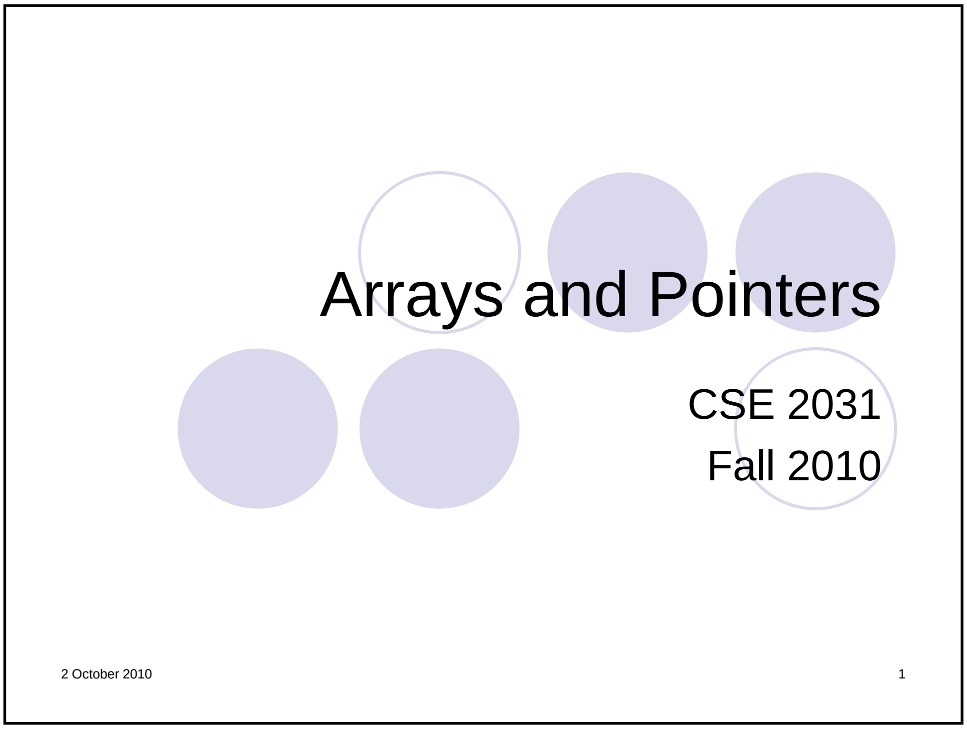
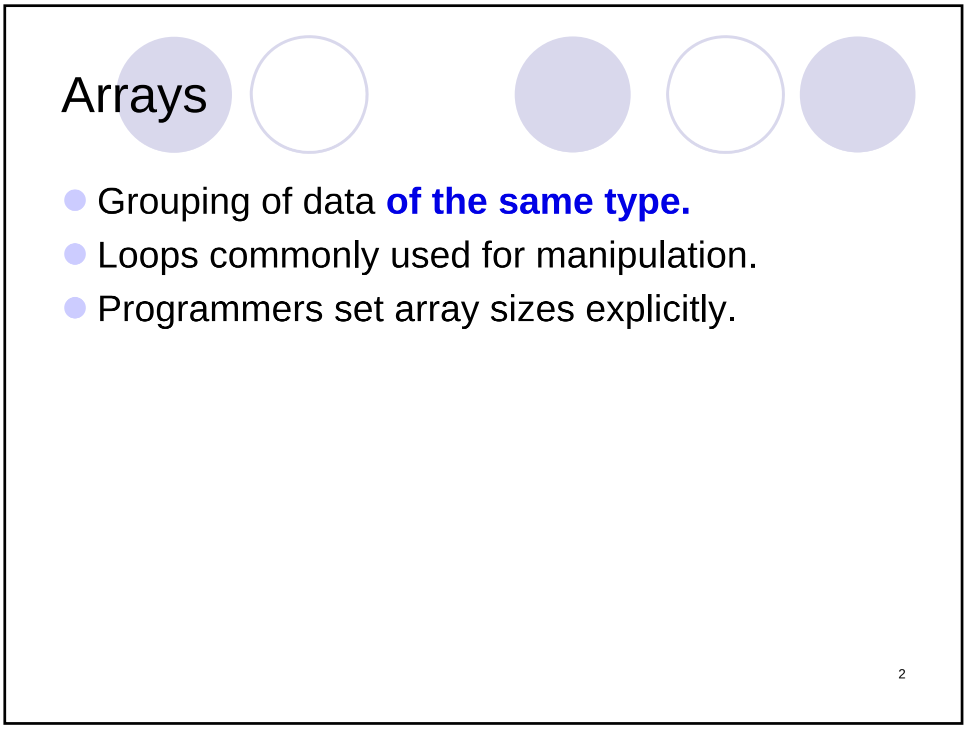


Arrays and Pointers

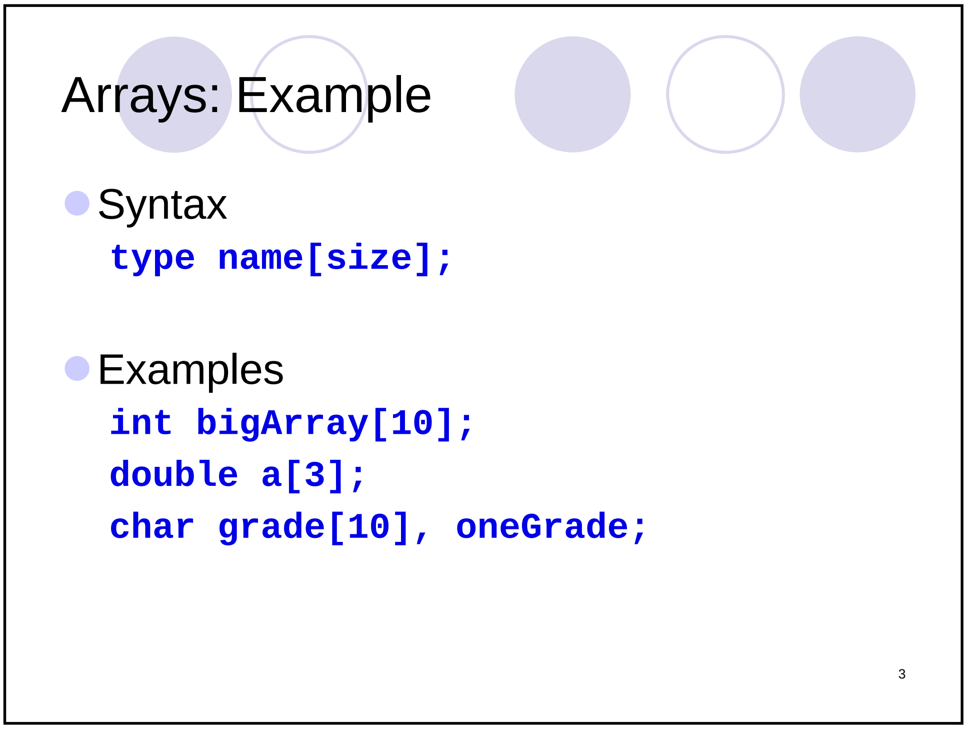
CSE 2031

Fall 2010



Arrays

- Grouping of data **of the same type.**
- Loops commonly used for manipulation.
- Programmers set array sizes explicitly.



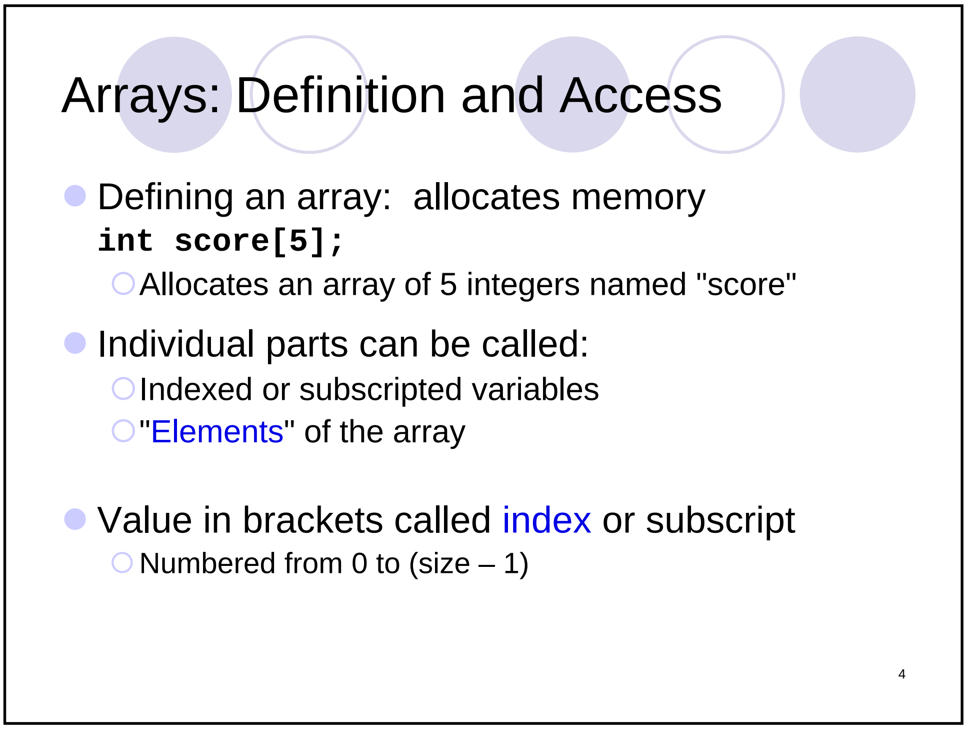
Arrays: Example

- Syntax

```
type name[size];
```

- Examples

```
int bigArray[10];
double a[3];
char grade[10], oneGrade;
```

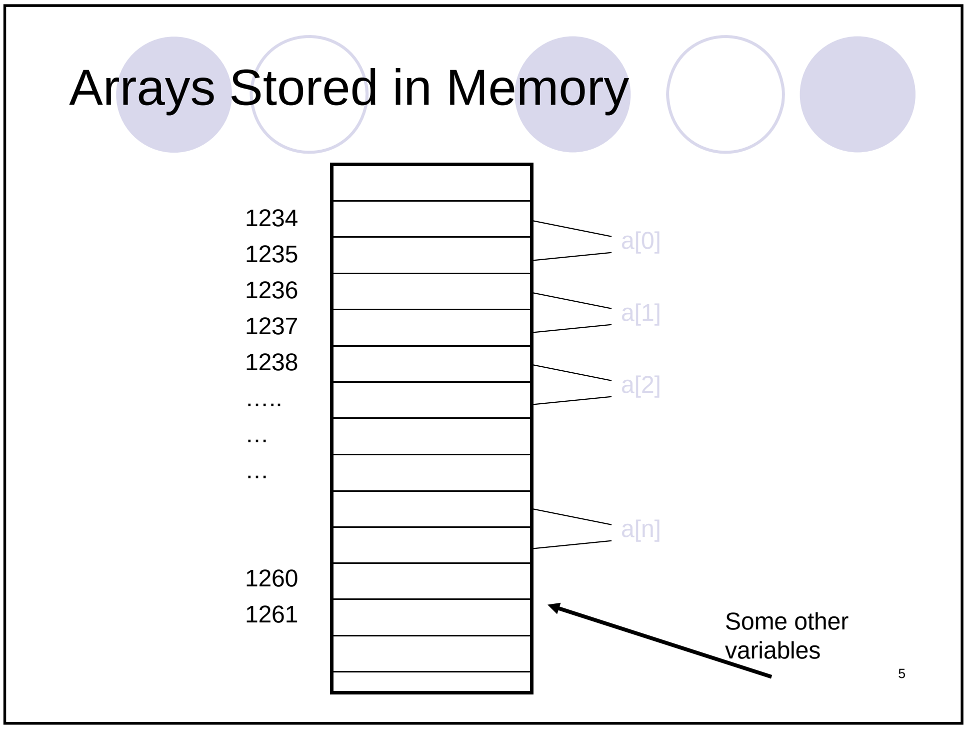


Arrays: Definition and Access

- Defining an array: allocates memory

```
int score[5];
```

  - Allocates an array of 5 integers named "score"
- Individual parts can be called:
  - Indexed or subscripted variables
  - "Elements" of the array
- Value in brackets called index or subscript
  - Numbered from 0 to (size – 1)



