C언어 강의자료

문정욱

C언어 더 알아보기 6

대입 연산자의 타입 변환

- 대입 연산자의 묵시적 타입 변환
  - 왼쪽 피연산자의 타입으로 변환
  - 만일 일반적인 묵시적 타입변환과 일치하지 않는 변환이 일어날 경우 프로그램을 컴파일 할 때 ***경고 메시지(warning message)*** 발생
  - 경고 메시지가 발생하면 *프로그래머의 의도를 명확하게 하기 위해* **명시적 타입 변환**을 *사용하는 것이 바람직*하다.

```
#include <stdio.h>

int main(void)
{
    short c=1;
    int i=2;
    long long ll=3;
    double d = 4.4;

    i = c;  // ok: char → int
    i = ll; // warning: long long → int
    i = d;  // warning: double → int

    i = c;          // ok
    i = (int) ll; // good
    i = (int) d;  // good

    return 0;
}
```

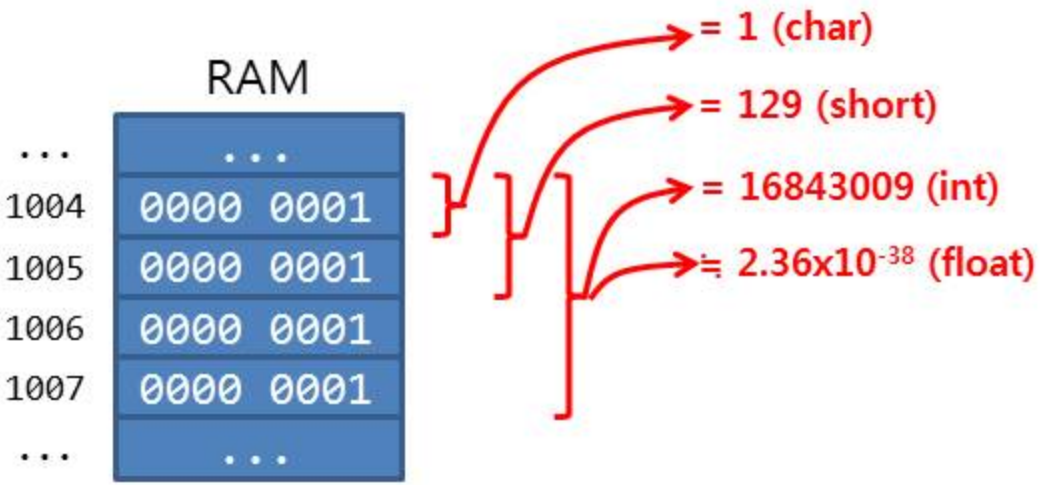
대입 연산자의 타입 변환

- 대입 연산자의 묵시적 타입 변환 (포인터 타입의 경우)
  - 참조타입이 다른 포인터 타입의 대입연산은 아주 조심해서 사용해야 한다.
  - 명시적 타입 변환을 하지 않으면 ***경고 메시지(warning message)*** 발생
  - 위험한 코딩이지만 *프로그래머의 의도가 반영된 코딩임을 명시 하기 위해* **명시적 타입 변환** *을 사용하는 것이 바람직*하다.

```
#include <stdio.h>

int main(void)
{
    char c;
    int i;
    char* pc = &c;
    int* pi = &i;

    pc = pi; // warning: int* → char*

    pc = (char*) pi; // ok, but be careful.
    return 0;
}
```

void 포인터(void*)

- 참조 타입의 의미
  - 포인터에 참조타입이 없고 주소 값만 저장되어 있으면 해당 주소의 메모리에 있는 데이터를 해석할 수 없다.
  - **값의 표현 방법**과 이를 저장하기 위한 **메모리 공간의 크기**를 알아야 값을 알 수 있다.



void 포인터(void*)

- void * 포인터 변수의 특징
  - 단순히 주소 값만 저장
  - **참조 타입이 없다.**
  - **참조 연산(*) 사용 불가능**
  - **가감산(+, -) 연산의 사용 불가능**
  - 명시적 타입 변환 후 참조 및 가감산 연산 사용 가능

```
#include <stdio.h>

int main(void)
{
    int a=3;
    void* p;

    p=&a;
    printf("%p\n", &a );
    printf("%p\n", p );
    printf("%d\n", *p );  // error
    printf("%p\n", p+1 ); // error
    printf("%d\n", *((int*)p) );
    printf("%p\n", (int*)p+1 );

    return 0;
}
```

함수 포인터

- 함수

RNT f(PRT);
RNT: ReturN Type, PRT: PaRameter Type f: Function Name

  - 기계어 명령(instruction)을 저장하는 메모리 공간
    - cf) 변수: 값(data)을 저장하는 메모리공간
- 함수 타입
  - 함수 = 주소 + 함수타입
    - 함수타입 = 반환타입 + 인자 타입

```
#include <stdio.h>

void f(int v)
{
    printf("%d\n",v);
}

int main(void)
{
    void (*p)(int);

    p = &f;
    return 0;
}
```

T(f) = void(int)

