



8. 연산자(operator)

- 연산자는 다음과 같은 범주로 구분할 수 있습니다.

산술(Arithmetic)

할당(Assignment)

비트(Bitwise)

C++에서만 사용가능(C++ specific)

콤마(Comma)

조건(Conditional)

논리(Logical)

후위표기(Postfix)

전처리(Preprocessor)

참조/역참조(Reference/Dereference)

관계(Relational)

sizeof

형 변환(casting)



- C++에서 아래의 연산자들은 **오버로드(overload)** 될 수 없습니다.

. C++ 직접 요소 선택(direct component selector)

* C++ 재참조(dereference)

:: C++ 범위 접근/해결(scope access/resolution)

?: 조건 연산자(Conditional)

-
- 앰퍼샌드ampersand(&)는 다음과 같이 해석될 수 있습니다.
 - (1) 비트 AND 연산자
 - (2) 주소(address-of) 연산자
 - (3) C++에서 참조에 의한 호출 변경자(reference modifier)
 - 2개의 심벌이 1개의 연산자 단위를 이룰 경우, 2개의 심벌 사이에 공백이 들어가면 안됩니다.



산술(Arithmetic)

- + 단항 부호연산자, 양수positive number를 의미합니다.
- 단항 부호연산자, 음수negative number를 의미합니다.
- + 덧셈addition 연산자
- 뺄셈subtraction 연산자
- * 곱셈multiplication 연산자
- / 나눗셈division 연산자
- % 나머지modulo 연산자
- ++ 단항 전위prefix 증가increment 연산자
- ++ 단항 후위postfix 증가 연산자
- 단항 전위 감소decrement 연산자
- 단항 후위 감소 연산자

-
- 가감승제 연산이 피연산자의 타입type에 따라 다른 연산을 한다는 것에 주의하세요. $2+3$ 과 $2.0+3.0$ 은 다릅니다.

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

```
void main() {  
    int i=31,j=39;  
    int k=10;  
    printf("%d,%d\n", i/k, j/k);  
    //      3,3  
}
```

- 정수 나눗셈의 결과는 소수점 이하는 항상 무시됩니다. 결과가 3.1이든 3.9이든 3이 됩니다.

-
- 나머지 연산자 %는 나눗셈의 나머지를 구합니다. $i=10/3$ 의 결과 i 는 3입니다. 하지만, $i=10\%3$ 의 결과 i 는 1입니다.
 - 아래의 예는 1에서 100사이의 정수 중 3의 배수를 출력합니다.

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

```
void main() {  
    int i=0;  
  
    while ((++i)<=100)  
        if (i%3==0) printf("%d, ", i);  
}
```

-
- 증감 연산자는 대입 연산자 =처럼 **부효과side effect**가 있습니다.

`++i`

- 위 문장은 i의 값을 1증가시킵니다. i의 값이 변하므로 부효과가 있습니다. 위 문장은 다음 문장과 동일합니다.

`i++`

- i를 1증가시키는 문장을 정리해 보도록 하겠습니다.

`i=i+1;`

`++i;`

`i++;`

`i+=1;`

“전위 증감 연산자는 식을 평가하기 전에 먼저 증감합니다. 후위 증감 연산자는 식을 평가한 후 증감됩니다.”

- $i=2, j=3$ 인 경우, $i=i+(++j)$ 와 $i=i+(j++)$ 는 다릅니다. 전자인 경우 $i=6$ 입니다. 후자인 경우 $i=5$ 입니다. 물론 둘 다 $j=4$ 입니다.

- ① 전위 연산자를 위해 코드를 생성하고,
- ② 표현식을 위해 코드를 생성한 다음,
- ③ 후위 연산자를 위해 코드를 생성합니다.

$$i=i+(++j)-(k--)+(j++)+k;$$

- 위의 문장은 다음과 같이 번역됩니다.

- ① $j=j+1;$
- ② $i=i+j-k+j+k;$
- ③ $k=k-1; j=j+1;$



할당(Assignment)

= 간단한 할당 연산자

*= /= %= += -= <<= >>= &= ^= |=

- 우리는 할당문의 **왼쪽에 있는 변수(left value: l-value)**를 해석하는 방법과 **오른쪽에 있는 변수(right value: r-value)**를 해석하는 방법이 다르다는 것에 주목할 필요가 있습니다.
- $i=2, j=3$ 일 때 아래 문장을 어떻게 해석할까요?

$i=j;$

-
- 왼쪽에는 항상 주소값을 구할 수 있는 변수가 위치해야 합니다.

$3=j;$

- 이러한 문장은 컴파일러에 의해 다음의 에러메시지를 발생합니다.

"L-value required"

- ++1같은 표현식의 경우도 마찬가지입니다. 컴파일러는 이 문장을 $1=1+1$ 로 해석하려고 하기 때문에 L-value가 틀렸다는 에러메시지를 출력합니다.

복잡한 할당문의 원리

$i = i + j * k;$

$i = i + j * k;$

$i = +j * k;$



$i += j * k;$

- 할당문도 표현식임에 주의하세요. $i = j;$ 라는 할당문은 j 의 값을 i 에 대입할 뿐만 아니라, $i = j$ 라는 문장 자체가 i 의 값을 가집니다.



비트(Bitwise)

- & 비트 논리곱(bitwise AND) 연산자
- | 비트 논리합(bitwise OR) 연산자
- ^ 비트 배타적 논리합(bitwise exclusive OR) 연산자
- ~ 단항 비트 논리부정(bitwise NOT) 연산자
- << 비트 왼쪽 쉬프트(bitwise shift left) 연산자
- >> 비트 오른쪽 쉬프트(bitwise shift right) 연산자

-
- 각 비트 연산자의 역할은 다음과 같습니다.

& bitwise AND: 두 비트가 모두 1이면, 1입니다. 그 외의 경우는 0입니다.

| bitwise inclusive OR: 두 비트가 모두 0이면, 0입니다. 그 외의 경우는 1입니다.

