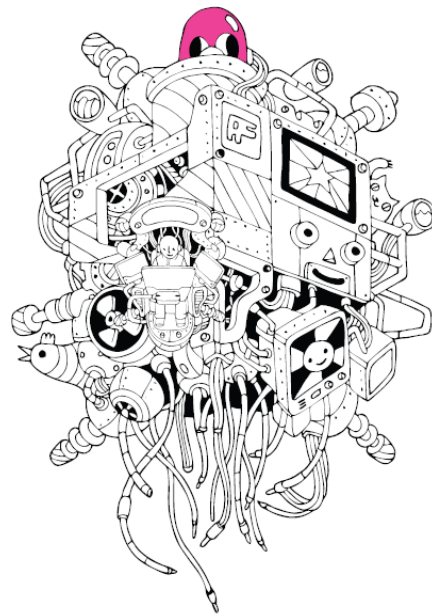


윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 01. 이것이 C언어다.

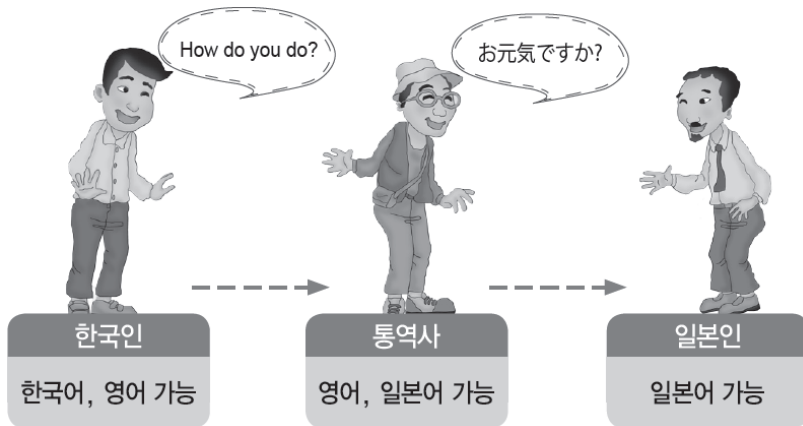
윤성우의 열혈 C 프로그래밍



Chapter 01-1. C언어의 개론적인 이야기

윤성우 저 열혈강의 C 프로그래밍 개정판

C언어는 프로그래밍 언어이다.



√ 프로그래밍 언어란 무엇인가?

사람과 컴파일러가 이해할 수 있는 약속된 형태의 언어를 의미한다. C언어도 프로그래밍 언어 중 하나이다.

√ 컴파일러는?

프로그래밍 언어로 작성된 프로그램을 컴퓨터가 이해할 수 있는 기계어로 번역하는 역할을 한다.

√ 기계어(Machine Language)란?

컴퓨터가 이해할 수 있는 0과 1로 구성된 형태의 언어를 뜻함.

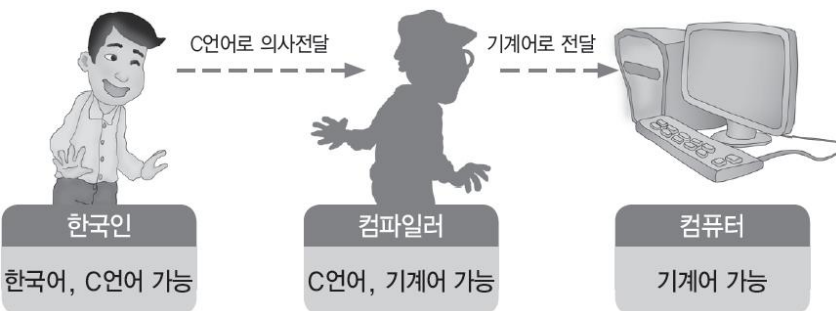
√ C언어를 공부한다는 것은?

문법을 이해하는 것.

표현능력을 향상시키는 것.

많이 사용할수록 표현에 능숙해진다.

다른 이의 표현을 참조할수록 표현이 부드러워진다.



C언어의 역사와 특징

√ C언어의 역사

1971년경 UNIX라는 운영체제의 개발을 위해 Dennis Ritchie와 Ken Thompson이 함께 설계한 범용적인 고급 (high-level)언어.

근원: ALGOL 60(1960) ▶ CPL(1963) ▶ BCPL(1969) ▶ B언어(1970)

√ C언어 등장 이전의 유닉스 개발

어셈블리(assembly) 언어라는 저급(low-level)언어로 만들어졌었다.

그런데 어셈블리 언어는 하드웨어에 따라서 그 구성이 달라지기 때문에 CPU 별로 유닉스를 각각 개발해야만 했다.

√ C언어 등장 이후 유닉스 개발

C언어의 구성은 CPU에 따라 나뉘지 않기 때문에 CPU별로 유닉스를 각각 개발할 필요가 없다.

√ 고급언어? 저급언어?

사람이 이해하기 쉬운 언어는 고급언어, 기계어에 가까울 수록 저급언어.

C언어는 고급언어이면서 메모리에 직접 접근이 가능하기 때문에 저급언어의 특성도 함께 지닌다고 이야기 한다.



C언어의 장점

√ C언어는 절차지향적 특성을 지닌다. 따라서 쉽게 익숙해질 수 있다.

인간의 사고하는 방식과 유사하다.

√ C언어로 작성된 프로그램은 이식성이 좋다.

CPU에 따라 프로그램을 재작성할 필요가 없다.

그러나 근래에는 C언어보다 이식성이 훨씬 뛰어난 언어들이 등장하고 있어서 장점으로 부각시키기에는 한계가 있다.

√ C언어로 구현된 프로그램은 좋은 성능을 보인다.

C언어를 이용하면 메모리의 사용량을 줄일 수 있고, 속도를 저하시키는 요소들을 최소화 할 수 있다.

단, 잘못 구현하면 오히려 성능이 좋지 못한 프로그램이 만들어지기도 한다.

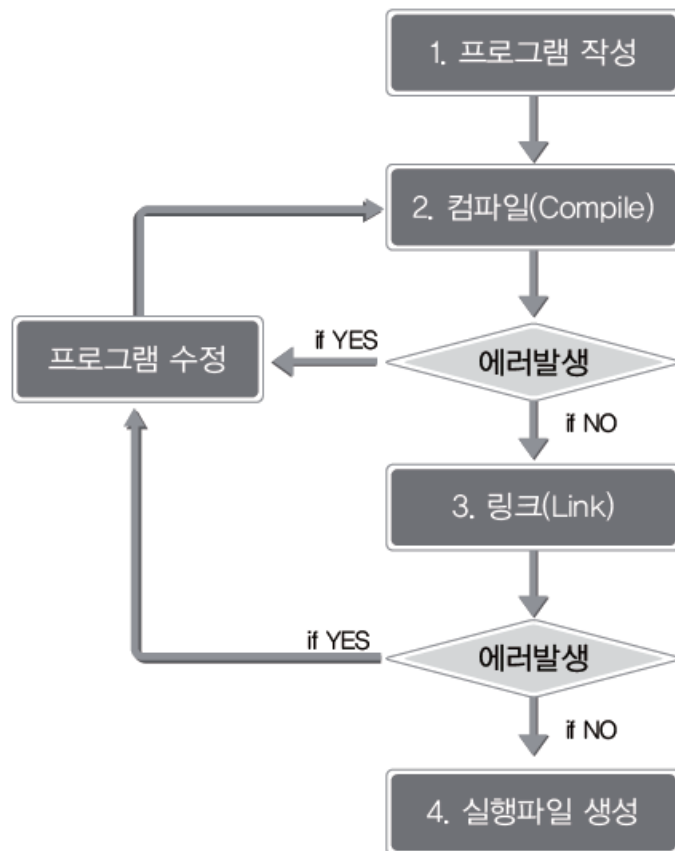
C언어의 장점은 앞으로 C언어를 공부해 나가면서 보다 정확히 이해하게 된다.



A collection of various cartoon objects including a cat, a fish, a laptop, a robot, a planet, a key, a die, a TV, a camera, and a ghost.

윤성우 저 열혈강의 C 프로그래밍 개정판

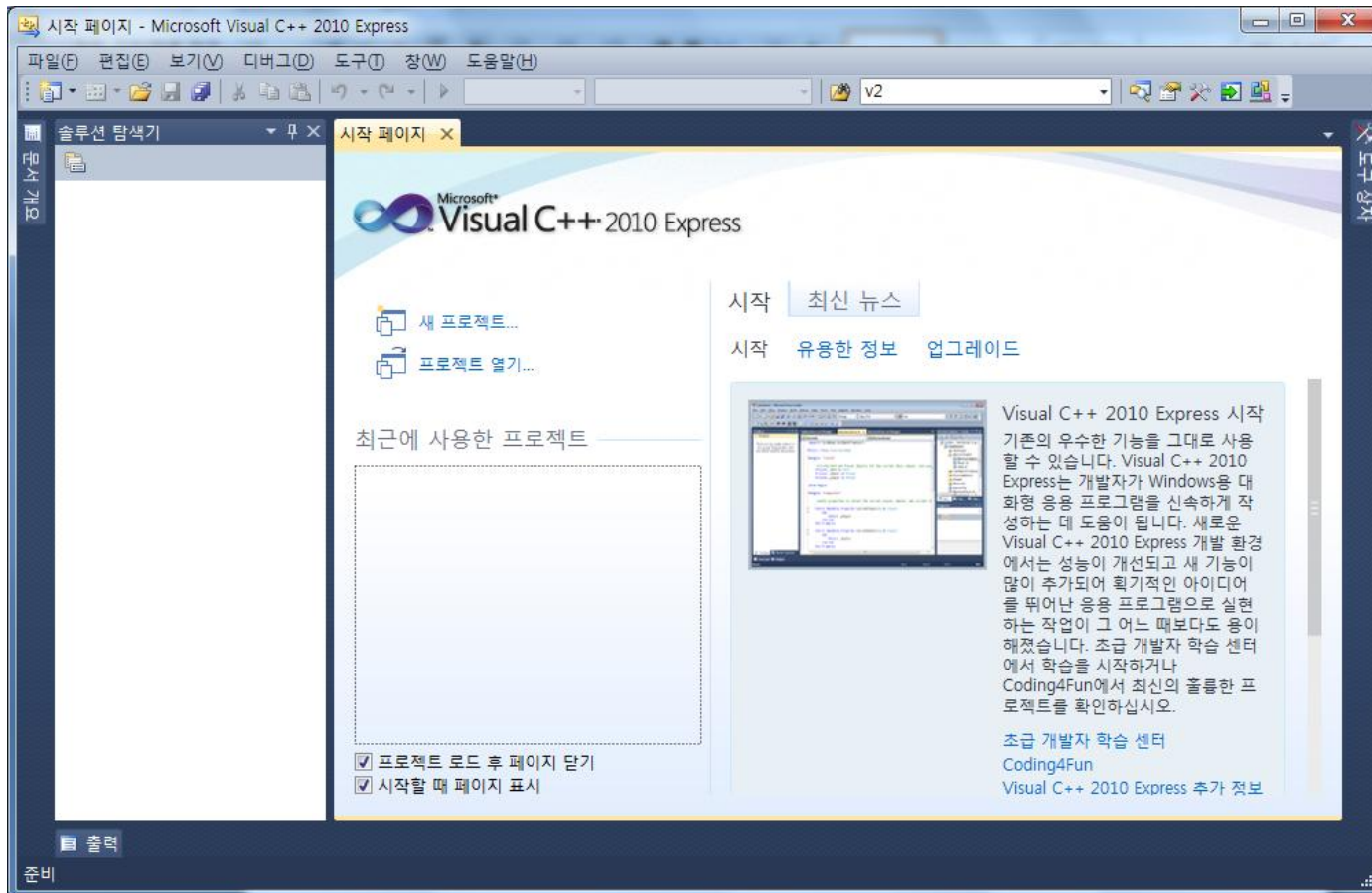
C 프로그램 완성과정의 전체적인 이해



- 첫 번째 단계 : 프로그램의 작성
- 두 번째 단계 : 작성한 프로그램의 컴파일
- 세 번째 단계 : 컴파일 된 결과물의 링크

VC++에서의 솔루션 생성1

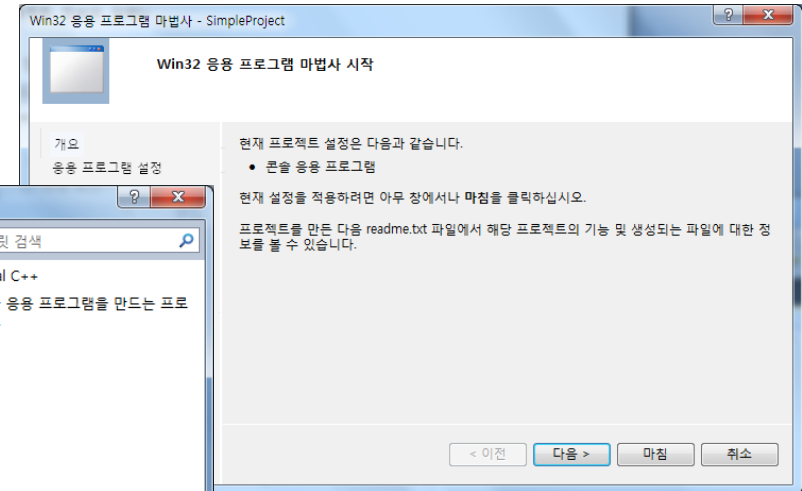
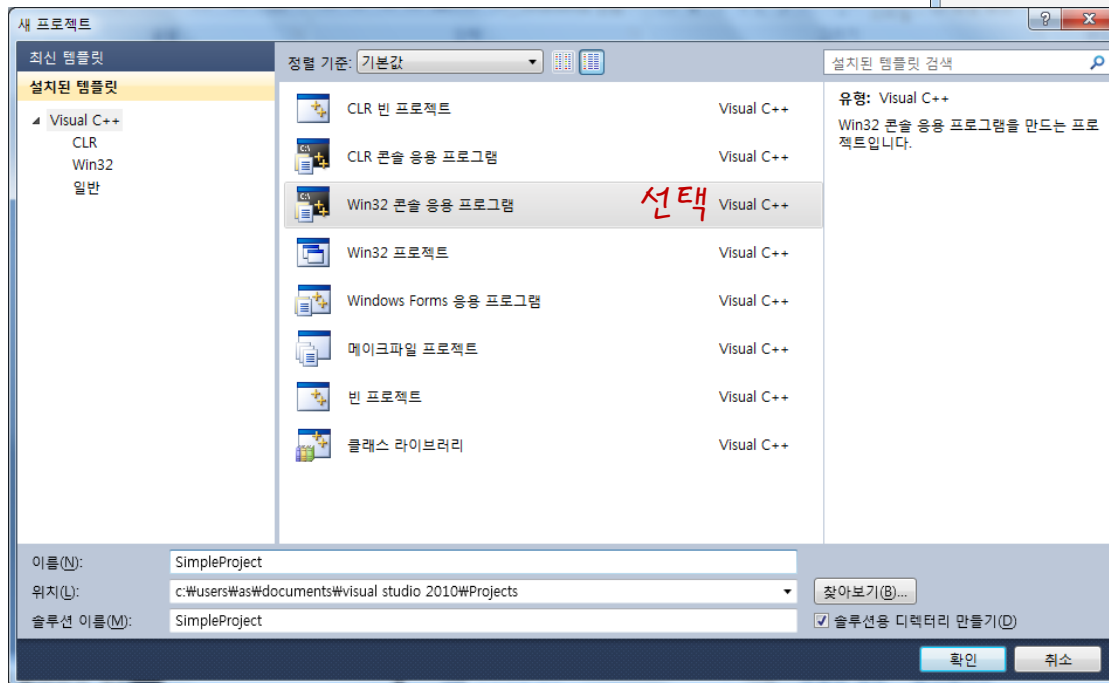
「 Visual C++ Express Edition의 실행 」



VC++에서의 솔루션 생성2

「 프로그래밍을 위한 작업공간의 마련 」

파일(F) → 새로 만들기(N) → 프로젝트(P)

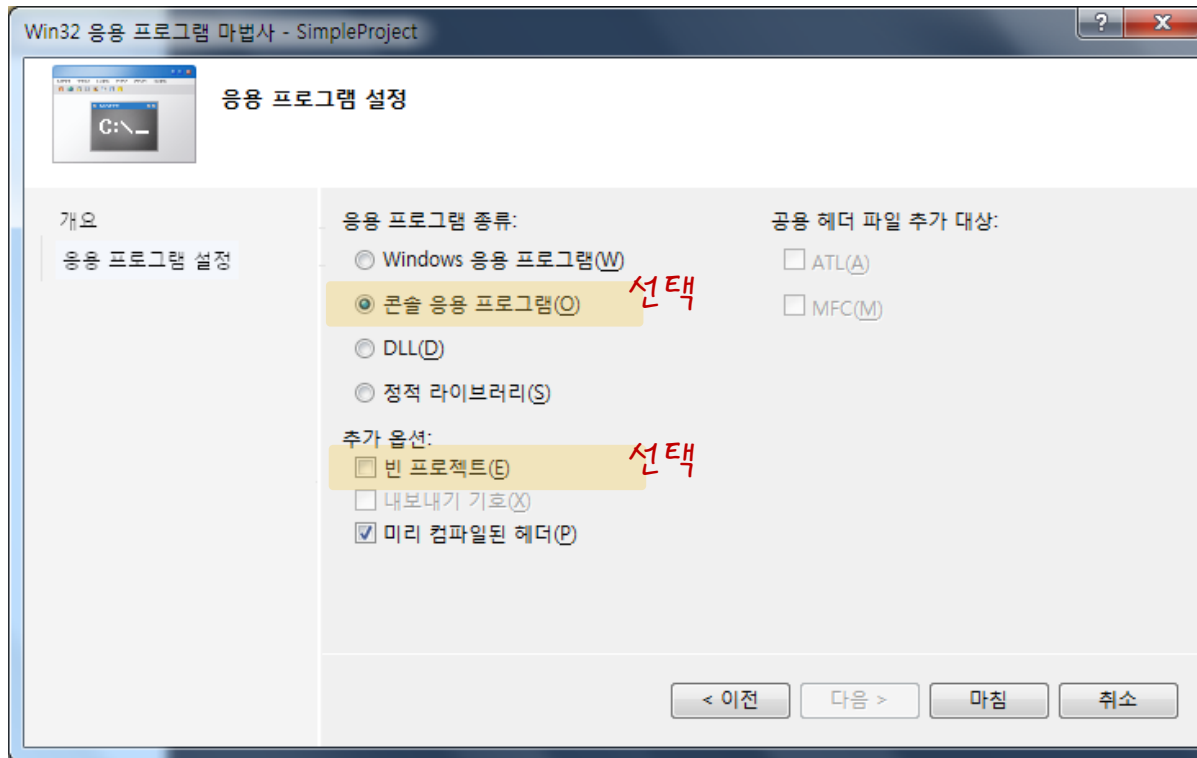


프로젝트의 이름을 SimpleProject로...
프로젝트의 이름을 입력하면 솔루션의 이름
은 자동으로 입력된다.

VC++에서의 솔루션 생성3

「 작업공간의 마련을 위한 최종 선택 」

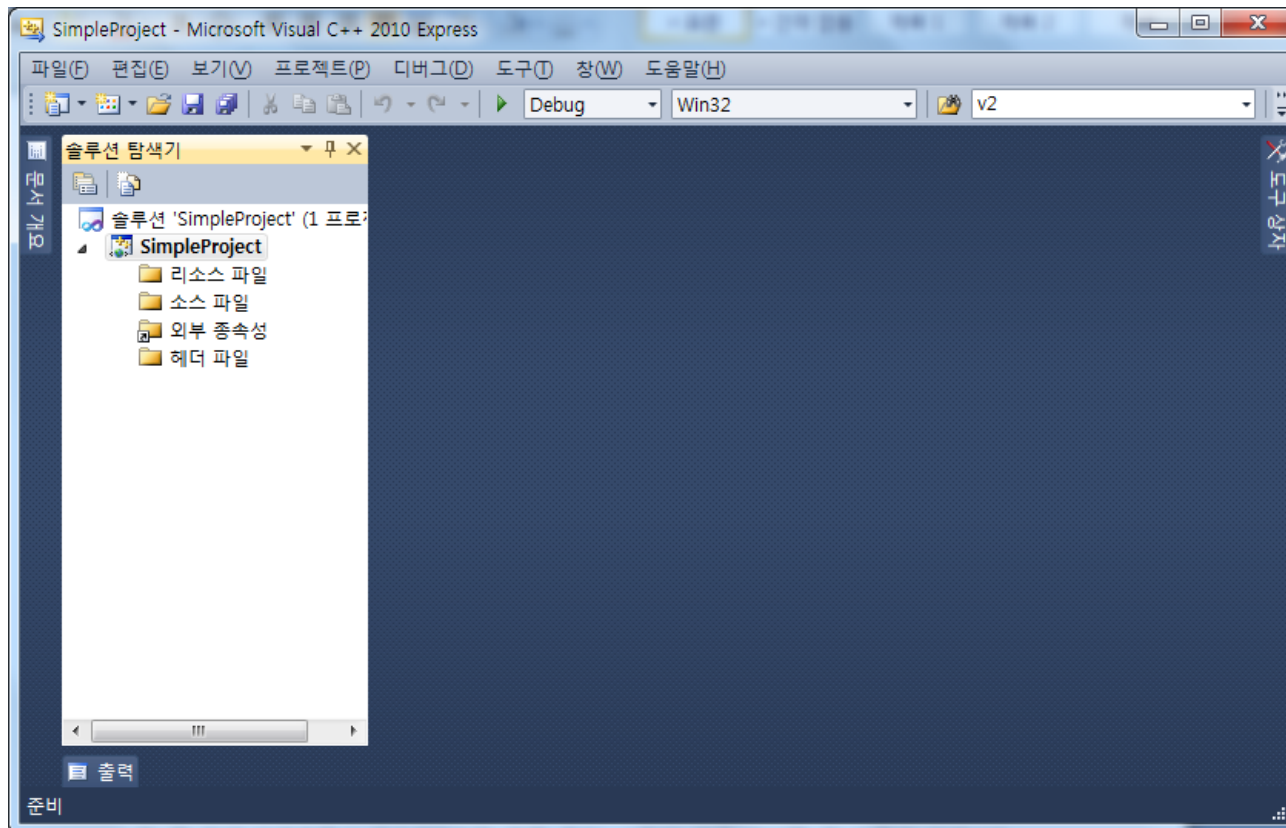
응용 프로그램의 종류에서 콘솔 응용 프로그램, 추가 옵션에서 빈 프로젝트 선택



VC++에서의 솔루션 생성4

「 작업공간의 마련 완료 」

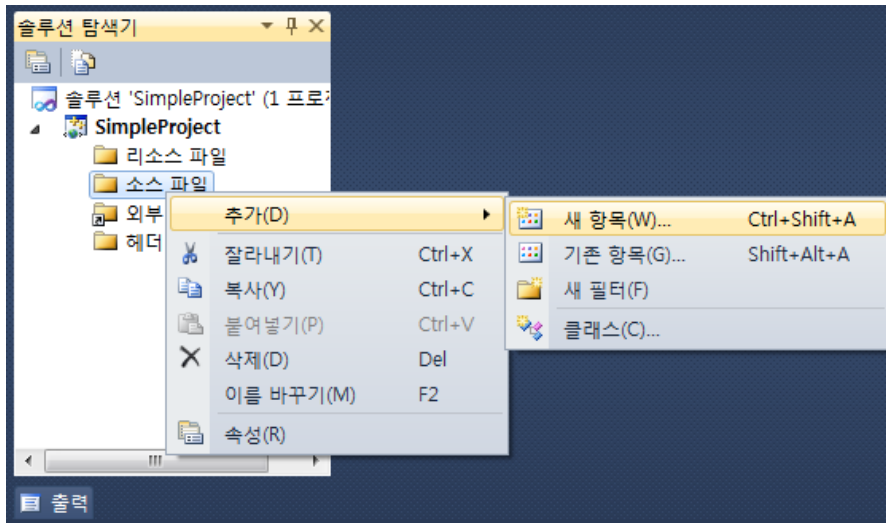
솔루션 탐색기에서 솔루션과 프로젝트가 생성되었음을 확인할 수 있다.



소스파일의 생성1

「 새로운 소스파일을 생성하여 프로젝트에 추가 」

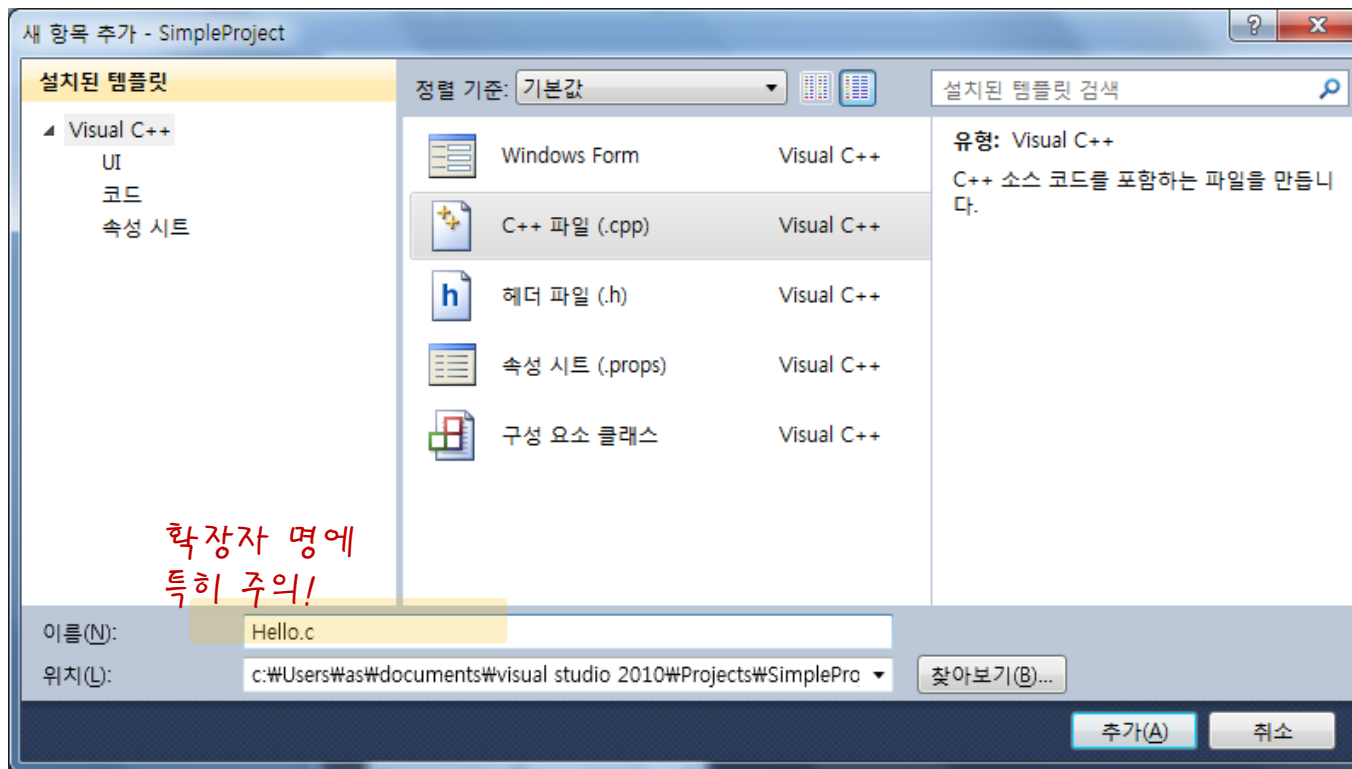
솔루션 탐색기의 소스 파일 위치에서 마우스 오른쪽 버튼을 눌러서 추가



소스파일의 생성2

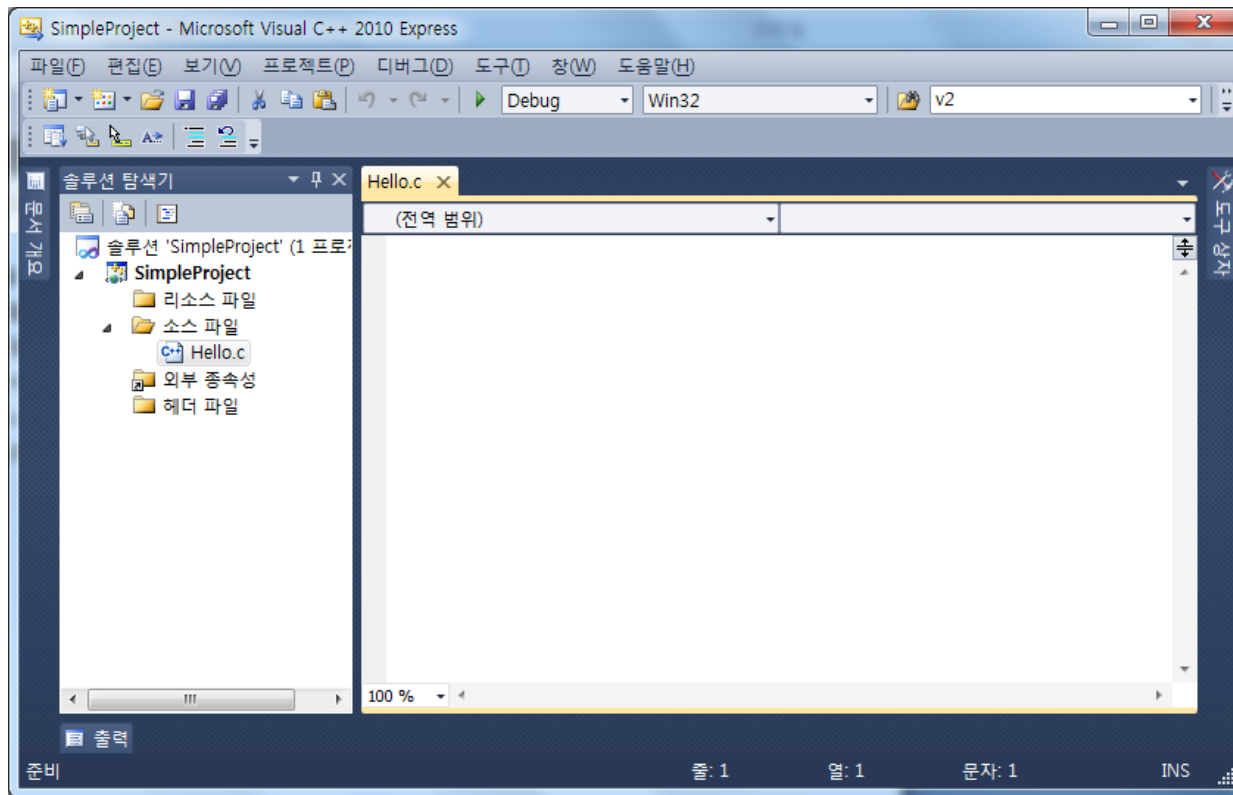
「 확장자가 .c인 소스파일 생성 」

확장자에 따라서 컴파일의 형식을 결정! 확장자를 .c로 해야 C언어 기반으로 컴파일 한다.



생성된 소스파일

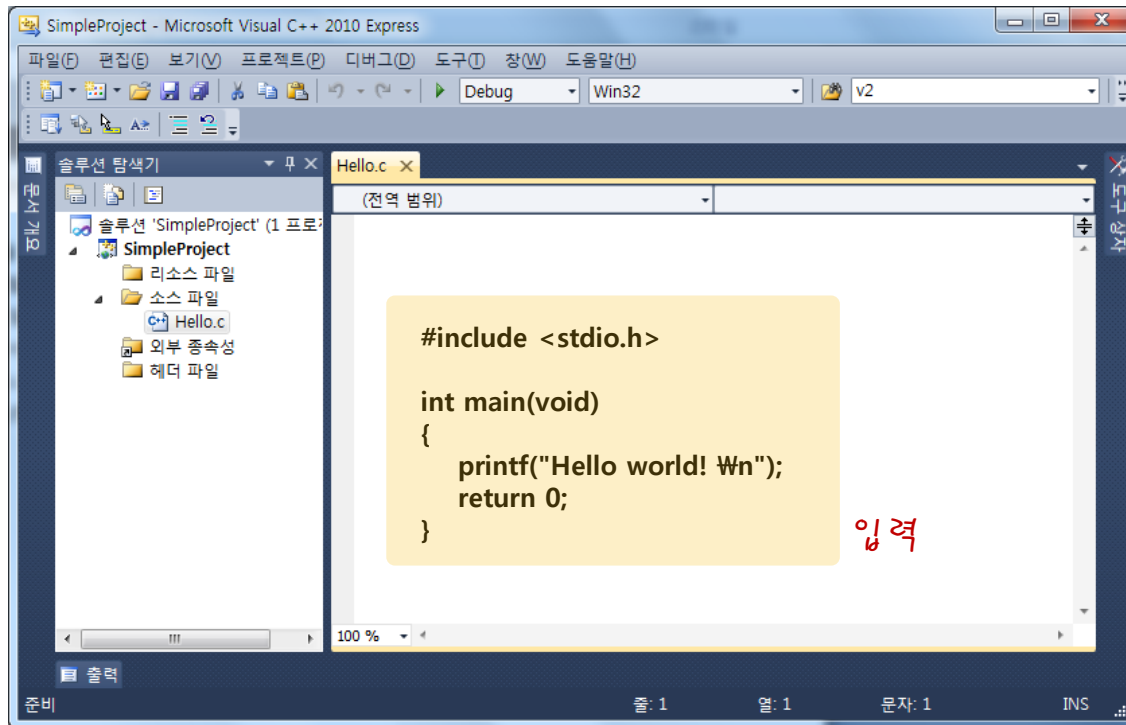
「 소스파일 생성 완료 」



생성된 소스파일과 프로그램의 입력

「 소스코드의 입력 」

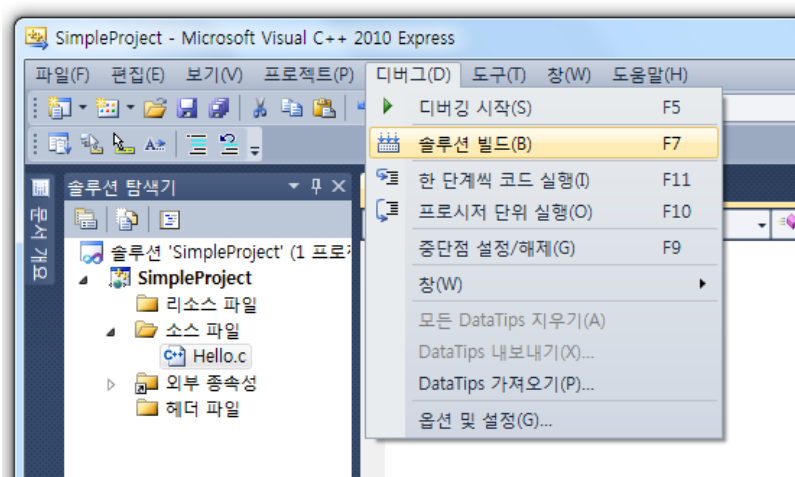
생성된 소스파일에 소스코드를 입력한다.



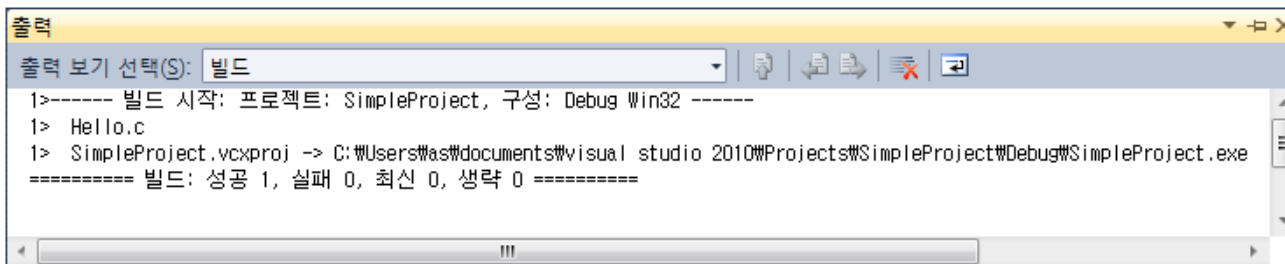
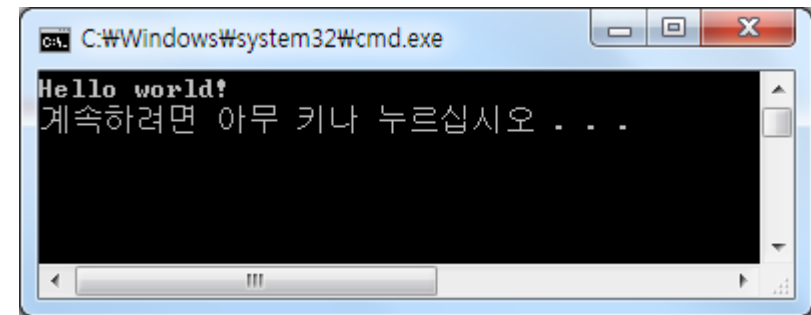
컴파일 및 실행결과 확인

「 컴파일과 실행결과의 확인 」

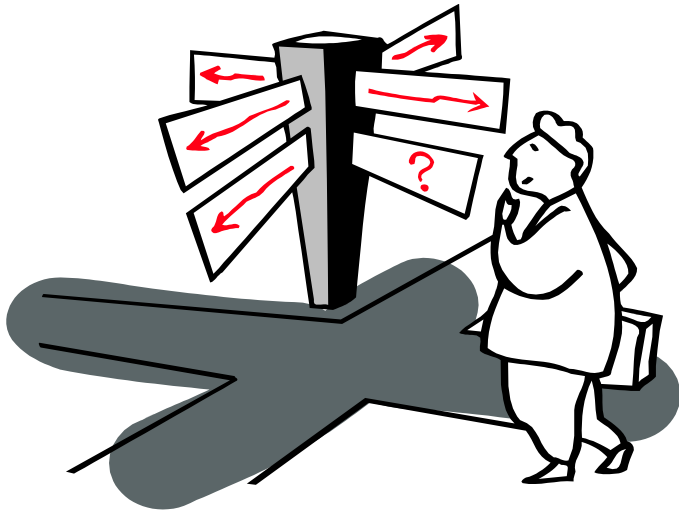
생성된 소스파일에 소스코드를 입력한다.



실행결과

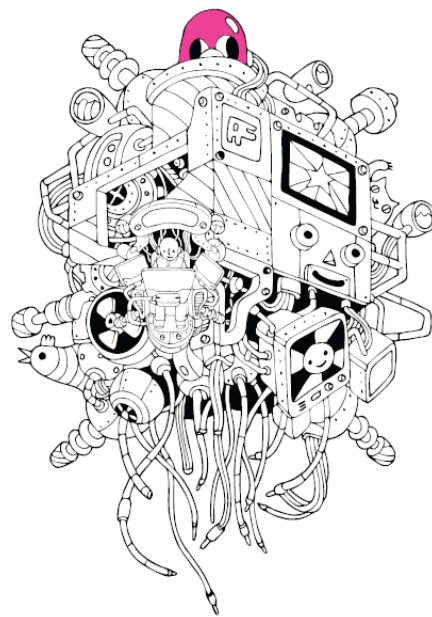


컴파일 결과



Chapter 이이 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 02. 프로그램의 기본구성

윤성우의 열혈 C 프로그래밍



Chapter 02-1. Hello world! 들여다보기

윤성우 저 열혈강의 C 프로그래밍 개정판

C언어의 기본단위인 '함수'의 이해

√ C언어의 기본단위는 함수이다.

함수를 만들고, 만들어진 함수의 실행순서를 결정하는 것이 C언어로 프로그램을 작성하는 것이다.

√ 함수의 기본특성

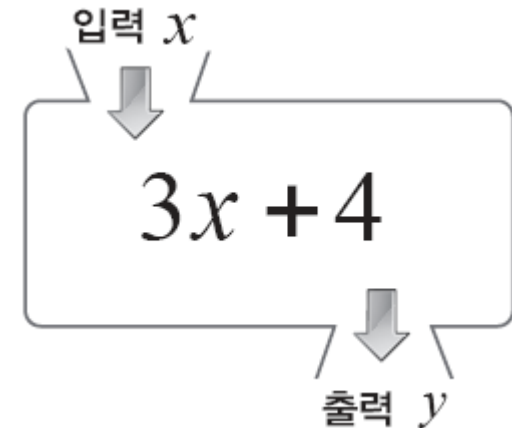
수학적으로 함수에는 입력과 출력이 존재한다.

√ C언어의 함수

C언어의 함수에도 입력과 출력이 존재한다.

√ C언어의 함수와 관련된 용어의 정리

- **함수의 정의** 만들어진 함수, 실행이 가능한 함수를 일컬음
- **함수의 호출** 함수의 실행을 명령하는 행위
- **인자의 전달** 함수의 실행을 명령할 때 전달하는 입력 값



C언어는 함수로 시작해서 함수로 끝이 난다.



예제 Hello.c에서의 함수는 어디에?

√ 프로그램의 시작

첫 번째 함수가 호출이 되면서 프로그램은 시작이 된다.

√ 제일 먼저 호출되는 함수는?

main이라는 이름의 함수! 따라서 C언어로 구현된 모든 프로그램은 시작점에 해당하는 main이라는 이름의 함수를 반드시 정의해야 한다.

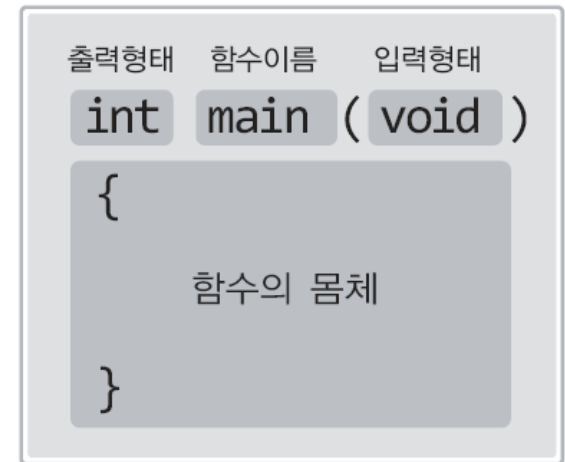
즉, main이라는 이름의 함수가 자동으로 호출이 되면서 프로그램은 실행된다.

√ 함수의 기능

함수의 기능은 중괄호 안에 표현이 되며, 중괄호 안에 표현된 함수의 기능을 가리켜 함수의 몸체라 한다.

√ C언어의 함수에 표시가 되는 세 가지

- **함수의 이름** 함수를 호출할 때 사용하게 되는 이름
- **출력형태** 실행의 결과! 일반적으로 반환형(return type)이라 한다.
- **입력형태** 함수를 호출할 때 전달하는 입력 값의 형태



```

int main(void)
{
    printf("Hello world! \n");
    return 0;
}
    
```

↓
순차적으로 실행

세미콜론

✓ 함수 내에 존재하는 문장의 끝에는 세미콜론 문자 ; 을 붙여준다.

세미콜론은 문장의 끝을 표현하기 위한 문자이다.

✓ 열 줄에 표현된 코드는 열 개의 문장인가?

하나의 문장이 둘 이상의 줄에 표시될 수도 있고, 한 줄에 둘 이상의 문장이 표시될 수도 있다.

즉, 줄 바꿈은 문장의 바꿈을 뜻하는 것이 아니다.

✓ 한 줄에 하나의 문장을 표시하는 것이 가장 일반적이고 또 보기도 좋다.

다음 세 main 함수는 모두 동일한 프로그램이다. 줄 바꿈의 차이가 프로그램의 차이로 이어지지 않는다.

```
int main(void)
{
    printf("Hello world! \n"); return 0;
}
```

```
int main(void)
{
    printf("Hello world! \n");
    return 0;
}
```

```
int main(void) { printf("Hello world! \n"); return 0; }
```



소스코드의 세세한 분석

```
#include <stdio.h>
int main(void)
{
    printf("Hello world! \n");
    return 0;
}
```

헤더파일 선언문

처음 보는 함수의 호출문

√ 표준함수

이미 만들어져서 기본적으로 제공이 되는 함수!

printf 함수는 표준함수이다.

√ 표준 라이브러리

표준함수들의 모임을 뜻하는 말이다.

즉, printf 함수는 표준 라이브러리의 일부이다

#include <stdio.h>

- stdio.h 파일의 내용을 이 위치에 가져다 놓으라는 뜻
- printf 함수의 호출을 위해서 선언해야 하는 문장
- stdio.h 파일에는 printf 함수호출에 필요한 정보 존재

printf("Hello world! \n");

- printf라는 이름의 함수를 호출하는 문장
- 인자는 문자열 "Hello world! \n"
- 인자는 소괄호를 통해서 해당 함수에 전달이 된다.

return 0;

- 함수를 호출한 영역으로 값을 전달(반환)
- 현재 실행중인 함수의 종료

윤성우의 열혈 C 프로그래밍



Chapter 02-2. 주석이 들어가야 완성된
프로그램

윤성우 저 열혈강의 C 프로그래밍 개정판

주석의 필요성과 블록단위 주석

√ 주석의 이해

주석은 소스코드에 삽입된 메모를 뜻한다. 이는 컴파일의 대상에서 제외가 되기 때문에 주석의 유무는 컴파일 및 실행의 결과에 영향을 미치지 않는다.

√ 주석의 필요성

코드의 분석은 글을 읽는 것 만큼 간단하지 않다. 때문에 코드를 분석해야 하는 남을 위해서, 그리고 코드를 작성한 작성자 스스로를 위해서라도 코드에 대한 설명인 주석을 간단히나마 달아놓을 필요가 있다. 즉 주석은 선택이 아닌 필수이다.

√ 블록 단위 주석

한 행의 주석처리

```
/* 주석처리 된 문장 */
```

```
/*
    주석처리 된 문장1
    주석처리 된 문장2
    주석처리 된 문장3
*/
```

여러 행의 주석처리

√ 행 단위 주석

```
// 주석처리 된 문장1
// 주석처리 된 문장2
// 주석처리 된 문장3
```

한 행 단위로의 주석처리

주석을 다는 방식은

프로젝트 별로 팀원과 상의하여 결정하게 된다.

주석 처리의 예

```
/*
제 목: Hello world 출력하기
기 능: 문자열의 출력
파일 이름: HelloComment.c
수정 날짜: 2014. 07. 15
작성자: 윤성우
*/
#include <stdio.h>    // 헤더파일 선언

int main(void)    // main 함수의 시작
{
    /*
    이 함수 내에서는 하나의 문자열을 출력한다.
    문자열은 모니터로 출력된다.
    */
    printf("Hello world! \n");    // 문자열의 출력
    return 0;    // 0의 반환
}    // main 함수의 끝
```

과도하게 처리된 주석(주석도 과하면 좋지 않다)!
주석을 다는 방법을 소개하기 위한 예제일 뿐이다.



주석처리에 있어서의 주의점

```
1.  /*
2.     주석처리 된 문장1
3.     /* 단일 행 주석처리 */
4.     주석처리 된 문장2
5.  */
```

잘못 달린 주석(컴파일 시 오류 발생)

```
1.  /*
2.     주석처리 된 문장1
3.     // 단일 행 주석처리
4.     주석처리 된 문장2
5.  */
```

잘 달린 주석(컴파일 시 오류 발생하지 않음)

주석을 달다 보면 주석이 겹치는(중첩되는) 경우가 발생하기도 한다. 그런데 블록 단위 주석은 겹치는 형태로 달 수 없다.



윤성우의 열혈 C 프로그래밍



Chapter 02-3. printf 함수의 기본적인 이해

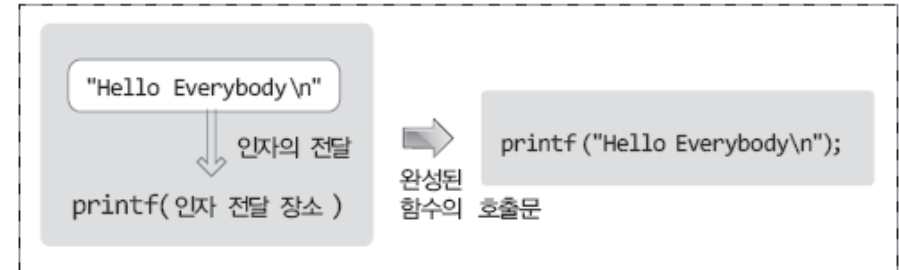
윤성우 저 열혈강의 C 프로그래밍 개정판

printf 함수를 이용한 정수의 출력

```
int main(void)
{
    printf("Hello Everybody\n");
    printf("%d\n", 1234);
    printf("%d %d\n", 10, 20);
    return 0;
}
```

실행결과

```
Hello Everybody
1234
10 20
```



%d

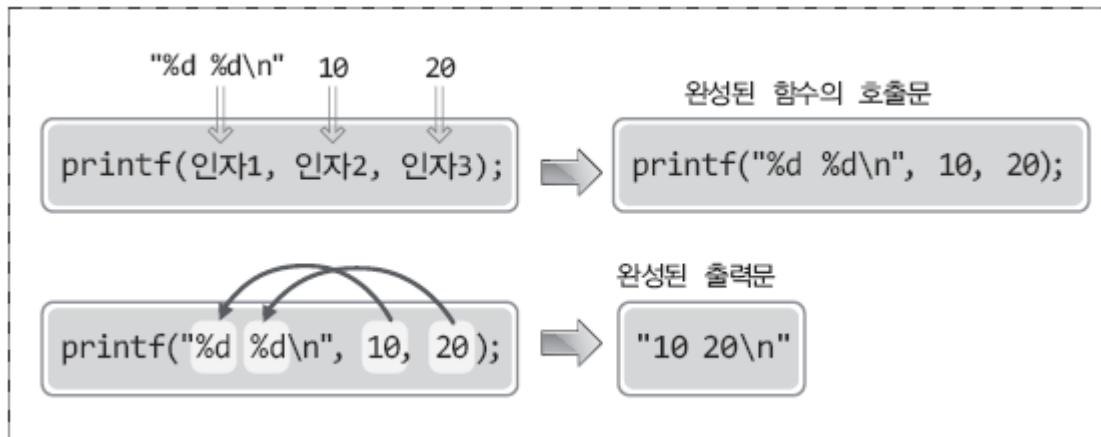
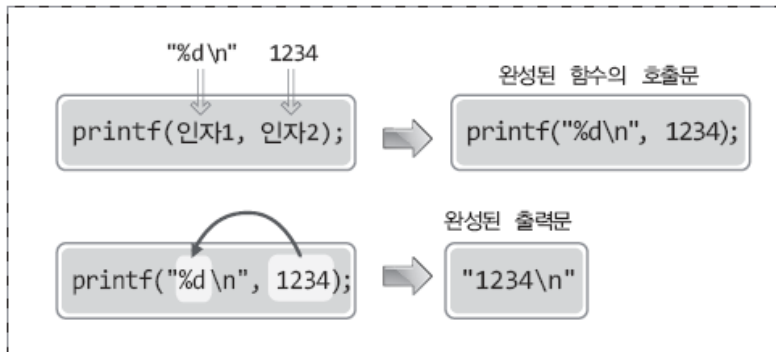
- 문자열에 삽입된 %d를 가리켜 '서식문자'라 한다.
- 서식문자는 출력의 형태를 지정하는 용도로 사용이 된다.
- %d는 부호가 있는 10진수 정수의 형태로 출력하라는 의미를 담는다!

\n은 이스케이프 시퀀스(escape sequence) 또는 특수문자라 불리며 개행을 의미하는 용도로 사용된다.

출력의 대상은?

- 큰 따옴표로 표시되는 문자열의 뒤에 이어서 표시를 하며,
- 콤마로 각각을 구분한다.
- 서식문자 %d가 두 개 등장하면, 출력의 대상도 두 개 등장해야 한다.

정수의 출력에 사용된 서식문자 %d

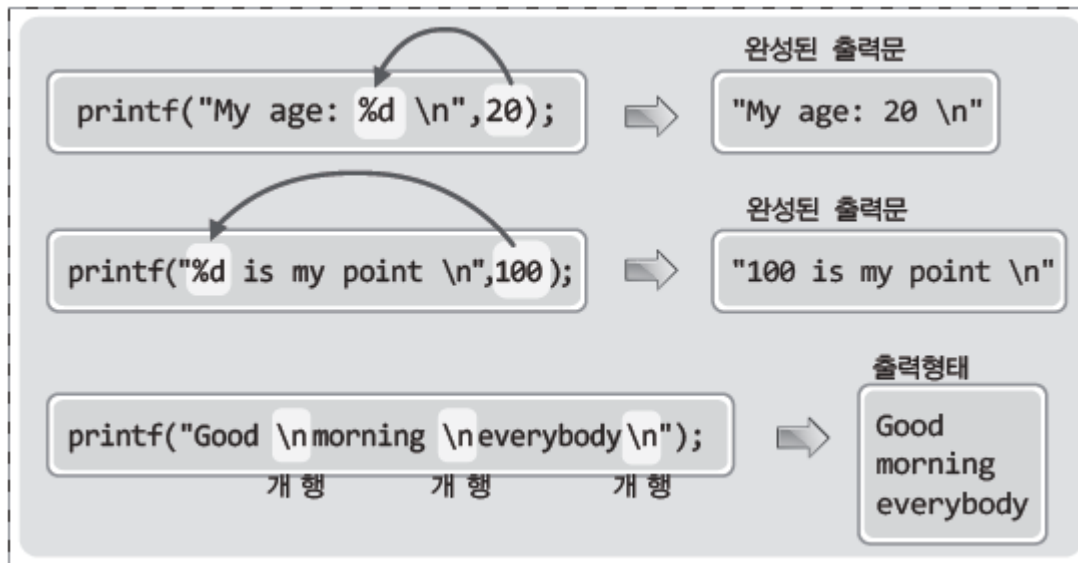


출력의 형태를 다양하게 조합하는 것이 가능하다.

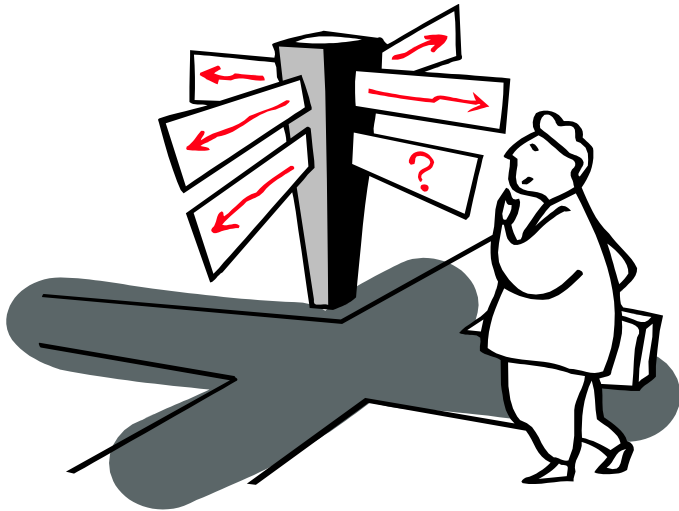
```
int main(void)
{
    printf("My age: %d \n", 20);
    printf("%d is my point \n", 100);
    printf("Good \nmorning \neverybody\n");
    return 0;
}
```

실행결과

```
My age: 20
100 is my point
Good
morning
everybody
```

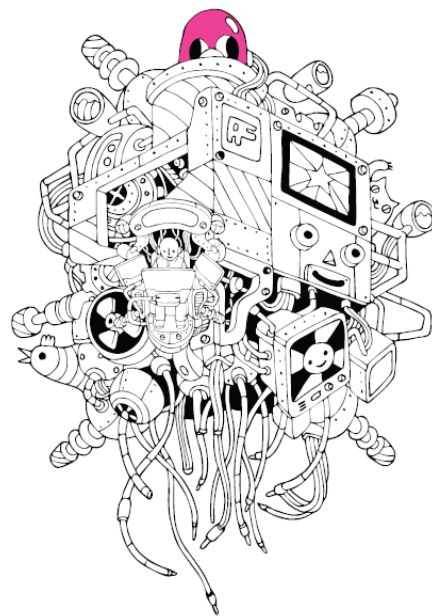


이후에는 보다 다양한 서식문자를 공부하게 된다. 그리고 그렇게 되면 보다 다양한 형태로 출력의 형태를 조합할 수 있게 된다.



Chapter 02가 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 03. 변수와 연산자

윤성우의 열혈 C 프로그래밍



Chapter 03-1. 연산을 위한 연산자와
값의 저장을 위한 변수

윤성우 저 열혈강의 C 프로그래밍 개정판

덧셈 프로그램의 구현에 필요한 + 연산자

```
int main(void)
{
    3+4;    // 3과 4의 합을 명령함
    return 0;
}
```

실행결과로는 아무것도 나타나지 않습니다.

+

- 컴파일 및 실행 시 문제가 발생하지 않으므로 인식 가능한 기호임이 확실하다.
- 실제로 +는 덧셈의 의미를 갖는다. 따라서 실행으로 인해서 3과 4의 합이 진행이 된다.
- +와 같은 기호를 가리켜 연산자라 한다.

연산의 결과는?

- + 연산만 요구를 하였지 그 결과를 출력하기 위한 어떠한 코드도 삽입되지 않았다.
- 따라서 아무런 출력도 이뤄지지 않는다.
- 연산의 결과를 저장해 두어야 원하는 바를 추가로 진행할 수 있다.
- 연산결과 또는 값의 저장을 위해서 C언어에서는 변수(variable)이라는 것을 제공한다.



변수를 이용한 데이터의 저장

√ 변수란?

값을 저장할 수 있는 메모리 공간에 붙여진 이름

변수라는 것을 선언하면 메모리 공간이 할당되고 할당된 메모리 공간에 이름이 붙는다.

√ 변수의 이름

변수의 이름을 통해서 할당된 메모리 공간에 접근이 가능하다.

값을 저장할 수도 있고 저장된 값을 참조할 수도 있다.

```
int main(void)
{
    int num;
    num=20;
    printf("%d", num);
    . . .
}
```

int num

- int 정수의 저장을 위한 메모리 공간의 할당
- num 할당된 메모리 공간의 이름은 num

num=20;

- 변수 num에 접근하여 20을 저장

printf("%d", num);

- num에 저장된 값을 참조(출력)



변수의 다양한 선언 및 초기화 방법

```
int main(void)
{
    int num1, num2;    // 변수 num1, num2의 선언
    int num3=30, num4=40; // 변수 num3, num4의 선언 및 초기화

    printf("num1: %d, num2: %d \n", num1, num2);
    num1=10;    // 변수 num1의 초기화
    num2=20;    // 변수 num2의 초기화

    printf("num1: %d, num2: %d \n", num1, num2);
    printf("num3: %d, num4: %d \n", num3, num4);
    return 0;
}
```

실행결과

```
num1: -858993460, num2: -858993460
num1: 10, num2: 20
num3: 30, num4: 40
```

int num1, num2;

- 변수를 선언만 할 수 있다.
- 콤마를 이용하여 둘 이상의 변수를 동시에 선언할 수 있다.
- 선언만 하면 값이 대입되기 전까지 쓰레기 값(의미 없는 값)이 채워진다.

int num3=30, num4=40;

- 선언과 동시에 초기화 할 수 있다.

변수선언 시 주의할 사항

```
int main(void)
{
    int num1;
    int num2;
    num1=0;
    num2=0;
    . . . .
}
```

컴파일 가능한
변수 선언

```
int main(void)
{
    int num1;
    num1=0;
    int num2;
    num2=0;
    . . . .
}
```

컴파일이 불가능할
수도 있는 변수선언

과거의 C 표준에서는 변수의 선언이 맨 앞에
올 것을 요구하였다. 그런데 지금도 그 표준을
따르는 컴파일러가 존재한다.

변수의 이름 규칙

- 첫째 변수의 이름은 알파벳, 숫자, 언더바(_)로 구성된다.
- 둘째 C언어는 대소문자를 구분한다. 따라서 변수 Num과 변수 num은 서로 다른 변수이다.
- 셋째 변수의 이름은 숫자로 시작할 수 없고, 키워드도 변수의 이름으로 사용할 수 없다(키워드
에 대해서는 잠시 후 설명한다).
- 넷째 이름 사이에 공백이 삽입될 수 없다.

의미 있는 이름을
짓는 것이 가장 중요하다!

잘못된 이름들

```
int 7ThVal;        // 변수의 이름이 숫자로 시작했으므로
int phone#;        // 변수의 이름에 #과 같은 특수문자는 올 수 없다.
int your name;     // 변수의 이름에는 공백이 올 수 없다.
```

변수의 자료형(Data Type)

✓ 정수형 두 가지 부류

정수형 변수와 실수형 변수

✓ 정수형 변수

정수 값의 저장을 목적으로 선언된 변수

정수형 변수는 char형, short형, int형, long형 변수로 나뉜다.

✓ 실수형 변수

실수 값의 저장을 목적으로 선언된 변수

실수형 변수는 float형 변수와 double형 변수로 나뉜다.

✓ 정수형 변수와 실수형 변수가 나뉘는 이유는?

정수를 저장하는 방식과 실수를 저장하는 방식이 다르기 때문

```
int num1=24
```

· num1은 정수형 변수 중 int형 변수

```
double num2=3.14
```

· num2는 실수형 변수 중 double형 변수

덧셈 프로그램의 완성

```
int main(void)
{
    int num1=3;
    int num2=4;
    int result=num1+num2;

    printf("덧셈 결과: %d \n", result);
    printf("%d+%d=%d \n", num1, num2, result);
    printf("%d와(과) %d의 합은 %d입니다.\n", num1, num2, result);
    return 0;
}
```

실행결과

덧셈 결과: 7
3+4=7
3와(과) 4의 합은 7입니다.

변수를 선언하여 덧셈의 결과를 저장했기 때문에 덧셈결과를 다양한 형태로 출력할 수 있다.



윤성우의 열혈 C 프로그래밍



Chapter 03-2. C언어의 다양한 연산자 소개

윤성우 저 열혈강의 C 프로그래밍 개정판

대입 연산자와 산술 연산자

연산자	연산자의 기능	결합방향
=	연산자 오른쪽에 있는 값을 연산자 왼쪽에 있는 변수에 대입한다. 예) num = 20;	←
+	두 피연산자의 값을 더한다. 예) num = 4 + 3;	→
-	왼쪽의 피연산자 값에서 오른쪽의 피연산자 값을 뺀다. 예) num = 4 - 3;	→
*	두 피연산자의 값을 곱한다. 예) num = 4 * 3;	→
/	왼쪽의 피연산자 값을 오른쪽의 피연산자 값으로 나눈다. 예) num = 7 / 3;	→
%	왼쪽의 피연산자 값을 오른쪽의 피연산자 값으로 나눴을 때 얻게 되는 나머지를 반환한다. 예) num = 7 % 3;	→

결합방향은 먼저 계산
되어지는 항목을 결정
하기 위하여 존재

```
int main(void)
{
    int num1=9, num2=2;
    printf("%d+%d=%d \n", num1, num2, num1+num2);
    printf("%d-%d=%d \n", num1, num2, num1-num2);
    printf("%d×%d=%d \n", num1, num2, num1*num2);
    printf("%d÷%d의 몫=%d \n", num1, num2, num1/num2);
    printf("%d÷%d의 나머지=%d \n", num1, num2, num1%num2);
    return 0;
}
```

함수호출 문장에 연산식이 있는 경우
연산이 이뤄지고 그 결과를 기반으로 함수의 호출이
진행된다.

9 + 2 = 11
9 - 2 = 7
9 × 2 = 18
9 ÷ 2의 몫 = 4
9 ÷ 2의 나머지 = 1

실행결과

복합 대입 연산자



```
int main(void)
{
    int num1=2, num2=4, num3=6;
    num1 += 3;    // num1 = num1 + 3;
    num2 *= 4;    // num2 = num2 * 4;
    num3 %= 5;    // num3 = num3 % 5;
    printf("Result: %d, %d, %d \n", num1, num2, num3);
    return 0;
}
```

실행결과

Result: 5, 16, 1

부호의 의미를 갖는 + 연산자와 - 연산자

```
int main(void)
{
    int num1 = +2;
    int num2 = -4;

    num1 = -num1;
    printf("num1: %d \n", num1);
    num2 = -num2;
    printf("num2: %d \n", num2);
    return 0;
}
```

int num1 = 2; 와 동일한 문장!

+를 연산자의 범주에 포함시켰기 때문에 컴파일이 가능하다.

