Software Systems

Lectures Week 7

Introduction to C

Dynamic memory – Text Files – Test 2

(Functions, Scope, Files, Structures)

Prof. Joseph Vybihal

Computer Science

McGill University

Part 1

Dynamic Memory

When can we define an array?

int array[10]; ← Yes

int n=10;
int array[n]; ← Yes

int n;
scanf("%d", &n);
int array[n]; ← No. Why?

Arrays are created at compile time.

The array limit problem

```
int array[10];
PERSON people[100];
```

Notice that we need to define the size of the array.

What if, at run-time, we realize we need more memory?

Dynamic Memory

Creating data structures while the program is running.

Steps:

1. At compile-time define the data structure type
2. At run-time ask the system for memory formatted according to your defined data structure type
3. If the system returns NULL then it was not successful
4. When you are finished using the data structure return the memory back to the system

C's dynamic memory functions

#include<stdlib.h>

- void *malloc(int size);
  - Creates one data structure of 'size'
- void *calloc(int multiples, int size);
  - Creates an array of data structures of type 'size'
- free(void *);
  - Returns the data structure's memory

Notice that the functions return a void* pointer. These pointers can point to anything regardless of type. Very powerful.

- It is customary to cast void* into the data structure type you want

Example

```
#include <stdlib>
int main(void) {
    int *array;
    int n;

    scanf("%d", &n);                    // notice we define size of array at run-time
    array = (int *) calloc(n, sizeof(int));   // int is 4 bytes, can replace sizeof with 4
    if (array == NULL) exit(1);

    *(array+2) = 5;                     // notice how we access data in array
    printf("%d", *(array+2));
    free(array);

    return 0;
}
```

Example

```
struct STUDENT {
    int age;
    float GPA;
};

struct STUDENT *x;

x = (struct STUDENT *) malloc(sizeof(struct STUDENT));
if (x == NULL) exit(1);

// two ways to access the contents
(*x).age = 5;
x->age = 5; // this is more common
```

Example

```
struct STUDENT *students, *aStudent;
int n, x;

scanf("%d", &n);
students = (struct STUDENT *) calloc(n, sizeof(struct STUDENT));
if (students == NULL) exit (1);

for(x=0; x<n, x++)  {
    aStudent = students+x;
    scanf("%d %f", &(aStudent->age), &(aStudent->GPA));
}

// or: &((students+x)->age)
```

Linked Lists

The previous examples assume the user knows the size of the array at some point…

what if the user will never know…

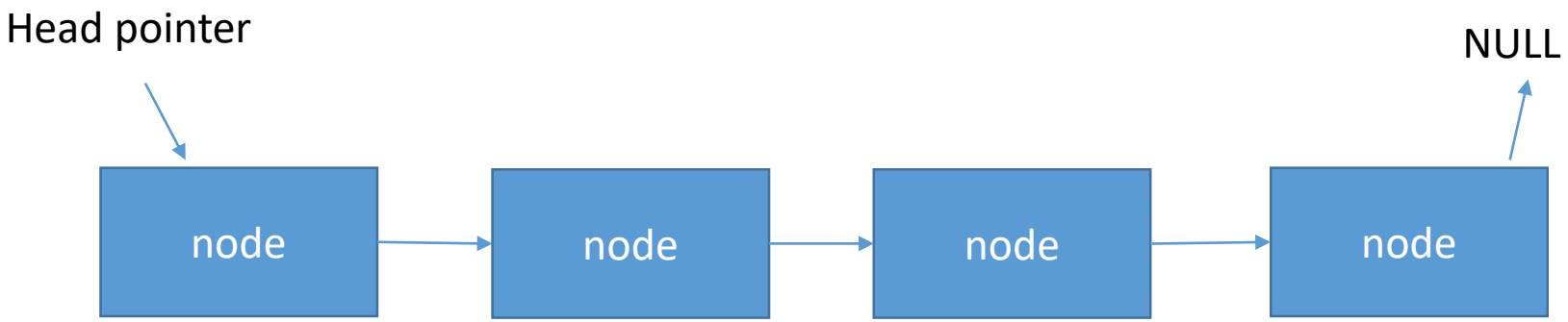
Eg:

- How many students will register for a course?
- How many cars will stop at the traffic light?

Linked Lists

A linked-list is a data structure that can grow or shrink in size gradually, as needed.

It looks like this:



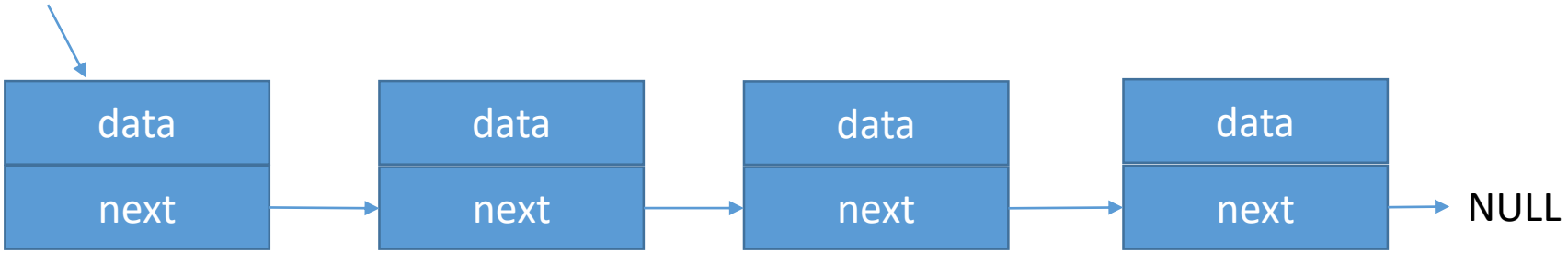
Node

```
struct NODE {
    int data;                 // the data we want to store in the node
    struct NODE *next;        // the pointer to the following node in the list
};

struct NODE *aNode;
aNode = (struct NODE *) malloc(sizeof(struct NODE));
if (aNode == NULL) exit(1);

aNode->data = 0;              // initialize to zero
aNode->next = NULL;           // The next pointer is not pointing to anything
```

