Unix
Bash
C
GNU
Systems

Software Systems

Lectures Week 8

Basic Software Engineering Techniques

(and Dynamic Memory in C)

Prof. Joseph Vybihal

Computer Science

McGill University

Week 8 Lecture 1

Dynamic Memory in C

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 does not know…

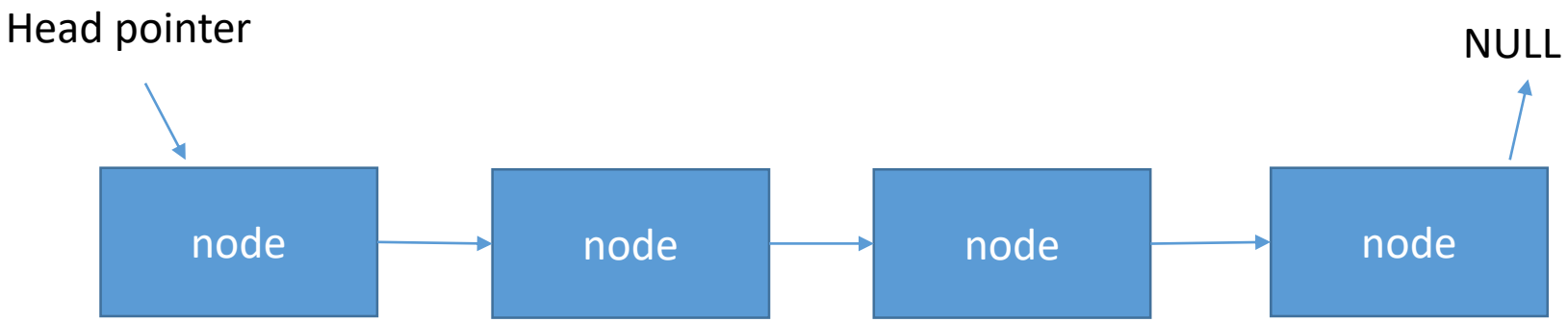
Eg:

- 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

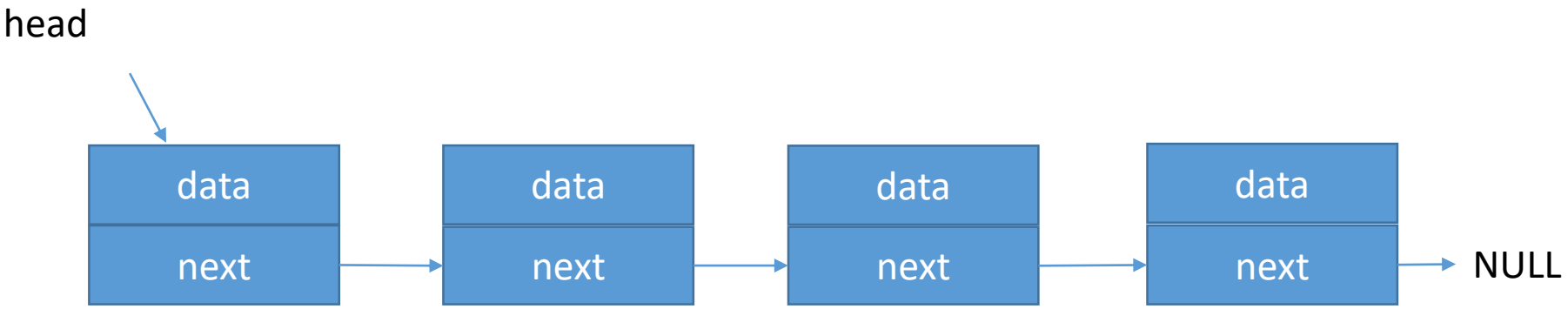
```
struct NODE {
    int data;                 // the data we want to store in the node
    struct NODE *next;        // the pointer to the following node in the list
};

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
```



