2020학년도 1학기 학생취업 포토폴리오

학부	컴퓨터공학부
학과	컴퓨터정보공학과
학번	20164114
성명	김태헌

목차

11장. 문자와 문자열

1. 11-1 문자와 문자열

-문자와 문자열의 개념

프로그램에서는 대부분이 수 또는 문자, 문자열이다. 문자는 영어의 알파벳, 한글의 한글자를 작은 따옴표 '' 로 둘러싸 'A'와 같이 표기하며 C언어에서 저장공간 1바이트 크키인 char자료형으로 지원한다. 이를 문자 상수 라고 한다.

문자의 모임인 일련의 문자를 문자열(String) 이라 하고 큰따옴표 "" 로 둘러싸 "java" 라고 표기한다.

=문자와 문자열 선언

C언어에서 문자는 자료형 char을 제공해주지만 문자열은 자료형을 따로 제공하지 않기 때문에 '문자배열'을 사용한다. 여기서 주의할 점은 문자열의 마지막에는 NULL문자 '/0'이 저장되어야 한다. 그러므로 배열의 크기가 저장되는 문자 수 보다 1이 더 커야된다.

-문자열 선언을 쉽게 하는법

*char java[] = {'J','A','V','A','/0'}; - 초기화 방법

*char java[] = "JAVA";

*char java[10] = "JAVA"; - 남는 배열의 공간에는 '/0'으로 채워짐

-printf()를 사용한 문자와 문자열 출력

함수 printf()는 형식제어문자 %c로 문자를 문자열은 배열이름 또는 문자 포인터를 사용하여 형식제어문자 %s로 출력한다. put(csharp)와 같이 사용하면 한 줄에 문자열을 출력한 후 다음줄에서 출력을 준비한다. printf(c)처럼 배열이름을 인자로 사용해도 문자열 출력이 가능하다.

=문자열 구성하는 문자참조

다른 방법으로는 문자열 상수를 문자 초인터에 저장하는 방법이 있다. 문자 포인터 변수에 문자열 상수를 저장할 수 있는데 문자열 출력도 함수 printf()에서 포인터 변수와 형식제어문자 %s로 간단히 처리할 수 있다. 단 이런 처리방식은 문자 포인터에 의한 선언으로 문자 하나하나의 수정은 할 수 없다.

그림

11-6

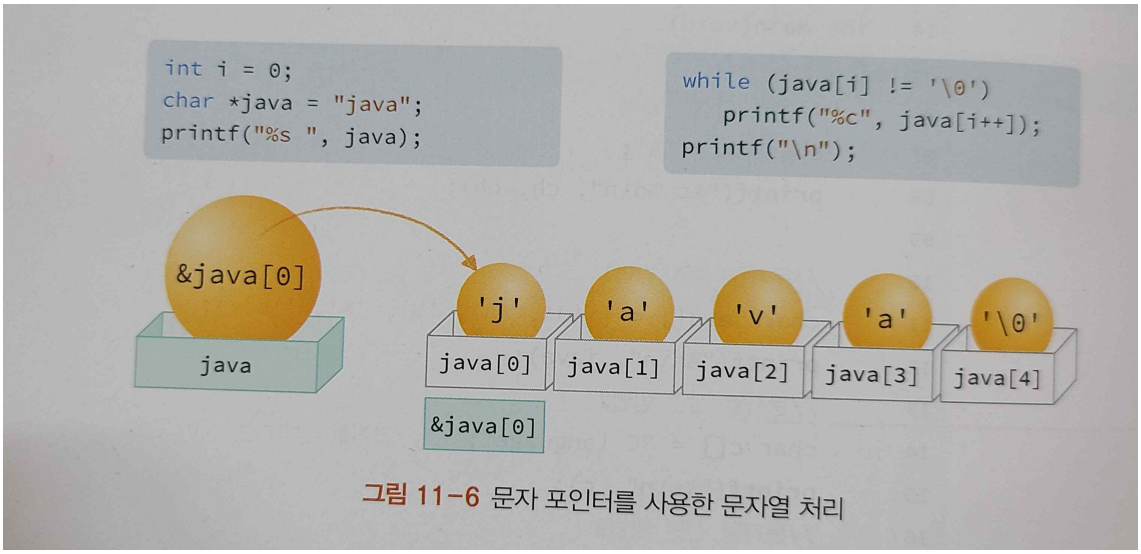


그림 11-6 문자 포인터를 사용한 문자열 처리

문자열을 구성하는 하나하나의 문자를 배열형식으로 직접 참조하여 출력하는 방식도 사용할 수 있는데 변수 java를 사용하여 문자를 수정하거나 수정될 수 있는 함수의 인자로 사용하면 실행 오류가 발생한다.

*출력할 문자열 끝을 '/0'문자로 검사하면 편리하다. 즉 반복문을 이용하여 문자가 '/0'이 아니면 문자를 출력하도록 하는 방식

*그러나 변수 java와 같이 문자열 상수를 저장하는 문자 포인터는 사용상 주의가 필요하고 변수 java가 가리키는 문자열은 상수이므로 수정할 수 없다.

='/0'문자에 의한 문자열 분리

함수 printf()에서 %s는 문자 포인터가 가리키는 위치에서 NULL 문자까지를 하나의 문자열로 인식한다. 그래서 문자열 사이에 한 문자를 '/0'으로 바꾸고 문자열을 출력하면 중간에 생긴 '/0'까지의 문자만 출력된다.

그림

11-7

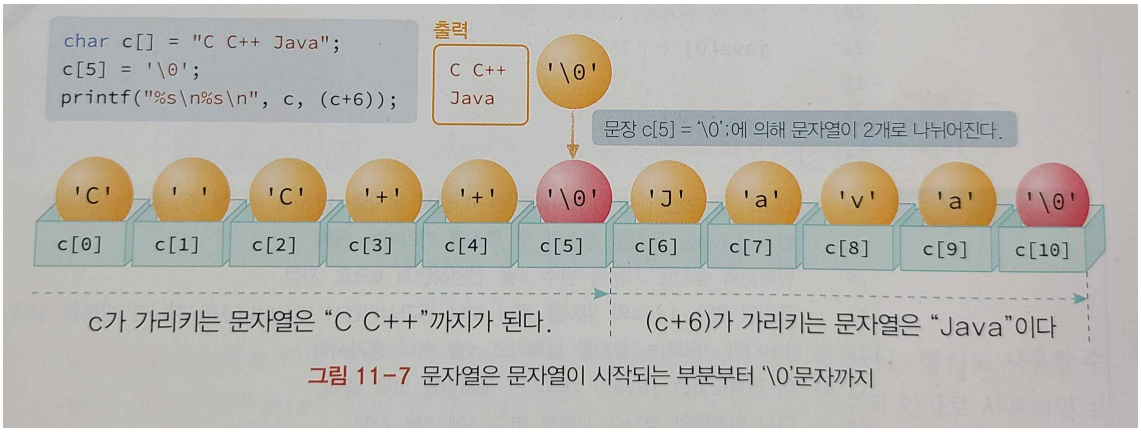


그림 11-7 문자열은 문자열이 시작되는 부분부터 '\0'문자까지

-다양한 문자 입출력

=버퍼처리 함수 getchar()

getchar() 와 pucher() 는 문자 입출력을 위한 함수로 getchar()는 문자의 입력에 putchar()는 문자의 출력에 사용된다.

*getchar()는 라인 버퍼링 방식을 사용하므로 문자하나를 입력한 후에 [enter]키를 눌러야 문자입력이 실행된다.

먼저 입력한 문자는 버퍼에 저장되어 있다가 [enter]키를 누르면 버퍼에서 프로그램으로 입력이 실행되는 방식인데 즉각적인 입력을 요구하는 시스템에는 맞지않다고 할 수 있다.

=함수 getche()

버퍼를 사용하지 않고 문자를 입력하는 함수인데 입력한 문자는 바로 모니터에 나타난다.

*함수를 이용하려면 헤더파일 conio.h를 삽입 해야한다.

=함수 getch()

위와 같이 버퍼를 사용하지 않고 바로 입력되는 함수인데 따로 출력함수를 쓰지 않는다면 화면에 보이지 않고 실행된다. 즉 echo 가 없다.

*함수를 이용하려면 헤더파일 conio.h를 삽입 해야한다.

그림 11-10

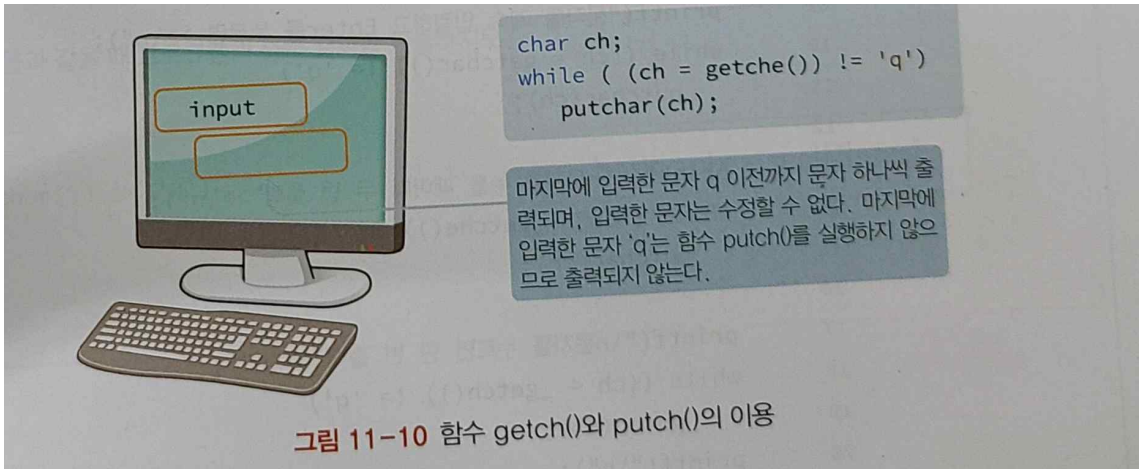


그림 11-10 함수 getch()와 putch()의 이용

=표를 통한 정리

함수	scanf("%c", &ch)	getchar()	getche() _getche()	getch() _getch()
헤더파일	stdio.h		conio.h	
버퍼 이용	버퍼 이용		버퍼 이용안함	
반응	[enter]키를 눌러야 작동		문자 입력마다 반응	
입력문자 표시(echo)	누르면 바로 표시		누르면 바로 표시	표시 안됨
입력문자 수정	가능		불가능	

-문자열입력

=문자배열 변수로 scanf()에서 입력

문자열을 입력받는 방법이다. scanf()는 공백으로 구분되는 하나의 문자열을 입력받을 수 있다.

*먼저 입력받은 문자열이 저장될 충분한 공간인 문자 배열 str을 선언한다.

*함수 scanf("%s", str)에서 형식제어문자 %s를 사용하여 문자열을 입력 받을 수 있다.

*함수 printf("%s", str)에서 %s를 사용하여 문자열을 출력한다.

문자열 입력은 충분한 공간의 문자배열이 있어야 가능한데 단순히 문자 포인터로는 문자열 저장이 불가능하다.

=gets()와 puts()

gets()는 한 행의 문자열 입력에 유용한 함수고 puts()는 한 행의 문자열을 출력하는 함수다.

*Visual C++의 함수 gets_s()는 현재 함수 gets()의 대체 함수로 사용을 권장한다.

gets(), puts(), gets_s()를 사용하려면 헤더파일 stdio.h를 삽입해야한다.

그림

11-11

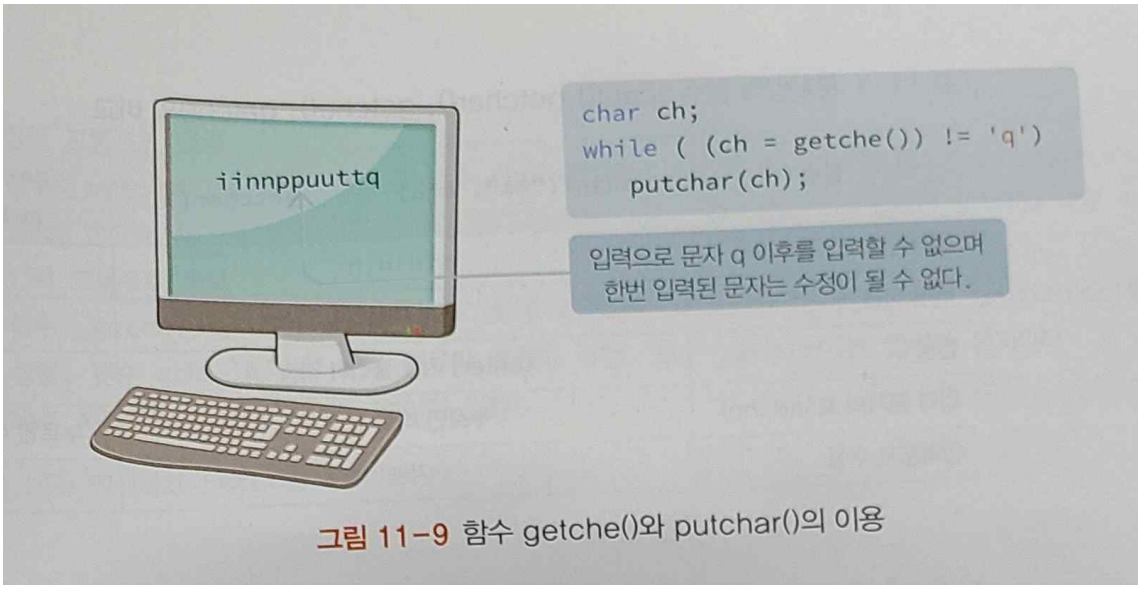


그림 11-9 함수 getche()와 putchar()의 이용

함수 gets() 는 [enter]키를 누를 깨까지 한 행을 버퍼에 저장 한 후 입력처리 하는데 주의할 점은 gets() 마지막에 입력된 '/n'가 '/0'fh 교체되어 인자인 배열에 저장된다. 그러므로 프로그램에서 한 행을 하나의 문자열로 간주하고 프로그래밍 할 수 있도록 한다.

그림 11-12

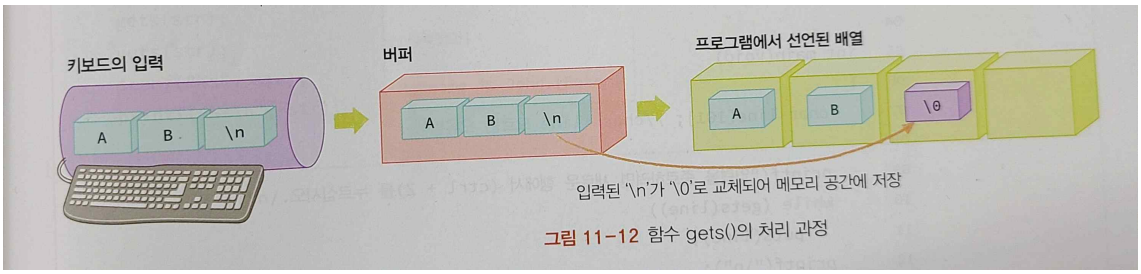


그림 11-12 함수 gets()의 처리 과정

gets()는 문자 배열을 인자로 받는데 배열의 크기가 입력한 문자열 보다 작을 경우 오류가 생긴다. puts() 출력함수로 문자열을 한 줄에 출력하는데 유용하게 사용될 수 있다. puts()는 오류가 발생하면 EOF를 반환한다.

*EOF(End Of File)는 파일의 끝이라는 의미로 stdio.h 헤더파일에 정수 1로 정의되어 있다.

함수 puts()는 gets()와 반대로 저장된 문자열의 마지막의 '/0'을 '/n'로 교체해 버퍼에 전송하고 버퍼에서 받은 내용을 모니터에 출력하기 때문에 문자열의 한 행 출력은 함수 puts()가 효과적이다.

2. 11-2 문자열 관련 함수

-문자배열 라이브러리와 문자열 비교

=다양한 문자열 라이브러리 함수

문자열 비교와 복사, 그리고 문자열 연결 등과 같은 다양한 문자열 처리는 헤더파일 string.h에 함수원형으로 선언된 라이브러리 함수로 제공된다. 함수에서 사용되는 자료형 size_t는 비부호

정수형(unsigned int type)이며, void*는 아직 정해지지 않는 다양한 포인터를 의미한다.
*문자의 배열 관련 함수는 헤더파일 string.h에 함수원형이 정의되어 있다.

=표를 통한 정리

함수원형	설명
void *memchr(const void *str, int c, size_t n)	메모리에서 str에서 n 바이트까지 문자 c를 찾아 그 위치를 반환
int memcmp(const void *str1, const void *str2, size_t n)	메모리 str1과 str2를 첫 n 바이트를 비교 검색하여 같으면 0, 다르면 음수 또는 양수 반환
void *memcpy(void *dest, const void *src, size_t n)	포인터 src 위치에서 dest에 n 바이트를 복사한 후 dest 위치 반환
void *memmove(void *dest, const void *src, size_t n)	포인터 src 위치에서 dest에 n 바이트를 복사한 후 dest 위치 반환
void *memset(void *src, int c, size_t n)	포인터 src 위치에서부터 n 바이트까지 문자 c를 지정한 후 src 위치 반환
size_t strlen(const char *str)	포인터 src 위치에서부터 널 문자를 제외한 문자열의 길이 반환

=함수 strcmp()
문자열 비교와 복사, 문자열 연결 등과 같은 다양한 문자열 처리는 헤더파일 string.h에 함수원형으로 선언된 라이브러리 함수로 제공된다. 문자열 관련 함수는 대부분 strxxx()로 명명된다. 대표적인 문자열 처리 함수인 strcmp()와 strncmp()는 두 문자열을 비교하는 함수다.

strcmp()는 인자인 두 문자열을 사전 상의 순서로 비교하는 함수다. strncmp()는 두 문자를 비교할 문자의 최대 수를 지정하는 함수이다.
*비교방법은 인자인 두 문자열을 구성하는 각 문자를 처음부터 비교해 나간다.
*비교 기준은 아스키 코드값으로 두 문자가 같다면 계속 다음 문자를 비교하여 문자가 다를때까지 계속 비교한다.
*결국 문자가 다른 경우 앞 문자가 작으면 음수, 뒤 문자가 작으면 양수, 같으면 0을 반환한다.
*대문자가 소문자보다 아스키 고드값이 작으므로 strcmp("java","javA")는 양수를 반환한다.
*마지막 문자 '/0'은 아스키 코드값이 0이므로 다른 어느 문자보다 작다.
*strcmp("ab","a")는 마지막에 문자 b와 '/0'를 비교하므로 양수를 반환한다.
그림 11-15

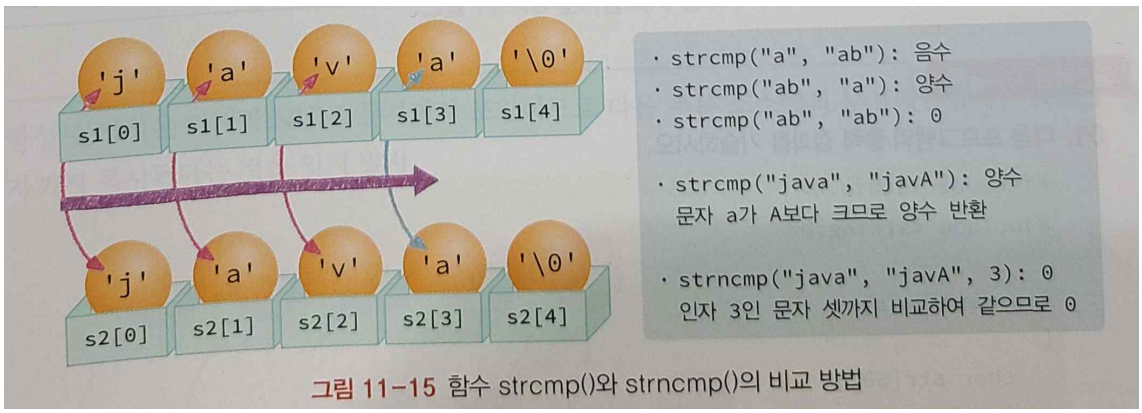


그림 11-15 함수 strcmp()와 strncmp()의 비교 방법

-문자열 복사와 연결
=함수 strcpy()

strcpy() 와 strncpy()는 문자열을 복사하는 함수인데 앞 인자에 뒤 인자 문자열을 복사한다. strncpy()는 복사되는 최대 문자 수를 마지막 인자 maxn으로 지정한다.

함상 문자열의 마지막 NULL 문자꼬지 포함하므로 다음 부분 소스에서 문자배열 d에 NULL 문자까지 복사된다.

그림 11-17

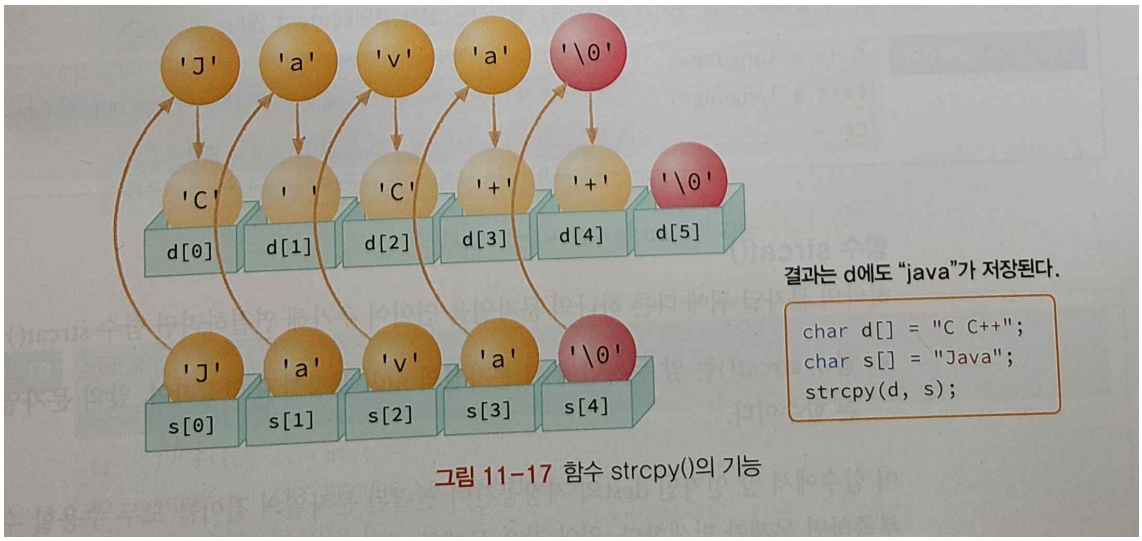


그림 11-17 함수 strcpy()의 기능

=함수 strcat()

하나의 문자열 뒤에 다른 하나의 문자열을 이을때는 strcat()를 사용한다.

*strcat()는 앞 문자열에 뒤 문자열의 null 문자까지 연결하여, 앞 문자열 주소를 반환한다.

단 앞 인자인 문자열의 저장공간이 연결된 문자열을 모두 수용할 공간보다 부족하면 문제가 생긴다. 위와 같은 문제를 예방하기 위한 함수가 strncat() 이다. 이 함수는 마지막 인자로 연결되는 문자의 수를 지정하고 그 이상 연결되지 않도록 한다. 여기서 정하는 문자 수는 널문자를 제외한 수이다.

그림 11-19

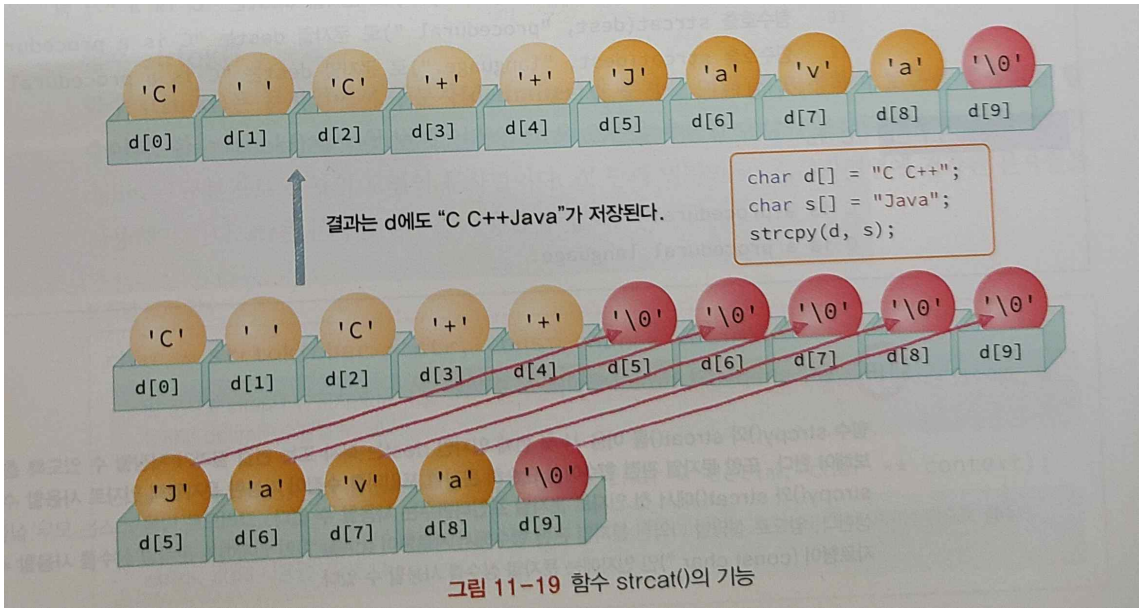


그림 11-19 함수 strcat()의 기능

-문자열 분리 및 다양한 문자열 관련 함수

=함수 strtok()

strtok()는 문자열에서 구분자인 문자를 지정해 토큰을 추출하는 함수다. 첫 번째 인자는 토큰을 추출할 문자열이고 두 번째 인자는 구분자로 문자의 모임인 문자열이다. 첫 번째 인자는 문자배열에 저장된 문자열을 사용해야 하고 문자열 상수는 사용할 수 없다.

