Discussion 2 Richard Guo
Fundamentals of C 01/23/09

Note: Last time, I implied that you are only responsible for Kilo, Mega, Giga. This is not the case. All the prefixes on last discussion's table are fair game for tests.

1. Answers to Last Time's Problems

Number Bases

Decimal	Binary	Hex
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

4 more bits needed to store 15M versus M. (log2(15) < 4)

Signed 64 bit integer: −9,223,372,036,854,775,808 to +9,223,372,036,854,775,807

12345 (decimal) = 3*8^4 + 7*8^1 + 1*8^0 = 12288 + 56 + 1 → 30071 (octal)

Hex allows us to easily concatenate binary (4 digits → 1 digit), just like octal but octal is smaller than base 10 while hex is larger. Thus, a number in octal can be accidentally interpreted as decimal. Finally, the size of a pointer is either 32 bits or 64 bits, both divisible by 4 (hex) but not by 3 (octal).

2. Directing Program Flow

- **If-else if-else Statements**

```
if (condition 1)
{
      // execute this
}
else if (condition 2) // implied that condition 1 was false
{
      // execute this
}
else // if all else fails
{
      // execute this
}
```

- **For loops**

For loops are generally safer than while loops because they force you to take into account the 3 components of loops: initialization, condition of execution, and incrementation.

```
int i;
for (i = 0; i < 50; i++) // loop will run 50 times
{
      // execute this
}
```

Note that you can leave components blank in the for statement.
`for (; ;) is a valid infinite loop.`

There are keywords *break* and *continue* that can be used in conjunction with loops.
Break jumps out of the loop. Continue skips execution of what remains in the current iteration but continues the loop.

```
int i;
for (i = 0; i < 50; i++)
{
      if (i == 25)
            break;
      if (i % 2 == 0)
      {
            printf ("even\n");
            continue;
      }
      printf ("odd\n");
}
```

3. Special Operators

- Operators
  - Arithmetic: +, -, *, /, %
  - Logical:
    - Relational:
      - > greater than
      - < less than
      - >= greater than or equal to
      - <= less than or equal to
    - Logical:
      - == equal to
      - != not equal to
      - && and
        - `if (condition1 && condition2)`
      - || or
    - Bitwise:
      - & bitwise and
      - | bitwise or
      - ^ bitwise XOR

- ~ bitwise inversion
- <<, >> bitwise shift
    - `x >> 4;`
    - `x << 10;`

- Assignment:
    - = sets left side equal to right side
    - +=, -=, *=, /=, %= sets left side equal to operation applied to left side
        - `x += 10 becomes x = x + 10;`
        - `x -= 20 becomes x = x – 20;`
    - ++, -- special increment/decrement
        - `--x; // decrement before performing other operations`
        - `x++; // increments after performing other operations`

4. Memory Management

So far, we've seen *statically* declared variables of known data types. How about allocation of an arbitrary amount of memory? There are two ways to do this - arrays and pointers.

- **Arrays**

```
data_type variable_name [size]; // syntax for declaring an array
int int_array[5]; // allocates a 20 byte contiguous block of uninitialized memory
int int_array[] = {1, 2, 3, 4, 5}; // initializes an array to hold 5 ints
```

An array by itself contains the *address* of the beginning of the block of memory. Elements are accessed by calling the index of an array. Indexes start at 0.

Assume for now that int_array is allocated starting at byte 20.

int_array2[1] would return 2 because it's accessing the data. What would int_array be? How about &int_array?

Arrays normally have no boundary checks, so to avoid running out of the array boundary, use #define to specify array length or store the size in the first element.

- **Pointers**

```
data_type *variable_name; // syntax for declaring pointer
```

A pointer contains the *address* of something. A pointer also has its own distinct address. * dereferences the pointer by going to what it points to. & is the “address of” operator that returns the address of a variable.

```
int blah = 123;
int *ptr = &blah;
printf ("%d %d %d", *ptr, ptr, &ptr);
```

What would the above statement print out? Try it yourself.

Why bother having pointers in the first place? Imaging that you are passing something to a subroutine:

```
void increment (int bar)
{
      bar++;
}
```

```
void increment (int *bar)
{
            *bar++;
}
```

Both forms increment the variable bar, but because bar is passed *by value* in the first case and *by address* in the second case, only the second case will result in bar being modified.

