

(Code) 기프 헷갈릴 만한 것 정리

#시험준비

- main은 return 0; return ; 둘다 가능
- 역슬래쉬 :\ , 큰따옴표 : ", 작은따옴표 : '
- 정수형은 %d, %i 둘다 가능하며 음수 표현이 가능하다.
- %g, %G는 소수점 이하 자리 수에 따라 %f, %e 둘중 하나를 선택한다.
- **%03d : 필드 폭 3칸 확보하고 오른쪽 정렬, 남은 자리는 0으로 채움
- %-3d : 필드 폭 3칸확보하고 왼쪽 정렬
- %+3d : 필드 폭 3칸 확보하고 오른쪽 정렬 (양수는 +, 음수는 - 출력)
- %8.2f : 총8칸 확보 후 소수점 2 이하 자리
- %10.4f :총 10칸 확보, 소수점 4 이하 자리
- 10% 할인 :: * 0.1
- 30% 할인 :: * 0.3
-
- 변수 선언은 제일 앞쪽에 한다.
- 변수 선언 시 맨 처음 숫자사용 안된다.
- 키워드를 변수명으로 사용 안된다.
- 변수는 대소문자 구분한다.
- const int a = 10;
- #define MAX 100
- #define 에서는 자료형과 세미콜론 사용 X
- 진법 상수
 - 10진수 : int dec = 10;
 - 16진수 : int hex = 0x10;
 - 8진수 : int oct = 010;
- CPU가 int형을 가장 빠르게 처리하는 이유
 - 대부분의 컴퓨터들은 32비트 이상의 시스템
 - CPU가 연산하는 기본 단위가 최소 32비트
- CPU는 문자를 인식하지 못하고 ASCII 코드를 참조해서 인식한다.
- 함수 종류
 - 표준 라이브러리 함수 : C에서 제공 (stdio.h, limits.h, float.h 등)
 - 사용자 정의 라이브러리 함수 : 사용자가 만듬
- 함수 장점
 - 안정성

- 재사용성
 - 에러수정 용이
 - 응집력 높고 복잡성 떨어지게 됨
 - **재귀함수는 시간과 메모리 공간의 효율 저하**
 - 함수는 호출하려고 하는 블록보다 위에 있어야 함.
 - 만약 위에 없는 경우 main위에 함수 선언 해주기
-
-

2진수 출력

```
int bi1 = 0b1010;  
int bi2 = 0b1111;  
  
printf("%d\n", bi1);  
printf("%d\n", bi2);
```

Note

2진수 표현 방식은 없지만, 16진수나 8진수를 앞에 써주고 2진수 쓴 후, %d로 출력하면 10진수로 값이 나옴
0b를 2진수 앞에 붙여야함!

지수표기

```
// 지수 표기  
double d1 = 3.14;  
double d2 = 3.14e2;  
double d3 = 3.14e-2;  
  
printf("d1 = %f\n", d1);  
printf("d2 = %f\n", d2);  
printf("d3 = %f\n", d3);
```

Note

지수 표기 법 !!

e2 :: 3.14 10^2

e-2 :: 3.14 10^-2

매크로 함수 만들기

```
#include <stdio.h>

#define PI 3.14
#define CIRCLE_AREA(r) (PI * (r) * (r))
#define CIRCLE_CIRCUM(r) (PI * 2 * (r))

int main(void){

    double radius = 2.0;
    printf("반지름 %.2f인 원의 넓이 : %.2f\n", radius, CIRCLE_AREA(radius));
    printf("반지름 %.2f인 원의 둘레 : %.2f\n", radius, CIRCLE_CIRCUM(radius));

    return 0;
}
```

변수의 주소값 알기

```
#include <stdio.h>
int main(void){

    int a =3;
    int b= 4;

    printf(" 변수 a의 시작 주소 : %x\n", &a);
    printf("변수 b의 시작 주소 : %x\n", &b);

    return 0;
}
```

Summary

변수의 주소를 알 때는 %x, & 연산자를 사용한다.

