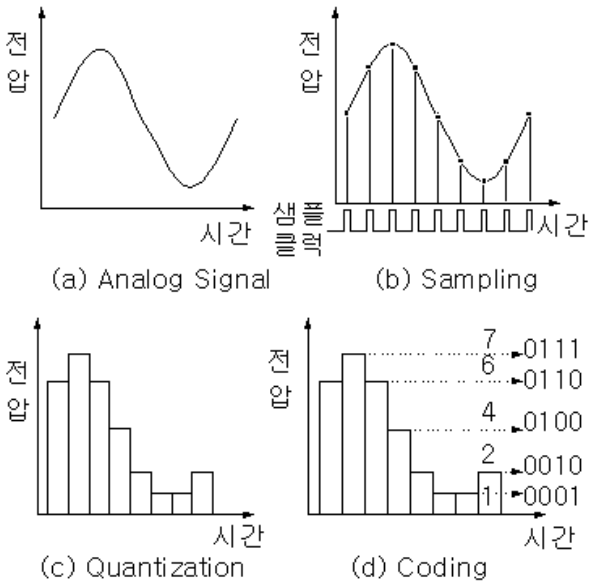
임베디드 C 프로그래밍

기본개념 : Digital VS Analog

- Analog
  - 연속되는 값으로 표현하는 정보
  - 자연에서 생성된 파장 정보를 재현
- Digital
  - 정보를 서로 다른 숫자(Digit)로 표현.
  - 0과 1이라는 인위적인 신호로 바꾸어 표현.
- Analog to Digital
  - Analog를 시간기준으로 sampling
  - Sampling된 data를 Quantization
  - Data 표현.



기본개념 : bit, byte and word

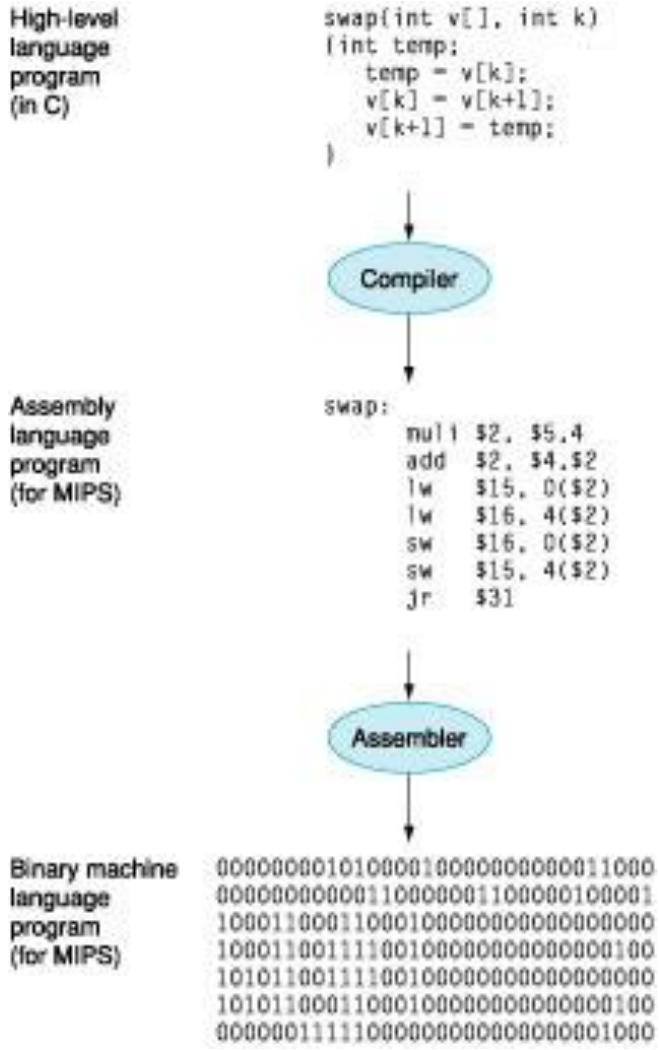
- bit
  - Binary digit의 약자
  - 이진수의 하나의 자리수를 의미.
  - 0 – 1, 참 – 거짓등의 배타적인 상태를 나타냄.
- byte
  - 비트가 여러 개 모인 것.
  - 원래는 크기가 명확히 정해져 있지 않지만,
    현재는 대개 8bit를 1-byte이다.
- Word
  - Embedded IC를 설계시 정해지는 메모리의 기본 단위.
  - 8-bit process : 1 word = 1-byte
  - 32-bit process : 1 word = 4-byte

기본개념 : binary, octal, decimal and hexadecimal

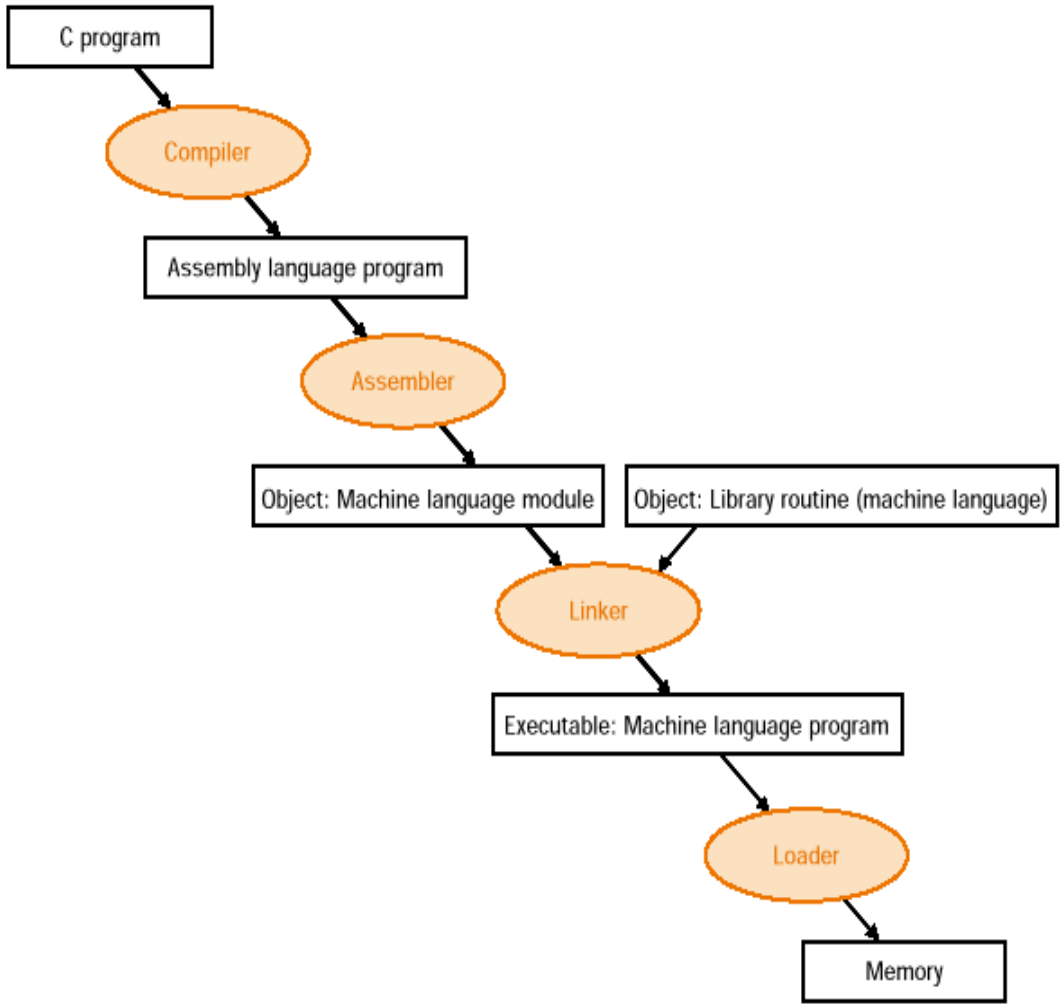
- binary
  - 이진법
  - 0과 1로 수를 표현
- octal
  - 8진법
  - 0 ~7까지로 수를 표현
- decimal
  - 10진법
  - 0 ~ 9까지로 수를 표현
- hexadecimal
  - 16진법
  - 0 ~ 9 + A, B, C, D, E, F로 수를 표현

From HLL to the Language of Hardware

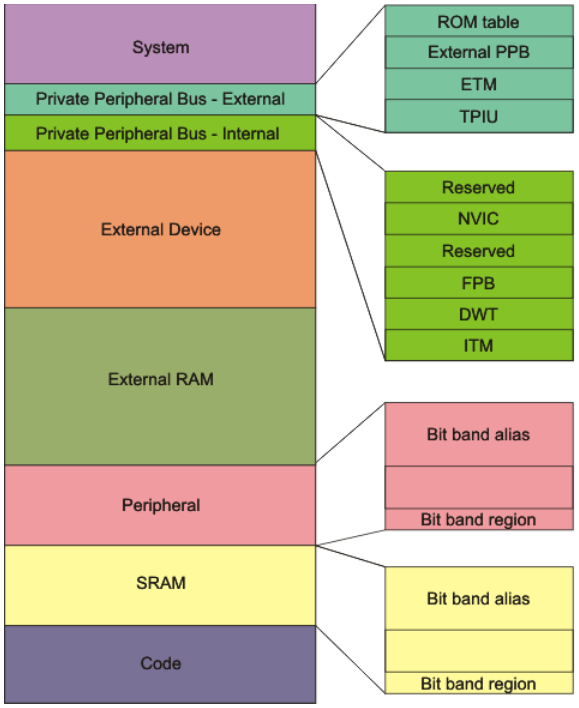
- High-level programming languages
  - Similar with natural language
  - Improved programmer productivity
  - Independent of the computer



프로그램 빌드 과정


C program
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

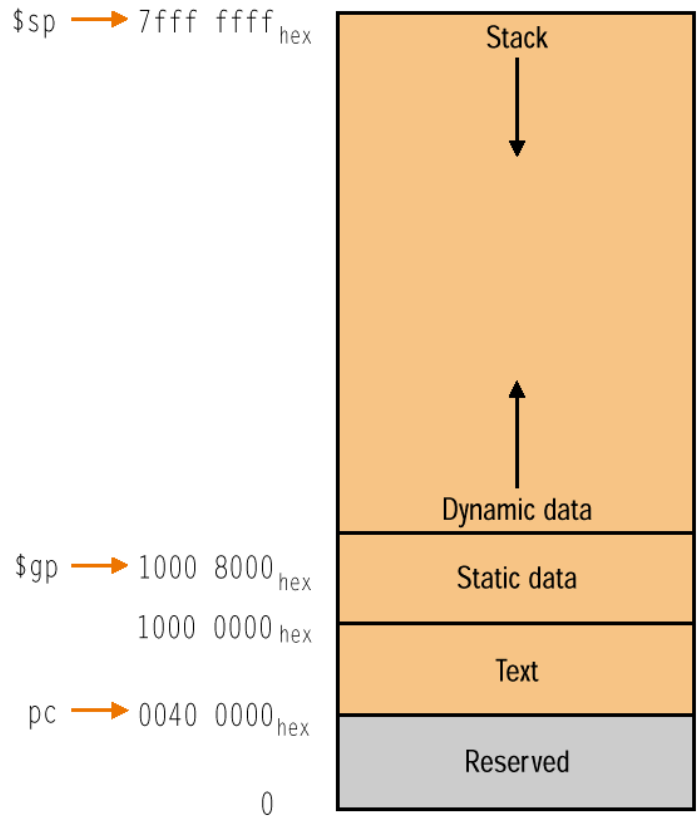
메모리 맵



[ARM Memory Map]

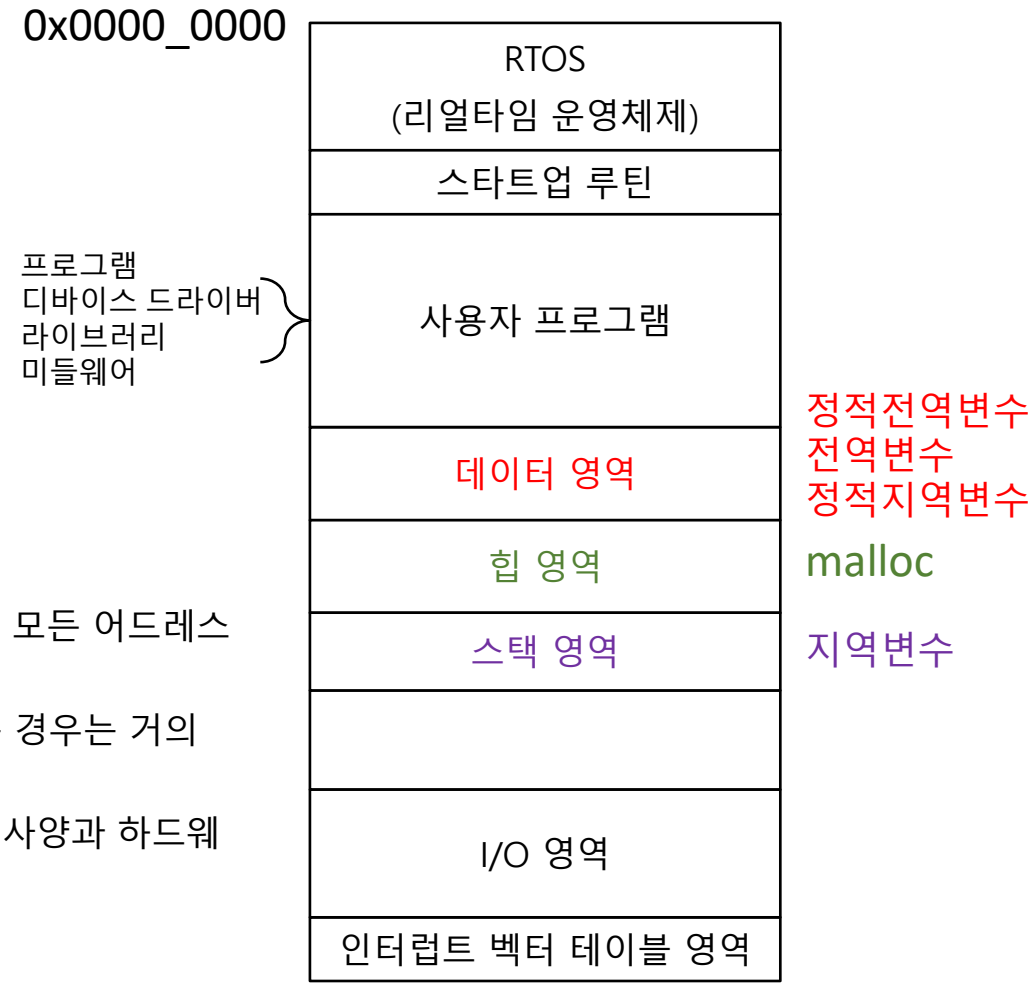
- 임베디드 프로그래밍에서는 Memory Map 파악이 중요함
  - 메모리 어느 영역에 무엇이 있는지
  - 각 메모리 영역의 크기는 얼마인지
- 스택의 움직임 및 메모리 상에 변수나 함수가 배치되는 방식을 이해함으로써 불필요한 문제를 피할 수 있음
- 메모리 공간상의 I/O(Input/Output) 포트로의 액세스 방법을 숙지하는 것은 임베디드 프로그래밍에서 유용하게 사용할 수 있음
- 메모리에 관련된 여러 사항들을 설명 하겠음

메모리 레이아웃


$sp
7fff ffff hex
Stack
Dynamic data
$gp
1000 8000 hex
Static data
1000 0000 hex
Text
pc
0040 0000 hex
Reserved
0

