|  |  |
| --- | --- |
| **Program / Project Name:** | Programming |
| **Checklist / Template Completed by:** | TBD |
| **Date Completed:** | Click here to enter a date. |

**Programming Guide**

**C**

Revision: 0.1

**Detailed Revision History**

|  |  |  |  |
| --- | --- | --- | --- |
| **Rev** | **Date** | **Editor(s)** | **Description of change** |
| 0.1 | 09-JUN-2019 | Sang-Gu Kang | Initial Draft |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Table of Contents**

[1 API Library 4](#_Toc11339749)

[1.1 문자열 처리 함수 4](#_Toc11339750)

[1.1.1 strcpy vs strcpy\_s 4](#_Toc11339751)

[2 수식과 입출력 4](#_Toc11339752)

[2.1 Type Cast 4](#_Toc11339753)

[2.1.1 static\_cast<type>(value) 4](#_Toc11339754)

[2.2 Type int 4](#_Toc11339755)

[2.2.1 서식 지정자 5](#_Toc11339756)

[2.2.2 min/max 6](#_Toc11339757)

[2.3 Literal 접미사 – 정수 6](#_Toc11339758)

[2.4 Type float double 7](#_Toc11339759)

[3 Header file 7](#_Toc11339760)

[3.1 stdint.h 7](#_Toc11339761)

# API Library

## 문자열 처리 함수

### strcpy vs strcpy\_s

가급적 strcpy\_s, sprintf\_s, strcat\_s와 같은 안전한 스트링 함수를 사용해야 한다. Visual studio에서 strcpy와 같은 함수를 사용하면 error로 처리된다. 이 함수들의 원형을 살펴보면 두 번째 인자가 size\_t numberOfElements임에 유의한다. Sizeof(buffer)가 아니라 \_countof(buffer)로 계산해야 한다.

errno\_t strcpy\_s(char \*strDest, size\_t numberOfElements, const char \*strSrc);

안전한 스트링 함수는 strDest 버퍼를 넘어서는 메모리 복사를 방지하는 함수이다. 기존의 strcpy같은 함수를 사용할 때에는, 잘못된 메모리 할당으로 어느 순간에는 문제가 없는 것처럼 돌아가다가 나중에 다른 곳에서 메모리 할당시 엉뚱하게 문제가 발생할 수 있고, 이는 디버깅 자체를 어렵게 만든다. strcpy\_s 함수는 메모리 에러를 발생시켜 주므로 메모리를 안정적으로 사용할 수 있도록 해준다.



# 수식과 입출력

## Type Cast

### static\_cast<type>(value)

형변환은 위와 같은 형식으로 엄격히 적용하는 것이 좋다. 이유는 인터넷을 참고한다.

### 자료형의 확장

실제로 프로그래밍을 할 때는 서로 다른 자료형으로 연산을 할 때가 많다. 정수와 실수를 함께 연산하면 결과값은 실수가 된다. 실수가 정수보다 표현 범위가 넓기 때문이다.

C언어에서는 자료형을 섞어서 쓰면 컴파일러에서 암시적 형 변환(implicit type conversion)을 하게 되는데 자료형의 크기가 큰 쪽, 표현 범위가 넓은 쪽으로 자동 변환된다. 이를 형 확장(type promotion)이라고 하며 값이 버려지지 않고 그대로 보전된다.

### 자료형의 축소

자료형의 크기가 작은 쪽, 표현 범위가 좁은 쪽으로 변환되는 것을 형 축소(type demotion)라고 한다. 형 축소가 일어나면 값의 손실이 일어날 수 있다. 이를 막기 위해 형 변환(type conversion, type casting)을 해야 한다.

## Type int

정수형 자료형은 크게 char, int가 있으며 앞에 부호 키워드(signed, unsigned)와 크기(short, long)를 붙여서 특성을 정의할 수 있다.



### 서식 지정자

printf 함수에서 자료형에 따라 서식 지정자를 달리 해야 한다.

* char, short, int: %d
* long: %ld
* long long: %lld
* unsigned char, unsigned short: %d
* unsigned int: %u
* unsinged long: %lu
* unsigned long long: %llu

%d: decimal  
%ld: long decimal  
%lld: long long decimal  
%u: unsigned decimal  
%lu: long unsigned decimal  
%llu: long long unsigned decimal

%p: pointer (memory address of variable)  
%x: hexadecimal

### min/max

int의 최소값은 -2,147,483,648로 되어 있지만, visual studio나 codeblock에서 이 값을 직접 넣어보면 waringing 혹은 error가 발생한다. 소스코드에서 정수의 최소값을 표현하려면 limits.h 헤더 파일을 사용해야 한다.



값을 계산하다가 오버플로우나 언더플로우 현상이 발생하면 의도치 않은 결과가 나올 수 있다. 따라서 프로그래밍할 때에는 정수 자료형의 크기를 항상 생각하고, 값이 범위를 넘어서지는 않는지 반드시 확인해야 한다.

## Literal 접미사 – 정수

컴파일 경고를 방지하려면 아래와 같이 뒤에 자료형 크기에 맞는 정수 리터럴 접미사를 붙여주면 된다.

unsigned int num3 = 4123456789U;

## Type float double



유효자릿수는 실수를 일정 자릿수만큼만 표현할 수 있다는 뜻이다. 만약 유효자릿수가 7이라면 0.123456789는 반올림하여 0.1234567로 표시된다. 즉, 정수 부분 1자리와 소수 부분 6자리로 7자리가 표시된다.

Note) Long double 은 잘 동작하지 않는 것 같음.

float에는 f를 붙여야 한다. Double은 생략, long double에는 l을 붙인다.  
예) float num1 = 2.97; (X) float num1 = 2.97f;

### min/max overflow/underflow

FLT\_MIN을 100000000.0과 같이 큰 수로 나누면 아주 작은 수가 되면서 언더플로우가 발생하는데 C언어에서는 실수 언더플로우를 0 또는 쓰레기 값으로 처리한다. 반대로 FLT\_MAX에 1000.0을 곱하면 저장할 수 있는 범위를 넘어서기 때문에 오버플로우가 발생한다. 정수와는 달리 실수는 오버플로우가 발생했을 때 최소값으로 되돌아가지 않고 무한대(infinity)가 되므로 inf가 출력된다.