T(&f) = void(int)*

T(p) = void(int)*

함수 포인터

- 함수 포인터
  - 함수의 주소를 저장하는 메모리 공간

```
RNT f(PRT);
```

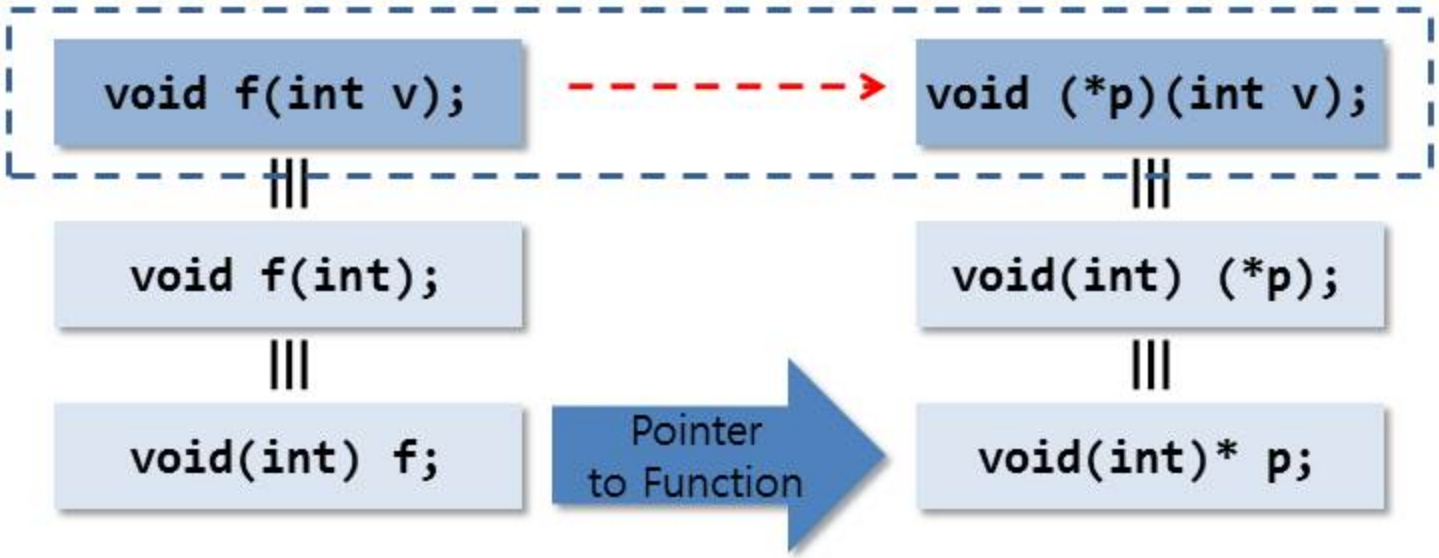
```
T(f) == RNT(PRT)
PT(f) == RNT(PRT)*
T(&f) == RNT(PRT)*
```

- 함수타입의 크기
  - 함수 타입의 크기는 없다.

```
#include <stdio.h>

void f(int v)
{
    printf("%d\n",v);
}

int main(void)
{
    void (*p)(int);

    p = &f;   // T(&f) == void(int)*
    return 0;
}
```

함수 포인터

함수 포인터 전환 과정



함수 포인터

- 함수의 묵시적 타입 변환

RNT(PRT) → RNT(PRT)*

- 함수 연산자

연산자	사용 방법	결과
()	함수주소(인자목록)	반환 값

- 함수 포인터의 특징
  - 함수 타입도 필요할 때 묵시적 타입변환이 일어난다.
  - 함수 포인터의 RT는 크기가 없다. 그러므로, 함수 포인터의 덧셈, 뺄셈 연산은 불가능 하다.

```
#include <stdio.h>

void f(int v)
{
    printf("%d\n",v);
}

int main(void)
{
    void (*p)(int);

    p=&f;
    p(3); // function operator

    p=f;  // implicity type conversion
    p(3); // function operator

    ++p;  // error
    return 0;
}
```

same result

함수 포인터

Pointer to Function 사용 예 1

```c
#include <stdio.h>

int add(int v1,int v2)
{
    return v1+v2;
}

int sub(int v1,int v2)
{
    return v1-v2;
}

void print_result(int v1,int v2,
                  int (**fl)(int,int))
                // int (*fl[])(int,int))
                // int (*fl[2])(int,int))
{
    int i;

    for(i=0;i<2;++i)
        printf("result=%d\n",fl[i](v1,v2));
}
```

```c
int main(void)
{
    int (*functlist[2])(int,int) = {&add,&sub};
    int a,b;

    scanf("%d%d",&a,&b);
    print_result(a,b,functlist);
    return 0;
}
```

입출력 결과

```
3 7
result=10
result=-4
계속하려면 아무 키나 누르십시오 . . .
```

함수 포인터

Pointer to Function 사용 예 2

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

int cmp(const void* v1,const void* v2)
{
    double diff;

    diff=(*(double*)v1) - (*(double*)v2);
    if(diff>0.0) return 1;
    if(diff<0.0) return -1;
    return 0;
}
```

```
void qsort(
    void *base,
    unsigned num,
    unsigned width,
    int (*f)(const void*, const void*)
);
```

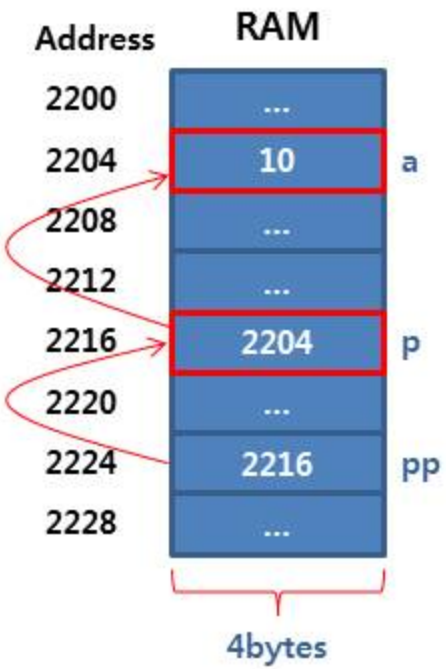
```
int main(void)
{
    double a[]={4.0, 8.0, 9.1, 3.4, 0.5};
    int n=sizeof(a)/sizeof(*a);
    int i;

    qsort(a,n,sizeof(double),cmp);

    for(i=0;i<n;++i)
        printf("%g ",a[i]);
    printf("\n");
    return 0;
}
```

입출력 결과