1. Memory Map

- Memory Map의 예
  - 0번지에서 인터럽트 벡터 테이블 영역까지 모든 어드레스에 무엇인가 배치되어 있는 예시
  - 실제 MCU의 메모리 공간 전부를 이용하는 경우는 거의 없음
  - 어느 영역에 무엇이 배치되는지는 MCU의 사양과 하드웨어 담당자의 설계에 의존

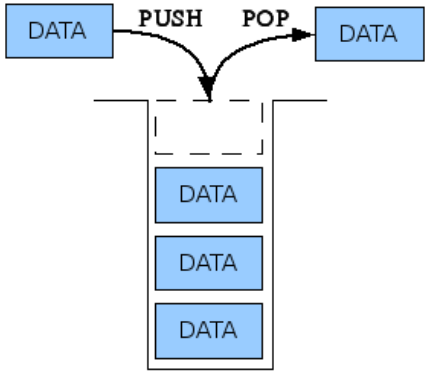


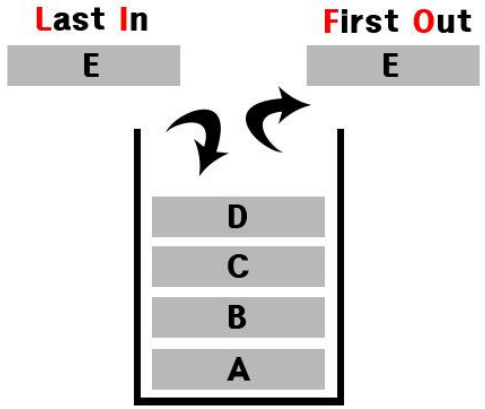
[Memory Map의 예]

Variables in C

	지역 변수	전역 변수	정적 변수	레지스터 변수
지정자	auto (생략가능)	extern (생략가능)	static	register
메모리	스택	데이터 세그먼트	데이터 세그먼트	레지스터
선언	함수 내부	어디든 가능	지역: 함수 내부 전역: 함수 외부	함수 내부
초기화	초기화 문장을 통해서만 실행, 초기화 안하면 쓰레기 값	초기화 문장 없이도 0으로 자동 초기화	초기화 문장 없이도 0으로 자동 초기화	초기화 문장을 통해서만 실행, 초기화 안하면 쓰레기 값
유효 범위	선언한 함수나 블록 내부	함수 외부 파일 외부	함수 내부 함수 외부 파일 내부	선언한 함수나 블록 내부
유효 시간 (life time)	선언된 함수가 실행될 때부터 종료될 때까지	프로그램 실행될 때부터 종료될 때까지	프로그램 실행될 때부터 종료될 때까지	선언된 함수가 실행될 때부터 종료될 때까지

2. 스택

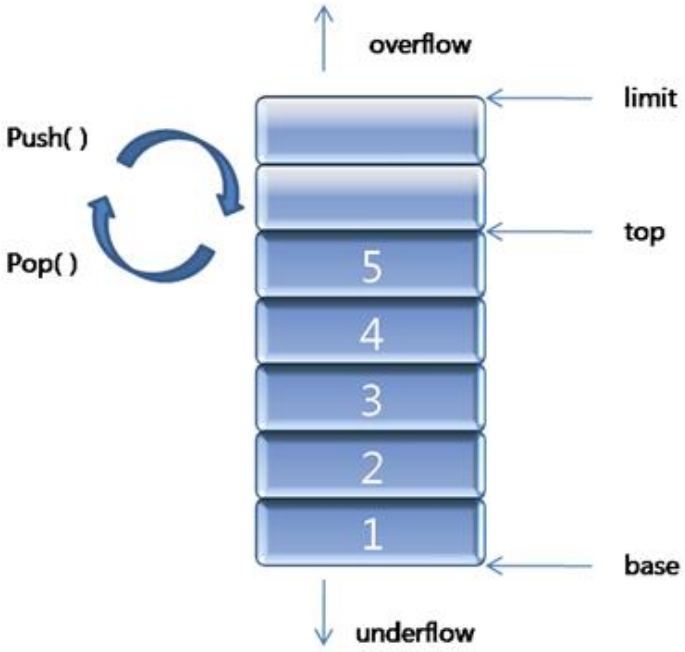




- C 프로그램의 동작을 이해하는데 있어 '스택 영역' 이해는 매우 중요
- 스택(Stack)은 영어로 '쌓다', '쌓아 올리다'라는 의미
  - 의미 그대로 스택은 데이터를 쌓아놓고 관리
- 스택에서의 데이터 관리
  - Push : 스택에 데이터를 저장하는 조작
  - Pop : 스택에서 데이터를 꺼내는 조작
  - FILO(First In, Last Out)

    : 처음 들어간(First In)것이 가장 나중에 나오는(Last Out) 특징을 가진 데이터 관리 기법

3. Stack Pointer



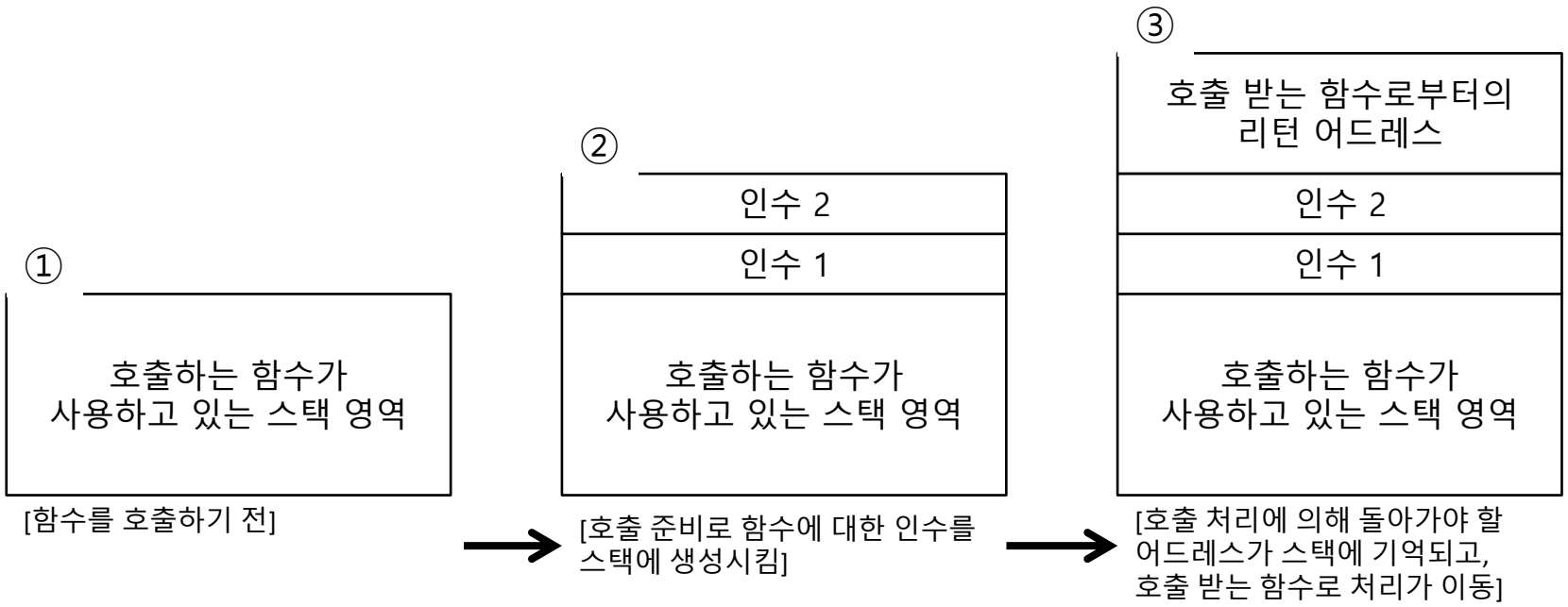
[Stack Pointer]

- 스택은 C 프로그램에서 자동 변수를 사용하거나 함수 호출과 연동해서 할당되는 임시 데이터 영역으로 사용
- 스택 포인터
  - CPU는 스택의 선두 영역의 어드레스를 스택 포인터라고 부르는 레지스터로 관리

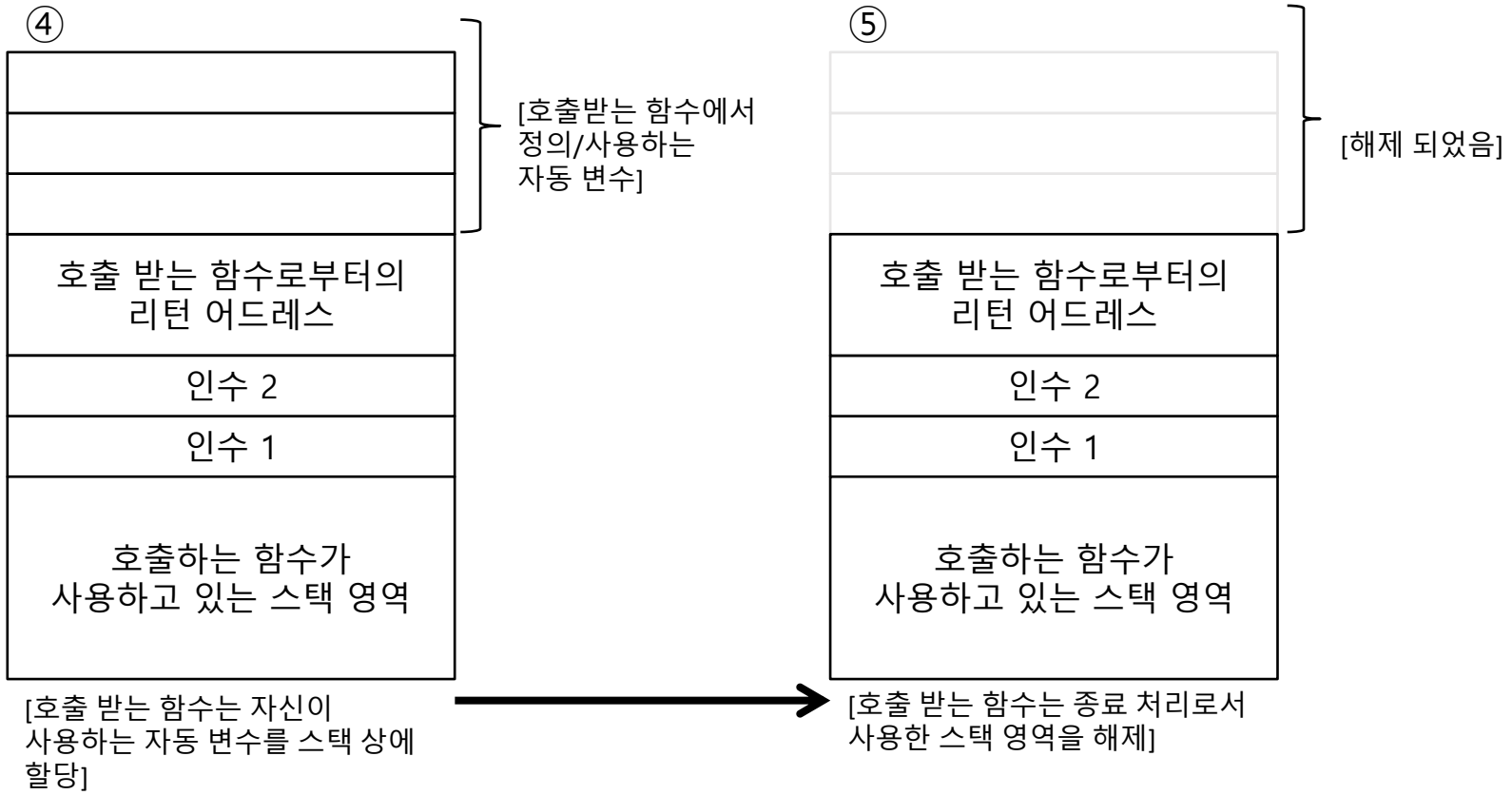
4. 함수 호출

- 함수를 호출할 때는 내부적 다음의 3가지 사항을 관리
  - 호출 받는 쪽에 인수를 전달 / 호출 받는 쪽으로부터 반환 값을 반환 값 전달 받음
  - 호출 받는 쪽으로 처리를 전달 / 함수를 종료하면 호출한 쪽으로 처리가 돌아가고 중단 상태였던 처리 재개
  - 호출 받는 쪽에 사용하는 변수 영역을 할당 / 함수를 종료할 때 할당한 변수 영역 해제
- 컴파일러 구현에 따라 차이 존재
- 해당 모든 처리는 스택 기능을 최대한 활용
- 스택 사용량의 계산이 틀리게 될 경우 스택 오버플로우가 발생
- 메모리 용량의 제약도 해결하면서 필요 충분한 크기의 스택 영역을 정의하는 것이 임베디드 프로그래밍에서 중요

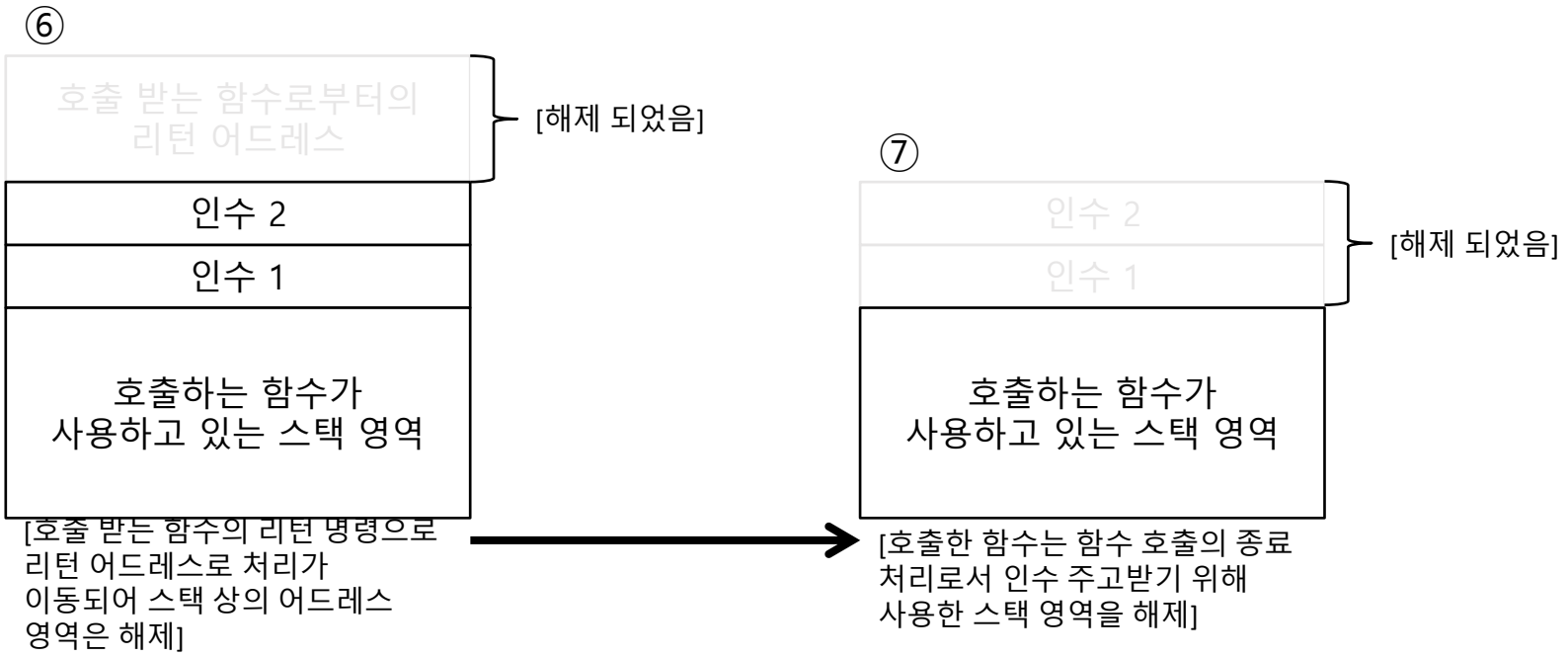
4. 함수 호출


①
호출하는 함수가
사용하고 있는 스택 영역
[함수를 호출하기 전]
②
인수 2
인수 1
호출하는 함수가
사용하고 있는 스택 영역
[호출 준비로 함수에 대한 인수를
스택에 생성시킴]
③
호출 받는 함수로부터의
리턴 어드레스
인수 2
인수 1
호출하는 함수가
사용하고 있는 스택 영역
[호출 처리에 의해 돌아가야 할
어드레스가 스택에 기억되고,
호출 받는 함수로 처리가 이동]

