CS1100
Introduction to Programming

Pointers

Madhu Mutyam
Department of Computer Science and Engineering
Indian Institute of Technology Madras

Accessing Arrays with Pointers

```
#include <stdio.h>
int myArray[ ] = {1,24,17,4,-5,100};
int *ptr;
int main(void){
  int i;
  ptr = &myArray[0];
  printf(“\n”);
  for (i = 0; i < 6; i++){
        printf("myArray[%d] = %d ", i, myArray[i]);
        printf(“value at ptr + %d is %d\n", i, *(ptr + i));
  }
  return 0;
}
```

ptr++ and ++ptr

- ++ptr and ptr++ are both equivalent to ptr + 1
  - though they are “incremented” at different times
- Replace the following statement

  ```
  printf(“value at ptr + %d is %d\n", i, *(ptr + i));
  ```

  with:

  ```
  printf("ptr + %d = %d\n",i, *ptr++);
  printf("ptr + %d = %d\n",i, *(++ptr));
  ```

**ptr++*

- **ptr++* is to be interpreted as returning the value pointed to by *ptr* and then incrementing the pointer value.
- This has to do with the precedence of the operators.
- (*ptr)++ would increment, not the pointer, but that which the pointer points to!
  - i.e. if used on the first character of the example string “IIT” the ‘I’ would be incremented to a ‘J’.

Arrays

- The name of the array is the address of the first element in the array
- In C, we can replace

  ```
  ptr = &myArray[0];
  ```

  with

  ```
  ptr = myArray;
  ```

  to achieve the same result
- Many texts state that the name of an array is a pointer

Array Names Are Not Pointers

- While we can write

  ```
  ptr = myArray;
  ```

- we cannot write

  ```
  myArray = ptr;
  ```

- The reason:
  - While ptr is a variable, myArray is a constant
  - That is, the location at which the first element of myArray will be stored cannot be changed once myArray has been declared

Pointer Types

- C provides for a pointer of type void. We can declare such a pointer by writing:

  `void *vptr;`

- A void pointer is a generic pointer
  - For example, a pointer to any type can be compared to a void pointer
- Type casts can be used to convert from one type of pointer to another under proper circumstances

Trying Out Pointers

```
#include <stdio.h>
int j = 1, k = 2; int *ptr;
main( ) {
ptr = &k;
printf(“\n j has the value %d and is stored at %p”,j,(void*)&j);
printf(“\n k has the value %d and is stored at %p”,k,(void*)
                                                          &k);
printf(“\n ptr has the value %p stored at %p”, ptr, (void *)
                                                        &ptr);
printf(“\nThe value of the integer pointed to by ptr is %d\n”,
                                                         *ptr);
}
```

Generic address of j

Dereferencing – will print r-value of k

Pointers and Strings

- C does not have a string type
  - languages like Pascal, Fortran have…
- In C, a string is an array of characters terminated with a binary zero character (written as '\0’)
  - remember this is not the character ‘0’
- One can manipulate strings as character arrays
- Character arrays can also be accessed by pointers

A Character Array

- One could create a string as follows,

  ```
  char myString[40];
  myString[0] = 'T';
  myString[1] = 'e';
  myString[2] = 'd’;
  myString[3] = '\0’;
  ```

  - Note - terminated with a *nul* character
- *nul* (or ‘\0’) ≠ NULL (pointer to nothing)
- Obviously this is very tedious

Ted Jenson’s tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

“Strings”

- One might write:

  `char myString[40] = {'T', 'e', 'd', '\0’};`

- But this also takes more typing than is convenient
- So, C permits:

  `char myString[40] = "Ted";`

  - Note that C automatically inserts ‘\0’
- Compiler sets aside a contiguous block of memory 40 bytes long
- The first four bytes contain Ted\0