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.
}
void 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
    }
}
```

Week 8 Lecture 2

Modular Programming in C

Readings: chapter 4

Modular Programming

A program composed from multiple files.

Two types exist:

- Single Source Multiple Text File programming
- Multiple Source Text File programming
  - Also known as: modular programming

Why is this useful?

- Creating your own libraries
- Working in a team of programmers

Single Source Multiple Text File

We create our program using many .c files, but we merge them at compile-time into a single document.

Bash-prompt $ vi file1.c

Bash-prompt $ vi file2.c

Bash-prompt $ vi file3.c

Bash-prompt $ gcc file1.c

Bash-prompt $ ./a.out

Notice:

- Three .c files are created, but
- We only compile the first source file
- There exists a command in the first source file that merges the other source files into itself

Benefits:
- More than one person can work on project.

Drawback:
- Compiling entire project each time.

#include

The #include directive has two formats:

- #include<library>
  - Merge a library into your program
- #include “path/textfile”
  - Merge a text file into your program

```
#include “file2.c”                  // loads source file from current directory

#include “/user/jack/file2.c”       // loads from specified directory
```

Example

Linkedlist.c

```
#include<stdio.h>
void printList(struct NODE *ptr) {
        .....
}
```

MyMax.c

```
#include<stdlib.h>
int max(int a, int b, int c) {
    int aMax = a;
    aMax = (b>aMax) ? b : aMax;
    aMax = (c>aMax) ? c : aMax;
    return aMax;
}
```

```
#include<stdio.h>
#include “Linkedlist.c”
#include “MyMax.c”

int main() {
   int a,b,c,result;
   scanf(“%d %d %d”, &a, &b, &c);
   result = max(a,b,c);
   printf(“%d\n”, result);
}
```

Main.c

```
Bash-prompt $ vi Linkedlist.c
Bash-prompt $ vi MyMax.c
Bash-prompt $ vi Main.c
Bash-prompt $ gcc Main.c
Bash-prompt $ ./a.out
```

Multiple Source Text File

We create our program using many .c files.

We compile each source file separately, and merge later.

Notice:

- The gcc –c command compiles each source file separately saving each .c file as a compiled .o file. These .o files cannot execute on their own.
- The gcc command by itself merges all the .o files into an a.out executable file.

```
Bash-prompt $ vi file1.c
Bash-prompt $ vi file2.c
Bash-prompt $ vi file3.c
Bash-prompt $ gcc –c file1.c file2.c file3.c
Bash-prompt $ gcc file1.o file2.o file3.o
Bash-prompt $ ./a.out
```

Benefits:
- More than one person can work on project.
- Only changed files needs to be compiled.
- Each file has its own named space.

Drawback:
- Compiling is more complicated.

Named Space

- A semi-private space.
  - Each .o file's global variables are, by default, local to only the .o file it was defined within.
  - This is not true for functions. All functions are public.
- Using the extern command a semi-private variable can be accessed publicly.

File1.c

```
int x;
void printX() {
   printf("%d\n", x);
}
```

File2.c

```
int main() {
   printf("%d\n", x);          // error
   printX();                   // legal
}
```

```
Bash-prompt $ gcc –c File1.c File2.c
Bash-prompt $ gcc File1.o File2.o
Bash-prompt $ ./a.out
```

Named Space

- Using the extern command a semi-private variable can be accessed publicly.

File1.c

```
int x;
void printX() {
   printf("%d\n", x);
}
```

File2.c

```
extern int x;
int main() {
   printf("%d\n", x);          // legal
   printX();                   // legal
}
```

```
Bash-prompt $ gcc –c File1.c File2.c
Bash-prompt $ gcc File1.o File2.o
Bash-prompt $ ./a.out
```

Example

Linkedlist.c

```
#include<stdio.h>
struct NODE *head;
void printList(struct NODE *ptr) {
            .....
}
```

MyMax.c

```
#include<stdlib.h>
int max(int a, int b, int c) {
    int aMax = a;
    aMax = (b>aMax) ? b : aMax;
    aMax = (c>aMax) ? c : aMax;
    return aMax;
}
```

Main.c

```
#include<stdio.h>
extern struct NODE *head;

int main() {
   int a,b,c,result;
   scanf("%d %d %d", &a, &b, &c);
   result = max(a,b,c);
   printf("%d\n", result);
}
```

