오류

    *(int *)vp = 35; // 가능

    return 0;
}
```

# main() 함수의 인수

- 지금까지의 main() 함수 형태

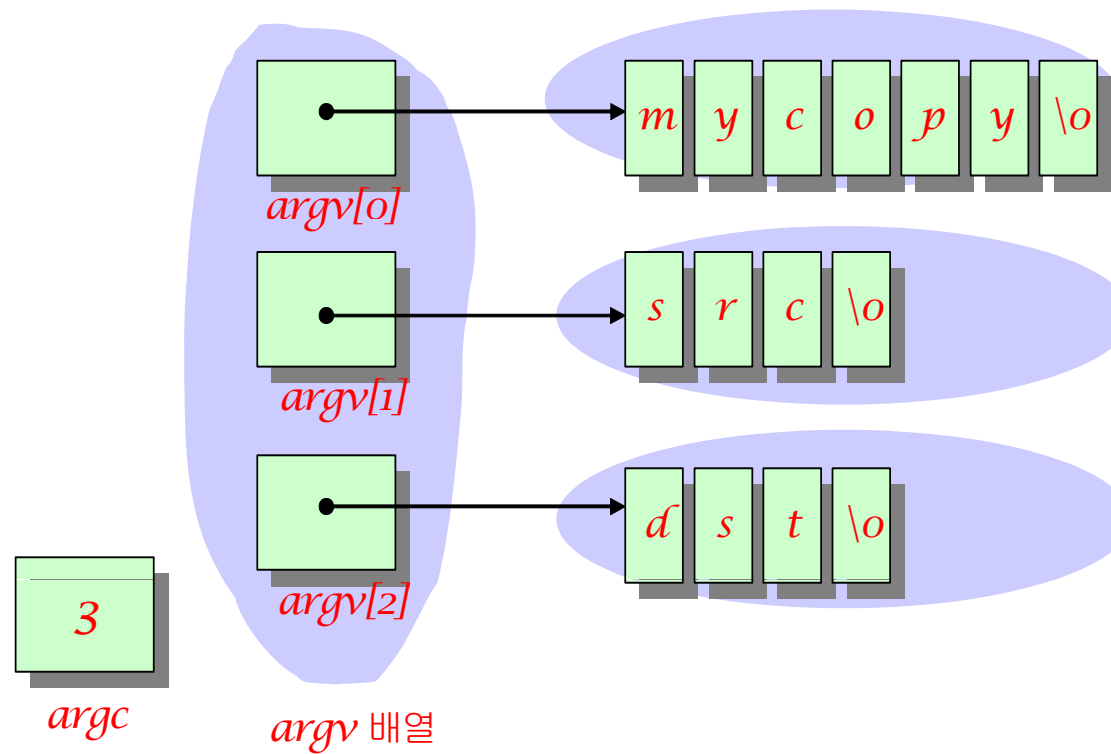
```
int main(void)
{
..
}
```

- 외부로부터 입력을 받는 main() 함수 형태

```
int main(int argc, char *argv[])
{
..
}
```

# 인수 전달 방법

```
C: \cprogram> mycopy src dst
```



# main\_arg.c



```
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i = 0;

