中国民航大学

Civil Aviation University of China

Computer Programming

Sino-European Institute of Aviation Engineering

Module 6 Pointor

Outline

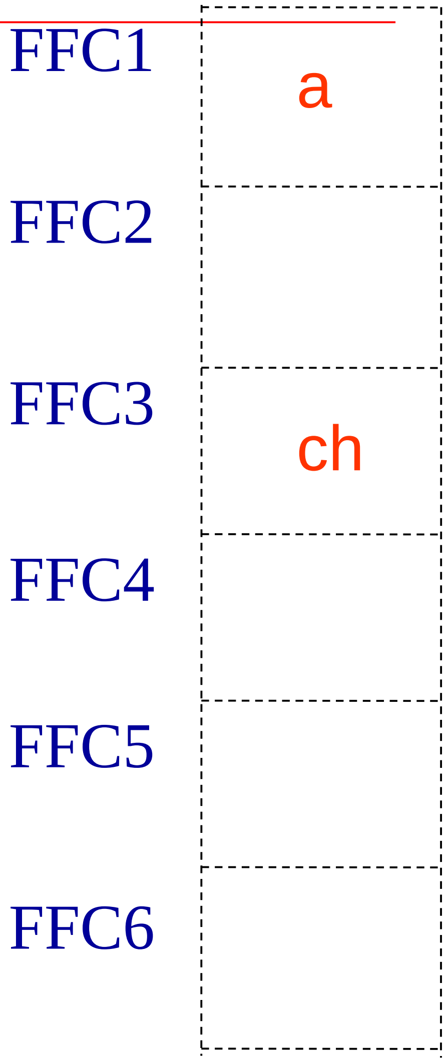
- Introduction
- Pointer Variable
- Pointer Operators
- Pointer and Function
- Pointer Expression and Arithmetic
- Pointer and Array
- Dynamic Allocation
- String, Character Array and Pointer

Introduction

- A variable in a program is stored in a certain number of bytes at a particular memory location in the machine.

  int a;   char ch;

- Each piece of memory should have a distinct number with it, named address.
- Can memory be accessed by its address?



FFC1 a

FFC2

FFC3 ch

FFC4

FFC5

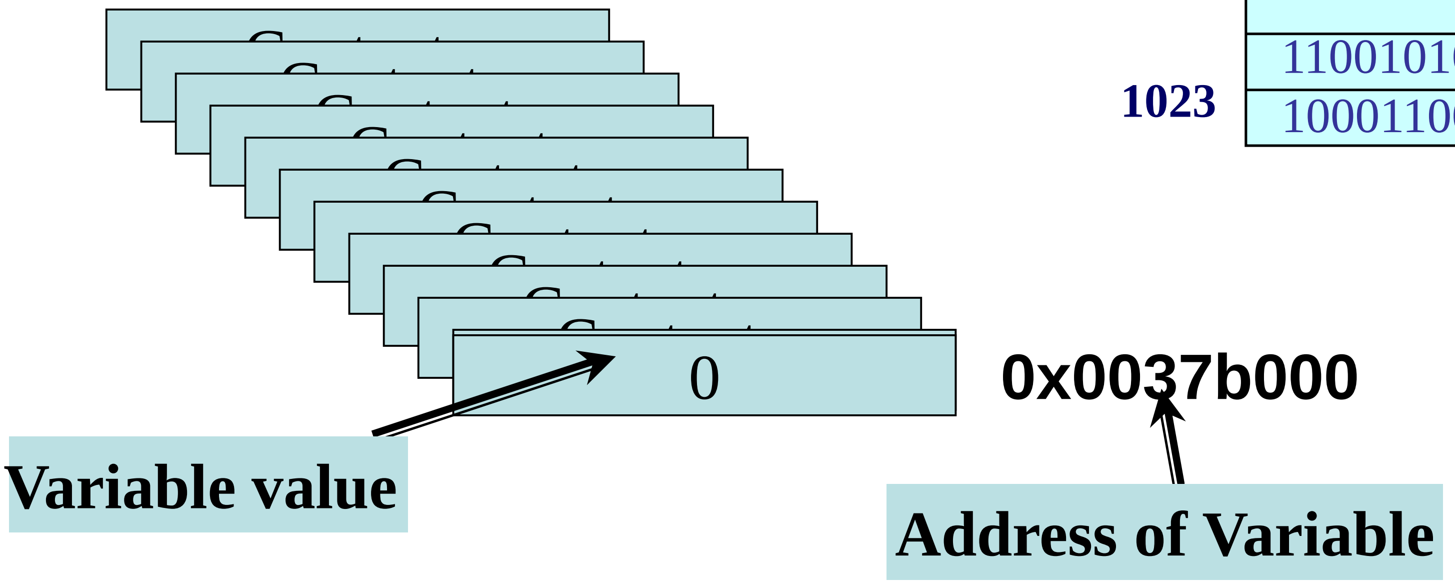
FFC6

Introduction

Random Access Memory Address

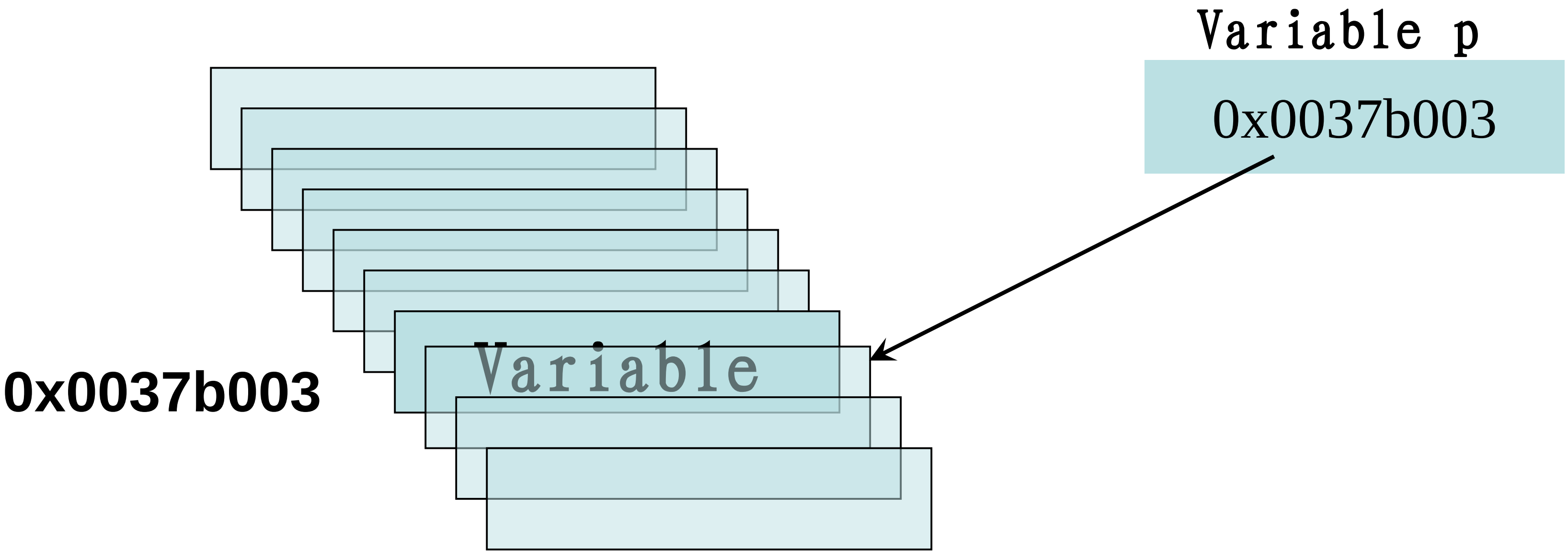
address	
0001	10011001
0002	11001010
0003	10001100
	00001011
	11001010
1023	10001100

int a=0;



Introduction

- How to store the address?



• A **pointer** is a variable that contains the address of another variable.

Introduction

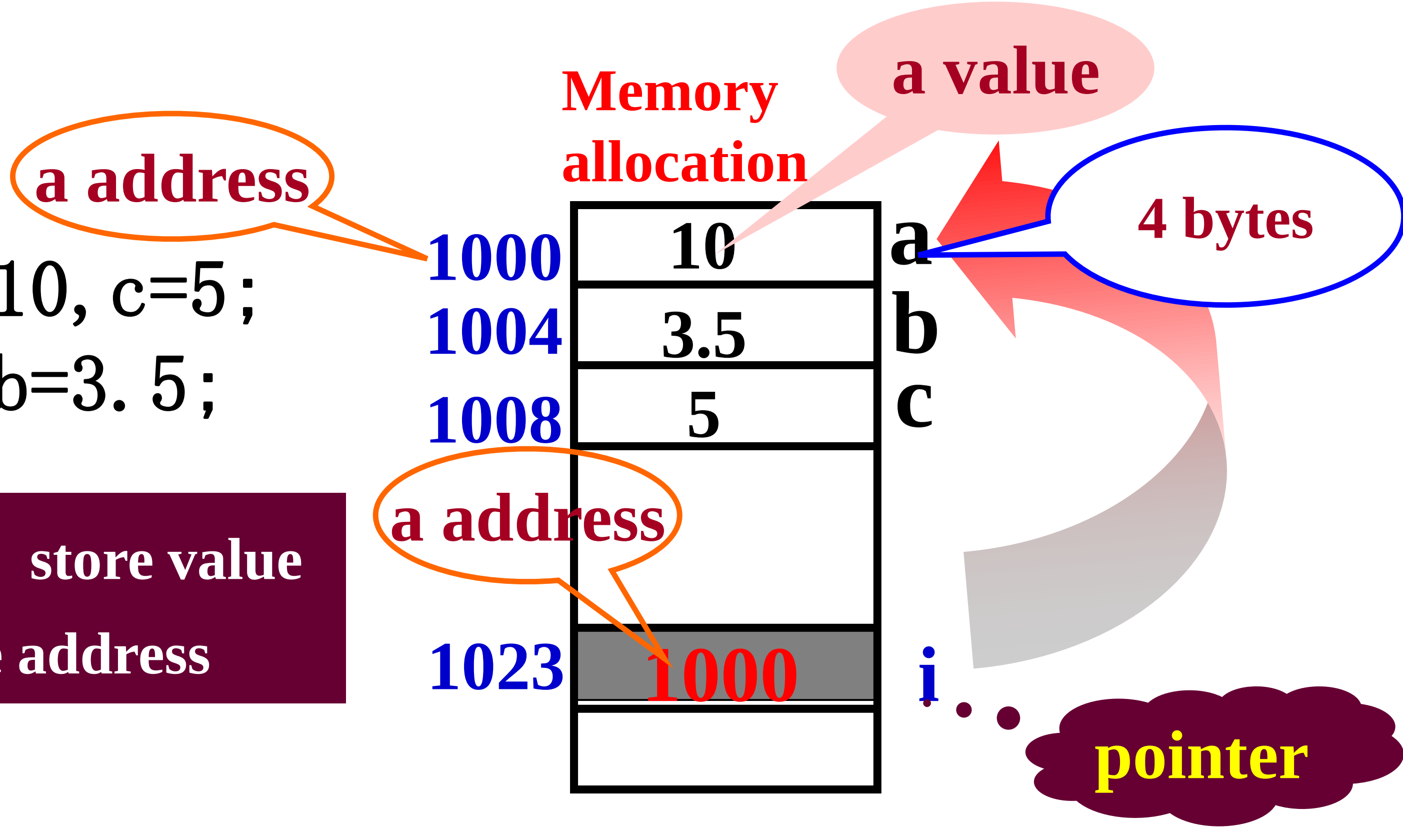
- Pointer variables, s kind of data type, simply called pointers
- Holding memory addresses as their values
- Characters
  - Powerful, but difficult to master
  - Simulate call-by-reference
  - Close relationship with arrays and strings

Introduction

```
int  a=10, c=5;
float  b=3.5;
```

a --is int variable， store value

i --is pointer, store address



address=pointer

Introduction

- How to r/w the data in memory？
  - through the address of variable to access the data
- Addressing Methods:
  - Direct Addressing
    - Through the address of the variable
  - Indirect Addressing
    - Through the variable which store the address of the variable

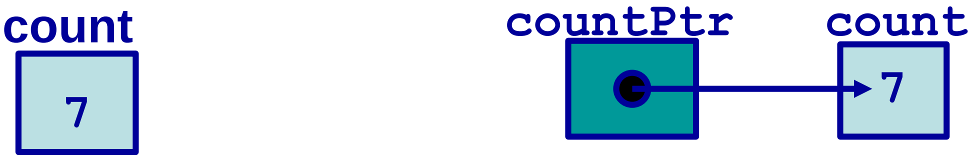
Example:

int a = 0; int *p = &a;

- **Direct access： &a**
- **Indirect access： *i**

Pointer Variable

- Pointer variables
  - Contain memory addresses as their values
  - Normal variables contain a specific value (direct reference)



  - Pointers contain address of a variable that has a specific value (indirect reference)
  - Indirection – referencing a pointer value

Pointer Variable

- Pointer declarations
  - * used with pointer variables

    base-type * pointer-variable;
  - Declares a pointer to an int (pointer of type int *)
  - Multiple pointers require using a * before each variable declaration

    ```
    int *myPtr1, *myPtr2;
    ```
  - Can declare pointers to any data type
  - Initialize pointers to 0, NULL, or an address
    - 0 or NULL – points to nothing (NULL preferred)

Pointer Variable

- Initialization

```
int  a, b;
int  *p1 = &a, *p2 = &b;
```

```
int  a, b;
int  *p1, *p2;
p1 = &a;
p2 = &b;
```

Pointer Operators

- Fundamental pointer operations

Two operators to manipulate pointer values:

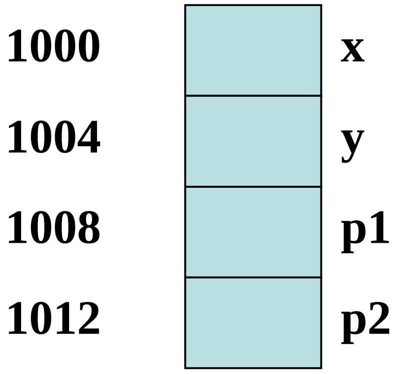
& Address-of

*** Value-pointed-to (dereferencing)**

Example:

```
int x,y;

int *p1, *p2;
```



Pointer Operators

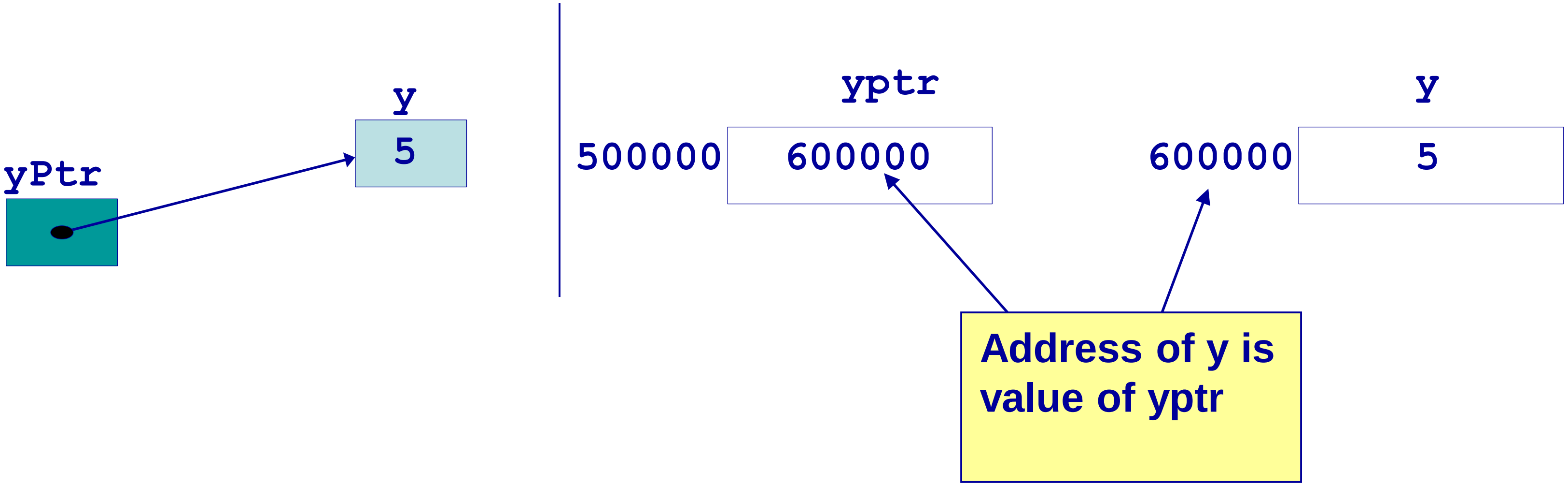
- Address operator &
  - Returns address of operand

    int y = 5;

    int *yPtr;

    yPtr = &y;   /* yPtr gets address of y */

    yPtr “points to” y



Address of y is value of yptr

Pointer Operators

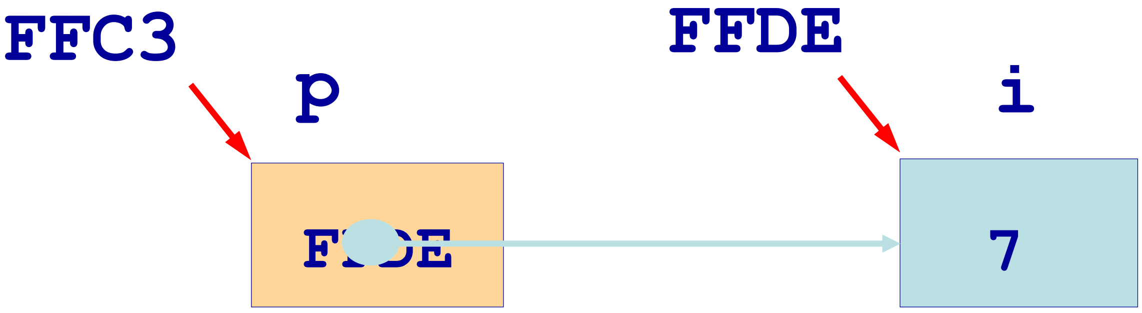
- Indirection/dereferencing operator *
  - Returns a synonym/alias of what its operand points to
  - *yptr returns y (because yptr points to y)
  - * can be used for assignment
    - Returns alias to an object

      ```
      *yptr = 7;   /*  changes y to 7  */
      ```
  - Dereferenced pointer (operand of *) must be an lvalue (no constants)
- * and & are inverses
  - They cancel each other out

Pointer Operators

- Example

```
int i;
i = 7;
int *p;       /* declares p ,pointer to int.*/
p=&i;         /* p pointing to i */
printf(“%d”, *p);
```



Pointer Operators

- * and & are two unary operators for pointer, they are inverses.
  - & : gives the address of an object.
  - * : accesses the object the pointer points to.

```
void main()
 {
    int i;
    int *p;
    i=10;
    p=&i;
    printf("i = %d, *p = %d\n”, i, *p);
    printf(“ value of p is %p\n", p);
  }
```

```
i = 10, *p = 10
value of p is FFDE
```

Pointer Operators

- The value of pointer can be changed during running period.
- It can be assigned to
  - Address of a normal variable

```
int *p, *q, i, j;
p = &i; q = &j;
```

  - Value of another pointer variable with same type

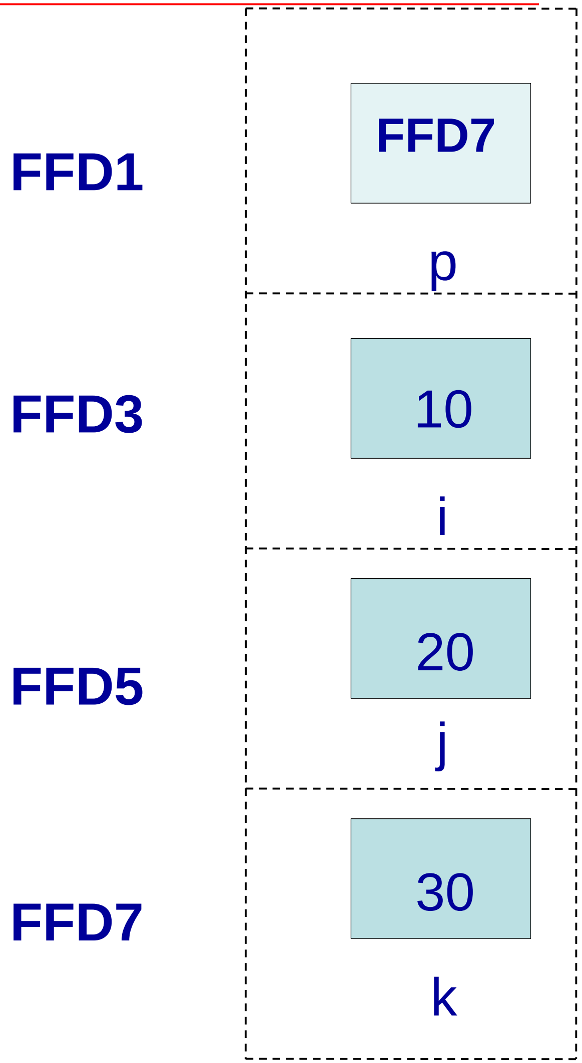
```
p = q;
```

Pointer Operators

- Change the pointer

```
int *p, i, j, k;
i = 10;
j = 20;
k = 30;
p = &i;
printf(“%d\n”, *p);
p = &j;
printf(“%d\n”, *p);
p = &k;
printf(“%d\n”, *p);
```

```
10
20
30
```



FFD1 FFD7 p

FFD3 10 i

FFD5 20 j

FFD7 30 k

Pointer Operators

- Do not point to an exact number of address.

```
p = 4000;              /*illegal*/
```

- Do not point at constants.

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

- Do not point at ordinary expressions.

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

- Do not point at register variables.

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

Pointer Operators

Pointers assignment

- int i, j, *p, *q;/* declaration of pointers */
- i = 5; j = 6;
- p = &i;   /* assign address of i to p */
- q = p;   /* assign value of p to q */
- *p = j;   /* assign value of j to the memory p pointed to */
- *q = --j;   /* assign value j – 1 to the memory q pointed to */
- p = &j;   /* assign address of j to p */
- p = 4000;   /* error */
- *p = &i;   /* error */

Pointer Operators

- Exchange 2 numbers using pointers

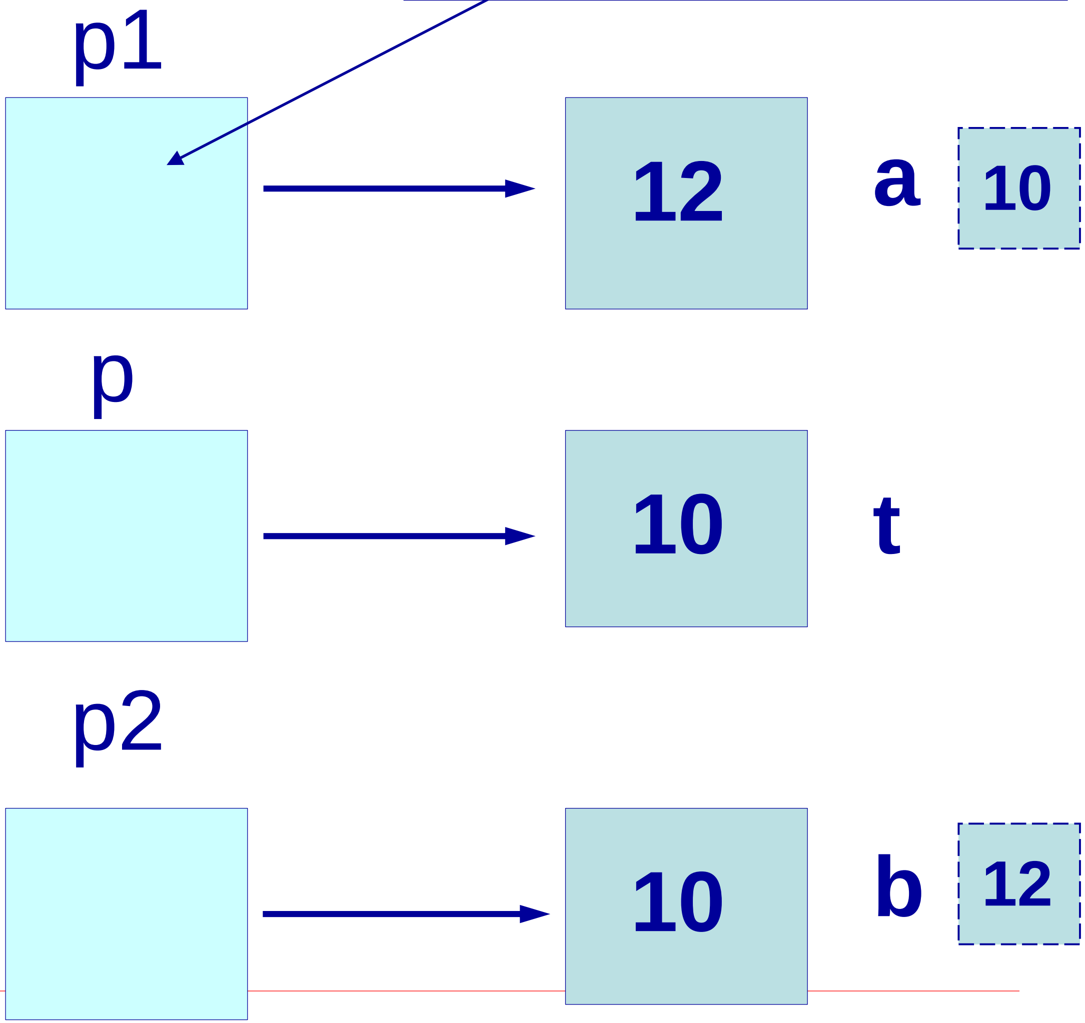
```
void main()
{
  int *p1,*p2,*p,a,b,t;
  scanf(“%d,%d”,&a,&b);
    p1=&a; p2=&b; p=&t;
  if(a<b)
   { *p=*p1; *p1=*p2; *p2=*p; }
  printf(“a=%d,b=%d\n”,a,b);
  printf(“max=%d,min=%d\n”,*p1,*p2”);
}
```

