CS 211 : Thurs 01/11 (lecture 03)

Prof. Hummel
(he/him)

- **Topics**: **input, structs, pointers, dynamic arrays**

January 2024

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	31			

www.a-printable-calendar.com

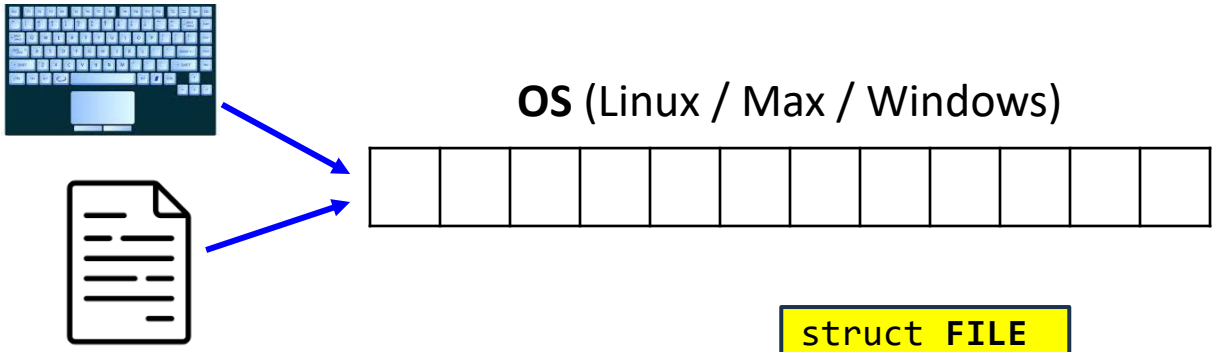
Northwestern University

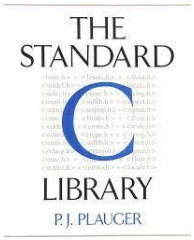
Notes:

- *Lecture slides available on Canvas*
- ***Project 01** due Friday @ 11:59pm, may be submitted up to 48 hours late (see syllabus). Gradescope is open for submissions (4 per day), test files are posted (Canvas/Piazza has link). Problems with EECS computers? Use replit.*
- ***HW 02** due Tuesday @ 11:59pm*
- *Watch for release of **Project 02** over the weekend*

how input works

- **Input is buffered by the OS**
- **C provides functions to call OS and access buffer**






```
struct FILE
{
  .
  .
  .
};
```

```
int fgetc(FILE* f)
{
  .
  .
  .

  return c;
}
```

Each call to **fgetc( )** advances buffer pointer within FILE structure…

Question #1

1) *Suppose the user types "apple" without the quotes and presses ENTER. What is output?*

```
int main()
{
  int c;

  while (fgetc(stdin) != 'e') {

     c = fgetc(stdin);
     printf("%c,", c);
  }

  return 0;
}
```

"stdin" is predefined pointer to the keyboard

A) a,p,p,l,e,

B) a,p,p,l,

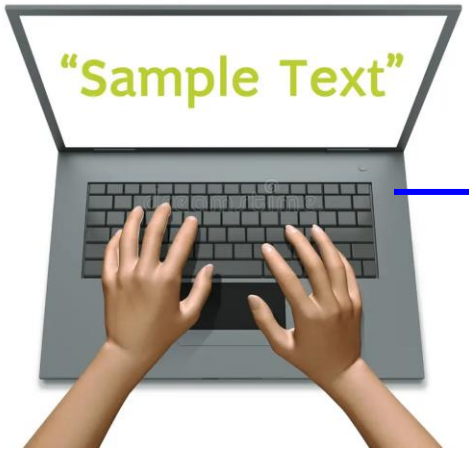
C) a,p,e

D) a,p,

E) p,l,

ungetc( )

- **ungetc(c, f) puts c back into the buffer pointed to by f…**



OS (Linux / Max / Windows)

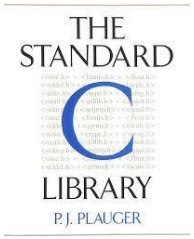
```
struct FILE
{
  .
  .
  .
};
```

```
int main()
{
    .
    .
    .

   ungetc(c, stdin);
```

```
int ungetc(int c, FILE* f)
{
  .
  .
  .

  return c;
}
```



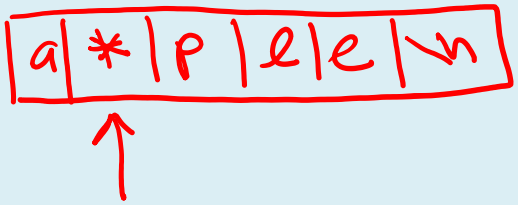
Question #2

2) *Suppose the user types "apple" without the quotes and presses ENTER. Which is the correct buffer and pointer after the ungetc?*