Pointers and arrays are almost interchangeable. Because addresses are allocated contiguously and an array stores the address of the first element, incrementing the address allows access of the next elements in the array.

```
int main ()
{
      int int_array[] = {1, 2, 3, 4, 5};
      int *blah;
      for (*blah=int_array; *blah < 6; blah++)
            printf ("%d %d ", *blah, blah);
      return 0;
}
```

output:
1 1055568624
2 1055568628
3 1055568632
4 1055568636
5 1055568640

Know the difference between *ptr++; and (*ptr)++;

- **Types of Memory**

4 types: Program, local, stack, and heap
The program and any global variables are loaded into lowest parts of memory at runtime.
The stack contains local variables, return addresses, and recursive function calls. Grows downwards.
The heap is space for dynamically allocated memory. Grows upwards.

- **Malloc, calloc, and free**

```
form: void * malloc (# of bytes)
int *ptr = (int *) malloc (10*sizeof(int)); // allocates array of 10 integers
int *ptr = (int *) calloc (10, sizeof(int)); // allocates and initializes to 0

free (ptr); // releases the memory ptr was allocated
```

5. Types of Errors

Segfault: access location that program is not allowed to access (caught by OS)
Bus error: same as segfault except caught by hardware
Syntax error: something wrong with C code correctness (e.g. missing semicolon); caught by compiler
Logic error: will compile, syntactically correct, but the thinking is wrong so output is unexpected

Memory leak: failure to free memory after it's allocated

6. Solved Problems

For each line, state how much memory is allocated (if any) and in which section of memory.

```
// program loaded into code space in memory
void increment (int *bar);
char * g_char = "hi"; // allocates 4 bytes for char* and 3 bytes for "hi" in local

int main () // 4 bytes for return address on stack
{
      int *ptr; // 4 bytes for int* in stack
      *ptr = (int *) calloc (1, sizeof(int)); // 4 bytes for int in heap

      increment (ptr); // 4 bytes for return address and 4 for copy of ptr in stack
}

void increment (int *bar)
{
      int temp = *bar; // 4 bytes for new int in stack
      free (bar); // -= bytes in heap due to release of memory
      *bar = (int *) malloc (sizeof(int)); // 4 bytes in heap
      *bar = temp;
}
```

In the following code, identify what i, *ptr, and ptr would contain depending on the code in the space.

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

int main()
{
      int i = 0;
      int *ptr = (int*)malloc(2*sizeof(int));
      *ptr=1;
      *(ptr+1)=3;
      // code goes in here
      return 0;
}
```

Code	i	*ptr	ptr (in bytes)
i = (*ptr)++;	1	2	ADDR
i = ++(*ptr);	2	2	ADDR
i = ++ptr;	ADDR+4	3	ADDR+4
i = ptr++;	ADDR	3	ADDR+4
i = *(ptr++);	1	3	ADDR+4
i = *(++ptr);	3	3	ADDR+4

Note that *ptr++ is the same as *(ptr++)

7. More Problems (solutions will be on *next* discussion's handout)

1. Identify all the errors in the following code without using a compiler. What type of error are they?

```
#include <stdlib.h>

int increment (int ptr);
int ptr;

void main ()
{
      int *ptr = (int *) malloc (sizeof(int));
      ptr = &ptr;
      for (int a = 0; a < 1; a++)
            *(ptr--) = increment(&ptr);
      return;
}

int increment (int ptr);
{
      return ptr++;
}
```

2. Write a program that prompts the user for a number N an then allocates an NxN matrix of integers. Set the blocks in the matrix to contain sequential integers values. Hint: use scanf to get user input. Print out the matrix.

3. Modify the above program to print out an (N-2)x(N-2) matrix where each value is equal to an average of the eight surrounding places in the original NxN matrix. Note that the borders are neglected because they do not have 8 surrounding places.

4. Modify program 2 so that instead of prompting the user for input, get the value N from argv.
e.g. someone should be able to call:
./your_prog 20
to set a 20x20 matrix containing the numbers from 0 to 399.