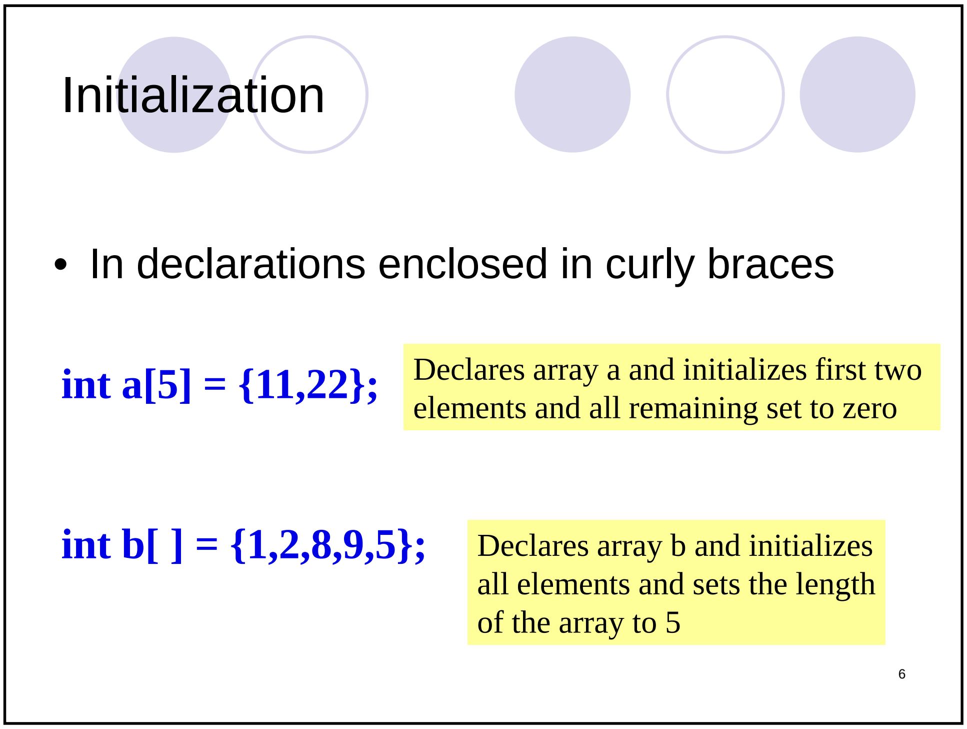
Arrays Stored in Memory

1234
1235
1236
1237
1238
…..
…
…

1260
1261

a[0]
a[1]
a[2]
a[n]

Some other variables



Initialization

- In declarations enclosed in curly braces

int a[5] = {11,22};

Declares array a and initializes first two elements and all remaining set to zero

int b[ ] = {1,2,8,9,5};

Declares array b and initializes all elements and sets the length of the array to 5

Array Access

```
x = ar[2];
ar[3] = 2.7;
```

- What is the difference between `ar[i]++, ar[i++], ar[++i]` ?

Strings

- No **`string`** type in C
- String = array of char
- char gretings[ ] = “Hello”

H	e	l	l	o	\0

Pointers

CSE 2031
Fall 2010

Pointers and Addresses (5.1)

- Memory address of a variable
- Declared with data type, * and identifier

```
type * pointer_var1, * pointer_var2, …;
```

- Example.

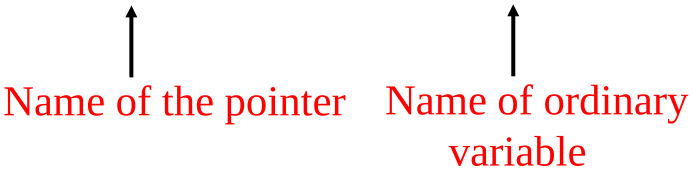
```
double * p;
int *p1, *p2;
```

- There has to be a * before EACH of the pointer variables

Pointers and Addresses (cont.)

- Use the **"address of"** operator (&)
- General form:

pointer_variable = &ordinary_variable



Name of the pointer

Name of ordinary variable

Using a Pointer Variable

- Can be used to access a value
- Unary operator * used

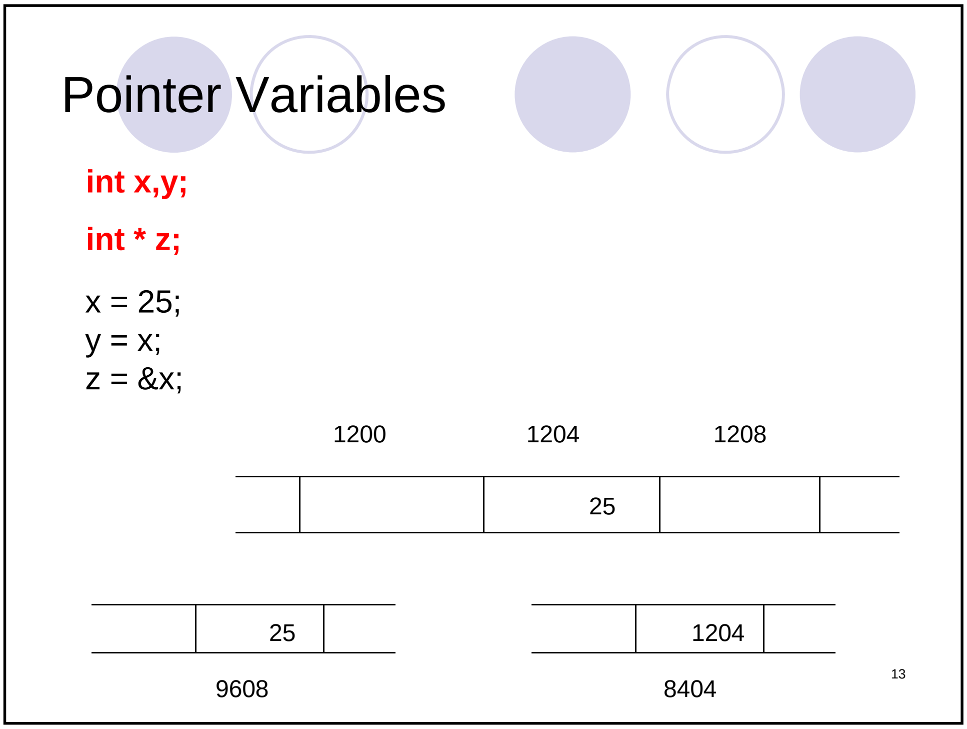
  * pointer_variable

  - In executable statement, indicates value
- Example

```
int *p1, v1;
v1 = 0;
p1 = &v1;
*p1 = 42;
printf(“%d\n“,v1);
printf(“%d\n,*p1);
```

Output:

```
42
42
```



Pointer Variables

int x,y;

int * z;

x = 25;
y = x;
z = &x;

1200 1204 1208

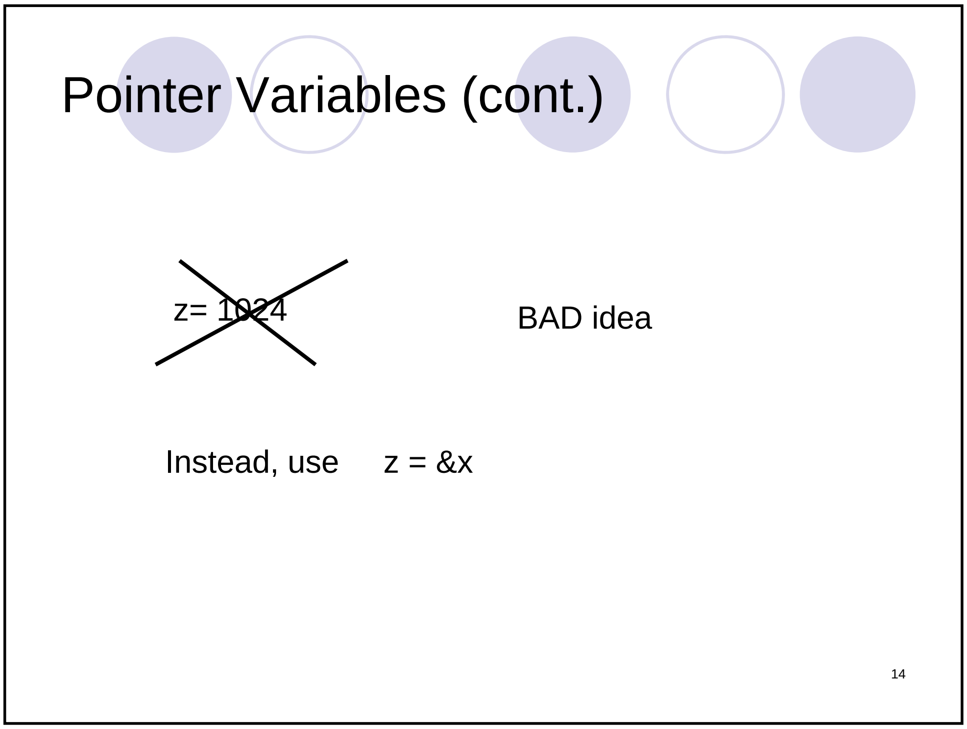
25

25

9608

1204

8404

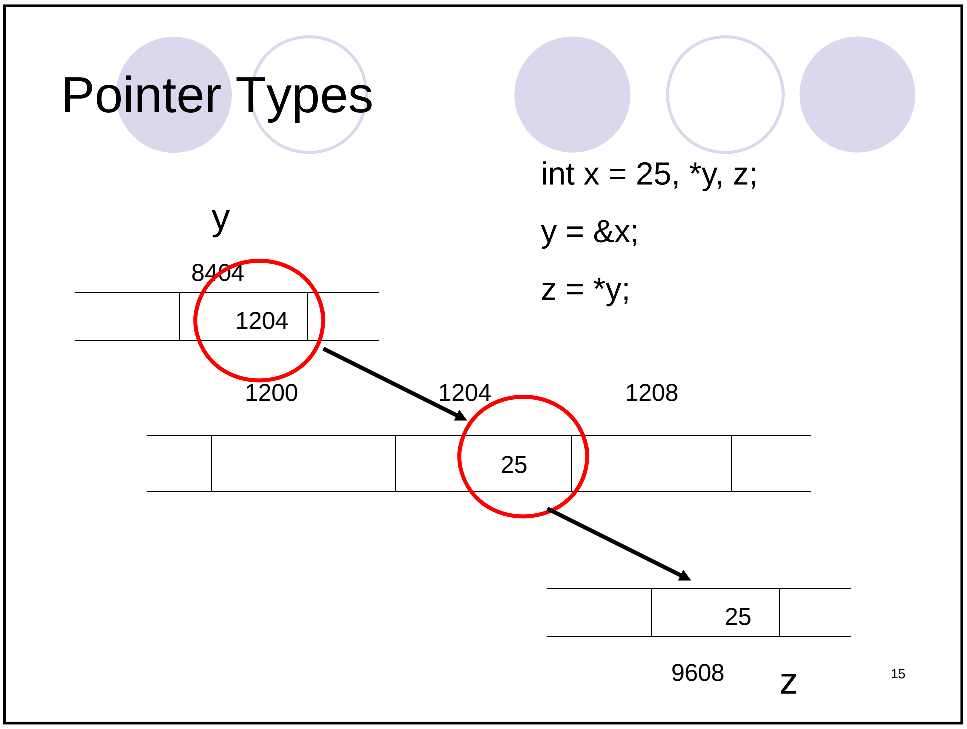


Pointer Variables (cont.)