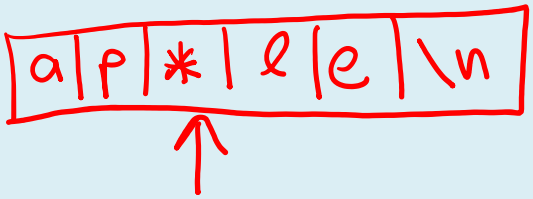
```
int main()
{
    int c;
    c = fgetc(stdin);
    c = fgetc(stdin);

    ungetc('*', stdin);
```

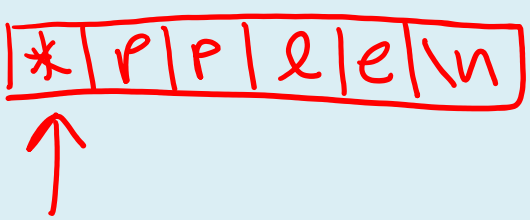
A)



B)



C)



Side-effects

- **fgetc( ) and ungetc( ) are examples of functions with "side-effects"**

- **Calling these functions may change memory**
  - *The buffer and buffer pointer*

Project 01: string literals

- **Here's an approach for handling string literals "..."**

```
if (c == '"')  // start of a string literal "..."
{
   T.id   = …
   T.line = …
   T.col  = …

   int i = 0;

   while (fgetc(input) != '"' && fgetc(input) != '\n') {

     // not yet at end, store and repeat:
     value[i] = fgetc(input);
     i++;

     (*colNumber)++;
   }
```

Project 01: input

- **Project 01 can input from the keyboard or a file…**

```
hummel@batgirl$ ./a.out
nuPython input (enter $ when you're done)>
print(x)
Token 25 ('print') @ (1, 1)
Token 1 ('(') @ (1, 6)
Token 25 ('x') @ (1, 7)
Token 2 (')') @ (1, 8)
```

```
hummel@batgirl$ ./a.out test01.py
Token 25 ('print') @ (1, 1)
Token 1 ('(') @ (1, 6)
Token 25 ('x') @ (1, 7)
Token 2 (')') @ (1, 8)
Token -1 ('{') @ (2, 1)
Token 25 ('a') @ (3, 3)
Token -1 ('=') @ (3, 5)
Token 25 ('b') @ (3, 7)
Token -1 ('}') @ (4, 1)
Token 0 ('$') @ (5, 1)
hummel@batgirl$
```

How does this work?

FILE* input

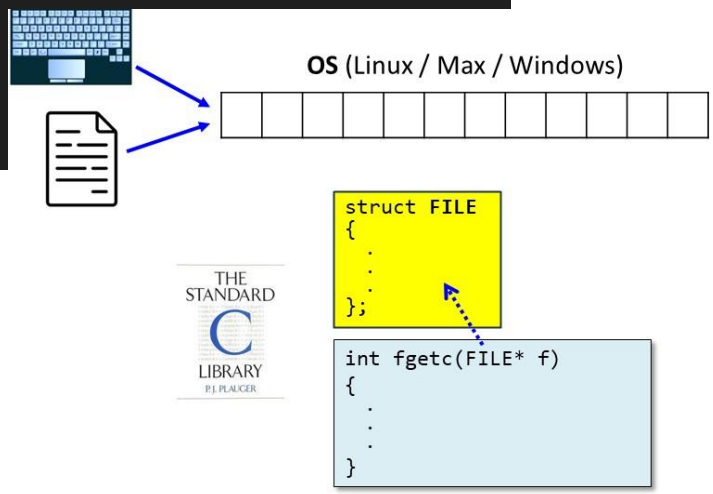
- **main( ) creates pointer to keyboard or file…**

```
20  //
21  // main
22  //
23  // usage: program.exe [filename.py]
24  //
25  // If a filename is given, the file is opened and serves as
26  // input to the scanner. If a filename is not given, then
27  // input is taken from the keyboard until $ is input.
28  //
29  int main(int argc, char* argv[])
30  {
31    FILE* input = NULL;
32    bool  keyboardInput = false;
33
34    if (argc < 2) {
35      //
36      // no args, just the program name:
37      //
38      input = stdin;
39      keyboardInput = true;
40    }
41    else {
42      //
43      // assume 2nd arg is a nuPython file:
44      //
45      char* filename = argv[1];
46
47      input = fopen(filename, "r");
48
```

scanner_nextToken( )