우리는 실생활에서 1과 1.0을 구분하지 않지만 컴퓨터는 1과 1.0을 구분한다.

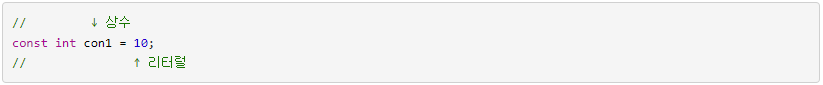
## Type char

정수 자료형인 char를 이용하여 문자 한 개를 저장한다. char에 문자를 저장할 때는 문자 자체를 저장하는 것이 아니라 문자에 해당하는 정숫값을 저장하게 된다. 이 규칙을 ASCII코드 라고 부른다.

## 상수

상수는 처음 선언할 때만 값을 할당할 수 있으며 그 다음부터는 값을 바꿀 수 없다.

상수와 리터럴을 구분해 볼 필요가 있는데, 다음과 같은 코드가 있을 때 con1은 상수, 10은 리터럴이다.



리터럴(literal)은 ‘문자 그대로’라는 뜻인데 C언어에서는 값 그 자체를 뜻한다. 상수(constant)는 변수처럼 리터럴이 저장된 공간이다.

상수는 프로그램을 작성할 때 고정되어 있어야 할 값을 변수에 저장해서 사용하다 보면 실수로 값을 바꿀 가능성이 있는데 이것을 방지하기 위해 사용하며 코드의 의도를 명확하게 만들 수 있다.

const는 자료형 앞에 붙여도 되고 자료형 뒤에 붙여도 된다.

## 절대값

abs, fabs, fabsf 함수를 사용하면 양수 또는 음수를 절대값(absolute value)으로 만들 수 있다. 이 함수도 math.h 헤더 파일에 선언되어 있다.  
-. abs(정수);  
 int abs(int \_X);  
 정수 절대값을 반환  
-. fabs(double형 실수);  
 double fabs(double \_X);  
 double형 실수 절대값을 반환  
-. fabsf(float형 실수);  
 float fabsf(float \_X);  
 float형 실수 절대값을 반환

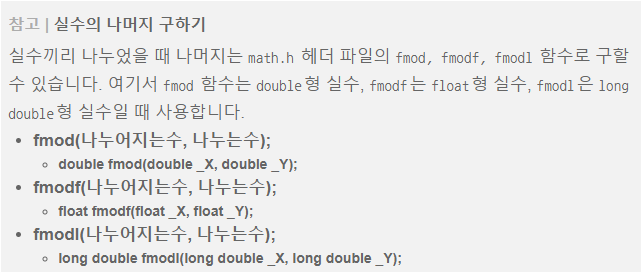
## 곱하기, 나누기, 나머지

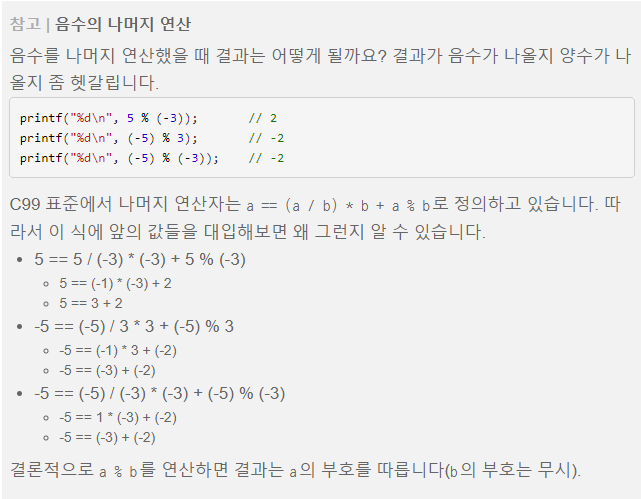
변수에 정수와 0을 저장해서 나누면 컴파일 에러는 발생하지 않지만 실행시 에러가 발생한다. 실무에서 정수를 0으로 나누는 버그는 생각보다 많이 나오므로 주의해야 한다.

실수 계산에서는 계산 결과에 오차가 발생한다. 예를 들어 2.73에 3.81을 곱하면 10.4013이 나와야 하는데 10.401299가 나온다.

실수도 소스 코드에서 0으로 직접 나누면 에러가 발생한다. 하지만 변수에 0.0을 저장해서 나누면 결과가 무한대(inf)로 나온다.  


나머지 연산은 정수에서만 사용할 수 있고 실수에서는 사용할 수 없다(컴파일 에러 발생). 또한, 7 % 0처럼 0으로 나눈 결과의 나머지는 구할 수 없다.