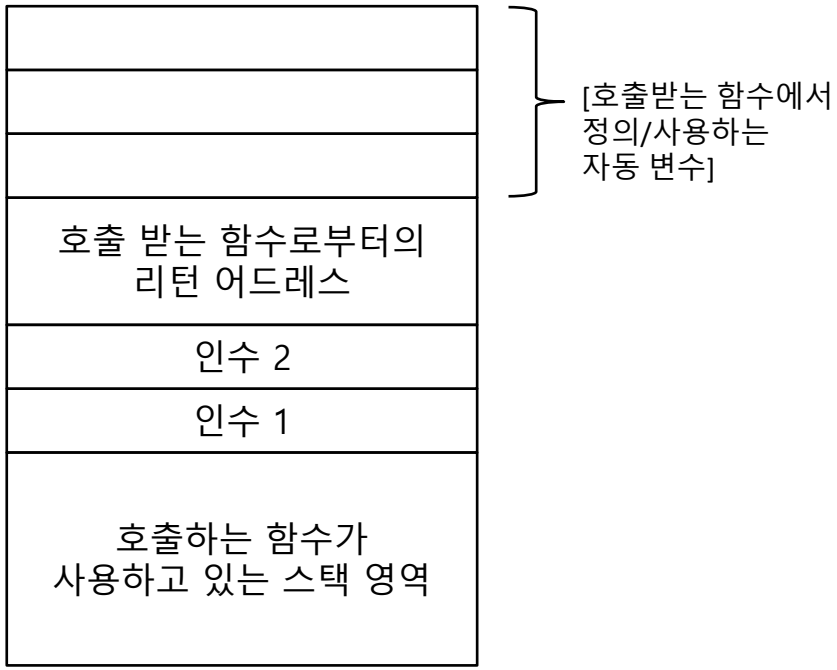
4. 함수 호출


④
[호출받는 함수에서
정의/사용하는
자동 변수]
호출 받는 함수로부터의
리턴 어드레스
인수 2
인수 1
호출하는 함수가
사용하고 있는 스택 영역
[호출 받는 함수는 자신이
사용하는 자동 변수를 스택 상에
할당]
⑤
[해제 되었음]
호출 받는 함수로부터의
리턴 어드레스
인수 2
인수 1
호출하는 함수가
사용하고 있는 스택 영역
[호출 받는 함수는 종료 처리로서
사용한 스택 영역을 해제]

4. 함수 호출


⑥
호출 받는 함수로부터의
리턴 어드레스
[해제 되었음]
인수 2
인수 1
호출하는 함수가
사용하고 있는 스택 영역
[호출 받는 함수의 리턴 명령으로
리턴 어드레스로 처리가
이동되어 스택 상의 어드레스
영역은 해제]
⑦
인수 2
인수 1
[해제 되었음]
호출하는 함수가
사용하고 있는 스택 영역
[호출한 함수는 함수 호출의 종료
처리로서 인수 주고받기 위해
사용한 스택 영역을 해제]

5. 변수 배치



- 자동 변수
  - C 프로그램에서 자동변수는 함수의 호출에서 스택 영역에 배치되어 보통은 함수의 호출과 함께 동적으로 할당/해제
  - 변수는 각각의 자료형에 의존하는 형태로 '자료형과 메모리와의 관계'에 따라 스택(메모리) 상에 위치
  - CPU는 스택상의 어드레스를 사용해서 자동 변수에 액세스
- static 변수
  - 자동 변수와 다르게 변수에 static 선언을 붙이면 그 변수를 스택 영역이 아닌 정적인(static) 데이터 영역에 존속 시킴
  - 변수별로 고정 어드레스를 할당해서 실현
  - static 선언의 변수에 액세스한다는 것은 메모리에서 고정 어드레스를 가지고 있는 변수에 액세스하게 된다는 의미

6. 자료형과 메모리와의 관계

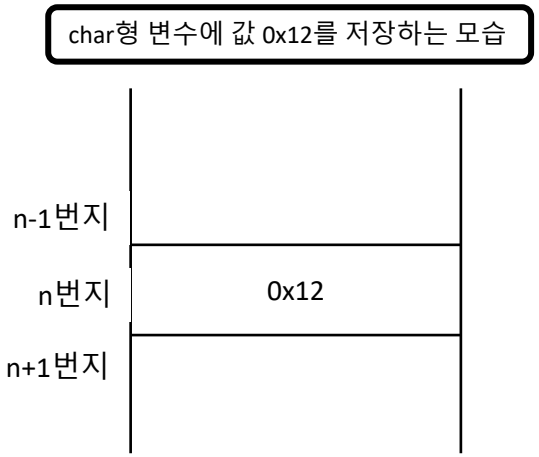
- 자료형과 메모리와의 관계
  - C 언어 내에는 다양한 자료형 존재
  - typedef 연산자를 사용해서 새로운 자료형을 자신이 정의 가능
  - C 언어로 임베디드 제품의 프로그램을 작성하거나 테스트할 경우 변수나 프로그램이 메모리 상에 어떤 식으로 배치되는가를 인식하면 메모리 풋프린트가 작은 프로그램이나 결함이 적은 프로그램을 얻는 것이 쉬워 짐
  - 몇 가지 자료형을 예로하여 데이터가 메모리 상에 어떤 식으로 배치되는지 설명 하겠음

※ 풋프린트(Footprint)는 발자국이라는 의미로 점유 면적이라는 의미로도 사용된다. 메모리 풋프린트는 메모리상의 점유 크기를 의미한다.

6. 자료형과 메모리와의 관계

- char 형
  - char형, signed char형, unsigned char형의 변수는 8비트의 데이터 길이 (즉, 1바이트)를 가지는 변수
  - 1바이트 변수는 메모리 상에서 다음 그림과 같이 배치

[1바이트 변수의 메모리 이미지]



6. 자료형과 메모리와의 관계

- short int형, int형, long int형
  - int형의 바이트 수는 처리 상태에 따라서 변하고 마찬가지로 short int형, long int형의 바이트 수도 처리 상태에 의존
  - 16비트 MCU나 32비트 MCU에서는 short int형은 2바이트, long형은 4바이트
  - 1바이트보다 긴 바이트 길이의 변수가 되면 메모리 상의 배치에 엔디언(endian)문제 발생

[2바이트 변수에서의 엔디언]

short int형 변수에 값 0x1234(10진수로 4660)을 저장하는 모습

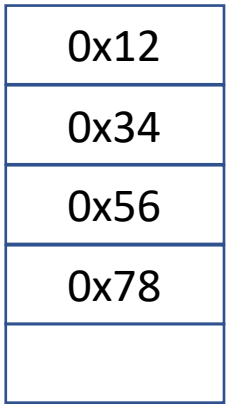
리틀 엔디언
메모리 덤프해 보면
n번지 34 12

빅 엔디언
메모리 덤프해 보면
n번지 12 34

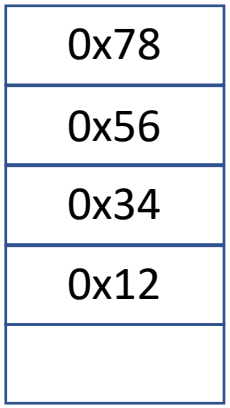
6. 메모리 바이트 순서(엔디안)

unsigned int x = 0x12345678;

- Big endian



- Little endian



6. 메모리 정렬

```
struct test {
    int a;
    short b;
    char c;
    int d;
    char e;
    char f;
    char g;
    short h;
};
```

Aligned access

3	2	1	0	
a[31:24]	a[23:16]	a[15:8]	a[7:0]	0x20000000
b[15:8]	b[7:0]	c[7:0]		0x20000004
d[31:24]	d[23:16]	d[15:8]	d[7:0]	0x20000008
e[7:0]	f[7:0]	g[7:0]		0x2000000C
h[15:8]	h[7:0]			0x20000010

Unaligned access

3	2	1	0	
a[31:24]	a[23:16]	a[15:8]	a[7:0]	0x20000000
b[15:8]	b[7:0]	c[7:0]	d[31:24]	0x20000004
d[23:16]	d[15:8]	d[7:0]	e[7:0]	0x20000008
f[7:0]	g[7:0]	h[15:8]	h[7:0]	0x2000000C
				0x20000010

6. 자료형과 메모리와의 관계

- 리틀 엔디언 / 빅 엔디언
  - 리틀 엔디언에서는 데이터 중에서 비중이 작은 바이트가 어드레스의 작은 쪽에 저장
  - 빅 엔디언에서는 비중이 큰 바이트가 어드레스의 작은 쪽에 저장
  - 엔디언은 통상 CPU에 의존

리틀엔디안 시스템에 저장된 값 : 0x0011

0	0	0	0	0	0	0	0			1	1	1	1	1	1	1	1
2^7	…	…	…	…	…	…	2^0			2^{15}	…	…	…	…	…	…	2^8

우리가 일반적으로 읽는 값 : 0x1100

1	1	1	1	1	1	1	1			0	0	0	0	0	0	0	0
2^{15}	…	…	…	…	…	…	2^8			2^7	…	…	…	…	…	…	2^0

7. 구조체와 공용체

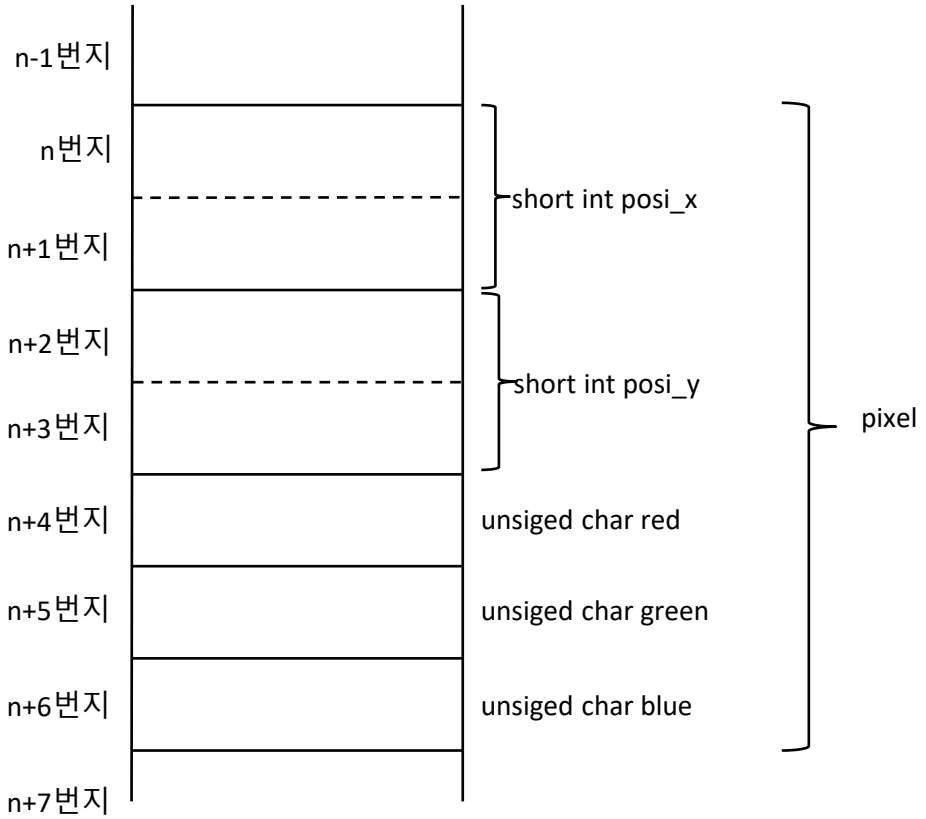
- 구조체
  - 내부에 구조를 가지고 있는 데이터
  - C 언어에서는 자유롭게 구제체를 정의
  - 새로운 자료형에 관해서도 기본 자료형과 마찬가지로 배열이나 포인터에 의한 조작 가능
  - 구조체 예시 : 비트맵 화상의 어떠한 한점에 대해 좌표와 색을 관리하는 변수 pixel 정의

```
struct s_sample
{
        short int           posi_x;
        short int           posi_y;
        unsigned char       red;
        unsigned char       green;
        unsigned char       blue;
} pixel;
```

7. 구조체와 공용체

- 구조체 메모리 이미지

```
struct s_sample
{
        short int           posi_x;
        short int           posi_y;
        unsigned char       red;
        unsigned char       green;
        unsigned char       blue;
} pixel;
```



7. 구조체와 공용체

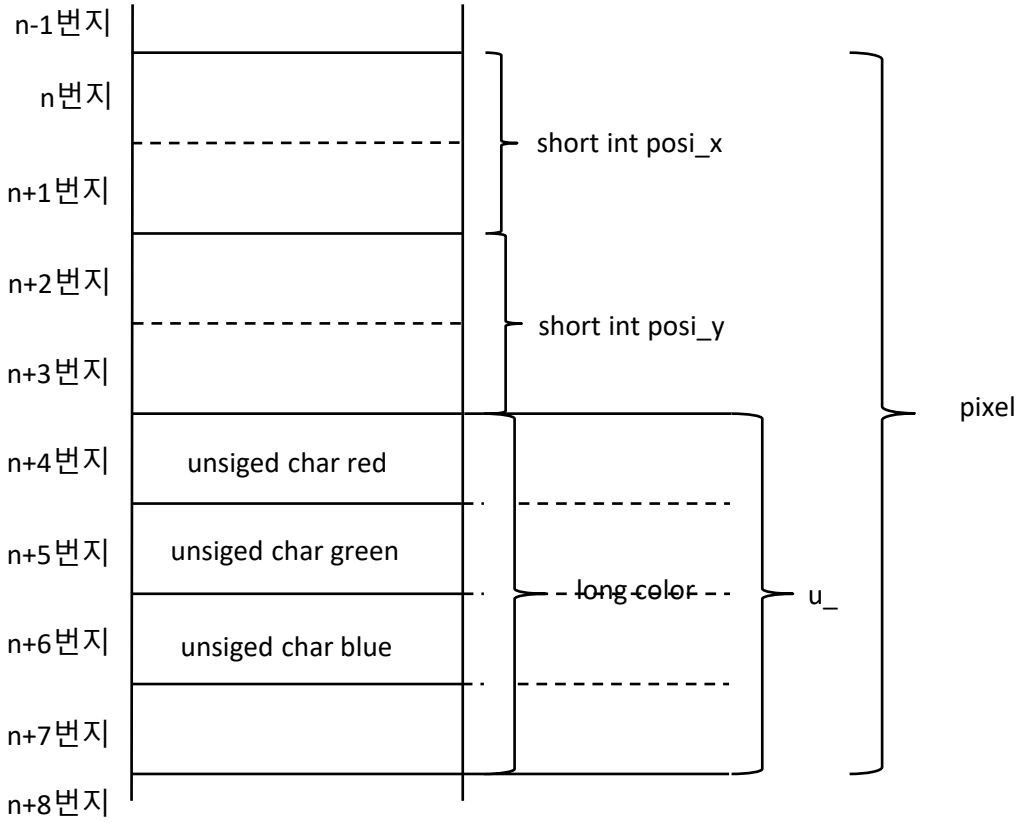
- 공용체 (Union)
  - 하나의 메모리 공간을 여러 개의 자료형이나 변수가 공유하는 것
  - 가장 큰 변수의 메모리를 만들어 나머지 변수들도 메모리를 같이 쓴다
  - 동시에 변수 여러 개에 접근 할 시에는 메모리를 사용할 수 없음, 이 같은 경우 구조체 사용
  - 공용체 예시 : 공용체를 구조체의 멤버로 내부에서 사용(red, green, blue의 각 색상 정보를 long 형의 변수 color로 일괄해서 다룰 수 있도록 함)

```
struct s_sample
{
        short int       posi_x;
        short int       posi_y;
        union
        {
                long color;
                struct
                {
                        unsigned char   red;
                        unsigned char   green;
                        unsigned char   blue;
                }
        } u_;
} pixel2;
```