```
Bash-prompt $ vi Linkedlist.c
Bash-prompt $ vi MyMax.c
Bash-prompt $ vi Main.c
Bash-prompt $ gcc –c Main.c Linkedlist.c MyMax.c
Bash-prompt $ gcc Main.o Linkedlist.o MyMax.o
Bash-prompt $ ./a.out
```

The .h File

In the last example we saw:

extern struct NODE *head;

The above command assumes that the definition for NODE exists in both the Linkedlist.c and Main.c files.

- This can be done in two ways:
  - Method 1: write the NODE definition in both files
  - Method 2: write the NODE definition in one file and share
    - The .h file uses Method 2

Example

Node.h

```
struct NODE {
   int data;
   struct NODE *next;
};
```

Linkedlist.c

```
#include<stdio.h>
#include “Node.h”

struct NODE *head;
void printList(struct NODE *ptr) {
            …..
}
```

Main.c

```
#include<stdio.h>
#include “Node.h”

extern struct NODE *head;

int main() {
   int a,b,c,result;
   scanf(“%d %d %d”, &a, &b, &c);
   result = max(a,b,c);
   printf(“%d\n”, result);
}
```

This is considered better form because you avoid duplication errors by writing the definition once.

Problem

Abigail, Bethany, and Sebastian want to work together on a telephone book program. A text file contains the names and numbers in the book (assume one name and number per line). The size of the text file is unknown. A user wants to enter the name and see the number.

How can they work together to create this C program (given what we have covered)?

The Pre-processor

Before gcc compiles your code it will pre-process it.

- This means it will make changes to your source file before it compiles it.

Common pre-processor commands used by developers:

- #include ← we have seen this already
- #define
- #ifdef
- #ifndef

#define

A technique by which a developer can define new terms or commands that are not part of the C language.

Syntax:

- #define TERM EXPRESSION
  - The EXPRESSION will be inserted into the source code everywhere it finds TERM

Examples:

- #define TRUE 1
- #define FALSE 0
- if (x == TRUE) { … }

Using #define as a macro

A variable #define command

Syntax:

- #define TERM(ARG1, ARG2) EXPRESSION
  - The EXPRESSION will be inserted into the source code everywhere it finds TERM
  - ARG1 and ARG2 will be inserted into the EXPRESSION

Example:

- #define MAX(A,B) (A>B)?A:B
- int result = MAX(5,10);

#ifdef and #ifndef

The pre-processor if-statement

Syntax:

- #ifdef TERM
  - If TERM has been previously #define'd then true, else false.
- #ifndef TERM
  - If TERM has not been previously #define'd then true, else false.

General syntax:

```
#ifdef TERM
    // multiple C programming statements go here
#else
    // optional #else part, multiple C programming statements go here
#endif
```

Optional Compiling Example

```
#define FRENCH
// #define ENGLISH
int main() {
    int age;
    #ifdef FRENCH
            printf("Votre age: ");
    #endif
    #ifdef ENGLISH
            printf("Your age: ");
    #endif
    scanf("%d", &age);
    :
}
```

This actually compiles your program into a French version without any English.

Instead of keeping two versions of your program you can keep one version and just compile into the language you want.

Crash proof .h file example

Node.h

```
#ifndef NODEH
#define NODEH
 struct NODE {
    int data;
    struct NODE *name;
 };
```

Often the developer who uses a library does not know all the inner workings of the library (or shared file) and can easily cause a situation where they include a file twice. #ifndef can help.

Lib.h

```
#include “Node.h”
 void addNode(struct NODE * ptr) {
                :
                :
 }
```