각진수 입력받아 10진수로 출력하기

```
#include
int main(void) {
printf("10진수 정수형 상수 %d + %d = %d 입니다. \n", 10, 20, 10+20);
printf("16진수 정수형 상수 %x + %x = %x 입니다. \n", 0x10, 0x20, 0x10+0x20);
printf(" 8진수 정수형 상수 %o + %o = %o 입니다. \n", 010, 020, 010+020);
```

```
return 0;
}
```

10진수 정수형 상수 10 + 20 = 30

16진수 정수형 상수 10 + 20 = 30

8진수 정수형 상수 10 + 20 = 30

```
#include <stdio.h>
int main(void){
printf("10진수 :: %d + %d = %d\n", 10, 20, 30);
printf("16진수 :: %d + %d = %d\n", 0x10, 0x20, 0x10 + 0x20);
printf("8진수 :: %d + %d = %d\n", 010, 020, 010 + 020);
return 0;
}
```

10지수 :: 10 + 20 = 30

16진수 :: 16 + 288 = 48

8진수 :: 8 + 16 = 24

Summary

출력값은 진수에 맞춰서 했지만, 실제 출력되는 값은 10진수로 나온다.

출력값을 모두 %d로 했을 경우, 해당 진법에 따른 계산이 10진수로 나오게 된다.

차이점 확인 !

Impoartant

아스키코드

A : 65

a : 97

0 : 48

```
#include <stdio.h>

#define PI 3.14
#define NUM 100
#define BUFFER_SIZE 200

int main(void) {

    NUM = 200; // 불가능
    PI = 4.14; // 불가능
    BUFFER_SIZE = 300; // 불가능

    printf("NUM : %d\n", NUM);
    printf("PI : %.2lf\n", PI);
    printf("BUFFER_SIZE : %d\n", BUFFER_SIZE);

    return 0;
}
```

```
#include <stdio.h>
int main(void) {

    const int NUM = 100;
    const double PI = 3.14;

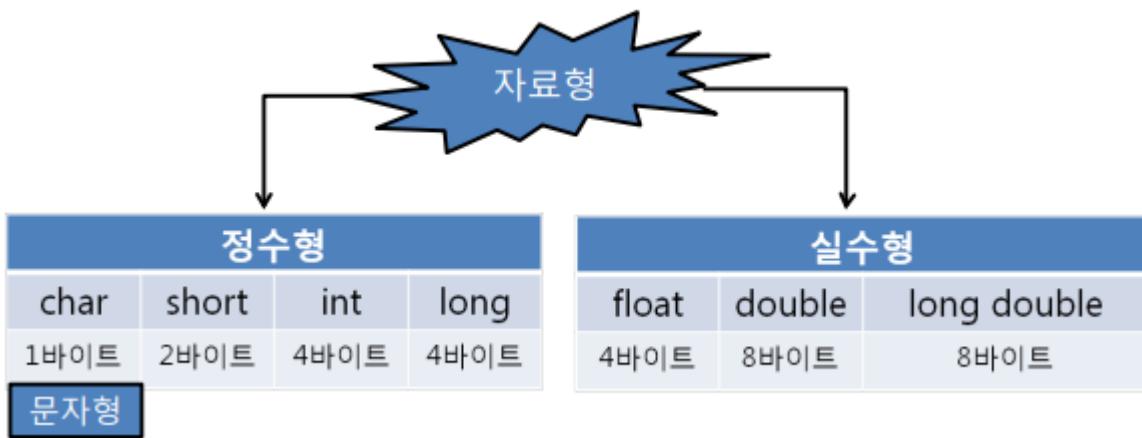
    NUM = 200; // 불가능
    PI = 4.14; // 불가능

    printf("NUM : %d\n", NUM);
    printf("PI : %.2lf\n", PI);
```

```
    return 0;  
}
```

Summary

#define과 const 둘 다 중간에 값 변경 불가능



Summary

사이즈 확인하는 방법

`sizeof(float)`

`sizeof(변수명) > sizeof(num1)`

long :: 4byte !!