7. 구조체와 공용체

- 공용체를 사용한 메모리 이미지 예

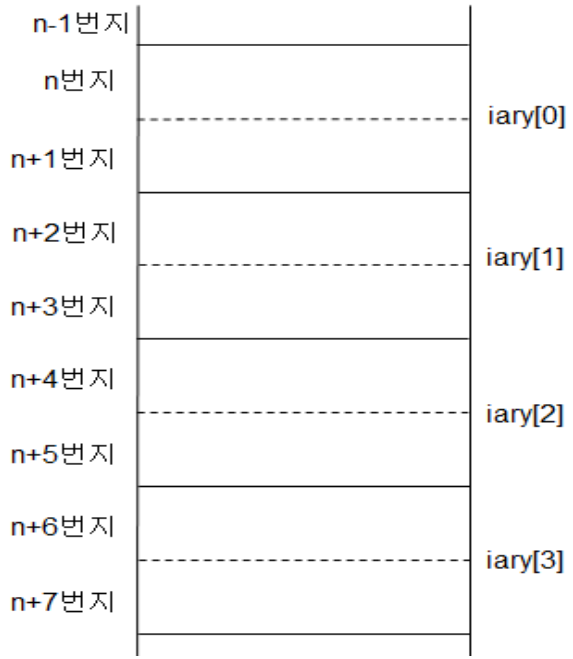
```
struct s_sample
{
        short int           posi_x;
        short int           posi_y;
        union
        {
                long color;
                struct
                {
                        unsigned char    red;
                        unsigned char    green;
                        unsigned char    blue;
                }
        } u_;
} pixel2;
```



- n+4번지 이후가 공용체 선언 부분으로 메모리 공간이 공용으로 되어 있음
- '공용'에 의해 프로그램에서는 각각의 unsigned char형의 멤버 또는 합쳐서 long형의 멤버 어느 쪽으로나 액세스 할 수 있음

8. 배열

- 배열
  - 배열은 지금까지 설명한 각 자료형 요소의 정렬
  - short, int 형의 배열은 메모리 상에서 다음과 같음



[short int iary[4]의 메모리 상에서의 이미지]

8. 배열

- 고급 배열 사용
  - 배열 이름만인 경우 배열의 선두 어드레스를 나타냄
    - short int *pointer;
    - pointer = iary;
    - ponter = &iary[0];

    이 둘은 동일한 의미를 나타냄
  - 배열의 m번째 요소 iary[m]을 변수 val에 대입하는 프로그램

    (val = iary[m]의 C 소스 코드는 실제로는 다음의 C 소스코드와 같은 식으로 배열 요소에 접근)

[배열에 대한 액세스의 등가 프로그램 1]

```
unsigned char *pbase;
short int val, m;
pbase = (unsigned char *) iary;
…
val = (short int)*(pbase + m*sizeof(short int));
```

[배열에 대한 액세스의 등가 프로그램 2]

```
short int *pbase;
short int val, m;
pbase = (short int *)iary;
…
val = (pbase + m);
```

8. 배열

- 구조체 배열
  - 배열의 m번째 요소인 pixel[m]의 멤버 posi_y를 변수 val에 대입하는 프로그램은 다음과 같이 나타낼 수 있음

[예 1]

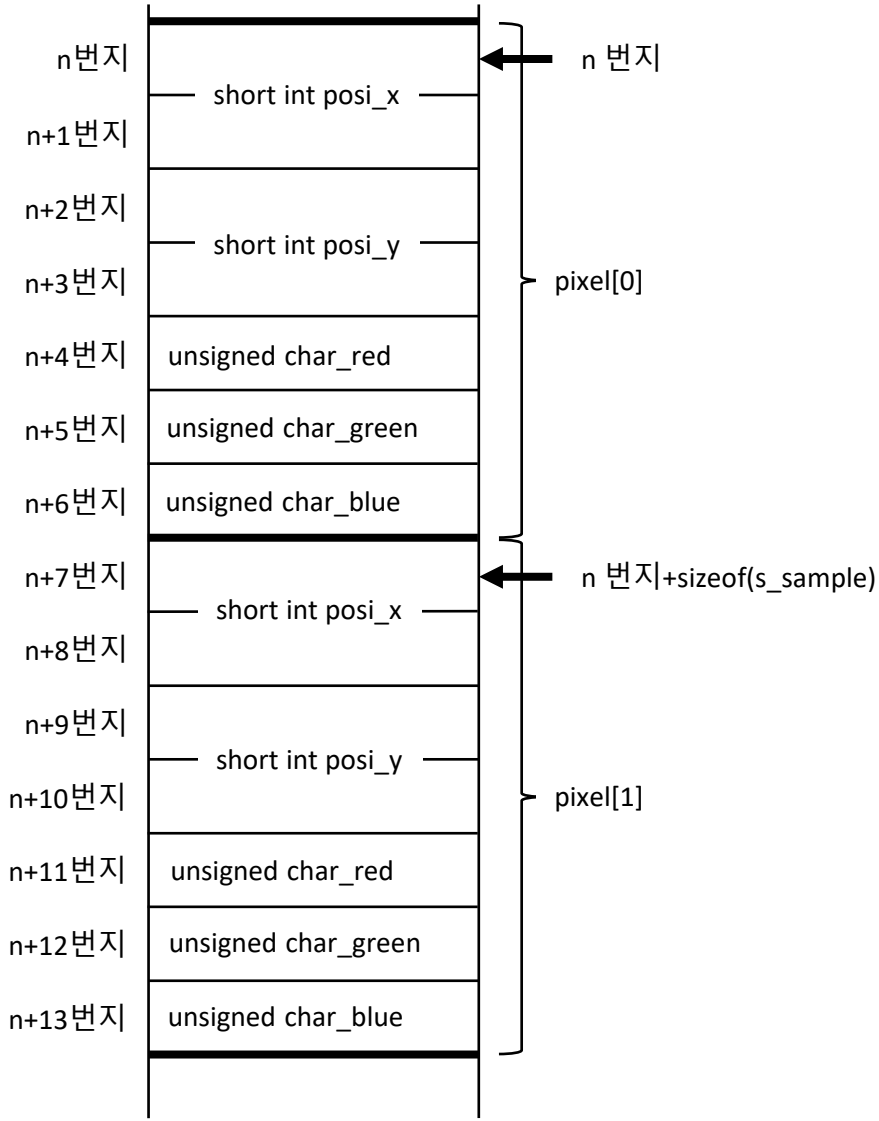
```
#define BUFSIZE  (10)

typedef struct {
        short int           posi_x;
        short int           posi_y;
        unsigned char       red;
        unsigned char       green;
        unsigned char       blue;
} s_sample;
unsigned char *pbase;
short int val, m;
s_sample sary[BUFSIZE], *p;
pbase = (unsigned char *)sary;
…
p = (s_sample *)(pbase + m*sizeof(s_sample));
val = p->posi_y;
```

8. 배열

- 구조체 배열의 메모리 이미지

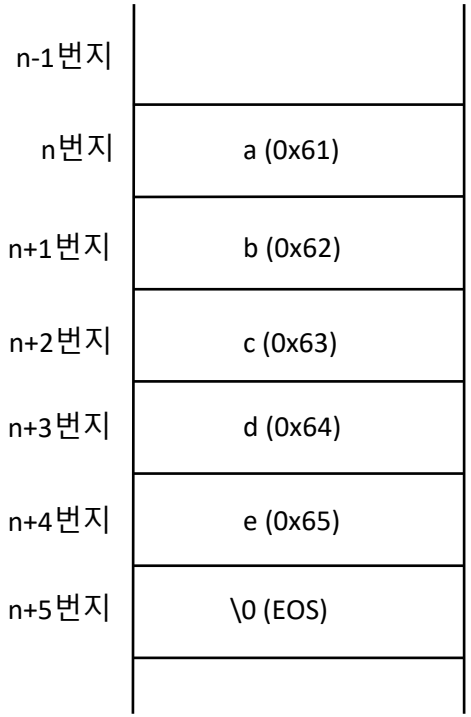
[예 1]

```
#define BUFSIZE  (10)

typedef struct {
        short int       posi_x;
        short int       posi_y;
        unsigned char   red;
        unsigned char   green;
        unsigned char   blue;
} s_sample;
unsigned char *pbase;
short int val, m;
s_sample sary[BUFSIZE], *p;
pbase = (unsigned char *)sary;
…
p = (s_sample *)(pbase + m*sizeof(s_sample));
val = p->posi_y;
```



9. 문자열

- 문자열
  - 문자열 데이터는 대부분이 가변형 크기를 가짐
  - 종료 문자 EOS의 존재를 잊어버리는 경우 많음
  - 문자열은 기본적으로 문자형의 배열이고 끝에는 반드시 0x00(' \ 0', 즉 EOS)가 덧붙여짐



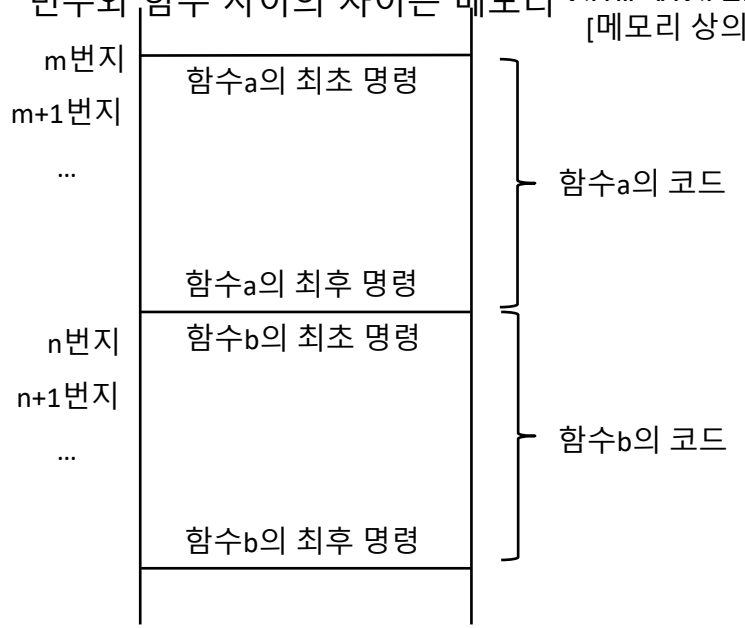
[문자열의 값 "abcede"를 저장한 모습]

[참 고]

- 'short int형, int형, long int형'에서 설명한 것처럼 CPU에는 엔디언이라는 속성 존재
- 임베디드 제품 사이에서 통신 등으로 데이터 교환을 하는 경우 해당 부분을 인지하고 있어야 함

10. 코드 배치와 포인터

- 코드 배치와 포인터
  - 메모리 공간에서는 변수와 마찬가지로 함수도 메모리 상에 실행 코드가 전개
  - 함수별로 고유한 선두 어드레스를 가짐
  - C++에서 인스턴스를 동적으로 생성하는 경우에도 생성된 인스턴스는 고유한 선두 어드레스로 관리
  - 함수도 포인터를 사용하여 호출 가능
  - 변수와 함수 사이의 차이는 메모리 상에 저장되어 있는 것이 값이 아니라 실행 코드라는 점임



11. I/O 포트

- I/O 포트
  - CPU는 디바이스의 I/O포트에 액세스함으로써 디바이스에 대해 데이터 입출력 실행
  - 메모리 맵 방식의 I/O에서는 어드레스 버스로 I/O 포트의 어드레스를 지정하고 데이터 버스로 포트에 읽기/쓰기를 함
  - 디바이스에 액세스하는 포트에는 자료형 존재
  - 포트의 폭을 의식하지 않고 프로그램을 작성하면 생각지 못한 결함이 발생할 가능 성 있음

[예 : 어드레스 0xff0c에 매핑된
8비트 폭 레지스터의 BIT1의 상태를 본다]

```
#define BIT1                (0x02)
unsigned char *port;
port = (unsigned char *)0xff0c;
if (*port & BIT1) {
    /* BIT1이 ON인 경우 처리 */
}
else {
    /* BIT1이 OFF인 경우의 처리 */
}
```

[예 : 어드레스 0xff0a에 매핑된
8비트 폭 레지스터에 값 0xc4을 설정한다]

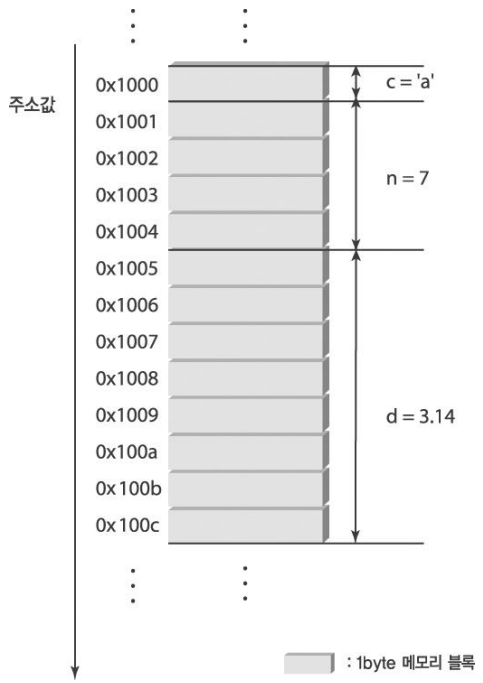
```
unsigned char *port;
port = (volatile unsigned char *)0xff0a
*port = 0xc4;
```

포인터

Overview

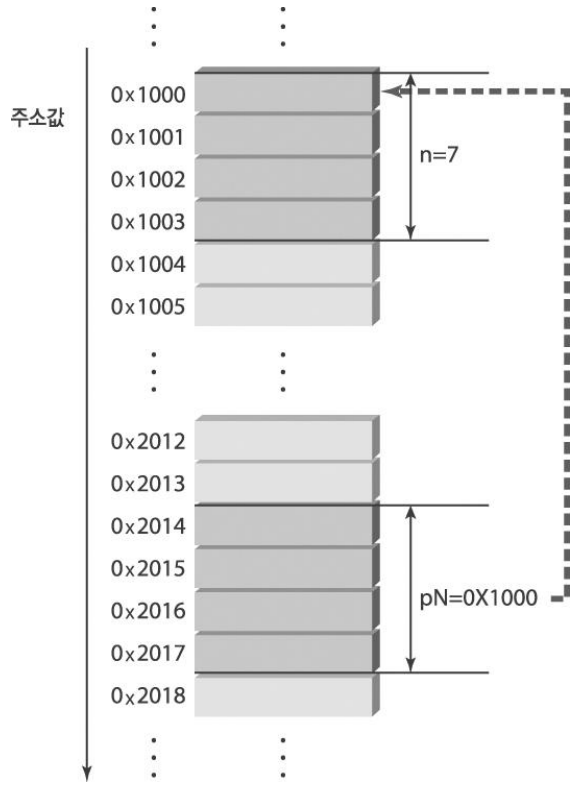
- 포인터와 포인터 변수
  - 메모리의 주소 값을 저장하기 위한 변수
  - "포인터"를 흔히 "포인터 변수"라 함
  - 주소 값과 포인터는 다른 것

```
int main(void)
{
    char c='a';
    int n=7;
    double d=3.14;
    . . . . .
```



포인터란 무엇인가?