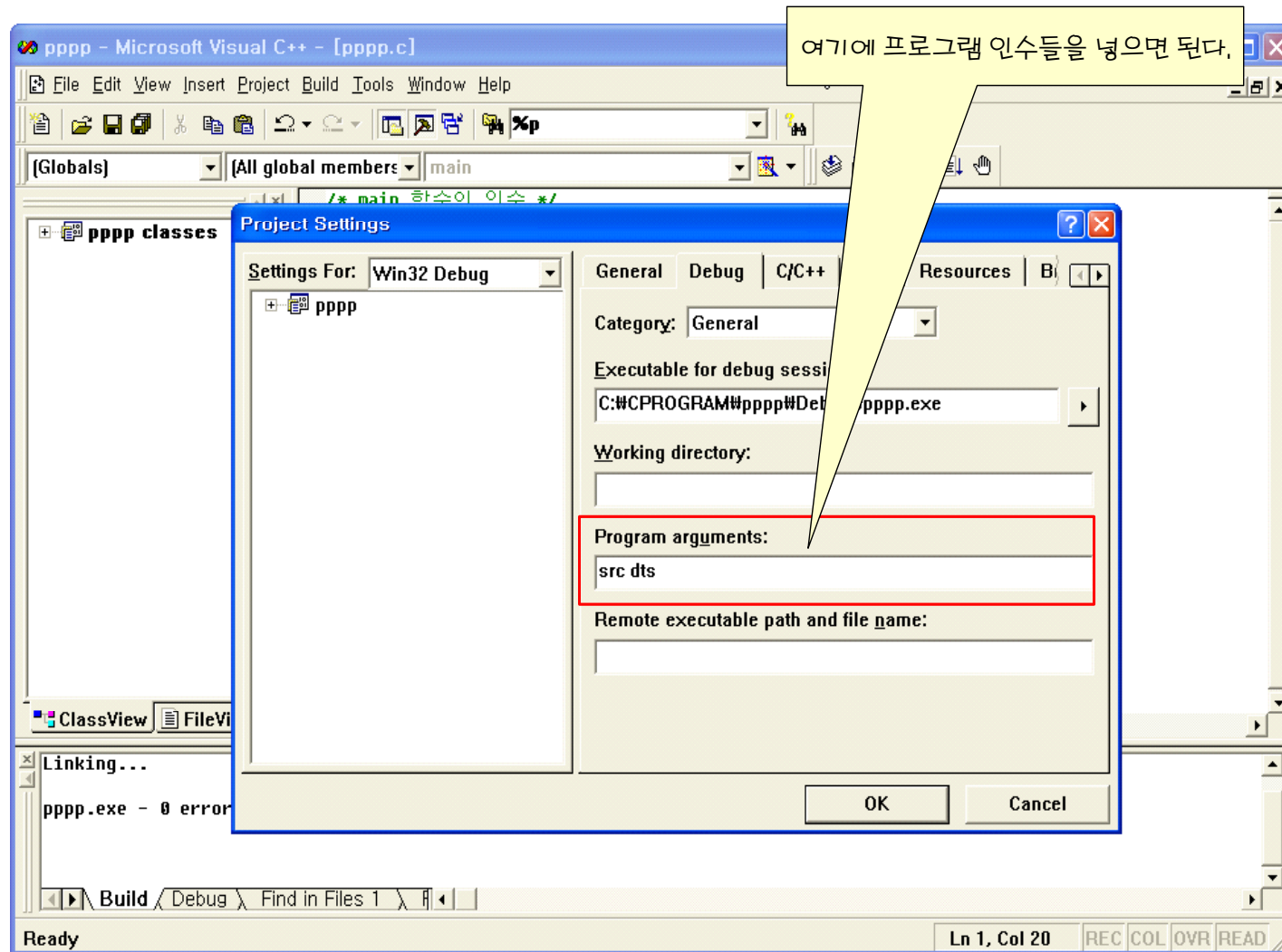
    for(i = 0; i < argc; i++)
        printf("명령어 라인에서 %d번째 문자열 = %s\n", i, argv[i]);

    return 0;
}
```



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
c:\cprogram\mainarg\Debug>mainarg src dst
명령어 라인에서 0번째 문자열 = mainarg
명령어 라인에서 1번째 문자열 = src
명령어 라인에서 2번째 문자열 = dst
c:\cprogram\mainarg\Debug>
```

# 비주얼 C++ 프로그램 인수 입력 방법



# mile2km.c



```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    double mile, km;

    if( argc != 2 ){
        printf("사용 방법: mile2km 거리\n");
        return 1;
    }
    mile = atof(argv[1]);
    km = 1.609 * mile;
    printf("입력된 거리는 %f km입니다. \n", km);

    return 0;
}
```



```
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
c:\cprogram\mainarg\Debug>mainarg 10
입력된 거리는 16.090000 km입니다.
c:\cprogram\mainarg\Debug>
```

# Q & A

