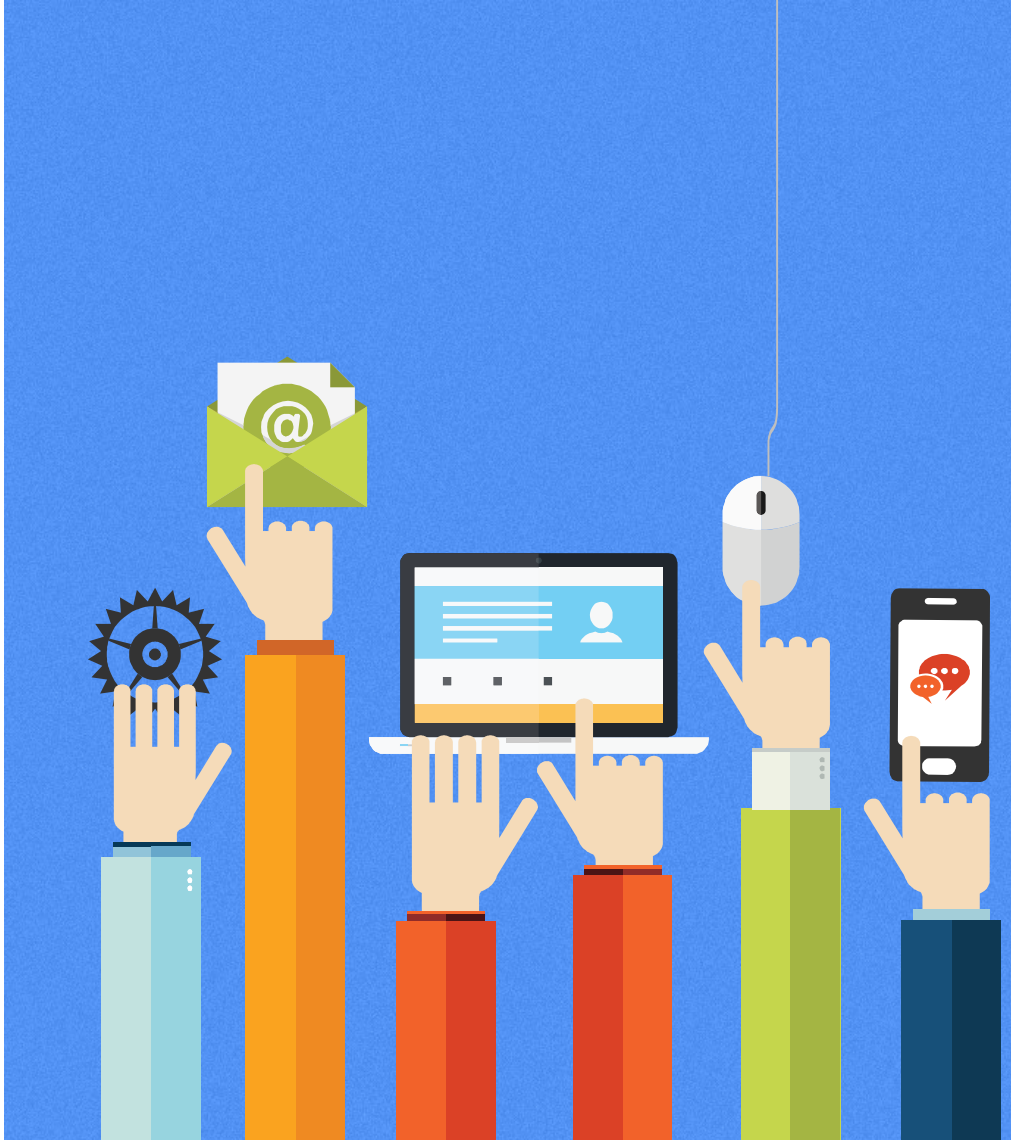


제16장

동적 메모리와 전처리

Perfect C

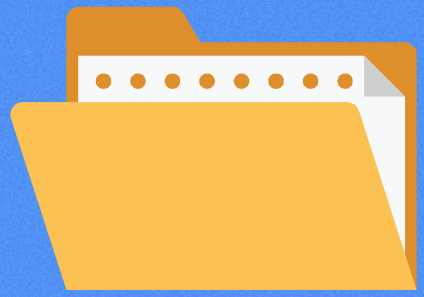
C언어로 배우는 프로그래밍 기초

단원 목표

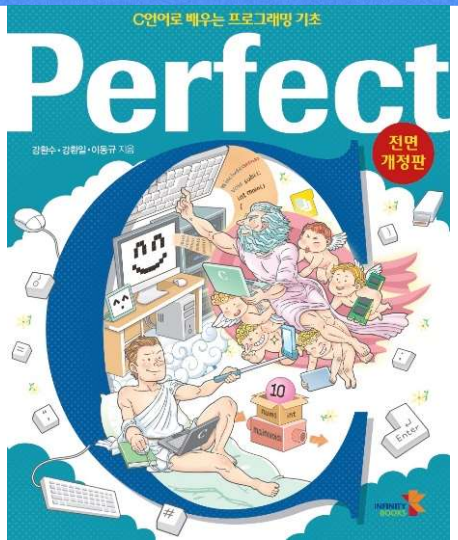
학습목표

- **동적 메모리 할당 방식을 이해하고 설명할 수 있다.**
  - 정적 할당과 동적 할당의 필요성
  - 함수 malloc(), calloc(), realloc(), free()의 사용
  - 동적메모리 할당으로 구현하는 자기참조 구조체
- **연결 리스트를 이해하고 설명할 수 있다.**
  - 연결 리스트의 이해와 구현
  - 연결 리스트의 노드, 헤드, 순회, 삽입, 삭제
  - 여러 파일로 구성하는 프로젝트와 사용자정의 헤더파일
  - 연결 리스트의 구현
- **전처리 지시자를 이해하고 설명할 수 있다.**
  - 매크로와 인자를 활용한 매크로
  - #if와 #endif
  - #ifdef와 #endif, #ifndef #endif
  - 전처리 연산자 #, #@, ##, defined

학습목차



01. 동적 메모리와 자기참조 구조체



동적 메모리 할당과 해제

- **정적 메모리 할당(static memory allocation)**
  - 변수와 배열, 구조체 모두 프로그램 실행 전에 필요한 만큼의 변수를 선언하여 사용
    - 프로그램 실행 중에 변수의 수를 늘리거나 줄이는 것이 불가능
  - 컴파일 이전에 저장공간 수나 크기를 정한 메모리 할당 방법을 정적 메모리 할당 방법
    - 프로그램이 실행되기 이전에 변수의 저장 공간 크기가 정해짐
    - 프로그램 또는 함수가 시작되면 메모리에 할당되어 사용
    - 그 모듈이나 프로그램이 종료되면 변수가 메모리에서 삭제되는 방식
    - 사용이 간편하나 실행 이전에 사용할 메모리의 공간 크기가 정해짐
      - 메모리의 사용 예측이 부정확한 경우 충분한 메모리를 미리 확보해야 하므로 비효율적
- **동적 메모리 할당(dynamic memory allocation)**
  - 프로그램 실행 중에 필요한 메모리를 할당하는 방법
  - 정적 할당 방식인 변수 선언에 비해 상대적으로 다소 어려움
  - 메모리 사용 예측이 정확하지 않고 실행 중에 메모리 할당이 필요하다면 동적 메모리 할당 방식이 적합

정적 메모리 할당 방식

```
int i;
long prod = 1;
int facto[6];
char *[] = {"algol", "pascal" "C",
      "C++", "Java","C#"};
```

동적 메모리 할당 방식

```
int *pi = NULL;
//메모리 할당 함수 malloc()으로 동적메모리 할당
pi = (int *)malloc(sizeof(int));
//동적메모리 할당 성공 검사
if (pi == NULL) {
   printf("메모리 할당에 문제가 있습니다.");
   exit(1);
};
//내용 값 저장
*pi = 3;
```

그림 16-1 정적 메모리 할당 방식과 동적 메모리 할당 방식

동적 메모리 관련 함수

- **동적 메모리**
  - 함수 malloc()의 호출로 힙(heap) 영역에 확보
  - 메모리는 사용 후 함수 free()를 사용해 해제
  - 만일 메모리 해제를 하지 않으면 메모리 부족과 같은 문제를 일으킬 수 있으니 꼭 해제하는 습관

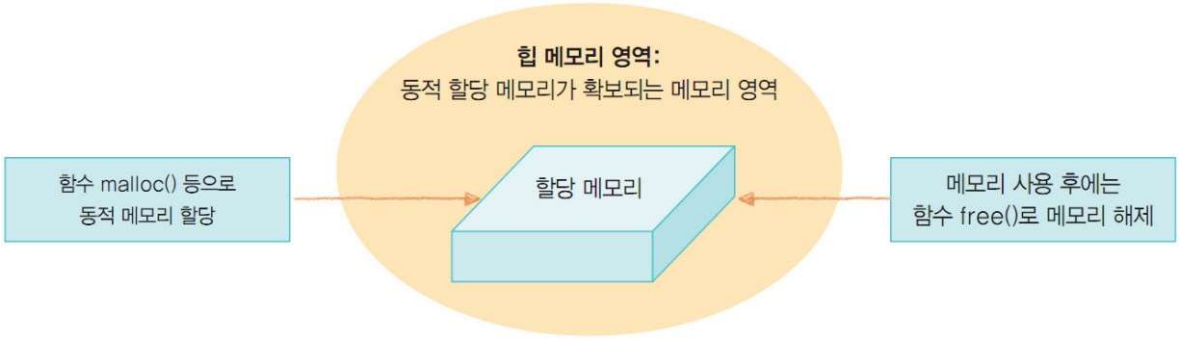


그림 16-2 동적 메모리 할당과 해제

- **동적 메모리 할당 함수: malloc(), calloc(), realloc() 3가지**
  - 반환 형이 void 포인터(void *)로 메모리 할당에 요구한 자료의 포인터 형으로 변환
  - 헤더파일 stdlib.h 필요
- **동적으로 할당된 메모리를 해제**
  - 함수 free()

표 16-1 동적 메모리 관련 함수

메모리	함수 원형	기능
메모리 할당 (기본값 없이)	`void * malloc(size_t)`	인자만큼의 메모리 할당 후 기본 주소 반환
메모리 할당 (기본값 0으로)	`void * calloc(size_t , size_t)`	뒤 인자 만큼의 메모리 크기로 앞 인자 수 만큼 할당 후 기본 주소 반환
기존 메모리 변경 (이전값 그대로)	`void * realloc(void *, size_t)`	앞 인자의 메모리를 뒤 인자 크기로 변경 후, 기본 주소 반환
메모리 해제	`void free(void *)`	인자를 기본 주소로 갖는 메모리 해제

메모리 할당 영역

- **메모리 영역**
  - 데이터(static data) 영역, 힙(heap) 영역, 스택(stack) 영역 세 부분
  - 힙 영역
    - 동적 메모리가 할당되는 부분
  - 데이터 영역
    - 전역변수와 정적변수가 할당되는 저장공간
  - 스택 영역
    - 함수 호출에 의한 형식 매개변수 그리고 함수 내부의 지역변수가 할당되는 저장공간

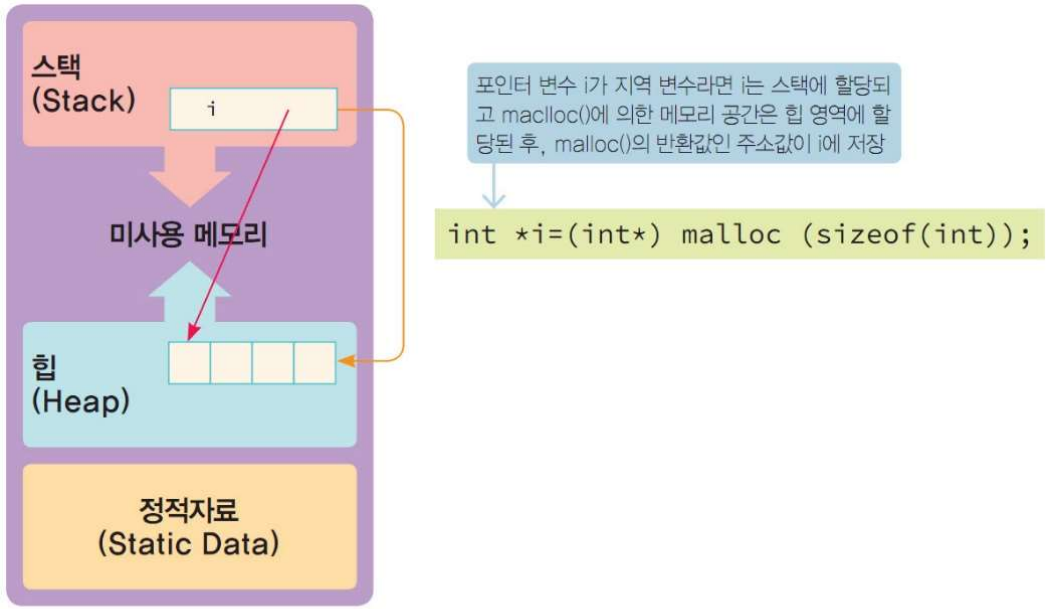


그림 16-3 메모리 영역

함수 malloc()

- **동적 메모리 할당 기능의 기본 함수**
  - malloc()

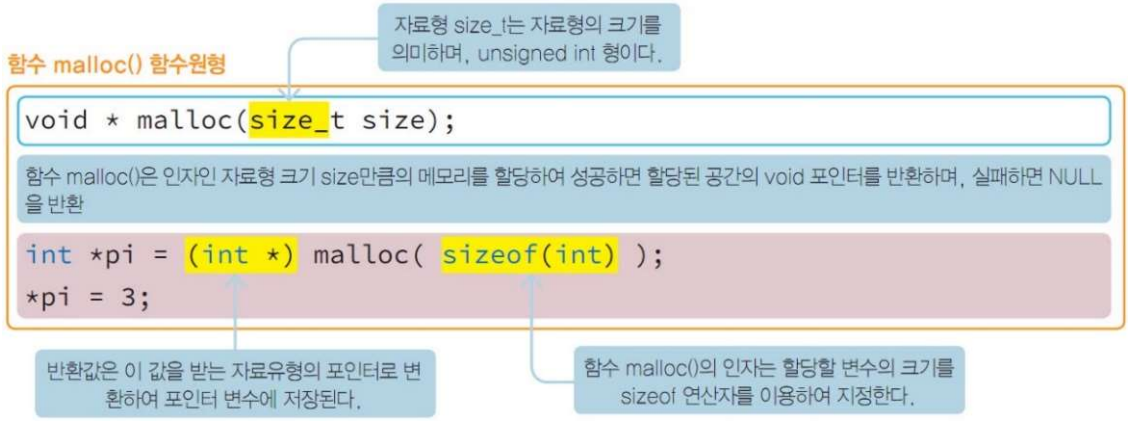


그림 16-4 함수 malloc()의 함수원형과 사용 방법

- **인자: 메모리 할당의 크기를 지정**
  - 할당된 메모리의 시작 주소를 반환
  - 반환값의 유형은 모든 자료형의 포인터로 이용할 수 있도록 void *를 사용
  - 확보된 공간의 주소는 int *의 변수에 저장
  - 간접연산자 *pi를 이용하여 원하는 값을 수정 가능

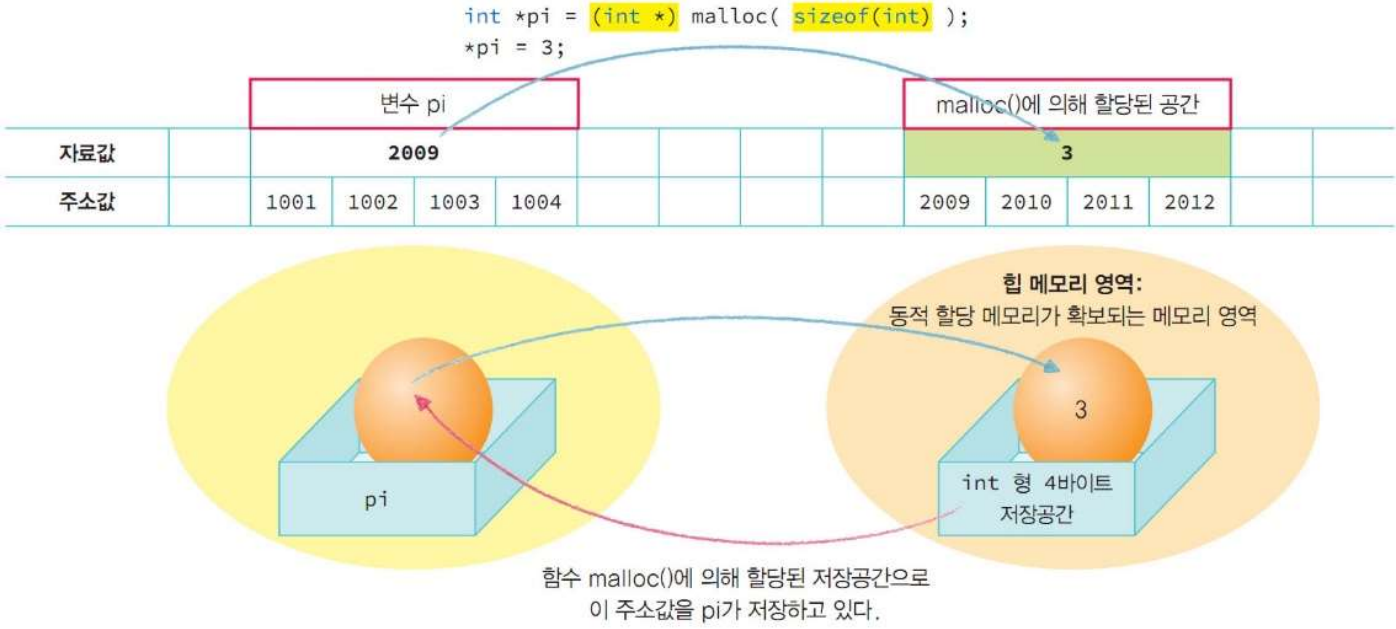


그림 16-5 함수 malloc()으로 정수형 저장공간 할당

함수 free()

