ARRAYS, POINTERS, AND STRINGS

One-dimensional Arrays

```
int grade 0, grade1, grade3;
```

- Array
  - a simple variable with an index, or subscript, added
  - The brackets [] are used for array subscripts.

```
int grade[3];
```

```
#define  N  100
int a[N];
for (i = 0; i < N; ++i)
   sum += a[i];        /* process element a[i] */
```

✓ The indexing of array elements always starts from 0.

One-dimensional Arrays

- Array Initialization
  - Array may be of storage class automatic, external, or static, but NOT register.
  - Arrays can be initialized using an array initializer.

```
float f[5] = {0.0, 1.0, 2.0, 3.0, 4.0};
int     a[100] = {0};  /* initializes all elements of a to zero*/
int     a[] = {2, 3, 5, -7}; ⇔  int   a[4] = {2, 3, 5, -7};
```

  - If there is **fewer initializers** for an array than the number specified, **the missing elements will be zero** for external, static, and automatic variables.
  - **external** or **static** array
    - If not initialized explicitly, then initialized to zero by default

One-dimensional Arrays

- Array Subscripting

  a[*expr*]

  int i, a[N];

  - The expression, **a[i]**
    - refers to (**i+1**)th element of the array **a**
    - If i has a value outside the range from 0 to N-1, then **Run-Time Error**
- The operators, **()** in function call and **[]** in array subscripting have
  - the highest precedence
  - left to right associativity

연산자		결합성
() [] -> .		left to right
! ~ ++ -- + - * & (*type*) sizeof		**right to left**
* / %	산술연산자	left to right
+ -		left to right
<< >>	비트연산자(쉬프트)	left to right
< <= > >=	관계연산자	left to right
== !=		left to right
&	비트연산자	left to right
^		left to right
|		left to right
&&	논리연산자	left to right
||		left to right
?:		**right to left**
= += -= *= /= %= &= ^= |= <<= >>=		**right to left**
,		left to right

Pointers

- Pointers
  - used to access memory and manipulate addresses
  - If v is a variable, then &v is the location, or address, in memory space.
    - &: unary address operator, right-to-left associativity
  - Pointer variables

```
int *p;      /*declares p to be of type pointer to int */
p = 0;
p = NULL;            /*equivalent to p = 0; */
p = &i;                        /* int i; */
p = (int *) 1776;    /* an absolute addr. in memory */
```

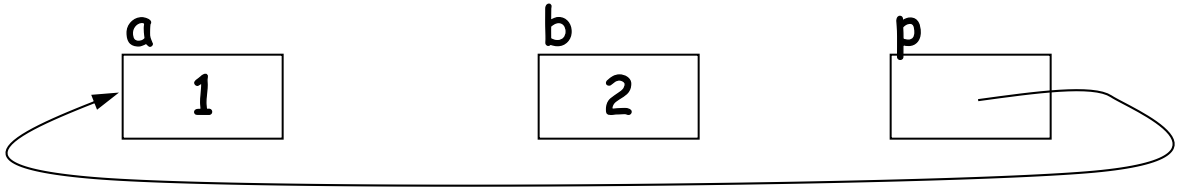
Pointers

- Pointers
  - If **p** is a pointer, then ***p** is the value of the variable of which **p** is the address.
    - ***:** unary "**indirection**" or "**dereferencing**" operator, right-to-left associativity
    - The direct value of **p** is a memory location.
    - ***p** is the indirect value of **p**-namely, the value at the memory location stored in **p**.

Pointers

int a = 1, b = 2, *p;

p = &a; "p is assigned the address of a"



b = *p; "b is assigned the value pointed to by p"

b = *p; ⇔ b = a;

Pointers

```
/* printing an address, or location */
#include <stdio.h>
int main(void)
{
   int i = 7, *p = &i;
   printf("%s%d\n%s%p\n", "   Value of i: ", *p,
            "Location of i: ", p);
   return 0;
}
```

```
   Value of i: 7
Location of i; effffb24
```

Pointers

Declarations and Initializations		
int i = 3, j = 5, *p = &i, *q = &j, *r; double x;		
Expression	Equivalent expression	Value
p == & i	p == (& i)	1
* * & p	* (* (& p))	3
r = & x	r = (& x)	/*illegal*/
7 * * p / * q + 7	((7 * (* p)) / (* q)) + 7	11
* (r = & j) *= *p	(* (r = (& j))) *= (* p)	15

Pointers

- Conversions during assignment between different pointer types are allowed
  - when one of the type is a pointer to **void**
  - when the right side is the constant **0**

Declarations and Initializations	
int *p; float *q; void *v;	
Legal assignments	Illegal assignments
p = 0;	p = 1;
p = (int *) 1;	v = 1;
p = v = q;	p = q;
p = (int *) q;	

Pointers

- Keep in mind the following prohibitions!
  - Do not point at constants.