그림

11-20

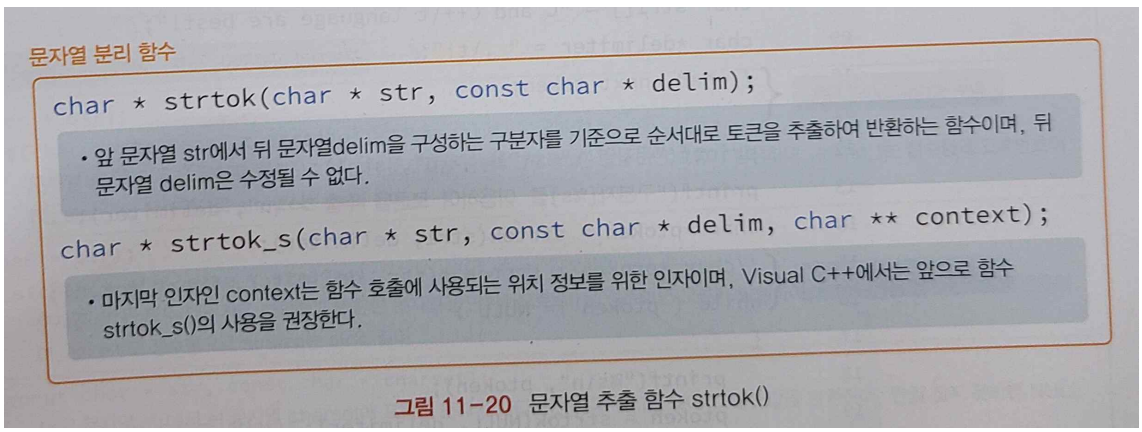


그림 11-20 문자열 추출 함수 strtok()

그림

11-21

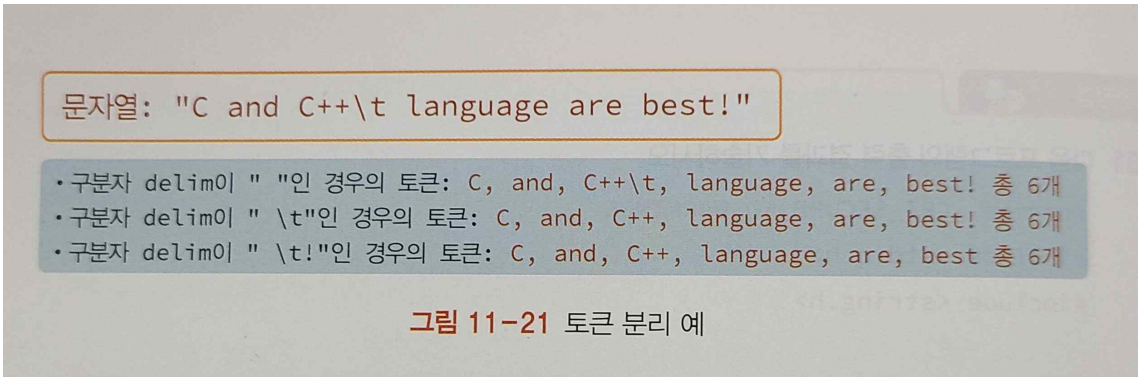


그림 11-21 토큰 분리 예

=문자열의 길이와 위치 검색

strlen()은 NULL 문자를 제외한 문자열 길이를 반환하는 함수다. strlwr()는 인자를 모두 소문자로 변환하여 반환한다. strupr()는 모두 대소문자로 변환하여 반환한다.

=표를 통한 정리

표

11-2

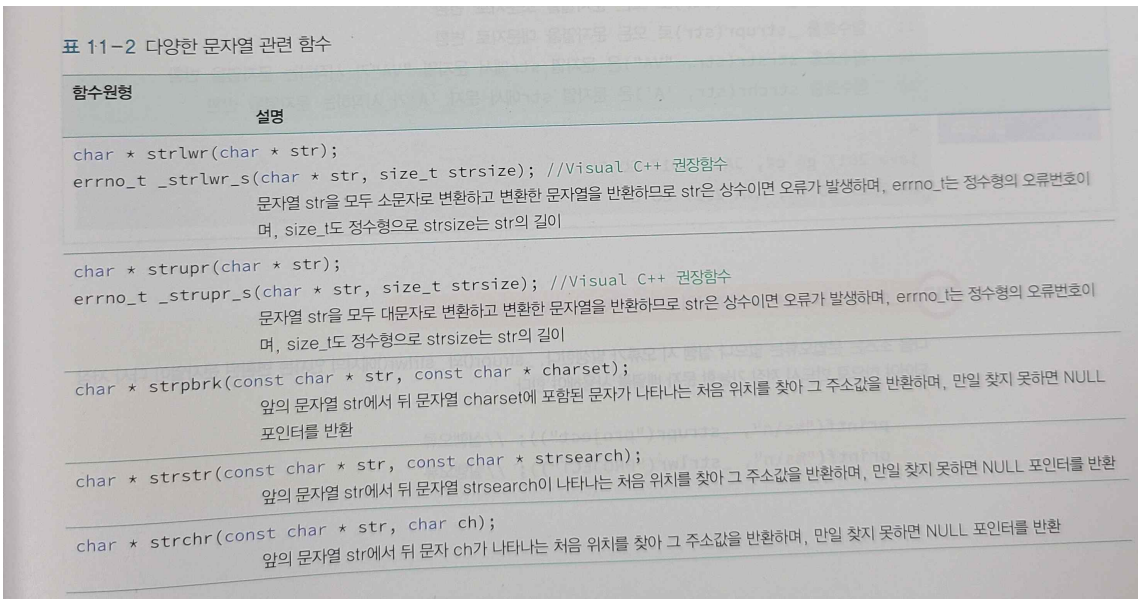


표 11-2 다양한 문자열 관련 함수

함수원형	설명
`char * strlwr(char * str);` `errno_t _strlwr_s(char * str, size_t strsize); //Visual C++ 권장함수`	문자열 str을 모두 소문자로 변환하고 변환한 문자열을 반환하므로 str은 상수이면 오류가 발생하며, errno_t는 정수형의 오류번호이며, size_t도 정수형으로 strsize는 str의 길이
`char * strupr(char * str);` `errno_t _strupr_s(char * str, size_t strsize); //Visual C++ 권장함수`	문자열 str을 모두 대문자로 변환하고 변환한 문자열을 반환하므로 str은 상수이면 오류가 발생하며, errno_t는 정수형의 오류번호이며, size_t도 정수형으로 strsize는 str의 길이
`char * strpbrk(const char * str, const char * charset);`	앞의 문자열 str에서 뒤 문자열 charset에 포함된 문자가 나타나는 처음 위치를 찾아 그 주소값을 반환하며, 만일 찾지 못하면 NULL 포인터를 반환
`char * strstr(const char * str, const char * strsearch);`	앞의 문자열 str에서 뒤 문자열 strsearch이 나타나는 처음 위치를 찾아 그 주소값을 반환하며, 만일 찾지 못하면 NULL 포인터를 반환
`char * strchr(const char * str, char ch);`	앞의 문자열 str에서 뒤 문자 ch가 나타나는 처음 위치를 찾아 그 주소값을 반환하며, 만일 찾지 못하면 NULL 포인터를 반환

3. 11-3 여러 문자열 처리

-문자 포인터 배열과 이차원 문자 배열

=문자 포인터 배열

여러 개의 문자열을 처리하는 하나의 방법은 문자 포인터 배열을 이용하는 방법이다. 하나의 문자 포인터가 하나의 문자열을 참조할 수 있으므로 문자 포인터 배열은 여러 개의 문자열을 참조할 수 있다. 다음 포인터 배열 pa의 선언 문장에 의한 메모리 구조를 살펴보면 다음과 같다.

그림

11-22

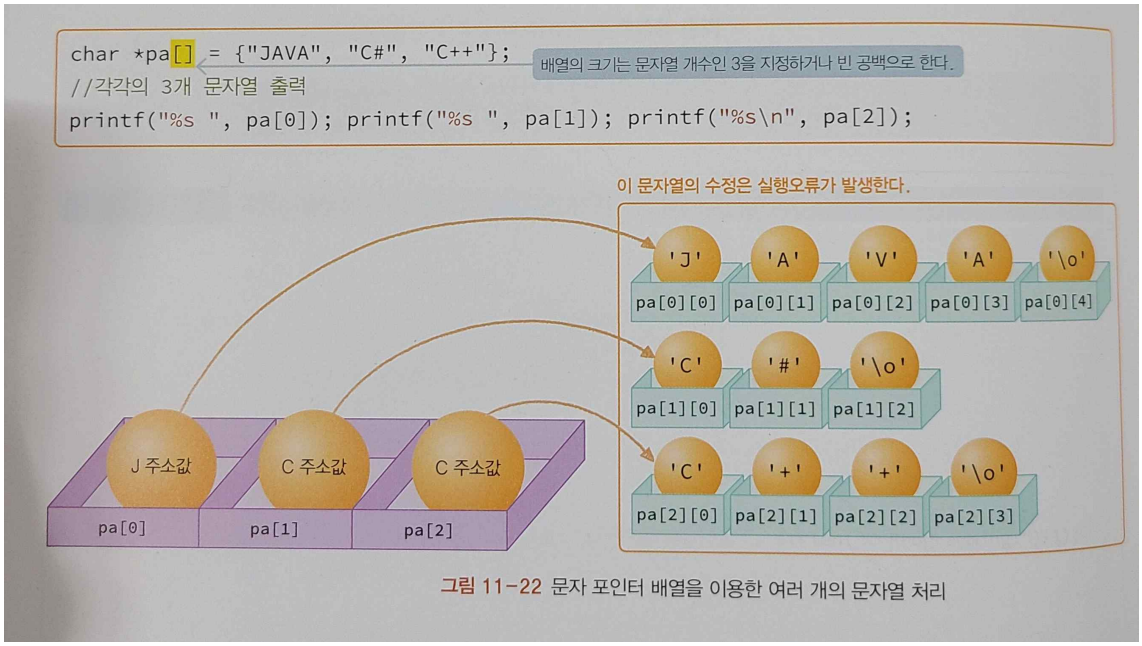


그림 11-22 문자 포인터 배열을 이용한 여러 개의 문자열 처리

문자 포인터 배열 이용 방법은 각각의 문자열 저장을 위한 최적의 공간을 사용하는 장점을 갖는다. 그러나 문자 포인터를 사용해서는 문자열 상수의 수정은 불가능하다. 문자 포인터 배열 pa를 이용하여 각 문자열을 출력하려면 pa[i]로 형식제어문자 %s를 이용한다.

=이차원 문자 배열

여러 개의 문자열을 처리하는 다른 방법은 문자의 이차원 배열을 이용하는 방법이다. 배열 선언에서 이차원 배열의 열 크기는 문자열 중에서 가장 긴 문자열의 길이보다 1 크게 지정해야 한다.

그림

11-23

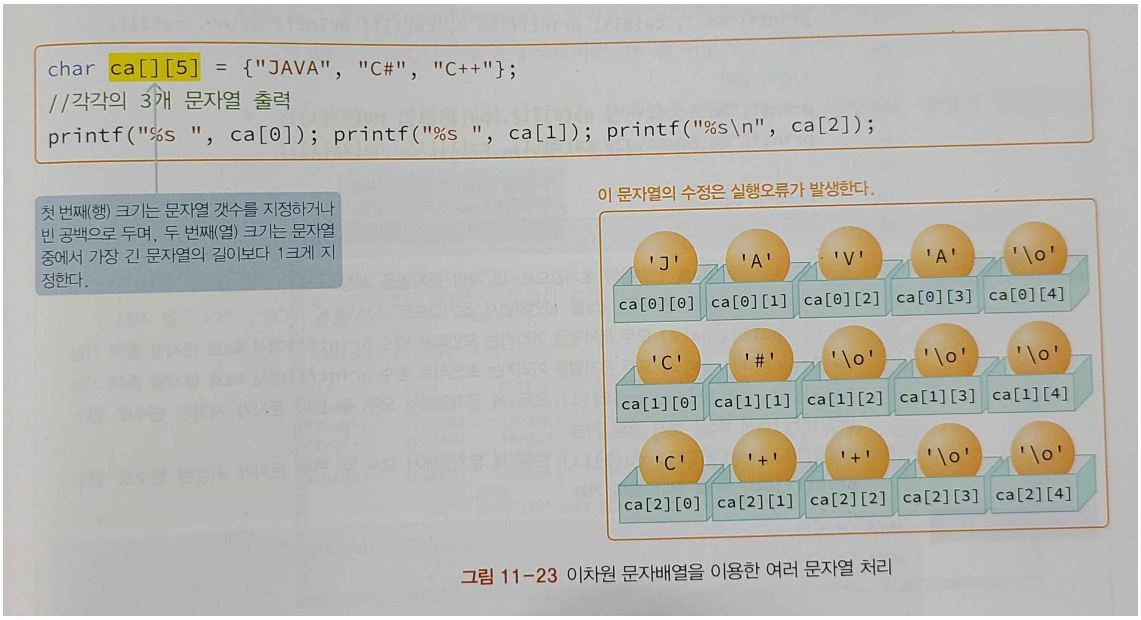


그림 11-23 이차원 문자배열을 이용한 여러 문자열 처리

이러한 배열처리는 문자열의 길이가 서로 다른 경우 남는 공간에 '/0'으로 채워져 공간을 낭비 할 수 있지만 문자열을 수정할 수 있다. ca[0][2] = 'v';와 같이 문자 수정이 가능하다는 소리이다. 각 문자열을 출력할때는 ca[i]로 형식제어문자 %s를 이용한다.

-명령행 인자

=main(int argc, char *argv[])

다음 도스 창을 살펴보면 도스 명령어로 "dir /w"를 사용한다. 만일 우리가 프로그램 dir를 개발한다면 옵션에 해당하는 "/w"를 어떻게 인식할까?

*다음과 같이 명령행에서 입력하는 문자열을 프로그램으로 전달하는 방법이 명령행 인자를 사용하는 방법이다.

그림

11-24

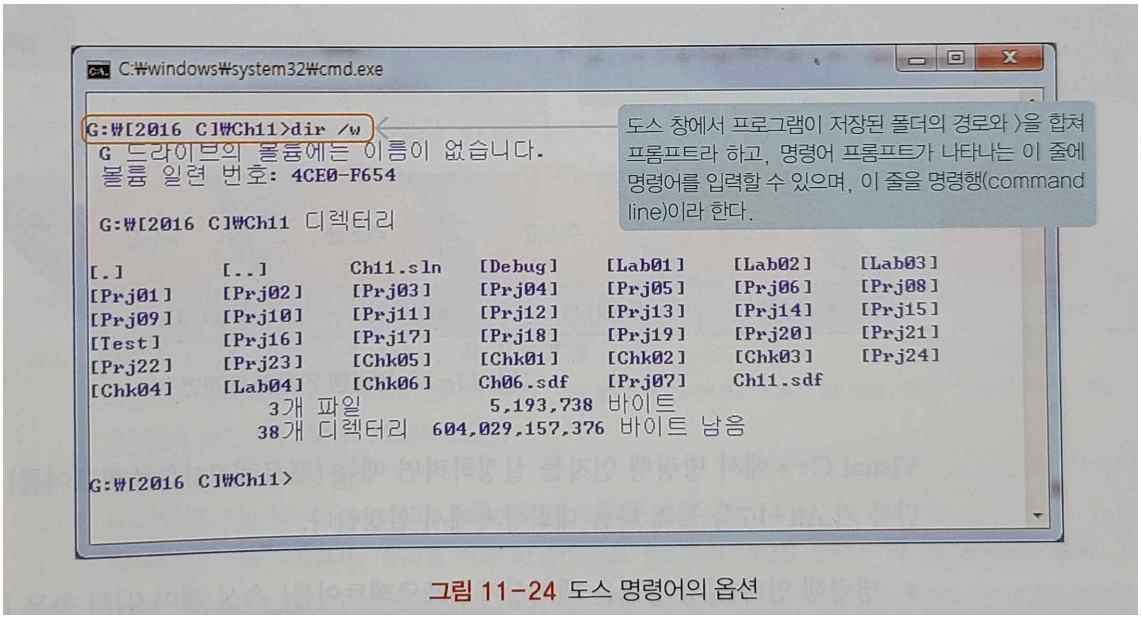


그림 11-24 도스 명령어의 옵션

프로그램에서 명령행 인자는 main() 함수의 인자로 기술된다. 지금까지는 명령행 인자를 이요하지 않고 main()의 인자를 void로 기술하였다.

*프로그램에서 명령행 인자를 받으려면 main() 함수에서 두 개의 인자 argc와 argv를 (int argc, char * argv[])로 기술해야 한다.

*매개변수 argc는 명령행에서 입력한 문자열의 수이며 argv[]는 명령행에서 입력한 문자열을 전달받는 문자 포인터 배열이다.

*주의할 점은 실행 프로그램 이름도 하나의 명령행 인자에 포함된다는 사실이다.

다음은 명령행에서 실행파일의 이름이 commandarg이고 옵션으로 C# c++ Java로, 프로그램을 실행한 결과를 보이고 있다. 이와 같은 명령행 인자로 프로그램을 실행하면 다음과 같은 구조의 문자열이 전달된다.

그림
11-25

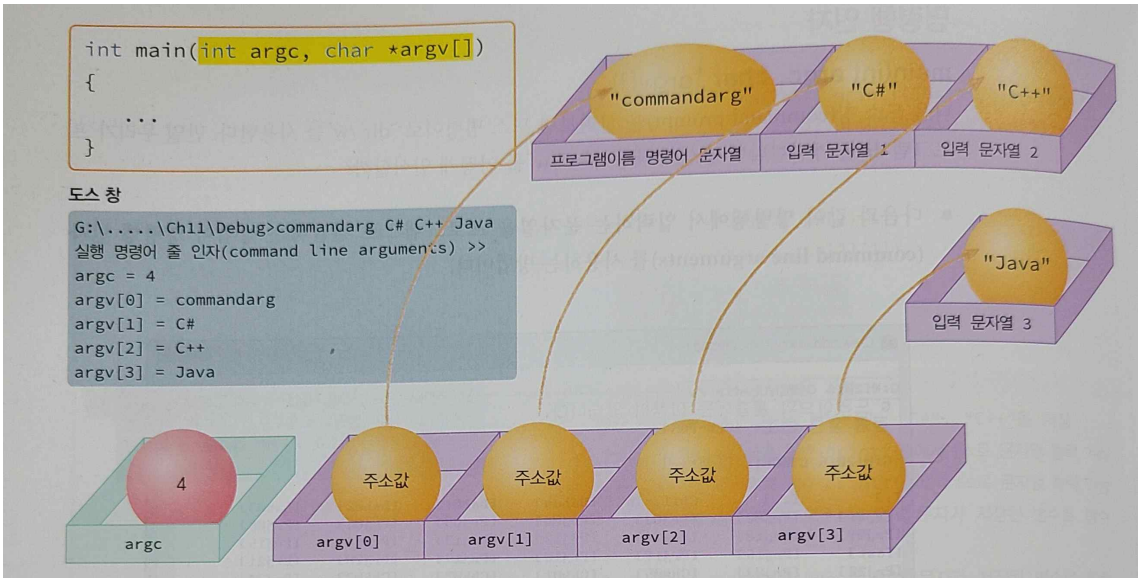


그림 11-25 명령행 인자와 매개변수

Visual C++에서 명령행 인자를 설정하려면 메뉴 [프로젝트/{프로젝트이름} 속성...]를 누르거나 단축키 Alt+F7을 눌러 다음 대화상자에서 설정한다.
*명령행 인자 설정 방법은 대화상자 [{프로젝트이름} 속성 페이지]의 항목 [디버깅]을 누르고 중간의 [명령인수]의 입력상자에 인자를 기술한다.
*이 입력 상자에는 실행파일 이름 뒤의 옵션만을 기술하면 된다.
그림 11-26

그림 11-27

12장. 변수 유효범위

1. 12-1 전역변수와 지역변수

-변수 범위와 지역변수
=변수 scope
변수의 참조가 유효한 범위를 변수의 유효 범위라 하는데 변수의 유효 범위는 크게 지역 유효 범위 와 전역 유효 범위로 나눌 수 있다.
*지역 유효 범위는 함수 또는 블록 내부에서 선언되어 그 지역에서 변수의 참조가 가능한 범위이다.
*전역 유효 범위는 2가지로 나뉜다. 하나는 하나의 파일에서만 변수의 참조가 가능한 범위다. 다른 하나는 프로젝트를 구성하는 모든 파일에서 변수의 참조가 가능한 범위다.
그림

12-1

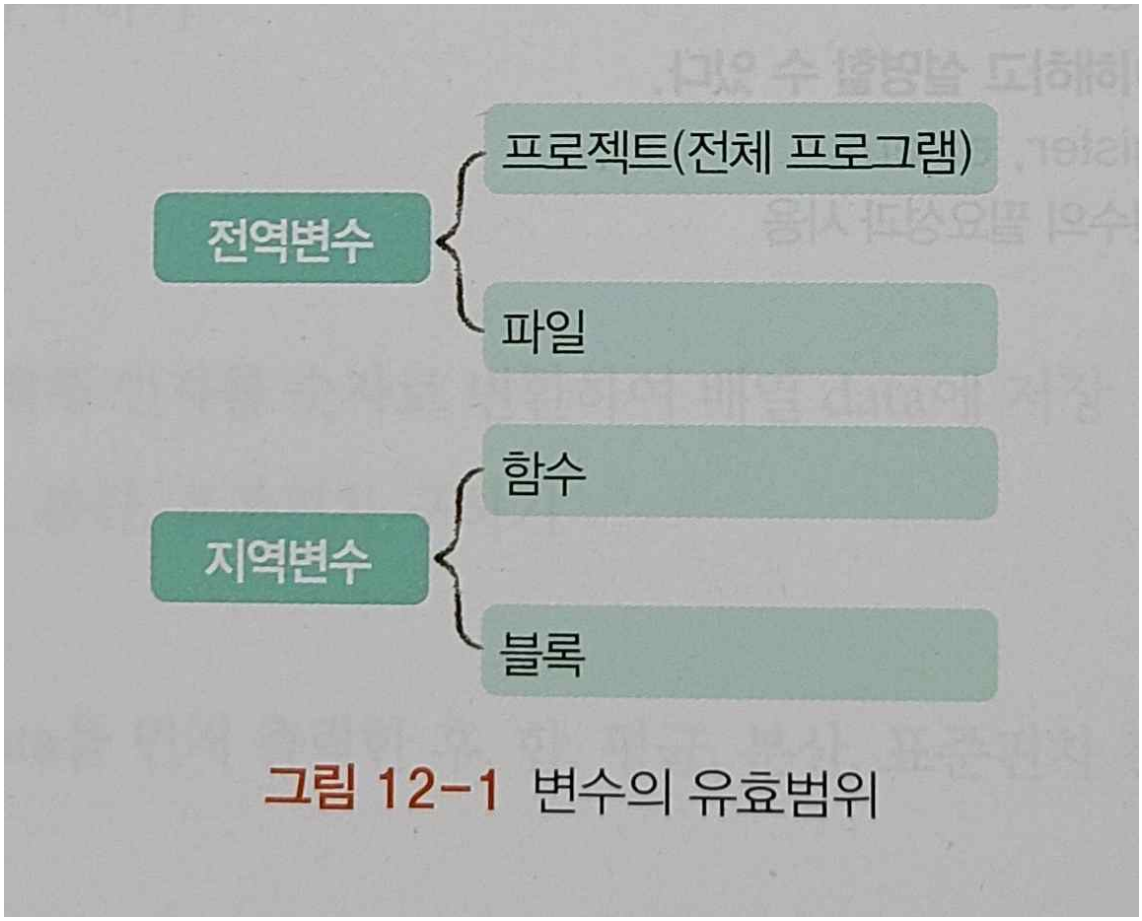


그림 12-1 변수의 유효범위

한의 프로젝트는 여러파일로 구성될 수 있다. 다음은 파일 main.c 그리고 sub1.c 와 sub2.c로 구성되는 프로그램이다.

*파일 main의 상단에 선언된 global은 프로젝트 전체 파일에서 사용될 수 있는 전역변수이다.

*파일 main의 함수 asub()에서 선언된 local은 rm 함수 내부에서만 사용할 수 있는 지역변수다.

*파일 sub2의 상단에 선언된 staticvar은 파일 sub2에서 사용될 수 있는 전역 변수이다.

*파일 sub1에서 파일 main의 상단에 선언된 global을 사용하려면 extern int global로 선언이 필요하다.

=지역변수

지역변수는 함수 또는 블록에서 선언된 변수이다. 지역변수는 내부변수 또는 자동변수라고도 부른다.

*함수나 블록에서 지역변수는 선언 문장 이후에 함수나 블록의 내부에서만 사용이 가능하다. 다른 함수나 블록에서는 사용될 수 없다.

*함수의 매개변수도 함수 전체에서 사용 가능한 지역변수와 같다.

*지역변수는 선언 후 초기화하지 않으면 쓰레기값이 저장되므로 주의해야 한다.

*지역변수는 그 변수가 선언된 함수 또는 블록에서 선언 문장이 실행되는 시점에서 메모리에 할당된다.
지역변수가 할당되는 메모리 영역을 스택(stack)이라 한다. 그리고 지역변수는 선언된 부분에서 자동으로 생성되고 함수나 블록이 종료되는 순간 메모리에서 자동으로 제거된다. 이러한 이유에서 지역변수는 자동변수라 한다. 지역변수 선언에서 자료형 앞에 키워드 auto가 사용될 수 있다. auto 생략 가능해서 없는 경우가 대부분이다.
그림
12-4

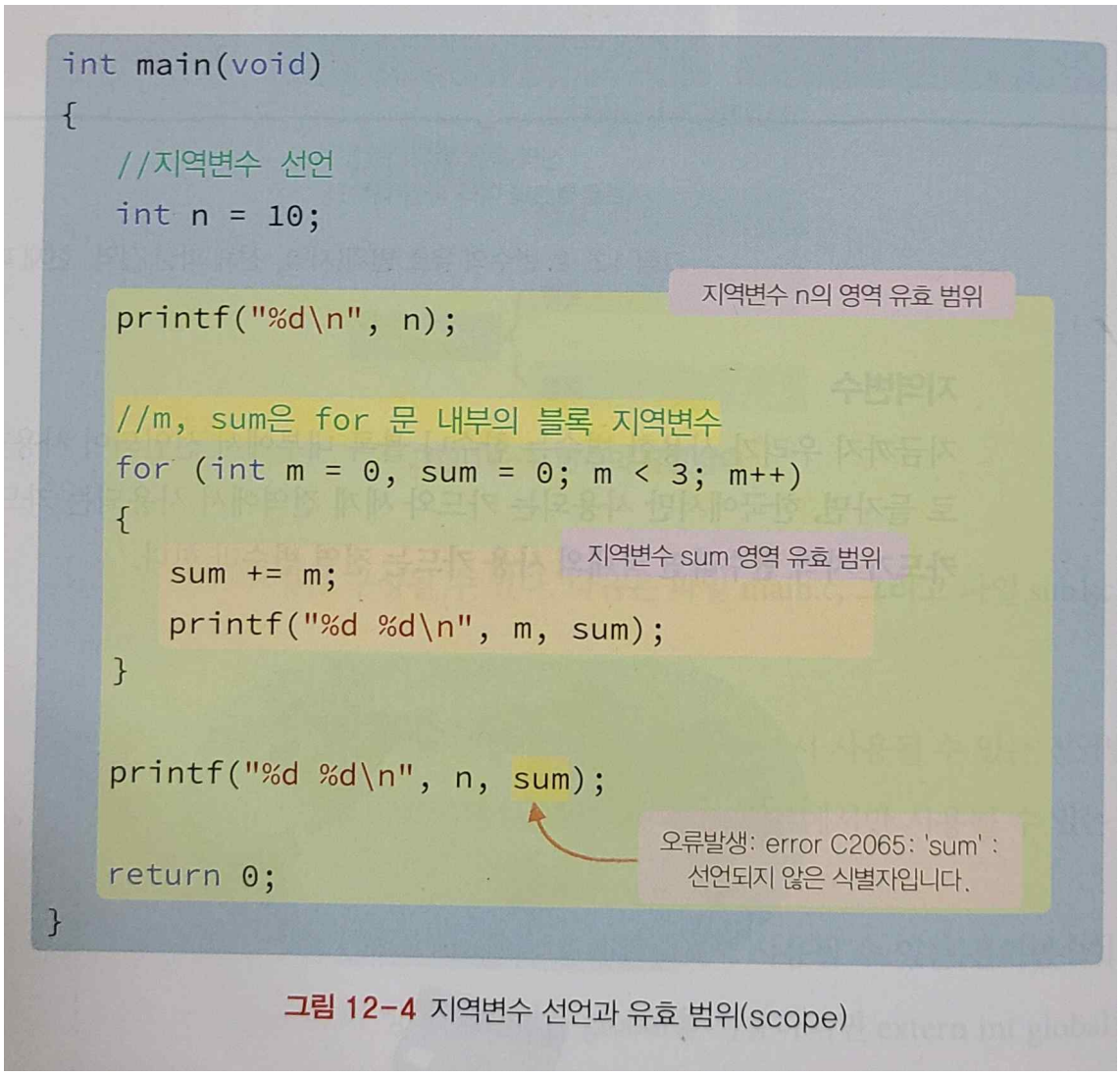


그림 12-4 지역변수 선언과 유효 범위(scope)

다음 예제 프로그램에서 for문 블록에서 선언된 지역변수 sum은 for 문 블록에서만 사용이 가능하고 sum은 블록 외부에서 참조가 불가능하다. 함수 sub(int param)에서와 같이 매개변수 param은 지역변수와 같이 사용될 수 있다.