Pointer Operators

```
p1 = &a;
p = &t;
p2 = &b;

if (a < b)
{
   *p = *p1;
   *p1 = *p2;
   *p2 = *p;
}
```

the value of pointer p1 and pointer p2 are not changed. a and b are changed



Pointer Operators

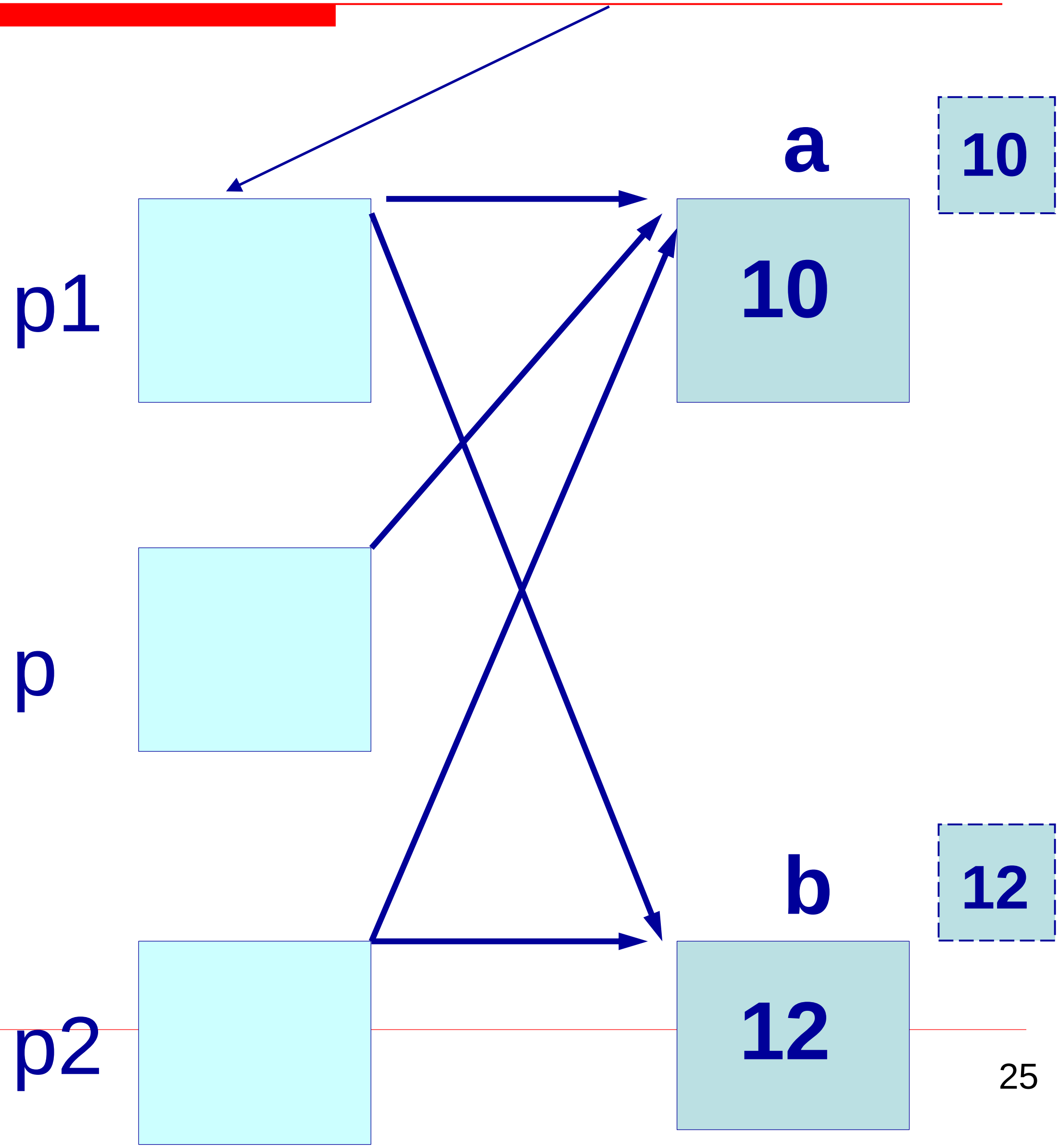
- Exchange 2 numbers using pointers in another way

```
void main()
{
  int *p1,*p2,*p,a,b;
  scanf(“%d,%d”,&a,&b);
  p1=&a; p2=&b;
  if(a<b)
      { p=p1; p1=p2; p2=p;}
  printf(“a=%d,b=%d\n”,a,b”);
  printf(“max=%d,min=%d\n”,*p1,*p2”);
}
```

Pointer Operators

```
p1 = &a;
p2 = &b;

if (a < b)
{
  p = p1;
  p1 = p2;
  p2 = p;
}
```


the value of pointer p1 and pointer p2 are changed, a and b are not changed
a
10
p1
10
p
b
12
p2
12

Pointer and Function

- Whenever variables are passed as arguments to a function, their values are copied to the corresponding function parameters, and the variables themselves are not changed in the calling environment.
- This "call-by-value" mechanism is strictly adhered to in C.
- To change the values of variables in the calling environment, other language provide the "call-by-reference" mechanism.

Pointer and Function

- For a function to effect “call-by-reference”, pointers must be used in the parameter list in the function definition.
- Then , when the function is called, addresses of variables must be passed as argument.

Pointer and Function

- Call by reference with pointer arguments
  - Pass address of argument using & operator
  - Allows you to change actual location in memory
  - Arrays are not passed with & because the array name is already a pointer
- * operator
  - Used as alias/nickname for variable inside of function

```
void double( int *number )
 {
   *number = 2 * ( *number );
 }
```

  - *number used as nickname for the variable passed

Pointer and Function

- The effect of “call-by-reference” is accomplished by
  - Declaring a function parameter to be a pointer.
  - Using the dereferenced pointer in the function body.
  - Passing an address as an argument when the function is called.

Pointer and Function

```
void swap(int a,int b)
{       int temp;
        temp = a;
        a = b;
        b = temp;
}
int main(void)
{       int i = 3,j = 5;
        swap(i,j);
        printf(“%d %d\n”,i,j);
        return 0;
}
```

Are the values of i and j changed?

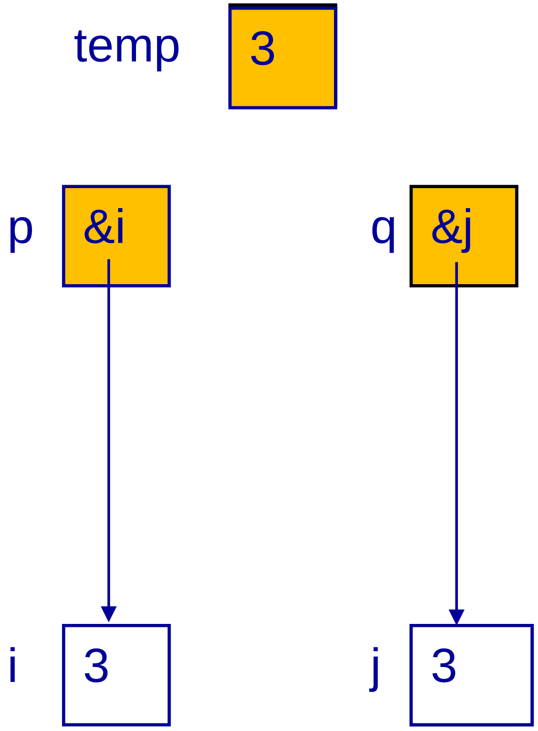
Pointer and Function

```
void swap(int *p,int *q)
{       int temp;
        temp = *p;
        *p = *q;
        *q = temp;
}
int main(void)
{       int i = 3,j = 5;
        swap(&i,&j);
        printf(“%d %d\n”,i,j);
        return 0;
}
```

Are the values of i and j changed?

Pointer and Function

```
void swap(int *p,int *q)
{       int temp;
        temp = *p;
        *p = *q;
        *q = temp;
}
int main(void)
{       int i = 3,j = 5;
        swap(&i,&j);
        printf("%d %d\n",i,j);
        return 0;
}
```



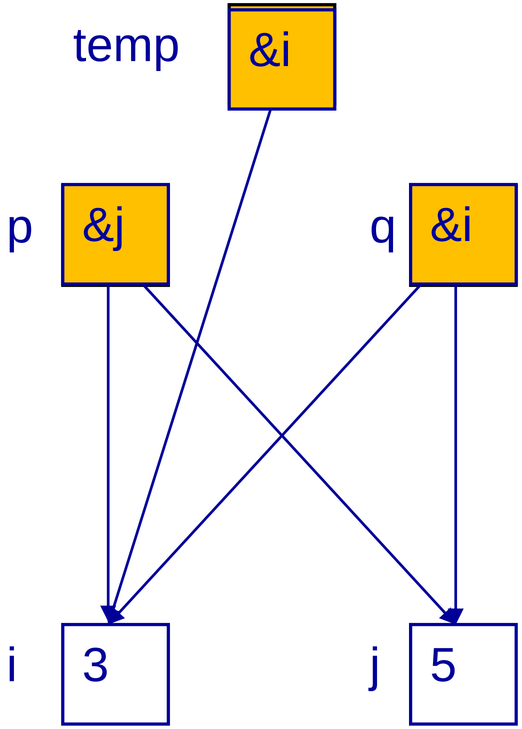
Pointer and Function

```
void swap(int *p,int *q)
{       int *temp;
        temp = p;
        p = q;
        q = temp;
}
int main(void)
{       int i = 3,j = 5;
        swap(&i,&j);
        printf(“%d %d\n”,i,j);
        return 0;
}
```

Are the values of i and j changed?

Pointer and Function

```
void swap(int *p,int *q)
{       int *temp;
        temp = p;
        p = q;
        q = temp;
}
int main(void)
{       int i = 3,j = 5;
        swap(&i,&j);
        printf(“%d %d\n”,i,j);
        return 0;
}
```



Pointer and Function

- pointer as return value

base-type * function(parameters list)

Return char value：

```
char min(char a[10])
{char i,m;
 m=a[0];
 for(i=1;i<10;i++)
  if(m>a[i]) m=a[i];
 return m;
 }
```

Return char * pointer

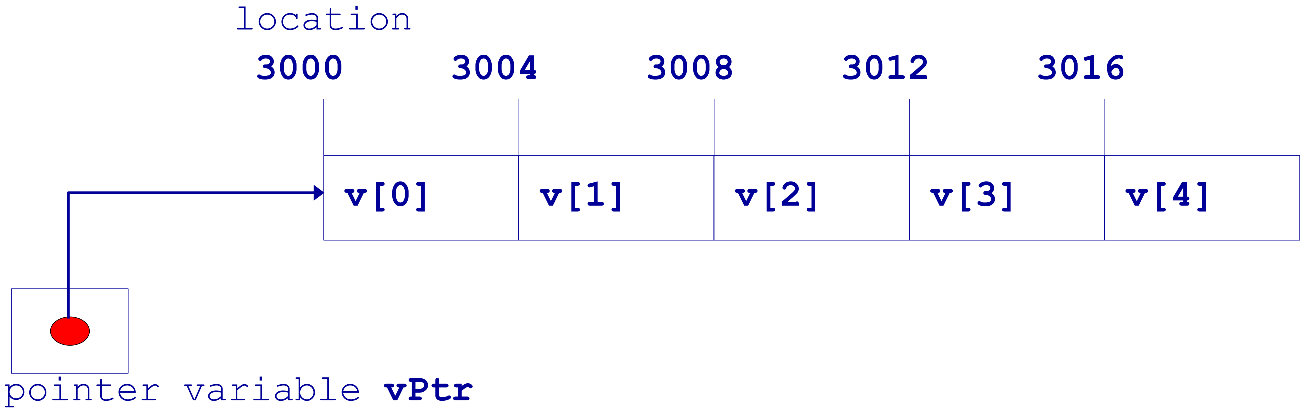
```
char *min(char a[10])
{char i,*m;
 m=&a[0];
 for(i=1;i<10;i++)
  if(*m>a[i]) m=&a[i];
 return m;
}
```

Pointer Expression and Arithmetic

- Arithmetic operations can be performed on pointers
  - Increment/decrement pointer (++ or −−)
  - Add an integer to a pointer(+ or += , − or −=)
  - Pointers may be subtracted from each other
  - Operations meaningless unless performed on an array

Pointer Expression and Arithmetic

- 5 element int array on machine with 4 byte ints
  - vPtr points to first element v[0]
    - at location 3000 (vPtr = 3000)
  - vPtr += 2; sets vPtr to 3008
    - vPtr points to v[2] (incremented by 2), but the machine has 4 byte ints, so it points to address 3008



Pointer Expression and Arithmetic

- Subtracting pointers
  - Returns number of elements from one to the other. If

    ```
    vPtr2 = v[ 2 ];
    vPtr = v[ 0 ];
    ```

  - `vPtr2 - vPtr` would produce 2
- Pointer comparison (<, == , >)
  - See which pointer points to the higher numbered array element
  - Also, see if a pointer points to `0`

Pointer Expression and Arithmetic

- Pointers of the same type can be assigned to each other
  - If not the same type, a cast operator must be used
  - Exception: pointer to `void` (type `void *`)
    - Generic pointer, represents any type
    - No casting needed to convert a pointer to `void` pointer
    - `void` pointers cannot be dereferenced

Pointer and Array

- Pointer and One-dimensional array
- Pointer and Two-dimensional array
  - Row pointer & row address
  - Column pointer & column addresses
- Array of Pointer and Pointer of Array
- Pointer to Pointer

Pointer and 1-D Arrays

- An array name by itself is an address, or pointer value, and pointers, as well as arrays can be subscripted.
- Although pointers and arrays are almost similar in terms of how they are used to access memory, they are different.
- A pointer variable can take different addresses as values. In constrast, an array name is an address, or pointers, that is fixed.

Pointer and 1-D Arrays

- Arrays and pointers closely related.
  - Array name like a constant pointer
  - Pointers can do array subscripting operations
- Declare an array b[5] and a pointer bPtr
  - To set them equal to one another use:

    ```
    bPtr = b;
    ```

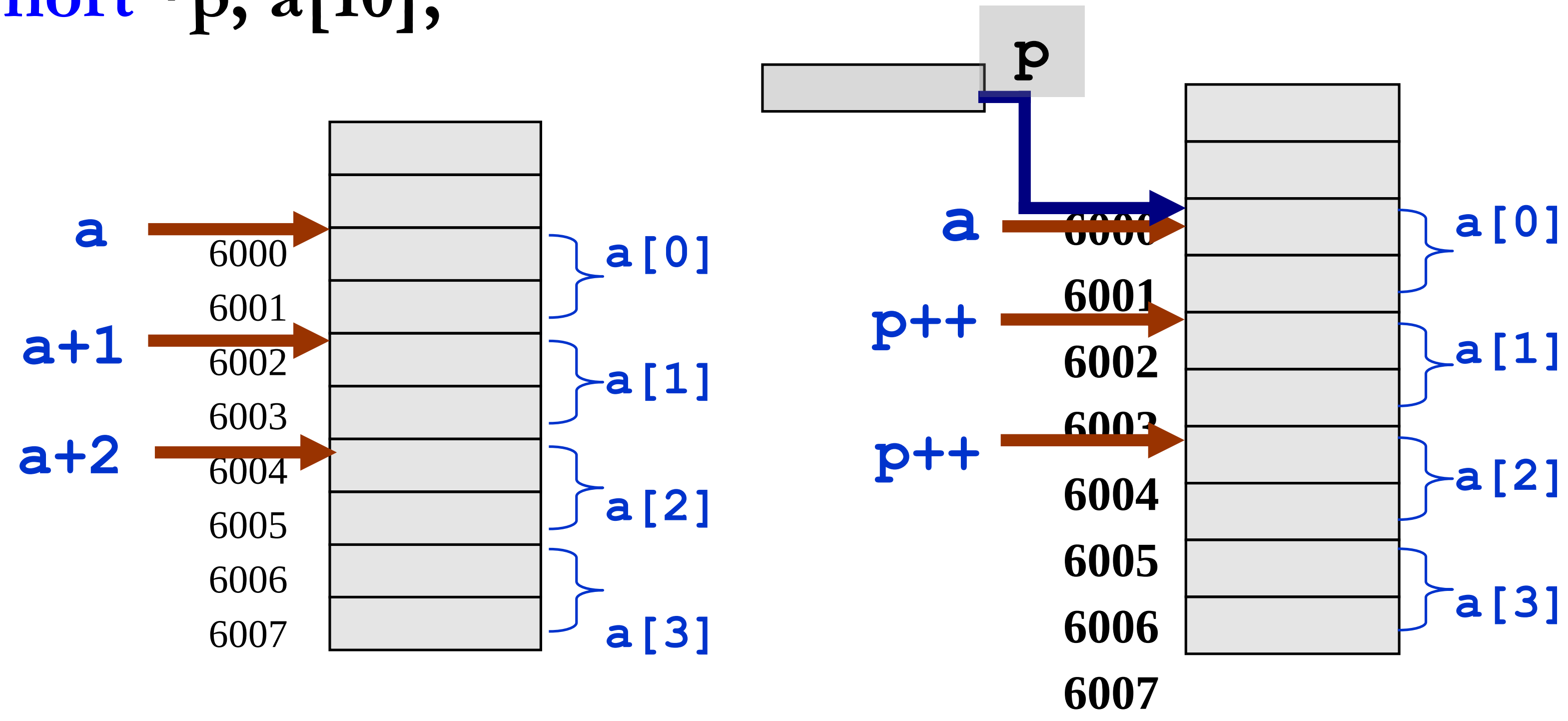
  - The array name (b) is actually the address of first element of the array b[5]

