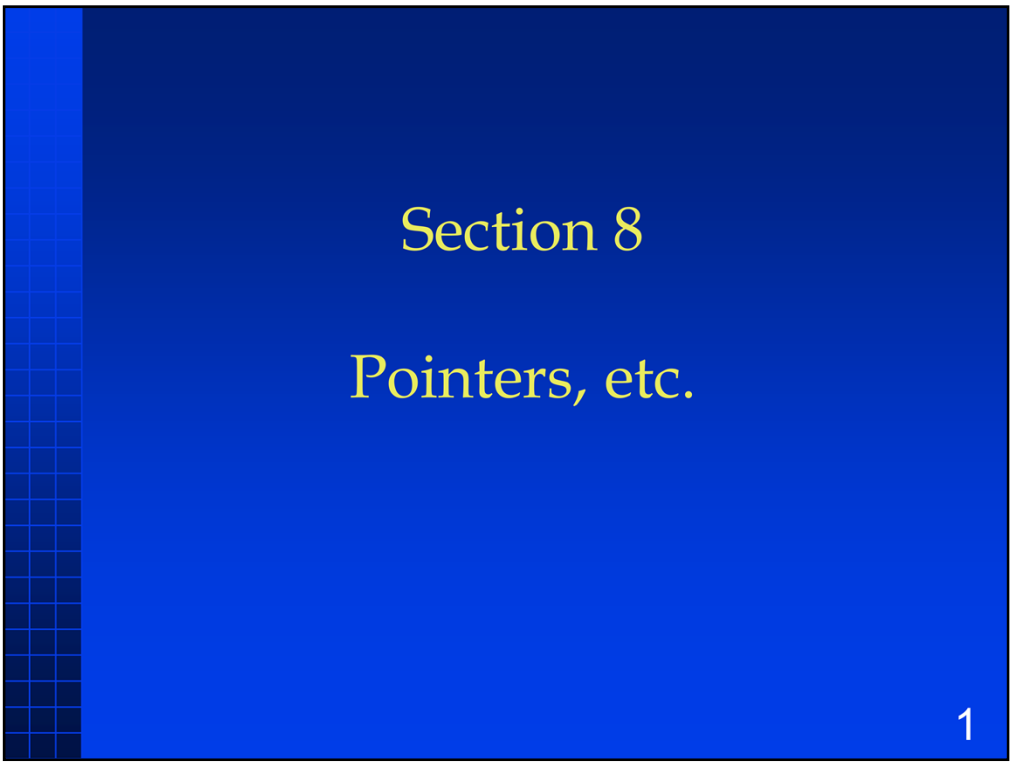


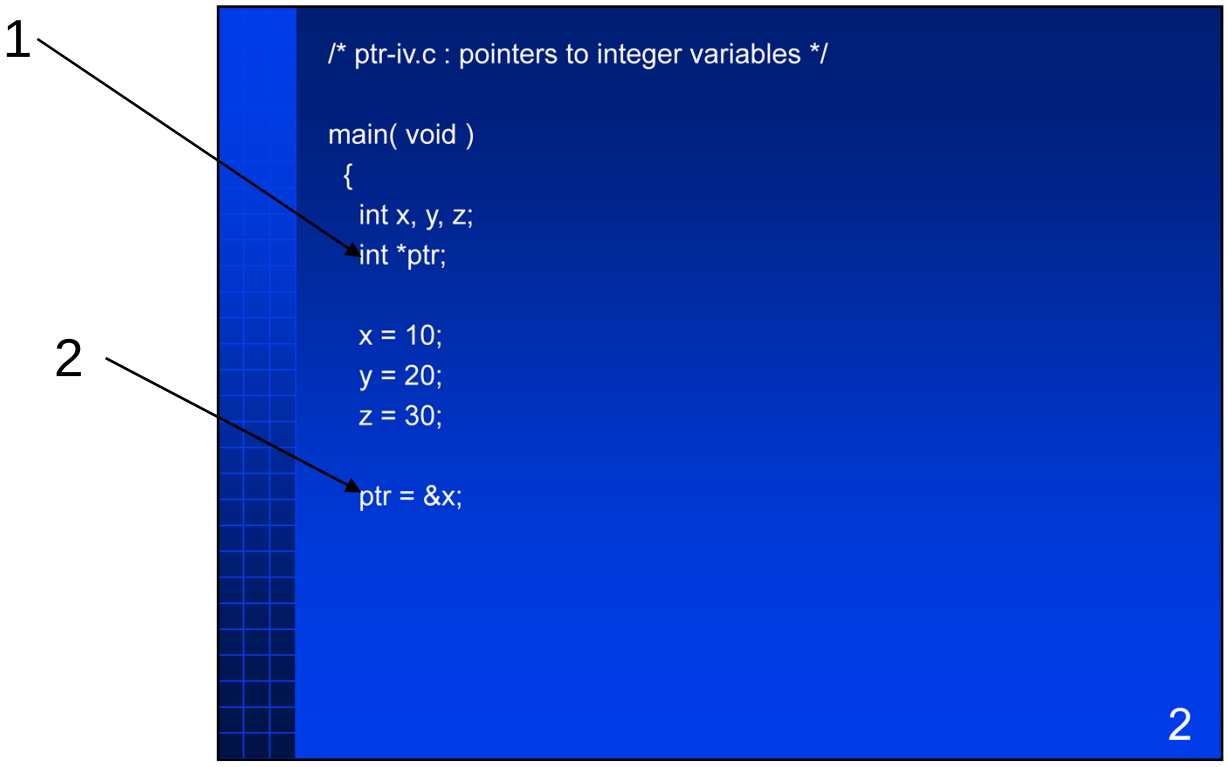
significant part of C's expressive power comes from ability to manipulate pointers. aka references, addresses.