-전역 변수와 extern

=전역변수

전역변수는 함수 외부에서 선언되는 변수이다. 전역변수는 외부변수라고도 부른다. 전역변수는 일반적으로 프로젝트의 모든 함수나 블록에서 참조할 수 있다.

*전역변수는 선언되면 자동으로 초기값이 자료형에 맞는 0으로 지정된다. 즉 정수형은 0, 문자형은 null 문자인 '/0', 실수형은 0.0, 포인터 형은 NULL값이 저장된다.

*함수나 블록에서 전역변수와 같은 이름으로 지역변수를 선언할 수있다. 이런 경우, 함수 내부나 블록에서 그 이름을 참조하면 지역변수로 인식한다. 그러므로 지역변수와 동일한 이름의 전역변수는 참조할 수 없게 된다. 가능한 이러한 변수는 사용하지 않도록 한다.

*전역변수는 프로젝트의 다른 파일에서도 참조가 가능하다. 단 다른 파일에서 선언된 전역변수를 참조하려면 키워드 extern을 사용하여 이미 다른 파일에서 선언된 전역변수임을 선언해야한다.

extern을 사용한 참조선언 구문은 변수선언 문장 맨 앞에 extern을 넣는 구조이다. extern 참조선언 구문에서 자료형은 생략할 수 있다. 키워드 extern을 사용한 변수 선언은 새로운 변수를 선언하는 것이 아니며, 단지 이미 존재하는 전역변수의 유효 범위를 확장하는 것이다.

=전역변수 장단점

동일한 파일에서도 extern을 사용해야 하는 경우가 발생할 수 있다. 전역변수의 선언 위치가 변수를 참조하려는 위치보다 뒤에 있는 경우, 전역변수를 사용하기 위해서는 extern을 사용한 참조선언이 필요하다. 그러나 다음과 같이 소스 파일 중간이나 하단에 전역변수를 배치하는 방법은 바람직하지 않다.

그림

12-6

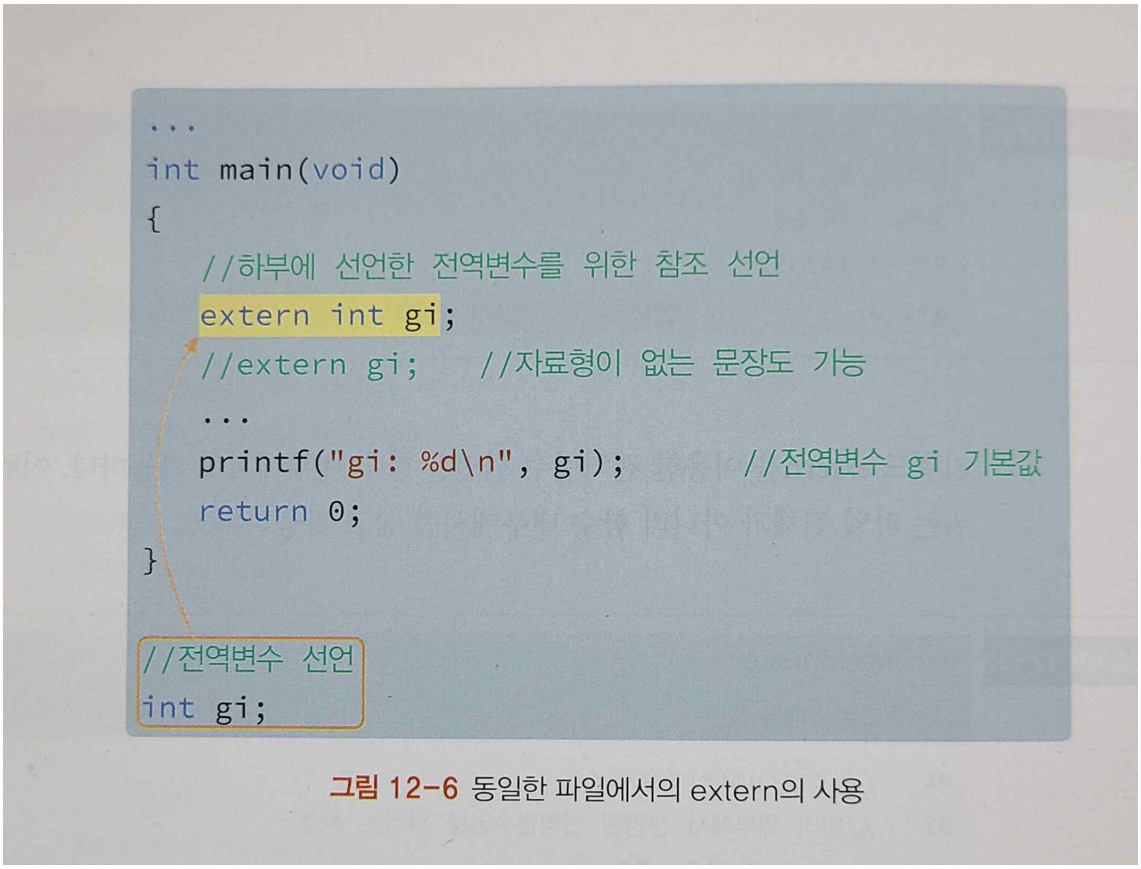


그림 12-6 동일한 파일에서의 extern의 사용

전역변수는 어디에서든지 수정할 수 있으므로 사용이 편한 장점이 있는데 전역변수에 예상하지 못한 값이 저장된다면 프로그램 어느 부분에서 수정되었는지 알기 어려운 단점이 있기 때문에 전역변수는 가능한 제한적으로 사용하는 것이 바람직하다.

2. 12-2 정적 변수와 레지스터 변수

-기억부류와 레지스터 변수

=auto, register, static, extern

변수 선언의 위치에 따라 전역, 지역변수로 나뉘는데 4가지 기억부류 auto, register, static, extern에 따라 할당되는 메모리 영역이 결정되고 메모리 할당과 제거시기가 결정된다.

*자동변수인 auto는 일반 지역변수로 생략될 수 있다.

그림

12-7

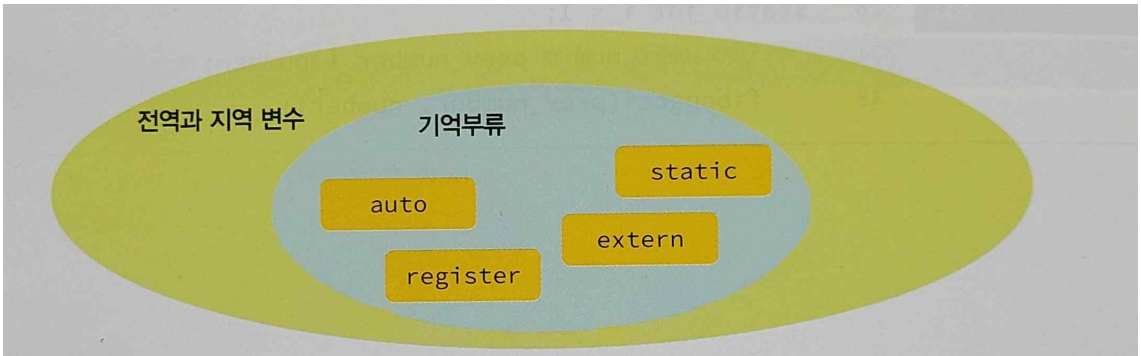


그림 12-7 전역과 지역, 다양한 기억 부류

extern은 컴파일러에게 변수가 이미 어딘가에 존재하고 이제 사용하겠다는 것을 알리는 구문에 사용되는 키워드인데 extern이 선언되는 위치에 따라 이 변수의 사용 범위는 전역 또는 지역으로 한정될 수 있다.

=표를 통한 정리

기억부류 종류	전역	지역
auto	x	o
register	x	o
static	o	o
extern	o	x

사용법은 변수선언 문장에서 자료형 앞에 하나의 키워드를 넣는 방식인데 extern을 제외하고 나머지 3개는 변수선언에서 초기값을 저장할 수 있다.

=키워드 register
일반적으로 변수는 메모리에 할당된다. 이 변수를 연산에 참여 시키려면 다시 CPU 내부에 레지스터에 불러들여 연산을 수행한다. 레지스터 변수는 저장공간이 일반 메모리가 아니라 CPU내부 레지스터에 할당된다.
그림 12-8밑에 그림

*CPU내부 레지스터에 저장되기 때문에 함수나 블록을 빠저나오면 소멸된다.
*일반 메모리보다 빠르게 참조된다.
*일반 메모리에 할당되는 변수가 아니기 때문에 주소연산자 &를 사용할 수 없다.
레지스터의 양이 제한되어 있기 때문에 모자라면 일반 지역변수로 저장된다. 주로 처리속도 증가를 위해 사용하고 반복문 횟수제어 제어변수에 이용하면 효과적이다.

-정적 변수
=키워드 static
자료형 앞에 static를 넣어서 선언하는데 정적변수는 정적 지역변수와 정적 전역변수로 나뉜다.

*초기 생성된 이후에 메모리에서 제거되지 않는다. 저장값을 유지하거나 수정할 수 있다.
*프로그램 시작시 메모리에 할당되고 종료시 메모리에서 제거된다.
*초기값을 지정하지 않으면 자동으로 0, '/0', NULL값이 저장된다.
*정적변수의 초기화는 단 한번만 수행되고 프로그램 중간에 더 이상 초기화 되지 않는다. 초기화는 상수로만 가능하다.

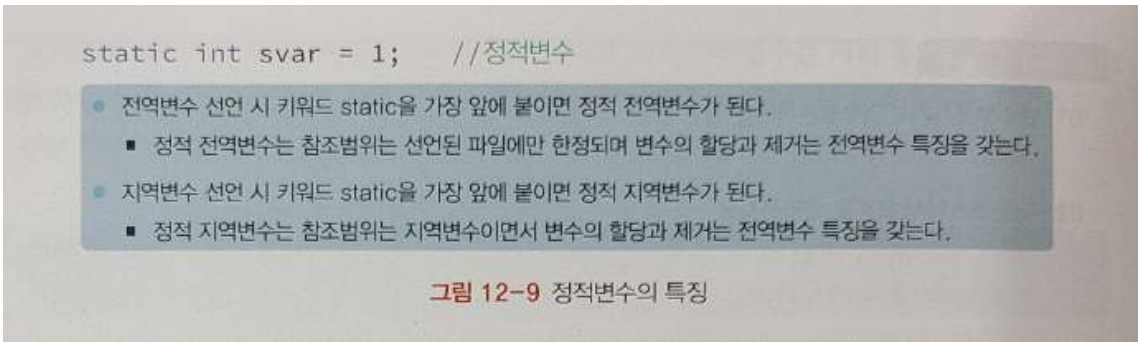


그림 12-9 정적변수의 특징

=정적 지역변수
함수나 블록에서 정적으로 선언되는 변수가 정적 지역변수이다. 정적 지역변수의 유효 범위는 선언된 블록 내부에서만 참조 가능하다. 정적 지역변수는 함수나 블록을 종료해도 메모리에서 제거되지 않고 계속메모리에 유지 관리되는 특성이 있다.
*변수 유효 범위(scope)는 지역변수와 같으나 할당된 저장공간은 프로그램이 종료되어야 메모리에서 제거되는 전역변수의 특징을 갖는다.
*즉 함수에서 이전에 호출되어 저장된 값을 유지하여 이번 호출에 사용될 수 있다.
다음 예제에서 정적 지역변수인 sindex는 함수가 종료되어도 메모리에 계속 변수값이 남아 있어 함수 increment()가 호출될 때마다 그 값이 1씩 증가한다. 그러므로 변수 sindex에는 함수 increment()가 호출된 횟수가 저장된다. 즉 정적 지역변수는 함수가 종료되더라도 그 값이 제거되지 않고 계속 남아 있어 그 다음의 함수 호출에서 이용될 수 있다. 그러나 자동 지역변수인 aindex는 함수 incremental 호출될 때마다 다시 새롭게 메모리에 할당되고, 함수가 종료되면 메모리에 서 제거 된다. 그러므로 aindex는 함상 1이 출력된다.

=정적 전역변수
함수 외부에서 정적으로 선언되는 변수가 정적 전역변수인데 일반 전역변수는 파일 소스가 다르더라도 extern을 사용하여 참조가 가능하다. 그러나 정적 전역변수는 선언된 파일 내부에서만 참조가 가능한 변수이다. 즉 정적 전역변수는 extern에 의해 다른 파일에서 참조가 불기능하다.

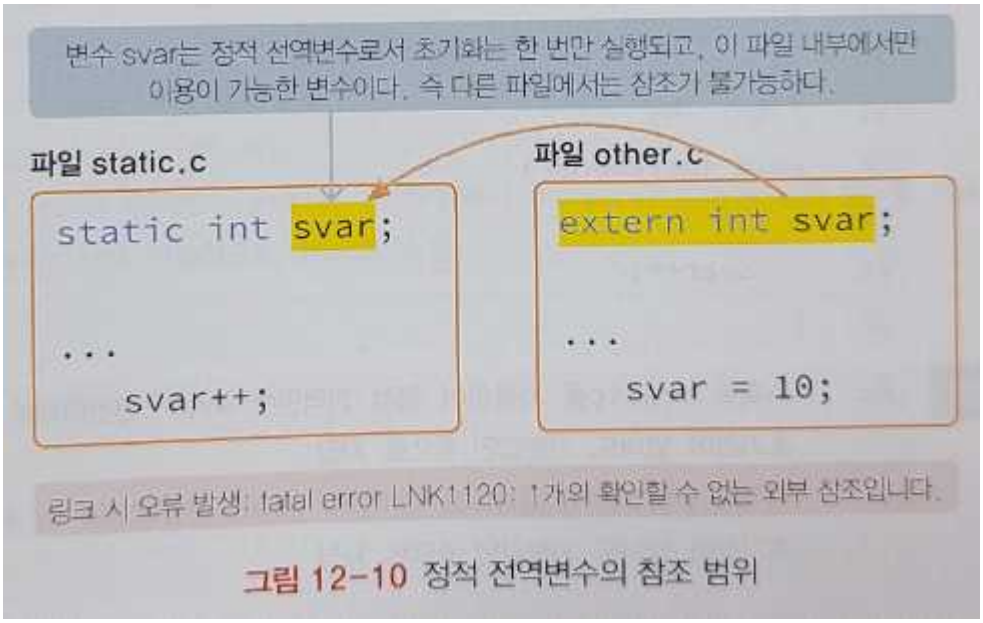


그림 12-10 정적 전역변수의 참조 범위

전역변수의 사용은 모든 함수에서 공유할 수 있는 저장공간을 이용할 수 있는 장점이 있으나 어느 한 함수에서 잘못 다루면 모든 함수에 영향을 미치는 단점도 있다. 특히 프로그램이 크고 복잡하면 전역변수의 사용은 원하지 않는 전역변수의 수정과 같은 부작용(side effect)의 위험성이 항상존재한다. 그러므로 가급적이면 전역변수의 사용을 자제하는 것이 좋으며, 부득이 전역변수를 이용하는 경우에는 파일에서만 전역 변수로 이용할 수 있는 정적 전역변수를 이용하는 것이 바람직하다

3. 12-3 메모리 영역과 변수 이용

-메모리영역

=데이터,스택,힙영역

메인 메모리의 영역은 프로그램 실행 과정에서 데이터(data)영역, 힙(heap)영역, 스택(stack)영역 세 부분으로 나뉜다. 이러한 메모리영역은 변수의 유효범위(sope)와 생존기(life time)에 결정적 역할을 하며, 변수는 기억부류(storage class)에 따라 할당되는 메모리 공간이 달라진다. 기억부류는 변수의 유효범위(scope) 와 생존기간(life time)을 결정한다.

기억부류는 변수의 저장공간의 위치가 데이터(data)영역, 힙(heap)영역, 스텍(stack)영역인지도 걸정하며, 초기값도 결정힌다.

그림 12-12 기억부류

*데이터 영역은 전역변수와 정적변수가 할당되는 저장공간이다.

*힙 영역은 동적 할당(dynamic allocation)되는 변수가 할당되는 저장공간이다.

*마지막으로 스택 영역은 함수 호출에 의한 형식 매개변수 그리고 함수 내부의 지역변수가 할당되는 저장공간이다.

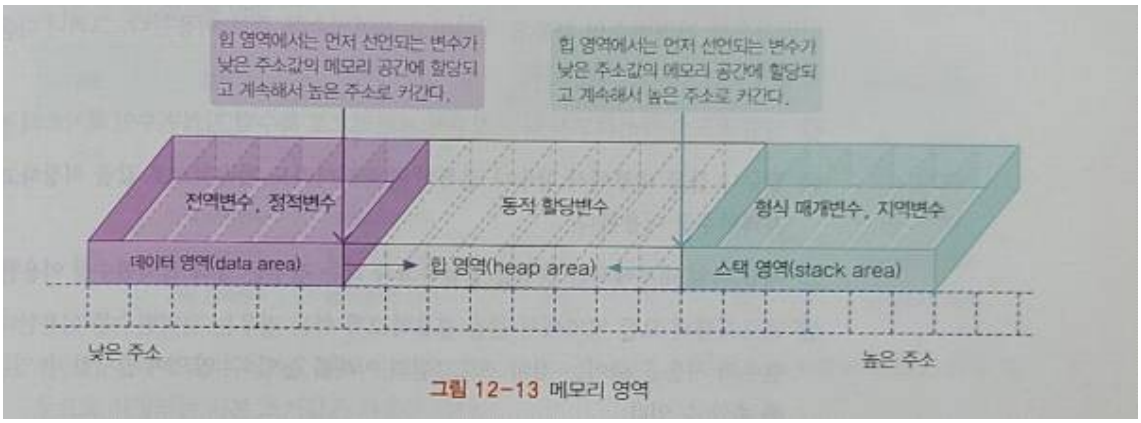


그림 12-13 메모리 영역

데이터영역은 메모리 주소가 낮은 값에서 높은 값으로 저장 장소가 할당된다. 데이터영역은 프로그램이 시작되는 시점에 정해진 크기대로 고정된 메모리 영역이 확보된다. 그러나 힙 영역과 스택 영역은 프로그램이 실행되면서 영역 크기가 계속적으로 변한다.

*즉 스택 영역은 메모리 주소가 높은 값에서 낮은 값으로 저장 장소가 할당된다. 그러므로 함수 호출과 종료에 따라 높은 주소에서 낮은 주소로 메모리가 할당되었다가 다시 제거되는 작업이 반복된다.

*힙 영역은 데이터 영역과 스택 영역 사이에 위치한다. 힙 영역은 메모리 주소가 낮은 값에서 높은 값으로 사용하지 않는 공간이 동적으로 할당된다.

-변수의 이용

=이용 기준

일반적으로 전역변수의 사용을 자제하고 지역변수를 주로 이용한다. 그러나 다음의 경우에는 그 특성에 맞는 변수를 이용한다.
1 실행 속도를 개선하고자 하는 경우에 제한적으로 특수한 지역변수인 레지스터 변수를 이용한다.
2 함수나 블록 내부에서 함수나 블록이 종료되더라도 계속적으로 값을 저장하고 싶을 때는 정적, 지역변수를 이용한다.
3 해당 파일 내부에서만 변수를 공유하고자 하는 경우는 정적 전역변수를 이용한다. 4 프로그램의 모든 영역에서 값을 공유하고자 하는 경우는 전역변수를 이용한다. 가능하면 전역
변수의 사용을 줄이는 것이 프로그램의 이해를 높일 수 있으며 발생할 수 있는 프로그램 문제를 줄일 수 있다. 변수가 정의되는 위치에 따라 전역변수와 지역변수를 정리하면 다음과 같다.

*변수가 할당되는 메모리 영역에 따라 변수의 할당과 제거의 시기가 결정된다. 즉 데이터 영역에 할당되는 전역변수와 정적 변수는 프로그램 시작 시 메모리가 할당되고, 프로그램 종료 시 메모리에서 제거된다.
*스택 영역과 레지스터에 할당되는 자동 지역변수와 레지스터 변수는 함수 또는 블록 시작 시 메모리가 할당되고, 함수 또는 블록 종료 시 메모리에서 제거된다.

표 12-2 변수의 종류

선언위치	상세 종류	키워드		유효범위	기억장소	생존기간
전역	전역 변수	참조선언	extern	프로그램 전역	메모리 (데이터 영역)	프로그램 실행 시간
	정적 전역변수	static		파일 내부		
지역	정적 지역변수	static		함수나 블록 내부		
	레지스터 변수	register			레지스터	함수 또는 블록 실행 시간
	자동 지역변수	auto (생략가능)			메모리 (스택 영역)	

변수의 종류에 따른 생존기간과 유효 범위만을 정리하면 다음과 같다.
*전역변수와 정적 변수는 모두 생존 기간이 프로그램 시작 시에 생성되어 프로그램 종료 시에 제거된다.
*다만 자동 지역변수와 레지스터 변수는 함수가 시작되는 시점에서 생성되어 함수가 종료되는 시점에서 제거된다.

다음 표에서 변수의 유효 범위는 O, X로 구분하여 그 지역에서 참조 가능하면 O, 아니면 X로 구분 한다.

표 12-3 변수의 유효 범위

구분	종류	메모리할당 시기	동일 파일 외부 함수에서의 이용	다른 파일 외부 함수에서의 이용	메모리해제 시기
전역	전역변수	프로그램시작	○	○	프로그램종료
	정적 전역변수	프로그램시작	○	×	프로그램종료
지역	정적 지역변수	프로그램시작	×	×	프로그램종료
	레지스터 변수	함수(블록) 시작	×	×	함수(블록) 종료
	자동 지역변수	함수(블록) 시작	×	×	함수(블록) 종료

변수의 종류에 따라 초기값 저장 문장의 실행 시점과 초기값이 명시적으로 지정되지 않을 경우 자동으로 지정되는 기본 초기값은 다음과 같다.

표 12-4 변수의 초기값

지역, 전역	종류	자동 저장되는 기본 초기값	초기값 저장
전역	전역변수	자료형에 따라 0이나 '\0' 또는 NULL 값이 저장됨.	프로그램 시작 시
	정적 전역변수		
지역	정적 지역변수		
	레지스터 변수	쓰레기값이 저장됨.	함수나 블록이 실행될 때마다
	자동 지역변수		

13장. 구조체와 공용체

1. 13-1 구조체와 공용체

-구조체 개념과 정의

=구조체 개념

추석이나 명절에 마트에서 다음과 같은 과일 또는 육류 등의 선물셋트를 흔히 볼 수 있다. 선물셋트는 인기가 있거나 관련 있는 상품들을 묶어 하나의 구성제품으로 판매하는 것이다. 이와 같이 지금까지 배운 정수나 문자, 실수나 포인터 그리고 이들의 배열 등을 묶어 하나의 자료형으로 이용하 는 것이 구조체이다.

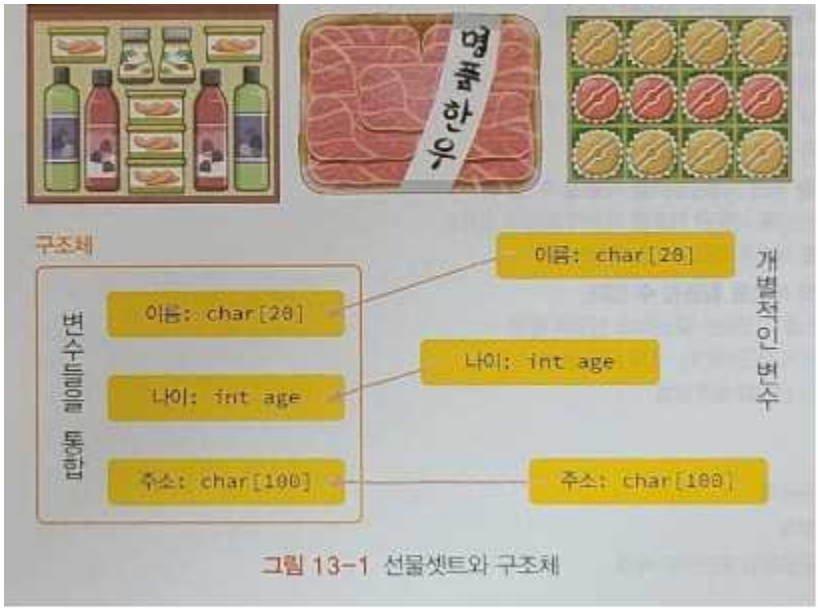


그림 13-1 선물셋트와 구조체

일상생활에서 서로 관련 있는 정보들을 하나로 묶어 처리하는 경우가 흔히 발생한다. 예를 들어 차 에 대한 정보, 계좌에 대한 정보, 책에 대한 정보 등을 들 수 있다. 또한 대학 관련 예를 들자면 학생 에 관한 정보, 교수에 관한 정보, 강좌에 관한 정보 등이다. 프로그램에서도 이와 같이 서로 다른 개 개의 자료 항목이 긴밀하게 연계된 하나의 단위로 관리되어야 하는 경우가 종종 발생한다. C 언어 는 이러한 요구사항을 구조체(struct)로 지원한다. 즉 연관성이 있는 서로 다른 개별적인 자료형의 변수들을 하나의 단위로 묶은 새로운 자료형을 구조체(structure)라 한다. 구조체는 연관된 멤버 로 구성되는 통합 자료형으로 대표적인 유도 자료형이다. 즉 기존 자료형으로 새로이 만들어진 자 료형을 유도 자료형(derived data types)'이라 한다.