    ```
    bPtr = &b[ 0 ]
    ```

  - Explicitly assigns bPtr to address of first element of b

Pointer and 1-D Arrays

short *p, a[10];



a[i] =*(a+i) = p[i]=*(p+i)

Pointer and 1-D Arrays

The name of an array is the address of the initial element.

a ⇔ &a[0]

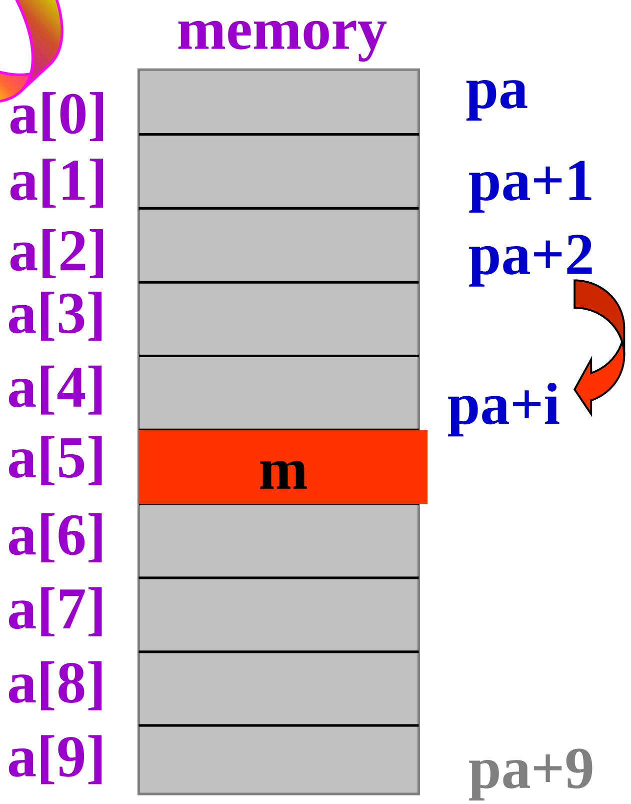
int a[10], *pa;

pa = &a[0]; *pa= a[0];

pa+i } pa=&a[i];

a+i } *pa=a[i];

pa=a;
pa=&a[0]


memory

a[0] pa

a[1] pa+1

a[2] pa+2

a[3]

a[4] pa+i

a[5] m

a[6]

a[7]

a[8]

a[9] pa+9

pa+i ⇔ a+i ⇔ &a[0]+i*m

(m is the memory size for each element)

*(pa+i) ⇔ *(a+i) ⇔ a[i]

(indirect access the array member)

pa+1 points to the next element, and pa+i points to the i-th element next to pa.

Pointer and 1-D Arrays

```
void main( )
{
    int  a[10];
    int  i;

    for (i=0; i<10; i++)
        scanf("%d", &a[i]);

    for (i=0; i<10; i++)
        printf("%d ", a[i]);
}
```

```
void main( )
{
    int  a[10];
    int  *p, i;

    for (p=a; p<(a+10); p++)
        scanf("%d", p);

    for (p=a; p<(a+10); p++)
        printf("%d ", *p);
}
```

Pointer and 1-D Arrays

```
void main( )
{   int a [ ]={ 1,2,3,4,5 };
    int i;
    for(i=0;i<5;i++)
    printf(“%d ”,a[i]);
}
```

```
void main( )
{   int a[ ]={1,2,3,4,5};
     int i;
     int *pa=a;
     for( i=0;i<5;i++)
     printf(“%d”,*(pa+i));
}
```

```
void main( )
{ int a[ ]={1,2,3,4,5 };
   int i;
   for(i=0;i<5;i++)
   printf(“%d ”, * ( a+ i ) );
}
```

Pointer and 1-D Arrays

- incrementing & decrementing pointers

```
int a[10],*p;
```

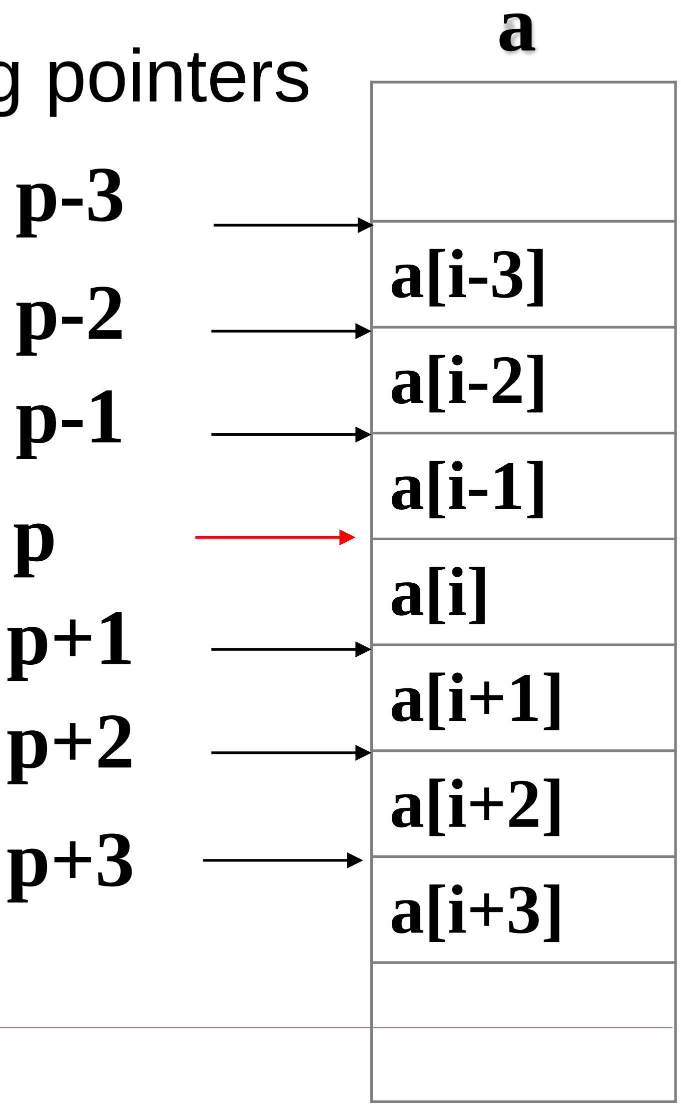
```
p=&a[5];
p++;  p points to  ?
p--;  p points to ?
```

a[6]

a[5]

a++? ×

a is constant



Pointer and 1-D Arrays

```
#include <stdio.h>
void main( )
{ char  s1[80], s2[80],*p1=s1,*p2=s2;
   gets(s1);                        /*enter s1 */
   while(*p1!=‘\0’)/*copy s1 to s2*/
   {   *p2 = *p1;
       p1++;p2++;   /*points to next character*/
   }
   puts(s2);
}
```

right?

Pointer and 1-D Arrays

- Relationship between two pointers

When two pointers points to the same array

```
int a[10],*p=&a[2],*q=&a[4];
```

p<q “true”   p!=q “true”

p>q “false”   p==q “false”

```
for(p=a,q=a+9;p<=q;p++)
   printf(“%d\n”,*p);
```

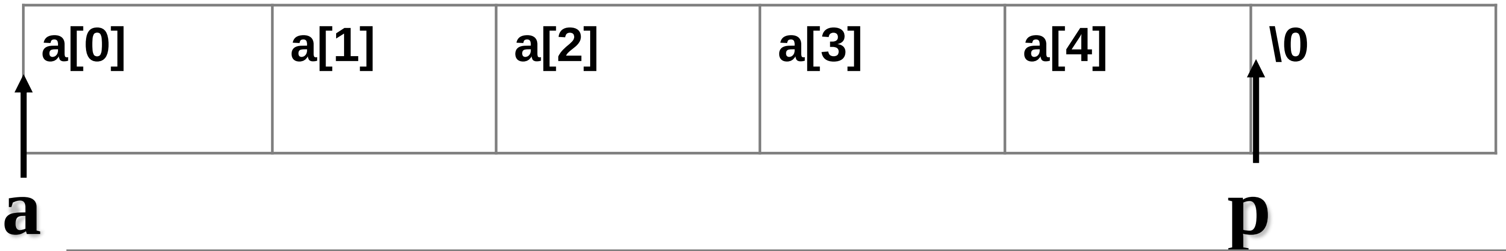
```
for(i=0,q=9;i<=q;i++)
printf(“%d\n”,a[i]);
```

Pointer and 1-D Arrays

- Subtraction between two pointers

When two pointers points to the same array

```
char a[6]=“china”,*p=&a[5];
```



a[0]	a[1]	a[2]	a[3]	a[4]	\0

a

p

p - a = 5, It is the elements number between p and a。 i.e. “china” length.

Pointer and 1-D Arrays

```
#include “stdio.h”
void main( )
 { char s[80];
    gets(s);
    printf(“%s  length=%d\n”, s, strlen(s) );
}
int strlen(char *s)           /* get the length of string */
 {char  *p;
   p=s;
   while(*p!=‘\0’)  p++; /*p points to the end of string*/
    return  p-s;                 /*return length of string */
}
```

Pointer and 1-D Arrays

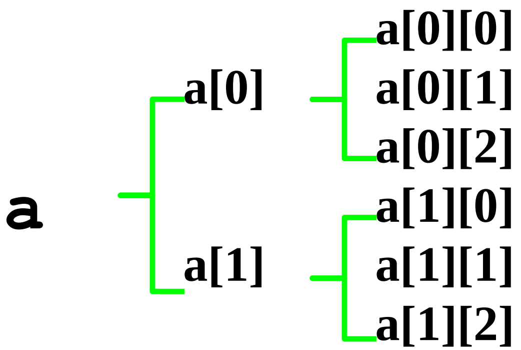
- Passing an Entire Array to a function using pointer
  - In previous chapter, we use the int a[] as parameter to passing an entire array to function.
  - Since the array name is just the address of the zeroth element, so we can passing the address of zeroth element to the function.

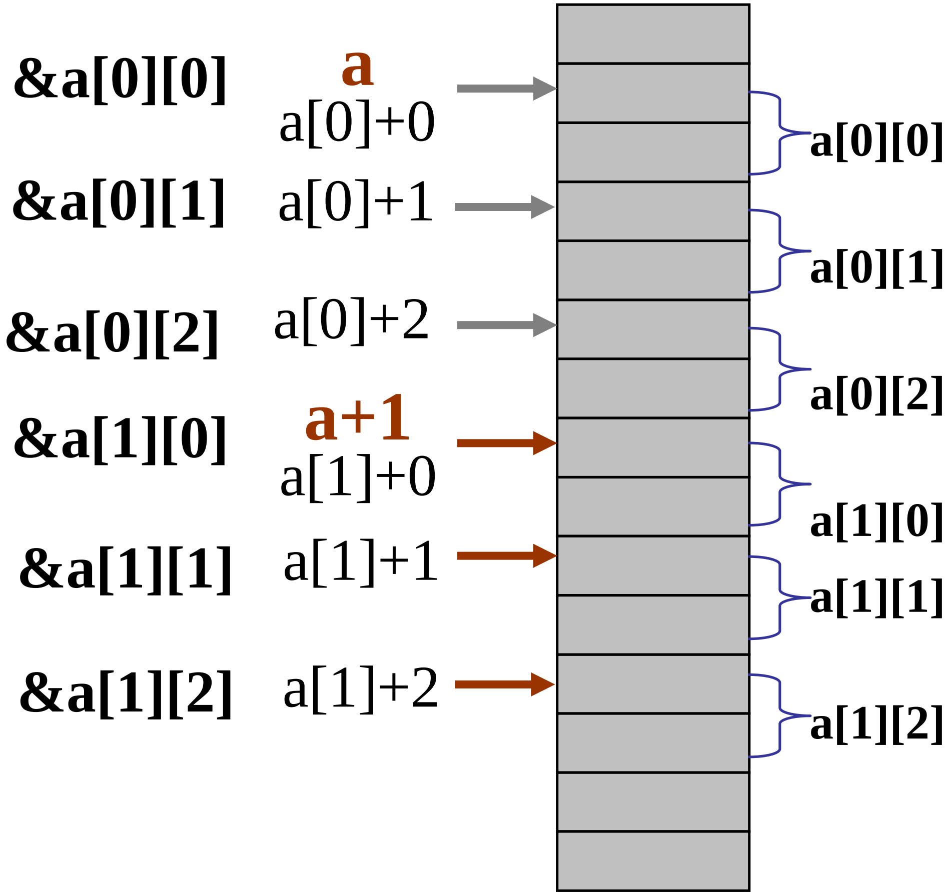
```
#include <stdio.h>
void display(int *,int);
void main()
{   int num[5] = {1,2,3,4,5};
    display(num,5);
}
void display(int a[],int n)
{
    int i;
    for(i=0;i<n;i++)
    {     printf(“%d”,a[i]);
    }
}
```

```
#include <stdio.h>
void display(int *,int);
void main()
{  int num[5] = {1,2,3,4,5};
   display(&num[0],5);
}
void display(int *j,int n)
{
   int i;
   for(i=0;i<n;i++)
   {    printf(“%d”,*j);
        j++;
   }
}
```

