1 **Introduction to Computer Science**

W 1113 – Lab (C)

Lab1

Suhit Gupta
1/29/04

2 **Rules**

- I will call on you
- You will learn a lot, I can assure you
  - Do the reading
  - Attendance is VERY important, pseudo mandatory
- Email me if you have any questions
- I am going to teach this as if it were a small group
  - You and I will get to know each other
  - Interrupt me, don't let me proceed unless you understand everything
  - Speak loudly
- No sleeping in the lab ☺
- Turn cell phones off

3 **Books**

- The two books I will use (yeah I know you weren't assigned them both)

4 **Introduction to C**

- Created by Dennis Ritchie in 1972
- Kernighan and Ritchie, wrote the canonical book

5 **Compile and Run**

- Basic compile and run
  - gcc <filename.c>
  - Therefore to run…
- Advanced options
  - gcc <filename.c> -o blah
  - Therefore to run…
- Makefile (and make)
  - What is it?

6 **Structure of program**

```
#include <stdio.h>
int main (void) {
    printf("Hello World!\n");
    return 0;
}
```

7 **Structure of program**

```
#include <stdio.h>
int main (void) {
    printf("Hello World!\n");
    return 0;
}
```

- Pre-processing directive
- Angle brackets mean that the file is found in the usual place

8 **Structure of program**

```
#include <stdio.h>
int main (void) {
    printf(“Hello World!\n”);
    return 0;
}
```

- Main function
- (void)
- Int over here means…
- {

9 Structure of program

```
#include <stdio.h>
int main (void) {
    printf(“Hello World!\n”);
    return 0;
}
```

- printf
- Hello World
- “…”
- ;

10 Structure of program

```
#include <stdio.h>
int main (void) {
    printf(“Hello World!\n”);
    return 0;
}
```

- Return
- 0

11 Structure of program

```
#include <stdio.h>
int main (void) {
    printf(“Hello World!\n”);
    return 0;
}
```

- End of program or the function

12 Comments

- //
- /* … */

13 Variables

```
#include <stdio.h>

int main (void) {
    int inches, feet, fathoms;

    fathoms = 7;
    feet = 6 * fathoms;
    inches = 12 * feet;
    printf(“Wreck of the Hesperus:\n”);
    printf(“Its depth at sea in different units:\n);
    printf(“ %d fathoms\n”, fathoms);
    printf(“ %d feet\n”, feet);
    printf(“ %d inches\n”, inches);
```

```
        return 0;
}
```

14 Variables II

```
#include <stdio.h>

int main (void) {
        char c;

        c = 'A';
        printf(" %c rocks\n", c);
        return 0;
}
```

15 Variables III

- Declare at the beginning of the program
- Name them intelligently
- Remember to assign values

16 I/O - output

- printf
- Special constructs like \n and \t
  - Also use \ to ignore next character (\\, \')
- %d, %c, etc.

17 Data types

- int
- char
- float
- string – next time

18 Miscellaneous

- #include <…>
- #include "filename"
- #define
  - Anywhere in the program

19 Assignment

- Type into cunix
  - man gcc
- Read Ch. 1-4 of *Practical C Programming*

1 **Introduction to Computer Science**

W 1113 – Lab (C)

Lab2

Suhit Gupta
2/5/04

2 ## Questions about the previous lab

3 ## Questions about HW1 (or HW0)

4 ## Who did man gcc?

- Tell me something interesting about it…

5 ## Recap

- Intro to Unix, Hardware, Server-Client relationships, concept behind telnet
- Intro to C
- Basic structure of a program
- Compiling and running programs
- Variables, and assigning values to them
- Data types and I/O
- \

6 **I/O**

Output in more detail

- printf("%s %c %f %c%c\n", "one", 2, 3.33, 'G', 'o');
- %3c – field width
- %7.2f
- HW1?

7 **I/O**

Input

- scanf – analogous to printf
- scanf("%d", &x);
- You can scan in different types of data from files, user input or command line parameters.

8 ## Conversion between data types

- atoi
- atof
- atol

- Usage -> a = atoi(b)
  - Here the value of b is converted from string to integer.

9 ## Command line parameters

- argv & argc
- ./a.out 2 3 (to add two numbers)

- int argc, char *argv[]

```
#include <stdio.h>

int main (int argc, char *argv[]) {
        int a, b;
        a = atoi(argv[1]);
        b = atoi(argv[2]);
        … //do things with a and b
}
```

10 Math operators

- +, -, *, /
- &
- |

11 Arrays

- What are arrays?

12 Method calls

- What are methods?

13 Assignment

- Read Ch. 5 and start Ch. 6 from the Practical C Programming book
- Read pg. 200-206 from the Practical C Programming book
- man gcc

- **HW1**

1 **Introduction to Computer Science**

W 1113 – Lab (C)

Lab3

Suhit Gupta
2/12/04

2 **Questions about the previous lab**

3 **Questions about HW1 (or HW0)**

4 **HW1 submit instructions**

5 **Recap from Lab 1**

- Intro to Unix, Hardware, Server-Client relationships, concept behind telnet
- Intro to C
- Basic structure of a program
- Compiling and running programs
- Variables, and assigning values to them
- Data types and I/O
- \

6 **Recap from Lab 2**

- Details on printf
- Details on scanf
- Conversion between data types
- Math operators
- Command Line Parameters

7 **Math ops continued**

- +, -, *, /, %
- ++, --
- +=, -=, *=, /=

8 **Other symbols**

- <, >, <<, >>,
- !, !=
- &, &&, |, ||
- #
- (), {}, []

9 **Arrays**

- What are arrays?
  - Arrays are sets of consecutive memory locations used to store data
- Typical array declaration
  - int data_list[3];
  - data_list[0], data_list[1], data_list[2]
  - Dimensionality
  - What is the index?
  - You can also initialize by doing the following
    - int data_list[3] = {1.0, 2.0, 3.0);

10 Code sample

```
#include <stdio.h>
#define N 5

int main (void) {
        float a[N], total, average;

        a[0] = 34.0;
        a[1] = 27.0;
        a[2] = 45.0;
        a[3] = 82.0;
        a[4] = 22.0;

        total = a[0] + a[1] + a[2] + a[3] + a[4];
        average = total/5.0;
        printf("Total is %f and Average is %f\n", total, average);
        return(0);
}

//run array.c
```

11 Multidimensional arrays

- int matrix [2][3];
- Now you assign and reference by saying
  - matrix [0][0];
  - matrix [0][1];
  - matrix [0][2];
  - matrix [1][0];
  - matrix [1][1];
  - matrix [1][2];

12 Strings

- Sequence of chars (an array of characters)

```
#include <stdio.h>

int main (void) {
        char name[6];

        name = "Suhit";

        printf("My name is %s\n", name);
        return(0);
}
```

13 Strings

- Sequence of chars (an array of characters)

```
#include <stdio.h>

int main (void) {
        char name[6];

        name = "Suhit";            // This is wrong

        printf("My name is %s\n", name);
        return(0);
}
```

14 Strings II

```
#include <stdio.h>

int main (void) {
        char name[6];

        name[0] = 'S';
        name[1] = 'u';
        name[2] = 'h';
        name[3] = 'i';
        name[4] = 't';
        name[5] = '\0';            //adding a null character at the end of the string
```

```
	printf("My name is %s\n", name);
	return(0);
}
```

15 Strings III

- #include <string.h>
  - to include special string manipulation thingies
    - strcpy
    - strcmp
    - strlen
    - strcat
    - strtok

