Kumoh National Institute of Technology

C Programming

Ch.7-1 포인터의 이해

컴퓨터공학과

C프로그래밍

Contents

1. C 메모리 구조 (추가)
2. 포인터란 무엇인가?
3. 포인터 관련 연산자: & 연산자와 * 연산자

- 추가 : 교재 외 추가 자료

C 메모리 구조

- 하나의 프로그램 실행(프로세스) 시 생성되는 메모리 구조

메모리 영역

데이터 (전역, 정적 변수 : a, c)
스택 (지역 변수 : x, y, b, p)
힙 (동적 변수 : malloc 변수)
코드 (함수 : Sum, main)

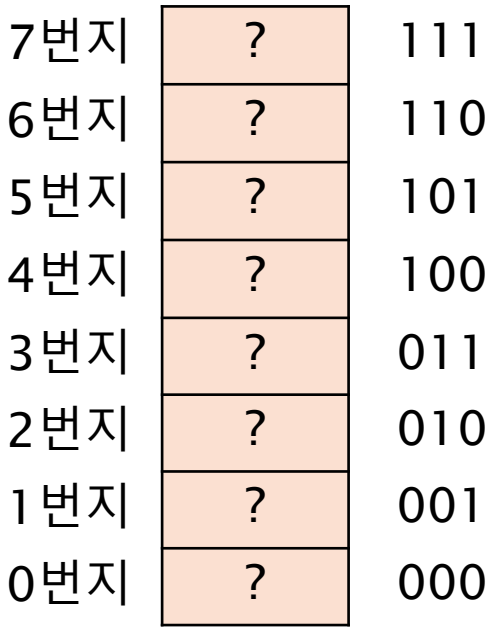
```c
int a = 1;

int Sum(int x, int y)
{
    static int c;
    c = x + y;
    return c;
}

void main(void)
{
    int b = 2;
    int *p = (int *)malloc(4); // 아직 잘 모름
    printf("%d + %d = %d\n", b, *p, b + (*p));
    free(p);
}
```

결론 : 변수와 함수 모두 메모리에 올라감!

메모리에는 바이트 단위로 주소가 붙어있음

- 만약 주소값을 3비트로 표현한다면?
  - 총 8바이트(2^3) 구별 가능
    - 0번지 ~ 7번지 (0 ~ 2^3-1)

- 실제로는 (32비트 시스템의 경우)
  4바이트(32비트)로 표현됨
  - 총 2^{32}바이트 구별 가능
    - $2^2 * 2^{10} * 2^{10} * 2^{10} = 4GByte$
    - 0 ~ $2^{32}-1$번지



어쨌든 변수와 함수코드 모두 메모리에 올라감!

Visual Studio : 32/64bit, Debug/Release 모드별 주소 차이

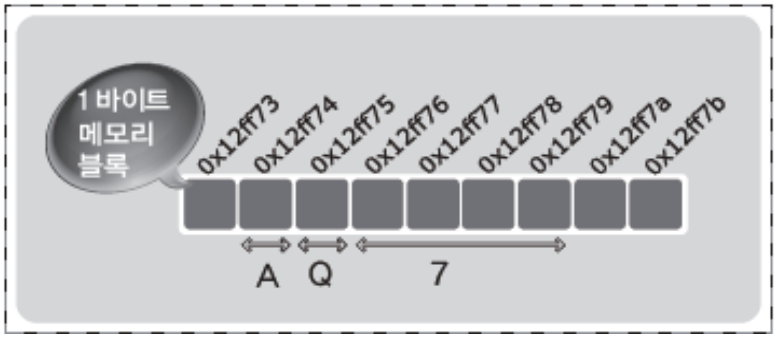
```c
int main(void)
{
    int a;
    int b;
    int c;

    printf("a: %p %d\n", &a, &a);
    printf("b: %p %d\n", &b, &b);
    printf("c: %p %d\n", &c, &c);

    return 0;
}
```

```
// 32bit, Debug : -12
a: 00F3FAC8 15989448
b: 00F3FABC 15989436
c: 00F3FAB0 15989424

// 32bit, Release : -4
a: 0137FE28 20446760
b: 0137FE24 20446756
c: 0137FE20 20446752

// 64bit, Debug : +32
a: 000000C312DEF754 316602196
b: 000000C312DEF774 316602228
c: 000000C312DEF794 316602260

// 64bit, Release : +4
a: 0000000B6A73F9B0 1785985456
b: 0000000B6A73F9B4 1785985460
c: 0000000B6A73F9B8 1785985464
```