~~z= 1024~~

BAD idea

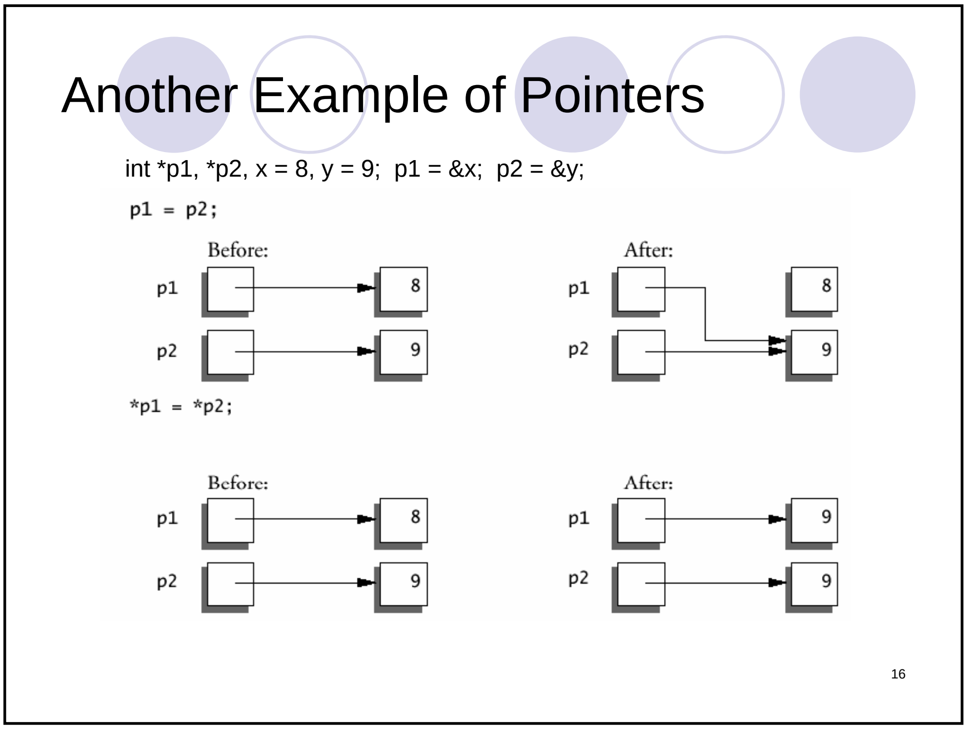
Instead, use z = &x



Pointer Types

```
int x = 25, *y, z;
y = &x;
z = *y;
```

15

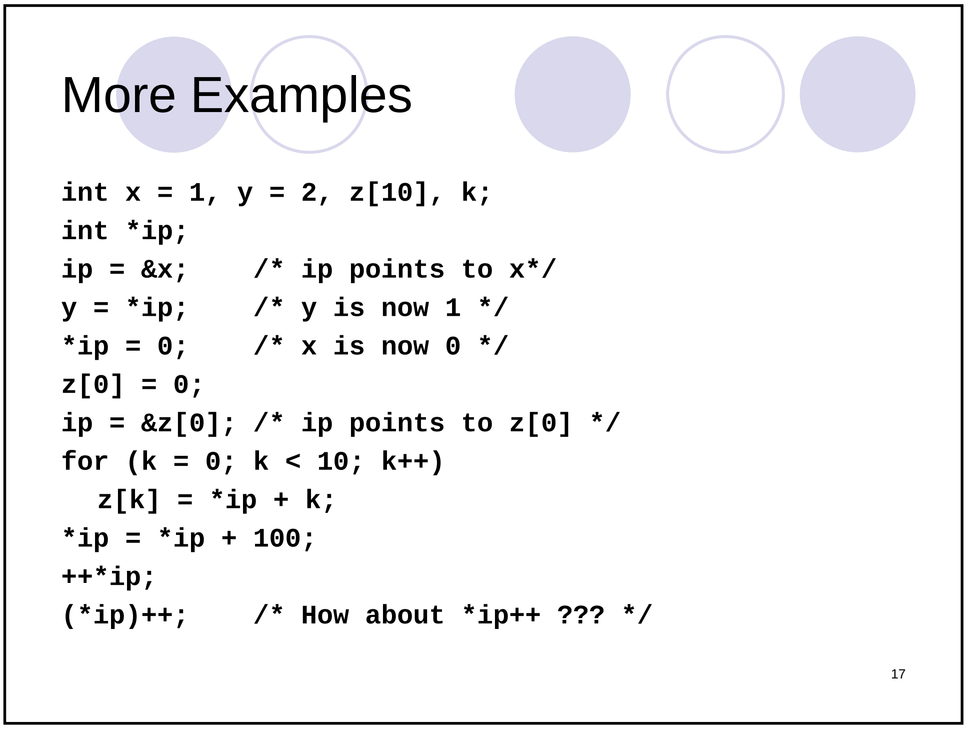


Another Example of Pointers

```
int *p1, *p2, x = 8, y = 9;  p1 = &x;  p2 = &y;
p1 = p2;
```

```
*p1 = *p2;
```

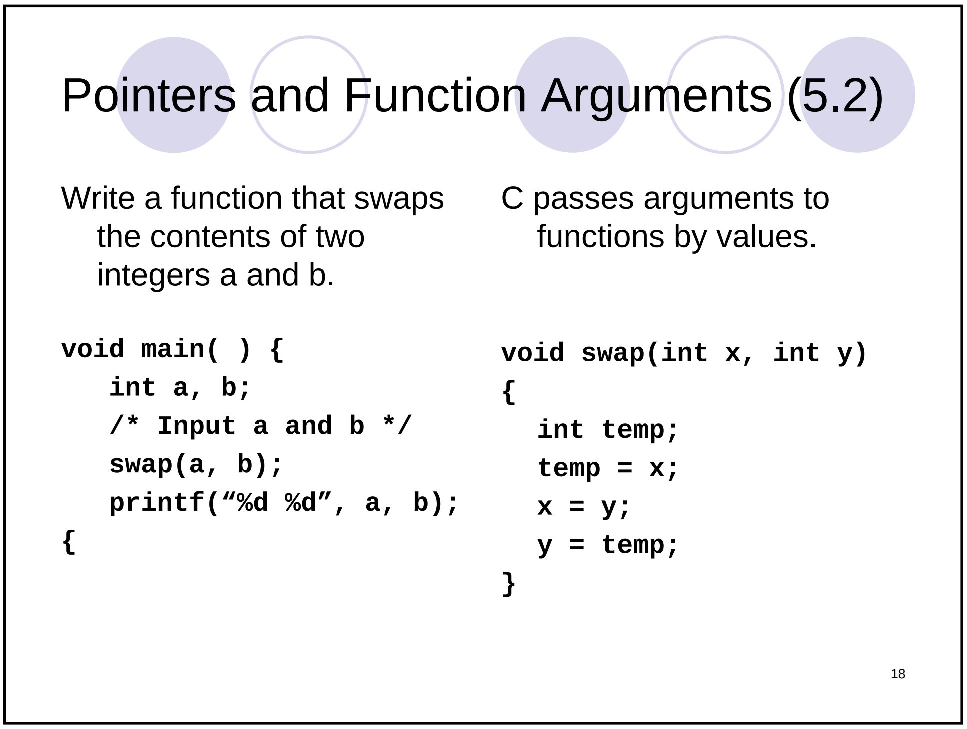
16



More Examples

```
int x = 1, y = 2, z[10], k;
int *ip;
ip = &x;    /* ip points to x*/
y = *ip;    /* y is now 1 */
*ip = 0;    /* x is now 0 */
z[0] = 0;
ip = &z[0]; /* ip points to z[0] */
for (k = 0; k < 10; k++)
  z[k] = *ip + k;
*ip = *ip + 100;
++*ip;
(*ip)++;    /* How about *ip++ ??? */
```

17



Pointers and Function Arguments (5.2)

Write a function that swaps the contents of two integers a and b.

```
void main( ) {
   int a, b;
   /* Input a and b */
   swap(a, b);
   printf(“%d %d”, a, b);
{
```

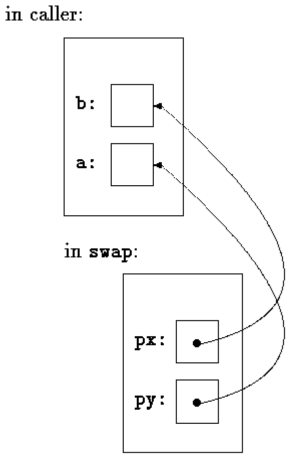
C passes arguments to functions by values.

```
void swap(int x, int y)
{
  int temp;
  temp = x;
  x = y;
  y = temp;
}
```

18

The Correct Version

```
void swap(int *px, int *py)
{
   int temp;
   temp = *px;
   *px = *py;
   *py = temp;
}

void main( ) {
    int a, b;
    /* Input a and b */
    swap(&a, &b);
    printf(“%d %d”, a, b);
{
```



Arrays and Pointers

Pointers and Arrays (5.3)

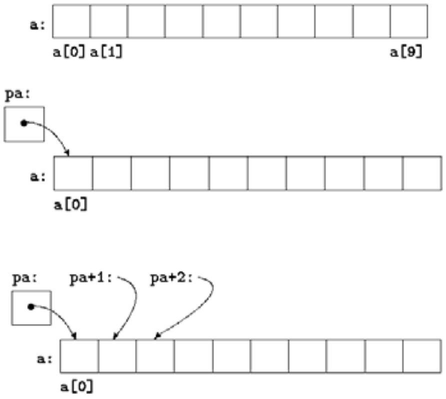
- Identifier of an array is equivalent to the address of its first element.

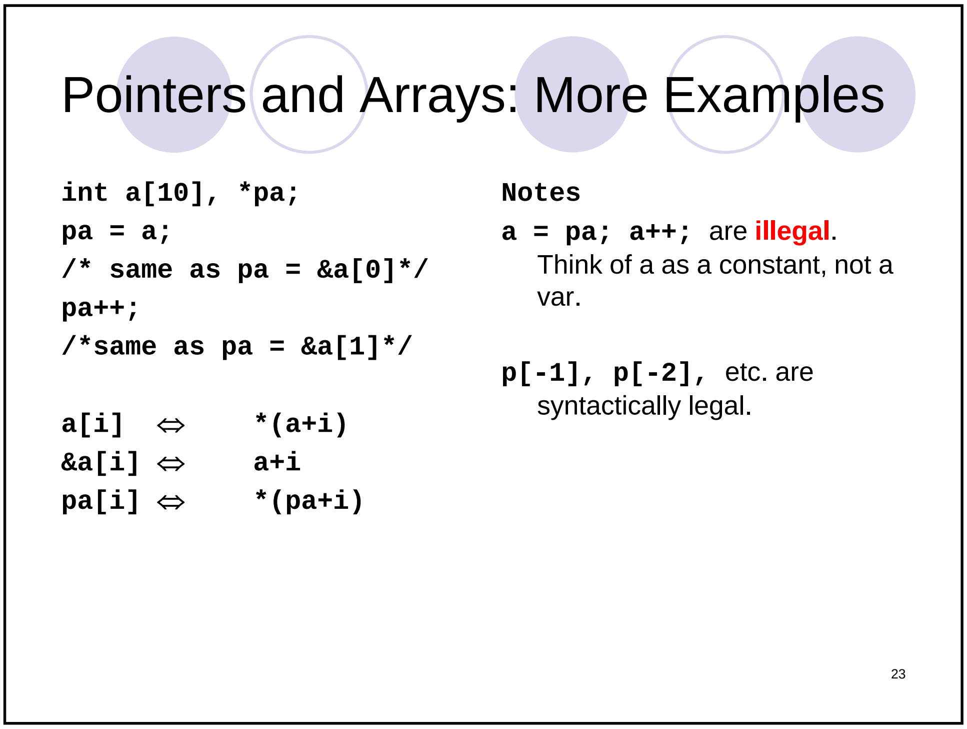
```
int numbers[20];
int * p;

p = numbers      // Valid
numbers = p      // Invalid
```

- **p** and **numbers** are equivalent and they have the same properties.
- Only difference is that we could assign another value to the pointer **p** whereas **numbers** will always point to the first of the 20 integer numbers of type int.