16 Strings IV

```
#include <stdio.h>
#include <string.h>

int main (void) {
	char name[6];
	//one character at the end is stored for null
	strcpy(name, "Suhit");

	printf("My name is %s\n", name);
	return(0);
}
```

17 Strings V

```
#include <stdio.h>
#include <string.h>

int main (void) {
	char name[60];
	/* last character is still reserved for null, store at most 59 characters */
	strcpy(name, "Suhit");

	printf("My name is %s\n", name);
	return(0);
}
```

18 Strings VI

```
#include <stdio.h>
#include <string.h>

char first_name [100];
char last_name [100];
char full_name [200];

int main (void) {
	strcpy(first_name, "Suhit");
	strcpy(last_name, "Gupta");

	strcpy(fullname, first_name);
	strcat(fullname, " ");
	strcat(fullname, last_name);

	printf("My full name is %s\n", full_name);
	return(0);
}

//run strings.c
```

19 Strings VII – Reading Strings

- fgets(name, sizeof(name), stdin);
  - *name* is the name of the character array
  - sizeof tells the program how much to read
  - stdin – keyboard

```
#include <stdio.h>
#include <string.h>

char line [100];
```

```
int main () {
       printf("Enter a line: ");;
           fgets(line, sizeof(line), stdin);

       printf("The length of the line is %d\n", strlen(line));
       return(0);
}

//Run strings2.c
```

20 Strings VIII

- fgets has last character as end-of-line (newline)
- Some people will munge the last newline char by doing the following
  - line[strlen(line)-1)] = '\0'
- Then use sscanf – like scanf, but used to scan strings
  - Usage : sscanf(name, format, &var1, &var2, …);
  - Why not use atoi?
    - Because you scan in different types of values and format them into different types of vals.
    - sscanf(in_string, "%d%d%d%s", &a, &b, &c, tmp);

21 BTW…

- In Ch. 5, read about different data types, like different types of int, types of float.
- Also read about hexadecimal and octal
- We will cover this in depth as the course goes on

22 Loops and conditionals

- if
  - need to know <, >, ==, !=
  - usage: *if (expr) {stmt…}*
    else if (expr) {stmt…}
    else {stmt}
- while
  - usage: *while (cond) {stmt…}*
  - break;

23 Next time…

- Iteration/loops
  - While
  - For
  - Do while
- Conditional statements
  - If
  - Switch
- Methods and method calls
  - Variable scope
  - Return values

24 Assignment

- Read Ch. 6 from the Practical C Programming book

- **HW1**

1 **Introduction to Computer Science**

W 1113 – Lab (C)

Lab4

Suhit Gupta
2/19/04

2 ## Questions about the previous lab

3 **Questions about HW2 (or HW1 or HW0)**

- Or submit instructions?

4 ## Recap from Lab 2

- Details on printf
- Details on scanf
- Conversion between data types
- Math operators
- Command Line Parameters

5 ## Recap from Lab 3

- Math operators
- Arrays (assignment and reference)
- Strings
  - string manipulation
  - fgets
  - sscanf

6 ## Quick quiz…

- BTW, I will be asking one (or two) questions every class that are in the reading only… brownie points ☺

- +=, -=, *=, /=
  - What do these do?

7 ## Function prototypes

- Usually, you declare variables before you can use them
  - similar with functions
  - however, you can
    - declare a function prototype at the beginning of the program
    - define the actual function workings later on
- Example
  - int add (int a, int b);
- This will be important in HW2

8 ## Function prototypes - code

```
#include <stdio.h>

int add (int a, int b);

int main() {
        int c;
        c=add(2, 3);
        printf("The total of 2 and 3 is %d\n", c);
}

int add (int first_number, int second_number) {
```

```
	int total;

	total = first_number + second_number;
	return total;
}
```

9 Function prototypes – code II

```
#include <stdio.h>

int add (int a, int b);

int main(int argc, char *argv[]) {
	int c, x, y;
	x=atoi(argv[1]);
	y=atoi(argv[2]);
	c=add(x, y);
	printf("The total of %d and %d is %d\n", x, y, c);
}

int add (int first_number, int second_number) {
	int total;

	total = first_number + second_number;
	return total;
}
```

10 BTW (a couple of comments about comments and style)

- Use comments
- Use tabs to write code cleanly
- Identify yourself as the author
- Placement of {}

11 Conditionals

- Conditional statements
  - if
  - switch

12 Conditionals

- Conditional statements
  - if
    - need to know <, >, ==, !=
    - usage: *if (expr) {stmt…}*
      else if (expr) {stmt…}
      else {stmt}
    - when do you not need {}
  - if followed by another if
    if (something) do something;
    if (something else) do something else;
  - The default case is the final *else*
  - Correctness
    - if (strcmp(string1, string2)) do something?
    - if (strcmp(string1, string2)==0) do something?

13 Conditionals II

- Switch

```
switch (val) {
	case 1:
	do some work;
	break;
	case 2:
	do some work;			// you don't have to necessarily have
	break;				// stuff here
	case 3:
	do some work;
	break;
	default:			//if needed
	do some work;
	break;
}
```

- What is the break statement?
- What happens if you don't use break?

14 Goto and the evils of it…

- DON'T USE GOTO
- What is GOTO

- Why is it a problem?

15 **Loops**

- Iteration/loops
  - While
  - For
  - Do while
- Difference between conditionals and loops

16 **Loops II**

- While
  - usage:
    - *while (cond) {stmt…}*
  - break;
  - continue;
- code

```
while(current_number<100) {
    do something;                    //what is wrong
}
```

17 **Loops II**

- While
  - usage:
    - *while (cond) {stmt…}*
  - break;
  - continue;
- code

```
while(current_number<100) {
    do something;                    //what is wrong
    i++; // or i-- as the case may be
}
```

18 **Loops III**

- Do while
  - usage:

```
do {
    blah;
} while (i>0);
```

  - Again, remember that the value of ‘i’ needs to be changed

19 **Loops IV**

- For
  - usage:
    - for (… ; … ; …) {
      do something here;
      }
  - There is other acceptable syntax (sort of)
- BTW, this is where the ++i and i++ becomes relevant and useful
- Everything in for can be done in a while
  - Think about it

20 **Loops V**

- The comma operator
  - Things are evaluated from left to right
- for (sum=0, i=1; i<=n; ++i)
  sum += i;
- for (sum=0, i=1; i<=n; sum += i, ++i)
  ;
- for (sum=0, i=1; i<=n; ++i, sum += i)

```
;          // this may give wrong results as i is
           // incremented before added to sum
```

21 Loops VI

- Why can we use the ; just like that
- Infinite loops – beware
  - while (1) { …}
  - for (; ;) {…}
    - Use it at your own risk (system administrator may kill ;-))
    - Use it instead of running your program again and again

22 What does the following do?

```
for (i = 1; i <= 10; ++i )
        ;
sum += i;
```

23 Assignment

- Read Ch. 6 from the Practical C Programming book

- **HW2**
  - Don’t wait till the last minute, *seriously.*

1 **Introduction to Computer Science**

W 1113 – Lab (C)

Lab5

Suhit Gupta
2/26/04

2 Questions about the previous lab

3 Questions about HW2

4 Recap from Lab 3

- Math operators
- Arrays (assignment and reference)
- Strings
  - string manipulation
  - fgets
  - sscanf

5 Recap from Lab 4

- Function prototypes
- Conditional statements
  - if
  - switch
- Loops
  - while
  - do while
  - for

6 Quick quiz…

- What does the following do in a for loop
  - && or ||
- What are double and long?