실행결과

```
num1: -2
num2: 4
```

```
num1=-num2;    // 부호 연산자의 사용
num1-=num2;    // 복합 대입 연산자의 사용
```

두 연산자를 혼동하지 않도록 주의한다.

```
num1 = -num2;    // 부호 연산자의 사용
num1 -= num2;    // 복합 대입 연산자의 사용
```

혼동을 최소화 하는 띄어쓰기



증가, 감소 연산자

연산자	연산자의 기능	결합방향
++num	값을 1 증가 후, 속한 문장의 나머지를 진행(선 증가, 후 연산) 예) val = ++num;	←
num++	속한 문장을 먼저 진행한 후, 값을 1 증가(선 연산, 후 증가) 예) val = num++;	←
--num	값을 1 감소 후, 속한 문장의 나머지를 진행(선 감소, 후 연산) 예) val = --num;	←
num--	속한 문장을 먼저 진행한 후, 값을 1 감소(선 연산, 후 감소) 예) val = num--;	←

결합방향 잘못됨 :
→ 이 방향이 맞음

```
int main(void)
{
    int num1=12;
    int num2=12;
    printf("num1: %d \n", num1);
    printf("num1++: %d \n", num1++);    // 후위 증가
    printf("num1: %d \n\n", num1);
    printf("num2: %d \n", num2);
    printf("++num2: %d \n", ++num2);    // 전위 증가
    printf("num2: %d \n", num2);
    return 0;
}
```

```
num1: 12
num1++: 12
num1: 13

num2: 12
++num2: 13
num2: 13
```

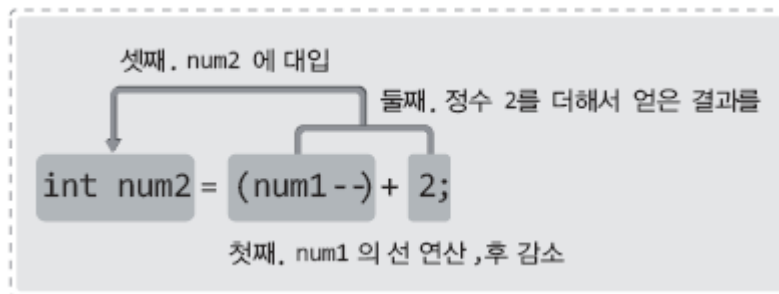
실행결과

증가, 감소 연산자 추가 예제

```
int main(void)
{
    int num1=10;
    int num2=(num1--)+2;    // 후위 감소
    printf("num1: %d \n", num1);
    printf("num2: %d \n", num2);
    return 0;
}
```

num1: 9
num2: 12

실행결과

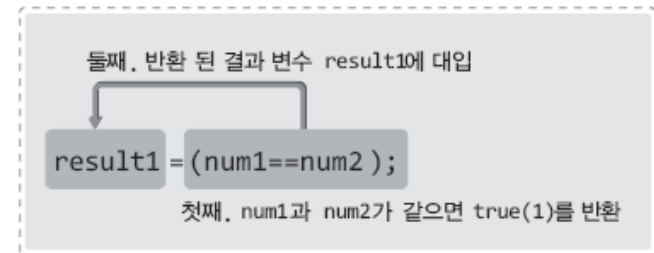


연산의 과정

관계 연산자

연산자	연산자의 기능	결합방향
<	예) $n1 < n2$ n1이 n2보다 작은가?	→
>	예) $n1 > n2$ n1이 n2보다 큰가?	→
==	예) $n1 == n2$ n1과 n2가 같은가?	→
!=	예) $n1 != n2$ n1과 n2가 다른가?	→
<=	예) $n1 <= n2$ n1이 n2보다 같거나 작은가?	→
>=	예) $n1 >= n2$ n1이 n2보다 같거나 큰가?	→

연산의 조건을 만족하면 참을 의미하는 1을 반환하고
만족하지 않으면 거짓을 의미하는 0을 반환하는 연산
자들이다.



C언어는 0이 아닌 모든 값을 참으로 간주한다. 다만

1이 참을 의미하는 대표적인 값일 뿐이다.

실행결과

```
result1: 0
result2: 1
result3: 0
```

```
int main(void)
{
    int num1=10;
    int num2=12;
    int result1, result2, result3;
    result1=(num1==num2);
    result2=(num1<=num2);
    result3=(num1>num2);
    printf("result1: %d \n", result1);
    printf("result2: %d \n", result2);
    printf("result3: %d \n", result3);
    return 0;
}
```

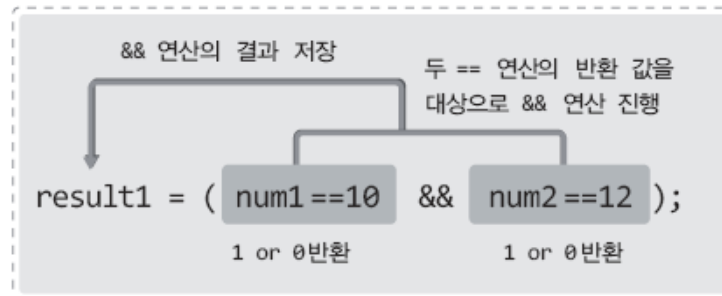
논리 연산자

연산자	연산자의 기능	결합방향
&&	예) A && B A와 B 모두 '참'이면 연산결과로 '참'을 반환(논리 AND)	→ A가 거짓이면 B는 연산되지 않음
	예) A B A와 B 둘 중 하나라도 '참'이면 연산결과로 '참'을 반환(논리 OR)	→ A가 참이면 B는 연산되지 않음
!	예) !A A가 '참'이면 '거짓', A가 '거짓'이면 '참'을 반환(논리 NOT)	←

```
int main(void)
{
    int num1=10;
    int num2=12;
    int result1, result2, result3;
    result1 = (num1==10 && num2==12);
    result2 = (num1<12 || num2>12);
    result3 = (!num1);
    printf("result1: %d \n", result1);
    printf("result2: %d \n", result2);
    printf("result3: %d \n", result3);
    return 0;
}
```

result1: 1
result2: 1
result3: 0

실행결과



왼쪽 예제에서 num1은 0이 아니므로 참과 거짓의 관계로 본다면 거짓에 해당한다. 따라서 ! 연산의 결과로 참을 의미하는 1이 반환되는 것이다.

정확한 설명 : !num1의 값은 !10 과 같으므로 참의 반대이기에 거짓이 됨.
따라서 결과는 0

coma 연산자

√ 콤마(,)

- 콤마도 연산자이다.
- 둘 이상의 변수를 동시에 선언하거나 둘 이상의 문장을 한 행에 삽입하는 경우에 사용되는 연산자이다.
- 둘 이상의 인자를 함수로 전달할 때 인자의 구분을 목적으로도 사용된다.
- 콤마 연산자는 다른 연산자들과 달리 연산의 결과가 아닌 '구분'을 목적으로 한다.

```
int main(void)
{
    int num1=1, num2=2;
    printf("Hello "), printf("world! \n");
    num1++, num2++;
    printf("%d ", num1), printf("%d ", num2), printf("\n");
    return 0;
}
```

실행결과

```
Hello world!
2 3
```



연산자의 우선순위와 결합방향

√ 연산자의 우선순위

- 연산의 순서에 대한 순위
- 덧셈과 뺄셈보다는 곱셈과 나눗셈의 우선순위가 높다.

√ 연산자의 결합방향

- 우선순위가 동일한 두 연산자 사이에서의 연산을 진행하는 방향
- 덧셈, 뺄셈, 곱셈, 나눗셈 모두 결합방향이 왼쪽에서 오른쪽으로 진행된다.

$$3 + 4 \times 5 \div 2 - 10$$

연산자의 우선순위에 근거하여 곱셈과 나눗셈이 먼저 진행된다.

결합방향에 근거하여 곱셈이 나눗셈보다 먼저 진행된다.



윤성우의 열혈 C 프로그래밍



Chapter 03-3. 키보드로부터의 데이터
입력과 C언어의 키워드

윤성우 저 열혈강의 C 프로그래밍 개정판

키보드로부터의 정수입력을 위한 scanf 함수의 호출

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

변수 num 에 저장하라.

scanf("%d", &num);

10진수 정수형태로 입력 받아서

- printf 함수에서의 %d는 10진수 정수의 출력을 의미한다.
- 반면 scanf 함수에서의 %d는 10진수 정수의 입력을 의미한다.
- 변수의 이름 num 앞에 & 를 붙인 이유는 이후에 천천히 알게 된다.

```
int main(void)
{
    int result;
    int num1, num2;
    printf("정수 one: ");
    scanf("%d", &num1);    // 첫 번째 정수 입력
    printf("정수 two: ");
    scanf("%d", &num2);    // 두 번째 정수 입력

    result=num1+num2;
    printf("%d + %d = %d \n", num1, num2, result);
    return 0;
}
```

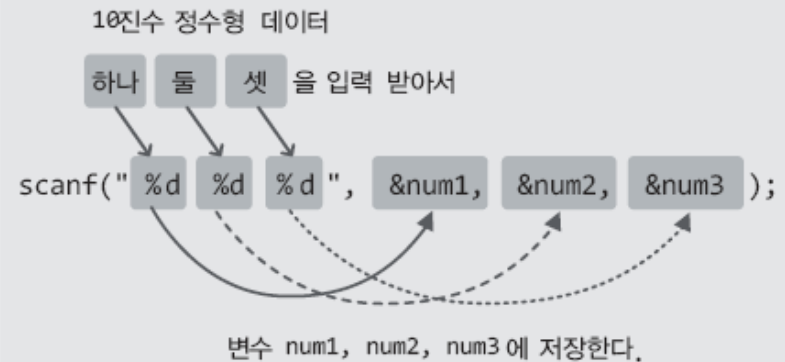
실행결과

정수 one: 3
정수 two: 4
3 + 4 = 7

입력의 형태를 다양하게 지정할 수 있다.

```
int main(void)
{
    int num1, num2, num3;
    scanf("%d %d %d", &num1, &num2, &num3);
    . . . .
}
```

한 번의 **scanf** 함수호출을 통해서 둘 이상의 데이터를 원하는 방식으로 입력 받을 수 있다.



```
int main(void)
{
    int result;
    int num1, num2, num3;
    printf("세 개의 정수 입력: ");
    scanf("%d %d %d", &num1, &num2, &num3);
    result=num1+num2+num3;
    printf("%d + %d + %d = %d \n", num1, num2, num3, result);
    return 0;
}
```

실행결과

세 개의 정수 입력: 4 5 6
4 + 5 + 6 = 15

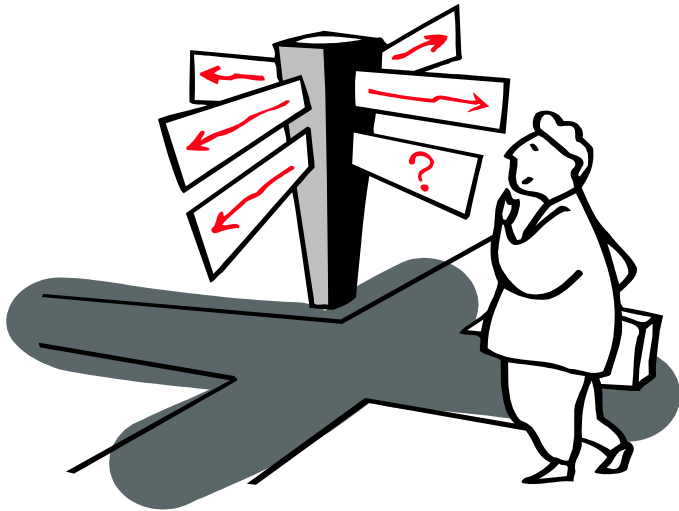
C언어의 표준 키워드

auto	_Bool	break	case
char	_Complex	const	continue
default	do	double	else
enum	extern	float	for
goto	if	_Imaginary	return
restrict	short	signed	sizeof
static	struct	switch	typedef
union	unsigned	void	volatile
while			

C언어의 문법을 구성하는, 그 의미가 결정되어 있는 단어들!

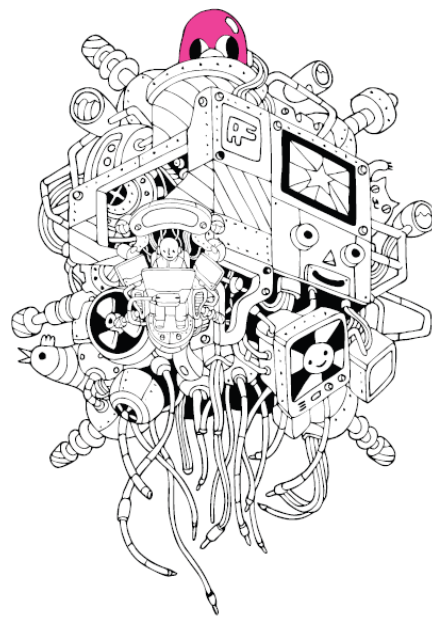
이러한 단어들을 가리켜 **키워드(keyword)**라 한다.





Chapter 03이 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 04. 데이터 표현방식의 이해

윤성우의 열혈 C 프로그래밍



Chapter 04-1. 컴퓨터가 데이터를 표현하는 방식

윤성우 저 열혈강의 C 프로그래밍 개정판

2진수란 무엇인가? 더불어 10진수, 16진수란 무엇인가?



2진수

- 두 개의 기호를 이용해서 값(데이터)를 표현하는 방식

10진수

- 열 개의 기호를 이용해서 값(데이터)을 표현하는 방식

N진수

- N개의 기호를 이용해서 값(데이터)을 표현하는 방식

10 진수	16 진수
9	9
10	A
11	B
12	C
13	D
14	E
15	F
16	10
17	11

자릿수 증가

자릿수 증가

10 진수	2진수
0	0
1	1
2	10
3	11
4	100
5	101

자릿수 증가

자릿수 증가

데이터의 표현단위인 비트(Bit)와 바이트(Byte)

1비트

0 1 1 0 1 1 0 1 0 1 1 1 1 0 1 1

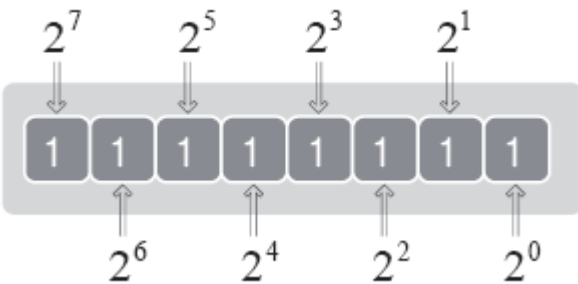
1바이트

0 1 1 0 1 1 0 1 0 1 1 1 1 0 1 1

2바이트

0 1 1 0 1 1 0 1 0 1 1 1 1 0 1 1

컴퓨터 메모리의 주소 값은 **1바이트당 하나의 주소가 할당**되어 있다. 따라서 바이트는 컴퓨터에 있어서 상당히 의미가 있는 단위이다.



왼쪽의 의미 있는 정보를 이용하면 2진수를 쉽게 10진수로 변환할 수 있다.

8진수와 16진수를 이용한 데이터 표현

```
int num1 = 10;    // 특별한 선언이 없으면 10진수의 표현
int num2 = 0xA;    // 0x로 시작하면 16진수로 인식
int num3 = 012;    // 0으로 시작하면 8진수로 인식
```

```
int main(void)
{
    int num1=0xA7, num2=0x43;
    int num3=032, num4=024;

    printf("0xA7의 10진수 정수 값: %d \n", num1);
    printf("0x43의 10진수 정수 값: %d \n", num2);
    printf(" 032의 10진수 정수 값: %d \n", num3);
    printf(" 024의 10진수 정수 값: %d \n", num4);

    printf("%d-%d=%d \n", num1, num2, num1-num2);
    printf("%d+%d=%d \n", num3, num4, num3+num4);
    return 0;
}
```

실행결과

```
0xA7의 10진수 정수 값: 167
0x43의 10진수 정수 값: 67
 032의 10진수 정수 값: 26
 024의 10진수 정수 값: 20
167-67=100
26+20=46
```

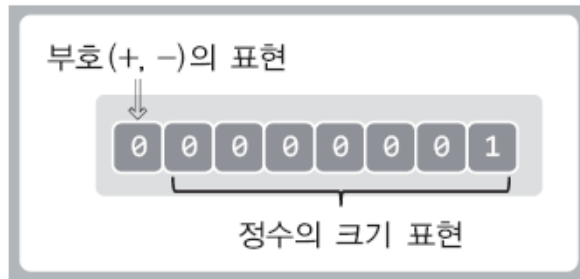

윤성우의 열혈 C 프로그래밍



Chapter 04-2. 정수와 실수의 표현방식

윤성우 저 열혈강의 C 프로그래밍 개정판

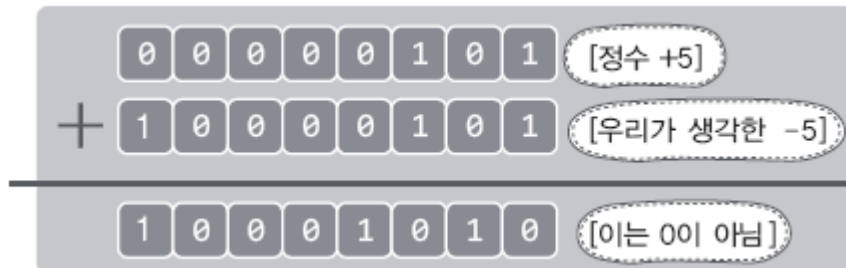
정수의 표현방식



- 가장 왼쪽 비트를 **MSB(Most Significant Bit)**라 한다.
- MSB는 부호를 나타내는 비트이다.
- MSB를 제외한 나머지 비트는 크기를 나타내는데 사용된다.

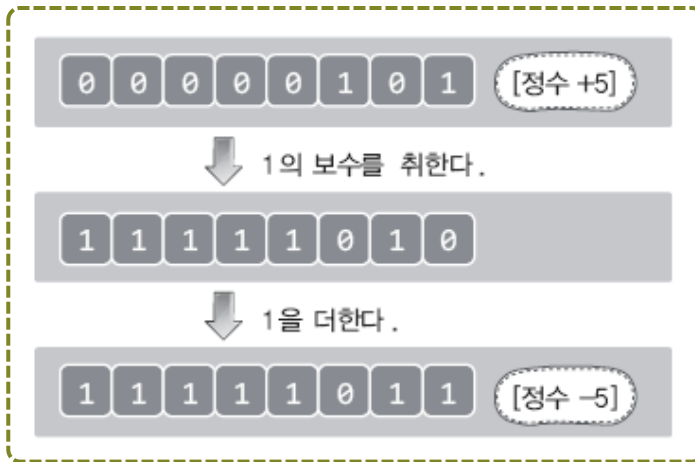
정수의 표현방식은 바이트의 크기와는 상관 없다.

바이트의 크기가 크면 그만큼 넓은 범위의 정수를 표현할 수 있을 뿐이다.



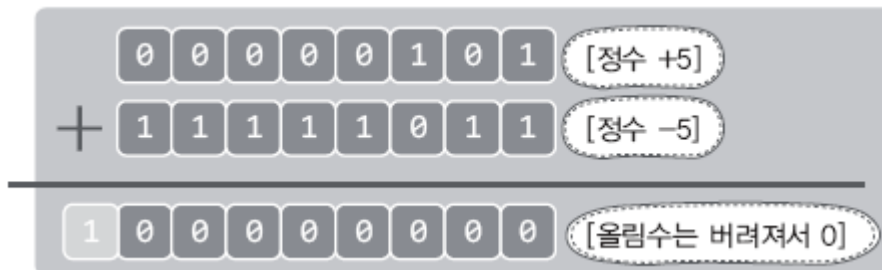
왼쪽에서는 양의 정수를 표현하는 방식으로 음의 정수를 표현하는 것이 적절치 않은 이유를 설명한다.

음의 정수 표현방식



음의 정수를 표현하는 방식

2의 보수를 취하여 음의 정수를 표현한다.



2의 보수 표현법이

음수를 표현하는데에 있어서 타당한지를 확인

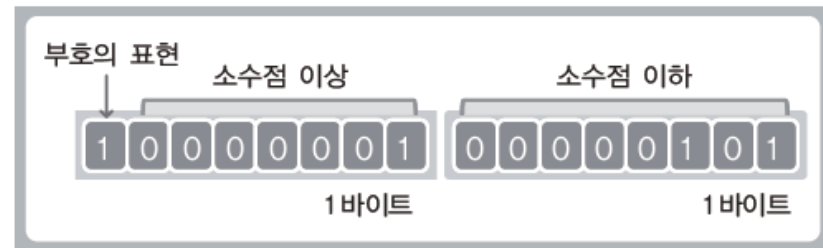


실수의 표현방식

다음의 방식과 같이 정수를 표현하는 방식과 유사하게 실수를 표현하면 다음의 문제가 따른다.

- 표현할 수 있는 실수의 수가 몇 개 되지 않는다.
- 3.12456과 3.12457 사이에 있는 무수히 많은 실수조차 제대로 표현하지 못한다.

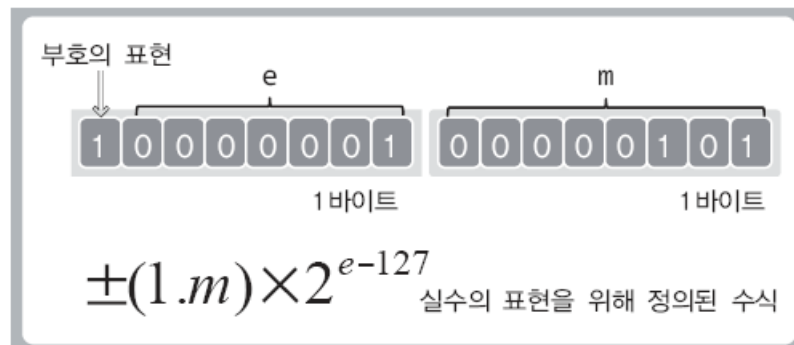
적절하지 못한 실수의
표현방법



따라서 실수의 표현방식은 정수의 표현방식과 다르다.

- 실수의 표현방식에서는 정밀도를 포기하는 대신에 표현할 수 있는 값의 범위를 넓힌다.
- 따라서 컴퓨터는 완벽하게 정밀한 실수를 표현하지 못한다.

오차가 존재하지만
적절한 실수의 표현방법



실수 표현의 오차 확인하기

```
int main(void)
{
    int i;
    float num=0.0;

    for(i=0; i<100; i++)
        num+=0.1;    // 이 연산을 총 100회 진행하게 됩니다.

    printf("0.1을 100번 더한 결과: %f \n", num);
    return 0;
}
```

실행결과

0.1을 100번 더한 결과: 10.000002

이론적으로 오차 없이 모든 실수를 완벽하게 표현할 능력이 있는 컴퓨팅 환경은 존재하지 않는다.
즉, 실수 표현에 있어서의 오차 발생은 C언어의 특성이 아닌 컴퓨터의 기본적인 특성이다.



윤성우의 열혈 C 프로그래밍



Chapter 04-3. 비트 연산자

윤성우 저 열혈강의 C 프로그래밍 개정판

비트 연산자(비트 이동 연산자)

연산자	연산자의 기능	결합방향
&	비트단위로 AND 연산을 한다. 예) num1 & num2;	→
	비트단위로 OR 연산을 한다. 예) num1 num2;	→
^	비트단위로 XOR 연산을 한다. 예) num1 ^ num2;	→
~	단항 연산자로서 피연산자의 모든 비트를 반전시킨다. 예) ~num; // num은 변화 없음, 반전 결과만 반환	←
<<	피연산자의 비트 열을 왼쪽으로 이동시킨다. 예) num<<2; // num은 변화 없음, 두 칸 왼쪽 이동 결과만 반환	→
>>	피연산자의 비트 열을 오른쪽으로 이동시킨다. 예) num>>2; // num은 변화 없음, 두 칸 오른쪽 이동 결과만 반환	→



& 연산자: 비트단위 AND

```
int main(void)
{
    int num1 = 15;    // 00000000 00000000 00000000 00001111
    int num2 = 20;    // 00000000 00000000 00000000 00010100
    int num3 = num1 & num2;    // num1과 num2의 비트단위 & 연산
    printf("AND 연산의 결과: %d \n", num3);
    return 0;
}
```

- 0 & 0 0을 반환
- 0 & 1 0을 반환
- 1 & 0 0을 반환
- 1 & 1 1을 반환

AND 연산의 결과: 4

실행결과

	00000000	00000000	00000000	000	01111
& 연산	00000000	00000000	00000000	000	10100
	00000000	00000000	00000000	000	00100

| 연산자: 비트단위 OR

```
int main(void)
{
    int num1 = 15;    // 00000000 00000000 00000000 00001111
    int num2 = 20;    // 00000000 00000000 00000000 00010100
    int num3 = num1 | num2;
    printf("OR 연산의 결과: %d \n", num3);
    return 0;
}
```

- 0 & 0 0을 반환
- 0 & 1 1을 반환
- 1 & 0 1을 반환
- 1 & 1 1을 반환

OR 연산의 결과: 31

실행결과

	00000000	00000000	00000000	000	01111
연산	00000000	00000000	00000000	000	10100
	00000000	00000000	00000000	000	11111

^ 연산자: 비트단위 XOR

```
int main(void)
{
    int num1 = 15;    // 00000000 00000000 00000000 00001111
    int num2 = 20;    // 00000000 00000000 00000000 00010100
    int num3 = num1 ^ num2;
    printf("XOR 연산의 결과: %d \n", num3);
    return 0;
}
```

• 0 & 0	0을 반환
• 0 & 1	1을 반환
• 1 & 0	1을 반환
• 1 & 1	0을 반환

XOR 연산의 결과: 27

실행결과

	00000000	00000000	00000000	000	01111
^ 연산	00000000	00000000	00000000	000	10100
	00000000	00000000	00000000	000	11011

~ 연산자

```
int main(void)
{
    int num1 = 15;    // 00000000 00000000 00000000 00001111
    int num2 = ~num1;
    printf("NOT 연산의 결과: %d \n", num2);
    return 0;
}
```

- ~ 0 1을 반환
- ~ 1 0을 반환

NOT 연산의 결과: -16

실행결과

~ 연산 전 00000000 00000000 00000000 00001111

~ 연산 후 11111111 11111111 11111111 11110000

<< 연산자: 비트의 왼쪽 이동(Shift)

- `num1 << num2` `num1`의 비트 열을 `num2`칸씩 왼쪽으로 이동시킨 결과를 반환
- `8 << 2` 정수 8의 비트 열을 2칸씩 왼쪽으로 이동시킨 결과를 반환

```
int main(void)
{
    int num = 15;    // 00000000 00000000 00000000 00001111

    int result1 = num<<1;    // num의 비트 열을 왼쪽으로 1칸씩 이동
    int result2 = num<<2;    // num의 비트 열을 왼쪽으로 2칸씩 이동
    int result3 = num<<3;    // num의 비트 열을 왼쪽으로 3칸씩 이동

    printf("1칸 이동 결과: %d \n", result1);
    printf("2칸 이동 결과: %d \n", result2);
    printf("3칸 이동 결과: %d \n", result3);
    return 0;
}
```

실행결과

1칸 이동 결과: 30
2칸 이동 결과: 60
3칸 이동 결과: 120

```
00000000 00000000 00000000 00011110    // 10진수로 30
00000000 00000000 00000000 00111100    // 10진수로 60
00000000 00000000 00000000 01111000    // 10진수로 120
```

왼쪽으로 한 칸씩 이동할 때마다 정수의 값은 두 배씩 증가한다.

반대로 오른쪽으로 한 칸씩 이동할 때마다 정수의 값은 반으로 줄어든다.

>> 연산자: 비트의 오른쪽 이동(Shift)

```
11111111 11111111 11111111 11110000    // -16
```



CPU에 따라서 달라지는 연산의 결과

```
00111111 11111111 11111111 11111100    // 0이 채워진 경우
```

```
11111111 11111111 11111111 11111100    // 1이 채워진 경우
```

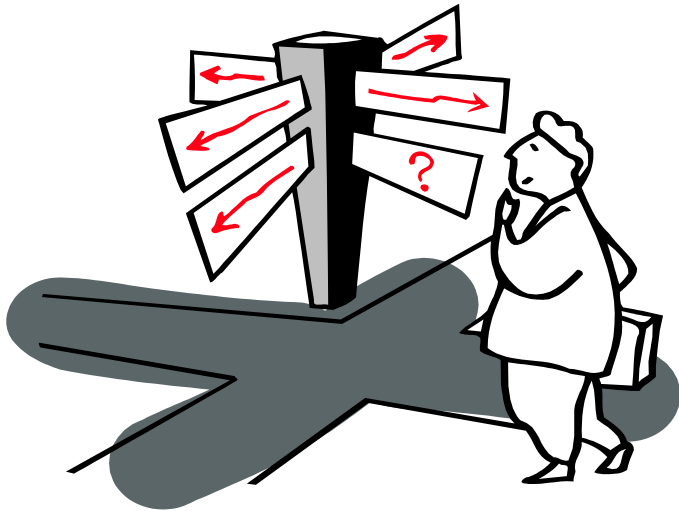
왼쪽 비트가 0으로 채워진 음수의 경우 오른쪽으로 비트를 이동시킨 결과는 CPU에 따라서 달라진다.
따라서 호환성이 요구되는 경우에는 >> 연산자의 사용을 제한해야 한다.

```
int main(void)
{
    int num = -16;    // 11111111 11111111 11111111 11110000
    printf("2칸 오른쪽 이동의 결과: %d \n", num>>2);
    printf("3칸 오른쪽 이동의 결과: %d \n", num>>3);
    return 0;
}
```

실행결과

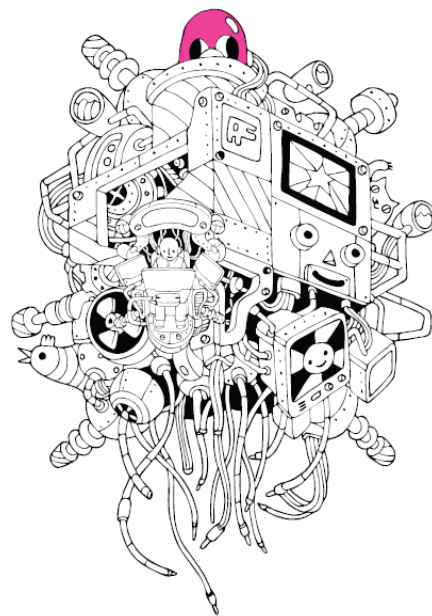
2칸 오른쪽 이동의 결과: -4

3칸 오른쪽 이동의 결과: -2



Chapter 이가 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 05. 상수와 기본 자료형

윤성우의 열혈 C 프로그래밍



Chapter 05-1. C언어가 제공하는 기본 자료형의 이해

윤성우 저 열혈강의 C 프로그래밍 개정판

자료형은 데이터를 표현하는 방법입니다.

실수를 저장할 것이냐? 정수를 저장할 것이냐!

- 값을 저장하는 방식이 실수냐 정수냐에 따라서 달라지기 때문에 용도를 결정해야 한다.

얼마나 큰 수를 저장할 것이냐!

- 큰 수를 표현하기 위해서는 많은 수의 바이트가 필요하다.

이름 이외에 메모리 공간
의 할당에 있어서 필요한
두 가지 정보



요청의 예

"아! 제가 정수를 저장할건데요. 크기는 4바이트로 하려고 합니다. 그 정도면 충분할
거예요. 그리고 변수의 이름은 num으로 할게요."

제대로 된 요청



C언어에서의 예

```
int num;
```

동일한 요청

자료형의 수는 데이터 표현방법의 수를 뜻한다. C언어가 제공하는 기본 자료형의 수가 10개라면, C언어가 제공하는 기본적인 데이터 표현방식의 수는 10개라는 뜻이 된다.



기본 자료형의 종류와 데이터의 표현범위

자료형		크기	값의 표현범위
정수형	char	1바이트	-128이상 +127이하
	short	2바이트	-32,768이상 +32,767이하
	int	4바이트	-2,147,483,648이상 +2,147,483,647이하
	long	4바이트	-2,147,483,648이상 +2,147,483,647이하
	long long	8바이트	-9,223,372,036,854,775,808이상 +9,223,372,036,854,775,807이하
실수형	float	4바이트	$\pm 3.4 \times 10^{-37}$ 이상 $\pm 3.4 \times 10^{+38}$ 이하
	double	8바이트	$\pm 1.7 \times 10^{-307}$ 이상 $\pm 1.7 \times 10^{+308}$ 이하
	long double	8바이트 이상	double 이상의 표현범위

- 컴파일러에 따라서 약간의 차이를 보인다.

C의 표준에서는 자료형별 상대적 크기를 표준화 할 뿐 구체적인 크기까지 언급하지는 않는다.

- 크게 정수형과 실수형으로 나뉜다.

데이터를 표현하는 방식이 정수형과 실수형 두 가지로 나뉘므로!

- 정수형에도 실수형에도 둘 이상의 기본 자료형이 존재한다.

표현하고자 하는 값의 크기에 따라서 적절히 선택할 수 있도록 다수의 자료형이 제공!



연산자 sizeof를 이용한 바이트 크기의 확인

```
int main(void)
{
    int num = 10;
    int sz1 = sizeof(num);
    int sz2 = sizeof(int);
    . . . .
}
```

변수 num과 int의 크기를 계산하여
그 결과로 sz1과 sz2를 초기화

sizeof 연산자의 피연산자로는 변수, 상수 및 자료형의 이름 등이 올 수 있다.
소괄호는 int와 같은 자료형의 이름에만 필수! 하지만 모든 피연산자를 대상으로
소괄호를 감싸주는 것이 일반적!

```
int main(void)
{
    char ch=9;
    int inum=1052;
    double dnum=3.1415;
    printf("변수 ch의 크기: %d \n", sizeof(ch));
    printf("변수 inum의 크기: %d \n", sizeof(inum));
    printf("변수 dnum의 크기: %d \n", sizeof(dnum));

    printf("char의 크기: %d \n", sizeof(char));
    printf("int의 크기: %d \n", sizeof(int));
    printf("long의 크기: %d \n", sizeof(long));
    printf("long long의 크기: %d \n", sizeof(long long));
    printf("float의 크기: %d \n", sizeof(float));
    printf("double의 크기: %d \n", sizeof(double));
    return 0;
}
```

실행결과

```
변수 ch의 크기: 1
변수 inum의 크기: 4
변수 dnum의 크기: 8
char의 크기: 1
int의 크기: 4
long의 크기: 4
long long의 크기: 8
float의 크기: 4
double의 크기: 8
```

정수의 표현 및 처리를 위한 일반적 자료형 선택

· 일반적인 선택은 int이다.

CPU가 연산하기에 가장 적합한 데이터의 크기가 int형의 크기로 결정된다.

연산이 동반이 되면 int형으로 형 변환이 되어서 연산이 진행된다.

따라서 연산을 동반하는 변수의 선언을 위해서는 int로 선언하는 것이 적합하다.

· char형 short형 변수는 불필요한가?

연산을 수반하지 않으면서(최소한의 연산만 요구가 되면서) 많은 수의 데이터를 저장해야 한다면,

그리고 그 데이터의 크기가 char 또는 short로 충분히 표현 가능하다면,

char 또는 short 로 데이터를 표현 및 저장하는 것이 적절하다.

```
int main(void)
{
    char num1=1, num2=2, result1=0;
    short num3=300, num4=400, result2=0;

    printf("size of num1 & num2: %d, %d \n", sizeof(num1), sizeof(num2));
    printf("size of num3 & num4: %d, %d \n", sizeof(num3), sizeof(num4));
    printf("size of char add: %d \n", sizeof(num1+num2));
    printf("size of short add: %d \n", sizeof(num3+num4));

    result1=num1+num2;
    result2=num3+num4;
    printf("size of result1 & result2: %d, %d \n", sizeof(result1), sizeof(result2));
    return 0;
}
```

+ 연산결과의 크기가 4바이트인 이유는
피연산자가 4바이트 데이터로 형 변환
되었기 때문이다.

실행결과

```
size of num1 & num2: 1, 1
size of num3 & num4: 2, 2
size of char add: 4
size of short add: 4
size of result1 & result2: 1, 2
```

실수의 표현 및 처리를 위한 일반적 자료형 선택

· 실수 자료형의 선택기준은 정밀도

실수의 표현범위는 float, double 둘 다 충분히 넓다.

그러나 8바이트 크기의 double이 float보다 더 정밀하게 실수를 표현한다.

· 일반적인 선택은 double이다.

컴퓨팅 환경의 발전으로 double형 데이터의 표현 및 연산이 덜 부담스럽다.

float형 데이터의 정밀도는 부족한 경우가 많다.

실수 자료형	소수점 이하 정밀도	바이트 수
float	6자리	4
double	15자리	8
long double	18자리	12

```
int main(void)
{
    double rad;
    double area;
    printf("원의 반지름 입력: ");
    scanf("%lf", &rad);

    area = rad*rad*3.1415;
    printf("원의 넓이: %f \n", area);
    return 0;
}
```

double형 변수의 출력 서식문자 %f - printf

double형 변수의 입력 서식문자 %lf - scanf

실행결과

원의 반지름 입력: 2.4

원의 넓이: 18.095040

unsigned를 붙여서 0과 양의 정수만 표현

정수 자료형	크기	값의 표현범위
char	1바이트	-128이상 +127이하
unsigned char		0이상 (128 + 127)이하
short	2바이트	-32,768이상 +32,767이하
unsigned short		0이상 (32,768 + 32,767)이하
int	4바이트	-2,147,483,648이상 +2,147,483,647이하
unsigned int		0이상 (2,147,483,648 + 2,147,483,647)이하
long	4바이트	-2,147,483,648이상 +2,147,483,647이하
unsigned long		0이상 (2,147,483,648 + 2,147,483,647)이하
long long	8바이트	-9,223,372,036,854,775,808이상 +9,223,372,036,854,775,807이하
unsigned long long		0이상 (9,223,372,036,854,775,808 + 9,223,372,036,854,775,807)이하

- 정수 자료형의 이름 앞에는 **unsigned** 선언을 붙일 수 있다.
- **unsigned**가 붙으면, MSB도 데이터의 크기를 표현하는데 사용이 된다.
- 따라서 표현하는 값의 범위가 양의 정수로 제한이 되며 양의 정수로 두 배 늘어난다.



윤성우의 열혈 C 프로그래밍



Chapter 05-2. 문자의 표현방식과 문자를 위한 자료형

윤성우 저 열혈강의 C 프로그래밍 개정판

문자의 표현을 위한 약속! 아스키(ASCII) 코드!

아스키 코드	아스키 코드 값
A	65
B	66
C	67
,	96
~	126

미국 표준 협회(ANSI: American National Standards Institute)

에 의해서 제정된

‘아스키(ASCII: American Standard Code for Information Interchange) 코드’

컴퓨터는 문자를 표현 및 저장하지 못한다. 따라서 문자를 표현을 목적으로 각 문자에 고유한 숫자를 지정한다.
인간이 입력하는 문자는 해당 문자의 숫자로 변환이 되어 컴퓨터에 저장 및 인식이 되고,
컴퓨터에 저장된 숫자는 문자로 변환이 되어 인간의 눈에 보여지게 된다.

```
int main(void)
{
    char ch1 = 'A';
    char ch2 = 'C';
    . . . . .
}
```



```
int main(void)
{
    char ch1 = 65;
    char ch2 = 67;
    . . . . .
}
```

컴파일 시 각 문자는
해당 아스키 코드 값으로 변환

따라서 실제로 컴퓨터에게 전달되는 데이
터는 문자가 아닌 숫자이다.

C 프로그램상에서 문자는 작은 따옴표로 묶어서 표현

문자는 이렇게 표현되는 거구나!

```
int main(void)
{
    char ch1='A', ch2=65;
    int ch3='Z', ch4=90;

    printf("%c %d \n", ch1, ch1);
    printf("%c %d \n", ch2, ch2);
    printf("%c %d \n", ch3, ch3);
    printf("%c %d \n", ch4, ch4);
    return 0;
}
```

서식문자 %c 해당 숫자의 아스키 코드 문자를 출력해라!

A 65

A 65

Z 90

Z 90

실행결과

· 문자를 char형 변수에 저장하는 이유

모든 아스키 코드 문자는 1바이트로도 충분히 표현가능

문자는 덧셈, 뺄셈과 같은 연산을 동반하지 않는다. 단지 표현에 사용될 뿐이다.

따라서 1바이트 크기인 char형 변수가 문자를 저장하기 최적의 장소이다.

문자는 int형 변수에도 저장이 가능하다.



A collection of various cartoon objects including a cat, a fish, a laptop, a robot, a planet, a key, a die, a TV, a camera, a ring, and a small alien.

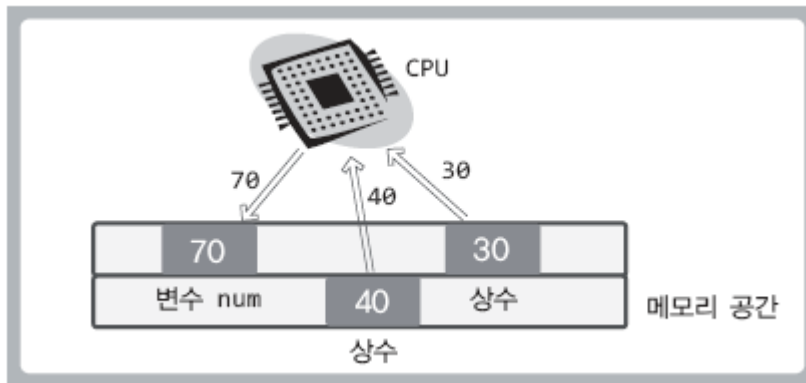
윤성우 저 열혈강의 C 프로그래밍 개정판

이름을 지니지 않는 리터럴 상수!

```
int main(void)
{
    int num = 30 + 40;
    . . . .
}
```

연산을 위해서는 30, 40과 같이 프로그램상에 표현되는 숫자도 메모리 공간에 저장되어야 한다.

이렇게 저장되는 값은 이름이 존재하지 않으니 변경이 불가능한 상수이다. 따라서 **리터럴 상수**라 한다.



- 단계 1. 정수 30과 40이 메모리 공간에 상수의 형태로 저장된다.
- 단계 2. 두 상수를 기반으로 덧셈이 진행된다.
- 단계 3. 덧셈의 결과로 얻어진 정수 70이 변수 num에 저장된다.

메모리 공간에 저장이 되어야 CPU의 연산대상이 된다.

리터럴 상수의 자료형

```
int main(void)
{
    printf("literal int size: %d \n", sizeof(7));
    printf("literal double size: %d \n", sizeof(7.14));
    printf("literal char size: %d \n", sizeof('A'));
    return 0;
}
```

실행결과

```
literal int size: 4
literal double size: 8
literal char size: 4
```

리터럴 상수도 자료형이 결정되어야 메모리 공간에 저장될 수 있다.

위 예제의 실행결과는 다음 사실을 의미한다.

- 정수는 기본적으로 **int**형으로 표현된다.
- 실수는 기본적으로 **double**형으로 표현된다.
- 문자는 기본적으로 **int**형으로 표현된다.



접미사를 이용한 다양한 상수의 표현

```
int main(void)
{
    float num1 = 5.789;    // 경고 메시지 발생
    float num2 = 3.24 + 5.12; // 경고 메시지 발생
    return 0;
}
```

실수는 **double**형 상수로 인식이 되어
데이터 손실에 대한 경고 메시지 발생

```
float num1 = 5.789f;    // 경고 메시지 발생 안 함
float num2 = 3.24F + 5.12F; // 소문자 f 대신 대문자 F를 써도 된다!
```

접미사를 통해서 상수의 자료형을 변경
할 수 있다.

접미사	자료형	사용의 예
U	unsigned int	unsigned int n = 1025U
L	long	long n = 2467L
UL	unsigned long	unsigned long n = 3456UL
LL	long long	long long n = 5768LL
ULL	unsigned long long	unsigned long long n = 8979ULL

[표 05-4: 정수형 상수의 표현을 위한 접미사]

접미사	자료형	사용의 예
F	float	float f = 3.15F
L	long double	long double f = 5.789L

[표 05-5: 실수형 상수의 표현을 위한 접미사]

이름을 지니는 심볼릭(Symbolic) 상수: const 상수

```
int main(void)
{
    const int MAX=100;    // MAX는 상수! 따라서 값의 변경 불가!
    const double PI=3.1415; // PI는 상수! 따라서 값의 변경 불가!
    . . . .
}
```

```
int main(void)
{
    const int MAX;    // 쓰레기 값으로 초기화 되어버림
    MAX=100;    // 값의 변경 불가! 따라서 컴파일 에러 발생!
    . . . .
}
```

상수의 이름은

모두 대문자로 표시하고,

둘 이상의 단어를 연결할 때에는 MY_AGE와 같이 언더바를 이용해서 두 단어를 구분하는 것이 관례!

윤성우의 열혈 C 프로그래밍



Chapter 05-4. 자료형의 변환

윤성우 저 열혈강의 C 프로그래밍 개정판

대입 연산의 과정에서 발생하는 자동 형 변환

```
double num1=245;    // int형 정수 245를 double형으로 자동 형 변환
int num2=3.1415;    // double형 실수 3.1415를 int형으로 자동 형 변환
```

대입 연산자의 왼편을 기준으로
형 변환이 발생한다.

정수 245는 245.0의 비트 열로 재구성이 되어 변수 num1에 저장된다.

실수 3.1415는 int형 데이터 3의 비트 열로 재구성이 되어 변수 num2에 저장된다.

```
int num3=129;
char ch=num3;    // int형 변수 num3에 저장된 값이 char형으로 자동 형 변환
```

4바이트 변수 num3에 저장된 4바이트 데이터 중 상위 3바이트가 손실되어 변수 ch에 저장된다.

00000000 00000000 00000000 10000001 ➡ 10000001

자동 형 변환의 방식 정리

형 변환의 방식에 대한 유형별 정리

- 정수를 실수로 형 변환 3은 3.0으로 5는 5.0으로(오차가 발생하게 된다).
- 실수를 정수로 형 변환 소수점 이하의 값이 소멸된다.
- 큰 정수를 작은 정수로 형 변환 작은 정수의 크기에 맞춰서 상위 바이트가 소멸된다.

```
int main(void)
{
    double num1=245;
    int num2=3.1415;
    int num3=129;
    char ch=num3;

    printf("정수 245를 실수로: %f \n", num1);
    printf("실수 3.1415를 정수로: %d \n", num2);
    printf("큰 정수 129를 작은 정수로: %d \n", ch);
    return 0;
}
```

실행결과

```
정수 245를 실수로: 245.000000
실수 3.1415를 정수로: 3
큰 정수 129를 작은 정수로: -127
```

정수의 승격에 의한 자동 형 변환

일반적으로 CPU가 처리하기에 가장 적합한 크기의 정수 자료형을 int로 정의한다.
따라서 int형 연산의 속도가 다른 자료형의 연산속도에 비해서 동일하거나 더 빠르다.



따라서 다음과 같은 방식의
형 변환 발생

```
int main(void)
{
    short num1=15, num2=25;
    short num3=num1+num2;    // num1과 num2가 int형으로 형 변환
    . . . .
}
```

이를 가리켜 '정수의 승격(Integral Promotion)'이라 한다.



피연산자의 자료형 불일치로 발생하는 자동 형 변환

```
double num1 = 5.15 + 19;
```

두 피연산자의 자료형은 일치해야 한다. 일치하지 않으면 일치하기 위해서 자동으로 형 변환이 발생한다.

아래의 자동 형 변환 규칙을 근거로 int형 데이터 19가 double형 데이터 19.0으로 형 변환이 되어 덧셈이 진행된다.



산술연산에서의
자동 형 변환 규칙

바이트 크기가 큰 자료형이 우선시 된다.

정수형보다 실수형을 우선시 한다.

이는 데이터의 손실을 최소화 하기 위한 기준이다.

명시적 형 변환: 강제로 일으키는 형 변환

```
int main(void)
{
    int num1=3, num2=4;
    double divResult;
    divResult = num1 / num2;
    printf("나눗셈 결과: %f \n", divResult);
    return 0;
}
```

num1과 num2가 정수이기 때문에 몫만 반환이 되는
정수형 나눗셈이 진행

실행결과

나눗셈 결과: 0.000000

divResult = (double)num1 / num2;

(type)은 type형으로의 형 변환을 의미한다.

num1이 double형으로 명시적 형 변환 그리고 num1과 num2의 / 연산 과정에서의 산술적 자동 형 변환!
그 결과 실수형 나눗셈이 진행되어 divResult에는 0.75가 저장된다.

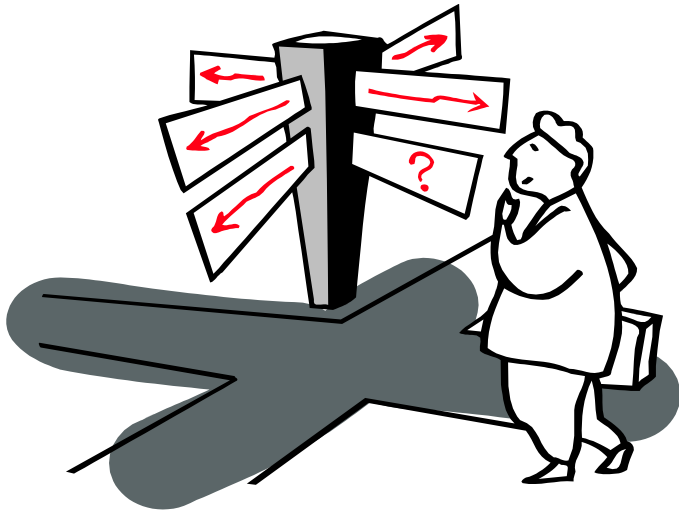
```
int main(void)
{
    int num1 = 3;
    double num2 = 2.5 * num1;
    . . . .
}
```



```
int main(void)
{
    int num1 = 3;
    double num2 = 2.5 * (double)num1;
    . . . .
}
```

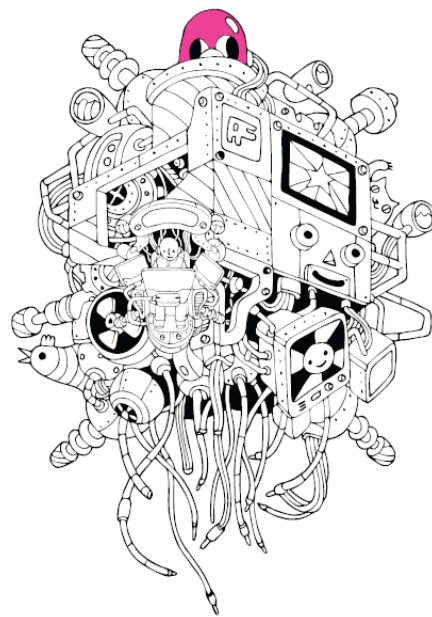
추천하는 코드 작성 스타일

자동 형 변환이 발생하는 위치에 명시적 형 변환 표시를 해서 형 변환이 발생함을 알리는 것이 좋다!



Chapter 05가 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 06. printf 함수와 scanf 함수 정리하기

A collection of various cartoon objects including a cat, a fish, a laptop, a robot, a planet, a key, a die, a TV, a camera, a ring, and a small alien.

윤성우 저 열혈강의 C 프로그래밍 개정판

printf 함수와 특수문자

```
int main(void)
{
    printf("I like programming \n");
    printf("I love puppy! \n");
    printf("I am so happy \n");
    return 0;
}
```

실행결과

```
I like programming
I love puppy!
I am so happy
```

printf 함수는 첫 번째 인자로 전달된 문자열을 출력한다.

```
printf("앞집 강아지가 말했다. "멍~! 멍~!" 정말 귀엽다.");
```

잘못된
printf 함수 호출문

"앞집 강아지가 말했다. " → 음 이것은 하나의 문자열이군!
멍~! 멍~! → 이건 뭐지?
" 정말 귀엽다." → 이것도 하나의 문자열이고!

큰 따옴표는 문자열의 시작과 끝으로 해석이 되니, 큰 따옴표 자체의 출력을 원하는 경우에는 큰 따옴표 앞에 \ 문자를 붙여주기로 하자!

컴파일러의 오해?

특수문자의 탄생 배경

```
printf("앞집 강아지가 말했다. \"멍~! 멍~!\" 정말 귀엽다.");
```

제대로 된
printf 함수 호출문

특수문자의 종류

특수문자	의미하는 바
\a	경고음
\b	백스페이스(backspace)
\f	폼 피드(form feed)
\n	개 행(new line)
\r	캐리지 리턴(carriage return)
\t	수평 탭
\v	수직 탭
\'	작은 따옴표 출력
\"	큰 따옴표 출력
\?	물음표 출력
\\	역슬래쉬 출력

\f와 \v는 모니터 출력이 아닌 프린터 출력을 위해 정의된 특수문자이기 때문에 모니터의 출력에 사용하면, 이상한 문자 출력!



printf 함수의 서식지정과 서식문자들

```
int main(void)
{
    int myAge=12;
    printf("제 나이는 10진수로 %d살, 16진수로 %X살입니다. \n", myAge, myAge);
    return 0;
}
```

서식문자를 이용해서 출력할 문자열의 형태를 조합해 낼 수 있다.

즉, 출력의 서식을 지정할 수 있다.

제 나이는 10진수로 12살, 16진수로 C살입니다.

실행결과

```
int main(void)
{
    int num1=7, num2=13;
    printf("%o %#o \n", num1, num1);
    printf("%x %#x \n", num2, num2);
    return 0;
}
```

7 07

d 0xd

실행결과

#을 삽입하면 8진수 앞에 0, 16진수 앞에 0x가 삽입된다.

서식문자	출력 대상(자료형)	출력 형태
%d	char, short, int	부호 있는 10진수 정수
%ld	long	부호 있는 10진수 정수
%lld	long long	부호 있는 10진수 정수
%u	unsigned int	부호 없는 10진수 정수
%o	unsigned int	부호 없는 8진수 정수
%x, %X	unsigned int	부호 없는 16진수 정수
%f	float, double	10진수 방식의 부동소수점 실수
%Lf	long double	10진수 방식의 부동소수점 실수
%e, %E	float, double	e 또는 E 방식의 부동소수점 실수
%g, %G	float, double	값에 따라 %f와 %e 사이에서 선택
%c	char, short, int	값에 대응하는 문자
%s	char *	문자열
%p	void *	포인터의 주소 값

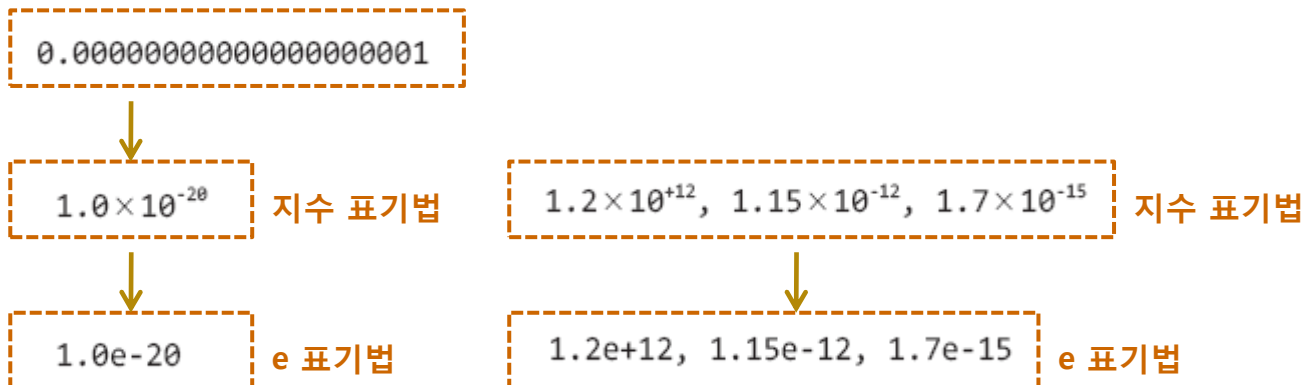
실수의 출력을 위한 서식문자들: %f, %e

```
int main(void)
{
    printf("%f \n", 0.1234);
    printf("%e \n", 0.1234);    // e 표기법 기반의 출력
    printf("%f \n", 0.12345678);
    printf("%e \n", 0.12345678);    // e 표기법 기반의 출력
    return 0;
}
```

실행결과

```
0.123400
1.234000e-001
0.123457
1.234568e-001
```

컴퓨터는 지수를 표현할 수 없으므로
e 표기법으로 지수를 대신 표현한다.



%g의 실수출력과 %s의 문자열 출력

```
int main(void)
{
    double d1=1.23e-3;    // 0.00123
    double d2=1.23e-4;    // 0.000123
    double d3=1.23e-5;    // 0.0000123
    double d4=1.23e-6;    // 0.00000123

    printf("%g \n", d1);   // %f 스타일 출력
    printf("%g \n", d2);   // %f 스타일 출력
    printf("%g \n", d3);   // %e 스타일 출력
    printf("%g \n", d4);   // %e 스타일 출력
    return 0;
}
```

%g는 실수의 형태에 따라서 %f와 %e 사이에서 적절한 형태의 출력을 진행한다.

%g와 %G의 차이점은 e 표기법의 e를 소문자로 출력하느냐 대문자로 출력하느냐에 있다.

실행결과

```
0.00123
0.000123
1.23e-005
1.23e-006
```

```
int main(void)
{
    printf("%s, %s, %s \n", "AAA", "BBB", "CCC");
    return 0;
}
```

실행결과

```
AAA, BBB, CCC
```

%s의 문자열 출력과 관련해서는 배열과 포인터 공부 후에 완벽히 이해하자!

일단은 %s의 사용법을 예제 기반으로 이해하자.

필드 폭을 지정하여 정돈된 출력 보이기

%8d

필드 폭을 8칸 확보하고, 오른쪽 정렬해서 출력을 진행한다.

%-8d

필드 폭을 8칸 확보하고, 왼쪽 정렬해서 출력을 진행한다.

```
int main(void)
{
    printf("%-8s %14s %5s \n", "이 름", "전공학과", "학년");
    printf("%-8s %14s %5d \n", "김동수", "전자공학", 3);
    printf("%-8s %14s %5d \n", "이을수", "컴퓨터공학", 2);
    printf("%-8s %14s %5d \n", "한선영", "미술교육학", 4);
    return 0;
}
```

실행결과

이 름	전공학과	학년
김동수	전자공학	3
이을수	컴퓨터공학	2
한선영	미술교육학	4

서식문자 사이에 들어가는 숫자는 필드의 폭을 의미한다.

기본 오른쪽 정렬이다. 따라서 -는 왼쪽 정렬을 의미하는 용도로 사용된다.

윤성우의 열혈 C 프로그래밍



Chapter 06-2. scanf 함수 이야기

윤성우 저 열혈강의 C 프로그래밍 개정판

정수 기반의 입력형태 정의하기

입력의 형식 어떻게 받아들일 거니?

입력의 장소 어디에 저장할까?

데이터를 입력 받는

scanf 함수에게 전달해야 할 두 가지 정보

%d 10진수 정수의 형태로 데이터를 입력 받는다.

%o 8진수 양의 정수의 형태로 데이터를 입력 받는다.

%x 16진수 양의 정수의 형태로 데이터를 입력 받는다.

서식문자의 의미는 출력을 입력으로만 변경
하면 printf 함수와 유사하다.

```
int main(void)
{
    int num1, num2, num3;
    printf("세 개의 정수 입력: ");
    scanf("%d %o %x", &num1, &num2, &num3);
    printf("입력된 정수 10진수 출력: ");
    printf("%d %d %d \n", num1, num2, num3);
    return 0;
}
```

실행결과

세 개의 정수 입력: 12 12 12

입력된 정수 10진수 출력: 12 10 18

실수 기반의 입력형태 정의하기

float형 데이터의 삽입을 위한 서식문자

printf 함수에서는 서식문자 **%f**, **%e** 그리고 **%g**의 의미가 각각 달랐다.
그러나 **scanf** 함수에서는 'float형 데이터를 입력 받겠다'는 동일한 의미를 담고 있다.

double형 long double형 데이터의 삽입을 위한 서식문자

%lf	double	%f 에 L 이 추가된 형태
%Lf	long double	%f 에 L 이 추가된 형태

float, double, long double의 데이터 출력 **%f**, **%f**, **%Lf**

float, double, long double의 데이터 입력 **%f**, **%lf**, **%Lf**

실행결과

실수의 입력과정에서
e 표기법을 사용해도 된다.

```
실수 입력1(e 표기법으로): 1.1e-3
입력된 실수 0.001100
실수 입력2(e 표기법으로): 0.1e+2
입력된 실수 10.000000
실수 입력3(e 표기법으로): 0.17e-4
입력된 실수 0.000017
```

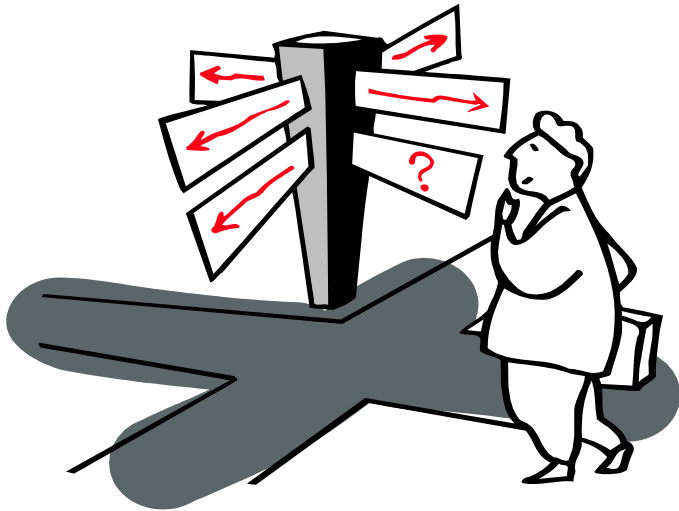
```
int main(void)
{
    float num1;
    double num2;
    long double num3;
    printf("실수 입력1(e 표기법으로): ");
    scanf("%f", &num1);
    printf("입력된 실수 %f \n", num1);

    printf("실수 입력2(e 표기법으로): ");
    scanf("%lf", &num2);
    printf("입력된 실수 %f \n", num2);

    printf("실수 입력3(e 표기법으로): ");
    scanf("%Lf", &num3);
    printf("입력된 실수 %Lf \n", num3);
    return 0;
}
```

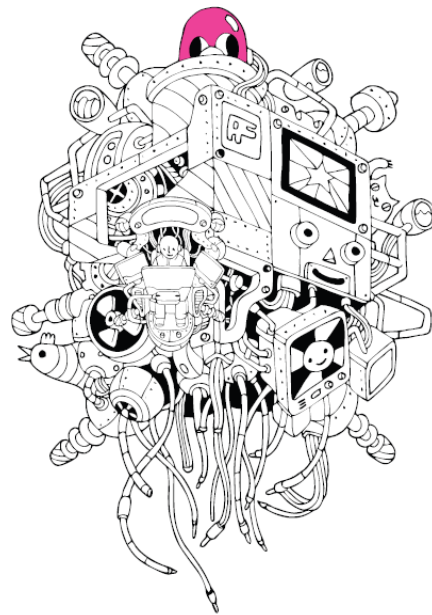

%s는 문자열의 입출력에 모두 사용된다는 사실 정도는 기억하고 있자!

이와 관련된 이해는 배열과 포인터를 공부한 다음으로 미루자!



Chapter 06이 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 07. 반복실행을 명령하는 반복문

윤성우의 열혈 C 프로그래밍



Chapter 07-1. while문에 의한
문장의 반복

윤성우 저 열혈강의 C 프로그래밍 개정판

반복문의 이해와 while문

· 반복문이란

하나 이상의 문장을 두 번 이상 반복 실행하기 위해서 구성하는 문장

· 반복문의 종류

while, do~while, for

반복의 대상이 한 문장이면 중괄호 생략 가능

```
while(num<5)
    printf("Hello world! %d \n", num++);
```

```
while(num<5)
    printf("Hello world! %d \n", num), num++;
```

```
int main(void)
{
    int num=0;
    while(num<5)
    {
        printf("Hello world! %d \n", num);
        num++;
    }
    return 0;
}
```

while 반복문

중괄호 내부 반복영역

반복의 목적이 되는 대상

변수 num은 반복의 횟수를 조절하기 위한 것!

```
Hello world! 0
Hello world! 1
Hello world! 2
Hello world! 3
Hello world! 4
```

실행결과

반복문 안에서도 들여쓰기 합니다.

들여쓰기를 하지 않은 것

```
int main(void)
{
int num=0;
while(num<5)
{
printf("Hello world! %d \n", num);
num++;
}
return 0;
}
```

들여쓰기를 한 것

```
int main(void)
{
    int num=0;
    while(num<5)
    {
        printf("Hello world! %d \n", num);
        num++;
    }
    return 0;
}
```

들여쓰기를 한 것과 하지 않은 것의 차이가 쉽게 눈에 들어온다!