Pointers and Arrays: Example

```
int a[10];

int *pa;
pa = &a[0]
x = *pa;
/*same as x = a[0]*/

int y, z;
y = *(pa + 1);
z = *(pa + 2);
```





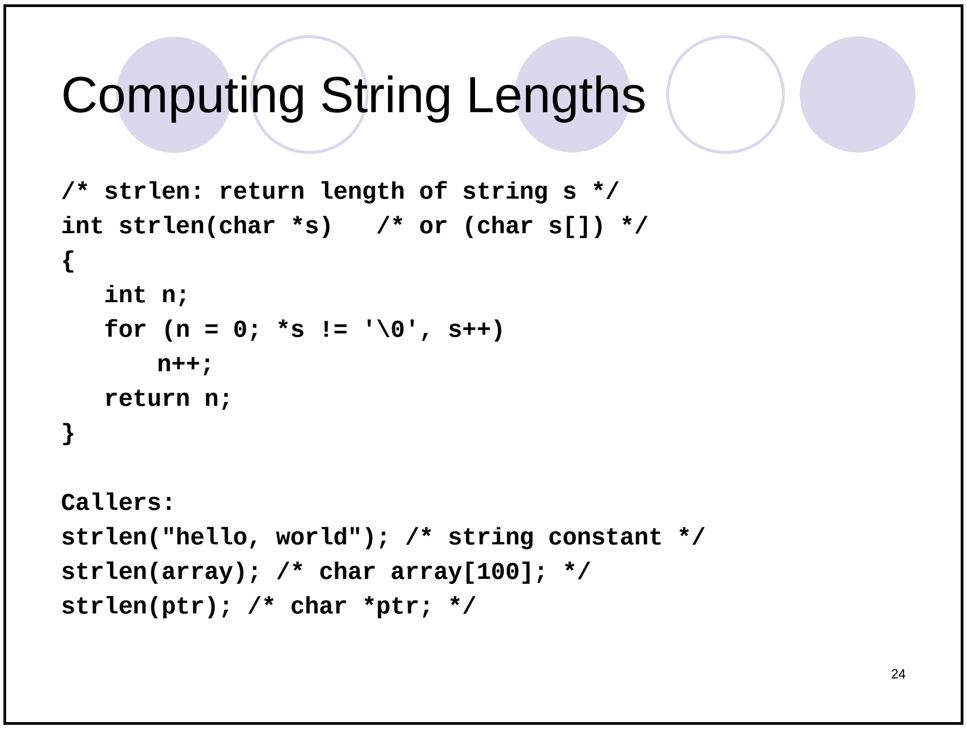
Pointers and Arrays: More Examples

```
int a[10], *pa;
pa = a;
/* same as pa = &a[0]*/
pa++;
/*same as pa = &a[1]*/

a[i]  ⇔    *(a+i)
&a[i] ⇔    a+i
pa[i] ⇔    *(pa+i)
```

Notes

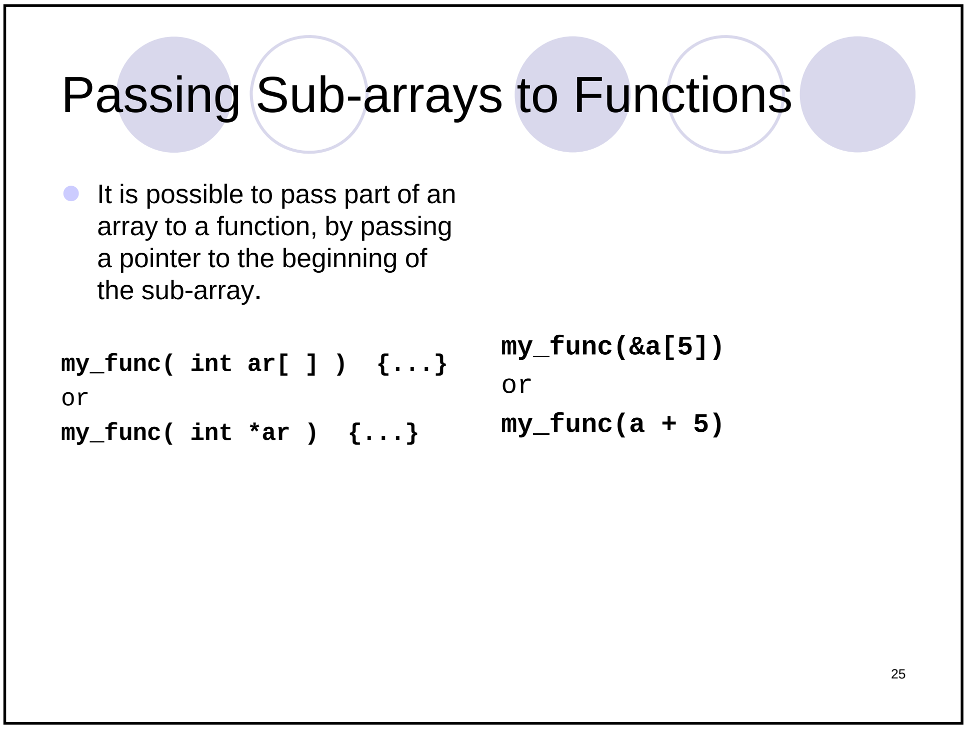
`a = pa; a++;` are **illegal**.
Think of a as a constant, not a var.

`p[-1], p[-2],` etc. are syntactically legal.



Computing String Lengths

```
/* strlen: return length of string s */
int strlen(char *s)   /* or (char s[]) */
{
   int n;
   for (n = 0; *s != '\0', s++)
       n++;
   return n;
}

Callers:
strlen("hello, world"); /* string constant */
strlen(array); /* char array[100]; */
strlen(ptr); /* char *ptr; */
```

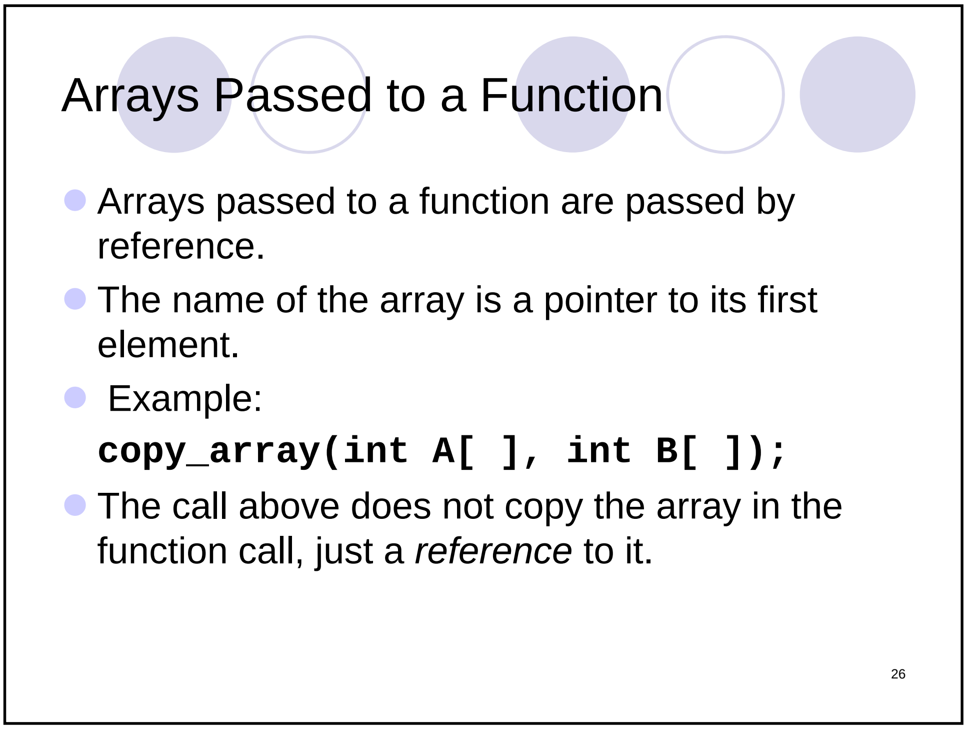


Passing Sub-arrays to Functions

- It is possible to pass part of an array to a function, by passing a pointer to the beginning of the sub-array.

```
my_func( int ar[ ] )  {...}
or
my_func( int *ar )  {...}
```

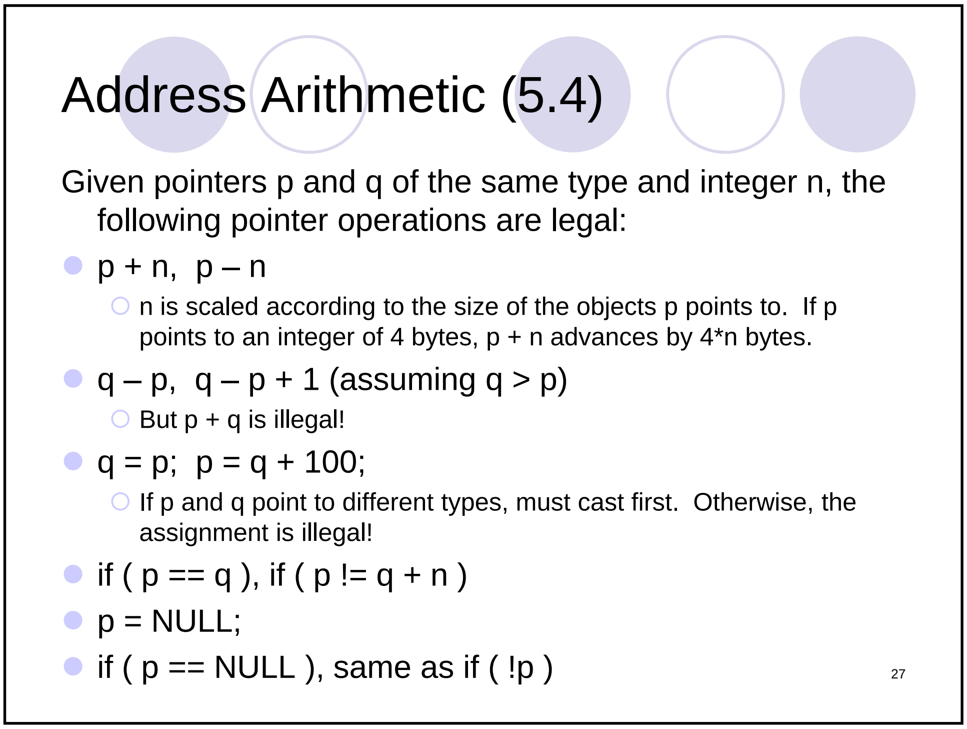
```
my_func(&a[5])
or
my_func(a + 5)
```



Arrays Passed to a Function

- Arrays passed to a function are passed by reference.
- The name of the array is a pointer to its first element.
- Example:

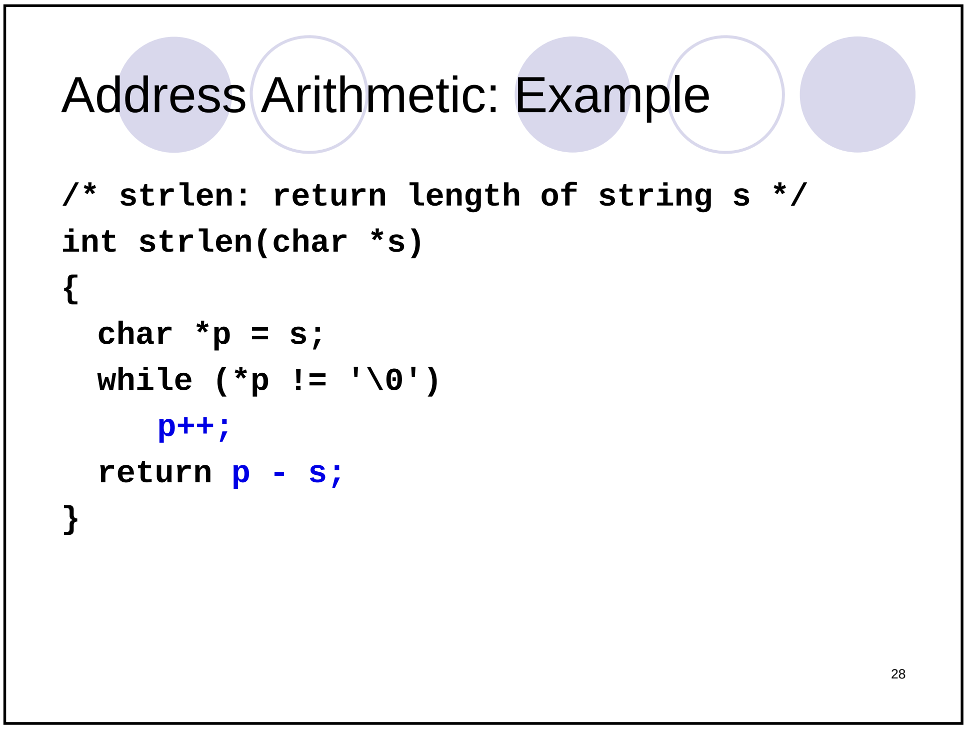
  ```
  copy_array(int A[ ], int B[ ]);
  ```
- The call above does not copy the array in the function call, just a *reference* to it.