7 Function prototypes revisited

- Usually, you declare variables before you can use them
  - similar with functions
  - however, you can
    - declare a function prototype at the beginning of the program
    - define the actual function workings later on
- Example
  - int add (int a, int b);
- This is important in HW2

8 Function prototypes – code I

```
#include <stdio.h>

int add (int first_number, int second_number) {
        int total;

        total = first_number + second_number;
        return total;
}

int main(int argc, char *argv[]) {
        int c, x, y;
        x=atoi(argv[1]);
```

```
        y=atoi(argv[2]);
        c=add(x, y);
        printf("The total of %d and %d is %d\n", x, y, c);
}
```

9 Function prototypes – code II

```
#include <stdio.h>

int add (int a, int b);

int main(int argc, char *argv[]) {
        int c, x, y;
        x=atoi(argv[1]);
        y=atoi(argv[2]);
        c=add(x, y);
        printf("The total of %d and %d is %d\n", x, y, c);
}

int add (int first_number, int second_number) {
        int total;

        total = first_number + second_number;
        return total;
}
```

10 Some more examples

```
#include <stdio.h>

//defining all my function prototypes
int add (int a, int b);
int minus (int a, int b);
int mult (int a, int b);
float div (int a, int b);

int main(int argc, char *argv[]) {
        //defining all my variables
        int addanswer, minusanswer, multanswer, x, y;
        float divanswer;
        //reading in all the input
        x=atoi(argv[1]);
        y=atoi(argv[2]);
        //performing calculations and printing the result
        addanswer=add(x, y);
        minusanswer=minus(x,y);
        multanswer=mult(x,y);
        divanswer=div(x,y);
        printf("The respective calculations of %d and %d are %d, %d, %d and %f\n", x, y, addanswer,
minusanswer, multanswer, divanswer);
}

//The add function
int add (int first_number, int second_number) {
        int total;

        total = first_number + second_number;
        return total;
}

//The subtraction function
int minus (int first_number, int second_number) {
        int total;

        total = first_number - second_number;
        return total;
}

//The multiplication function
int mult (int first_number, int second_number) {
        int total;

        total = first_number * second_number;
        return total;
}

//The division function - note that this one returns a float
float div (int first_number, int second_number) {
        float total;

        total = (float) first_number / (float) second_number;
        return total;
}
```

11 Here is a problem – use functions

- Brainstorming (real world example)
  - Planning your trip to Europe
  - Changing currency during your Eurotrip
  - Booking Flights
  - Booking Hotel Room and/or Youth Hostels
  - Sightseeing
  - Look up the weather
- What are the different methods?

12 Conditionals revisited

- Conditional statements
  - if
  - switch

13 Conditionals

- Conditional statements
  - if
    - need to know <, >, ==, !=, <=, >=
    - &&, ||
    - usage: *if (expr) {stmt…}*
else if (expr) {stmt…}
else {stmt}
    - when do you not need {}
  - if followed by another if
if (something) do something;
if (something else) do something else;
  - The default case is the final *else*
  - Correctness
    - if (strcmp(string1, string2)) do something?

- if (strcmp(string1, string2)==0) do something?

14 ## Conditionals II

- Switch

```
switch (val) {
    case 1:
    do some work;
    break;
    case 2:
    do some work;                  // you don't have to necessarily have
    break;                         // stuff here
    case 3:
    do some work;
    break;
    default:                       //if needed
    do some work;
    break;
}
```

- What is the break statement?
- What happens if you don't use break?

15 ## Loops

- Iteration/loops
  - While
  - For
  - Do while
- Difference between conditionals and loops

16 ## Loops II

- While
  - usage:
    - *while (cond) {stmt…}*
  - break;
  - continue;
- code

```
while(current_number<100) {
    do something;                  //what is wrong
}
```

17 ## Loops II

- While
  - usage:
    - *while (cond) {stmt…}*
  - break;
  - continue;
- code

```
while(current_number<100) {
    do something;                  //what is wrong
    i++; // or i-- as the case may be
}
```

18 ## Loops III

- Do while
  - usage:

```
do {
    blah;
} while (i>0);
```

  - Again, remember that the value of 'i' needs to be changed

19 ## Loops IV

- For
  - usage:
    - for (initial statement ; condition ; iteration statement) {
      do something here;
      }

- There is other acceptable syntax (sort of)

- BTW, this is where the ++i and i++ becomes relevant and useful
- Everything in for can be done in a while
  - Think about it

20 ## Loops V

- The comma operator
  - Things are evaluated from left to right
- for (sum=0, i=1; i<=n; ++i)

  sum += i;
- for (sum=0, i=1; i<=n; sum += i, ++i)

  ;
- for (sum=0, i=1; i<=n; ++i, sum += i)

  ; *// this may give wrong results as i is*
  // incremented before added to sum

21 ## Loops VI

- Why can we use the ; just like that
- Infinite loops – beware
  - while (1) { …}
  - for (; ;) {…}
    - Use it at your own risk (system administrator may kill ;-))
    - Use it instead of running your program again and again

22 ## What does the following do?

```
for (i = 1; i <= 10; ++i )
        ;
sum += i;
```

23 ## Back to the Europe Trip example

- Now that we know loops, how would we use them to call our methods nicely

24 ## Assignment

- Read Ch. 8 and 9 from the Practical C Programming book
- Start reading Ch. 7

- **HW2**
  - Due soon.

1 **Introduction to Computer Science**

W 1113 – Lab (C)

Lab 6

Janak J Parekh
3/3/04

2 Recap from Lab 5

- Function prototypes
- Functions
- Conditionals
- Loops

3 Agenda

- Elements for HW#3
  - Variable scoping
  - Two-dimensional arrays
- Good coding practices
- Debugging
- Midterm review…

4 Variable scope

- Variables can be declared in different parts of your program, and this affects how they're accessible
- *Global* variables are declared outside any function
- *Local* variables are declared inside a function, or any arbitrary code block
- In C, local variables *must* be declared at the top of the block
- The "closest" one in the same block takes precedence

5 Example

```
#include<stdio.h>
int i = 5;
int main(void) {
    int i = 10;
    {
        int i = 12;
    }
    printf("%d\n", i);
}
```

- Yes, this is legitimate syntax! What's the answer?

6 A note on code blocks…

- Be *very* careful in identifying code blocks; use { } and proper indentation to keep your code clear
- If-else if-else: note that the latter two are *optional*, but should clearly correspond to the "original if" if present… legitimate syntax:

```
if(a) {
    if(b) { … }
    else { … }
} else { … }
```

7 Why global variables?

- If you have some piece of information used by lots of functions in the same program, no need to pass them as variables if they're already accessible
- However, be careful *not* to make everything global
- We'll get more used to structuring data later in the semester…

8 Permanent vs. temporary variables

- Book makes distinction – probably beyond the "scope" of this class
- Modern computers have a much larger stack
- Unless you're doing very special stuff, don't worry about it
- **static:** The most confusing keyword in C, ever

9 Two-dimensional arrays

- Easy to set up:
  - int a[10][20];
  - a[10][12] = 6;
  - Might want to "zero out" the array initially… how?
- Special meaning with strings
  - char strs[10][20];
  - You can treat this as a 2D array of chars, *or* as a 1D array of strings
  - In the latter, how many strings, and how many chars in each?
  - strcpy(strs[3], "Hello world");

10 Good coding practices

- Comment!
- Proper variable, function naming
  - In general, variables and functions have an initial lowercase, uppercase later
  - int numRecords = 0;
  - Indentation is very important, especially in keeping track of scope
    - emacs will help you in this
    - I've debugged people's code just by indenting it!