consider a variable. sometimes we're not interested in the contents of the variable, we want the variable itself (box and contents analogy)

useful for efficiency, eg arrays, don't want to transmit entire set of values

also useful for accessing underlying hardware: computer memory is big array of integers or bytes/characters

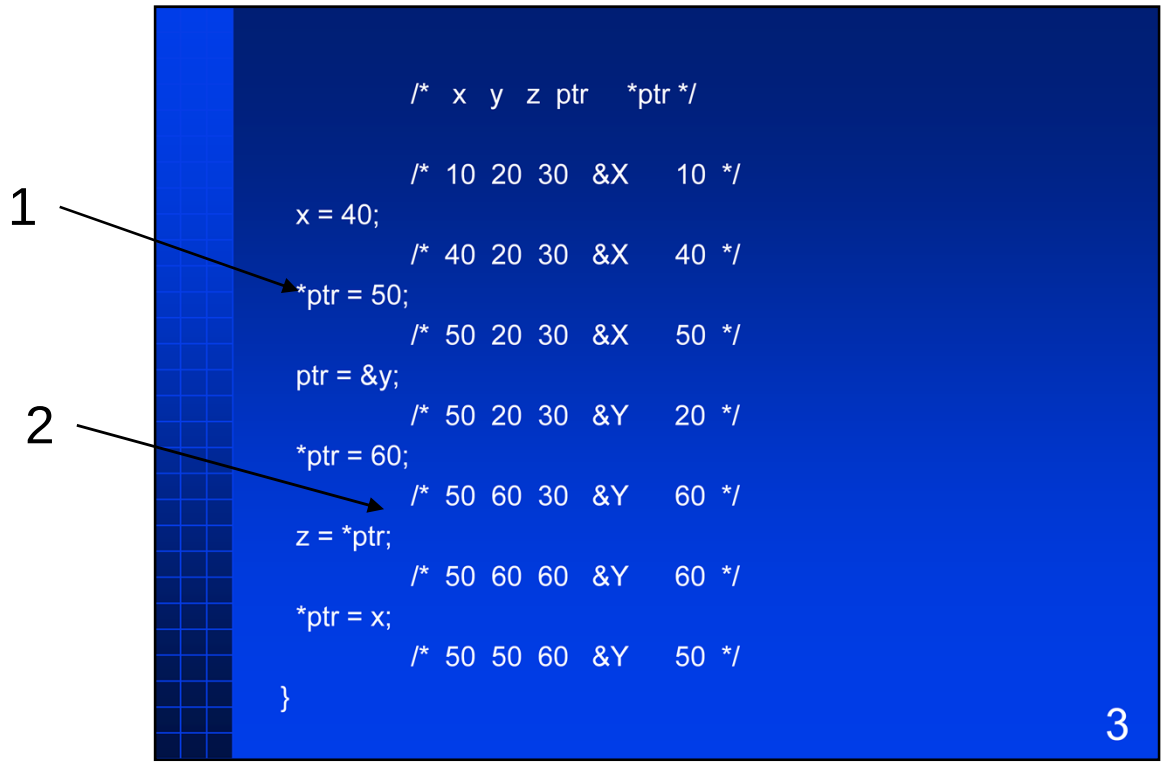
later, will see equivalence of pointers and arrays



1) ptr is a variable that contains a reference to an integer. ptr is not an int, it is a pointer to an int.

2) ptr is assigned a reference to the variable x. ptr contains the address of x.

unary & is the “address of” operator



explain slide:

comments show contents of variables. ignore last column for a second

starts out as...

assign 40 to x

1) assign 50 to wherever ptr points; assign 50 to the variable whose address is contained in ptr. opposite of & (taking reference) -- call dereferencing. aka indirection.