Address Arithmetic (5.4)

Given pointers p and q of the same type and integer n, the following pointer operations are legal:

- p + n, p – n
  - n is scaled according to the size of the objects p points to. If p points to an integer of 4 bytes, p + n advances by 4*n bytes.
- q – p, q – p + 1 (assuming q > p)
  - But p + q is illegal!
- q = p; p = q + 100;
  - If p and q point to different types, must cast first. Otherwise, the assignment is illegal!
- if (p == q), if (p != q + n)
- p = NULL;
- if (p == NULL), same as if (!p)



Address Arithmetic: Example

```
/* strlen: return length of string s */
int strlen(char *s)
{
  char *p = s;
  while (*p != '\0')
     p++;
  return p - s;
}
```

Address Arithmetic: Summary

- Legal:
  - assignment of pointers of the same type
  - adding or subtracting a pointer and an integer
  - subtracting or comparing two pointers to members of the same array
  - assigning or comparing to zero (NULL)
- Illegal:
  - add two pointers
  - multiply or divide or shift or mask pointer variables
  - add float or double to pointers
  - assign a pointer of one type to a pointer of another type (except for void *) without a cast

Character Pointers and Functions (5.5)

- A *string constant* (“hello world”) is an array of characters.
- The array is terminated with the null character '\0' so that programs can find the end.

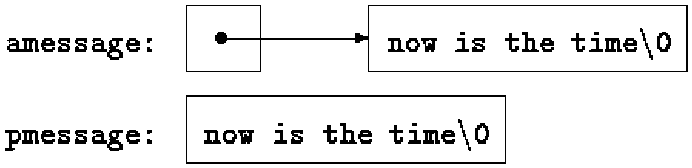
```
char *pmessage;
pmessage = "now is the time";
```

  - assigns to pmessage a pointer to the character array. This is *not a string copy; only pointers are involved.*
  - C does not provide any operators for processing an entire string of characters as a unit.

Important Difference between ...

```
char amessage[] = "now is the time"; /* an array */
char *pmessage = "now is the time"; /* a pointer */
```

- amessage will always refer to the same storage.
- pmessage may later be modified to point elsewhere.



Example: String Copy Function

```
/* strcpy: copy t to s; array
   subscript version */
void strcpy(char *s, char *t)
{
  int i;
  i = 0;
  while ((s[i] = t[i]) != '\0')
    i++;
}
```

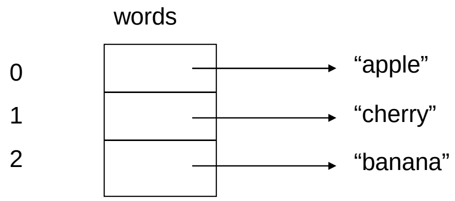
```
/* strcpy: copy t to s; pointer
   version */
void strcpy(char *s, char *t)
{
  int i;
  i = 0;
  while ((*s = *t) != '\0') {
    s++; t++;
  }
}
```

```
/* strcpy: copy t to s; pointer
   version 2 */
void strcpy(char *s, char *t)
{
while ((*s++ = *t++) != '\0') ;
}
```

Arrays of Pointers (5.6)

```
char *words[ ]={“apple”, “cherry”, “banana”};
```

- **`words`** is an array of pointers to **`char`**.
- Each element of **`words`** (**`words[0], words[1], words[2]`**) is a pointer to **`char`**.



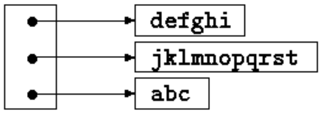
words

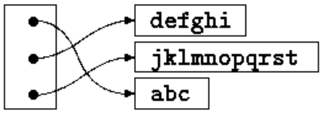
	words	
0		“apple”
1		“cherry”
2		“banana”

Arrays vs. Pointers

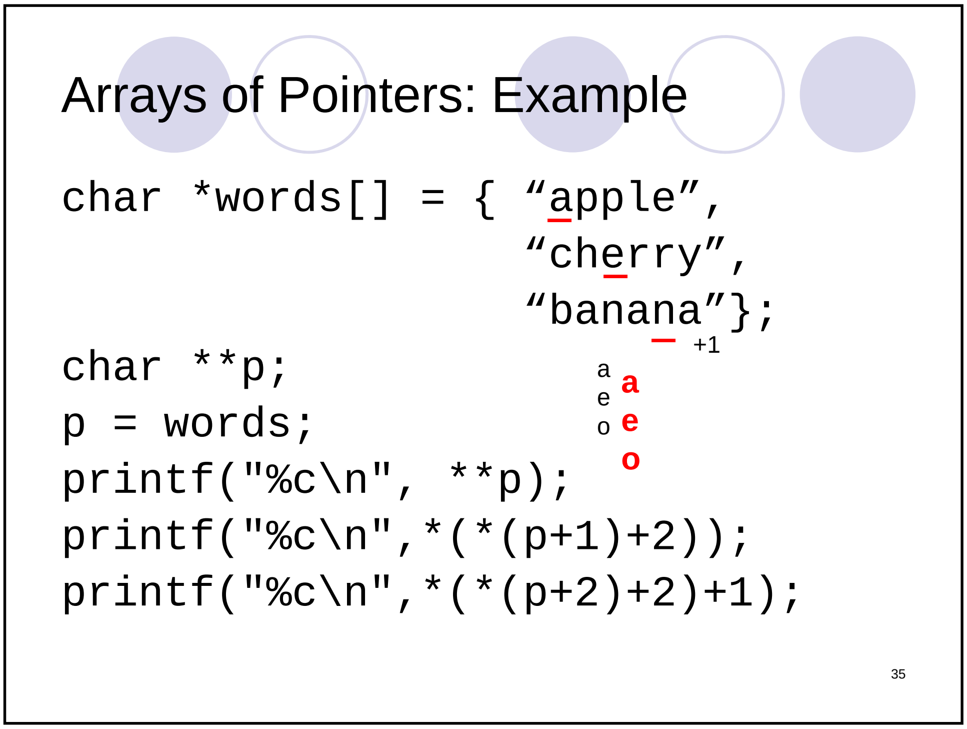
What is the difference between the previous example and the following?

```
char words[][10] = { “apple”,
                     “cherry”,
                     “banana”};
```





Previous example



Arrays of Pointers: Example

```
char *words[] = { “apple”,
                  “cherry”,
                  “banana”};
char **p;
p = words;
printf("%c\n", **p);
printf("%c\n",*(*(p+1)+2));
printf("%c\n",*(*(p+2)+2)+1);
```

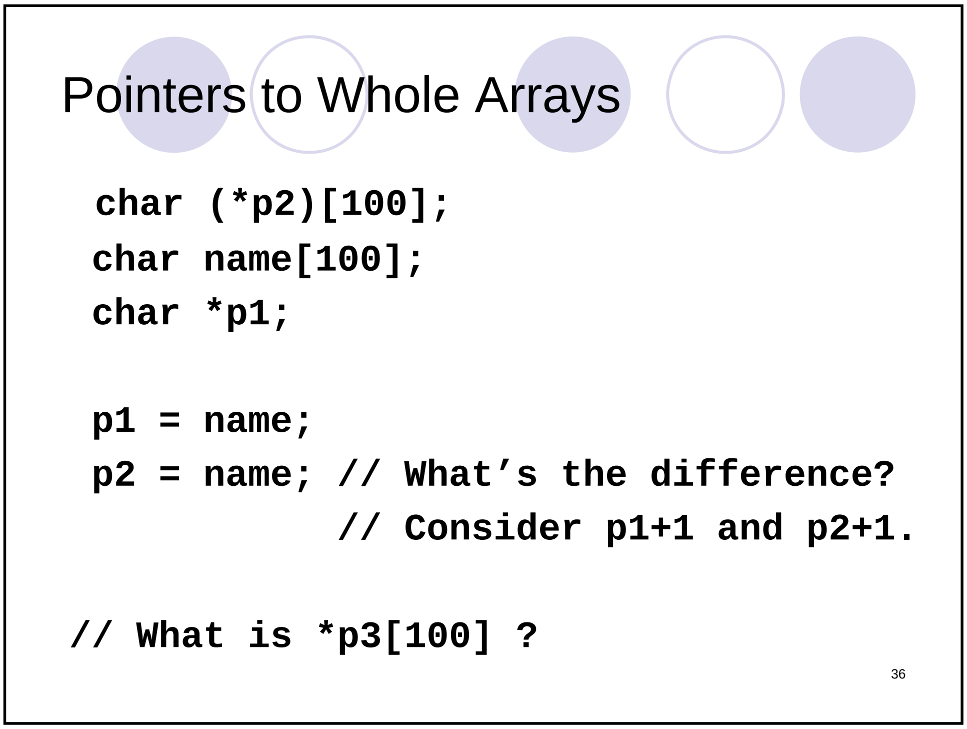
+1

a a

e e

o o

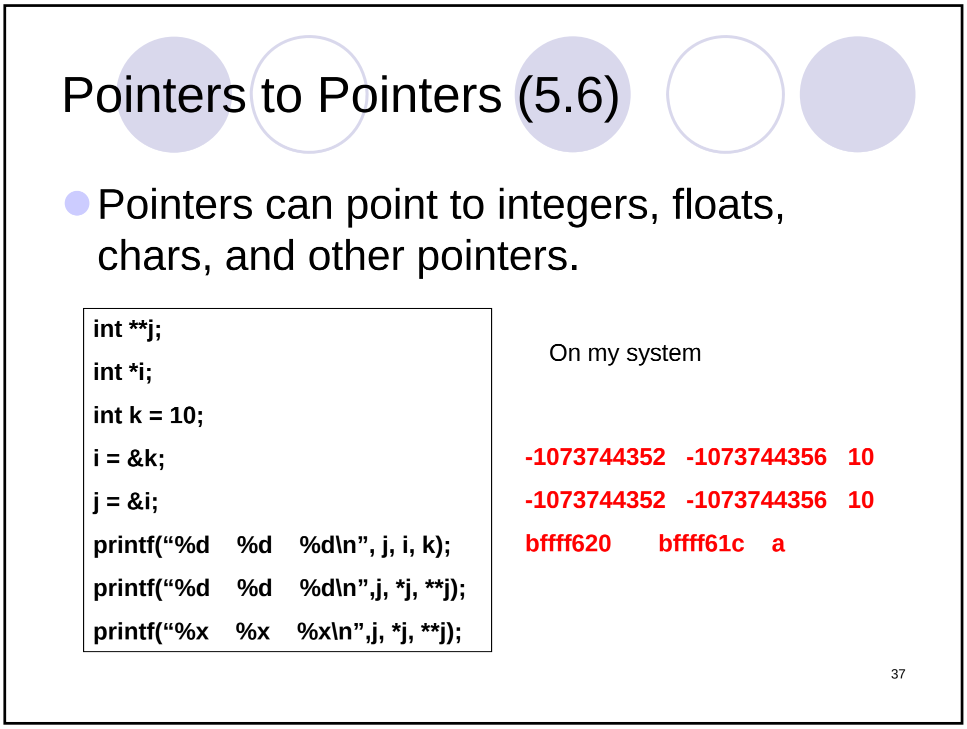
35



Pointers to Whole Arrays

```
 char (*p2)[100];
char name[100];
char *p1;

p1 = name;
p2 = name; // What’s the difference?
           // Consider p1+1 and p2+1.

// What is *p3[100] ?
```

