○ Confused C Language Concepts

○ 상수와 기본 자료형

● 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
```

- 주의할 것은, 표현의 방식이 다르다고 해서 저장되는 값이 달라지는 것은 아니라는 점이다. 위에 세 문장에 의해서 변수에 초기화 되는 값은 모두 동일하다. 다만, 초기화에 사용된 표현의 방식에만 차이가 있을 뿐이다.

● 이름을 지니지 않는 리터럴(Literal) 상수

```
int main(void)
{
    int num = 30 + 40; // 30과 40은 상수이다!
}
```

- 위 덧셈이 가능하다는 것은 30 + 40의 연산을 CPU가 진행해야 한다는 뜻인데, 그러기 위해서는 30이라는 정수도, 40이라는 정수도 메모리상에 존재해야 한다. 그래야 CPU의 연산대상이 될 수 있기 때문이다.
- 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.
- 이렇듯 변수와 달리 이름이 없는 상수를 가리켜 '리터럴 상수' 또는 '리터럴'이라 한다.
- int형으로 표현 가능한 정수형 상수는 int형으로, double형으로 ……

● 접미사를 이용한 다양한 상수의 표현

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
F	float	float f = 2.15F
L	long double	long double f = 5.789L

〈대소문자 구분 X〉

● 피연산자의 자료형 불일치로 발생하는 자동 형 변환

- 피연산자의 자료형이 일치하지 않아서 발생하는 자동 형 변환은 데이터의 손실을 최소화하는 방향으로 진행된다.
- int -〉 long -〉 long long -〉 float -〉 double -〉 long double
- char와 short는 'Integral Promotion'에 의해 둘 다 int형 정수로 형 변환된다.
- long long 보다 float가 표현할 수 있는 값의 범위 더 넓다.

● scanf 함수 이야기

● 정수 기반의 입력형태 정의하기

- %d: 10진수 정수의 형태로 데이터를 입력 받는다.
- %o: 8진수 양의 정수의 형태로 데이터를 입력 받는다.
- %x: 16진수 양의 정수의 형태로 데이터를 입력 받는다.

```c
#include <stdio.h>

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;
}
```

```
세 개의 정수 입력: 12 12 12
입력된 정수 10진수 출력: 12 10 18
Press any key to continue . . .
```

● 실수 기반의 입력형태 정의하기

- printf 함수에서는 서식문자 %f, %e, %g의 의미가 각각 달랐다. 그러나 scanf 함수에서는 'float형 데이터를 입력 받겠다'는 동일한 의미를 담고 있다.
- double형을 받을 때는,
  - %lf double %f에 l이 추가된 형태
  - %Lf long double %f에 L이 추가된 형태

```c
int main(void)
{
    float num1;
    double num2;
    long double num3;

    printf("실수 입력1(e 표기법으로): ");
    scanf_s("%f", &num1);
    printf("입력된 실수 %f \n", num1);

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

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

```
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
Press any key to continue . . .
```

● printf 함수의 이해

● 특수문자의 종류

특수문자	의미
\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

● 서식문자의 종류와 그 의미

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 *	포인터 주소 값

- 그리고 8진수를 8진수답게, 16진수를 16진수답게 출력하고 싶다면, 서식문자 %o 그리고 서식문자 %x의 사이에 문자 #을 넣어서 출력하면 된다.

```
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
Press any key to continue . . .
```

- 실수의 출력에 사용되는 서식문자 %f와 %e에서 소수점 이하 자릿수가 긴 실수라면 %e로 출력하는 것이 수의 크기를 파악하기에 좋을 것이다.
- 서식문자 %g는 실수의 형태에 따라서 %f와 %e 사이에서 적절한 형태의 출력을 진행한다.
- %g와 %G의 차이점은 e 표기법의 e를 소문자로 출력하느냐 대문자로 출력하는가에 있다.

● 필드 폭을 지정하여 정돈된 출력 보이기

- %8d
  - 필드 폭을 8칸 확보하고, 오른쪽 정렬해서 출력을 진행한다.
- %-8d
  - 필드 폭을 8칸 확보하고, 왼쪽 정렬해서 출력을 진행한다.