- 그림을 통한 포인터의 이해
  - 컴퓨터의 주소 체계에 따라 크기가 결정
  - 32비트 시스템 기반 : 4 바이트



포인터란 무엇인가?

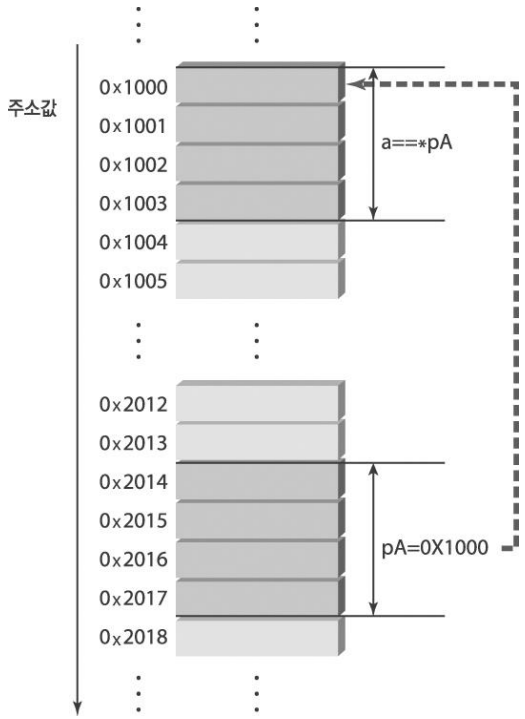
- 포인터의 타입과 선언
  - 포인터 선언 시사용되는 연산자 : *
  - A형 포인터(A*) : A형 변수의 주소값을 저장

```
int main(void)
{
   int *a;          // a라는 이름의 int형 포인터
   char *b;         // b라는 이름의 char형 포인터
   double *c;       // c라는 이름의 double형 포인터
    .....
```

포인터란 무엇인가?

- 주소 관련 연산자
  - & 연산자 : 변수의 주소 값 반환
  - * 연산자 : 포인터가 가리키는 메모리 참조

```
int main(void)
{
   int a=2005;
   int *pA=&a;
   printf("%d", a);   //직접 접근
   printf("%d", *pA);   // 간접 접근
   . . . . .
```



포인터란 무엇인가?

- 포인터에 다양한 타입이 존재하는 이유
  - 포인터 타입은 참조할 메모리의 크기 정보를 제공

```
#include <stdio.h>
 int main(void)
{
    int a=10;
    int *pA = &a;
    double e=3.14;
    double *pE=&e;

    printf("%d %f", *pA, *pE);
    return 0;
}
```

잘못된 포인터의 사용

- 사례1

```
int main(void)
{
   int *pA;     // pA는 쓰레기 값으로 초기화 됨
   *pA=10;
   return 0;
}
```

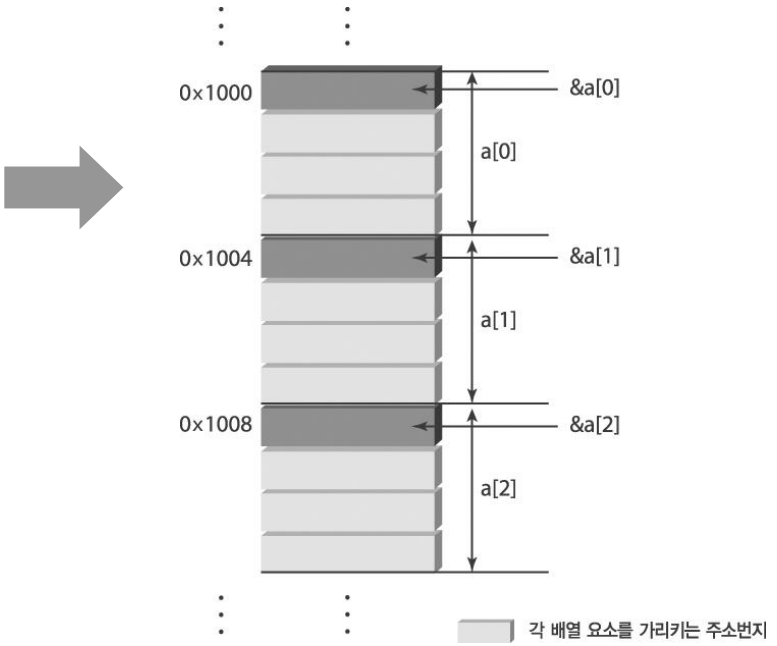
- 사례2

```
int main(void)
{
   int* pA=100;   // 100이 어딘 줄 알고???
   *pA=10;
   return 0;
 }
```

포인터와 배열

- 배열의 이름의 정체
  - 배열 이름은 첫 번째 요소의 주소 값을 나타낸다

```
int a[5]={0, 1, 2, 3, 4}
```



포인터와 배열

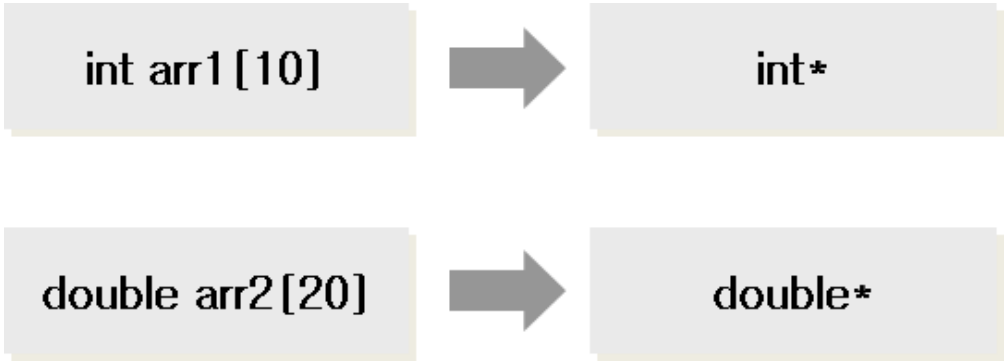
- 배열 이름과 포인터 비교
  - 배열 이름은 첫 번째 요소의 주소 값을 나타낸다

비교 조건 \ 비교 대상	포인터	배열 이름
이름이 존재하는가	물론 있음	당연히 있음
무엇을 나타내는가	메모리의 주소	메모리의 주소
변수인가 상수인가	변수	상수

```
int main(void)
{
    int a[5]={0, 1, 2, 3, 4};
    int b=10;
    a=&b;   //a는 상수이므로 오류, a가 변수였다면 OK!
}
```

포인터와 배열

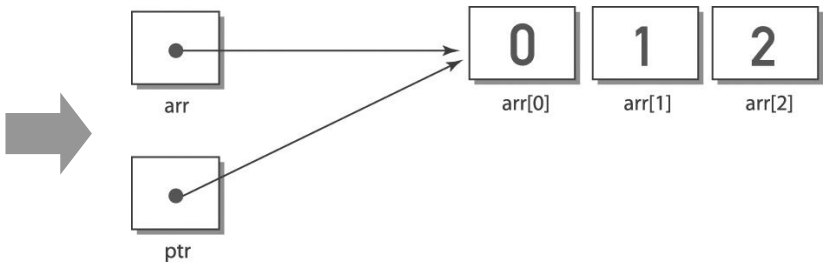
- 배열 이름의 타입
  - 배열 이름도 포인터이므로 타입이 존재
  - 배열 이름이 가리키는 배열 요소에 의해 결정



포인터와 배열

- 배열 이름의 활용
  - 배열 이름을 포인터처럼, 포인터를 배열 이름처럼 활용하는 것 가능

```
/* pointer_array2.c */
#include <stdio.h>

int main(void)
{
    int arr[3]={0, 1, 2};
    int *ptr;

    ptr=arr;

    printf("%d, %d, %d ₩n", ptr[0], ptr[1], ptr[2]);
    return 0;
}
```



포인터 연산

- 포인터 연산이란?
  - 포인터가 지니는 값을 증가 혹은 감소시키는 연산을 의미

```
ptr1++;
ptr1 += 3;
--ptr1;
ptr2=ptr1+2;
```

포인터 연산

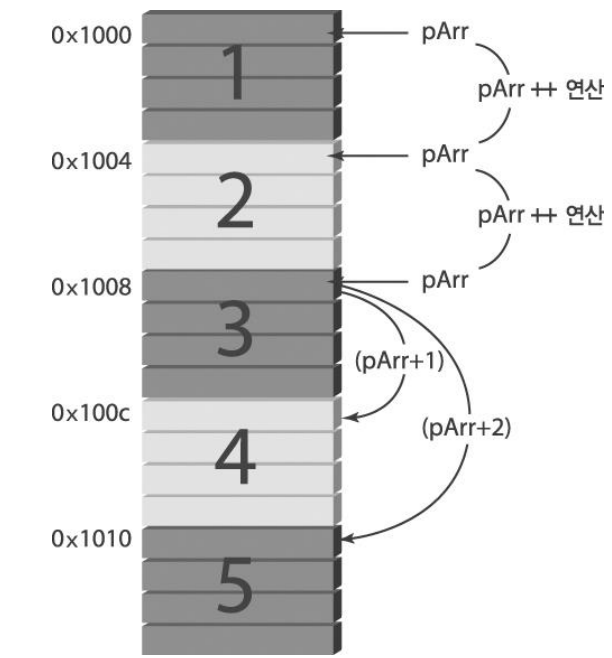
- 포인터 연산
  - 포인터가 가리키는 대상의 자료형에 따라서 증가 및 감소되는 값이 차이를 지님

```
/* pointer_op.c */
#include <stdio.h>

int main(void)
{
    int* ptr1=0;                    // int* ptr1=NULL; 과 같은 문장
    char* ptr2=0;                          // char* ptr2=NULL; 과 같은 문장
    double* ptr3=0;                 // double* ptr3=NULL; 과 같은 문장

    printf("%d 번지, %d 번지, %d 번지 ₩n", ptr1++, ptr2++, ptr3++);
    printf("%d 번지, %d 번지, %d 번지 ₩n", ptr1, ptr2, ptr3);

    return 0;
}
```

포인터 연산

- 포인터 연산을 통한 배열 요소의 접근

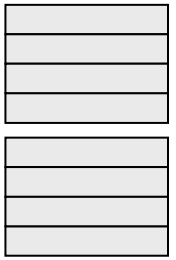
```
/* pointer_array3.c */
#include <stdio.h>

int main(void)
{
   int arr[5]={1, 2, 3, 4, 5};

   int* pArr=arr;
   printf("%d ₩n", *pArr);

   printf("%d ₩n", *(++pArr));
   printf("%d ₩n", *(++pArr));

   printf("%d ₩n", *(pArr+1));
   printf("%d ₩n", *(pArr+2));

   return 0;
}
```



포인터 연산

- 포인터 배열 활용

```
/* two_same.c */
#include <stdio.h>

int main(void)
{
   int arr[2]={1, 2};
   int* pArr=arr;

   printf("%d, %d ₩n", arr[0], *(arr+1));

   printf("%d, %d ₩n", pArr[0], *(pArr+1));

   return 0;
}
```



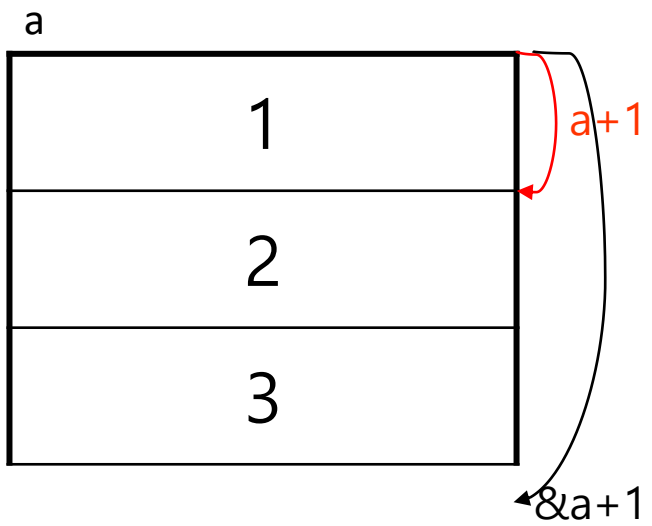
arr[i] == *(arr+i)

→ arr이 "포인터"이거나 "배열 이름"인 경우

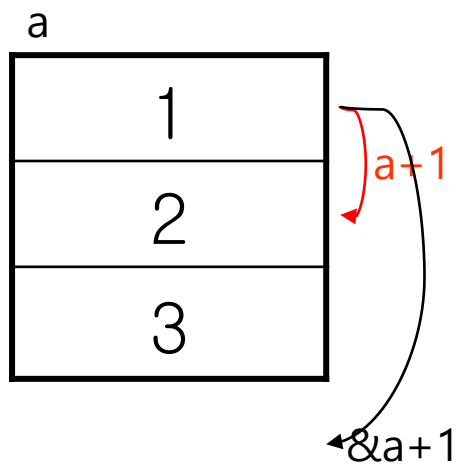
1차원 배열에서 배열의 이름

a => 배열 첫 요소에 대한 주소
=> 한 단위는 배열 한 칸

&a => 배열의 시작 주소
=> 한 단위는 배열전체



1중 포인터에서 +1의 의미



a == 0x100 a + 1 = 0x104

&a == 0x100 &a + 1 == 0x10C

a int[0]의 배열의 주소 : int * (1중 포인터)

&a int[3] 1차원 배열의 주소 : int (*)[3] (2중 포인터)

2차원 배열에서 배열의 이름

a[2][3] = {{1, 2, 3}, {4, 5, 6}};

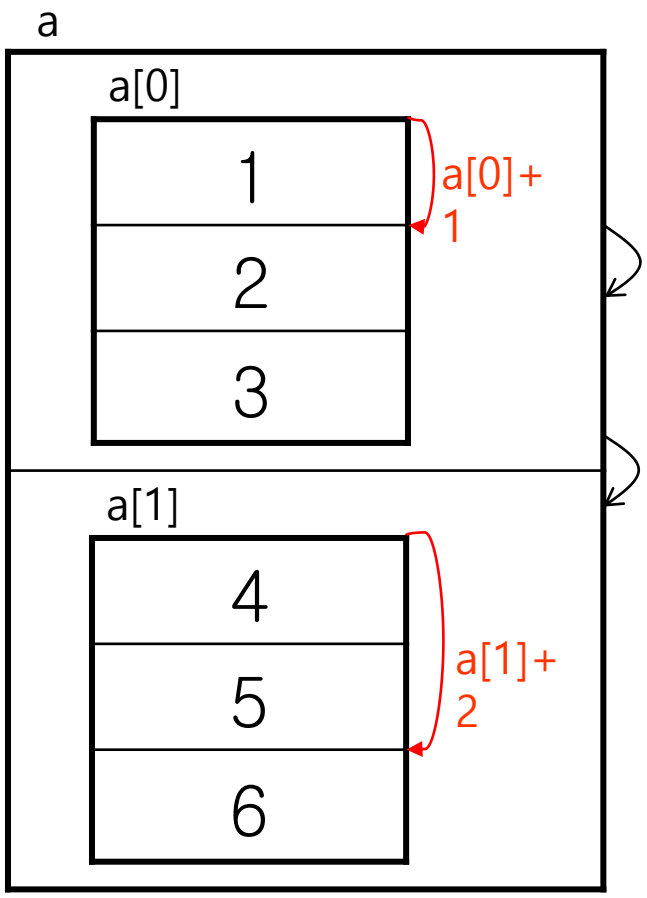
주소 표현			
배열 이름	실제주소	배열	주소표현
a[0] == *a	0x100	1	a[0] == *a
	0x104	2	a[0]+1 == *a+1
	0x108	3	a[0]+2 == *a+2
a[1] == *(a+1)	0x10C	4	a[1] == *(a+1)
	0x110	5	a[1]+1 == *(a+1)+1
	0x114	6	a[1]+2 == *(a+1)+2