11 Good coding practices (II)

- Initial values for (most) variables
  - int i = 0;
  - int a[10] = { 0 };
  - Especially important in C – no presumed default
- Avoid very long functions: split up functionality
- Avoid overly complex logic if possible

12 Debugging tips

- gcc -Wall
  - Compile with "all warnings"
  - Often can catch errors this way
  - Sometimes will return some "optional" errors
- printf()
  - When stuck, print out intermediate results as your program runs

13 Using a debugger

- Especially with C code that crashes, it's hard to tell *why* the C code crashed
  - "Segmentation fault" isn't a very good answer
  - It'll only get worse when we learn pointers
- You can run your code through a debugger and see why it crashed
- Let's try a simple example…

14 Bad code

```
int main(void) {
  char c;
  strcpy(c, "This is a test");
}
```

- OK, this looks obvious here, but if you have a few hundred lines of code…
- Not surprisingly, it crashes

15 gdb – the GNU debugger

- First, compile your code with "-g"
  - gcc -g -o test test.c
- Then, run it with gdb
  - gdb test
- Common gdb commands
  - run
  - list – look at code
  - bt – "backtrace" along the function call stack
  - up/down – move among function call stack
  - break – add a "breakpoint"
- This is a whirlwind tour

16 gdb's unfriendly?

- Buy a commercial IDE
- Or, try ddd, which is a graphical frontend to gdb
  - Lots of features – I'll only scratch the surface in my "tour"
- You probably don't need to use a debugger for HW#3, but it'll be important for later homeworks

17 Midterm review…

- Any specific questions, first?
- Let's run through the slides

1 **Introduction to Computer Science**

W 1113 – Lab (C)

Lab7

Suhit Gupta
3/11/04

2 ## Questions about the previous lab

3 ## Questions about HW3

4 ## Recap from Lab 5

- Basically a recap from Lab 4
- Function prototypes
- Conditional statements
  - if
  - switch
- Loops
  - while
  - do while
  - for

5 ## Recap from Lab 6

- Code blocks
- Global variable scoping
- Two dimensional arrays
  - arrays of strings
- Debugging

6 ## Readme

- Write a README file
- Write a good README file
- It doesn’t have to be overly verbose

7 ## Comments

- Writing comments
- Writing good comments
- Often, naming variables well is a form of self-commenting code

8 ## Function prototypes

- Who does not understand them?
- Three types of submissions in HW2
  - everything in main() {…}
  - function before main, so you did not have to use function prototypes
  - function after main, but lucky this time

9 ## Preprocessors

- I already went over these two but here is a recap, and some more detail
- #include
  - /usr/include – stdio.h, stdlib.h, math.h, string.h, ctype.h, limits.h
  - If you use include math.h, then you need a –lm at the end of your compile command

10 Preprocessors II

- #define
  - convention – in caps
  - You can define macros as well
  - #define FOO bar
  - #define FOR_ALL for (i=0; i<ARRAY_SIZE; i++)

```
...
FOR_ALL {
        data[i] = 0;
}
```

  - #define SQR(x) ((x)*(x))
    - note the extra parentheses
- Both define and include end at EOL, however, you can continue with a \

11 Preprocessors III

- #ifdef (pg. 146) + #ifndef, #undef, #endif, #else
  - Conditional compilation

```
#ifdef DEBUG
    printf ("The code reaches this point\n");
#endif
```

- Now you can use #define DEBUG or #undef DEBUG

12 Bit operators

- ~ (unary operator) – Not
- | - Or
- & - And
- ^ - Xor (exclusive or)

13 Shift operators

- << - Left shift
  - Shifting left by 1 multiplies by 2
  - Shifting left by 2 multiplies by 4, or 2^2
  - Therefore, shifting left by n, multiplies by 2^n
- >> - Right shift (see Part II, Question 2, midterm)
  - Right shift divides by 2

14 Debugging

- "gcc –Wall <filename.c> will generate warnings
- gdb
  - gcc –Wall –g <filename.c>
- ddd
  - You run these two on a.out
  - run, bt, breakpoint, skip, step, lots of commands
  - step is good for loops

15 HW3 and Midterm questions…

- If we have time.

16 Assignment

- Read Ch. 10, 11 from the Practical C Programming book
- Read Ch. 12 for next class

- **HW3**
  - Don't wait too long

1 **Introduction to Computer Science**

W 1113 – Lab (C)

Lab8

Suhit Gupta
3/25/04

2 **Questions about the first half of the semester?**

3 **Questions about HW3 or HW4**

4 **Recap from Lab 6**

- Code blocks
- Global variable scoping
- Two dimensional arrays
  - arrays of strings
- Debugging

5 **Recap from Lab 7**

- Writing a README and comments
- Function prototypes (but I am still not sure everyone gets it)
- Preprocessors
  - #include
  - #define
- Bit Operators
- Debugging

6 **More on preprocessors**

- #ifndef
  - Allows for code to be compiled if symbol is *not* defined.

```
#ifndef DEBUG
    printf("This is production code");
#endif
```

- #else
  - basically does the same thing

```
#ifdef DEBUG
    printf("This is test code");
#else DEBUG
    printf("This is production code");
#endif
```

- You can use these techniques to debug as well as write regular code
  - Helps in commenting
  - /* lots of code */

7 **More on preprocessors**

- You can use these techniques to debug as well as write regular code
  - Helps in commenting

```
/***** I want to comment this testing section
    section_report();
        /* Handle the end of section stuff */
        dump_table();
***** end of commented out section */
```

  - What is wrong with this code?
- You can fix it by writing

```
#ifdef DEBUG
    section_report();
    /* Handle the end of section stuff */
    dump_table();
#endif
```

8 Structs

- Used to define your own types

```
struct structure-name {
 field-type field-name;
 field-type field-name;
 ….
} variable-name;
```

9 Structs II

- So an example would be

```
struct bin {
 char name [30];        // name of the part
 int quantity;          // how many in the bin
 int cost;              // the cost of the single part
} printer_cable_bin;          // where we put the cables
```

- Here printer_cable_bin is a variable of type struct bin
- You can omit the variable name

10 Structs III

- The dot operator
  - In order to access one of the fields of the struct, for a particular variable, use the form *variable.field*
  - *eg: printer_cable_bin.cost = 1295;*
  - *eg: total_cost = printer_cable_bin.cost * printer_cable_bin.quantity*

11 Structs IV

- I said earlier that you don't have to define variables when defining the struct
- So can I do, later in the code –
  - bin printer_cables_bin; (i.e. just like I use int or char)
  - Answer: No
- How to do it correctly
  - struct bin printer_cables_bin;
  - But this doesn't define any of the values inside of bin, therefore those remain undefined
  - So you can either assign them one at a time or you can do the following

```
struct bin printer_cable_bin = {
 "Printer Cables",
 0,
 1295
}; // However, this notation can only be used at the time of declaration
```

- Note the semicolons and the commas

12 Structs V

- (Shortcut) Initializing values –

```
struct bin {
 char name [30]; // name of the part
 int quantity;          // how many in the bin
 int cost;              // the cost of the single part
} printer_cable_bin = {
 "Printer Cables",
 0,
 1295
};
```