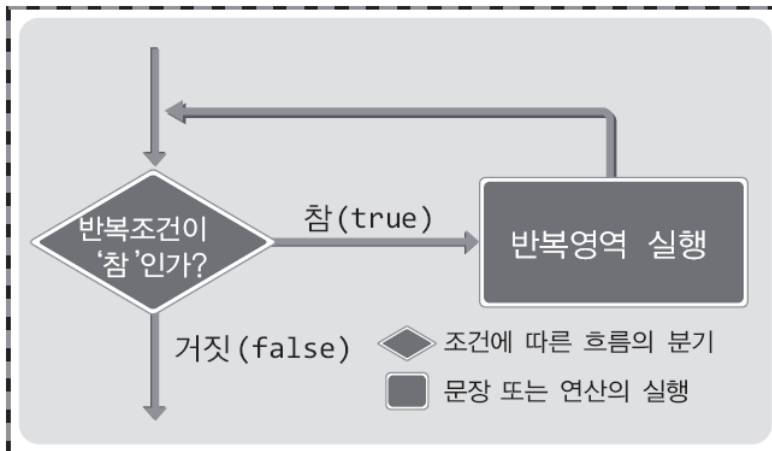


while문의 구성과 실행흐름의 세세한 관찰

```
int main(void)
{
    int num=0;
    while(num<3)  // 3회 반복
    {
        printf("Hello world! %d \n", num);
        num++;
    }
    . . . .
}
```



반복의 과정은?



flow chart 기준에서의 while문

구구단의 출력

```
int main(void)
{
    int dan=0, num=1;
    printf("몇 단? ");
    scanf("%d", &dan);

    while(num<10)
    {
        printf("%d×%d=%d \n", dan, num, dan*num);
        num++;
    }
    return 0;
}
```

```
몇 단? 7
7×1=7
7×2=14
7×3=21
7×4=28
7×5=35
7×6=42
7×7=49
7×8=56
7×9=63
```

실행결과

구구단은 반복문을 이해하는데 사용되는 대표적인 예제이다.

이후에 반복문의 중첩에서는 구구단 전체를 출력하는 예제를 접한다.

무한루프의 구성

```
while( 1 )
{
    printf("%d×%d=%d \n", dan, num, dan*num);
    num++;
}
```

숫자 1은 '참'을 의미하므로 반복문의 조건은 계속해서 '참'이 된다.

이렇듯 반복문의 탈출조건이 성립하지 않는 경우 무한루프를 형성한다고 한다.

이러한 무한루프는 실수로 만들어지는 경우도 있지만, break문과 함께 유용하게 사용되기도 한다.



while문의 중첩

while문 안에 while문이 존재하는 상태를 의미한다. 아래의 예제에서는 while문을 중첩시켜서 구구단 전체를 출력한다. 이 예제를 통해서 중첩된 while문의 코드 흐름을 이해하자.

```
int main(void)
{
    int cur=2;
    int is=0;

    while(cur<10) // 2단부터 9단까지 반복
    {
        is=1; // 새로운 단의 시작을 위해서
        while(is<10) // 각 단의 1부터 9의 곱을 표현
        {
            printf("%d×%d=%d \n", cur, is, cur*is);
            is++;
        }
        cur++; // 다음 단으로 넘어가기 위한 증가
    }

    return 0;
}
```

바깥쪽 while문

안쪽 while문



윤성우의 열혈 C 프로그래밍

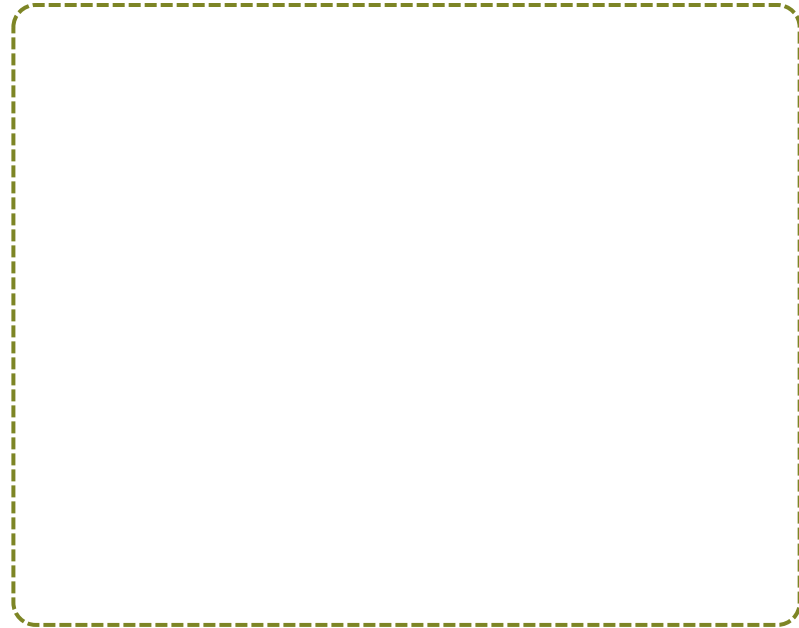


Chapter 07-2. do~while문에 의한
문장의 반복

윤성우 저 열혈강의 C 프로그래밍 개정판

do~while문의 기본구성

```
do
{
    printf("Hello world! \n");
    num++;
} while(num<3);
```



반복의 과정은?

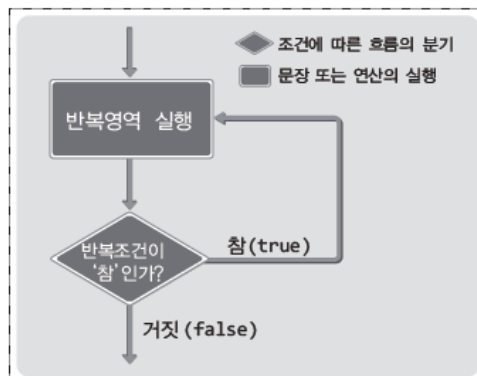
반복조건을 반복문의 마지막에 진행하는 형태이기 때문에
최소한 1회는 반복영역을 실행하게 된다. 이것이 while문과의 가장 큰 차이점이다.

do~while문이 자연스러운 상황

```
while(num<10)
{
    printf("%d×%d=%d \n", dan, num, dan*num);
    num++;
}
```

↕ 동일한 횟수를 반복하는 반복문들

```
do
{
    printf("%d×%d=%d \n", dan, num, dan*num);
    num++;
} while(num<10);
```



do~while문의 순서도

```
int main(void)
{
    int total=0, num=0;
    do
    {
        printf("정수 입력(0 to quit): ");
        scanf("%d", &num);
        total += num;
    }while(num!=0);
    printf("합계: %d \n", total);
    return 0;
}
```

정수 입력(0 to quit): 1
정수 입력(0 to quit): 2
정수 입력(0 to quit): 3
정수 입력(0 to quit): 4
정수 입력(0 to quit): 5
정수 입력(0 to quit): 0
합계: 15

실행결과

최소한 1회 이상 실행되어야 하는 반복문은
do~while문으로 구성하는 것이 자연스럽다.

윤성우의 열혈 C 프로그래밍



Chapter 07-3. for문에 의한
문장의 반복

윤성우 저 열혈강의 C 프로그래밍 개정판

반복문의 필수3요소

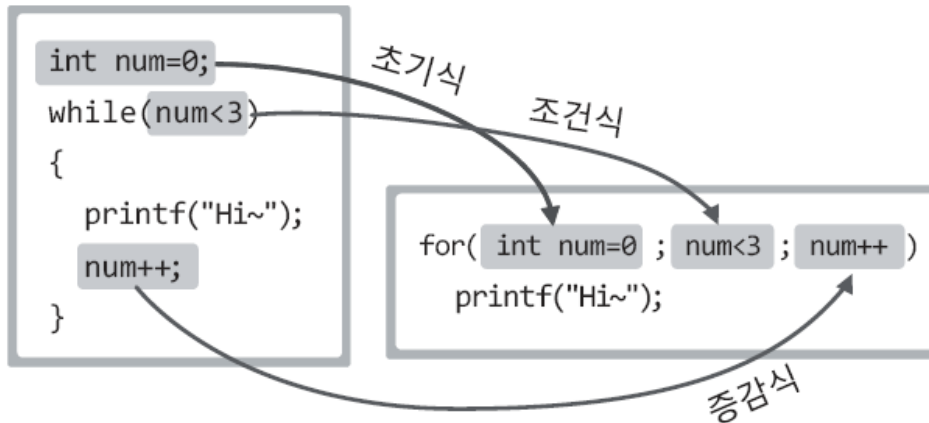
```
int main(void)
{
    int num=0;    // 필수요소 1. 반복을 위한 변수의 선언
    while(num<3)  // 필수요소 2. 반복의 조건검사
    {
        printf("Hi~");
        num++;    // 필수요소 3. 반복의 조건을 '거짓'으로 만들기 위한 연산
    }
    . . . .
}
```

정해진 횟수의 반복을 위해서는 하나의 변수가 필요하다.
 그 변수를 기반으로 하는 조건검사가 필요하다.
 조건검사가 false가 되게 하기 위한 연산이 필요하다.

이 세 가지를 한 줄에 표시하도록
 돕는 것이 for문이다.

위의 while문에서 보이듯이 반복문에 필요한 세 가지 요소가 여러 행에 걸쳐서 분산되어 있다.
 따라서 반복의 횟수가 바로 인식 불가능하다.

for문의 구조와 이해



```

for( 초기식 ; 조건식 ; 증감식 )
{
    // 반복의 대상이 되는 문장들
}
    
```

```

int main(void)
{
    int num;
    for(num=0; num<3; num++)
        printf("Hi~");
    . . .
}
    
```

일부 컴파일러는 여전히 초기식에서의 변수 선언을 허용하지 않는다.
for문의 반복영역도 한 줄이면 중괄호 생략 가능!

for문의 흐름 이해

for문의 구성요소

- ✓ 초기식 본격적으로 반복을 시작하기에 앞서 딱 한번 실행된다.
- ✓ 조건식 매 반복의 시작에 앞서 실행되며, 그 결과를 기반으로 반복유무를 결정!
- ✓ 증감식 매 반복실행 후 마지막에 연산이 이뤄진다.

for문 흐름의 핵심

int num=0에 해당하는 초기화는 반복문의 시작에 앞서 딱 1회 진행!
num<3에 해당하는 조건의 검사는 매 반복문의 시작에 앞서 진행!
num++에 해당하는 증감연산은 반복영역을 실행한 후에 진행!

첫 번째 반복의 흐름

1 → 2 → 3 → 4 [num=1]

두 번째 반복의 흐름

2 → 3 → 4 [num=2]

세 번째 반복의 흐름

2 → 3 → 4 [num=3]

네 번째 반복의 흐름

2 [num=3] 따라서 탈출!

```

1      2      4
for( int num=0 ; num<3 ; num++ )
{ 3
    printf("Hi~");
}
    
```


for문 기반의 다양한 예제

```
int main(void)
{
    int total=0;
    int i, num;
    printf("0부터 num까지의 덧셈, num은? ");
    scanf("%d", &num);

    for(i=0; i<num+1; i++)
        total+=i;

    printf("0부터 %d까지 덧셈결과: %d \n", num, total);
    return 0;
}
```

0부터 num까지의 덧셈, num은? 10
0부터 10까지 덧셈결과: 55

실행결과

다양한 예제를 통해서 for문에 익숙해지자!

오른쪽 예제에서 보이듯이 불필요하다면, 초기식, 조건식, 증감식을 생략할 수 있다.
단 조건식을 생략하면 참으로 인식이 되어 무한루프를 형성하게 된다.

실수 입력(minus to quit) : 3.2323
실수 입력(minus to quit) : 5.1891
실수 입력(minus to quit) : 2.9297
실수 입력(minus to quit) : -1.0
평균: 3.783700

실행결과

```
int main(void)
{
    double total=0.0;
    double input=0.0;
    int num=0;

    for( ; input>=0.0 ; )
    {
        total+=input;
        printf("실수 입력(minus to quit) : ");
        scanf("%lf", &input);
        num++;
    }
    printf("평균: %f \n", total/(num-1));
    return 0;
}
```

for문의 중첩

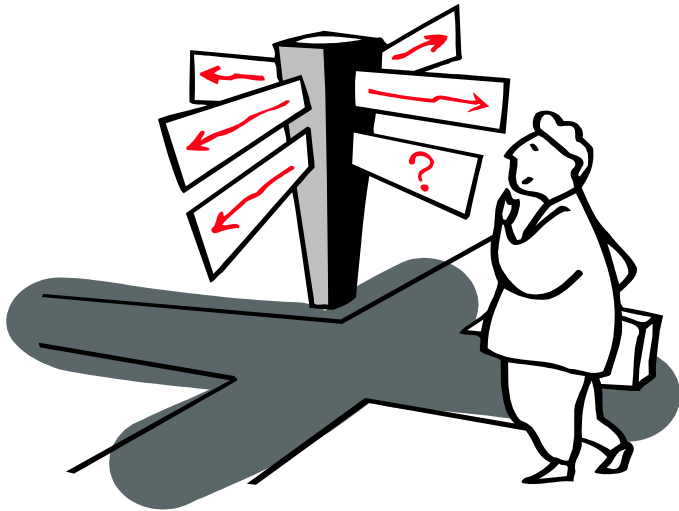
```
int main(void)
{
    int cur, is;

    for(cur=2; cur<10; cur++)
    {
        for(is=1; is<10; is++)
            printf("%d×%d=%d \n", cur, is, cur*is);
        printf("\n");
    }
    return 0;
}
```

for문의 중첩은 while, do~while문의 중첩과 다르지 않다.

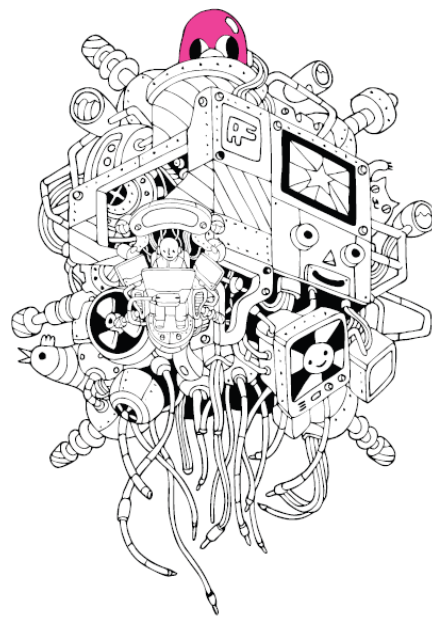
구구단 전체를 출력하는 원편의 예제를 통해서 for문의 중첩을 이해하자.





Chapter 07이 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 08. 조건에 따른 흐름의 분기

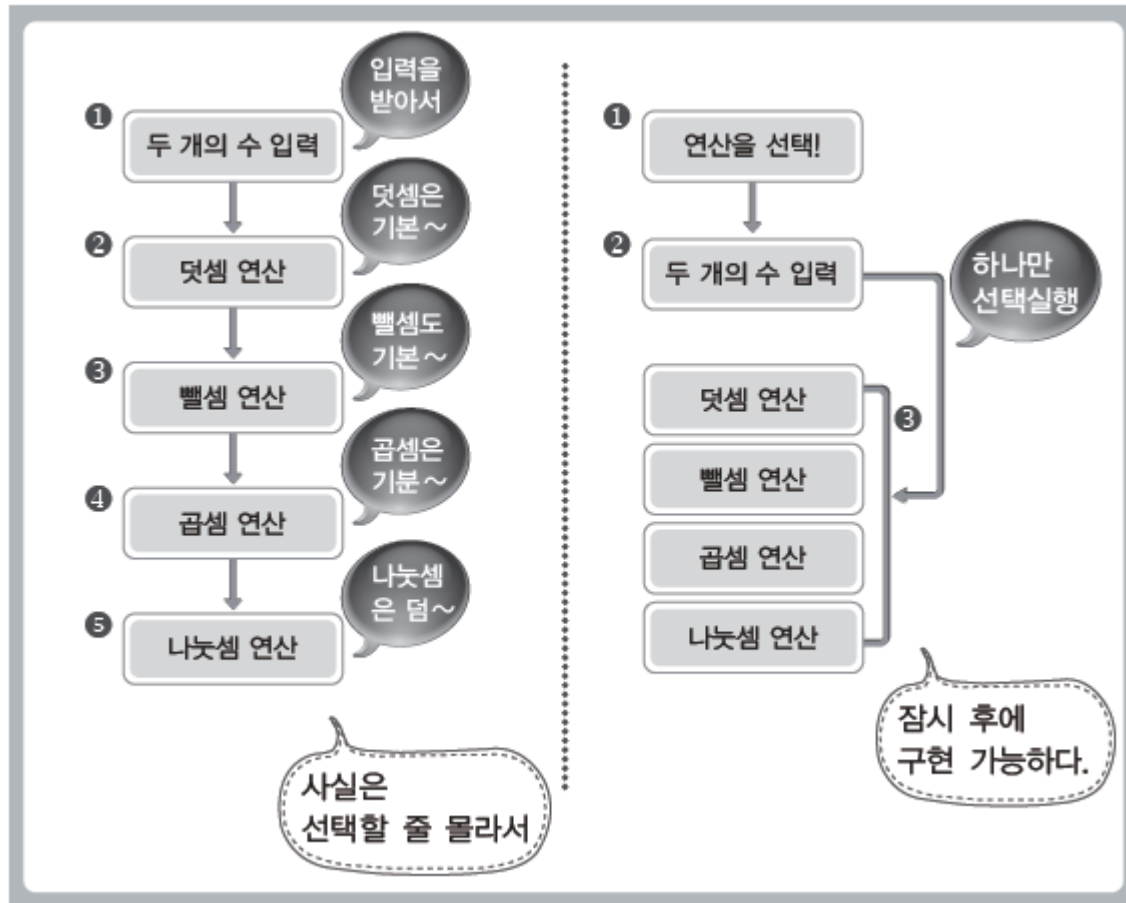
윤성우의 열혈 C 프로그래밍



Chapter 08-1. 조건적 실행과 흐름의 분기

윤성우 저 열혈강의 C 프로그래밍 개정판

흐름의 분기가 필요한 이유



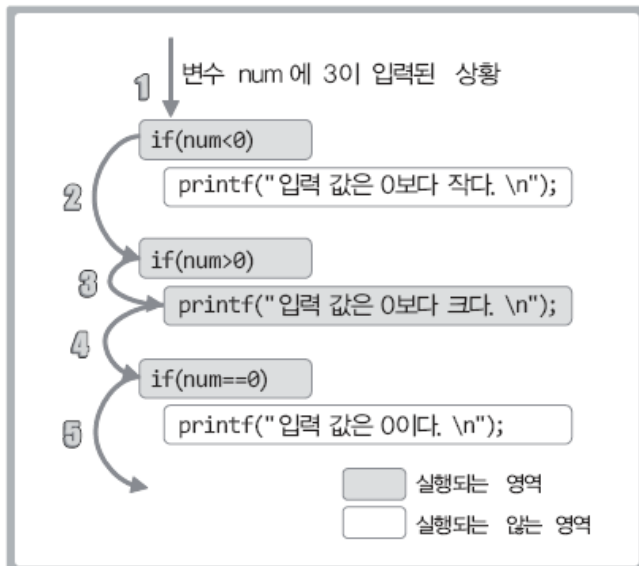
분기하지 못하면 프로그램 사용자는
사칙연산 중 하나를 선택하지 못한다!

프로그램을 구현하다 보면 상황에 따라서 선택적으로 실행해야 하는 영역도 존재하기 마련!

if문을 이용한 조건적 실행

```
if(num1>num2) num1이 num2보다 크면 실행
{
    printf("num1이 num2보다 큽니다. \n");
    printf("%d > %d \n", num1, num2);
}
```

```
if(num1>num2) 한 줄이면 중괄호 생략 가능
    printf("num1이 num2보다 큽니다. \n");
```



```
int main(void)
{
    int num;
    printf("정수 입력: ");
    scanf("%d", &num);

    if(num<0)    // num이 0보다 작으면 아래의 문장 실행
        printf("입력 값은 0보다 작다. \n");

    if(num>0)    // num이 0보다 크면 아래의 문장 실행
        printf("입력 값은 0보다 크다. \n");

    if(num==0)   // num이 0이면 아래의 문장 실행
        printf("입력 값은 0이다. \n");

    return 0;
}
```

정수 입력: 3
입력 값은 0보다 크다. *실행결과1*

정수 입력: 0
입력 값은 0이다. *실행결과2*

if문을 이용한 계산기 프로그램

```
int main(void)
{
    int opt;
    double num1, num2;
    double result;

    printf("1.덧셈 2.뺄셈 3.곱셈 4.나눗셈 \n");
    printf("선택? ");
    scanf("%d", &opt);
    printf("두 개의 실수 입력: ");
    scanf("%lf %lf", &num1, &num2);

    if(opt==1)
        result = num1 + num2;
    if(opt==2)
        result = num1 - num2;
    if(opt==3)
        result = num1 * num2;
    if(opt==4)
        result = num1 / num2;

    printf("결과: %f \n", result);
    return 0;
}
```

이제 계산기 프로그램에 실질적으로 더 가까운 형태가 되었다.

프로그램 구성상 사칙연산 중 하나만 실행이 된다. 그럼에도 불구하고 프로그램 사용자가 덧셈연산을 선택할지라도 총 4번의 조건검사(if문을 통한)를 진행한다는 불합리한 점이 존재한다.

이러한 불합리한 점의 해결에 사용되는 것이 if~else문이다.

실행결과

```
1.덧셈 2.뺄셈 3.곱셈 4.나눗셈
선택? 3
두 개의 실수 입력: 2.14 5.12
결과: 10.956800
```

예제 Mul3Mul4.c도
공부하자!

if~else문을 이용한 흐름의 분기

```
if(num1>num2)    num1이 num2보다 크면 실행
{    // if 블록
    printf("num1이 num2보다 큽니다. \n");
    printf("%d > %d \n", num1, num2);
}
else    num1이 num2보다 크지 않으면 실행
{    // else 블록
    printf("num1이 num2보다 크지 않습니다. \n");
    printf("%d <= %d \n", num1, num2);
}
```

if~else문은 하나의 문장임에 주목하자!

따라서 if와 else 사이에 다른 문장이 삽입될 수 없다.

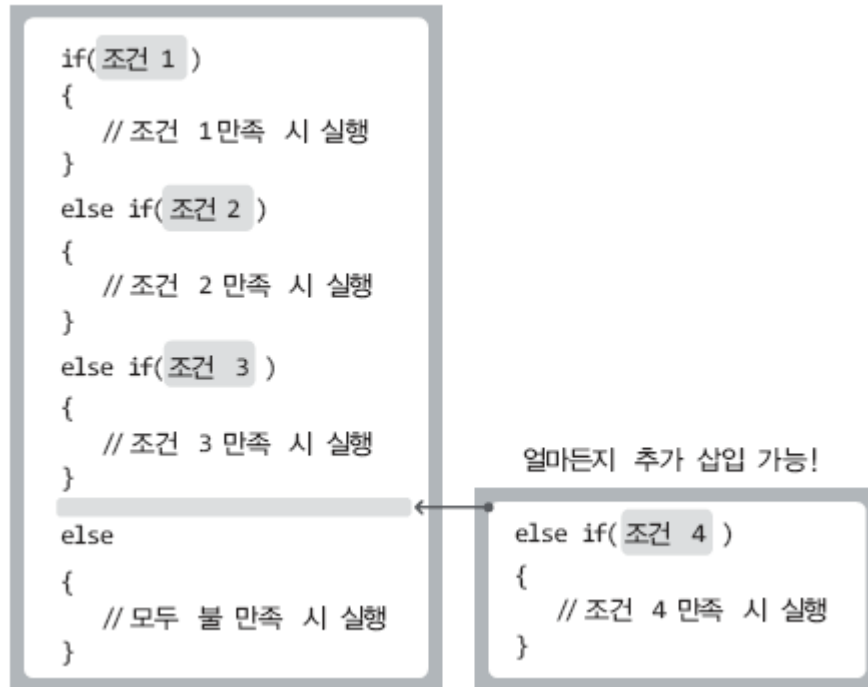
```
int main(void)
{
    int num;
    printf("정수 입력: ");
    scanf("%d", &num);
    if(num<0)
        printf("입력 값은 0보다 작다. \n");
    else
        printf("입력 값은 0보다 작지 않다. \n");

    return 0;
}
```

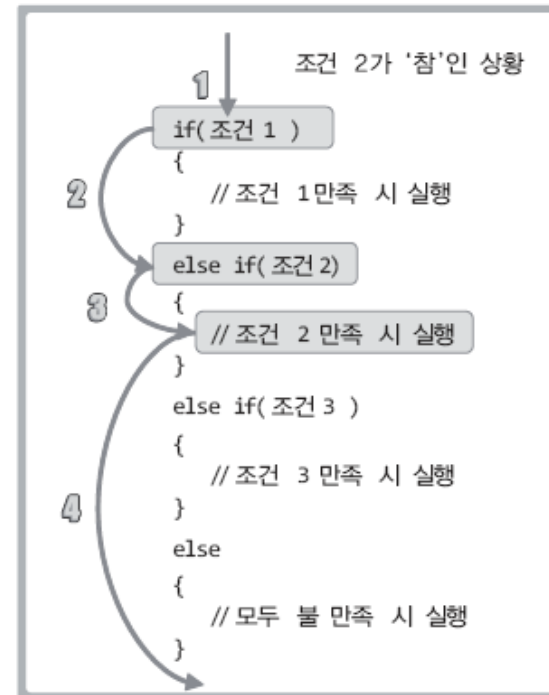
실행결과

정수 입력: 7
입력 값은 0보다 작지 않다.

if...else if...else의 구성



if...else if...else문의 구성



if...else if...else문의 흐름

if...else if...else문의 적용

```
int main(void)
{
    int opt;
    double num1, num2;
    double result;
    printf("1.덧셈 2.뺄셈 3.곱셈 4.나눗셈 \n");
    printf("선택? ");
    scanf("%d", &opt);
    printf("두 개의 실수 입력: ");
    scanf("%lf %lf", &num1, &num2);

    if(opt==1)
        result = num1 + num2;
    else if(opt==2)
        result = num1 - num2;
    else if(opt==3)
        result = num1 * num2;
    else
        result = num1 / num2;

    printf("결과: %f \n", result);
    return 0;
}
```

합리적으로 완성된 사칙연산 계산기 프로그램



if...else if...else의 진실

```
if(num<0)
    printf("입력 값은 0보다 작다. \n");
else if(num>0)
    printf("입력 값은 0보다 크다. \n");
else
    printf("입력 값은 0이다. \n");
```

```
if(num<0)
{
    printf("입력 값은 0보다 작다. \n");
}
else
{
    if(num>0)
        printf("입력 값은 0보다 크다. \n");
    else
        printf("입력 값은 0이다. \n");
}
```

if~else문은 하나의 문장임을 상기!

```
if(num<0)
    printf("입력 값은 0보다 작다. \n");
else
    if(num>0)
        printf("입력 값은 0보다 크다. \n");
    else
        printf("입력 값은 0이다. \n");
```

else에 하나의 if~else문이 속한 상황.

속한 문장이 하나일 때에는 중괄호를 생략할 수 있다!

조건 연산자: 피 연산자가 세 개인 '삼항 연산자'

```
(num1>num2) ? (num1) : (num2);
```

```
(조건) ? data1 : data2
```

조건이 참이면 data1 반환, 거짓이면 data2 반환

```
int num3 = (num1>num2) ? (num1) : (num2);
```

```
int num3 = num1;  num1>num2가 참이면
```

```
int num3 = num2;  num1>num2가 거짓이면
```

```
int main(void)
{
    int num, abs;
    printf("정수 입력: ");
    scanf("%d", &num);

    abs = num>0 ? num : num*(-1);
    printf("절댓값: %d \n", abs);
    return 0;
}
```

실행결과

정수 입력: -79
절댓값: 79

윤성우의 열혈 C 프로그래밍



Chapter 08-2. 반복문의 생략과 탈출:
continue & break

윤성우 저 열혈강의 C 프로그래밍 개정판

break! 이제 그만 빠져나가자!

```
int main(void)
{
    int sum=0, num=0;

    while(1)
    {
        sum+=num;
        if(sum>5000)
            break; // break문 실행! 따라서 반복문 탈출
        num++;
    }

    printf("sum: %d \n", sum);
    printf("num: %d \n", num);
    return 0;
}
```

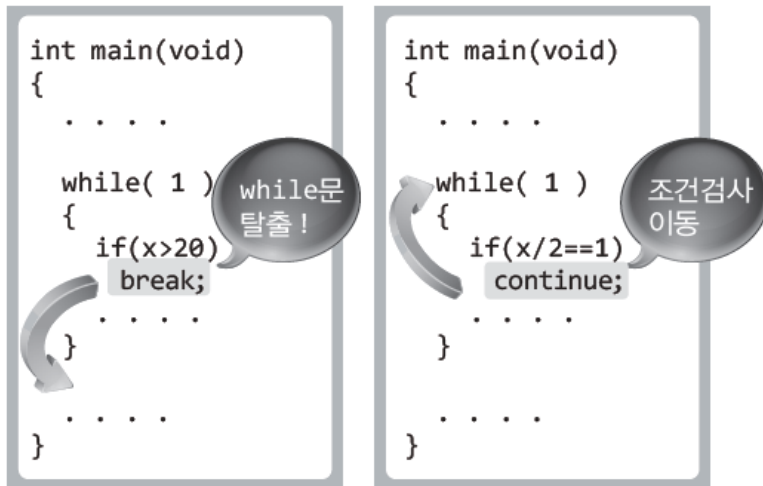
break문은 자신을 감싸는 반복문 하나를 빠져나간다.

if문과 함께 사용이 되어서 특정 조만이 만족될 때 반복문을 빠져나가는 용도로 주로 사용된다.

실행결과

```
sum: 5050
num: 100
```

continue! 나머지 생략하고 반복조건 확인하러



continue문은 반복문을 빠져나가지 않는다!
다만 반복조건을 확인하러 올라갈 뿐이다.
그리고 반복조건이 여전히 '참'이라면 반복영역을
처음부터 실행하게 된다.

```
int main(void)
{
    int num;
    printf("start! ");

    for(num=1; num<20; num++)
    {
        if(num%2==0 || num%3==0)
            continue;
        printf("%d ", num);
    }
    printf("end! \n");
    return 0;
}
```

start! 1 5 7 11 13 17 19 end!

실행결과

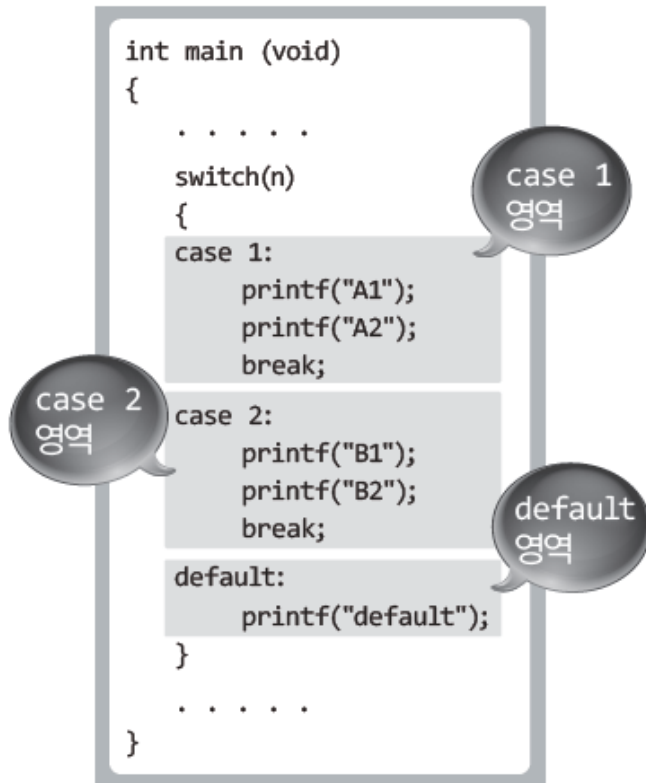
윤성우의 열혈 C 프로그래밍



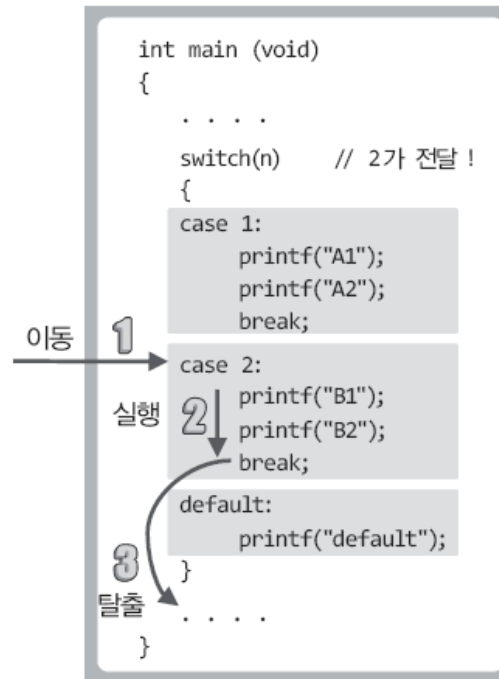
Chapter 08-3. switch문에 의한 선택적
실행과 goto문

윤성우 저 열혈강의 C 프로그래밍 개정판

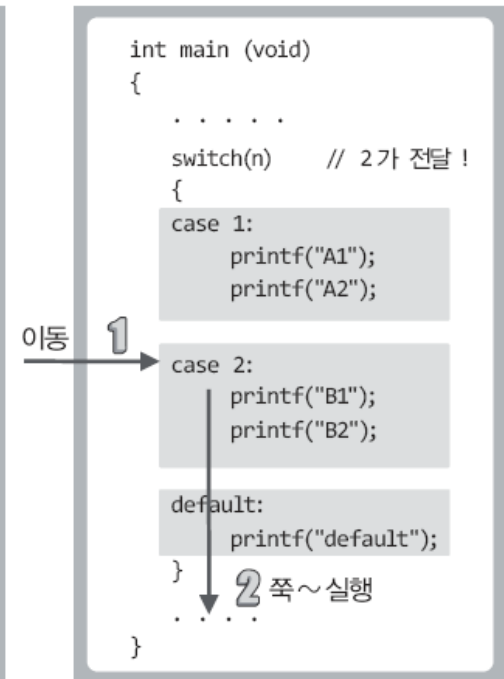
switch문의 구성과 기본기능



switch문의 기본구성



삽입되어 있는 break문이 갖는 의미



switch문 관련 예제

```
int main(void)
{
    int num;
    printf("1이상 5이하의 정수 입력: ");
    scanf("%d", &num);
    switch(num)
    {
        case 1:
            printf("1은 ONE \n");
            break;
        case 2:
            printf("2는 TWO \n");
            break;
        case 3:
            printf("3은 THREE \n");
            break;
        case 4:
            printf("4는 FOUR \n");
            break;
        case 5:
            printf("5는 FIVE \n");
            break;
        default:
            printf("I don't know! \n");
    }
    return 0;
}
```

실행결과1

1이상 5이하의 정수 입력: 3
3은 THREE

실행결과2

1이상 5이하의 정수 입력: 5
5는 FIVE

실행결과3

1이상 5이하의 정수 입력: 7
I don't know!

break문을 생략한 형태의 switch문 구성

```
int main(void)
{
    char sel;
    printf("M 오전, A 오후, E 저녁 \n");
    printf("입력: ");
    scanf("%c", &sel);

    switch(sel)
    {
        case 'M':
        case 'm':
            printf("Morning \n");
            break;
        case 'A':
        case 'a':
            printf("Afternoon \n");
            break;
        case 'E':
        case 'e':
            printf("Evening \n");
            break; // 사실 불필요한 break문!
    }
    return 0;
}
```

원편의 예제와 같은 경우 다음과 같이 두 case 레이블을 한 줄에 같이 표시하기도 한다.

case 'M': case 'm':

.....

case 'A': case 'a':

.....

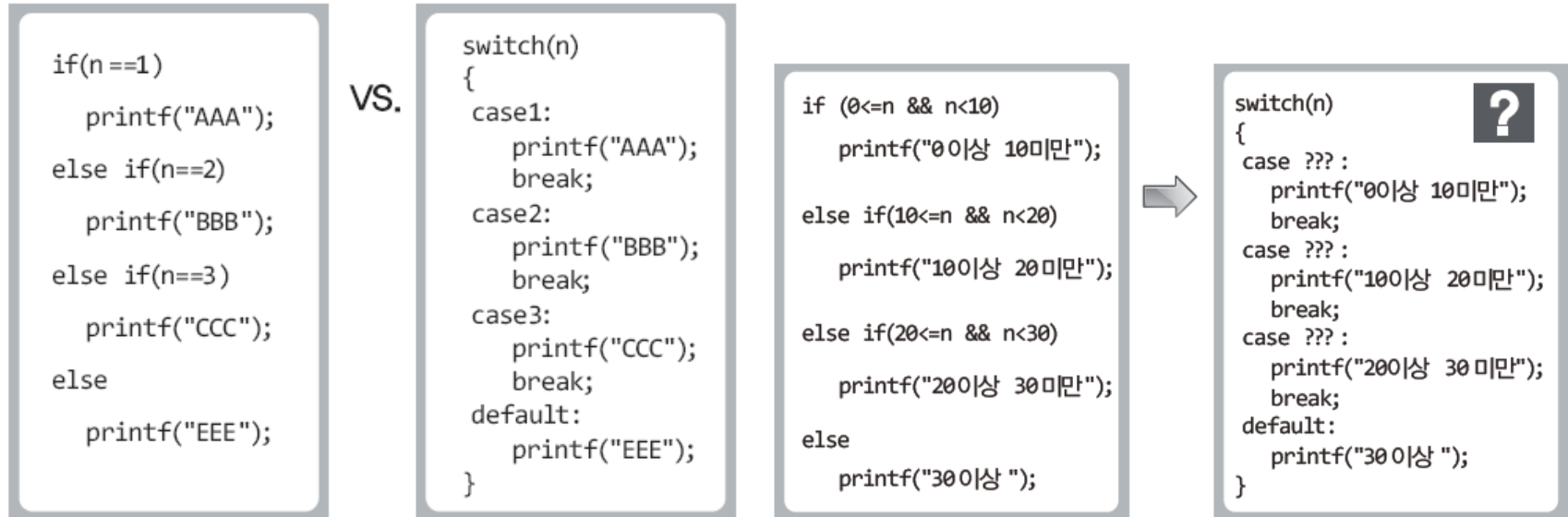
case 'E': case 'e':

.....

실행결과

```
M 오전, A 오후, E 저녁
입력: M
Morning
```

switch vs. if...else if...else



if...else if...else보다 switch문을 선호한다.
switch문이 더 간결해 보이기 때문이다.

모든 if...else if...else문을 switch문으로
대체할 수 있는 것은 아니다.

마지막으로 goto에 대해서 소개합니다.

```
int main(void)
```

```
{
```

```
    . . . .
```

```
rabbit: 위치를 표시하는 rabbit 레이블
```

```
    . . . .
```

```
    goto rabbit: 레이블 rabbit으로 무조건 이동!
```

```
    . . . .
```

```
}
```

goto는 단점이 많다. 따라서 이해는 하되 활용은
하지 말자!

실행결과

자연수 입력: 2
2를 입력하셨습니다!

```
int main(void)
```

```
{
```

```
    int num;
```

```
    printf("자연수 입력: ");
```

```
    scanf("%d", &num);
```

```
    if(num==1)
```

```
        goto ONE;
```

```
    else if(num==2)
```

```
        goto TWO;
```

```
    else
```

```
        goto OTHER;
```

```
ONE:
```

```
    printf("1을 입력하셨습니다! \n");
```

```
    goto END;
```

```
TWO:
```

```
    printf("2를 입력하셨습니다! \n");
```

```
    goto END;
```

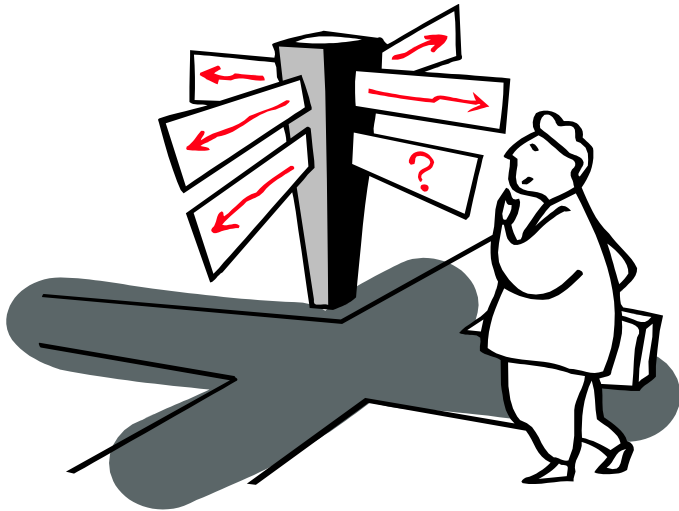
```
OTHER:
```

```
    printf("3 혹은 다른 값을 입력하셨습니다! \n");
```

```
END:
```

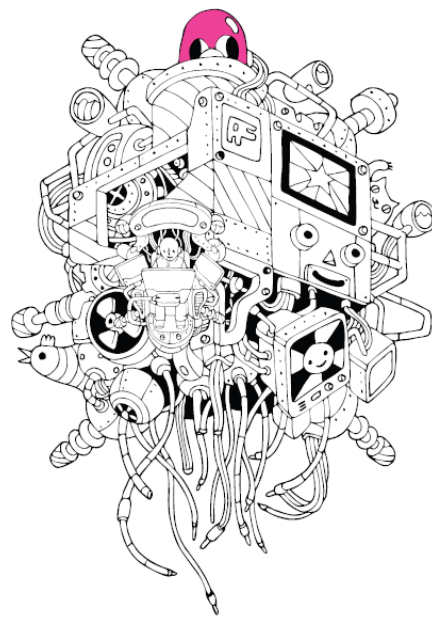
```
    return 0;
```

```
}
```



Chapter 08이 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 09. C언어의 핵심! 함수!

윤성우의 열혈 C 프로그래밍



Chapter 09-1. 함수를 정의하고 선언하기

윤성우 저 열혈강의 C 프로그래밍 개정판

함수를 만드는 이유

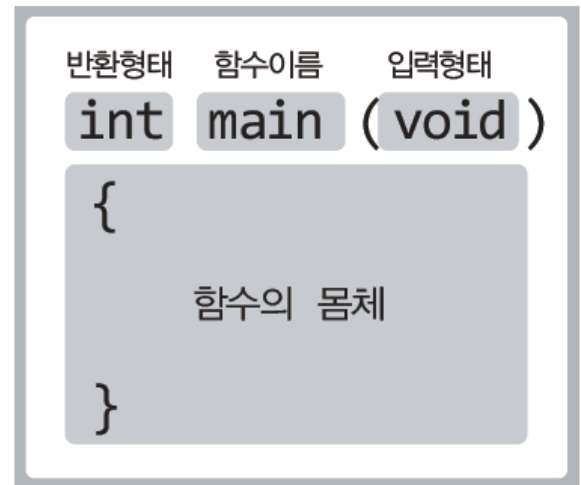
"Divide and Conquer!"

다수의 작은 단위 함수를 만들어서 프로그램을 작성하면 큰 문제를 작게 쪼개서 해결하는 효과를 얻을 수 있다.

그러나 함수를 만드는 이유 및 이점은 이보다 훨씬 다양하다.

main 함수를 포함하여 함수의 크기는 작을수록 좋다. 무조건 작다고 좋은 것은 아니지만, 불필요하게 큰 함수가 만들어지지 않도록 주의해야 한다.

하나의 함수는 하나의 일만 담당하도록 디자인 되어야 한다. 물론 **하나의 일**이라는 것은 매우 주관적인 기준이다. 그러나 이러한 주관적 기준 역시 프로그래밍에 대한 경험이 쌓이면 매우 명확한 기준이 된다.



함수의 입력과 출력: printf 함수도 반환을 합니다.

```
int main(void)
{
    int num1, num2;
    num1=printf("12345\n");
    num2=printf("I love my home\n");
    printf("%d %d \n", num1, num2);
    return 0;
}
```

printf 함수도 사실상 값을 반환한다. 다만 반환값이 필요 없어서 반환되는 값을 저장하지 않았을 뿐이다.
printf 함수는 출력된 문자열의 길이를 반환한다.

```
12345
I love my home
6 15
```

실행결과

함수가 값을 반환하면 반환된 값이 함수의 호출문을 대체한다고 생각하면 된다.

예를 들어서 아래의 printf 함수 호출문이 6을 반환한다면,

```
num1=printf("12345\n");
```

함수의 호출결과와는 다음과 같이 되어 대입연산이 진행된다.

```
num1=6;
```



함수의 구분

유형 1: 전달인자 있고, 반환 값 있다! 전달인자(○), 반환 값(○)

유형 2: 전달인자 있고, 반환 값 없다! 전달인자(○), 반환 값(×)

유형 3: 전달인자 없고, 반환 값 있다! 전달인자(×), 반환 값(○)

유형 4: 전달인자 없고, 반환 값 없다! 전달인자(×), 반환 값(×)

전달인자와 반환 값의 유무에 따른 함수의 구분!

전달인자 반환 값 모두 있는 경우

전달인자는 int형 정수 둘이며, 이 둘을 이용한 덧셈을 진행한다.
덧셈결과는 반환이 되며, 따라서 반환형도 int형으로 선언한다.
마지막으로 함수의 이름은 Add라 하자!

```

A.  B.  C.
int  Add (int num1, int num2)
{
    int result = num1 + num2;
    D. return result;
}
    
```



A. 반환형
B. 함수의 이름
C. 매개변수
D. 값의 반환

함수호출이 완료되면 호출한 위치로
이동해서 실행을 이어간다.

실행결과

덧셈결과1: 7
덧셈결과2: 13

```

int Add(int num1, int num2)
{
    return num1+num2; 덧셈이 선 진행되고
                        그 결과가 반환됨
}

int main(void)
{
    int result;
    result = Add(3, 4);
    printf("덧셈결과1: %d \n", result);
    result = Add(5, 8);
    printf("덧셈결과2: %d \n", result);
    return 0;
}
    
```

전달인자나 반환 값이 존재하지 않는 경우

```
void ShowAddResult(int num)  // 인자전달 (O), 반환 값 (X)
{
    printf("덧셈결과 출력: %d \n", num);
}
```

```
int ReadNum(void)  // 인자전달 (X), 반환 값 (O)
{
    int num;
    scanf("%d", &num);
    return num;
}
```

```
void HowToUseThisProg(void)  // 인자전달 (X), 반환 값 (X)
{
    printf("두 개의 정수를 입력하시면 덧셈결과가 출력됩니다. \n");
    printf("자! 그럼 두 개의 정수를 입력하세요. \n");
}
```



4가지 함수 유형을 조합한 예제

```
int Add(int num1, int num2)    // 인자전달 (0), 반환 값 (0)
{
    return num1+num2;
}

void ShowAddResult(int num)    // 인자전달 (0), 반환 값 (X)
{
    printf("덧셈결과 출력: %d \n", num);
}

int ReadNum(void)             // 인자전달 (X), 반환 값 (0)
{
    int num;
    scanf("%d", &num);
    return num;
}

void HowToUseThisProg(void)    // 인자전달 (X), 반환 값 (X)
{
    printf("두 개의 정수를 입력하시면 덧셈결과가 출력됩니다. \n");
    printf("자! 그럼 두 개의 정수를 입력하세요. \n");
}
```

```
int main(void)
{
    int result, num1, num2;
    HowToUseThisProg();
    num1=ReadNum();
    num2=ReadNum();
    result = Add(num1, num2);
    ShowAddResult(result);
    return 0;
}
```

실행결과

두 개의 정수를 입력하시면 덧셈결과가 출력됩니다.
자! 그럼 두 개의 정수를 입력하세요.
12 24
덧셈결과 출력: 36

값을 반환하지 않는 return

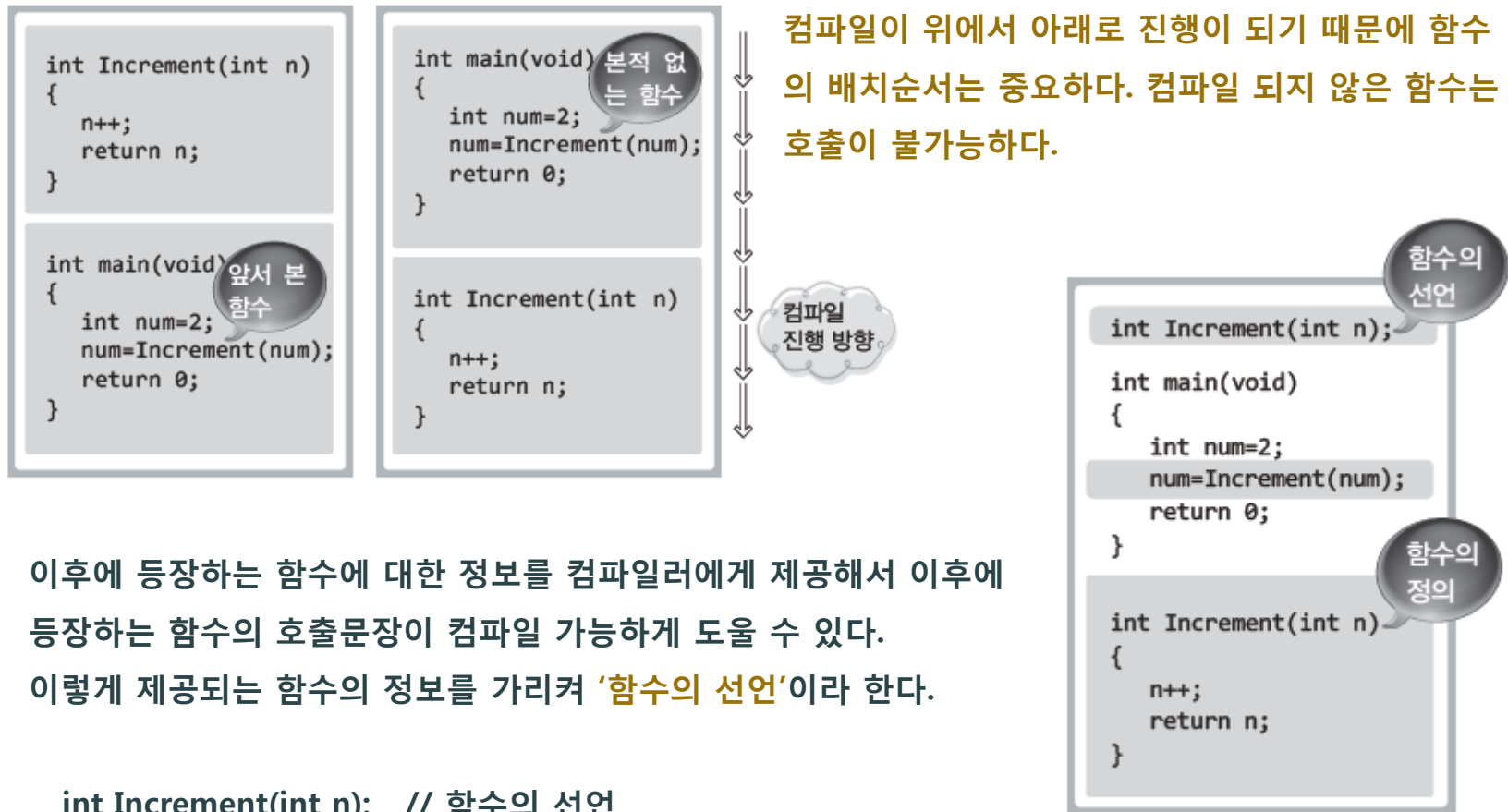
```
void NoReturnType(int num)
{
    if(num<0)
        return;    // 값을 반환하지 않는 return문!
    . . . .
}
```

return문에는 '값의 반환'과 '함수의 탈출'이라는 두 가지 기능이 담겨있다.

위의 예제에서 보이듯이 값을 반환하지 않는 형태로 return문을 구성하여 값을 반환하지 않되 함수를 빠져나가는 용도로 사용할 수 있다.



함수의 정의와 그에 따른 원형의 선언



이후에 등장하는 함수에 대한 정보를 컴파일러에게 제공해서 이후에 등장하는 함수의 호출문장이 컴파일 가능하게 도울 수 있다.
이렇게 제공되는 함수의 정보를 가리켜 '함수의 선언'이라 한다.

```
int Increment(int n); // 함수의 선언
```

```
int Increment(int); // 위와 동일한 함수선언, 매개변수 이름 생략 가능
```

다양한 종류의 함수 정의1

```
int main(void)
{
    printf("3과 4중에서 큰 수는 %d 이다. \n", NumberCompare(3, 4));
    printf("7과 2중에서 큰 수는 %d 이다. \n", NumberCompare(7, 2));
    return 0;
}

int NumberCompare(int num1, int num2)
{
    if(num1>num2)
        return num1;
    else
        return num2;
}
```

중간에도 얼마든지 *return*문이 올 수 있다.

실행결과

3과 4중에서 큰 수는 4 이다.
7과 2중에서 큰 수는 7 이다.

```
printf("3과 4중에서 큰 수는 %d 이다. \n", NumberCompare(3, 4));
printf("7과 2중에서 큰 수는 %d 이다. \n", NumberCompare(7, 2));
```

```
printf("3과 4중에서 큰 수는 %d 이다. \n", 4);
printf("7과 2중에서 큰 수는 %d 이다. \n", 7);
```

위의 두 문장한 *NumberCompare* 함수호출 이후 왼쪽과 같이 된다.

다양한 종류의 함수 정의2

```
int AbsoCompare(int num1, int num2); // 절댓값이 큰 정수 반환
int GetAbsoValue(int num); // 전달인자의 절댓값을 반환

int main(void)
{
    int num1, num2;
    printf("두 개의 정수 입력: ");
    scanf("%d %d", &num1, &num2);
    printf("%d와 %d중 절댓값이 큰 정수: %d \n",
        num1, num2, AbsoCompare(num1, num2));
    return 0;
}

int AbsoCompare(int num1, int num2)
{
    if(GetAbsoValue(num1) > GetAbsoValue(num2))
        return num1;
    else
        return num2;
}

int GetAbsoValue(int num)
{
    if(num < 0)
        return num * (-1);
    else
        return num;
}
```

```
if(GetAbsoValue(num1) > GetAbsoValue(num2))
    . . . .

if( 5 > 9 )    GetAbsoValue 함수호출 이후
    . . . .
```

이 예제에서 보이듯이 함수의 호출문장은 어디
에든 놓일 수 있다.

실행결과

```
두 개의 정수 입력: 5 -9
5와 -9중 절댓값이 큰 정수: -9
```

윤성우의 열혈 C 프로그래밍



Chapter 09-2. 변수의 존재기간과 접근범위 1: 지역변수

윤성우 저 열혈강의 C 프로그래밍 개정판

함수 내에만 존재 및 접근 가능한 지역변수

```
int SimpleFuncOne(void)
{
    int num=10;    // 이후부터 SimpleFuncOne의 num 유효
    num++;
    printf("SimpleFuncOne num: %d \n", num);
    return 0;    // SimpleFuncOne의 num이 유효한 마지막 문장
}

int SimpleFuncTwo(void)
{
    int num1=20;    // 이후부터 num1 유효
    int num2=30;    // 이후부터 num2 유효
    num1++, num2--;
    printf("num1 & num2: %d %d \n", num1, num2);
    return 0;    // num1, num2 유효한 마지막 문장
}

int main(void)
{
    int num=17;    // 이후부터 main의 num 유효
    SimpleFuncOne();
    SimpleFuncTwo();
    printf("main num: %d \n", num);
    return 0;    // main의 num이 유효한 마지막 문장
}
```

함수 내에 선언되는 변수를 가리켜 지역변수라 한다.

지역변수는 선언된 이후로부터 함수 내에서만 접근이 가능하다.

한 지역(함수) 내에 동일한 이름의 변수 선언 불가능하다.

다른 지역에 동일한 이름의 변수 선언 가능하다.

해당 지역을 빠져나가면 지역변수는 소멸된다. 그리고 호출될 때마다 새롭게 할당된다.

실행결과

```
SimpleFuncOne num: 11
num1 & num2: 21 29
main num: 17
```

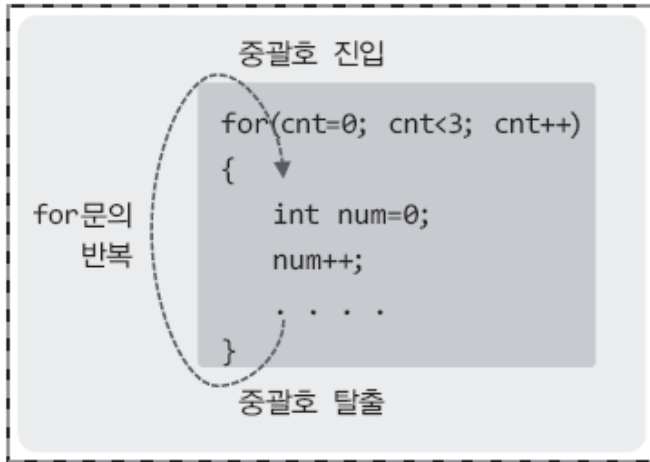
메모리 공간의 할당과 소멸 관찰하기

```
int SimpleFuncOne(void)
{
    int num=10;    // 이후부터 SimpleFuncOne의 num 유효
    num++;
    printf("SimpleFuncOne num: %d \n", num);
    return 0;    // SimpleFuncOne의 num이 유효한 마지막 문장
}

int SimpleFuncTwo(void)
{
    int num1=20;    // 이후부터 num1 유효
    int num2=30;    // 이후부터 num2 유효
    num1++, num2--;
    printf("num1 & num2: %d %d \n", num1, num2);
    return 0;    // num1, num2 유효한 마지막 문장
}

int main(void)
{
    int num=17;    // 이후부터 main의 num 유효
    SimpleFuncOne();
    SimpleFuncTwo();
    printf("main num: %d \n", num);
    return 0;    // main의 num이 유효한 마지막 문장
}
```

다양한 형태의 지역변수



for문의 중괄호 내에 선언된 변수도 지역변수이다. 그리고 이 지역변수는 for문의 중괄호를 빠져나가면 소멸된다.
따라서 for문의 반복횟수만큼 지역변수가 할당되고 소멸된다.

지역변수는 외부에 선언된 동일한 이름의 변수를 가린다.

실행결과

if문 내 지역변수 num: 17
main 함수 내 지역변수 num: 1

주석처리 후 실행결과

if문 내 지역변수 num: 11
main 함수 내 지역변수 num: 11

```
int main(void)
{
    int num=1;
    if(num==1)
    {
        int num=7; // 이 행을 주석처리 하고 실행결과 확인하자!
        num+=10;
        printf("if문 내 지역변수 num: %d \n", num);
    }
    printf("main 함수 내 지역변수 num: %d \n", num);
    return 0;
}
```

if문 내에 선언된 변수 num이 main 함수의 변수 num을 가린다.

지역변수의 일종인 매개변수

지역변수



매개변수는 일종의 지역변수이다.

매개변수도 선언된 함수 내에서만 접근이 가능하다.

선언된 함수가 반환을 하면, 지역변수와 마찬가지로 매개변수도 소멸된다.



윤성우의 열혈 C 프로그래밍



Chapter 09-3. 전역변수, static 변수,
register 변수

윤성우 저 열혈강의 C 프로그래밍 개정판

전역변수의 이해와 선언방법

```
void Add(int val);
int num;    // 전역변수는 기본 0으로 초기화됨

int main(void)
{
    printf("num: %d \n", num);
    Add(3);
    printf("num: %d \n", num);
    num++;    // 전역변수 num의 값 1 증가
    printf("num: %d \n", num);
    return 0;
}

void Add(int val)
{
    num += val;    // 전역변수 num의 값 val만큼 증가
}
```

num: 0
num: 3
num: 4

실행결과

전역변수는 함수 외부에 선언된다.

프로그램의 시작과 동시에 메모리 공간에 할당되어 종료 시까지 존재한다.

별도의 값으로 초기화하지 않으면 0으로 초기화된다.

프로그램 전체 영역 어디서든 접근이 가능하다.

```
int Add(int val);
int num=1;

int main(void)
{
    int num=5;
    printf("num: %d \n", Add(3));
    printf("num: %d \n", num+9);
    return 0;
}

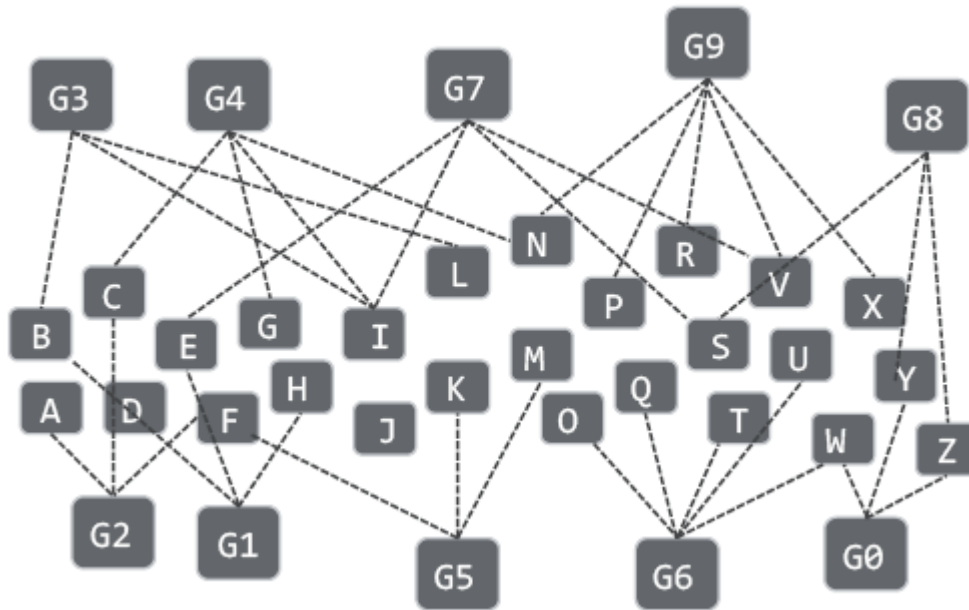
int Add(int val)
{
    int num=9;
    num += val;
    return num;
}
```

지역변수의 이름이
전역변수의 이름을 가린다.

num: 12
num: 14

실행결과

전역변수! 많이 써도 되는가?



G0~G9의 전역변수와 함수와의 접근관계의 예시

전역변수! 많이 쓰면 좋지 않다. 전역변수의 변경은 전체 프로그램의 변경으로 이어질 수 있으며 전역변수에 의존적인 코드는 프로그램 전체 영역에서 찾아야 한다. 어디서든 접근이 가능한 변수이므로...

지역변수에 static 선언을 추가한 static 변수

```
void SimpleFunc(void)
{
    static int num1=0;    // 초기화하지 않으면 0 초기화
    int num2=0;          // 초기화하지 않으면 쓰레기 값 초기화
    num1++, num2++;
    printf("static: %d, local: %d \n", num1, num2);
}

int main(void)
{
    int i;
    for(i=0; i<3; i++)
        SimpleFunc();
    return 0;
}
```

선언된 함수 내에서만 접근이 가능하다.
(지역변수 특성)

딱 1회 초기화되고 프로그램 종료 시까지
메모리 공간에 존재한다. (전역변수 특성)

실행결과

```
static: 1, local: 1
static: 2, local: 1
static: 3, local: 1
```

“난 사실 전역변수랑 성격이 같아. 초기화하지 않으면 전역변수처럼 0으로 초기화되고, 프로그램 시작과 동시에 할당 및 초기화되어서 프로그램이 종료될 때까지 메모리 공간에 남아있지! 그럼 왜 이 위치에 선언 되었냐고? 그건 접근의 범위를 SimpleFunc로 제한하기 위해서야!”

static 지역변수의 발언!

프로그램이 실행되면 static 지역변수는 해당 함수에 존재하지 않는다.

static 지역변수는 좀 써도 되나요?

√ 전역변수가 필요한 이유 중 하나는 다음과 같다.

선언된 변수가 함수를 빠져나가도 계속해서 메모리 공간에 존재할 필요가 있다.

√ 함수를 빠져나가도 계속해서 메모리 공간에 존재해야 하는 변수를 선언하는 방법은 다음 두 가지이다.

전역변수, *static* 지역 변수

√ *static* 지역변수는 접근의 범위가 전역변수보다 훨씬 좁기 때문에 훨씬 안정적이다.

static 지역변수를 사용하여 전역변수의 선언을 최소화하자.



보다 빠르게! register 변수

```
int SoSimple(void)
{
    register int num=3;
    . . . .
}
```

register는 힌트를 제공하는 키워드이다. 컴파일러는 이를 무시하기도 한다.

그리고 레지스터는 CPU 내부에 존재하는, 때문에 접근이 가장 빠른 메모리 장치이다.

