Nested For Loops, Functions, Arrays, and File I/O

July 11, 2005

Announcements

- Project 1 due today
  - Sign up for demos on CPM
- Mid-semester survey during break
- Returning midterms at end of class
- Final is August 12 at 7 p.m. in Gore 306
  - Also posted on course web site

Nested For Loops

- Two questions when writing single for loop
  - What are you repeating?
  - How many times are you repeating?
- **Nested for loops**
  - May repeat a loop!
  - Example: printing the square of asterisks

Printing a square of asterisks

- Previously, we counted the total number of asterisks and used an if statement to add newline characters
- Rethinking…
  - What are we repeating?
  - How many times do we repeat?

Draw 3 stars in each row

```
* * *
* * *
* * *
```

Drawing 3 stars? That sounds like a loop!

Nested For Loops

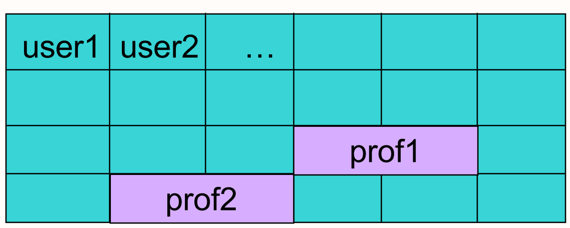
- Inside loop
  - Done once for each iteration of outside loop
- Outside loop
- Each loop has its own counter variable
- Example:

```
For each row
      For each column
            Print a star
```

- Code simple nested for loop with two different counter variables (nestedfor.c)

Handling your quota

Disk Space



- Disk space is broken into pieces, one for each user.
- You need to manage your allotted piece.
  - Around 512 MB

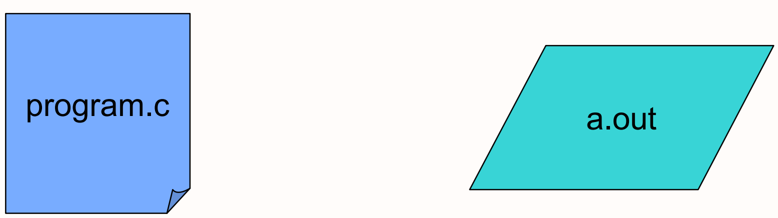
Unix Commands: Handling your quota

- List the amount of **disk usage** for the files, look for largest files (contain MB of data)
  - du -sh <file list>
  - du -sh <file list> | grep M

Called "pipe".
Means pass output of command to input of next

- How much disk space am I using?
  - quota -v
- Find the a.out files
  - find . -name a.out
- Remove the a.out files
  - find . -name a.out | xargs rm

Different types of "space" requirements: disk versus memory

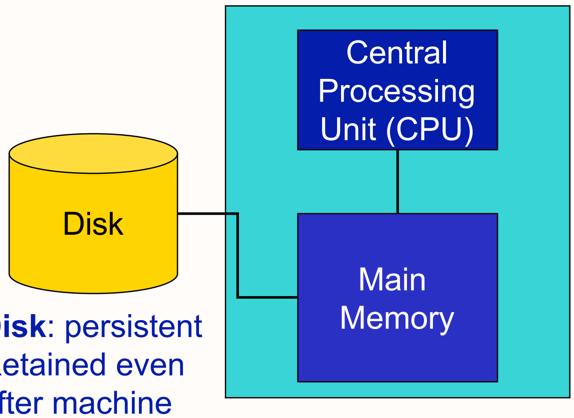


Takes up relatively little **disk** space

Executable takes up more disk space than the .c file

When executed, requires **memory** space

Computer Architecture



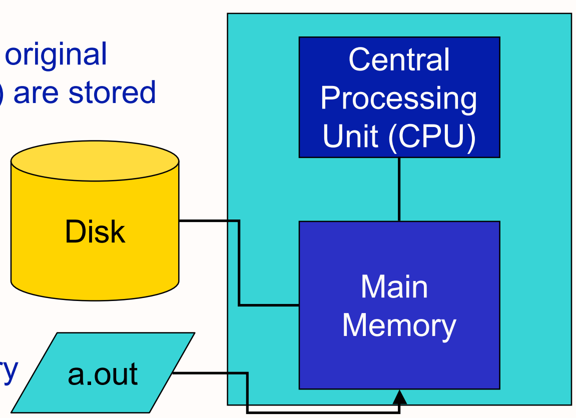
Disk: persistent Retained even after machine shuts down

Main memory: volatile memory gets erased when program exits or machine shuts down

Note: picture is not to scale
CPU << Main Memory << Disk

Computer Architecture

Executable (and original program text file) are stored on disk



Load executable into main memory and process

What affects memory requirements

- During execution of the program
- How many variables used?
  - Appropriate variable types/sizes
- Often have a tradeoff between time and space
  - Can compute faster if store more information in variables, but requires more memory!