- Note the commas and the semicolon

13 Structs VI

- Structs typically go outside all methods
- You can have them inside methods but then those are local only to the method, this is NOT RECOMMENDED

```
#include<stdio.h>

int main(void) {
  struct a {
    int b;
```

```
  double c;
};

struct a suhit; /* = { 6 , 7.213432 };*/

suhit.b = 5;
suhit.c = 3.2;

printf("%d\n", suhit.b);
printf("%f\n", suhit.c);

return 0;
}
```

14 Unions

- There are like structs, however they have only one memory space.

  union *structure-name* {
   field-type field-name;
   field-type field-name;
   ….
  } *variable-name*;

15 Unions II

```
struct bin {
 char name [30];   // name of the part
 int quantity;     // how many in the bin
 double cost;      // the cost of the single part
} printer_cable_bin;        // where we put the cables

VS

union bin {
 char name [30];   // name of the part
 int quantity;     // how many in the bin
 double cost;      // the cost of the single part
} printer_cable_bin;        // where we put the cables
```

- Make space for largest variable

16 Unions III

- You can overwrite quantities, in union

  printer_cables_bin.name = "Printer Cables"
  printer_cables_bin.cost = 10;
  printf("The name of the bin is %s\n", printer_cables_bin.name);
  - What will the produce?
  - Answer: Unexpected result
  - You must keep track of which field you used
- So why use this?
  - Memory space saving

17 Typedefs

- Struct allows you to create a data type/structure
- Typedefs allow the programmer to define their own variable type

18 Typedefs II

- Usage
  - typedef *type-declaration*;
  - where *type-declaration* is the same as variable declaration, except that a type name is used instead of a variable name
  - eg: typedef int count; //creates a new type count that is the //same as an integer
  - Now you can say – count a; //equal to int a;

19 Typedefs III

- But you can get more complex

– typedef int group[10];
  - You can now say group classroom, which will create a variable classroom of 10 integers

```
main() {
 typedef int group[10];
 group class;
 for (i=1; i<10; i++)
                  class[i] = 0;
 return 0;
}
```

20 Typedefs IV

- But you can get more complex
  – typedef struct bin bin
    - This creates a variable type bin of type *struct bin*, and you can now say bin printer_cables_bin, instead of struct bin printer_cables_bin

```
struct bin {
 char name [30];
 int quantity;
 int cost;
};

typedef struct bin bin;

bin printer_cables_bin = {"Printer Cables", 10, 1290};
```

21 Enums

- This is designed for variables that contain only a limited set of values
- Traditionally, if you wanted to set up the days of a week, you would -

```
typedef int week_day;
const int Sunday = 0;
const int Monday = 1;
const int Tuesday = 2;
const int Wednesday = 3;
const int Thursday = 4;
const int Friday = 5;
const int Saturday = 6;

week_day today = Tuesday;
```

22 Enums II

- That was cumbersome
- You can say

```
enum week_day {Sunday, Monday, Tuesday,          Wednesday, Thursday, Friday, Saturday};

enum week_day today = Tuesday;
```

- Usage

enum *enum-name* (tag-1, tag-2, ….} variable-name;

23 Enums III

- You can omit variable-name, like in struct and union
- C implements the enum type as compatible with integer, so it is legal to say
  – today = 5; //though this may throw a warning
                // will make today Thursday

24 Enums IV – more examples

```
enum week_day {Sunday, Monday, Tuesday,       Wednesday, Thursday, Friday, Saturday};
enum day d1, d2; // makes d1 and d2 of type                  // enum day

d1=Friday;
if (d1==d2)
     …
```

25 Enums V – more examples

- You can use it to do switches

```
enum week_day {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

typedef enum day day;

day find_next_day(day d) {

    day next_day;

    switch(d) {
            case Sunday:
                    next_day = Monday;
                    break;
            case Monday:
                    next_day = Tuesday;
                    break;
            …       …
            case Saturday:
                    next_day = Sunday;
                    break;
    }
    return next_day;
}
```

26 Arrays of Structs

```
struct time (
    int hour;
    int minute;
    int second;
};

const int MAX_LAPS = 4;
strcut time lap[MAX_LAPS];

lap[count].hour = hour;
lap[count].minute = minute;
lap[count].second = second;
++count;
```

27 Arrays of Structs II

- Another way of initializing

```
struct time start_stop[2] = {
    {10, 0, 0},
    {12, 0, 0}
};
```

28 Structs with arrays

```
struct mailing {
    char name[60];
    char address1[60];
    char address2[60];
    char city[40];
    char state[2];
    long int zip;
};

struct mailing list[MAX_ENTRIES];

list[count].name[0]=S;
```

29 Casting

- (type) expression
- You already know this

```
int a;
float b, total;
total = (float)a + b;
```

30 Assignment

- Read Ch. 12 from the Practical C Programming book
- Start reading Ch. 13 for next class
- This class is going to get hard (pointers and memory allocation)
- **HW4**
  - Don't wait too long

1 **Introduction to Computer Science**

W 1113 – Lab (C)

Lab9

Suhit Gupta
4/1/04

2 **Questions about HW4**

3 **Recap from Lab 7**

- Writing a README and comments
- Function prototypes (but I am still not sure everyone gets it)
- Preprocessors
  - #include
  - #define
- Bit Operators
- Debugging

4 **Recap from Lab 8**

- preprocessors
- struct
- union
- typedef
- enum

5 **Pointer Basics**

- A pointer is a variable in C that contains a memory location.
- Pointers are used in programs to access memory and manipulate addresses.
  - We have already seen it briefly in scanf() where usage was scanf("%d", &v);

6 **Pointer Basics II**

- Declaration
  - int *p;
  - This creates 'p', which is of type "pointer to int"
  - The legal range of values for any pointer always includes the special address 0 and a set of positive integers that are interpreted as machine addresses on the system
- & is used to "point to" the address of a variable
  - This is used to dereference a variable's memory location
  - Officially - & is an operator that retrieves the memory address of a variable

7 **Pointer Basics III**

- Examples
  - p = &i; // p has the memory location of i
    // therefore *p points to i
  - p = 0;// shows assignment of p to 0
  - p = NULL; // same as p = 0;
  - p = (int *) 1307; // p now has an absolute // address in memory // We do this by using a cast // This is typically not done, why?

8 **Pointer Basics IV**

- Typical example (ptrexample0.c)

```
int var;          // Declare an integer var
int *p;           // Declare p as a pointer to an integer

var = 4;          // Set the value of var to be 4
p = &var;    // Set p to be the address of var

printf ("%d", p);     // Is this accurate?

*p = 5;           // Sets the value of the thing p is pointing to, to 5
p = 5;            // What will this do?
```

9 Pointer Addressing/Dereferencing

```
int a, b;
int *p;
a = b = 7;
p = &a;

printf("%d\n", *p);          // What is printed?
*p = 3;
printf("%d\n", a);           // What is printed?
```

10 Pointer Addressing/Dereferencing

```
p = &b;

*p = 2 * *p – a;
printf("b = %d\n", b);       // What does this print?
```

11 * and & relationship

- Simply put, the dereference operator (*) is the inverse of the address operator (&).

```
double x, y, *p;

p = &x;
y = *p;

// Here, p is assigned to address of x. Then y is assigned to the
// value of object pointed to by p

y = *&x;
y = x;
//How do these two statements relate to the above two?
```

(ptrexample1.c)

12 Multiple pointers can point to one location

```
int something;

int *first_ptr;
int *second_ptr;

something = 1;

first_ptr = &something;
second_ptr = first_ptr;
```