주소값을 저장하기 위한 포인터 변수

- 포인터와 포인터 변수
  - 포인터 변수 : 메모리의 주소값을 저장하기 위한 변수
  - 포인터 : 포인터 상수와 포인터 변수를 모두 포함
    - cf) int 정수 : int 정수 상수와 int 정수 변수 포함
  - 일반적으로 포인터 ≒ 포인터 변수
- 변수(와 함수)는 메모리의 특정 영역에 저장됨

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



변수 num의 시작 주소(0x12ff76) 값을 어떤 변수에 저장하고 싶은데... → 포인터 변수

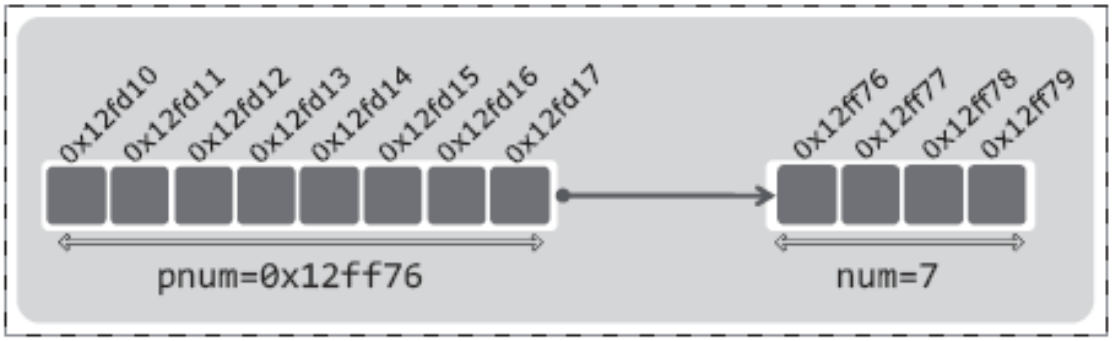
포인터 변수와 & 연산자 맛보기

- int num;이 저장되어 있는 메모리의 시작 주소값을 포인터 변수 pnum에 넣어보자!

```
int main(void)
{
    int num = 7;
    int* pnum;     // 포인터 변수 선언
    pnum = &num;   // & 연산자 : num의 시작 주소값 반환
    ......
}
```

pnum은 int형 변수의 주소값을 저장하는 포인터 변수

- 메모리 저장 상태 (64비트 시스템 가정)



포인터 변수도 변수다!
- 메모리에 위치함
- 32비트 시스템 : 4바이트 차지
- 64비트 시스템 : 8바이트 차지

포인터 변수 pnum이 변수 num을 가리키는 형태임

포인터 변수 선언하기

- 포인터 변수의 타입 : 가리키고자 하는 변수의 자료형에 따라 결정됨
  - 차지하는 메모리 크기는 모두 동일 (32비트 시스템 : 4바이트)

```
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

sizeof로 변수의 크기 확인
```
sizeof(char *)
sizeof(int *)
sizeof(double *)
sizeof(unsigned int *)
```

일반화

```
type* ptr;
```
type형 변수의 주소 값을 저장하는 포인터 변수 ptr의 선언

포인터의 형(type)

- int* pnum; 에서 pnum의 타입은?
  - char a; int b; double c; 에서 a, b, c의 타입은?

```
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 선언 시 * 의 위치에 따른 차이는 없음. 다음은 모두 동일
int *ptr;     // 권장. 여러 개 선언 시 int *pa, *pb, *pc;
int* ptr;     // int* pa, pb, pc;의 경우 pa만 포인터
int * ptr;
```

변수의 시작 주소값을 반환하는 & 연산자

- & 주소 연산자 : 변수에 적용됨
  - 주소에도 타입이 있다!
  - num의 주소

    → num이 int이므로 int의 주소

    → int형 포인터

    → 그렇다면 int형 포인터 변수(int *)에 저장해야 되겠네…

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

- 잘못된 예

```
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.