정수형 최대 최소 확인

```
#include <stdio.h>  
#include <limits.h>  
  
int main(void) {  
  
    printf("char의 최솟값 %d, 최댓값 %d\n", CHAR_MIN, CHAR_MAX);  
    printf("short의 최솟값 %d, 최댓값 %d\n", SHRT_MIN, SHRT_MAX);  
    printf("int의 최솟값 %d, 최댓값 %d\n", INT_MIN, INT_MAX);
```

```

printf("long의 최솟값 %ld, 최댓값 %ld\n", LONG_MIN, LONG_MAX);

signed char num1 = 130; // -128 ~ 127 데이터 표현
unsigned char num2 = 130; // 0~127 + 128 = 0~256 데이터 표현

printf("signed char num1 :: %d \n", num1); // -- (a)
printf("unsigned char num2 :: %d \n", num2); //--(b)

printf("unsigned char의 최댓값 %u\n", UCHAR_MAX);
printf("unsigned short의 최댓값 %u\n", USHRT_MAX);
printf("unsigned int의 최댓값 %u\n", UINT_MAX);
printf("unsigned long의 최댓값 %lu\n", ULONG_MAX);

return 0;
}

```

Note

#include <limits.h> // 자료형 최댓값, 최솟값 알 수 있는 라이브러리
 unsigned를 사용하려면 **%u or %lu** 사용할 것

Important

정수형의 양수 범위를 2배로 늘리는 unsigned 자료형이 있다.

unsigned : 0과 양수만 표현 (음수 표현 X)

unsigned는 MIN값 X

기존 자료형의 최대 범위 + 1한 값을 추가로 더해준다.

ex : char (-128 ~ 127) > **unsigned char (0 ~ 256 (127 + 128))**

Example

(a) : -126, (b) : 130

```

#include <stdio.h>

int main(void) {

    char num1 = -129;
    char num2 = 128;

    printf("%d \n", num1); -- (a)****
    printf("%d \n", num2); -- (b)

    num1 = -130;
    num2 = 129;

    printf("%d \n", num1); -- (c)
    printf("%d \n", num2); -- (d)

    return 0;
}

```

: Example

(a) : 127, (b) : -128, (c) : 126, (d) : -127

오버플로우

- 정수형의 오버플로우와 언더 플로우는 순환된 값을 출력한다.
- 오버플로우 : 자료형에 저장할 수 있는 최대값 보다 큰 수 저장하는 경우
- 언더플로우 : 자료형에 저장할 수 있는 최소값 보다 작은 수 저장하는 경우

Important

최댓값에서 +1만큼 오버플로우 발생하면 ?

: -128 출력

최대값에서 +2만큼 오버플로우 발생하면 ?

: -127 출력

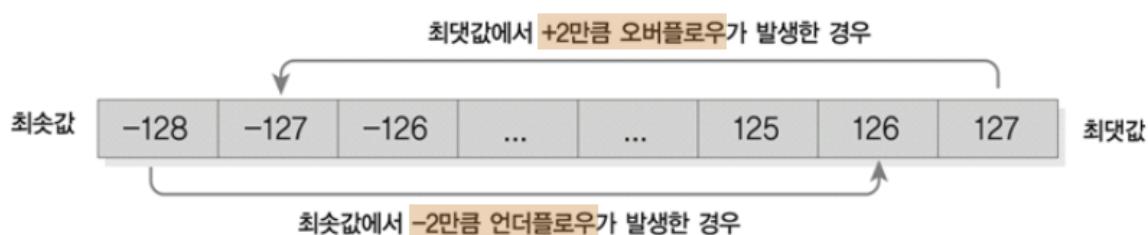


최소값에서 -1만큼 오버풀로우 발생하면?

: 127

최소값에서 -2만큼 오버풀로우 하면 ?