2차원 배열에서 배열의 이름

a[2][3] = {{1, 2, 3}, {4, 5, 6}};

값 표현			
배열 이름	실제주소	배열	주소표현
a[0] == *a	0x100	1	a[0][0] == *a[0] == **a
	0x104	2	a[0][1] == *(a[0]+1) == *(*a+1)
	0x108	3	a[0][2] == *(a[0]+2) == *(*a+2)
a[1] == *(a+1)	0x10C	4	a[1][0] == *a[1] == **(a+1)
	0x110	5	a[1][1] == *(a[1]+1) == *(*(a+1)+1)
	0x114	6	a[1][2] == *(a[1]+2) == *(*(a+1)+2)

다중 포인터에서 +1의 의미



a+1

&a+1

a[0] == 0x100 a[0] + 1 = 0x104
a == 0x100 a + 1 = 0x10C
&a == 0x100 &a + 1 == 0x118

a[0] int의 주소 : int * (1중 포인터)
a int[3] 1차원 배열의 주소 : int (*)[3] (2중 포인터)
&a int[2][3] 2차원 배열의 주소 : int (*)[2][3] (3중 포인터)

void 포인터

- void형 포인터란 무엇인가?
  - 자료형에 대한 정보가 제외된, 주소 정보를 담을 수 있는 형태의 변수
  - 포인터 연산, 메모리 참조와 관련된 일에 활용 할 수 없음

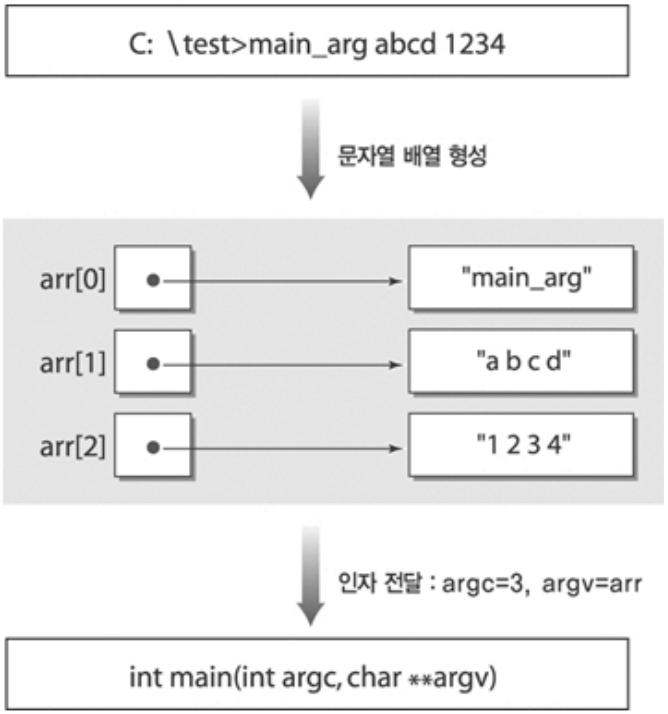
```
int main(void)
{
    char c='a';
    int n=10;
    void * vp;    // void 포인터 선언
    vp=&c;
    vp=&n;
    .....
```

```
int main(void)
{
    int n=10;
    void * vp=&n;
    *vp=20;              // Error!
    vp++;                // Error!
    .....
```

void 포인터

- main 함수의 인자 전달

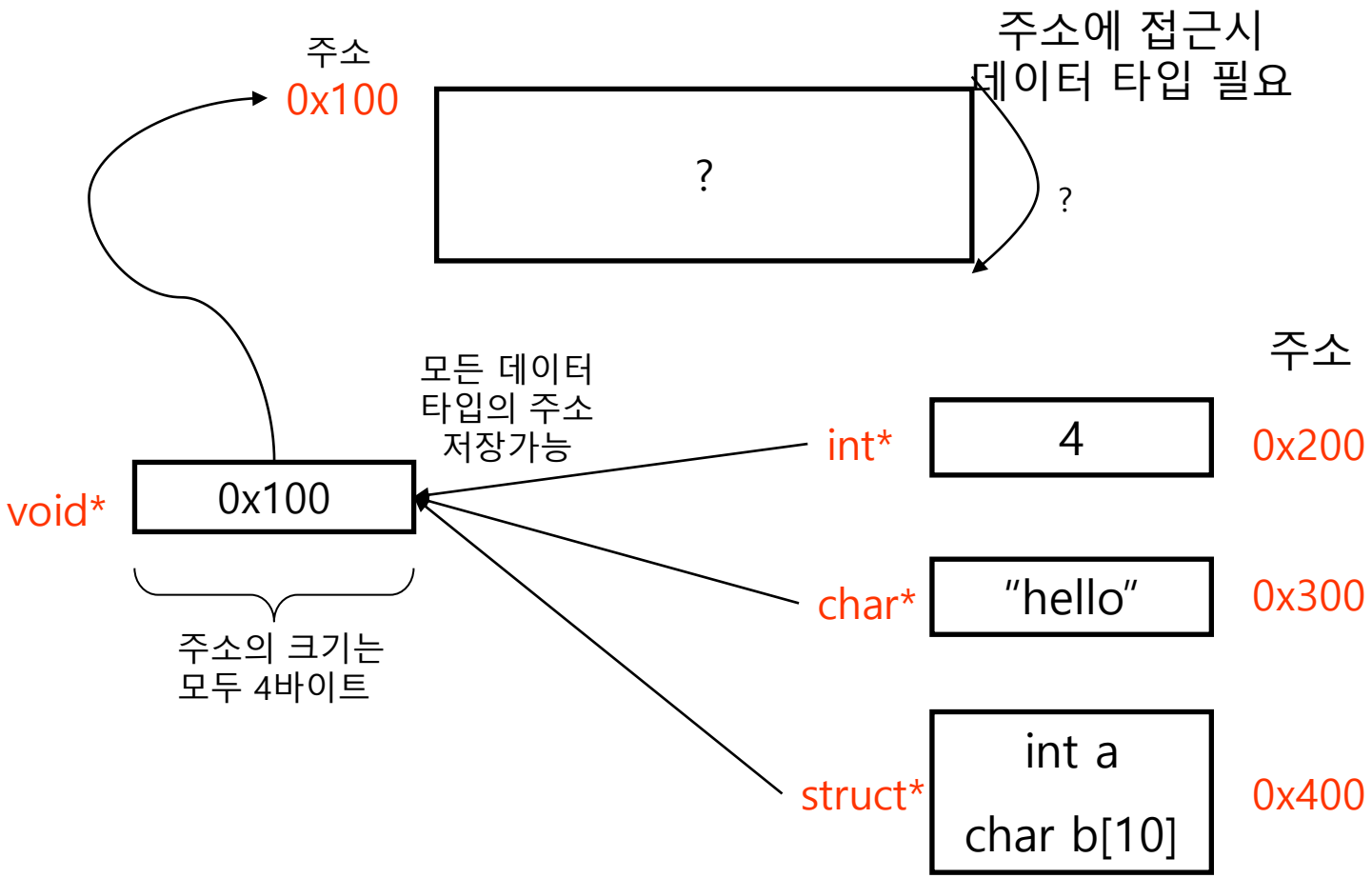
```
/* main_arg.c */
#include <stdio.h>

int main(int argc, char **argv)
{
    int i=0;
    printf("전달된 문자열의 수 : %d ₩n", argc);

    for(i=0; i<argc; i++)
        printf("%d번째 문자열 : %s ₩n", i+1, argv[i]);

    return 0;
}
```



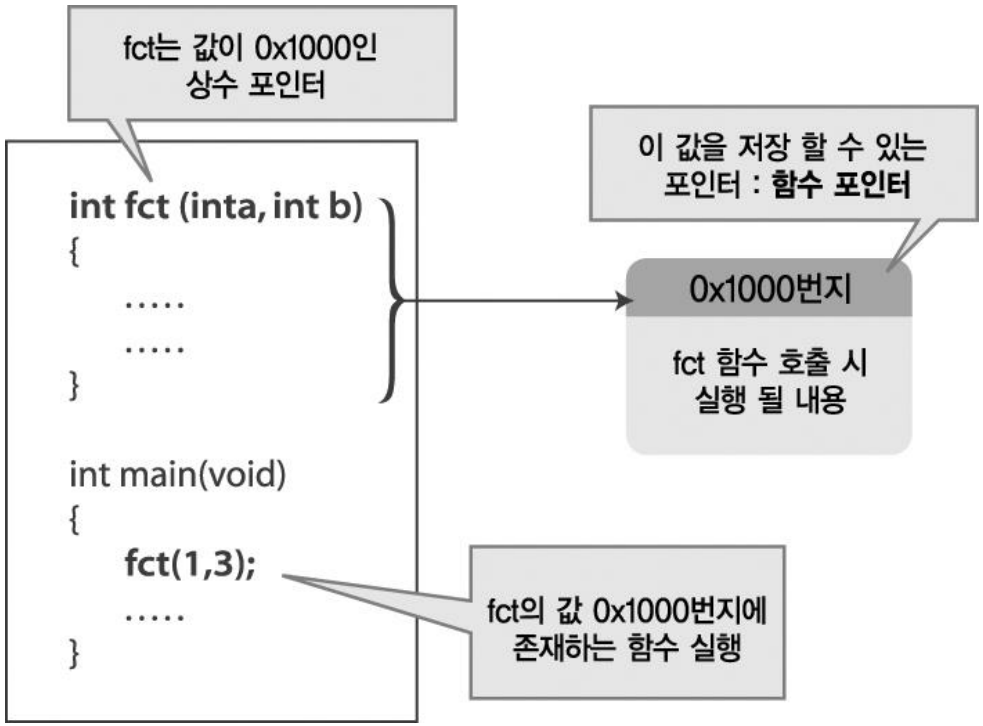
Example 3 – Void Pointer

```
1     #include <stdio.h>
2
3     //여러 타입의 데이터를 입력 받기 위해 인자의 형을 void 포인터로 지정
4     void add(void *p, void *q, void *s, int op);
5
6     void main(void) {
7         int a = 1, b = 2, sum_i;
8         float x = 1.5, y = 2.5, sum_f;
9
10        add(&a, &b, &sum_i, 1);
11        add(&x, &y, &sum_f, 2);
12
13        printf("int의 합=%d\n", sum_i);
14        printf("float의 합=%f\n", sum_f);
15    }
16
17    void add(void *p, void *q, void *s, int op) {
18        if(op == 1)
19            *(int *)s = *(int *)p + *(int *)q; // void 포인터는 연산 시 캐스팅
20        else if(op == 2)
21            *(float *)s = *(float *)p + *(float *)q;
```

Example 3 – Void Pointer



함수 포인터의 이해


fct는 값이 0x1000인
상수 포인터
int fct (inta, int b)
{
.....
.....
}
int main(void)
{
fct(1,3);
.....
}
이 값을 저장 할 수 있는
포인터 : 함수 포인터
0x1000번지
fct 함수 호출 시
실행 될 내용
fct의 값 0x1000번지에
존재하는 함수 실행

함수 포인터

- 함수 이름의 포인터 타입을 결정짓는 요소
  - 리턴 타입 + 매개 변수 타입

```
int fct1 (int a)
{
    a++
    return a;
}
```

```
int (*fPtr1) (int);
```

```
double fct2 (double a, double b)
{
    double add=a+b;
    return add;
}
```

```
double (*fPtr2) (double, double);
```

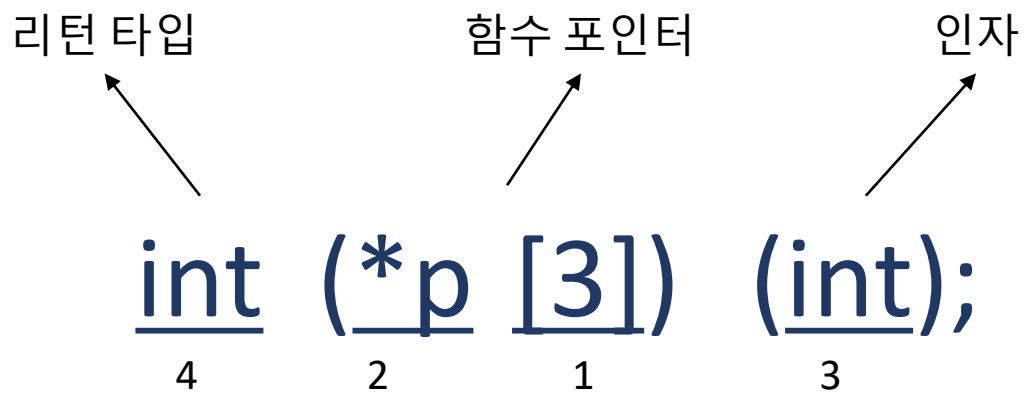
Example 4 – Function Pointer

```
1    #include <stdio.h>
2
3    int a(int);
4    int b(int);
5    int c(int);
6    int (*p[3])(int) = {a, b, c}  //함수 포인터 배열을 만들어
                                   //함수 주소 저장
7
8    void main(void) {
9        int x, y, z, i;
10       printf("\n메뉴\n1. 제곱\n");
11       printf("2. 3제곱\n");
12       printf("3. 4제곱\n");
13       printf("\n원하는 작동을 선택하시오\n");
14       scanf("%d", &i);
```

Example 4 – Function Pointer

```
15
16      // 선택한 메뉴에 따라 배열첨자를 이용해 함수호출
17      z = p[i-1](4);
18
19      printf("%d\n", z);
20  }
21  int a(int k) {
22      return k*k;
23  }
24  int b(int k) {
25      return k*k*k;
26  }
27  int c(int k) {
28      return k*k*k*k;
29  }
```

Example 4 – Function Pointer



포인터의 사용

- 대용량 데이터의 함수 전달
  - Call-by-reference
  - Remove pointer chain

- Heap 사용
  - malloc, free function

Large Data Access

<Ex1. Call-by-value>

```
1   #include <stdio.h>
2   void f1(struct Test x);
3
4   struct Test{
5       int a;
6       float b;
7       char c;
8   };
9
```

Large Data Access

```
10  void main {
11      struct Test t1;
12      t1.a = 3;
13      t1.b = 5.5;
14      t1.c = ‘x’;
15      f1(t1);
16  }
17
18  void f1(struct Test x) {
19      printf(“%d, %f, %c\n”, x.a, x.b, x.c);
20  }
```

Large Data Access

<Ex2. Call-by-reference>

```
1   #include <stdio.h>
2   void f1(struct Test *x);
3
4   struct Test{
5       int a;
6       float b;
7       char c;
8   };
9
```

Large Data Access

```
10  void main() {
11      struct Test t1, *p;
12      p = &t1;
13      t1.a = 3;
14      t1.b = 5.5;
15      t1.c = ‘x’;
16      f1(p);
17  }
18
19  void f1(struct Test *x) {
20      printf(“%d, %f, %c\n”, x->a, x->b, x->c);
21  }
```

Make read-only parameter

```
1   #include <stdio.h>
2   void f1(const struct Test *x);
3
4   struct Test{
5       int a;
6       float b;
7       char c;
8   };
9
```