```
0.5 3.4 4 8 9.1
계속하려면 아무 키나 누르십시오 . . .
```

포인터의 포인터

- Pointer to Pointer



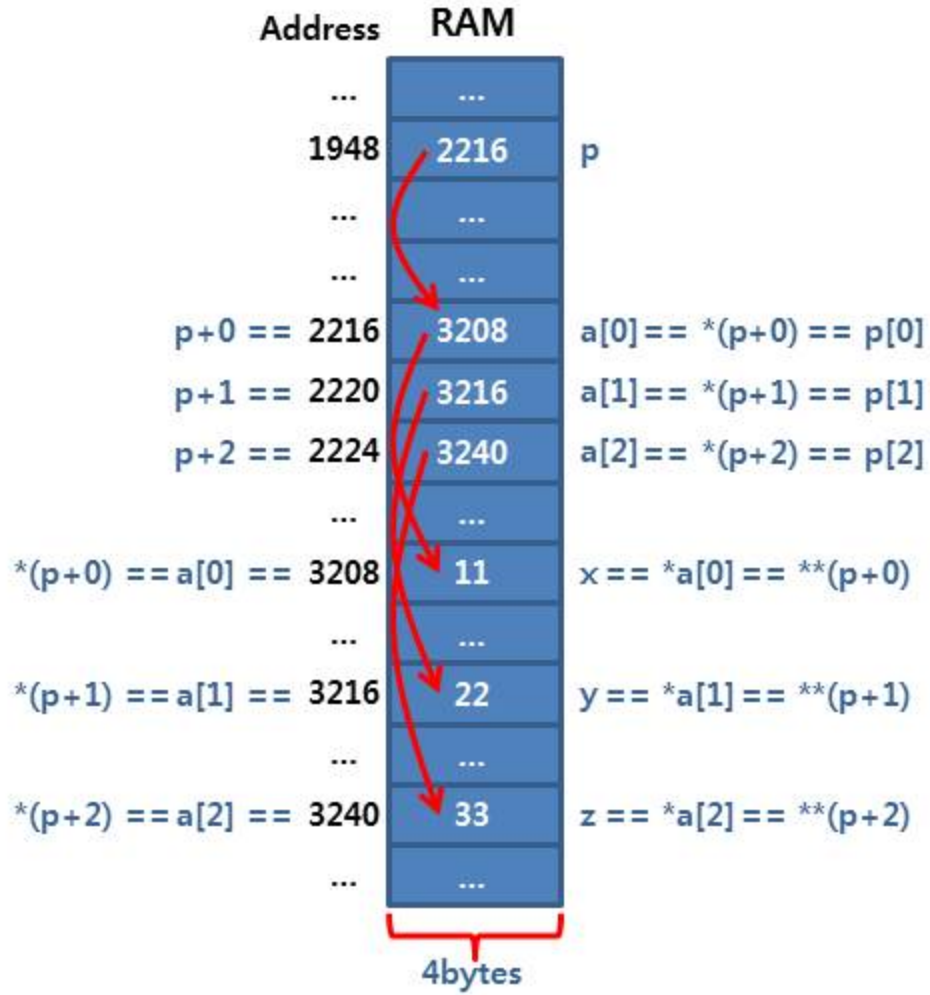
입출력 결과

```
a   = 17
&a  = 0035FB4C
p   = 0035FB4C
*pp = 0035FB4C
pp  = 0035FB40
계속하려면 아무 키나 누르십시오 . . .
```

```c
#include <stdio.h>

int main(void)
{
    int a = 10;
    int* p;
    int** pp;

    p=&a;
    pp=&p;

    *p+=3;
    **pp+=4;

    printf("a   = %d\n",a);
    printf("&a  = %p\n",&a);
    printf("p   = %p\n",p);
    printf("*pp = %p\n",*pp);
    printf("pp  = %p\n",pp);
    return 0;
}
```

포인터의 포인터

- Pointer to Pointer




```c
#include <stdio.h>

void increase(int** p)
{
    ++ **(p+0);  // **(p+0) == *a[0] == x
    ++ **(p+1);  // **(p+1) == *a[1] == y
    ++ **(p+2);  // **(p+2) == *a[2] == z
}

int main(void)
{
    int x=11, y=22, z=33;
    int* a[3] = { &x, &y, &z};

    increase( a );
    printf("%d %d %d\n", x, y, z);
    return 0;
}
```

입출력 결과

```
12 23 34
계속하려면 아무 키나 누르십시오 . . .
```

포인터의 포인터

```c
#include <stdio.h>

void increase(int** p)
{
    ++ **(p+0);  // **(p+0) == *a[0] == x
    ++ **(p+1);  // **(p+1) == *a[1] == y
    ++ **(p+2);  // **(p+2) == *a[2] == z
}

int main(void)
{
    int x=11, y=22, z=33;
    int* a[3] = { &x, &y, &z};

    increase( a );
    printf("%d %d %d\n", x, y, z);
    return 0;
}
```

```c
#include <stdio.h>

void increase(int* p[3])
{
    ++ *p[0];  // *p[0] == *a[0] == x
    ++ *p[1];  // *p[1] == *a[1] == y
    ++ *p[2];  // *p[2] == *a[2] == z
}

int main(void)
{
    int x=11, y=22, z=33;
    int* a[3] = { &x, &y, &z};

    increase( a );
    printf("%d %d %d\n", x, y, z);
    return 0;
}
```

포인터 사용시 주의할 점

- 포인터 변수의 초기화
  - NULL을 대입
    - 포인터가 **아무것도 가리키고 있지 않다**는 의미로 해석하면 된다.
    - NULL은 헤더파일 stdio.h에 아래와 같이 정의되어 있다.