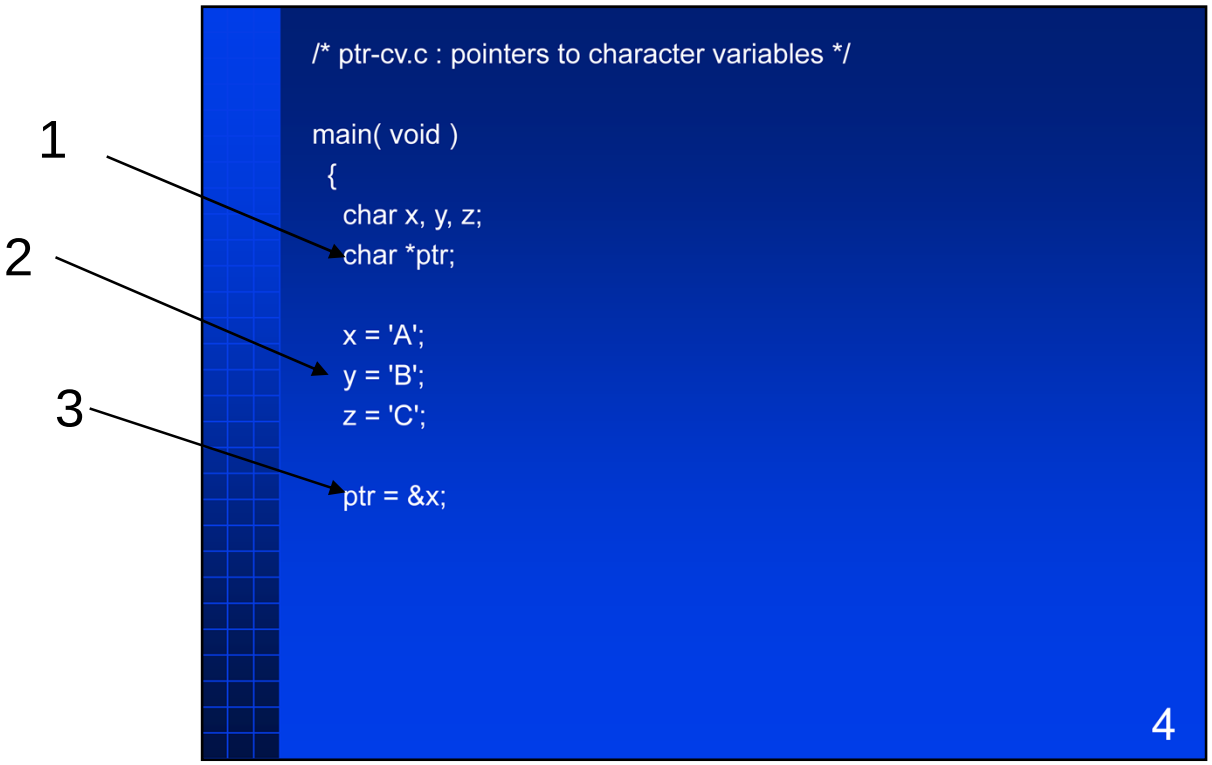
in this case, dereference as a target of assignment,

last column shows current dereference of ptr. go back and review.

[explain each line]

...