13 Convince yourself

14 Call by reference

- Pointers can be used as function arguments
- We have been typically using call by value
- Remember the swap function

```
#include <stdio.h>

int swap (int a, int b);
int main () {
        int x=3, y=7;
        printf("%d %d\n", x, y);
        swap (x,y);
        printf("%d %d\n", x, y);
        return 0;
}

int swap (int a, int b) {
        int tmp;
        tmp=a;
        a=b;
        b=tmp;
        return a;       // I can return only one value, what do I return?
} //ptrexample2.c
```

15 Call by reference II

- Note that the call-by-value has problems in that only the method's local values are affected.
- Therefore we need something else
  - Pointers to the rescue
  - We call other functions and pass parameters by reference
  - New code looks like

16 Call by reference III

```
#include <stdio.h>

int swap (int *, int *);

int main() {
          int x=3, y=7;

          printf("%d %d\n", x, y);

          swap (&x,&y);
          printf("%d %d\n", x, y);
          return 0;
}

int swap (int *p, int *q) {
          int tmp;

          tmp = *p;
          *p = *q;
          *q = tmp;
}
//ptrexample3.c
```

17 Call by reference IV

- Another example

```
#include <stdio.h>

void inc_count (int *count_ptr)

int main () {
        int count = 0;

        while (count < 10)
                inc_count(&count);

        return 0;
}

void inc_count(int *count_ptr) {
        (*count_ptr)++;
}
```

18 Assignment

- Read Ch. 13 from the Practical C Programming book

- **HW4**

1 **Introduction to Computer Science**

W 1113 – Lab (C)

Lab10

Suhit Gupta
4/8/04

2 **Questions about HW5**

- I highly recommend that you start early
- It is not an easy assignment

3 **Recap from Lab 8**

- preprocessors
- struct
- union
- typedef
- enum

4 **Recap from Lab 9**

- Pointer basics
- Pointer addressing/dereferencing
- * and & relationship
- Call by reference

5 **const Pointers**

- Declaring constant pointers is a bit tricky

const int result = 5;

- Now result is 5, so result=10; is illegal
  - BTW, why would I use const and not #define
- However, the following does not limit answer_ptr as above

*const chat *answer_ptr = "Forty-Two";*

- Instead, it tells the compiler that whatever answer_ptr is pointing to, is a contant
- So now the data cannot be changed but the pointer can

6 **Pointer Arithmetic**

- What do the following return?
  - given –> char data ='a'; char *ptr = &data;

1. &data
2. ptr
3. &ptr
4. *ptr
5. *ptr+1
6. *(ptr+1)
7. ++ptr
8. ptr++
9. *++ptr
10. *(++ptr)
11. *ptr++
12. (*ptr)++
13. ++*ptr++
14. ++*++ ptr

7 **Pointers and Arrays**

- As shown from before, C allows pointer arithmetic. And this is actually very helpful with arrays

char array[5];
char *array_ptr = &array[0];

- This means, array_ptr is array[0], array_ptr+1 is array[1], and so on…
- However (*array_ptr) + 1 is not array[1], instead it is array[0] + 1
  - ptrexample4.c
- Now this is a horrible way of representing array, so why use this?

8

Pointers and Arrays II

```
#include <stdio.h>

#define ARRAY_SIZE 10

char array[ARRAY_SIZE + 1] = “0123456789”;

int main() {
        int index;
        printf(“&array[index] (array+index) array[index]\n”);
        for (index=0; index<ARRAY_SIZE; ++i) {
                        printf(“0x%-10p 0x%-10p 0x%x\n”, \
                        &array[index], (array+index), array[index]);
        return 0;
}
//ptrexample9.c
```

- What does this program do?

9

Pointers and Arrays III

- Arrays are actually pointers to a sequential set of memory locations
  - char a[10]; means ‘a’ points to the array’s 0th memory location
- Feel like horror movie revelation?
- However, this actually helps us with pointers
  - you don’t have to pass the address of the array, you can just pass the array itself

10

Pointers and Arrays IV

```
#include <stdio.h>

char strA[80] = "A string to be used for demonstration purposes";
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 */
   pB = strB;                   /* point pB at string B */
   putchar('\n');               /* move down one line on the screen */
   while(*pA != '\0')           /* line A (see text) */
   {
      *pB++ = *pA++;   /                 * line B (see text) */
   }
   *pB = '\0';                  /* line C (see text) */
   puts(strB);                  /* show strB on screen */
   return 0;
}                                        //ptrexample5.c
```

11

Pointers and Strings

- You can use pointers to separate strings
- Assume given string is of the form “First/Last”
- You can find the / using *strchr* (used to find a character in a string, and it returns a pointer to the first occurrence of the character
  - Then replace it with a NULL
- OR, using pointers, you don’t have to reaplce anything
  - just have a pointer point to the beginning of the string (this is easy since we just learned about arrays, and we know that strings are arrays)
  - make a new pointer to point to the location after the ‘/’
- No over-writing needed, you preserve the original data

12

Pointers and structures

- Another motivation for pointers, reduces the amount of data to be moved
- Reminder no structures – ptrexample6.c
- What does the following do?

```
struct mailing {
        char name[60];
        char address1[60];
        char address2[60];
        char city[40];
        char state[2];
```

```
    long int zip;
} list[MAX_ENTRIES];
```

13 Pointers and structures II

- The code on the previous slide create a mailing list struct
- We may need to sort the mailing lists
- Each entry is fairly long (note the size of each array)
  - btw… how long is each entry of the struct?
- So that is a lot of data to move around
- A solution: declare an array of pointers and then sort the pointers

14 Pointers and structures III

- Therefore, looks at the following piece of code

```
struct mailing *list_ptrs[MAX_ENTRIES];
int current;

for (current=0; current=number_of_entries; ++current) {
    list_ptrs[current] = &list[current];
}
```

- What does the above piece of code do?
  - Instead of moving a 226 byte structure aroung, we only move 4 byte pointers
  - Therefore sorting is much faster

15 Pointers and structures IV

- Accessing pointer structures is similar to regular structures
- Remember the '.' operator
  - It is replaced with the '->' operator in pointers to structures, rather than the structure itself

```
struct SIMPLE {
    int a;
    int b;
    int c;
}
```

- Things are fairly trivial here, as before…
  - struct SIMPLE simple;
  - simple.a = 1;
  - etc.

16 Oh btw…

```
typedef struct {
    int a;
    int b;
    int c;
} SIMPLE;
```

- What does this do?
- And how is it different from

```
typedef struct SIMPLE {
    int a;
    int b;
    int c;
} s;
```

17 Pointers and structures V

```
struct COMPLEX {
    float f;
    int a[20];
    long *lp;
    struct SIMPLE s;
    struct SIMPLE sa[10];
    struct SIMPLE *sp;
}
```

- struct COMPLEX comp;
- ((comp.sa) [4]).c
  - same as comp.sa[4].c

18 **Pointers and structures VI**

- However, if you have
  - struct COMPLEX *cp;
  - Then, you can only have
    - (*cp).f
    - But this is a pain to write everytime, so -> is used instead
    - cp->f
- There is now tons of fun you can have with
  * & . ->
- Combine these to access nested structs, pointers to structs, plain structs, whatever…

19 **Command line arguments**

- Next motivation for pointers - we have already seen this
- main (int argc, char *argv[]) {
- The array argv[] contains the actual arguments
  - however it is of type *pointer to a character array*