- **메모리 해제에 이용되는 함수**
  - free()

함수free() 함수원형

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

동적으로 할당된 메모리를 제거한다.

```
free(pi);
```

그림 16-6 함수 free()의 함수원형과 사용 방법

- **free(pi)**
  - 함수 malloc()의 반환 주소를 저장한 변수 pi를 해제
    - 인자로 해제할 메모리 공간의 주소값을 갖는 포인터를 이용하여 호출
  - 변수 pi가 가리키는 4바이트의 자료값이 해제되어 더 이상 사용할 수 없음

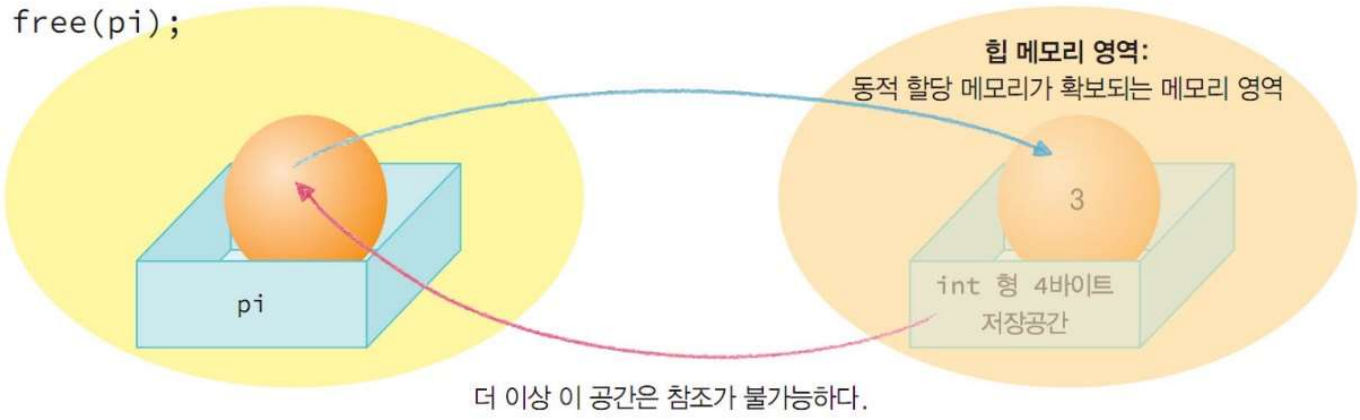


그림 16-7 함수free()에 의해 할당된 저장공간의 해제

동적 메모리 사용

예제 malloc.c

- 동적 메모리 할당 함수 malloc()

함수 malloc()

- 함수 malloc()을 이용하여 int형 메모리 공간을 하나 할당하여 값 3을 저장한 후 출력
- 만일 메모리 공간이 부족하여 요청된 공간을 할당하지 못한다면 함수 malloc()은 NULL을 반환

실습예제 16-1 malloc.c

함수malloc()을 이용하여 int 형 저장공간을 확보하여 처리

```
01  // file: malloc.c
02  #include <stdio.h>
03  #include <stdlib.h>
04
05  int main(void)
06  {
07     int *pi = NULL;
08     //메모리 할당 함수 malloc()으로 동적메모리 할당
09     pi = (int *)malloc(sizeof(int));
10     //동적메모리 할당 성공 검사
11     if (pi == NULL) {
12        printf("메모리 할당에 문제가 있습니다.");
13        exit(1);
14     };
15     //내용 값 저장
16     *pi = 3;
17     printf("주소값: *pi = %d, 저장값: p = %d\n", pi, *pi);
18
19     //메모리 해제
20     free(pi);
21
22     return 0;
23  }
```

만일을 대비해서 함수 malloc()의 반환값을 점검하는 모듈이 필요하다.

설명

```
07  동적메모리 할당의 주소를 저장하기 위한 포인터
09  함수 malloc()으로 동적메모리 할당 후 int 형 포인터인 pi에 저장하기 위해
    자료형 변환 (int *)이 필요
11  오류로 pi가 NULL이면 종료
16  동적메모리 할당에 의한 공간에 3 저장
17  동적메모리 공간의 주소값과 저장값 출력
20  동적메모리 공간의 메모리 해제
```

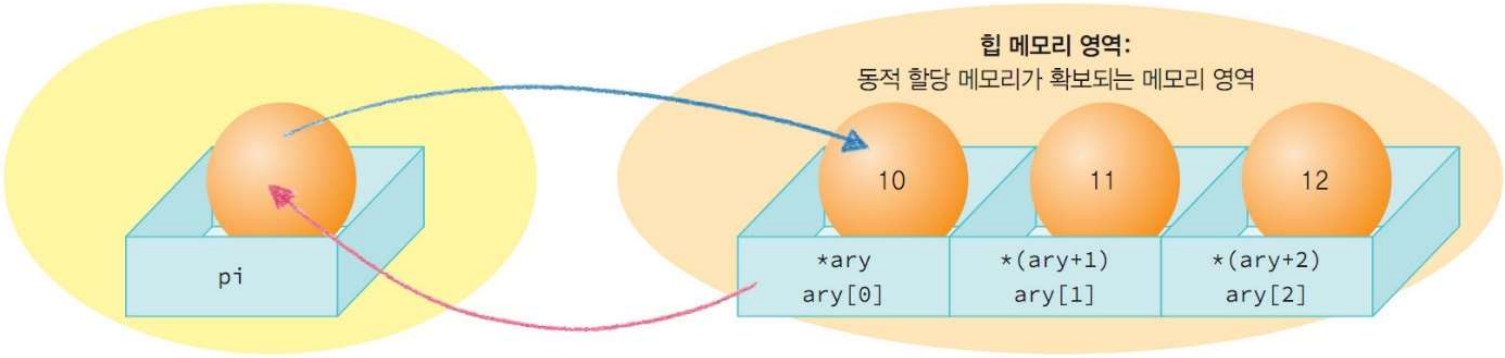
실행결과

```
주소값: *pi = 4631344, 저장값: p = 3
```

함수 malloc()에 의한 배열 공간 할당

- **배열과 같이 동일한 메모리 공간 여러 개를 동적으로 확보하는 방법**
  - int형 배열을 확보하는 방법
    - 함수 malloc()이 반환하는 값은 포인터인 int *의 변수로 저장
    - malloc()의 인자
      - sizeof(int) * (확보하려는 배열의 원소의 개수)로 지정
    - 변수 ary를 이용하여 다음과 같이 배열 형태와 같이 이용 가능

```
int *ary;
ary = (int *) malloc( sizeof(int)*3 );
ary[0] = 10; ary[1] = 11; ary[2] = 12;
```



함수 malloc()에 의해 할당된 저장공간은 int형 3개이며 주소값을 ary에 저장하고 있다. ary[i]로 각 원소를 참조할 수 있다.

그림 16-8 함수 malloc()으로 정수형 저장공간 할당

동적 배열 사용

예제 arraymalloc.c

- 동적 할당은 이와 같이 실행시간에 메모리 공간의 수가 결정되는 프로그램에 적합
- 표준입력으로 입력될 성적의 갯수를 먼저 입력 받아 그 수만큼 메모리를 동적으로 할당
  - 동적으로 할당된 메모리에 표준입력으로 받은 여러 성적을 저장하여 합과 평균을 구하여 출력

실습예제 16-2 arraymalloc.c

함수malloc()을 이용하여 동일한 저장공간 여러 개를 확보하여 처리

```c
// file: arraymalloc.c
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   int n = 0;
   printf("입력할 점수의 개수를 입력 >> ");
   scanf("%d", &n);

   int *ary = NULL;
   //동적 메모리 할당
   if ((ary = (int *)malloc(sizeof(int)*n)) == NULL)
   {
      printf("메모리 할당에 문제가 있습니다.");
      exit(1);
   };
   //표준입력과 처리
   printf("%d개의 점수 입력 >> ", n);
   int sum = 0;
   for (int i = 0; i < n; i++)
   {
      scanf("%d", (ary + i));
      sum += *(ary + i); //sum += ary[i];
   }
   //표준입력 자료와 결과 출력
   printf("입력 점수: ");
   for (int i = 0; i < n; i++)
      printf("%d ", *(ary + i));
   printf("\n");
   printf("합: %d  평균: %.1f\n", sum, (double)sum / n);

   //동적 메모리 해제
   free(ary);

   return 0;
}
```

할당된 저장공간의 주소값 연산식이다.

할당된 저장공간의 참조 연산식이다.

설명

```
10    동적메모리 할당으로 확보할 저장공간의 수를 표준입력으로 받아 변수 n에 저장
12    동적메모리 할당의 주소를 저장하기 위한 포인터
14    함수 malloc()으로 동적메모리 할당 후 int 형 포인트인 pi에 저장하기 위해
      자료형 변환(int *)이 필요, 할당되는 메모리의 총 크기는 인자인 sizeof(int)*n,
      오류로 ary가 NULL이면 종료
24    실제 표준입력으로 동적메모리 할당된 공간에 값을 저장
25    24 행에서 저장된 모든 값들을 변수 sum에 계속 저장
29~30 동적메모리 할당된 공간의 모든 값을 출력, *(ary + i)는 ary[i]로도 참조 가능함
32    합과 평균을 출력
35    동적메모리 공간의 메모리 해제
```

실행결과

```
입력할 점수의 개수를 입력 >> 6
6개의 점수 입력 >> 98 78 67 99 82 87
입력 점수: 98 78 67 99 82 87
합: 511  평균: 85.2
```

함수 calloc()

- **함수 calloc()은 메모리 공간을 확보**
  - 초기값을 자료형에 알맞게 0을 저장
    - 함수 malloc()은 메모리 공간을 확보하고 초기값을 저장하지 않으면
      - 쓰레기값이 저장
  - 함수 원형이 헤더 파일 stdlib.h에 정의
    - 앞의 인자: 할당되는 원소의 개수, 뒤의 인자: 한 원소의 크기

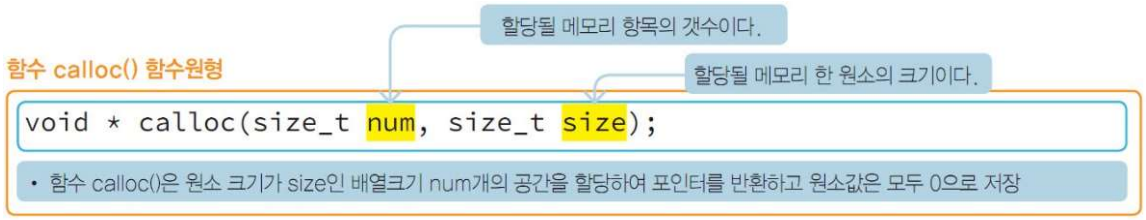


그림 16-9 함수 calloc()의 함수원형과 사용 방법

- **함수 호출 calloc(3, sizeof(int))로 할당**
  - int형 원소 3개
  - 저장공간에 기본값 0

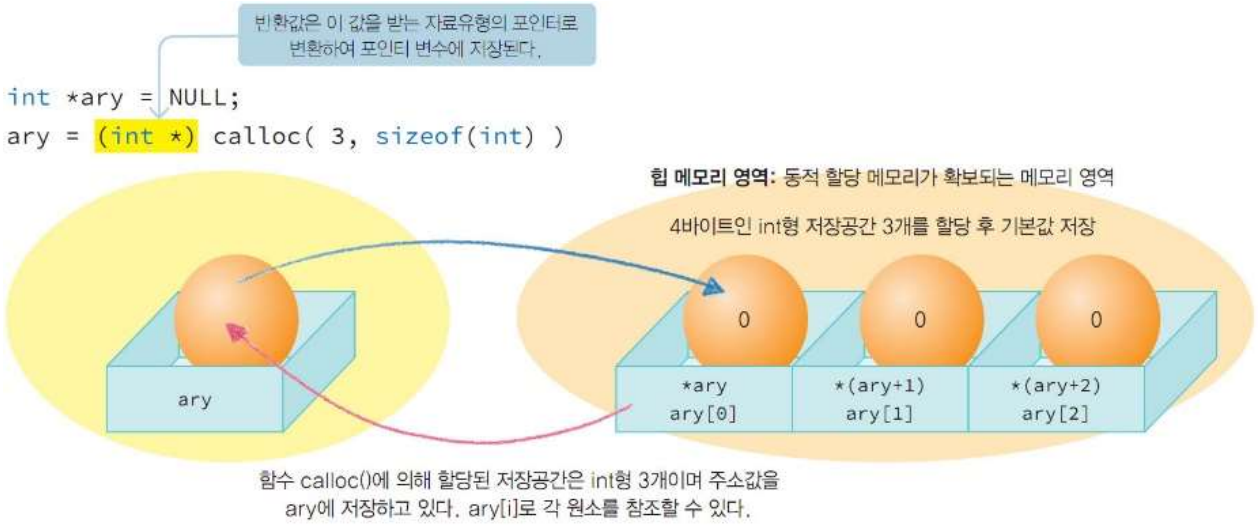


그림 16-10 함수 calloc()의 저장공간 할당과 기본값 저장

함수 calloc()

예제 calloc.c

실습예제 16-3 calloc.c

함수calloc()을 이용하여 int형 원소 3개를 할당

```c
// file: calloc.c
#include <stdio.h>
#include <stdlib.h>
void myprintf(int *ary, int n);

int main(void)
{
   int *ary = NULL;
   //동적메모리 할당
   if ((ary = (int *)calloc(3, sizeof(int))) == NULL)
   {
      printf("메모리 할당이 문제가 있습니다.\n");
      exit(EXIT_FAILURE);
   }
   myprintf(ary, 3); //모두 기본값인 0 출력

   //동적메모리 해제
   free(ary);
   myprintf(ary, 3); //모두 쓰레기값 출력

   return 0;
}

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

정수 1로서 헤더파일 stdlib.h에 정의되어 있는 상수이다.

할당된 저장공간의 참조 연산식이다.

설명

- 동적메모리 할당을 위해 함수 calloc()을 사용, 첫 인자 3은 동적메모리 할당 개수
- EXIT_FAILURE는 정수 1로서 헤더파일 stdlib.h에 정의되어 있는 상수
- 함수 calloc()으로 할당되는 메모리의 초기값은 모두 0으로 저장된 것을 확인
- 동적메모리 공간의 메모리 해제
- 동적메모리 공간의 메모리 해제 이후 메모리 공간에는 아무 의미 없는 쓰레기값이 있는 것을 확인
- 함수 myprintf()에서 첫 번째 매개변수 출력할 첫 주소이며, 두 번째 인자는 출력할 원소 수
- 동적메모리 할당된 공간의 모든 값을 출력, *(ary + i)는 ary[i]로도 참조 가능함

실행결과

