Agenda

Standard I/O

Reading and writing text files

Pointer-based data structures

Standard I/O

Three default streams

- stdin (scanf, fscanf)
- stdout (printf, fprintf)
- stderr (fprintf(stderr, <format string>, <data>))

stdin, **stdout**, **stderr** are tied to files with descriptors 0, 1, and 2 respectively

At the command line we can redirect these streams to files

- ./a.out < infile.txt
- ./a.out > outfile.txt
- ./a.out &> allfile.txt (all output)
- ./a.out 2> errfile.txt (error output only)
- ./a.out 1> outfile.txt is the same as ./a.out > outfile.txt

File I/O

Reading/writing files is done using a **file pointer**

```
FILE *infile; FILE *outfile;

infile = fopen("input.txt", "r");
if (infile == NULL) {
   printf("Error: unable to open file %s\n", "input.txt");
   exit(1);
}

outfile = fopen("output.txt", "w");
if (outfile == NULL) {
   printf("Error: unable to open outfile\n");
   exit(1);
}

fclose(infile);
fclose(outfile);
```

Standard I/O (C)

```
int a;
 char word[32];
 printf("Enter an integer: ");
 scanf(" %d", &a);

 printf("Enter a string: ");
 scanf(" %s", word);

 printf("\nYou entered: %d %s\n",
              a, word);
 fprintf(stderr, "Test printing an error\n");
```

```
int a;
 char word[32];
 fprintf(stdout, "Enter an integer: ");
 fscanf(stdin, " %d", &a);

 fprintf(stdout, "Enter a string: ");
 fscanf(stdin, " %s", word);

 fprintf(stdout, "\nYou entered: %d
%s\n",
              a, word);
 fprintf(stderr, "Test printing an error\n");
```

printf(...) is the same as fprintf(stdout, ...)

When we redirect output/input, they are fed into/out of these commands (demo)

Reading Text Files (C)

```
  const char* filename = "grades.txt";
  FILE* fp = fopen(filename, "r");
  if (!fp) // fp is NULL on error
  {
    printf("Cannot load file: %s\n",
filename);
    return 1;
  }
  int num = 0;
  float sum = 0;
  char line[1024];
  while (fgets(line, 1024, fp))
  {
    int grade;
    sscanf(line, " %d", &grade);
    sum += grade;
    num++;
  }
  printf("The average is %.2f\n", sum/num);
  fclose(fp);
```

```
99
79
51
92
56
89
63
```

Helpful text file commands

- fprintf, fscanf
- fgets, fputs (gets/puts a line of input)
- fgetc, fputc (gets/puts a single character)

Helpfull string commands

- strlen: get the length
- strcpy: copy one string to another
- strncmp, strcmp: returns 0 if two strings are the same
- strstr: check if a string contains a substring
- strtok: tokenize a string

Example: strtok

```
void print(char** list, int n) {
  for (int i = 0; i < n; i++)   {
    printf("%s, ", list[i]);
  }
  printf("\n");
}
```

```
int main() {
   char line[1024];
   strcpy(line, "cat bear dog");

   int n = 0;
   char* list[16]; // max of 16 words can be put in the list
   char* token = strtok(line, delims);
   while (token && n < 16)    {
      list[n] = token;
      token = strtok(NULL, delims);
      n++;
   }
   print(list, n);
}
```

Discuss: How does redirected input/output compare to reading/writing from files?

Pointer-based data structures

E.g. linked lists, trees, graphs

No built-in data structures in C!

Nodes in these types of data structures use **structs**

struct pointers connect nodes together

Dive into systems calls these “self-referential structs” (Section 2.7.5)

Example

```
struct node {
   int data;         // used to store a list element's data value
   struct node *next;  // used to point to the next node in the list
};
```

```
struct node *head, *temp;
int i;

head = NULL;  // an empty linked list

head = malloc(sizeof(struct node));  // allocate a node
assert (head != NULL); // quit if head is null
head->data = 10;    // set the data field
head->next = NULL;  // set next to NULL (there is no next element)

// add 2 more nodes to the head of the list:
for (i = 0; i < 2; i++) {
   temp = malloc(sizeof(struct node));  // allocate a node
   assert (temp != 0); // quit if temp is null
   temp->data = i;     // set data field
   temp->next = head;  // set next to point to current first node
   head = temp;       // change head to point to newly added node
}
```

Draw the Stack/Heap

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

```
struct node *head, *temp;
int i;

head = NULL;

head = malloc(sizeof(struct node));
assert (head != NULL);
head->data = 10;
head->next = NULL;

for (i = 0; i < 2; i++) {
   temp = malloc(sizeof(struct node));
   assert (temp != 0);
   temp->data = i;
   temp->next = head;
   head = temp;
}
```

Exercise: Linked list in Java

What is a linked list?

What are some uses of linked lists? (e.g. when are they useful?)

How do you implement a linked list in Java?