```
#define  NULL  0
```

```
#include <stdio.h>

int main(void)
{
    int* p = NULL;
    int* q = 0;

    return 0;
}
```

같은 의미

포인터 사용시 주의할 점

- 포인터에게 NULL의 의미
  - 포인터는 0번지 메인 메모리 영역을 가리킬 수는 있다. 하지만, 그 내용을 참조하거나 변경해서는 안 된다.
  - **포인터에 0(zero)가 저장되어 있다는 것은 의미 없는 값이 저장되어 있다는 것을 뜻하며, 해당 포인터는 아무 대상도 가리키고 있는 않는 것을 뜻한다.**
  - **이때 포인터에 변수참조연산자(*)를 사용하면 실행오류가 발생할 수 있다.**

※ 주의
메인 메모리의 0번지는 운영체제에서 관리하는 영역이다. 임의의 응용프로그램이 권한 없이 이 영역을 참조하려 할 경우 운영체제는 해당 응용프로그램을 강제로 종료시킨다. (보안 기능이 부족한 운영체제는 이를 용납하기도 함)

```
#include <stdio.h>

int main(void)
{
    int a;
    int* p = NULL;

    *p = 3; // run-time error

    p = &a;
    *p = 3; // ok
    return 0;
}
```

포인터 p는 아무것도 가리키고 있지 않다.

포인터 p에 변수 a의 주소를 저장하면 p는 a를 가리키게 된다.

포인터 사용시 주의할 점

- 초기화 되지 않은 포인터의 사용
  - 초기화되지 않은 변수에는 쓰레기 값(garbage value)가 저장되어 있다.
  - **이때 포인터에 변수참조연산자(*)를 사용하면 실행오류가 발생할 수 있다.**

※ 주의
다행이 포인터에 저장된 쓰레기 값이 접근이 허용된 메모리 영역의 주소 값일 경우, 포인터에 변수참조연산자(*)을 사용하면 실행오류가 발생하지 않을 수 있다.
하지만, 허용된 메모리 영역 외의 주소 값일 경우 실행오류가 발생하게 된다.

포인터에 쓰레기 값이 저장된 상태에서 변수참조연산자(*)를 사용하는 것은 당장에 실행 오류가 발생하지 않을 수도 있지만, 다른 시간대에 다른 환경에서 실행 오류가 발생할 수 있으므로 이는 매우 위험한 코딩 방식이다.

```
#include <stdio.h>

int main(void)
{
    int a;
    int* p; // garbage value

    *p = 3; // run-time error

    p = &a;
    *p = 3; // ok
    return 0;
}
```

포인터 p는 아무것도 가리키고 있지 않다.

포인터 p에 변수 a의 주소를 저장하면 p는 a를 가리키게 된다.

포인터 사용시 주의할 점

잘못된 포인터의 사용

```
#include <stdio.h>

int main(void)
{
    char *p;          // oops

    scanf("%s", p );  // run-time error
    return 0;
}
```

바람직한 사용

```
#include <stdio.h>

int main(void)
{
    char a[512];      // good

    scanf("%s", a );  // ok
    return 0;
}
```

포인터 사용시 주의할 점

- 블록의 실행이 끝나면서 포인터가 이미 사라진 변수를 가리키는 경우
  - 포인터가 이미 **사라진 변수의 주소를 저장**하고 있는 것은 문제가 되지 않는다.
  - 다만, 이 포인터 변수에 **변수참조연산자(*)를 사용**하면 **실행 오류**가 발생할 수 있다.

```
#include <stdio.h>

int main(void)
{
    int* p;

    {
        int a;

        p = &a;
        *p = 99;   // ok
    }

    *p = 3;        // run-time error
    return 0;
}
```

내부 블록의 지역 변수

블록이 끝나면 변수 a는 사라진다.

사라진 변수를 가리키는 포인터 p

포인터 사용시 주의할 점

- 함수 호출이 끝나면서 포인터가 이미 사라진 변수를 가리키는 경우
  - 포인터가 이미 **사라진 변수의 주소를 저장**하고 있는 것은 문제가 되지 않는다.
  - 다만, 이 포인터 변수에 **변수참조연산자(*)를 사용**하면 **실행 오류**가 발생할 수 있다.

```
#include <stdio.h>

int* p;

void funct(void)
{
    int a;

    p = &a;
    *p = 99;   // ok
}

int main(void)
{
    funct();

    *p = 3;    // run-time error
    return 0;
}
```

내부 블록의 지역 변수

함수가 끝나면 변수 a는 사라진다.

사라진 변수를 가리키는 포인터 p

Call by Value

- C언어의 인자 전달 방식
  - Call by Value
    - 값을 전달하는 방식

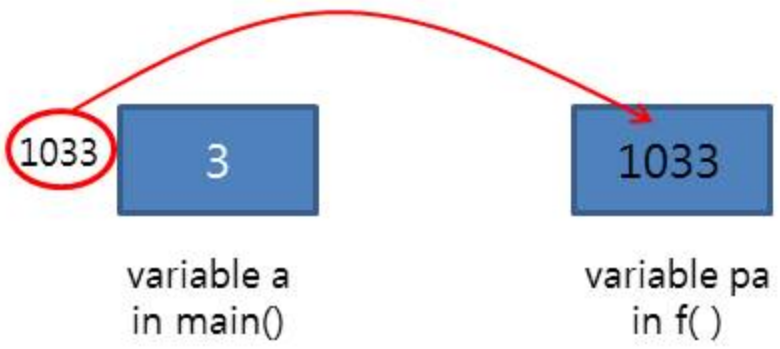
입출력 결과

```
4
3
계속하려면 아무 키나 누르십시오 . . .
```

```
#include <stdio.h>

void f(int a)
{
    ++a;
    printf("%d\n",a);
}

int main(void)
{
    int a=3;

    f(a);
    printf("%d\n",a);
    return 0;
}
```

Call by Address

- C언어의 인자 전달 방식
  - Call by Address
    - 주소값을 전달하는 방식



입출력 결과