20 **Command line arguments**

- Now you can learn to use flags
- What are flags?
  - “-v”, “-h” after your program will set some setting, or call your program in a particular mode
- This is typically done in most programs
- Note most ‘man’ pages
- “-h” flag used in addition to the README

21 **Pointer to a pointer**

- int **c; declares c as a pointer to a pointer to an integer

int a = 12;
int *b = &a;
int **c = &b;

- Pointers to pointers follow the same rules as just regular pointers

22 **How not to use pointers…**

- What is wrong with the following?

int *a;
*a = 12;

- a doesn’t have a place to put 12

23 **Final motivation for pointers**

- We will see this next time
- malloc();
- You can use this function to allocate memory to certain variables or arrays
- You can then point to this memory using pointers
- This is also useful in dealing with peripherals of a computer
- We will also see more on arrays and multi-dimensional arrays
- But all this for next time ☺

24 **Assignment**

- Read Ch. 17 from the Practical C Programming book
- **HW5**

1 **Introduction to Computer Science**

W 1113 – Lab (C)

Lab11

Suhit Gupta
4/15/04

2 ## Questions about HW5

3 ## Recap from Lab 8

- preprocessors
- struct
- union
- typedef
- enum

4 ## Recap from Lab 9

- Pointer basics
- Pointer addressing/dereferencing
- * and & relationship
- Call by reference

5 ## Recap from Lab 10

- const Pointers
- Pointer arithmetic
- Pointers and Arrays
- Pointers and Strings
- Pointers and Structs
- Command Line Arguments (Pointers)
- Pointer to a Pointer
- How not to use pointers

6 ## A small segway…

- You guys asked questions about the printf statement here last time

```
printf("&array[index] (array+index) array[index]\n");
      for (index=0; index<ARRAY_SIZE; ++index)
                printf("0x%-10p 0x%-10p 0x%x\n", \
                &array[index], (array+index), array[index]);
```

- Here “-10” left justifies the text
- The %x prints out hexadecimal
- For lots more information on printf
  - man printf
  - man 3 printf
  - man 3c printf
  - man –s 3c printf

7 ## Storing an indeterminate amount of data

- How would you store an indeterminate amount of data?
- You create a bank, but you don’t know how many accounts you are going to have
- Two ways to fix this
  - Growable arrays
    - If the array fills up, create an array twice its size and copy all the elements over
  - Linked Lists

8 ## Pointers and linked lists

- Instead of statically declaring an array, we can create a bunch of nodes and link them together

```
struct node {
    struct node *next_ptr;
    int value;
}
```

- If you wanted to create a large number of these nodes

```
struct node node_1;
struct node node_2;
```

- BTW, do you guys know what linked lists are?

9 Pointers and linked lists II

- However, you can only declare a limited number of nodes.
  - well, ok, so you can create a lot, but if you didn't know how many you would need, then you have a problem.
- Therefore you can allocate memory dynamically

10 function malloc()

- malloc();
  - *usage:* void *malloc (unsigned int);
  - It allocates storage for a variable and returns a pointer.
  - It is used to create things out of thin air ☺
  - Up to now, we use pointers to point to predefined variables
  - With malloc we can allocate memory without having to predefine a variable
  - The void * mean that malloc returns a generic pointer

11 malloc examples

```
#include <stdlib.h>
main() {
    char *string_ptr;
    string_ptr = malloc (80);
}
```

- This allocates storage for a character string 80 bytes long ('\0' included)

12 malloc examples

- More precisely

```
#include <stdlib.h>
main() {
    char *string_ptr;
    string_ptr = malloc (80 * sizeof(char));
}
```

13 malloc examples II

- You may be allocating lots of variables of type struct, each of which has large arrays. Therefore you are allocating real space in memory for each instance

14 free()

- It is the opposite of malloc
- malloc allocates memory
- You can de-allocate it using free
- free takes a pointer as an argument, just as malloc returns a pointer
- *Usage*: free(pointer);
  - Here pointer is what was returned by malloc

- Not freeing / Double freeing is bad

15 free() example

```
#include <stdlib.h>
main() {
      char *string_ptr;
      string_ptr = malloc (80);

      free(string_ptr);
      string_ptr = NULL;
}
```

- You typically NULL out the pointer as well
- If you don't use free, you will keep eating the allocated memory every time you call the respective function

16 Heaps and Stacks

- How does all of this happen in memory?
- There are two ways that this is all stored in memory
  - Heaps
  - Stacks
- Stacks used for regular variables that you have seen so far
- Heaps used for malloc();

17 Heaps and Stacks II

- When you call a function, space for all the local function variables, etc. are created in memory, in a stack frame
  - When you leave the function, all that memory is cleaned up
- However, when you allocate space using malloc, it is allocated in a heap
  - It is not cleaned up when leaving a function
  - Therefore you have to use free

18 Dangling pointers

- A dangling pointer is a surviving reference to an object that no longer exists at that address. Dangling pointers typically arise from one of:
  - A premature free, where an object is freed, but a reference is retained;
  - Retaining a reference to a stack-allocated object, after the relevant stack frame has been popped.

19 Bad code (preliminary free)

```
int main(void) {
  int *result = malloc(sizeof(int));
  *result = 6;
  free(result);
  printf("result is %d\n", *result);
}
```

20 Bad code (stack memory)

```
int main(void) {
  int *result = square(6);
  printf("result is %d\n", *result);
}

int *square(int i) {
  int j = i * i;
  return &j;
```

}

21 Back to linked lists

- So how does malloc help us here?

```
struct linked_list {
    char data[30];
    struct linked_list *next_ptr;
}
struct linked_list *first_ptr = NULL;
```

- So we want to use malloc instead of creating an array of linked lists that will limit the number of nodes in the linked list to the size of the array
- How can we do this?

22 Pointers and Linked Lists contd…

```
new_node_ptr = malloc(sizeof(struct linked_list));
```

- This created the new node and allocates the correct amount of memory

```
(*new_node_ptr).data = item;
```

- This will store the value of item into data

```
(*new_node_ptr).next_ptr = first_ptr;
```

- The node now points to first_ptr

```
first_ptr = new_node_ptr;
```

- The new element is now the first element

23 One other concept like malloc()

- calloc()
  - *Usage*: void *calloc (int n, int size_of_n);
  - similar to malloc(), except that you give it that second argument of the number of elements followed by the size of each of those elements
  - Slightly cleaner than malloc(sizeof(foo) * nElements)

24 More code examples

- Average n numbers in a dynamically-defined array
- Add an element to the *end* of the linked list instead of the beginning
- (HARD!) Delete an element from a linked list

25 Assignment

- Read Ch. 14 from the Practical C Programming book

- **HW5**

1 **Introduction to Computer Science**

W 1113 – Lab (C)

Lab12

Suhit Gupta
4/22/04

2 ## Questions about HW6

3 ## Recap from Lab 10

- const Pointers
- Pointer arithmetic
- Pointers and Arrays
- Pointers and Strings
- Pointers and Structs
- Command Line Arguments (Pointers)
- Pointer to a Pointer
- How not to use pointers

4 ## Recap from Lab 11

- malloc
- free
  - Dangling pointers
- calloc
- Pointers and Linked Lists

5 ## A repeat of the linked list example

- So how does malloc help us here?

```
struct linked_list {
      char data[30];
      struct linked_list *next_ptr;
}
struct linked_list *first_ptr = NULL;
```

- So we want to use malloc instead of creating an array of linked lists that will limit the number of nodes in the linked list to the size of the array
- How can we do this?