“이 변수는 내가 빈번히 사용하거든, 그래서 접근이 가장 빠른 레지스터에 저장하는 것이 성능향상에 도움이 될 거야” *register* 변수 선언의 의미



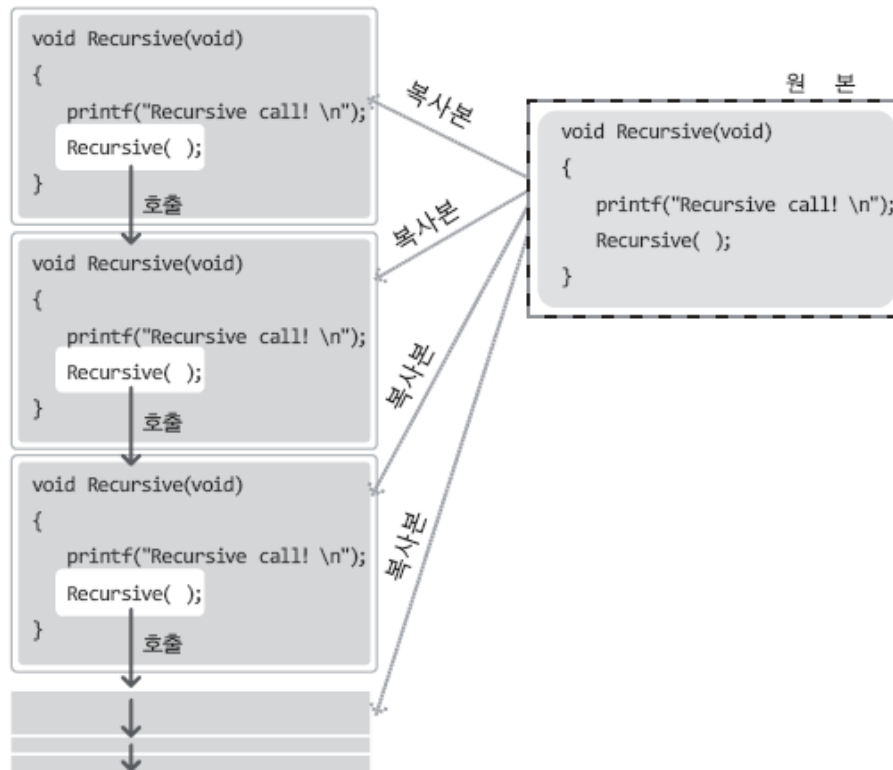
윤성우의 열혈 C 프로그래밍



Chapter 09-4. 재귀함수에 대한 이해

윤성우 저 열혈강의 C 프로그래밍 개정판

재귀함수의 기본적인 이해



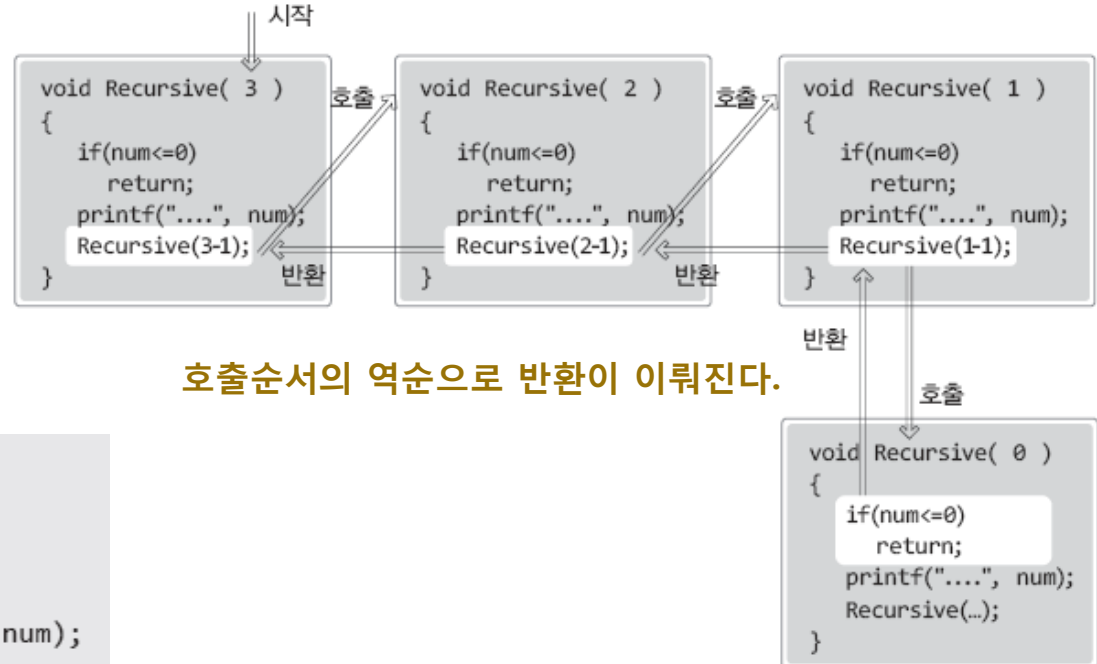
재귀함수 호출의 이해!

```

void Recursive(void)
{
    printf("Recursive call! \n");
    Recursive(); // 나! 자신을 재 호출한다.
}
    
```

자기자신을 재호출하는 형태로 정의된 함수를
가리켜 재귀함수라 한다.

탈출조건이 존재하는 재귀함수의 예



```

void Recursive(int num)
{
    if(num<=0)    // 재귀의 탈출조건
        return; //재귀의 탈출!
    printf("Recursive call! %d \n", num);
    Recursive(num-1);
}

int main(void)
{
    Recursive(3);
    return 0;
}
    
```

실행결과

```

Recursive call! 3
Recursive call! 2
Recursive call! 1
    
```

재귀함수의 디자인 사례

$n! = n \times (n-1) \times (n-2) \times (n-3) \times \dots \times 2 \times 1$

$(n-1)!$



$n! = n \times (n-1)!$



팩토리얼에 대한 수학적 표현

$$f(n) = \begin{cases} n \times f(n-1) & \dots n \geq 1 \\ 1 & \dots n = 0 \end{cases}$$



$n \times f(n-1) \dots n \geq 1$ 에 대한 코드 구현

```
if(n>=1)
    return n * Factorial(n-1);
```

$f(n)=1$ 에 대한 코드 구현

```
if(n==0)
    return 1;
```



```
if(n==0)
    return 1;
else
    return n * Factorial(n-1);
```

팩토리얼 함수의 예

```
int Factorial(int n)
{
    if(n==0)
        return 1;
    else
        return n * Factorial(n-1);
}

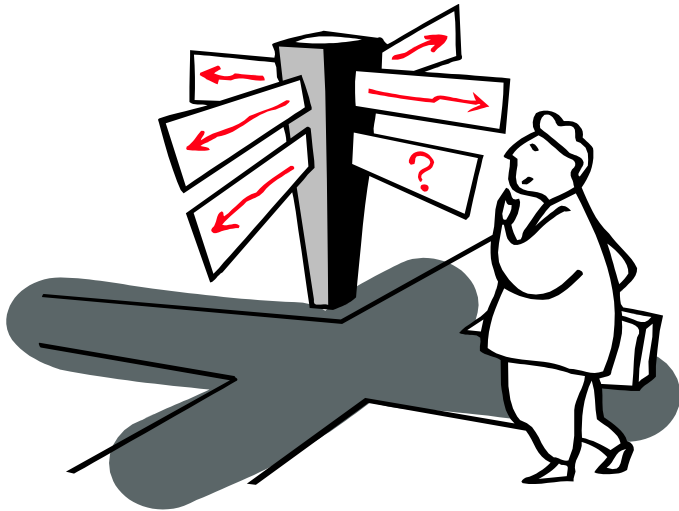
int main(void)
{
    printf("1! = %d \n", Factorial(1));
    printf("2! = %d \n", Factorial(2));
    printf("3! = %d \n", Factorial(3));
    printf("4! = %d \n", Factorial(4));
    printf("9! = %d \n", Factorial(9));
    return 0;
}
```

C언어가 재귀적 함수호출을 지원한다는 것은 그만큼 표현할 수 있는 범위가 넓다는 것을 의미한다!

C언어의 재귀함수를 이용하면 재귀적으로 작성된 식을 그대로 코드로 옮길 수 있다.

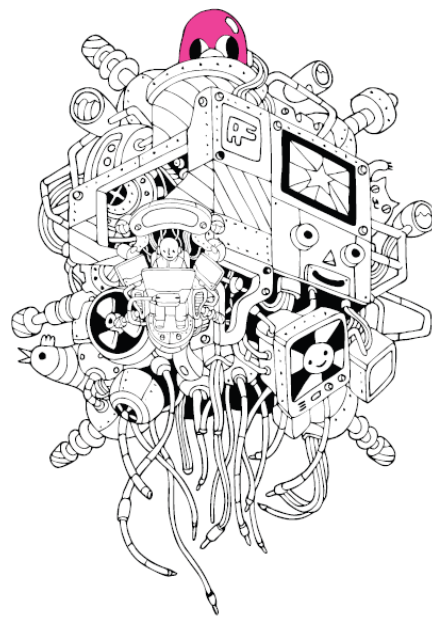
실행결과

```
1! = 1
2! = 2
3! = 6
4! = 24
9! = 362880
```



Chapter 09가 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 11. 1차원 배열

윤성우의 열혈 C 프로그래밍



Chapter 11-1. 배열의 이해와 배열의
선언 및 초기화 방법

윤성우 저 열혈강의 C 프로그래밍 개정판

배열이란 무엇인가?

```
int main(void)
{
    int floor101, floor102, floor103, floor104;    // 1층 101호부터 104호까지
    int floor201, floor202, floor203, floor204;    // 2층 201호부터 204호까지
    int floor301, floor302, floor303, floor304;    // 3층 301호부터 304호까지
    . . . . .
}
```

다수의 정보를 저장하기
위해서는 다수의 배열을
선언해야 한다.

위와 같이 다수의 변수를 선언해야 하는 경우 매우 번거로울 수 있다. 그래서 **다수의 변수선언을 용이하게 하기 위해서 배열이라는 것이 제공된다.** 배열을 이용하면 하나의 선언을 통해서 둘 이상의 변수를 선언할 수 있다.

배열은 단순히 다수의 변수선언을 대신하지 않는다. **다수의 변수로는 할 수 없는 일을 배열을 선언하면 할 수 있다.**

배열은 1차원의 형태로도 2차원의 형태로도 선언할 수 있다. 이번 Chapter에서는 1차원 형태의 배열에 대해서 학습한다.



1차원 배열 선언에 필요한 것 세 가지

```
int oneDimArr [4];
```

1차원 배열 선언의 예

int	배열을 이루는 요소(변수)의 자료형
oneDimArr	배열의 이름
[4]	배열의 길이



생성되는 배열의 형태

```
int arr1[7];           // 길이가 7인 int형 1차원 배열 arr1
float arr2[10];        // 길이가 10인 float형 1차원 배열 arr2
double arr3[12];       // 길이가 12인 double형 1차원 배열 arr3
```

다양한 배열 선언의 예

선언된 1차원 배열의 접근

```
arr[0]=10;    // 배열 arr의 첫 번째 요소에 10을 저장해라!
arr[1]=12;    // 배열 arr의 두 번째 요소에 12를 저장해라!
arr[2]=25;    // 배열 arr의 세 번째 요소에 25를 저장해라!
```

1차원 배열 접근의 예



일반화

arr[idx]=20; → "배열 arr의 idx+1번째 요소에 20을 저장해라!"

```
int main(void)
{
    int arr[5];
    int sum=0, i;

    arr[0]=10, arr[1]=20, arr[2]=30, arr[3]=40, arr[4]=50;

    for(i=0; i<5; i++)
        sum += arr[i];

    printf("배열요소에 저장된 값의 합: %d \n", sum);
    return 0;
}
```

원편의 예제를 통해서 느낄 수
있는 배열의 또 다른 매력은?

실행결과

배열요소에 저장된 값의 합: 150

배열! 선언과 동시에 초기화하기

```
int arr1[5]={1, 2, 3, 4, 5};
```

초기화 리스트로 초기화



초기화 결과



순서대로 초기화

```
int arr3[5]={1, 2};
```

초기화 값 부족한 경우



부족한 부분 0으로 채워짐



```
int arr2[ ]={1, 2, 3, 4, 5, 6, 7};
```

초기화 리스트는 존재하고 배열의
길이정보 생략된 경우



컴파일러가 배열의 길이정보 채움

```
int arr2[7]={1, 2, 3, 4, 5, 6, 7};
```

1차원 배열의 선언, 초기화 및 접근 관련 예제

```
int main(void)
{
    int arr1[5]={1, 2, 3, 4, 5};
    int arr2[ ]={1, 2, 3, 4, 5, 6, 7};
    int arr3[5]={1, 2};
    int ar1Len, ar2Len, ar3Len, i;

    printf("배열 arr1의 크기: %d \n", sizeof(arr1));
    printf("배열 arr2의 크기: %d \n", sizeof(arr2));
    printf("배열 arr3의 크기: %d \n", sizeof(arr3));

    ar1Len = sizeof(arr1) / sizeof(int); // 배열 arr1의 길이 계산
    ar2Len = sizeof(arr2) / sizeof(int); // 배열 arr2의 길이 계산
    ar3Len = sizeof(arr3) / sizeof(int); // 배열 arr3의 길이 계산

    for(i=0; i<ar1Len; i++)
        printf("%d ", arr1[i]);
    printf("\n");

    for(i=0; i<ar2Len; i++)
        printf("%d ", arr2[i]);
    printf("\n");

    for(i=0; i<ar3Len; i++)
        printf("%d ", arr3[i]);
    printf("\n");

    return 0;
}
```

*sizeof 연산의 결과로
배열의 바이트 크기 정보 반환*

*배열의 길이를 계산하는 방식
에 주목!*

*배열이기에 for문을 통한 순차
적 접근이 가능하다.*

*다수의 변수라면 반복문을 통한
순차적 접근 불가능!*

실행결과

```
배열 arr1의 크기: 20
배열 arr2의 크기: 28
배열 arr3의 크기: 20
1 2 3 4 5
1 2 3 4 5 6 7
1 2 0 0 0
```

윤성우의 열혈 C 프로그래밍



Chapter 11-2. 배열을 이용한 문자열
변수의 표현

윤성우 저 열혈강의 C 프로그래밍 개정판

char형 배열의 문자열 저장과 널 문자

```
char str[14]="Good morning!";
```

배열에 문자열 저장



저장결과



문자열의 끝에 널 문자라 불리는 \0가 삽입되었음에
주목! 널 문자는 문자열의 끝을 의미한다.

```
int main(void)
{
    char str[]="Good morning!";
    printf("배열 str의 크기: %d \n", sizeof(str));
    printf("널 문자 문자형 출력: %c \n", str[13]);
    printf("널 문자 정수형 출력: %d \n", str[13]);

    str[12]='?'; // 배열 str에 저장된 문자열 데이터는 변경 가능!
    printf("문자열 출력: %s \n", str);
    return 0;
}
```

실행결과

배열 str의 크기: 14
 널 문자 문자형 출력:
 널 문자 정수형 출력: 0
 문자열 출력: Good morning?

널 문자와 공백 문자의 비교

```
int main(void)
{
    char nu = '\0';    // 널 문자 저장
    char sp = ' ';     // 공백 문자 저장
    printf("%d %d", nu, sp);    // 0과 32 출력
    return 0;
}
```

널 문자를 %c를 이용해서 출력 시 아무것도 출력되지 않는다. 그렇다고 해서 널 문자가 공백 문자는 아니다.

널 문자의 아스키 코드 값은 0이고, 공백 문자의 아스키 코드 값은 32이다.

널 문자는 모니터 출력에서 의미를 갖지 않는다. 그래서 아무것도 출력이 되지 않을 뿐이다.



scanf 함수를 이용한 문자열의 입력

```
int main(void)
{
    char str[50];
    int idx=0;

    printf("문자열 입력: ");
    scanf("%s", str); // 문자열을 입력 받아서 배열 str에 저장!
    printf("입력 받은 문자열: %s \n", str);

    printf("문자 단위 출력: ");
    while(str[idx] != '\0')
    {
        printf("%c", str[idx]);
        idx++;
    }
    printf("\n");
    return 0;
}
```

scanf 함수의 호출을 통해서 입력 받은
문자열의 끝에도 널 문자가 존재함을 확
인하기 위한 문장

scanf 함수를 이용해서 문자열 입력 시
서식문자 %s를 사용한다.

scanf("%s", str);

위와 같이 배열이름 *str*의 앞에는
& 연산자를 붙이지 않는다.

실행결과

문자열 입력: Simple
입력 받은 문자열: Simple
문자 단위 출력: Simple

```
char arr1[ ] = {'H', 'i', '~'};
char arr2[ ] = {'H', 'i', '~', '\0'};
```

*arr1*은 문자열이 아닌 문자 배열, 반면 *arr2*는 문자열!
널 문자의 존재여부는 문자열의 판단여부가 된다.

문자열의 끝에 널 문자가 필요한 이유

문자열의 시작은 판단할 수 있어도 문자열의 끝은 판단이 불가능하다! 때문에 문자열의 끝을 판단할 수 있도록 널 문자가 삽입이 된다.

```
int main(void)
{
    char str[50]="I like C programming";
    printf("string: %s \n", str);

    str[8]='\0';    // 9번째 요소에 널 문자 저장
    printf("string: %s \n", str);

    str[6]='\0';    // 7번째 요소에 널 문자 저장
    printf("string: %s \n", str);

    str[1]='\0';    // 2번째 요소에 널 문자 저장
    printf("string: %s \n", str);
    return 0;
}
```

배열의 시작위치에 문자열이 저장되기 시작한다. 따라서 시작위치는 확인이 가능하다. 하지만 배열의 끝이 문자열의 끝은 아니므로 널 문자가 삽입되지 않으면 문자열의 끝은 확인이 불가능하다.

실행결과

```
string: I like C programming
string: I like C
string: I like
string: I
```

위 예제에서 보이듯이 printf 함수도 배열 str의 시작위치를 기준으로해서 널 문자를 만날 때까지 출력을 진행한다. 따라서 널 문자가 없으면 printf 함수도 문자열의 끝을 알지 못한다.

scanf 함수의 문자열 입력 특성

```
int main(void)
{
    char str[50];
    int idx=0;

    printf("문자열 입력: ");
    scanf("%s", str); // 문자열을 입력 받아서 배열 str에 저장!
    printf("입력 받은 문자열: %s \n", str);

    printf("문자 단위 출력: ");
    while(str[idx] != '\0')
    {
        printf("%c", str[idx]);
        idx++;
    }
    printf("\n");
    return 0;
}
```

앞서 보인 원편의 예제를 실행할 때 다음과 같이 문자열을 입력하면

He is my friend

다음의 실행결과를 보인다.

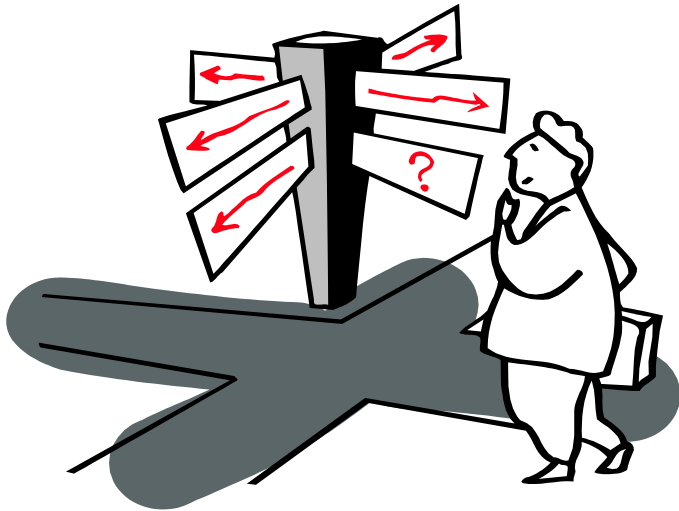
입력 받은 문자열: He

문자 단위 출력: He

scanf 함수는 공백을 기준으로 데이터의 수를 구분한다. 따라서 공백을 포함하는 문자열을 한번의 scanf 함수호출을 통해서 읽어 들이지는 못한다.

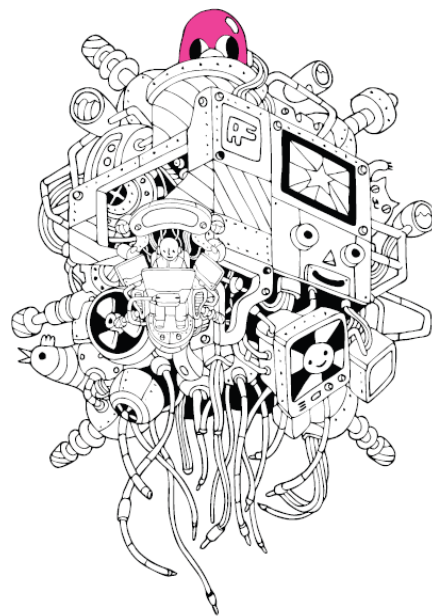
공백을 포함하는 문자열의 입력에 사용되는 함수는 이후에 별도로 설명합니다.





Chapter 11이 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 12. 포인터의 이해

윤성우의 열혈 C 프로그래밍

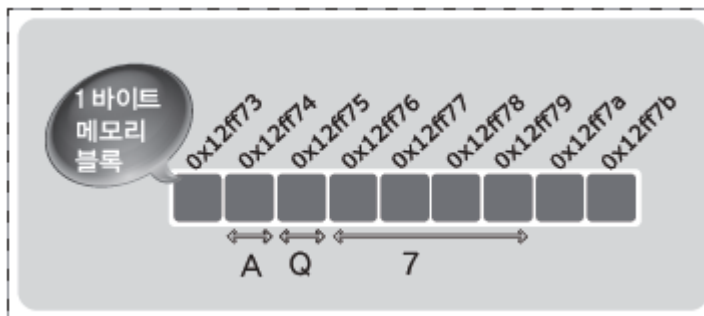


Chapter 12-1. 포인터란 무엇인가?

윤성우 저 열혈강의 C 프로그래밍 개정판

주소 값의 저장을 목적으로 선언되는 포인터 변수

```
int main(void)
{
    char ch1='A', ch2='Q';
    int num=7;
    . . . .
}
```



변수 num이 저장되기 시작한 주소 0x12ff76이 변수 num의 주소 값이다.

이러한 정수 형태의 주소 값을 저장하는 목적으로 선언되는 것이 포인터 변수이다.

포인터 변수와 & 연산자 맛보기

“정수 7이 저장된 int형 변수 num을 선언하고 이 변수의 주소 값을 저장할 위한 포인터 변수 pnum을 선언하자. 그리고 나서 pnum에 변수 num의 주소 값을 저장하자.”



코드로 옮긴 결과

```
int main(void)
{
    int num=7;
    int * pnum;
    pnum = &num;
    . . . .
}
```

포인터 변수 pnum의 선언
num의 주소 값을 pnum에 저장

포인터 변수의 크기는 시스템의 주소 값 크기에 따라서 다르다.

16비트 시스템 → 주소 값 크기 16비트 → 포인터 변수의 크기 16비트!

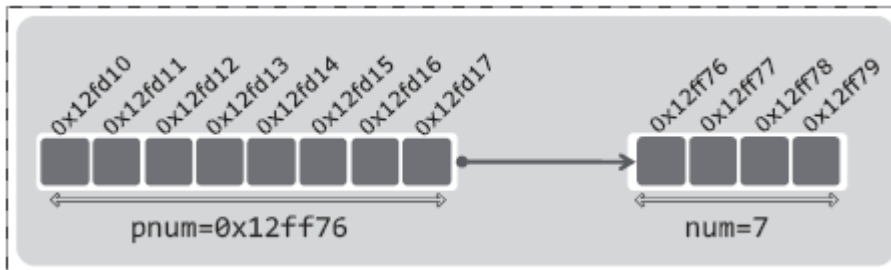
32비트 시스템 → 주소 값 크기 32비트 → 포인터 변수의 크기 32비트!

int * pnum 의 선언에서...

pnum	포인터 변수의 이름
int *	int형 변수의 주소 값을 저장하는 포인터 변수의 선언



메모리의 저장상태



이 상태를 다음과 같이 표현한다.

포인터 변수 pnum이 변수 num을 가리킨다.

포인터 변수 선언하기

가리키고자 하는 변수의 자료형에 따라서 포인터 변수의 선언방법에는 차이가 있다.
포인터 변수에 저장되는 값은 모두 정수로 값의 형태는 모두 동일하지만, 그래도 선언하는 방법에 차이가 있다(차이가 있는 이유는 메모리 접근과 관련이 있다).

```
int * pnum1;
```

int * 는 int형 변수를 가리키는 pnum1의 선언을 의미함

```
double * pnum2;
```

double * 는 double형 변수를 가리키는 pnum2의 선언을 의미함

```
unsigned int * pnum3;
```

unsigned int * 는 unsigned int형 변수를 가리키는 pnum3의 선언을 의미함



일반화

```
type * ptr;
```

type형 변수의 주소 값을 저장하는 포인터 변수 ptr의 선언

포인터의 형(Type)

```
int *           int형 포인터
int * pnum1;    int형 포인터 변수 pnum1

double *        double형 포인터
double * pnum2; double형 포인터 변수 pnum2
```



일반화

```
type *          type형 포인터
type * ptr;     type형 포인터 변수 ptr
```

포인터 변수 선언에서 * 의 위치에 따른 차이는 없다. 즉, 다음 세 문장은 모두 동일한 포인터 변수의 선언문이다.

```
int * ptr;      // int형 포인터 변수 ptr의 선언
int* ptr;      // int형 포인터 변수 ptr의 선언
int * ptr;      // int형 포인터 변수 ptr의 선언
```



윤성우의 열혈 C 프로그래밍



Chapter 12-2. 포인터와 관련 있는
& 연산자와 * 연산자

윤성우 저 열혈강의 C 프로그래밍 개정판

변수의 주소 값을 반환하는 & 연산자

```
int main(void)
{
    int num = 5;
    int * pnum = &num;
    . . . .
}
```

& 연산자는 변수의 주소 값을 반환하므로 상수가 아닌 변수가 피연산자이어야 한다. & 연산자의 반환 값은 포인터 변수에 저장을 한다.

```
int main(void)
{
    int num1 = 5;
    double * pnum1 = &num1; // 일치하지 않음!

    double num2 = 5;
    int * pnum2 = &num2; // 일치하지 않음!
    . . . .
}
```

num1은 int형 변수이므로 pnum1은 int형 포인터 변수이어야 함

num2는 double형 변수이므로 pnum2는 double형 포인터 변수이어야 함.

int형 변수 대상의 & 연산의 반환 값은 int형 포인터 변수에, double형 변수 대상의 & 연산의 반환 값은 double형 포인터 변수에 저장한다.

포인터가 가리키는 메모리를 참조하는 * 연산자

```
int main(void)
{
    int num=10;
    int * pnum=&num;
    *pnum=20;
    printf("%d", *pnum);
    . . .
}
```

pnum이 num을 가리킨다.

pnum이 가리키는 공간(변수)에 20을 저장

pnum이 가리키는 공간(변수)에 저장된 값 출력

*pnum은 num을 의미한다.

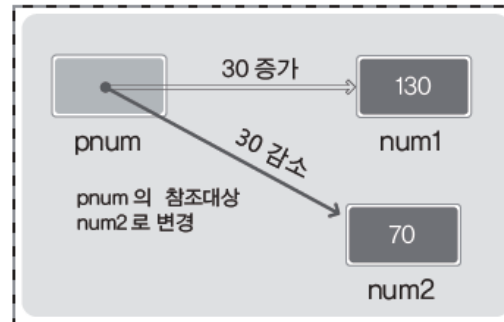
따라서 num을 놓을 자리에 *pnum을 놓을 수 있다.

```
int main(void)
{
    int num1=100, num2=100;
    int * pnum;

    pnum=&num1;    // 포인터 pnum이 num1을 가리킴
    (*pnum)+=30;   // num1+=30; 과 동일

    pnum=&num2;    // 포인터 pnum이 num2를 가리킴
    (*pnum)-=30;   // num2-=30; 과 동일

    printf("num1:%d, num2:%d \n", num1, num2);
    return 0;
}
```



실행결과

num1:130, num2:70

다양한 포인터 형이 존재하는 이유

포인터 형은 메모리 공간을 참조하는 방법의 힌트가 된다. 다양한 포인터 형을 정의한 이유는 * 연산을 통한 메모리의 접근기준을 마련하기 위함이다.

int형 포인터 변수로 * 연산을 통해 메모리(변수) 접근 시

4바이트 메모리 공간에 부호 있는 정수의 형태로 데이터를 읽고 쓴다.

double형 포인터 변수로 * 연산을 통해 메모리(변수) 접근 시

8바이트 메모리 공간에 부호 있는 실수의 형태로 데이터를 읽고 쓴다.

```
int main(void)
{
    double num=3.14;
    int * pnum=&num;
    printf("%d", *pnum);
    . . . .
}
```

형 불일치! 컴파일은 된다.
pnum이 가리키는 것은 double형 변수인데, pnum이 int형 포인터 변수이므로 int형 데이터처럼 해석!

주소 값이 정수임에도 불구하고 int형 변수에 저장하지 않는 이유는 int형 변수에 저장하면 메모리 공간의 접근을 위한 * 연산이 불가능하기 때문이다.

잘못된 포인터의 사용과 널 포인터

```
int main(void)
{
    int * ptr;
    *ptr=200;
    . . . .
}
```

위험한 코드

ptr이 쓰레기 값으로 초기화 된다. 따라서 200이 저장되는 위치는 어디인지 알 수 없다! 매우 위험한 행동!

```
int main(void)
{
    int * ptr=125;
    *ptr=10;
    . . . .
}
```

위험한 코드

포인터 변수에 125를 저장했는데 이곳이 어디인가? 역시 매우 위험한 행동!

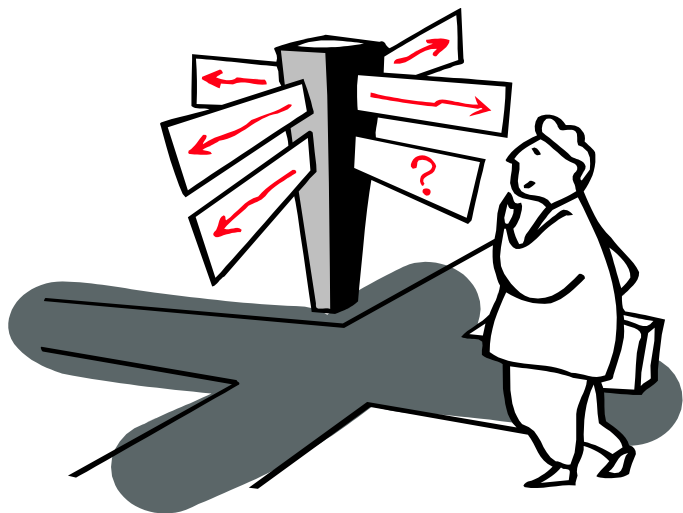
```
int main(void)
{
    int * ptr1=0;
    int * ptr2=NULL;
    . . . .
}
```

안전한 코드

잘못된 포인터 연산을 막기 위해서 특정한 값으로 초기화하지 않는 경우에는 **널 포인터**로 초기화하는 것이 안전하다.

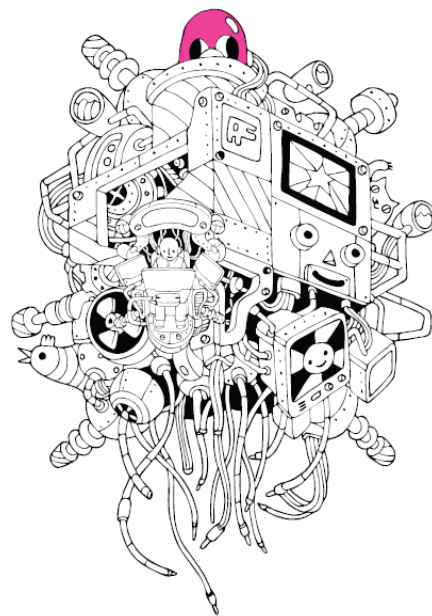
널 포인터 NULL은 숫자 0을 의미한다. 그리고 0은 0번지를 뜻하는 것이 아니라, 아무것도 가리키지 않는다는 의미로 해석이 된다.





Chapter 12가 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 13. 포인터와 배열! 함께 이해하기

윤성우의 열혈 C 프로그래밍



Chapter 13-1. 포인터와 배열의 관계

윤성우 저 열혈강의 C 프로그래밍 개정판

배열의 이름은 무엇을 의미하는가?

아래의 예제에서 보이듯이 배열의 이름은 배열의 시작 주소 값을 의미하는(배열의 첫 번째 요소를 가리키는) 포인터이다. 단순히 주소 값이 아닌 포인터인 이유는 **메모리 접근에 사용되는 * 연산이 가능하기** 때문이다.

```
int main(void)
{
    int arr[3]={0, 1, 2};
    printf("배열의 이름: %p \n", arr);
    printf("첫 번째 요소: %p \n", &arr[0]);
    printf("두 번째 요소: %p \n", &arr[1]);
    printf("세 번째 요소: %p \n", &arr[2]);
    // arr = &arr[i]; // 이 문장은 컴파일 에러를 일으킨다.
    return 0;
}
```

배열의 이름은 변수가 아닌 상수 형태의 포인터이기에 대입연산이 불가능하다.

비교조건	비교대상	포인터 변수	배열의 이름
이름이 존재하는가?		존재한다	존재한다
무엇을 나타내거나 저장하는가?		메모리의 주소 값	메모리의 주소 값
주소 값의 변경이 가능한가?		가능하다	불가능하다.

배열 이름과 포인터 변수의 비교

배열의 이름: 0012FF50
 첫 번째 요소: 0012FF50
 두 번째 요소: 0012FF54
 세 번째 요소: 0012FF58

실행결과



배열 요소간 주소 값의 크기는 4바이트임을 알 수 있다(모든 요소가 붙어있다는 의미).

1차원 배열 이름의 포인터 형

1차원 배열 이름의 포인터 형 결정하는 방법

- 배열의 이름이 가리키는 변수의 자료형을 근거로 판단
- int형 변수를 가리키면 int * 형
- double형 변수를 가리키면 double * 형

int arr1[5]; 에서 **arr1**은 int * 형

double arr2[7]; 에서 **arr2**는 double * 형

```
int main(void)
{
    int arr1[3]={1, 2, 3};
    double arr2[3]={1.1, 2.2, 3.3};

    printf("%d %g \n", *arr1, *arr2);
    *arr1 += 100;    배열 이름을 대상으로 포인터 연산
    *arr2 += 120.5;  은 하고 있음에 주목!
    printf("%d %g \n", arr1[0], arr2[0]);
    return 0;
}
```

실행결과

```
1 1.1
101 121.6
```

arr1이 int형 포인터이므로 * 연산의 결과로 4바이트 메모리 공간에 정수를 저장

arr2는 double형 포인터이므로 * 연산의 결과로 8바이트 메모리 공간에 실수를 저장

포인터를 배열의 이름처럼 사용할 수도 있다.

```
int main(void)
{
    int arr[3]={1, 2, 3};
    arr[0] += 5;
    arr[1] += 7;
    arr[2] += 9;
    . . . .
}
```

arr은 int형 포인터이니 int형 포인터를 대상으로 배열접근을 위한 [idx] 연산을 진행한 셈이다.

실제로 포인터 변수 ptr을 대상으로 ptr[0], ptr[1], ptr[2]와 같은 방식으로 메모리 공간에 접근이 가능하다.

```
int main(void)
{
    int arr[3]={15, 25, 35};
    int * ptr=&arr[0];    // int * ptr=arr; 과 동일한 문장

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

포인터 변수를 이용해서 배열의 형태로 메모리 공간에 접근하고 있음에 주목!

실행결과

```
15 15
25 25
35 35
15 15
```

윤성우의 열혈 C 프로그래밍



Chapter 13-2. 포인터 연산

윤성우 저 열혈강의 C 프로그래밍 개정판

포인터를 대상으로 하는 증가 및 감소연산

```
int main(void)
{
    int * ptr1=0x0010;
    double * ptr2=0x0010;
    printf("%p %p \n", ptr1+1, ptr1+2);
    printf("%p %p \n", ptr2+1, ptr2+2);
    printf("%p %p \n", ptr1, ptr2);
    ptr1++;
    ptr2++;
    printf("%p %p \n", ptr1, ptr2);
    return 0;
}
```

적절치 않은 초기화

원편과 같이 포인터 변수에 저장된 값을 대상으로 하는
증가 및 감소연산을 진행할 수 있다(곱셈, 나눗셈 등등
은 불가).

그리고 이것도 포인터 연산의 일종이다.

실행결과

```
00000014 00000018
00000018 00000020
00000010 00000010
00000014 00000018
```

예제의 실행결과를 통해서 다음 사실을 알 수 있다.

- ▶ **int형 포인터 변수** 대상의 증가 감소 연산 시 **sizeof(int)**의 크기만큼 값이 증가 및 감소한다.
- ▶ **double형 포인터 변수** 대상의 증가 감소 연산 시 **sizeof(double)**의 크기만큼 값이 증가 및 감소한다.



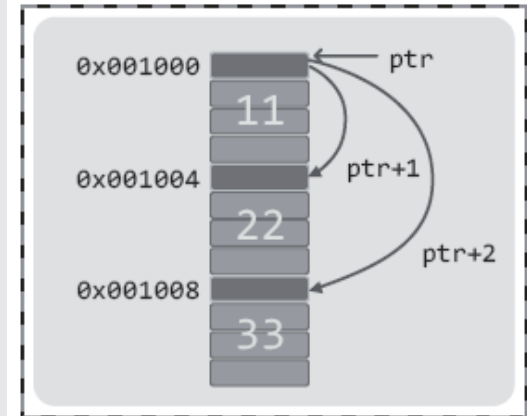
일반화

- ▶ **type형 포인터 변수** 대상의 증가 감소 연산 시 **sizeof(type)**의 크기만큼 값이 증가 및 감소한다.

포인터를 대상으로 하는 증가 및 감소연산

```
int main(void)
{
    int arr[3]={11, 22, 33};
    int * ptr=arr;    // int * ptr=&arr[0]; 과 같은 문장
    printf("%d %d %d \n", *ptr, *(ptr+1), *(ptr+2));

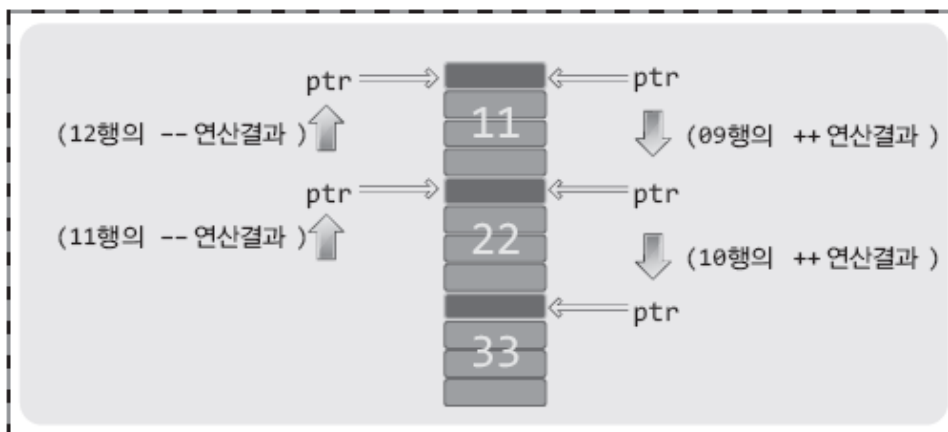
    printf("%d ", *ptr); ptr++;    // printf 함수호출 후, ptr++ 실행
    printf("%d ", *ptr); ptr++;
    printf("%d ", *ptr); ptr--;    // printf 함수호출 후, ptr-- 실행
    printf("%d ", *ptr); ptr--;
    printf("%d ", *ptr); printf("\n");
    return 0;
}
```



11 22 33

11 22 33 22 11

실행결과



int형 포인터 변수의 값은 4씩 증가 및 감소를 하니,
int형 포인터 변수가 **int형 배열을 가리키면**,
int형 포인터 변수의 값을 증가 및 감소시켜서
배열 요소에 순차적으로 접근이 가능하다.

중요한 결론! $arr[i] == *(arr+i)$

```
int main(void)
{
    int arr[3]={11, 22, 33};
    int * ptr=arr;
    printf("%d %d %d \n", *ptr, *(ptr+1), *(ptr+2));
    . . .
}
```

배열이름도 포인터이니, 포인터 변수를 이용한 배열의 접근방식을 배열의 이름에도 사용할 수 있다. 그리고 배열의 이름을 이용한 접근방식도 포인터 변수를 대상으로 사용할 수 있다. 결론은 **arr이 포인터 변수의 이름이건 배열의 이름이건**
 $arr[i] == *(arr+i)$

```
printf("%d %d %d \n", *(ptr+0), *(ptr+1), *(ptr+2));    // *(ptr+0)는 *ptr과 같다.
printf("%d %d %d \n", ptr[0], ptr[1], ptr[2]);
printf("%d %d %d \n", *(arr+0), *(arr+1), *(arr+2));    // *(arr+0)는 *arr과 같다.
printf("%d %d %d \n", arr[0], arr[1], arr[2]);
```

윤성우의 열혈 C 프로그래밍



Chapter 13-3. 상수 형태의 문자열을
가리키는 포인터

윤성우 저 열혈강의 C 프로그래밍 개정판

두 가지 형태의 문자열 표현

```
char str1[ ] = "My String";
char * str2 = "Your String";
```



문자열의 저장방식



```
int main(void)
{
    char * str = "Your team";
    str = "Our team"; // 의미 있는 문장
    ....
}
```

```
int main(void)
{
    char str[ ] = "Your team";
    str = "Our team"; // 의미 없는 문장
    ....
}
```

str1은 문자열이 저장된 배열이다. 즉, 문자 배열이다. 따라서 변수성향의 문자열이다.

str2는 문자열의 주소 값을 저장한다. 즉, 자동 할당된 문자열의 주소 값을 저장한다. 따라서 상수성향의 문자열이다.

두 가지 형태의 문자열 표현의 예

```
int main(void)
{
    char str1[]="My String";    // 변수 형태의 문자열
    char * str2="Your String";  // 상수 형태의 문자열
    printf("%s %s \n", str1, str2);

    str2="Our String";    // 가리키는 대상 변경
    printf("%s %s \n", str1, str2);

    str1[0]='X';    // 문자열 변경 성공!
    str2[0]='X';    // 문자열 변경 실패!
    printf("%s %s \n", str1, str2);
    return 0;
}
```

변수 성향의 **str1**에 저장된 문자열은 변경이 가능!

반면 상수 성향의 **str2**에 저장된 문자열은 변경이 불가능!

간혹 상수 성향의 문자열 변경도 허용하는 컴파일러가 있으나, 이러한 형태의 변경은 바람직하지 못하다!



어디서든 선언할 수 있는 상수 형태의 문자열

```
char * str = "Const String";
```



문자열 저장 후 주소 값 반환

```
char * str = 0x1234;
```

문자열이 먼저 할당된 이후에
그 때 반환되는 주소 값이 저장되는 방식이다.

```
printf("Show your string");
```



문자열 저장 후 주소 값 반환

```
printf(0x1234);
```

위와 동일하다.
문자열은 선언 된 위치로 주소 값이 반환된다.

```
WhoAreYou("Hong");
```



문자열을 전달받는 함수의 선언

```
void WhoAreYou(char * str) { . . . }
```

문자열의 전달만 보더라도
함수의 매개변수 형(type)을 짐작할 수 있다.

윤성우의 열혈 C 프로그래밍



Chapter 13-4. 포인터 변수로 이뤄진
배열: 포인터 배열

윤성우 저 열혈강의 C 프로그래밍 개정판

포인터 배열의 이해

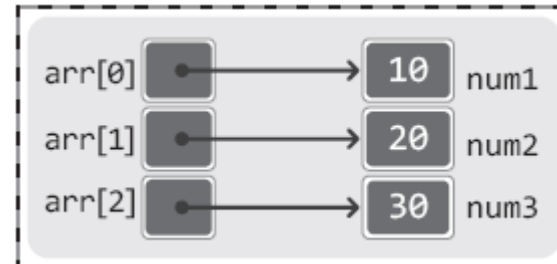
```
int * arr1[20];    // 길이가 20인 int형 포인터 배열 arr1
double * arr2[30]; // 길이가 30인 double형 포인터 배열 arr2
```

```
int main(void)
{
    int num1=10, num2=20, num3=30;
    int* arr[3]={&num1, &num2, &num3};

    printf("%d \n", *arr[0]);
    printf("%d \n", *arr[1]);
    printf("%d \n", *arr[2]);
    return 0;
}
```

실행결과

10
20
30



포인터 배열이라 해서 일반 배열의 선언과 차이가 나지는 않는다.

변수의 자료형을 표시하는 위치에 **int**나 **double**을 대신해서 **int *** 나 **double *** 가 올 뿐이다.

문자열을 저장하는 포인터 배열

```
int main(void)
{
    char * strArr[3]={"Simple", "String", "Array"};
    printf("%s \n", strArr[0]);
    printf("%s \n", strArr[1]);
    printf("%s \n", strArr[2]);
    return 0;
}
```

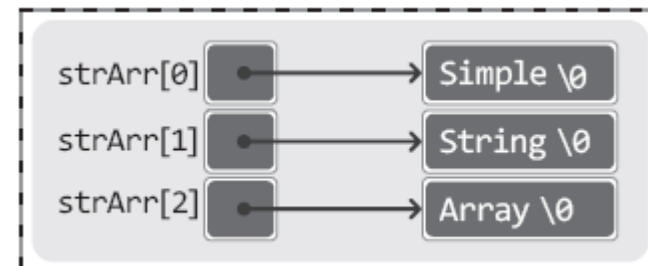
실행결과

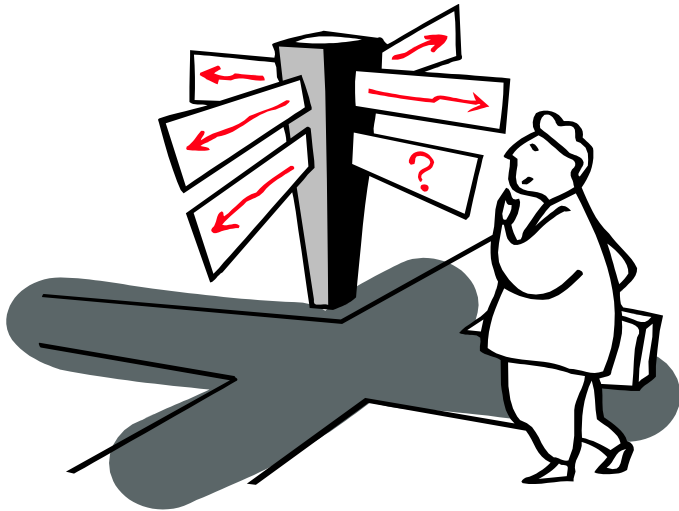
Simple
String
Array

```
char * strArr[3]={"Simple", "String", "Array"};
```



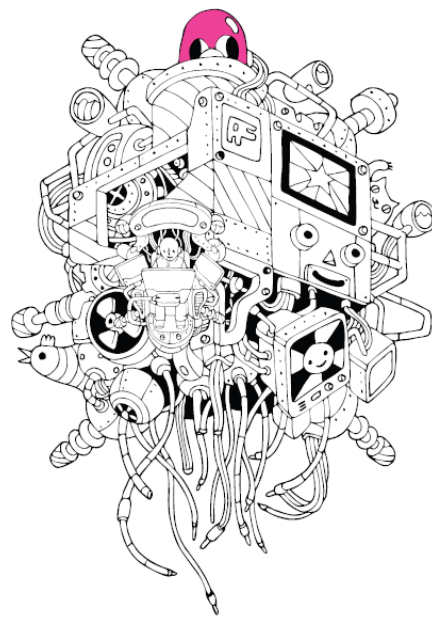
```
char * strArr[3]={0x1004, 0x1048, 0x2012};    // 반환된 주소 값은 임의로 결정하였다.
```





Chapter 13이 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 14. 포인터와 함수에 대한 이해

윤성우의 열혈 C 프로그래밍



Chapter 14-1. 함수의 인자로
배열 전달하기

윤성우 저 열혈강의 C 프로그래밍 개정판

인자전달의 기본방식은 값의 복사이다!

```
int SimpleFunc(int num) { . . . . }
int main(void)
{
    int age=17;
    SimpleFunc(age);
    . . . .
}
```

age에 저장된 값이 매개변수 num에 복사가 된다.

실제 전달되는 것은 age가 아니라 age에 저장된 값이다.

배열을 함수의 매개변수에 전달하는 이유는 함수 내에서 배열에 저장된 값을 참조하도록 하기 위함이다. 그런데 배열을 통째로 전달하지 않아도 이러한 일이 가능하다.

위의 코드에서 보이는 바와 같이, 배열을 함수의 인자로 전달하려면 배열을 통째로 복사할 수 있도록 배열이 매개변수로 선언되어야 한다. 그러나 C언어는 매개변수로 배열의 선언을 허용하지 않는다. 결론! 배열을 통째로 복사하는 방법은 C언어에 존재하지 않는다.

따라서 배열을 통째로 복사해서 전달하는 방식 대신에, 배열의 주소 값을 전달하는 방식을 대신 취한다.



배열을 함수의 인자로 전달하는 방식

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

배열의 이름은 int형 포인터!

배열의 이름은 int형 포인터! 따라서 int형 포인터 변수에 배열의 이름이 지니는 주소 값을 저장할 수 있다.



위의 예제를 통해서 다음과 같은 코드의 구성이 가능함을 유추할 수 있다.

```
int main(void)
{
    int arr[3]={1, 2, 3};
    SimpleFunc(arr);
    . . . .
}
```

배열이름 arr이 지니는 주소 값의 전달

```
void SimpleFunc(int * param)
{
    printf("%d %d", param[0], param[1]);
}
```

포인터 변수를 이용해서도 배열의 형태로

접근가능!

배열이름 arr인 int형 포인터이므로

매개변수는 int형 포인터 변수!

배열을 함수의 인자로 전달하는 예제

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

int main(void)
{
    int arr1[3]={1, 2, 3};
    int arr2[5]={4, 5, 6, 7, 8};
    ShowArrayElem(arr1, sizeof(arr1) / sizeof(int));
    ShowArrayElem(arr2, sizeof(arr2) / sizeof(int));
    return 0;
}
```

```
1 2 3
4 5 6 7 8
```

실행결과

실행결과

```
2 3 4
4 5 6
7 8 9
```

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

void AddArrayElem(int * param, int len, int add)
{
    int i;
    for(i=0; i<len; i++)
        param[i] += add;
}

int main(void)
{
    int arr[3]={1, 2, 3};
    AddArrayElem(arr, sizeof(arr) / sizeof(int), 1);
    ShowArrayElem(arr, sizeof(arr) / sizeof(int));
    AddArrayElem(arr, sizeof(arr) / sizeof(int), 2);
    ShowArrayElem(arr, sizeof(arr) / sizeof(int));
    AddArrayElem(arr, sizeof(arr) / sizeof(int), 3);
    ShowArrayElem(arr, sizeof(arr) / sizeof(int));
    return 0;
}
```

배열을 함수의 인자로 전달받는 함수의 또 다른 선언

```
void ShowArrayElem (int * param, int len) { . . . . }
void AddArrayElem (int * param, int len, int add) { . . . . }
```



동일한 선언

```
void ShowArrayElem (int param[], int len) { . . . . }
void AddArrayElem (int param[], int len, int add) { . . . . }
```

매개변수의 선언에서는 **int * param**과 **int param[]**이 동일한 선언이다. 따라서 배열을 인자로 전달받는 경우에는 **int param[]**이 더 의미있어 보이므로 주로 사용된다.

```
int main(void)
{
    int arr[3]={1, 2, 3};
    int * ptr=arr;    // int ptr[]=arr; 로 대체 불가능
    . . . .
}
```

하지만 그 이외의 영역에서는 **int * ptr**의 선언을 **int ptr[]**으로 대체할 수 없다.

윤성우의 열혈 C 프로그래밍



Chapter 14-2. Call-by-value
vs. Call-by-reference

윤성우 저 열혈강의 C 프로그래밍 개정판

값을 전달하는 형태의 함수호출: Call-by-value

함수를 호출할 때 단순히 값을 전달하는 형태의 함수호출을 가리켜 **Call-by-value**라 하고, 메모리의 접근에 사용되는 주소 값을 전달하는 형태의 함수호출을 가리켜 **Call-by-reference**라 한다. 즉, Call-by-value와 Call-by-reference를 구분하는 기준은 함수의 인자로 전달되는 대상에 있다.

```
void NoReturnType(int num)
{
    if(num<0)
        return;
    . . . .
}
```

call-by-value

call-by-value와 call-by-reference라는 용어를 기준으로 구분하는 것이 중요한 게 아니다.

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

call-by-reference

중요한 것은 각 함수의 특징을 이해하고 적절한 형태의 함수를 정의하는 것이다.

call-by-value 형태의 함수에서는 **함수 외부에 선언된 변수에 접근이 불가능**하다. 그러나 call-by-reference 형태의 함수에서는 **외부에 선언된 변수에 접근이 가능**하다.

잘못 적용된 Call-by-value

```
void Swap(int n1, int n2)
{
    int temp=n1;
    n1=n2;
    n2=temp;
    printf("n1 n2: %d %d \n", n1, n2);
}

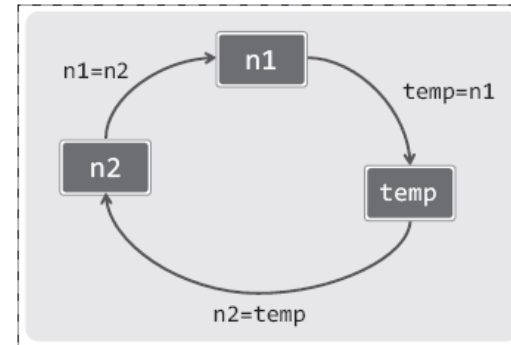
int main(void)
{
    int num1=10;
    int num2=20;
    printf("num1 num2: %d %d \n", num1, num2);

    Swap(num1, num2);    // num1과 num2에 저장된 값이 서로 바뀌길 기대!
    printf("num1 num2: %d %d \n", num1, num2);
    return 0;
}
```

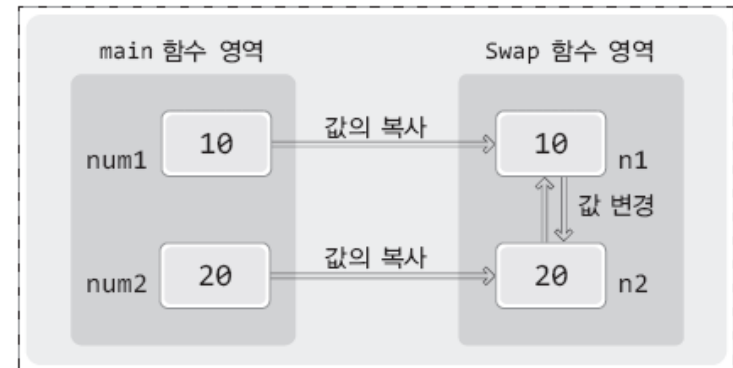
```
num1 num2: 10 20
n1 n2: 20 10
num1 num2: 10 20
```

*call-by-value*가 적절치 않은 경우

실행결과



Swap 함수 내에서의 값의 교환

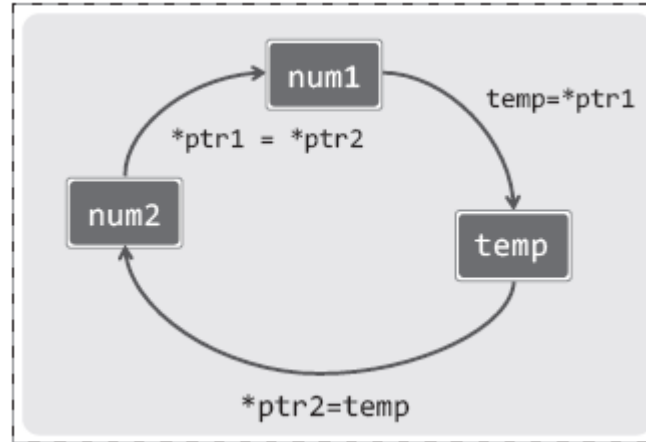


Swap 함수 내에서의 값의 교환은 외부에 영향을 주지 않는다.

주소 값을 전달하는 형태의 함수호출: Call-by-reference

```
void Swap(int * ptr1, int * ptr2)
{
    int temp = *ptr1;
    *ptr1 = *ptr2;
    *ptr2 = temp;
}

int main(void)
{
    int num1=10;
    int num2=20;
    printf("num1 num2: %d %d \n", num1, num2);
    Swap(&num1, &num2);
    printf("num1 num2: %d %d \n", num1, num2);
    return 0;
}
```



Swap 함수 내에서 함수 외부에 있는 변수간
값의 교환

```
num1 num2: 10 20
num1 num2: 20 10
```

실행결과

Swap 함수 내에서의 *ptr1은 main 함수의 num1
Swap 함수 내에서의 *ptr2는 main 함수의 num2
를 의미하게 된다.

scanf 함수호출 시 & 연산자를 붙이는 이유는?

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

변수 *num* 앞에 & 연산자를 붙이는 이유는?

scanf 함수 내에서 외부에 선언된 변수 *num*에 접근 하기 위해서는 *num*의 주소 값을 알아야 한다. 그래서 scanf 함수는 변수의 주소 값을 요구한다.

```
int main(void)
{
    char str[30];
    scanf("%s", str);
    . . . .
}
```

배열 이름 *str* 앞에 & 연산자를 붙이지 않는 이유는?

*str*은 배열의 이름이고 그 자체가 주소 값이기 때문에 & 연산자를 붙이지 않는다. *str*을 전달함은 scanf 함수 내부로 배열 *str*의 주소 값을 전달하는 것이다.



윤성우의 열혈 C 프로그래밍



Chapter 14-3.포인터 대상의 const 선언

윤성우 저 열혈강의 C 프로그래밍 개정판

포인터 변수의 참조대상에 대한 const 선언

```
int main(void)
{
    int num=20;
    const int * ptr=&num;
    *ptr=30;    // 컴파일 에러!
    num=40;     // 컴파일 성공!
    . . . .
}
```

원편의 *const* 선언이 갖는 의미

포인터 변수 *ptr*을 이용해서 *ptr*이 가리키는 변수에 저장된 값을 변경하는 것을 허용하지 않겠습니다!

그러나 변수 *num*에 저장된 값 자체의 변경이 불가능한 것은 아니다.

다만 *ptr*을 통한 변경을 허용하지 않을뿐이다.



포인터 변수의 상수화

```
int main(void)
{
    int num1=20;
    int num2=30;
    int * const ptr=&num1;
    ptr=&num2;    // 컴파일 에러!
    *ptr=40;      // 컴파일 성공!
    . . . .
}
```

원편의 *const* 선언이 갖는 의미

포인터 변수 *ptr*에 저장된 값을 상수화 하겠다. 즉, *ptr*에 저장된 값은 변경이 불가능하다. *ptr*이 가리키는 대상의 변경을 허용하지 않는다.

```
const int * ptr=&num;
int * const ptr=&num;
```



```
const int * const ptr=&num;
```

두 가지 *const* 선언을 동시에 할 수 있다.

const 선언이 갖는 의미

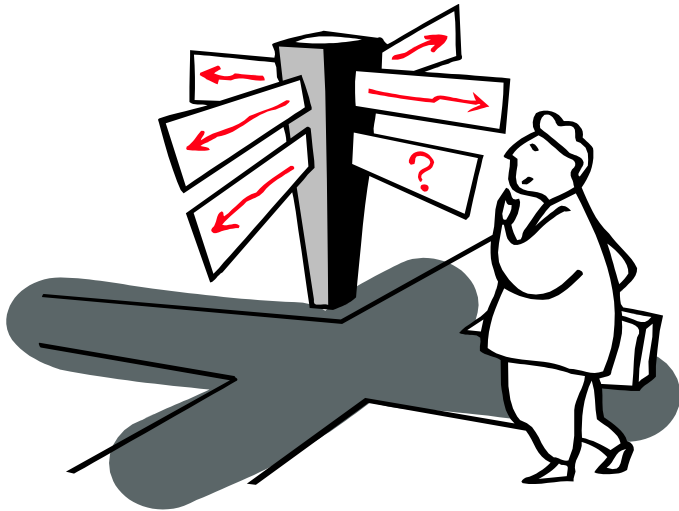
```
int main(void)
{
    double PI=3.1415;
    double rad;
    PI=3.07; // 실수로 잘못 삽입된 문장, 컴파일 시 발견 안됨
    scanf("%lf", &rad);
    printf("circle area %f \n", rad*rad*PI);
    return 0;
}
```



안전성이 높아진 코드

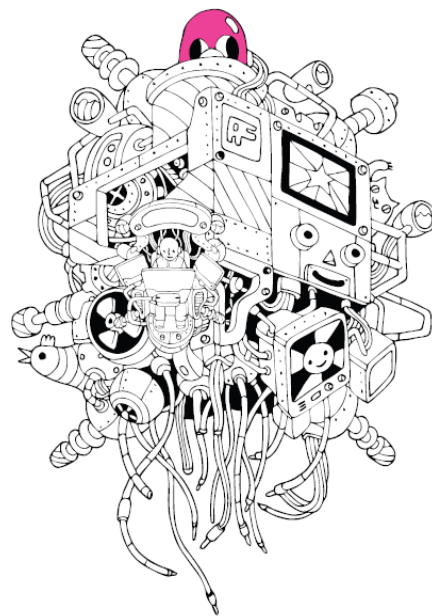
```
int main(void)
{
    const double PI=3.1415;
    double rad;
    PI=3.07; // 컴파일 시 발견되는 오류상황
    scanf("%lf", &rad);
    printf("circle area %f \n", rad*rad*PI);
    return 0;
}
```

const 선언은 추가적인 기능을 제공하기 위한 것이 아니라, 코드의 안전성을 높이기 위한 것이다. 따라서 이러한 **const**의 선언을 소홀히하기 쉬운데, **const**의 선언과 같이 코드의 안전성을 높이는 선언은 가치가 매우 높은 선언이다.



Chapter 14가 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 16. 다차원 배열

윤성우의 열혈 C 프로그래밍



Chapter 16-1. 다차원 배열의 이해와 활용

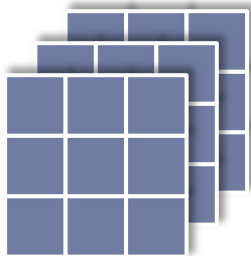
윤성우 저 열혈강의 C 프로그래밍 개정판

2차원, 3차원 배열 OK! 4차원, 5차원 배열 NO!

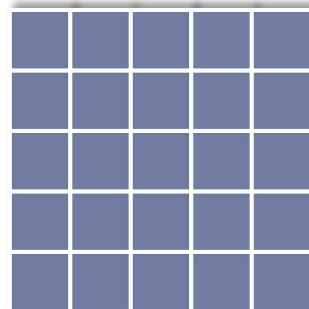
<code>int arrOneDim[10];</code>	길이가 10인 1차원 int형 배열
<code>int arrTwoDim[5][5];</code>	가로, 세로의 길이가 각각 5인 2차원 int형 배열
<code>int arrThreeDim[3][3][3];</code>	가로, 세로, 높이의 길이가 각각 3인 3차원 int형 배열



1차원 배열 *arrOneDim*



3차원 배열 *arrThreeDim*



2차원 배열 *arrTwoDim*

문법적으로는 4차원 5차원 배열의 선언도 가능
하지만 그것은 의미를 부여하기 힘든, 의미가
없는 배열이다.



다차원 배열을 의미하는 2차원 배열의 선언

2차원 배열의 선언 방식 → **TYPE arr[세로길이][가로길이];**

	1열	2열	3열	4열
1행	[0][0]	[0][1]	[0][2]	[0][3]
2행	[1][0]	[1][1]	[1][2]	[1][3]
3행	[2][0]	[2][1]	[2][2]	[2][3]

int arr1[3][4];

	1열	2열	3열	4열	5열	6열
1행	[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]
2행	[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]

int arr2[2][6];

```
int main(void)
{
    int arr1[3][4];
    int arr2[7][9];
    printf("세로3, 가로4: %d \n", sizeof(arr1));
    printf("세로7, 가로9: %d \n", sizeof(arr2));
    return 0;
}
```

실행결과

세로3, 가로4: 48
세로7, 가로9: 252

2차원 배열요소의 접근

```
int arr[3][3];
```



배열 생성

	1열	2열	3열
1행	0	0	0
2행	0	0	0
3행	0	0	0

```
arr[0][0]=1;
```



0 0 접근

	1열	2열	3열
1행	1	0	0
2행	0	0	0
3행	0	0	0

```
arr[0][1]=2;
```



0 1 접근

	1열	2열	3열
1행	1	2	0
2행	0	0	0
3행	0	0	0

일반화

```
arr[N-1][M-1]=20;  
printf("%d", arr[N-1][M-1]);
```

세로 N , 가로 M 의 위치에 값을 저장 및 참조

```
arr[2][1]=5;
```



2 1 접근

	1열	2열	3열
1행	1	2	0
2행	0	0	0
3행	0	5	0



2차원 배열요소 접근관련 예제

```
int main(void)
{
    int villa[4][2];
    int popu, i, j;
    /* 가구별 거주인원 입력 받기 */
    for(i=0; i<4; i++)
    {
        for(j=0; j<2; j++)
        {
            printf("%d층 %d호 인구수: ", i+1, j+1);
            scanf("%d", &villa[i][j]);
        }
    }

    /* 빌라의 층별 인구수 출력하기 */
    for(i=0; i<4; i++)
    {
        popu=0;
        popu += villa[i][0];
        popu += villa[i][1];
        printf("%d층 인구수: %d \n", i+1, popu);
    }
    return 0;
}
```

1층 1호 인구수: 2
 1층 2호 인구수: 4
 2층 1호 인구수: 3
 2층 2호 인구수: 5
 3층 1호 인구수: 2
 3층 2호 인구수: 6
 4층 1호 인구수: 4
 4층 2호 인구수: 3
 1층 인구수: 6
 2층 인구수: 8
 3층 인구수: 8
 4층 인구수: 7

실행결과

2차원 배열의 메모리상 할당의 형태

0x1001번지, 0x1002번지, 0x1003번지, 0x1004번지, 0x1005번지

1차원적 메모리의 주소 값

0x12-0x24번지, 0x12-0x25번지, 0x12-0x26번지, 0x12-0x27번지

0x13-0x24번지, 0x13-0x25번지, 0x13-0x26번지, 0x13-0x27번지

0x14-0x24번지, 0x14-0x25번지, 0x14-0x26번지, 0x14-0x27번지

. . . .

2차원적 메모리의 주소 값

실제 메모리는 1차원의 형태로 주소 값이 지정이 된다.

따라서 아래와 같은 형태로 2차원 배열의 주소 값이 지정된다.

0x1000	0	arr[0][0]
0x1004	1	arr[0][1]
0x1008	2	arr[1][0]
0x100C	3	arr[1][1]
0x1010	4	arr[2][0]
0x1014	5	arr[2][1]

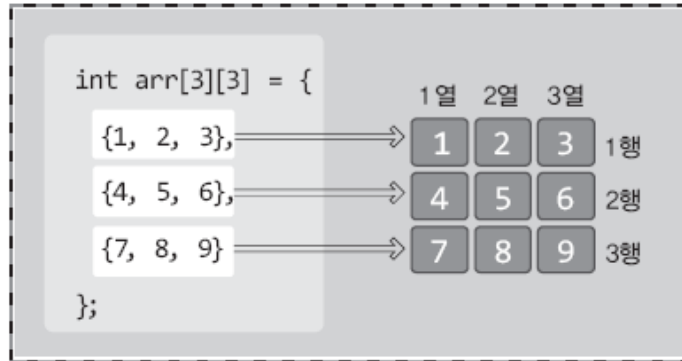
2차원 배열의
실제 메모리
할당형태

실행결과

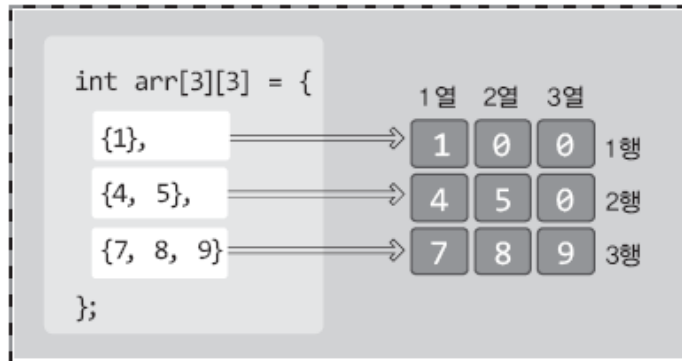
002AFD54
002AFD58
002AFD5C
002AFD60
002AFD64
002AFD68

```
int main(void)
{
    int arr[3][2];
    int i, j;
    for(i=0; i<3; i++)
        for(j=0; j<2; j++)
            printf("%p \n", &arr[i][j]);
    return 0;
}
```

2차원 배열 선언과 동시에 초기화 하기



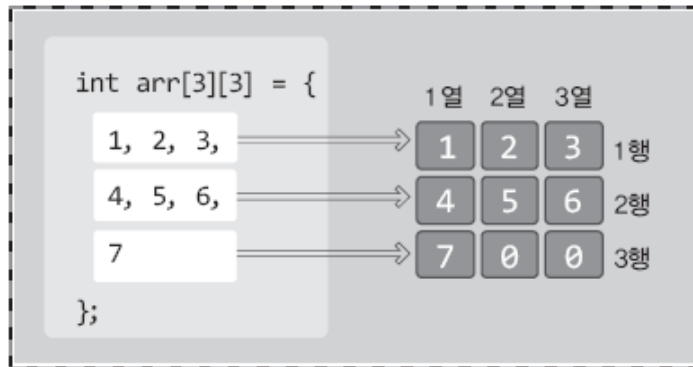
초기화 리스트 안에는 행 단위로 초기화할 값들을 별도의 중괄호로 명시한다.