36



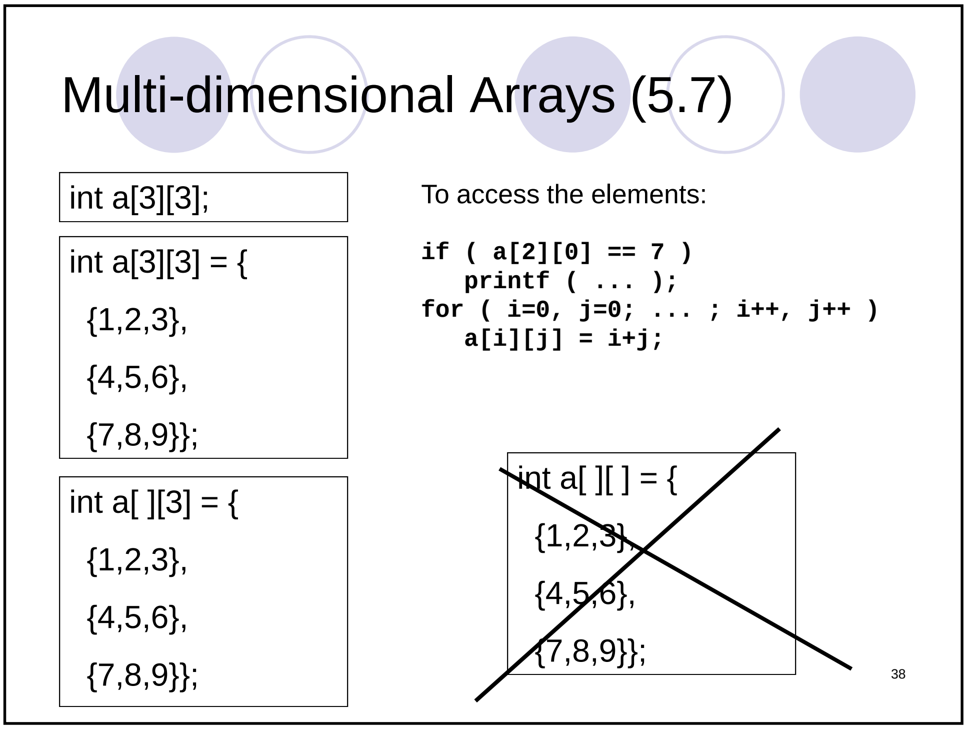
Pointers to Pointers (5.6)

- Pointers can point to integers, floats, chars, and other pointers.

```
int **j;
int *i;
int k = 10;
i = &k;
j = &i;
printf("%d   %d   %d\n", j, i, k);
printf("%d   %d   %d\n",j, *j, **j);
printf("%x   %x   %x\n",j, *j, **j);
```

On my system

```
-1073744352  -1073744356  10
-1073744352  -1073744356  10
bffff620     bffff61c     a
```



Multi-dimensional Arrays (5.7)

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

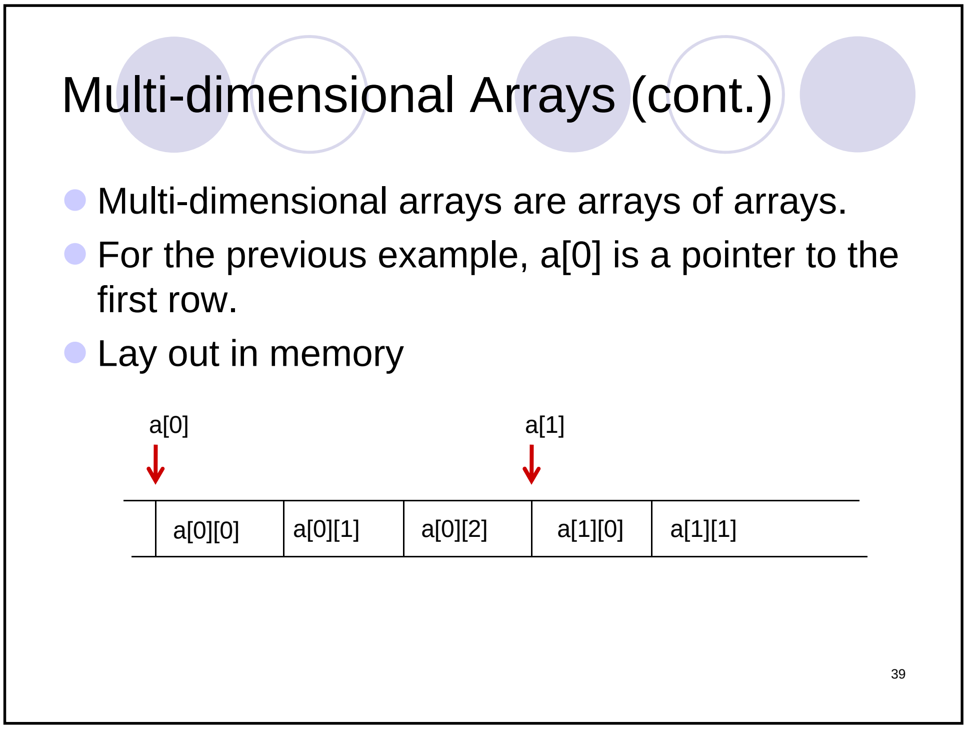
```
int a[3][3] = {
  {1,2,3},
  {4,5,6},
  {7,8,9}};
```

```
int a[ ][3] = {
  {1,2,3},
  {4,5,6},
  {7,8,9}};
```

To access the elements:

```
if ( a[2][0] == 7 )
   printf ( ... );
for ( i=0, j=0; ... ; i++, j++ )
   a[i][j] = i+j;
```

~~int a[ ][ ] = {~~
~~{1,2,3},~~
~~{4,5,6},~~
~~{7,8,9}};~~

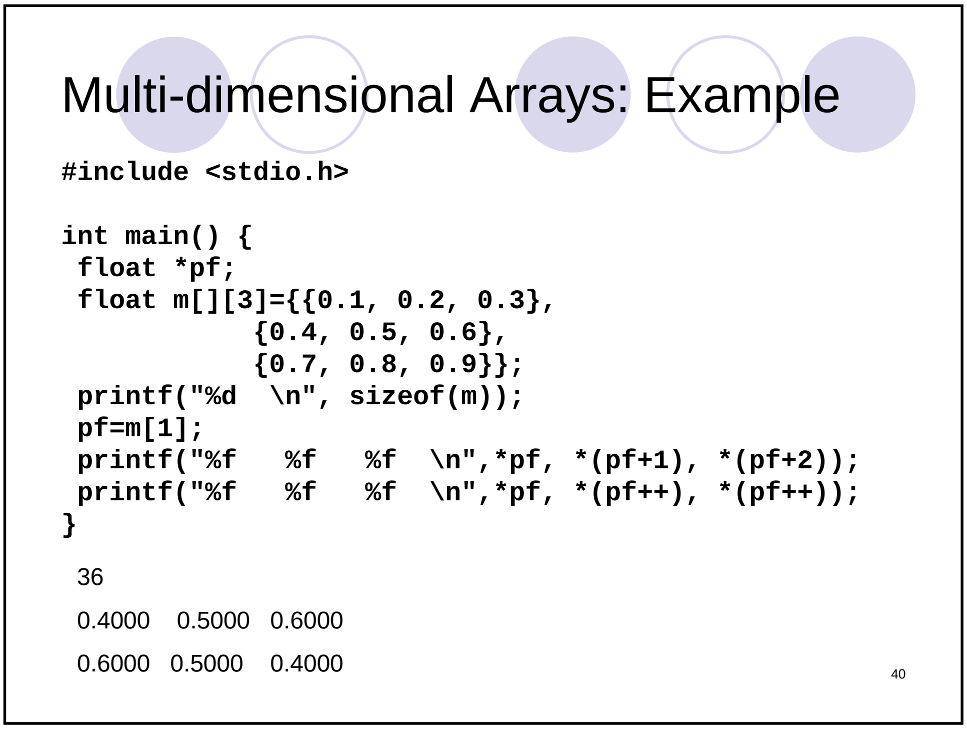


Multi-dimensional Arrays (cont.)

- Multi-dimensional arrays are arrays of arrays.
- For the previous example, a[0] is a pointer to the first row.
- Lay out in memory

a[0] ↓ a[1] ↓

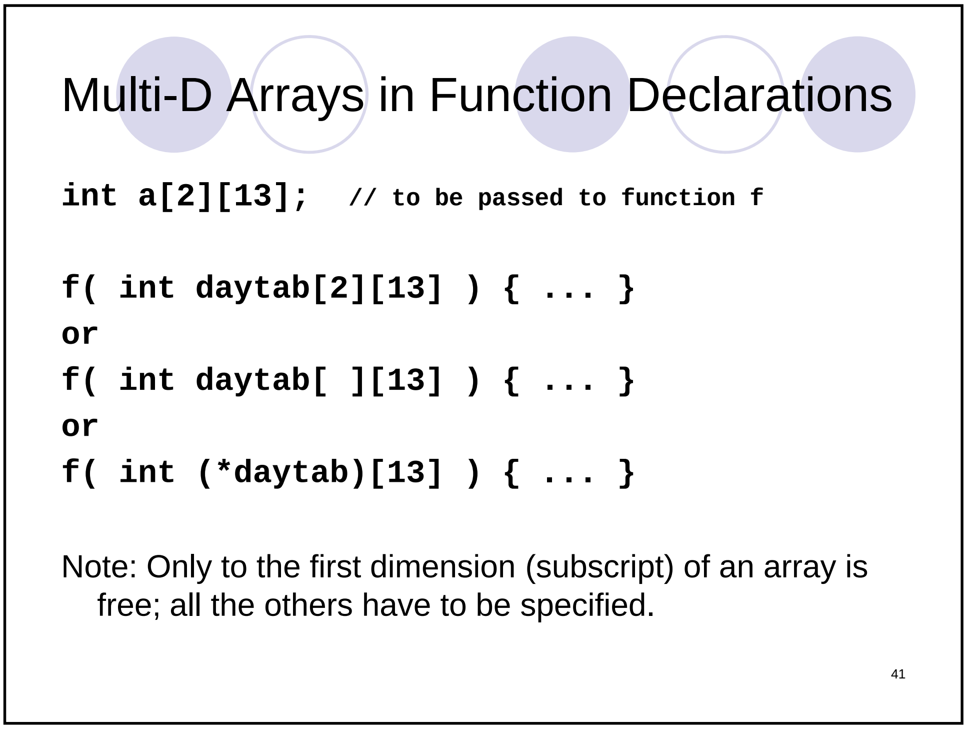
a[0][0]	a[0][1]	a[0][2]	a[1][0]	a[1][1]



Multi-dimensional Arrays: Example

```
#include <stdio.h>

int main() {
 float *pf;
 float m[][3]={{0.1, 0.2, 0.3},
           {0.4, 0.5, 0.6},
           {0.7, 0.8, 0.9}};
 printf("%d  \n", sizeof(m));
 pf=m[1];
 printf("%f   %f   %f  \n",*pf, *(pf+1), *(pf+2));
 printf("%f   %f   %f  \n",*pf, *(pf++), *(pf++));
}
```