```
ary[0] = 0
ary[1] = 0
ary[2] = 0
ary[0] = -572662307
ary[1] = -572662307
ary[2] = -572662307
```

함수 realloc()

- **이미 확보한 저장공간을 새로운 크기로 변경**
  - 함수 realloc()에 의하여 다시 확보하는 영역
    - 기존의 영역을 이용하여 그 저장 공간을 변경하는 것이 원칙
      - 새로운 영역을 다시 할당하여 이전의 값을 복사할 수도 있음
    - 성공적으로 메모리를 할당하면 변경된 저장공간의 시작 주소를 반환
      - 실패하면 NULL을 반환
    - 인자
      - 첫 인자: 변경할 저장공간의 주소
      - 두 번째 인자: 변경하고 싶은 저장공간의 총 크기
- **함수 realloc()에서 첫 번째 인자가 NULL**
  - 함수 malloc()과 같은 기능을 수행
  - 즉 지정된 크기만큼의 새로운 공간을 할당

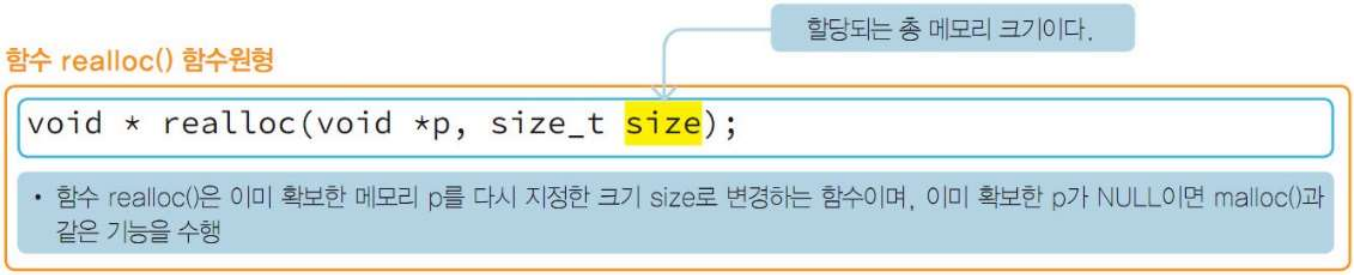


그림 16-11 함수 realloc()의 함수원형과 사용 방법

함수 realloc() 사용

- **cary는 int 형 3 개의 저장공간의 시작 주소를 저장**
  - 모든 변수의 값은 기본값인 0으로 저장
- **함수 realloc()을 이용하여 변수 cary가 가리키는 메모리의 내용을 수정**
  - 크기가 4인 배열로 재할당하는 구문과 메모리 구조
  - 대부분 reary와 cary 값이 동일
  - 함수 realloc()에 의하여 확장되는 공간은 malloc()과 같이 기본값 0이 저장되지 않음
    - reary[3]의 내용을 none으로 표기한 것은 기본값 0이 저장되지 않았음을 의미

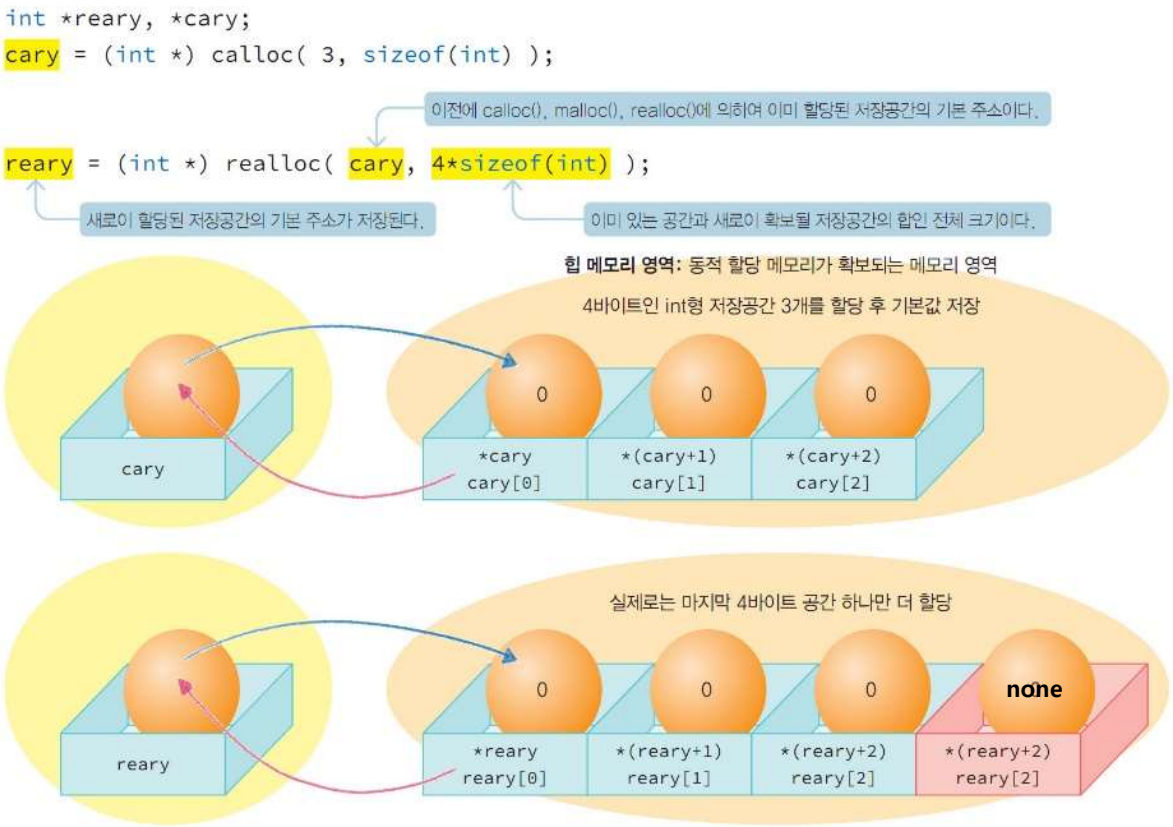


그림 16-12 함수 realloc()에 의한 메모리 공간의 재할당

함수 realloc() 예제

예제 realloc.c

실습예제 16-4 realloc.c

함수realloc()을 이용하여 이미 할당된 메모리를 변경

```c
// file: realloc.c
#include <stdio.h>
#include <stdlib.h>
void myprintf(int *ary, int n);

int main(void)
{
   int *reary, *cary;
   if ((cary = (int *)calloc(3, sizeof(int))) == NULL)
   {
      printf("메모리 할당이 문제가 있습니다.\n");
      exit(EXIT_FAILURE);
   }
   if ((reary = (int *)realloc(cary, 4 * sizeof(int))) == NULL)
   {
      printf("메모리 할당이 문제가 있습니다.\n");
      exit(EXIT_FAILURE);
   }

   myprintf(reary, 4); //앞 3개는 기본값인 0 출력, 마지막 하나는 다른 값
   free(reary);

   return 0;
}

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

이전에 할당된 공간이 3개이므로 실제로는 하나 더 할당된다.

설명

- reary는 realloc()으로 cary는 calloc()으로 동적메모리 할당되는 메모리 공간의 주소를 저장
- 동적메모리 할당을 위해 함수 calloc()을 사용, 자료형 int 3개를 할당
- EXIT_FAILURE는 정수 1로서 헤더파일 stdlib.h에 정의되어 있는 상수
- 동적메모리 할당을 위해 함수 recalloc()을 사용, 이전에 calloc()으로 할당된 공간에서 하나 더 늘려 int형 4개를 할당, 추가된 하나의 공간에는 기본값이 저장되지 않고 쓰레기값이 저장
- 함수 recalloc()으로 할당된 주소 reary에서 4개의 int를 출력, 앞 3개는 기본값인 0 출력, 마지막 하나는 쓰레기값 출력

실행결과

```
ary[0] = 0 ary[1] = 0 ary[2] = 0 ary[3] = -842150451
```

확장된 메모리 공간에는 기본값이 저장되지 않아 쓰레기값이 출력된다.

자기참조 구조체 정의

- **자기참조 구조체(self reference struct)**
  - 구조체의 멤버 중의 하나가 자기 자신의 구조체 포인터 변수를 갖는 구조체
- **구조체 selfref**
  - 멤버로 int 형 n과 struct selfref * 형 next로 구성
    - 즉 멤버 next의 자료형은 지금 정의하고 있는 구조체의 포인터 형
  - 구조체 selfref는 자기 참조 구조체
    - 구조체의 멤버 중의 하나가 자기 자신의 구조체 포인터 변수
  - 구조체는 자기 자신 포인터를 멤버로 사용할 수 있으나
    - 자기 자신은 멤버로 사용 불가능

```
struct selfref {
    int n;
    struct selfref *next;
    //struct selfref one;    //컴파일 오류 발생
}
```

error C2079: 'one'은(는) 정의되지 않은 struct 'selfref'을(를) 사용합니다.

그림 16-13 자기 참조 구조체

간단한 연결 리스트

- **연결 리스트(linked list)**
  - 자기 참조 구조체는 동일 구조체의 표현을 여러 개 만들어 연결할 수 있는 기능

```c
//❶ 우선 구조체 struct selfref를 하나의 자료형인 list 형으로 정의
typedef struct selfref list;

//❷ 두 구조체 포인터 변수 first와 second를 선언한 후,
//   함수 malloc()을 이용하여 구조체의 멤버를 저장할 수 있는 저장공간을 할당
list *first = NULL, *second = NULL;
first = (list *)malloc(sizeof(list));
second = (list *)malloc(sizeof(list));

//❸ 구조체 포인터 first와 second의 멤버 n에 각각 정수 100, 200을 저장하고,
//   멤버 next에는 각각 NULL을 저장
first->n = 100;
first->next = NULL;
second->n = 200;
second->next = NULL;
```

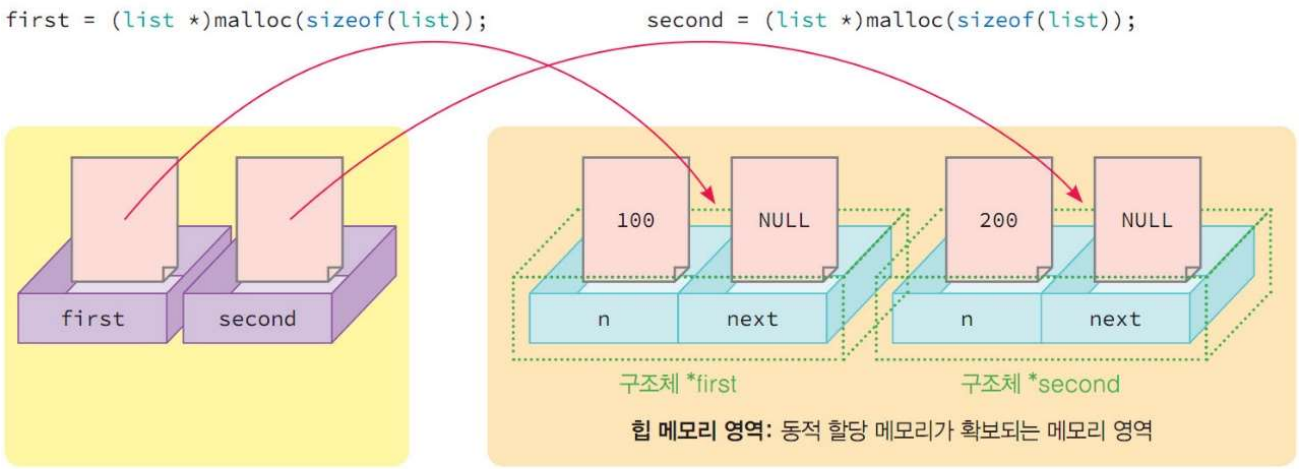


그림 16-14 구조체 멤버로 포인터가 있는 구조체의 동적 할당

구조체 포인터의 활용

- **위 그림으로는 아직 구조체 *first와 *second는 아무 관련이 없음**
  - 만일 구조체 *first가 다음 *second 구조체를 가리키도록 하려면
    - 문장 first->next = second; 가 필요
      - 즉 구조체 *first의 멤버 next에 구조체 포인터 second의 내용인 주소값을 저장
    - 이제 first가 가리키는 구조체 멤버 next를 사용하여 다음 구조체를 연결

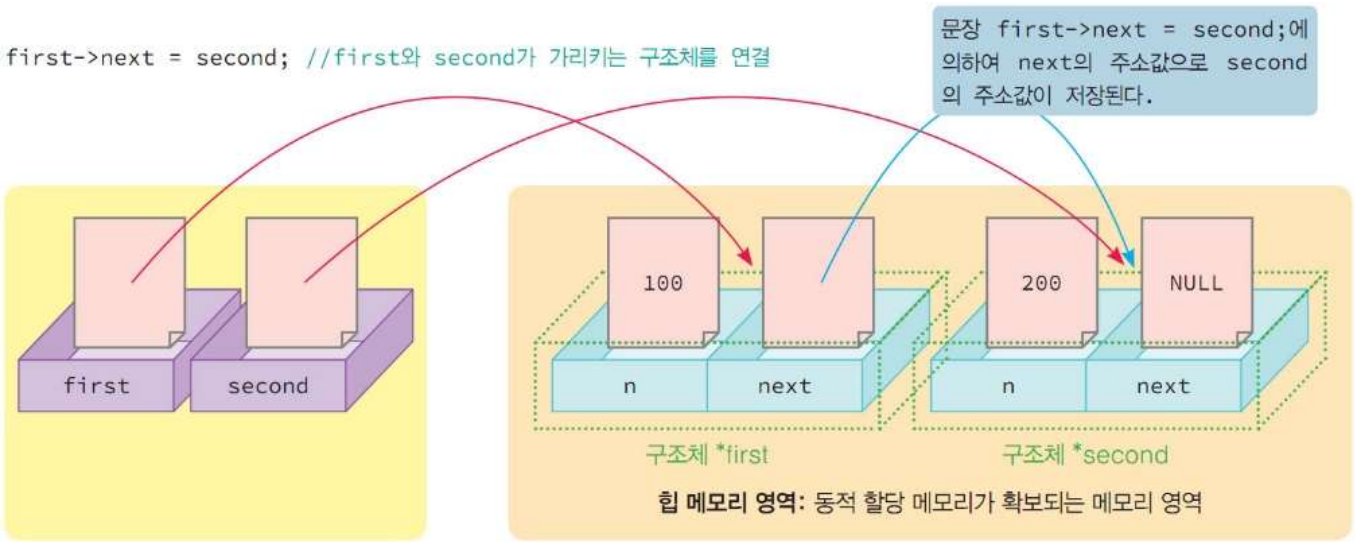


그림 16-15 구조체 멤버의 포인터에 연결하려는 구조체의 주소값을 저장

리스트 기초

예제 selfrefstruct.c

실습예제 16-5 selfrefstruct.c

자기참조 구조체를 이용한 연결 리스트의 기본 학습

```c
// file: selfrefstruct.c
#include <stdio.h>
#include <stdlib.h>

//자기참조 구조체 정의
struct selfref {
   int n;
   struct selfref *next;
   //struct selfref one;   //컴파일 오류 발생
};

int main(void)
{
   //❶ 우선 구조체 struct selfref를 하나의 자료형인 list 형으로 정의
   typedef struct selfref list;

   //❷ 두 구조체 포인터 변수 first와 second를 선언한 후,
   //   함수 malloc()을 이용하여 구조체의 멤버를 저장할 수 있는 저장공간을 할당
   list *first = NULL, *second = NULL;
   first = (list *)malloc(sizeof(list));
   second = (list *)malloc(sizeof(list));

   //❸ 구조체 포인터 first와 second의 멤버 n에 각각 정수 100, 200을 저장하고,
   //   멤버 next에는 각각 NULL을 저장
   first->n = 100;
   first->next = NULL;
   second->n = 200;
   second->next = NULL;

   //구조체 *first가 다음 *second 구조체를 가리키도록 하는 문장
   first->next = second;

   printf("구조체 크기= %d\n\n", sizeof(list));
   printf("첫 번째 구조체: ");
   printf("\t자료의 주소값(first) = %u\n", first);
   printf("\t자료값(first->n) = %d\n", first->n);
   printf("\t자료값(first->next) = %u\n", first->next);
   printf("\t자료값(first->next->n) = %d\n\n", first->next->n);

   printf("두 번째 구조체: ");
   printf("\t자료의 주소값(second) = %u\n", second);
   printf("\t자료값(second->n) = %d\n", second->n);
   printf("\t자료값(second->next) = %u\n", second->next);

   //동적메모리 할당 해제
   free(first);
   free(second);

   return 0;
}
```

08 멤버 next의 자료형이 struct selfref *로 자기 자신 유형의 포인터로 선언,
이와 같이 구조체의 멤버 중의 하나가 자기 자신의 구조체 포인터 변수를 갖는 구조체를
자기참조 구조체(self reference struct)라 함
20 자료형 변환 (list *)은 반드시 필요하며, first가 가리키는 곳이 구조체가 있는 메모리 공간
25 포인터 first로 멤버를 참조하려면 연산자 ->을 사용
31 first가 가리키는 구조체의 next에 second의 내용 값인 주소를 저장하므로 구조체 *first가
다음 *second 구조체를 가리키도록 함
37 first->next의 주소값이 바로 41 행의 second의 주소값과 동일
38 first->next->n은 바로 second->n과 동일한 200이 출력
46~47 할당된 동적메모리 해제

```
구조체 크기 = 8

첫 번째 구조체:          자료의 주소값(first) = 5577656
        자료값(first->n) = 100
        자료값(first->next) = 5577712
        자료값(first->next->n) = 200

두 번째 구조체:          자료의 주소값(second) = 5577712
        자료값(second->n) = 200
        자료값(second->next) = 0
```

LAB 직원을 위한 자기참조 구조체 구현

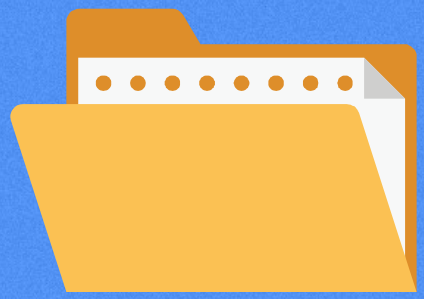
- **직원의 정보를 표현하는 struct employee를 정의**
  - 2개의 구조체를 동적 메모리로 할당하여 적절한 정보를 입력
  - 구조체 employee는 자기 자신을 가리키는 자기참조 구조체
    - 하나의 구조체 필드 next가 다른 구조체를 가리키도록 구현
- **결과**
  - 20146730 고윤호 2000000
  - 20146729 신민아 1000000
  - 20146729 신민아 1000000

Lab 16-1 structemployee.c

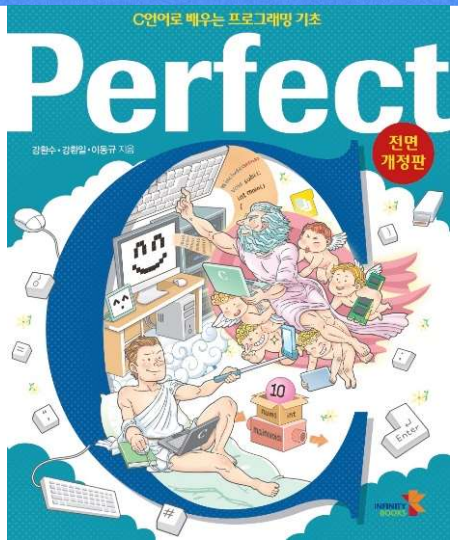
```c
// file: structemployee.c
#define _CRT_SECURE_NO_WARNINGS //scanf() 오류를 방지하기 위한 상수 정의
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//자기참조 구조체 정의
typedef struct employee {
   int id;
   char *name;
   ____________________;
   ____________________;
} employee;

int main(void)
{
   employee *one = (employee *)malloc(sizeof(employee));
   employee *you = (employee *)malloc(sizeof(employee));

   one->id = 20146729;
   one->salary = 1000000;
   one->name = (char *)malloc(strlen("신민아") + 1);
   strcpy(one->name, "신민아");

   you->id = 20146730;
   you->salary = 2000000;
   you->name = ____________________________________);
   strcpy(_____________________________);

   you->next = one;
   printf("%d %s %d\n", you->id, you->name, you->salary);
   printf("%d %s %d\n", one->id, one->name, one->salary);
   printf("%d %s %d\n", you_____________,
          you_____________, you_____________);

   //동적메모리 할당 해제
   free(one);
   free(you);

   return 0;
}
```