6 ## Pointers and Linked Lists contd…

new_node_ptr = malloc(sizeof(struct linked_list));

- This created the new node and allocates the correct amount of memory

(*new_node_ptr).data = item;

- This will store the value of item into data

(*new_node_ptr).next_ptr = first_ptr;

- The node now points to first_ptr

first_ptr = new_node_ptr;

- The new element is now the first element

7 ## File I/O

- Now that you know pointers and malloc, you are ready for file I/O
- *Usage*: FILE *file;
- To open a file – fopen();
- *Usage*: void *fopen(*name, mode*);
  - file = fopen (name, mode);

- NULL is returned on error
- *name* is the actual name of the file
- *mode* indicate the property with which to open the file

8 Options for mode

- *mode* indicates whether the file is open for reading or writing
- ‘w’ for writing
- ‘r’ for reading
- Example

```
FILE *in_file;
in_file = fopen(“input.txt”, “r”);
if (in_file == NULL) {
      fprintf (stderr, “Error: Could not open the input file ‘input.txt’\n);
      exit (8);
}
```

9 Close a file – fclose()

- fclose() will close a file
- *Usage*: fclose (pointer to file);
- status = fclose(in_file);
  - You don’t need status
    - fclose(in_file);
      - This will just throw away the return value
  - ‘status’ will be 0 is file was closed successfully
  - It will be non-zero is there is an error
    - Do a man on fclose to see the different error codes

10 Simple operations

- fputc – This function writes a single character to a file
  - *Usage*: fputc (character, file)
- fputs – This function writes a string to a file
  - *Usage*: fputs (string, size, file)
  - *Usage*: fputs (string, sizeof(string), file)
    - This will return a pointer to the string if successful or NULL if there is an error
  - Sometimes there are problems when you try to write strings that are very long

11 Simple operations II

- fgetc – This function gets a single character from a file
  - *Usage*: fputc (character, file)
  - Typically used when you have a stream of data coming in and you need to read the characters coming in one at a time
- fgets – This function gets a string to a file (similar to fputs)
  - *Usage*: fgets (string, size, file)
  - *Usage*: fgets (string, sizeof(string), file)
    - This will return a pointer to the string if successful or NULL if there is an error
  - Read the text book as well as the man page to see the intricacies with fgets
    - You need to worry about the \n, \0, etc at the end of the string as well as the end of the file

12 More operations

- fprintf
  - *Usage*: count = fprintf (file, format, parameter1, parameter2, …)
    - count is the number of characters sent (-1 if error)
    - format describes how the arguments are to be printed
    - parameters – to be converted and sent
- Similar function
  - sprintf
    - *Usage*: sprintf (string, format, parameter1, parameter2, …)

13 More operations II

- fscanf
  - *Usage*: fscanf (file, format, &parameter1, …)

- And similar to fscanf is sscanf
  - *Usage*: fscanf (string, format, &parameter1, …)

14 Example

```
#include <stdio.h>
#include <stdlib.h>

int main() {
        char name [100];
        FILE *in_file;

        printf ("Name of file? ");
        fgets(name, sizeof(name), stdin);

        in_file = fopen(name, "r");

        if (in_file == NULL) {
                fprintf(stderr, "Could not open the file\n");
                exit (8);
        }
        printf ("File found\n");
        fclose(in_file);
        return 0;
}
```

15 Example II

```
#include <stdio.h>
#include <stdlib.h>
const char FILE_NAME[] = "input.txt";

int main() {
        int count = 0;
        FILE *in_file;
        int ch;

        in_file = fopen(name, "r");
        if (in_file == NULL) {
                fprintf(stderr, "Could not open the file\n");
                exit (8);
        }
        while (1) {
                ch = fgetc(in_file);
                if (ch == EOF)
                        break;
                count++;
        }
        printf ("Number of characters in %s is %d\n", FILE_NAME, count);
        fclose(in_file);
        return 0;
}
```

16 Example III

```
#include <stdio.h>
#include <stdlib.h>
#ifndef __MSDOS__
#include <unistd.h>
#endif __MSDOS__

int main() {
        int cur_char;
        FILE *out_file;

        out_file = fopen ("test.out", "w");
        if (out_file == NULL) {
                fprintf(stderr, "Cannot open output file\n");
                exit (8);
        }
        for (curr_char = 0; cur_char < 128; cur_char++)
                fputc(cur_char, outfile);
        fclose (out_file);
        return 0;
}
```

17 Advanced concept - strtok()

- Used to tokenize a given string
- *Usage*: char *strtok (char *s1, const char *s2)
- It searches for tokens in s1, using the character in s2 as token separator
- If s1 contains one or more tokens
  - the first token in s1 is found
  - the character immediately following it is overwritten with a NULL
  - the remainder of s1 is stored elsewhere
  - the address of the first character in the token is returned
  - subsequent calls with s1 equal to NULL return the base address of a string supplied by the system that contains the next token
  - If no additional tokens are available, NULL is returned

18
Example using strtok

```
char s1[] = “ this is,an   example ; ”;
char s2[] = “,;  ”;

printf (“\“%s\””, strtok (s1, s2));
while ((p=strtok(NULL, s2)) != NULL) // p here is a pointer to the
       printf(“ \“%s\””, p);                                 // character we are checking
putchar(‘\n’);
```

- This will print out
  - “this” “is” “an” “example”

19
strdup()

- Duplicates a string
- *Usage*: char *strdup(const char *s);
- Basically, given a string, it will duplicate it
  - it will return a pointer to the duplicate string

20
Things to remember

- Always close the file before leaving the program
- Functions can take file pointers as arguments
  - void my_func (FILE *, FILE *) { … }
- All functions take file pointers and not the file names themselves

21
Assignment

- Read Ch. 18 from the Practical C Programming book

- **HW6**

1 **Introduction to Computer Science**

W 1113 – Lab (C)

Lab13

Suhit Gupta
4/29/04

2 Questions about HW6

3 Question about review session

- Wednesday or Thursday?

4 Recap from Lab 11

- malloc
- free
  - Dangling pointers
- calloc
- Pointers and Linked Lists

5 Recap from Lab 12

- Pointers and Linked Lists
- File *
  - fopen()
  - fclose()
- Input and Output to/from files
- strtok() and strdup()

6 Short Lab today

- We will cover two topics
  - Modularity
  - Makefiles

7 Modularity

- You would want to deal with modularity in two cases
  - If you have multiple people working on the same “project”
  - If you want to reuse one piece of code in multiple places

8 Example – calendar.c

- Look at the solutions
- Now, imagine that each function in this piece of code needed to be written by a different programmer
- Separate out all the functions into separate files
- Each file gets a .h, but no main()
- The main file
  - contains the main() function
  - includes all the .h files (in “ ”)

9 Let us look at a real example

- From the text book…
  - Ch. 18, pg 308, 311 and 318

10 Makefiles

- How does Java compile pieces of code?
- How does C do it?
- How would you compile multiple files together
- Dependencies

11 The GNU make utility

- http://www.gnu.org/manual/make-3.79.1/html_node/make_toc.html
- The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.
- You have to have a Makefile
- Run make to start rules in the Makefile file.

12 Example of a Makefile

13 From the example

- To use this makefile to create the executable file called ‘edit’, type: make
- make clean
- You can also define variables/macros
  - CC = gcc
  - $(CC)

14 The stuff I covered today

- This will not be on the final exam
- Good knowledge though

- Question about C or about the course in general

15 Assignment

- **HW6**

- **Have a good Final Exam!**