    ```
    &3                    /* illegal */
    ```

  - Do not point at ordinary expression.

    ```
    &(k+99)               /* illegal */
    ```

  - Do not point at register variable.

    ```
    register  v;
    &v                    /* illegal */
    ```

Call-by-Reference

- "Call-by-value" mechanism
- "Call-by-reference" mechanism
  - for changing the values of variables in the calling environment
  - Pointers must be used in parameter list in the function definition.

1. Declaring a function parameter to be a pointer
2. Using the dereferenced pointer in the function body
3. Passing an address as an argument when calling the function

Call-by-Reference

```
#include <stdio.h>
void swap(int *, int *);
int main(void)
{
    int i = 3, j = 5;
    swap(&i, &j);
    printf("%d %d\n", i, j);  /* 5 3 is printed */
    return 0;
}
void swap(int *p, int *q)
{
    int tmp;
    tmp = *p;
    *p = *q;
    *q = tmp;
}
```

Arrays and Pointers

- An array name
  - by itself is an address, or pointer value
- Arrays and Pointers
  - can be subscripted.
  - A pointer var. can take different addr. as values
  - An array name is an FIXED address, or pointer.

Arrays and Pointers

```
#define N 100
int a[N], i, *p, sum = 0;
```

a[i] ⇔ *(a+i) : the value of the ith element of the array, a

- a + i
  - A pointer arithmetic
  - has as its value the ith offset from the base addr. of the array, a
  - points to the ith element of the array (counting from 0)

p[i] ⇔ *(p+i)

- p + i
  - is the ith offset from the value of p.
  - The actual addr. produced by such an offset depends on the type that p points to

p = a; ⇔ p = &a[0];

p = a + 1; ⇔ p = &a[1];

Arrays and Pointers

```
#define  N  100
int a[N], i, *p, sum =0;
for ( i=0; i < N; ++i)
    sum += a[i];
```

```
for (i = 0; i < N; ++i)
    sum += *(a+i);
```

```
for (p = a; p < &a[N]; ++p)
    sum += *p;
```

```
p = a;
for (i = 0; i < N; ++i)
    sum += p[i];
```

- ✓ Note that because **a** is a constant pointer, expressions such as

  a = p ++a a += 2 &a

  are illegal.

Pointer Arithmetic and Element Size

- Pointer arithmetic
  - If the variable **p** is a pointer to a particular type,

    p + 1 p + i ++p p += i

```
double a[2], *p, *q;
p = a;                    /* points to base of array */
q = p +1;                 /* equivalent to q = &a[1] */
printf("%d\n", q-p); /* 1 is printed */
printf("%d\n", (int) q – (int) p); /* 8 is printed */
```

- **q – p**
  - yields the **int** value representing the number of array elements between **p** and **q**

Arrays as Function Arguments

- In a function definition, a formal parameter that is declared as an array is actually a pointer.
  - When an array is passed as an argument to a function, the base addr. of the array is passed "call-by-value"

```
double sum(double a[], int n)
{   /* n is the size of a[] */
    int i;
    double sum = 0.0;
    for ( i = 0; i < n; ++i)
      sum += a[i];
    return sum;
}
```

⇔

```
double sum(double *a, int n)
{
    ....
```

Arrays as Function Arguments

Various ways that **sum()** might be called	
Invocation	What gets computed and returned
sum(v, 100)	v[0] + v[1] + … + v[99]
sum(v, 88)	v[0] + v[1] + … + v[87]
sum(&v[7], k-7)	v[7] + v[8] + … + v[k-1]
sum(v+7, 2*k)	v[7] + v[8] + … + v[2*k + 6]