Variable Sizes in Memory

- We can find out the size of a variable
  - Using sizeof()
- See sizeof.c

Review: local variables

- Parameters and variables in a function
  - Cannot be accessed or used by other functions (except by being passed as arguments or as return values)
- Created (or **allocated**) on function entry
- **Deallocated** on function return
  - Remember our stacks
- Parameters are initialized by copying the value of the argument ("**call-by-value**")
- Localize information, reduce iteractions

Dealing with lots of data

- Recall the lab problem of finding the minimum, maximum, and average of students' grades
  - We basically "threw away" each input value
  - Only kept what we needed in the current min and max variables and a running sum
  - *What if we wanted to keep those grades for later processing?*

Grades (sentinel -1):

90 85 70 68 57 91 82 81 95 75 78 -1

Solutions to storing student grades

- Could create a variable for each student grade:
  - int grade1, grade2, grade3, …;
- Problem?

Arrays!

- Named, ordered collection of variables of the same type

Indices

Values

0	90
1	85
2	70
3	68
4	57
5	91
6	82
7	81
8	95
9	75
10	78

Conceptually, array contains 11 variables.

Arrays!

Example declaration: int grades[11];

Need to specify the size of the array

Indices: identify the separate variables

0	90
1	85
2	70
3	68
4	57
5	91
6	82
7	81
8	95
9	75
10	78

Values: the data stored in the variables

Access array's content:

grades[0] is 90
grades[10] is 78
2*grades[4] is 104

Array Declaration Syntax

- type identifier[size];
  - size must be a *positive* int constant or literal
- int grades[11];
  - grades is of type array of int with size 11
  - grades[0], grades[1], …, grades[10] are elements of the array grades
    - Each is a variable of type int
  - the bounds are the lowest and highest values of the indices (0 and 10 in this example)

Storing and Retrieving Data in Arrays

- Array: collection of variables
  - An **element** is a variable, can be used anywhere that a simple variable of that type can be used
    - Assignment, expressions, I/O
  - Must be declared before use

Storing and Retrieving Data in Arrays

- An array is treated differently than a single variable
  - Can't assign or compare arrays with =, ==, <, …
  - Can't use printf or scanf on an entire array
  - Can do them one element at a time
  - Examples:
    - grades[4] = 100;
    - total += grades[i];
    - if(grades[0] == grades[1]) {/*do something*/}

Index Rule

- An array index must evaluate to an int between 0 and n-1 (inclusive), where n is the size of the array
  - Accessing at indices < 0 or >= n will cause problems (e.g., seg. faults or weird values, etc.)
- Example:
  - The size-1
  - grades[i+3+k]   // OK as long as 0 <= i+3+k <= 10
- The index may be simple: grades[0]
- Or complex:
  - grades[(int) (3.1*fabs(sin(2.0*PI*sqrt(29.067))))]

Keeping Track of Elements in-use

- When we're reading in grades, we don't know how big to make the array
  - Arrays must be of fixed size
  - Declare the array bigger than you think you'll need

```
#define MAXGRADES 200
int grades[MAXGRADES];
```

0	90
1	85
2	70
3	68
4	57
…	…
199	-

Keeping Track of Elements in-use

- Need to keep track of which entries contain valid entries
  - Keep all valid entries at beginning of array
  - Another variable with number of valid elements

After element 10, all entries are "empty".
Keep variable numGrades = 11

0	90
1	85
2	70
3	68
4	57
…	…
199	-

Practice programs

- Change averaging student grades to maintain the grades
- Extension: How do we calculate and print the number of grades that are above average?
  - Demonstrate tradeoff between execution space and time!

Array Initialization

- Initialization:

```
int grades[5];
grades[0]=90;
grades[1]=85;
grades[2]=70;
…
```

What are the valid indices for this array?

- Single-line initialization:

```
int grades[5] = {90, 85, 70, 68, 57};
```

- Implicit initialization:

```
int grades[5] = {90};
```

Rest of entries are implicitly initialized to 0

Using Array Elements in Functions

- Adding two numbers together

```
int sum( int a, int b ) {
    return a+b;
}
```

- Declare an array of integers
- Declare another array of integers
  - Store the result of adding two consecutive elements in the array

Using Whole Arrays in Functions

- Compute average in a function:

How to pass array (empty brackets)

The number of valid entries in the array

```
double computeAverage( int a[], int num ) {
        int i; double total;
        for( i=0; i < num; i++ ) {
            total+=a[i];
        }
        return total/num;
}
```

Anything stylistically nice about this function?