정답

```c
int salary;
struct employee *next;
you->name = (char *)malloc(strlen("고윤호") + 1);
strcpy(you->name, "고윤호");
printf("%d %s %d\n", you->next->id, you->next->name, you->next->salary);
```



02. 연결 리스트



배열의 장단점

- **프로그램 언어를 개발된 순서인 C, C++, Java 순서로 나열 방법**
  - 표
  - 배열

1	C
2	C++
3	Java

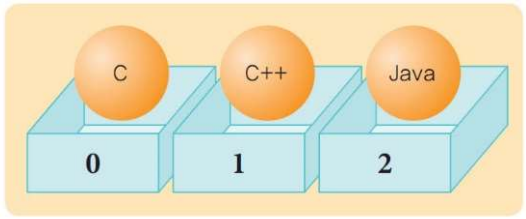


그림 16-16 개발된 순서로 나열한 프로그래밍 언어이름과 배열 표현

- **자료를 순차적으로 저장하기 가장 쉬운 방법: 배열**
  - 배열이름과 첨자(index)를 사용, 원하는 원소를 직접 임의참조(random access)가 가능
  - 단점1
    - 컴파일 전에 배열의 크기가 이미 결정, 실행 중간에 배열크기 수정 불가능
  - 단점2
    - 맨 앞이나 중간에 새로운 자료를 삽입, 삽입되는 자료 이후의 원소가 모두 이동
  - 단점3
    - 중간에 하나 삭제하는 경우도 마찬가지 어려움

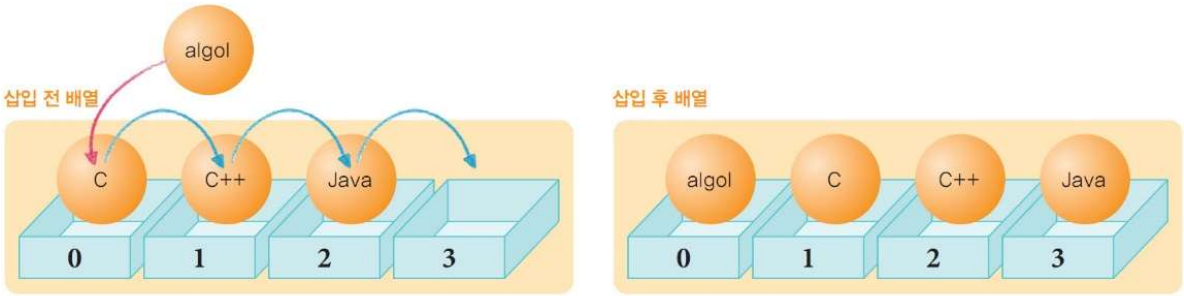


그림 16-17 배열 처음 위치에 새로운 원소 삽입

연결 리스트 개요

- **연결 리스트**
  - 배열과 함께 순차적 자료 표현에 적합한 구조
- **연결 리스트 예**
  - 헤드(head)는 "미수"를 가리키고
    - "미수"는 다시 "현순"을 가리키고
    - 계속해서 "윤원", "현화", "수성", "나혜"
    - 그리고 다시 나혜는 마지막이라 가리키는 사람이 없는 것(NULL)과 같은 구조

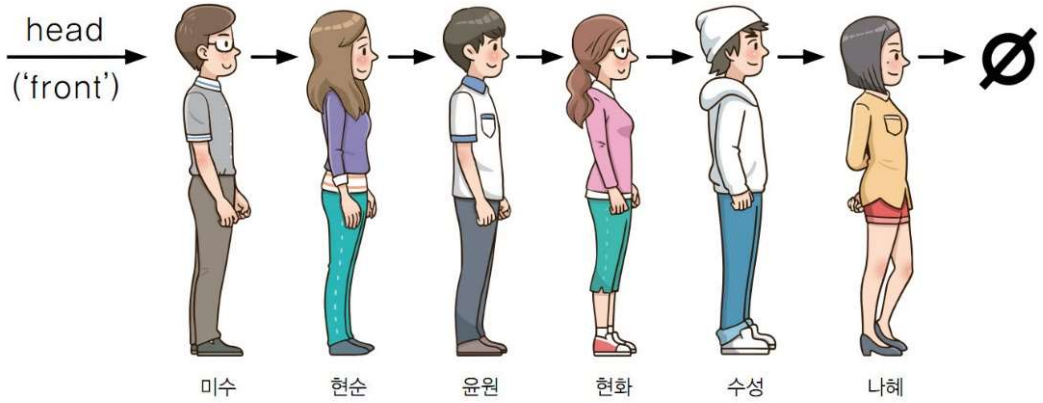


그림 16-18 연결 리스트의 이해

  - 헤드에서 시작하여 가리키는 곳을 계속 따라가면 순차적 자료를 표현
  - 원소인 노드(node)가 순차적으로 연결된 자료구조
    - 노드는 배열의 원소에 해당
    - 자료(data)와 링크(link)로 구성
    - 노드는 자기참조 구조체로 정의
  - 첨자대신 링크(link)라는 포인터로 다음 노드를 가리키는 구조

노드의 표현

- **노드의 자료: 필요한 여러 변수의 조합으로 구성**
  - 노드의 링크: 자기 구조체의 포인터로 구현
- **헤드(head)**
  - 항상 첫 번째 노드를 가리키는 포인터
- **테일(tail)**
  - 마지막 노드를 가리키는 포인터

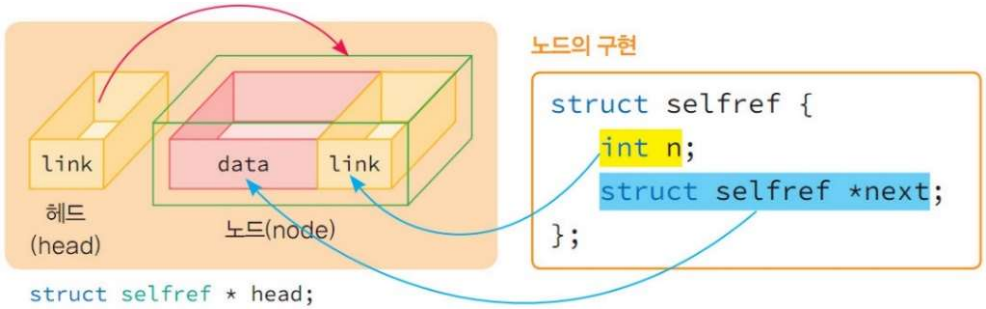


그림 16-19 연결 리스트의 헤드와 노드

- **연결 리스트로 표현한 그림**
  - 헤드 포인터 노드에서 시작해서 화살표를 따라 이동하면 자료를 순서대로 참조
  - 연결 리스트에서 마지막 노드의 링크는 NULL로 저장
    - 만일 연결 리스트에 노드가 하나도 없다면 헤드는 NULL

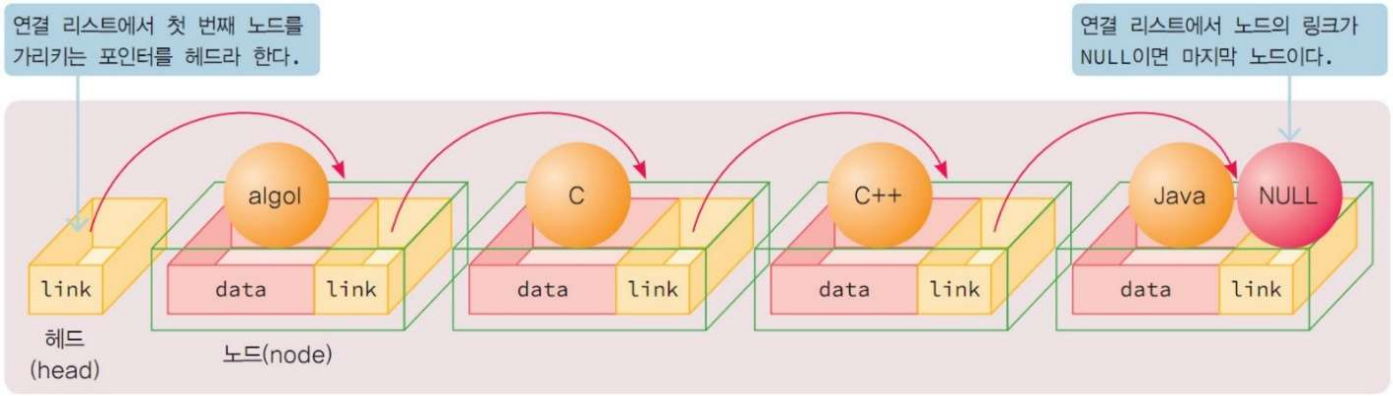


그림 16-20 연결 리스트의 헤드와 노드

연결 리스트 장단점

- **연결 리스트 장점**
  - 항목 수를 프로그램 내부에서 메모리가 허용하는 한 늘릴 수 있다는 것
    - 배열과는 달리 프로그램 실행 전에 미리 기억장소를 확보해 둘 필요가 없음
  - 프로그램 실행 중이라도 필요할 때 노드를 동적으로 생성
    - 기존의 연결 리스트에 삽입 또는 추가 가능
  - 기억장소를 비순차적으로 사용
    - 연결 리스트에서 중간에 노드를 삽입 또는 삭제하더라도 배열에 비하여 다른 노드에 영향을 덜 미침
  - 결론적으로 연결 리스트는 동적으로 노드를 생성
    - 리스트 크기의 증가 감소에 따라 효율적으로 대처할 수 있으며
    - 노드의 삽입과 삭제와 같은 자료의 재배치를 빠르게 처리
- **단점**
  - 배열에 비하여 임의 접근(random access)에 많은 시간이 소용
  - 노드 검색은 헤드에서부터 링크를 따라가는 순차적 검색만이 가능

노드 순회와 추가

- **노드 순회(node traversal)**
  - 연결 리스트에서 모든 노드를 순서대로 참조하는 방법
  - 헤드부터 계속 노드 링크의 포인터로 이동하면 가능
    - 링크가 NULL이면 마지막 노드
    - 노드 순회 방법을 이용하여 각 노드의 자료를 참조할 수 있으며 원하면 출력도 가능
- **노드 추가**
  - 새로운 노드를 하나 생성, 연결 리스트의 마지막 노드로 추가
  - "C#" 노드를 만들어 기존의 연결 리스트에 추가하는 방법
    - ❶ 첫 번째로 추가할 노드를 먼저 생성한 후, 자료를 저장하고 링크를 NULL로 저장
    - ❷ 기존 연결 리스트를 순회하여 마지막으로 이동
    - ❸ 마지막 노드의 링크를 새로 생성한 노드의 주소값으로 저장하여 연결 리스트의 마지막 노드로 연결

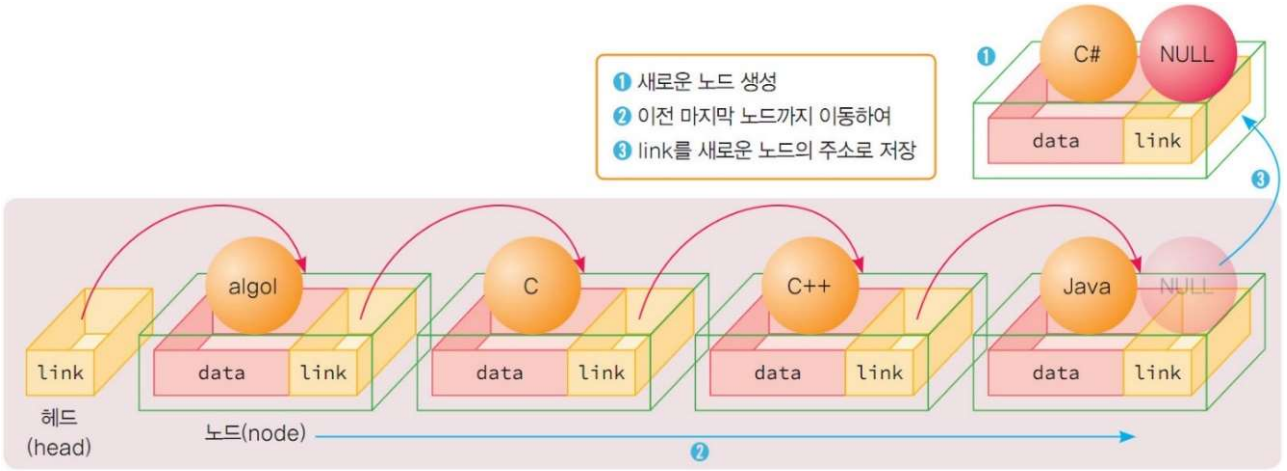


그림 16-21 연결 리스트에서 노드 추가

노드 삽입

- **연결 리스트 중간에 한 노드를 삽입하는 과정**
  - 기존의 연결 리스트인 노드 "C"와 노드 "C++" 사이에 노드 "Objective-C"를 삽입하는 과정
    - ❶ 가장 먼저 삽입 노드를 동적으로 생성하여 적당한 자료를 저장
    - ❷ 이제 삽입하려는 바로 이전 노드인 노드 "C"로 이동
    - ❸ 삽입하는 "Objective-C" 노드의 링크에 노드 "C"의 링크를 저장
    - ❹ 다음에는 노드 "C"의 링크를 새로 삽입하는 "Objective-C" 노드를 가리키도록 삽입하는 "Objective-C" 노드의 주소값을 저장

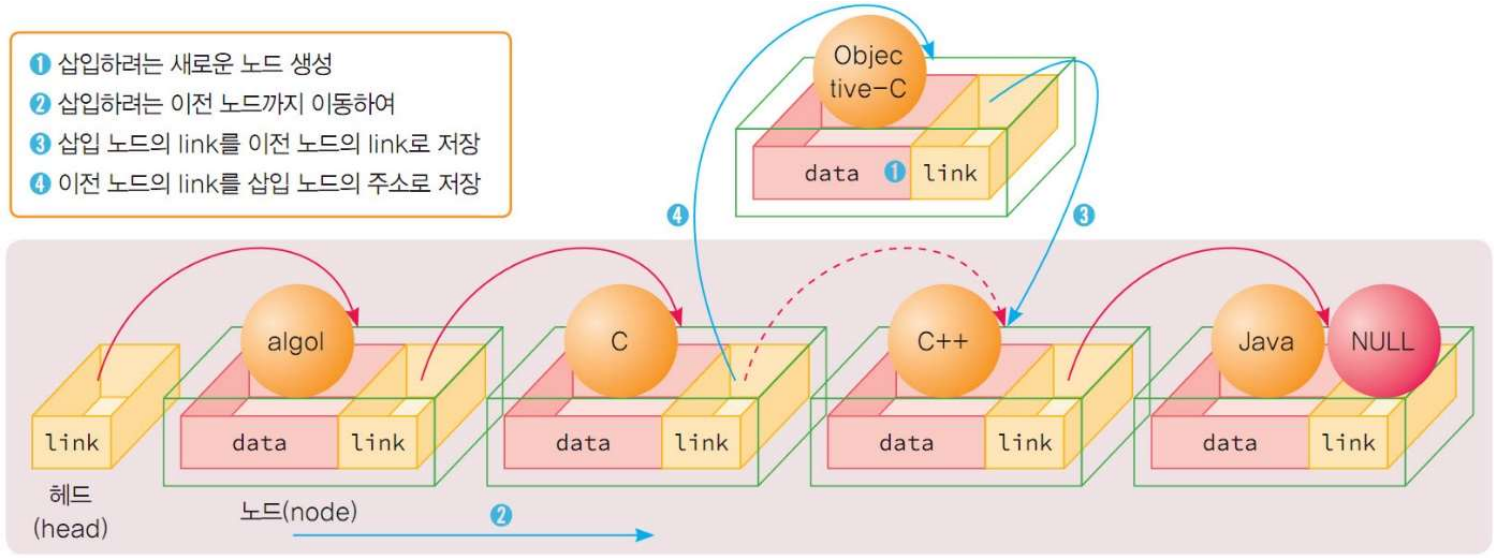


그림 16-22 연결 리스트에서 삽입

- **노드 삽입의 다른 조건**
  - 삽입 노드의 위치를 알려주는 방법
    - 위 과정 중에서 2번의 이동이 없이 바로 노드 "C"를 알려주고 노드 "C"의 다음에 삽입
    - 위 과정 중 2번을 제외하고 처리하면 새로운 노드를 삽입 가능

노드 삭제

- **연결 리스트에서 노드 하나를 삭제하는 과정**
  - 기존의 연결 리스트에서 노드 "C++"를 삭제하려면 가장 먼저 삭제하려는 노드 바로 이전 노드 "C"로 이동
  - 삭제하려는 노드 "C++"를 포인터 변수에 저장
  - ❶ 노드 "C"의 링크를 삭제하려는 노드 "C++"의 링크 노드 값으로 저장
  - ❷ 이제 마지막으로 삭제 노드 "C++"를 메모리에서 제거