Dynamic Memory Allocation with calloc() and malloc()

- **calloc()** and **malloc()**
  - declared in **stdlib.h**
  - **calloc()** : contiguous allocation
  - **malloc()** : memory allocation

```
#include <stdio.h>
#include <stdlib.h>
void swap(int *, int *);
int main(void)
{
    int *a;                          /* to be used as an array */
    int n;                           /* the size of the array */
    ......                           /* get n from somewhere */
    a = calloc(n, sizeof(int));      /* get space for a */
    .....                    /* use a as an array */
}
```

Dynamic Memory Allocation with calloc() and malloc()

- **calloc()** and **malloc()**

  ptr = calloc(n, sizeof(int));

  - The allocated memory is initialized with all bits set to zero.

  ptr = malloc(n * sizeof(int));

  - does not initialize the memory space

  if(a != NULL)

  …

  - Space having been dynamically allocated MUST be returned to the system upon function exit.

    free(ptr);

    - **ptr** must be the base addr. of space previously allocated.

Strings

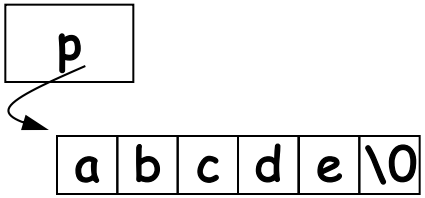
- Strings
  - one-dimensional array of type char
  - terminated by the end-of-string sentinel '\0', or null character (0x00)
  - The size of a string must include the storage needed for the null character.
    - "abc" : a char. array of size 4
  - String constant, like an array name by itself, is treated as a pointer

```
char *p = "abc";
printf("%s %s\n", p, p+1);   /*abc bc is printed */
```

Strings

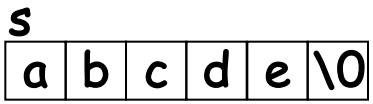
char *p = "abcde";

- allocates space in memory for **p**
- puts the string constant "**abcde**" in memory somewhere else,
- and initializes **p** with the base addr. of the string constant



char s[] = "abcde"; ⇔ char s[]={'a', 'b', 'c', 'd', 'e', '\0'};

- allocates 6 bytes of memory for the array **s**.



Strings

```
/*Count the number of words in a string */
#include <ctype.h>
int word_cnt(const char *s)
{
    int cnt = 0;

    while (*s != '\0') {
        while ( isspace(*s)   )                    /* skip white space */
          ++s;

        if( *s != '\0') {                          /* found a word */
            ++cnt;
            while ( !isspace(*s) && *s != '\0' ) /* skip the word */
                ++s;
        }
    }
    return cnt;
}
```

String-Handling Functions in the Standard Library

- A standard header file, **string.h**

```
char   *strcat(char *s1, const char *s);
int      strcmp(const char *s1, const char *s2);
```

  - If **s1 < s2**, returns negative integer
  - If **s1 = s2**, returns zero
  - If **s1 > s2**, returns positive integer

```
char   *strcpy(char *s1, const char *s2);
size_t strlen(const char *s);
```

  - the number of char.s before '\0'

```
size_t strlen(const char *s)
{
    size_t n;
    for(n = 0; *s != '\0'; ++s)
        ++n;
    return n;
}
```

String-Handling Functions in the Standard Library

```
char *strcpy(char *s1, register const char *s2)
{
    register char *p = s1;
    while (*p++ = *s2++)
        ;
    return s1;
}
```

- ✓ *p++ ⇔ *(p++)
  p itself is being incremented.
- ✓ (*p)++
  would increment what p is pointing to.

```
char *strcat(char *s1, register const char *s2)
{
    register char *p = s1; /* go to the end */
    while (*p)
        ++p;
    while (*p++ = *s2++)   /* copy */
        ;
    return s1;
}
```

- ✓ while (*p) ⇔ while (*p != '\0')

!! Note carefully that it is the programmer's responsibility to allocate sufficient space for the strings that are passed as arguments to these functions

String-Handling Functions in the Standard Library

Declarations and Initializations	
`char s1[] = "beautiful big sky country",` `  s2[] = "how now brown cow";`	
Expression	Value
strlen(s1)	25
strlen(s2 + 8)	9
strcmp(s1, s2)	negative integer
Expression	What gets printed
printf("%s", s1 + 10);	big sky country
strcpy(s1 + 10, s2 + 8);	
strcat(s1, "s!");	
printf("%s", s1);	beautiful brown cows!