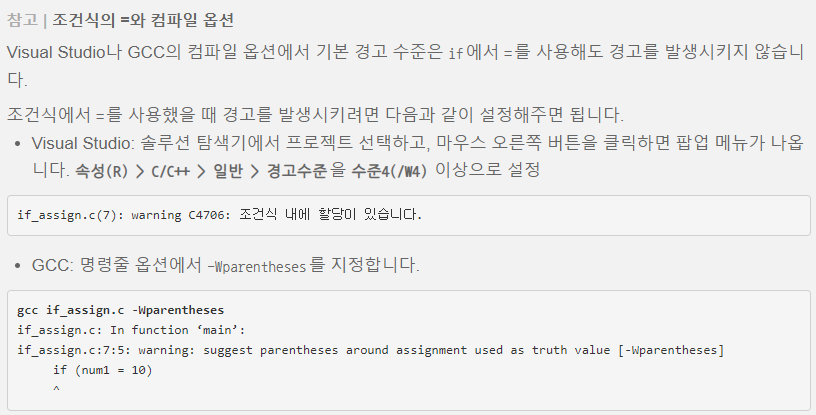


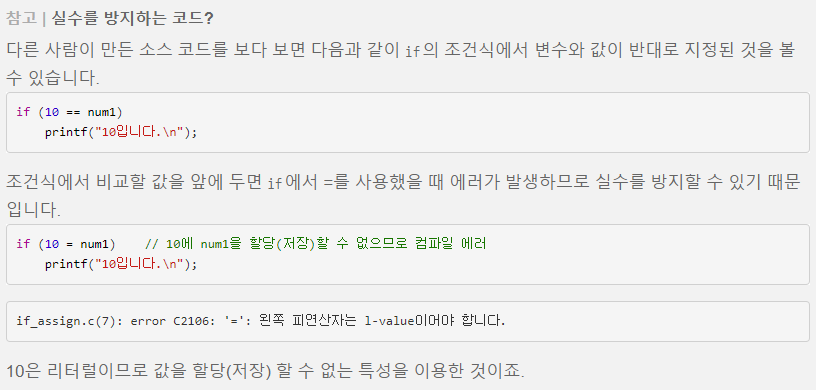
## If 조건

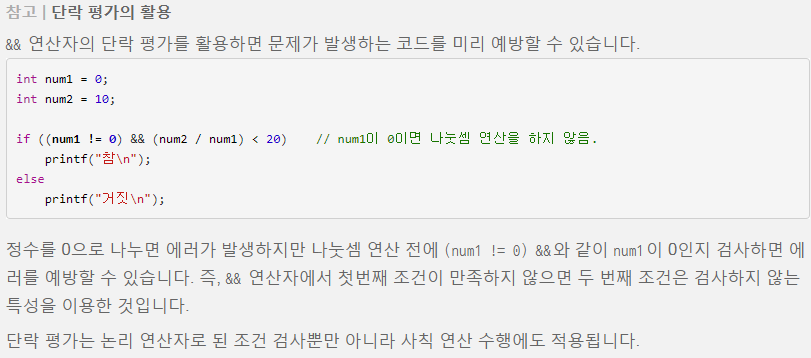
실수는 연산한 뒤 반올림 오차가 발생할 수 있으므로 반올림 오차를 감안하여 비교한다. 문자 자료형은 c1 == ‘a’처럼 문자로 비교하거나, c1 == 97처럼 ASCII 코드로 비교해도 된다.

세 가지 조건을 판단할 때  
If ( ( year % 4 == 0) && (year % 100 != 0) || (year % 400 == 0 )

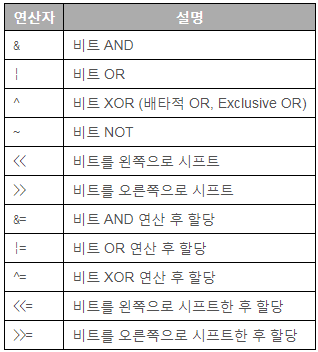
실수와 실수를 비교할 때 다음과 같은 문장을 사용하는 것은 문제가 될 수 있다.  
if (result == expectedResult ) // 0.2와 같은 단순한 값은 정확하게 표현되지만 복잡한 값은 정확하게 표현되지 않기 때문. 따라서 아래와 같이 오차를 감안해서 비교한다.  
if ( fabs(result – expectedResult) < 0.00001)



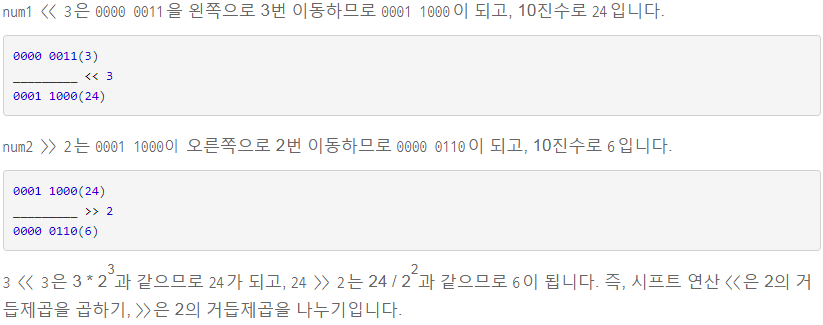




## 비트 연산자



비트 연산자는 비트로 옵션을 설정할 때 주로 사용하며 저장 공간을 아낄 수 있는 장점이 있다. 특히 이런 방식을 플래그(flag)라고 부른다.



부호 있는 자료형에 저장된 값을 shift하게 되면 모자라는 공간은 모두 부호 비트의 값으로 채워지게 된다.



# Pointer

변수의 메모리 주소를 구해보았는데 메모리 주소는 어디에 저장해야 할까? C언어에서 메모리 주소는 포인터(pointer) 변수에 저장한다.

포인터 변수를 선언할 때는 자료형 뒤에 \*(asterisk)를 붙인다. \*의 위치에 따른 차이는 없으며 모두 같은 뜻이다.

포인터의 크기를 sizeof로 구해보면 32비트에서는 4byte, 64비트에서는 8byte가 나온다.

## 역참조 연산자(dereference)

포인터 변수에는 메모리 주소가 저장되어 있다. 이때 메모리 주소가 있는 곳으로 이동해서 값을 가져오고 싶다면 역참조(dereference) 연산자 \*를 사용한다.

포인터를 선언할 때도 \*를 사용하고 역참조를 할 때도 \*를 사용한다. 같은 \* 기호를 사용해서 헷갈리기 쉽지만 선언과 사용을 구분해서 생각하면 된다. 즉, 포인터를 선언할 때 \*는 “이 변수가 포인터 변수이다”라고 알려주는 역할이고, 포인터에 사용할 때 \*는 “포인터의 메모리 주소를 역참조하겠다”라는 뜻이다.



역참조 연산자는 자료형을 바꾸는 효과를 낸다. 즉, int \*numPtr;에서 \*numPtr처럼 역참조하면 int 가 된다.



## 다양한 자료형의 포인터 선언

C언어에서 사용할 수 있는 모든 자료형은 포인터로 만들 수 있다.

### 상수와 포인터

포인터에도 const 키워드를 붙일 수 있는데 const의 위치에 따라 특성이 달라진다.

#### 상수를 가리키는 포인터 (pointer to constant)



num1이 const int 이므로 이 변수의 주소를 넣을 수 있는 포인터는 cons tint \* 로 선언해야 한다. Pointer to constant는 메모리 주소에 저장된 값을 변경할 수 없다는 뜻이다.