The List

```
struct NODE *head;   // always points to the first node in a list
                     // the last node always points to NULL
```

head



Traverse an assumed list example

```
int main() {
    struct NODE * head = ... A list previously created…

    printNodes(head);
}

void printNodes(struct NODE *ptr) {          // copy of head pointer
    struct NODE *temp = ptr;
    while (temp != NULL) {                   // stop at end of list
            printf("%d\n", temp->data);
            temp = temp->next;               // move to the following node in list
    }
}
```

Creating a list example

```
int main() {
    struct NODE * head = NULL;
    int x, newData;
    for(x=0; x<10; x++) head = addNode(head, newData);  // newData scanf'd in loop, not shown.
}
struct NODE* addNodes(struct NODE *ptr, int someData) {
        struct NODE *temp = (struct NODE *) malloc(sizeof(struct NODE));
        if (temp == NULL) return NULL;          // NULL to designate error
        temp->data = someData;
        if (ptr == NULL)                        // First node in list
            temp->next = NULL;
        else
            temp->next = ptr;                   // Chain to list (at head of list)
        return temp;                            // return as the new head of the list
    }
}
```

Question

How would we define the structure that stores students in a linked list?

- Assume student has:
  - Name
  - Age
  - GPA

Part 2

Sequential Text Files

Sequential Files

Letter.txt

Dear Mom,
Please send money.
Love Bob.

LOGICAL VIEW

Files on disk are actually linear structures like 1D arrays but without cell index numbers.

Start address

D	e	a	r	/r	/n	/t	P	….	EOF

End of file character

ACTUAL PHYSICAL VIEW

STDIO.H

The stdio.h library has file commands:

- fopen - to access the file from the start address
- fclose - to terminate file access
- fgetc - to read a single character from the file
- fgets - to read one entire line from a file
- fputc - to write a single character to the file
- fputs - to write one entire line to the file
- fscanf - to read a formatted line from the file
- fprintf - to write a formatted line to the file
- feof - end of the file test

The get, put and printf commands work much like their console and stream versions

fopen

To access a file.

Syntax:

- FILE *fopen(FILENAME, MODE);

Where:

- FILE - a built-in pointer type to reference a file
  - On success returns a pointer to the file
  - On failure returns a NULL pointer
- FILENAME - a Unix path/filename descriptor as a string
- MODE:
  - rt - read from text file (file must exist)
  - wt - write to text file (if file exists, overwrites)
  - at - append to text file (if file exists it appends, or creates file)

Example

```
#include <stdio.h>
#include <stdlib.h>
void displayFile (FILE *p) {
    char c;
    while(!feof(p)) {              // while not end of file p
        c = fgetc(p);              // read one character into c
        putc(c);                   // print c to the screen
    }
}
void main() {
    FILE *q = fopen("letter.txt","rt");
    if (q == NULL) exit(1);        // terminate with an error code
    displayFile(q);
    fclose(q);
}
```

Example

```
#include <stdio.h>
#include <stdlib.h>
void copyFile (FILE *source, FILE *destination) {
    char c;
    while(!feof(source)) {
        c = fgetc(source);
        fputc(c, destination);
    }
}
void main() {
    FILE *s = fopen("letter.txt","rt"), *d = fopen("copy.txt","wt");
    if (s == NULL || d == NULL) exit(1); // terminate with an error code
    copyFile(s, d);
    fclose(s); fclose(d);
}
```

Example

```
#include <stdio.h>
#include <stdlib.h>  #include<string.h>  // cannot define beside in real life…
void copySkipWord (FILE *source, FILE *destination, char *word) {
    char array[1000];                       // must assume a max size…
    while(!feof(source)) {
        fgets(array,999,source);            // 999 since fgets inserts a \0 at the end
        if (strstr(array, word) == 0)       // the word is not in the array
            fputs(array, destination);
    }
}
void main() {
    FILE *s = fopen("letter.txt","rt"), *d = fopen("copy.txt","wt");
    if (s == NULL || d == NULL) exit(1);   // terminate with an error code
    copySkipWord(s, d, "bob");
    fclose(s); fclose(d);
}
```

Important

End of file issue: in the previous example the last line of the file would be repeated twice. Why?

This is the correct way to do it:

```
fgets(array,999,source);
while(!feof(source)) {
        if (strstr(array, word) == 0) // the word is not in the array
                fputs(array, destination);
        fgets(array,999,source);
    }
}
```

The CSV File

- CSV = Comma Separated Vector
- Is a text file with a specific format
- Format:

```
Data1,data2,data3,data4
Data5,data6,data5,data8
```

- Notice that the file is composed of units of data that are separated from each other by commas or CR/LF

The CSV File

- Each row is called a “record”
  - It stores data about a common artifact
- Each unit of data is called a “field”
  - It stores a single fact related to the artifact

```
Mary,Smith,18,3.7
Tom,Bombadil,98,4.0
```

- The above example stores information about students: first name, last name, age, and gpa.

Example

How can we read each record?

How can we parse each record into its fields?

Part 3

Test 2