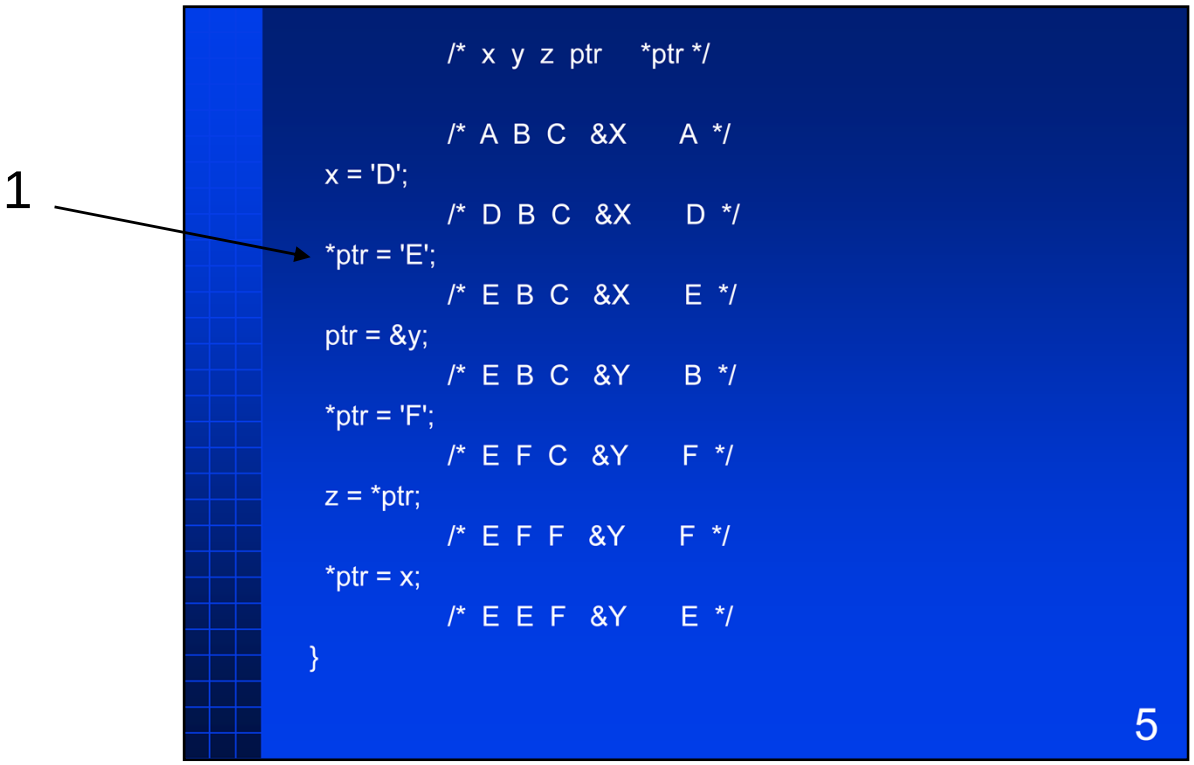
2) dereference as an expression



1) ptr is a pointer to a single character

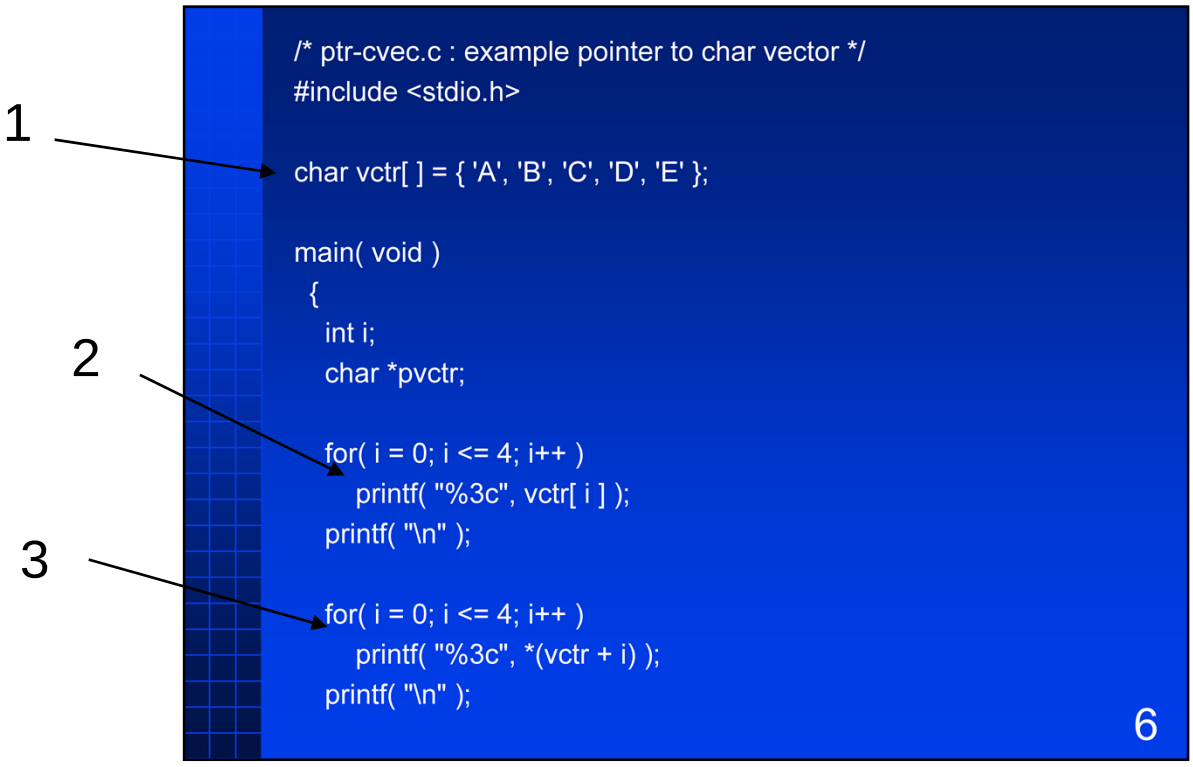
2) single-char values

3) ptr is assigned the address of x. note that this is the same expression as assigning an integer address. & yields a pointer in any case, the definition specify precisely what it points to



same structure as before, comment columns show values

1) dereferencing a pointer, same as before. since its a ptr to char, we assign a char.



have seen that array-names are already a reference, don't need & to get address of array. what would be an appropriate thing to store the address of an array? a pointer -- pointer to what? pointer to same thing as the basetype of the array.

Eg: arrayname is pointer to first element, ignore the rest [chalkboard]

So arrayname is reference is pointer to basetype. what is subscripting?

start at first element and move along. can do this with [], can also do with pointer arithmetic [chalkboard: +1 to char*]

So subscripting is equivalent to pointer arithmetic.

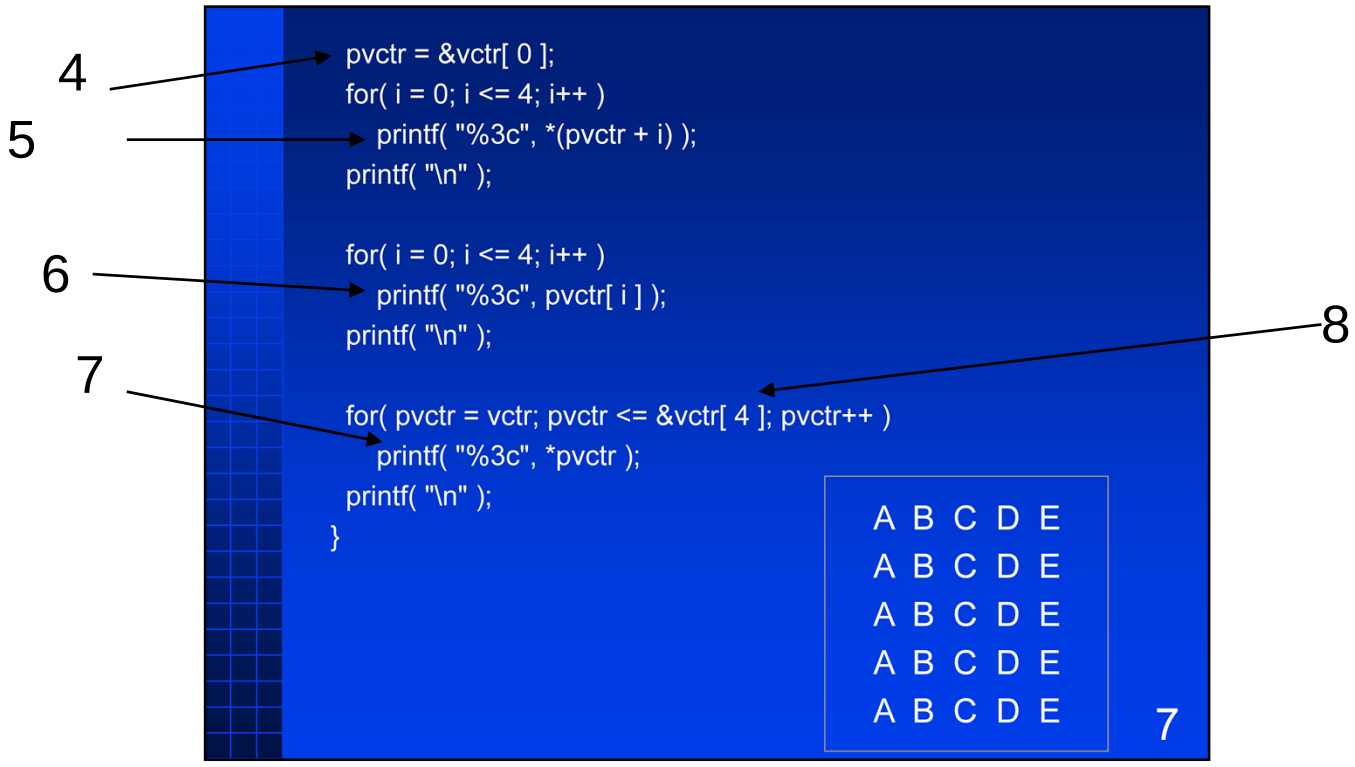
These concepts are fundamental to C: can use pointer variables and array-names interchangeably