^ bitwise exclusive OR: 1의 수가 홀수 개이면, 1입니다. 그 외의 경우는 0입니다■.

~ bitwise complement: 단항 연산자이므로 피연산자가 1개입니다. 0과 1을 토글(toggle)합니다. 즉 1의 보수를 구합니다.

>> bitwise shift right: 비트열(bit sequence)을 오른쪽으로 이동(shift)합니다. 빈 자리에는 0혹은 1로 채워집니다■.

<< bitwise shift left: 비트열을 왼쪽으로 이동합니다. 빈 자리에는 0으로 채워집니다.

-
- 다음의 진리값 truth value 테이블을 참조하세요.

E1	E2	E1 & E2	E1 E2	E1 ^ E2
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0



비트 연산자의 진리표

-
- 아래 프로그램의 결과를 계산해 보세요.

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

```
void main() {
```

```
    int i=2, //0000 0000 0000 0010
```

```
        j=3, //0000 0000 0000 0011
```

```
        k=5; //0000 0000 0000 0101
```

```
    printf("%d,%d,%d,%d,%d\n", i&j, i|j, i^j^k, i<<2, k>>1);
```

```
    //      2  3  4  8  2
```

```
}
```

-
- 쉬프트shift 연산자는 이진수의 특성상 2의 지수 곱셈에 대한 특별한 의미를 가집니다.

$$i << n$$

- 위 표현식은 $i * 2^n$ 과 동일합니다(오버플로우overflow는 고려하지 않았습니다).

$$i >> n$$

- 위 문장은 $i / 2^n$ 과 동일합니다(언더플로우underflow는 고려하지 않았습니다).

-
- 우리는 또한 &와 |의 **비트 마스크(bit mask)**기능에 주목할 필요가 있습니다.
 - (1) bit set: 비트열의 특정한 부분을 1로 만듭니다.
 - (2) bit clear: 비트열의 특정한 부분을 0으로 만듭니다.
 - 아래의 예제는 16진 정수 비트열의 11,10,9,8 위치의 비트를 0으로 지우고, 7,6,5,4 위치의 비트를 1로 설정합니다.

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

```
void main() {  
    unsigned int i=0x1234;//0001 0010 0011 0100  
  
    printf("%x\n", i&0xf0ff);  
    //      1034  
    printf("%x\n", i|0x00f0);  
    //      12f4  
    printf("%x\n", i&0xf0ff|0x00f0);  
    //      10f4  
}
```

-
- ^ 연산자는 특정한 숫자를 토글toggle시키기 위해 사용할 수 있습니다■.

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

```
void main() {  
    char i=3;  
    printf("%d\n", ~i);  
    //      -4  
}
```



C++에서만 사용가능(C++ specific)

- 아래와 같은 연산자들은 C++에서만 사용할 수 있습니다.

:: 범위 접근/해결 연산자(Scope access (or resolution) operator)

.* 클래스 멤버의 포인터의 재참조(Dereference pointers to class members)■

->* 클래스 포인터 멤버의 포인터의 재참조(Dereference pointers to pointers to class members)

**new**

- C++에서는 여러 가지 복잡한 기능을 지원해야 하므로 새로운 **동적 메모리 할당 (dynamic memory allocation) 연산자 new**가 추가되었습니다.
- C에서는 동적으로 메모리를 할당하기 위해, 주로 malloc()이라는 표준 함수를 이용했습니다. 예를 들어 10개의 short 정수를 저장하기 위해서, 메모리를 할당하는 경우 - 20바이트 할당 - 다음과 같은 문장을 사용할 수 있습니다.

```
i=(short*)malloc(sizeof(short)*10);
```

- i는 short *로 선언되어 있어야하며, 포인터 연산의 정확성을 보장하기 위해, (short *)를 사용하여, 형 변환(casting)을 해주는 것이 필요합니다.

```
free(i);
```

- 사용에서 명시적인(explicit) 형 변환(type conversion)이 왜 필요한지 알아보시다. 아래 프로그램의 출력 결과는 얼마일까요?

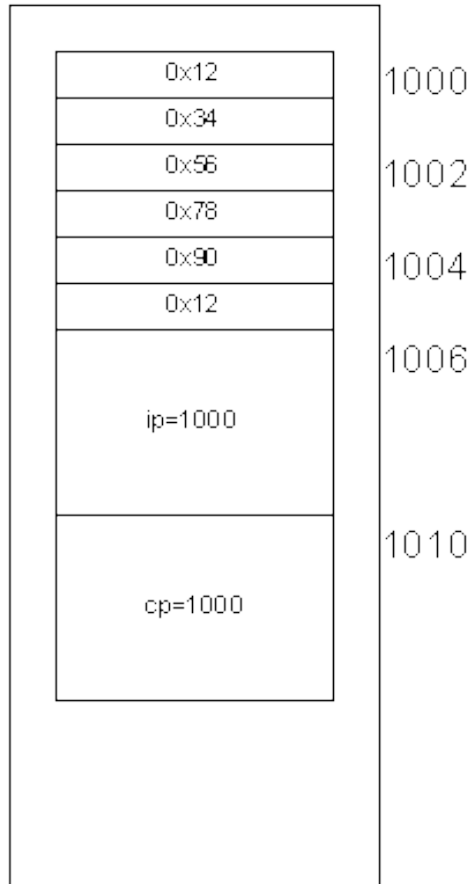
```
#include <stdio.h>
#include <stdlib.h>

void main() {
    short a[]={0x1234,0x5678,0x9012};
    short *ip=a;
    char *cp=(char *)a;

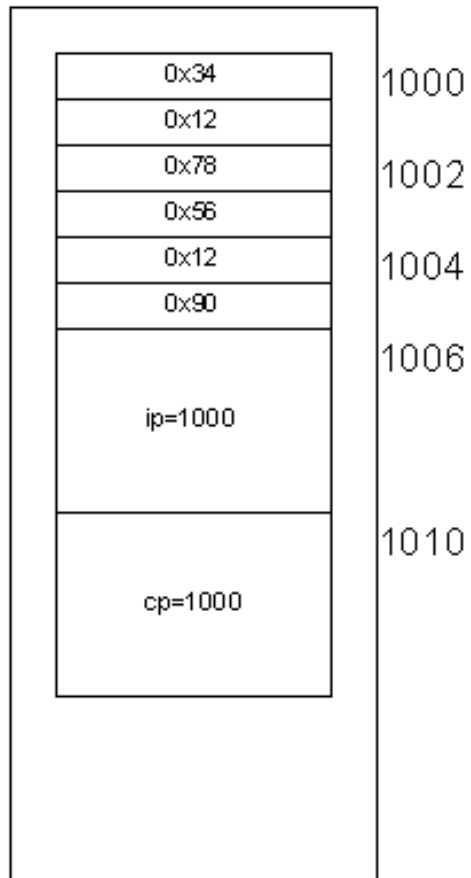
    printf("%x,%x\n",*(ip+1),*(cp+1));
    //      5678,12
} //main
```