: 126



Important

실수형은 double을 사용하여 정밀도를 높힌다.

Note

float pi = 3.14; 를 하면 자동으로 double (컴파일러 자동)으로 형변환이 되는데, 이를 막기 위해서 f를 붙힌다.

float pi = 3.14f;

실수형의 최대 최소 및 실수에서의 지수 표현

```
#include <stdio.h>
#include <float.h>
```

```

int main(void) {

    printf("float 최솟값 %e, 최댓값 %e \n", FLT_MIN, FLT_MAX);
    printf("double 최솟값 %e, 최댓값 %e \n", DBL_MIN, DBL_MAX);

    //

    float num1 = 3.4e+30;
    double num2 = 3.4e+30;

    printf("%f, %e \n", num1, num2); // float은 오차 발생
    printf("%lf, %le \n", num1, num2);

    return 0;
}

```

Note

%e 지수 표현을 사용하여 나타내기

실수형에서 최대/최소를 나타낼 때 float(%f, %e) 사용하면 오차가 발생

double(%lf, %le)를 사용하기

출력시 잘려서 나올 수 있음.

FLT_MIN | FLT_MAX

DBL_MIN | DBL_MAX

Important

num1 은 3.4e+30;이고 float 은 +38까지 표현 가능하지만 정밀도에서 떨어지기 때문에
double 사용 할 것

```

char a = 'a';
char A = 'A';

printf("ASCII 코드 값 : %c > %d, %c > %d\n", a, a, A, A);

```

Note

아스키 코드 모를 때 출력하는 방법
'문자'를 %d로 출력하면 아스키코드 나옴

```
#include <stdio.h>

typedef int money;

int main(void) {

    money num1 = 3000;
    money num2 = 10000;
    money num3 = 2000;
    money num4 = 0;

    num4 = num1 + num2 + num3;
    printf("total money : %d won \n", num4);

    return 0;
}
```

Summary

typedef : 기본 자료형을 개발자가 지정하는 것.

자료형 변환

- 작은게 큰걸로 바뀌면 데이터 손실 X
- 큰게 작은걸로 바뀌면 데이터 손실 O

자동형변환

- 모든 경우 가능한데, 데이터 손실이 있기 때문에 강제형 변환을 권장하는 경우

자료 형 변환 결과 값 비교 (강제형변환)

==> 결과 값 예상해보기 ==

```
#include <stdio.h>
int main(void) {
    int num1 = 10, num2 = 3;---
    double result = 0.0;

    result = num1 / num2;
    printf("결과 0(num1 / num2) : %lf\n", num1 / num2);
    printf("결과 1 (result = num1/num2): %lf\n", result);
    printf("결과 2 ((double)num1 / num2): %lf\n", (double)num1 / num2);
    printf("결과 3 (num1 / (double)num2) : %lf\n", num1 / (double)num2);
    printf("결과 4 ((double)num1 / (double)num2) : %lf\n", (double)num1 /
(double)num2);

    return 0;
}
```

📎 Result

결과 0 : 0.000000
결과 1 : 3.000000
결과 2 : 3.333333
결과 3 : 3.333333
결과 4 : 3.333333

📎 Note

1. **int를 값을 float / double로 출력하려면 0.00000 출력됨**
2. **실수 출력(%lf, %f)에 int 값 넣으면 0뜬다.**
3. **반대도 마찬가지**

⑤ 관계 연산자

- 관계를 비교하여 참(True)과 거짓(False)으로 결론짓는 연산자

관계 연산자	예	설명	결과
>	a>b	a가 b보다 클지를 비교	1(참), 0(거짓)
<	a<b	a가 b보다 작을지를 비교	1(참), 0(거짓)
>=	a>=b	a가 b보다 크거나 같을지를 비교	1(참), 0(거짓)
<=	a<=b	a가 b보다 작거나 같을지를 비교	1(참), 0(거짓)
==	a==b	a가 b보다 같을지를 비교	1(참), 0(거짓)
!=	a!=b	a가 b보다 같지 않을지를 비교	1(참), 0(거짓)