Remove Pointer Chain

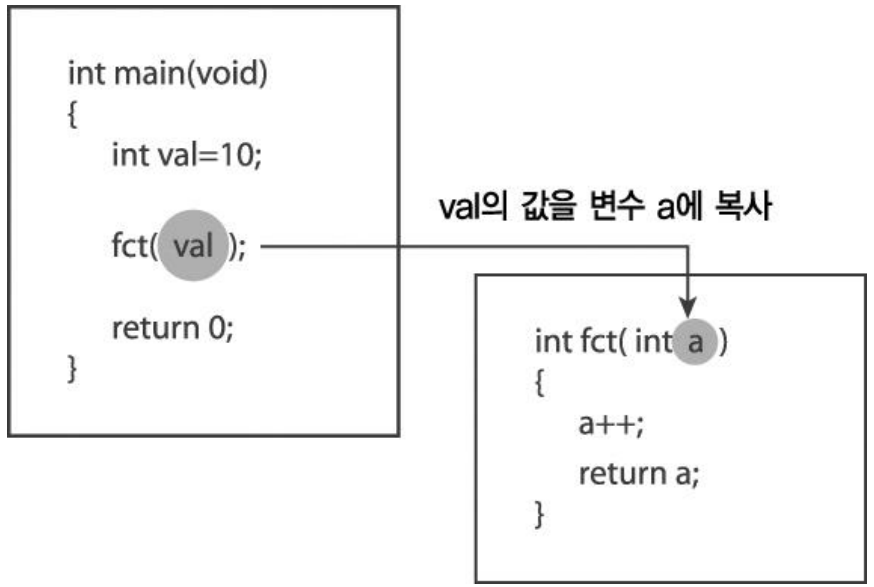
일반 코드	포인터 체인 제거

struct Point{
  int x, y, z;
};
 	
struct Obj{
  Point *p1, *d;
};
 	
void draw(struct Obj *a) {
  a->p1->x = 0;
  a->p1->y = 0;
  a->p1->z = 0;
}
	struct Point{
	  int x, y, z;
	};
	
	struct Obj{
	  Point *p1, *d;
	};
	
	void draw(struct Obj *a) {
	  struct Point *k = a->p1;
	  k->x = 0;
	  k->y = 0;
	  k->z = 0;
	}

함수 호출 규약

함수의 인자로 배열 전달

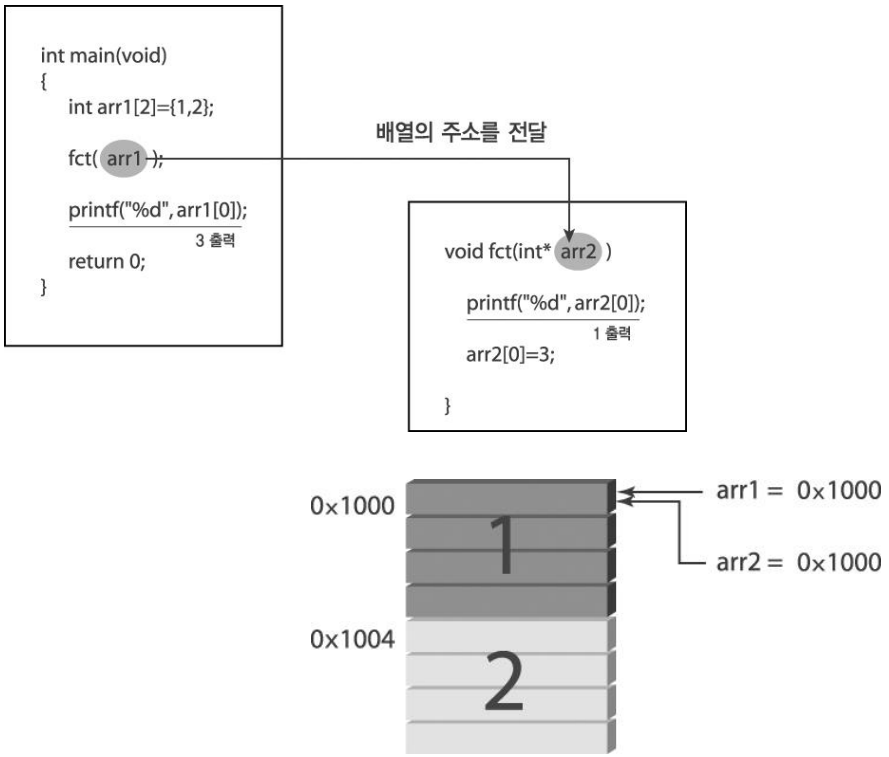
- 기본적인 인자의 전달 방식
  - 값의 복사에 의한 전달



함수의 인자로 배열 전달

- 배열의 함수 인자 전달 방식
  - 배열 이름(배열 주소, 포인터)에 의한 전달

```
#include <stdio.h>
void fct(int *arr2);

int main(void)
{
    int arr1[2]={1, 2};

    fct(arr1);
    printf("%d ₩n", arr1[0]);
    return 0;
}
void fct(int *arr2)
{
    printf("%d ₩n", arr2[0]);
    arr2[0]=3;
}
```



함수의 인자로 배열 전달

- 배열 이름, 포인터의 sizeof 연산
  - 배열 이름 : 배열 전체 크기를 바이트 단위로 반환
  - 포인터 : 포인터의 크기(4)를 바이트 단위로 반환

```
#include <stdio.h>

int main(void)
{
   int arr[5];
   int* pArr=arr;

   printf("%d ₩n", sizeof(arr) );          // 20 출력
   printf("%d ₩n", sizeof(pArr) );         // 4 출력
   return 0;
}
```

함수의 인자로 배열 전달

- "int *pArr" vs. "int pArr[]"
  - 둘 다 같은 의미를 지님
  - 선언 "int pArr[]"은 함수의 매개 변수 선언 시에만 사용 가능

```
int function(int pArr[])
{
   int a=10;
   pArr=&a;          // pArr이 다른 값을 지니게 되는 순간
   return *pArr;
}
```

Call-By-Value와 Call-By-Reference

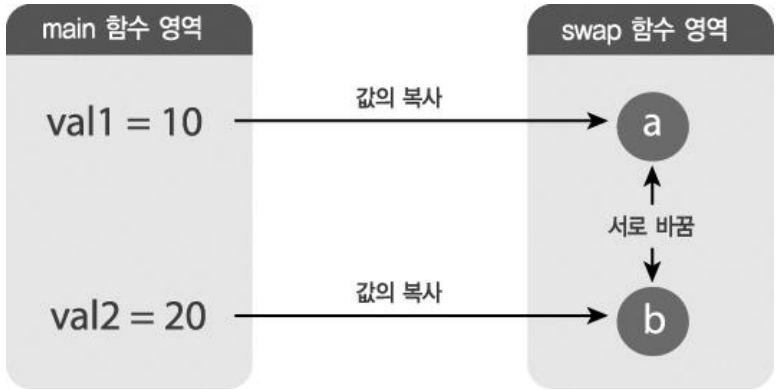
- Call-By-Value
  - 값의 복사에 의한 함수의 호출
  - 가장 일반적인 함수호출 형태

```
#include <stdio.h>
int add(int a, int b);

int main(void)
{
    int val1=10;
    int val2=20;
    printf(" 결  과 : ", add(val1, val2);

    return 0;
}
int add(int a, int b)
{
    return a+b;
}
```

Call-By-Value와 Call-By-Reference

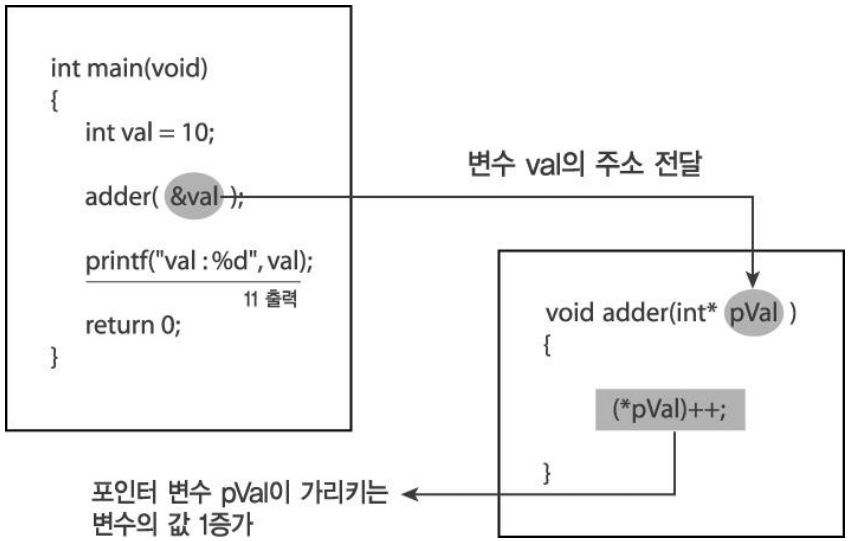
- Call-By-Value에 의한 swap

```
int main(void)
{
    int val1=10;
    int val2=20;
    swap(val1, val2);

    printf("val1 : %d ₩n", val1);
    printf("val2 : %d ₩n", val2);
    return 0;
}
void swap(int a, int b)
{
    int temp=a;
    a=b;
    b=temp;

    printf("a : %d ₩n", a);
    printf("b : %d ₩n", b);
}
```



Call-By-Value와 Call-By-Reference

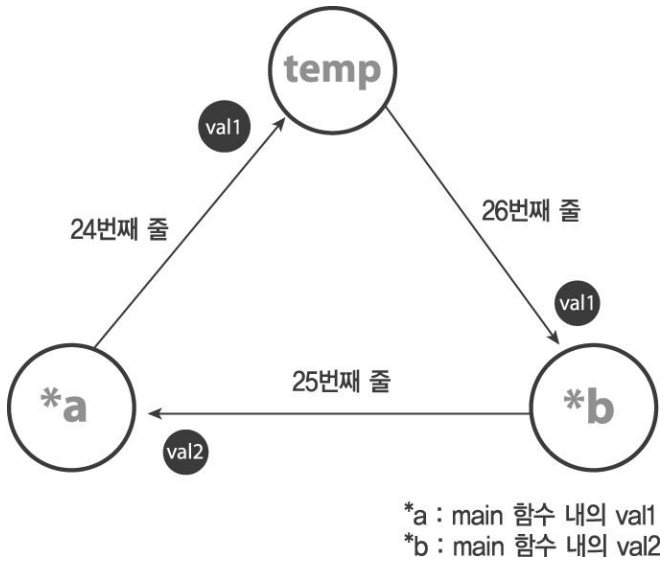
- Call-By-Reference
  - 참조(참조를 가능케 하는 주소 값)를 인자로 전달하는 형태의 함수 호출



Call-By-Value와 Call-By-Reference

- Call-By-Reference에 의한 swap

```
int main(void)
{
    int val1=10;
    int val2=20;
    printf("Before val1 : %d ₩n", val1);
    printf("Before val2 : %d ₩n", val2);
    swap(&val1, &val2);      //val1, val2 주소 전달

    printf("After val1 : %d ₩n", val1);
    printf("After val2 : %d ₩n", val2);
    return 0;
}
void swap(int* a, int* b)
{
    int temp=*a;
    *a=*b;
    *b=temp;
}
```



Call-By-Value와 Call-By-Reference

- scanf함수 호출 시 &를 붙이는 이유
  - case 1

```
int main(void)
{
    int val;
    scanf("%d", &val);
    .....
```

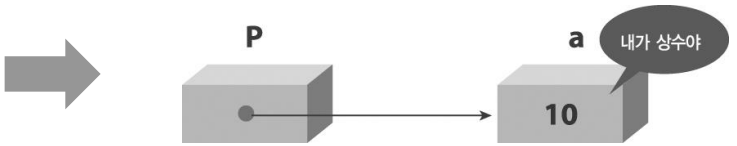
  - case 2

```
int main(void)
{
    char str[100];
    printf("문자열 입력 : ");
    scanf("%s", str);
    .....
```

포인터와 const 키워드

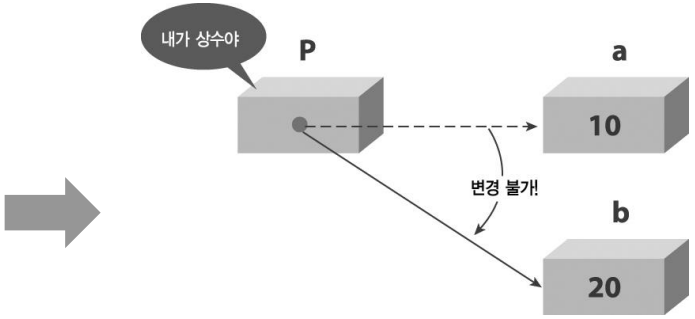
- 포인터가 가리키는 변수의 상수화

```
int a = 10;
const int * p = &a;
*p=30          // Error!
a=30           // OK!
```



- 포인터 상수화

```
int a=10;
int b=20;
int * const p = &a;
p=&b           // Error!
*p=30          // OK!
```



포인터와 const 키워드

- const 키워드를 사용하는 이유
  - 컴파일 시 잘못된 연산에 대한 에러 메시지
  - 프로그램을 안정적으로 구성

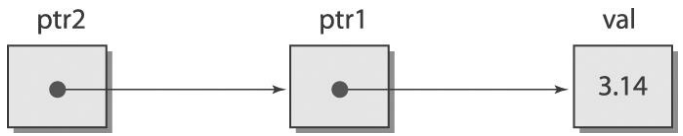
```
#include <stdio.h>
float PI=3.14;

int main(void)
{
    float rad;
    PI=3.07;     // 분명히 실수!!

    scanf("%f", &rad);
    printf("원의 넓이는 %f ₩n", rad*rad*PI);
    return 0;
}
```

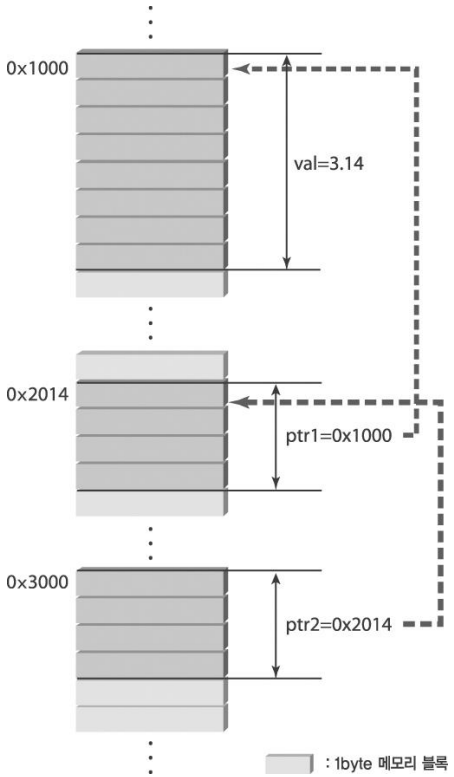
```
#include <stdio.h>
const float PI=3.14;

int main(void)
{
    float rad;
    PI=3.07;     // Compile Error 발생!

    scanf("%f", &rad);
    printf("원의 넓이는 %f ₩n", rad*rad*PI);
    return 0;
}
```

포인터의 포인터

- 포인터의 포인터
  - 더블 포인터라고 불림
  - 싱글 포인터의 주소 값을 저장하는 용도의 포인터

```
int main(void)
{
   double val=3. 14;
   double *ptr1 = &val;    // 싱글 포인터
   double **ptr2 = &ptr1; //더블 포인터
   . . .
```