#### 포인터 자체가 상수 (constant pointer)

이때는 \* 뒤에 const를 붙인다.



numPtr은 포인터 자체가 상수이므로 다른 포인터(메모리 주소)를 할당할 수 없다. 즉, constant pointer는 메모리 주소를 변경할 수 없다는 뜻이다.

#### 포인터가 상수이면서 상수를 가리키는 상황 (constant pointer to constant)

이때는 포인터를 선언하는 자료형에도 const를 붙이고 \* 뒤에도 const를 붙인다.



constant pointer to constant는 메모리 주소도 변경할 수 없고 메모리 주소에 저장된 값도 변경할 수 없다는 뜻이다.

## void 포인터

자료형이 정해지지 않은 포인터로 void 키워드와 \* 로 선언한다. 직접 자료형을 변환하지 않아도 암시적으로 자료형이 변환되는 방식이다. 단, void 포인터는 자료형이 정해지지 않았으므로 역참조를 할 수 없다.

역참조도 할 수 없는 void 포인터는 함수에서 다양한 자료형을 받아들일 때, 함수의 반환 포인터를 다양한 자료형으로 된 포인터에 저장할 때, 자료형을 숨기고 싶을 때 다양한 형태로 사용되고 있다.

## Memory

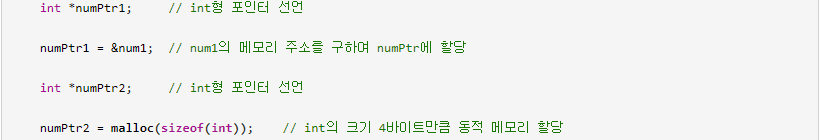
포인터에 변수의 메모리 주소를 저장하는 방식으로 포인터를 사용했다면, 포인터에 원하는 만큼 메모리 공간을 할당받아 사용하는 방법을 이해한다.

메모리는 malloc -> 사용 -> free 패턴으로 사용한다.

메모리를 사용하려면 malloc 함수로 사용할 메모리 공간을 확보해야 한다(memory allocation). 이때 필요한 메모리 크기는 바이트 단위로 지정한다. 메모리 할당, 해제 함수는 stdlib.h 헤더 파일에 선언되어 있다.

포인터 = malloc(크기)  
-. void \* malloc(size\_t size);  
-. 성공하면 메모리 주소를 반환, 실패하면 NULL을 반환

원하는 시점에 원하는 만큼 메모리를 할당할 수 있다고 하여 동적 메모리 할당이라 부른다.(dynamic memory allocation)  


아래에서 numPtr1에는 일반 변수의 메모리 주소를 할당했고, numPtr2에는 malloc 함수로 메모리를 할당했다. 변수는 스택(stack)에 생성되며 malloc 함수는 힙(heap) 부분의 메모리를 사용한다.  
  
스택과 힙의 가장 큰 차이점은 메모리 해제이다. 스택에 생성된 변수는 사용한 뒤 따로 처리를 해주지 않아도 되지만 malloc 함수를 사용하여 힙에서 할당한 메모리는 반드시 해제를 해 주어야 한다.

malloc 함수로 할당한 메모리에 값을 저장할 때는 \*numPtr = 10; 처럼 포인터를 역참조한 뒤 값을 저장하면 된다.

다음과 같이 malloc 함수에 sizeof(int) \* 1024 \* 1024 \* 1024를 넣으면 어떻게 될까?  
  
문제없이 할당이 된다. 계산을 해 보면 4294967296인데 4GB이다. 더 큰 메모리도 할당 가능하지만 실제로 값을 저장할 때는 시스템의 한계(RAM, 디스크) 이상 저장할 수 없다. 할당은 가능하다.

memset 함수를 사용하면 메모리의 내용을 원하는 크기만큼 특정 값으로 설정할 수 있다.(memory set) string.h

memset 함수는 주로 다음과 같이 설정할 값을 0으로 지정하여 메모리의 내용을 모두 0으로 만들 때 주로 사용한다.  


NULL이 들어 있는 포인터를 널 포인터(null pointer)라고 하며 아무것도 가리키지 않는 상태를 뜻한다. 따라서 역참조는 할 수 없다.  


실무에서는 다음과 같이 포인터가 NULL인지 아닌지 확인한 뒤 NULL이면 메모리를 할당하는 패턴을 주로 사용한다.  

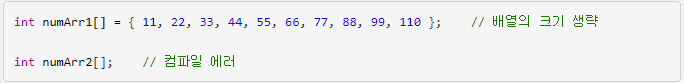

## 배열



{ }를 사용하여 배열에 값을 할당하는 방법은 배열을 선언할 때만 사용할 수 있으며 이미 선언된 배열에는 사용할 수 없다. 배열의 인덱스는 항상 0부터 시작한다.

배열을 선언할 때 [ ]는 “이 변수가 배열이고 크기는 얼마다”라고 알려주는 역할이고, 배열을 사용할 때 [ ]는 “배열의 요소에 접근하겠다”는 뜻이다.

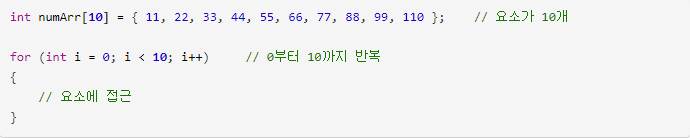
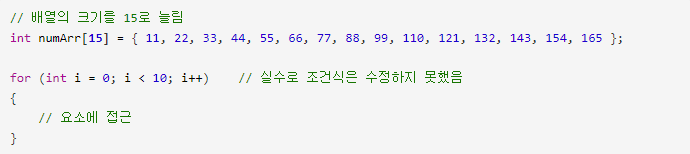
배열의 인덱스가 0부터 시작하는 이유는 메모리 주소가 0부터 시작하기 때문이다. 배열도 포인터이므로 인덱스가 0부터 시작하면 요소 접근과 포인터 연산이 일치하게 된다.

