Pointers

- A pointer in C holds the **memory address** of a value
  - the value of a pointer is an address
  - the value of the memory location pointed at can be obtained by “dereferencing the pointer” (retrieving the contents of that address)

\[
\begin{array}{c|c|c}
\text{memory location} & \text{address} \\
\hline
129 & 10000 \\
77 & 10001 \\
31542 & 10002 \\
108 & 10003 \\
97 & 10004 \\
542 & 10005 \\
100652 & 10006 \\
\end{array}
\]
C pointers vs. Java references

C pointers
• a pointer is the address of a memory location
  – no explicit type information associated with it
• arithmetic on pointers is allowed, e.g.:
  *(p+27)

Java references
• a reference is an alias for an object
  – references have associated type information
• arithmetic on references not allowed
Declaring pointer variables

• Two new operators (unary, prefix):
  – & : “address of”
  – * : “dereference”

• Declarations:
  – a pointer x to something of type T is declared as
    
    \[ T \ *x \]

Example:

\begin{verbatim}
    int *p; // p: pointer to an int
    char **w; // w: pointer to a pointer to a char
\end{verbatim}
Using pointer-related operators

- If $x$ is a variable, $&x$ is the address of $x$
- If $p$ is a pointer, $*p$ is the value of whatever points to
- $*(\&p) \equiv p$ always
Arrays

• An array in C is just a contiguous sequence of memory locations
  – size of each element depends on type
  – the length of the array is \textit{not} part of the array type
  – the language does not require that array accesses be checked to be within the array bounds
    • out-of-bound accesses result in bugs, security flaws ("buffer overflow vulnerabilities")
More arrays

• Consider an array declared as:

```c
int A[20];
```

  – the value of $A[i]$ is the contents of the memory location occupied by element $i$ of $A$;

  – the value of $A$ is the address of the array $A$, i.e., $&A[0]$);

• this does not have size information associated with it.
More arrays

• To pass an array as an argument to a function, you pass the array name
  – since the value of the array name is the address of the array, what is actually passed is a pointer to the array
• This does not have size information associated
  – the called function does not know how big the array is
  – need to provide a mechanism for callee to figure this out:
    • either pass the size of the array separately; or
    • terminate the array with a known value (e.g., 0)
\textbf{scanf()} and pointers

• To read input using scanf(), we have to provide:
  – a format string with conversion specifications (\%d, \%s, etc.) that says what kind of value is being read in; and
  – a pointer to (i.e., the address of) a memory area where the value is to be placed

• Reading in an integer:
  \begin{verbatim}
  int x;
  scanf("\%d", &x); // \&x \equiv \text{address of } x
  \end{verbatim}

• Reading in a string:
  \begin{verbatim}
  char str[...];
  scanf("\%s", str); // str \equiv \text{address of the array } str
  \end{verbatim}
Example 1

str_rev is a function of type “char *”, i.e., returns a pointer to a character

the argument is an array (its size is not part of its type)

array ≈ pointer

string library functions
Example 1...

```c
/* File: str_reverse.c
 * This program implements a function to reverse a string.
 */

#include <stdio.h>
#include <string.h>
/*
 * str_rev() returns a string that is the reverse of the argument string s.
 */
char *str_rev(char s[])
{
    int i, len, n;
    char *t;
    if (s == NULL) return NULL;
    t = strdup(s); /* allocates a new string t that duplicates s */
    len = strlen(s);
    for (i = 0, n = len-1; n >= 0; i++, n--) {
        t[n] = s[i];
    }
    t[len] = '\0';
    return t;
}

main()
{
    char s[32];
    while (scanf("%s", s) != EOF) {
        printf("the reverse of %s is %s\n", s, str_rev(s));
    }
}
```
Example 1...

```plaintext
% gcc str_reverse.c
% ./a.out
abcde
the reverse of abcde is edcba
abc123def
the reverse of abc123def is fed321cba
% ```
Example 2: string reversal using pointers

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

/* str_rev() returns a string that is the reverse of the argument string s. */
char *str_rev(char *s)
{
    int i, len, n;
    char *t, *ptr;

    if (s == NULL) return NULL;

    t = strdup(s); /* allocates a new string t that duplicates s */
    len = strlen(s);
    for (ptr = t+len-1; *s != '\0'; s++, ptr--) {
        *ptr = *s;
    }

    return t;
}

main()
{
    char s[32];

    while ( scanf("%s", s) != EOF ) {
        printf("the reverse of %s is %s\n", s, str_rev(s));
    }

    return 0;
}
```
Example 2...