```
int arr[3][3] = {
    {1, 0, 0},
    {4, 5, 0},
    {7, 8, 9}
};
```

채워지지 않은 빈 공간은 0으로 채워진다.

2차원 배열 선언과 동시에 초기화 하기2



별도의 중괄호를 사용하지 않으면 좌 상단부터 시작해서 우 하단으로 순서대로 초기화된다.



한 줄에 표현해도 된다.

```
int arr[3][3]={1, 2, 3, 4, 5, 6, 7};
```



마찬가지로 빈 공간은 0으로 채워진다.

```
int arr[3][3]={1, 2, 3, 4, 5, 6, 7, 0, 0};
```


2차원 배열 선언과 동시에 초기화 하기(예제)

```
int main(void)
{
    int i, j;

    /* 2차원 배열 초기화의 예 1 */
    int arr1[3][3]={
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    /* 2차원 배열 초기화의 예 2 */
    int arr2[3][3]={
        {1},
        {4, 5},
        {7, 8, 9}
    };

    /* 2차원 배열 초기화의 예 3 */
    int arr3[3][3]={1, 2, 3, 4, 5, 6, 7};
```

```
    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
            printf("%d ", arr1[i][j]);
        printf("\n");
    }
    printf("\n");

    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
            printf("%d ", arr2[i][j]);
        printf("\n");
    }
    printf("\n");

    for(i=0; i<3; i++)
    {
        for(j=0; j<3; j++)
            printf("%d ", arr3[i][j]);
        printf("\n");
    }
    return 0;
}
```

실행결과

```
1 2 3
4 5 6
7 8 9

1 0 0
4 5 0
7 8 9

1 2 3
4 5 6
7 0 0
```

배열의 크기를 알려주지 않고 초기화하기

```
int arr[][]={1, 2, 3, 4, 5, 6, 7, 8};
```

8 by 1 ??

4 by 2 ??

2 by 4 ??

두 개가 모두 비면 컴파일러가 채워 넣을 숫자를 결정하지 못한다.

```
int arr1[][4]={1, 2, 3, 4, 5, 6, 7, 8};
```

```
int arr2[][2]={1, 2, 3, 4, 5, 6, 7, 8};
```

세로 길이만 생략할 수 있도록 약속되어 있다.



컴파일러가 세로 길이를 계산해 준다.

```
int arr1[2][4]={1, 2, 3, 4, 5, 6, 7, 8};
```

```
int arr2[4][2]={1, 2, 3, 4, 5, 6, 7, 8};
```

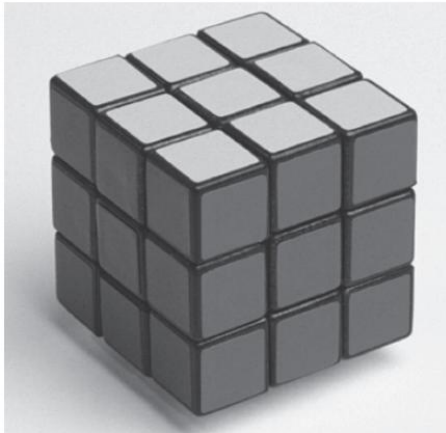
윤성우의 열혈 C 프로그래밍



Chapter 16-2. 3차원 배열

윤성우 저 열혈강의 C 프로그래밍 개정판

3차원 배열의 논리적 구조



```
int main(void)
{
    int arr1[2][3][4];
    double arr2[5][5][5];
    printf("높이2, 세로3, 가로4 int형 배열: %d \n", sizeof(arr1));
    printf("높이5, 세로5, 가로5 double형 배열: %d \n", sizeof(arr2));
    return 0;
}
```

```
높이2, 세로3, 가로4 int형 배열: 96
높이5, 세로5, 가로5 double형 배열: 1000
```

실행결과

```
int arr1[2][3][4];
```

높이 2, 세로 3, 가로 4인 int형 3차원 배열(세로 3, 가로 4인 배열이 두 개 겹친 형태)

```
double arr2[5][5][5];
```

높이, 세로, 가로가 모두 5인 double형 3차원 배열(세로 5, 가로 5인 배열이 5개 겹친 형태)

3차원 배열의 선언과 접근

```
int main(void)
{
    int mean=0, i, j;
    int record[3][3][2]={
        {
            {70, 80},    // A 학급 학생 1의 성적
            {94, 90},    // A 학급 학생 2의 성적
            {70, 85}     // A 학급 학생 3의 성적
        },
        {
            {83, 90},    // B 학급 학생 1의 성적
            {95, 60},    // B 학급 학생 2의 성적
            {90, 82}     // B 학급 학생 3의 성적
        },
        {
            {98, 89},    // C 학급 학생 1의 성적
            {99, 94},    // C 학급 학생 2의 성적
            {91, 87}     // C 학급 학생 3의 성적
        }
    };
};
```

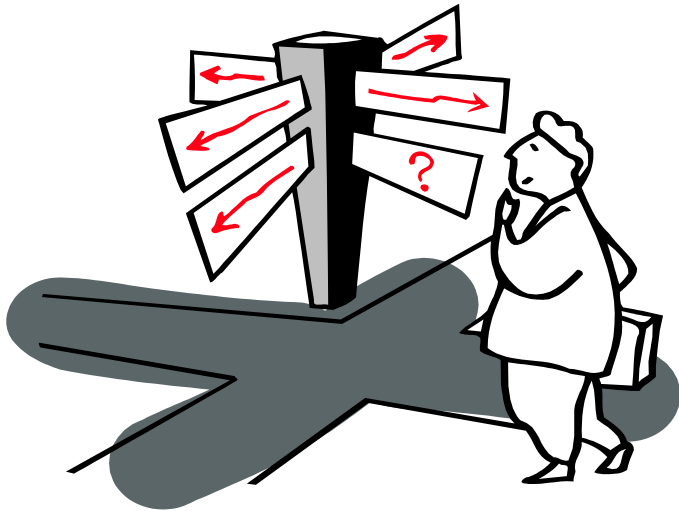
```
for(i=0; i<3; i++)
    for(j=0; j<2; j++)
        mean += record[0][i][j];
printf("A 학급 전체 평균: %g \n", (double)mean/6);

mean=0;
for(i=0; i<3; i++)
    for(j=0; j<2; j++)
        mean += record[1][i][j];
printf("B 학급 전체 평균: %g \n", (double)mean/6);

mean=0;
for(i=0; i<3; i++)
    for(j=0; j<2; j++)
        mean += record[2][i][j];
printf("C 학급 전체 평균: %g \n", (double)mean/6);
return 0;
}
```

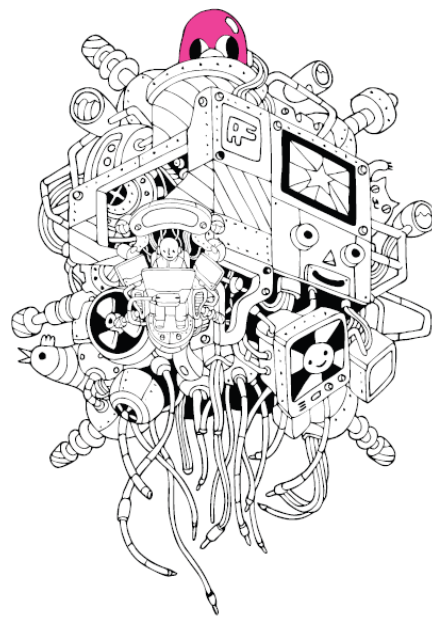
A 학급 전체 평균: 81.5
B 학급 전체 평균: 83.3333
C 학급 전체 평균: 93

실행결과



Chapter 1b이 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 17. 포인터의 포인터

윤성우의 열혈 C 프로그래밍

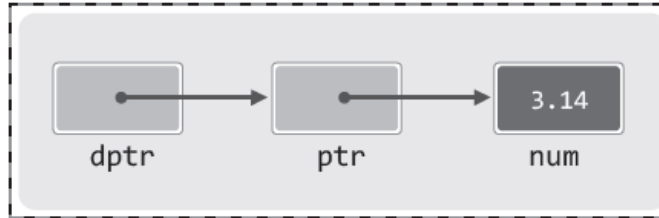


Chapter 17-1. 포인터의 포인터에 대한 이해

윤성우 저 열혈강의 C 프로그래밍 개정판

포인터 변수를 가리키는 이중 포인터 변수

```
int main(void)
{
    double num=3.14;
    double * ptr=&num;
    double ** dptr =&ptr;
    .....
}
```

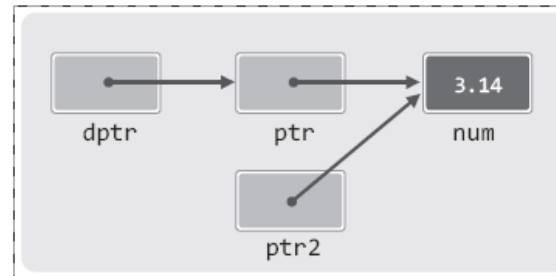


포인터 변수의 주소 값을 저장하는 것이 이중 포인터 변수(더블 포인터 변수)이다.

위의 상황에서 *dptr은 포인터 변수 ptr을...
*(dptr)은 변수 num을 의미하게 된다.

```
int main(void)
{
    double num = 3.14;
    double *ptr = &num;
    double **dptr = &ptr;
    double *ptr2;

    printf("%9p %9p \n", ptr, *dptr);
    printf("%9g %9g \n", num, **dptr);
    ptr2 = *dptr; // ptr2 = ptr 과 같은 문장
    *ptr2 = 10.99;
    printf("%9g %9g \n", num, **dptr);
    return 0;
}
```



이 상황에서 변수 num에 접근하는 네 가지 방법은?

0032FD00	0032FD00
3.14	3.14
10.99	10.99

실행결과

포인터 변수의 Swap 1

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

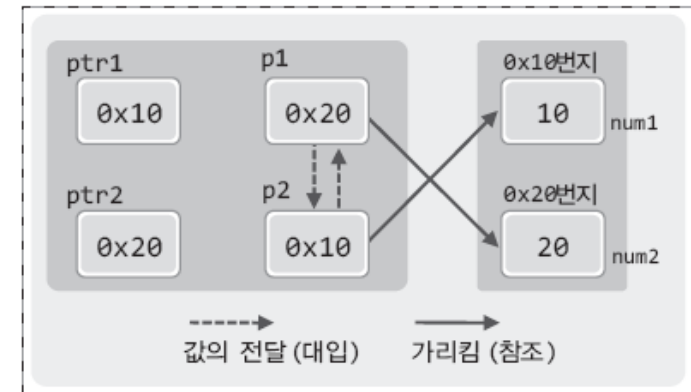
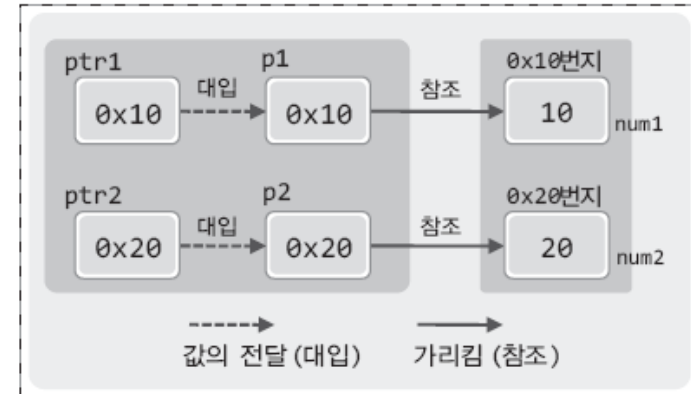
*ptr1과 ptr2의 swap은 성공하는
가? 문제점은?*

```
int main(void)
{
    int num1=10, num2=20;
    int *ptr1, *ptr2;
    ptr1=&num1, ptr2=&num2;
    printf("*ptr1, *ptr2: %d %d \n", *ptr1, *ptr2);

    SwapIntPtr(ptr1, ptr2);
    printf("*ptr1, *ptr2: %d %d \n", *ptr1, *ptr2);
    return 0;
}
```

```
*ptr1, *ptr2: 10 20
*ptr1, *ptr2: 10 20
```

실행결과



원래 예제의 실행결과! 결과적으로 ptr1과 ptr2에 저장된 값은 서로 바뀌지 않는다.

포인터 변수의 Swap 2

```
void SwapIntPtr(int **dp1, int **dp2)
{
    int *temp = *dp1;
    *dp1 = *dp2;
    *dp2 = temp;
}
```

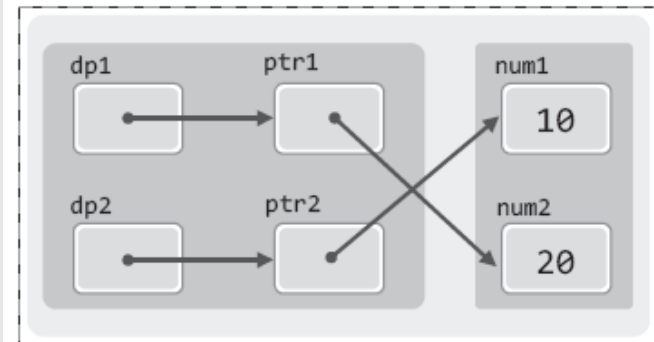
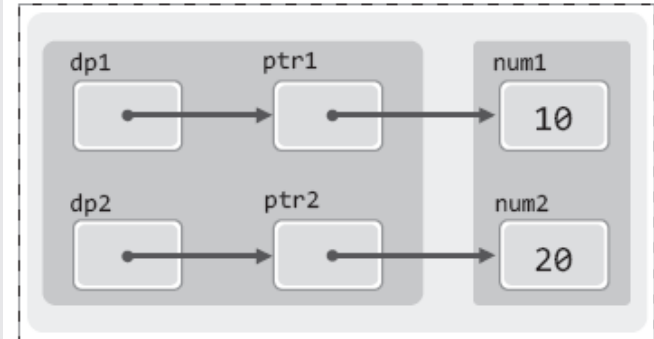
포인터 변수에 저장된 값의 변경
이 목적이므로 포인터 변수의 주소
값을 함수에 전달해야 한다.

```
int main(void)
{
    int num1=10, num2=20;
    int *ptr1, *ptr2;
    ptr1=&num1, ptr2=&num2;
    printf("*ptr1, *ptr2: %d %d \n", *ptr1, *ptr2);

    SwapIntPtr(&ptr1, &ptr2); // ptr1과 ptr2의 주소 값 전달!
    printf("*ptr1, *ptr2: %d %d \n", *ptr1, *ptr2);
    return 0;
}
```

```
*ptr1, *ptr2: 10 20
*ptr1, *ptr2: 20 10
```

실행결과



이중 포인터를 이용해서 두 포인터 변수
의 swap에 성공한다.

포인터 배열과 포인터 배열의 포인터 형

```
int * arr1[20];
double * arr2[30];
```

Ch 13의 후반에 학습한 포인터 변수로 이뤄진 배열(포인터 배열)

`int arr1[3];` 에서 `arr1`의 포인터 형은 `int *` `double arr2[3];` 에서 `arr2`의 포인터 형은 `double *`
 이렇듯 1차원 배열이름의 포인터 형은 배열 이름이 가리키는 대상을 기준으로 결정된다.
 따라서 `int * arr1[20];` 에서 `arr1`의 포인터 형은 `int **`
`double * arr2[30];` 에서 `arr2`의 포인터 형은 `double **`

```
int main(void)
{
    int num1=10, num2=20, num3=30;
    int *ptr1=&num1;
    int *ptr2=&num2;
    int *ptr3=&num3;

    int * ptrArr[]={ptr1, ptr2, ptr3};
    int **dptr=ptrArr;

    printf("%d %d %d \n", *(ptrArr[0]), *(ptrArr[1]), *(ptrArr[2]));
    printf("%d %d %d \n", *(dptr[0]), *(dptr[1]), *(dptr[2]));
    return 0;
}
```

```
10 20 30
10 20 30
```

실행결과

윤성우의 열혈 C 프로그래밍



Chapter 17-2. 다중 포인터 변수와
포인터의 필요성

윤성우 저 열혈강의 C 프로그래밍 개정판

이중 포인터를 가리키는 삼중 포인터

```
int ***tptr;
```

삼중 포인터 변수!

이중 포인터 변수의 주소 값을 담는 용도로 선언된다.

```
int main(void)
{
    int num=100;
    int *ptr=&num;
    int **dptr=&ptr;
    int ***tptr=&dptr;

    printf("%d %d \n", **dptr, ***tptr);
    return 0;
}
```

실행결과

100 100

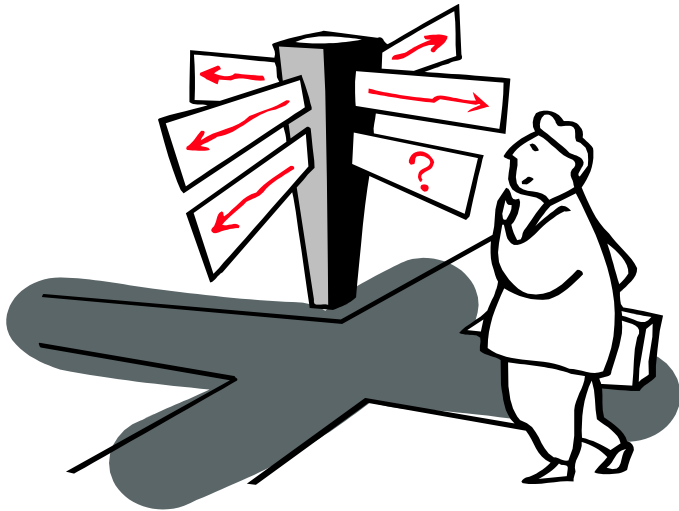
이중 포인터 변수의 개념을 그대로 확장해서 이해할 수 있는 것이 삼중 포인터 변수이다!



포인터의 필요성은 어디서 찾아야 하는가?

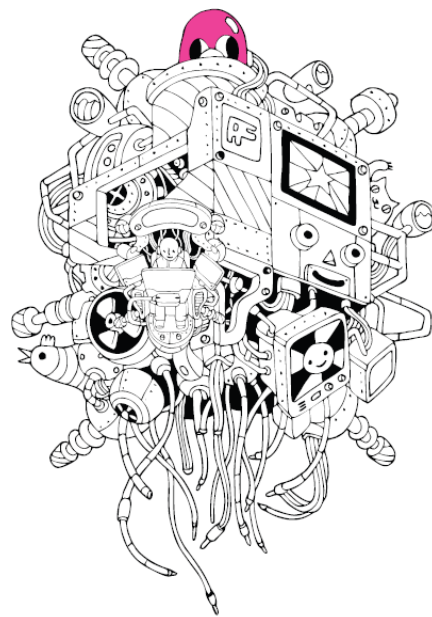
- . scanf 함수와 같이 함수 내에서 함수 외부에 선언된 변수의 접근을 허용하기 위해서.
- . 메모리의 동적 할당 등등 PART 04에서 공부하는 내용을 통해서 포인터의 필요성을 다양하게 이해하게 된다.
- . 향후에 자료구조라는 과목을 공부하게 되면 보다 넓게 필요성을 이해할 수 있게 된다.





Chapter 17이 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 18. 다차원 배열과 포인터의 관계

윤성우의 열혈 C 프로그래밍



Chapter 18-1. 2차원 배열이름의
포인터 형

윤성우 저 열혈강의 C 프로그래밍 개정판

1차원 배열이름과 2차원 배열이름의 포인터 형

```
int arr[10];
```

1차원 배열이므로 arr은 int형 포인터 (int *)

```
int * parr[20];
```

1차원 배열이므로 parr은 int형 이중 포인터 (int **)

```
int arr2d[3][4];
```

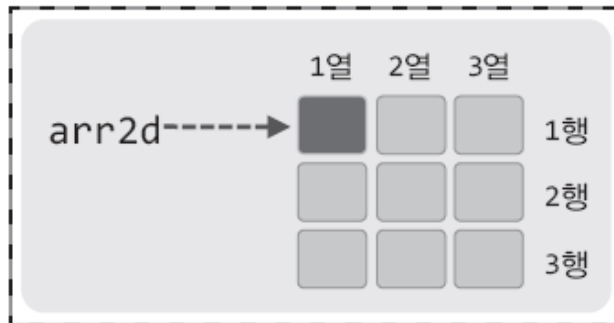
int형 1차원 배열도, int 포인터 형 1차원 배열도 아니므로 arr2d는 int형 포인터 형도, int형 이중 포인터 형도 아니다. 2차원 배열이름의 포인터 형을 결정짓는 방법은 별도로 존재한다.



2차원 배열이름이 가리키는 것들은?

2차원 배열이름의 포인터 형을 결정지으려면 우선 2차원 배열이름이 가리키는 대상이 무엇인지 알아야 한다. 그런데 1차원 배열과 달리 이것만으로 포인터 형이 결정되지 않는다.

```
int arr2d[3][3];
```



배열이름 `arr2d`가 가리키는 것은 인덱스 기준으로 `[0][0]`에 위치한 첫 번째 요소



2차원 배열의 경우 `arr2d[0]`, `arr2d[1]`, `arr2d[2]`도 의미를 지닌다. 각각 1행, 2행, 3행의 첫 번째 요소를 가리키는 주소 값의 의미를 지닌다.

그럼 arr2d와 arr2d[0]는 같은 것인가?

```
int main(void)
{
    int arr2d[3][3];
    printf("%d \n", arr2d);
    printf("%d \n", arr2d[0]);
    printf("%d \n\n", &arr2d[0][0]);

    printf("%d \n", arr2d[1]);
    printf("%d \n\n", &arr2d[1][0]);

    printf("%d \n", arr2d[2]);
    printf("%d \n\n", &arr2d[2][0]);

    printf("sizeof(arr2d): %d \n", sizeof(arr2d));
    printf("sizeof(arr2d[0]): %d \n", sizeof(arr2d[0]));
    printf("sizeof(arr2d[1]): %d \n", sizeof(arr2d[1]));
    printf("sizeof(arr2d[2]): %d \n", sizeof(arr2d[2]));
    return 0;
}
```

arr2d는 2차원 배열 전체를 의미한다. 반면 arr2d[0]는 2차원 배열의 첫 번째 행을 의미한다.

실행결과

4585464

4585464

4585464

4585476

4585476

4585488

4585488

sizeof(arr2d): 36

sizeof(arr2d[0]): 12

sizeof(arr2d[1]): 12

sizeof(arr2d[2]): 12

배열이름 arr2d를 대상으로 sizeof 연산을 하는 경우 배열 전체의 크기를 반환

arr2d[0], arr2d[1], arr2d[2]를 대상으로 sizeof 연산을 하는 경우 각 행의 크기를 반환

배열이름 기반의 포인터 연산

```
int iarr[3];      // iarr은 int형 포인터
double darr[7];   // darr은 double형 포인터
```

```
printf("%p", iarr+1);
printf("%p", darr+1);
```

iarr은 **int형 포인터이기 때문에** +1의 결과로 sizeof(int)의 크기만큼 값이 증가한다.

darr은 **double형 포인터이기 때문에** +1의 결과로 sizeof(double)의 크기만큼 값이 증가한다.

이렇듯 포인터 연산의 결과는 포인터 형에 의존적이다. 따라서 2차원 배열이름의 포인터 형을 결정짓기 위한 힌트는 포인터 연산의 결과를 주목하면 얻을 수 있다.

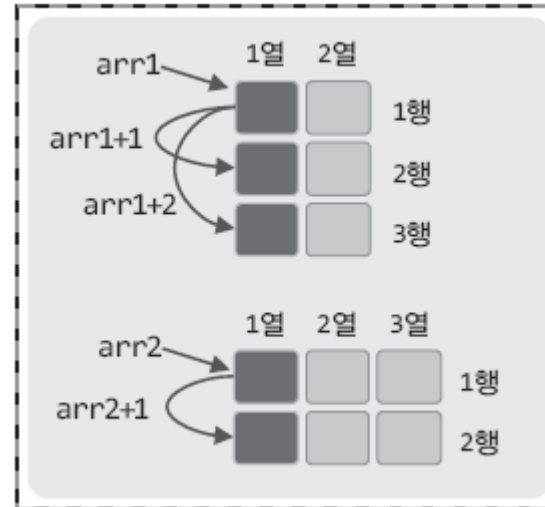


2차원 배열이름 대상의 포인터 연산 결과

```
int main(void)
{
    int arr1[3][2];
    int arr2[2][3];

    printf("arr1: %p \n", arr1);
    printf("arr1+1: %p \n", arr1+1);
    printf("arr1+2: %p \n\n", arr1+2);

    printf("arr2: %p \n", arr2);
    printf("arr2+1: %p \n", arr2+1);
    return 0;
}
```



2차원 배열이름을 대상으로 값
을 1씩 증가 및 감소하는 경우
그 결과는 각 행의 첫 번째 요
소의 주소 값이다.

```
arr1: 004BFBE0
arr1+1: 004BFBE8
arr1+2: 004BFBF0

arr2: 004BFBC0
arr2+1: 004BFBC3
```

arr1과 arr2는 둘 다 int형 2차원 배열이다. 그러나 **가로의 길이가 다르기 때문에** 포인터 연산결과로 증가 및 감소하는 값의 크기에는 차이가 있다. 즉, **arr1과 arr2**는 **의 포인터 형은 일치하지 않는다.**

이렇듯 2차원 배열이름의 포인터 형은 배열의 가로길이에 따라서도 나뉜다. 그리고 이러한 특징 때문에 2차원 배열이름의 포인터 형 결정이 쉽지 않은 것이다.

실행결과

최종결론! 2차원 배열이름의 포인터 형 1

1. 2차원 배열이름의 포인터 형을 결정짓는 두 가지 요소!

1. 가리키는 대상은 무엇인가?
2. 배열이름(포인터)를 대상으로 값을 1 증가 및 감소 시 실제로는 얼마가 증가 및 감소하는가?

2. 왜 다른가?

1차원 배열이름의 자료형은 1차원 배열이름이 가리키는 대상만으로 결정이 되는데 2차원 배열이름의 자료형은 그렇지 않은 이유는?

3. 포인터 형을 통한 메모리의 접근과 주소 값의 증가

1차원 배열의 경우에는 배열이름이 가리키는 대상을 기준으로 메모리의 접근방법과 포인터 연산시의 증가 및 감소의 크기가 결정되었다. 그러나 2차원 배열에서는 위의 두 가지 정보가 모두 존재해야 이 둘을 결정지을 수 있다.

최종결론! 2차원 배열이름의 포인터 형 2

어색하지만 이것이 `arr`의 포인터 형을 설명하는 최선의 방법이다.

`int arr[3][4]`의 포인터 형은?

`int`형 포인터, `double`형 포인터와 같이 딱 떨어지는 명칭이 존재하지 않는다.

1. 가리키는 대상 `int`형 변수
2. 포인터 연산의 결과 `sizeof(int)×4`의 크기단위로 값이 증가 및 감소



이러한 유형의 포인터 변수 `ptr`의 선언

`int (*ptr) [4];`

2차원 배열을 가리키는 포인터 변수
이므로 배열 포인터 변수라 한다.

`int (*ptr) [4]`

`ptr`은 포인터!

`int (*ptr) [4]`

`int`형 변수를 가리키는 포인터 !

`int (*ptr) [4]`

포인터 연산 시 4칸씩 건너뛰는 포인터!

2차원 배열이름의 포인터 형 결정짓는 연습

case 1

```
char (*arr1)[4];  
double (*arr2)[7];
```



“arr1은 char형 변수를 가리키면서, 포인터 연산 시 sizeof(char)×4의 크기단위로 값이 증가 및 감소하는 포인터 변수”

“arr2는 double형 변수를 가리키면서, 포인터 연산 시 sizeof(double)×7의 크기단위로 값이 증가 및 감소하는 포인터 변수”

case 2

“int형 변수를 가리키면서, 포인터 연산 시 sizeof(int)×2의 크기단위로 값이 증가 및 감소하는 포인터 변수 ptr1”

“float형 변수를 가리키면서, 포인터 연산 시 sizeof(float)×5의 크기단위로 값이 증가 및 감소하는 포인터 변수 ptr2”



```
int (*ptr1)[2];  
float (*ptr2)[5];
```

2차원 배열이름의 포인터 관련 예제

```
int main(void)
{
    int arr1[2][2]={
        {1, 2}, {3, 4}
    };
    int arr2[3][2]={
        {1, 2}, {3, 4}, {5, 6}
    };
    int arr3[4][2]={
        {1, 2}, {3, 4}, {5, 6}, {7, 8}
    };
    int (*ptr)[2];
    int i;
    ptr=arr1;
    printf("*** Show 2,2 arr1 **\n");
    for(i=0; i<2; i++)
        printf("%d %d \n", ptr[i][0], ptr[i][1]);

    ptr=arr2;
    printf("*** Show 3,2 arr2 **\n");
    for(i=0; i<3; i++)
        printf("%d %d \n", ptr[i][0], ptr[i][1]);

    ptr=arr3;
    printf("*** Show 4,2 arr3 **\n");
    for(i=0; i<4; i++)
        printf("%d %d \n", ptr[i][0], ptr[i][1]);
    return 0;
}
```

arr1, arr2, arr3는 둘 다 int형 2차원 배열이면서 가로 길이가 같으므로 포인터 형이 동일하다.

실행결과

```
** Show 2,2 arr1 **
1 2
3 4
** Show 3,2 arr2 **
1 2
3 4
5 6
** Show 4,2 arr3 **
1 2
3 4
5 6
7 8
```

윤성우의 열혈 C 프로그래밍



Chapter 18-2. 2차원 배열이름의 특성과 주의사항

윤성우 저 열혈강의 C 프로그래밍 개정판

'배열 포인터'와 '포인터 배열'을 혼동하지 말자

```
int * whoA [4];    // 포인터 배열
int (*whoB) [4];  // 배열 포인터
```

포인터 배열

포인터 변수로 이루어진 배열

배열 포인터

배열을 가리킬 수 있는 포인터 변수

```
int main(void)
{
    int num1=10, num2=20, num3=30, num4=40;
    int arr2d[2][4]={1, 2, 3, 4, 5, 6, 7, 8};
    int i, j;

    int * whoA[4]={&num1, &num2, &num3, &num4};    // 포인터 배열
    int (*whoB)[4]=arr2d;    // 배열 포인터

    printf("%d %d %d %d \n", *whoA[0], *whoA[1], *whoA[2], *whoA[3]);
    for(i=0; i<2; i++)
    {
        for(j=0; j<4; j++)
            printf("%d ", whoB[i][j]);
        printf("\n");
    }
    return 0;
}
```

실행결과

```
10 20 30 40
1 2 3 4
5 6 7 8
```

2차원 배열을 함수의 인자로 전달하기

```
int main(void)
{
    int arr1[2][7];
    double arr2[4][5];
    SimpleFunc(arr1, arr2);
    . . .
}
```

매개변수의 선언 위치에서만 동일한 선언으로 간주된다.

```
void SimpleFunc( int (*parr1)[7], double (*parr2)[5] ) { . . . }
```



동일한 선언



동일한 선언

```
void SimpleFunc( int parr1[][7], double parr2[][5] ) { . . . }
```

2차원 배열을 함수의 인자로 전달하는 예제

```
void ShowArr2DStyle(int (*arr)[4], int column)
{
    // 배열요소 전체출력
    int i, j;
    for(i=0; i<column; i++)
    {
        for(j=0; j<4; j++)
            printf("%d ", arr[i][j]);
        printf("\n");
    }
    printf("\n");
}
```

```
int Sum2DArr(int arr[][4], int column)
{
    // 배열요소의 합 반환
    int i, j, sum=0;
    for(i=0; i<column; i++)
        for(j=0; j<4; j++)
            sum += arr[i][j];
    return sum;
}
```

```
1 2 3 4
5 6 7 8
```

```
1 1 1 1
3 3 3 3
5 5 5 5
```

```
arr1의 합: 36
arr2의 합: 36
```

실행결과

정의된 두 함수의 인자로 전달되는 2차원 배열의 **가로길이는 결정되어 있다.**

반면, **세로 길이 정보는 결정되어 있지 않고** 두 번째 인자를 통해서 추가로 전달하고 있다. 이점에 주목하자!

```
int main(void)
{
    int arr1[2][4]={1, 2, 3, 4, 5, 6, 7, 8};
    int arr2[3][4]={1, 1, 1, 1, 3, 3, 3, 3, 5, 5, 5, 5};

    ShowArr2DStyle(arr1, sizeof(arr1)/sizeof(arr1[0]));
    ShowArr2DStyle(arr2, sizeof(arr2)/sizeof(arr2[0]));
    printf("arr1의 합: %d \n", Sum2DArr(arr1, sizeof(arr1)/sizeof(arr1[0])));
    printf("arr2의 합: %d \n", Sum2DArr(arr2, sizeof(arr2)/sizeof(arr2[0])));
    return 0;
}
```

배열의 세로길이 계산방식

2차원 배열에서도 arr[i]와 *(arr+i)는 같다.

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

Ch13에서 1차원 배열과 포인터 변수를 대상으로 내린 결론! 2차원 배열에서도 그대로 적용이 된다!

```
a[0]: 001AFDC8
*(a+0): 001AFDC8
a[1]: 001AFDD0
*(a+1): 001AFDD0
a[2]: 001AFDD8
*(a+2): 001AFDD8

6, 6
6, 6
6, 6
```

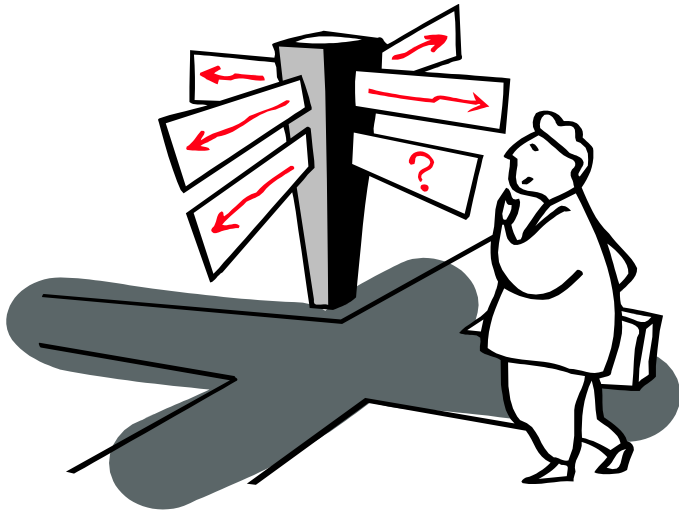
실행결과

```
int arr[3][2]={ {1, 2}, {3, 4}, {5, 6} };
```

```
arr[2][1]=4;
(*(arr+2))[1]=4;
*(arr[2]+1)=4;
*(*(arr+2)+1)=4;
```

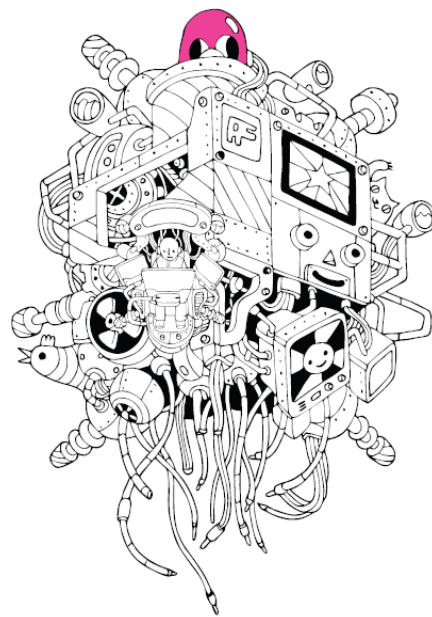
위에 선언된 배열 arr을 대상으로 인덱스 기준 [2][1] 번째 요소에 4를 저장하는, 모두 동일한 결과를 보이는 문장이다.

```
int main(void)
{
    int a[3][2]={ {1, 2}, {3, 4}, {5, 6} };
    printf("a[0]: %p \n", a[0]);
    printf("*(a+0): %p \n", *(a+0));
    printf("a[1]: %p \n", a[1]);
    printf("*(a+1): %p \n", *(a+1));
    printf("a[2]: %p \n", a[2]);
    printf("*(a+2): %p \n", *(a+2));
    printf("%d, %d \n", a[2][1], (*(a+2))[1]);
    printf("%d, %d \n", a[2][1], *(a[2]+1));
    printf("%d, %d \n", a[2][1], *(*a+2)+1));
    return 0;
}
```

Chapter 18이 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 19. 함수 포인터와 void 포인터

윤성우의 열혈 C 프로그래밍



Chapter 19-1. 함수 포인터와 void 포인터

윤성우 저 열혈강의 C 프로그래밍 개정판

함수 포인터의 이해

1. 함수 포인터

1. **함수의 이름**은 함수가 저장된 메모리 공간을 가리키는 포인터이다(함수 포인터).
2. 함수의 이름이 의미하는 주소 값은 **함수 포인터 변수**를 선언해서 저장할 수 있다.
3. 함수 포인터 변수를 선언하려면 함수 포인터의 형(type)을 알아야 한다.

2. 함수 포인터의 형(type)

1. 함수 포인터의 형 정보에는 **반환형**과 **매개변수 선언**에 대한 정보를 담기로 약속
2. 즉, 함수의 반환형과 매개변수 선언이 동일한 두 함수의 함수 포인터 형은 일치한다.

3. 함수 포인터 형 결정

`int SimpleFunc(int num)` 반환형 **int**, 매개변수 **int형 1개**

`double ComplexFunc(double num1, double num2)` 반환형 **double**, 매개변수 **double형 2개**

적절한 함수 포인터 변수의 선언

```
int (*fptr) (int)
```

fptr은 포인터!

```
int (*fptr) (int)
```

반환형이 int인 함수 포인터!

```
int (*fptr) (int)
```

매개변수 선언이 int 하나인 함수 포인터!

함수 포인터 변수를 선언하는 방법

```
int SoSimple(int num1, int num2) { . . . . }
```

```
int (*fptr) (int, int);
```

SoSimple 함수이름과 동일한 형의 변수 선언

```
fptr=SoSimple;
```

상수의 값을 변수에 저장

```
fptr(3, 4);
```

// SoSimple(3, 4)와 동일한 결과를 보임

함수 포인터 변수에 저장된 값을 통해서도 함수호출 가능!

함수 포인터 변수 관련 예제

```
void SimpleAdder(int n1, int n2)
{
    printf("%d + %d = %d \n", n1, n2, n1+n2);
}

void ShowString(char * str)
{
    printf("%s \n", str);
}

int main(void)
{
    char * str="Function Pointer";
    int num1=10, num2=20;

    void (*fptr1)(int, int) = SimpleAdder;
    void (*fptr2)(char *) = ShowString;

    /* 함수 포인터 변수에 의한 호출 */
    fptr1(num1, num2);
    fptr2(str);
    return 0;
}
```

10 + 20 = 30
Function Pointer

실행결과

교재에 있는 UsefulFunctionPointer.c를 통해서 함수 포인터 변수가 매개변수로 선언이 됨을 확인하기 바랍니다.

형(Type)이 존재하지 않는 void 포인터

```
void * ptr;
```

어떠한 주소 값도 저장이 가능한 void형 포인터

형 정보가 존재하지 않는 포인터 변수이기에 어떠한 주소 값도 저장이 가능하다.

형 정보가 존재하지 않기 때문에 메모리 접근을 위한 * 연산은 불가능하다.

```
void SoSimpleFunc(void)
{
    printf("I'm so simple");
}

int main(void)
{
    int num=20;
    void * ptr;

    ptr=&num;    // 변수 num의 주소 값 저장
    printf("%p \n", ptr);

    ptr=SoSimpleFunc;    // 함수 SoSimpleFunc의 주소 값 저장
    printf("%p \n", ptr);
    return 0;
}
```

```
int main(void)
{
    int num=20;
    void * ptr=&num;
    *ptr=20;    // 컴파일 에러!
    ptr++;    // 컴파일 에러!
    . . .
}
```

형 정보가 존재하지 않으므로!!

001AF974

00F61109

실행결과

윤성우의 열혈 C 프로그래밍



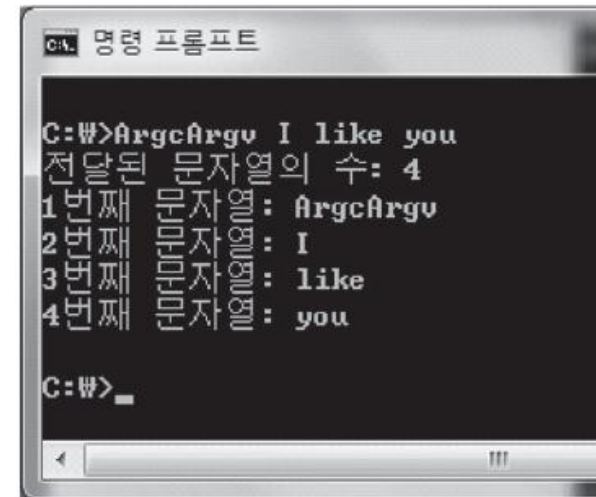
Chapter 19-2. main 함수로의 인자 전달

윤성우 저 열혈강의 C 프로그래밍 개정판

main 함수를 통한 인자의 전달

```
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;
}
```



```
C:\>ArgcArgv I like you
전달된 문자열의 수: 4
1번째 문자열: ArgcArgv
2번째 문자열: I
3번째 문자열: like
4번째 문자열: you
C:\>_
```

인자를 전달하는 방식

char * argv[]

```
void SimpleFunc(TYPE * arr) { . . . . }
void SimpleFunc(TYPE arr[]) { . . . . }
```

매개 변수 선언에서는 예외적으로 ***arr**을 **arr[]**으로 대신할 수 있다!
앞서 두 차례 확인한 내용!

 그대로 적용한다.

```
void SimpleFunc(char **arr) { . . . . }
void SimpleFunc(char * arr[]) { . . . . }
```

즉, char * arr[]는 char형 이중 포인터이다.



char * argv[] 관련 예제

```
void ShowAllString(int argc, char * argv[])
{
    int i;
    for(i=0; i<argc; i++)
        printf("%s \n", argv[i]);
}

int main(void)
{
    char * str[3]={
        "C Programming",
        "C++ Programming",
        "JAVA Programming"
    };
    ShowAllString(3, str);
    return 0;
}
```

문자열의 주소 값을 모은 배열이므로 char형 포인터 배열을 선언!
str의 포인터 형은 char**

```
C Programming
C++ Programming
JAVA Programming
```

실행결과



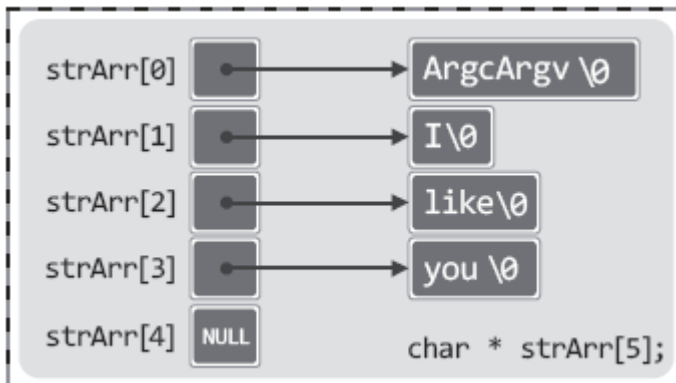
인자의 형성과정

c:\>ArgvArgv I like you

문자열의 구분

문자열 1	"ArgvArgv"
문자열 2	"I"
문자열 3	"like"
문자열 4	"you"

문자열의 구성



```

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

    while(argv[i]!=NULL)
    {
        printf("%d번째 문자열: %s \n", i+1, argv[i]);
        i++;
    }
    return 0;
}
    
```

C:\> ArgvEndNULL "I love you"

전달된 문자열의 수: 2

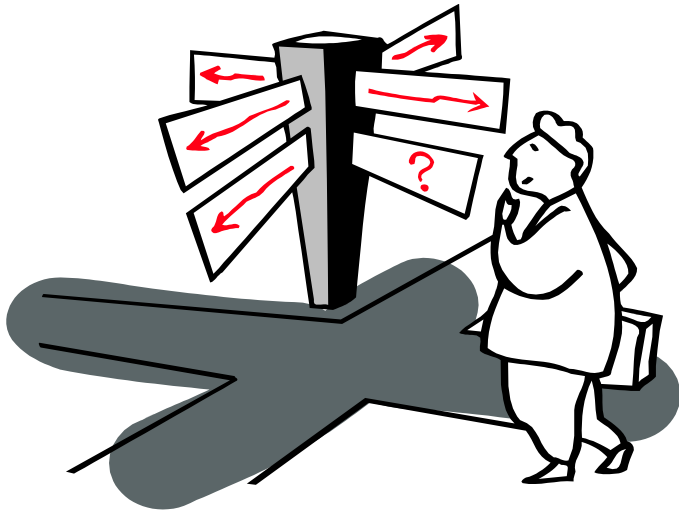
1번째 문자열: ArgvEndNULL

2번째 문자열: I love you

실행결과

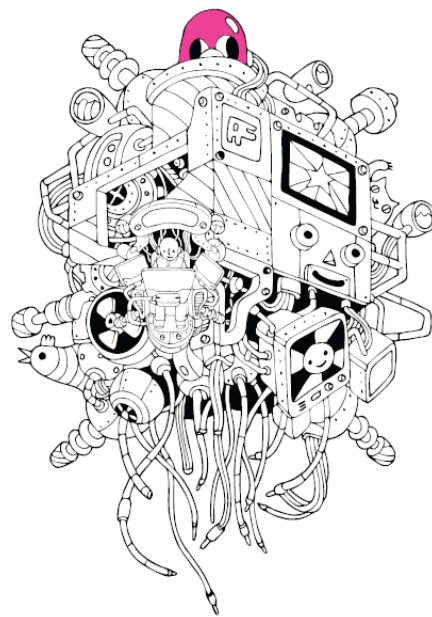
문자열 기반 함수의 호출

main(4, strArr);



Chapter 19가 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 21. 문자와 문자열 관련 함수

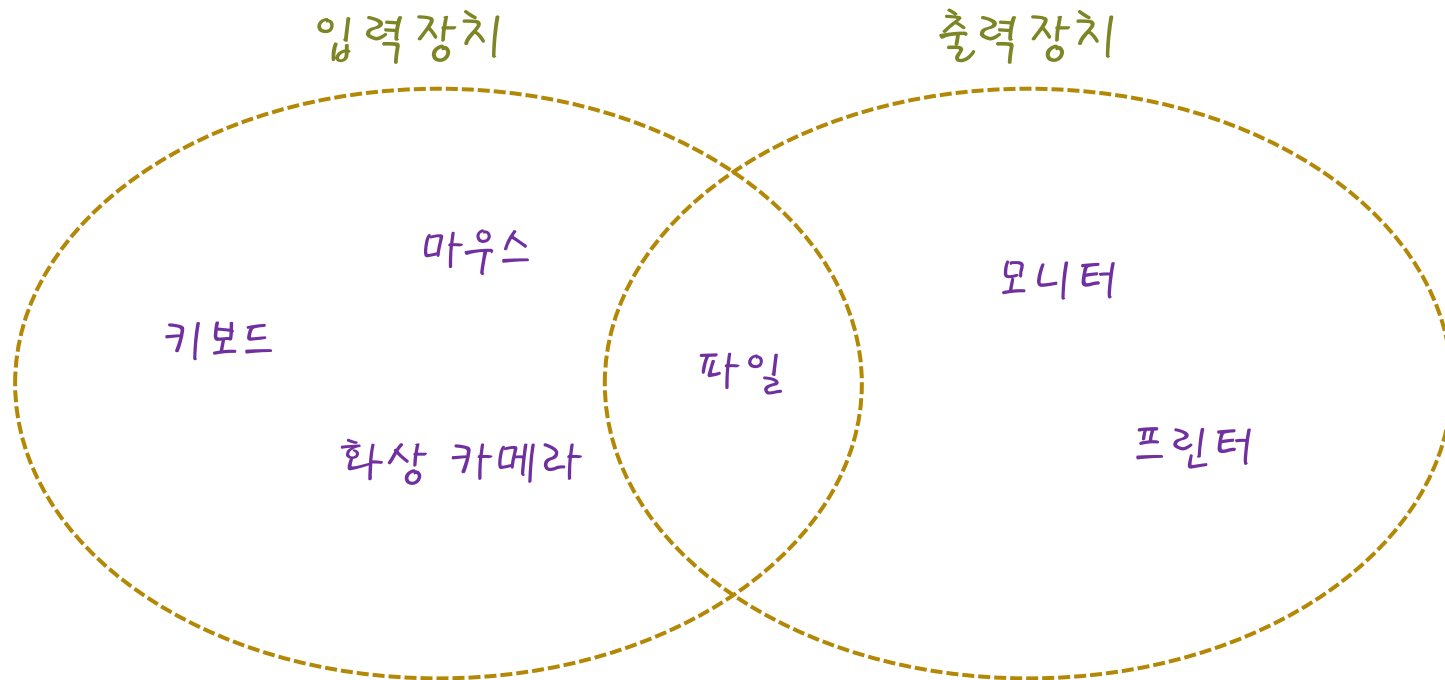
윤성우의 열혈 C 프로그래밍



Chapter 21-1. 스트림과 데이터의 이동

윤성우 저 열혈강의 C 프로그래밍 개정판

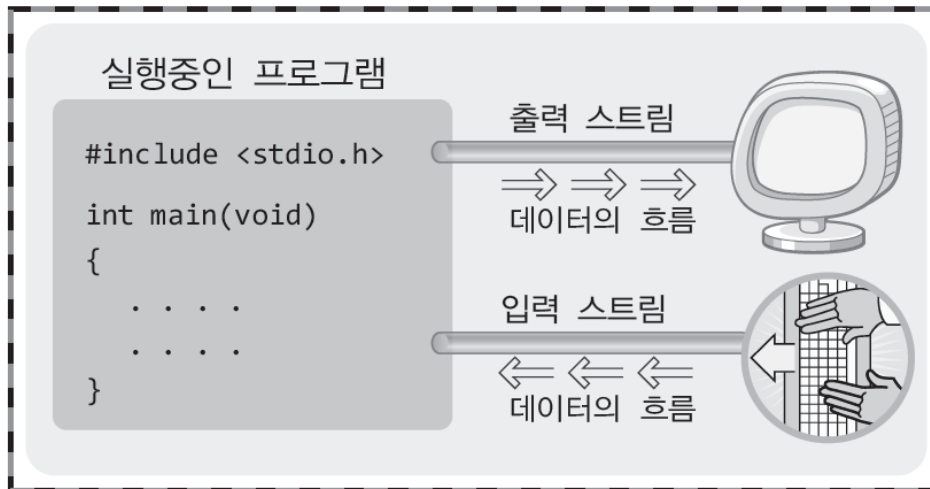
무엇이 입력이고 무엇이 출력인가



입출력 장치는 매우 포괄적이다.

데이터를 컴퓨터 내부로 받아들이는 것이 입력이고 외부로 전송하는 것이 출력이다.

데이터의 이동수단이 되는 스트림



콘솔 입출력을 위한 스트림은 프로그램이 시작되면 OS에 의해서 자동으로 생성된다.

➤ 데이터의 입 출력이 가능한 이유!

출력의 경로가 되는 출력 스트림과 입력의 경로가 되는 입력 스트림이 존재하기 때문

➤ 입출력 스트림이란?

OS가 데이터의 입출력을 위해 놓아주는 소프트웨어적인 형태의 다리!

스트림의 생성과 소멸

• stdin	표준 입력 스트림	키보드 대상으로 입력
• stdout	표준 출력 스트림	모니터 대상으로 출력
• stderr	표준 에러 스트림	모니터 대상으로 출력

- √ stdin과 stdout은 각각 **표준 입력 스트림**과 **표준 출력 스트림**을 의미하는 이름들이다.
- √ stderr은 **표준 에러 스트림**이라 하며, 출력의 대상은 stdout과 마찬가지로 모니터이다.
- √ 출력 리다이렉션이라는 것을 통해서 stdout과 stderr이 향하는 데이터 전송의 방향을 각각 달리 할 수 있다.
- √ stdin, stdout, stderr은 모두 프로그램 시작과 동시에 자동으로 형성되고 프로그램 종료시 자동으로 소멸된다.
- √ 이외의 스트림들은 프로그래머가 직접 형성해야 한다. 예를 들어 파일 입출력을 위한 스트림은 직접 형성해야 한다.

스트림이라 불리는 이유는 데이터의 이동을 한 방향으로만 형성하기 때문이다. 물이 한 방향으로 흐르듯 스트림도(스트림은 물의 흐름을 의미함) 한 방향으로만 데이터가 이동한다.



윤성우의 열혈 C 프로그래밍



Chapter 21-2. 문자 단위 입출력 함수

윤성우 저 열혈강의 C 프로그래밍 개정판

문자 입출력 함수

✓ 하나의 문자를 출력하는 두 함수

```
#include <stdio.h>
int putchar(int c); putchar 함수는 인자로 전달된 문자를 모니터에 출력한다.
int fputc(int c, FILE * stream); fputc 함수의 두 번째 인자를 통해서 출력의 대상을 지정한다.
```

➡ 함수호출 성공 시 쓰여진 문자정보가, 실패 시 EOF 반환

`fputc`의 두 번째 인자로 `stdout`이 전달되면 이 `putchar` 함수와 동일한 결과를 보인다.

✓ 하나의 문자를 입력 받는 두 함수

```
#include <stdio.h>
int getchar(void); getchar 함수는 키보드로 입력된 문자의 정보를 반환한다.
int fgetc(FILE * stream); fgetc 함수의 두 번째 인자를 통해서 입력의 대상을 지정한다.
```

➡ 파일의 끝에 도달하거나 함수호출 실패 시 EOF 반환

`getchar` 함수와 `fgetc` 함수의 관계는 `putchar` 함수와 `fputc` 함수의 관계와 같다.



문자 입출력 관련 예제

```
int main(void)
{
    int ch1, ch2;

    ch1=getchar(); // 문자 입력
    ch2=fgetc(stdin); // 엔터 키 입력

    putchar(ch1); // 문자 출력
    fputc(ch2, stdout); // 엔터 키 출력
    return 0;
}
```

문자의 입력을 완성하는 엔터 키의 입력도 하나의 문자로 인식이 된다.
따라서 이 역시도 입출력이 가능하다.

p
p

실행결과

첫 번째 P는 입력이 된 P, 두 번째 P는 출력된 P

문자를 *int*형 변수에 저장하는 이유는 EOF를 설명하면서 함께 설명한다.



문자 입출력에서의 EOF

√ EOF의 의미

- ▶ EOF는 End Of File의 약자로서, 파일의 끝을 표현하기 위해서 정의해 놓은 상수이다.
- ▶ 파일을 대상으로 fgetc 함수가 호출되었을 때 파일에 끝에 도달을 하면 EOF가 반환된다.

√ 콘솔 대상의 fgetc, getchar 함수호출로 EOF를 반환하는 경우

- ▶ 함수호출의 실패
- ▶ Windows에서 Ctrl+Z 키, Linux에서 Ctrl+D 키가 입력이 되는 경우

```
int main(void)
{
    int ch;
    while(1)
    {
        ch=getchar();
        if(ch==EOF)
            break;
        putchar(ch);
    }
    return 0;
}
```

키보드에는 EOF가 존재하지 않는다.

따라서 EOF를 Ctrl+Z 키와 Ctrl+D 키로 약속해 놓은 것이다.

예제에서 보듯이. 하나의 문장이 입력되어도

문장을 이루는 모든 문자들이 반복된 getchar 함수의 호출을 통해서 입력될 수 있다.

```
Hi~
Hi~
I like C lang.
I like C lang.
^Z
```

실행결과

반환형이 int이고, int형 변수에 문자를 담는 이유는?

```
int getchar(void);
int fgetc(FILE * stream);
```

✓ 반환형이 char형이 아닌 int형인 이유는?

- ▶ char형은 예외적으로 signed char가 아닌 unsigned char로 표현하는 컴파일러가 존재한다.
- ▶ 파일의 끝에 도달했을 때 반환하는 EOF는 -1로 정의되어 있다.
- ▶ char를 unsigned char로 표현하는 컴파일러는 EOF에 해당하는 -1을 반환하지 못한다.
- ▶ int는 모든 컴파일러가 signed int로 처리한다. 따라서 -1의 반환에 무리가 없다.



윤성우의 열혈 C 프로그래밍



Chapter 21-3. 문자열 단위 입출력 함수

윤성우 저 열혈강의 C 프로그래밍 개정판

문자열 출력 함수: puts, fputs

```
#include <stdio.h>
int puts(const char * s);
int fputs(const char * s, FILE * stream);
```

➔ 성공 시 0이 아닌 값을, 실패 시 EOF 반환

인자로 전달되는 문자열을 출력한다. 단 fputs 함수는 두 번째 인자를 통해서 출력의 대상을 지정할 수 있다.

```
int main(void)
{
    char * str="Simple String";
    printf("1. puts test ----- \n");
    puts(str);
    puts("So Simple String");
    printf("2. fputs test ----- \n");
    fputs(str, stdout); printf("\n");
    fputs("So Simple String", stdout); printf("\n");
    printf("3. end of main ----\n");
    return 0;
}
```

puts 함수가 호출되면 문자열 출력 후 자동으로 개행이 이뤄지지만, fputs 함수가 호출되면 문자열 출력 후 자동으로 개행이 이뤄지지 않는다는 사실에 주목!

```
1. puts test -----
Simple String
So Simple String
2. fputs test -----
Simple String
So Simple String
3. end of main ----
```

실행결과

문자열 입력 함수: gets, fgets

```
#include <stdio.h>
char * gets(char * s);
char * fgets(char * s, int n, FILE * stream);
```

➔ 파일의 끝에 도달하거나 함수호출 실패 시 NULL 포인터 반환

```
int main(void)
{
    char str[7]; // 7바이트의 메모리 공간 할당
    gets(str); // 입력 받은 문자열을 배열 str에 저장
    . . . .
}
```

이 경우 입력되는 문자열의 길이가 배열을 넘어설 경우 할당 받지 않은 메모리를 참조하는 오류가 발생한다.