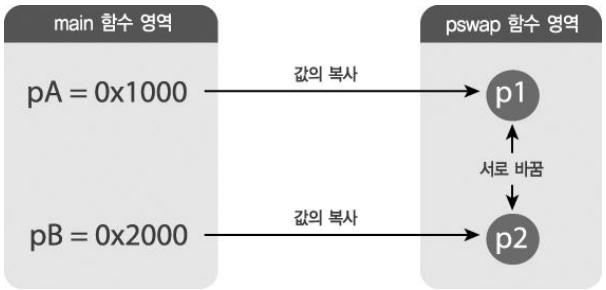


포인터의 포인터

- 구현 사례 1 : 효과 없는 swap 함수의 호출

```c
/* ptr_swap1.c */
#include <stdio.h>

void pswap(int *p1, int *p2);

int main(void)
{
    int A=10, B=20;
    int *pA, *pB;
    pA=&A, pB=&B;

    pswap(pA, pB);

    // 함수 호출 후
    printf("pA가 가리키는 변수 : %d ₩n", *pA);
    printf("pB가 가리키는 변수 : %d ₩n", *pB);

    return 0;
}
```

```c
void pswap(int *p1, int *p2)
{
    int *temp;
    temp=p1;
    p1=p2;
    p2=temp;
}
```

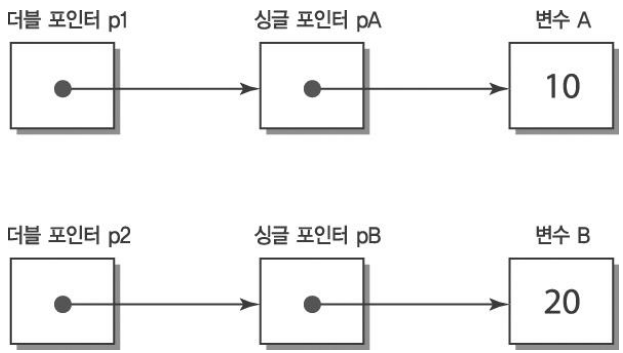


포인터의 포인터

- 구현 사례 2 : 더블 포인터 입장에서의 swap

```
/* ptr_swap2.c */
#include <stdio.h>

void pswap(int **p1, int **p2);

int main(void)
{
    int A=10, B=20;
    int *pA, *pB;
    pA=&A, pB=&B;

    pswap(&pA, &pB);

    //함수 호출 후
    printf("pA가 가리키는 변수 : %d ₩n", *pA);
    printf("pB가 가리키는 변수 : %d ₩n", *pB);

    return 0;
}
```

```
void pswap(int **p1, int **p2)
{
    int *temp;
    temp=*p1;
    *p1=*p2;
    *p2=temp;
}
```



포인터의 포인터

- 포인터 배열과 포인터 타입
  - 1차원 배열의 경우 배열이름이 가리키는 대상을 통해서 타입 결정
  - 포인터 배열이라 하더라도 동일함

```
int* arr1[10];
double* arr2[20];
char* arr3[30];
```

비트 조작

Specific Address Access

- Pointer

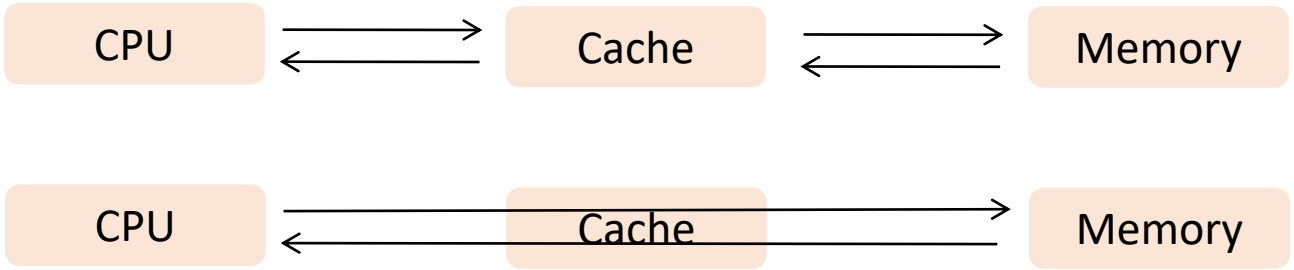
```
char *p = (char *)0x20001000;
*p = 0x80;
```

- Direct access without pointer

```
0x20001000 = 0x80;                          (x)
*(*)0x20001000 = 0x80;                      (x)
*(char*)0x20001000 = 0x80;                  (x)
*(unsigned char*)0x20001000 = 0x80;         (▲)
*(volatile unsigned char*)0x20001000 = 0x80;   (o)

#define PA (*(volatile unsigned char*)0x20001000)
PA = 0x80;
```

Volatile

- No cache



- No optimization

```
while (1)
    PA = 0x8000;
```

```
PA = 0x8000;
```

비트조작 명령어 정리

- & 연산자 : 비트단위로 AND 연산
- | 연산자 : 비트단위로 OR 연산
- ^ 연산자 : 비트단위로 XOR 연산
- ~ 연산자 : 피연산자의 모든 비트를 반전
- << 연산자 : 비트를 왼쪽으로 shift 연산
  - num1 << num2 : num1의 비트를 num2 비트만큼 왼쪽으로 이동.(이동후 나머지는 0으로 채운다)
- >> 연산자 : 비트를 오른쪽으로 shift 연산
  - num1 >> num2 : num1의 비트를 num2 비트만큼 오른쪽으로 이동.(이동후 나머지는 0으로 채운다)

&	결과
0&0	0
0&1	0
1&0	0
1&1	1

|	결과
0|0	0
0|1	1
1|0	1
1|1	1

^	결과
0^0	0
0^1	1
1^0	1
1^1	0

~	결과
~0	0
~1	1

Specific Bit Set Operation

5번 비트를 1로 설정해라.

```
PA  = 0x0303;
PA |= 0b00100000;  // 0x20

PA |= 0x1 << 5;
```

```
      0000_0011_0000_0011
OR    0000_0000_0010_0000
      0000_0011_0010_0011
```

2,3,5번 비트를 1로 설정해라.

```
PA |= (0x1 << 2) + (0x1 << 3) + (0x1 << 5);
PA |= (0x3 << 2) + (0x1 << 5);
```

Specific Bit Clear Operation

8번 비트를 0으로 설정해라.

```
PA  = 0x0303;
PA &= 0b1111_1110_1111_1111;

PA &= ~(0x1 << 8);
```

```
  0000_0011_0000_0011
AND
  1111_1110_1111_1111
  0000_0010_0000_0011
```

1,8,9번 비트를 0으로 설정해라.

```
PA &= ~((0x1 << 1) + (0x1 << 8) + (0x1 << 9));
PA &= ~((0x1 << 1) + (0x3 << 8));
```

Specific Bit Toggle Operation

8번 비트를 toggle해라.

```
PA  = 0x0303;
PA ^= 0b0000_0001_0000_0000;

PA ^= 0x1 << 8;
```

```
        0000_0011_0000_0011
XOR     0000_0001_0000_0000
        0000_0010_0000_0011
```

1,8,9번 비트를 toggle해라.

```
PA ^= (0x1 << 1) + (0x1 << 8) + (0x1 << 9);
PA ^= (0x1 << 1) + (0x3 << 8);
```

Macro Bit Operation

```
#define CLEAR_BIT(data, bit) ((data) &= ~(0x1 << (bit c)))
#define CLEAR_BITS(data, area, bit)  ((data) &= ~((area) << (bit)))
#define SET_BIT(data, bit) ((data) |= (0x1 << (bit)))
#define SET_BITS(data, area, bit)  ((data) |= ((area) << (bit)))
#define TOGGLE_BIT(data, bit) ((data) ^= (0x1 << (bit)))
#define TOGGLE_BITS(data, area, bit)  ((data) ^= ((area) << (bit)))
#define CHECK_BIT(data, bit) ((data) & (0x1 << (bit)))
#define EXTRACT_BITS(data, area, bit) (((data) >> (loc)) & (area))
```

```
Example)
CLEAR_BIT(a, 5) // 5번비트 클리어
CLEAR_BITS(a, 0x7, 3)  // 5,4,3번의 연속 3비트 클리어
EXTRACT_BITS(a, 0x7, 4); // 6,5,4 번 비트를 추출하여 b에 대입
```

라이브러리

Overview

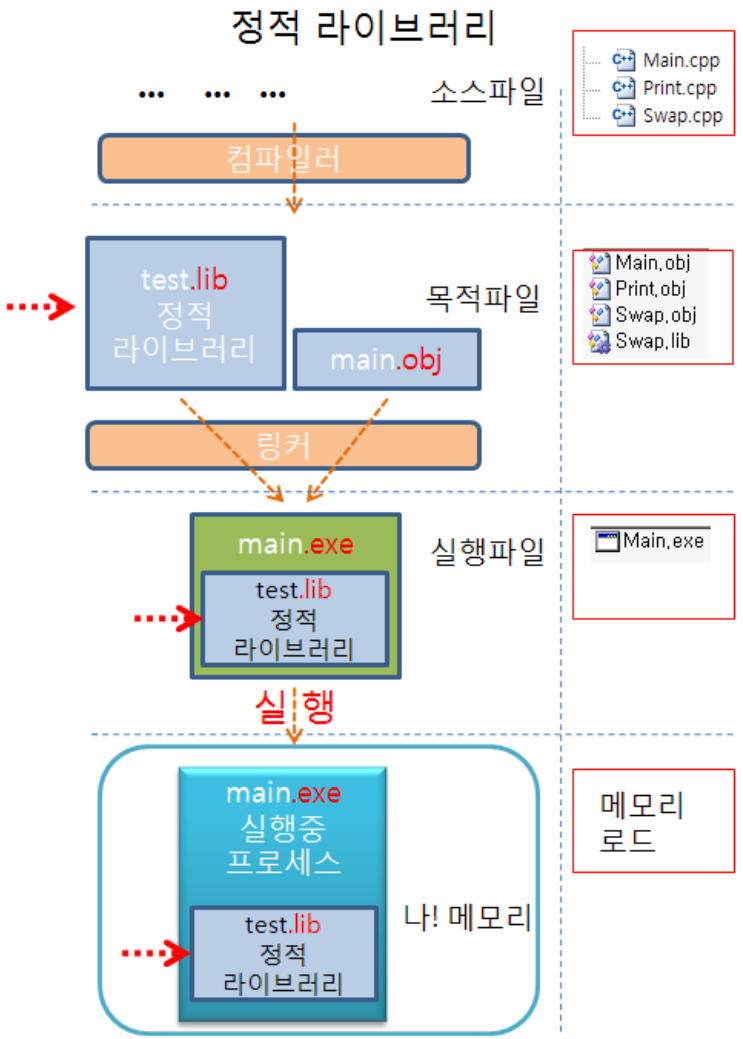
- 라이브러리(Library)
  - 다른 프로그램과 링크되기 위하여 존재하는 하나 이상의 서브루틴이나 function들이 저장된, 파일들의 모음
  - 함께 링크될 수 있도록 보통 컴파일된 형태(object module)로 존재
  - 라이브러리는 코드 재사용을 위해 조직화된 초창기 방법 중의 하나이며, 많은 다른 프로그램에서 사용할 수 있도록 운영체계나 소프트웨어 개발 환경제공자들에 의해 제공되는 경우가 많음
  - 라이브러리 내에 있는 루틴들은 두루 쓸 수 있는 범용일 수도 있지만, 3차원 애니메이션 그래픽 등과 같이 특별한 용도의 function으로 설계될 수도 있음

라이브러리 종류

- 라이브러리 종류
  - 정적 라이브러리
    - 정적 라이브러리는 컴파일러가 소스 파일을 컴파일 할 때 참조되는 프로그램 모듈
    - 정적 라이브러리는 루틴 외부 함수와 변수들의 집합으로, 컴파일러, 링커, 바인더 등에 의해 목표된 애플리케이션으로 복사되어 오브젝트 파일과 독립적으로 실행할 수 있는 실행 파일을 생성하는데 사용
  - 동적 라이브러리
    - 소프트웨어의 일종으로, 말 그대로 동적 링크를 사용한 라이브러리
    - 여러 프로그램이 공통으로 필요로 하는 기능을 프로그램과는 분리하여 필요할 때에만 불러내어 쓸 수 있게 만들어 놓은 라이브러리
    - 예)
      - [마우스가 지금 화면 어디에 있는지를 조사]
      - 해당 기능은 다양한 프로그램(응용 프로그램)이 공통적으로 사용하려는 기능으로 여겨지므로, 그 부분만을 모듈화하고, 여러 프로그램들이 사용할 수 있도록 하는 것이 효율적임
      - 이 같은 기능을 동적 라이브러리로서 만들어 놓는 경우가 많음

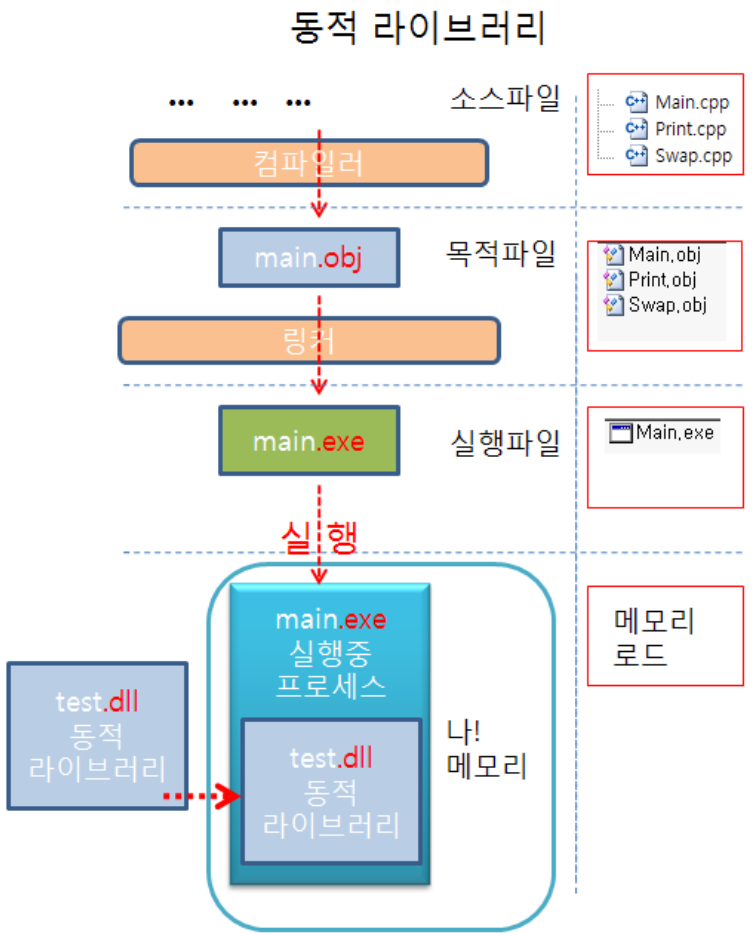
정적 라이브러리

- 정적 라이브러리
  - 정적 라이브러리는 단순히 보통의 목적파일(object file)의 모음
  - 공유 라이브러리의 이점들 때문에 예전만큼 많이 쓰이지는 않는다
  - 하지만 현재에도 사용은 되고 있고 라이브러리의 개념을 이해하기 쉽다



동적 라이브러리

- 동적 라이브러리
  - 정확히는 동적 연동 라이브러리라고 함
  - 라이브러리를 하나의 메모리 공간에 맵핑한 후, 여러 프로그램에서 공유하여 활용
  - 메모리, 용량 절약 차원의 장점
  - 라이브러리 업데이트 등의 유연성을 가지고 있음
  - 라이브러리 의존성에 따른 관리 필요

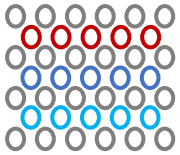


라이브러리 비교

[정적 라이브러리] [동적 라이브러리]

Q & A

Thank you for your attention



Architecture and Compiler
for Embedded Systems Lab.

School of Electronics Engineering, KNU

ACE Lab. (jcho@knu.ac.kr)

ACE Lab.

KNU