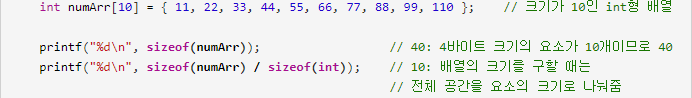
  
배열을 선언할 때 값을 초기화한다면 배열의 크기를 생략할 수 있다. 단, 초기화를 하지 않을 때는 생략할 수 없다.

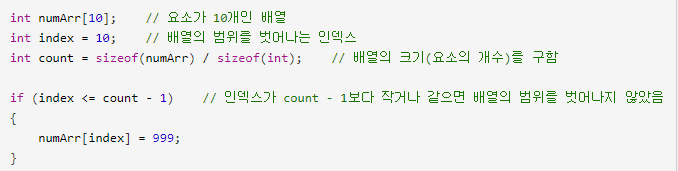
배열의 요소에 접근할 때 인덱스로 음수를 지정하거나, 배열의 크기를 벗어난 인덱스를 지정해도 컴파일 에러가 발생하지 않는다. 실행을 해보면 쓰레기값이 출력된다. 즉, 배열의 범위를 벗어난 인덱스에 접근하면 배열이 아닌 다른 메모리 공간에 접근하게 된다.

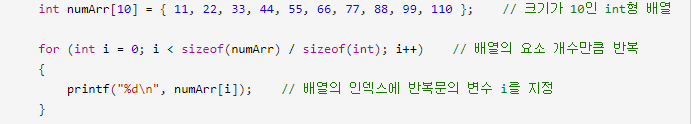
만약 배열의 범위를 벗어난 접근을 하여 값을 할당해버리면 엉뚱한 메모리에 값을 저장하게 되어 프로그램이 정상적으로 실행되지 않을 수 있으므로 주의해야 한다.

### 배열의 크기 (반복문에 자동 적용)

반복문으로 크기가 10인 배열의 요소를 모두 출력한다면,  
  
배열의 크기를 15개로 늘려야 하는 상황이 발생했으나 실수로 반복문의 조건식을 수정하지 못한다면,  
  
수정할 곳이 여러 개로 늘어나면 실수할 가능성은 그만큼 높아진다. 실수를 방지하기 위해서는 배열의 크기가 바뀌었을 때 배열의 크기를 알아서 계산하도록 만들면 된다.

배열이 차지하는 전체 공간과 요소의 개수는 sizeof 연산자를 활용하면 간단하게 구할 수 있다.  


프로그래머는 항상 인덱스가 배열의 범위를 벗어났는지 주의하면서 코딩을 해야 한다. 배열의 크기(요소 개수)를 구해 놓고, 배열에 접근하기 전에 인덱스가 요소 개수 -1을 넘지 않는지 확인하는 것도 좋은 방법이다.  


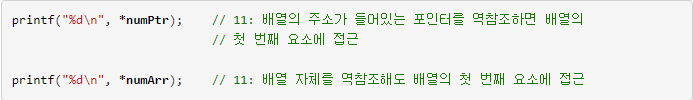
반복문을 사용하여 배열의 요소를 모두 출력하면 다음과 같다.  


### 배열과 포인터

배열은(배열 이름) 첫 번째 요소의 주소값을 담고 있다. 즉, 배열은 주소값이기 때문에 포인터에 넣을 수 있다. 따라서 다음과 같이 포인터에 배열을 넣은 뒤 포인터에서 인덱스로 요소에 접근할 수 있다.



int \* numPtr = numArr;처럼 배열을 포인터에 바로 할당할 수 있다. 단, 자료형이 같아야 하며 1차원 배열이라면 \* 가 한 개인 단일 포인터여야 한다.

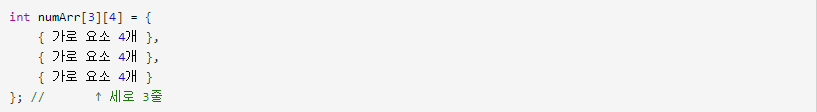
배열을 포인터에 할당한 뒤 포인터를 역참조해보면 배열의 첫 번째 요소의 값이 나온다. 마찬가지로 배열 자체도 역참조해보면 배열의 첫 번째 요소의 값이 나온다. 즉, 실제로는 배열도 포인터라고 할 수 있다.  


배열의 주소가 들어있는 포인터는 인덱스를 통하여 요소에 접근할 수 있다.  

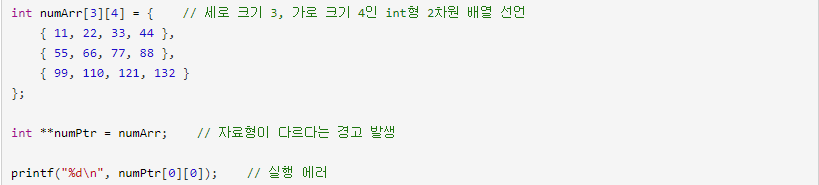

### 2차원 배열

첫 번째 대괄호에는 세로 크기, 두 번째 대괄호에는 가로 크기를 지정한다. 일반적으로 2차원 공간은 가로X세로로 표기하지만 배열로 만들 때는 세로x가로로 표기한다.

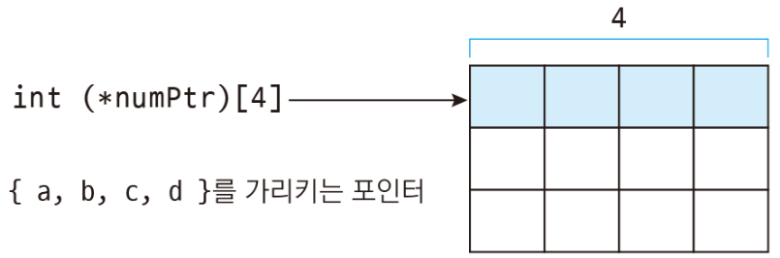
* 자료형 배열이름[세로크기][가로크기]
* 자료형 배열이름[세로크기][가로크기] = {{값,값,값}, {값,값,값}}

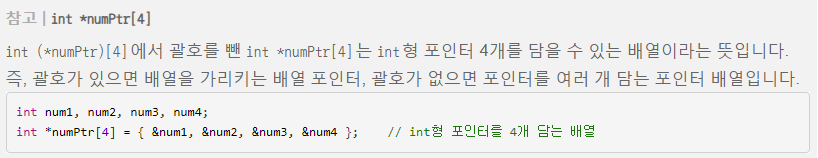
다음은 세로 크기가 3, 가로 크기가 4인 int형 2차원 배열을 선언하는 방법이다. 2차원 배열을 선언하면서 초기화하려면 { }를 사용하는데 가로 요소들을 먼저 묶어주고, 가로 줄을 세로 크기만큼 다시 묶어준다. { } 안의 값과 줄 개수는 가로, 세로 크기보다 작아도 되지만 크면 안된다.  