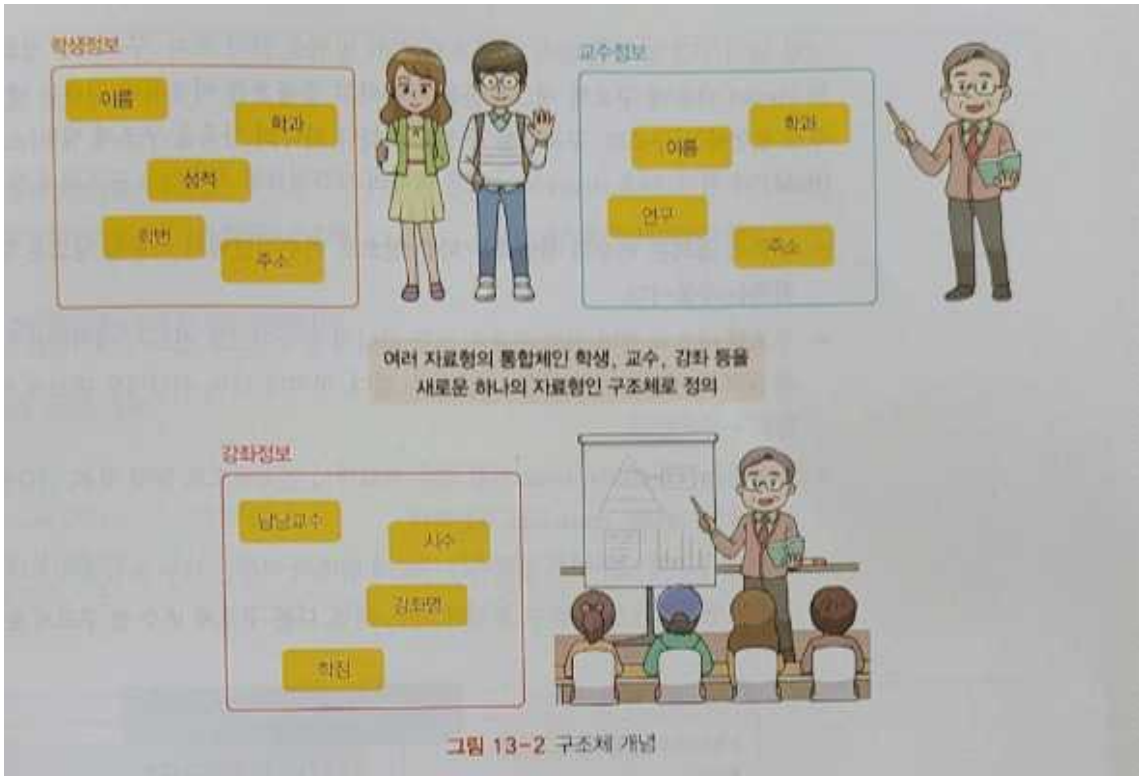


그림 13-2 구조체 개념

구조체 정의 자료형 int나 double과 같은 일반 자료형은 자료형을 바로 사용할 수 있으나 구조체를 자료형으로 사용하려면 먼저 구조체를 정의해야 한다. 와플이나 붕어빵을 만들려면 와플 기계나 붕어빵 기계가 필요하듯이 구조체를 사용하려면 먼저 구조체를 만들 구조체 틀(template)을 정의하여야 한다.

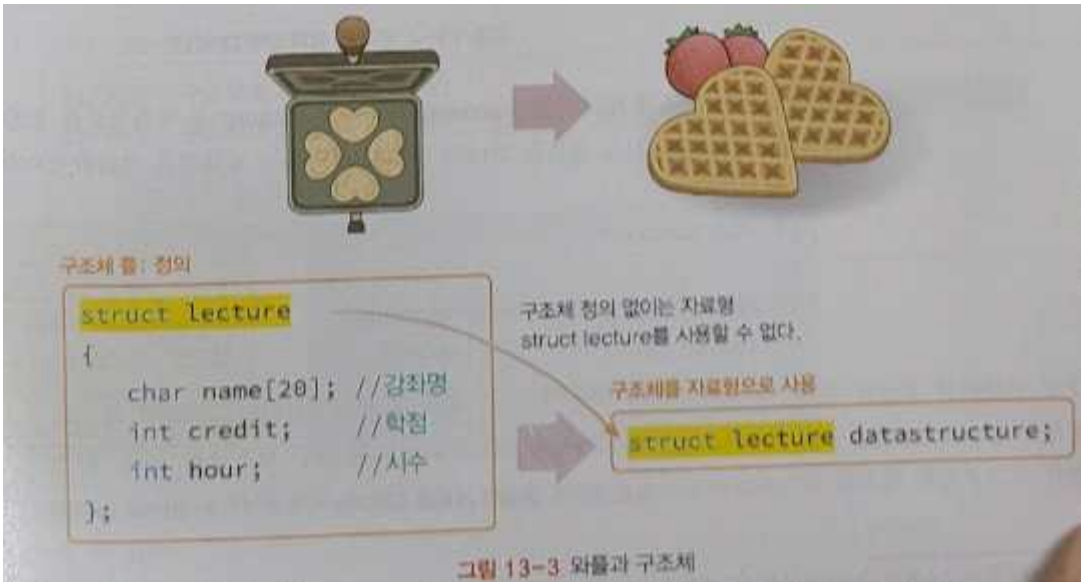


그림 13-3 와플과 구조체

그럼 먼저 구조체 틀을 만드는 구조체 정의 방법을 알아 보자. 구조체를 정의하는 방법은 키워 드 struct 다음에 구조체 태그이름을 기술하고 중괄호를 이용하여 원하는 멤버를 여러 개의 변 수로 선언하는 구조다. 구조체를 구성하는 하나 하나의 항목을 구조체 멤버(member) 또는 필드 (field)라 한다. 다음 struct lecture는 대학의 강좌정보를 처리하는 구조체의 한 예이다.

* 구조체 정의는 변수의 선언과는 다른 것으로 변수선언에서 이용될 새로운 구조체 자료형을 정의하는 구문이다.
*구조체 내부의 멤버 선언 구문은 모두 하나의 문장이므로 반드시 세미콜론으로 종료해야 하며, 각 구조체 멤버의 초기값을 대입할 수 없다. 마지막 멤버 선언에도 반드시 세미콜론이 빠지지 않도록 주의한다.
*멤버가 int credit; int hour;처럼 같은 자료형이 연속적으로 놓일 경우, 간단히 콤마 연산자를 사용해 int credit, hour;로도 가능하다.
*또한 한 구조체 내부에서 선언되는 구조체 멤버의 이름은 모두 유일해야 한다.
*구조체 멤버로는 일반 변수, 포인터 변수, 배열, 다른 구조체 변수 및 구조체 포인터도 허용된다.

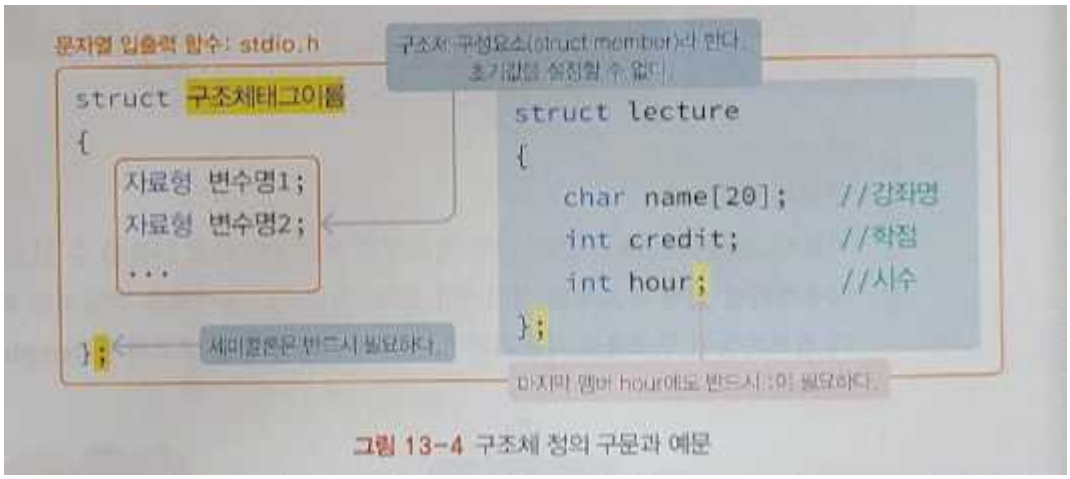


그림 13-4 구조체 정의 구문과 예문

다음 소스에서 구조체 태그이름이 account인 struct account 는 계좌정보를 표현하는 구조체로 계

좌주이름, 계좌번호, 잔고 정보를 하나의 단위로 처리하는 자료형을 정의한 것이다.

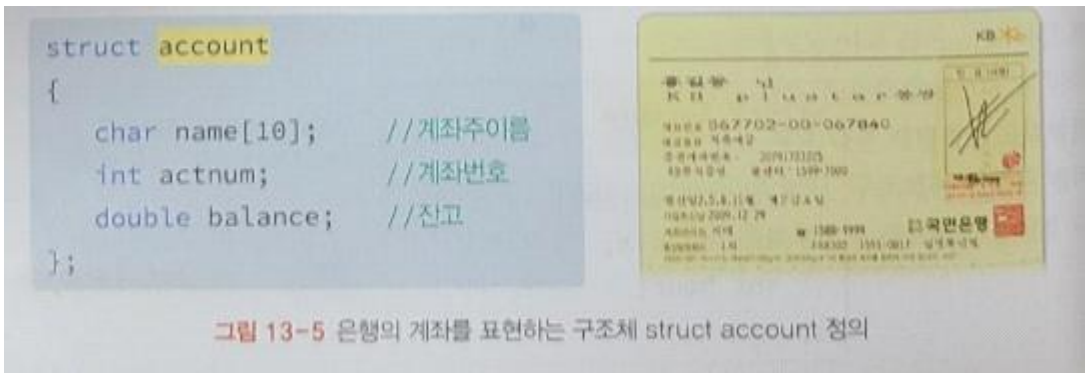


그림 13-5 은행의 계좌를 표현하는 구조체 struct account 정의

-구조체 변수 선언과 초기화

=구조체 변수 선언

구조체가 정의되었다면 이제 구조체형 변수 선언이 가능하다. 즉 구조체 struct account가 새로운 자료유형으로 사용될 수 있다. 새로운 자료형 struct account 형 변수 mine을 선언하려면 struct account mine;으로 선언한다. 기존의 일반 변수선언과 같이 동시에 여러 개의 변수선언도 가능하다.

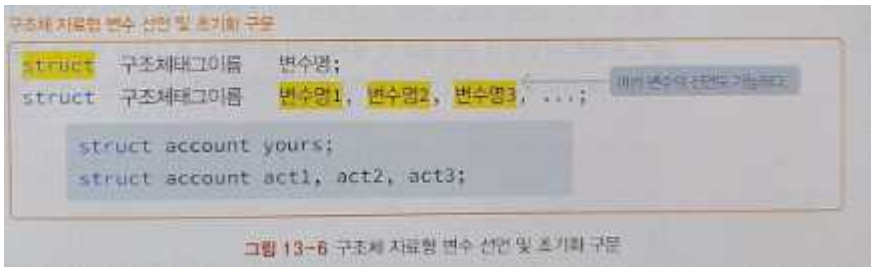


그림 13-6 구조체 자료형 변수 선언 및 초기화 구문

구조체 변수를 선언하는 다른 방법은 다음과 같이 구조체 정의와 변수 선언을 함께하는 방법 이다. 다음 문장으로 구조체 struct account를 정의하면서 동시에 변수 myaccount는 구조체 struct account 형의 변수로 선언된다. 이 문장 이후 struct account도 새로운 자료형으로 사용 될 수 있다.

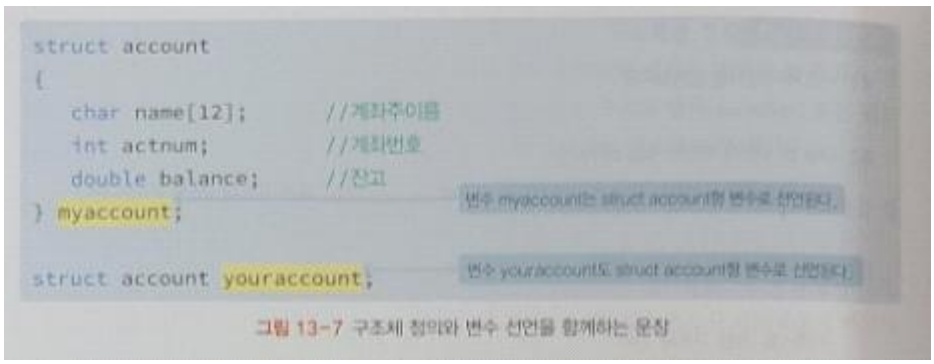


그림 13-7 구조체 정의와 변수 선언을 함께하는 문장

=구조체 변수의 초기화

배열과 같이 구조체 변수도 선언 시 중괄호를 이용한 초기화 지정이 가능하다. 초기화 값은 다음과 같이 중괄호 내부에서 구조체의 각 멤버 정의 순서대로 초기값을 쉼표로 구분하여 기술한다. 배열 과 같이 초기값에 기술되지 않은 멤버값은 자료형에 따라 기본값인 0, 0.0, '/0' 등으로 저장된다.

```
struct 구조체태그이름   변수명 = {초기값1, 초기값2, 초기값3, ...};

struct account
{
   char name[12];       //계좌주이름
   int actnum;          //계좌번호
   double balance;      //잔고
};

struct account mine = {"홍길동", 1001, 300000 };
```

그림 13-9 구조체 변수의 초기화

=구조체의 멤버 접근 연산자, 와 변수 크기

선언된 구조체형 변수는 접근연산자 를 사용하여 멤버를 참조할 수 있다. 즉 문장 yours. actnum=1002;은 변수 yours의 멤버 actnum에 1002를 저장하는 기능을 수행한다. 이와 같이 yours.name과 yours.balance로 구조체 멤버 name과 balance도 참조할 수 있다. 접근연산자는 는 참조연산자라고도 부른다.

```
구조체변수이름.멤버
mine.actnum = 1002;   mine.balance = 300000;
```

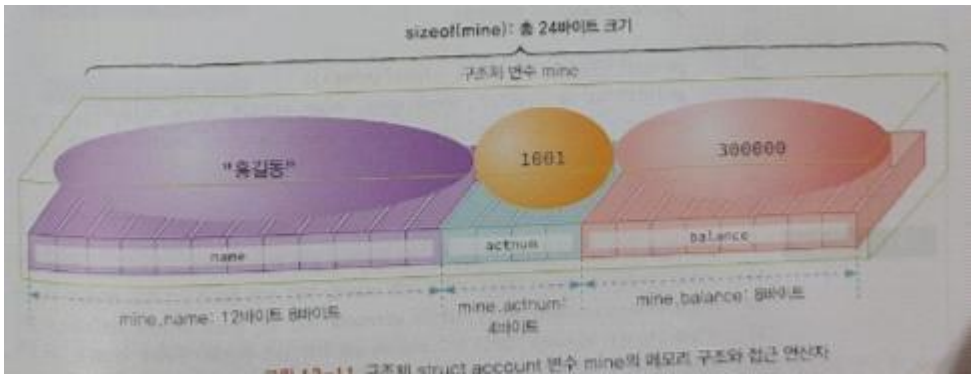
그림 13-10 구조체 멤버 접근 연산자

구조체 struct account 의 변수 mine은 다음과 같은 구조로 메모리에 할당된다. 그러므로 변수 mine 저장공간 크기는 name이 12바이트, actnum이 4바이트, balance가 8바이트, 모두 더하면 총 24바이트이다. 변수 mine의 크기는 sizeof(mine)로 알아 볼 수 있다. 그러나 우리가 계산하는 멤버 저장공간 크기의 합인 구조체의 크기는 실제 sizeof 결과보다 작거나 같다.

*일반적으로 컴파일러는 시스템의 효율성을 위하여 구조체 크기를 산술적인 구조체의 크기보다 크게 할당할 수 있다.

*시스템은 정보를 4바이트 혹은 8바이트 단위로 전송 처리하므로 이에 맞도록 메모리를 할당하다 보면 중간에 사용하지 않는 바이트를 삽입할 수 있다.

*그러므로 실제 구조체의 크기는 멤버의 크기의 합보다 크거나 같다.



-구조체 활용

=구조체 멤버로 사용되는 구조체

구조체 멤버로 이미 정의된 다른 구조체 형 변수와 자기 자신을 포함한 구조체 포인터 변수를 사용 할 수 있다. 다음 구조체 struct date는 년, 월, 일 정보를 저장할 수 있는 구조체이다.

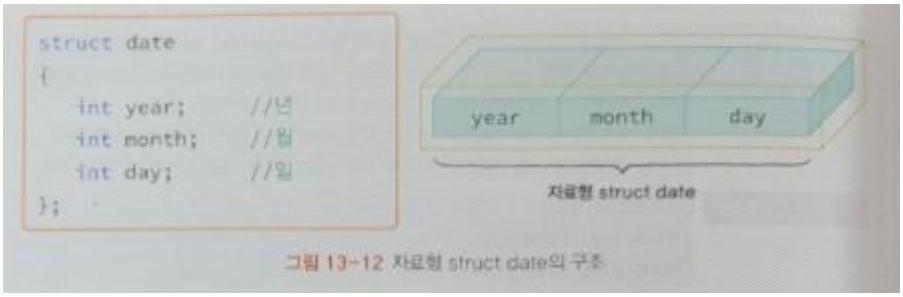


그림 13-12 자료형 struct date의 구조

구조체 struct account에 계좌 개설일자를 저장할 멤버로 open을 추가해보자. 이 open의 자료형 으로 위에서 정의한 struct date를 사용할 수 있을 것이다. 즉 새로운 구조체 account를 다음과 같 이 정의할 수 있다. 즉 구조체 account의 멤버로 구조체 struct date를 사용하고 있다. 그러므로 struct account 변수 me의 메모리 구조는 다음과 같다.

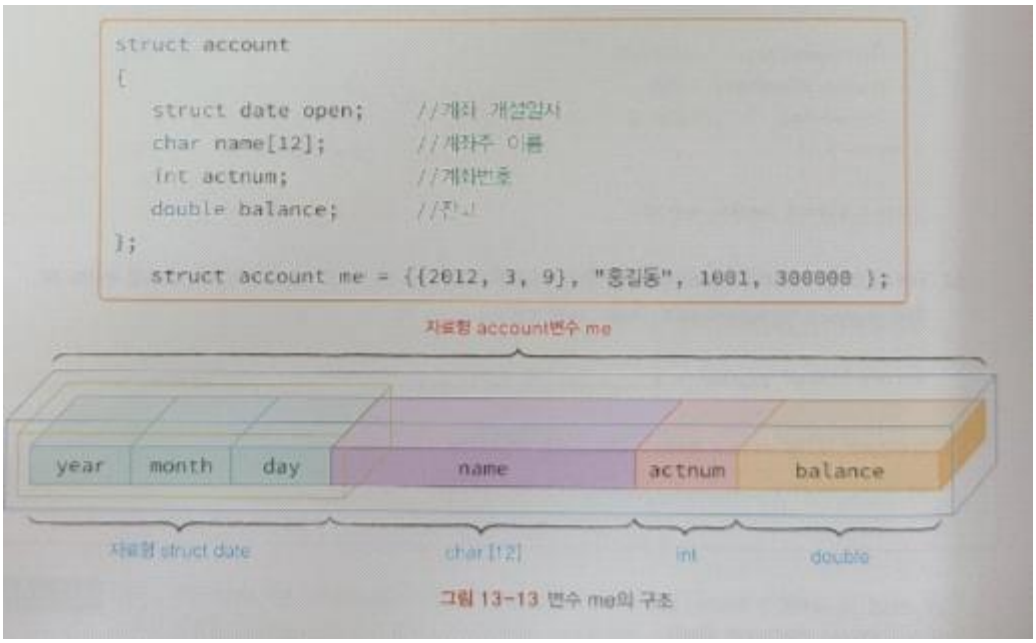


그림 13-13 변수 me의 구조

=구조체 변수의 대입과 동등비교

동일한 구조체형의 변수는 대입문이 가능하다. 즉 구조체 멤버마다 모두 대입할 필요 없이 변수 대 입으로 한번에 모든 맴버의 대입이 가능하다. 다음 소스와 같이 struct student 형의 변수 hong과 one 사이의 대입이 가능하다. 다음 소스와 같이 struct student 형의 변수 hong과 one 사이의 대입이 가능하다.

```
struct student
{
    int snum;    //학번
    char *dept;    //학과 이름
    char name[12];    //학생 이름
};
struct student hong = { 201800001, "컴퓨터정보공학과", "홍길동" };
struct student one;

one = hong;
```

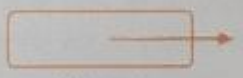
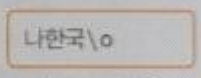
그림 13-15 구조체 변수의 대입

struct student 형의 변수 hong과 one에서 (one == bae)와 같은 동등 비교는 사용할 수 없다. 그러므로 만일 구조체를 비교하려면 구조체 멤버, 하나 하나를 비교해야 한다. 즉 struct student에서 학번을 비교하고 문자열인 이름과 학번은 문자열이므로 라이브러리 strcmp()를 사용하면 hong과 one의 내용을 모두 비교할 수 있다.

```
if ( one == bae ) //오류
  printf("내용이 같은 구조체입니다.\n");

if (one.snum == bae.snum)
  printf("학번이 %d로 동일합니다.\n", one.snum);

if (one.snum == bae.snum && !strcmp(one.name, bae.name) && !strcmp(one.dept, bae.dept))
  Printf("내용이 같은 구조체입니다.\n");
```

그림 13-16 구조체의 동등 비교

표 13-1 char 포인터와 char 배열의 비교

char 포인터	char 배열
char *dept; //학과 이름	char name[12]; //학생 이름
char *dept = "컴퓨터정보공학과";	char name[12] = "나한국";
 변수 dept	 변수 name[12]
변수 dept는 포인터로 단순히 문자열 상수를 다루는 경우 효과적	변수 name은 배열로 12바이트 공간을 가지며 문자열을 저장하고 수정 등이 필요한 경우 효과적
dept = "컴퓨터정보공학과";	name = "나한국"; //오류
단지 문자열 상수의 첫 주소를 저장하므로 문자열 자체를 저장하거나 수정하는 것은 불가능하므로 다음 구문은 사용 불가능	문자열 자체를 저장하는 배열이므로 문자열의 저장 및 수정이 가능하고 문자열 자체를 저장하는 다음 구문 사용도 가능이 가능
strcpy(dept, "컴퓨터정보공학과"); //오류	strcpy(name, "배성문");
scanf("%s", dept); //오류	scanf("%s", name);

-공용체 활용

=공용체 개념

하나의 차고에 일반 세단과 SUV를 각각 주차한다고 생각해보자. 주차 공간이 하나이므로 세단과 SUV를 동시에 주차시킬 수는 없으나 한 대의 주차는 가능하다. 이러한 겸용 주차장과 비슷한 개념 이 공용체이다. 동일한 저장 장소에 여러 자료형을 저장하는 방법으로, 공용체를 구성하는 멤버에 한번에 한 종류 만 저장하고 참조할 수 있다.

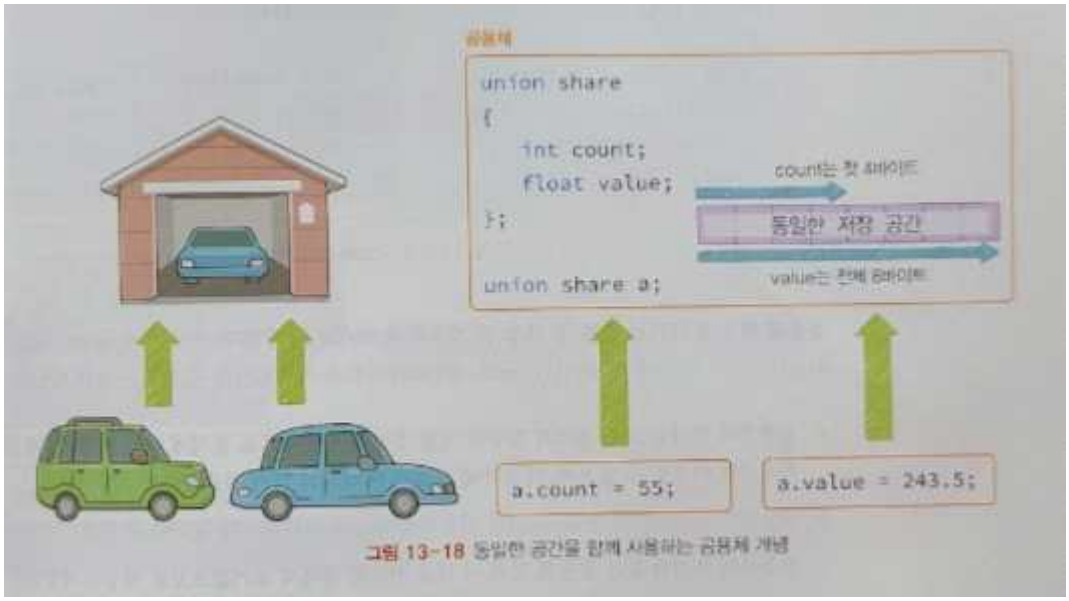


그림 13-18 동일한 공간을 함께 사용하는 공용체 개념

=union을 사용한 공용체 정의 및 변수 선언

공용체(union)는 서로 다른 자료형의 값을 동일한 저장공간에 저장하는 자료형이다. 공용체 선 언 방법은 union을 struct로 사용하는 것을 제외하면 구조체선언 방법과 동일하다. 다음은 공용체 union data를 정의하는 구문이다. 물론 구조체와 같이 공용체를 정의하면서 바로 변수를 선언할 수 있다.

=공용체 멤버 접근

공용체 변수로 멤버를 접근하기 위해서는 구조체와 같이 접근연산자 를 사용한다. 즉 문장 data2. ch = 'A';은 공용체 변수 data2의 멤버 ch에 문자 'A'를 저장하는 구문이다. 이 문장 이후에 멤버 cnt나 real의 출력은 가능하나 의미는 없다. I 유형이 char인 ch를 접근하면 8바이트 중에서 첫 1바이트만 참조하며, int인 ont를 접근하면 전체 공간의 첫 4바이트만 참조하고, double인 real을 접근하면 8바이트 공간을 모두 참조한다. 공용체 멤버의 참조는 가능하나 항상 마지막에 저장한 멤버로 접근해야 원하는 값을 얻을 수 있다. 그러므로 공용체를 참조할 경우 정확한 멤버를 사용하는 것은 프로그래머의 책임이다.