Multidimensional Arrays

- C lang. allows arrays of any type, including arrays of arrays.
- Two-dimensional array
  - using two bracket pairs, [][]

```
int a[100];          a one-dimensional array
int b[2][7];         a two-dimensional array
int c[5][3][2];      a three-dimensional array
```

- k-dimensional array
  - allocates space for $s_1 \times s_2 \times \ldots \times s_k$ elements, where s_i represents the size of ith dimension.
  - Starting at the base addr. of the array, all the elements are stored contiguously in memory.

Multidimensional Arrays

- Two-dimensional array

  int a[3][5];

	col 1	col 2	col 3	col 4	col 5
row 1	a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]
row 2	a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]
row 3	a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]

Expressions equivalent to a[i][j]
*(a[i] + j)
(*(a + i))[j]
(((a + i)) + j)
*(&a[0][0] + 5*i + j)

- The array name **a** by itself is equivalent **&a[0]**; it is a pointer to an array of 5 ints.
- The base addr. of the array is **&a[0][0]**, not **a**.
  - Starting at the base addr. of the array, compiler allocate for 15 ints.

Storage Mapping Function

- mapping between pointer values and array indices
  - int a[3][5]
    - a[i][j] : *(&a[0][0] + 5*i + j)
  - int a[7][9][2]
    - a[i][j][k] : *(&a[0][0][0] + 9*2*i + 2*j + k)

Multidimensional Arrays

- Formal Parameter Declarations
  - When a multidimensional array is a formal parameter in a function definition, all sizes except the first must be specified
    - so that the compiler can determine the correct mapping.

```
int    a[3][5];
int    sum(int a[][5])
{
   int i, j, sum = 0;
   for(i=0; i<3; ++i)
      for(j=0; j<5; ++j)
         sum += a[i][j];

   return sum;
}
```

✓ int a[][5] ⇔ int a[3][5] ⇔ int (*a)[5]

✓ int b[3];

int b[] ⇔ int b[3] ⇔ int *b

Multidimensional Arrays

- Initialization
  - The indexing is by rows.
  - All sizes except the first must be given explicitly

```
int a[2][3] = {1,2,3,4,5,6};
int a[2][3] = {{1,2,3}, {4,5,6}};
int a[][3] = {{1,2,3}, {4,5,6}};
int a[][3] = {{1,0,0}, {4,5,0}};  ⇔  int a[][3] = {{1}, {4,5}};

int a[2][3] = {0};
```

Use of typedef

```
#define      N  3
typedef double scalar;
typedef scalar vector[N];
typedef scalar matrix[N][N];
```

```
void add(vector x, vector y, vector z)
{  /* x = y + z */
    int     i;
    for (i = 0; i < N; ++i)
        x[i] = y[i] + z[i];
}
scalar dot_product(vector x, vector y)
{
    int     i;
    scalar sum = 0.0;
    for (i = 0; i < N; ++i)
        sum += x[i] * y[i];
    return sum;
}
void multiply(matrix a, matrix b, matrix c)
{  /* a = b * c */
    int    i, j, k;
    for (i = 0; i < N; ++i) {
        for (j = 0; j < N; ++j) {
            a[i][j] = 0.0;
            for (k = 0; k < N; ++k)
                a[i][j] += b[i][k] * c[k][j];
        }
    }
}
```

Arrays of Pointers

- Array elements can be of any type, including a pointer type.
- An array with elements of type **char ***
  - an array of strings

[input]
A is for apple or alphabet pie
which all gets a slice of, come taste it and try.