- **Scanner uses FILE pointer to read input stream…**

```
119  struct Token scanner_nextToken(FILE* input, int* lineNumber, int* colNumber, char* value)
120  {
121    assert(input != NULL);
122    assert(lineNumber != NULL);
123    assert(colNumber != NULL);
124    assert(value != NULL);
125
126    struct Token T;
127
128    //
129    // repeatedly input characters one by one until a token is found:
130    //
131    while (true)
132    {
133      //
134      // Get the next input character:
135      //
136      int c = fgetc(input);
137
```



Pointers

- **We are starting to see why C has pointers…**
- **Reason #1: abstraction**

I can write a single function that reads from keyboard or file --- the pointer hides the details of which one it is.

```
119  struct Token scanner_nextToken(FILE* input, int* lineNumber, int* colNumber, char* value)
120  {
121    assert(input != NULL);
122    assert(lineNumber != NULL);
123    assert(colNumber != NULL);
124    assert(value != NULL);
125
126    struct Token T;
127
128    //
129    // repeatedly input characters one by one until a token is found:
130    //
131    while (true)
132    {
133      //
134      // Get the next input character:
135      //
136      int c = fgetc(input);
137
```

Pointers

- **Reason #2: allows function to change memory**

The scanner function can advance the input buffer, the line #, the column #, and return the token value --- because of these pointers

```
119  struct Token scanner_nextToken(FILE* input, int* lineNumber, int* colNumber, char* value)
120  {
121    assert(input != NULL);
122    assert(lineNumber != NULL);
123    assert(colNumber != NULL);
124    assert(value != NULL);
125
126    struct Token T;
127
128    //
129    // repeatedly input characters one by one until a token is found:
130    //
131    while (true)
132    {
133      //
134      // Get the next input character:
135      //
136      int c = fgetc(input);
137
```

main( ) + scanner_nextToken( )

main( )

```
64   int lineNumber = -1;
65   int colNumber = -1;
66   char value[256] = "";
67   struct Token T;
68
69   //
70   // setup to start scanning:
71   //
72   scanner_init(&lineNumber, &colNumber, value);
73
74   if (keyboardInput)  // prompt the user if appropriate:
75   {
76     printf("nuPython input (enter $ when you're done)>\n");
77   }
78
79   //
80   // call scanner to process input token by token until we see ; or $
81   //
82   T = scanner_nextToken(input, &lineNumber, &colNumber, value);
83
84   while (T.id != nuPy_EOS)
85   {
86     printf("Token %d ('%s') @ (%d, %d)\n", T.id, value
87
88     T = scanner_nextToken(input, &lineNumber, &colNumb
89   }
90
```

```
119  struct Token scanner_nextToken(FILE* input, int* lineNumber, int* colNumber, char* value)
120  {
121    assert(input != NULL);
122    assert(lineNumber != NULL);
123    assert(colNumber != NULL);
124    assert(value != NULL);
125
126    struct Token T;
127
128    //
129    // repeatedly input characters one by one until a token is found:
130    //
131    while (true)
132    {
133      //
134      // Get the next input character:
135      //
136      int c = fgetc(input);
137
```

Pointers

- **Reason #3: building data structures**

Dynamic array:

```
int main()
{
  int* array;

  array = <<allocate a chunk of memory to start>>;

  .
  .  // if array gets full, allocate a bigger array and copy elements
  .
```

Linked-list:

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

```
int main()
{
  struct Node* list;

  list = NULL; // empty to start

  .
  .  // insert one node for each data element
  .
```

Live coding on replit.com

- Login to replit.com
- Open team…
- Open project "Lecture 03"

Dynamic arrays (part 01)

```
int main()
{
  .
  .
  .

  //
  // input numbers into an array:
  //
  int  capacity = 100;  // initial capacity
  int* A = (int*) malloc(capacity * sizeof(int));

  int N = 0;

  while (!feof(input)) {
    int number;
    int count = fscanf(input, "%d", &number);

    if (count != 1)  // input failed:
      break;

    A[N] = number;  // store number:
    N++;
  }

  //
  // loop complete, print stats and first/last values:
  //
  printf("number of values: %d\n", N);
  printf("capacity of array: %d\n", capacity);
  printf("first: %d\n", A[0]);
  printf("last: %d\n", A[N-1]);
```

- **Let's write a program that reads integers from a file and dynamically adapts to the file size**

```
if (N == capacity) { // it's full:
  capacity = N * 2;
  A = (int*) realloc(A, capacity * sizeof(int));
}
```

Question #3

3) *For the large text file with 80,000 numbers, what was the array capacity?*

A) 100

B) 80,000

C) 102,400

D) 1,000,000

Question #4

4) Do you think it will snow tomorrow?

A) Yes

B) No

What should I be working on?

1. ***Project #01*** *is due Friday @ 11:59pm…*
2. ***HW #02*** *is due Tuesday @ 11:59pm…*
3. *Watch for release of* ***Project 02*** *over the weekend*