- 출력결과는 놀랍게도 5678,12 입니다.

- 인텔 CPU의 **역워드(inverted word)** 구조에 대해서 이해해야 합니다.



- 인텔 호환 마이크로 CPU를 사용하는 PC인 경우, 실제의 메모리 그림은 아래와 같습니다.



-
- ip와 cp모두 1000임에는 틀림없습니다. 하지만 차이점이 존재합니다. ip는 short 포인터 (short *)이므로, ip+1은 다음과 같이 해석한다.

“ip에서 1번째 떨어진 short 정수”

- ip+1은 1001이 아니라, 1002입니다. 또한 *(ip+1)도 1002번지의 내용이 아니라, 1002번지와 1003번지의 내용입니다. 그러므로 결과는 0x5678입니다.
- cp+1은 다음과 같이 해석합니다.

“cp에서 1번째 떨어진 1바이트 정수”

- cp+1은 1001이며, *(cp+1)은 0x12가 맞습니다.

-
- 일반적으로 포인터 ip의 증감 n의 의미는 다음과 같습니다.

$$ip+n \equiv ip+sizeof(*ip)*n$$

- 그러므로 ip가 int *라면, ip+2는 ip를 8증가시킵니다.

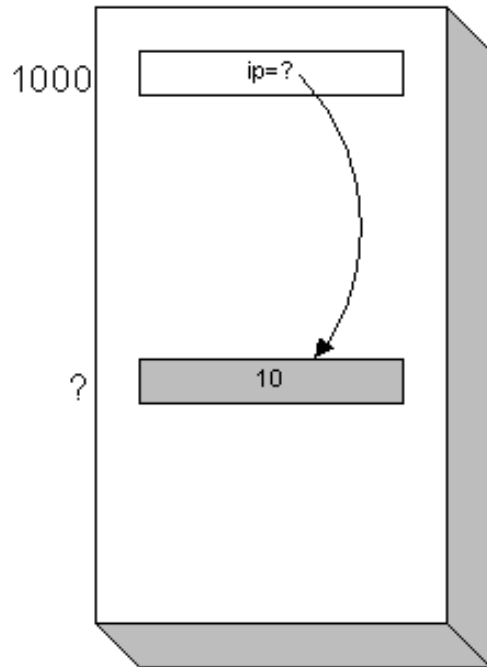
-
- 이제 **동적 메모리 할당**에 대해서 알아보시다. 아래의 프로그램 소스는 어디가 잘못일까요?

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

void main() {
    int *ip;

    *ip=10;
    printf("%d\n", *ip);
} //main
```

- 아래 그림을 보면 ip에는 의미없는 값이 들어 있는 것을 알 수 있습니다.



땡글링 포인터dangling pointer

-
- 동적 메모리 할당을 사용하도록, 아래와 같이 고쳐야 합니다.

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

void main() {
    int *ip;

    ip=(int *)malloc(4);
    *ip=10;
    printf("%d\n", *ip);
    free(ip);
} //main
```

-
- C++에서는 위의 소스를 다음과 같이 new/delete를 사용하여 구현 가능합니다.

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

```
void main() {  
    int *ip;  
  
    ip=new int;  
    *ip=10;  
    printf("%d\n",*ip);  
    delete ip;  
} //main
```

- stdlib.h를 포함하는 것은 더 이상 필요하지 않습니다.
- new에 더 이상의 형 변환이 필요하지 않습니다.
- new int 는 sizeof(int)크기의 메모리 4바이트를 할당하여 시작 주소를 리턴합니다.

-
- new의 문법은 다음과 같습니다.

[::]new <type>[(초기값)]

- new앞에 범위 해결사 ::는 옵션(option)입니다.
- 오버로딩된 new와 구분하기 위해서 ::new 처럼 사용할 필요가 발생합니다.
- 이것은 오버로딩된 new를 사용한다는 것이 아니라, 원래의 전역 new를 사용한다는 것을 보장해 줍니다.
- new 뒤에 적는 type은 생략해서는 안 됩니다. new type은 sizeof(type)만큼의 메모리를 할당해서 시작 주소를 리턴합니다.
- 아래의 문장은 4바이트를 할당합니다.

```
long *lp;  
lp=new long;
```

-
- delete는 할당된 크기에 상관없이, 다음과 같이 사용합니다.

```
delete ip;
```

- new는 메모리 할당과 동시에 초기화를 하는 것을 허락합니다.

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

```
void main() {
```

```
    int *ip;
```

```
    ip=new int(10);
```

```
    printf("%d\n", *ip);
```

```
    delete ip;
```

```
}//main
```

-
- 이제 10개의 int형 변수를 할당하기 위해서는 어떻게 해야 하는지 알아보시다.

`[::]new type'['표현식']`

- 표현식을 둘러싼 브래킷(bracket; [or])은 선택 사항을 나타내는 심벌이 아니라, 문법 심벌입니다.
- 아래의 문장은 정수 3개를 저장하기 위해서 12바이트의 메모리를 할당■합니다.

`new int[3]`

- 아래의 문장에서 ip는 12바이트의 시작주소를 가리킵니다.