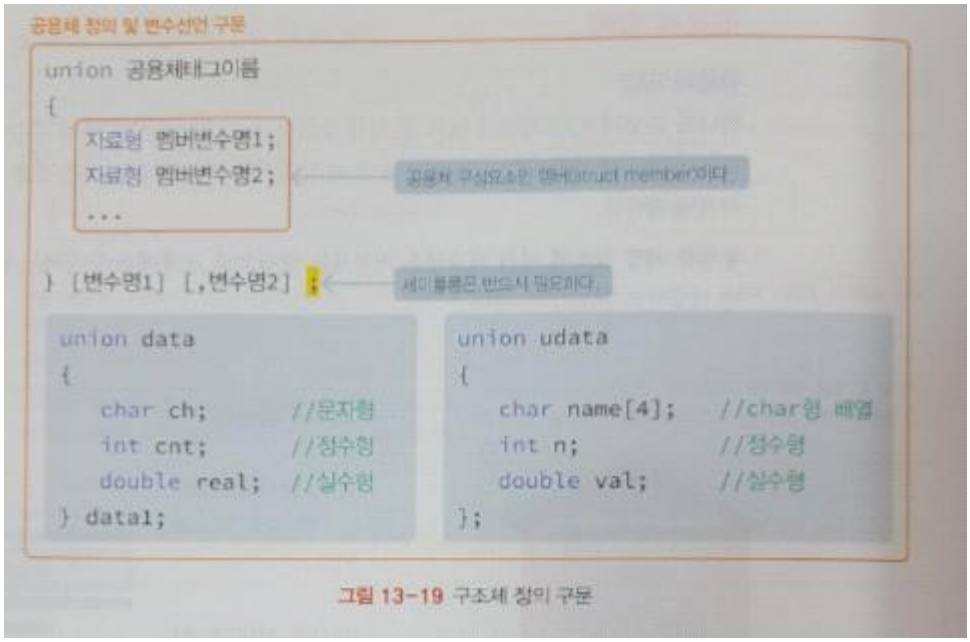


그림 13-19 구조체 정의 구문

공용체 변수의 크기는 멤버 중 가장 큰 자료형의 크기로 정해진다. 즉 위에서 union data의 변수 data1은 멤버 중 가장 큰 크기인 double 형의 8바이트의 저장공간을 세 멤버가 함께 이용한다.

*공용체의 멤버는 모든 멤버가 동일한 저장 공간을 사용하므로 동시에 여러 멤버의 값을 동시에 저장하여 이용할 수 없으며, 마지막에 저장된 단 하나의 멤버 자료값만을 저장한다.

*공용체도 구조체와 같이 typedef를 이용하여 새로운 자료형으로 정의할 수 있다.

*공용체의 초기화 값은 공용체 정의 시 처음 선언한 멤버의 초기값으로만 저장이 가능하다.

공용체인 변수 영역이 다르며,

만일 다른 멤버로 초기값을 지정하면 컴파일 시 다음과 같은 경고가 발생한다. 초기값으로 동일한 유형의 다른 변수의 대입도 가능하다.

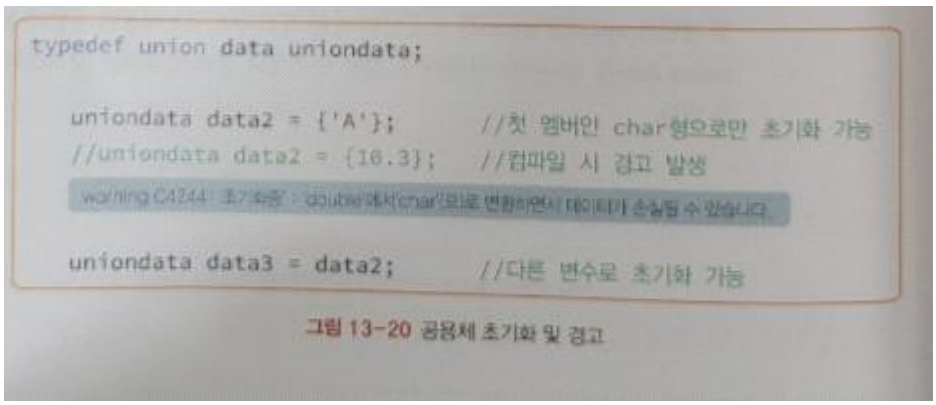


그림 13-20 공용체 초기화 및 경고

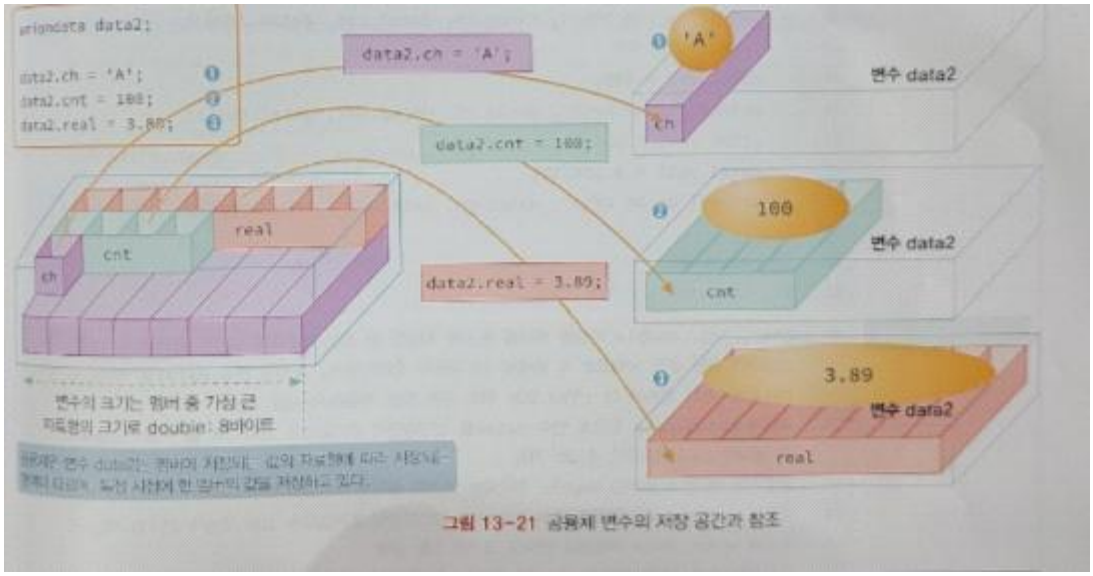


그림 13-21 공용체 변수의 저장 공간과 참조

2. 13-2 자료형 재정의

-자료형 재정의 typedef

=typedef 구문

typedef는 이미 사용되는 자료 유형을 다른 새로운 자료형 이름으로 재정의할 수 있도록 하는 키 워드이다. 문장 typedef int profit;은 profit을 int와 같은 자료형으로 새롭게 정의하는 문장이다.

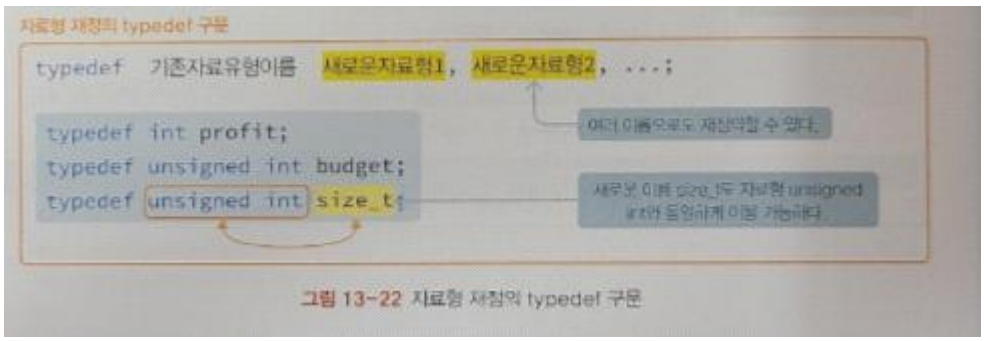


그림 13-22 자료형 재정의 typedef 구문

일반적으로 자료형을 재정의하는 이유는 프로그램의 시스템 간 호환성과 편의성을 위해 필요하다. 터보 C++ 컴파일러에서 자료유형 int는 저장공간 크기가 2바이트이나 Visual C++에서는 4바이 트이다. 그러므로 다음과 같이 Visual C++에서 작성한 프로그램은 터보 C++에서는 문제가 발생 한다. 2바이트로는 2000000을 저장할 수 없기 때문이다.

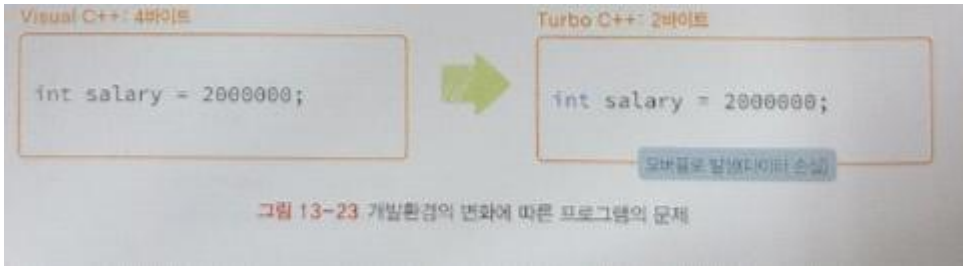


그림 13-23 개발환경의 변화에 따른 프로그램의 문제

이 문제를 해결하는 방안을 살펴보자. Visual C++에서는 다음과 같이 int를 myint로 재정의한 후, 모든 int 형을 myint형으로 선언하여 이용한다. 만일 이 소스를 터보 C++에서 컴파일 한다면, typedef 문장에서 int를 long으로 수정하면 아무 문제 없이 다른 소스는 수정 없이 그대로 이용 가 능하다.

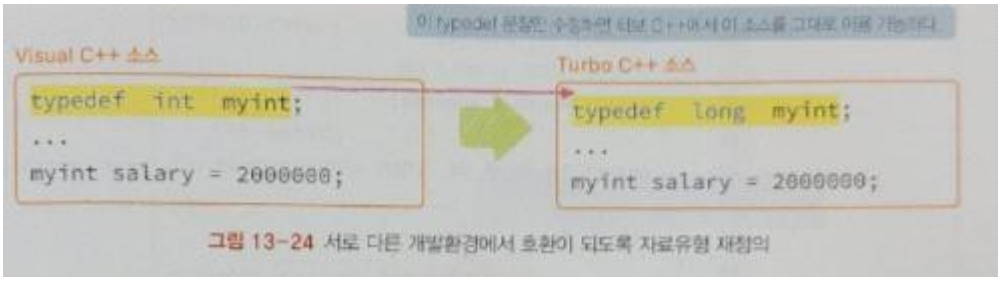


그림 13-24 서로 다른 개발환경에서 호환이 되도록 자료유형 재정의

다음 문장과 같이 자료형 int를 여러 개의 새로운 자료형 이름 integer와 word로 재정의하는 것도 가능하다.

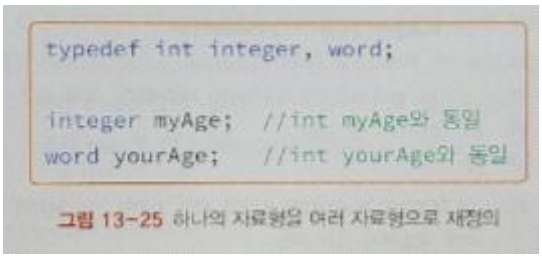


그림 13-25 하나의 자료형을 여러 자료형으로 재정의

문장 typedef도 일반 변수와 같이 그 사용 범위를 제한한다. 그러므로 함수 내부에서 재정의되면 선언된 이후의 그 함수에서만 이용이 가능하다. 만일 함수 외부에서 재정의된다면 재정의된 이후 그 파일에서 이용이 가능하다.

-구조체 자료형 재정의

=struct를 생략한 새로운 자료형

구조체 자료형은 struct date 처럼 항상 키워드 struct를 써야 하나? typedef 사용하여 구조체를 한 단어의 새로운 자료형으로 정의하면 이러한 불편을 덜 수 있다. 구조체 struct date 가 정의된 상태 에서 typedef 사용하여 구조체 struct date를 date로 재정의할 수 있다. 이제 새로운 자료유형인 date는 struct date와 함께 동일한 자료유형으로 이용이 가능하다. 물론 date가 아닌 datetype 등 다른 이름으로도 재정의가 가능하다.

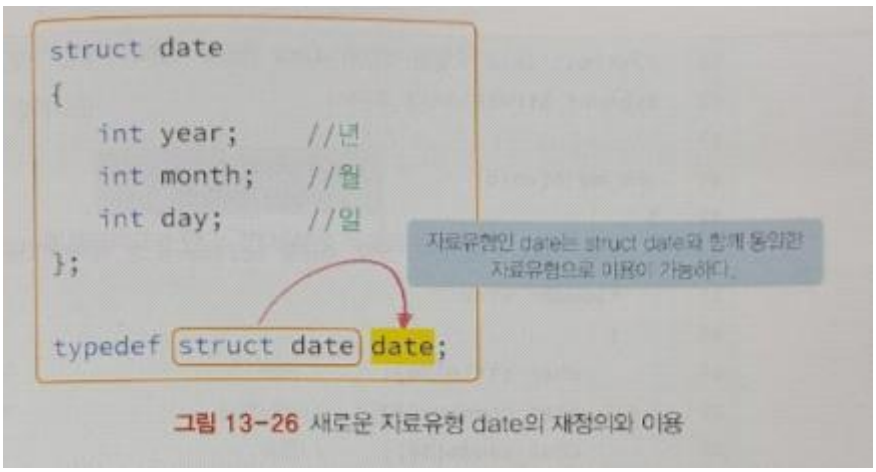


그림 13-26 새로운 자료유형 date의 재정의와 이용

typedef를 이용하는 다른 방법은 구조체 정의 자체를 typedef와 함께 처리하는 방법이다. 아래 typedef 구문에서 새로운 자료형으로 software 형이 정의되며, 이 구문 이후에는 software를 구 조체 자료형으로 변수 선언에 사용할 수 있다. 이 경우 구조체 태그이름은 생략 가능하다. 구조체 software형은 멤버로 구조체 date형 변수 release를 갖는다.

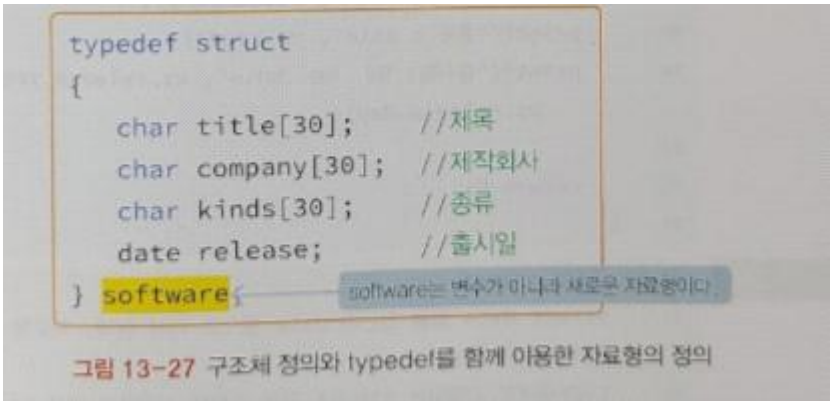


그림 13-27 구조체 정의와 typedef를 함께 이용한 자료형의 정의

3. 13-3 구조체와 공용체의 포인터와 배열

-구조체 포인터

=포인터 변수 선언

포인터는 각각의 자료형 저장 공간의 주소를 저장하듯이 구조체 포인터는 구조체의 주소값을 저장 할 수 있는 변수이다. 구조체 포인터 변수의 선언은 일반 포인터 변수 선언과 동일하다. 다음은 대학 강좌를 처리하는 구조체 자료형 lecture를 선언한 구문이다. 구조체 포인터 변수 p는 lecture *p로 선언된다.

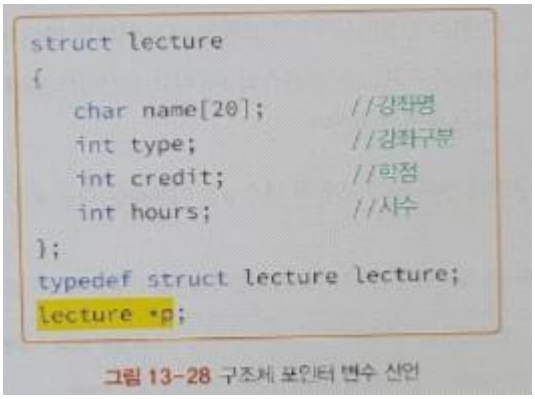


그림 13-28 구조체 포인터 변수 선언

구조체 포인터 변수는 다른 포인터 변수와 사용 방법이 동일하다. 변수 os를 선언한 후, 문장 lecture *p = &os ;로 lecture 포인터 변수 p에 &os를 저장한다. 이로써 포인터 p로 구조체 변수 os 멤버 참조가 가능하다.

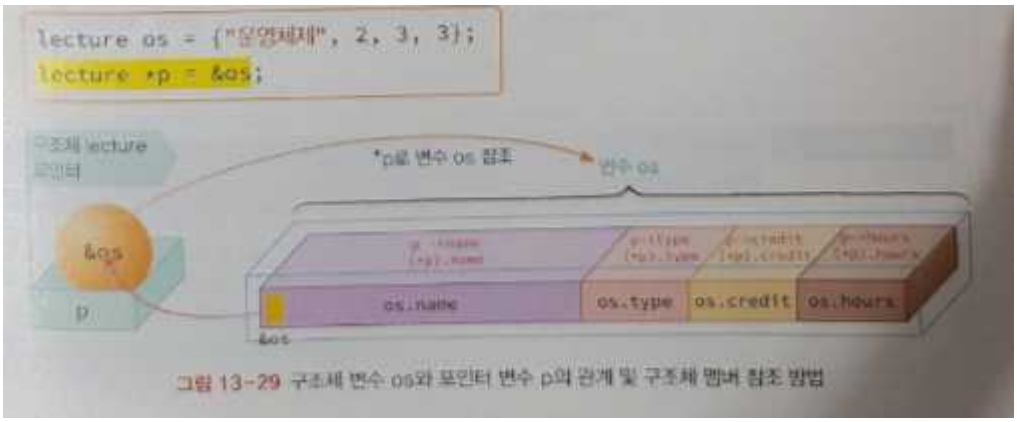


그림 13-29 구조체 변수 os와 포인터 변수 p의 관계 및 구조체 멤버 참조 방법

=포인터 변수의 구조체 멤버 접근 연산자 -)

구조체 포인터 멤버 접근연산자 ->는 p->name과 같이 사용한다. 연산식 p->name은 포인터 p 가 가리키는 구조체 변수의 멤버 name을 접근하는 연산식이다. 마찬가지로 p->type, p->credit, l p->hours는 각각 os.type, os.credit, os.hours를 참조하는 연산식이다.

*구조체 포인터의 접근연산자인 ->는 두 문자가 연결된 하나의 연산자이므로 - 와〉 사이에 공백이 들어가서는 절대 안된다.

*연산식 p->name은 접근연산자(.)와 간접연산자(*)를 사용한 연산식 (*p).name으로도 사용 가능하다. 그러나 (*p).name은 *p.name과는 다르다는 것에 주의하자.

*연산식 *p.name은 접근연산자(.)가 간접연산자(*)보다 우선순위가 빠르므로 *(p.name)과 같은 연산식이다. p가 포인터이므로 p.name 는 문법오류가 발생한다. *구조체 포인터 멤버 접근연산자 ->와 구조체 변수의 구조체 멤버 접근연산자 의 연산자 우선 순위는 간접연산자 *를 포함한 다른 어떠한 연산자 우선순위보다 가장 높다.

*연산자 ->와 은 우선순위 1위이고 결합성은 좌에서 우이며, 연산자 *은 우선순위 2이고 결합성은 우에서 좌이다.

구조체 변수 os와 포인터 변수 p를 이용하는 다음 4가지 구문을 잘 구별하도록 하자.

표 13-2 구조체 변수와 구조체 포인터 변수를 이용한 멤버의 참조

접근 연산식	구조체 변수 os와 구조체 포인터변수 p인 경우의 의미
p->name	포인터 p가 가리키는 구조체의 멤버 name
(*p).name	포인터 p가 가리키는 구조체의 멤버 name
*p.name	*(p.name)이고 p가 포인터이므로 p.name은 문법오류가 발생
*os.name	*(os.name)를 의미하며, 구조체 변수os의 멤버 포인터 name이 가리키는 변수로, 이 경우는 구조체 변수 os 멤버 감체명의 첫 문자임, 다만 한글인 경우에는 실행 오류
*p->name	*(p->name)을 의미하며, 포인터 p이 가리키는 구조체의 멤버 name이 가리키는 변수로 이 경우는 구조체 포인터 p이 가리키는 구조체의 멤버 감체명의 첫 문자임, 마찬가지로 한글인 경우에는 실행 오류

=공용체 포인터

공용체 변수도 포인터 변수 사용이 가능하며, 공용체 포인터 변수로 멤버를 접근하려면 접근연산자 ->를 이용한다. 다음은 공용체 변수 value를 가리키는 포인터 p를 선언하여 p가 가리키는 공용 체 멤버 ch에 'a'를 저장하는 소스이다.

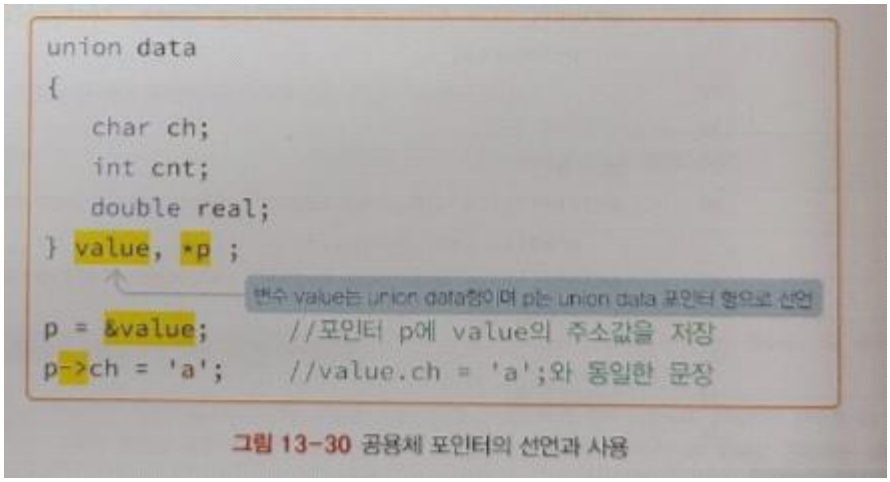


그림 13-30 공용체 포인터의 선언과 사용

구조체와 같이 연산식 p->ch는 포인터가 가리키는 공용체 변수의 멤버ch를 접근하는 연산식이다. 마찬가지로 p->cnt, p->real은 각각 value.cnt, value.real을 참조하는 연산식이다.

-구조체 배열

=구조체 배열 변수 선언

다른 배열과 같이 동일한 구조체 변수가 여러 개 필요하면 구조체 배열을 선언하여 이용할 수 있다. 다음은 구조체 lecture의 배열크기 3인 c를 선언하고 초기값을 저장하는 구문이다. 구조체 배열 의 초기값 지정 구문에서는 중괄호가 중첩되게 나타난다. 외부 중괄호는 배열 초기화의 중괄호이 며, 내부 중괄호는 배열원소인 구조체 초기화를 위한 중괄호이다.

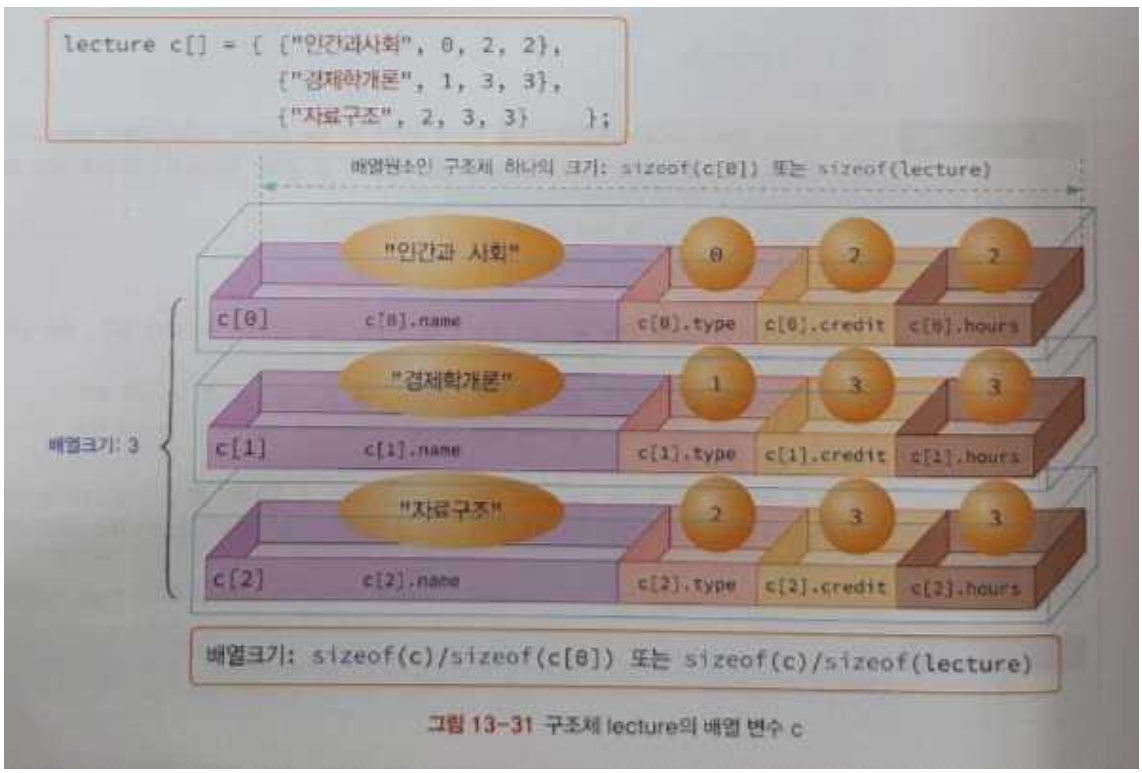


그림 13-31 구조체 lecture의 배열 변수 c

문장 lecture *p = c;와 같이 구조체 배열이름은 구조체 포인터 변수에 대입이 가능하다. 이제

구조 체 포인터 변수 p를 이용한 p[i]로 배열원소 접근이 가능하다.

```
lecture *p = c;

//p로도 참조 가능
for (i = 0; i < arysize; i++)
  printf("%16s %10s %5d %5d\n", p[i].name, lectype[p[i].type],p[i].credit, p[i].hours);
```

그림 13-32 배열의 주소를 저장

14장. 함수와 포인터 활용

1. 14-1 함수의 인자전달 방식

-값에 의한 호출과 참조에 의한 호출

=함수에서 값의 전달

C 언어는 함수의 인자 전달 방식이 기본적으로 값에 의한 호출(call by value) 방식이다. 값에 의한 호출 방식이란 함수 호출 시 실인자의 값이 형식인자에 복사되어 저장된다는 의미이다. 다음 함수 increase(int origin, int increment)는 origin += increment; 를 수행하는 간단한 함수이다. 함수 main()에서 변수 amount에 10을 저장한 후 increase(amount, 20) 로 함수를 호출한다. 이 후 변수 amount를 출력해 보자. 출력값이 30이 아니라 여전히 10이다.