Example: Linked list C vs Java

```c
struct node {
  int val;
  struct node* next;
};

int main() {
  struct node n1 = {1, NULL};
  struct node n2 = {2, NULL};
  struct node n3 = {3, NULL};

  struct node* list;
  list = &n1;
  n1.next = &n2;
  n2.next = &n3;

  for (struct node* n = list; n != NULL; n = n->next) {
    printf("Val: %d\n", n->val);
  }
}
```

```java
class LinkedList {

  public static void main(String[] args) {
    Node n1 = new Node(1);
    Node n2 = new Node(2);
    Node n3 = new Node(3);

    Node list = null;
    list = n1;
    n1.next = n2;
    n2.next = n3;

    for (Node n = list; n != null; n = n.next) {
      System.out.println("Val: " + n.val);
    }
  }
}
```

Visualize the linked list from this code

```
struct node {
  int val;
  struct node* next;
};

int main() {
  struct node n1 = {1, NULL};
  struct node n2 = {2, NULL};
  struct node n3 = {3, NULL};

  struct node* list;
  list = &n1;
  n1.next = &n2;
  n2.next = &n3;

  for (struct node* n = list; n != NULL; n = n->next) {
    printf("Val: %d\n", n->val);
  }
}
```

Example: Dynamic Linked list (using malloc/free)

```
struct node {
  int val;
  struct node* next;
};

struct node* insert_front(int val, struct node* head)
{
  struct node* n = malloc(sizeof(struct node));
  if (n == NULL) {
    printf("ERROR: Out of space!\n");
    exit(1);
  }
  n->val = val;
  n->next = head;
  return n;
}
```

```
void print(struct node* list) {
  for (struct node* n = list; n != NULL; n = n->next) {
    printf("Val: %d\n", n->val);
  }
}
```

```
int main() {
  struct node* n3 = insert_front(2, NULL);
  struct node* n2 = insert_front(1, n3);
  struct node* n1 = insert_front(0, n2);

  free(n1);
  free(n2);
  free(n3);
}
```

Visualize: insert_front

Assume insert_front is called twice, like so:

```
struct node* n2 = insert_front(1, NULL);
struct node* n1 = insert_front(0, n2);
```

```
struct node {
  int val;
  struct node* next;
};

struct node* insert_front(int val, struct node* head)
{
  struct node* n = malloc(sizeof(struct node));
  if (n == NULL) {
    printf("ERROR: Out of space!\n");
    exit(1);
  }
  n->val = val;
  n->next = head;
  return n;
}
```

Visualize: print

```
void print(struct node* list) {
  for (struct node* n = list; n != NULL; n = n->next) {
    printf("Val: %d\n", n->val);
  }
}
```

Assume the list contains the numbers 0, 1, and 2.

Exercise: Visualize this program

```
void print(struct node* list) {
  for (struct node* n = list; n != NULL; n = n->next) {
    printf("Val: %d\n", n->val);
  }
}
```

Assume we have a linked list with 3 nodes. The first node has value 10, the second has value -3, and the third has value 5. Visualize the execution of `print`

Exercise: Draw the stack diagram

```
struct node* insert_front(int val, struct node* head)
{
  struct node* n = malloc(sizeof(struct node));
  if (n == NULL) {
    printf("ERROR: Out of space!\n");
    exit(1);
  }
  n->val = val;
  n->next = head;
  return n;
}
```

```
int main() {
   struct node* n1 = insert_front(0, NULL);

   free(n1);
   // draw the stack diagram here
}
```

Example: Dynamic linked list – version 2

```
struct node {
  int val;
  struct node* next;
};

void insert_front(int val, struct node* head) {
  struct node* n = malloc(sizeof(struct node));
  if (n == NULL) {
    printf("ERROR: Out of space!\n");
    exit(1);
  }
  n->val = val;
  n->next = head;
  head = n;
}
```

```
void print(struct node* list) {
  for (struct node* n = list; n != NULL; n = n->next) {
    printf("Val: %d\n", n->val);
  }
}

int main() {
  struct node* head = NULL;
  insert_front(0, head);
  insert_front(1, head);
  insert_front(2, head);
  print(head);
}
```

This doesn’t work. Why?

Example: Dynamic linked list – version 2

```
void insert_front(int val, struct node* head) {
  struct node* n = malloc(sizeof(struct node));
  if (n == NULL) {
    printf("ERROR: Out of space!\n");
    exit(1);
  }
  n->val = val;
  n->next = head;
  head = n;
}

int main() {
  struct node* head = NULL;
  insert_front(0, head);
  insert_front(1, head);
  print(head);
}
```

Example: Dynamic linked list – version 2 (fixed)

```
void insert_front(int val, struct node** head) {
  struct node* n = malloc(sizeof(struct node));
  if (n == NULL) {
    printf("ERROR: Out of space!\n");
    exit(1);
  }
  n->val = val;
  n->next = head;
  *head = n;
}

int main() {
  struct node* head = NULL;
  insert_front(0, &head);
  insert_front(1, &head);
  print(head);
}
```

Data Structure design in C

Summary: Use structs to encapsulate data and functions to perform operations

Exercise: Design a struct for a re-sizable string type. What functions would you need?

Aside: Writing C programs with multiple files

A **header** file includes definitions for structs, constants, and functions so that they can be re-used

Examples: stdio.h, stdlib.h

Define your own headers for your own code:

```
#include “myfile.h”
```

Demo: mystring.h

```
// Example header file structure
#ifndef _filename_H_ // define a macro to avoid redefinitions
#define _filename_H_

// define structs and constants
#define ARRAY_SIZE 32
struct dataT
{
   int x;
   char buffer[ARRAY_SIZE];
};

// use extern for functions
extern void init_data(struct dataT* data);

#endif // close #ifdef
```

Exercise: Binary tree

How would you implement a binary tree in C? What structs would you need? What functions would you need?