`int *ip;`

`ip=new int[3];//이것은 정수 3개를 할당한다.`

-
- 기본 형 여러 개로 메모리를 할당한 경우 delete의 문법 또한 다릅니다. new int[3]처럼 할당된 메모리는 delete[] 로 해제해야 합니다.

```
int *ip;  
ip=new int[3];  
...  
delete[] ip;
```

-
- 아래의 예제를 참고하세요.

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

```
void main() {
```

```
    int *ip;
```

```
    ip=new int[3];
```

```
    *ip=1; *(ip+1)=2; *(ip+2)=3;
```

```
    printf("%d,%d,%d\n",*ip, *(ip+1), *(ip+2));
```

```
    delete[] ip;
```

```
}//main
```

-
- 기본 형 여러 개로 메모리를 할당하면서 초기화하는 다음과 같은 문장은 허락되지 않습니다.

```
ip=new int[3](1,2,3);
```

- 후에 배열 연산자 오버로딩을 이용하면, 아래와 같은 문장이 가능하도록 코드를 작성할 수 있습니다. 이때 new는 연산자는 적절히 오버로딩 되어야 합니다.

```
ip=new(1,2,3) int[3];
```

-
- 우리는 내용 연산자(contents-of operator) []에 대해 알고 있습니다.
 - *(ip+n)은 ip[n]과 동일합니다.

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

```
void main() {
```

```
    int *ip;
```

```
    ip=new int[3];
```

```
    ip[0]=1; ip[1]=2; ip[2]=3;
```

```
    printf("%d,%d,%d\n",ip[0], ip[1], ip[2]);
```

```
    delete[] ip;
```

```
}//main
```

-
- new를 사용하여 2차원처럼 사용할 수 있는 메모리를 할당할 수 있습니다.

```
#include <iostream>
```

```
void display(long double **);
```

```
void de_allocate(long double **);
```

```
int m = 3;
```

```
    // THE NUMBER OF ROWS.
```

```
int n = 5;
```

```
    // THE NUMBER OF COLUMNS.
```

```
int main(void) {
```

```
    long double **data;
```

```
    data = new long double*[m];           // STEP 1: SET UP THE ROWS.
```

```
    for (int j = 0; j < m; j++)
```

```
        data[j] = new long double[n];    // STEP 2: SET UP THE COLUMNS
```

```
    for (int i = 0; i < m; i++)
        for (int j = 0; j < n; j++)
            data[i][j] = i + j;                // ARBITRARY INITIALIZATION

    display(data);
    de_allocate(data);
    return 0;
}

void display(long double **data) {
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++)

            cout << data[i][j] << " ";
        cout << "\n" << endl;
    }
}

void de_allocate(long double **data) {
```

```
for (int i = 0; i < m; i++)
    delete[] data[i];                // STEP 1: DELETE THE COLUMNS

delete[] data;                        // STEP 2: DELETE THE ROWS
}
```

-
- C++11의 **초기화 리스트initialization list**를 사용하면 new[]로 메모리를 할당 할 때, 각각의 초기화를 지정하는 것이 가능합니다.
 - Visual Studio 2013이상에서 이 문법을 지원합니다.

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

```
void main()
```

```
{
```

```
    int *ip;
```

```
    ip = new int[ 3 ]{1,2,3};
```

```
    printf( "%d,%d,%d\\n", *ip, *( ip + 1 ), *( ip + 2 ) );
```

```
    delete[] ip;
```

```
}//main
```

- 초기화 리스트의 문법은 new type{}입니다. {와 }안에 할당하는 단위 개수 만큼의 초기값을 적습니다.



delete

- delete는 아래의 두 가지 문법이 있습니다.

[::]delete <pointer>;

[::]delete[] <pointer>;



typeid

- C++의 typeid를 사용하면 실행시간에 표현식이나 형의 형 정보type information를 구할 수 있습니다.
- typeid는 type_info라는 클래스의 상수 참조를 리턴합니다.
- typeid를 사용하기 위해서는 typeid.h를 포함해야 합니다.

```
#include <iostream>
#include <string>
#include <typeinfo>
```

```
struct Base {}; // non-polymorphic
struct Derived : Base {};
```

```
struct Base2 { virtual void foo() {} }; // polymorphic
struct Derived2 : Base2 {};
```

```
int main()
{
    int myint = 50;
    std::string mystr = "string"
    double *mydoubleptr = nullptr

    std::cout << "myint has type: " << typeid(myint).name() << '\n'
        << "mystr has type: " << typeid(mystr).name() << '\n'
        << "mydoubleptr has type: " << typeid(mydoubleptr).name() << '\n'

    // Non-polymorphic lvalue is a static type
    Derived d1;
    Base& b1 = d1;
    std::cout << "reference to non-polymorphic base: " << typeid(b1).name() << '\n'

    Derived2 d2;
    Base2& b2 = d2;
    std::cout << "reference to polymorphic base: " << typeid(b2).name() << '\n'
}
```

-
- 출력결과는 다음과 같습니다.

```
myint has type: int
```

```
mystr has type: class std::basic_string<char,struct std::char_traits<char>,class  
std::allocator<char> >
```

```
mydoubleptr has type: double *
```

```
reference to non-polymorphic base: struct Base
```

```
reference to polymorphic base: struct Derived2
```

- typeid가 동작하기 위해서는 프로젝트 설정에서 실행시간 형식 정보를 사용하도록 설정해 주어야 합니다.