```
4
4
계속하려면 아무 키나 누르십시오 . . .
```

```
#include <stdio.h>

void f(int* pa)
{
	++(*pa);
	printf("%d\n",*pa);
}

int main(void)
{
	int a=3;

	f(&a);
	printf("%d\n",a);
	return 0;
}
```

Call by Address

- 주소 전달의 예
  - scanf 함수에서는 값을 저장할 변수의 주소를 인자로 전달해야 한다.
  - 배열의 경우 배열의 주소를 인자로 전달한다.
    - 배열의 주소는 첫 번째 요소의 주소와 동일하다.

입출력 결과

```
1
1
2
2
3
3
계속하려면 아무 키나 누르십시오 . . .
```

```c
#include <stdio.h>

int main(void)
{
    int a;
    int b[3]={99,99,99};

    scanf("%d",&a);
    printf("%d\n",a);

    scanf("%d",b);    // b → &b[0]
    printf("%d\n",b[0]);

    scanf("%d",b+1);  // b+1 → &b[0]+1
    printf("%d\n",b[1]);
    return 0;
}
```

Call by Address

- 주소 전달의 예
  - scanf 함수에서 주소를 전달할 때 주의할 점
    - 주소를 전달할 때 주소 값의 참조형(reference type)을 고려해야 한다.

입출력 결과

```
0
0
2
99
계속하려면 아무 키나 누르십시오 . . .
```

```
#include <stdio.h>

int main(void)
{
    int a;
    int b[3]={99,99,99};

    scanf("%d",&b);      // undesirable
    printf("%d\n",b[0]);

    scanf("%d",&b+2);    // run-time error
    printf("%d\n",b[2]);
    return 0;
}
```

T(&b) == int[3]*
&b+2 == address(&b) + sizeof(int[3])*2
== address(&b) + 24

Address Return

- 잘못된 포인터 반환
  - 경우에 따라 원하는 동작할 수도 있지만, 예측할 수 없는 결과가 나오기도 한다.
  - 경우에 따라 **실행오류(run-time error)**가 발생할 수 있다.

```
#include <stdio.h>

int* f(int a)
{
    ++a;
    return &a;
}

int main(void)
{
    int a=3;
    int* p;

    p = f(a);
    printf("%d\n",*p);
    return 0;
}
```

이미 사라진 변수의 주소 값 반환

Address Return

- 바람직한 포인터 반환
  - Global, File-scope, Block-scope 변수는 포인터 반환이 가능하다.
  - 이유는 Data Segment 영역에 있으므로, 프로그램이 종료될 때까지 변수가 사라지지 않기 때문이다.

입출력 결과

```
4
계속하려면 아무 키나 누르십시오 . . .
```

```c
#include <stdio.h>

int* f(int a)
{
	static int b; /* block-scope */
	b=++a;
	return &b;
}

int main(void)
{
	int a=3;
	int* p;

	p=f(a);
	printf("%d\n",*p);
	return 0;
}
```

Address Return

Global Variable의 사용

```
#include <stdio.h>

int b; // global variable

int* f(int a)
{
      b=++a;
      return &b;
}

int main(void)
{
      int a=3;
      int* p;

      p=f(a);
      printf("%d\n",*p);
      return 0;
}
```

File-scope Variable의 사용

```
#include <stdio.h>

static int b; // file-scope variable

int* f(int a)
{
      b=++a;
      return &b;
}

int main(void)
{
      int a=3;
      int* p;

      p=f(a);
      printf("%d\n",*p);
      return 0;
}
```

논리 연산자

- 논리 연산자의 실행 순서
  - 연산 결과가 결정되면 나머지 부분은 더 이상 계산하지 않는다.

```
#include <stdio.h>

int f(int n)
{
    printf("f(%d)==%d\n",n,n-1);
    return n-1;
}

int main(void)
{
    if(f(1) && f(2)) printf("true\n");
    else printf("false\n");

    if(f(2) && f(3)) printf("true\n");
    else printf("false\n");

    if(f(1) || f(2)) printf("true\n");
    else printf("false\n");

    if(f(2) || f(3)) printf("true\n");
    else printf("false\n");

    return 0;
}
```

입출력 결과

```
f(1)==0
false
f(2)==1
f(3)==2
true
f(1)==0
f(2)==1
true
f(2)==1
true
계속하려면 아무 키나 누르십시오 . . .
```

논리 연산자

잘못된 논리 연산 순서

```
#include <stdio.h>

int main(void)
{
    int a[6]={3,2,6,8,10,9};
    int s=6;
    int value;
    int i;

    value=8;
    for(i=0; a[i]!=value && i<s ;++i)
        ;
    printf("index == %d\n",i);

    return 0;
}
```

i==s 일 경우
메모리 참조 오류 발생
(Run-time Error)

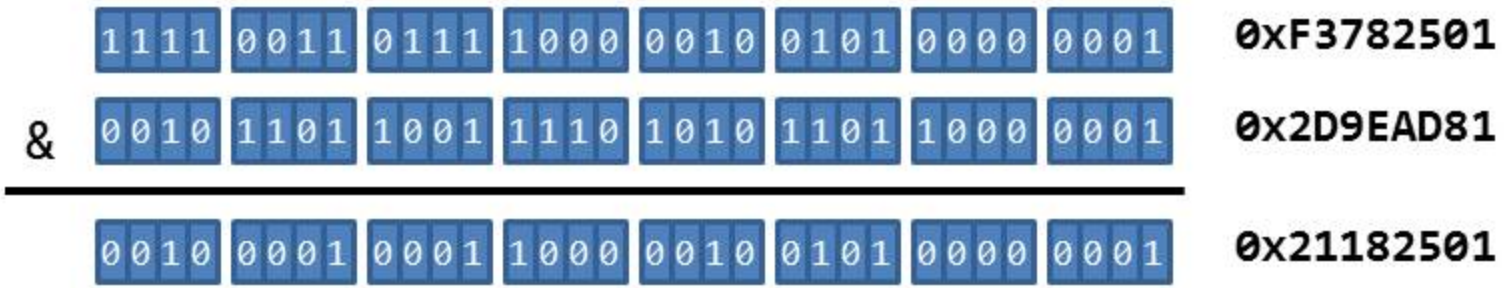
올바른 논리 연산 순서