#### 2차원 배열을 포인터에 넣기

1차원 배열은 단일 포인터에 넣을 수 있었다. 그렇다면 2차원 배열은 이중 포인터에 넣을 수 있지 않을까?  
  
컴파일을 해보면 다음과 같이 자료형이 다르다는 경고가 발생하고, 실행을 해보면 에러가 발생하여 numPtr[0][0] 값은 출력되지 않는다.  


2차원 배열을 포인터에 담으려면 다음과 같이 특별한 방법이 필요하다.  
-. 자료형 (\*포인터이름)[가로크기];  
즉, 포인터를 선언할 때 \*와 포인터 이름을 괄호로 묶어준 뒤 [ ]에 가로 크기를 지정한다. 풀어서 설명하면 가로 크기가 4인 배열을 가리키는 포인터라는 뜻이다.  

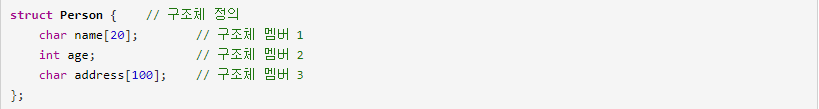


### 포인터와 배열 응용

포인터를 배열처럼 사용하는 방법은 간단하다. 포인터에 malloc 함수로 메모리를 할당해 주면 된다.  
-. 자료형 \*포인터이름 = malloc(sizeof(자료형) \* 크기);

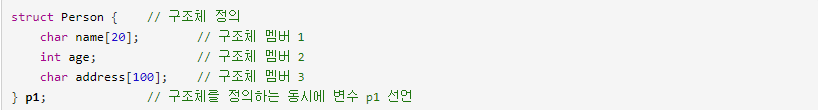
# 구조체

구조체는 정의만 해서는 사용을 할 수가 없다. 따라서 구조체도 변수로 선언해서 사용한다.  
-. struct 구조체이름 변수이름;

구조체 안에 들어있는 변수를 멤버라고 한다.  


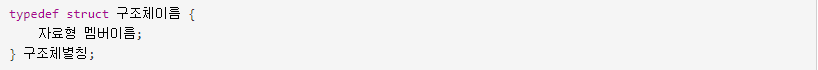
구조체는 보통 main 함수 바깥에 정의한다. 만약 함수 안에 구조체를 정의하면 해당 함수 안에서만 구조체를 사용할 수 있다.

정의한 구조체를 사용하려면 구조체 변수를 선언해야 한다. 이때는 구조체 이름 앞에 반드시 struct 키워드를 붙여준다. 다음은 Person 구조체 타입의 변수 p1을 선언한다는 뜻이다.  


닫는 중괄호와 세미콜론 사이에 변수를 지정해 주면 구조체를 정의하는 동시에 변수를 선언할 수 있다.  
  
구조체 변수 p1은 main 함수 바깥에 선언되어 있다. P1은 전역변수이다.

구조체 변수를 선언하는 동시에 초기화  
-. struct 구조체이름 변수이름 = { .멤버이름1=값1, .멤버이름2=값2 };  
-. struct 구조체이름 변수이름 = { 값1, 값2 };  


## typedef 구조체(익명 구조체)

구조체 변수를 선언할 때 일일이 struct 키워드를 붙이기 귀찮을 때 이를 생략하는 방법은 typedef로 구조체를 정의하면서 별칭(alias)을 지정해 주면 된다.  


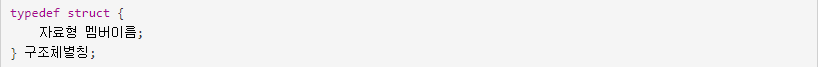
typedef로 구조체의 별칭을 만들었다면 변수는 다음과 같이 선언한다.  
-. 구조체별칭 변수이름;

### typedef 활용하기

typedef는 자료형의 별칭을 만드는 기능이다. 구조체 뿐만 아니라 모든 자료형의 별칭을 만들 수 있다.  
-. typedef 자료형 별칭  
-. typedef 자료형\* 별칭





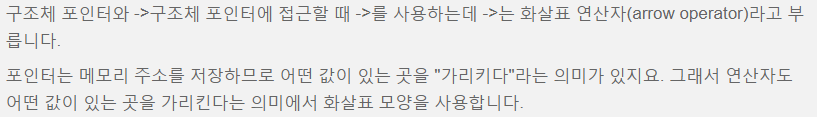
익명 구조체(anonymous structure)를 사용하면 구조체 이름을 지정하지 않아도 된다. 즉, typedef로 구조체를 정의하면서 이름을 생략할 수 있다.  
  
변수는 구조체 별칭으로 선언하면 된다.  
-. 구조체별칭 변수이름;

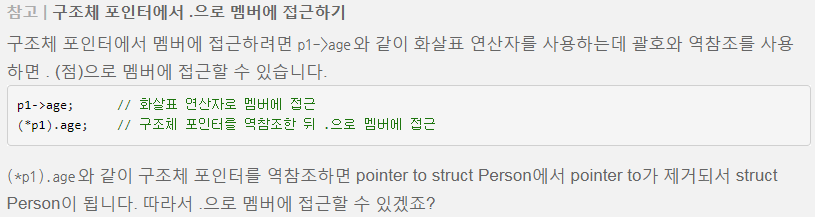
## 구조체 포인터

구조체는 멤버 변수가 여러 개 들어 있어서 크기가 큰 편이다. 구조체 변수를 일일이 선언해서 사용하는 것보다는 포인터에 메모리를 할당해서 사용하는 편이 효율적이다. 함수를 만들어서 구조체를 사용할 때 포인터를 자주 활용하게 된다.