```
int main(void)
{
    char str[7];
    fgets(str, sizeof(str), stdin);
    . . . // stdin으로부터 문자열 입력 받아서 str에 저장
}
```

stdin으로부터 문자열을 입력 받아서 **str**에 저장을 하되, 널 문자를 포함하여 **sizeof(str)**의 크기만큼 저장을 해라.



fgets 함수의 호출의 예

```
int main(void)
{
    char str[7];
    int i;

    for(i=0; i<3; i++)
    {
        fgets(str, sizeof(str), stdin);
        printf("Read %d: %s \n", i+1, str);
    }
    return 0;
}
```

12345678901234567890

실행결과1

Read 1: 123456

Read 2: 789012

Read 3: 345678

6개의 문자씩 끊어서 읽고 있다.

즉, 한번의 fgets 함수호출당 최대 6개의 문자만 읽혀진다.

We 엔터

Read 1: We

실행결과2

엔터 키의 입력도 문자열의 일부로

like 엔터

Read 2: like

받아 들임을 보임

you 엔터

Read 3: you

Y & I 엔터

Read 1: Y & I

실행결과3

공백을 포함하는 문자열을 읽어 들임

ha ha 엔터

Read 2: ha ha

을 보임

^^ -- 엔터

Read 3: ^^ --

윤성우의 열혈 C 프로그래밍



Chapter 21-4. 표준 입출력 버퍼

윤성우 저 열혈강의 C 프로그래밍 개정판

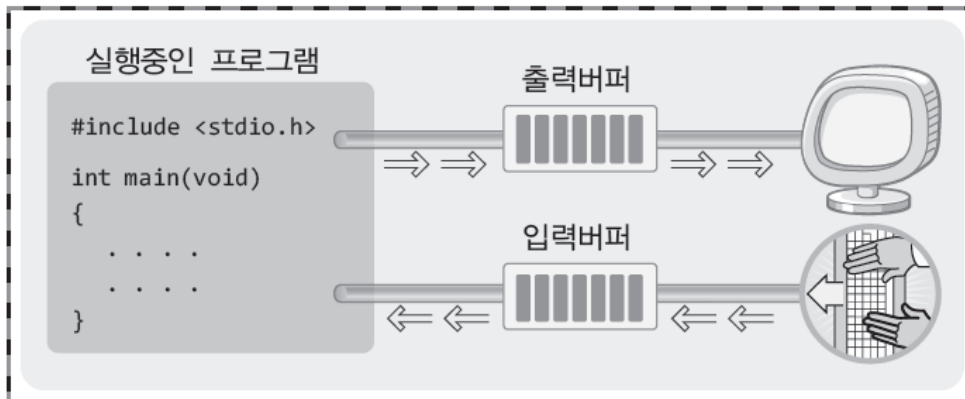
표준 입출력 기반의 버퍼와 버퍼링의 이유

✓ 입출력 버퍼

- ▶ 버퍼는 특정 크기의 메모리 공간을 의미한다.
- ▶ 운영체제는 입력과 출력을 돕는 입출력 버퍼를 생성하여 제공한다.
- ▶ 표준 입출력 함수를 기반으로 데이터 입출력 시 입출력 버퍼를 거친다.

✓ 입출력 버퍼에 데이터가 전송되는 시점

- ▶ 호출된 출력함수가 반환이 되는 시점이 출력버퍼로 데이터가 완전히 전송된 시점이다.
- ▶ 엔터를 입력하는 시점이 키보드로 입력된 데이터가 입력버퍼로 전달되는 시점이다.



버퍼링을 하는 이유는 데이터 이동의 효율과 관련이 있다. 데이터를 모아서 전송하면, 하나씩 전송하는 것보다 효율적이다.



출력버퍼를 비우는 fflush 함수

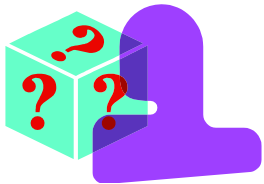
```
#include <stdio.h>
int fflush(FILE * stream);
```

➔ 함수호출 성공 시 0, 실패 시 EOF 반환

▶ 인자에 해당하는 출력버퍼를 비운다.

출력버퍼를 비운다는 것은 출력버퍼에 저장된 데이터를 지우는 것이 아니라,
출력버퍼에 저장된 데이터를 목적지로 최종 전송함을 뜻한다.

▶ fflush(stdout) → 출력버퍼를 지워라!



출력버퍼의 경우와 달리 **입력버퍼의 비움은 입력버퍼에 저장된 데이터의 소멸**을 뜻한다.
그리고 fflush 함수는 **출력버퍼를 대상으로 정의된 함수**이다. 따라서 fflush(stdin) 과 같은 형태의 함수호출은 그 결과를 보장받지 못한다.

그렇다면 입력버퍼는 어떻게 비워야 할까?

입력버퍼는 어떻게 비워야 하나요?

```
int main(void)
{
    char perID[7];
    char name[10];

    fputs("주민번호 앞 6자리 입력: ", stdout);
    fgets(perID, sizeof(perID), stdin);

    fputs("이름 입력: ", stdout);
    fgets(name, sizeof(name), stdin);

    printf("주민번호: %s \n", perID);
    printf("이름: %s \n", name);
    return 0;
}
```

주민번호 앞 6자리만 입력 받기 위해서 배열의

길이가 널 문자 포함 7이다.

입력버퍼를 비우는 함수

```
void ClearLineFromReadBuffer(void)
{
    while(getchar()!='\n');
}
```

```
int main(void)
{
    char perID[7];
    char name[10];

    fputs("주민번호 앞 6자리 입력: ", stdout);
    fgets(perID, sizeof(perID), stdin);
    ClearLineFromReadBuffer(); // 입력버퍼 비우기

    fputs("이름 입력: ", stdout);
    fgets(name, sizeof(name), stdin);

    printf("주민번호: %s\n", perID);
    printf("이름: %s\n", name);
    return 0;
}
```

주민번호 앞 6자리 입력: 950915 실행결과1

이름 입력: 주민번호: 950915

이름:

엔터 키가 남아서 문제가 되는 상황

주민번호 앞 6자리 입력: 950709-1122345 실행결과2

이름 입력: 주민번호: 950709

이름: -1122345

말 안 듣는 사람들 때문에 문제되는 상황

어떠한 경우에도 주민번호 6자리만 입력 받도록
재 구현된 예제

윤성우의 열혈 C 프로그래밍



Chapter 21-5. 입출력 이외의 문자열
관련 함수

윤성우 저 열혈강의 C 프로그래밍 개정판

문자열의 길이를 반환하는 함수: strlen

```
#include <string.h>
size_t strlen(const char * s);
```

➡ 전달된 문자열의 길이를 반환하되, 널 문자는 길이에 포함하지 않는다.

size_t의 일반적인 선언

typedef unsigned int size_t;

typedef에 관해서는 후에 설명

```
int main(void)
{
    char str[]="1234567";
    printf("%u \n", strlen(str));
    . . .      // 문자열의 길이 7이 출력
}
```

실행결과

```
문자열 입력: Good morning
길이: 13, 내용: Good morning

길이: 12, 내용: Good morning
```

```
void RemoveBSN(char str[])
{
    int len=strlen(str);
    str[len-1]=0;
}
```

마지막에 삽입되는

널 문자를 없애는 예제

```
int main(void)
{
    char str[100];
    printf("문자열 입력: ");
    fgets(str, sizeof(str), stdin);
    printf("길이: %d, 내용: %s \n", strlen(str), str);

    RemoveBSN(str);
    printf("길이: %d, 내용: %s \n", strlen(str), str);
    return 0;
}
```

문자열을 복사하는 함수들: strcpy, strncpy

```
#include <string.h>
char * strcpy(char * dest, const char * src);
char * strncpy(char * dest, const char * src, size_t n);
```

➔ 복사된 문자열의 주소 값 반환

대표적인 문자열 복사 함수

```
int main(void)
{
    char str1[30]="Simple String";
    char str2[30];
    strcpy(str2, str1);
    . . . // str1의 문자열을 str2에 복사
}
```

str1에 저장된 문자열을 str2에 단순히 복사!

strcpy 함수를 호출하는 경우 배열의 범위를 넘어서 복사가 진행될 위험이 있다.

```
int main(void)
{
    char str1[30]="Simple String";
    char str2[30];
    strncpy(str2, str1, sizeof(str2));
    . . .
}
```

str1에 저장된 문자열을 str2에 복사하되 최대 sizeof(str2)의 반환 값 크기만큼 복사한다.

strncpy 함수를 잘못 사용한 예

```
int main(void)
{
    char str1[20]="1234567890";
    char str2[20];
    char str3[5];

    /*** case 1 ***/
    strcpy(str2, str1);
    puts(str2);

    /*** case 2 ***/
    strncpy(str3, str1, sizeof(str3));
    puts(str3);

    /*** case 3 ***/
    strncpy(str3, str1, sizeof(str3)-1);
    str3[sizeof(str3)-1]=0;
    puts(str3);
    return 0;
}
```

배열 길이 *str1*에 딱 맞는 길이만큼만 복

사를 하겠다는 의도의 문장

실행결과

1234567890

12345?234567890

1234

두 번째 *strncpy* 함수호출 후의 결과에 이상이 보이는 이유는 복사하는 과정에서 문자열의 끝을 의미하는 널 문자가 복사되지 않았기 때문이다. 문자열을 복사할 때에는 항상 널 문자의 복사까지 고려해야 한다.

문자열을 덧붙이는 함수들: strcat, strncat

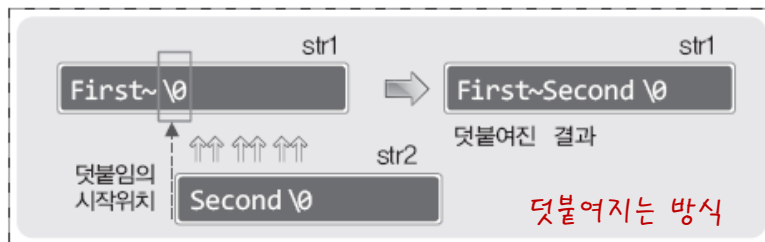
```
#include <string.h>
char * strcat(char * dest, const char * src);
char * strncat(char * dest, const char * src, size_t n);
```

➔ 덧붙여진 문자열의 주소 값 반환

strncat 함수는 덧붙일 문자열의
최대 길이를 제한한다.

최대 n 개의 문자를 덧붙이되 n 문자 포함하여 $n+1$ 개의 문자를 덧붙인다.

```
int main(void)
{
    char str1[30]="First~";
    char str2[30]="Second";
    strcat(str1, str2);
    . . . . // str1의 문자열 뒤에 str2를 복사
}
```



```
int main(void)
{
    char str1[20]="First~";
    char str2[20]="Second";
    char str3[20]="Simple num: ";
    char str4[20]="1234567890";

    /**** case 1 ****/
    strcat(str1, str2);
    puts(str1);

    /**** case 2 ****/
    strncat(str3, str4, 7);
    puts(str3);
    return 0;
}
```

실행결과

```
First~Second
Simple num: 1234567
```

문자열을 비교하는 함수들: strcmp, strncmp

```
#include <string.h>
int strcmp(const char * s1, const char * s2);
int strncmp(const char * s1, const char * s2, size_t n);
```

→ 두 문자열의 내용이 같으면 0, 같지 않으면 0이 아닌 값 반환

- s1이 더 크면 0보다 큰 값 반환
- s2가 더 크면 0보다 작은 값 반환
- s1과 s2의 내용이 모두 같으면 0 반환

strncmp는 최대 n개의 문자를 비교

- ▶ 크고 작음은 **아스키코드 값**을 근거로 한다.
- ▶ A보다 B가, B보다 C가 아스키 코드 값이 더 크고 A보다 a가, B보다 b가 아스키 코드 값이 더 크니, **사전편찬순서를 기준으로 뒤에 위치할 수록 더 큰 문자열로 인식해도 된다.**

printf("%d", strcmp("ABCD", "ABCC")); *0보다 큰 값이 출력*

printf("%d", strcmp("ABCD", "ABCDE")); *0보다 작은 값이 출력*

두 문자열이 같으면 0, 다르면 0이 아닌 값을 반환한다고 인식하고 있어도 충분하다!



문자열 비교의 예

```
int main(void)
{
    char str1[20];
    char str2[20];
    printf("문자열 입력 1: ");
    scanf("%s", str1);
    printf("문자열 입력 2: ");
    scanf("%s", str2);

    if(!strcmp(str1, str2))
    {
        puts("두 문자열은 완벽히 동일합니다.");
    }
    else
    {
        puts("두 문자열은 동일하지 않습니다.");

        if(!strncmp(str1, str2, 3))
            puts("그러나 앞 세 글자는 동일합니다.");
    }
    return 0;
}
```

실행결과

문자열 입력 1: Simple
문자열 입력 2: Simon
두 문자열은 동일하지 않습니다.
그러나 앞 세 글자는 동일합니다.

그 이외의 변환함수들

<code>int atoi(const char * str);</code>	문자열의 내용을 int형으로 변환
<code>long atol(const char * str);</code>	문자열의 내용을 long형으로 변환
<code>double atof(const char * str);</code>	문자열의 내용을 double형으로 변환

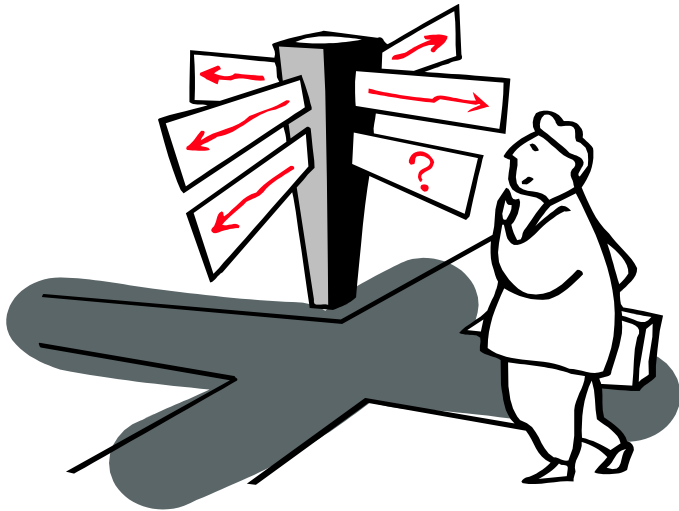
헤더파일 `stdlib.h`에 선언

```
int main(void)
{
    char str[20];
    printf("정수 입력: ");
    scanf("%s", str);
    printf("%d \n", atoi(str));
    printf("실수 입력: ");
    scanf("%s", str);
    printf("%g \n", atof(str));
    return 0;
}
```

위의 함수들을 모른다면 문자열에 저장된 숫자 정보를 int형 또는 double형으로 변환하는 일은 번거로운 일이 될 수 있다.

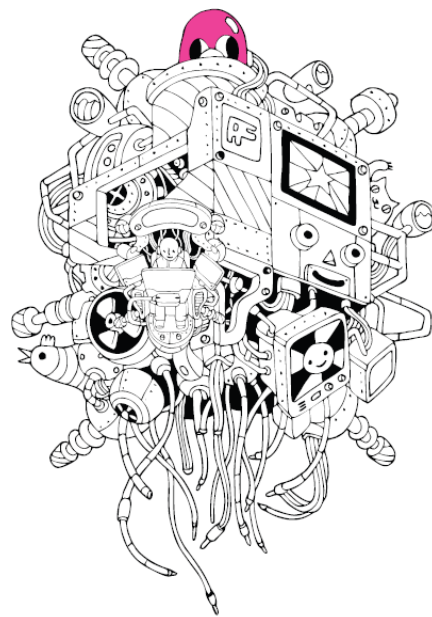
실행결과

```
정수 입력: 15
15
실수 입력: 12.456
12.456
```



Chapter 21이 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 22. 구조체와 사용자 정의 자료형1

윤성우의 열혈 C 프로그래밍



Chapter 22-1. 구조체란 무엇인가?

윤성우 저 열혈강의 C 프로그래밍 개정판

구조체의 정의

```
int xpos; // 마우스의 x 좌표
int ypos; // 마우스의 y 좌표
```

마우스의 좌표정보를 저장하고 관리하기 위해서는
x좌표와 y좌표를 저장할 수 있는 두 개의 변수가 필요하다.

xpos와 ypos는 서로 독립된 정보를 표현하지 않고 하나의 정보를 표현한다. 따라서 이 둘은 늘 함께한다.

```
struct point // point라는 이름의 구조체 정의
{
    int xpos; // point 구조체를 구성하는 멤버 xpos
    int ypos; // point 구조체를 구성하는 멤버 ypos
};
```

구조체를 이용해서 xpos와 ypos를 하나로 묶었다.
이 둘을 묶어서 point라는 이름의 새로운 자료형을 정의!

int가 자료형의 이름인것 처럼 point도 자료형의 이름이다.

단, 프로그래머가 정의한 자료형이기에 '사용자 정의 자료형(user defined data type)'이라 한다.

```
struct person
{
    char name[20]; // 이름 저장
    char phoneNum[20]; // 전화번호 저장
    int age; // 나이 저장
};
```

개인의 이름과 전화번호 나이 정보를
person이라는 구조체 정의를 통해서 묶고 있다.

배열도 구조체의 멤버로 선언이 가능!

구조체 변수의 선언과 접근

구조체 변수선언의 기본 형태

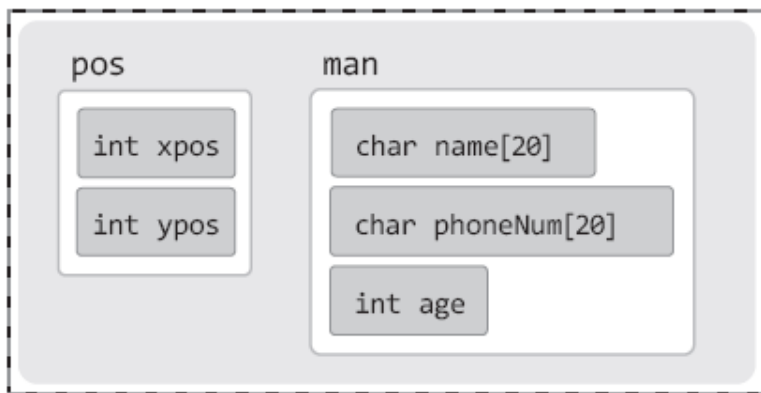
```
struct type_name val_name ;
```



```
struct point pos;  
struct person man;
```



구조체 변수선언의 예



구조체 변수선언의 결과

멤버의 접근방식

구조체 변수의 이름 . 구조체 멤버의 이름



```
pos.xpos=20;
```

구조체 변수 *pos*의 멤버 *xpos*에 20을 저장

```
printf("%s \n", man.name);
```

*man*의 멤버 *name*에 저장된 문자열 출력

구조체 변수의 선언과 접근관련 예제1

```
struct point    // 구조체 point의 정의
{
    int xpos;
    int ypos;
};

int main(void)
{
    struct point pos1, pos2;
    double distance;
    fputs("point1 pos: ", stdout);
    scanf("%d %d", &pos1.xpos, &pos1.ypos);

    fputs("point2 pos: ", stdout);
    scanf("%d %d", &pos2.xpos, &pos2.ypos);

    /* 두 점간의 거리 계산 공식 */
    distance=sqrt((double)((pos1.xpos-pos2.xpos) * (pos1.xpos-pos2.xpos)+
        (pos1.ypos-pos2.ypos) * (pos1.ypos-pos2.ypos)));

    printf("두 점의 거리는 %g 입니다. \n", distance);
    return 0;
}
```

이 예제에서 호출하는 함수 *sqrt*는 제곱근을 반환하는 함수
로써 헤더파일 *math.h*에 선언된 수학관련 함수이다.

실행결과

```
point1 pos: 1 3
point2 pos: 4 5
두 점의 거리는 3.60555 입니다.
```

구조체 변수의 선언과 접근관련 예제2

```
struct person
{
    char name[20];
    char phoneNum[20];
    int age;
};

int main(void)
{
    struct person man1, man2;
    strcpy(man1.name, "안성준");
    strcpy(man1.phoneNum, "010-1122-3344");
    man1.age=23;

    printf("이름 입력: "); scanf("%s", man2.name);
    printf("번호 입력: "); scanf("%s", man2.phoneNum);
    printf("나이 입력: "); scanf("%d", &(man2.age));

    printf("이름: %s \n", man1.name);
    printf("번호: %s \n", man1.phoneNum);
    printf("나이: %d \n", man1.age);

    printf("이름: %s \n", man2.name);
    printf("번호: %s \n", man2.phoneNum);
    printf("나이: %d \n", man2.age);

    return 0;
}
```

구조체의 멤버라 하더라도 일반적인 접근의 방식을 그대로 따른다. 구조체의 멤버로 배열이 선언되면 배열의 접근방식을 취하면 되고, 구조체의 멤버로 포인터 변수가 선언되면 포인터 변수의 접근방식을 취하면 된다.

```
이름 입력: 김수정
번호 입력: 010-0001-0002
나이 입력: 27
이름: 안성준
번호: 010-1122-3344
나이: 23
이름: 김수정
번호: 010-0001-0002
나이: 27
```

실행결과

구조체 정의와 동시에 변수 선언하기

```
struct point
{
    int xpos;
    int ypos;
} pos1, pos2, pos3;
```

point라는 이름의 구조체를 정의함과 동시에
point 구조체의 변수 pos1, pos2, pos3를 선언하는 문장이다.

```
struct point
{
    int xpos;
    int ypos;
};
struct point pos1, pos2, pos3;
```

위와 동일한 결과를 보이는 구조체의 정의와 변수의 선언이다.

구조체를 정의함과 동시에 변수를 선언하는 문장은 잘 사용되지 않는다.

그러나 문법적으로 지원이 되고 또 간혹 사용하는 경우도 있다.



구조체 변수의 초기화

```
struct point
{
    int xpos;
    int ypos;
};

struct person
{
    char name[20];
    char phoneNum[20];
    int age;
};

int main(void)
{
    struct point pos={10, 20};
    struct person man={"이승기", "010-1212-0001", 21};
    printf("%d %d \n", pos.xpos, pos.ypos);
    printf("%s %s %d \n", man.name, man.phoneNum, man.age);
    return 0;
}
```

초기화 방식이 배열과 유사하다.

초기화 할 데이터들을 중괄호 안에 순서대로 나열하면 된다..

실행결과

10 20

이승기 010-1212-0001 21

윤성우의 열혈 C 프로그래밍



Chapter 22-2. 구조체와 배열
그리고 포인터

윤성우 저 열혈강의 C 프로그래밍 개정판

구조체 배열의 선언과 접근

```
struct point
{
    int xpos;
    int ypos;
};

int main(void)
{
    struct point arr[3];
    int i;
    for(i=0; i<3; i++)
    {
        printf("점의 좌표 입력: ");
        scanf("%d %d", &arr[i].xpos, &arr[i].ypos);
    }

    for(i=0; i<3; i++)
        printf("[%d, %d] ", arr[i].xpos, arr[i].ypos);

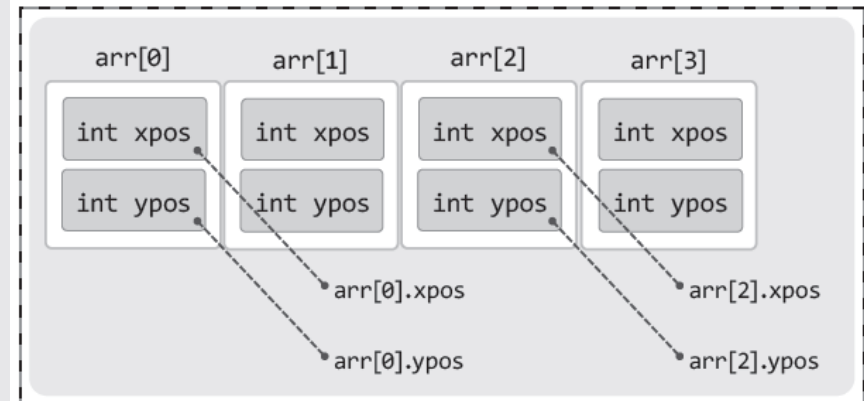
    return 0;
}
```

struct point arr[4];

길이가 4인 구조체 배열의 선언방법



선언된 배열의 형태



실행결과

```
점의 좌표 입력: 2 4
점의 좌표 입력: 3 6
점의 좌표 입력: 8 9
[2, 4] [3, 6] [8, 9]
```

구조체 배열의 초기화

```
struct person man={"이승기", "010-1212-0001", 21};
```

구조체 변수의 초기화

구조체 변수 하나를 초기화하기 위해서 하나의 중괄호를 사용하듯이...

```
struct person arr[3]={
    {"이승기", "010-1212-0001", 21},    // 첫 번째 요소의 초기화
    {"정지영", "010-1313-0002", 22},    // 두 번째 요소의 초기화
    {"한지수", "010-1717-0003", 19}     // 세 번째 요소의 초기화
};
```

구조체 배열의 초기화

구조체 배열을 초기화하기 위해서 배열요소 각각의 초기화 값을 중괄호로 묶어서 표현한다.

구조체 배열의 초기화 예제

```
struct person
{
    char name[20];
    char phoneNum[20];
    int age;
};

int main(void)
{
    struct person arr[3]={
        {"이승기", "010-1212-0001", 21},    // 첫 번째 요소의 초기화
        {"정지영", "010-1313-0002", 22},    // 두 번째 요소의 초기화
        {"한지수", "010-1717-0003", 19}     // 세 번째 요소의 초기화
    };

    int i;
    for(i=0; i<3; i++)
        printf("%s %s %d \n", arr[i].name, arr[i].phoneNum, arr[i].age);

    return 0;
}
```

실행결과

```
이승기 010-1212-0001 21
정지영 010-1313-0002 22
한지수 010-1717-0003 19
```

구조체 변수와 포인터

```
struct point pos={11, 12};
```

```
struct point * pptr=&pos;
```

구조체 *point*의 포인터 변수 선언

```
(*pptr).xpos=10;
```

*pptr*이 가리키는 구조체 변수의 멤버 *xpos*에 접근

```
(*pptr).ypos=20;
```

*pptr*이 가리키는 구조체 변수의 멤버 *ypos*에 접근

구조체 포인터 변수를 대상으로 하는
포인터 연산 및 멤버의 접근방법

```
(*pptr).xpos=10; ↔ pptr->xpos=10;
```

```
(*pptr).ypos=20; ↔ pptr->ypos=20;
```

-> 연산자를 기반으로 하는 구조체 변수
의 멤버 접근 방법

구조체 변수와 포인터 관련 예제

```
struct point
{
    int xpos;
    int ypos;
};

int main(void)
{
    struct point pos1={1, 2};
    struct point pos2={100, 200};
    struct point * pptr=&pos1;

    (*pptr).xpos += 4;
    (*pptr).ypos += 5;
    printf("[%d, %d] \n", pptr->xpos, pptr->ypos);

    pptr=&pos2;
    pptr->xpos += 1;
    pptr->ypos += 2;
    printf("[%d, %d] \n", (*pptr).xpos, (*pptr).ypos);
    return 0;
}
```

프로그래머들이 주로 사용하는 연산자이니
-> 연산자의 사용에 익숙해지자.

실행결과

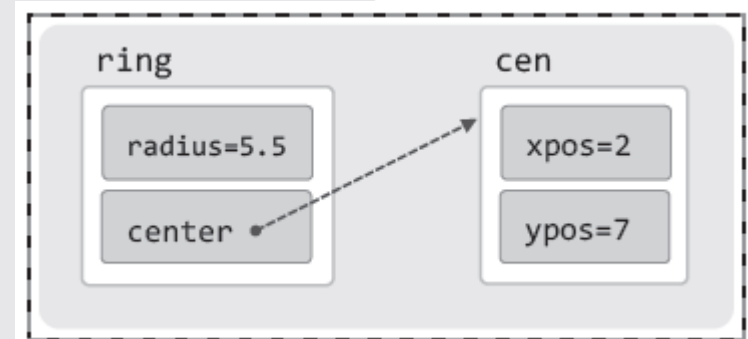
```
[5, 7]
[101, 202]
```

포인터 변수를 구조체의 멤버로 선언하기1

```
struct point
{
    int xpos;
    int ypos;
};
struct circle
{
    double radius;
    struct point * center;
};
```

구조체 변수의 멤버로 구조체 포인터 변수가 선언될 수 있다!

```
int main(void)
{
    struct point cen={2, 7};
    double rad=5.5;
    struct circle ring={rad, &cen};
    printf("원의 반지름: %g \n", ring.radius);
    printf("원의 중심 [%d, %d] \n", (ring.center)->xpos, (ring.center)->ypos);
    return 0;
}
```



실행결과

원의 반지름: 5.5
원의 중심 [2, 7]

포인터 변수를 구조체의 멤버로 선언하기2

```
struct point
{
    int xpos;
    int ypos;
    struct point * ptr;
};
```

type형 구조체 변수의 멤버로 type형 포인터 변수를 둘 수 있다.

```
int main(void)
{
    struct point pos1={1, 1};
    struct point pos2={2, 2};
    struct point pos3={3, 3};

    pos1.ptr = &pos2;    // pos1과 pos2를 연결
    pos2.ptr = &pos3;    // pos2와 pos3를 연결
    pos3.ptr = &pos1;    // pos3를 pos1과 연결

    printf("점의 연결관계... \n");
    printf("[%d, %d]와(과) [%d, %d] 연결 \n",
        pos1.xpos, pos1.ypos, pos1.ptr->xpos, pos1.ptr->ypos);
    printf("[%d, %d]와(과) [%d, %d] 연결 \n",
        pos2.xpos, pos2.ypos, pos2.ptr->xpos, pos2.ptr->ypos);
    printf("[%d, %d]와(과) [%d, %d] 연결 \n",
        pos3.xpos, pos3.ypos, pos3.ptr->xpos, pos3.ptr->ypos);
    return 0;
}
```

실행결과

점의 연결관계...

```
[1, 1]와(과) [2, 2] 연결
[2, 2]와(과) [3, 3] 연결
[3, 3]와(과) [1, 1] 연결
```


구조체 변수와 첫 번째 멤버의 주소 값

```
struct point
{
    int xpos;
    int ypos;
};

struct person
{
    char name[20];
    char phoneNum[20];
    int age;
};

int main(void)
{
    struct point pos={10, 20};
    struct person man={"이승기", "010-1212-0001", 21};

    printf("%p %p \n", &pos, &pos.xpos);
    printf("%p %p \n", &man, man.name);
    return 0;
}
```

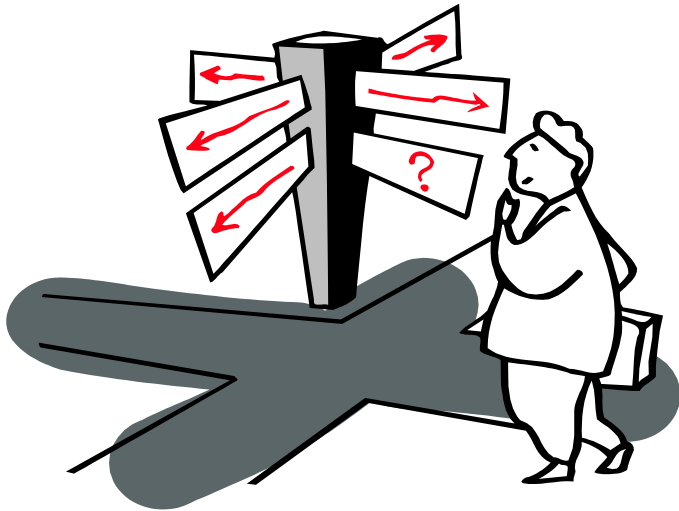
구조체 변수의 주소 값과 구조체 변수의 첫 번째 멤버의 주소 값은 일치한다.

응용 프로그램 분야에서는 이 사실을 이용해서 프로그램을 작성하기도 한다.

실행결과

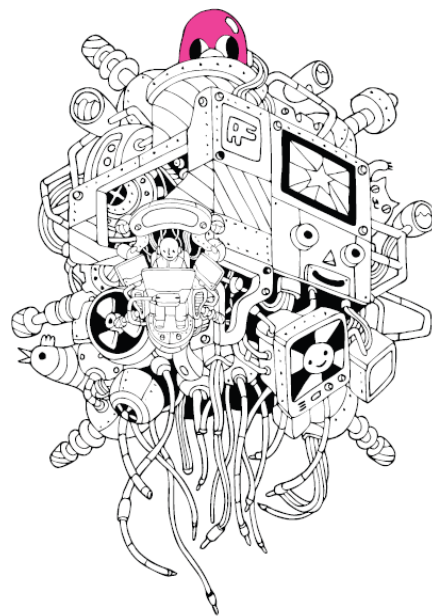
003EF7B8 003EF7B8

003EF784 003EF784



Chapter 22가 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 23. 구조체와 사용자 정의 자료형2

윤성우의 열혈 C 프로그래밍



Chapter 23-1. 구조체의 정의와
typedef 선언

윤성우 저 열혈강의 C 프로그래밍 개정판

typedef 선언

```
typedef int INT;
```

자료형의 이름 *int*에 *INT*라는 이름을 추가로 붙여줍니다.



위의 *typedef* 선언으로 인해서!!!

INT num; *int num;* 과 동일한 선언

*INT * ptr;* *int * ptr;* 과 동일한 선언

새로 부여된 이름	대상 자료형
INT	int
PTR_INT	int *
UINT	unsigned int
PTR_UINT	unsigned int *
UCHAR	unsigned char
PTR_UCHAR	unsigned char *



정의되는 이름들

실행결과

120, 190, Z

```
typedef int INT;
typedef int * PTR_INT;
typedef unsigned int UINT;
typedef unsigned int * PTR_UINT;
typedef unsigned char UCHAR;
typedef unsigned char * PTR_UCHAR;

int main(void)
{
    INT num1 = 120;           // int num1 = 120;
    PTR_INT pnum1 = &num1;    // int * pnum1 = &num1;
    UINT num2 = 190;          // unsigned int num2 = 190;
    PTR_UINT pnum2 = &num2;    // unsigned int * pnum2 = &num2;
    UCHAR ch = 'Z';           // unsigned char ch = 'Z';
    PTR_UCHAR pch = &ch;       // unsigned char * pch = &ch;
    printf("%d, %u, %c \n", *pnum1, *pnum2, *pch);
    return 0;
}
```

구조체 정의와 typedef 선언

```
struct point
{
    int xpos;
    int ypos;
};

typedef struct point Point;
```

구조체 *point* 정의 후

*struct point*에 *Point*라는 이름을 부여하기 위한 *typedef* 선언 추가!



합친 형태

```
typedef struct point
{
    int xpos;
    int ypos;
} Point;
```

구조체 *point*의 정의와

*Point*에 대한 *typedef* 선언을 한데 묶은 형태

구조체 정의와 typedef 선언 관련 예제

```
struct point
{
    int xpos;
    int ypos;
};
```

typedef struct point Point; 구조체 point의 정의와 typedef 선언

```
typedef struct person
{
    char name[20];
    char phoneNum[20];
    int age;
} Person;
```

구조체 person의 정의와

Person이라는 이름의 typedef 선언을 하나로!

```
int main(void)
{
    Point pos={10, 20};
    Person man={"이승기", "010-1212-0001", 21};
    printf("%d %d \n", pos.xpos, pos.ypos);
    printf("%s %s %d \n", man.name, man.phoneNum, man.age);
    return 0;
}
```

실행결과

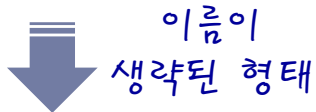
10 20

이승기 010-1212-0001 21

구조체의 이름 생략

```
typedef struct person
{
    char name[20];
    char phoneNum[20];
    int age;
} Person;
```

typedef 선언으로 인해서 새로운 이름 *Person*이 정의되었으니,
구조체의 이름 *persons*은 큰 의미가 없다.



```
typedef struct 
{
    char name[20];
    char phoneNum[20];
    int age;
} Person;
```

따라서 이렇듯 구조체의 이름을 생략하는 것도 가능하다.

윤성우의 열혈 C 프로그래밍



Chapter 23-2. 함수로의 구조체 변수
전달과 반환

윤성우 저 열혈강의 C 프로그래밍 개정판

함수의 인자로 전달되고 return문에 의해 반환되는 구조체 변수1

```
typedef struct point
{
    int xpos;
    int ypos;
} Point;

void ShowPosition(Point pos)
{
    printf("[%d, %d] \n", pos.xpos, pos.ypos);
}

Point GetCurrentPosition(void)
{
    Point cen;
    printf("Input current pos: ");
    scanf("%d %d", &cen.xpos, &cen.ypos);
    return cen; 구조체 변수 cen이 통째로 반환된다.
}

int main(void)
{
    Point curPos=GetCurrentPosition();
    ShowPosition(curPos);
    return 0; ShowPosition 함수의 매개변수에
    curPos에 저장된 값이 통째로 복사된다.
}
```

실행결과

Input current pos: 2 4
[2, 4]

배열까지도 통째로 복사

```
typedef struct person
{
    char name[20];
    char phoneNum[20];
    int age;
} Person;
```

구조체의 멤버로 배열이 선언된 경우
구조체 변수를 인자로 전달하거나 반환 시
배열까지도 통째로 복사가 이뤄진다.

실행결과

```
name? Jung
phone? 010-12XX-34XX
age? 22
name: Jung
phone: 010-12XX-34XX
age: 22
```

```
void ShowPersonInfo(Person man)
{
    printf("name: %s \n", man.name);
    printf("phone: %s \n", man.phoneNum);
    printf("age: %d \n", man.age);
}

Person ReadPersonInfo(void)
{
    Person man;
    printf("name? "); scanf("%s", man.name);
    printf("phone? "); scanf("%s", man.phoneNum);
    printf("age? "); scanf("%d", &man.age);
    return man;
}

int main(void)
{
    Person man=ReadPersonInfo();
    ShowPersonInfo(man);
    return 0;
}
```

구조체 기반의 Call-by-reference

```
typedef struct point
{
    int xpos;
    int ypos;
} Point;

void OrgSymTrans(Point * ptr)    // 원점대칭
{
    ptr->xpos = (ptr->xpos) * -1;
    ptr->ypos = (ptr->ypos) * -1;
}

void ShowPosition(Point pos)
{
    printf("[%d, %d] \n", pos.xpos, pos.ypos);
}

int main(void)
{
    Point pos={7, -5};
    OrgSymTrans(&pos);    // pos의 값을 원점 대칭이동시킨다.
    ShowPosition(pos);
    OrgSymTrans(&pos);    // pos의 값을 원점 대칭이동시킨다.
    ShowPosition(pos);
    return 0;
}
```

구조체 변수 대상의 *Call-by-reference*는 일반 변수의 *Call-by-reference*와 동일하다.

실행결과

[-7, 5]

[7, -5]

구조체 변수를 대상으로 가능한 연산1

```
typedef struct point
{
    int xpos;
    int ypos;
} Point;
```

```
int main(void)
{
```

```
    Point pos1={1, 2};
```

```
    Point pos2;
```

```
    pos2=pos1; // pos1의 멤버 대 pos2의 멤버간 복사가 진행됨
```

```
    printf("크기: %d \n", sizeof(pos1)); // pos1의 전체 크기 반환
```

```
    printf("[%d, %d] \n", pos1.xpos, pos1.ypos);
```

```
    printf("크기: %d \n", sizeof(pos2)); // pos2의 전체 크기 반환
```

```
    printf("[%d, %d] \n", pos2.xpos, pos2.ypos);
```

```
    return 0;
```

```
}
```

구조체 변수간 대입 연산의 결과로 멤버 대 멤버 복사가 이뤄진다는 사실을
확인하자!

실행결과

크기: 8

[1, 2]

크기: 8

[1, 2]

구조체 변수를 대상으로 가능한 연산2

```
typedef struct point
{
    int xpos;
    int ypos;
} Point;
```

구조체 변수를 대상으로는 덧셈 및 뺄셈 연산이 불가능하다.

따라서 필요하다면 덧셈함수와 뺄셈함수를 정의해야 한다.

실행결과

```
[7, 15]
[3, -3]
```

```
Point AddPoint(Point pos1, Point pos2)
{
    Point pos={pos1.xpos+pos2.xpos, pos1.ypos+pos2.ypos};
    return pos;
}

Point MinPoint(Point pos1, Point pos2)
{
    Point pos={pos1.xpos-pos2.xpos, pos1.ypos-pos2.ypos};
    return pos;
}

int main(void)
{
    Point pos1={5, 6};
    Point pos2={2, 9};
    Point result;

    result=AddPoint(pos1, pos2);
    printf("[%d, %d] \n", result.xpos, result.ypos);
    result=MinPoint(pos1, pos2);
    printf("[%d, %d] \n", result.xpos, result.ypos);
    return 0;
}
```

구조체 Point의 덧셈 함수

구조체 Point의 뺄셈 함수

윤성우의 열혈 C 프로그래밍



Chapter 23-3. 구조체의 유용함에 대한
논의와 중첩 구조체

윤성우 저 열혈강의 C 프로그래밍 개정판

구조체를 정의하는 이유

- ▶ 연관 있는 데이터를 하나로 묶을 수 있는 자료형을 정의할 수 있다. **구조체의 정의 이유!**
- ▶ 연관 있는 데이터를 묶으면 데이터의 표현 및 관리가 용이해진다.
- ▶ 데이터의 표현 및 관리가 용이해지면 그만큼 합리적인 코드를 작성할 수 있다.

```
typedef struct student
{
    char name[20];        // 학생 이름
    char stdnum[20];      // 학생 고유번호
    char school[20];      // 학교 이름
    char major[20];       // 선택 전공
    int year;             // 학년
} Student;

void ShowStudentInfo(Student * sptr)
{
    printf("학생 이름: %s \n", sptr->name);
    printf("학생 고유번호: %s \n", sptr->stdnum);
    printf("학교 이름: %s \n", sptr->school);
    printf("선택 전공: %s \n", sptr->major);
    printf("학년: %d \n", sptr->year);
}
```

인자 전달 시 용이하다.

```
int main(void)
{
    Student arr[7];
    int i;
    for(i=0; i<7; i++)
    {
        printf("이름: "); scanf("%s", arr[i].name);
        printf("번호: "); scanf("%s", arr[i].stdnum);
        printf("학교: "); scanf("%s", arr[i].school);
        printf("전공: "); scanf("%s", arr[i].major);
        printf("학년: "); scanf("%d", &arr[i].year);
    }
    for(i=0; i<7; i++)
        ShowStudentInfo(&arr[i]);
    return 0;
}
```

하나의 배열 선언으로 종류가 다른 데이터들을 한데 저장할 수 있다..

중첩된 구조체의 정의와 변수의 선언

```
typedef struct point
{
    int xpos;
    int ypos;
} Point;
```

```
typedef struct circle
{
    Point cen;
    double rad;
} Circle;
```

```
void ShowCircleInfo(Circle * cptr)
{
    printf("[%d, %d] \n", (cptr->cen).xpos, (cptr->cen).ypos);
    printf("radius: %g \n\n", cptr->rad);
}

int main(void)
{
    Circle c1={{1, 2}, 3.5};
    Circle c2={{2, 4}, 3.9};
    ShowCircleInfo(&c1);
    ShowCircleInfo(&c2);
    return 0;
}
```

앞서 정의한 구조체는 이후에 새로운 구조체를 선언하는데 있어서
기본 자료형의 이름과 마찬가지로 사용이 될 수 있다.

실행결과

```
[1, 2]
radius: 3.5

[2, 4]
radius: 3.9
```

윤성우의 열혈 C 프로그래밍



Chapter 23-4. 공용체(Union Type)의 정의와 의미

윤성우 저 열혈강의 C 프로그래밍 개정판

구조체 vs. 공용체: 선언방식의 차이



```
typedef struct sbbox
{
    int mem1;
    int mem2;
    double mem3;
} SBox;
```



```
typedef union ubox
{
    int mem1;
    int mem2;
    double mem3;
} UBox;
```

정의 방법에 있어서의 차이는 키워드 *struct*를 쓰느냐, 아니면 키워드 *union*을 쓰느냐에 있다!

구조체 vs. 공용체: 실행결과를 통한 관찰

공용체 변수를 이루는 멤버의 시작 주소 값이

모두 동일함을 관찰하고 공용체 변수의 크기 값을 관찰한다!

실행결과

```
002CFC28 002CFC2C 002CFC30
002CFC18 002CFC18 002CFC18
16 8
```

```
int main(void)
{
    SBox sbx;
    UBox ubx;
    printf("%p %p %p \n", &sbx.mem1, &sbx.mem2, &sbx.mem3);
    printf("%p %p %p \n", &ubx.mem1, &ubx.mem2, &ubx.mem3);
    printf("%d %d \n", sizeof(SBox), sizeof(UBox));
    return 0;
}
```

```
typedef struct sbox
{
    int mem1;
    int mem2;
    double mem3;
} SBox;

typedef union ubox
{
    int mem1;
    int mem2;
    double mem3;
} UBox;
```

구조체 vs. 공용체: 메모리적 차이

```
typedef union ubox    // 공용체 ubox의 정의
{
    int mem1;
    int mem2;
    double mem3;
} UBox;

int main(void)
{
    UBox ubx;    // 8바이트 메모리 할당
    ubx.mem1=20;
    printf("%d \n", ubx.mem2);

    mem1에 저장된 데이터를 덮어쓴다.
    ubx.mem3=7.15;
    printf("%d \n", ubx.mem1);
    printf("%d \n", ubx.mem2);
    printf("%g \n", ubx.mem3);
    return 0;
}
```

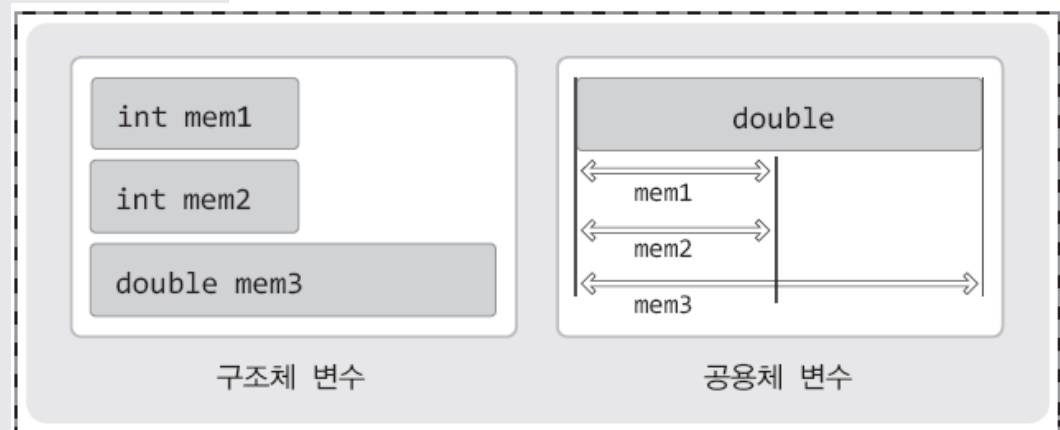
20

-1717986918

-1717986918

7.15

실행결과



공용체의 유용함1: 문제의 제시

- 민선: 수진아! 교수님이 과제를 내 주셨어
- 수진: 뭔데?
- 민선: 프로그램 사용자로부터 int형 정수 하나를 입력 받으래
- 수진: 그래서?
- 민선: 입력 받은 정수의 상위 2바이트와 하위 2바이트 값을 양의 정수로 출력!
- 수진: 그게 다야?
- 민선: 그 다음엔 상위 1바이트와 하위 1바이트에 저장된 값의 아스키 문자 출력!
- 수진: 그거 공용체를 이용해 보라는 깊은 뜻이 담겨있는 것 같은데?



```
typedef struct dbshort
{
    unsigned short upper;
    unsigned short lower;
} DBShort;

typedef union rdbuf
{
    int iBuf;
    char bBuf[4];
    DBShort sBuf;
} RDBuf;
```

해결책이 되는 공용체의 정의

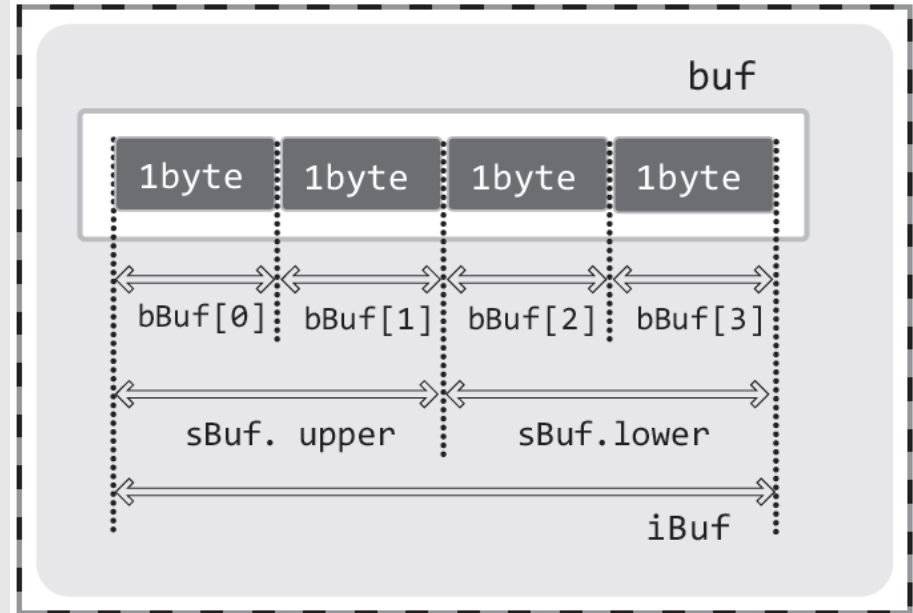
공용체의 유용함2: 문제의 해결

```
typedef struct dbshort
{
    unsigned short upper;
    unsigned short lower;
} DBShort;

typedef union rdbuf
{
    int iBuf;
    char bBuf[4];
    DBShort sBuf;
} RDBuf;

int main(void)
{
    RDBuf buf;
    printf("정수 입력: ");
    scanf("%d", &(buf.iBuf));

    printf("상위 2바이트: %u \n", buf.sBuf.upper);
    printf("하위 2바이트: %u \n", buf.sBuf.lower);
    printf("상위 1바이트 아스키 코드: %c \n", buf.bBuf[0]);
    printf("하위 1바이트 아스키 코드: %c \n", buf.bBuf[3]);
    return 0;
}
```



실행결과

```
정수 입력: 1145258561
상위 2바이트: 16961
하위 2바이트: 17475
상위 1바이트 아스키 코드: A
하위 1바이트 아스키 코드: D
```

윤성우의 열혈 C 프로그래밍



Chapter 23-5. 열거형(Enumerated Type) 의 정의와 의미

윤성우 저 열혈강의 C 프로그래밍 개정판

열거형의 정의와 변수의 선언

▶ 열거형 `syllable`의 정의의 의미

"`syllable`형 변수에 저장이 가능한 정수 값들을 결정하겠다"

▶ 열거형 `syllable`의 정의의 예

```
enum syllable    // syllable이라는 이름의 열거형 정의
{
    Do=1, Re=2, Mi=3, Fa=4, So=5, La=6, Ti=7
};
```

Do, Re, Mi, Fa...

를 열거형 상수라 한다.

Do를 정수 1을 의미하는 상수로 정의한다.

그리고 이 값은 `syllable`형 변수에 저장이 가능하다

▶ `syllable`형 변수의 선언

```
enum syllable tone;    // syllable형 변수 tone의 선언
```

구조체 공용체와 마찬가지로 `typedef` 선언을 추가하여 `enum` 선언을 생략할 수 있다.

열거형의 정의와 변수선언의 예

```
typedef enum syllable
{
    Do=1, Re=2, Mi=3, Fa=4, So=5, La=6, Ti=7
} Syllable;
```

typedef 선언이 추가된 열거형의 정의 및 선언

실행결과

```
도는 하얀 도라지 ♪
레는 둥근 레코드 ♪
미는 파란 미나리 ♪ ♪
파는 예쁜 파랑새 ♪ ♭
술은 작은 술방울 ♪ ♪ ♪
라는 라디오고요~ ♪ ♪ ♭ ♪
시는 졸졸 시냇물 ♪ ♭ ♪ ♪
```

```
void Sound(Syllable sy)
{
    switch(sy)
    {
        case Do:
            puts("도는 하얀 도라지 ♪"); return;
        case Re:
            puts("레는 둥근 레코드 ♪"); return;
        case Mi:
            puts("미는 파란 미나리 ♪ ♪"); return;
        case Fa:
            puts("파는 예쁜 파랑새 ♪ ♭"); return;
        case So:
            puts("술은 작은 술방울 ♪ ♪ ♪"); return;
        case La:
            puts("라는 라디오고요~ ♪ ♪ ♭ ♪"); return;
        case Ti:
            puts("시는 졸졸 시냇물 ♪ ♭ ♪ ♪"); return;
    }
    puts("다 함께 부르세~ 도레미파 솔라시도 솔 도~ 짹~");
}

int main(void)
{
    Syllable tone;
    for(tone=Do; tone<=Ti; tone+=1)
        Sound(tone);
    return 0;
}
```

*열거형 상수는 선언 이후 어디서건
쓸 수 있는 상수가 된다.*

열거형 상수의 값이 결정되는 방식

```
enum color {RED, BLUE, WHITE, BLACK};
```



동일한 선언

```
enum color {RED=0, BLUE=1, WHITE=2, BLACK=3};
```

열거형 상수의 값은 명시되지 않으면 0부터 시작해서 1씩 증가한다.

```
enum color {RED=3, BLUE, WHITE=6, BLACK};
```



동일한 선언

```
enum color {RED=3, BLUE=4, WHITE=6, BLACK=7};
```

값이 명시되지 않는 상수는 앞에 정의된 상수 값에서 1이 증가한다.



열거형의 유용함

```
typedef enum syllable
{
    Do=1, Re=2, Mi=3, Fa=4, So=5, La=6, Ti=7
} Syllable;
```

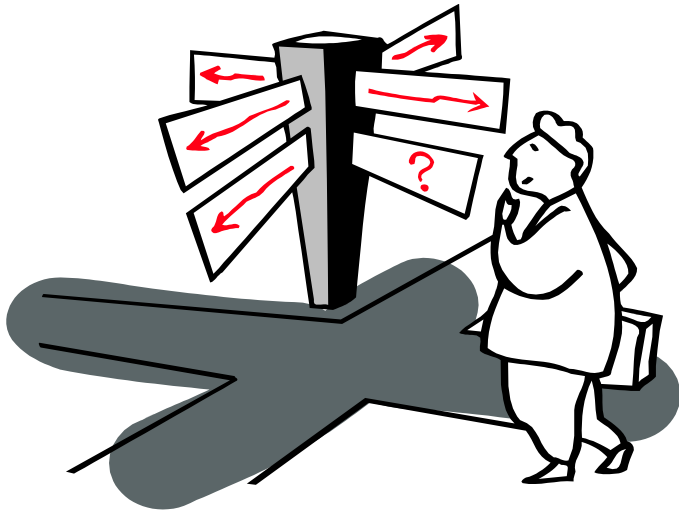
Syllable이라는 이름의 자료형 안에서 음계에
관련있는 상수들을 모두 묶어서 정의하였다!

```
enum {
    Do=1, Re=2, Mi=3, Fa=4, So=5, La=6, Ti=7 };
```

변수의 선언이 목적이 아닌 상수의 선언이 목
적인 경우 이렇듯 열거형의 이름과 **typedef** 선
언을 생략하기도 한다.

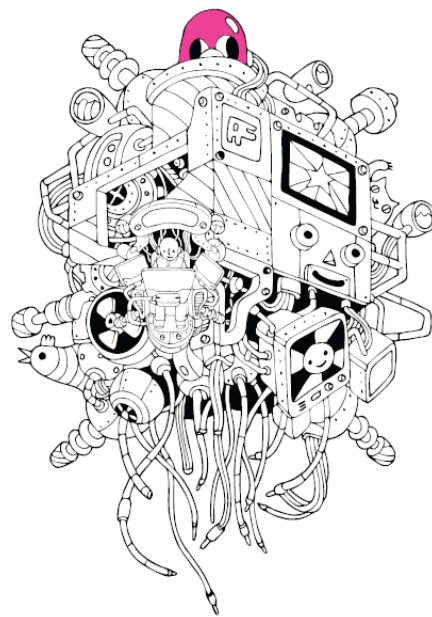
열거형의 유용함은 둘 이상의 연관이 있는 이름을 상수로 선언함으로써
프로그램의 가독성을 높이는데 있다.





Chapter 23이 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 24. 파일 입출력

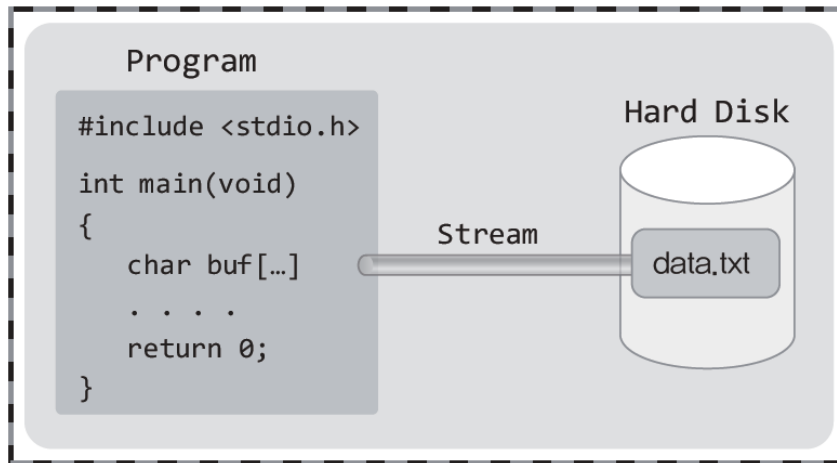
윤성우의 열혈 C 프로그래밍



Chapter 24-1. 파일과 스트림 그리고
기본적인 파일의 입출력

윤성우 저 열혈강의 C 프로그래밍 개정판

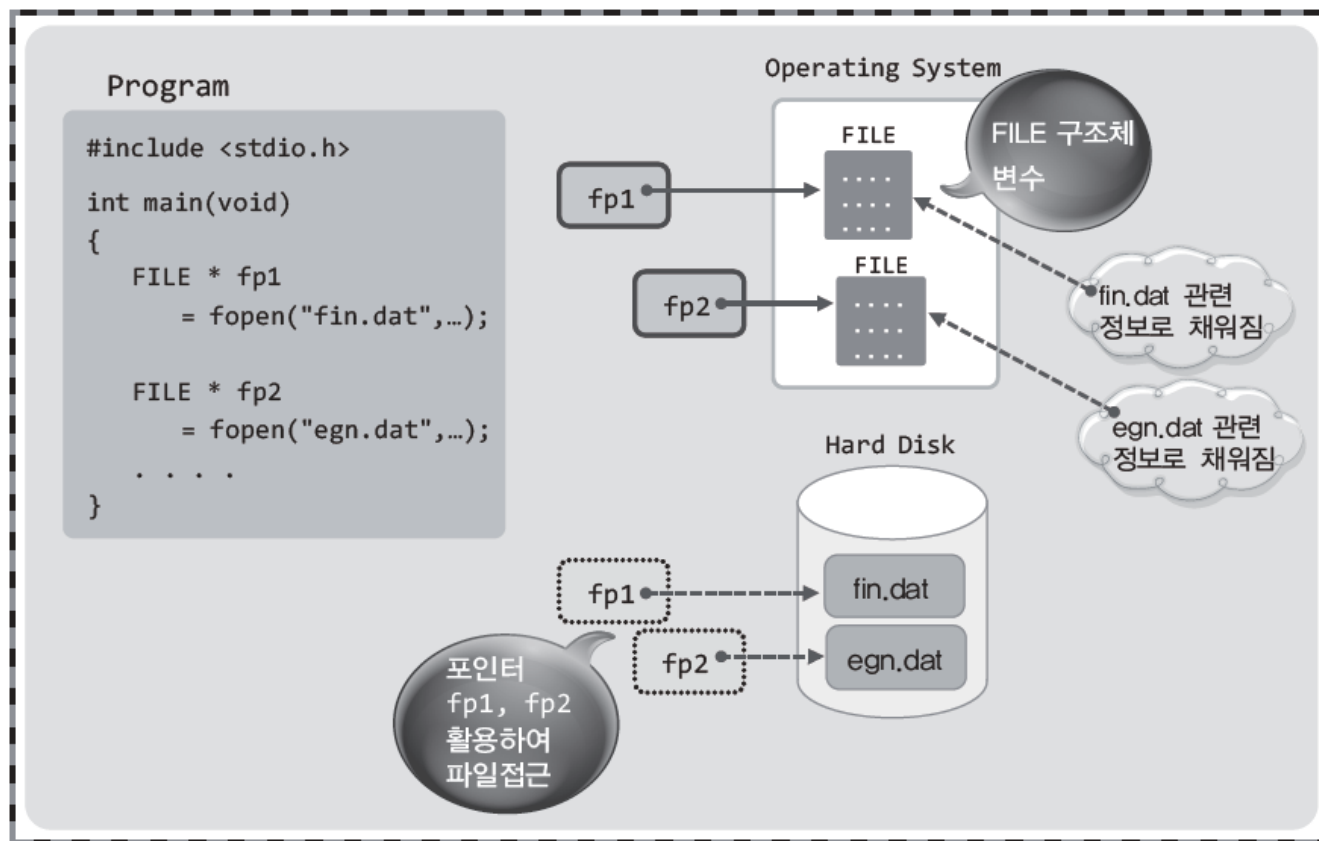
파일에 저장되어 있는 데이터를 읽고 싶어요.



콘솔 입출력과 마찬가지로 파일로부터의 데이터 입출력을 위해서는 **스트림**이 형성되어야 한다.

파일과의 스트림 형성은 데이터 입출력의 기본이다.

fopen 함수를 통한 스트림의 형성과 FILE 구조체



fopen 함수호출 시 생성되는 FILE 구조체 변수와 이를 참조하는 FILE 구조체 포인터 변수의 관계를 이해하자!

fopen 함수호출의 결과

스트림을 형성할 파일의 이름 형성할 스트림의 종류

```
#include <stdio.h>
FILE * fopen(const char * filename, const char * mode);
```

➔ 성공 시 해당 파일의 FILE 구조체 변수의 주소 값, 실패 시 NULL 포인터 반환

- **fopen** 함수가 호출되면 FILE 구조체 변수가 생성된다.
- 생성된 FILE 구조체 변수에는 파일에 대한 정보가 담긴다.
- FILE 구조체의 포인터는 사실상 파일을 가리키는 '지시자'의 역할을 한다.

fopen 함수가 파일과의 스트림 형성을 요청하는 기능의 함수이다.



출력 스트림의 생성

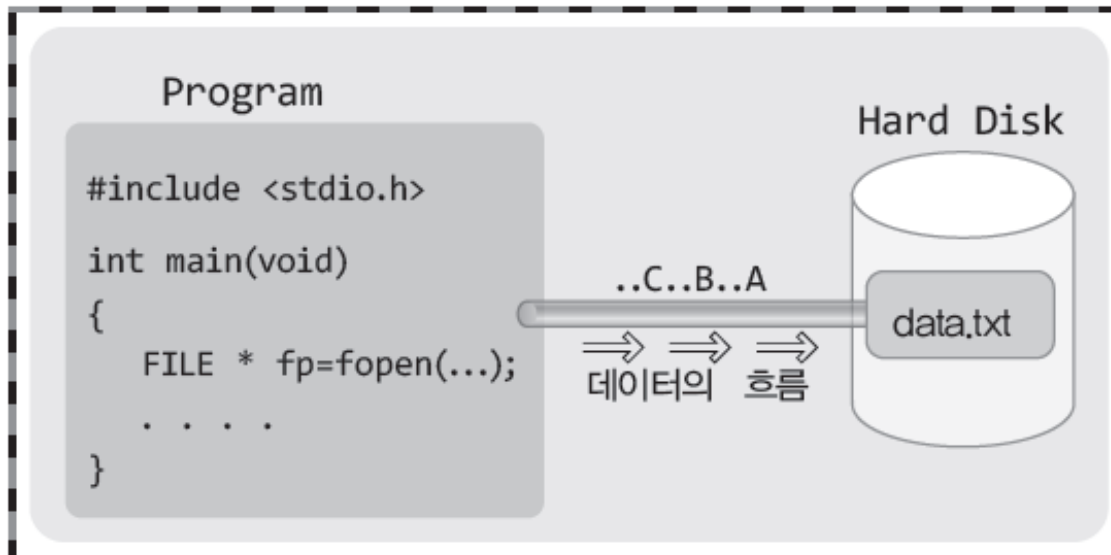
"wt"에는 출력 스트림의
의미가 담겨있다.

```
FILE * fp = fopen("data.txt", "wt");
```

“파일 data.txt와 스트림을 형성하되 wt 모드로 스트림을 형성해라!”



출력 스트림의 형성 결과



포인터 변수 fp 에 저장된 값이
 $data.txt$ 의 스트림에 데이터를
전송하는 도구가 된다.

입력 스트림의 생성

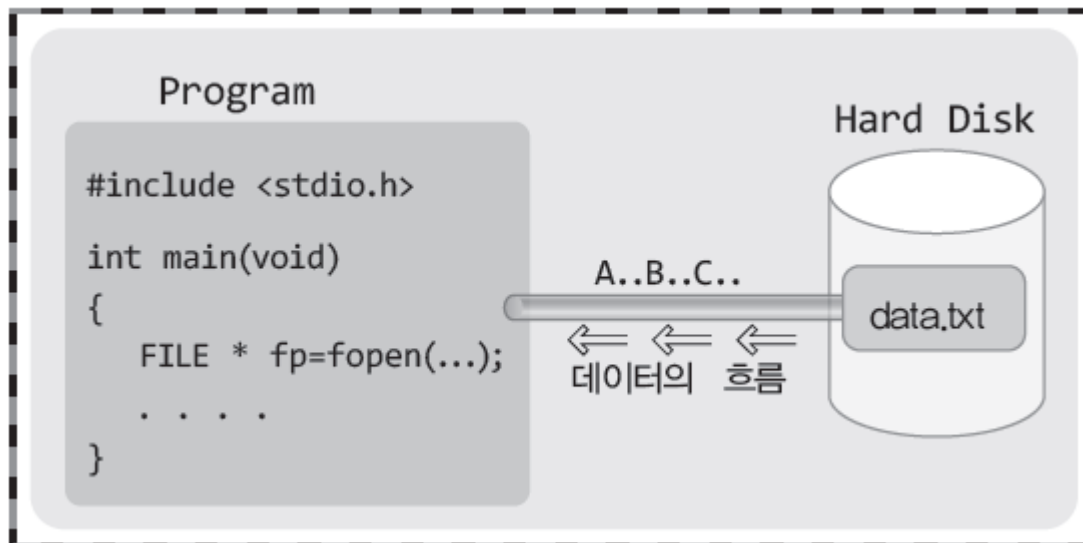
"rt"에는 입력 스트림의
의미가 담겨있다.