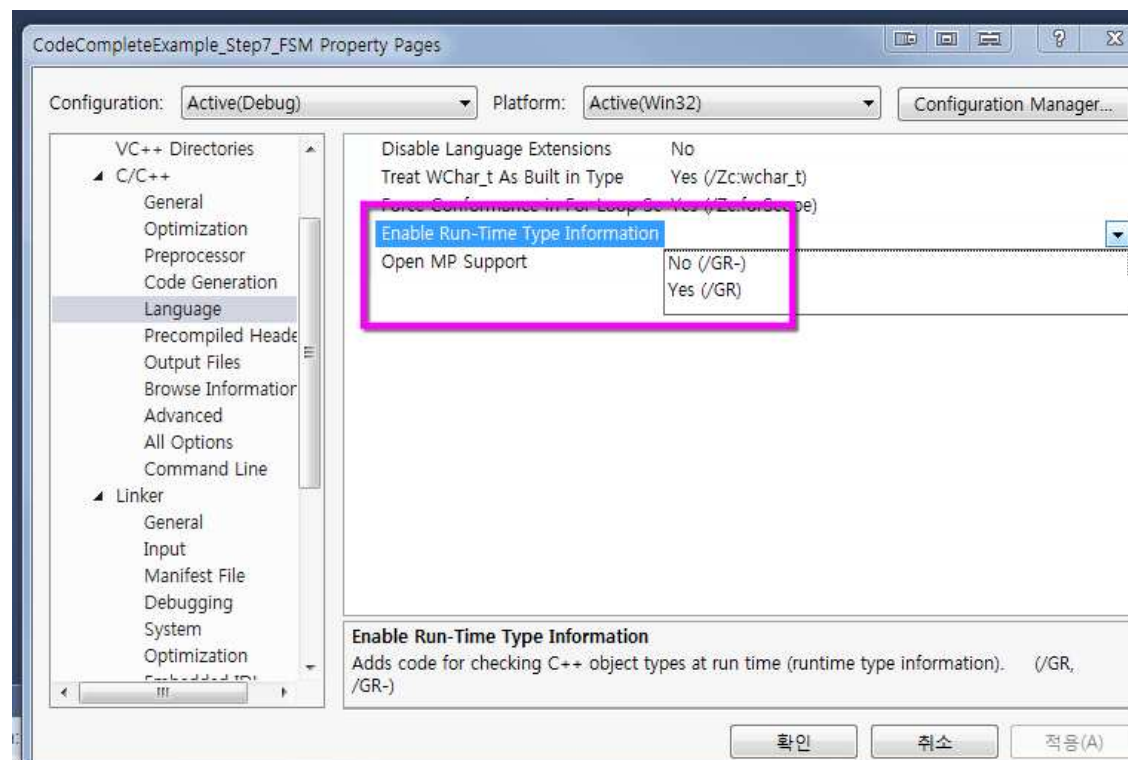


그림: Visual Studio 2013의 런타임 형식 정보 활성화 설정



콤마(Comma)

- 콤마 연산자의 문법은 다음과 같습니다.

표현식[표현식][...]

- 표현식은 콤마에 의해 계속 연결될 수 있으며, 콤마에 의해 연결된 표현식은 하나의 문장입니다.
- 문장의 값은 마지막 표현식의 값이 됩니다. 그러므로 다음 문장의 값은 3입니다.

1,2,3

- 아래의 소스는 3을 i에 할당합니다.

i=(1,2,3);

-
- 문법에서도 언급했듯이, 콤마 연산자로 연결할 수 있는 것은 표현식만입니다. 아래의 소스의 결과에 주목하세요.

```
#include <stdio.h>

void main() {
    int i;

    printf("%d\n", (1,2,3)); //값은 3입니다.
    //      3
    i=(3,2,1); //마지막 값인 1이 i에 할당됩니다.
    printf("%d\n", i);
    //      1
} //main
```

- 표현식만으로 연결되어야 하므로, 다음 문장은 에러입니다.

```
i=1, int j, k=2;
```

-
- i를 1부터 10까지 변화시키면서, j를 5부터, 50까지 5의 배수로 크기를 변화시켜야 한다고 생각해 봅시다. 우리는 for문을 사용하여 소스를 다음과 같이 작성할 수 있습니다.

```
j=5;
for (i=1;i<=10;++i) {
    ...
    j+=5;
} //for
```

- 콤마 연산자를 사용하여 다음과 같이 소스를 수정할 수 있습니다.

```
for (i=1,j=5;i<=10;++i,j+=5)
    ...
```


-
- 아래 프로그램의 결과를 예측해 보세요.

```
#include <stdio.h>

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

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

    f((i=1,j=2),k=3);
}
```

- 결과는 2,3이 출력됩니다.



조건(Conditional)

- 조건 연산자의 문법은 다음과 같습니다.

표현식0 ? 표현식1 : 표현식2

- 위 문장은 표현식0의 값이 0이 아니면, 즉 참(true)이면 표현식1의 값으로, 아니면 표현식2의 값으로 결정됩니다. 그러므로 아래 문장의 값은 2입니다.

0 ? 1 : 2

-
- 아래 프로그램의 결과는 5입니다.


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

void main() {
    int m,i=2,j=5,k=3;

    m=(i>j)?i:(j>k)?j:k;
    cout << m <<endl;
} //main
```

-
- 우리는 조건 연산자를 이용하여 MAX혹은 MIN이라는 매크로 함수(macro function)을 다음과 같이 만들 수 있을 것입니다.

```
#define MAX(a,b) ((a)>(b)?(a):(b))
```

 매크로 함수의 파라미터를 괄호로 감싸야 함에 유의하세요. 그렇게 하지 않는다면, 연산자 우선 순위에 의해 심각한 문제가 발생할 수 있습니다. 이것은 '20. 전처리 명령어'에서 상세히 다룹니다.

```
#define MIN(a,b) ((a)<(b)?(a):(b))
```



논리(Logical)

&& 논리곱(logical AND) 연산자

|| 논리합(logical OR) 연산자

! 단항 논리부정(logical NOT) 연산자

- 수학에서는 논리 연산자의 표현식이 논리 표현식이지만, C에서는 수치 표현식입니다.
- 논리 연산의 결과는 참/거짓이 아니라 수(number)입니다.

- 0을 거짓으로 보고, 1을 참으로 본다면, 진리표는 아래와 같습니다.

값		&&	
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1



논리 연산자의 진리표

값	!
0	1
1	0



논리 부정의 진리표

-
- 0은 거짓이지만, 0이 아니면 항상 참이라는 것에 주목하세요. 1은 참입니다.
 - 100도 참이며, -1도 참입니다.
 - !0은 참입니다.
 - !1, !100, !-1은 참을 부정하는 문장으로 간주하므로, 거짓(0)입니다.

-
- 아래 프로그램의 결과는 얼마일까요?