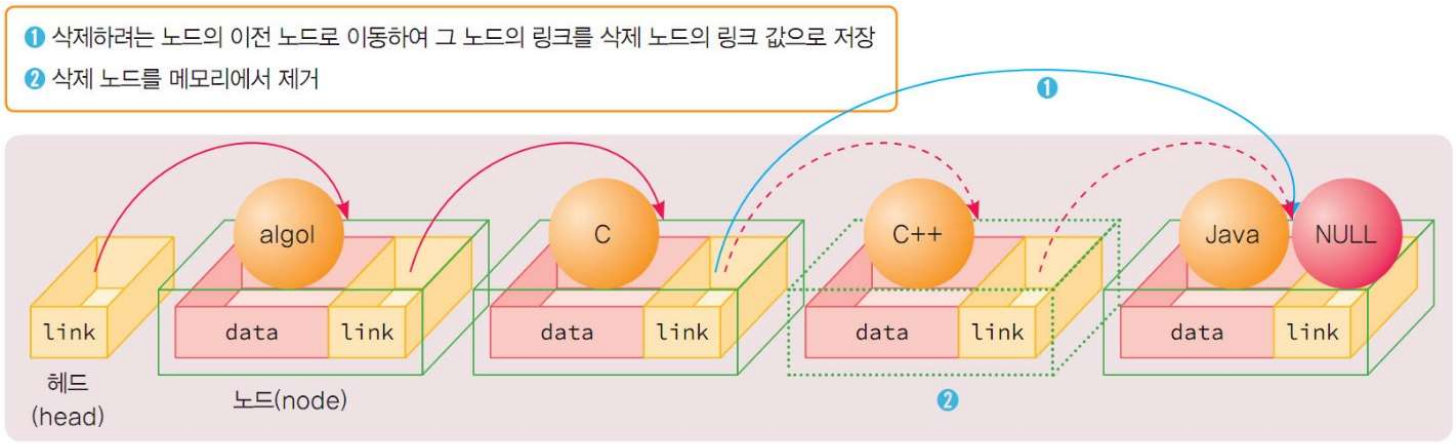


그림 16-23 연결 리스트에서 노드 삭제

구조체 정의와 생성

- **연결 리스트의 노드를 표현하는 구조체 struct linked_list를 정의**
  - 구조체 struct linked_list의 멤버
    - 문자열을 저장할 char * 변수인 name
    - 그리고 다른 구조체를 가리킬 포인터 next를 구성
  - 문장 typedef를 이용
    - struct linked_list를 간단히 NODE로 정의
    - 이 NODE의 포인터를 LINK로 정의

```
struct linked_list {
    char *name;
    struct linked_list   *next;
};
typedef   struct linked_list NODE;
typedef   NODE *LINK;
```

그림 16-24 구조체와 관련 자료형 정의

구조체 정의와 생성

- **함수 createNode()**
  - 구조체 NODE를 하나 생성해 멤버 name에 인 자로 전달되는 문자열를 저장
  - 노드 하나를 생성해서 " Java" 문자열을 저장
    - 함수호출 createNode("Java")로 수행
- **구현**
  - 변수 cur가 가리키는 구조체의 멤버 name은 문자 포인터
  - cur->name에 저장할 문자의 수만큼 메모리를 할당
    - 함수 malloc()에서 할당할 메모리의 크기
      - strlen(str)+1개의 문자 크기를 확보
  - 함수 strcpy()를 이용
    - "Java"가 저장된 문자열 str을 cur->name에 복사
  - 현재 생성된 노드의 멤버 cur->next
    - 초기에 NULL을 입력
    - 필요하면 나중에 다른 구조체를 가리키도록 대입

```
//노드를 생성하는 함수
LINK createNode(char *name)
{
   //새로 생성되는 노드의 주소를 저장할 변수 cur를 선언
   LINK   cur;
   //함수 malloc()으로 할당된 저장공간의 주소를 변수 cur에 저장

   cur = (LINK) malloc(sizeof(NODE));
   ...
   //언어 이름을 저장할 문자배열을 동적 할당하여 name에 저장
   cur->name = (char *)malloc(sizeof(char) * (strlen(name) + 1));
   strcpy(cur->name, name);
   //다음 노드는 모르므로 NULL로 저장
   cur->next = NULL;

   return cur;
}
```

그림 16-25 노드를 생성하는 함수 createNode()

구조체 노드 추가(1)

- **연결 리스트에 구조체 포인터 cur를 추가하는 함수 append()**
  - cur 노드를 연결 리스트 head의 마지막 노드에 추가하는 함수
  - 인자
    - 시작 노드를 가리키는 head
    - 추가될 노드인 cur
  - 방법
    - 연결 리스트는 인자인 노드 head가 가리키는 노드에서 시작
    - 시작 노드를 반환

```
//cur 노드를 연결 리스트 head의 마지막 노드에 추가하는 함수
LINK append(LINK head, LINK cur)
{
    //지역 변수 nextNode를 선언하고 초기값으로 head를 저장
    LINK nextNode = head;
    ...
    return head;
}
```

그림 16-26 함수 append() 구현

구조체 노드 추가(2)

- **함수 append()를 이용하여 노드가 하나도 없는 연결 리스트에 노드 C를 추가하는 과정**
  - 연결 리스트의 마지막 노드를 찾아가는데 필요한 지역 변수 nextNode를 선언
    - 초기값으로 head를 저장
  - 만일 현재 연결 리스트에 아무 노드가 없는 상태라면
    - 무조건 새로 추가되는 노드가 첫 번째 노드가 될 것
      - head에 cur를 대입하고 head를 반환하면서 함수를 종료

```
//지역 변수 nextNode를 선언하고 초기값으로 head를 저장
LINK nextNode = head;
//만일 현재 헤드가 가리키는 것이 없다면, 즉 연결 리스트의 노드가 하나도 없는 경우
if (head == NULL)
{
   head = cur; //추가하려는 노드가 head가 됨
   return head;
}
```

실제 name은 문자 포인터이므로 문자열 "C"가 저장된 저장 공간의 주소값을 갖는다. 편의를 위해 이와 같이 간단히 표현한다.

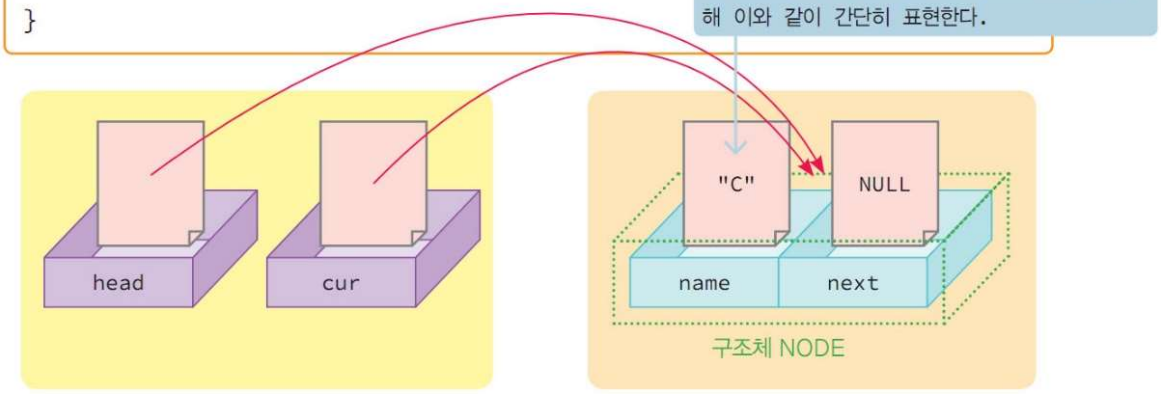


그림 16-27 노드가 전혀 없는 연결 리스트에 처음으로 C 노드를 추가

구조체 노드 추가(3)

- **연결 리스트 마지막 노드에 노드 C#을 추가하는 과정**
  - while() 모듈을 이용하면 nextNode는 멤버 next의 값이 NULL인 마지막 노드로 이동
  - nextNode는 처음에 head에서부터 마지막 노드를 찾아갈 때까지 계속 다음 노드를 가리키는 노드
  - 결국 멤버 next의 값이 NULL인 마지막 노드를 가리킴
  - 이제 마지막 노드에 새로운 노드 C#을 연결
    - 마지막 모드를 가리키는 nextNode를 이용하여 nextNode -> next에 cur를 대입

```
//멤버 next가 NULL일 때까지 이동하여 마지막 노드까지 이동
while (nextNode->next != NULL)
{
   nextNode = nextNode->next;
}
//추가 노드를 현재 노드의 next에 저장
nextNode->next = cur;
```

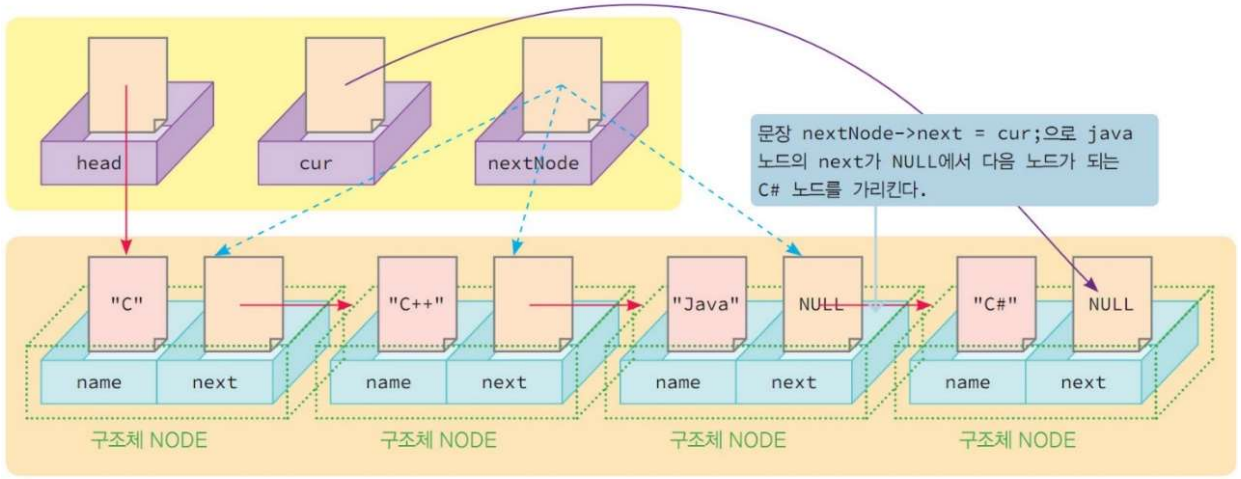


그림 16-28 연결 리스트에 C# 노드가 추가되어 수정된 연결 리스트

구조체 노드 출력

- **연결 리스트 자료 출력 함수 printList()**
  - 연결 리스트의 시작 노드를 가리키는 head 노드를 인자로 받아서 연결 리스트를 순회하면서 모든 노드의 자료를 출력하는 함수
  - 출력된 노드의 수를 반환

```
//연결 리스트의 모든 노드 출력 함수
int printList(LINK head)
{
    int cnt = 0; //방문한 노드의 수를 저장
    LINK nextNode = head;
    //nextNode를 이용하여 연결 리스트의 처음부터 끝까지 순회
    while (nextNode != NULL)
    {
        //리스트의 순서로 노드를 방문하여 방문 횟수와 문자열 자료를 출력
        ...

    }

    //총 노드 방문 횟수를 반환하고 함수를 종료
    return cnt;
}
```

그림 16-29 함수 printList()

프로그래밍 언어 종류를 연결 리스트로 구현

- **프로그래밍 언어 이름을 표준입력으로 받아 계속 연결 리스트에 추가하는 프로그램**
  - 2 개의 소스 파일
    - linkedlist.c, listlib.c
  - 하나의 헤더파일
    - linkedlist.h로 구성
  - 상대적으로 큰 규모의 프로그램을 작성하려면 소스파일도 여러 개로 나누는 것이 필요
  - 헤더파일도 프로그래머가 직접 만들 필요가 있음
    - 프로그래머가 직접 만든 헤더파일을 사용자 정의 헤더파일
      - 사용자정의 헤더파일도 시스템 헤더파일과 같이 함수원형, 매크로, 자료형 재정의에 관련된 문장으로 구성
- **비주얼 스튜디오 프로젝트 Linked List로 구현**
  - 파일 linkedlist.c
    - 함수 main() 구현
  - 파일 listlib.c
    - 연결 리스트에 필요한 함수 createNode(), append(), printList() 구현
  - 헤더파일 linkedlist.h
    - 필요한 시스템 헤더 파일을 삽입
    - 함수 createNode(), append(), printList()의 함수원형을 정의
    - 구조체 linked_list 정의와 관련된 자료유형을 정의

Linked List 프로젝트: 소스 3개 파일, 목적파일, 실행파일

- 헤더파일은 컴파일이 필요한 파일이 아님
- 프로그램 소스 2개에서 컴파일이 성공
  - 목적파일
    - linkedlist.obj
    - listlib.obj가 생성
- 다시 링크가 성공
  - 프로젝트 이름의 실행파일 Linked List.exe가 생성

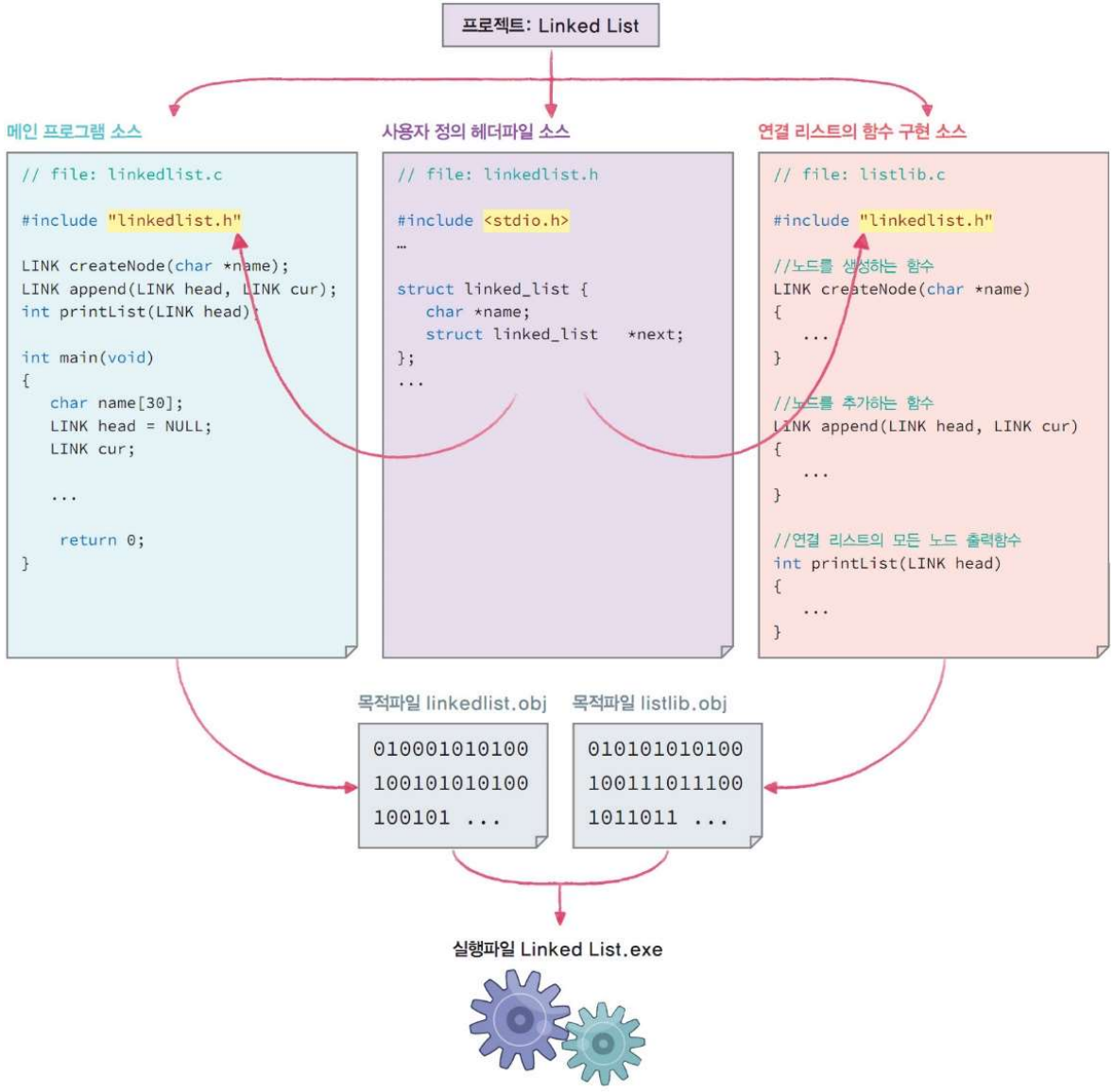


그림 16-30 프로젝트 linkedlist의 소스파일과 실행파일

프로그래밍 언어 종류를 연결 리스트로 구현

- **사용자 정의 헤더파일 linkedlist.h**
  - 표준입출력에 필요한 헤더파일 <stdio.h>의 삽입 문장 기술
  - 두 구현 소스파일에서 헤더파일 linkedlist.h을 삽입
  - 지시자 #include에서 사용자정의 헤더 파일을 삽입 방법
    - 큰 따옴표를 사용

```
#include "linkedlist.h"    //"사용자헤더파일.h" 기술
#include <stdio.h>         //<시스템헤더파일.h> 기술
```