다른 자료형과 마찬가지로 구조체도 포인터를 선언할 수 있으며, 구조체 포인터에는 malloc 함수를 사용하여 동적 메모리를 할당할 수 있다.  
-. struct 구조체이름 \* 포인터이름 = malloc(sizeof(struct 구조체이름));

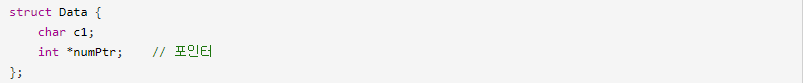
  
구조체 이름 앞에 반드시 struct 키워드를 붙여야 한다. 즉, 포인터를 선언할 때도, sizeof로 크기를 구할 때도 struct 키워드를 넣어 준다.

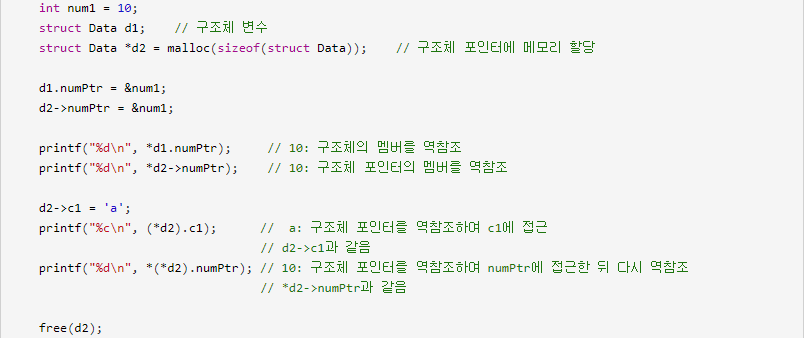


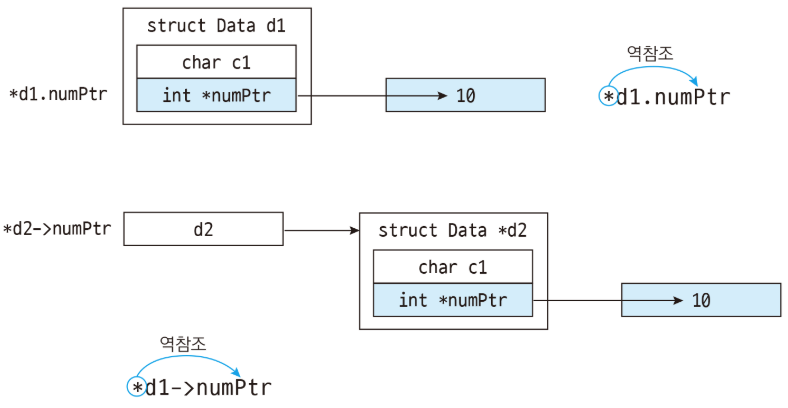


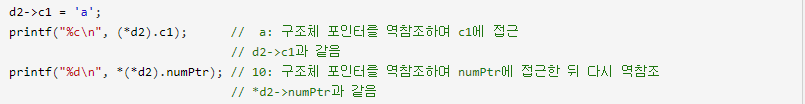
### 구조체의 멤버가 포인터일 때 역참조

구조체의 멤버가 포인터일 때 역참조를 하려면 맨 앞에 \*를 붙이면 된다. 이때 구조체 변수 앞에 \*가 붙어 있더라도 멤버의 역참조이지 구조체 변수의 역참조가 아니다.  
-. \* 구조체변수.멤버  
-. \* 구조체포인터->멤버







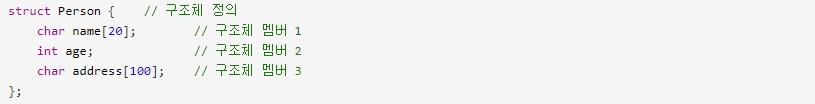
만약 역참조한 것을 괄호로 묶으면 구조체 변수를 역참조한 뒤 멤버에 접근한다는 뜻이 된다. \*(\*d2).numPtr처럼 구조체 포인터를 역참조하여 numPtr에 접근한 뒤 다시 역참조할 수도 있다.  
-. (\*구조체포인터).멤버  
-. \*(\*구조체포인터).멤버  
  
여기서 (\*d2).c1은 d2->c1과 같고, \*(\*d2).numPtr은 \*d2->numPtr과 같다. 즉, 구조체 포인터를 역참조한 뒤 괄호로 묶으면 ->연산자에서 .연산자를 사용하게 되므로 포인터가 일반 변수로 바뀐다는 뜻이다.

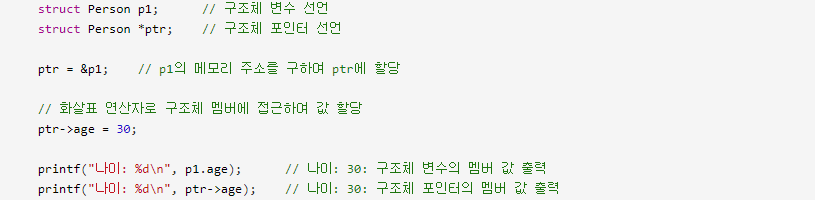
struct 키워드로 포인터를 선언하고 메모리를 할당했으니 typedef로 정의한 구조체별칭으로도 포인터를 선언하고 메모리를 할당할 수 있다.  
-. 구조체별칭 \*포인터이름 = malloc(sizeof(구조체별칭));



### 구조체 포인터에 구조체 변수의 주소 할당

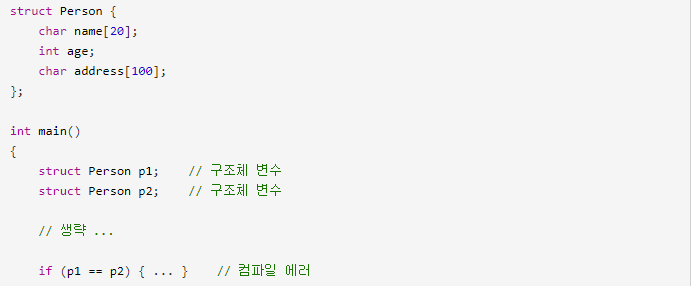
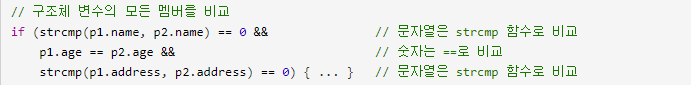
malloc 함수로 동적 메모리를 할당하지 않고 구조체 포인터를 사용하려면, 구조체 변수에 &(주소연산자)를 사용하면 된다.  
-. 구조체포인터 = &구조체변수;