```
FILE * fp = fopen("data.txt", "rt");
```

“파일 data.txt와 스트림을 형성하되 rt 모드로 스트림을 형성해라!”



입력 스트림의 형성 결과



포인터 변수 *fp*에 저장된 값이
*data.txt*의 스트림으로부터 데이터
를 수신하는 도구가 된다.

파일에 데이터를 써봅시다.

현재 디렉터리에 저장된 파일 data.txt를 찾는다.

```
int main(void)
{
    FILE * fp=fopen("data.txt", "wt");
    if(fp==NULL) {
        puts("파일오픈 실패!");
        return -1;    // 비정상적 종료를 의미하기 위해서 -1을 반환
    }

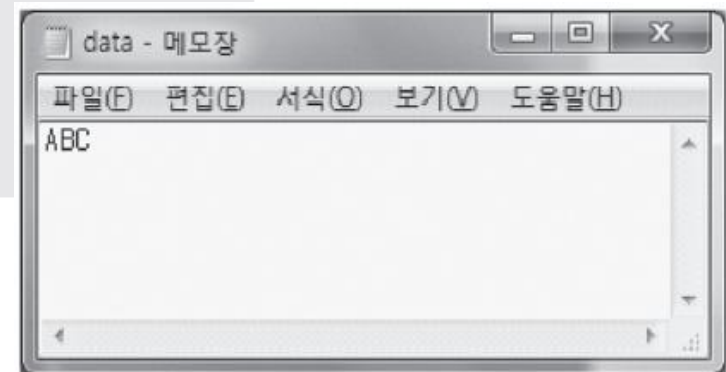
    fputc('A', fp);
    fputc('B', fp);
    fputc('C', fp);
    fclose(fp);    // 스트림의 종료
    return 0;
}
```

현재 디렉터리는 실행파일이 저장된 디렉터리이거나
프로젝트 파일이 저장된 디렉터리이다!

← 문자 A를 fp가 가리키는 파일에 저장해라!

```
FILE * fp = fopen("C:\\Project\\data.txt", "wt");
```

fopen 함수호출 시 경로를 완전히 명시할 수도 있다.



메모장으로 파일을 열어서 확인해 본다.

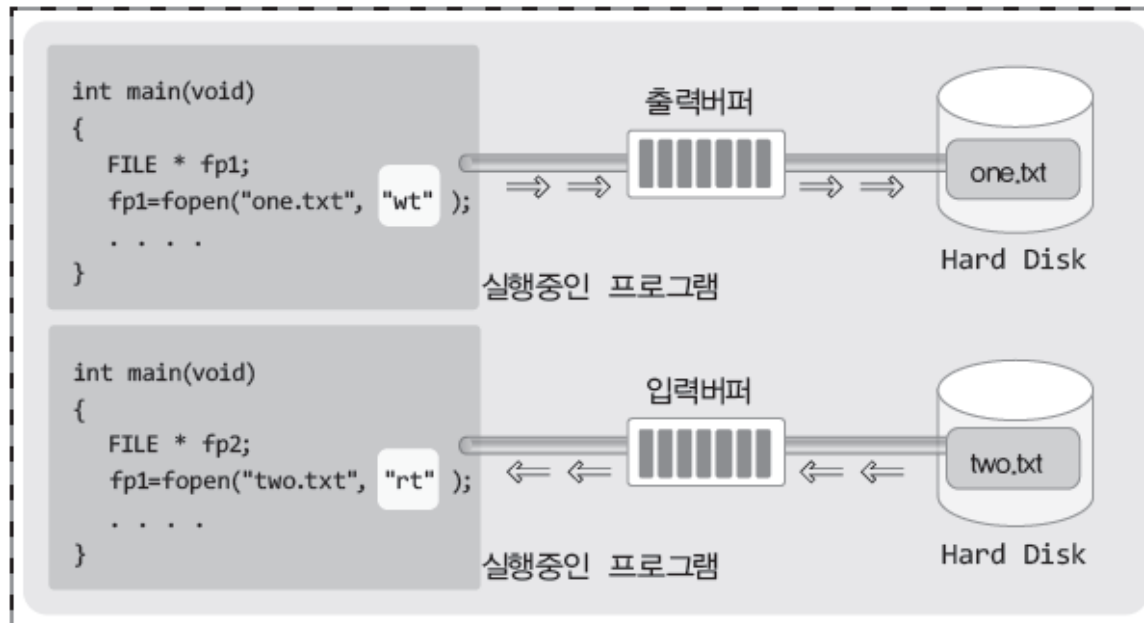
스트림의 소멸을 요청하는 fclose 함수

```
#include <stdio.h>
int fclose(FILE * stream);
```

➔ 성공 시 0, 실패 시 EOF를 반환

fclose 함수호출이 동반하는 두 가지

- 운영체제가 할당한 자원의 반환
- 버퍼링 되었던 데이터의 출력



fclose 함수가 호출되어야 스트림 형성 시 할당된 모든 리소스가 소멸이 된다. 따라서 파일이
오픈 된 상태로 놔두는 것은 좋지 않다.

Ch21에서 호출한 적 있는 fflush 함수

```
#include <stdio.h>
int fflush(FILE * stream);
```

➔ 함수호출 성공 시 0, 실패 시 EOF 반환

콘솔 대상으로 *fflush* 함수를 설명한바 있다.

대상이 파일로 바뀌었을 뿐 달라지는 것은 없다.

- 출력버퍼를 비운다는 것은 출력버퍼에 저장된 데이터를 목적지로 전송한다는 의미
- 입력버퍼를 비운다는 것은 입력버퍼에 저장된 데이터를 소멸시킨다는 의미
- fflush 함수는 출력버퍼를 비우는 함수이다.
- fflush 함수는 입력버퍼를 대상으로 호출할 수 없다.

```
int main(void)
{
    FILE * fp = fopen("data.txt", "wt");
    . . . .
    fflush(fp);    // 출력 버퍼를 비우라는 요청!
    . . . .
}
```

이렇듯 *fflush* 함수의 호출을 통하여 *fclose* 함수를 호출하지 않고도 출력버퍼만 비울 수 있다.

그렇다면 파일의 입력버퍼는 어떻게 비우는가?

이를 위한 별도의 함수가 정의되어 있는가?

파일로부터 데이터를 읽어 봅시다.

```
int main(void)
{
    int ch, i;
    FILE * fp=fopen("data.txt", "rt");
    if(fp==NULL) {
        puts("파일오픈 실패!");
        return -1;
    }
    for(i=0; i<3; i++)
    {
        ch=fgetc(fp);
        printf("%c \n", ch);
    }
    fclose(fp);
    return 0;
}
```

*fp로부터 하나의 문자를 읽어서
변수 ch에 저장해라!*

A
B
C

이전에 문자가 써진 순서대로 읽힌다!

실행결과



A collection of various cartoon objects including a cat, a fish, a laptop, a robot, a planet, a key, a die, a TV, a camera, a ring, and a small alien.

윤성우 저 열혈강의 C 프로그래밍 개정판

스트림의 구분 기준 두 가지(Basic)

• 기준1

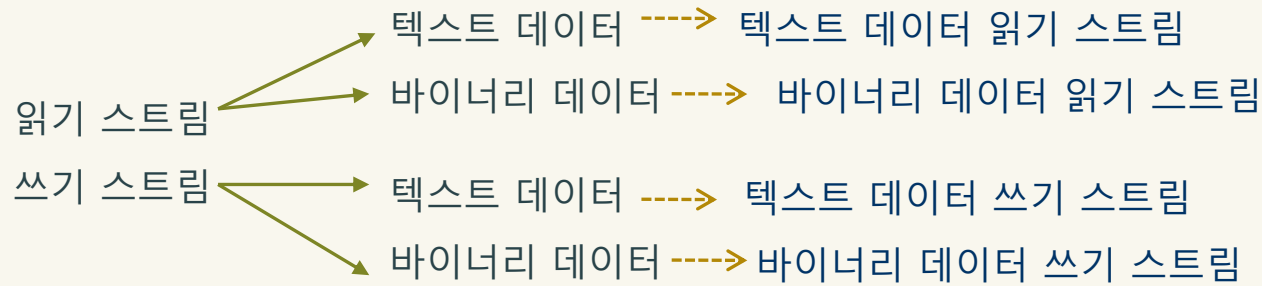
읽기 위한 스트림이냐? 쓰기 위한 스트림이냐?

파일에 데이터를 쓰는데 사용하는 스트림과 데이터를 읽는데 사용하는 스트림은 구분이 된다.

• 기준2

텍스트 데이터를 위한 스트림이냐? 바이너리 데이터를 위한 스트림이냐?

추력의 대상이 되는 데이터의 종류에 따라서 스트림은 두 가지로 나뉜다.



기본적인 스트림의 구분!
그러나 실제로는 더 세분화!

스트림을 구분하는 기준1: Read or Write

√ 스트림의 성격은 R/W를 기준으로 다음과 같이 세분화 된다.

모드(mode)	스트림의 성격	파일이 없으면?
r	읽기 가능	에러
w	쓰기 가능	생성
a	파일의 끝에 덧붙여 쓰기 가능	생성
r+	읽기/쓰기 가능	에러
w+	읽기/쓰기 가능	생성
a+	읽기/덧붙여 쓰기 가능	생성

➔ 모드의 +는 읽기, 쓰기가 모두 가능한 스트림의 형성을 의미한다.

➔ 모드의 a는 쓰기가 가능한 스트림을 의미하되 여기서 말하는 쓰기는 덧붙여 쓰기이다.



스트림을 구분하는 기준2: 텍스트 모드, 바이너리 모드

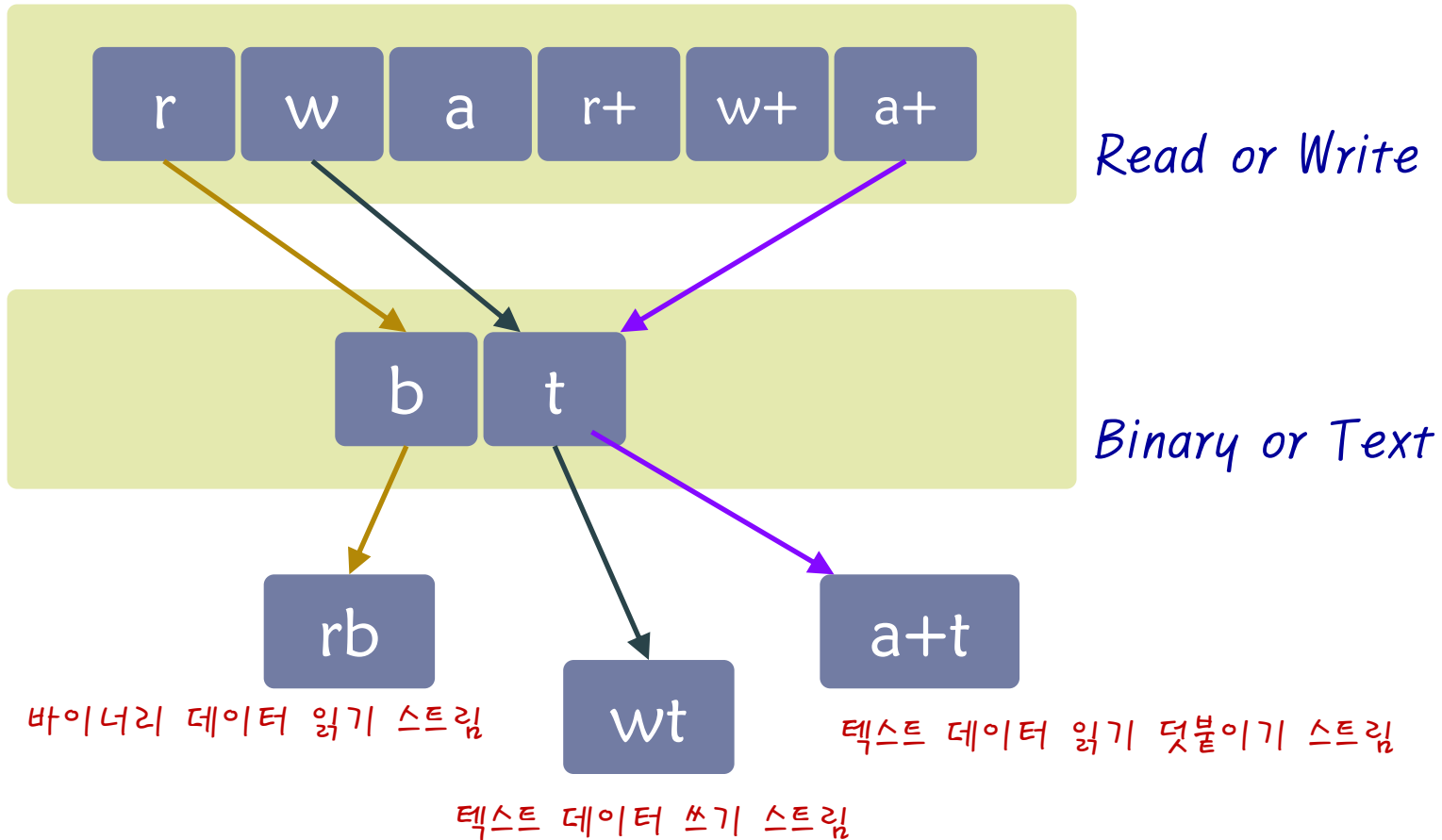
√ 스트림의 성격은 데이터의 종류에 따라서 다음과 같이 두 가지로 나뉜다.

- ▶ 텍스트 모드 스트림 (t) : 문자 데이터를 저장하는 스트림
- ▶ 바이너리 모드 스트림 (b) : 바이너리 데이터를 저장하는 스트림

√ 문자 데이터와 바이너리 데이터

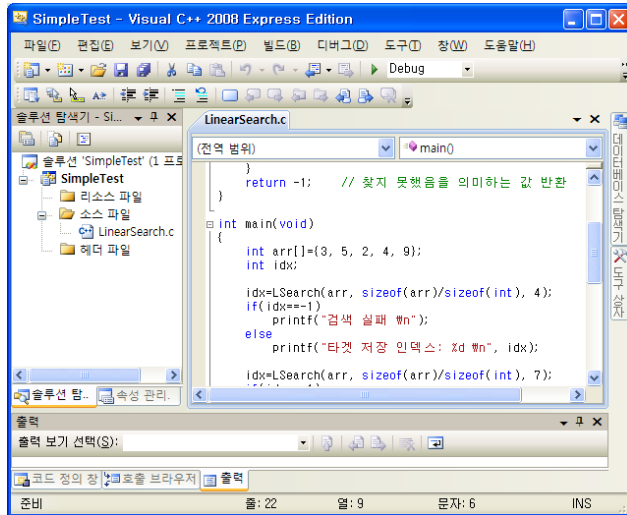
- ▶ 문자 데이터 : 사람이 인식할 수 있는 유형의 문자로 이뤄진 데이터
 - 파일에 저장된 문자 데이터는 Windows의 메모장으로 열어서 문자 확인이 가능
 - 예 : 도서목록, 물품가격, 전화번호, 주민등록번호
- ▶ 바이너리 데이터 : 컴퓨터가 인식할 수 있는 유형의 데이터
 - 메모장과 같은 편집기로는 그 내용이 의미하는 바를 이해할 수 없다.
 - 예 : 음원 및 영상 파일, 그래픽 디자인 프로그램에 의해 저장된 디자인 파일

파일의 개방모드 조합!



`t`도 `b`도 붙지 않으면 텍스트 모드로 파일 개방

텍스트 스트림이 별도로 존재하는 이유1



C언어는 개행을 `\n`으로 표시하기로 약속하였다.
따라서 개행 정보를 저장할 때 C 프로그램상에서 우리는 `\n`을 저장한다.



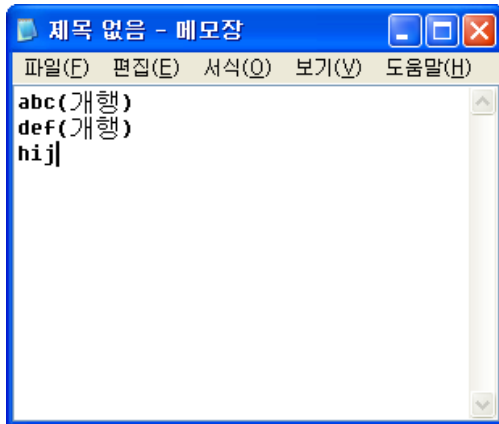
개행 정보로 저장된 `\n`은 문제가 되지 않을까?

text.txt

텍스트 스트림이 별도로 존재하는 이유2

text.txt

운영체제 별로 개행을 표시하는 방법에는 차이가 있다. 만약
에 개행을 `\n`으로 표현하지 않는 운영체제가 있다면?

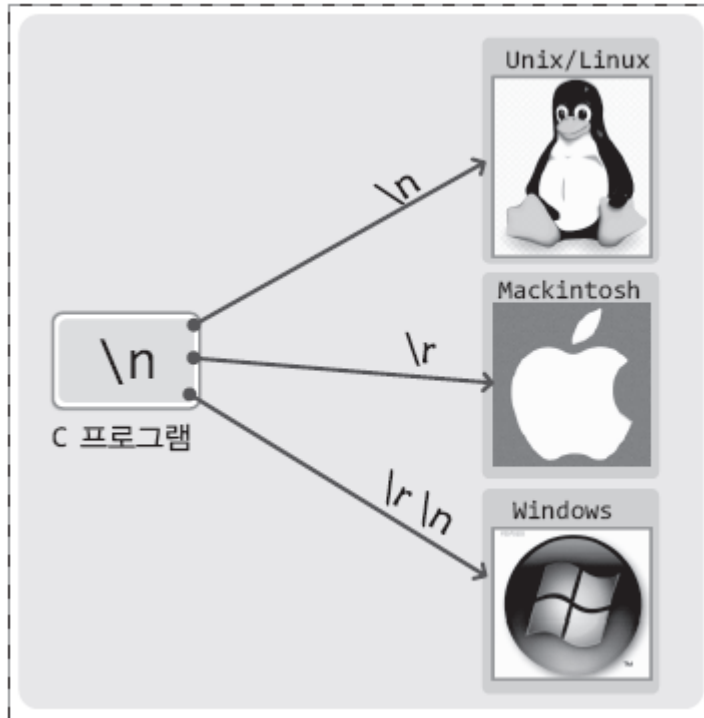


개행을 `\n`으로 표현하지 않는 운영체제는 `\n`을
전혀 다르게 해석하게 된다.

운영체제 별 개행의 표시 방법

- | | |
|-----------|-------------------|
| ▶ Windows | <code>\r\n</code> |
| ▶ Linux | <code>\n</code> |
| ▶ Mac | <code>\r</code> |

텍스트 스트림이 별도로 존재하는 이유3



개행 정보를 정확히 저장하기 위해서는 위와 같은 종류의 변환 과정을 거쳐야 한다.

텍스트 모드로 데이터를 입출력 하면 이러한 형태의 변환이 운영체제에 따라서 자동으로 이뤄진다.

윤성우의 열혈 C 프로그래밍



Chapter 24-3. 파일 입출력 함수의 기본

윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 21에서 학습한 파일 입출력 함수들

텍스트 데이터 입출력 함수들

```
int fputc(int c, FILE * stream);           // 문자 출력
int fgetc(FILE * stream);                  // 문자 입력
int fputs(const char * s, FILE * stream);  // 문자열 출력
char * fgets(char * s, int n, FILE * stream); // 문자열 입력
```

당시에는 매개변수 *stream*에 *stdin* 또는 *stdout*을 인자로 전달하여 콘솔을 대상으로 입출력을 진행하였지만, 위의 함수들은 *FILE* 구조체의 포인터를 인자로 전달하여 파일을 대상으로 입출력을 진행할 수 있는 함수들이다.



파일 입출력의 예

write 순서대로 read해야 한다!

```
int main(void)
{
    FILE * fp=fopen("simple.txt", "wt");
    if(fp==NULL) {
        puts("파일오픈 실패!");
        return -1;
    }

    fputc('A', fp); 문자 A와 B가
    fputc('B', fp); fp가 가리키는 파일에 저장
    fputs("My name is Hong \n", fp);
    fputs("Your name is Yoon \n", fp);
    fclose(fp); 두 개의 문자열이 fp가 가리
    return 0; 키는 파일에 저장
}
```

파일에 저장된 문자열의 끝에는 널이 존재하지 않는다.
때문에 파일을 대상으로 문자열을 입출력 할 때에는 개
행을 의미하는 \n은 문자열의 마지막에 넣어줘야 한
다. \n은 기준으로 문자열을 구분하기 때문이다.

```
int main(void)
{
    char str[30];
    int ch;
    FILE * fp=fopen("simple.txt", "rt");
    if(fp==NULL) {
        puts("파일오픈 실패!");
        return -1;
    }

    ch=fgetc(fp);
    printf("%c \n", ch);
    ch=fgetc(fp);
    printf("%c \n", ch);

    fgets(str, sizeof(str), fp);
    printf("%s", str); \n을 만날때까지 read
    fgets(str, sizeof(str), fp);
    printf("%s", str); \n을 만날때까지 read

    fclose(fp);
    return 0;
}
```

A 실행결과
B
My name is Hong
Your name is Yoon

feof 함수 기반의 파일복사 프로그램

```
#include <stdio.h>
int feof(FILE * stream);
```

➔ 파일의 끝에 도달한 경우 0이 아닌 값 반환

파일의 끝을 확인해야 하는 경우 이 함수가 필요하다. 파일 입력 함수는 오류가 발생하는 경우에도 EOF를 반환한다. 따라서 EOF의 반환원인을 확인하려면 이 함수를 호출해야 한다.

feof 함수호출을 통해서
EOF 반환 원인을 확인!

```
int main(void) 문자 단위 파일복사 프로그램
{
    FILE * src=fopen("src.txt", "rt");
    FILE * des=fopen("dst.txt", "wt");
    int ch;
    if(src==NULL || des==NULL) {
        puts("파일오픈 실패!");
        return -1;
    }
    while((ch=fgetc(src))!=EOF)
        fputc(ch, des);
    if(feof(src)!=0)
        puts("파일복사 완료!");
    else
        puts("파일복사 실패!");
    fclose(src);
    fclose(des);
    return 0;
}
```

EOF가 반환이 되면...

문자열 단위 파일복사 프로그램

```
int main(void)
{
    FILE * src=fopen("src.txt", "rt");
    FILE * des=fopen("des.txt", "wt");
    char str[20];

    if(src==NULL || des==NULL) {
        puts("파일오픈 실패!");
        return -1;
    }

    while(fgets(str, sizeof(str), src)!=NULL)
        fputs(str, des);

    if(feof(src)!=0)
        puts("파일복사 완료!");
    else
        puts("파일복사 실패!");

    fclose(src);
    fclose(des);
    return 0;
}
```

EOF가 반환이 되면...

문자 단위로 복사를 진행하느냐 문자열 단위로
복사를 진행하느냐의 차이만 있을 뿐!

feof 함수호출을 통해서
EOF 반환 원인을 확인!



바이너리 데이터의 입출력: fread

```
#include <stdio.h>
size_t fread(void * buffer, size_t size, size_t count, FILE * stream);
```

➔ 성공 시 전달인자 count, 실패 또는 파일의 끝 도달 시 count보다 작은 값 반환

```
int main(void)
{
    int buf[12];
    . . . .
    fread((void*)buf, sizeof(int), 12, fp);
    . . . .
```

*sizeof(int) 크기의 데이터 12개를 fp로부터
읽어 들여서 배열 buf에 저장하라!*



바이너리 데이터의 입출력: fwrite

```
#include <stdio.h>
size_t fwrite(const void * buffer, size_t size, size_t count, FILE * stream);
```

➔ 성공 시 전달인자 count, 실패 시 count보다 작은 값 반환

```
int main(void)
{
    int buf[7]={1, 2, 3, 4, 5, 6, 7};
    ....
    fwrite((void*)buf, sizeof(int), 7, fp);
    ....
```

*sizeof(int) 크기의 데이터 7개를 buf로부터
읽어서 fp에 저장해라!*



바이너리 파일 복사 프로그램

```
int main(void)
{
    FILE * src=fopen("src.bin", "rb");
    FILE * des=fopen("dst.bin", "wb");
    char buf[20];
    int readCnt;
    if(src==NULL || des==NULL) {
        puts("파일오픈 실패!");
        return -1;
    }
}
```

1.

파일의 끝에 도달해서 *buf*를 다 채우지 못한 경우에 참이 된다!

2.

feof 함수호출의 결과가 참이면 파일의 끝에 도달했다는 의미이므로 마지막으로 읽은 데이터를 파일에 저장하고 프로그램을 종료한다!

```
while(1)
{
    readCnt=fread((void*)buf, 1, sizeof(buf), src);
    1. if(readCnt<sizeof(buf))
    {
        if(feof(src)!=0)
        {
            2. fwrite((void*)buf, 1, readCnt, des);
            puts("파일복사 완료");
            break;
        }
        else
            puts("파일복사 실패");

        break;
    }
    fwrite((void*)buf, 1, sizeof(buf), des);
}
fclose(src);
fclose(des);
return 0;
}
```


윤성우의 열혈 C 프로그래밍



Chapter 24-4. 텍스트 데이터와 바이너리 데이터를 동시에 입출력 하기

윤성우 저 열혈강의 C 프로그래밍 개정판

서식에 따른 데이터 입출력: fprintf, fscanf

```
char name[10]="홍길동";
char sex='M';
int age=24;
fprintf(fp, "%s %c %d", name, sex, age);
```

fprintf 함수를 이용하면 어떻게 텍스트 & 바이너리 데이터를 동시에 출력할 수 있을까?

fprintf 함수는 *printf* 함수와 그 사용방법이 매우 유사하다. 다만 *fp*를 대상으로 조합이 된 문자열이 출력(저장)될 뿐이다.

```
char name[10];
char sex;
int age;
fscanf(fp, "%s %c %d", name, &sex, &age);
```

fscanf 함수를 이용하면 어떻게 텍스트 & 바이너리 데이터를 동시에 입력할 수 있을까?

sprintf 함수는 *printf* 함수와 그 사용방법이 매우 유사하다. 다만 *fp*를 대상으로 서식문자의 조합 형태대로 데이터가 입력될 뿐이다.



fprintf & fscanf 관련 예제

```
int main(void)
{
    char name[10];
    char sex;
    int age;

    FILE * fp=fopen("friend.txt", "wt");
    int i;
    for(i=0; i<3; i++)
    {
        printf("이름 성별 나이 순 입력: ");
        scanf("%s %c %d", name, &sex, &age);
        getchar(); // 버퍼에 남아있는 \n의 소멸을 위해서
        fprintf(fp, "%s %c %d", name, sex, age);
    }
    fclose(fp);
    return 0;
}
```

저장하는 데이터가 문자열이므로
텍스트 모드로 개방한다!

정은영 F 22
한수정 F 26
이영호 M 31

실행결과

이름 성별 나이 순 입력: 정은영 F 22
이름 성별 나이 순 입력: 한수정 F 26
이름 성별 나이 순 입력: 이영호 M 31

실행결과

```
int main(void)
{
    char name[10];
    char sex;
    int age;

    FILE * fp=fopen("friend.txt", "rt");
    int ret;
    while(1)
    {
        ret=fscanf(fp, "%s %c %d", name, &sex, &age);
        if(ret==EOF)
            break;
        printf("%s %c %d \n", name, sex, age);
    }
    fclose(fp);
    return 0;
}
```

Text/Binary의 집합체인 구조체 변수 입출력

```
typedef struct fren
{
    char name[10];
    char sex;
    int age;
} Friend;
```

```
int main(void)
{
    FILE * fp;
    Friend myfren1;
    Friend myfren2;

    /*** file write ***/
    fp=fopen("friend.bin", "wb");
    printf("이름, 성별, 나이 순 입력: ");
    scanf("%s %c %d", myfren1.name, &(myfren1.sex), &(myfren1.age));
    fwrite((void*)&myfren1, sizeof(myfren1), 1, fp);
    fclose(fp);
    /*** file read ***/
    fp=fopen("friend.bin", "rb");
    fread((void*)&myfren2, sizeof(myfren2), 1, fp);
    printf("%s %c %d \n", myfren2.name, myfren2.sex, myfren2.age);
    fclose(fp);
    return 0;
}
```

바이너리 모드로 통째로
구조체 변수를 저장

바이너리 모드로 통째로
구조체 변수를 복원

이름, 성별, 나이 순 입력: Jungs M 27
Jungs M 27

실행결과

구조체 변수의 입출력은 생각보다 어렵지 않다.

fread & fwrite 함수 기반으로 통째로 입출력 하면 된다.

윤성우의 열혈 C 프로그래밍



Chapter 24-5. 임의 접근을 위한 '파일
위치 지시자'의 이동

윤성우 저 열혈강의 C 프로그래밍 개정판

파일 위치 지시자란?

- FILE 구조체의 멤버 중 하나.
- read 모드로 오픈 된 파일 위치 지시자: "어디까지 읽었더라?"에 대한 답
- write 모드로 오픈 된 파일 위치 지시자: "어디부터 이어서 쓰더라?"에 대한 답
- 즉, Read/Write에 대한 위치 정보를 갖고 있다.

따라서 파일 입출력과 관련이 있는 *fputs*, *fread*, *fwrite*와 같은 함수가 호출될 때마다 파일 위치 지시자의 참조 위치는 변경이 된다.



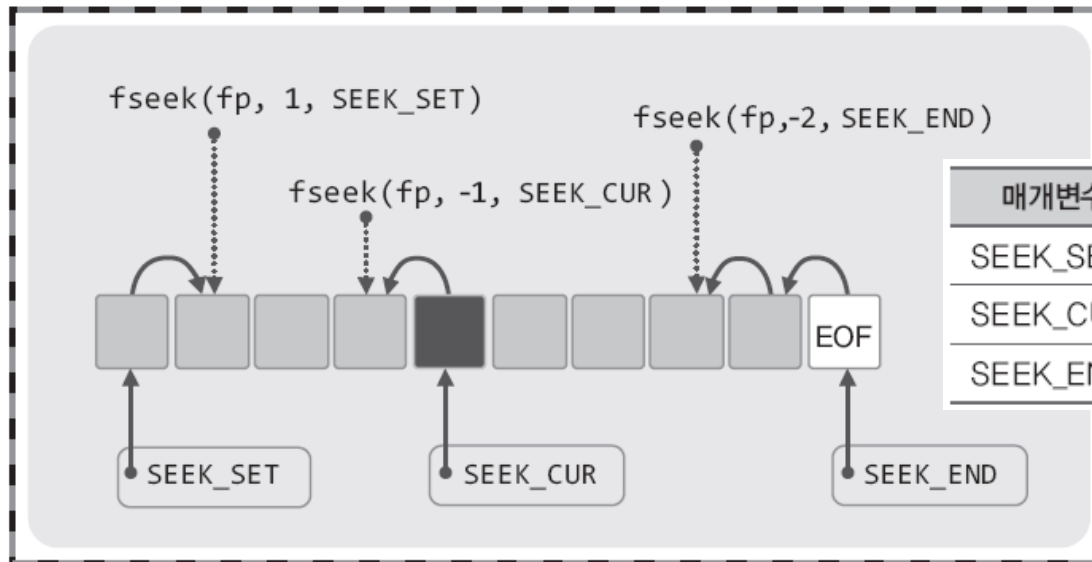
파일 위치 지시자의 이동: fseek

```
#include <stdio.h>
int fseek(FILE * stream, long offset, int wherefrom);
```

➔ 성공 시 0, 실패 시 0이 아닌 값을 반환

파일 위치 지시자의

참조 위치를 변경시키는 함수



매개변수 wherefrom 이...	파일 위치 지시자는...
SEEK_SET(0) 이라면	파일 맨 앞에서부터 이동을 시작
SEEK_CUR(1) 이라면	현재 위치에서부터 이동을 시작
SEEK_END(2) 이라면	파일 맨 끝에서부터 이동을 시작

fseek 함수의 호출결과로 인한 파일 위치 지시자의 이동 결과

fseek 함수의 호출의 예

```
int main(void)
{
    /* 파일생성 */
    FILE * fp=fopen("text.txt", "wt");
    fputs("123456789", fp);
    fclose(fp);

    /* 파일개방 */
    fp=fopen("text.txt", "rt");

    /* SEEK_END test */
    fseek(fp, -2, SEEK_END);
    putchar(fgetc(fp));

    /* SEEK_SET test */
    fseek(fp, 2, SEEK_SET);
    putchar(fgetc(fp));

    /* SEEK_CUR test */
    fseek(fp, 2, SEEK_CUR);
    putchar(fgetc(fp));

    fclose(fp);
    return 0;
}
```

/ 2 3 4 5 6 7 8 9 e eof)

/ 2 3 4 5 6 7 8 9 e eof)

/ 2 3 4 5 6 7 8 9 e eof)

/ 2 3 4 5 6 7 8 9 e eof)

/ 2 3 4 5 6 7 8 9 e eof)

/ 2 3 4 5 6 7 8 9 e eof)

실행결과

836

현재 파일 위치 지시자의 위치는?: ftell

```
#include <stdio.h>
long ftell(FILE * stream);
```

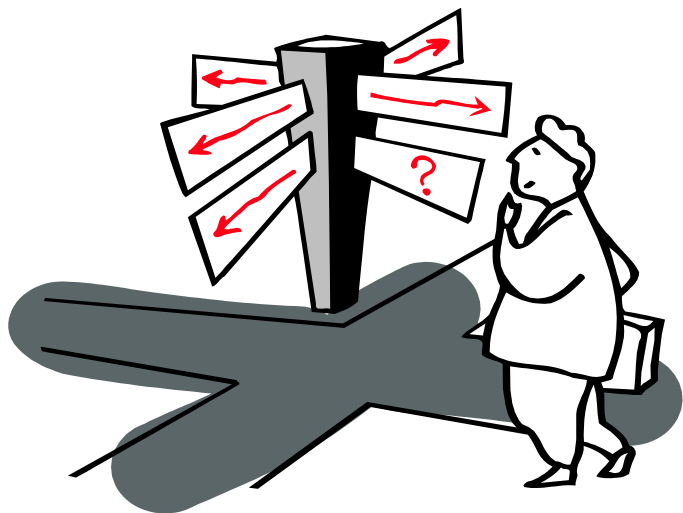
➔ 파일 위치 지시자의 위치 정보 반환

현재 파일 위치자의 위치 정보를 반환하는 함수!

```
int main(void)
{
    long fpos;
    int i;
    /* 파일생성 */
    FILE * fp=fopen("text.txt", "wt");
    fputs("1234-", fp);
    fclose(fp);
    /* 파일개방 */
    fp=fopen("text.txt", "rt");
    for(i=0; i<4; i++)
    {
        putchar(fgetc(fp));
        fpos=ftell(fp); 현재 위치 저장
        fseek(fp, -1, SEEK_END);
        putchar(fgetc(fp));
        fseek(fp, fpos, SEEK_SET);
    } 저장해 놓은 위치 복원
    fclose(fp);
    return 0;
}
```

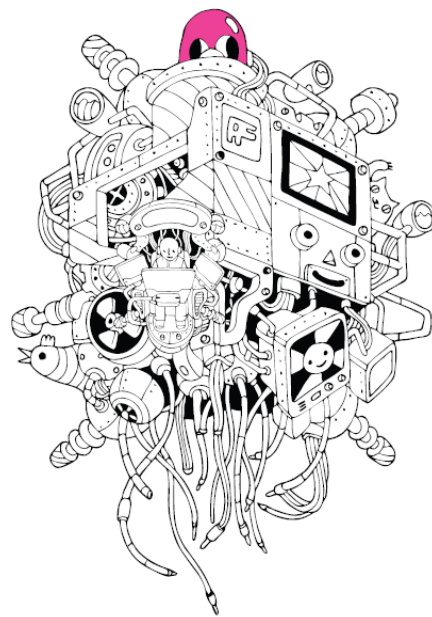
실행결과

1-2-3-4-



Chapter 24가 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 25. 메모리 관리와 메모리의 동적 할당

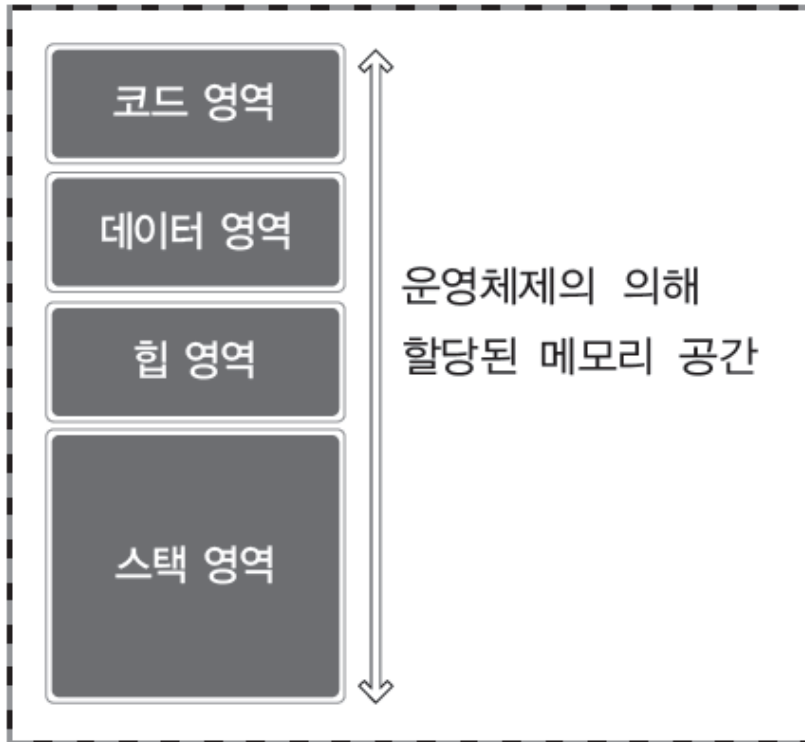
윤성우의 열혈 C 프로그래밍



Chapter 25-1. C언어의 메모리 구조

윤성우 저 열혈강의 C 프로그래밍 개정판

메모리의 구성



메모리 공간을 나뉘놓은 이유는 커다란 서랍장의 수납공간이 나뉘어 있는 이유와 유사하다.

메모리 공간을 나뉘어서 유사한 성향의 데이터를 묶어서 저장을 하면, 관리가 용이해지고 메모리의 접근 속도가 향상된다.

메모리 영역별로 저장되는 데이터의 유형

코드 영역

실행할 프로그램의 코드가 저장되는 메모리 공간.
CPU는 코드 영역에 저장된 명령문을 하나씩 가져다가 실행

데이터 영역

전역변수와 static 변수가 할당되는 영역.
프로그램 시작과 동시에 할당되어 종료 시까지 남아있는 특징의 변수가 저장되는 영역

힙 영역

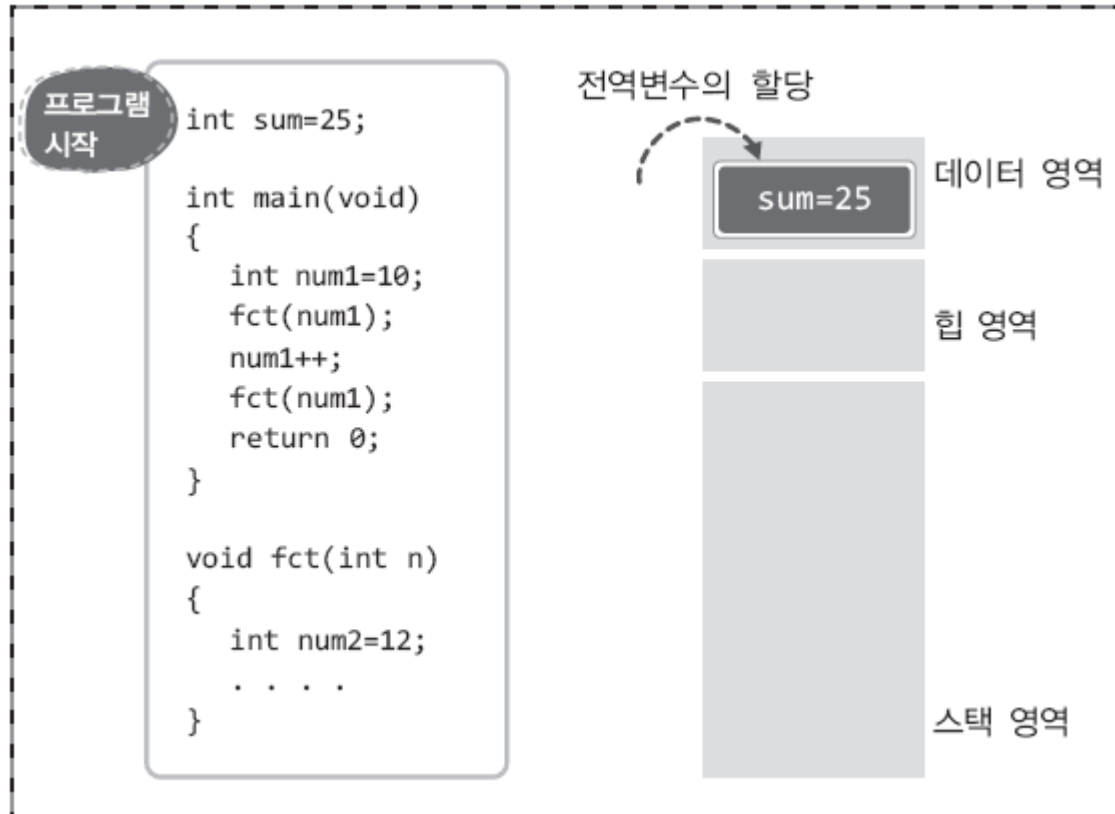
프로그래머가 원하는 시점에 메모리 공간에 할당 및 소멸을 하기 위한 영역

스택 영역

지역변수와 매개변수가 할당되는 영역
함수를 빠져나가면 소멸되는 변수를 저장하는 영역



프로그램의 실행에 따른 메모리의 상태 변화1

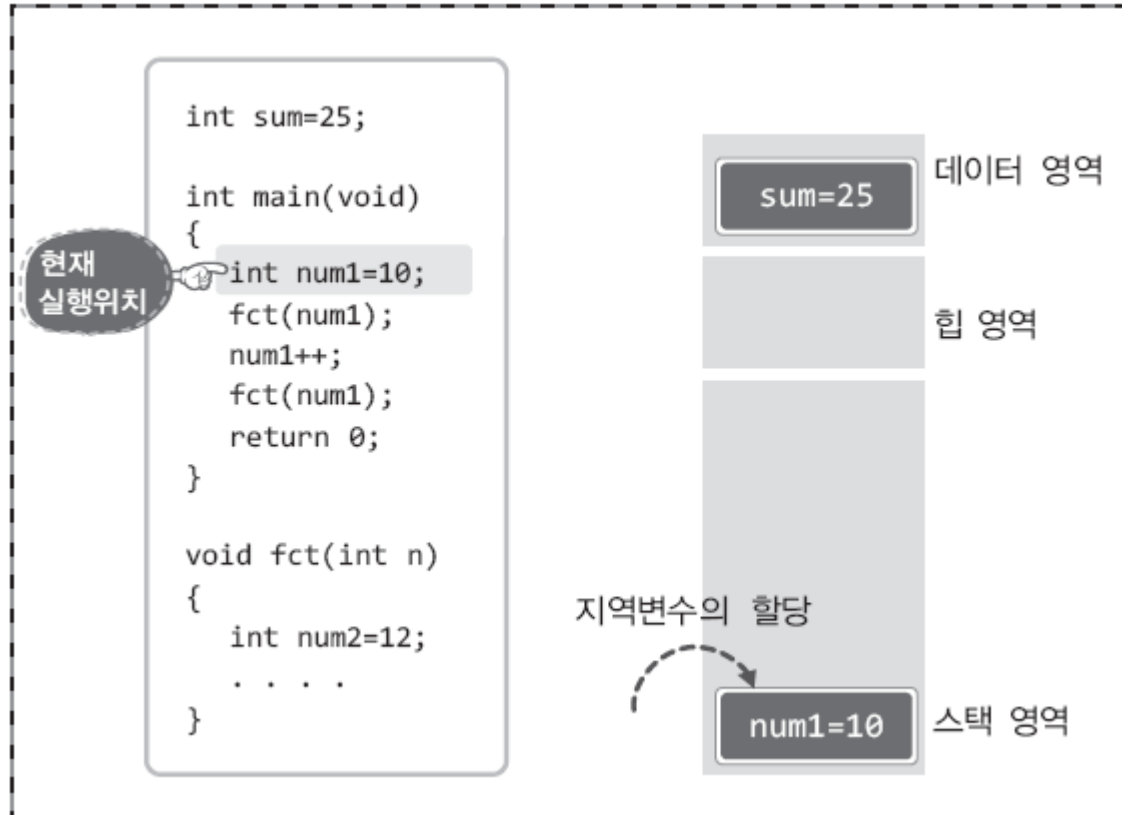


프로그램의 시작:

전역변수의 할당 및 초기화

실행의 흐름 /

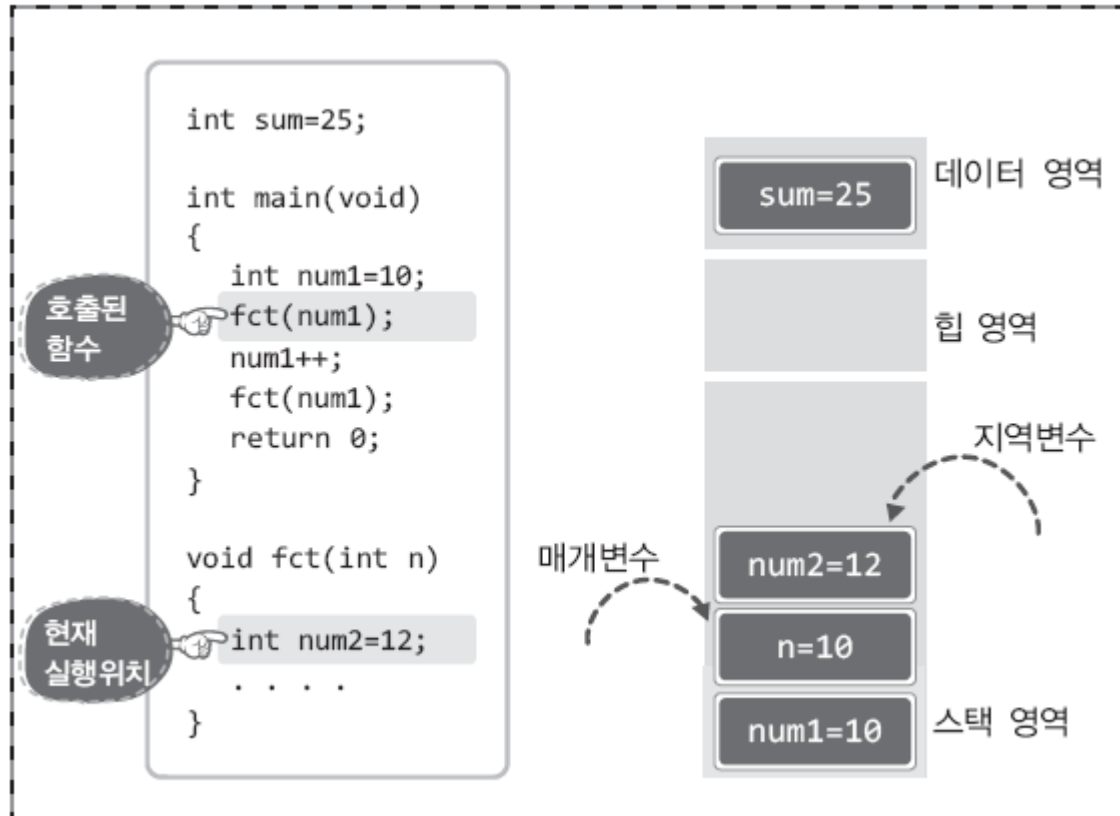
프로그램의 실행에 따른 메모리의 상태 변화2



main 함수의 호출 및 실행

실행의 흐름2

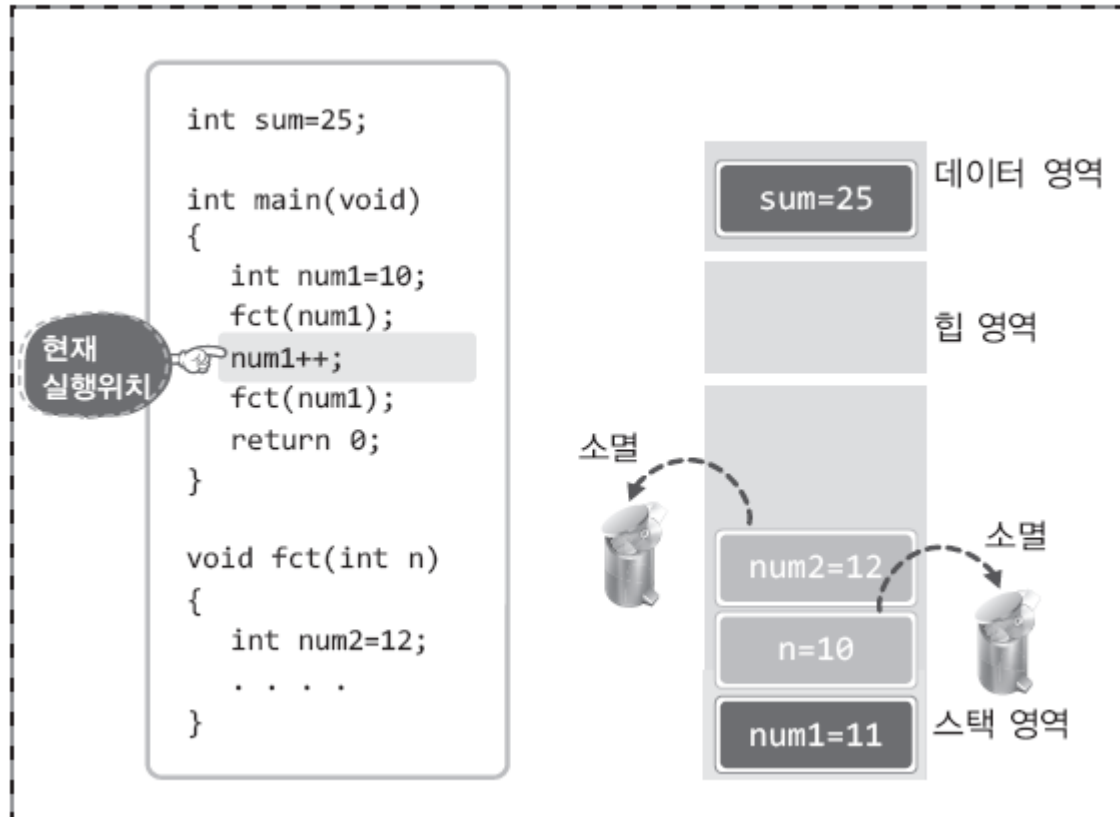
프로그램의 실행에 따른 메모리의 상태 변화3



fct 함수의 호출

실행의 흐름3

프로그램의 실행에 따른 메모리의 상태 변화4

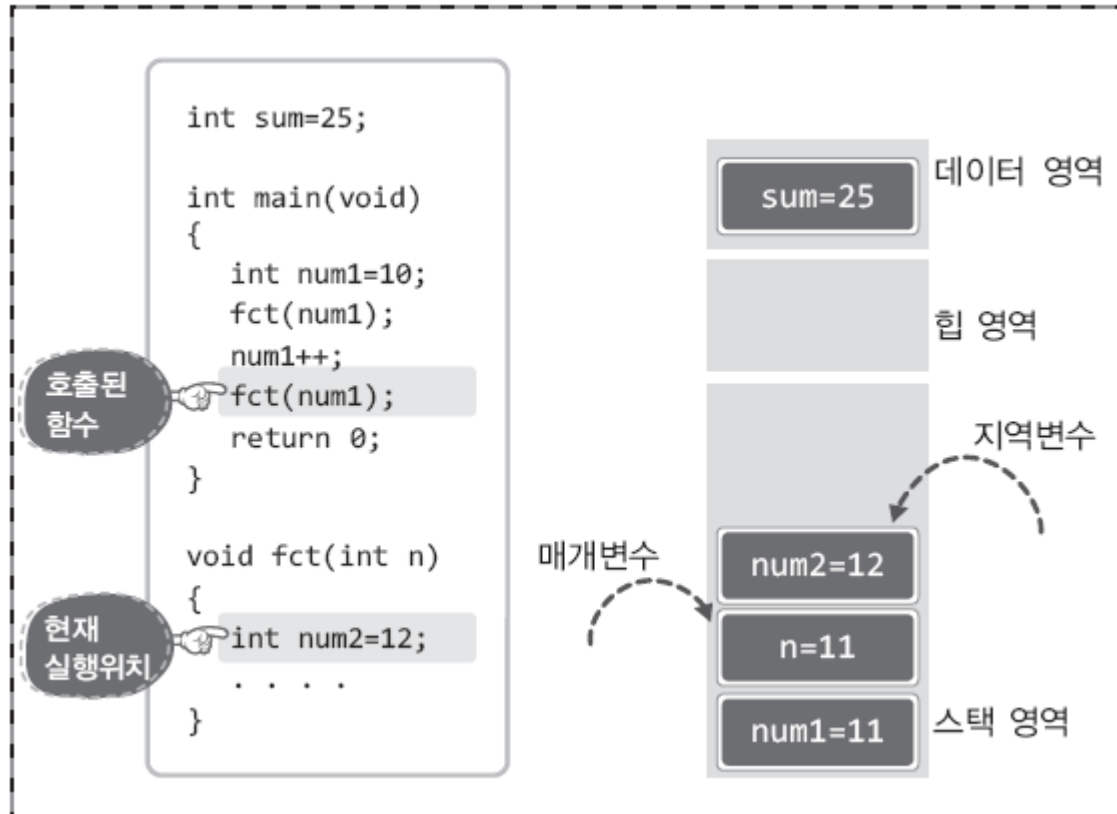


fct 함수의 반환

그리고 main 함수 이어서 실행

실행의 흐름4

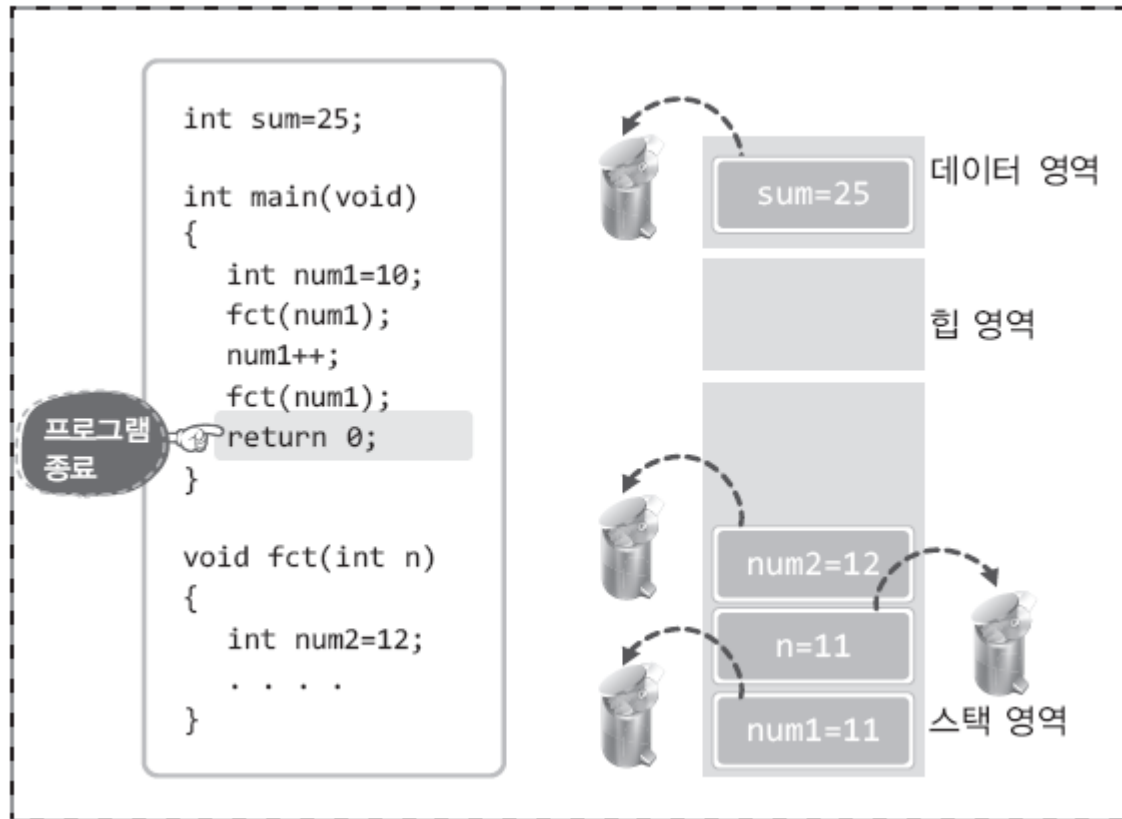
프로그램의 실행에 따른 메모리의 상태 변화5



fct 함수의 재호출 및 실행

실행의 흐름5

프로그램의 실행에 따른 메모리의 상태 변화6



fct 함수의 반환
및 *main* 함수의 반환

실행의 흐름6 (프로그램 종료)

함수의 호출순서가 `main → fct1 → fct2`이라면 스택의 반환은(지역변수의 소멸은) 그의 역순인 `fct2 → fct1 → main`으로 이루어진다는 특징을 기억하자!

윤성우의 열혈 C 프로그래밍



Chapter 25-2. 메모리의 동적 할당

윤성우 저 열혈강의 C 프로그래밍 개정판

전역변수와 지역변수로 해결이 되지 않는 상황

```
char * ReadUserName(void)
{
    char name[30];
    printf("What's your name? ");
    gets(name);
    return name; 무엇이 반환하는가?
}
```

```
int main(void)
{
    char * name1;
    char * name2;
    name1=ReadUserName();
    printf("name1: %s \n", name1);
    name2=ReadUserName();
    printf("name2: %s \n", name2);
    return 0;
}
```

변수 `name`은 `ReadUserName` 함수호출 시 할당이 되어야 하고,
`ReadUserName` 함수가 반환을 하더라도 계속해서 존재해야 한다.
 그런데 전역변수도 지역변수도 이러한 유형에는 부합하지 않는다!



혹시 전역변수가 답이 된다고 생각하는가?

```
char name[30];
char * ReadUserName(void)
{
    printf("What's your name? ");
    gets(name);
    return name;
}

int main(void)
{
    char * name1;
    char * name2;
    name1=ReadUserName();
    printf("name1: %s \n", name1);
    name2=ReadUserName();
    printf("name2: %s \n", name2);
    printf("name1: %s \n", name1);
    printf("name2: %s \n", name2);
    return 0;
}
```

전역변수는 답이 될 수 없음을 보이는 예제 및 실행결과

실행결과

```
What's your name? Yoon sung woo
name1: Yoon sung woo
What's your name? Choi jun kyung
name2: Choi jun kyung
name1: Choi jun kyung
name2: Choi jun kyung
```

힙 영역의 메모리 공간 할당과 해제

```
#include <stdlib.h>
void * malloc(size_t size);    // 힙 영역으로의 메모리 공간 할당
void free(void * ptr);        // 힙 영역에 할당된 메모리 공간 해제
```

➔ malloc 함수는 성공 시 할당된 메모리의 주소 값, 실패 시 NULL 반환

```
int main(void)
{
    void * ptr1 = malloc(4);    // 4바이트가 힙 영역에 할당
    void * ptr2 = malloc(12);   // 12바이트가 힙 영역에 할당
    . . . . .
    free(ptr1);                // ptr1이 가리키는 4바이트 메모리 공간 해제
    free(ptr2);                // ptr2가 가리키는 12바이트 메모리 공간 해제
    . . . . .
}
```

반환형이 void형 포인터임에 주목!

*malloc & free 함수 호출의
기본 모델*



malloc 함수의 반환형이 void형 포인터인 이유

```
void * ptr1 = malloc(sizeof(int));
void * ptr2 = malloc(sizeof(double));
void * ptr3 = malloc(sizeof(int)*7);
void * ptr4 = malloc(sizeof(double)*9);
```

malloc 함수의 일반적인 호출형태



```
void * ptr1 = malloc(4);
void * ptr2 = malloc(8);
void * ptr3 = malloc(28);
void * ptr4 = malloc(72);
```

sizeof 연산 이후 실질적인 malloc의 호출

malloc 함수는 인자로 숫자만 하나 전달받을 뿐이니 할당하는 메모리의 용도를 알지 못한다. 따라서 메모리의 포인터 형을 결정짓지 못한다. 따라서 다음과 같이 형 변환의 과정을 거쳐서 할당된 메모리의 주소 값을 저장해야 한다.

```
int * ptr1 = (int *)malloc(sizeof(int));
double * ptr2 = (double *)malloc(sizeof(double));
int * ptr3 = (int *)malloc(sizeof(int)*7);
double * ptr4 = (double *)malloc(sizeof(double)*9);
```

malloc 함수의
가장 모범적인 호출형태

힙 영역으로의 접근

```
int main(void)
{
    int * ptr1 = (int *)malloc(sizeof(int));
    int * ptr2 = (int *)malloc(sizeof(int)*7);
    int i;

    *ptr1 = 20;
    for(i=0; i<7; i++)
        ptr2[i]=i+1;

    printf("%d \n", *ptr1);
    for(i=0; i<7; i++)
        printf("%d ", ptr2[i]);

    free(ptr1);
    free(ptr2);
    return 0;
}
```

이렇듯 힙 영역으로의 접근은 포인터를
통해서만 이뤄진다.

```
int * ptr = (int *)malloc(sizeof(int));
if(ptr==NULL)
{
    // 메모리 할당 실패에 따른 오류의 처리
}
```

메모리 할당 실패 시 malloc 함수는 NULL을 반환

실행결과

```
20
1 2 3 4 5 6 7
```

‘동적 할당’이라 하는 이유!

컴파일 시 할당에 필요한 메모리 공간이 계산되지 않고, 실행 시 할당에 필요한 메모리 공간이
계산되므로!

free 함수를 호출하지 않으면?

- free 함수를 호출하지 않으면?

할당된 메모리 공간은 메모리라는 중요한 리소스를 계속 차지하게 된다.

- free 함수를 호출하지 않으면 프로그램 종료 후에도 메모리를 차지하는가?

프로그램이 종료되면 프로그램 실행 시 할당된 모든 자원이 반환된다.

- 꼭 free 함수를 호출해야 하는 이유는 무엇인가?

fopen 함수와 쌍을 이루어 fclose 함수를 호출하는 것과 유사하다.

- 예제에서 조차 늘 free 함수를 호출하는 이유는 습관을 들이기 위해서인가?

맞다! fopen, fclose가 늘 쌍을 이루듯 malloc, free도 쌍을 이루게 하자!



문자열 반환하는 함수를 정의하는 문제의 해결

```
char * ReadUserName(void)
{
    char * name = (char *)malloc(sizeof(char)*30);
    printf("What's your name? ");
    gets(name);
    return name;
}
```

할당!

ReadUserName 함수가 호출될 때마다 새로운 메모리 공간이 할당이 되고 이 메모리 공간은 함수를 빠져나간 후에도 소멸되지 않는다!

```
int main(void)
{
    char * name1;
    char * name2;
    name1=ReadUserName();
    printf("name1: %s \n", name1);
    name2=ReadUserName();
    printf("name2: %s \n", name2);

    printf("again name1: %s \n", name1);
    printf("again name2: %s \n", name2);
    free(name1);
    free(name2);
    return 0;
}
```

소멸!

```
What's your name? Yoon Sung Woo
name1: Yoon Sung Woo
What's your name? Hong Sook Jin
name2: Hong Sook Jin
again name1: Yoon Sung Woo
again name2: Hong Sook Jin
```

실행결과

calloc & realloc

```
#include <stdlib.h>
void * calloc(size_t elt_count, size_t elt_size);
```

➔ 성공 시 할당된 메모리의 주소 값, 실패 시 NULL 반환

malloc 함수와의 가장 큰 차이점은 메모리 할당을 위한 인자의 전달방식

$\text{elt_count} \times \text{elt_size}$ 크기의 바이트를 동적 할당한다. 즉, elt_size 크기의 블록을 elt_count 의 수만큼 동적할당! 그리고 *malloc* 함수와 달리 모든 비트를 0으로 초기화!