can subscript pointers, can do pointer dereferencing to array-names.

1) array of characters, not a "string" (no nullchar). all strings are array of char, but not all array of char are string.

2) simple array subscripting, [] yields a single character

3) ptr arith on array-name (compute address and dereference)



4) pvctr gets address of first element of vctr

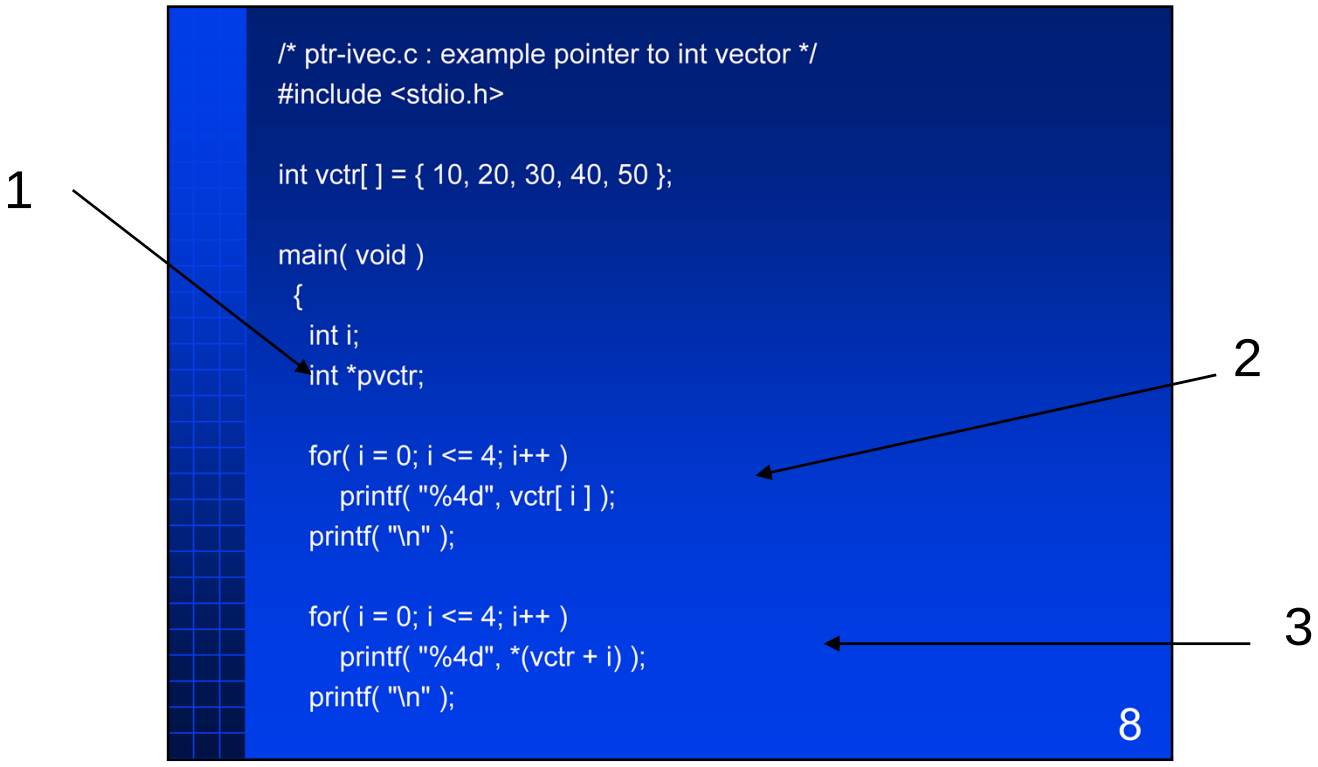
can also use pvctr = vctr since array name is already reference

5) pure pointer arithmetic. note same as array-name case

6) subscripting a pointer. the [] are really operators that are defined to work on things containing an address -- any address will do. in this case, operation is pretty much identical to preceding

7) cursoring: changing the pointer variable itself (moves the pointer along the array)

8) loop termination: address comparison &vctr[4] is the address of the last (5th) element. loop continues as long as not advances past end of array



Now, similar to previous, but array of integers instead if array of characters.