```
#include <stdio.h>
#include “Node.h”
#include “Lib.h”

 void main() {
      :
      :
 }
```

Week 8 Lecture 3

GNU Tools

Readings: chapter 4, http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/ and https://sourceware.org/binutils/docs/gprof/

What is GNU?

GNU = Gnu is Not Unix

- Open source project to create a Unix-like but free environment
- It contains a Unix-like operating system: HURD kernel
- It contains many development tools

In the next two lectures we will look at development tools

- Some based on GNU
- Others not based on GNU

The make Tool

Modular programming:

- Great for team programming
- Fast way to compile large projects
- But…. Complicated to compile

The GNU make Tool helps us compile complicated projects.

The make Tool can also do other software engineering activities, like backups, etc.

The make Tool

Composed of:

- The make program
- The makefile text file

Developers define their software engineering rules within the text file named makefile.

The make program uses that text file.

If your favorite IDE uses the term Project, then inside somewhere is a makefile and a make program.

Basic makefile Text File

```
program: file1.o file2.o
  gcc –o program file1.o file2.o
file1.o: file1.c
  gcc –c file1.c
file2.o: file2.c
  gcc –c file2.c
backup:
  cp *.c /home/project/backup
```

This means:

- gcc –c file1.c file2.c
- gcc –o program file1.o file2.o

It can also mean:

- cp *.c /home/project/backup

Some power exists in selectively executing portions of the makefile.

Basic makefile Text File

```
program: file1.o file2.o
  gcc –o program file1.o file2.o
file1.o: file1.c
  gcc –c file1.c
file2.o: file2.c
  gcc –c file2.c
backup:
  cp *.c /home/project/backup
```

Method 1:

- Bash-prompt $ make

This invokes the make program.

It assumes a makefile exists in the same directory and executes the first line. In our example: program: file1.o file2.o

Notice the recursive definition.

Basic makefile Text File

```
program: file1.o file2.o
  gcc –o program file1.o file2.o
file1.o: file1.c
  gcc –c file1.c
file2.o: file2.c
  gcc –c file2.c
backup:
  cp *.c /home/project/backup
```

Method 2:

- Bash-prompt $ make backup

This invokes the make program and specifically targets a portion of the makefile.

It assumes a makefile exists in the same directory and in our example executes: cp *.c /home/project/backup

Notice it executes like Bash.

gprof
Optimizing for Speed

Unacceptably Slow

- An application that is unacceptably slow is an application that takes longer time to complete an operation than desired.
- Thus, the application needs to be optimized.
  - The code of the application needs to be improved so the application can accomplish the same tasks with less resources.
  - The behaviour of the application should remain unchanged.
- Before optimizing, we need to identify the portion of code that is slow.

Definitions of slow

Big Oh

- Logical speed of the algorithm

Clock Time

- Actual speed as related to computer hardware MHz, RAM, Devices

Advantages and Problems

- Big Oh:
  - Advantage: true speed independent from hardware
  - Problems: long to do on large pieces of code
- Clock:
  - Advantages: can be automated
  - Problems: get better CPU, program runs faster

Profilers

- Profilers are dynamic performance analysis tools
  - As opposed to lint which are static analysis tools.
- They record data as the application executes
- This data is often used to determine which part of an application needs optimization.

Using GPROF

$ gcc –pg file.c
(creates a gmon.out binary file)

$ gprof –b a.out gmon.out > textfile

$ gprof -b a.out gmon.out < input > output

Options:

-b not verbose

-s merge all gmon files

-z table of functions never called

The Flat Profile

Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
38.88	24.20	24.20	6	4033.33	4033.33	array_type_organpipe
38.28	48.03	23.83	6	3971.67	3972.64	array_type_random
15.20	57.49	9.46	12	788.33	1181.16	qsort
7.44	62.12	4.63	1598	2.90	2.92	choose_pivot
0.06	62.16	0.04	918	0.04	0.04	write
0.06	62.20	0.04				mcount
0.03	62.22	0.02	11715	0.00	0.00	swap_elem
.						
.						
.						
0.00	62.25	0.00	1	0.00	0.00	sigvec

- % overall
- order of execution (generally)
- how long fn exec in total
- then # of times it was called in total
- length of exec for fn alone. 4.033 sec
- length of exec for fn plus children
- fn name

how long fn exec in total / 62.25 sec = 24.2 sec / 62.25 sec = 38.88%

length of exec for fn alone. 4.033 sec * 6 = 24.2 sec.

% time
This is the percentage of the total execution time your program spent in this function. These should all add up to 100%.

cumulative seconds
This is the cumulative total number of seconds the computer spent executing this functions, plus the time spent in all the functions above this one in this table.

self seconds
This is the number of seconds accounted for by this function alone. The flat profile listing is sorted first by this number.

calls
This is the total number of times the function was called, blank otherwise.

self ms/call
This represents the average number of milliseconds spent in this function per call.

total ms/call
Average number of milliseconds spent in this function and its descendants per call.

name
This is the name of the function. The flat profile is sorted by this field alphabetically after the self seconds field is sorted.

The Call Graph