```
#include <stdlib.h>
void * realloc(void * ptr, size_t size);
```

➔ 성공 시 새로 할당된 메모리의 주소 값, 실패 시 NULL 반환

*ptr*이 가리키는 힙의 메모리 공간을 *size*의 크기로 늘리거나 줄인다!

malloc, *calloc*, *realloc* 함수호출을 통해서 할당된 메모리 공간은 모두 *free* 함수호출을 통해서 해제한다.



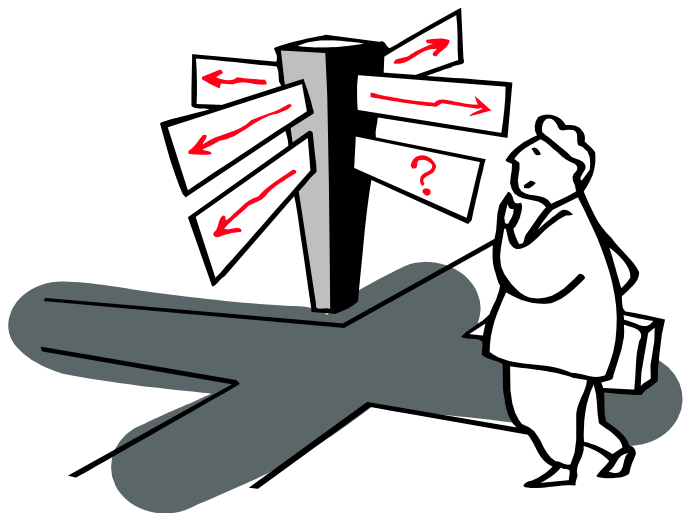
realloc 함수의 보충설명

```
int main(void)
{
    int * arr = (int *)malloc(sizeof(int)*3); // 길이가 3인 int형 배열 할당
    . . .
    arr = (int *)realloc(arr, sizeof(int)*5); // 길이가 5인 int형 배열로 확장
    . . .
}
```

- malloc 함수! 그리고 realloc 함수가 반환한 주소 값이 같은 경우
 - ➔ 기존에 할당된 메모리 공간을 이어서 확장할 여력이 되는 경우
- malloc 함수! 그리고 realloc 함수가 반환한 주소 값이 다른 경우
 - ➔ 기존에 할당된 메모리 공간을 이을 여력이 없어서 새로운 공간을 마련하는 경우

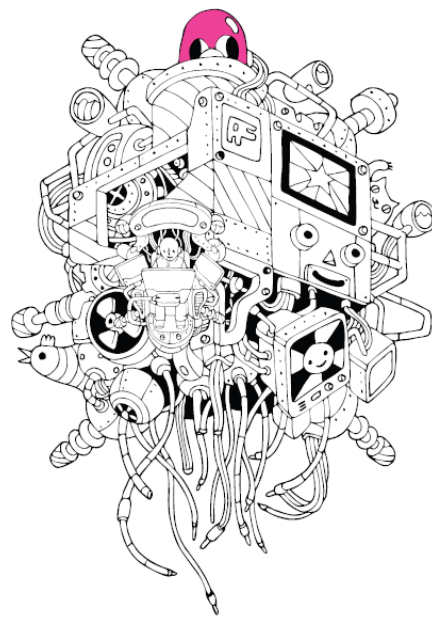
새로운 공간을 마련해야 하는 경우에는 메모리의 복사과정이 추가됨에 주목!





Chapter 25가 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 26. 매크로와 선행처리기

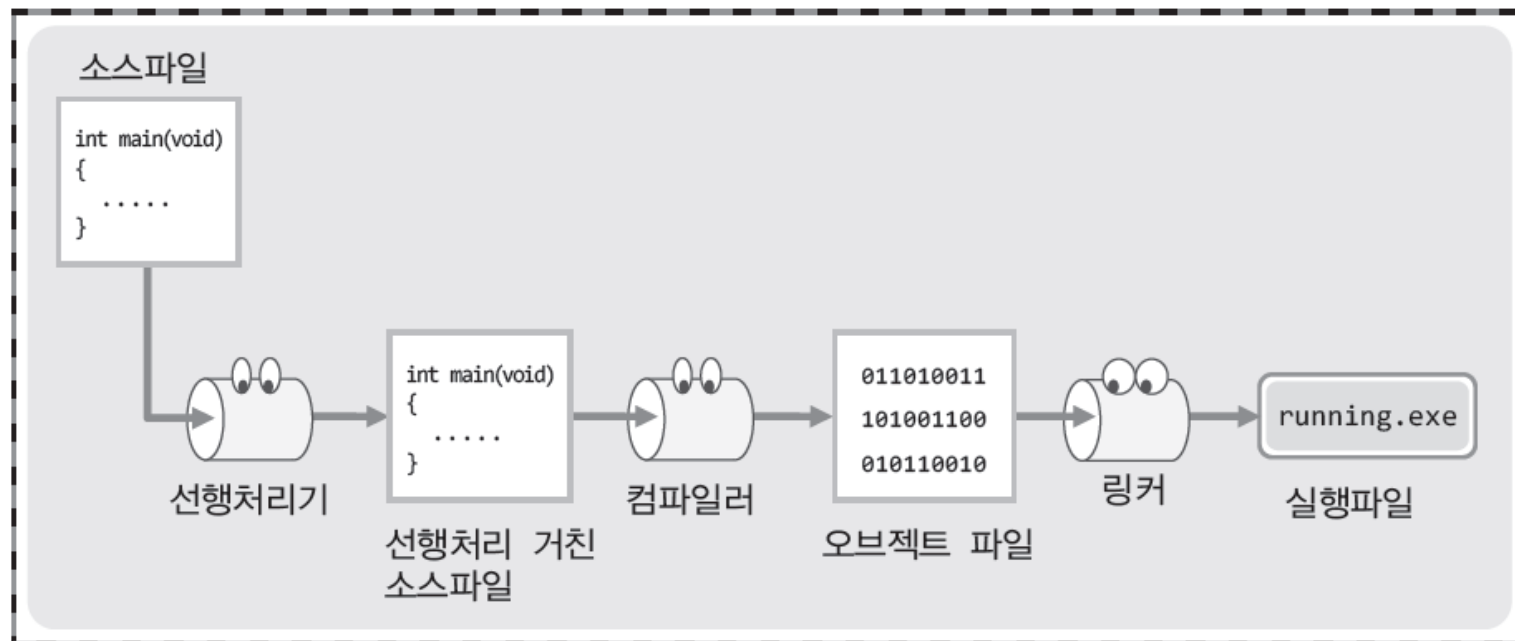
윤성우의 열혈 C 프로그래밍



Chapter 26-1. 선행처리기와 매크로

윤성우 저 열혈강의 C 프로그래밍 개정판

선행처리는 컴파일 이전의 처리를 의미합니다.

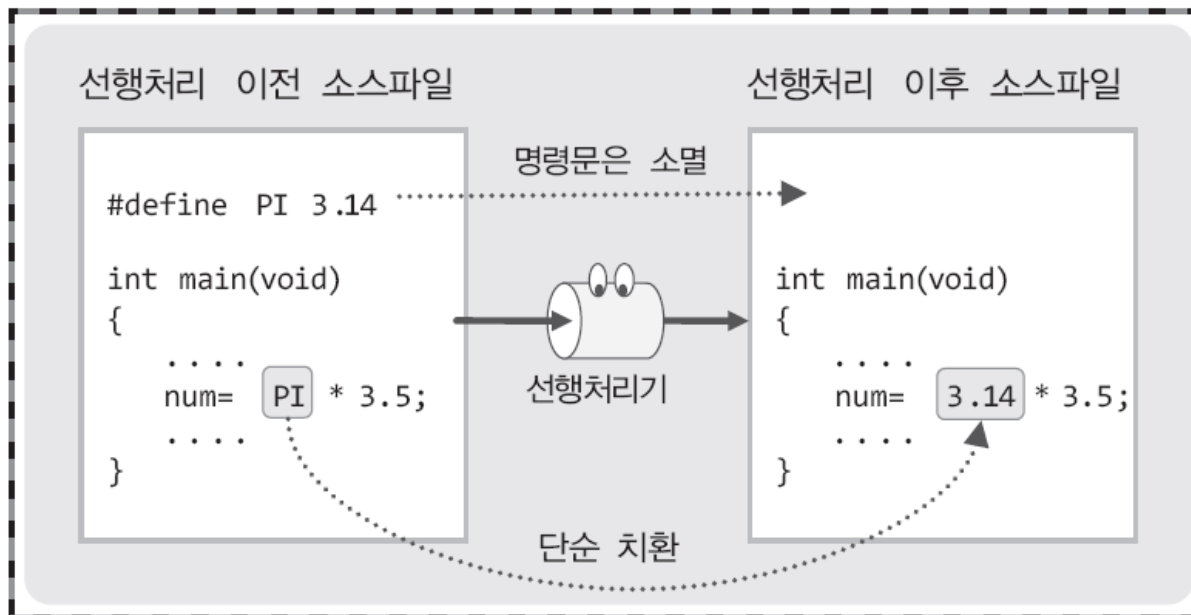


일반적으로 컴파일의 과정에 포함이 되어 이야기하지만, 선행처리의 과정과 컴파일의 과정은 구분이 된다.

선행처리기의 일 간단히 맛보기

#define PI 3.14 가 의미하는 바는 다음과 같다.

“PI를 3.14로 치환하라!” .



컴파일러에 비해서 선행처리기의 역할은 매우 간단하다. 쉽게 말해서 '단순한 치환'의 작업을 거친다. 그리고 선행처리기에게 무엇인가를 명령하는 문장은 #으로 시작한다.

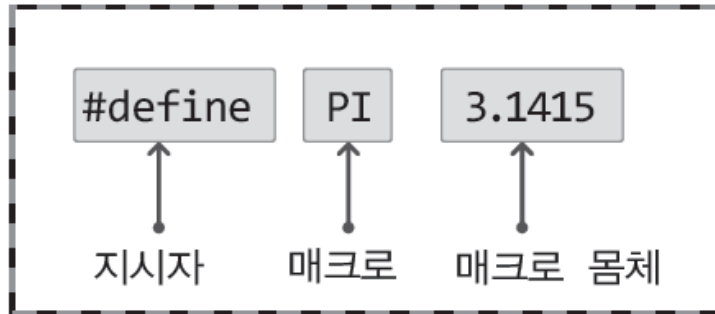
윤성우의 열혈 C 프로그래밍



Chapter 26-2. 대표적인 선행처리 명령문

윤성우 저 열혈강의 C 프로그래밍 개정판

#define: Object-like macro



PI를 3.1415로 무조건 치환하라!

```

#define NAME      "홍길동"
#define AGE       24
#define PRINT_ADDR puts("주소: 경기도 용인시\n");
int main(void)
{
    printf("이름: %s \n", NAME);
    printf("나이: %d \n", AGE);
    PRINT_ADDR;
    return 0;
}
    
```

```

printf("이름: %s \n", "홍길동");
printf("나이: %d \n", 24);
puts("주소: 경기도 용인시 \n");
    
```

치환된 결과물!

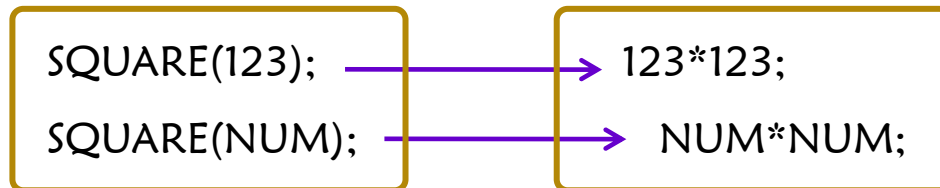
실제 컴파일 되는 문장들!

실행결과

```

이름: 홍길동
나이: 24
주소: 경기도 용인시
    
```

#define: Function-like macro



이러한 변환의 과정을 가리켜
'매크로 확장(macro expansion)'이라 한다.

SQUARE(X) 패턴에 부합!

따라서 치환!

매크로 확장의 예

```
#define SQUARE(X) X*X
int main(void)
{
    int num=20;

    /* 정상적 결과 출력 */
    printf("Square of num: %d \n", SQUARE(num));
    printf("Square of -5: %d \n", SQUARE(-5));
    printf("Square of 2.5: %g \n", SQUARE(2.5));

    /* 비정상적 결과 출력 */
    printf("Square of 3+2: %d \n", SQUARE(3+2));
    return 0;
}
```

```
Square of num: 400
Square of -5: 25
Square of 2.5: 6.25
Square of 3+2: 11
```

실행결과

3과 2의 합인 5가 SQUARE 함수의 인자가 되어, 25가 출력되는 것이 상식적이다! 그러나 출력결과는 다르다!

잘못된 매크로의 정의와 소괄호의 해결책

```
#define SQUARE(X)  X*X
```

SQUARE(3+2)	➔	3+2*3+2	➔	11	오류의 원인
SQUARE((3+2))	➔	(3+2)*(3+2)	➔	25	사용하기 불편

```
#define SQUARE(X)  (X)*(X)
```

SQUARE(3+2)	➔	(3+2)*(3+2)	➔	25	이 경우는 해결
num=120/SQUARE(2)	➔	120 / (2)*(2)			여전히 문제!

```
#define SQUARE(X)  ((X)*(X))
```

num=120/SQUARE(2)	➔	120 / ((2)*(2))			최상의 해결책!
-------------------	---	-----------------	--	--	----------

매크로를 두 줄에 걸쳐서 정의하는 방법

```
#define SQUARE(X)
((X)*(X))
```

이렇듯 두 줄에 걸쳐서 매크로를 정의하면 에러가 발생한다!

```
#define SQUARE(X) \
((X)*(X))
```

첫 번째 줄의 끝에 \을 삽입하면 에러가 발생하지 않는다.
이는 \이 매크로의 정의가 이어짐을 뜻하기 때문이다.



먼저 정의된 매크로의 사용

```
#define PI 3.14
#define PROUDCT(X, Y) ((X)*(Y))
#define CIRCLE_AREA(R) (PROUDCT((R), (R))*PI)

int main(void)
{
    double rad=2.1;
    printf("반지름 %g인 원의 넓이: %g \n", rad, CIRCLE_AREA(rad));
    return 0;
}
```

반지름 2.1인 원의 넓이: 13.8474

실행결과

위 예제에서 보이듯이, 앞 줄에서 먼저 정의된 매크로는 새로운 매크로를 정의하는데 있어서 사용될 수 있다.

일반함수와 비교한 매크로 함수의 장점

장점 1.

매크로 함수는 일반 함수에 비해 실행속도가 빠르다.

함수의 호출을 완성하기 위해서는 별도의 메모리 공간이 필요하고 호출된 함수로의 이동 및 반환의 과정을 거쳐야 한다. 반면 매크로 함수는 정의된 몸체로 치환이 이뤄지니 이러한 일이 불필요하며, 때문에 실행속도가 빨라질 수 밖에 없다.

장점 2.

자료형에 따라서 별도로 함수를 정의하지 않아도 된다.

전달되는 인자의 자료형에 구분을 받지 않으므로 자료형에 의존적이지 않다.



매크로 함수의 단점

단점

1. 정의하기가 정말로 까다롭다.
2. 디버깅하기가 쉽지 않다. *선행처리 후 컴파일러에 의해서 예러가 감지되므로*

매크로 함수로 정의하면?

```
int DiffABS(int a, int b)
{
    if(a>b)
        return a-b;
    else
        return b-a;
}
```

잘못 정의된 매크로

```
#define DIFF_ABS(X, Y)  ( (x)>(y) ? (x)-(y) : (y)-(x) )
int main(void)
{
    printf("두 값의 차: %d \n", DIFF_ABS(5, 7));
    printf("두 값의 차: %g \n", DIFF_ABS(1.8, -1.4));
    return 0;
}
```

그러나 컴파일러는 컴파일 된 이후의 내용을 기준으로 오류를 이야기한다.

함수를 매크로로 정의하기 위한 조건

조건 1.

작은 크기의 함수

한 두줄 정도 크기의 작은 함수가 아니면 매크로로 정의하는 것이 쉽지 않아서 오류발생 확률이 높아진다. 그리고 if~else, for와 같이 실행의 흐름을 컨트롤 하는 문장도 매크로로 정의하기 쉽지 않다.

조건 2.

호출의 빈도수가 높은 함수

함수를 매크로로 정의하는 데에는 성능적 측면이 고려된다. 그런데 호출의 빈도수가 높지 않으면 애써 매크로로 함수를 정의하는 수고를 할 이유가 줄어든다.



윤성우의 열혈 C 프로그래밍



Chapter 26-3. 조건부 컴파일을 위한 매크로

윤성우 저 열혈강의 C 프로그래밍 개정판

#if ... #endif : 참이라면

```
#define ADD 1
#define MIN 0

int main(void)
{
    int num1, num2;
    printf("두 개의 정수 입력: ");
    scanf("%d %d", &num1, &num2);
```

```
#if ADD    // ADD가 '참'이라면
    printf("%d + %d = %d \n", num1, num2, num1+num2);
#endif
```

```
#if MIN    // MIN이 '참'이라면
    printf("%d - %d = %d \n", num1, num2, num1-num2);
#endif
```

```
    return 0;
}
```

실행처리 과정에서 *ADD*가 '참' 이
면 ~*endif* 까지 컴파일 대상에 포함

실행처리 과정에서 *MIN*이 '참' 이
면 ~*endif* 까지 컴파일 대상에 포함

실행결과

```
두 개의 정수 입력: 5 4
5 + 4 = 9
```

#ifdef ... #endif : 정의되었다면

```
// #define ADD 1
#define MIN 0
int main(void)
{
    int num1, num2;
    printf("두 개의 정수 입력: ");
    scanf("%d %d", &num1, &num2);
```

ADD와 MIN의 정의 여부만 중요한 상황이라면 각각의 매크로에 값을 지정할 필요가 없다. 즉, 다음과 같이 정의해도 된다.

#define ADD

#define MIN

```
#ifdef ADD // 매크로 ADD가 정의되었다면
    printf("%d + %d = %d \n", num1, num2, num1+num2);
#endif
```

ADD가 정의되었다면 ~endif 까지 컴파일 대상에 포함

```
#ifdef MIN // 매크로 MIN이 정의되었다면
    printf("%d - %d = %d \n", num1, num2, num1-num2);
#endif
```

MIN이 정의되었다면 ~endif 까지 컴파일 대상에 포함

```
    return 0;
}
```

실행결과

```
두 개의 정수 입력: 7 2
7 - 2 = 5
```


#ifndef . . . #endif : 정의되지 않았다면

```
#ifndef ADD      ADD가 정의되지 않았다면 ~endif 까지 컴파일 대상에 포함
    printf("%d + %d = %d \n", num1, num2, num1+num2);
#endif
```

```
#ifndef MIN      MIN이 정의되지 않았다면 ~endif 까지 컴파일 대상에 포함
    printf("%d - %d = %d \n", num1, num2, num1-num2);
#endif
```



#else의 삽입: #if, #ifdef, #ifndef에 해당

```
#define HIT_NUM 5

int main(void)
{
    #if HIT_NUM==5
        puts("매크로 상수 HIT_NUM은 현재 5입니다.");
    #else
        puts("매크로 상수 HIT_NUM은 현재 5가 아닙니다.");
    #endif
    return 0;
}
```

포함 조건

매크로 `HIT_NUM`이 5가 아닐 경우 포함되는 문장

실행결과 매크로 상수 `HIT_NUM`은 현재 5입니다.

`#else`를 추가해서

조건이 ‘참’ 이 아닌 경우에 컴파일의 대상에 포함시킬 문장들을 구성할 수 있다.

#elif의 삽입: #if에만 해당

```
#define HIT_NUM 7

int main(void)
{
    #if HIT_NUM==5
        puts("매크로 상수 HIT_NUM은 현재 5입니다.");
    #elif HIT_NUM==6
        puts("매크로 상수 HIT_NUM은 현재 6입니다.");
    #elif HIT_NUM==7
        puts("매크로 상수 HIT_NUM은 현재 7입니다.");
    #else
        puts("매크로 상수 HIT_NUM은 5, 6, 7은 확실히 아닙니다.");
    #endif

    return 0;
}
```

조건에 따라서 이중에서 하나의 문장만 컴파일의 대상으로 포함
이 된다.

매크로 상수 HIT_NUM은 현재 7입니다.

실행결과

#elif와 #else를 추가해서 if ~ else if ~ else 구문과 동일한 형태를 구성할 수 있다.

윤성우의 열혈 C 프로그래밍



Chapter 26-4. 매개 변수의 결합과
문자열화

윤성우 저 열혈강의 C 프로그래밍 개정판

문자열 내에서는 매크로의 매개변수 치환 불가

[문자열 치환을 목적으로 정의된 매크로]

```
#define STRING_JOB(A, B)    "A의 직업은 B입니다."
```

치환의 대상이 되는 *A*와 *B*가 문자열 안에 존재함에 주목하자!



[매크로의 확장 결과]

STRING_JOB(이동춘, 나무꾼) ➡ "A의 직업은 B입니다."

STRING_JOB(한상순, 사냥꾼) ➡ "A의 직업은 B입니다."

이동춘과 나무꾼, 그리고 한상순과 사냥꾼이 기대대로 치환되지 않았다. 문자열 안에서는 치환이 일어나지 않기 때문이다. 이럴 때 고려해 볼 수 있는 연산자가 # 연산자이다!

문자열 내에서 매크로 매개변수 치환: # 연산자

[# 연산자를 이용해서 정의된 매크로]

```
#define STR(ABC) #ABC
```

매개변수 *ABC*에 전달되는 인자를 문자열 "*ABC*"로 치환해라!



[# 연산자 기반의 매크로 확장 결과]

STR(123)



"123"

STR(12, 23, 34)



"12, 23, 34"

연산자를 이용한 문제의 해결!

다음 문장 선언은

```
char * str="ABC" "DEF";
```

다음의 문장 선언과 같음과

```
char * str="ABCDEF"
```

연산자를 근거로 하여 앞서 제시한 문제를 해결한 결과

둘 이상의 문자열 선언을 나란히 하면, 이는 하나의 문자열 선언으로 인식이 된다.

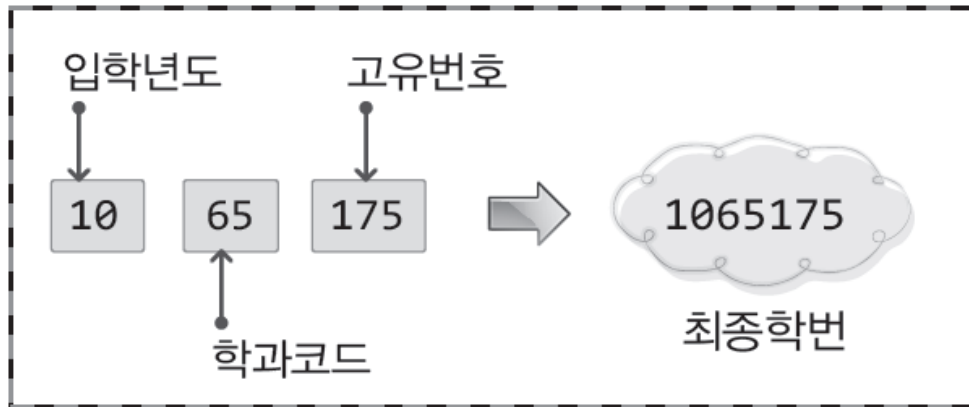


```
#define STRING_JOB(A, B)  #A "의 직업은 " #B "입니다."
int main(void)
{
    printf("%s \n", STRING_JOB(이동춘, 나무꾼));
    printf("%s \n", STRING_JOB(한상순, 사냥꾼));
    return 0;
}
```

실행결과

이동춘의 직업은 나무꾼입니다.
한상순의 직업은 사냥꾼입니다.

매크로 연산자 없이 단순 연결은 불가능



세 개의 숫자를 단순 연결하여 학번을 구성해야 한다고 가정해 보자. 그리고 이를 위한 매크로를 정의해 보자.

```
#define STNUM(Y, S, P)
```

YSP

치환 대상 YSP를 연결해 놓으면 이는 그냥 YSP로 인식이 된다.

```
#define STNUM(Y, S, P)
```

Y S P

치환은 제대로 이뤄지나 Y와 S와 P가 연결되어 있지 않아서 10 65 175 와 같이 치환이 이뤄진다.

```
#define STNUM(Y, S, P)
```

((Y)*100000+(S)*1000+(P))

연산자를 모르는 상태에서의 최선의 해결책!

단순 연결 관련 예제 기반 문제점 확인

```
// #define STNUM(Y, S, P)  YSP
// #define STNUM(Y, S, P)  Y S P
#define STNUM(Y, S, P)  ((Y)*100000+(S)*1000+(P))

int main(void)
{
    printf("학번: %d \n", STNUM(10, 65, 175));
    printf("학번: %d \n", STNUM(10, 65, 075));
    return 0;
}
```

1065175

1065061

실행결과

실행결과를 보면 1065075로 치환이 이뤄지지 않았음을 알 수 있다. 이는 075가 8진수로 해석된 결과이다. 따라서 이 경우에는 다음과 같이 문장을 구성해야 한다.

`printf("학번: %d \n", STNUM(10, 65, 75));`

위 예제에서 보이는 매크로도 좋은 해결책이 되지 못한다. 필요한 것은 '단순 연결'인데, 8진수 인식의 문제로 인해서 생각할 요소가 발생하기 때문이다! 이는 매크로 STNUM의 사용에 있어서 주의를 요하는 요소가 되므로 좋은 매크로 정의라 할 수 없다!

필요한 형태로 단순 결합: ## 연산자

[##연산자를 이용해서 정의된 매크로]

```
#define CON(UPP, LOW) UPP ## 00 ## LOW
```

매개변수 *UPP*와 *LOW*에 달되는 인자를 *UPP00LOW*로 단순히 연결해서 치환해라!



[## 연산자 기반의 매크로 확장 결과]

```
int num = CON(22, 77);
```

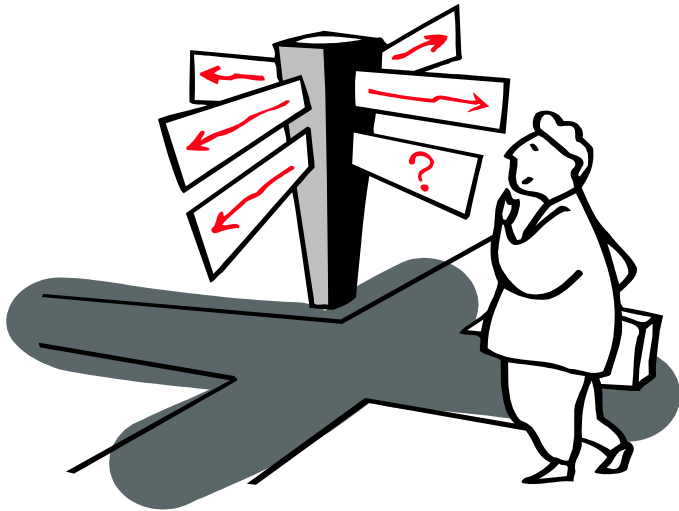


```
220077
```

이렇듯 ## 연산자를 이용해서 다음과 같이 매크로를 정의해야 학번을 단순치환 할 수 있다.

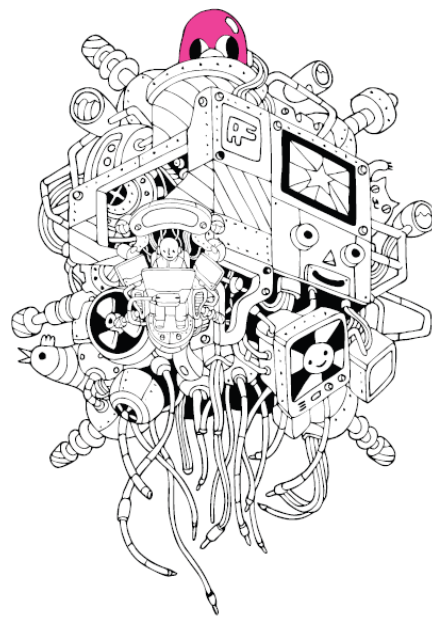
```
#define STNUM(Y, S, P) Y ## S ## P
```





Chapter 26이 끝났습니다. 질문 있으신지요?

윤성우의 열혈 C 프로그래밍



윤성우 저 열혈강의 C 프로그래밍 개정판

Chapter 27. 파일의 분할과 헤더파일의 디자인

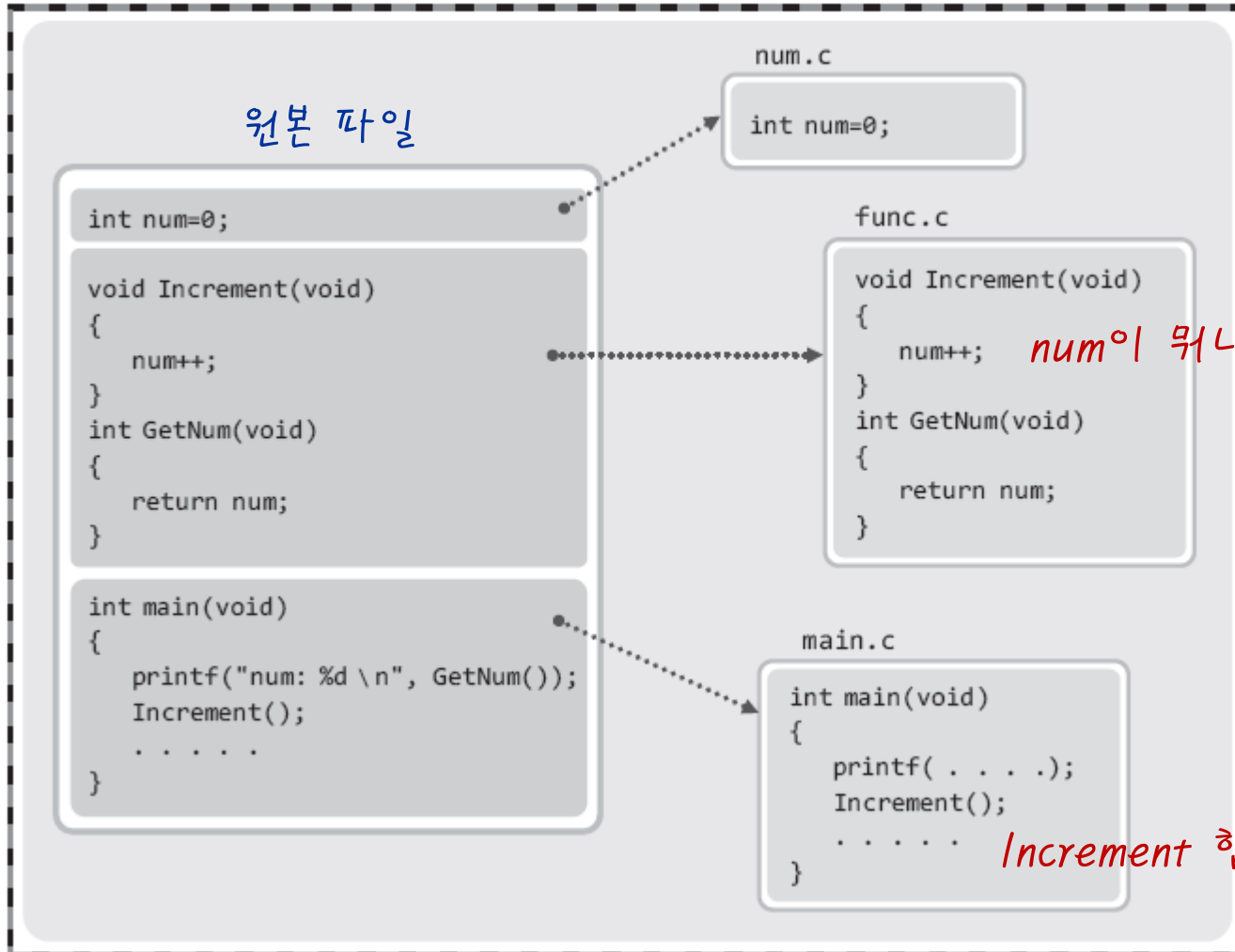
윤성우의 열혈 C 프로그래밍



Chapter 27-1. 파일의 분할

윤성우 저 열혈강의 C 프로그래밍 개정판

파일을 그냥 나눠도 될까요?



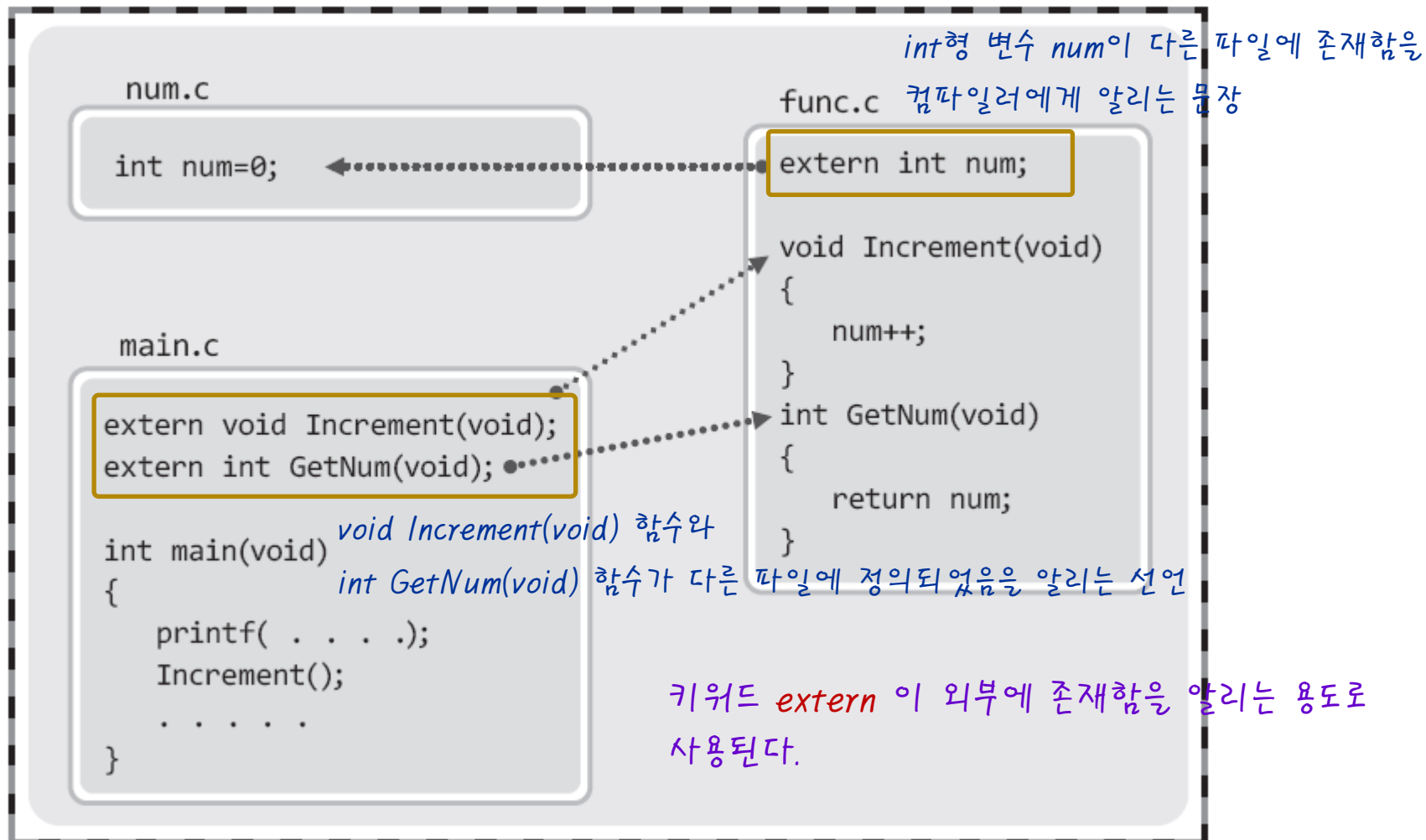
컴파일러는

파일 단위로 컴파일!

num이 뭐냐?

Increment 함수는 어디 있는 거야?

외부 선언 및 정의 사실을 컴파일러에게 알려줘야..



전역변수의 static 선언의 의미

지역변수로 선언되는 경우

```
void SimpleFunc(void)
{
    static int num=0;
    ....
}
```

함수 내에서만 접근이 가능한, 전역변수와 마찬가지로 한번 메모리 공간에 저장되면 종료 시까지 소멸되지 않고 유지되는 변수의 선언

전역변수로 선언되는 경우

```
static int num=0;

void SimpleFunc(void)
{
    ....
}
```

이 경우 int num은 전역변수이다. 단 외부 소스파일에서 접근이 불가능한 전역변수가 된다. 즉, 접근의 범위를 파일로 제한하게 된다.



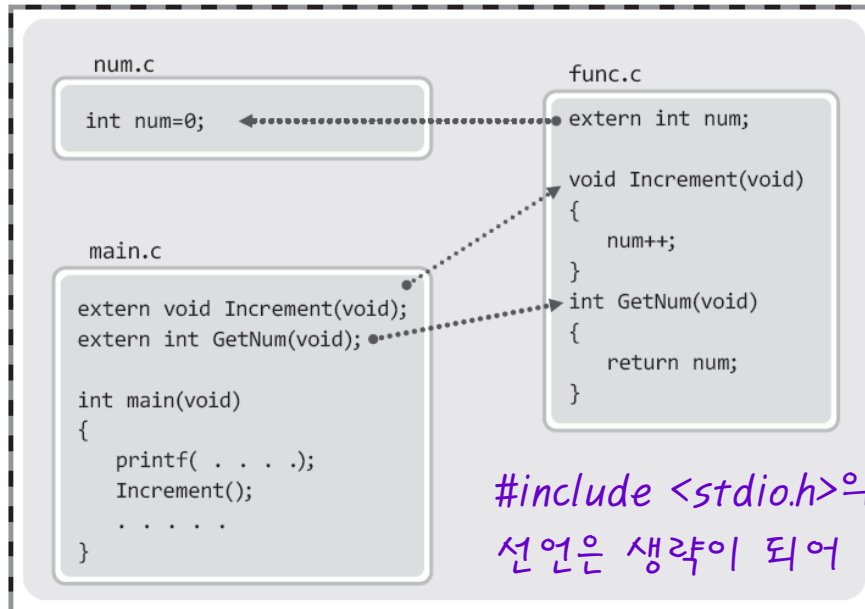
윤성우의 열혈 C 프로그래밍



Chapter 27-2. 둘 이상의 파일을 컴파일
하는 방법과 static에 대한 고찰

윤성우 저 열혈강의 C 프로그래밍 개정판

파일부터 정리하고 시작합니다.



이 세 개의 파일을 하나의 프로젝트 안에 담아서 하나의 실행파일을 생성해 보는 것이 목적!

다중파일 컴파일 방법 두 가지

- 첫 번째 방법
→ 파일을 먼저 생성해서 코드를 삽입한 다음에 프로젝트에 추가한다.
- 두 번째 방법
→ 프로젝트에 파일을 추가한 다음에 코드를 삽입한다.

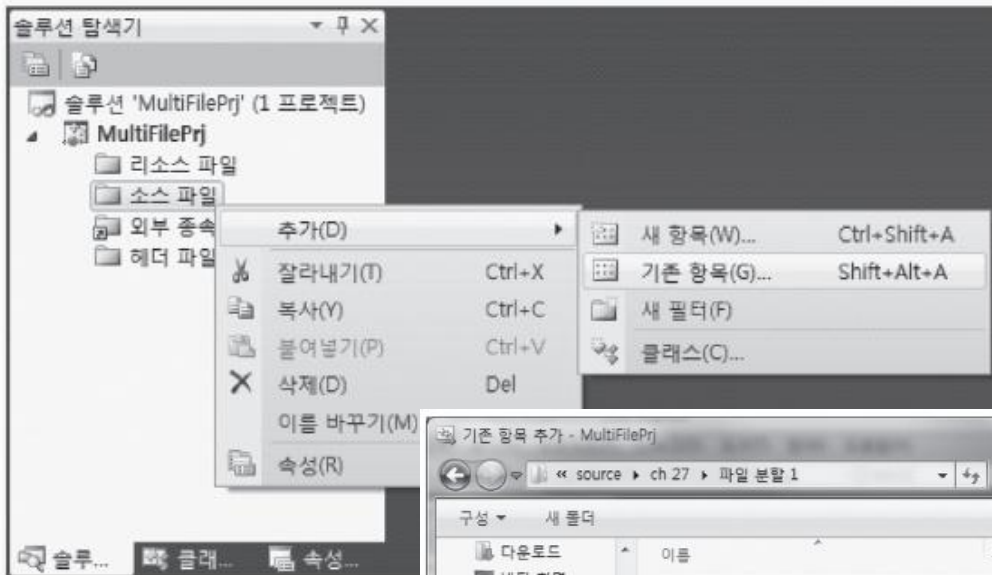
존재하는 파일, 프로젝트에 추가하는 방법

이미 존재하는 소스파일을 추가하는 방법

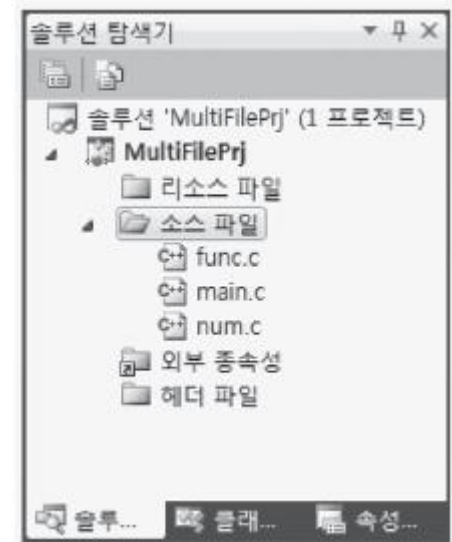
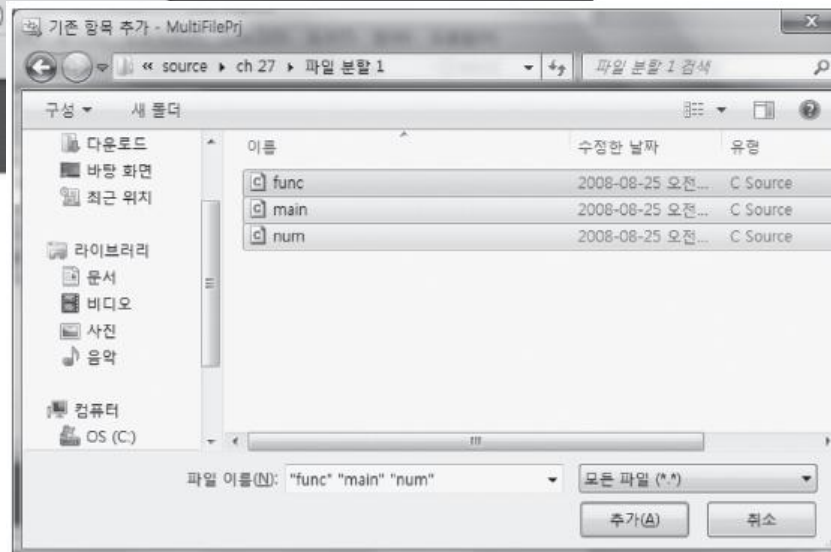
1단계

아래에서 보이듯이 다수의 파일이 하나의 프로젝트 안에 포함되었음이 솔루션 탐색기에 나타나야 한다.

추가결과 확인

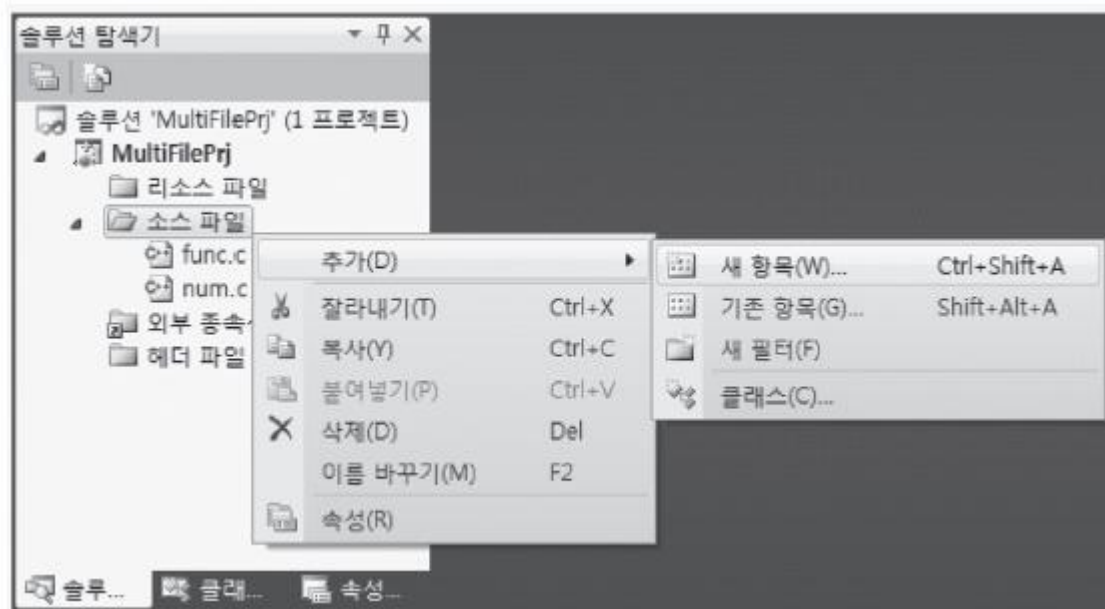


2단계



프로젝트에 새로운 파일을 추가하는 방법

새로운 소스파일을 만들어서 추가하는 방법



이는 기존에 해왔던, 소스파일을 새로 생성해서 프로젝트에 추가하는 방법과 100% 동일하다.
그 과정을 재차 진행하면 새로운 소스파일을 생성해서 프로젝트 내에 포함시킬 수 있다.

함수에도 static 선언을 할 수 있습니다.

함수를 대상으로 하는 *static* 선언

```
static void MinCnt(void)
{
    cnt--;
}
```

함수의 static 선언은 전역변수의 static 선언과 그 의미가 동일하다. 즉, 외부 소스파일에서의 접근을(호출을) 허용하지 않기 위한 선언이다.



윤성우의 열혈 C 프로그래밍



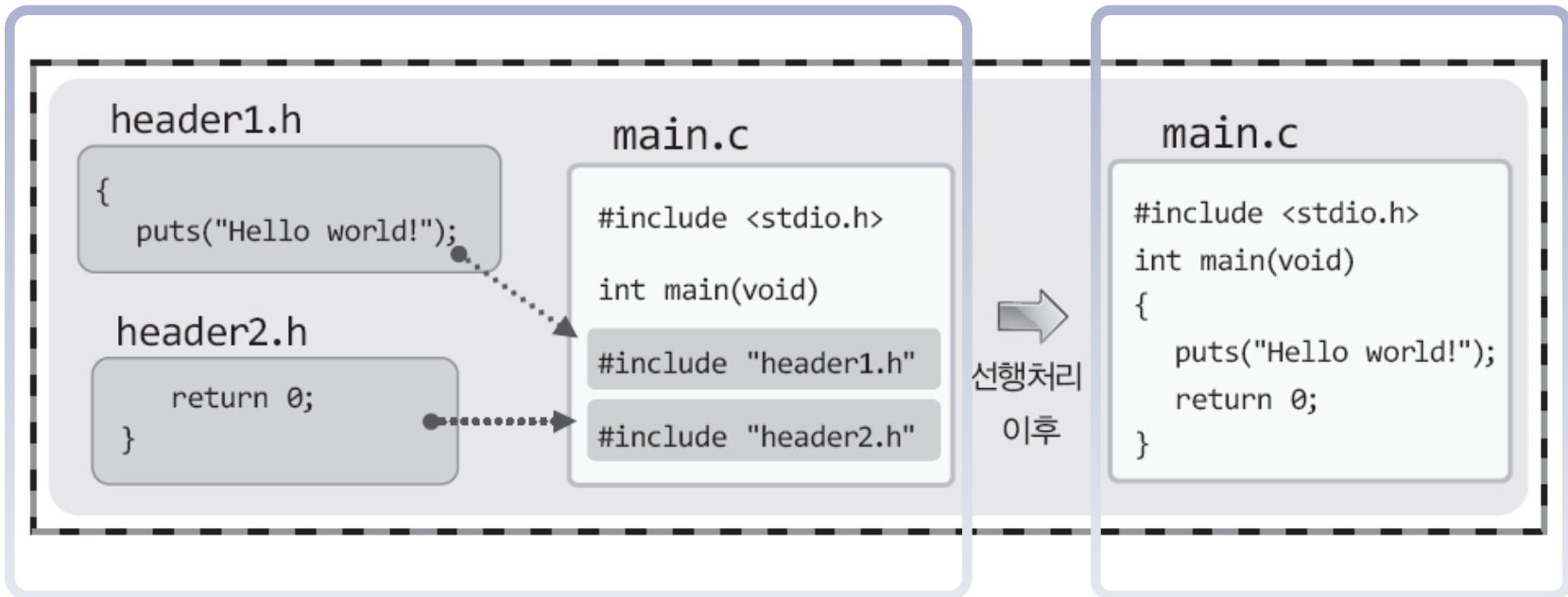
Chapter 27-3. 헤더파일의 디자인과 활용

윤성우 저 열혈강의 C 프로그래밍 개정판

#include 지시자와 헤더파일의 의미

두 개의 헤더파일과 하나의 소스파일로 이뤄진 프로젝트

선행처리 이후의 결과



위의 그림을 통해서 이해할 수 있듯이 #include 지시자는 헤더파일을 단순히 포함시키는 기능을 제공한다. 그리고 기본적으로 헤더파일에는 무엇이든 넣을 수 있다. 그러나 아무것이나 넣어서는 안 된다.

헤더파일을 include 하는 두 가지 방법

표준 헤더파일의 포함

```
#include <헤더파일 이름>
```

표준헤더 파일을 포함시킬 때 사용하는 방식이다. 표준헤더 파일이 저장된 디렉터리에서 헤더파일을 찾아서 포함을 시킨다.

프로그래머가 정의한 헤더파일의 포함

```
#include "헤더파일 이름"
```

프로그래머가 정의한 헤더파일을 포함시킬 때 사용하는 방식이다. 이 방식을 이용하면 이 문장을 포함하는 소스파일이 저장된 디렉터리에서 헤더파일을 찾게 된다.



절대경로의 지정과 그에 따른 단점

이렇듯 헤더파일의 경로를 명시할 수도 있다.

```
#include "C:\CPower\MyProject\header.h"
```

Windows의 절대경로 지정방식.

```
#include "/CPower/MyProject/header.h"
```

Linux의 절대경로 지정방식

- ▶ 절대경로를 지정하면 프로그램의 소스파일과 헤더파일을 임의의 위치로 이동시킬 수 없다(동일 운영체제를 기반으로 하더라도).
- ▶ 운영체제가 달라지면 디렉터리의 구조가 달라지기 때문에 경로지정에 대한 부분을 전면적으로 수정해야 한다.

상대경로의 지정 방법

상대경로 기반의 #include 선언

#include "header.h" 이 문장을 포함하는 소스파일이 저장된 디렉터리
:현재 디렉터리

#include "Release\header0.h" 현재 디렉터리의 서브인 Release 디렉터리

#include "..\CProg\header1.h" 현재 디렉터리의 상위 디렉터리의
서브인 Cprog 디렉터리

#include "..\..\MyHeader\header2.h" 현재 디렉터리의 상위 디렉터리의 상위
디렉터리의 서브인 MyHeader 디렉터리

위와 같은 형태로(상대경로의 지정방식을 기반으로) 헤더파일 경로를 명시하면 프로그램의 소스코드가 저장되어 있는 디렉터리를 통째로 이동하는 경우 어디서든 컴파일 및 실행이 가능해진다.

헤더파일에 무엇을 담으면 좋겠습니까?

헤더파일에 삽입이 되는 가장 일반적인 선언의 유형

```
extern int num;  
extern int GetNum(void);    // extern 생략 가능
```

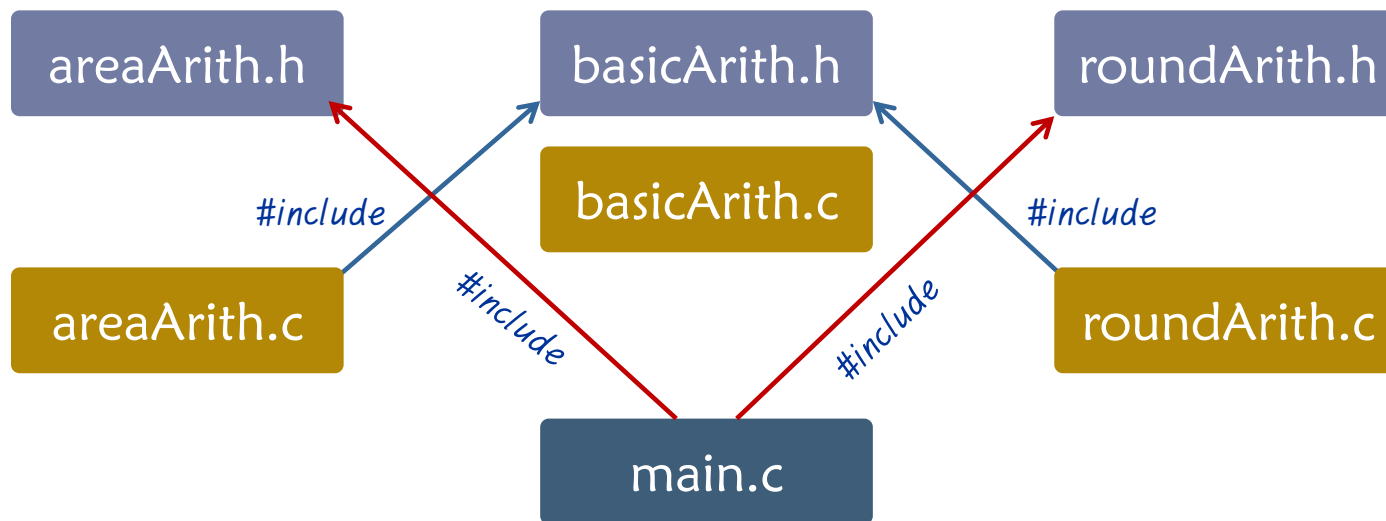
- ▶ 총 7개의 소스파일과 헤더파일로 이뤄진 예제를 통해서 다음 두 가지에 대한 정보를 얻자!
 - 소스파일을 나누는 기준
 - 헤더파일을 나누는 기준 및 정의의 형태
- ▶ 예제의 소스파일과 헤더파일의 구성
 - basicArith.h basicArith.c
 - areaArith.h areaArith.c
 - roundArith.h roundArith.c
 - main.c



헤더파일과 소스파일의 포함관계

▶ 예제의 소스파일과 헤더파일의 구성 및 내용

- basicArith.h basicArith.c → 수학과 관련된 기본적인 연산의 함수의 정의 및 선언
- areaArith.h areaArith.c → 넓이계산과 관련된 함수의 정의 및 선언
- roundArith.h roundArith.c → 둘레계산과 관련된 함수의 정의 및 선언
- main.c



basicArith.h & basicArith.c

basicArith.h : 기본연산 함수의 선언

```
#define PI 3.1415
double Add(double num1, double num2);
double Min(double num1, double num2);
double Mul(double num1, double num2);
double Div(double num1, double num2);
```

매크로의 정의는 파일단위로 유효하다.
그래서 PI와 같은 상수의 선언은 헤더파일에 정의하고, 이를 필요한 모든 소스파일이 PI가 선언된 헤더파일을 포함하는 형태를 띈다.

basicArith.c : 기본연산 함수의 정의

```
double Add(double num1, double num2)
{
    return num1+num2;
}

double Min(double num1, double num2)
{
    return num1-num2;
}

double Mul(double num1, double num2)
{
    return num1*num2;
}

double Div(double num1, double num2)
{
    return num1/num2;
}
```



areaArith.h & areaArith.c

```
double TriangleArea(double base, double height);  
double CircleArea(double rad);
```

areaArith.h : 넓이 계산 함수의 선언

```
#include "basicArith.h"
```

```
double TriangleArea(double base, double height)  
{  
    return Div(Mul(base, height), 2);  
}  
  
double CircleArea(double rad)  
{  
    return Mul(Mul(rad, rad), PI);  
}
```

areaArith.c : 넓이 계산 함수의 정의



roundArith.h & roundArith.c

```
double RectangleRound(double base, double height);  
double SquareRound(double side);
```

roundArith.h : 둘레계산 함수의 선언

```
#include "basicArith.h"
```

```
double RectangleRound(double base, double height)  
{  
    return Mul(Add(base, height), 2);  
}  
double SquareRound(double side)  
{  
    return Mul(side, 4);  
}
```

roundArith.c : 둘레계산 함수의 정의



main.c

```
#include <stdio.h>
#include "areaArith.h"
#include "roundArith.h"

int main(void)
{
    printf("삼각형 넓이(밑변 4, 높이 2): %g \n",
        TriangleArea(4, 2));
    printf("원 넓이(반지름 3): %g \n",
        CircleArea(3));

    printf("직사각형 둘레(밑변 2.5, 높이 5.2): %g \n",
        RectangleRound(2.5, 5.2));
    printf("정사각형 둘레(변의 길이 3): %g \n",
        SquareRound(3));

    return 0;
}
```

실행결과

```
삼각형 넓이(밑변 4, 높이 2): 4
원 넓이(반지름 3): 28.2735
직사각형 둘레(밑변 2.5, 높이 5.2): 15.4
정사각형 둘레(변의 길이 3): 12
```


구조체의 정의는 어디에? : 문제의 제시

구조체의 정의도 파일 단위로만 그 선언이 유효하다. 따라서 필요하다면 동일한 구조체의 정의를 소스파일마다 추가시켜 줘야 한다.

소스파일 *intdiv.c*

```
typedef struct div
{
    int quotient;    // 몫
    int remainder;   // 나머지
} Div;
```

```
Div IntDiv(int num1, int num2)
{
    Div dval;
    dval.quotient=num1/num2;
    dval.remainder=num1%num2;
    return dval;
}
```

소스파일 *main.c*

```
typedef struct div
{
    int quotient;    // 몫
    int remainder;   // 나머지
} Div;
```

```
extern Div IntDiv(int num1, int num2);

int main(void)
{
    Div val=IntDiv(5, 2);
    printf("몫: %d \n", val.quotient);
    printf("나머지: %d \n", val.remainder);
    return 0;
}
```

같은 구조체 정의를 둘 이상의 소스파일에 직접 추가시킨다는 것 자체가 부담!

구조체의 정의는 어디에? : 해결책의 제시

헤더파일 *stdiv.h*

```
typedef struct div
{
    int quotient;    // 몫
    int remainder;   // 나머지
} Div;
```

#include

```
#include "stdiv.h"
Div IntDiv(int num1, int num2)
{
    Div dval;
    dval.quotient=num1/num2;
    dval.remainder=num1%num2;
    return dval;
}
```

소스파일 *intdiv2.c*

구조체의 정의도 헤더파일에 넣고두고

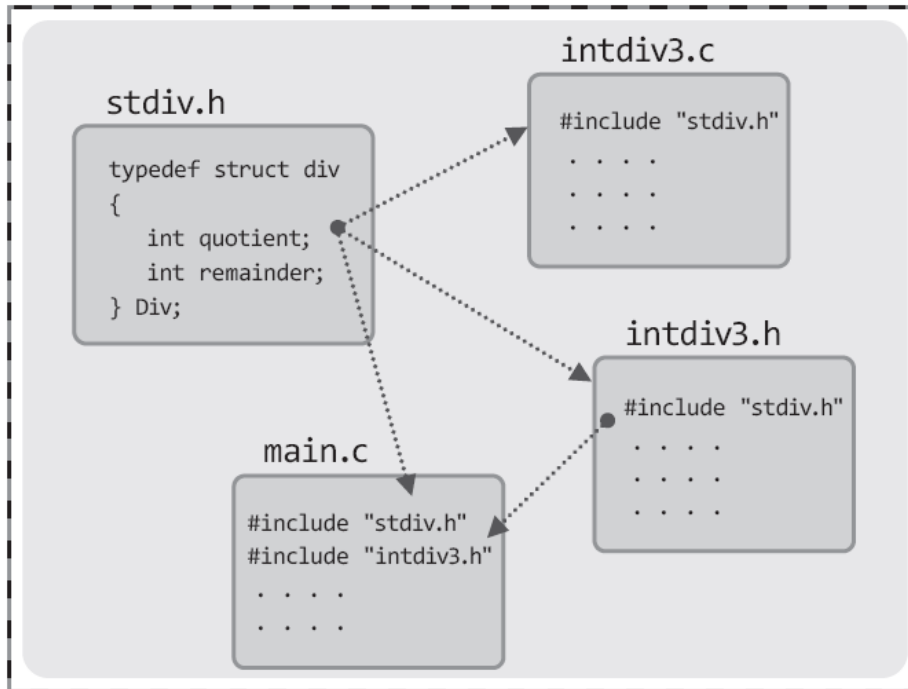
필요할 때마다 include 하는 것이 일반적이다!

#include

```
#include "stdiv.h"
extern Div IntDiv(int num1, int num2);
int main(void)
{
    Div val=IntDiv(5, 2);
    printf("몫: %d \n", val.quotient);
    printf("나머지: %d \n", val.remainder);
    return 0;
}
```

소스파일 *main.c*

헤더파일의 중복삽입 문제



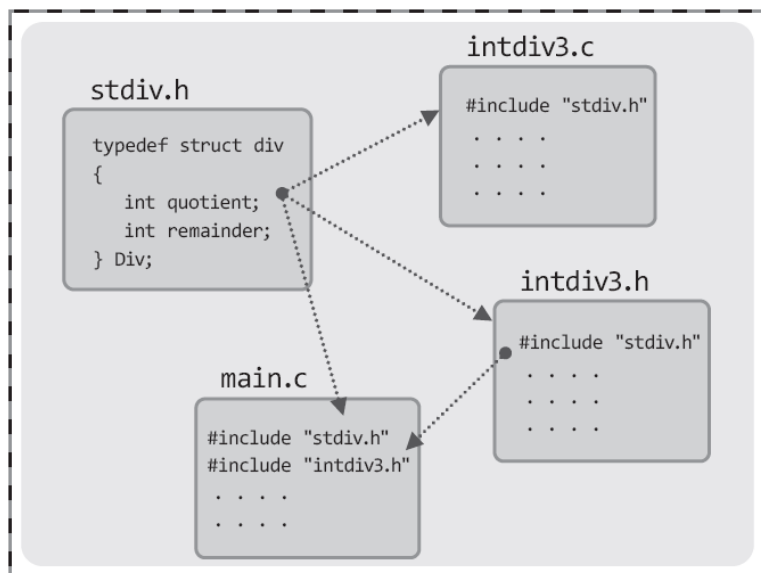
헤더파일을 직접적으로 또는 간접적으로 두 번 이상 포함하는 것 자체는 문제가 아니다. 그러나 두 번 이상 포함시킨 헤더파일의 내용에 따라서 문제가 될 수 있다.

일반적으로 선언(예로 함수의 선언)은 두 번 이상 포함시켜도 문제되지 않는다. 그러나 정의(예로 구조체 및 함수의 정의)는 두 번 이상 포함시키면 문제가 된다.

main.c는 결과적으로 구조체 Div의 정의를 두 번 포함하는 꼴이 된다! 그런데 구조체의 정의는 하나의 소스 파일 내에서 중복될 수 없다!

조건부 컴파일을 활용한 중복삽입 문제의 해결

중복 삽입 문제의 해결책



```

#ifndef __STDIV2_H__
#define __STDIV2_H__

typedef struct div
{
    int quotient;    // 몫
    int remainder;  // 나머지
} Div;

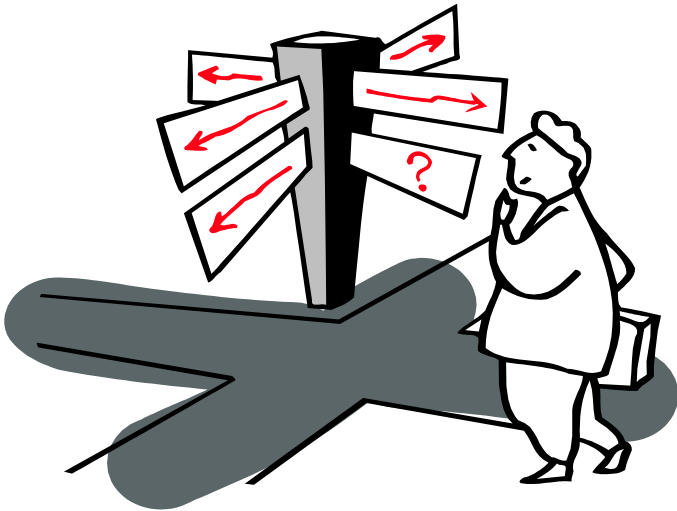
#endif
    
```

매크로 `__STDIV2_H__` 와 `#ifndef`의 효과가 `main.c`에서 어떻게 나타나는지 그려보자!

위와 같은 이유로 모든 헤더파일은 `#ifndef~#endif`로 감싸는 것이 안전하고 또 일반적이다!

강의가 끝났습니다.

'윤성우의 열혈 C 프로그래밍'을 사랑해 주신 여러분께 진심으로 감사드립니다.



Chapter 27이 끝났습니다. 질문 있으신지요?