Pointer and 2-D Arrays

```
int a[2][3];
```

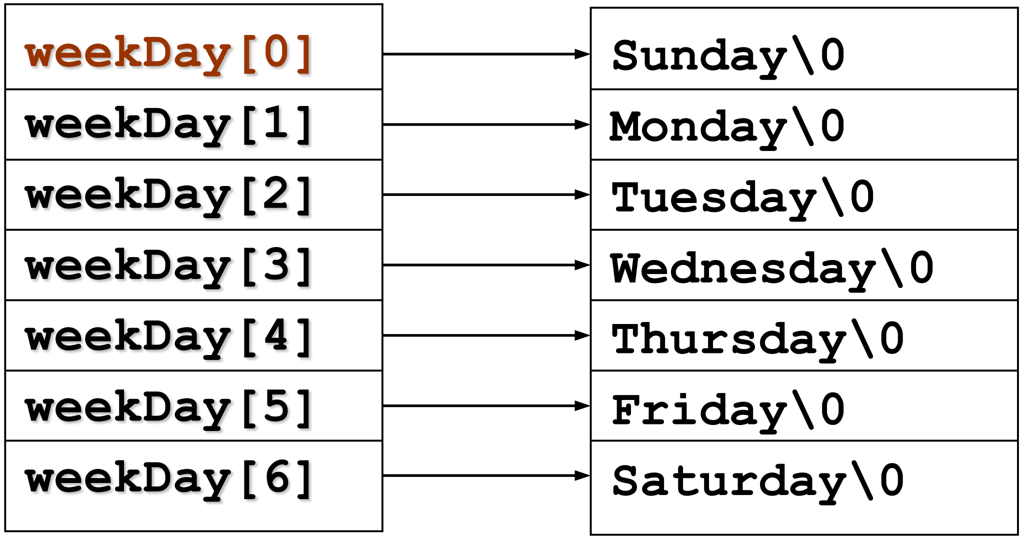




Pointer and 2-D Arrays

Char weekDay[7][10]= {"Sunday","Monday","Tuesday", "Wednesday","Thursday","Friday", "Saturday"};

0	Sunday
1	Monday
2	Tuesday
3	Wednesday
4	Thursday
5	Friday
6	Saturday



weekDay[0] → Sunday\0

weekDay[1] → Monday\0

weekDay[2] → Tuesday\0

weekDay[3] → Wednesday\0

weekDay[4] → Thursday\0

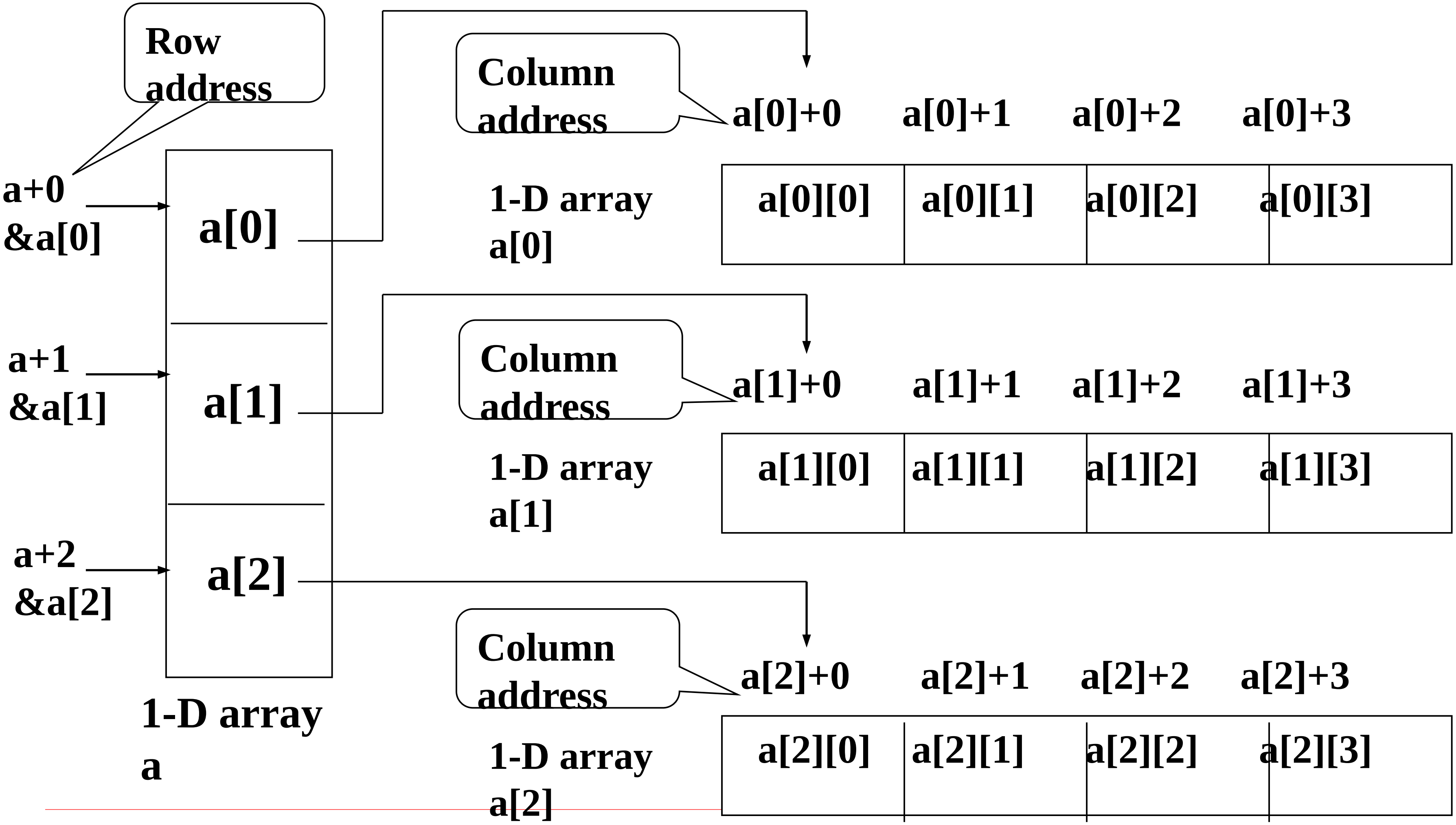
weekDay[5] → Friday\0

weekDay[6] → Saturday\0

```c
#include  <string.h>
void main()
{
    int    i, pos;
    int    findFlag = 0;
    char   x[10];
    char   weekDay[][10] = {"Sunday","Monday","Tuesday",
        "Wednesday","Thursday","Friday","Saturday"};
    printf("Please enter a string:");
    scanf("%s", x);
    for (i=0; i<7 && !findFlag; i++)
    {
        if (strcmp(x, weekDay[i]) == 0)
        {
                pos = i;
                findFlag = 1;
        }
    }
    if (findFlag)
      printf("%s is %d\n", x, pos);
    else
      printf("Not found!\n");
}
```

Address ofweekDay[i][0]

Pointer and 2-D Arrays



Row address

a+0
&a[0]

a[0]

a+1
&a[1]

a[1]

a+2
&a[2]

a[2]

1-D array
a

Column address

a[0]+0 a[0]+1 a[0]+2 a[0]+3

1-D array
a[0]

a[0][0]	a[0][1]	a[0][2]	a[0][3]

Column address

a[1]+0 a[1]+1 a[1]+2 a[1]+3

1-D array
a[1]

a[1][0]	a[1][1]	a[1][2]	a[1][3]

Column address

a[2]+0 a[2]+1 a[2]+2 a[2]+3

1-D array
a[2]

a[2][0]	a[2][1]	a[2][2]	a[2][3]

Pointer and 2-D Arrays

Expression	Meaning
a	two dimensional array name, points to one dimensional array a[0],the address of row 0
a+i,&a[i]	the address of row i
a[i]+0,*(a+i)+0， ,&a[i][0]	the address of the element row i column 0
a[i]+j,*(a+i)+j,&a[i][j]	the address of the element row i column j
(a[i]+j),(*(a+i)+j),a[i][j]	the element row 1 column 2

Pointer and 2-D Arrays

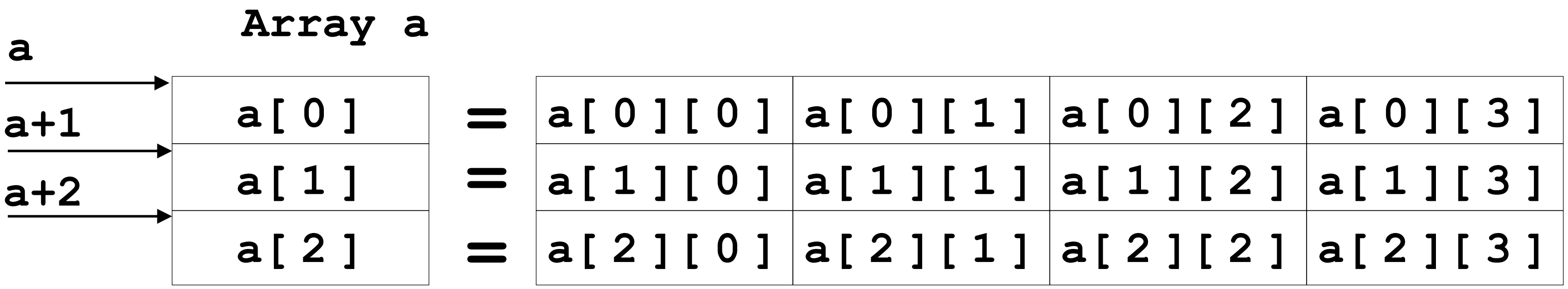
- Expressions equivalent to &a[i][j]
  - &a[i][j]
  - a[i]+j
  - *(a+i)+j
  - &(*(a+i))[j]

- Expressions equivalent to a[i][j]
  - a[i][j]
  - *(a[i]+j)
  - *(*(a+i)+j)
  - (*(a+i))[j]

Pointer and 2-D Arrays

Pointer of Array

- Since a,a+1,a+2 are the address of row 0,row 1 and row2, each row actually is a 1-D array.
- So a,a+1,a+2 can also be treated as the pointer pointing to an array, which are called “pointer of array”.



Array a

a	a[0]	=	a[0][0]	a[0][1]	a[0][2]	a[0][3]
a+1	a[1]	=	a[1][0]	a[1][1]	a[1][2]	a[1][3]
a+2	a[2]	=	a[2][0]	a[2][1]	a[2][2]	a[2][3]

Pointer and 2-D Arrays

- Assuming that we declare an 2-D array int a[3][4], the array name a is a pointer to a 1-D array which has 4 elements.
- We can delcare a pointer to a 4-element 1-D array as int (*p)[4];
- The general form :

```
basic_type  (*pointer_name)[array_size]
```

Pointer and 2-D Arrays

- Relationship between 2-D array and pointer of array

int a[5][10] && int (*p)[10];

  - 2-D array name is a pointer pointing to an 1-D array with 10 elements
  - a+i point to the ith row of 2-D array *(*(a+i)+j) ⇔ a[i][j]
  - int x[ ][10] ⇔ int (*p)[10]

2*5*10 byte

2 byte

Pointer and 2-D Arrays

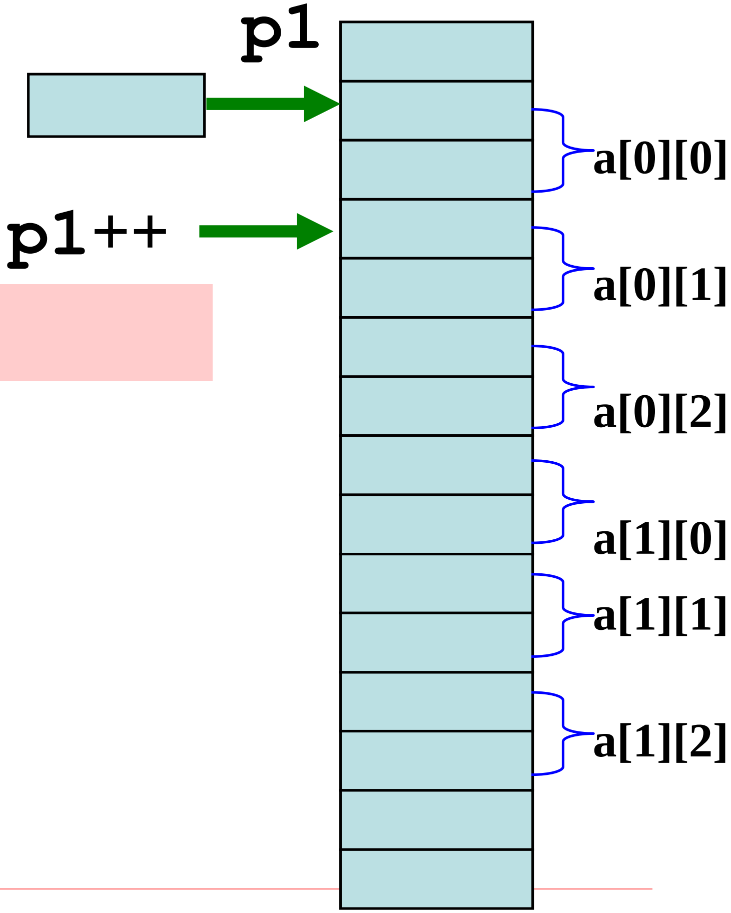
- Column pointer of a[2][3]

int *p1;

p1 = *a; //initialized using column address

offset: i*3+j

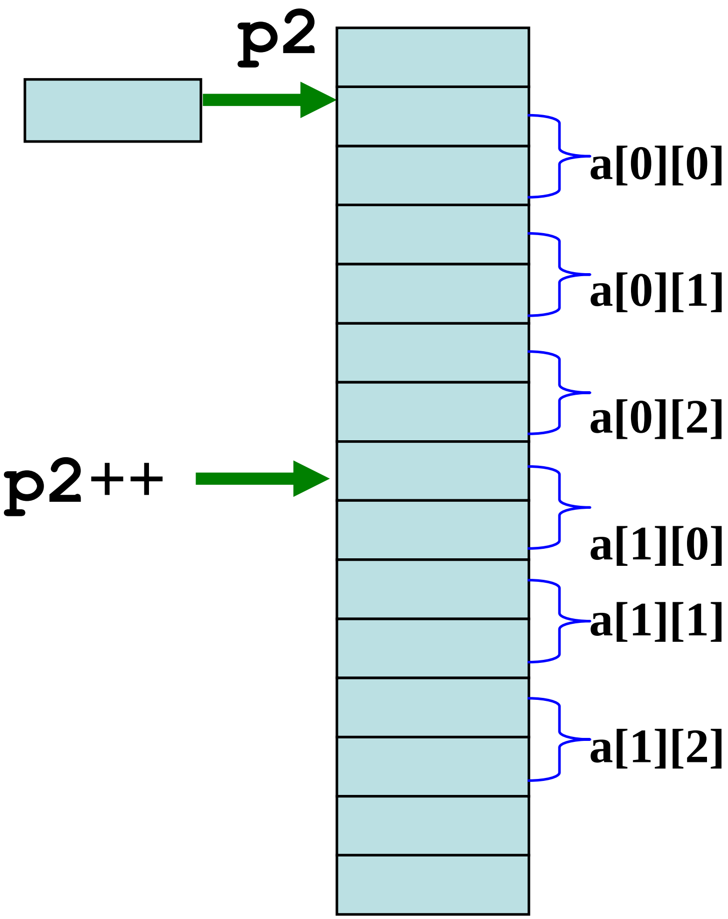
```
for (i=0; i<2; i++)
     for (j=0; j<3; j++)
              printf("%d",*(p1+i*3+j));
```