- **Goto statements**

```c
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("2을 입력하셨습니다! \n");
    goto END;
OTHER:
    printf("3 혹은 다른 값을 입력하셨군요! \n");

END:
    return 0;
}
```

- **포인터와 배열**
  - 중요한 결론! Arr[i] == *(Arr + i)

- **포인터와 다차원 배열**

```c
#include <stdio.h>
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): %d \n", sizeof(arr2d));
    printf("sizeof(arr2d): %d \n", sizeof(arr2d));
    printf("sizeof(arr2d): %d \n", sizeof(arr2d));
    return 0;
}
```

```
10091352
10091352
10091352

10091364
10091364

10091376
10091376

sizeof(arr2d): 36
sizeof(arr2d): 36
sizeof(arr2d): 36
sizeof(arr2d): 36
Press any key to continue . . .
```

- arr2d[0], arr2d[1], arr2d[2]은 각각 0, 1, 2 행의 시작주소를 나타낸다. 따라서 배열 전체를 의미하는 arr2d와 arr2d[0]은 다르다.
- 위와 같은 배열의 포인터 형을 묻는다면, “배열이름 arr2d이 가리키는 대상은 int형 변수이고, arr2d의 값을 1 증가하면 실제로는 sizeof(int) * 4의 크기만큼 주소 값이 증가하는 포인터 형이다.”
- 따라서 가리키는 대상이 int형 변수이면서 포인터 연산 시 sizeof(int) * 4의 크기단위로 증가 또는 감소하는 포인터 변수 ptr은 다음과 같이 선언한다.
  - int (*ptr) [4];
  - 위 수식을 보충 설명하면, 변수 ptr 앞에 * 선언은 ptr이 포인터 선언이 되게 한다. 그리고 [4]는 포인터 연산 시 4칸씩 건너뛰되, ptr이 가리키는 변수가 int형 변수이기 때문에, int형 편수를 4칸씩 건너뛰다는 의미가 되는 것이다. 그리고 위와 같은 포인터 편수는 2차원 배열을 가리키는 용도로만 사용되기 때문에 이러한 유형의 포인터 변수를 가리켜 ‘배열 포인터 변수’라 한다.

```c
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 arr1 **\n");
    for (i = 0; i < 3; i++)
        printf("%d %d \n", ptr[i][0], ptr[i][1]);

    ptr = arr3;
    printf("** Show 4,2 arr1 **\n");
    for (i = 0; i < 4; i++)
        printf("%d %d \n", ptr[i][0], ptr[i][1]);
    return 0;
}
```

```
** Show 2,2 arr1 **
1 2
3 4
** Show 3,2 arr1 **
1 2
3 4
5 6
** Show 4,2 arr1 **
1 2
3 4
5 6
7 8
Press any key to continue . . .
```

- **2차원 배열이름의 특성과 주의사항**
- **'배열 포인터'와 '포인터 배열'을 혼동하지 말자**

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

- whoA와 whoB의 외형적 유일한 차이점은 소괄호의 유무이다. 그런데 whoA는 배열 선언이고, whoB는 포인터 변수 선언이다. 조금 더 구체적으로 말하면, whoA는 int형 포인터 변수로 이뤄진 int형 포인터 배열이고, whoB는 가로길이가 4인 int형 2차원 배열을 가리키는 용도의 포인터 변수이다.

```
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;
}
```

```
10 20 30 40
1 2 3 4
5 6 7 8
Press any key to continue . . .
```

● 2차원 배열을 함수의 인자로 전달하기

■ 아래의 코드에서 호출되고 있는 SimpleFunc 함수의 원형을 선언해보자.

```
int arr1[2][7];
double arr2[4][5];
SimpleFunc(arr1, arr2);
```

■ arr1의 주소 값을 전달받을 수 있는 매개변수의 이름을 parr1이라 하고, arr2의 주소 값을 전달받을 수 있는 매개변수의 이름을 parr2라 하면, 이 둘은 각각 다음과 같이 선언되어야 한다.

- ➢ int (*parr1) [7]
- ➢ double (*parr2) [5]

■ 따라서 SimpleFunc 함수는 반환형이 void라고 가정하면, 이 함수는 다음의 형태로 정의 되어야 한다.

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

■ 그리고 이를 대신해서 다음과 같이 정의하는 것도 가능하다.

- ➢ void SimpleFunc(int parr1 [] [7], double parr2 [] [5]) {}

■ 왜냐하면, 다음의 두 매개변수 선언은 동일한 선언이기 때문이다.

