# Pointers and Arrays

## © 2005 Devendra Tewari

# Introduction

## Pointers are variables that store memory addresses

## They store the address of a memory region that stores a particular type of data

## The size of a pointer is determined by the address size of the CPU

### int \* p;

### int i = 10;

### p = &i;

# Pointer declaration

## A pointer variable is declared using the \* operator

### int \* p;

## \* is called the dereferencing operator because \*p gives the value of the variable p points to

## The & operator is used to recover the address of a variable in memory, it cannot be applied to expressions, constants or register variables

### p = &i;

# Pointer assignment and usage

## Pointer can be assigned to one another

### int i = 10;

### int \* ip = &i;

### int \* iq = ip; /\* iq now points to i \*/

## Operator precedence in usage scenarios of \* operator

### \*ip += 1;

### ++\*ip;

### (\*ip)++;

#### increment value pointed to by ip

#### \*ip++; would be incorrect in last example

# Function arguments

## Arguments are passed to a function by value, even pointer arguments

## Pointers provide a mechanism for functions to alter the value of referenced variables

## Write a function that swaps the value of it's arguments

# Arrays

## Arrays provide contiguous storage to several elements of the same type

### int a [10];

#### declares an array of 10 integers

## Elements for external, static and automatic variables are initialized to zero

## The array index is zero based

# Array initialization

## Arrays can be initialized during declaration

### int days[] = {5, 10, 15, 25, 30};

#### compiler fills in the size and fills the array

### char name[] = "name";

#### right-hand side is a string constant

### char name[] = {'n', 'a', 'm', 'e'};

## Arrays can be initialized using assignment statements or using loops

### int days[5];

### days[0] = 5;

# Pointers and Arrays

## Arrays and pointers are related

### int a[5] = {0 , 1, 2, 3, 4};

### int \* p = &a[0];

### int \* q = a;

#### a always points to the start of the array and cannot be changed

# Pointer operations

## Pointers can be incremented in integer steps

## p++ points to the next element

### what does \*p++ = 10 do? (hint – see operator precedence table)

## p-- points to the previous element

### what does \*--p = 10 do? (hint – see operator precedence table)

## p+=i points to i elements beyond the current position

## p-=i points to i elements before the current position

## p = 0 or p = NULL makes p a null pointer i.e. a pointer that does not point to anything in particular, a valid pointer value

# Strings

## Strings constants are arrays of char

### char name [] = "name";

## Since an array of char can be assigned to a pointer to char, a pointer to char can refer to a string constant

### char \* name = "name";

## strlen can be used to calculate length of a string

### strlen(name) returns 4

## A string is internally padded with NULL character or '\0', name is thus internally 5 characters long

## Write a function to replace strcpy

# Multi-dimensional Arrays

## Declaration

### int a[10][20];

#### 10 rows and 20 columns, *contiguous* storage for 200 integers

## Initialization

### int a[][2] = {{1,2}, {3}};

### int (a[])[2] = {{1,2}, {3}};

#### The number of columns (length of each row) needs to be known beforehand, try printing a[1][1], what do you get?

# Array of Pointers

## This is how you would construct an array of string constants

### char a[][7] = {"hello", "world!"};

### printf("%s %s\n", a[0], a[1]);

## The multi-dimensional array above is similar to an array of pointers to char

### char \* a[] = {"hello", "world!"};

### printf("%s %s\n", a[0], a[1]);

# Pointers v. multi-dimensional arrays

## int a[2][2] = {{1,2},{3,4}};

## int \*b[2], \*\*c, \*d;

## b[0] = a[0]; b[1] = a[1];

## c = b; d = (int \*)a;

## printf("%d\n", a[1][1]);

## printf("%d\n", \*(\*(a + 1) + 1));

## printf("%d\n", b[1][1]);

## printf("%d\n", \*(\*(b + 1) + 1));

## printf("%d\n", c[1][1]);

## printf("%d\n", \*(\*(c + 1) + 1));

## printf("%d\n", d[3]);

## printf("%d\n", \*(d + 3));

# Command line arguments

### main (int argc, char \* argv[])

#### argc is the number of arguments in the command-line that invoked the program, always at least 1 because the program name is itself an argument

#### argv is an array of pointers to char, each element points to a string

# Pointers to Functions

## Pointers can point to functions, although functions are very different from variables, they do have an address where they begin

## Declare a pointer to a function

### int (\*p)(int \* a, int \* b)

## Assign a function

### p = add;

## Call the function

### int a = b = 2;

### (\*p)(&a, &b);

# void pointer

## Any pointer type can be assigned to, or passed to a function as, a void pointer

### int \* ip;

### void \* vp = ip;

## void pointer can be cast to any pointer type

### char \* cp = (char \*) vp;

## Useful for making generic functions that apply to various types

## Be careful with casting void \* to another type, know what you are doing

# Dynamic memory allocation

## Pointers not yet initialized are dangerous if they are not NULL pointers

## Pointers can be initialized to point to storage dynamically allocated using malloc and calloc

## free must be used to release the memory allocated using the above functions

# malloc

## void \* malloc(size\_t n)

### Allocates n bytes of storage and returns a void pointer to it

## int \* ip = (int \*)malloc(10 \* sizeof(int));

## free(ip);

### sizeof is an operator that returns the size of the object or type specified

# calloc

## void \* calloc(size\_t n, size\_t size)

### Allocates memory for n objects of size size and returns a void pointer to it

### The memory assigned is initialized to zeros

## int \* ip = (int \*)calloc(10, sizeof(int));

## free(ip);

# Memory problems

## Using an unallocated pointer

## Writing to memory outside the allocated region (buffer overflow)

## Freeing memory not allocated using malloc or calloc

## Not freeing memory allocated using malloc and calloc (memory leak)

# Detecting using memwatch

## memwatch [1] is distributed as a single source file memwatch.c and it's accompanying header file memwatch.h

## Source files you want to watch for memory problems must include memwatch.h and be recompiled using the following compiler options:

### -DMEMWATCH -DMW\_STDIO

## memwatch prints an error message in the standard output and produces a detailed log file listing the memory problems it encounters

# Exercise

## Write a program that sorts an array of strings. Use your favorite sorting algorithm (bubble sort, insertion sort, etc). Write your own replacement for strcmp to compare the strings. Write a generic sort function that can work with arrays of other types

# Tools and References

## memwatch – http://www.linkdata.se/sourcecode.html