Strings: Input and Output

- The function *gets*( ) accepts the name of the string as a parameter, and fills the string with characters that are input from the keyboard till newline character is encountered. At the end, a null terminator is appended.
  - Not a popular function because there are no built-in checks
- char ****gets***(char *s);
- ***gets***(*str*) – reads from standard input into *str*
- ***puts***(*str*) – writes to standard output from *str*

gets may Overwrite Memory

```
char b1[] = “ABCDE”;
char b2[] = “LMNOF”;
char b3[] = “ZYXWV”;
puts(b1);
puts(b2);
puts(b3);
puts(“Input:”);
gets(b2);
puts(b1);
puts(b2);
puts(b3);
```

A sample run

```
puts(b1); ABCDE
puts(b2); LMNOP
puts(b3); ZYXWV
puts(…); Input: 1234
gets(b2);
puts(b1); ABCDE
puts(b2); 1234
puts(b3); ZYXWV
```

Another run

```
puts(b1); ABCDE
puts(b2); LMNOP
puts(b3); ZYXWV
puts(…); Input: 1234567890
gets(b2);
puts(b1); 7890
puts(b2); 1234567890
puts(b3); ZYXWV
```

Character Pointers

```
#include <stdio.h>
char strA[80] = "A string to be used for demonstration";
char strB[80];
int main(void)
{
   char *pA;          /* a pointer to type character */
   char *pB;          /* another pointer to type character */
   puts(strA);        /* show string A */
   pA = strA;         /* point pA at string A */
   puts(pA);          /* show what pA is pointing to */
```

--continued →

Copying Strings…

```
   pB = strB;                /* point pB at string B */
   putchar('\n');       /* move down one line on the screen */
   while(*pA != '\0')        /* while string  */
    {
     *pB++ = *pA++;      /* copy and increment pointer */
    }
   *pB = '\0';               /* insert end-of-string */
   puts(strB);               /* show strB on screen */
   return 0;
}
```

Ted Jenson’s tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

A Version of *strcpy*

```
char *myStrcpy(char *destination, char *source)
{
 char *p = destination;
 while (*source != '\0')
   {
     *p++ = *source++;
   }
 *p = '\0';
 return destination;
}
```

Ted Jenson’s tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm

Equivalent Definition Using Arrays

```
char *myStrcpy(char dest[], char source[])
{
 int i = 0;
 while (source[i] != '\0')
  {
    dest[i] = source[i];
    i++;
  }
 dest[i] = '\0';
 return dest;
}
```

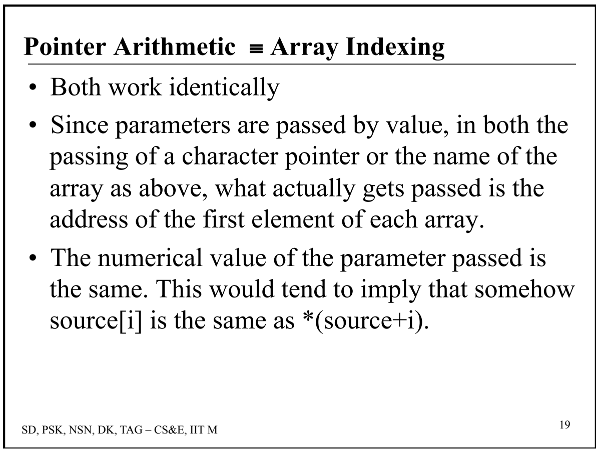
```
char *myStrcpy(char *destination, char *source)
char *p = destination;
while (*source != '\0')

*p++ = *source++;

*p = '\0';
return destination;
```

Copying Arrays Using Pointers

- Exercise – define a function to copy part of an integer array into another. Access the elements using pointers.
- Function prototype:

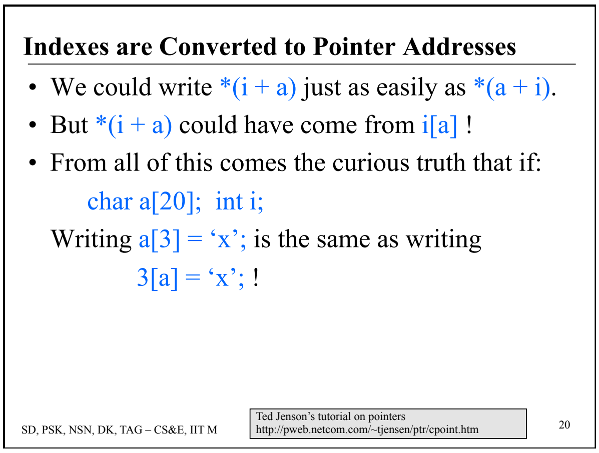
  void intCopy(***int*** *ptrA, ***int*** *ptrB, ***int*** num);

  – where num is the number of integers to be copied.



Pointer Arithmetic ≡ Array Indexing

- Both work identically
- Since parameters are passed by value, in both the passing of a character pointer or the name of the array as above, what actually gets passed is the address of the first element of each array.
- The numerical value of the parameter passed is the same. This would tend to imply that somehow source[i] is the same as *(source+i).



Indexes are Converted to Pointer Addresses

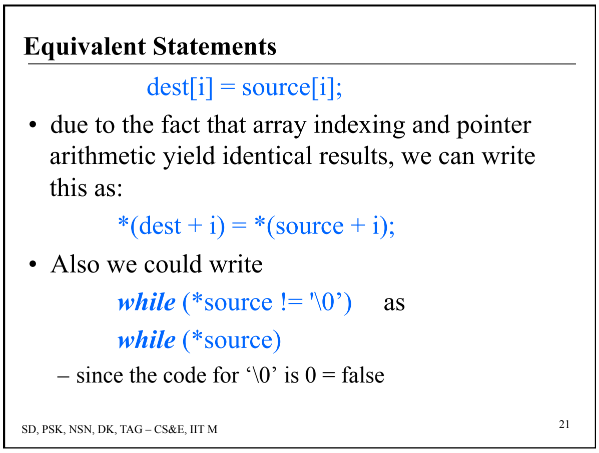
- We could write *(i + a) just as easily as *(a + i).
- But *(i + a) could have come from i[a] !
- From all of this comes the curious truth that if:

  char a[20]; int i;

  Writing a[3] = ‘x’; is the same as writing

  3[a] = ‘x’; !

Ted Jenson’s tutorial on pointers
http://pweb.netcom.com/~tjensen/ptr/cpoint.htm



Equivalent Statements

dest[i] = source[i];

- due to the fact that array indexing and pointer arithmetic yield identical results, we can write this as:

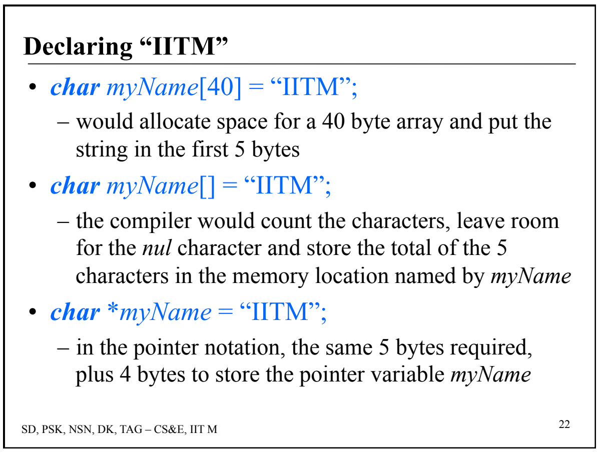
  *(dest + i) = *(source + i);

- Also we could write

  while (*source != '\0’) as

  while (*source)

  – since the code for ‘\0’ is 0 = false



Declaring “IITM”

- ***char*** *myName*[40] = “IITM”;
  – would allocate space for a 40 byte array and put the string in the first 5 bytes
- ***char*** *myName*[] = “IITM”;
  – the compiler would count the characters, leave room for the *nul* character and store the total of the 5 characters in the memory location named by *myName*
- ***char*** **myName* = “IITM”;
  – in the pointer notation, the same 5 bytes required, plus 4 bytes to store the pointer variable *myName*