- ➢ int (*parr1) [7] = int parr1 [] [7]
- ➢ double (*parr2) [5] = double parr2 [] [5]

■ 물론 이 둘은 매개변수의 선언에서만 같은 의미를 지니므로, 그 의외의 영역으로 까지 확대해서 동일한 선언으로 간주하면 안된다.

```c
void ShowAarr2Dstyle(int(*arr)[4], int column)
{
    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)
{
    int i, j, sum = 0;
    for (i = 0; i < column; i++)
        for (j = 0; j < 4; j++)
            sum += arr[i][j];
    return sum;
}
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 };

    ShowAarr2Dstyle(arr1, sizeof(arr1) / sizeof(arr1[0]));
    ShowAarr2Dstyle(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;
}
```

```
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
Press any key to continue . . .
```

● 2차원 배열에서도 arr[i]와 *(arr + i)는 같다

- int arr[3][2] = {{1, 2}, {3, 4}, {5, 6}};
- 이 배열에서 6이 저장된 인텍스 [2][1]의 위치의 값을 4로 변경시키기 위해서는 다음의 문장을 실행해야 한다.
  - arr[2][1] = 4;
- 그런데 이는 다음의 문장들로 대신할 수 있다.
  - (*(arr + 2))[1] = 4;
  - *(arr[2] + 1) = 4;
  - *(*(arr + 2) + 1) = 4;

```c
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 + 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;
}
```

```
a[0]: 008FFC84
*(a + 0): 008FFC84
a[1]: 008FFC8C
*(a + 1): 008FFC94
6, 6
6, 6
6, 6
Press any key to continue . . .
```

● 함수 포인터와 void 포인터

```c
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; int num2 = 20;

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

    fptr1(num1, num2);
    fptr2(str);
    return 0;
}
```

```
10 + 20 = 30
Function Pointer
Press any key to continue . . .
```

■ 변수만 메모리 공간에 저장되는 것은 아니다. 프로그램 실행의 흐름을 구성하는 함수들도 바이너리 형태로 메모리 공간에 저장되어서 호출 시 실행이 된다. 그리고 이렇게 메모리상에 저장된 함수의 주소 값을 저장하는 포인터 변수가 바로 '함수 포인터 변수'

이다.

● 대표적인 선행처리 명령문

```c
#define NAME "권지용"
#define AGE 30
#define PRINT_ADDR puts("주소: 서울특별시\n");
int main(void)
{
    printf("이름: %s \n", NAME);
    printf("나이: %d \n", AGE);
    PRINT_ADDR;
    return 0;
}
```

```c
#define SQUARE(X) X*X
int main(void)
{
    int num = 20;
    printf("Square of num: %d \n", SQUARE(num));
    return 0;
}
```

● 매크로 함수의 장점

- ■ 매크로 함수는 일반 함수에 비해 실행속도가 빠르다.
- ■ 자료형에 따라서 별도로 함수를 정의하지 않아도 된다.

● 매크로 함수의 단점

- ■ 정의하기가 까다롭다.
- ■ 디버깅하기 쉽지 않다.

- **조건부 컴파일(Conditional Compilation)을 위한 매크로**
  - 매크로 지시자 중에는 특정 조건에 따라 소스코드의 일부를 삽입하거나 삭제할 수 있도록 디자인 된 지시자가 있다.

- **#if #end: 참이라면**

```c
#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 참이라면
    printf("%d + %d = %d \n", num1, num2, num1 + num2);
#endif

#if MIN // MIN이 참이라면
    printf("%d - %d = %d \n", num1, num2, num1 - num2);
#endif
    return 0;
}
```

- **#ifdef #endif: 정의되었다면**

```c
//#define ADD 1
#define MIN 0
int main(void)
{
    int num1, num2;
    printf("두 개의 정수 입력: ");
    scanf("%d %d", &num1, &num2);

#ifdef ADD // ADD가 정의되었다면
    printf("%d + %d = %d \n", num1, num2, num1 + num2);
#endif

#ifdef MIN // MIN이 정의되었다면
    printf("%d - %d = %d \n", num1, num2, num1 - num2);
#endif
    return 0;
}
```

- **#ifndef #endif: 정의되지 않았다면**
- **#else의 삽입: #if, #ifdef, #ifndef에 해당**
- **#elif의 삽입: #if에만 해당**

- **매개변수의 결합과 문자열화**
- **문자열 내에서 매크로의 매개변수 치환이 발생하게 만들기: # 연산자**

```c
#define STRING_JOB(A, B) #A "의 직업은 " #B "입니다."
int main(void)
{
    printf("%s \n", STRING_JOB(권지용, 아티스트));
    return 0;
}
```

- **필요한 형태대로 단순하게 결합하기: 매크로 ## 연산자**
  - #define CON(UPP, LOW) UPP ## 00 ## LOW
    - CON(22, 77) -〉 220077