그림 16-31 지시자 #include에서 〈시스템 헤더파일〉과 "사용자 헤더파일" 차이

- **비주얼 스튜디오에서 사용자 헤더 파일과 소스 파일 추가**
  - 프로젝트의 [헤더 파일]에서 오른쪽 버튼을 눌러 나온 메뉴 [추가/새항목]을 선택
  - [새 항목 추가]에서 헤더 파일을 선택
    - 사용자 헤더파일 이름 linkedlist를 입력
  - 소스파일 추가는 파일이 필요한 만큼 가능
    - 본 프로젝트: 2 개의 소스 파일 linkedlist.c, linkedlib.c
  - 솔루션 탐색기
    - 프로젝트 linked list 에 추가된 소스파일과 헤더파일을 확인

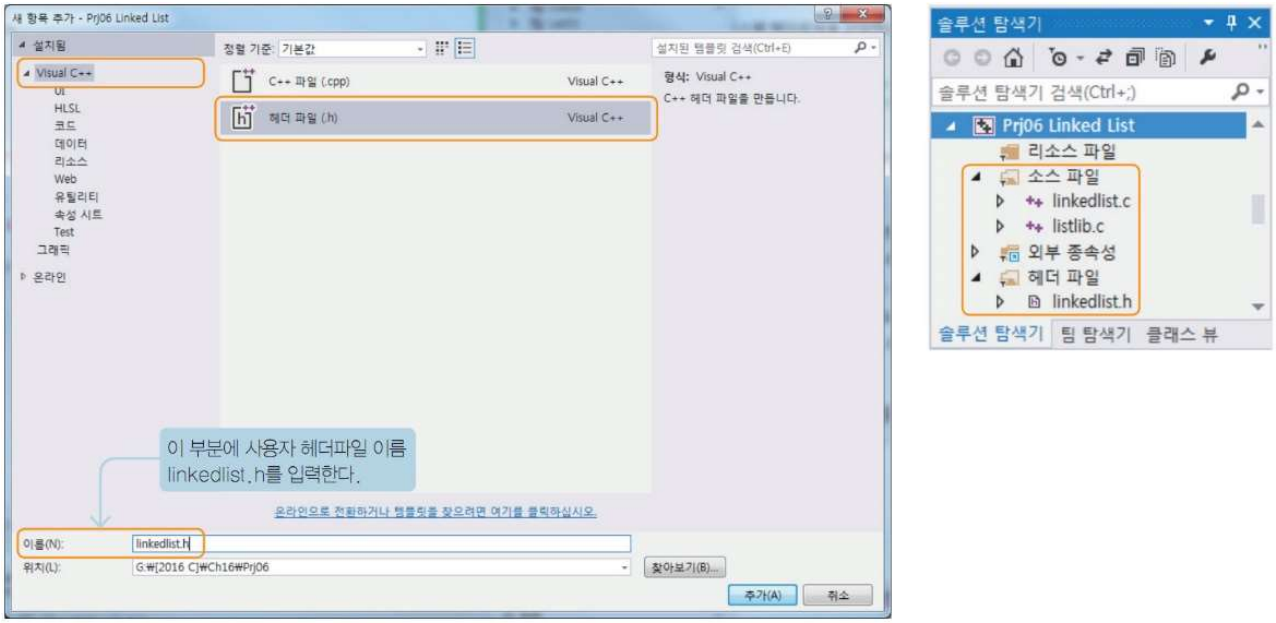


그림 16-32 사용자 헤더파일을 추가하는 대화상자와 추가된 헤더파일 linkedlist.h

프로젝트 Linked List 구현 내용

- **함수 main()**
  - 변수 name[]은 표준입력으로 받은 프로그램 이름 문자열을 저장할 문자 배열
  - 변수 head는 연결 리스트의 헤드로 사용되고,
  - 변수 cur는 현재 새로이 생성된 노드를 가리키는 포인터
- **함수 gets()**
  - 표준입력으로 문자열을 받고, 문장 while () 문을 이용하여 ctrl + Z를 누를 때까지 표준입력을 계속 받도록
- **함수 createNode()**
  - 문자열 name과 같은 문자열이 저장된 구조체 노드를 생성하여 변수 cur에 대입
  - 변수 cur를 인자로 함수 append()를 호출하여 연결 리스트에 추가
  - 새로운 노드가 하나 추가될 때마다 연결 리스트의 모든 노드를 순차적으로 출력

헤더 파일

예제 linkedlist.h

- 사용자 정의 헤더파일

전체 프로그램

- 연결 리스트를 구현한 프로그램
- 구현 소스
  - linkedlist.c, listlib.c
- 헤더파일
  - linkedlist.h 파일

실습예제 16-6 linkedlist.h

연결 리스트를 생성하고 출력하는 프로그램을 위한 헤더파일

```
01  // file: linkedlist.h
02  #define _CRT_SECURE_NO_WARNINGS //scanf(), gets() 등 오류를 방지하기 위한 상수 정의
03  #include <stdio.h>
04  #include <stdlib.h>
05  #include <string.h>
06
07  //자기참조 구조체 정의
08  struct linked_list {
09     char *name;
10     struct linked_list *next;
11  };
12  //struct linked_list를 NODE로 재정의
13  typedef   struct linked_list NODE;
14  //NODE *를 LINK로 재정의
15  typedef   NODE * LINK;
```

함수 구현

예제 listlib.c

- 함수 3개 구현

실습예제 16-8 listlib.c

연결 리스트를 생성하고, 추가, 출력하는 함수3개를 구현한 소스

```c
// file: listlib.c
#include "linkedlist.h"

//노드를 생성하는 함수
LINK createNode(char *name)
{
   //새로 생성되는 노드의 주소를 저장할 변수 cur를 선언
   LINK   cur;
   //함수 malloc()으로 할당된 저장공간의 주소를 변수 cur에 저장
   cur = (LINK) malloc(sizeof(NODE));
   if (cur == NULL)
   {
      printf("노드 생성을 위한 메모리 할당에 문제가 있습니다.\n");
      return NULL;
   }
   //언어 이름을 저장할 문자배열을 동적 할당하여 name에 저장
   cur->name = (char *)malloc(sizeof(char) * (strlen(name) + 1));
   strcpy(cur->name, name);
   //다음 노드는 모르므로 NULL로 저장
   cur->next = NULL;

   return cur;
}

//cur 노드를 연결 리스트 head의 마지막 노드에 추가하는 함수
LINK append(LINK head, LINK cur)
{
   //지역 변수 nextNode를 선언하고 초기값으로 head를 저장
   LINK nextNode = head;
   //만일 현재 헤드가 가리키는 것이 없다면, 즉 연결 리스트의 노드가 하나도 없는 경우
   if (head == NULL)
   {
      head = cur; //추가하려는 노드가 head가 됨
      return head;
   }
   //멤버 next가 NULL일 때까지 이동하여 마지막 노드까지 이동
   while (nextNode->next != NULL)
   {
      nextNode = nextNode->next;
   }
   //추가 노드를 현재 노드의 next에 저장
   nextNode->next = cur;

   return head;
}

//연결 리스트의 모든 노드 출력 함수
int printList(LINK head)
{
   int cnt = 0; //방문한 노드의 수를 저장
   LINK nextNode = head;
   //nextNode를 이용하여 연결 리스트의 처음부터 끝까지 순회
   while (nextNode != NULL)
   {
      //리스트의 순서로 노드를 방문하여 방문 횟수와 문자열 자료를 출력
      printf("%3d번째 노드는 %s\n", ++cnt, nextNode->name);
      nextNode = nextNode->next;
   }

   //총 노드 방문 횟수를 반환하고 함수를 종료
   return cnt;
}
```

설명

17 문자열 마지막이 \0이 저장되려면 malloc()의 인자에서 sizeof(char) * (strlen(name) + 1)로 기술

57 nextNode를 다음 노드로 이동

함수 main() 구현

예제 linkedlist.c

- 표준입력으로 받은 이름으로 연결리스트 구성

실습예제 16-7 linkedlist.c

연결 리스트를 생성하고 출력하는 메인 프로그램

```c
// file: linkedlist.c
#include "linkedlist.h"

LINK createNode(char *name);
LINK append(LINK head, LINK cur);
int printList(LINK head);

int main(void)
{
   //표준입력으로 받은 프로그램 이름 문자열을 저장할 문자 배열
   char name[30];
   LINK head = NULL; //연결 리스트의 헤드로 사용
   LINK cur; //현재 새로이 생성된 노드를 가리키는 포인터

   printf("이름을 입력하고 Enter를 누르세요. >> \n");
   while (gets(name) != NULL)
   {
      cur = createNode(name);   //노드 동적 할당
      if (cur == NULL) {
         printf("동적메모리 할당에 문제가 있습니다.\n");
         exit(1);
      }
      head = append(head, cur);   //맨 뒤에 노드 추가
      printList(head);      //연결 리스트 모두 출력
   }

   return 0;
}
```

설명

```
02  사용자정의 헤더 파일을 삽입하려면 큰 따옴표를 사용하여 "linkedlist.h" 기술
16  ctrl + Z를 누르면 while() 종료
23  함수 append()를 호출하여 cur 노드를 맨 뒤에 추가
24  연결 리스트 모두 출력
```

실행결과

```
이름을 입력하고 Enter를 누르세요. >>
C
  1번째 노드는 C
C++
  1번째 노드는 C
  2번째 노드는 C++
C#
  1번째 노드는 C
  2번째 노드는 C++
  3번째 노드는 C#
^Z
```

LAB 노드 3개로 구성된 연결 리스트(1)

- **구조체 struct node**
  - int 형의 정보를 담는 x 필드
  - 다른 노드를 가리키는 next 필드로 구성
- **구현**
  - 먼저 노드 2개를 생성하여 연결 리스트로 연결
  - 포인터 head는 항상 연결 리스트의 첫 노드를 가리키게 하며
  - 마지막 노드의 필드 next는 NULL
  - 2개의 노드 중 마지막 노드로 이동하여 하나의 노드를 새로 생성
    - 필드 x를 30으로 저장하고 이 노드가 마지막 노드가 되도록
  - 연결 리스트의 모든 노드를 순회하면서 필드 x 값을 출력
- **결과**
  - 1번째 노드는 20
  - 2번째 노드는 10
  - 3번째 노드는 30

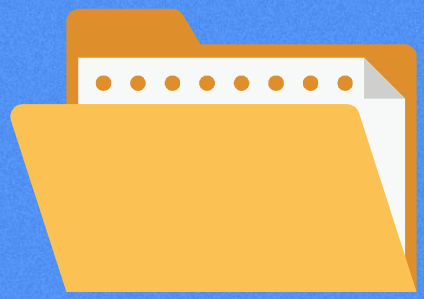
LAB 노드 3개로 구성된 연결 리스트(2)

Lab 16-2 nodelist.c

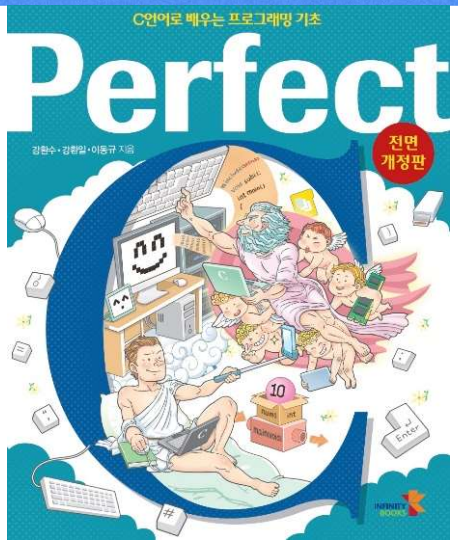
```c
#include <stdio.h>
#include <stdlib.h>

struct node {
   int x;
   _____________________________;
};

int main(void)
{
   //노드 두 개를 생성하여 자료와 링크를 대입
   struct node *one = _____________________________;
   one->x = 10;
   one->next = NULL;
   struct node *two = malloc(sizeof(struct node));
   two->x = 20;
   two->next = one;

   struct node *head = ___________________;
   struct node *cur = head;
   if (cur) {
      while (cur->next != NULL)
         cur = cur->next;
   }

   //노드를 생성하여 주소를 저장
   cur->next = malloc(sizeof(struct node));
   cur = cur->next;

   cur->next = NULL;
   cur->x = 30;

   cur = head;
   int cnt = 0;
   while (cur != NULL)
   {
      //리스트의 순서로 노드를 방문하여 방문 횟수와 문자열 자료를 출력
      printf("%3d번째 노드는 %d\n", ++cnt, cur->x);
      ___________________________;
   }

   return 0;
}
```

정답

```c
06  struct node *next;
12     struct node *one = malloc(sizeof(struct node));
19     struct node *head = two;
39        cur = cur->next;
```



03. 전처리



다양한 전처리 명령어

- **전처리기가 처리하는 지시자의 종류**
  - #include
    - 헤더파일을 삽입
  - #define
    - 기호상수를 정의
- **조건부로 필요한 문장을 컴파일에 참여시키는 지시자**
  - 조건부 컴파일 문장은 개발단계에서만 실행에 참여하는 문장 처리에 적합
  - 프로그램 개발에서 버전과 플랫폼에 따라 다르게 컴파일 조건부 컴파일 기능
    - 유용하게 사용

표 16-2 전처리 명령어 종류

명령어	설명
#define	기호상수 정의
#elif	조건부 컴파일 블록의 표시
#else	조건부 컴파일 블록의 마지막을 표시
#endif	조건 부 컴파일 블록의 종료 표시
#if	주어진 연산식이 참이면 컴파일

명령어	설명
#ifdef	주어진 이름이 정의되었다면 컴파일
#ifndef	주어진 이름이 정의되지 않았다면 컴파일
#include	지정된 헤더파일 내용을 현재에 복사
#undef	지정한 기호상수를 삭제

예약 매크로 예제

예제 sysmacro.c

- 예약 매크로(predefined macro)
  - 시스템에서 이미 정의되어 있는 매크로
  - ANSI 표준 매크로로 프로그램 디버깅에 활용

실습예제 16-9 sysmacro.c

이미 정의된 매크로의 이용

```
01  // file: sysmacro.c
02  #include <stdio.h>
03
04  int main(void)
05  {
06     printf("__DATE__ -> %s\n", __DATE__);
07     printf("__FILE__ -> %s\n", __FILE__);
08     printf("__LINE__ -> %d\n", __LINE__);
09     printf("__TIME__ -> %s\n", __TIME__);
10     printf("__TIMESTAMP__ -> %s\n", __TIMESTAMP__);
11
12     return 0;
13  }
```

설명

```
06  가장 최근에 소스 파일을 컴파일한 날짜로 [Mmm dd yyyy]으로 표시
07  현재 소스 파일 이름으로 절대경로로 표시
08  현재 소스 파일에서 이 문장이 있는 줄 번호
09  가장 최근에 소스 파일을 컴파일한 시각으로 [시:분:초]로 표시
10  현재 소스 파일을 마지막으로 수정한 시각으로 [요일 월 날짜 시:분:초 년]으로 표시
```

실행결과

```
__DATE__ -> Dec 11 2017
__FILE__ -> d:\creative c sources\ch16\prj08\sysmacro.c
__LINE__ -> 9
__TIME__ -> 17:35:08
__TIMESTAMP__ -> Sat Dec 11 17:35:07 2017
```

소스 파일의 최종 수정된 날짜와 시각 정보를 표시

표 16-3 예약 매크로

매크로	설명
__DATE__	가장 최근에 소스 파일을 컴파일한 날짜로 [Mmm dd yyyy]으로 표시
__FILE__	현재 소스 파일 이름으로 절대경로로 표시
__LINE__	현재 소스 파일에서 이 문장이 있는 줄 번호
__TIME__	가장 최근에 소스 파일을 컴파일한 시각으로 [시:분:초]로 표시
__TIMESTAMP__	현재 소스 파일을 마지막으로 수정한 시각으로 [요일 월 날짜 시:분:초 년]으로 표시

명령행에서 전처리 결과 보기

- **비주얼 스튜디오 개발도구**
  - 명령어 컴파일러 cl.exe를 제공
    - 전처리기가 처리한 결과 소스를 표준출력
      - 컴파일러 cl 명령어에서 /E 옵션을 사용
- **예제 소스 파일 sysmacro.c의 전처리 결과를 보는 명령어**
  - 결과 소스가 길어 그 내용을 마지막 부분만 보기 가능
  - 도스 연산자 >>를 사용
    - 표준출력을 지정한 파일이름 preresult.c에 저장
    - 옵션 /P를 사용하면 바로 확장자가 i인 소스 파일이름으로 전처리 결과를 저장 가능