*함수 호출 시 변수 amount의 값 10이 매개변수인 origin에 복사되고, 20이 매개변수인 increment에 복사된다. 이러한 방식을 값에 의한 호출이라 한다.

함수 increase() 내부실행에서 매개변수인 origin 값이 30으로 증가되고 다시 제어는 main() 함수 로 돌아온다. main() 내부에서 amount의 값은 여전히 변하지 않고 10이다. 즉 변수 amount와 매 개변수 origin은 아무 관련성이 없다. 그러므로 origin은 증가해도 amount의 값은 변하지 않는다. 그러므로 C 언어에서 값에 의한 호출 방식을 사용해서는 함수 외부의 변수를 함수 내부에서 수정 할 수 없는 특징이 있다.

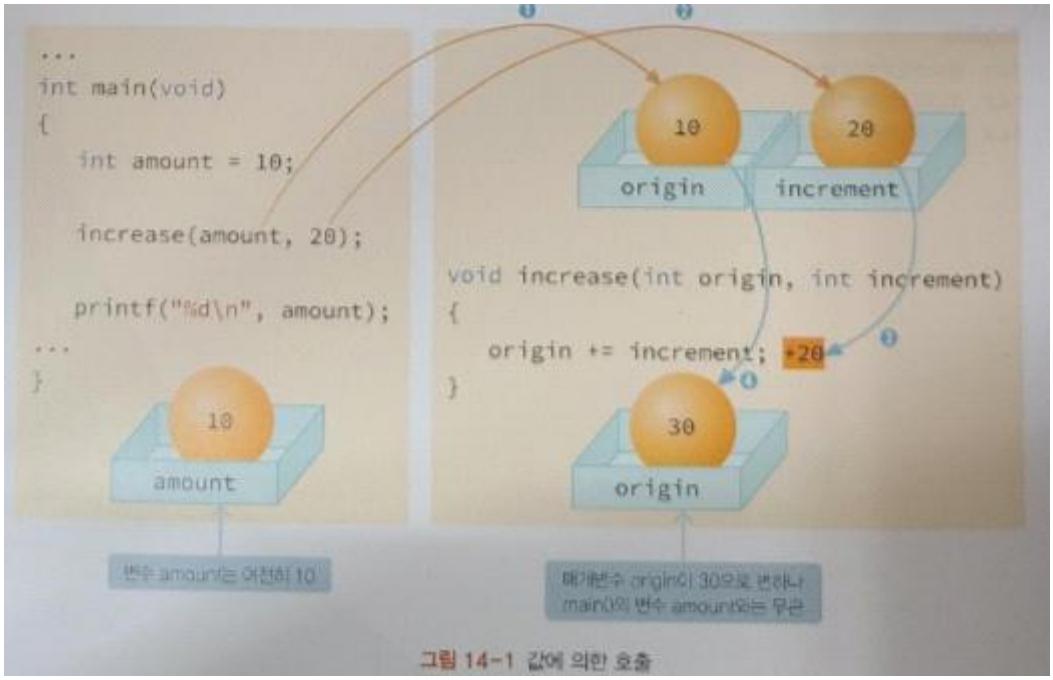


그림 14-1 값에 의한 호출

=함수에서 주소의 전달

C 언어에서 포인터를 매개변수로 사용하면 함수로 전달된 실인자의 주소를 이용하여 그 변수를 참조할 수 있다. 이와 같이 함수에서 주소의 호출을 참조에 의한 호출(call by reference)이라 한다.

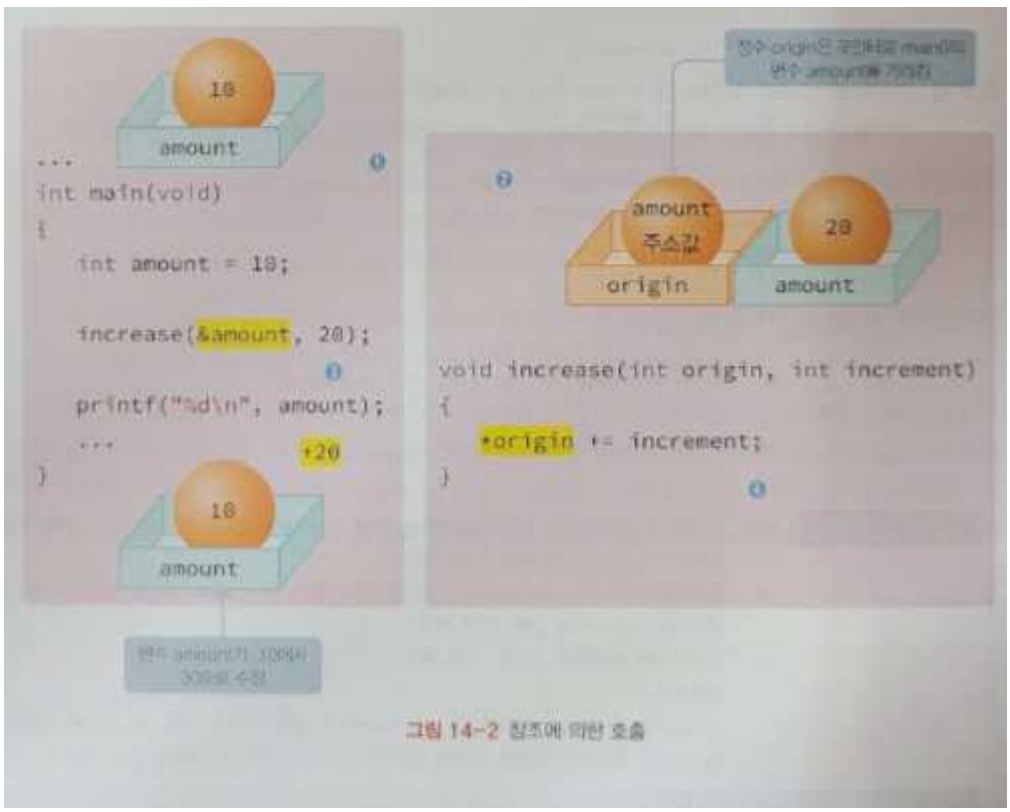


그림 14-2 참조에 의한 호출

-배열의 전달

=배열이름으로 전달

함수의 매개변수로 배열을 전달하는 것은 배열의 첫 원소를 참조 매개변수로 전달하는 것과 동일하다. 다음과 같이 배열을 매개변수로 하는 함수 sum()을 구현해보자. 함수 sum()은 실수형 배열의 모든 원소의 합을 구하여 반환하는 함수이다. 함수 sum()의 형식매개변수는 실수형 배열과 배열크기로 한다. 첫 번째 형식매개변수에서 배열자체에 배열크기를 기술하는 것은 아무 의미가 없다. 그러므로 double ary[5]보다는 double ary[]라고 기술하는 것을 권장한다.

*실제로 함수 내부에서 실인자로 전달된 배열의 배열 그기를 알 수 없다.

*그 이유는 매개변수를 double ary[] 처럼 배열형태로 기술해도 단순히 double Hary 처럼 포인터 변수로 인식하기 때문이다.

*그러므로 배열크기를 두 번째 인자로 사용한다.

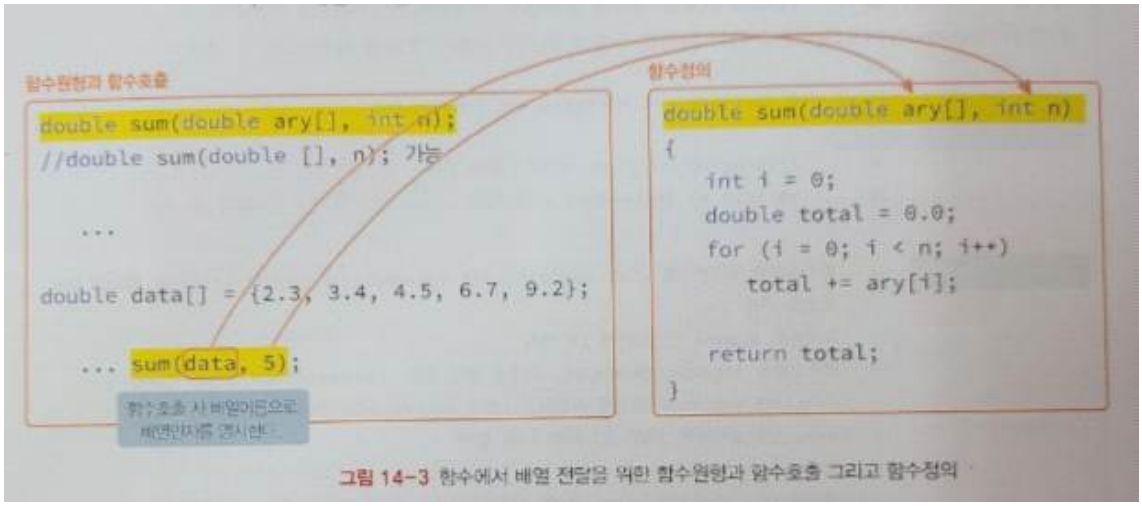


그림 14-3 함수에서 배열 전달을 위한 함수원형과 함수호출 그리고 함수정의

만일 배열크기를 인자로 사용하지 않는다면 정해진 상수를 함수정의 내부에서 사용해야 할 것이다. 그러나 이런 방법은 배열크기가 변하면 소스를 수정해야 하므로 비효율적이다. 배열크기에 관계없이 배열 원소의 합을 구하는 함수를 만들려면 배열크기도 하나의 인자로 사용해야 한다.

함수정의가 구현되면 함수원형을 선언한 뒤 함수호출이 가능하다. 함수원형에서 매개변수는 배열이름을 생략하고 double []와 같이 기술할 수 있다. 함수호출에서 배열 인자에는 반드시 배열이름으로 sum(data, 5)와 같이 기술해야 한다.

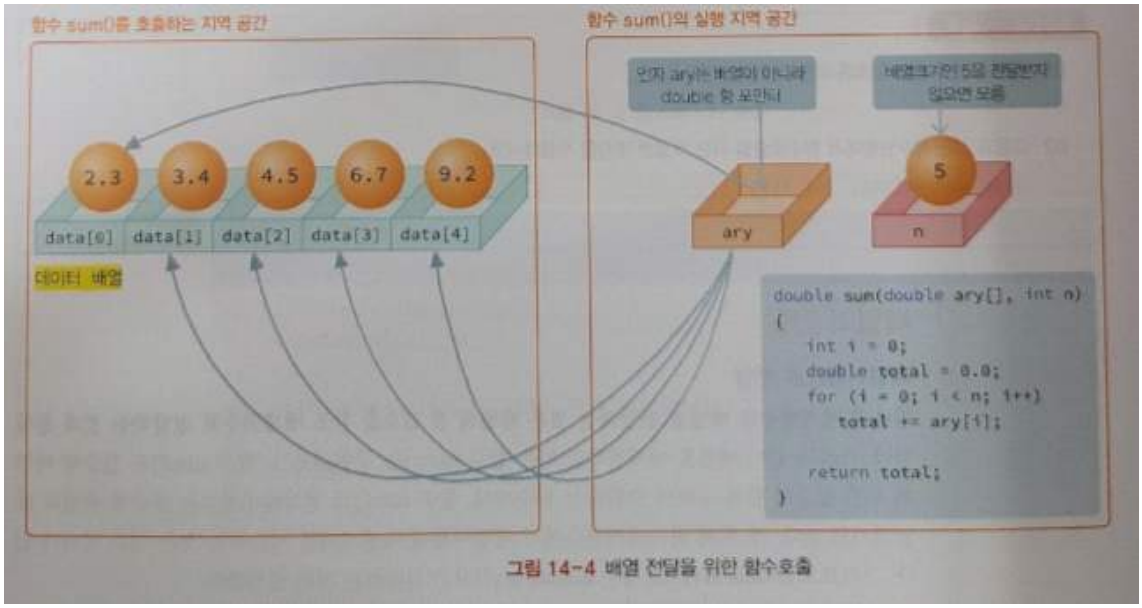


그림 14-4 배열 전달을 위한 함수호출

=다양한 배열원소 참조 방법

다음 소스의 배열 point에서 간접연산자를 사용한 배열원소의 접근 방법은 *(point + i)이다. 그러므로 배열의 합을 구하려면 sum += *(point + i); 문장을 반복한다.

*문장 int address = point;로 배열 point를 가리키는 포인터 변수 address를 선언하여 point를 저장해 보자. 이제 문장 sum += *(address++)으로도 배열의 합을 구할 수 있다.

*그러나 배열이름 point는 주소 상수이기 때문에 sum += *(point++)는 사용할 수 없다. 증가연산식 point++의 피연산자로 상수인 point를 사용할 수 없기 때문이다.

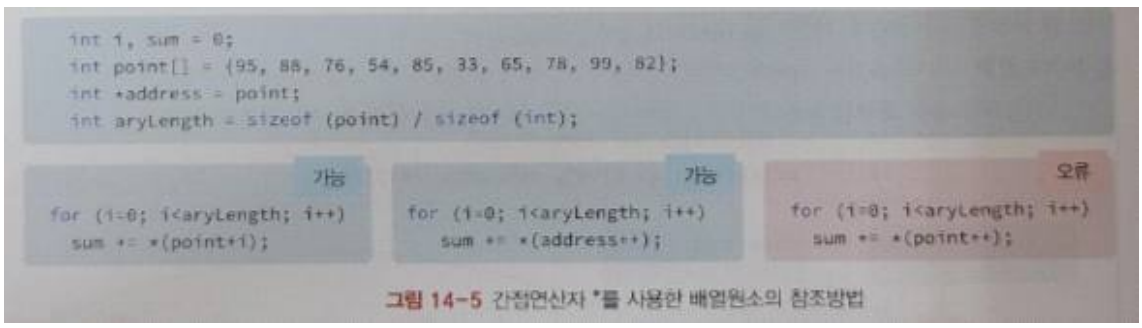


그림 14-5 간접연산자 *를 사용한 배열원소의 참조방법

함수헤더에 배열을 인자로 기술하는 다양한 방법에 대해 알아보자.

*함수헤더에 int ary[]로 기술하는 것은 int *ary로도 대체 가능하다.

=배열크기 계산방법

배열이 함수인자인 경우 대부분 배열크기도 함수인자로 하는 경우가 흔하다. 이 배열크기를 상수로 기술하는 것보다 연산자 sizeof를 이용하여 배열크기를 계산해 보낸다면 더욱 좋은 프로그램이 된다. 저장공간의 크기를 바이트 수로 반환하는 연산자 sizeof를 이용하면 쉽게 배열크기를 알 수 있다.

*연산자 sizeof를 이용한 식 (sizeof(배열이름) / sizeof(배열원소))의 결과는 배열크기이다.

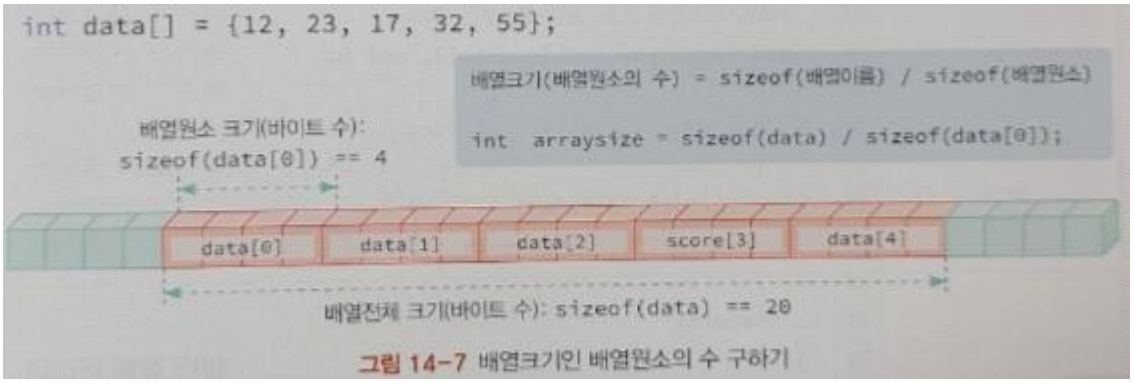


그림 14-7 배열크기인 배열원소의 수 구하기

=다차원 배열 전달

이차원 배열을 함수 인자로 이용하는 방법을 알아보기 위해 이차원 배열에서 모든 원소의 합을 구하는 함수를 구현해 보자.

*다차원 배열을 인자로 이용하는 경우, 함수원형과 함수정의의 헤더에서 첫 번째 대괄호 내부의 크기를 제외한 다른 모든 크기는 반드시 기술되어야 한다.

*그러므로 이차원 배열의 행의 수를 인자로 이용하면 보다 일반화된 함수를 구현할 수 있다.

함수 sum()은 인자인 이차원 배열값을 모두 더하는 함수이며, 함수 printarray()는 인자인 이차원 배열값을 모두 출력하는 함수이다.

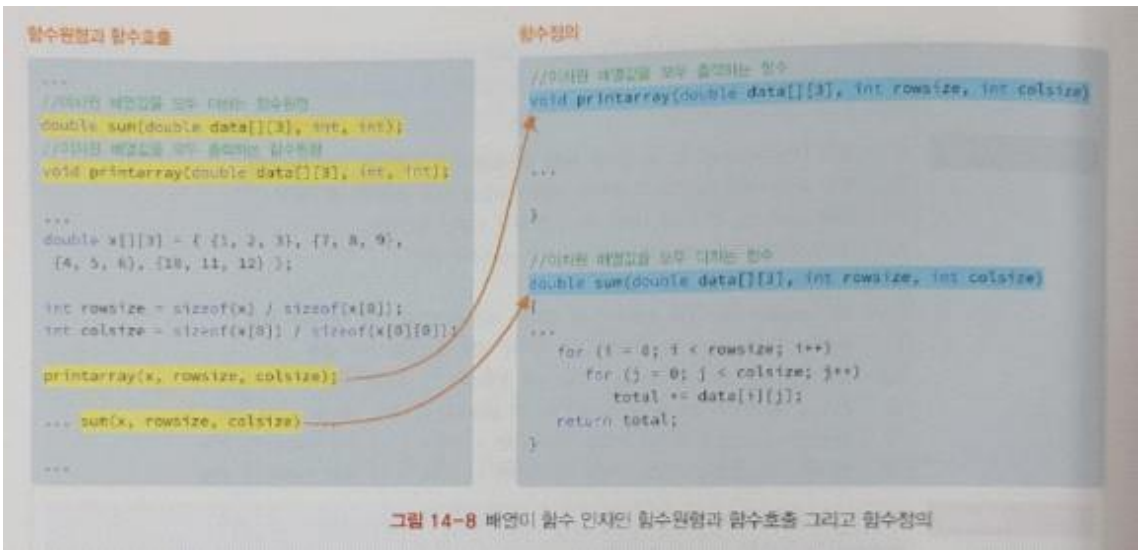


그림 14-8 배열이 함수 인자인 함수원형과 함수호출 그리고 함수정의

함수 sum()을 호출하려면 배열이름과 함께 행과 열의 수가 필요하다.

*이차원 배열의 행의 수는 다음과 같이 (sizeof(x) / sizeof(x[0])로 계산할 수 있다.

*또한 이차원 배열의 열의 수는 다음과 같이 (sizeof(x[0]) / sizeof(x[0][0]))로 계산할 수 있다.

*여기서 sizeof(x)는 배열 전체의 바이트 수를 나타내며 sizeof(x[0])는 1행의 바이트 수, sizeof(x[0][0])은 첫 번째 원소의 바이트 수를 나타낸다.

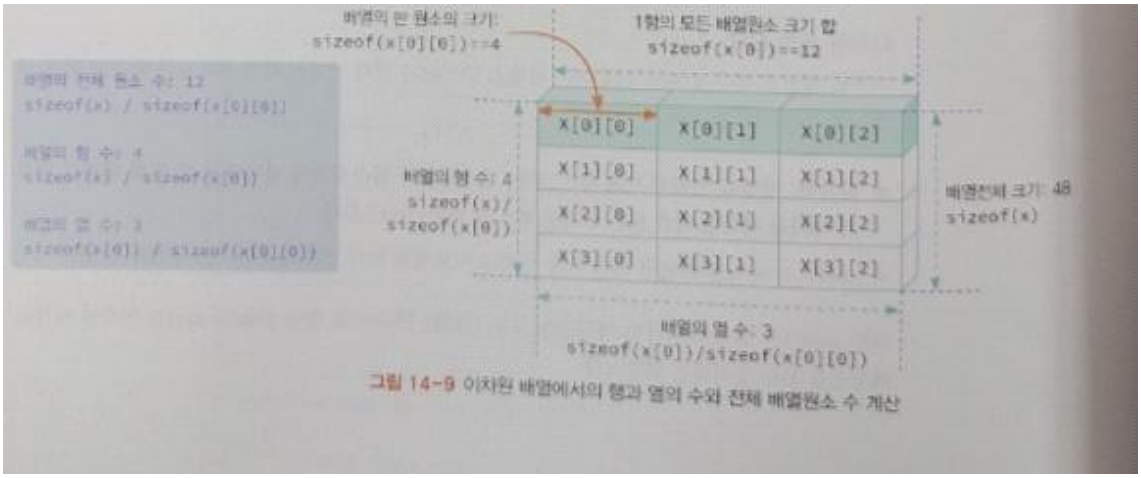


그림 14-9 이차원 배열에서의 행과 열의 수와 전체 배열원소 수 계산

-가변 인자

=가변 인자가 있는 함수머리

함수 printf()는 다음과 같은 함수원형으로 되어 있으며, 첫 인자는 char * Format을 제외하고는 이후에 ... 표시가 되어 있다. 함수 printf()를 호출하는 경우를 살펴보면, 출력할 인자의 수와 자료형이 결정되지 않은 채 함수를 호출하는데, 출력할 인자의 수와 자료형은 인자 Format에 %d 등 으로 표현되어 있는 것을 말 수 있다. 함수 scanf()도 마찬가지이다.

```
//함수 printf()의 함수원형
int printf(const char *_Format, ...); //...이 무엇일까?

//함수 사용 예
printf("%d%d%f", 3, 4, 3.678); //인자가 총 4개
printf("%d%d%f%f%f", 7, 9, 2.45, 3.678, 8.98); //인자가 총 5개
```

그림 14-10 함수 printf()의 함수원형과 함수호출 사용 예

이와 같이 함수에서 인자의 수와 자료형이 결정되지 않은 함수 인자 방식을 가변 인자(variable argument)라 한다.

*함수의 가변 인자란 함수에서 처음 또는 앞 부분의 매개변수는 정해져 있으나 이후 매개변수 수와 각각의 자료형이 고정적이지 않고 변하는 인자를 말한다. *함수의 매개변수에서 중간 이후부터 마지막에 위치한 가변 인자만 가능하다.

*함수 정의 시 가변인자의 매개변수는 ... 으로 기술한다. 즉 함수 vatest의 함수 헤드는 void vatest(int n, ...)와 같이, 가변 인자인 ... 의 앞 부분에는 반드시 매개변수가 int n처럼 고정적이어야 하며, 이후 가변인자인 ...을 기술할 수 있다.

*가변인자 ... 시작 전 첫 고정 매개변수는 이후의 가변인자를 처리하는데 필요한 정보를 지정하는데 사용한다.

```
int vatest(int n, ...);
double vasum(char *type, int n, ...);
double vafun1(char *type, ..., int n); //오류, 마지막이 고정적일 수 없음
double vafun2(...); //오류, 처음부터 고정적일 수 없음
```

그림 14-11 함수원더에서의 가변 인자 표현

=가변 인자가 있는 함수 구현

함수에서 가변 인자를 구현하려면 가변인자 선언, 가변인자 처리 시작, 가변인자 얻기, 가변인자 처리 종료 4단계가 필요하다. 가변인자 처리 절차 4단계를 정리하면 다음과 같으며, 이를 구현하기 위해서는 헤더파일 stdarg.h가 필요하다.

*가변인자 선언은 마치 변수선언처럼 가변인자로 처리할 변수를 하나 만드는 일로서, 자료형인 va_list는 가변 인자를 위한 char *로 헤더파일 stdarg.h에 정의되어 있다.

*가변인자 처리 시작은 위에서 선언된 변수에서 마지막 고정 인자를 지정해 가변 인자의 시작 위치를 알리는 방법으로, 함수 va_start()는 헤더파일 stdarg.h에 정의되어 있는 매크로 함수이다.

*가변인자 얻기 처리에서는 가변인자 각각의 자료형을 지정하여 가변인자를 반환 받는 절차로 함수 va_arg()도 헤더파일 stdarg.h에 정의된 매크로 함수이다. 매크로 함수 va_arg()의 호출로 반환된 인자로 원하는 연산을 처리할 수 있다.

*가변인자 처리 종료는 표현 그대로 가변 인자에 대한 처리를 끝내는 단계로 va_end() 함수는 헤더파일 stdarg.h에 정의된 매크로 함수이다.

표 14-1 가변인자 처리를 위한 네 가지 절차

구문	처리 절차	설명
va_list argp;	❶ 가변인자 선언	va_list로 변수 argp를 선언
va_start(va_list argp, prevarg)	❷ 가변인자 처리 시작	va_start()는 첫 번째 인자로 va_list로 선언된 변수이름 argp과 두 번째 인자는 가변인자 앞의 고정인자 prevarg 를 지정하여 가변인자 처리 시작
type va_arg(va_list argp, type)	❸ 가변인자 얻기	va_arg()는 첫번째 인자로 va_start()로 초기화된 va_list 변수 argp를 받으며, 두번째 인자로는 가변 인자로 전달된 값의 type을 기술
va_end(va_list argp)	❹ 가변인자 처리 종료	va_list로 선언된 변수이름 argp의 가변인자 처리 종료

가변인자 처리 절차와 가변인자가 있는 함수 sum(int numargs, ...)에서의 구현을 정리하면 다음으로 요약할 수 있다. 함수 sum()에서 가변인자 앞의 첫 고정인자인 numargs는 가변인자의 수이며, int 형인 가변인자를 처리하여 그 결과를 반환하는 함수이다.

*가변인자 ... 시작 전 첫 고정 매개변수는 이후의 가변인자를 처리하는데 필요한 정보를 지정하는데 사용한다. 그러므로 다음의 numargs처럼 반드시 가변인자의 수일 필요는 없다.