[output]
A
a
all
alphabet
…
which

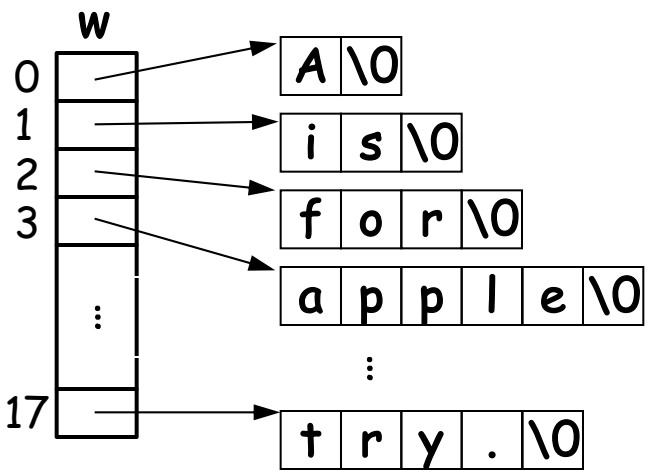
Arrays of Pointers

[sort.h]

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

#define MAXWORD      50   /* max word size */
#define N            300  /* array size of w[] */

void error_exit_calloc_failed(void);
void error_exit_too_many_words(void);
void error_exit_word_too_long(void);
void sort_words(char *w[], int n);
void swap(char **p, char **q);
void wrt_words(char *w[], int n);
```

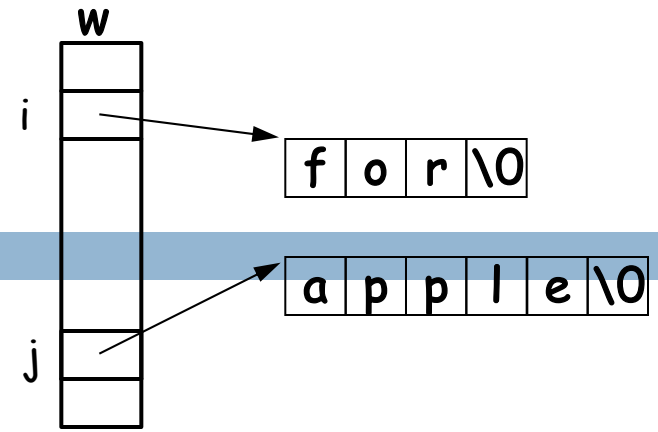
[main.c] /*Sort words lexicographically. */

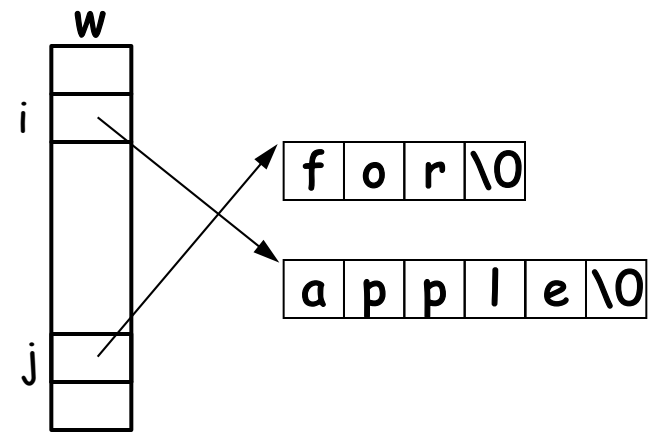
```
#include "sort.h"

int main(void)
{
    char word[MAXWORD]; /* work space */
    char *w[N];         /* an array of pointers */        ⇔ char *(w[N]);
    int n;              /* number of words to be sorted */
    int i;

    for (i=0; scanf("%s", word) == 1; ++i) {
        if (i >= N)
            error_exit_too_many_words();
        if (strlen(word) >= MAXWORD)
            error_exit_word_too_long ();

        w[i] = calloc(strlen(word) + 1, sizeof(char));
        if (w[i] == NULL)
            error_exit_calloc_failed();
        strcpy(w[i], word);
    }
    n = i;

    sort_words(w, n);           /* sort the words */
    wrt_words(w, n);            /* write sorted list of words */
    return;
}
```



[sort.c]

```
#include "sort.h"
void sort_words(char *w[], int n)
{
    int i,j;

    for (i=0; i<n; ++i)
        for (j=i+1; j<n, ++j)
            if (strcmp(w[i], w[j]) > 0)
                    swap(&w[i], &w[j]);
}
```





[swap.c]

```
#include "sort.h"
void swap(char **p, char **q)
{
    char *tmp;

    tmp = *p;
    *p = *q;
    *q = tmp;
}
```

- ✓ The addr. of the pointers are passed so that the pointer values themselves can be changed in the calling environment by the function call.
- ✓ an addr. of a pointer to **char**
  ⇔ a pointer to pointer to **char**

[wrt.c]

```
#include "sort.h"
void wrt_words(char *w[], int n)
{
    int i;

    for (i=0; i<n; ++i)
        printf("%s\n", w[i]);
}
```

Arguments to main()

- Two arguments, **argc** and **argv**, can be used with **main()**.