Using Whole Arrays in Functions

- Compute average in a function:

Note that prototype includes the brackets too

```
double computeAverage( int a[], int num );

int main() {
        int grades[MAXGRADES];
        double gradeAvg;
        …
        gradeAvg = average(grades, numGrades);
}
```

Don't need the brackets when passing the array

Whole Arrays As Parameters

- Entire arrays as parameters work differently than variables
  - Array is never copied, i.e., pass-by-reference
    - We'll talk more about pass-by-ref next week
  - If modify the array in the function, the array changes outside the function
- Arrays do not contain information about their size
  - Must pass the size of the array as an additional parameter (or use a constant)

Reading Data From a File

- Inputting all those grades by hand is tedious
- Put data in a file and read from the file
  - New data type: FILE
    - Defined in stdio.h
  - Initialization:
    - FILE *file_ptr;

Need the "*"

data.txt

90 85 70
68 57 91
82 81 95
75 78

Checklist for reading files

- Open the file
  - Specify what opening the file for (read or write)
- Check that the file actually opened
- Read the file
  - Use **fscanf**, similar to scanf
  - Adds the file pointer as the first parameter
- Close the file

Opening the file

Either
"r" for read
"w" for write

- Prototype:
  - FILE* **fopen**(char* filename, char* mode);
  - Returns NULL if there was some problem
    - File does not exist, incorrect permissions
- Example usage:
  - file_ptr = fopen("data.txt", "r");
- Check that file opened

```
if( file_ptr == NULL ) {
    printf("File 'data.txt' did not open.\n");
    exit(1);
}
```

Reading from the file

- int fscanf(FILE* stream, char* format, ...);
  - Returns number of matches (variables defined)
    - 0 if incorrect format specifier
    - EOF (a defined constant) if no more to read
      - From stdio.h
      - Means "End-of-File"
      - Use: while(fscanf(...) != EOF) { /* do stuff */ }
  - Keeps track of where you are in the file

Reading from the file

- int fscanf(FILE* stream, char* format, ...);
  - Returns number of matches (variables defined)
  - Keeps track of where you are in the file

Before any read

data.txt

90 85 70
68 57 91
82 81 95
75 78

After one call to
fscanf(file_ptr, "%d", &input);
input now has value 90.

Note that don't need "*" before file_ptr variable in function call.

Close the file

- Prototype:
  - int fclose(FILE *stream);
- Example use:
  - fclose(file_ptr);

Practicing Reading from a File

- Modify the grade program to read from a file
  - 2 different stopping criteria
    - a sentinel value
    - EOF

Reading from the terminal (stdin)

- Can read from the terminal using fprintf and fscanf
  - **stdin** is FILE* variable
  - stdin is short for "standard in"

Writing Data to a File

- Instead of writing to a terminal, may want to keep the program output in a more permanent form
- Similar to printing to the terminal:
  - **fprintf**: adds file pointer as first parameter
- Prototype:
  - int fprintf(FILE* stream, char* format, ...);
- Example use:
  - fprintf(file_ptr, "Val is %d.\n", val);
- Can use **stdout** to write to the terminal

Writing Data to a File

- fprintf also keeps track of where written in the file

Before writing

out.txt

start out

After one call to
fprintf(file_ptr, “start out\n”);

Note that don’t need “*” before file_ptr variable in function call.

Data Structures

- Functions help us organize programs
- How can we organize data?
  - Data structures!
    - Organize large amounts of data
    - Organize variable amounts of data
    - Organize related data
- In this course, we will structure data using arrays and structs

Multidimensional arrays

- int mult[4][4];

Number of rows — Number of columns

	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	4	6
3	0	3	6	9

- Each row is an array of size 4 (columns)
  - mult[1] is the array {0, 1, 2, 3}, mult[2][2] is 4

Multidimensional Arrays

- Can have lots of dimensions!
- But what is the most logical way to organize the data?
- Consider storing the high and low temperatures for every day for several years
  - What data structure/array form would you use to store that information?

Initializing multidimensional arrays

- Can initialize arrays like 1-d arrays
  - int array[2][3] = {1, 2, 3, 4, 5, 6}
  - int array[4][3] = {{1, 2, 3},
    {4, 5, 6},
    {7, 8,9},
    {10,11,12}}
    - If leave out any of value, implicitly set to 0

Multidimensional Arrays as Parameters

- void mod_array(int a[][COLS], int rows);

Need to specify the size of the dimensions for all but the first dimension

- Calling function:
  - int matrix[ROWS][COLS];
  - modMatrix(matrix, ROWS);

Sorting Numbers in an Array

- We may want to sort the values in an array
  - Sorting grades makes it easier to find the median grade
- Consider a small example:
  - Sort **int array[3]** such that array[0] <= array[1] <= array[2]
- Can we extend the basic idea to larger arrays?

Midterm

- 20% of your grade
- Everyone gets 2 bonus points to make exam out of 150 points
- Solutions to problems
  - Field width *includes* precision and decimal point
  - Negating a condition