```
int i=2,j=3,k=4;
```

```
if (i<j && j<=k)  
    printf("logical operator\n");
```

- 문자열 "logical operator"는 출력됩니다. 왜 문자열이 출력되는가요?

"i<j는 관계 비교가 참이므로 1입니다. j<=k역시 1입니다. '1 AND 1'은 1이므로 printf()가 실행됩니다."

- if문이 실행된 이유는 괄호 안의 조건이 참이기 때문이 아니라, 괄호 안의 표현식이 1이기 때문입니다.
- C에서 참/거짓이란 값은 존재하지 않습니다.
- 0이면 조건 비교를 거짓으로 간주하고, 0이 아니면 참으로 간주합니다.

-
- 아래 프로그램의 출력결과는 1입니다.

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

```
void main() {
```

```
    int i=2, j=3, k=4;
```

```
    printf("%d\n", i<j && j<=k);
```

```
}//main
```

- “논리 연산자와 관계 연산자의 결과는 0 아니면 1입니다. 즉 숫자 표현식입니다.”

-
- 논리 연산자를 다룰 때, 주의 해야할 사항은 '**짧은 평가(short circuit)**'에 관한 것입니다.

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

```
void main() {
```

```
    int i=2,j=3,k=4;
```

```
    if (i<j || j<=k)//짧은 평가에 의해, j<=k는 평가하지 않습니다.
```

```
        printf("short circuit\n");
```

```
}//main
```

- 아래에 논리 연산자의 전체적인 예를 들었습니다.

```
#include <stdio.h>


void main() {
    int i=2,j=3,k=4;

    printf("%d\n",i && j);
    printf("%d\n",!i && !j);
    printf("%d\n",i>j || j==k);
    printf("%d\n",!(i>j));
    printf("%d\n",!(!(i<j)));
    printf("%d\n",!(j!=k));
} //main
```

- 1 0 0 1 1 0이 출력됩니다.

-
- 종종 **무한 루프(infinite loop)**를 만들어야 하는 경우가 생깁니다.

```
while(1) { ... }  
do { ... } while(1);  
for(;1;) { ... }■
```

 for(;;) { ... } 역시 무한 루프입니다. for문에서 생략된 비교 문장은 참을 의미합니다. 어떤 방법이든 for를 사용한 무한 루프는 직관적이지 않으므로 추천하지 않습니다.

- while(-1) { ... } 역시 무한 루프입니다.



후위표기(Postfix)

• C에서 연산자처럼 느껴지지 않는 몇몇 연산자들은 **후위표기법(postfix notation)**을 사용합니다.

() 표현식을 그룹화 하기 위해, 조건 표현식을 독립시키기 위해, 함수 호출을 위해, 함수의 파라미터를 지정하기 위해 사용합니다. **함수 호출 연산자(function call operator)**라 합니다.

[] 배열의 첨자(subscript)를 가리키기 위해, 포인터가 가리키는 곳의 내용을 참조하기 위해 사용합니다. **첨자 연산자** 혹은 **내용 연산자(contents-of operator)**라 합니다.

{ } 복합문(compound statement)의 시작과 끝을 나타내기 위해 사용합니다. 복합문은 하나의 문장 취급됩니다. 블록(**block**)이라고 합니다.

. 구조체(structure), 공용체(union)와 클래스(class)의 멤버를 접근(access)하기 위해 사용합니다. **멤버 연산자(member operator)**라 합니다.

-> 포인터로 선언된 구조체 등의 멤버를 접근하기 위해 사용합니다. 포인터 멤버 연산자(pointer member operator)라 합니다.

-
- **함수 호출 연산자 ()**는 함수 호출이라는 특별한 목적이외에도 문법 구조를 이루는 심벌로 자주 등장합니다.
 - $2+3*4$ 는 14입니다. 하지만 $(2+3)*4$ 는 20입니다.
 - 이것은 $2+3$ 이라는 것을 그룹화한 것을 의미합니다. ()는 연산자의 우선 순위가 가장 높습니다.

-
- 함수 호출 연산자에 대해 살펴봅시다. 아래의 소스를 보세요. 무엇이 잘못 되었는가요?

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

void main() {
    int i;

    printf("Press y key");
    i=getch;//0이 부분이 잘못된가요?
    if (i=='y')
        printf("You pressed small y key\n");
} //main
```

- 아마도 사용자는 getch의 뒤에 괄호 ()를 적는 것을 빠뜨린 것 같습니다.

-
- 함수 호출이 일어나기 위해서는 함수 호출 연산자를 함수의 시작 주소 뒤에 명시해 주어야 합니다. 수정된 소스는 아래와 같습니다.

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

void main() {
    int i;

    printf("Press y key");
    i=getch(); //함수 호출이 일어납니다.
    if (i=='y')
        printf("You pressed small y key\n");
} //main
```


-
- 여러 개의 문장이 모여 하나의 문장을 이룰 때 이를 **복합문(compound statement)**이라고 합니다.
 - `{ }` 는 복합문, 즉 블록 **구조(block structure)**를 만들기 위해 사용합니다.

```
int i=2,j=3;
```

```
if (i<j)  
    printf("i is greater than j.\n");
```

- `i<j` 가 참이므로, `printf()`를 실행합니다. 만약 괄호 안의 조건이 참이 될 때, 실행할 문장이 두 문장 이상이라면, 어떻게 할 것인가요?

```
if (i<j)  
    printf("i is greater than j.\n");  
    printf("it's joke.\n");
```

-
- 어떻게 이 문제를 해결할 것인가요? 바로 복합문 연산자, 즉 2개의 문장을 하나의 문장으로 만드는 것입니다.