Pointer and 2-D Arrays

- Row pointer of a[2][3]

```
int (*p2)[3];
p2 = a;
    for (i=0; i<2; i++)
            for (j=0; j<3; j++)
                    printf("%d",*(*(p2+i)+j));
```



Pointer and 2-D Arrays

- When we take the two dimensional array name as the argument, we need to declare a pointer of array as the paramenter in our function.

```
int a[3][4];
int (*p)[4];
```

Pointer and 2-D Arrays

- Function with pointer augments
  - Pointer variable point to variable
  - Pointer variable point to 1-D array
  - 2-D array name

```
int   a[3][4];  int  (*p1)[4]=a;  int  *p2=a[0];
```

Actual argument	Formal argument
Array name a	Array name int x[][4]
Array name a	Pointer int (*q)[4]
Pointer p1	Array name int x[][4]
Pointer p1	Pointer int (*q)[4]
Pointer p2	Pointer int *q

For 3 students, given 4 scores of each student, calculate the average score, and output the score of nth student

```
main()
{ void average(float  *p,int  n);
  void search(float  (*p)[4],int  n);
  float score[3][4]=
{{65,67,79,60},{80,87,90,81},
{90,99,100,98}};
  average(*score,12);
  search(score,2);
}
```

Column pointer

Row pointer

float   p[][4]

p

p

65	52	79	60
80	87	90	81
90	99	100	98

```
void average(float *p,int n)
{   float  *p_end, sum=0,aver;
    p_end=p+n-1;
    for(;p<=p_end;p++)
            sum=sum+(*p);
    aver=sum/n;
    printf("average=%5.2f\n",aver);
}
void search(float  (*p)[4], int n)
{   int i;
    printf(" No.%d  :\n",n);
    for(i=0;i<4;i++)
      printf("%5.2f  ",*(*(p+n)+i));
}
```

⇔ p[n][i]

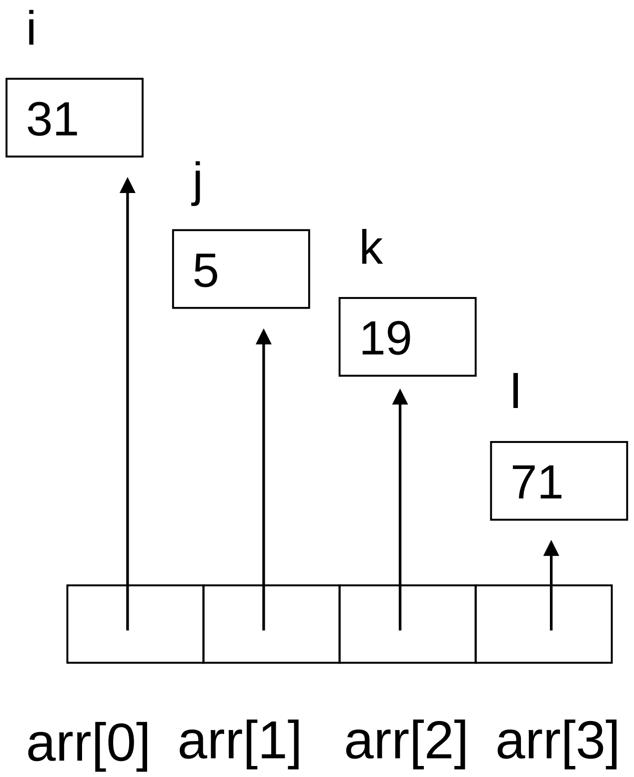
Array of Pointers

- The way there can be an array of ints or an array of floats, similarly, there can be an array of pointers.
- Since a pointer variable always contains an address, an array of pointers would be nothing but a collection of addresses.
- The general form to declare an array of pointers :

```
basic_type  *pointer_name[array_size]
```

Array of Pointers

```
#include <stdio.h>
void main()
{   int *arr[4];
    int i=31,j=5,k=19,l=71,m;
    arr[0]=&i;
    arr[1]=&j;
    arr[2]=&k;
    arr[3]=&l;
    for(m=0;m<=3;m++)
        printf(“%d”,*(arr[m]));
}
```

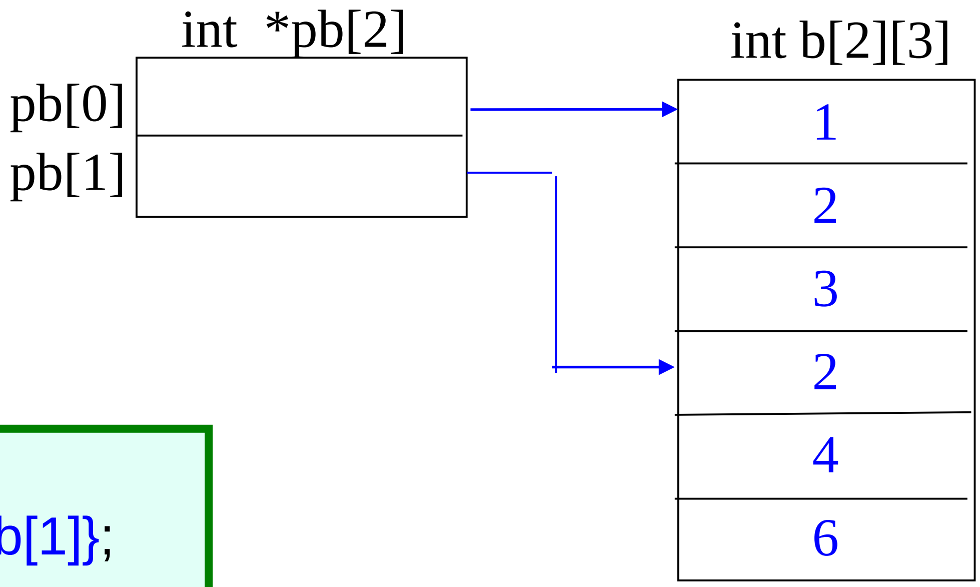


Array of Pointers

- Initialization

```
main()
{   int b[2][3],*pb[2];
    pb[0]=b[0];
    pb[1]=b[1];
    ……..
}
```

```
main()
{   int b[2][3],*pb[ ]={b[0],b[1]};
    ……..
}
```

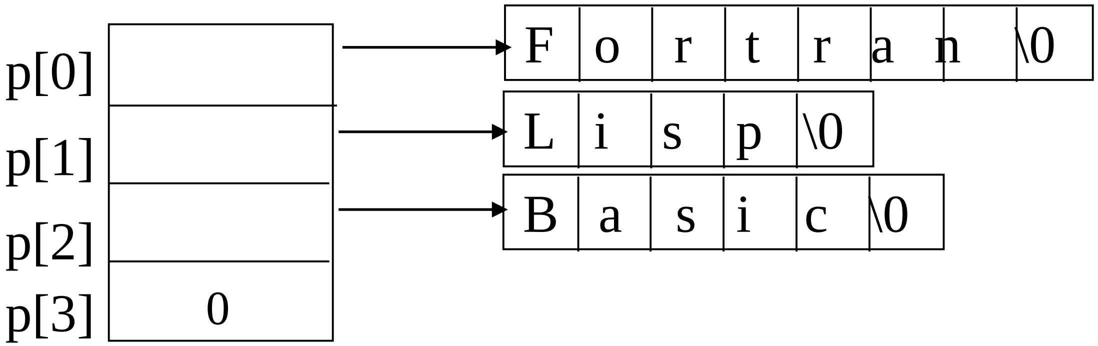


Array of Pointers

```
main()
{    char a[]="Fortran";
     char b[]="Lisp";
    char c[]="Basic";
     char *p[4];
     p[0]=a; p[1]=b; p[2]=c; p[3]=NULL;
       ……..
}
```

```
main()
{   char *p[4];
    p[0]= "Fortran";
    p[1]= "Lisp";
    p[2]= "Basic";
    p[3]=NULL;
      ……..
}
```

```
main()
{ char
*p[]={"Fortran",
"Lisp",
"Basic",NULL};
     ……..
}
```

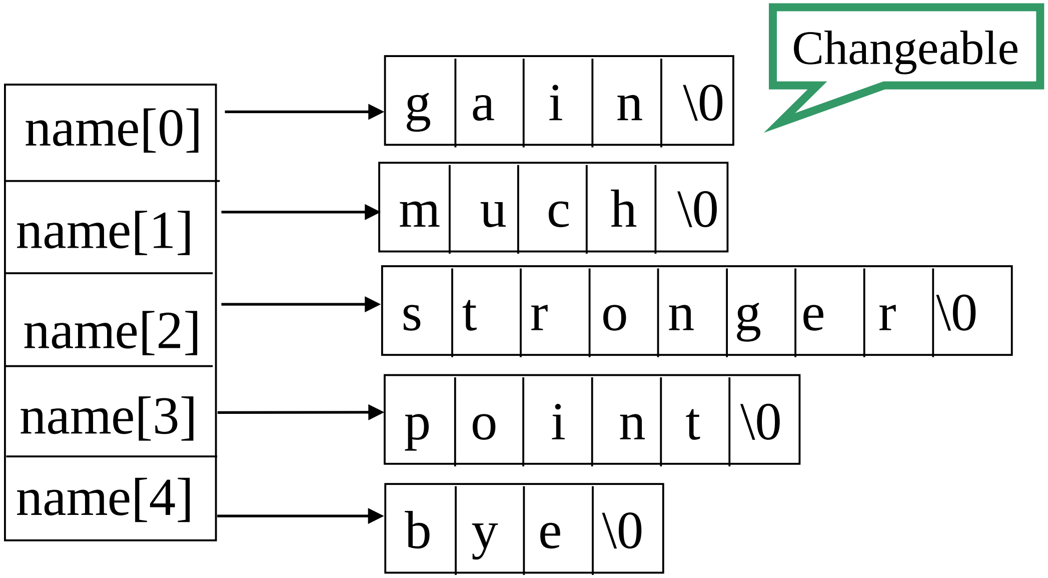


Array of Pointers

char name[5][9]={“gain”,“much”,“stronger”, “point”,“bye”};

g	a	i	n	\0				
m	u	c	h	\0				
s	t	r	o	n	g	e	r	\0
p	o	i	n	t	\0			
b	y	e	\0					

fixed



Changeable

name[0] → g a i n \0

name[1] → m u c h \0

name[2] → s t r o n g e r \0

name[3] → p o i n t \0

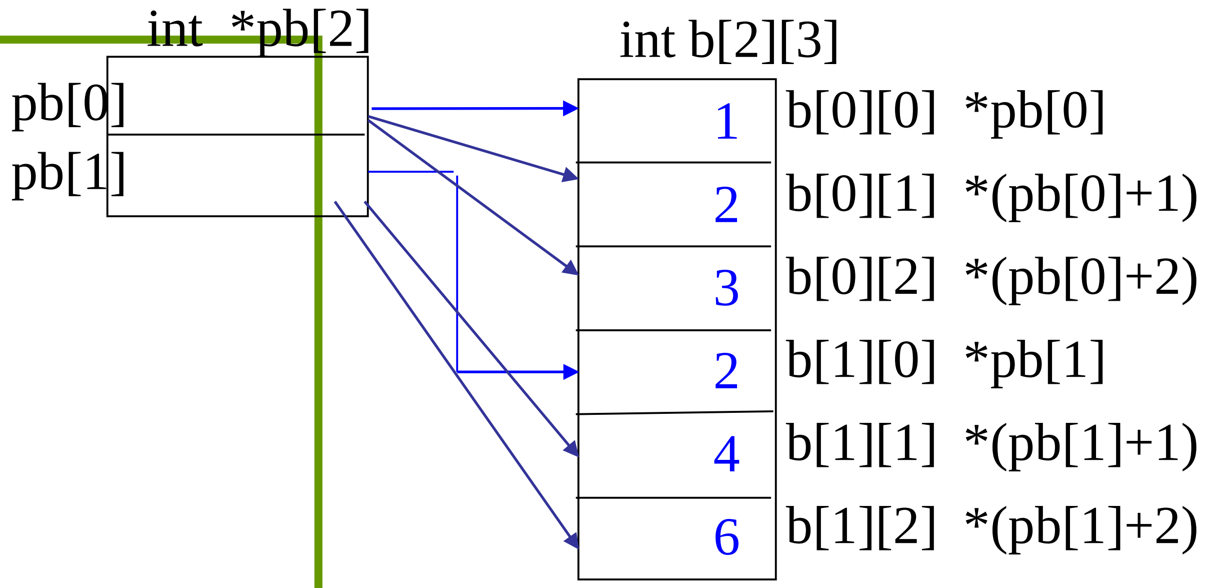
name[4] → b y e \0

char *name[5]={“gain”,“much”,“stronger”, “point”,“bye”};