```
#include <stdio.h>

int main(void)
{
    int a[6]={3,2,6,8,10,9};
    int s=6;
    int value;
    int i;

    value=8;
    for(i=0; i<s && a[i]!=value ;++i)
        ;
    printf("index == %d\n",i);

    return 0;
}
```

i==s 일 경우
이 부분 실행 안 함

Bitwise Operator

- 비트 AND



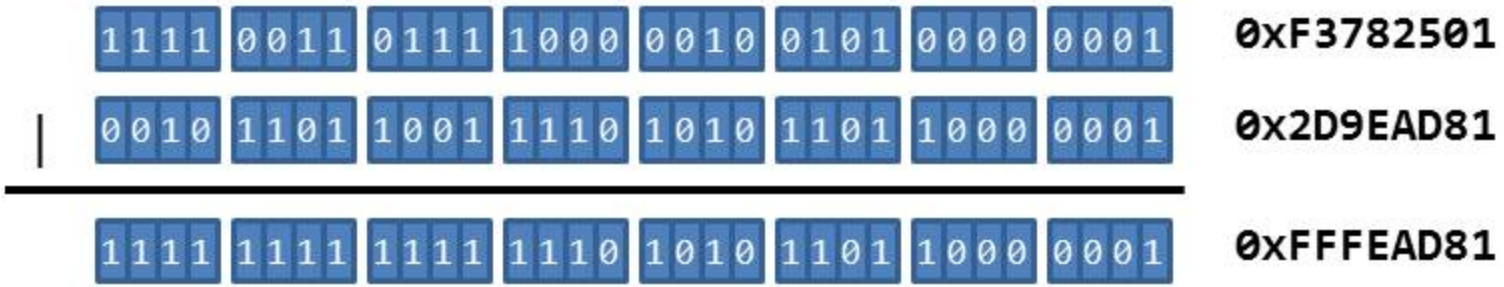
입출력 결과

```
21182501
계속하려면 아무 키나 누르십시오 . . .
```

```c
#include <stdio.h>

int main(void)
{
    int a,b;

    a = 0xF3782501;
    b = 0x2D9EAD81;
    printf("%08X\n", a & b );
    return 0;
}
```

Bitwise Operator

- 비트 OR



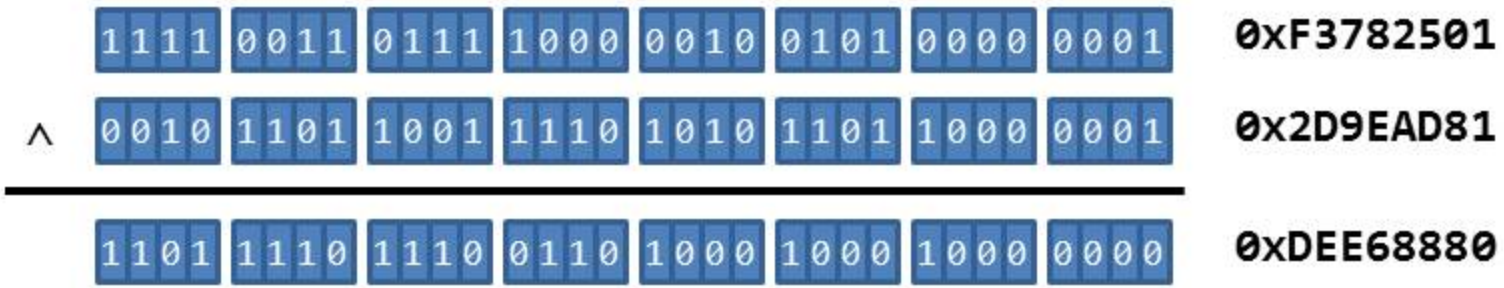
입출력 결과

```
FFFEAD81
계속하려면 아무 키나 누르십시오 . . .
```

```c
#include <stdio.h>

int main(void)
{
    int a,b;

    a = 0xF3782501;
    b = 0x2D9EAD81;
    printf("%08X\n", a | b );
    return 0;
}
```

Bitwise Operator

- 비트 XOR



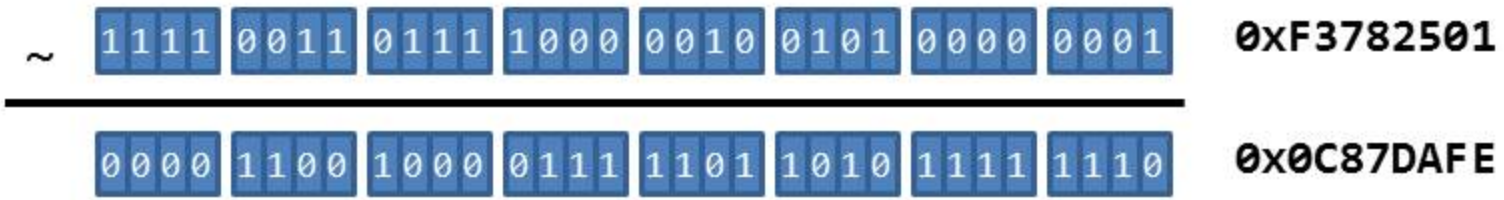
입출력 결과