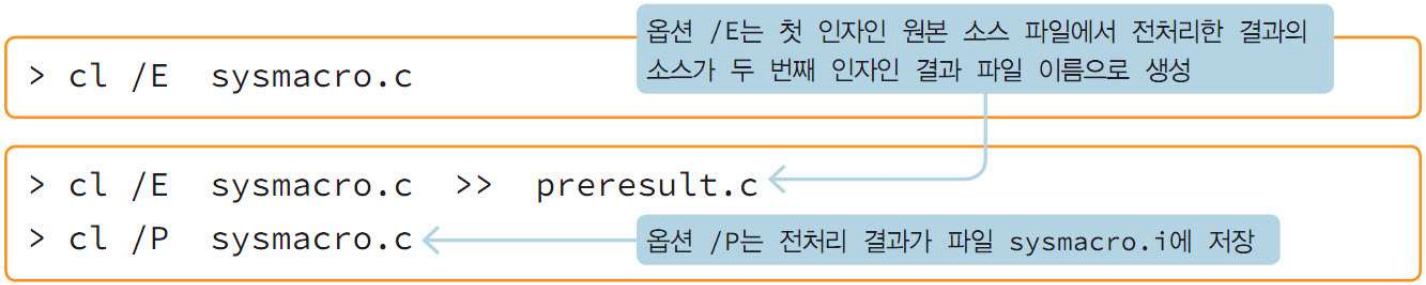


그림 16-33 전처리 결과를 파일에 저장하는 명령어

기본 조건부 컴파일 명령어 #if #endif

- **기본적인 조건부 컴파일 지시자**
  - 전처리 #if #endif
    - #if 다음의 조건식 expression이 0이 아니면
      - #endif 또는 다른 옵션의 전처리 지시자 사이의 모든 문장을 컴파일
    - 조건식 expression이 0이면 컴파일에서 제외
    - #if는 반드시 #endif로 종료
  - #if에서는 조건식에 괄호는 생략 가능
    - 괄호를 사용해도 무방
    - 조건 절의 문장이 여러 개라도 블록을 사용할 필요가 없음
  - #if와 #endif 사이
    - C 문장뿐 아니라 #define과 같은 다른 전처리기 지시자도 가능
    - 옵션으로 else if를 의미하는 #elif와 else인 #else 가능

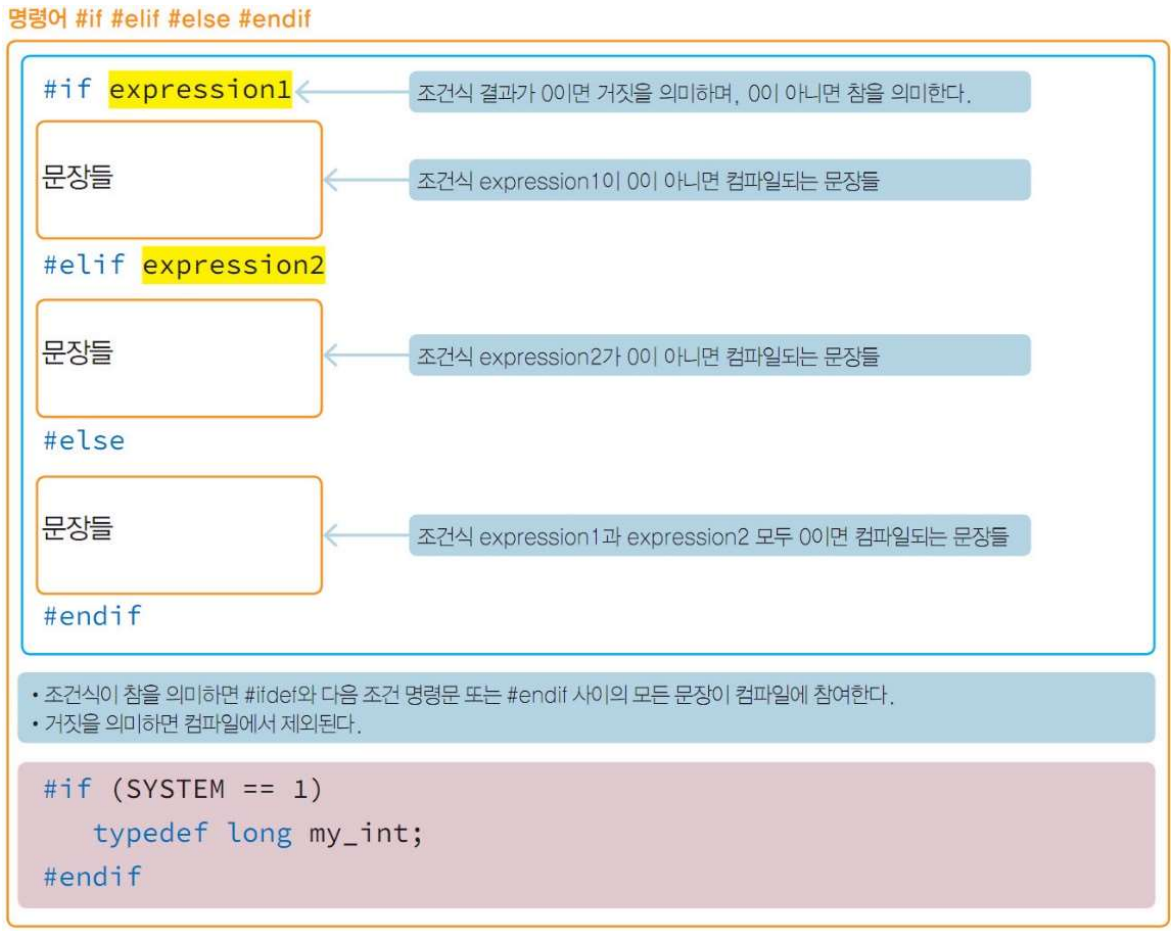


그림 16-35 전처리 지시자 #if #endif

#if 조건식

- **명령문 #if 조건식**
  - 기호상수와 정수 상수, 문자 상수만 사용 가능
  - 실수 상수와 문자열 상수, 변수 등은 사용 불가능
  - 그 결과도 반드시 정수
  - 조건식에는 관계연산자와 논리연산자 그리고 사칙연산을 사용 가능
    - 조건식에 변수는 사용 불가능
- **다음은 잘못된 #if 조건식**

```
#if SYSTEM < 2.0
#define TEST 100
#endif
```

```
#define PL "Java"
#if PL == "Java"
#define TEST 100
#endif
```

```
int a = 10;
#if a == 10
typedef long my_int;
#endif
```

그림 16-36 잘못 사용된 #if 조건식

전처리 조건

예제 preif.c

- 플랫폼에 따라 int 형을 다른 자료형으로 사용

실습예제 16-10 preif.c

전처리기 지시자 #if를 이용

```
01  // file: preif.c
02  #include <stdio.h>
03
04  //상수 정의
05  #define WINDOWS 1
06  #define MAC 2
07  #define UNIX 3
08  //#define SYSTEM WINDOWS
09  #define SYSTEM UNIX
10
11  //전처리 #if #elif #else #endif
12  #if (SYSTEM == WINDOWS)
13  typedef long my_int;
14  #elif SYSTEM == MAC
15  typedef int my_int;
16  #elif SYSTEM == UNIX
17  typedef long long my_int;
18  #else
19  typedef short my_int;
20  #endif
21
22  int main(void)
23  {
24     my_int n = 17;
25     printf("변수크기: %d, 저장값: %d\n", sizeof(n), n);
26
27     return 0;
28  }
```

설명

```
12  조건이 SYSTEM == WINDOWS 이면 13행인 my_int를 long으로 사용
14  조건이 SYSTEM == MAC 이면 15행인 my_int를 int로 사용
16  조건이 SYSTEM == UNIX 이면 17행인 my_int를 long long으로 사용
18  else이면 my_int를 short로 사용
24  변수 n을 my_int로 선언하고 초기값으로 17 저장
25  변수의 저장공간 크기와 변수 저장값 저장
```

실행결과

```
변수크기: 8, 저장값: 17
```

전처리 연산자 defined

- **전처리 연산자 defined (기호상수)**
  - 기호상수가 정의되었다면 0이 아닌 값을, 정의가 되지 않았다면 0을 반환
  - 피연산자의 괄호는 생략 가능
  - 연산자 defined는 지시자 #if의 조건식에서 사용 가능

```
#if (defined WINDOWS)
   typedef long my_int;
#endif
```

그림 16-37 전처리 연산자 defined

- **if defined #ifdef #endif**
  - 전처리 #ifdef #endif도 조건부 컴파일 지시자
  - #ifdef 다음에 나오는 기호상수가 이미 정의되었다면
    - #endif까지 모든 문장을 컴파일하고,
  - 그렇지 않으면
    - 컴파일에서 제거
  - #ifdef는 반드시 #endif로 종료
  - #ifdef와 #endif 사이에는 C 문장뿐 아니라 전처리 지시자도 사용 가능

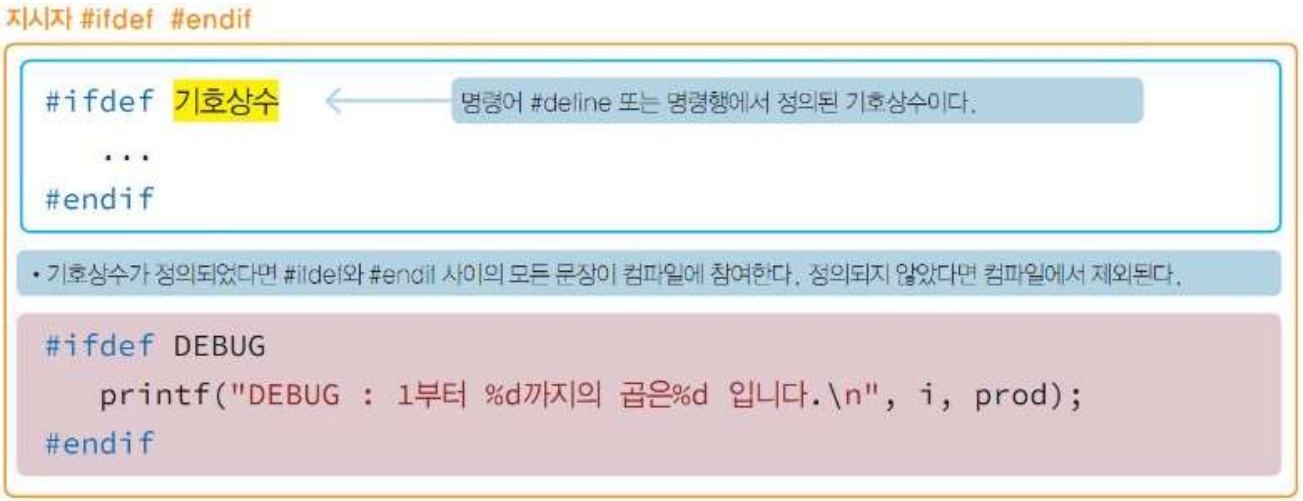


그림 16-38 전처리 지시자 #ifdef #endif

기호상수 정의 방법

- **기호상수 DEBUG가 정의되어 있으면**
  - 중간 결과를 출력하는 문장이 컴파일
- **그렇지 않으면**
  - 이 문장은 컴파일에서 제외
- **기호상수 DEBUG**
  - 지시자 #define으로 정의 가능
  - 비주얼 스튜디오에서도 직접 기호상수를 정의 가능
    - 메뉴 [프로젝트/ 프로젝트 속성 ...]를 선택
    - 표시된 [프로젝트 속성 페이지] 대화상자
      - [전처리기 정의]에서 DEBUG를 삽입

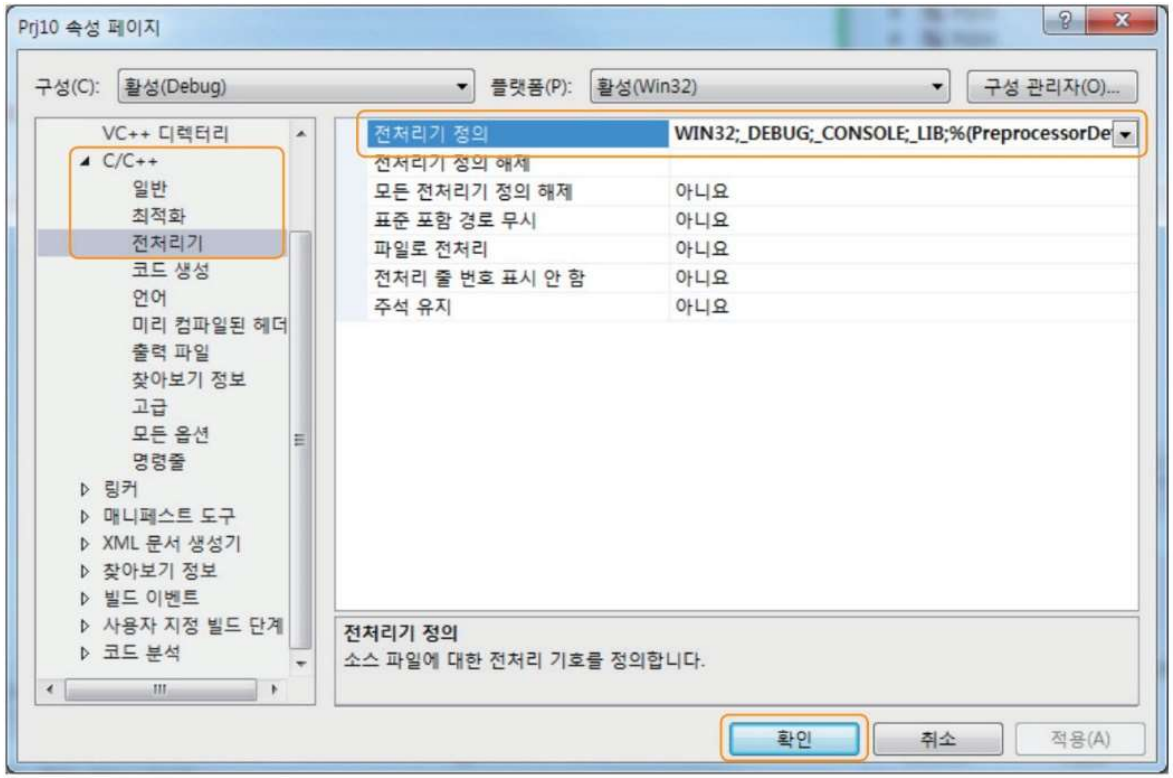


그림 16-39 비주얼 스튜디오에서 기호상수를 정의하는 대화상자

컴파일에 참여

예제 ifdef.c

- 기호상수 DEBUG가 정의되어 있다면
  - printf("DEBUG : 1부터 %d까지의 곱은 %d입니다.₩n", i, prod); 출력문을 컴파일
- 만일 DEBUG가 정의되어 있지 않다면
  - #ifdef 내부의 printf()문은 컴파일에 참여할 수 없고, 당연히 실행해도 그 부분은 실행되지 않음

실습예제 16-11 ifdef.c

전처리 지시자 #ifdef 이용한 디버깅 방법

```
01  // file: preifdef.c
02  #include <stdio.h>
03  #define DEBUG
04  #define LIMIT 20
05
06  int main(void)
07  {
08     long prod = 1;
09     for (int i = 1; i <= LIMIT; i++)
10     {
11        prod *= i;
12
13  #ifdef DEBUG
14        if (i % 5 == 0)
15           printf("DEBUG : 1부터% d까지의 곱은 %d입니다.\n", i, prod);
16  #endif
17
18     }
19     printf("1부터 %d까지의 곱은 %d입니다.\n", LIMIT, prod);
20
21     return 0;
22  }
```

이 기호상수 정의 문을 제거하면 중간 과정의 결과는 보이지 않는다.

기호상수 DEBUG가 정의되지 않는다면 다음과 같은 한 행의 결과만 표시된다.

1부터 20까지의 곱은 -2102132736입니다.