1) pointer to integer

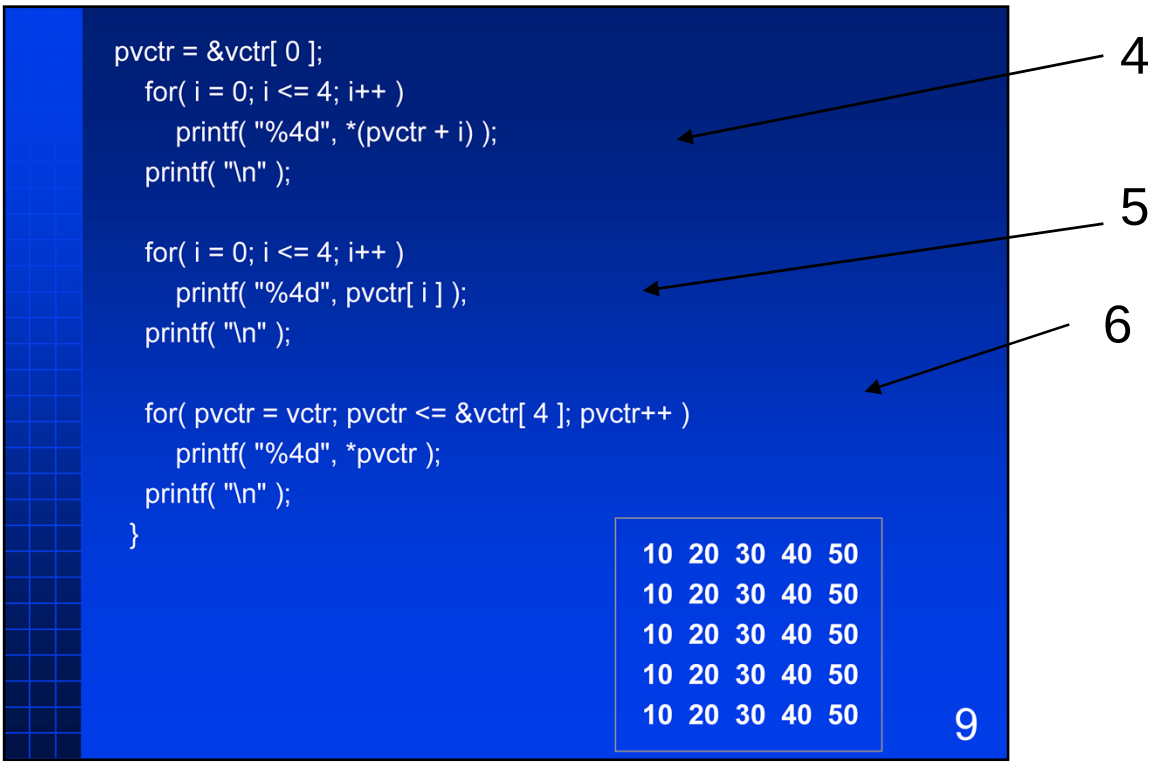
2) simple array subscripting

3) pointer arithmetic on array-name (compute address and dereference)

--> same code as before (vctr + i): how can this work? adding one would point at the middle of an integer

--> answer: C adjust for this size of the thing being derference of for the size of the thing to which a pointer points. So, if vctr points at an 4-byte integer, (vctr+i) becomes (vctr + i*sizeof(int))