```
DEE68880
계속하려면 아무 키나 누르십시오 . . .
```

```c
#include <stdio.h>

int main(void)
{
    int a,b;

    a = 0xF3782501;
    b = 0x2D9EAD81;
    printf("%08X\n", a ^ b );
    return 0;
}
```

Bitwise Operator

- 1's Complement



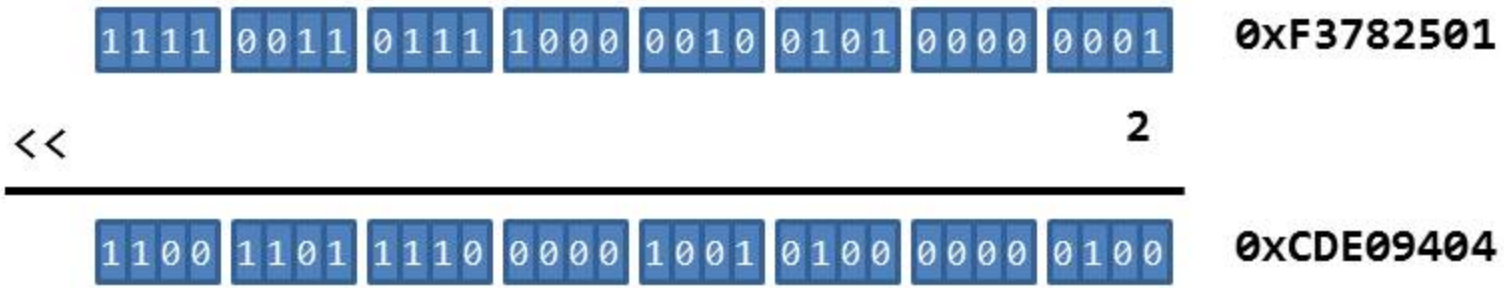
입출력 결과

```
0C87DAFE
계속하려면 아무 키나 누르십시오 . . .
```

```c
#include <stdio.h>

int main(void)
{
    int a;

    a = 0xF3782501;
    printf("%08X\n", ~a );

    return 0;
}
```

Bitwise Operator

- Left Shift



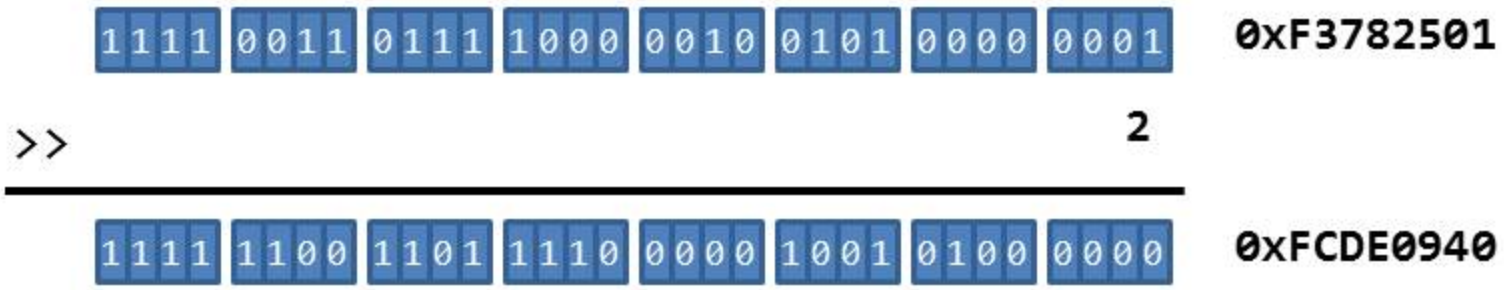
입출력 결과

```
CDE09404
계속하려면 아무 키나 누르십시오 . . .
```

```c
#include <stdio.h>

int main(void)
{
    int a;

    a = 0xF3782501;
    printf("%08X\n", a << 2 );

    return 0;
}
```

Bitwise Operator

- Right Shift (signed)



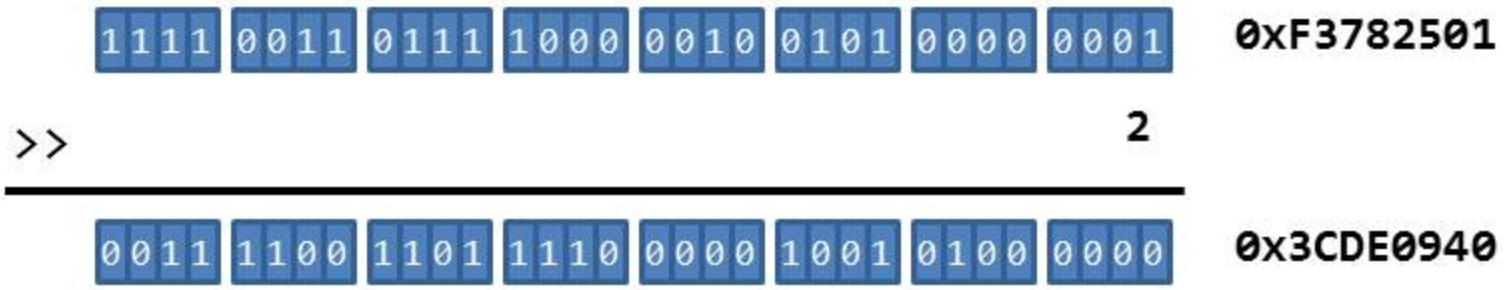
입출력 결과

```
FCDE0940
계속하려면 아무 키나 누르십시오 . . .
```

```c
#include <stdio.h>

int main(void)
{
    int a;

    a = 0xF3782501;
    printf("%08X\n", a >> 2 );

    return 0;
}
```

Bitwise Operator

- Right Shift (unsigned)



입출력 결과

```
3CDE0940
계속하려면 아무 키나 누르십시오 . . .
```

```c
#include <stdio.h>

int main(void)
{
    unsigned int a;

    a = 0xF3782501;
    printf("%08X\n", a >> 2 );

    return 0;
}
```

Assignment Operator

- Assignment

대입연산자	의미
a &= b	a = a & b
a |= b	a = a | b
a ^= b	a = a ^ b
a <<= b	a = a << b
a >>= b	a = a >> b

입출력 결과