Element of array of pointer equal to the row name of 2-D array,
But the former is pointer variable, the latter is address constant

Array of Pointers

```
main()
{   int b[2][3],*pb[2];
    int i,j;
    for(i=0;i<2;i++)
      for(j=0;j<3;j++)
              b[i][j]=(i+1)*(j+1);
    pb[0]=b[0];
    pb[1]=b[1];
    for(i=0;i<2;i++)
      for(j=0;j<3;j++,pb[i]++)
              printf("b[%d][%d]:%2d\n",i,j,*pb[i]);
}
```



1	b[0][0]	*pb[0]
2	b[0][1]	*(pb[0]+1)
3	b[0][2]	*(pb[0]+2)
2	b[1][0]	*pb[1]
4	b[1][1]	*(pb[1]+1)
6	b[1][2]	*(pb[1]+2)

Array of Pointers

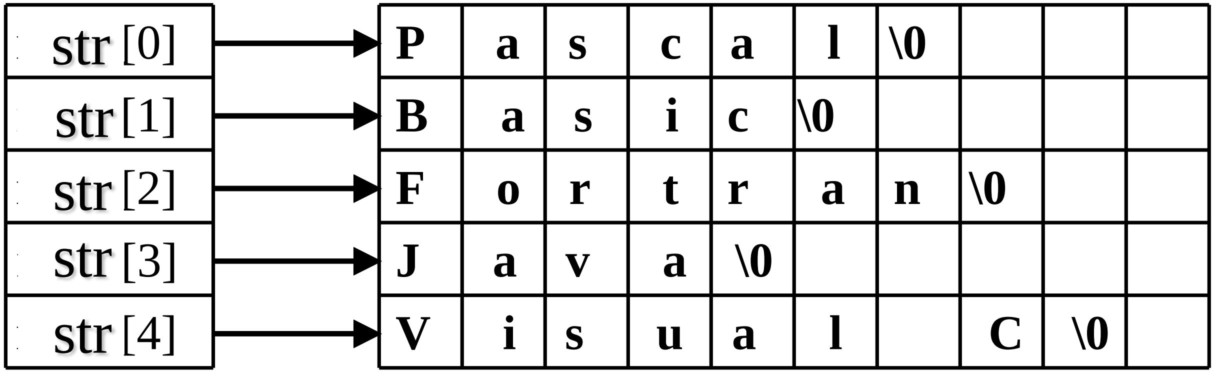
- Sort the strings in lexicographic order

```
char  str[5][10] = {"Pascal","Basic","Fortran",
                 "Java","Visual C"};
char temp[10]={0};
/////////////////////////
for (i=0; i<5-1; i++)
    for (j = i+1; j<5; j++)
           if (strcmp(str[j], str[i]) < 0)
           {
                      strcpy(temp,str[i]);
                       strcpy(str[i],str[j]);
                       strcpy(str[j],temp);
           }
```

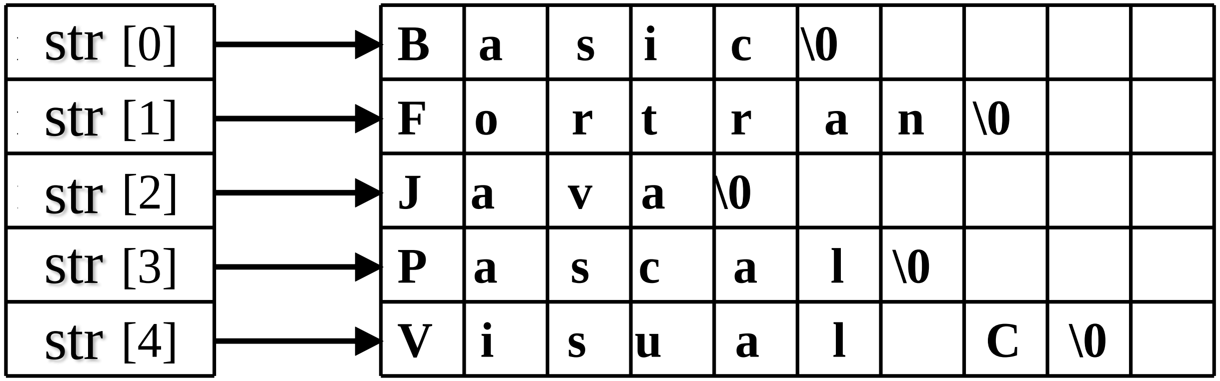
2-D array

Array of Pointers

- Memory layout before and after sort



str [0]	P	a	s	c	a	l	\0			
str [1]	B	a	s	i	c	\0				
str [2]	F	o	r	t	r	a	n	\0		
str [3]	J	a	v	a	\0					
str [4]	V	i	s	u	a	l		C	\0	



str [0]	B	a	s	i	c	\0				
str [1]	F	o	r	t	r	a	n	\0		
str [2]	J	a	v	a	\0					
str [3]	P	a	s	c	a	l	\0			
str [4]	V	i	s	u	a	l		C	\0	

Array of Pointers

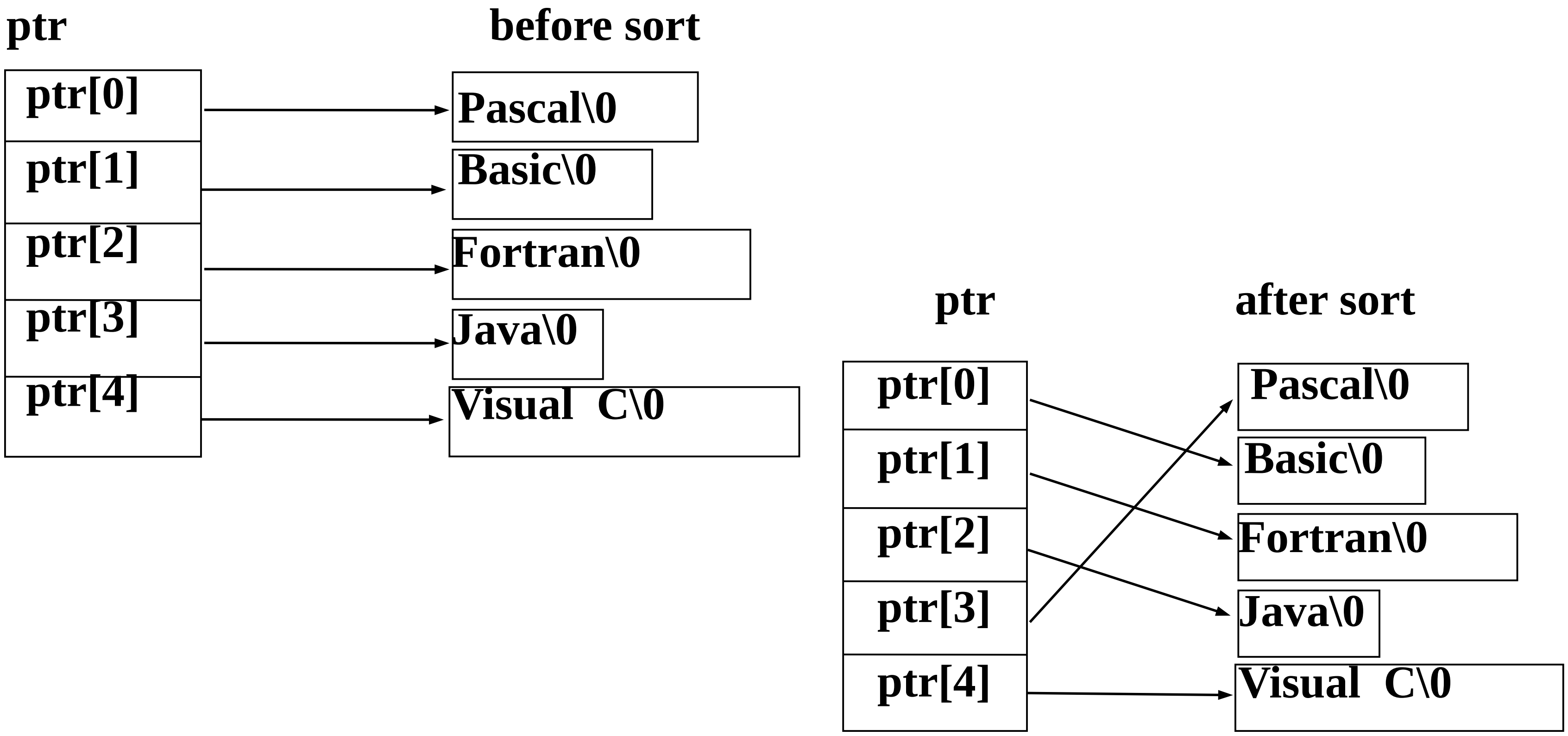
- Sort the strings in lexicographic order

```
char  *ptr[N] = {"Pascal","Basic","Fortran",
    "Java","Visual C"};
char *temp=NULL;
/////////////////////////////
for (i=0; i<N-1; i++)
    for (j = i+1; j<N; j++)
         if (strcmp(ptr[j], ptr[i]) < 0)
          {
                    temp = ptr[i];
                    ptr[i] = ptr[j];
                   ptr[j] = temp;
          }
```

Array of pointers

Array of Pointers

- Memory layout before and after sort



ptr

before sort

ptr[0]

ptr[1]

ptr[2]

ptr[3]

ptr[4]

Pascal\0

Basic\0

Fortran\0

Java\0

Visual C\0

ptr

after sort

ptr[0]

ptr[1]

ptr[2]

ptr[3]

ptr[4]

Pascal\0

Basic\0

Fortran\0

Java\0

Visual C\0

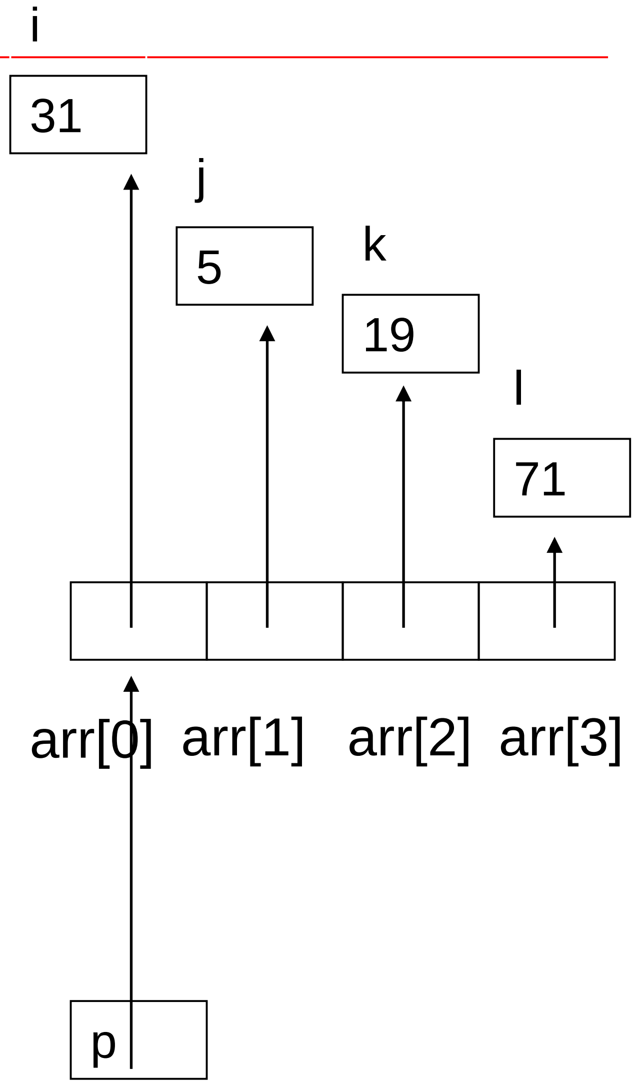
Pointer to Pointer

- When we declare an array of pointers such as *int *arr[4]*,consider the array name *arr*, since *arr[0*] is a pointer to *int*, and *arr* is the address of *arr[0], arr* can be treated as a pointer to pointer.
- The general form to declare a pointer to pointer :

  *basic_type **pointer_name*

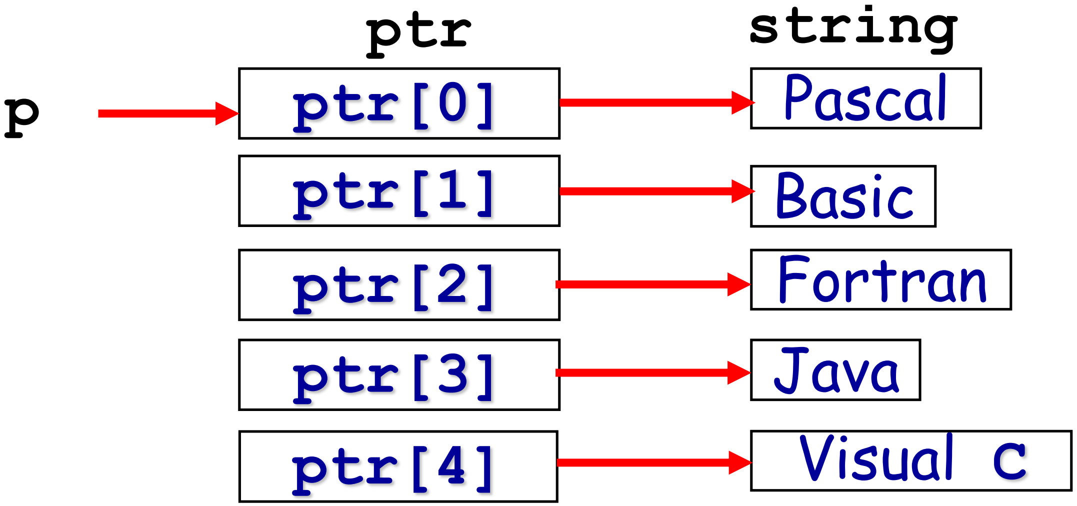
Pointer to Pointer

```
#include <stdio.h>
void main()
{   int *arr[4],**p;
    int i=31,j=5,k=19,l=71,m;
    arr[0]=&i;   arr[1]=&j;
    arr[2]=&k;  arr[3]=&l;
    p=arr;
    for(m=0;m<=3;m++)
          printf("%d",*(*(p+m));
}
```



Pointer to Pointer

```
void main()
{
  int i;
  char*ptr[] = {"Pascal","Basic","Fortran",
                  "Java","Visual C"};
  char **p;
  p = ptr;
     for (i=0; i<5; i++)
     {
    printf("%s\n", *p);
    p++;
  }
}
```



Dynamic Allocation

- Memory allocation

```
void * malloc ( unsigned size );
```

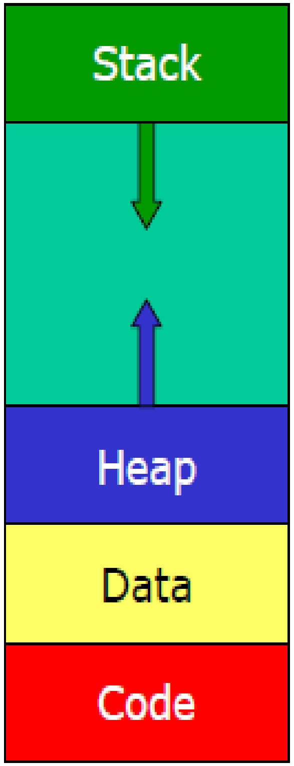
If memory allocation succeed, return the initial address of the memory block, or else, return NULL.