```
if (i<j) {  
    printf("i is greater than j.\n");  
    printf("it's joke.\n");  
}
```

- 블록 구조(block structure)는 다음과 같은 특징이 있습니다.
 - (1) 하나의 문장 취급됩니다.
 - (2)-a 블록의 첫 부분에, 변수 선언을 가질 수 있습니다.(예전의 C언어)
 - (2)-b 블록 안에서는 어디나 변수 선언을 가질 수 있습니다.(C++언어)
 - [(3) 겹쳐질(nesting) 수 있습니다.]
- (3)번은 블록 안에 얼마든지 블록을 만들 수 있음을 의미합니다.

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

```
void main() {  
    printf("a");  
    {  
        printf("b");  
        {  
            printf("c");  
        }  
    }  
}
```

-
- . 와 -> 는 C에서 구조체/공용체의 **멤버 연산자**로 사용되었습니다. 물론 C++에서 클래스의 멤버를 접근하기 위해서 사용됩니다.
 - 구조체는 여러 개의 서로 다른 데이터 형(data type)을 **필드(field)**로 가지는 데이터 형입니다.
 - 클래스와의 호환성을 위해 필드란 말 대신 멤버란 말을 사용하기로 합니다.

```
#include <stdio.h>
//#include <string.h>
#include <iostream>

struct STest {
    int age;
    char name[80];
}; //struct STest

void main() {
    struct STest s={30, "Seo JinTaek"};

    cout << s.age << endl; //STest의 멤버 age를 참고하기 위해 . 을 사용한
                           //다.
    cout << s.name << endl;
} //main
```

- 구조체가 선언되지 않고 구조체를 가리키는 포인터가 선언된 경우 어떻게 할 것인가요?

```
#include <stdio.h>
#include <string.h>
#include <iostream>
```

```
struct STest {
    int age;
    char name[80];
}; //struct STest
```

```
void main() {
    struct STest *s;
```

```
    s=new STest; //포인터이므로 메모리 할당이 필요합니다.
```

```
    (*s).age=30;
```

```
    strcpy((*s).name, "seojt"); //(*s).name="seojt"는 왜 안되는가요?
```

```
    cout << (*s).age << endl;
```

```
    cout << (*s).name << endl;
```

```
    delete s;//new로 할당한 메모리는 반드시 delete합니다.  
}//main
```

- 우리는 `s`가 구조체가 아니라, `s`가 가리키는 내용, 즉 `*s`가 구조체라는 것을 알고 있습니다.
- 그러므로 구조체 `*s`의 멤버를 접근하기 위해 `.`을 사용하여, `(*s).age` 처럼 사용합니다.
- 연산자의 우선 순위 때문에 괄호가 반드시 필요합니다.
- `(*s).age`는 `s->age`와 동일합니다.
- 즉 `->` 는 구조체 포인터 변수에서 쉽게 멤버를 접근하기 위해 사용되는 연산자입니다.

```
#include <stdio.h>
#include <string.h>
#include <iostream>

struct STest {
    int age;
    char name[80];
}; //struct STest

void main() {
    struct STest *s;

    s=new STest;
    s->age=30;
    strcpy(s->name, "seojt");
    cout << s->age << endl;
    cout << s->name << endl;
    delete s;
} //main
```




전처리(Preprocessor)

스트링화 연산자

토큰 연결(token concatenation) 연산자

- #은 큰따옴표(")가 없는 문자 순서(string sequence)를 문자열로 만듭니다.

```
#define stringit(x) #x
```

- 위 문장을 이용해서 프로그램 소스에서 아래와 같이 사용하면,

```
stringit(Seo JinTaek)
```

- 컴파일 전에 "Seo JinTaek"라고 치환됩니다.

-
- ##은 두 개의 토큰(token)을 컴파일 전에 연결하는 연산자입니다.

```
#define tokencat(x,y) x##y
```

- 위에서 처럼 tokencat을 정의했을 때, 다음과 같이 사용할 수 있습니다.

```
tokencat(i,j)
```

- 그러면, 컴파일 전에(전처리 시간에) ij 로 치환됩니다.

-
- 아래의 소스를 참고하세요.

```
#include <iostream>
```

```
//#define charit(x) #@x//0| 연산자 #@는 각자가 사용하는 컴파일러의 도움말  
//을 참고하세요.
```

```
#define stringit(x) #x
```

```
#define tokencat(x,y) x##y
```

```
void main(void)
```

```
{
```

```
    int i=1,j=2,ij=3;
```

```
    cout << stringit(hello) << '\n';
```

```
    cout << tokencat(i,j) << '\n';
```

```
}
```



참조/역참조(Reference/Dereference)

& 참조 연산자 혹은 주소 연산자(address-of operator)

* 역참조 연산자



재참조 연산자, 간접지정 연산자(indirect operator)라고도 합니다.

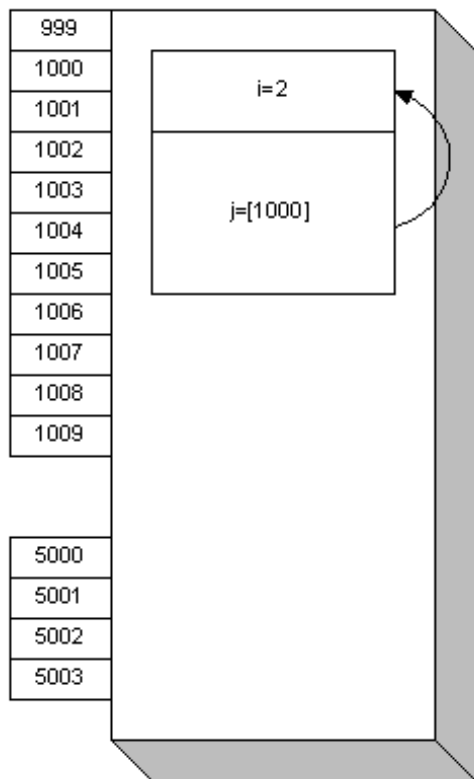
- &는 데이터가 저장된 곳의 주소를 얻기 위해 사용합니다.

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