so each increment of i adds 4 instead of 1

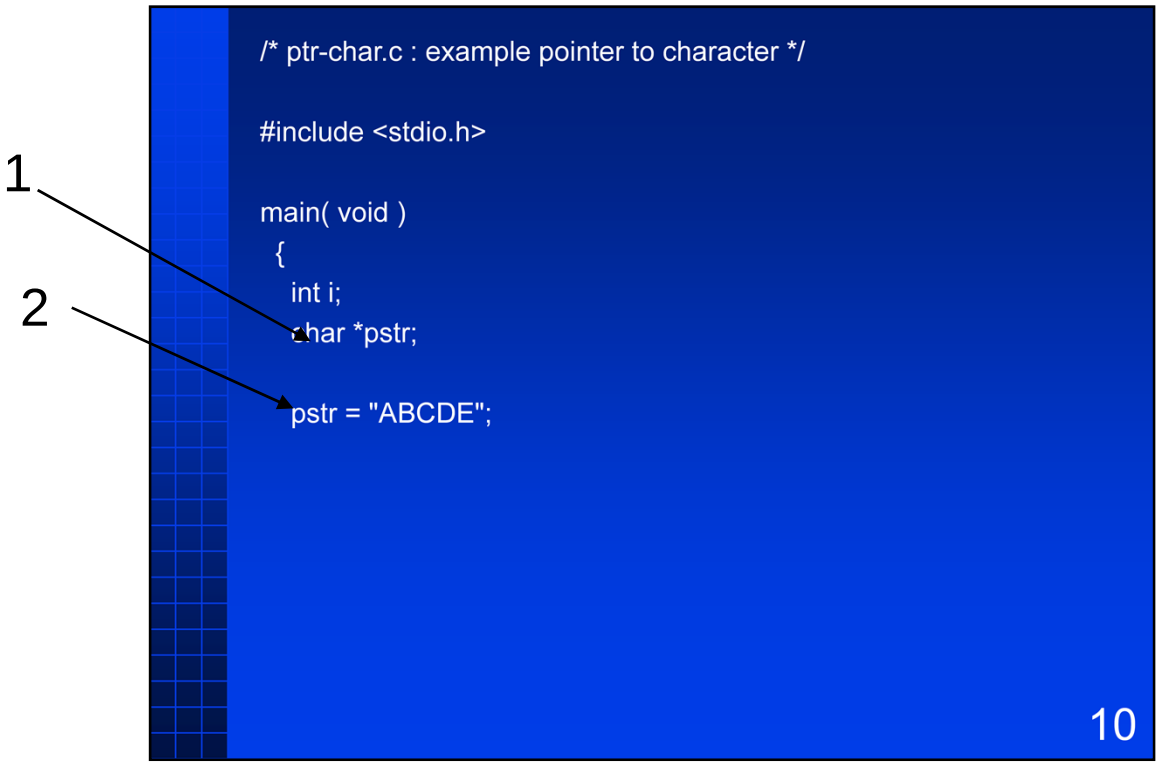


4) pure pointer arithmetic. +i adds sizeof(*pvctr)

5) subscripting a pointer. full definition of subscripting is

x[i] == *(x + i*sizeof(*x))

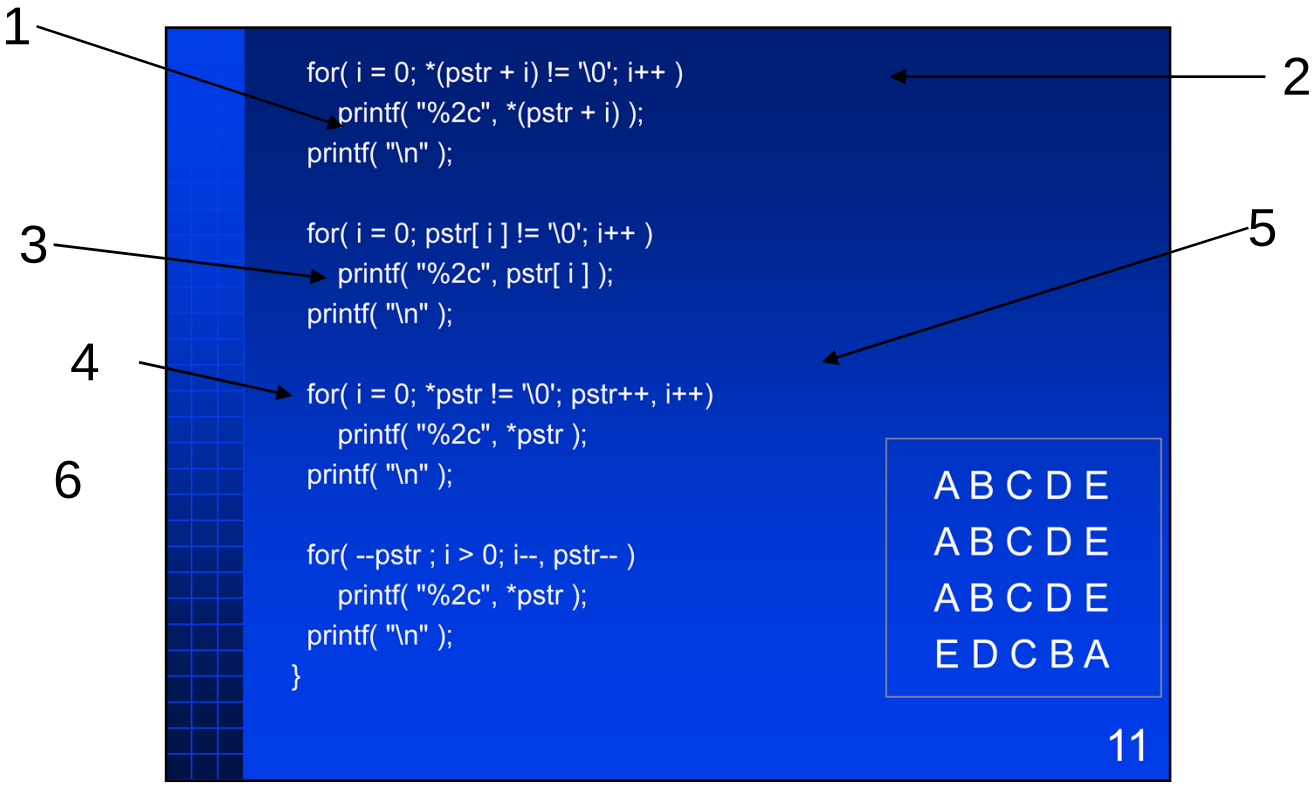
6) cursoring, moving the pointer. ++ here means “plus the size of the thing to which I point”



Now, use character pointer to manipulate traditional null-terminated strings. look at ways to get at individual characters within strings.

1) pointer to character

2) assign the address of the string literal to the pointer var. a literal string is represented as an array of char (as discussed), so assigning the string is really just assigning the address. the string literal automatically contains a null (C adds it).