설명

03 상수 DEBUG를 정의

13 상수 DEBUG가 정의되었다면 14 ~ 15행을 컴파일하므로, 중간과정인 5, 10, 15, 20까지의 곱도 출력될 수 있으며, 3행을 빼버리면 중간과정이 안 나오고 20까지의 곱의 결과만 계산하여 19행의 결과만 출력

19 1에서 20까지의 곱의 결과를 출력

실행결과

```
DEBUG : 1부터 5까지의 곱은 120입니다.
DEBUG : 1부터 10까지의 곱은 3628800입니다.
DEBUG : 1부터 15까지의 곱은 2004310016입니다.
DEBUG : 1부터 20까지의 곱은 -2102132736입니다.
1부터 20까지의 곱은 -2102132736입니다.
```

명령어 #ifndef

- **전처리 지시자 #ifndef**
  - #ifdef와 반대로 기호상수가 정의되지 않았다면
    - #ifndef와 #endif 사의의 모든 문장이 컴파일에 참여
  - 정의되었다면: 컴파일에서 제외
- **기호상수 NAME_SIZE가 정의되어 있지 않다면**
  - NAME_SIZE를 30으로 정의하는 문장
  - 실제로 헤더파일을 여러 개 사용하다 보면 기호상수 정의가 중복될 수 있는데
    - 컴파일 시 경고가 발생
  - #ifndef 를 사용한 기호상수 정의
    - 단순히 #define NAME_SIZE 30이 아니라
    - 헤더파일에서 기호상수를 중복되게 정의하는 것을 막아주는 역할

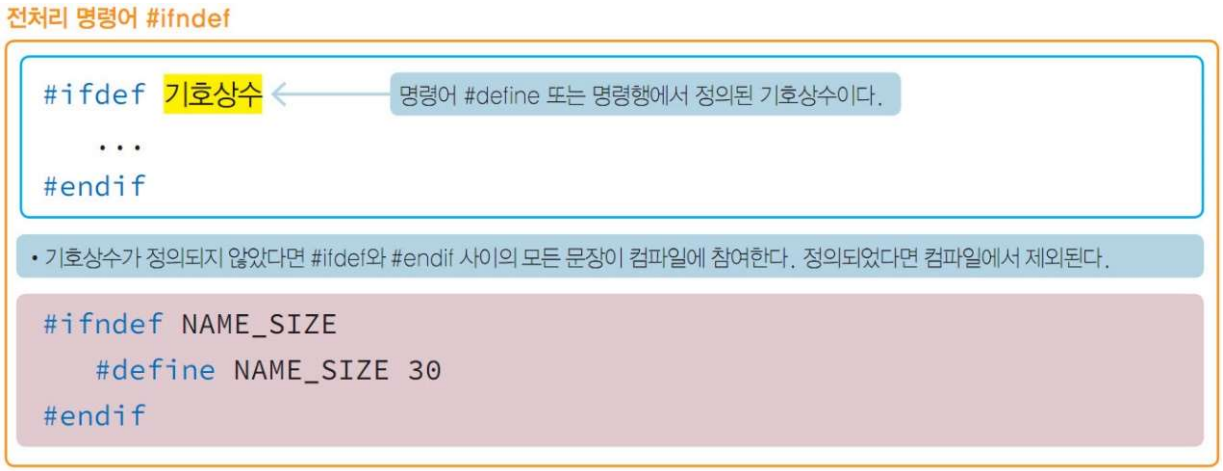


그림 16-40 전처리 지시자 #ifndef #endif

#undef

- **기호상수 삭제 #undef**
  - #undef는 이미 정의된 기호 상수를 해지하는 지시자
  - 다음 구문
    - 20으로 정의된 기호상수 SIZE
      - 전처리기 지시자 #undef SIZE로 그 효력을 상실하게 함

```
#define SIZE 20
#ifdef SIZE
#undef SIZE
#endif
```

그림 16-41 지시자 #undef

  - 일반적으로 기호상수를 삭제하기 전
    - #ifdef로 이전에 정의됨을 확인한 후 삭제하는 것을 추천

전처리 연산자 종류

- **전처리 연산자**
  - #, #@, ##, defined 모두 4개
- **연산자 #, #@, ##**
  - 매크로 정의 지시자 #define에서 사용
- **연산자 defined**
  - 이미 보았듯이 조건부 컴파일 지시자 #if와 #elif 등에서 사용

표 16-4 전처리 연산자

연산자	이름	사용 예	기능
#	문자열 만들기 연산자 (stringzing operator)	#인자	인자 앞 뒤에 큰 따옴표를 붙여 인자를 문자열로 만드는 연산자
#@	문자 만들기 연산자 (charizing operator)	#@인자	인자 앞 뒤에 작은 따옴표를 붙여 인자를 문자로 만드는 연산자
##	토큰 붙이기 연산자 (token-pasting operator)	인자##인자	인자를 다른 토큰들과 연결해주는 연산자
defined	정의 검사 연산자 (defined operator)	defined WINDOWS	상수로 정의되어 있는지 검사하는 연산자

문자열 만들기 연산자 #(1)

- **매크로 정의 시에 뒤에 나오는 인자를 문자열로 만드는 연산자**
  - 매크로에서 인자의 앞 뒤에 인용 부호 "인자"를 넣어 문자열로 만듬
  - 매크로 PRT(a) 정의
    - 인자 a의 사용을 #a로 하는 경우
      - 이 매크로 호출 시에 인자의 앞 뒤에 인용 부호 "a"를 넣어 문자열로 만듬
- **매크로 PRT(i)의 호출**
  - 실인자 #i는 "i"로 대체되며 실인자 i는 그대로 i로 대체
  - 문자열 "i"와 " = %2d 일때,"를 나열
    - 그대로 문자열의 연결을 의미
      - 두 문자열이 연결된 문자열 "i = %2d 일때,"

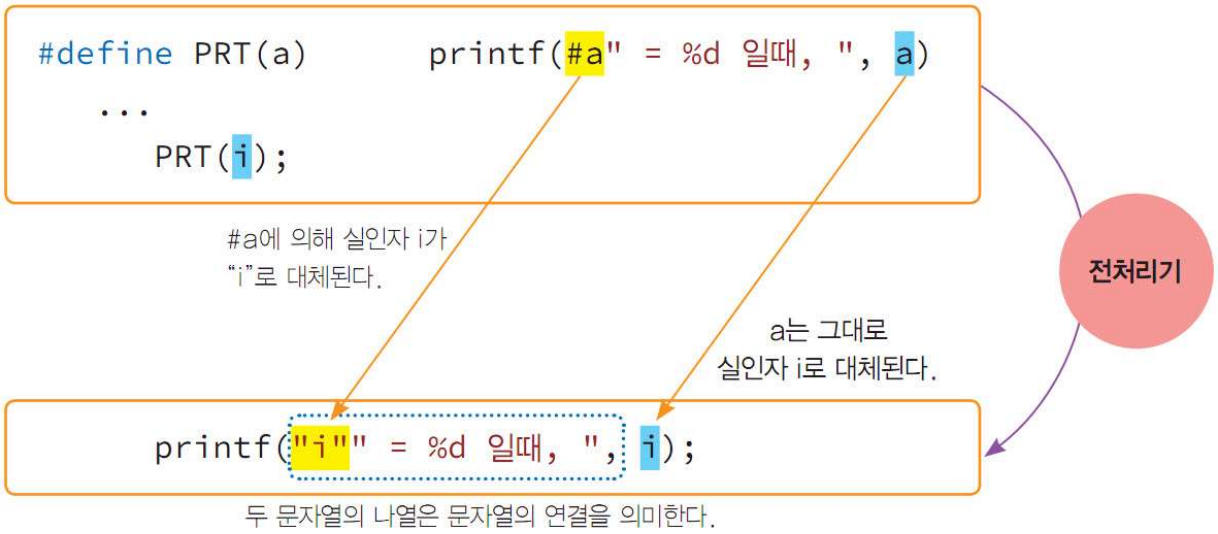


그림 16-42 문자열 만들기 연산자 #의 예1

문자열 만들기 연산자 #(2)

- **매크로 정의 APRT(a)**
  - APRT(fact[i])의 호출은 어떤 문장이 실행될까?
  - printf("facto[i]"" = %3d₩t", facto[i]); 문장으로 실행

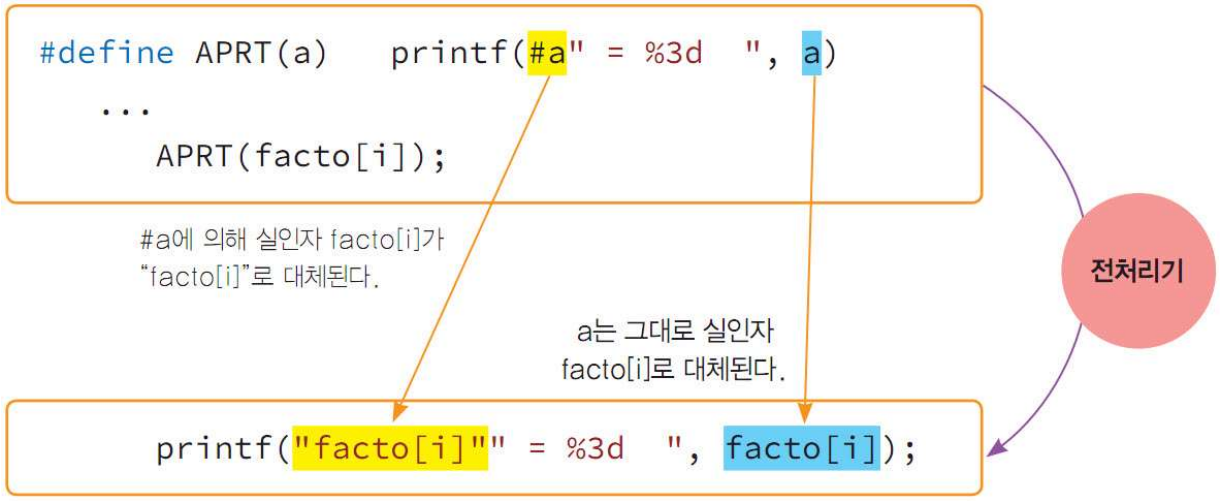


그림 16-43 문자열 만들기 연산자 #의 예2

문자 만들기 연산자 #@

- **문자 만들기 연산자 #@ 인자 x의 매크로 정의**
  - #@x와 같이 형식인자 앞에 위치
    - 연산자 #@는 뒤에 나오는 인자를 앞 뒤에 작은 따옴표를 붙여 문자로 만들어 줌
  - 다음 매크로 CHPRT(a)는 인자 a를 문자로 만들어 함수 printf()로 출력하는 매크로
    - 매크로 호출 CHPRT($)로 문자 $를 출력

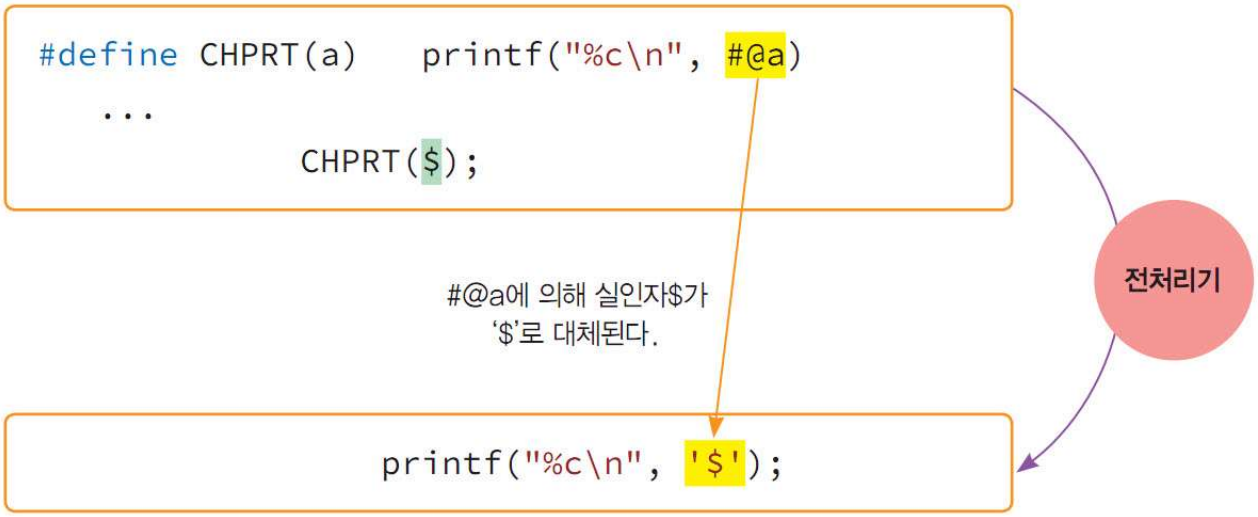


그림 16-44 문자 만들기 연산자 #@의 예

토큰 붙이기 연산자

- **좌우의 토큰을 연결(concatenation)하는 기능을 수행**
  - 매크로 AIPRT(a, i)에서 연산식 a##[i]는 a와 [i]를 연결
  - 매크로 호출 AIPRT(facto, i)
    - 실인자 연산식 facto##[i]에 의해 facto와 i를 연결한 facto[i]로 대체

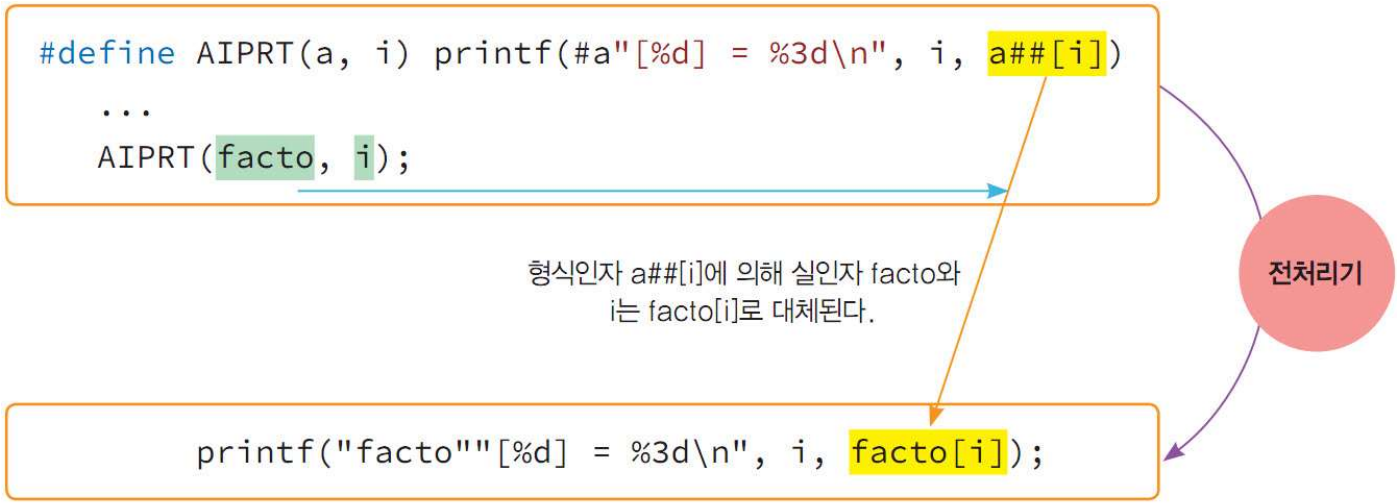


그림 16-45 토큰 붙이기 연산자 ##

전처리 연산자 예제

예제 operator.c

- 전처리 연산자 #
- 전처리 연산자 #@
- 전처리 연산자 ##

실습예제 16-12 operator.c

전처리 연산자 #, #@, ##의 사용

```c
// file: operator.c
#include <stdio.h>

#define CHPRT(a)        printf("%c\n", #@a)
#define PRT(a)          printf(#a" = %d 일때, ", a)
#define APRT(a)         printf(#a" = %3d  ", a)
#define AIPRT(a, i)     printf(#a"[%d] = %3d\n", i, a##[i])

int main(void)
{
   int prod = 1;
   int facto[6];
   CHPRT($); //4행 매크로 호출

   for (int i = 1; i <= 5; i++)
   {
      prod *= i;
      facto[i] = prod;
      PRT(i); //5행 매크로 호출
      APRT(facto[i]); //6행 매크로 호출
      AIPRT(facto, i); //7행 매크로 호출
   };

   return 0;
}
```

설명

```
#@인자는 인자 앞 뒤에 작은 따옴표를 붙여 인자를 문자로 만드는 연산자로 #@a는 'a'로 만듦
#인자는 인자 앞 뒤에 큰 따옴표를 붙여 인자를 문자열로 만드는 연산자로 #a는 "a"로 만듦
#a" = %3d  "는 결국 "a" " = %3d  "가 되어 문자열 "a" " = %3d  "이 됨
##은 두 다른 토큰들과 연결해주는 연산자로 a##[i]는 a[i]가 됨
PRT(i)는 i에 따라 'i = 4 일때'와 같이 출력
APRT(facto[i])는 'i에 따라 facto[i] = 24'와 같이 출력
AIPRT(facto, i)는 'i에 따라 facto[4] = 24'와 같이 출력
```

실행결과

```
$
i = 1 일때, facto[i] =   1  facto[1] =   1
i = 2 일때, facto[i] =   2  facto[2] =   2
i = 3 일때, facto[i] =   6  facto[3] =   6
i = 4 일때, facto[i] =  24  facto[4] =  24
i = 5 일때, facto[i] = 120  facto[5] = 120
```

LAB 다양한 자료형의 변수 내용을 서로 교환하는 매크로(1)

- **스왑(swap)**
  - 두 변수의 내용을 서로 교환하는 수행
  - 매크로 SWAP_INT(a, b, temp)
    - int 형 변수 두 a, b를 이미 선언되어 있는 변수 temp를 사용하여 두 변수를 교환하는 매크로로 구현
  - 매크로 SWAP_DOUBLE(a, b)
    - double형 변수 두 a, b를 매크로 내부에서 직접 변수 _temp를 선언해 두 변수를 교환하는 매크로로 구현
  - 매크로 SWAP_TYPE(type, a, b)
    - 자료형 type 형인 변수 두 a, b를 매크로 내부에서 직접 자료형 type으로 변수 _swap_temp를 선언해 두 변수를 교환하는 매크로로 구현
- **교현**
  - 정수를 교환하는 매크로와 실수를 교환하는 매크로 각각 2번 호출
- **결과**
  - 30 20
  - 20 30
  - 30 20
  - 123.45 43.87
  - 43.87 123.45
  - 123.45 43.87

LAB 다양한 자료형의 변수 내용을 서로 교환하는 매크로(2)

Lab 16-3 swaptype.c

```c
// swaptype.c:
#include <stdio.h>

//자료형 int a, b의 교환 매크로, temp를 사용
#define SWAP_INT(a, b, temp)   \
      temp = a; _____; ________;

#define SWAP_DOUBLE(a, b)   \
      ____________ _temp = a; a = b; b = _temp;

//자료형을 지정하여 a, b의 교환 매크로, _swap_temp를 내부에서 지정해서 사용
#define SWAP_TYPE(type, a, b) \
       { \
         _________________________ \
         _swap_temp = (b); \
         (b) = (a); \
         _________________________ \
       }

int main(void)
{
   int a = 30, b = 20, c;
   printf("%d %d\n", a, b);
   SWAP_INT(a, b, c);
   printf("%d %d\n", a, b);
   SWAP_TYPE(int, a, b);
   printf("%d %d\n", a, b);

   double x = 123.45, y = 43.87;
   printf("%.2f %.2f\n", x, y);
   SWAP_DOUBLE(x, y);
   printf("%.2f %.2f\n", x, y);
   SWAP_TYPE(_____________________________________);
   printf("%.2f %.2f\n", x, y);

   return 0;
}
```

정답

```
06    temp = a; a = b; b = temp;
09    double _temp = a; a = b; b = _temp;
14    type _swap_temp; \
17    (a) = _swap_temp; \
33  SWAP_TYPE(double, x, y);
```

Thank you