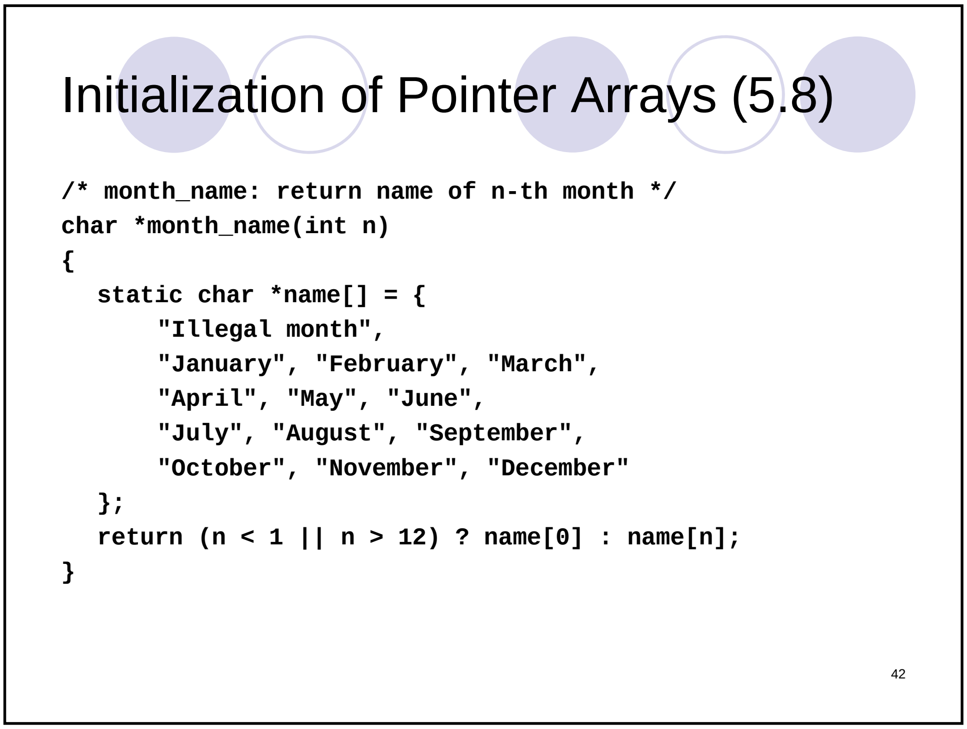
```
36
0.4000   0.5000  0.6000
0.6000  0.5000   0.4000
```



Multi-D Arrays in Function Declarations

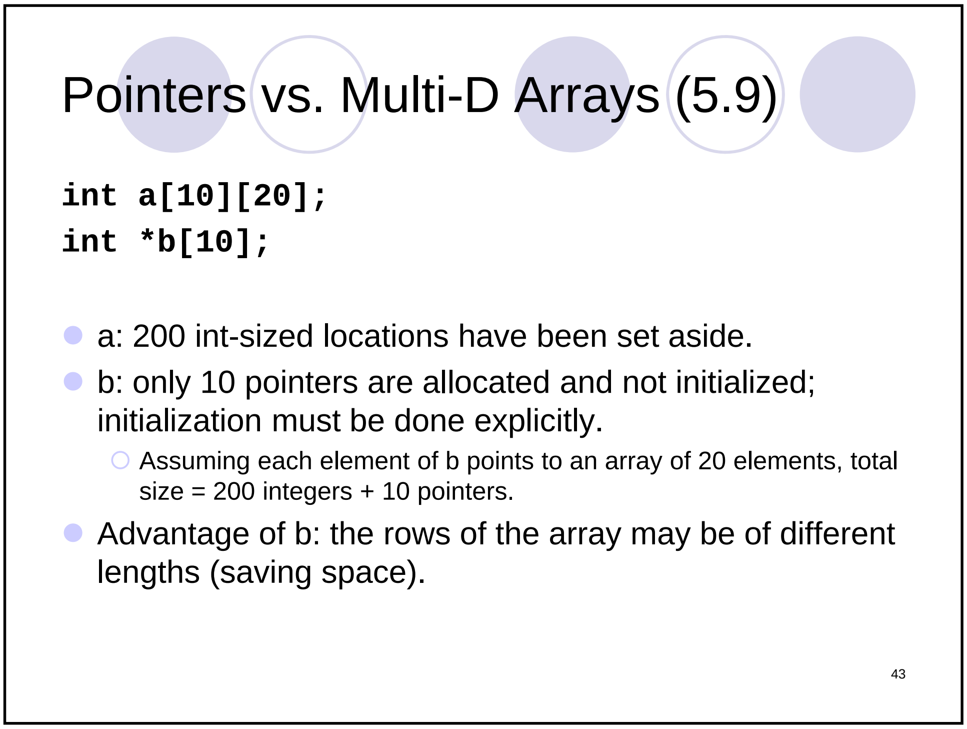
```
int a[2][13];  // to be passed to function f

f( int daytab[2][13] ) { ... }
or
f( int daytab[ ][13] ) { ... }
or
f( int (*daytab)[13] ) { ... }
```

Note: Only to the first dimension (subscript) of an array is free; all the others have to be specified.



Initialization of Pointer Arrays (5.8)

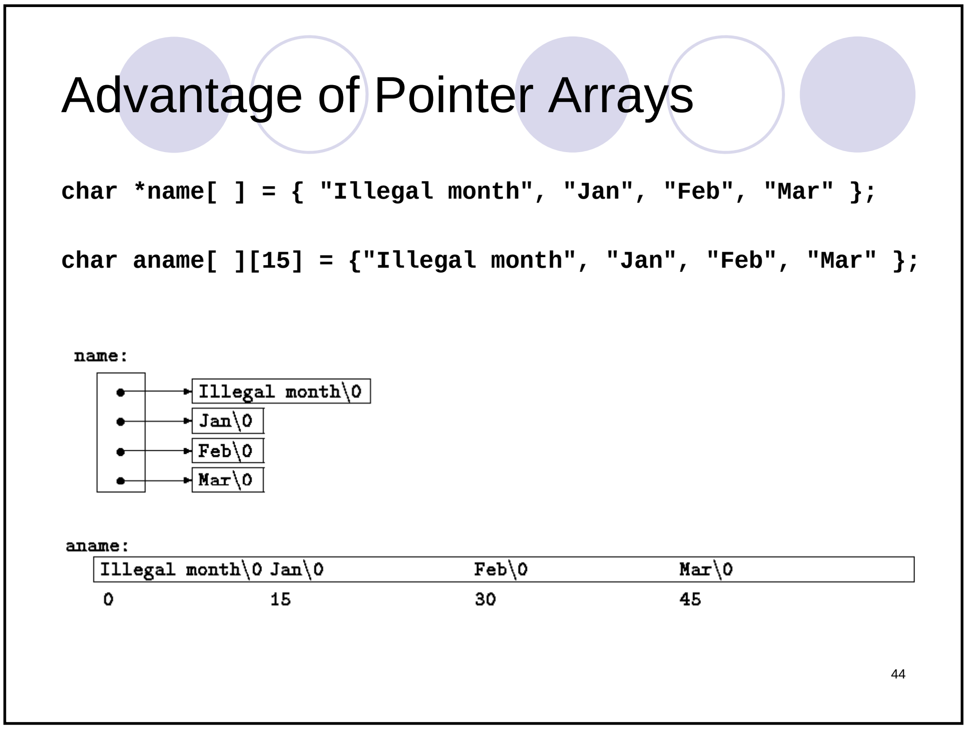
```
/* month_name: return name of n-th month */
char *month_name(int n)
{
  static char *name[] = {
      "Illegal month",
      "January", "February", "March",
      "April", "May", "June",
      "July", "August", "September",
      "October", "November", "December"
  };
  return (n < 1 || n > 12) ? name[0] : name[n];
}
```



Pointers vs. Multi-D Arrays (5.9)

```
int a[10][20];
int *b[10];
```

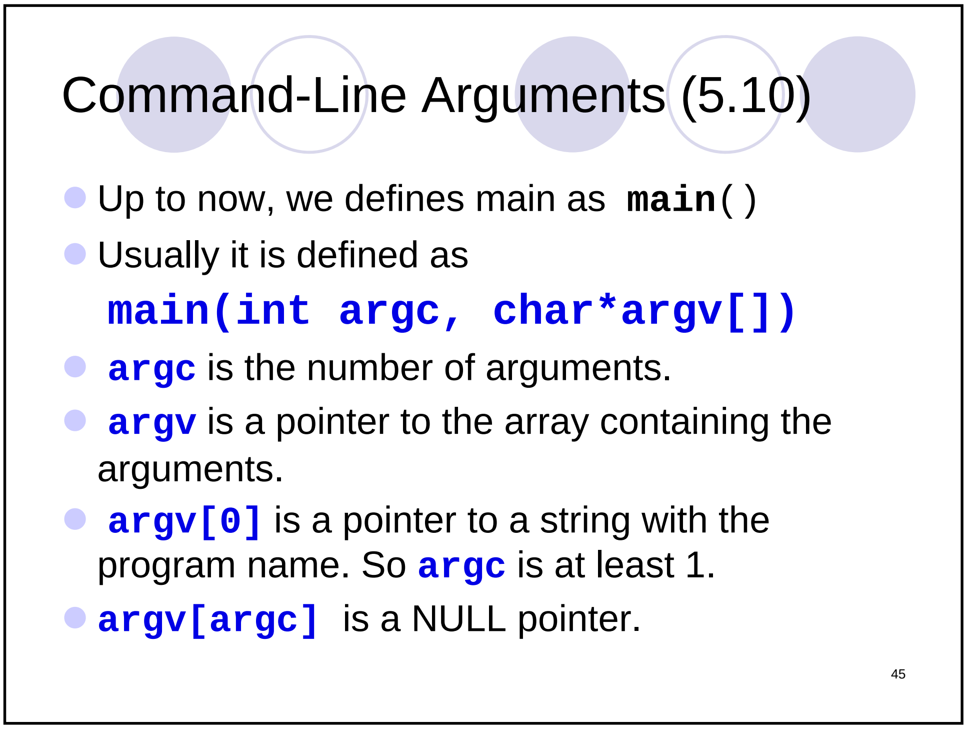
- a: 200 int-sized locations have been set aside.
- b: only 10 pointers are allocated and not initialized; initialization must be done explicitly.
  - Assuming each element of b points to an array of 20 elements, total size = 200 integers + 10 pointers.
- Advantage of b: the rows of the array may be of different lengths (saving space).



Advantage of Pointer Arrays

```
char *name[ ] = { "Illegal month", "Jan", "Feb", "Mar" };

char aname[ ][15] = {"Illegal month", "Jan", "Feb", "Mar" };
```

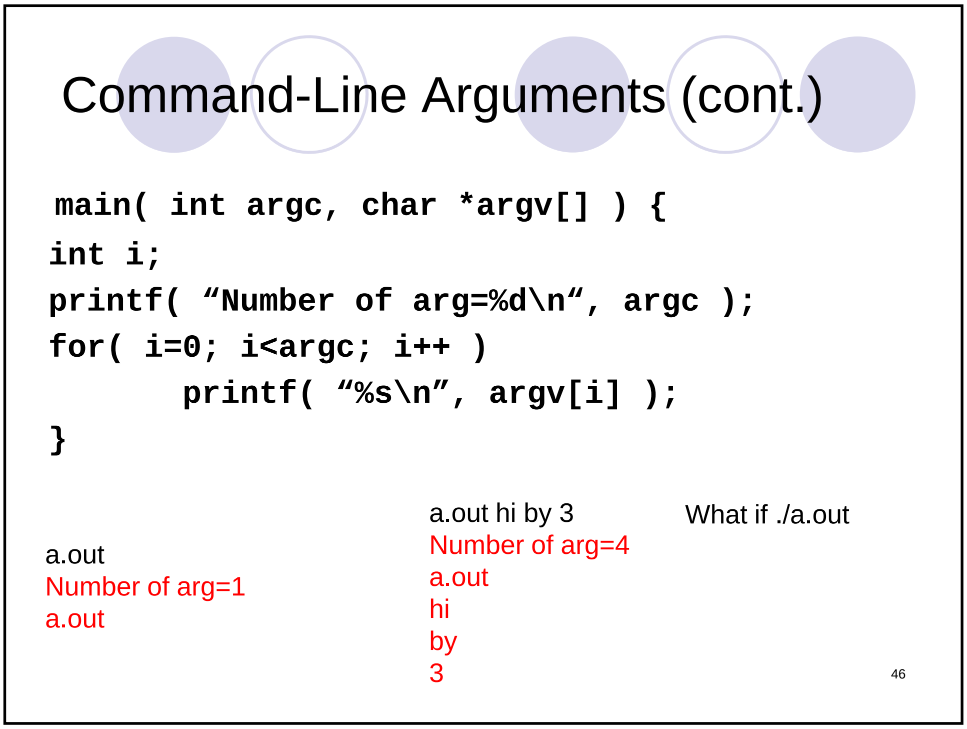
```
name:
  •──▶ Illegal month\0
  •──▶ Jan\0
  •──▶ Feb\0
  •──▶ Mar\0

aname:
 Illegal month\0 Jan\0          Feb\0          Mar\0
 0               15             30             45
```



Command-Line Arguments (5.10)

- Up to now, we defines main as **`main()`**
- Usually it is defined as

  `main(int argc, char*argv[])`
- **`argc`** is the number of arguments.
- **`argv`** is a pointer to the array containing the arguments.
- **`argv[0]`** is a pointer to a string with the program name. So **`argc`** is at least 1.
- **`argv[argc]`** is a NULL pointer.