```
21182501
FFFEAD81
DEE68880
CDE09404
FCDE0940
계속하려면 아무 키나 누르십시오 . . .
```

```c
#include <stdio.h>

int main(void)
{
    int a;

    a   = 0xF3782501;
    a  &= 0x2d9EAD81;
    printf("%08X\n", a);

    a   = 0xF3782501;
    a  |= 0x2d9EAD81;
    printf("%08X\n", a);

    a   = 0xF3782501;
    a  ^= 0x2d9EAD81;
    printf("%08X\n", a);

    a   = 0xF3782501;
    a <<= 2;
    printf("%08X\n", a);

    a   = 0xF3782501;
    a >>= 2;
    printf("%08X\n", a);
    return 0;
}
```

Bit Mask

- Bit Mask의 용도
  - Bit의 선택적 제거 또는 선택

Bit mask	결과	의미
x | 1	1	1로 제거
x & 1	x	선택
x | 0	x	선택
x & 0	0	0로 제거

```
#include <stdio.h>

int main(void)
{
    int a;

    a = 0xF3782501;
    printf("%08X\n", a);
    printf("%08X\n", a & 0x0000FFFF);
    printf("%08X\n", a | 0x0000FFFF);
    return 0;
}
```

입출력 결과

```
F3782501
00002501
F378FFFF
계속하려면 아무 키나 누르십시오 . . .
```

곱셈, 나눗셈

- 비트 연산의 활용
  - 곱셉

    $a << n == a * 2^n$

  - 나눗셈

    $a >> n == a / 2^n$

  - 장점
    - 연산속도가 빠르다.

입출력 결과

```
F3782501 -210230015
E6F04A02 -420460030
E6F04A02 -420460030
F9BC1280 -105115008
F9BC1281 -105115007
계속하려면 아무 키나 누르십시오 . . .
```

```c
#include <stdio.h>

int main(void)
{
    int a;

    a = 0xF3782501;
    printf("%08X %d\n", a, a);
    printf("%08X %d\n", a << 1, a << 1);
    printf("%08X %d\n", a  * 2, a  * 2);
    printf("%08X %d\n", a >> 1, a >> 1);
    printf("%08X %d\n", a  / 2, a  / 2);
    return 0;
}
```

연산자

연산자 우선 순위 (최종)

	연산자		결합 순서
함수/배열/구조체 연산자	( ) [ ] -> .		left → right
단항 연산자	* & (type) sizeof	! ~ ++ -- + -	right → left
산술 연산자	* / %		left → right
	+ -		left → right
비트 연산자(이동)	<< >>		left → right
관계 연산자	< <= > >=		left → right
	== !=		left → right
비트 연산자(논리)	&		left → right
	^		left → right
	|		left → right
논리 연산자	&&		left → right
	||		left → right
조건 연산자	? :		right → left
대입 연산자	= += -= *= /= %=	&= ^= |= <<= >>=	right → left
콤마 연산자	, (comma operator)		left → right

복잡한 타입의 재정의

```c
#include <stdio.h>

typedef int M[2][3];

void f(M m)
{
    printf("%d in f()\n", sizeof(M) );
    printf("%d in f()\n", sizeof(m) );
}

int main(void)
{
    M a = {
        {1,2,3},
        {4,5,6}
    };
    f( a );
    printf("%d in main()\n", sizeof(M) );
    printf("%d in main()\n", sizeof(a) );
    return 0;
}
```

T(M) == int[3][2]

int[3][2] --> int[3]*

T(m) == int[3]*

T(a) == int[3][2]

int[3][2] --> int[3]*

입출력 결과

```
24 in f()
4 in f()
24 in main()
24 in main()
계속하려면 아무 키나 누르십시오 . . .
```

```c
#include <stdio.h>

typedef int F(int);
typedef F* PF;

int f1(int j) { return 1+j; }
int f2(int j) { return 9-j; }

void print(PF pf1,PF pf2, int n)
{
    int n1,n2,i,j;

    for(j=0;j<10;j=j+1) {
        n1=pf1(j);
        n2=pf2(j);
        for(i=0;i<n1;i=i+1) printf("* ");
        for(i=0;i<n2;i=i+1) printf(". ");
        printf("\n");
    }
}

int main(void)
{
    print(f1,f2,10);
    return 0;
}
```

T(F) == int(int)

T(PF) == int(int)*

T(pf1) == T(pf2) == int(int)*

T(f1) == T(f2) == int(int)

int(int) --> int(int)*

복잡한 타입의 재정의

```
#include <stdio.h>

typedef int F(int);
typedef F* PF;
typedef PF PFL[2];

int f1(int j) { return 1+j; }
int f2(int j) { return 9-j; }

void print(PFL pf_l, int n)
{
    int n1,n2,i,j;

    for(j=0;j<10;j=j+1) {
        n1=pf_l[0](j);
        n2=pf_l[1](j);
        for(i=0;i<n1;i=i+1) printf("* ");
        for(i=0;i<n2;i=i+1) printf(". ");
        printf("\n");
    }
}
```

T(F) == int(int)
T(PF) == int(int)*
T(PFL) == int(int)*[2]

int(int)*[2] --> int(int)**
T(pf_l) == int(int)**

```
int main(void)
{
    PFL pf_list = { &f1, &f2 };

    print(pf_list,10);
    return 0;
}
```

T(pf_list) == int(int)*[2]

T(&f1) == T(&f2) == int(int)*

입출력 결과

```
*
* *
* * *
* * * *
* * * * *
* * * * * *
* * * * * * *
* * * * * * * *
* * * * * * * * *
* * * * * * * * * *
계속하려면 아무 키나 누르십시오 . . .
```