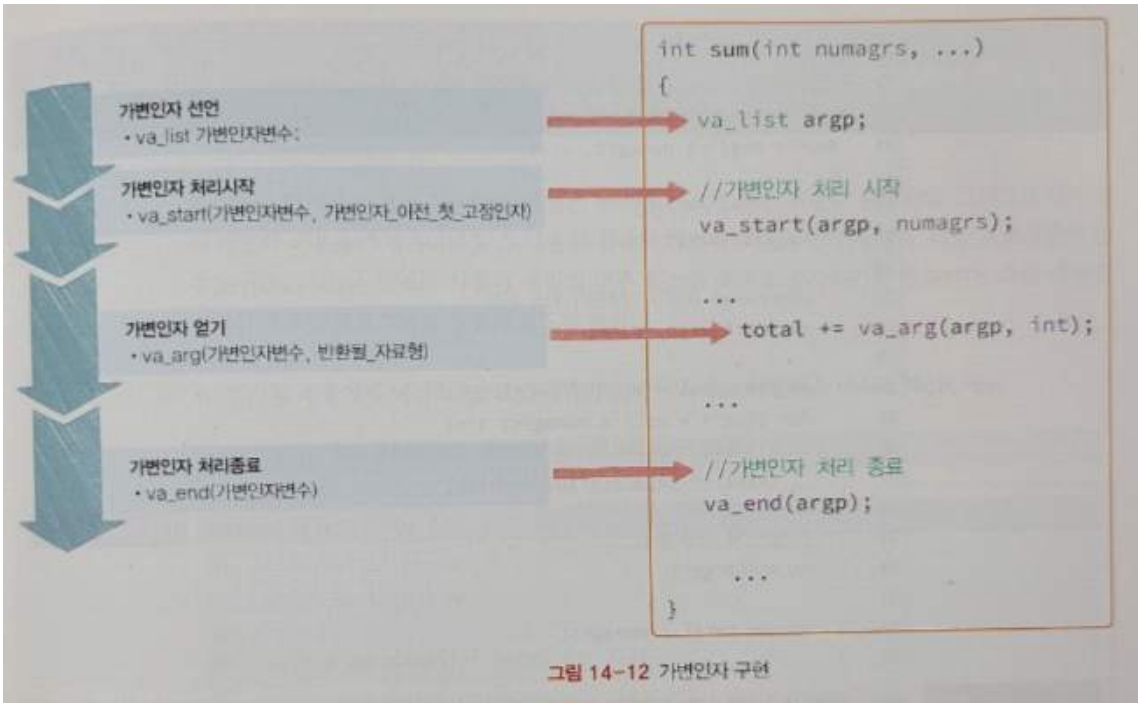


그림 14-12 가변인자 구현

2. 14-2 포인터 전달과 반환

-매개변수와 반환으로 포인터 사용

=주소연산자 &

함수에서 매개변수를 포인터로 이용하면 결국 참조에 의한 호출이 된다. 함수의 매개변수로 포인 터를 이용하는 다른 예를 살펴보자.

*함수원형 void add(int *, int, int); 에서 첫 매개변수가 포인터인 int *이다.

*함수 add()는 두 번째와 세 번째 인자를 합해 첫 번째 인자가 가리키는 변수에 저장하는 함수이다.

*이 함수를 호출하는 방법은 합이 저장될 변수인 sum을 선언하여 주소값인 &sum을 인자로 호 출한다.

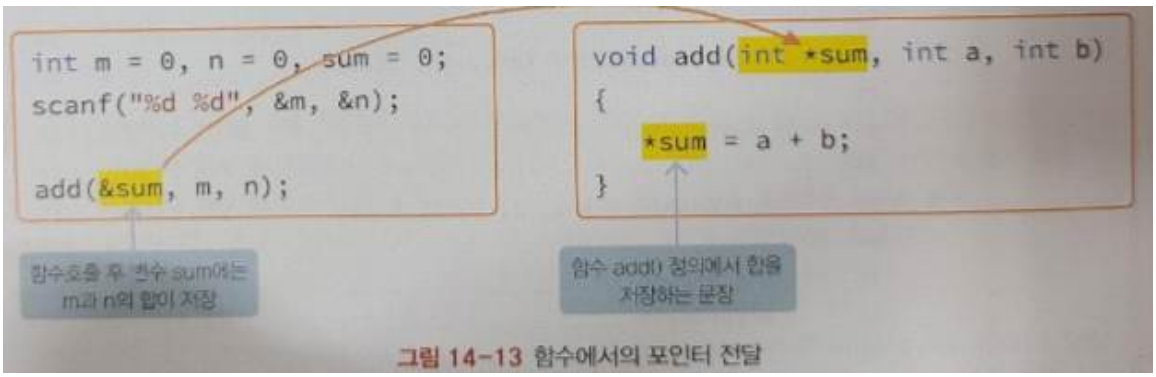


그림 14-13 함수에서의 포인터 전달

=주소값 반환

함수의 결과를 포인터로 반환하는 예를 살펴보자. 함수원형을 int * add(int *, int, int) 로 하는 함 수 add()는 반환값이 포인터인 int *이다. 함수 add() 정의에서 두 수의 합을 첫 번째 인자가 가리 키는 변수에 저장한 후 포인터인 첫 번째 인자를 그대로 반환한다. 함수 add()를

*add(&sum, m, n)호출하면 변수 sum에 합 a+b가 저장된다. 또한 반환값인 포인터가 가리키는 변수인 sum을 바 로 참조할 수 있다.

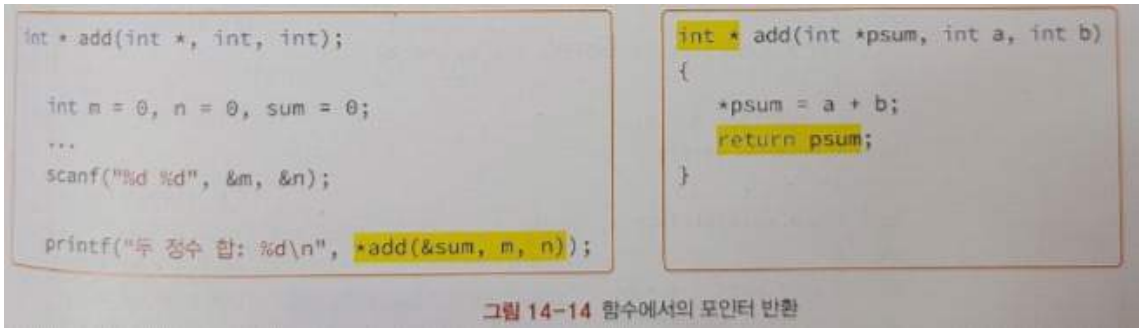


그림 14-14 함수에서의 포인터 반환

다음 예제의 함수 multiply()에서 인자인 두 수의 곱을 지역변수인 mult에 저장한 후 &mult로 포인터를 반환한다. 지역변수는 함수가 종료되는 시점에 메모리에서 제거되는 변수이다. 그러므로 지역변수 주소값의 반환은 문제를 발생시킬 수 있다. 실제로 Visual C++에서 컴파일 시 다음과 같은 경고가 발생한다. 그러므로 제거될 지역변수의 주소값은 반환하지 않는 것이 바람직하다.

-상수를 위한 const 사용

=키워드 const

포인터를 매개변수로 이용하면 수정된 결과를 받을 수 있어 편리하다. 그러나 이러한 참조에 의한 호출은 매개변수가 가리키는 변수값이 원하지 않는 값으로 수정될 수 있다. 이러한 포인터 인자의 잘못된 수정을 미리 예방하는 방법이 있다. 즉 수정을 원하지 않는 함수의 인자 앞에 키워드 const 를 삽입하여 참조되는 변수가 수정될 수 없게 한다.

*키워드 const는 인자인 포인터 변수가 가리키는 내용을 수정할 수 없도록 한다.

*다음과 같이 인자를 const double *a와 const double *b로 기술하면 *a와 b를 대입연산자의 l-value로 사용할 수 없다. 즉 a와 *b를 이용하여 그 내용을 수정할 수 없다.

*상수 키워드 const의 위치는 자료형 앞이나 포인터변수 *a 앞에도 가능하므로, const double a와 double const *a 는 동일한 표현이다.

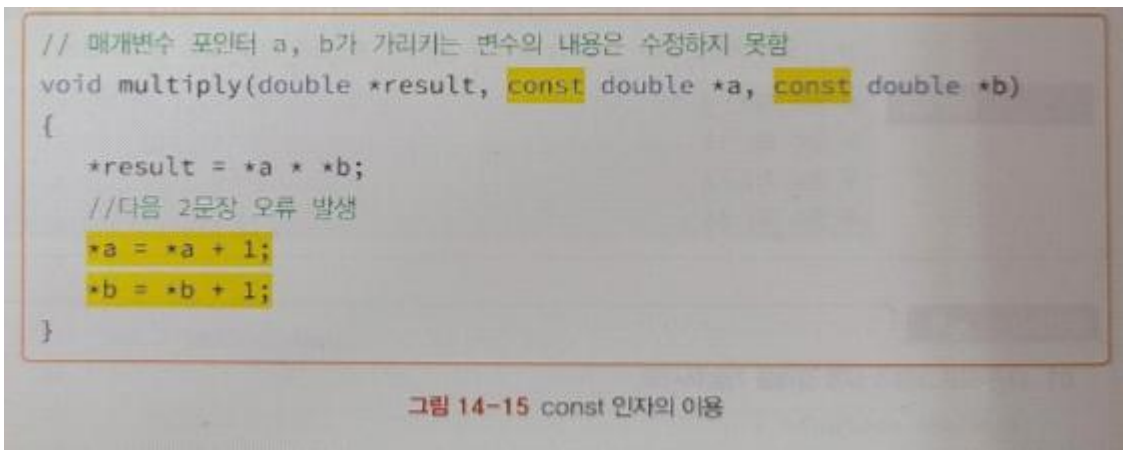


그림 14-15 const 인자의 이용

-함수의 구조체 전달과 반환

=복소수를 위한 구조체

구조체 complex를 이용하여 복소수 연산에 이용되는 함수를 만들어보자. 우선 복소수의 자료형인 구조체 complex는 실수부와 허수부를 나타내는 real과 img를 멤버로 구성한다. 복소수 자료형으로 complex를 재정의를 하여 구조체 자료형으로 사용하도록 하자.

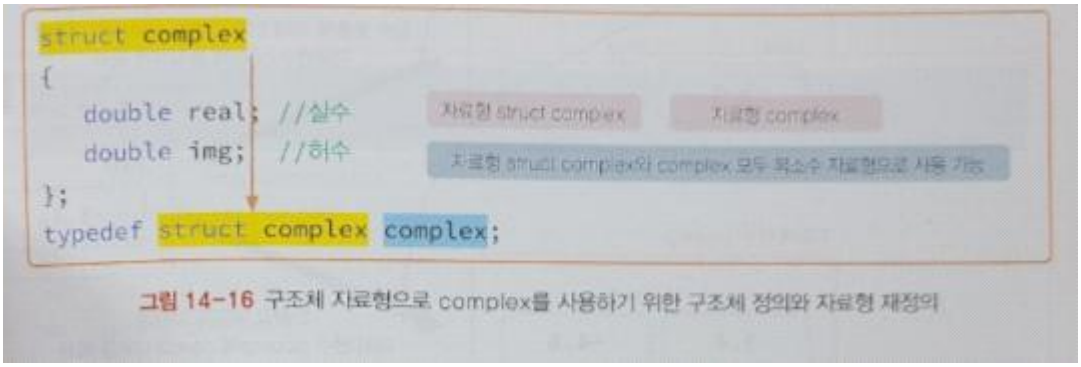


그림 14-16 구조체 자료형으로 complex를 사용하기 위한 구조체 정의와 자료형 재정의

인자와 반환형으로 구조체 사용 함수 paircomplexl()는 인자인 복소수의 켤레 복소수(pair complex number)를 구하여 반 환하는 함수로 정의한다. 복소수 (α + bi)의 켤레 복소수는 (α - bi)이므로 다음 소스에서 변수 pcomp에는 {3.4, -4.8} 이 저장될 것이다. 함수 paircomplexl()을 구현해 보자. 이 함수는 유형이 complex인 인자의 켤레 복소수를 구하여 그 결과를 반환하므로 다음과 같이 구현할 수 있다. 함수 에서 구조체의 반환 방법도 다른 일반 변수와 같다.

*구조체는 함수의 인자와 반환값으로 이용이 가능하다. 다음 함수는 구조체 인자를 값에 의한 호출(call by value) 방식으로 이용하고 있다.

*즉 함수 paircomplexl() 내부에서 구조체 지역변수 com을 하나 만들어 실인자의 구조체 값을 모두 복사하는 방식으로 구조체 값을 전달 받는다.

*구조체 자체를 전달하기 위해 대입하고 다시 반환값을 대입하기 위해서는 시간이 소요된다.

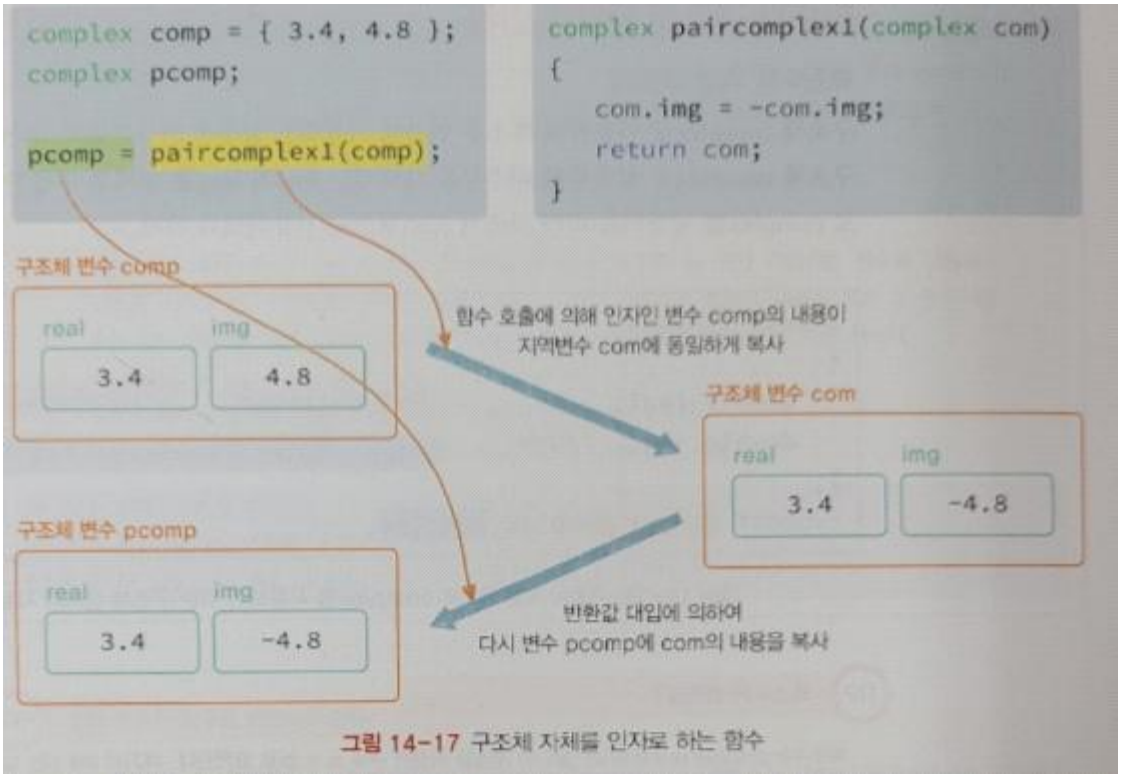


그림 14-17 구조체 자체를 인자로 하는 함수

이 함수를 참조에 의한 호출(call by reference) 방식으로 바꾸어보자. 다음 함수 paircomplex2()는 인자를 주소값으로 저장하여, 실인자의 변수 comp의 값을 직접 수정하는 방식이다. 이 함수를 호 출하기 위해서는 &pcomp처럼 주소값을 이용해 호출한다.

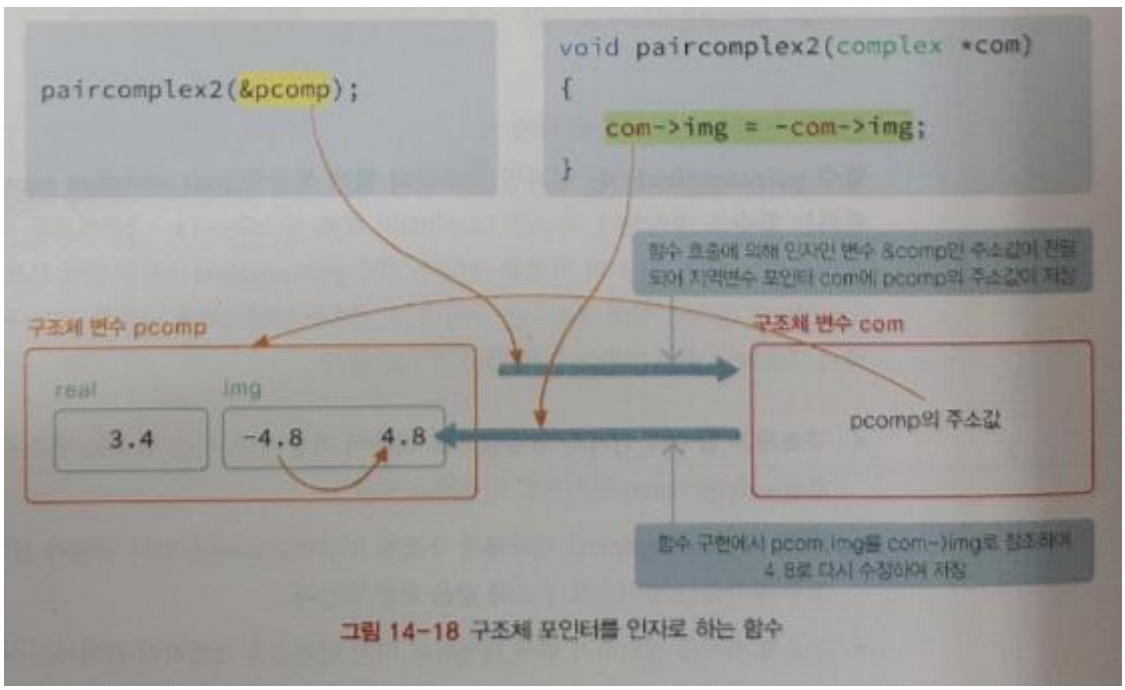


그림 14-18 구조체 포인터를 인자로 하는 함수

구조체를 함수의 인자로 사용하는 방식은 다른 변수와 같이 값에 의한 호출과 참조에 의한 호출 방 식을 사용할 수 있다.

*구조체가 크기가 매우 큰 구조체를 값에 의한 호출의 인자로 사용한다면 매개변수의 메모리 할

당과 값의 복사에 많은 시간이 소요될 수 있다.
*이에 반해 주소값을 사용하는 참조에 의한 호출 방식은 메모리 할당과 값의 복사에 드는 시간이 없는 장점이 있다.

3. 14-3 함수 포인터와 void 포인터

-함수 포인터
=함수 주소 저장 변수
포인터의 장점은 다른 변수를 참조하여 읽거나 쓰는 것도 가능하다는 것이다. 이처럼 하나의 함수 이름으로 필요에 따라 여러 함수를 사용하면 편리할 수 있다. 이것을 가능하도록 하는 것이 함수 포인터이다.
그림처럼 함수 포인터 pfun은 함수 add()와 mult() 그리고 subt()로도 사용 가능하도록 한다.

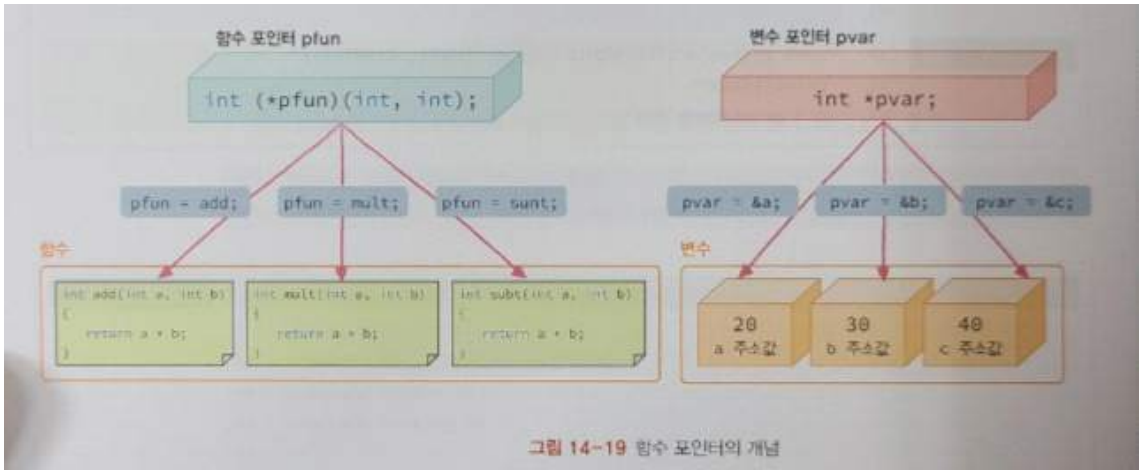


그림 14-19 함수 포인터의 개념

함수 포인터(pointer to function)는 함수의 주소값을 저장하는 포인터 변수이다. 즉 함수 포인터는 함수를 가리키는 포인터를 말한다. 함수도 변수처럼 메모리 어딘가에 저장되어 있으며 그 주소를 갖고 있다.
*함수 포인터는 반환형, 인자목록의 수와 각각의 자료형이 일치하는 함수의 주소를 저장할 수 있는 변수이다.
*함수 포인터를 선언하려면 함수원형에서 함수이름을 제외한 반환형과 인자목록의 정보가 필요하다.
변수이름이 pf인 함수 포인터를 하나 선언해 보자. 함수 포인터 pf는 함수원형이 void add(double*, double, double);인 함수의 주소를 저장하려고 한다. 이때 필요한 것이 함수원형에서 반환형인 void와 인자목록인 (double *, double, double) 정보이다.
*함수 포인터 pf는 문장 void (*pf)(double*, double, double); 로 선언될 수 있다.
*물론 인자목록에서 변수이름은 사용할 수도 있으나 생략하는 편이 더 간편하다.
*여기서 주의할 점은 (*pf)와 같이 변수이름인 pf 앞에는 *이 있어야 하며 반드시 괄호를 사용해야 한다는 것이다.
*만일 괄호가 없으면 pf는 함수 포인터 변수가 아니라 void *를 반환하는 함수이름이 되고, 이 문장은 함수원형이 되어 버린다.

```
//잘못된 함수 포인터 선언
void *pf(double*, double, double);    //함수원형이 되어 pf는 원래 함수이름

   void (*pf)(double*, double, double); //함수 포인터
   pf = add; //변수 pf에 함수 add의 주소값을 대입 가능
```

그림 14-21 함수 포인터 변수 선언과 대입

물론 위 함수 포인터 변수 pf는 함수 add()만을 가리킬 수 있는 것이 아니라 add()와 반환형과 인자목록이 같은 함수는 모두 가리킬 수 있다. 즉 subtract()의 반환형과 인자목록이 add()와 동일하 다면 pt는 subtract()도 가리킬 수 있다. 문장 pf = subtract; 와 같이 함수 포인터에는 괄호가 없이 함수이름만으로 대입해야 한다.

*함수 이름 add나 subtract는 주소 연산자를 함께 사용하여 &add나 &subtract로도 사용 가능하다.

*그러나 다음과 같이 subtract()와 add()와 같이 함수호출로 대입해서는 오류가 발생한다.

```
void (*pf2)(double *z, double x, double y) = add();     //오류발생
pf2 = subtract();                                       //오류발생
pf2 = add;                                              //가능
pf2 = &add;                                             //가능
pf2 = subtract;                                         //가능
pf2 = &subtract;                                        //가능
```

그림 14-22 함수 포인터 변수의 대입

=함수 포인터를 이용한 함수 호출

다음 예제에서 함수 add()의 구현을 살펴보자. 함수 add()에서 x + y의 결과를 반환하지 않고 포인 터 변수 z에 저장하고 있다. 이와 같이 함수 호출에서 함수 인자를 포인터 변수로 사용하면 함수 내 부에서 수정한 값이 그대로 실인자로 반영되는 것을 알 수 있다.

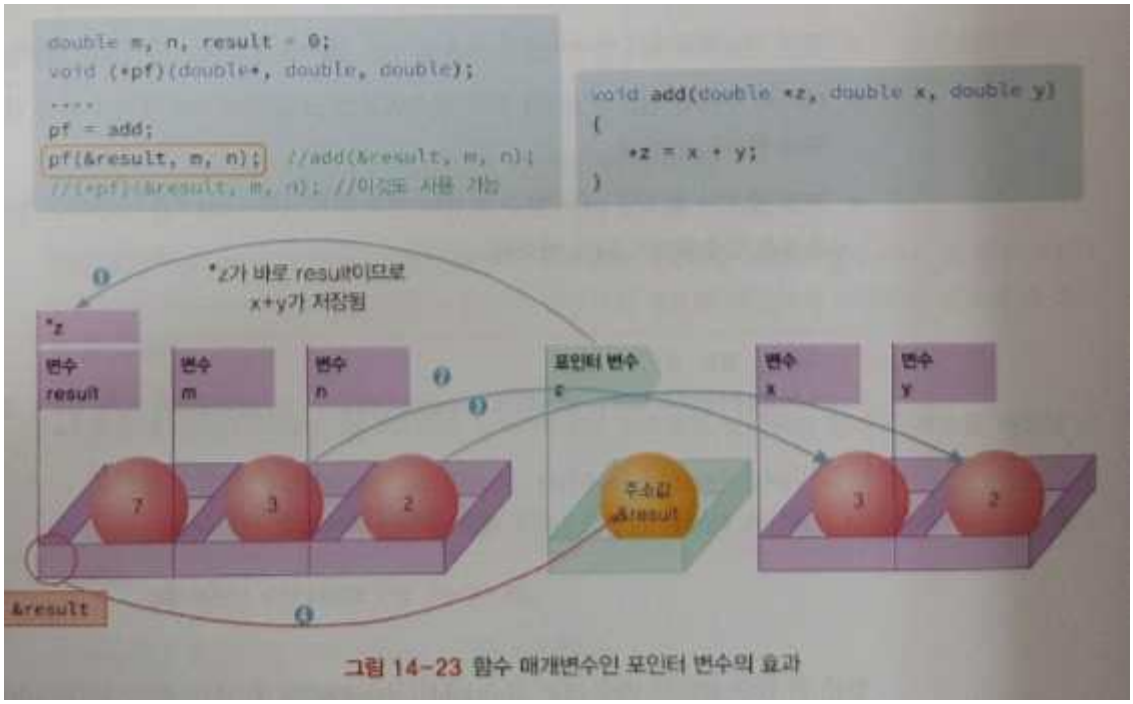


그림 14-23 함수 매개변수인 포인터 변수의 효과

문장 pf = add;로 함수 포인터 변수인 pf에 함수 add()의 주소값이 저장되면, 변수 pf를 이용하여 add() 함수를 호출할 수 있다.