포인터가 가리키는 메모리를 참조하는 * 연산자 (1)

- 포인터 변수가 가리키는 메모리의 변수를 사용하고 싶다!
  - 역참조 연산자 * 사용

현재 pnum이 num을 가리키므로
*pnum = 20;과 num = 20;은 동일

```
int main(void)
{
    int num=10;
    int * pnum=&num;        pnum이 num을 가리킨다.
    *pnum=20;               pnum이 가리키는 공간(변수)에 20을 저장
    printf("%d", *pnum);    pnum이 가리키는 공간(변수)에 저장된 값 출력
    . . . .
}
```

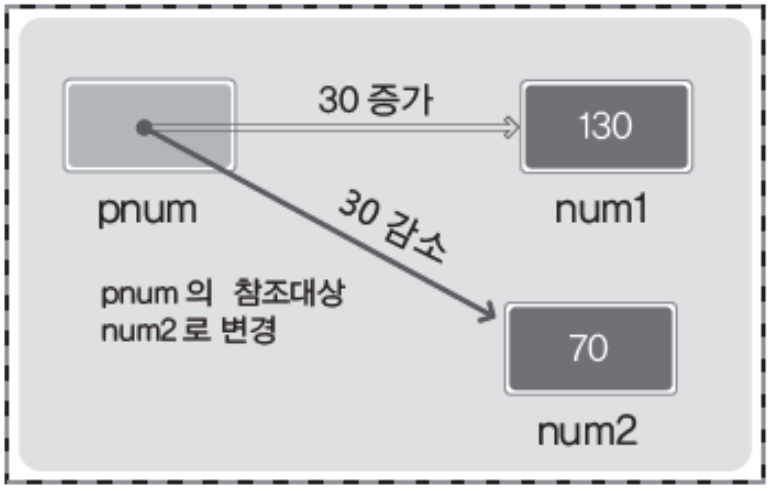
int* pnum = # 이후에 num과 *pnum은 동격이다.

포인터가 가리키는 메모리를 참조하는 * 연산자 (2)

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

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

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

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

```
num1:130, num2:70
```

포인터 변수도 변수이다!
따라서 값이 변할 수 있다(다른 변수를 가리킬 수 있다).



다양한 포인터 형이 존재하는 이유

- 포인터 형 : 메모리 공간을 참조하는 방법을 알려줌
  - ~~pointer라는 타입이 있다면 pointer a = #~~
    - ~~a가 가리키는 곳을 어떻게 해석하지?~~
  - int* a; // a가 가리키는 곳에 int 변수가 있다!
    - 역참조 연산자 (*a) : 4바이트 정수로 해석
  - double* b; // b가 가리키는 곳에 double 변수가 있다!
    - 역참조 연산자 (*b) : 8바이트 실수로 해석
- 맞지 않는 타입의 포인터 변수에 대입하면?

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

자동 형변환 발생 :
(double*) → (int*)
- 실행은 된다. 4바이트 int로 해석
- 프로그래머가 책임만 지면 된다^^

잘못된 포인터의 사용과 널 포인터

- 위험한 코드의 예 1
  - ptr이 쓰레기값으로 초기화됨
  - ptr이 가리키는 곳에 200을 넣어라
    - 거기가 뭐하는 곳인데? 위험

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

- 위험한 코드의 예 2
  - 125번지는 뭐하는 곳인데? 위험
  - 상수값을 바로 포인터 변수에 넣지 않음

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

- 안전한 코드의 예
  - 포인터 변수의 값을 0으로 초기화
    - if (ptr1 == 0) // 검사
      - 아직 유효하지 않음

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

NULL : 널포인터(내부값 0)

이번 장에서 배운 것

- 프로그램을 실행(프로세스)하면 코드와 변수 모두 메모리에 저장된다.
- 주소값은 0번지부터 1바이트 단위로 붙게 되며 32비트 시스템의 경우 4바이트를 사용하여 주소값을 나타낸다.
- 어떤 변수가 저장된 메모리의 시작 주소는 주소 연산자 &를 사용하면 알아낼 수 있다.
- 포인터 변수는 주소값을 저장하는 변수로서 어떤 변수가 저장된 주소를 저장하느냐에 따라 타입(형)이 정해진다. 예를 들어 int 변수의 주소값을 저장하려면 int형 포인터 변수(int *)를 사용한다.
- 역참조 연산자(*)를 사용하면 포인터 변수가 현재 가리키고 있는 변수를 참조할 수 있다.
- 포인터 변수 생성 시 NULL 포인터 값으로 초기화하면 나중에 포인터 변수가 유효한 주소값을 가지고 있는지를 확인할 수 있다.