⑥ 논리 연산자

- **&&** : AND 연산자 (논리곱)
- **||** : OR 연산자 (논리합)
- **!** : NOT 연산자 (논리 부정)

A	B	A && B	A B	!A	!B
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	1
1	1	1	1	0	0

Note

NOT 연산자는 변수에 어떤 값이 들어가던 1이 아니면 0을 뺍는다.

```
int num3 = 5;
int result = !num3;
printf("result 값 : %d\n", result);
```

답 : 0(거짓)

비트 연산자

비트 연산자

맨 왼쪽 비트는 부호 비트 , 1:(-), 0:(+)

십진수 = 16진수 = 8진수

10 = 0xa = 012

1. 2진수

1. 총 4바이트 메모리 공간 사용

2. 16진수

1. 2진수 메모리 공간 기준으로

2. 오른쪽 기준으로 4비트 단위로 묶음

3. 8진수

1. 2진수 메모리 공간 기준으로

2. 오른쪽 기준으로 3비트 단위로 묶음

⑧ 비트 연산자

- 데이터를 비트 단위로 처리하는 연산자

비트 연산자	연산식	설명
&	a & b	비트 단위 AND 연산
	a b	비트 단위 OR 연산
^	a ^ b	비트 단위 XOR 연산
~	~a	비트 단위 NOT 연산
<<	a << 3	왼쪽으로 세 칸 이동
>>	a >> 1	오른쪽으로 한 칸 이동

⌚ Important

모두 비트 단위로 계산된다.

& : 두 개의 비트가 모두 1일 때 1 반환 (비트 단위 AND 연산)

| : 두개의 비트 중 하나만 1이여도 1 반환 (비트 단위 OR 연산)

^ : 두개의 비트가 서로 같지 않을 때 1 반환 (비트 단위 XOR 연산)

~ : 보수 연산으로 비트 반전 시킴 (비트 단위 NOT 연산)

<< : 비트를 왼쪽으로 이동 a<<3 :: 왼쪽으로 3칸 이동

>> : 비트를 오른쪽으로 이동 a>>1 :: 오른쪽으로 1칸 이동

```

#include <stdio.h>
int main(void) {

    int num1 = 20, num2 = 16;
    int result = 0;

    result = num1 & num2;
    printf("비트 단위 & 연산 결과 : %d\n", result);

    result = num1 | num2;
    printf("비트 단위 | 연산 결과 : %d\n", result);

    result = ~num1;
    printf("비트 단위 ~ 연산 결과 : %d\n", result);

    num1 = 10;
    result = num1 << 2;
    //0000 1010 : 10
    //0010 1000 : 40 (num1 * 2의 2승)
    printf("비트 단위 << 연산 결과 : %d\n", result);

    num1 = 10;
    num2 = -10;
    int result1 = 0, result2 = 0;

    result1 = num1 << 1;

    result2 = num2 >> 1;

    printf("비트 단위(num1) >> 연산 결과 : %d\n", result1);
    printf("비트 단위(num2) >> 연산 결과 : %d\n", result2);

    return 0;
}

```

Result

비트 단위 & 연산 결과 : 16
 비트 단위 | 연산 결과 : 20
 비트 단위 ~ 연산 결과 : -21
 비트 단위 << 연산 결과 : 40

```

0000 1010 : 10
0010 1000 : 40 (num1 2의 2승)
비트 단위(num1) >> 연산 결과 : 20
0000 1010 : 10
0001 0100 : 20 (num1 2의 1승)
비트 단위(num2) >> 연산 결과 : -5
1000 1010 : -10
1000 0101 : -5 (num1 / 2의 1승)

```

우선 순위	연산자	연산 방향
1	() [] -> .	왼쪽에서 오른쪽
2	~ ~ ++ -- + - * &	오른쪽에서 왼쪽
3	* / %	왼쪽에서 오른쪽
4	+ -	왼쪽에서 오른쪽
5	<< >>	왼쪽에서 오른쪽
6	< <= > >=	왼쪽에서 오른쪽
7	== !=	왼쪽에서 오른쪽