1) pure pointer arithmentic. compute and dereference to a single character

2) termination condition: while not at a nullchar

3) subscripting a pointer variable

4) cursor (moving the pointer) forwards. termination same (not null), but simpler expression, since pstr itself is moving

5) Note comma op. want to advance pointer and counter together. counter doesn't actually do anything for us in this loop (used in next one)

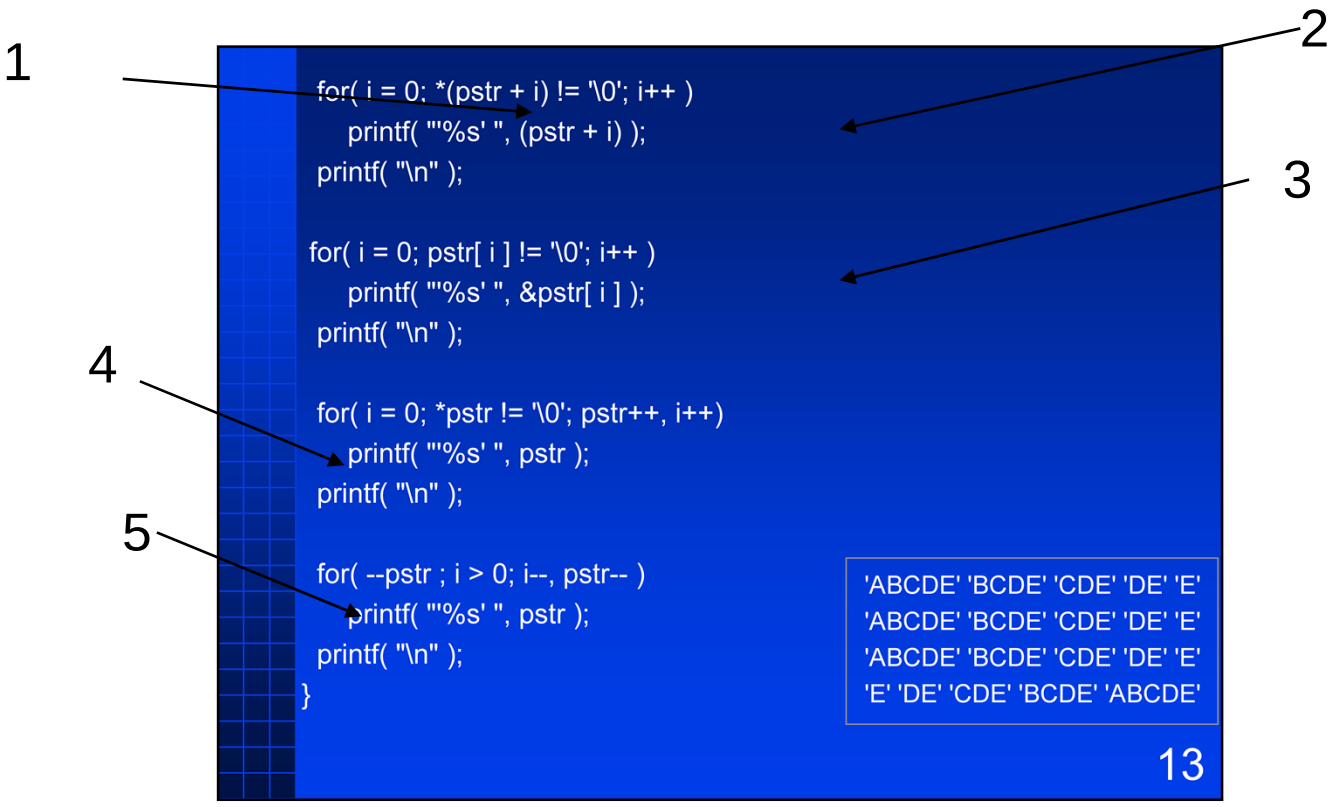
6) cursor backwards. note clever predecrement to move from nullchar (where previous loop ended). use i here for termination test. Note comma operator

```
/* ptr-str.c : example pointer to string */

#include <stdio.h>

main( void )
 {
   int i;
   char *pstr;

   pstr = "ABCDE";
```

12

similar ideas. instead of dealing with individual chars, manipulate strings.

printf control strings will have %s instead of %c



loop structure: display substrings of varying lengths. terminate at nullchar

1) print strings instead of a char

2) no dereference: %s wants the address of a string, (pstr+i) is an address expression that yields the address of a null-terminated string.

3) we want the address of a string, pstr[i] is a char, toss in the & to get the address of the character, suitable as string address

4) move the pointer along the string. no dereference.

5) move the pointer backwards. still no dereference.

```
/* ptr-strf.c : example pointer, string, char */
#include <stdio.h>
#include <string.h>

main( void )
 {
   char *pstr1, *pstr2;

   pstr1 = "ABCDE";
   pstr2 = strchr( pstr1, 'C' );

   printf( "pstr1: %s\n", pstr1 );
   if( pstr2 != NULL ) {
      printf( "pstr2: %s\n", pstr2 );
      printf( "*pstr2: %c\n", *pstr2 );
   }
 }
```

```
pstr1: ABCDE
pstr2: CDE
*pstr2: C
```

14

another example of relationship between pointers, characters and string

1) strchr is one of many functions that manipulate string. see library reference. many are standard, many more are not.

Strchr looks for occurrence of single character in a string. returns pointer to string, or null if not found

2) display string, no dereference, string address for %s

3) display character, dereference, single char for %c

String literals, variables and pointers

```
{
  char str[100];
  char *pstr1, *pstr2;

  strcpy( str, "ABCDE" );
  pstr1 = str;
  pstr2 = “ABCDE”;
}
```

- str is a variable, storage is read/write
- “ABCDE” is a literal, might be read-only
- *pstr1 can be modified, *pstr2 should not

15

preceding examples used pointer to string literal, not typical use.

in fact, trying to change a string literal might cause an error

more typical usage is to defin storage (eg string variable), and then define a pointer to it

can’t change the value of a string variable (eg don’t change str)