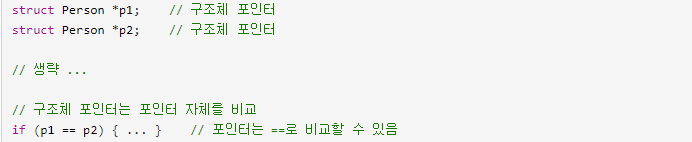




### Q&A

-. 구조체(공용체)의 최대 멤버 개수는 C언어 표준에서는 1,023개로 정의되어 있으며 컴파일러마다 달라질 수 있다.

-. 두 구조체의 내용이 같은지 알기 위해 비교 연산자를 사용하면 컴파일 에러가 발생한다. 이때는 어쩔 수 없이 구조체 변수의 모든 멤버를 비교해야 한다. 그러나 구조체를 포인터로 선언하고 메모리를 할당했을 때는 포인터 자체를 ==로 비교할 수 있다. 두 포인터에 저장된 메모리 주소가 같다면 두 구조체의 내용은 같다. 같은 메모리 공간을 가리키는데 안에 저장된 값이 서로 다를 수는 없기 때문이다.  
  


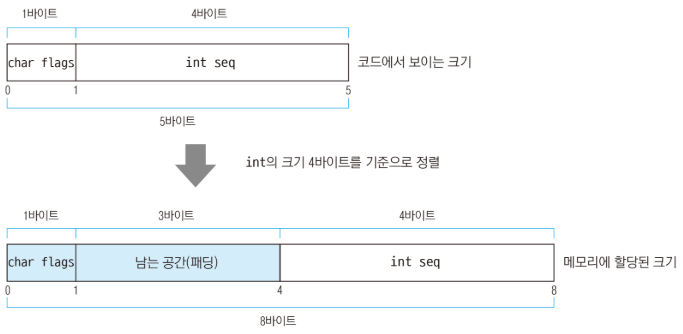


## 구조체 멤버 정렬과 크기

### 구조체 크기, 멤버 위치

sizeof 연산자를 사용하면 알 수 있다.  
-. sizeof ( struct 구조체 )  
-. sizeof ( 구조체별칭 )  
-. sizeof ( 구조체변수 )  
-. sizeof 구조체변수

C 언어에서는 구조체를 정렬할 때 멤버 중에서 가장 큰 자료형 크기의 배수로 정렬한다.

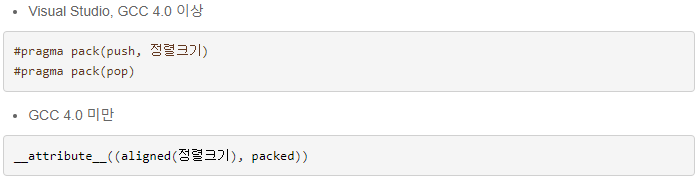


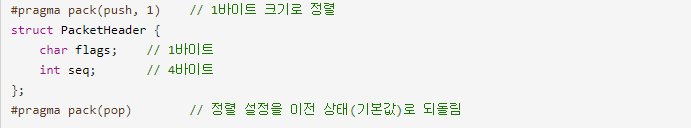
1 바이트 크기의 char flags 뒤에는 4바이트를 맞추기 위해 남는 공간에 3 바이트가 더 들어간다. 이렇게 구조체를 정렬할 때 남는 공간을 채우는 것을 padding이라고 부른다.

구조체에서 멤버의 위치(offset)를 구할 때는 offsetof 매크로를 사용한다(stddef.h 헤더 파일에 정의되어 있다).  
-. offsetof(struct 구조체, 멤버)  
-. offsetof(구조체별칭, 멤버)

### 구조체 정렬 크기 조절

C언어에서는 구조체를 정렬하는 표준 방법이 없다. 그러나 각 컴파일러에서 제공하는 특별한 지시자를 사용하면 구조체 정렬 크기를 조절할 수 있다.

  
pack 안의 1은 1바이트 단위로 정렬하라는 의미이다. 만약 GCC 버전이 4.0 미만이라면 #pragma pack(push, 1), #pragma pack(pop) 대신 \_\_attribute\_\_((aligned(1), packed))를 사용한다.



# GCC

## 수학 라이브러리 링크

sqrt 함수를 사용하려면 컴파일 옵션에 –lm 옵션을 지정해야 한다(-lm은 수학 라이브러리를 링크한다는 뜻).



# 참고

## Pseudo code

의사코드는 실제 프로그래밍 언어가 아닌 사람의 언어로 프로그래밍 언어를 표현한 것이다. 보통 특정 프로그래밍 언어를 사용하지 않고 알고리즘이나 컴퓨터 명령을 기술할 때 사용한다.



# Header file

## limits.h

정수 자료형의 최소값, 최대값이 정의되어 있다.



## stdint.h

CPU와 OS에 따라 정수 자료형의 크기도 그때 그때 달라져서 혼란을 가져왔기 때문에 C99 표준부터 stdint.h 헤더 파일이 추가되었다.

int8\_t  
int16\_t  
int32\_t  
int64\_t  
uint8\_t  
uint16\_t  
uint32\_t  
uint64\_t

이런 자료형은 크기를 정확하게 표현해야 하는 파일 압축 및 암호화, 네트워크 프로그래밍을 할 때 특히 유용하다. stdint의 최소, 최대값은 stdint.h 헤더 파일 안에 정의되어 있으므로 limits.h 헤더 파일을 사용하지 않아도 된다.

## stdlib.h

melloc, free 함수가 선언되어 있음.

## string.h

memset 함수.

## stdbool.h

bool, true, false