*포인터 변수 pf를 이용한 함수 add()의 호출방법은 add() 호출과 동일하다. 즉 pf(&result, m, l n); 로 add(&result, m, n) 호출을 대체할 수 있다. 이 문장이 실행되면 변수 result에는 m + n

의 결과가 저장된다. 즉 함수 add()에서 m + n이 반영된 변수 result를 사용할 수 있다.

*함수 호출 pf(&result, m, n)은 (*pf)(&result, m, n)로도 가능하다. 반드시 *pf에 괄호를 넣은 (*pf)이 필요하다.

-함수 포인터 배열

=함수 포인터 배열 개념

포인터 배열과 같이 함수 포인터가 배열의 원소로 여러 개의 함수 포인터를 선언하는 함수 포인터 배열을 생각할 수 있다. 즉 함수 포인터 배열(array of function pointer)은 함수 포인터가 원소인 배열이다.

*다음 그림에서 크기가 3인 함수 포인터 배열 pfinary는 문장 int (*pfunary[3](int, int); 으로 선언된다.

*배열 pfunary의 각 원소가 가리키는 함수는 반환값이 int이고 인자목록이 (int, int)이어야 한다.

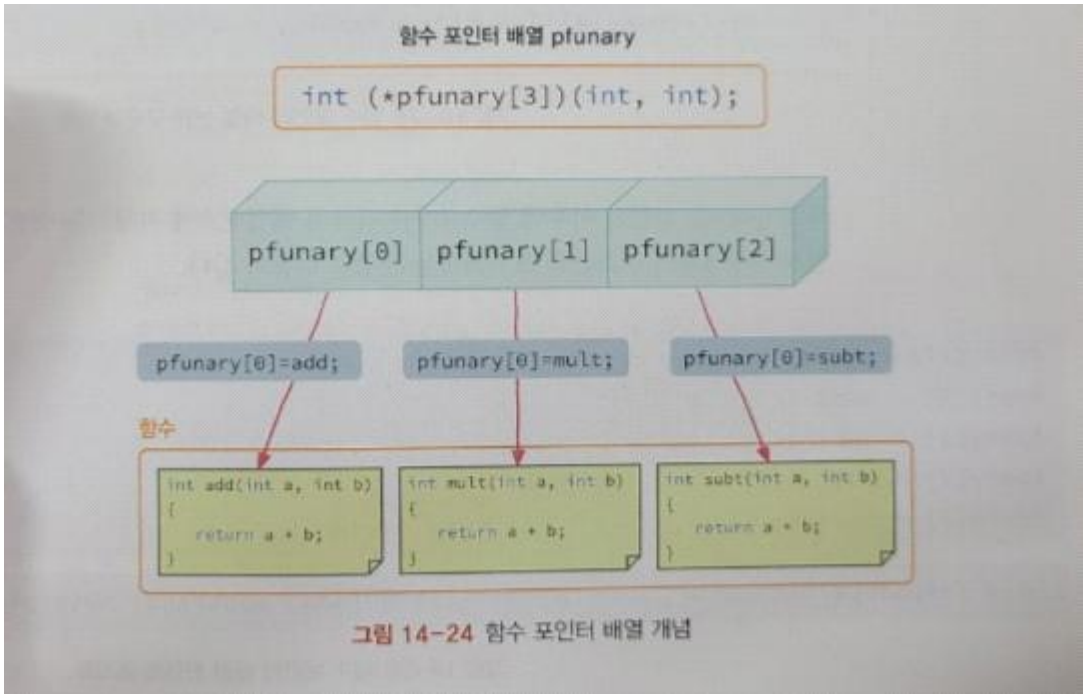


그림 14-24 함수 포인터 배열 개념

=함수 포인터 배열 선언

다음은 함수 포인터 배열선언 구문을 나타낸다. 다음 부분 소스의 배열 fpary의 각 원소가 가리키는 함수는 반환값이 void이고 인자목록이 (double*, double, double)인 경우의 배열 선언이다. 배열 fpary을 선언한 이후에 함수 4개를 각각의 배열원소에 저장하는 방법과 배열 fpary을 선언하 면서 함수 4개의 주소값을 초기화하는 문장은 다음과 같다.

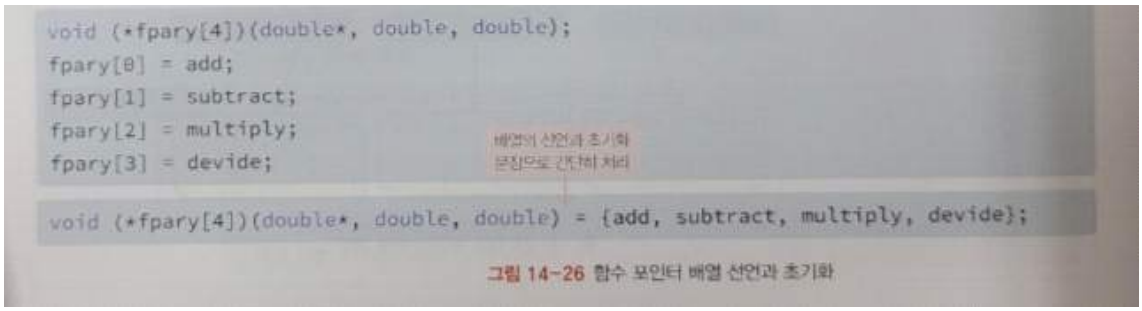


그림 14-26 함수 포인터 배열 선언과 초기화

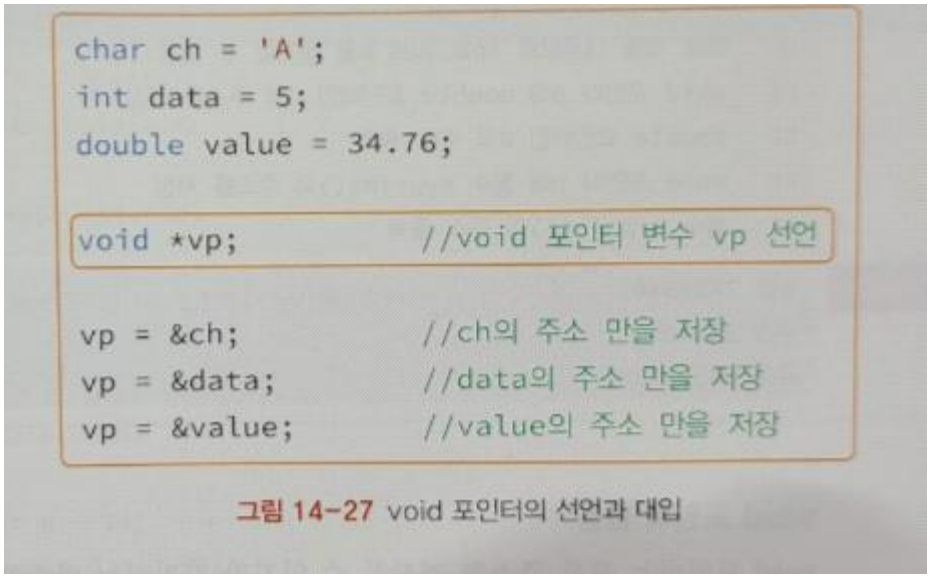
-void 포인터

=void 포인터 개념

포인터는 주소값을 저장하는 변수인 int *, double * 처럼 가리키는 대상의 구체적인 자료형의 포인터로 사용하는 것이 일반적이다. 주소값이란 참조를 시작하는 주소에 불과하며 자료형을 알아야 참조할 범위와 내용을 해석할 방법을 알 수 있는 것이다. 그렇다면 void 포인터(void *)는 무엇일까?

*바로 void 포인터는 자료형을 무시하고 주소값만을 다루는 포인터이다. 그러므로 포인터 void 포인터는 대상에 상관없이 모든 자료형의 주소를 저장할 수 있는 만능 포인터로 사용할 수 있다.

*void 포인터에는 일반 변수 포인터는 물론 배열과 구조체 심지어 함수 주소도 담을 수 있다.



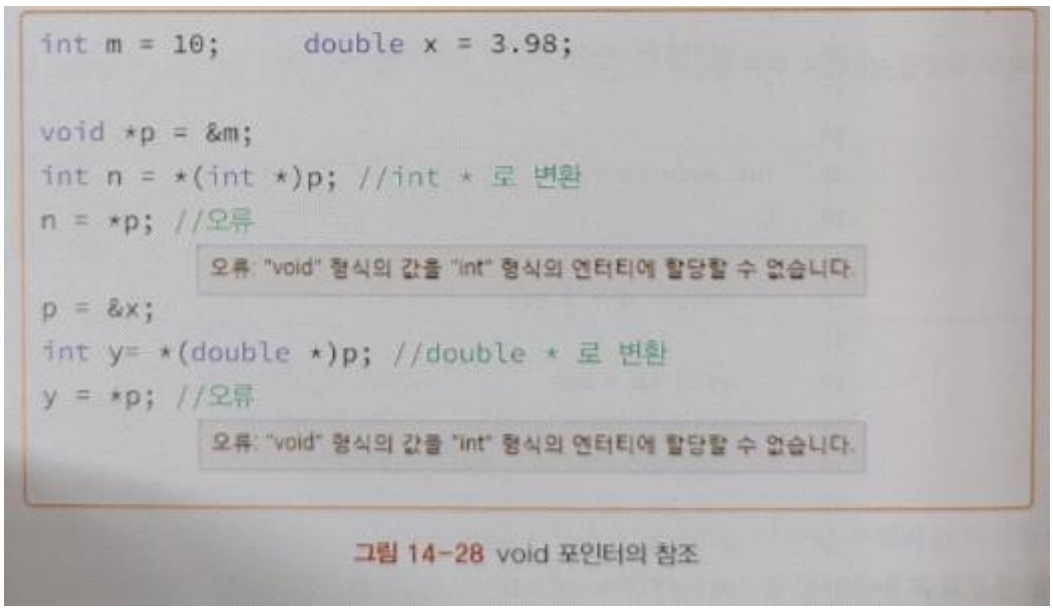
```
char ch = 'A';
int data = 5;
double value = 34.76;

void *vp;              //void 포인터 변수 vp 선언

vp = &ch;              //ch의 주소 만을 저장
vp = &data;            //data의 주소 만을 저장
vp = &value;           //value의 주소 만을 저장
```

그림 14-27 void 포인터의 선언과 대입

=void 포인터 활용

void 포인터는 모든 주소를 저장할 수 있지만 가리키는 변수를 참조하거나 수정이 불가능하다. void 포인터로 참조 변수를 참조할 수 없는 것은 당연한 일이다. 주소값으로 변수를 참조하려면 결 국 자료형으로 참조범위를 알아야 하는데 void 포인터는 이러한 정보가 전혀 없이 주소 만을 담는 변수에 불과하기 때문이다.

*void 포인터는 자료형 정보는 없이 임시로 주소 만을 저장하는 포인터이다. 그러므로 실제 void 포인터로 변수를 참조하기 위해서는 자료형 변환이 필요하다.



```
int m = 10;     double x = 3.98;

void *p = &m;
int n = *(int *)p; //int * 로 변환
n = *p; //오류
        오류: "void" 형식의 값을 "int" 형식의 엔터티에 할당할 수 없습니다.
p = &x;
int y= *(double *)p; //double * 로 변환
y = *p; //오류
        오류: "void" 형식의 값을 "int" 형식의 엔터티에 할당할 수 없습니다.
```

그림 14-28 void 포인터의 참조

15장. 파일처리

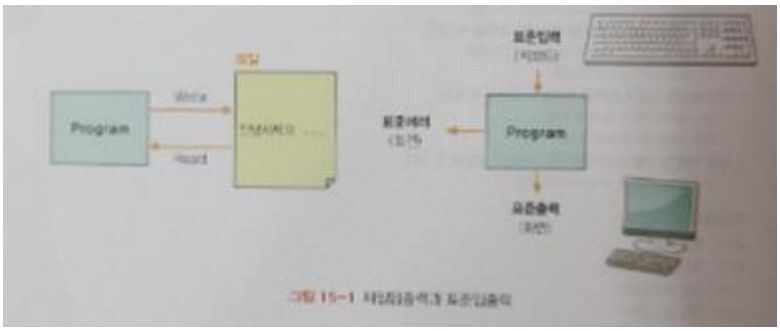
1. 15-1 파일 기초

-텍스트 파일과 이진 파일

=파일의 필요성

일상 생활에서 자주 이용하는 한글 또는 워드와 같은 워드프로세서는 어떻게 개발하였을까? 지금까지 메모장이나 워드프로세서로만 파일을 만들었을 것이다. 이러한 워드프로세서 없이 프로그램에서 프로그램의 결과로 구성되는 파일을 직접 만들 수 있을까? 또한 이미 있는 파일의 내용을 읽을 수 있을까? 이런 의문이 있었다면 이 단원으로 이제 그 의문이 어느 정도 풀릴 것이다.

지금까지 프로그램의 출력은 콘솔에서, 입력은 키보드에서 수행하였다. 만일 프로그램에서 출력을 파일에 한다면 파일이 생성되는 것이다. 반대로 키보드에서 표준 입력하던 입력을 파일에서 입력하면 파일 입력이 된다.



변수와 같이 프로그램에서 내부에서 할당되어 사용되는 주기억장치의 메모리 공간은 프로그램이 종료되면 모두 사라진다. 그러나 보조기억장치인 디스크에 저장되는 파일(file)은 직접 삭제하지 않은 한 프로그램이 종료되더라도 계속 저장할 수 있다. 그러므로 프로그램에서 사용하던 정보를 종료 후에도 계속 사용하고 싶다면 프로그램에서 파일에 그 내용을 저장해야 한다. 한 예로 학생 성적 처리를 프로그램을 통하여 결과를 얻어내 그 처리 결과를 지속적으로 저장하려면 파일에 저장해야 할 것이다.

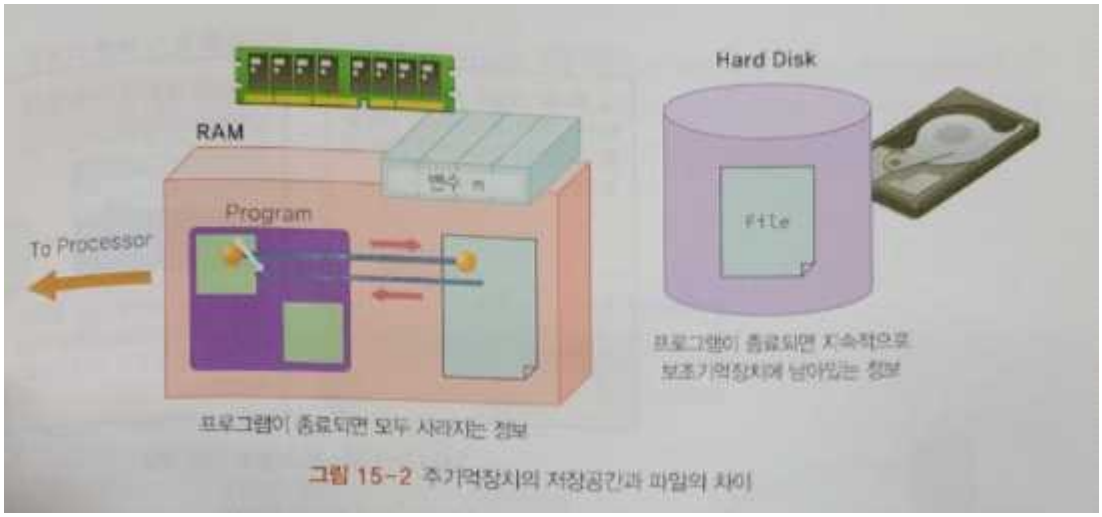


그림 15-2 주기억장치의 저장공간과 파일의 차이

=텍스트 파일과 이진파일

파일은 보조기억장치의 정보저장 단위로 자료의 집합이다. 파일은 텍스트 파일(text file)과 이진 파일(binary file) 두 가지 유형으로 나뉜다. 대표적인 텍스트 파일은 메모장(notepad) 같은 편집기로 작성된 파일이며, 이진 파일은 실행파일과 그림 파일, 음악 파일, 동영상 파일 등을 예로 들 수 있다.

*텍스트 파일은 문자 기반의 파일로서 내용이 아스키 코드(ascii code)와 같은 문자 코드값으로 저장된다.

* 메모리에 저장된 실수와 정수와 같은 내용도 텍스트 파일에 저장될 때는 문자 형식으로 변환되어 저장된다. 그러므로 텍스트 파일은 텍스트 편집기를 통하여 그 내용을 볼 수 있고 수정할 수도 있다.

이진 파일은 텍스트 파일과 다르게 그림 파일, 동영상 파일, 실행 파일과 같이 각각의 목적에 알맞은 자료가 이진 형태(binary format)로 저장되는 파일이다. 또한 C 프로그램 내부에서 관리하던 변수의 내용을 그대로 이진 형태로 파일에 저장하면 이진 파일이 된다.

*이진 파일은 컴퓨터 내부 형식으로 저장되는 파일이다.

* 이진 파일에서 자료는 메모리 자료 내용에서 어떤 변환도 거치지 않고 그대로 파일에 기록된다. 그러므로 입출력 속도도 텍스트 파일에 비해 빠르다.

*이러한 이진 파일은 메모장과 같은 텍스트 편집기로는 그 내용을 볼 수 없다. 이진 파일의 자는 그 내용을 이미 알고 있는 특정한 프로그램에 의해 인지될 때 의미가 있다.

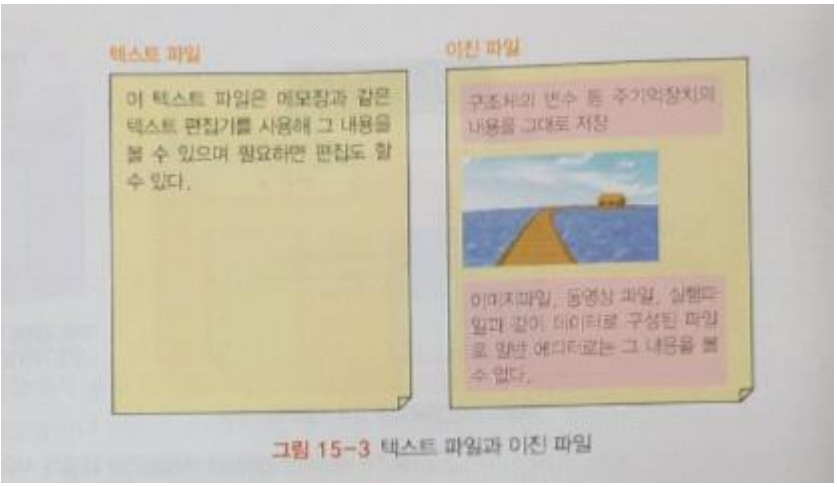


그림 15-3 텍스트 파일과 이진 파일

=입출력 스트림

자료의 입력과 출력은 자료의 이동이라고 볼 수 있으며 자료가 이동하려면 이동 경로가 필요하다. 산속의 물이 개울가(stream)를 통해 하천으로 내려오듯이 자료도 이동 경로인 스트림(stream)이 연결되어야 이동할 수 있다. 입출력 시 이동 통로가 바로 입출력 스트림(io stream)이다.
키보드에서 프로그램으로 자료가 이동하는 경로가 바로 표준입력 스트림이며 함수 scanf()는 바로 표준 입력 스트림에서 자료를 읽을 수 있는 함수이다. 반대로 프로그램에서 모니터의 콘솔로 자료가 이동하는 경로가 표준출력 스트림이며 함수 printf()는 바로 표준출력 스트림으로 자료를 보낼 수 있는 함수이다.

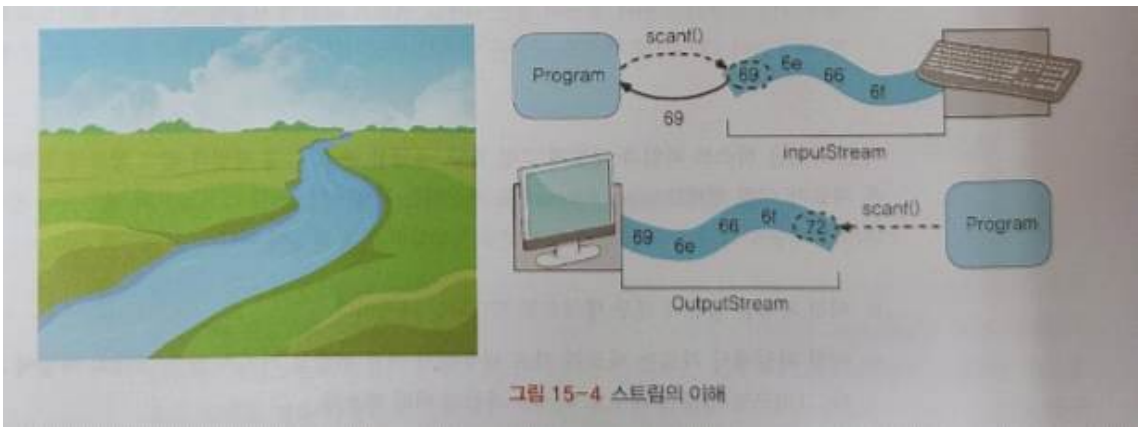
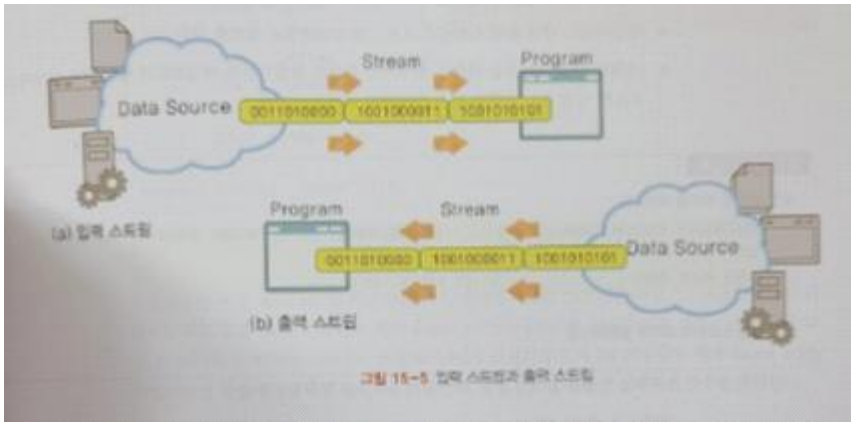


그림 15-4 스트림의 이해

다른 곳에서 프로그램으로 들어오는 경로가 입력 스트림(input stream)이며, 자료가 떠나는 시작 부분이 자료 원천부(data source)로 이 부분이 키보드이면 표준입력이며, 파일이면 파일로부터 자료를 읽는 것이며, 터치스크린이면 스크린에서 터치 정보를 알 수 있고, 네트워크이면 다른 곳에서 프로그램으로 네트워크를 통해 자료가 전달되는 것이다. 반대로 프로그램에서 다른 곳으로 나가는경로가 출력 스트림(input stream)이며, 자료의 도착 장소가 자료 목적부(data destination)로 이 부분이 콘솔이면 표준출력이며, 파일이면 파일에 원하는 값을 쓸 수 있으며, 프린터이면 프린터에 출력물이 나오고, 네트워크이면 네트워크 출력이 되어 다른 곳으로 자료가 이동되는 것이다.



=파일 스트림 이해

프로그램에서 보조기억장치에 파일로 정보를 저장하거나 파일에서 정보를 참조하려면 파일(file) 에 대한 파일 스트림(file stream)을 먼저 연결해야 한다. 파일 스트림이란 보조기억장치의 파일과 프로그램을 연결하는 전송경로이다.

2. 15-2 텍스트 파일 입출력

파일에 서식화된 문자열 입출력

함수 fprintf()와 fscanf()

서식화된 자료의 파일 입출력 함수를 알아보자. 텍스트 파일에 자료를 쓰거나 읽기 위하여 함수 fprintf()와 fscanf() 또는 fscanf_s()를 이용한다. 이들 함수를 이용하기 위해서는 헤더 파일 stdio. h를 포함해야 한다. 함수 fprintf()와 fscanf(), fscanf s()의 함수 원형을 살펴보면 다음과 같다. 현 재 Visual C++에서 함수 fscanf()는 함수 fscanf s()로 대체되어 fscanf s() 사용을

권장하고 있다.
위 함수들의 첫 번째 인자는 입출력에 이용될 파일이고, 두 번째 인자는 입출력에 이용되는 제어 문자열이며, 다음 인자들은 입출력될 변수 또는 상수 목록이다. 함수 원형에서 기호 ...은 인자 수가 정해지지 않은 다중 인자임을 의미한다. 또한 함수 fprintf()와 fscanf() 또는 fscanf s()의 첫 번 째 인자에 각각 stdin 또는 stdout를 이용하면 표준 입력, 표준 출력으로 이용이 가능하다.
o fprintf(stdout, "제어문자열", ...)은 printf("제어문자열", ...)와 같이 표준출력 기능을 수행한다.
*기호 상수 stdin, stdoute stderr과 함께 헤더 파일 stdio.h에 정의되어 있는 값으로 각각 표준 입력, 표준출력, 표준에러를 의미한다.
*실제로 이 stdin, stdout, stderr은 파일을 가리키는 포인터로 C 언어가 제공하는 표준파일이라 한다.

3. 15-3 이진 파일 입출력
텍스트와 이진 파일 입력과 출력
함수 fprinf()와 fsanf_s()
함수 fprintf()와 fscanf(), fscanf_s()는 자료의 입출력을 텍스트 모드로 처리한다. 함수 fprintf()에 의해 출력된 텍스트 파일은 텍스트 편집기로 그 내용을 볼 수 있으며, 텍스트 파일의 내용은 모두 지 정된 아스키 코드와 같은 문자 코드값을 갖고 있어 그 내용을 확인할 수 있을 뿐만 아니라 인쇄할 수 있다. 함수 fprintf()를 이용하여 int 형 변수 cnt의 값을 파일 f에 출력하는 과정을 살펴보자.
함수 fprintf()로 실제 정수 int형 변수 cnt를 출력하면, 실제로 파일에 저장되는 자료는 정수값 10 에 해당하는 각 문자의 아스키 값이다 그러므로 파일 f에는 정수값 10에 해당하는 각각의 문자 1 과 '0'의 아스키 코드값이 저장된다.
함수 fwrite()와 fread() 텍스트 파일과는 다르게 이진파일(binary file)은 C 언어의 자료형을 모두 유지하면서 바이트 단 위로 저장되는 파일이다. 이진 모드로 블록 단위 입출력을 처리하려면 함수 fwrite()와 fread() 르 이용한다. 함수 fwrite()와 fread()는 헤더파일 stdio.h에 다음과 같은 함수원형으로 정의되어 있다