```
/* Echoing the command line arguments. */
#include <stdio.h>
int main(int argc, char *argv[])
{
    int i;

    printf("argc = %d\n", argc);
    for (i=0; i< argc; ++i)
        printf("argv[%d] = %s\n", i, argv[i]);

    return 0;
}
```

```
[Command]
my_echo a is for apple

[Output]
argv = 5
argv[0] = my_echo
argv[1] = a
argv[2] = is
argv[3] = for
argv[4] = apple
```

Ragged Arrays

- Ragged array
  - an array of pointers whose elements are used to point to arrays of varying sizes

```
#include <stdio.h>
int main(void)
{
    char a[2][15] = {"abc:", "a is for apple"};
    char *p[2]     = {"abc:", "a is for apple"};
    printf("%c%c%c %s %s\n", a[0][0], a[0][1], a[0][2], a[0], a[1]);
    printf("%c%c%c %s %s\n", p[0][0], p[0][1], p[0][2], p[0], p[1]);
}
```

- [output]

```
abc abc: a is for apple
abc abc: a is for apple
```

Ragged Arrays

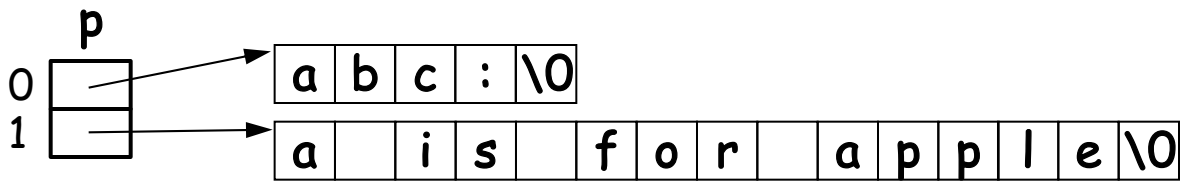
- **char a[2][15] = {"abc:", "a is for apple"};**
  - Space for 30 **chars** is allocated
  - Each of **a[0]** and **a[1]** is an array of 15 **chars**
  - **a[0]** and **a[1]** are strings
  - **char a[2][15] = {{'a', 'b', 'c', ':', '\0'}, {'a', ' ', 'i', 's', … '\0'}};**

- **Compiler generates a storage mapping function for accessing array elements**

Ragged Arrays

- **char *p[2] = {"abc:", "a is for apple"};**
  - one-dimensional array of pointers to **char**
  - It causes space for two pointers to be allocated.
  - **p[0]** is initialized to point at **"abc:"**, a string constant of 5 **chars**, thus there is no way to modify "abc:" (e.g. p[0][3]='d' is not allowed)
  - **p[1]** is initialized to point at **"a is for apple"**, a string constant of 15 **chars**.
  - **p** does its work in less space than **a**
  - **p[0][14]** vs. **a[0][14]**

- **Compiler does not generate a storage mapping function for accessing array elements** ⇒ faster working than 2D array **a**



Function Pointers

- pointers to functions
  - arguments
  - array elements
  - return values

```
double sum-square( double f(double), int m, int n)
/* ⇔  double sum-square( double (*f)(double), int m, int n) */
{
   int k;
   double sum=0.;

   for (k=m; k<=n; ++k) {
      sum+=f(k)*f(k); /* ⇔  sum+=(*f)(k) * (*f)(k);  */
   }
   return sum;
}
```

연산자		결합성
() [] -> .		left to right
! ~ ++ -- + - * & (*type*) sizeof		**right to left**
* / %	산술연산자	left to right
+ -		left to right
<< >>	비트연산자(쉬프트)	left to right
< <= > >=	관계연산자	left to right
== !=		left to right
&	비트연산자	left to right
^		left to right
|		left to right
&&	논리연산자	left to right
||		left to right
?:		**right to left**
= += -= *= /= %= &= ^= |= <<= >>=		**right to left**
,		left to right