Command-Line Arguments (cont.)

```
 main( int argc, char *argv[] ) {
int i;
printf( “Number of arg=%d\n“, argc );
for( i=0; i<argc; i++ )
       printf( “%s\n”, argv[i] );
}
```

```
a.out
Number of arg=1
a.out
```

```
a.out hi by 3
Number of arg=4
a.out
hi
by
3
```

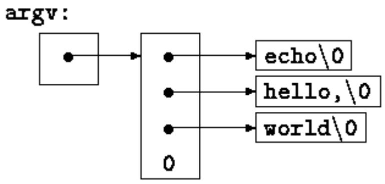
What if ./a.out

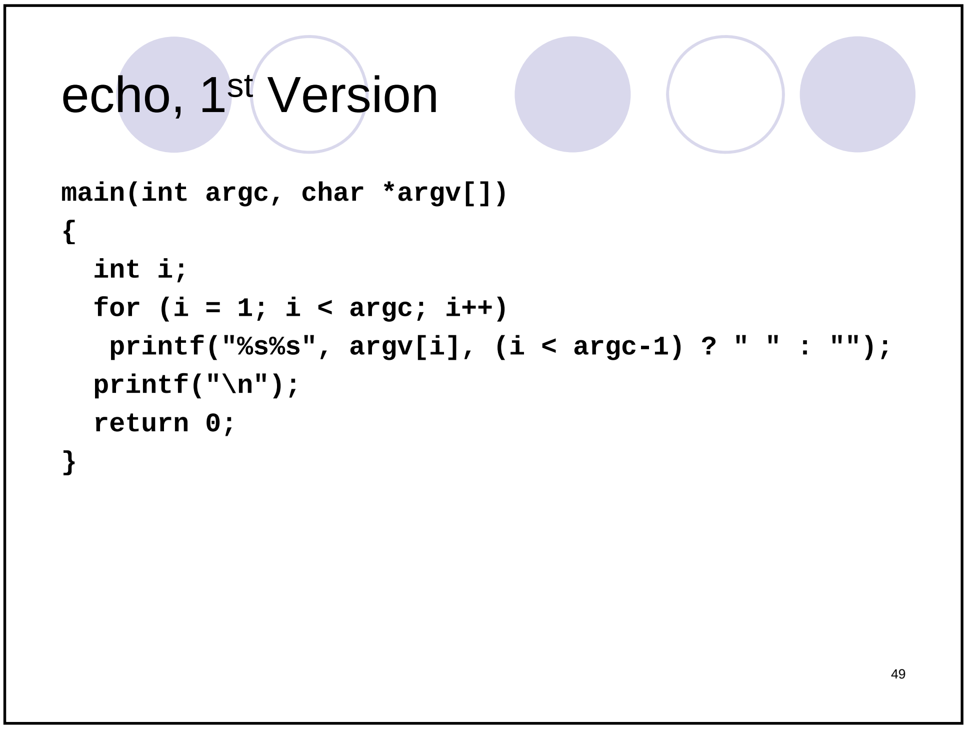
Example

- Write a program name echo (echo.c) which echoes its command-line arguments on a single line, separated by blanks.
- Command: **`echo hello, world`**
- Output: **`hello, world`**

Example: Diagram

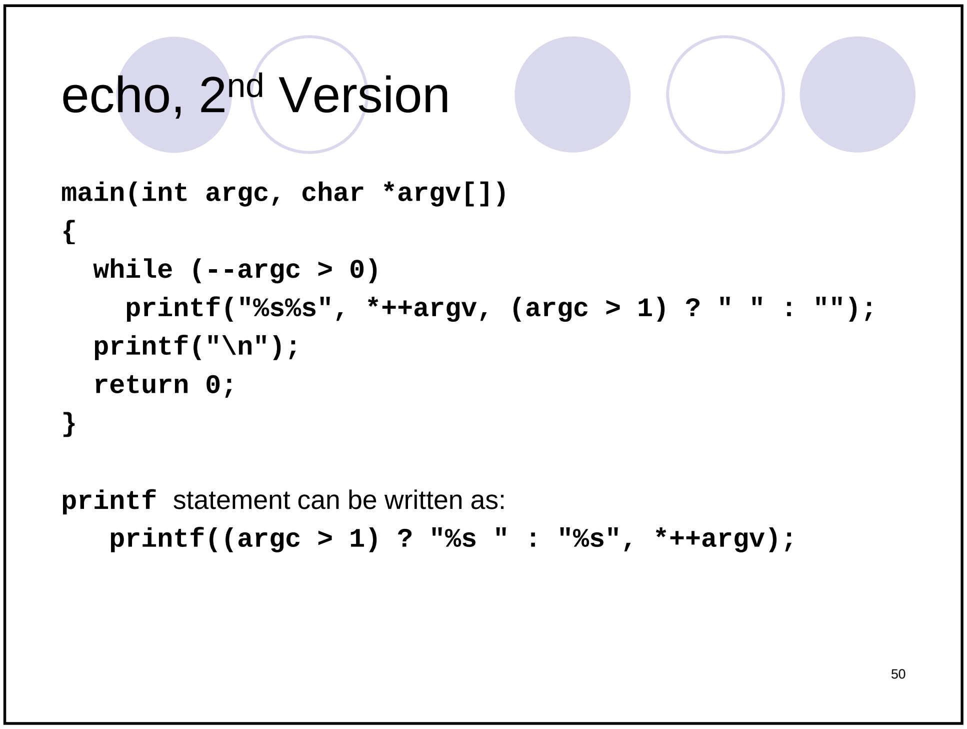
- Write a program name echo (echo.c) which echoes its command-line arguments on a single line, separated by blanks.
- Command: **`echo hello, world`**
- Output: **`hello, world`**





echo, 1st Version

```
main(int argc, char *argv[])
{
  int i;
  for (i = 1; i < argc; i++)
   printf("%s%s", argv[i], (i < argc-1) ? " " : "");
  printf("\n");
  return 0;
}
```

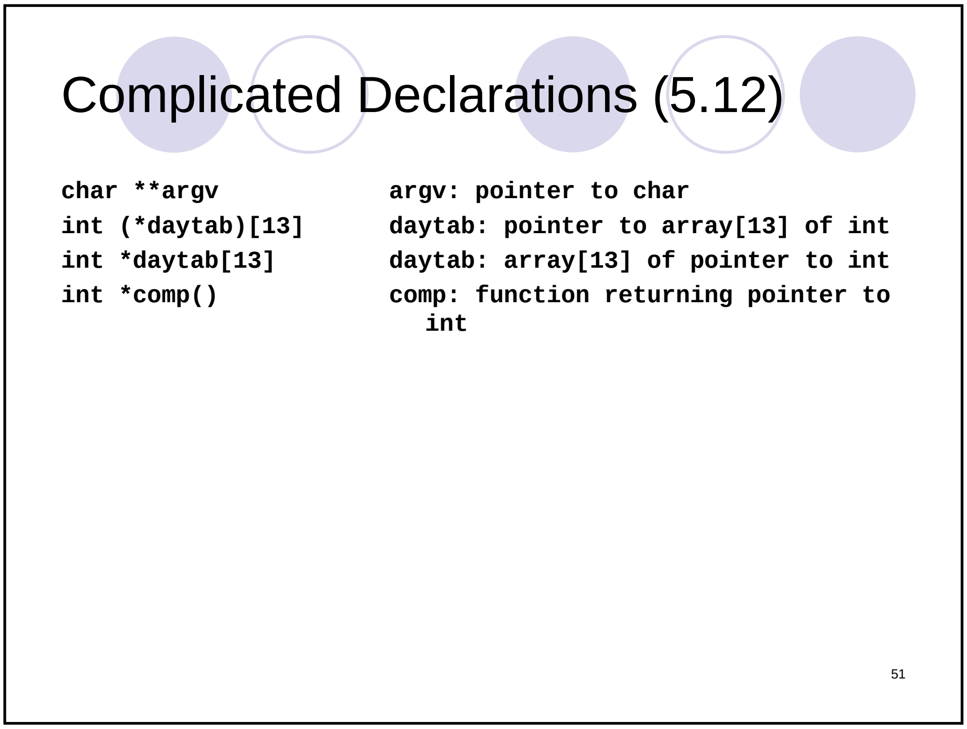


echo, 2nd Version

```
main(int argc, char *argv[])
{
  while (--argc > 0)
    printf("%s%s", *++argv, (argc > 1) ? " " : "");
  printf("\n");
  return 0;
}
```

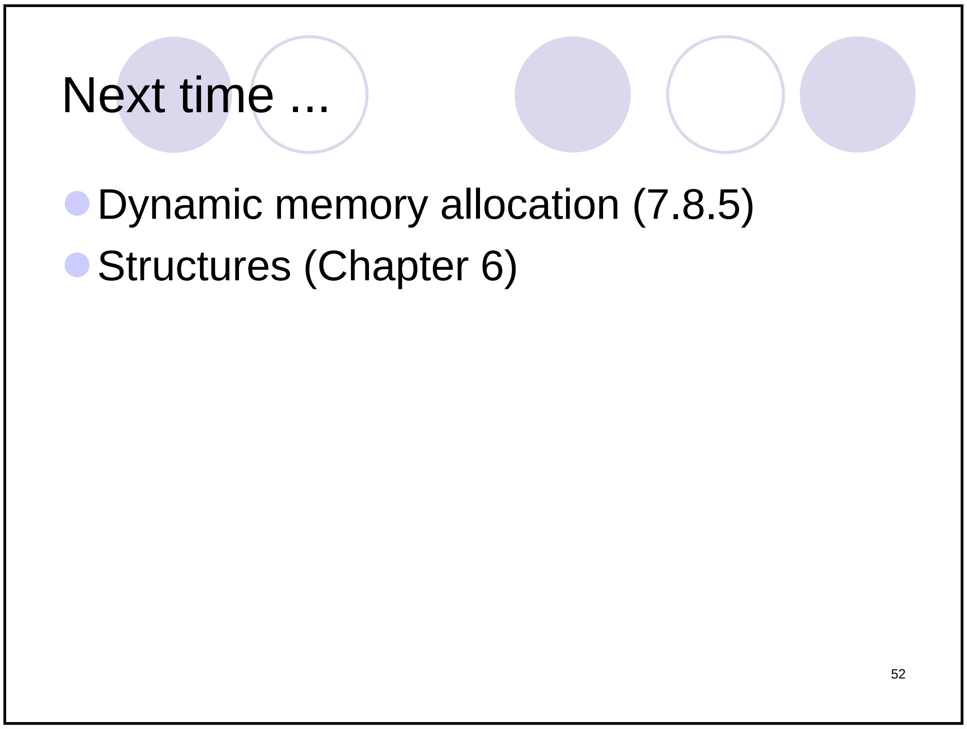
`printf` statement can be written as:

```
   printf((argc > 1) ? "%s " : "%s", *++argv);
```



Complicated Declarations (5.12)

`char **argv`	`argv: pointer to char`
`int (*daytab)[13]`	`daytab: pointer to array[13] of int`
`int *daytab[13]`	`daytab: array[13] of pointer to int`
`int *comp()`	`comp: function returning pointer to int`



Next time ...

- Dynamic memory allocation (7.8.5)
- Structures (Chapter 6)