우선 순위	연산자	연산방향
8	&	왼쪽에서 오른쪽
9	^	왼쪽에서 오른쪽
10		왼쪽에서 오른쪽
11	&&	왼쪽에서 오른쪽
12		왼쪽에서 오른쪽
13	? :	오른쪽에서 왼쪽
14	= += -= *= /= %= &= ^= != <<= >>=	오른쪽에서 왼쪽
15	,	왼쪽에서 오른쪽

반복문 (for)

```

//1. for문 무한루프
for(int i =0; 1;i++){
    //코드
}

```

```

//2. 선언을 미리
int num=10;
for( ; num >0 num--){
    //코드
}

```

```
//3. 조건문 없는 경우  
for(int i=0; ; i++){  
    if(i>10) break; / 무한루프 탈출하기 위함  
}
```

Note

while : 조건문 아니면 바로 벗어남

do~while : 조건문 나중에 확인

Important

switch ~ case문

1. default : case에 해당하는 조건이 없을 때 실행

2. break : switch 문 종료

Important

switch ~ case

1. 관계 연산이 올 수 없다. (정수형, 문자형)

2. 실수형 불가능

2. break 사용 가능, continue 사용 불가능

```
#include <stdio.h>  
double divide(double x, double y);  
void information(void);  
  
int main(void){  
    //함수 호출 코드  
    // ...  
    //...
```

```

}

double divide(double x, double y){
    //함수 코드
}

void information(void){
    //함수 코드
}

```

재귀함수

- 함수 내에서 자기 자신을 호출하는 함수
 - 재귀호출 (Recursive Call): 자기 자신을 호출하는 행위
 - 시간과 메모리 공간의 효율이 저하 -> 개발에 신중히 사용할 것
-

1. 지역변수 (Local Variable)

- 함수 내에서, 중괄호 **내부**에서, 함수의 매개변수로 (함수의 입력 변수로)사용
- 중괄호 지역을 빠져나가면 메모리 **자동으로 소멸**
- 초기화 하지 않으면 **쓰레기 값** 저장
- 메모리 생성 : 중괄호 내에서 초기화 될 때
- 메모리 소멸 : 중괄호 내에서 탈출 할 때

2. 전역변수 (Global Variable)

- 중괄호 **외부**에서 사용
- 초기화 하지 않아도 **자동 0으로 설정**
- 메모리 생성 : 프로그램 시작될 때
- 메모리 소멸 : 프로그램 종료될 때

3. 정적변수(Static Variable)

- 자료형 앞에 static 키워드
- 프로그램 **종료되지 않으면** 메모리 소멸 일어나지 않음
- 초기값 지정 안해도 **자동으로 0설정**
- 프로그램 시작되면 **초기화는 한 번만 수행**
- 메모리 생성 : 중괄호 내에서 초기화 될 때**
- 메모리 소멸 : 프로그램 종료될 때**

4. 외부 변수

- 외부 파일에 선언된 변수 참조하는 변수
- 자료형 앞에 **extern 키워드** 사용

- **다른 파일에 있는 전역변수 사용
`extern int num1;`
- 특정 전역변수를 **외부에서 참조 못하게 하려면 static 키워드 사용**

5. 레지스터 변수(Register Variable)

- **CPU 내부 레지스터에 변수 할당하는 변수**
 - 처리속도 빠름
 - 자료형 앞에 **register 키워드** 사용
-

변수의 종류와 범위

1. 코드영역 : 실행 코드 or 함수들 저장 (실행코드, 함수)
2. 스택영역 : 매개변수 or 중괄호 내부에 정의된 변수 (지역변수, 매개변수) 저장
3. 데이터영역 : 전역변수와 정적변수 저장
4. 힙 영역 : 동적으로 메모리 할당하는 변수 저장