Example:

```
int  * p;
p = (int *) malloc( 10 * sizeof(int ) ); //dynamic array
if (p == NULL)
      printf(“No memory available”);
```

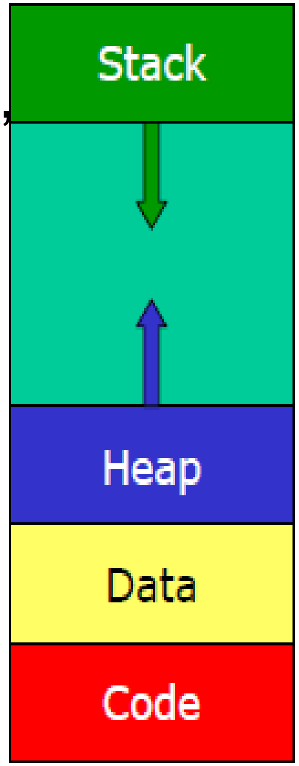
Dynamic Allocation

- Stack-stores variables that are local to functions
  - All **static** memory is allocated from the stack
  - when a functions is called, its automatic variables are allocated on the top of the stack
  - when it ends its variables are de-allocated



Dynamic Allocation

- Heap
  - place for variables that are created with 'new' and disposed by 'unchecked_deallocation'
  - Dynamic memory is allocated from the heap
  - Data: initalized variables including global and static variables
  - Code (text): program instructions to be executed



Dynamic Allocation

Stack vs. Heap

- Stack
    - Grows “down”
    - Operations always take place at the top,Push and pop are well organized
    - Support for nested functions and recursion
- Heap
    - Grows “up”
    - The order in which objects are created or destroyed is completely under the control of the programmer
    - You can have ‘holes’
    - Dynamic memory management
    - Memory fragmentation - memory fragments into small blocks over lifetime of program

Dynamic Allocation

```
//main.c
int a = 0; //Global Initialization area
char *p1; //Initialization area
int main(void)
{
        int b; //stack
        char s[ ] = “abc”; // stack
        char *p2; // stack
        char *p3 = “123456”; //123456\0: Constant area, p3: Stack
        static int c =0;   /Global (static) Initialization area
        p1 = (char *)m    oc(
        p2 = (char *)m    oc(          namic allocation, heap */
        strcpy(p1, “123456”); /*123456\0: Constant area
        return 0;
    }
```

Same place

Dynamic Allocation

- Freeing memory

void free(void* ptr);

```
#include <stdlib.h>
void main ( )
{       int *p;
         p = (int *) malloc( 10 * sizeof(int) );
         printf( “\n Result:” );
         try ( p, 10);
         free(p);
}
void try ( int a[ ], int m )
{            int k;
             for ( k=0; k<m; k++ )       a [ k ] = k*10;
             for ( k=0; k<m; k++ )       printf ( “%d,” , a[k]);
}
```

String, Character Array and Pointer

- String
  - characters ending in the null terminator ('\0'), which indicates where a string terminates in memory.
  - access via character array or character pointer
- Character Array
  - An array of characters
  - char string[100];
- Character Pointer
  - points to the first character in the string
  - char* p;

String, Character Array and Pointer

- Definition

char str[10]; //array	char *ptr;//pointer

- Initialization

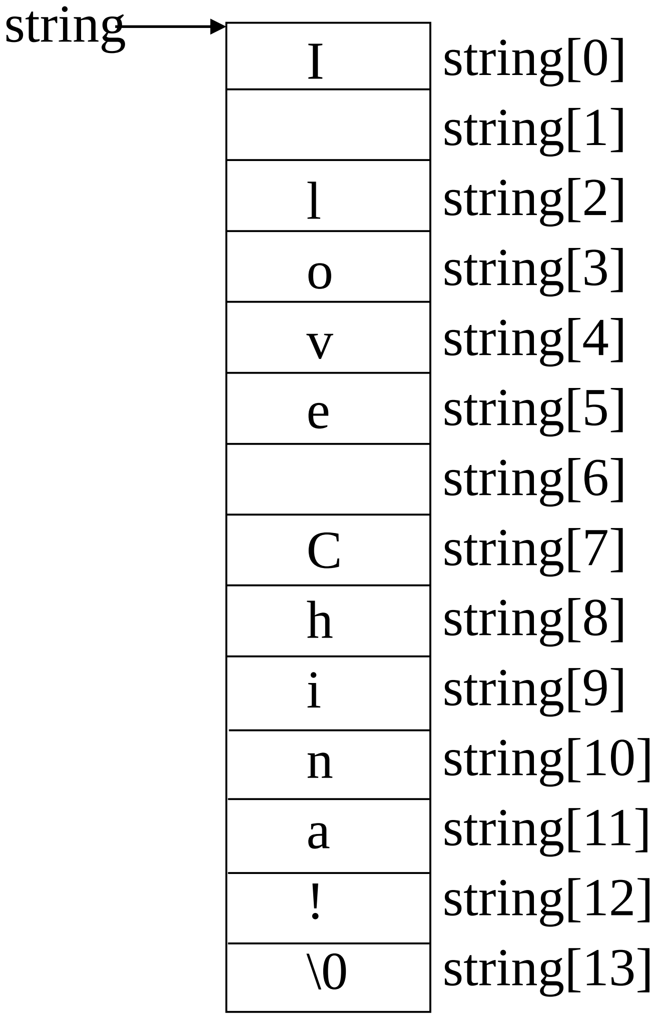
char str[10] = “china”;	char *ptr;
or:	
char str[10];	ptr = “china”;
strcpy(str, “china”);	

String, Character Array and Pointer

□ Character array

```
main( )
     {   char string[]="I love China!";
         printf("%s\n",string);
          printf("%s\n",string+7);
     }
```

```
I love China!
China!
```



string

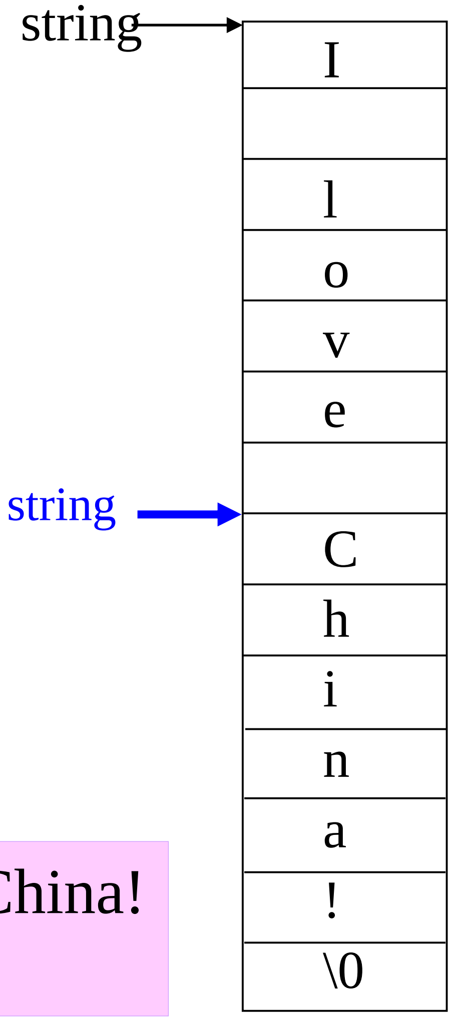
I	string[0]
	string[1]
l	string[2]
o	string[3]
v	string[4]
e	string[5]
	string[6]
C	string[7]
h	string[8]
i	string[9]
n	string[10]
a	string[11]
!	string[12]
\0	string[13]

String, Character Array and Pointer

- Character pointer

```
main( )
    {   char  *string="I  love China!";
        printf("%s\n",string);
        string+=7;
        while(*string)
         {      putchar(string[0]);
                string++;
          }
     }
```

*string!=0



```
I love China!
China!
```

String, Character Array and Pointer

```
char  str[10];
scanf("%s", str);   /*right*/
```

Allocate memory

```
char *a;
scanf(“%s”, a);
/*wrong */
```

Doesn’t allocate memory

```
char *a;
char str[10];
a = str;
scanf("%s", a);
/*right*/
```

String, Character Array and Pointer

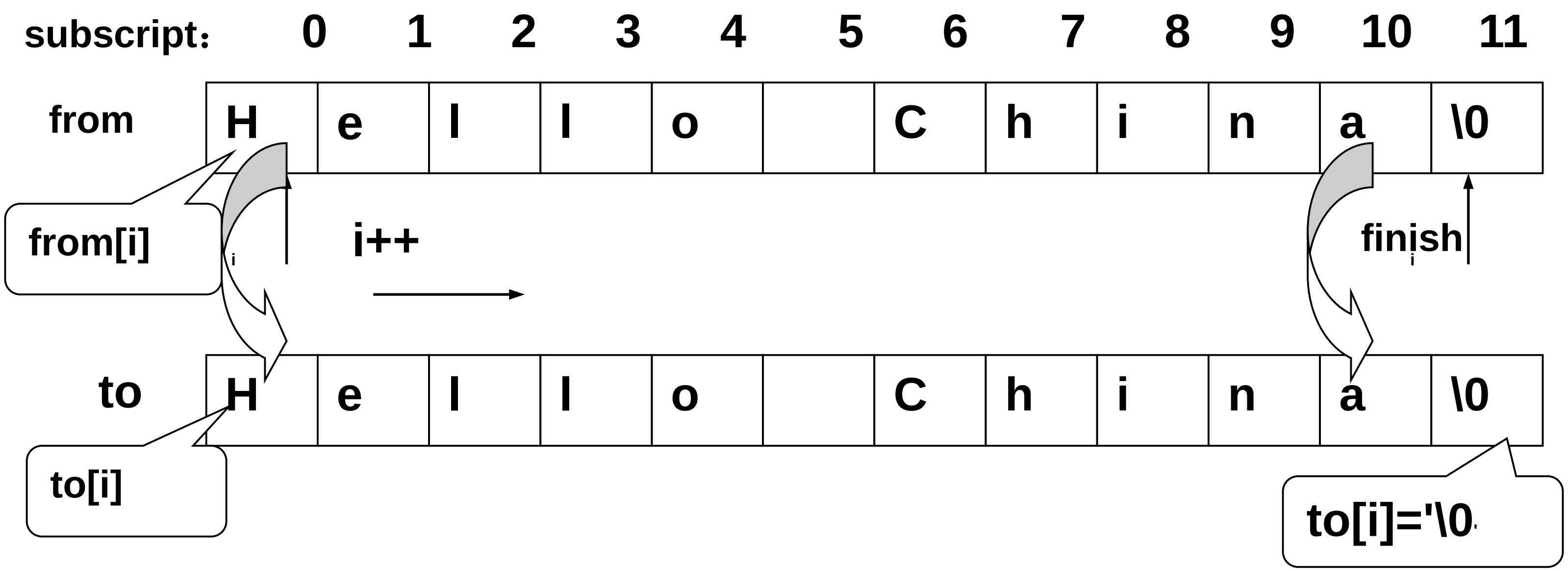
- String Copy

```
void  MyStrcpy(char to[ ], char from[ ])
{
    int  i = 0;
    while (from[i] != '\0')
    {
          to[i] = from[i];
          i++;
    }
    to[i] = '\0';
}
```

Character Array

String, Character Array and Pointer

- String Copy



subscript:	0	1	2	3	4	5	6	7	8	9	10	11
from	H	e	l	l	o		C	h	i	n	a	\0
to	H	e	l	l	o		C	h	i	n	a	\0

String, Character Array and Pointer

- String Copy

```
void  MyStrcpy(char *to, const char *from)
{
    while (*from != '\0')
    {
          *to = *from;
          from++;
          to++;
    }
    *to = '\0';
}
```

Character Pointer

Summary

int *p;	p is a pointer variable which points to int type data.
int *q[4];	q is a pointer array in where there are four pointer elements. Each pointer points to an integer.
int (*w)[4];	w is an array pointer which points to a one-dimension array. Array includes four integer elements.
int *g( );	g is a function. * means return value of function g is a pointer which points to an integer.
int (*y) ();	y is a pointer. () means pointer y points to a function and return value is integer.

Summary

- Pointer variables contain memory addresses as their values.
- An 1-D array name by itself is an address, or pointer value, and pointers.
- An 2-D array name by itself is the address of row 0, which is treated as a pointer of 1-D array.
- Since a pointer variable always contains an address, an array of pointers would be nothing but a collection of addresses.

Thank you!