```
void main() {  
    short i=2;  
    int *j;
```

```
    j=&i;  
} //main
```

- i 가 1000번지에 할당되었고, j 가 1002번지에 할당되었다면 메모리의 구조는 다음과 같습니다.



-
- j가 가리키는 값 - i값, 즉 1000번지에 있는 값 - 을 참조하기 위해서는 어떻게 할 것인가요?
 - 바로 역참조 연산자 *를 사용하는 것입니다. *j는 j가 가리키는 값, 즉 정수 2를 의미합니다.

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

```
void main() {
    int i=2;
    int *j;

    j=&i;
    printf("%d,%p,%p\n", *j, j, &j);
} //main
```

메모리의 구조가 위 그림과 같다면, 출력 결과는 다음과 같습니다.

2,[1000],[1002]



관계(Relational)

- == 같음 연산자(equality operator)
- != 같지 않음 연산자(inequality operator)
- < 작다 ■ 연산자(less-than operator)
- > 크다 연산자(greater-than operator)
- <= 작거나 같다 연산자(less-than or equal operator)
- >= 크거나 같다 연산자(greater-than or equal operator)



sizeof

sizeof <expression>

sizeof (<type>)

- **sizeof 연산자**는 표현식이나 형이 차지하는 바이트 수 (number)를 구하기 위해 사용합니다.



-
- 아래의 예제는 16비트 컴파일러에서 컴파일 되었으므로, 결과는 다음과 같습니다.

2,2

2,4

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

```
void main() {
```

```
    int i,j;
```

```
    i=sizeof(int);
```

```
    j=sizeof i;
```

```
    printf("%d,%d\n",i,j);
```

```
    printf("%d,%d\n",sizeof 2,sizeof 2L);
```

```
}//main
```

-
- 아래의 소스는 무엇이 잘못된가요?

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

void main() {
    int *ip;

    ip=(int *)malloc(2); //왜 2바이트를 할당해야 하는가요?
    //! 위 문장은 정수 할당을 보장하지 못합니다.
    *ip=2;
    printf("%d\n", *ip);
    free(ip);
} //main
```

-
- 위의 소스는 아래와 같이 바르게 고쳐서 사용하는 것이 마땅합니다.

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

void main() {
    int *ip;

    ip=(int *)malloc(sizeof(int)); //확실하게 정수 할당을 보장합니다.
    *ip=2;
    printf("%d\n", *ip);
    free(ip);
} //main
```

-
- sizeof를 포인터 변수에 대해서 쓸 때 주의해야 할 사항이 있습니다. 아래의 소스는 어디가 잘못된가요?

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

```
struct STest {
    int age;
    char name[80];
}; //struct STest
```

```
void main() {
    struct STest *s;

    s=(struct STest*)malloc(sizeof(s)); //s는 4바이트이다.
    s->age=30;
    strcpy(s->name, "seojt");
    printf("%s : %d\n", s->name, s->age);
}
```

```
    free(s);  
} //main
```

- s는 포인터이므로 4바이트입니다.
- 그러므로 sizeof(*s)가 되는 것이 맞습니다. 확실하게 sizeof(struct STest)라고 하는 것이 좋습니다.



형 변환(casting: type conversion)

- 아래 프로그램의 결과는 얼마일까요?

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

```
void main() {
```

```
    char c=12;
```

```
    short i=0x1234;
```

```
    c=i;//이 문장에서 무엇이 일어났는가요?
```

```
    printf("%x\n",c);
```

```
}
```

- 결과는 다음과 같습니다.



-
- 아래의 예를 보고, 잘못된 부분을 수정해 보세요.

```
#include <stdio.h>

void main() {
    char c=12;
    short i=0x1234;

    i=c;
    printf("%x\n", c);
}
```

- 실행 결과는 다음과 같습니다.

C

-
- 다음과 같이 소스를 수정하는 것이 바람직합니다.

```
#include <stdio.h>

void main() {
    char c=12;
    short i=0x1234;

    i=(short)c;
    printf("%x\n", c);
}
```

- 물론 위의 소스는 이전 소스와 동작과 결과가 같습니다. 하지만, 분명한 형 변환을 명시하므로, 이전의 소스보다 읽기 좋습니다.

-
- 아래의 소스를 보세요. 무엇이 잘못된가요?

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

void main() {
    int* ip;

    ip=malloc(sizeof(int));
    *ip=100;
    printf("%d\n", *ip);
    free(ip);
}
```

- Visual C++에서 실행하면 다음과 같은 에러 메시지가 발생합니다.

cannot convert from 'void *' to 'int *'

-
- 소스는 다음과 같이 수정되어야 합니다.

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

void main() {
    int* ip;

    ip=(int*)malloc(sizeof(int));
    *ip=100;
    printf("%d\n", *ip);
    free(ip);
}
```

- 실행 결과는 다음과 같습니다.

함수형 형변환(functional casting)

- 정수형(int)의 변수 i를 문자형(char)의 변수 c로 변환하기 위한 문장은 다음과 같습니다.

```
int i=65;  
char c;  
c=(char)i;
```

- 위의 형 변환 문장은 함수 호출처럼 사용할 수 있습니다.

```
c=char(i);
```



실습문제

1. x^y 를 계산하는 최적의 방법을 기술하세요.

2. 16^n 을 계산하기 위해 <<를 어떻게 이용할 수 있습니까?

3. 어셈블리어의 회전 연산자(rotate operator)를 함수로 구현하세요.

4. C++에서 사용 가능한 모든 연산자를 체계적으로 구분하고 간단히 설명하세요.

5. 아래의 소스에서 잘못된 부분이 있으면, 잘못된 부분을 수정하세요.

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

```
void main() {
```

```
line4:    printf("hello");
```

```
line5:    printf("world");
```

```
}
```

6. 아래의 소스에서 t.*대신에 t->*를 사용하도록 하려면, 소스를 어떻게 수정해야 할까요?

```
void SetValue(CTest &t,int i)
{
    int CTest::*ip;

    ip=&CTest::a;
    t.*ip=i;
} //CTest::SetValue
```