```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.00    0.92                 main [1]
                0.06    0.85       1/1           test [2]
                0.01    0.00       1/1           some_other_test [5]
-----------------------------------------------
                0.06    0.85       1/1           main [1]
[2]     98.9    0.06    0.85       1         test [2]
                0.00    0.85       1/1           another_test [3]
-----------------------------------------------
                0.00    0.85       1/1           test [2]
[3]     92.3    0.00    0.85       1         another_test [3]
                0.85    0.00       1/1           yet_another_test [4]
-----------------------------------------------
                0.85    0.00       1/1           another_test [3]
[4]     92.3    0.85    0.00       1         yet_another_test [4]
-----------------------------------------------
                0.01    0.00       1/1           main [1]
[5]      1.1    0.01    0.00       1         some_other_test [5]
-----------------------------------------------
```


```
index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00    0.05                 start [1]
                0.00    0.05       1/1           main [2]
                0.00    0.00       1/2           on_exit [28]
                0.00    0.00       1/1           exit [59]
-----------------------------------------------
                0.00    0.05       1/1           start [1]
[2]    100.0    0.00    0.05       1           main [2]
                0.00    0.05       1/1           report [3]
-----------------------------------------------
                0.00    0.05       1/1           main [2]
[3]    100.0    0.00    0.05       1           report [3]
                0.00    0.03       8/8           timelocal [6]
                0.00    0.01       1/1           print [9]
                0.00    0.01       9/9           fgets [12]
                0.00    0.00      12/34          strncmp <cycle 1> [40]
                0.00    0.00       8/8           lookup [20]
                0.00    0.00       1/1           fopen [21]
                0.00    0.00       8/8           chewtime [24]
                0.00    0.00       8/16          skipspace [44]
-----------------------------------------------
[4]     59.8    0.01        0.02       8+472     <cycle 2 as a whole> [4]
                0.01        0.02     244+260         offtime <cycle 2> [7]
                0.00        0.00     236+1           tzset <cycle 2> [26]
-----------------------------------------------
```

The Primary Line

The line that describes the function which the entry is about.

```
index  % time    self  children called     name
[3]      100.0    0.00    0.05       1         report [3]
```

index
Entries are numbered with consecutive integers, for cross-referencing.

% time
This is the percentage of the total time that was spent in this function, including time spent in subroutines called from this function. Cannot be totalled.

self
This is the total amount of time spent in this function. This should be identical to the number printed in the seconds field for this function in the flat profile.

children
This is the total amount of time spent in calls made by this function.

called
This is the number of times the function was called.

Function's Callers

A function's entry has a line for each function it was called by.

```
index  % time    self  children called     name
...
                0.00    0.05       1/1          main [2]
[3]      100.0    0.00    0.05       1        report [3]
```

self
An estimate of the amount of time spent when it was called from main.

children
An estimate of the amount of time spent in subroutines when called from main.

called
Two numbers: the number of times was called from main, followed by the total number of non-recursive calls from all its callers.

Function's Children

A line for each other function that it called.

```
index  % time    self  children called     name
[2]      100.0    0.00    0.05       1         main [2]
                  0.00    0.05       1/1           report [3]
```

self
An estimate of the amount of time spent directly when called from main.

children
An estimate of the amount of time spent in subroutines of report when report was called from main.

called
Two numbers, the number of calls to report from main followed by the total number of non-recursive calls to report.