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

/* str_rev() returns a string that is the reverse of the argument string s. */
char *str_rev(char *s)
{
    int i, len, n;
    char *t, *ptr;

    if (s == NULL) return NULL;

    t = strdup(s); /* allocates a new string t that duplicates s */
    len = strlen(s);
    for (ptr = t+len-1; *s != '\0'; s++, ptr--) {
        *ptr = *s;
    }

    return t;
}

main()
{
    char s[32];

    while ( scanf("%s", s) != EOF ) {
        printf("the reverse of %s is %s\n", s, str_rev(s));
    }

    return 0;
}
```
Example 2...

% gcc str_reverse-1.c
% ./a.out

abcde
the reverse of abcde is edcba

1234567
the reverse of 1234567 is 7654321
%
When 1 = 4

```c
#include <stdio.h>

cat ptr-arith.c

/* File: ptr-arith.c */

Purpose: To illustrate some effects of pointer arithmetic */

#include <stdio.h>

char  cvar;
int   ivar;
long long l1var;

int main() {
    char *cptr = &cvar;
    int *iptr = &ivar;
    long long *l1ptr = &l1var;

    long long val1, val2;
    val1 = cptr; cptr += 1; val2 = cptr;
    printf(">>> [char *]: old = %ld; new = %ld ... difference = %d\n", val1, val2, val2-val1);

    val1 = iptr; iptr += 1; val2 = iptr;
    printf(">>> [int *]: old = %ld; new = %ld ... difference = %d\n", val1, val2, val2-val1);

    val1 = l1ptr; l1ptr += 1; val2 = l1ptr;
    printf(">>> [long long *]: old = %ld; new = %ld ... difference = %d\n", val1, val2, val2-val1);

    return 0;
}
```

 pointers of different types

 pointer arithmetic:
 add 1 to pointers of different types
When 1 = 4...

-o: “put the output in the file specified, instead of the default a.out”

but each pointer was incremented by 1!!!
What’s going on

• Pointer arithmetic is performed relative to the size of the pointee type
  – for char* pointers, “+= 1” increments by 1
  – for int* pointers, “+= 1” increments by 4 (if size of int = 4)
  ★ in general, “+= 1” will increment a pointer by the size (in bytes) of the type being pointed at
    • analogously for other arithmetic

• Reason: portability:
  • want code to be able to step through an array of values without worrying about architecture-dependent issues of their size
Figuring out sizes: sizeof()

```c
#include <stdio.h>

char   cvar;
int    ivar;
long long llvar;

int main() {
    char *cptr = &cvar;
    int *iptr = &ivar;
    long long *llptr = &llvar;

    printf("Sizes: type variable *pointer pointer\n");
    printf("----- -------- ------- ------- \n");
    printf(" char: %d %d %d %d \n",
            sizeof(char), sizeof(cvar), sizeof(*cptr), sizeof(*cptr));
    printf(" int: %d %d %d %d \n",
            sizeof(int), sizeof(ivar), sizeof(*iptr), sizeof(*iptr));
    printf(" long long: %d %d %d %d \n",
            sizeof(long long), sizeof(llvar), sizeof(*llptr), sizeof(*llptr));

    return 0;
}
```

- `sizeof()` invoked with a type name
- `sizeof()` invoked with a variable name
- `sizeof()` invoked with a pointer dereference
Figuring out sizes: sizeof()

```c
char    cvar;
int     ivar;
long long llvar;

int main() {
    char *cptr = &cvar;
    int *iptr = &ivar;
    long long *llptr = &llvar;

    printf("Sizes: type variable *pointer pointer\n");
    printf("----- ---- -------- -------- --------\n");
    printf(" char:    %d    %d %d %d\n",
           sizeof(char), sizeof(cvar), sizeof(*cptr), sizeof(cptr));
    printf(" int:    %d    %d %d %d\n",
           sizeof(int), sizeof(ivar), sizeof(*iptr), sizeof(iptr));
    printf(" long long: %d    %d %d %d\n",
           sizeof(long long), sizeof(llvar), sizeof(*llptr), sizeof(llptr));

    return 0;
}
```

eanson@lectura:~/cs352/fall16/slides/programs$ gcc sizeof.c 2>res
eanson@lectura:~/cs352/fall16/slides/programs$ a.out

Sizes: type variable *pointer pointer

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>char:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>int:</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>long long:</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td></td>
</tr>
</tbody>
</table>
More sizeof()

- sizeof() applied to an array returns the total size of that array
  - but be careful about implicit array/pointer conversions

```c
% cat sizeof-1.c
/*
 * File: sizeof-1.c
 * Purpose: illustrate the use of sizeof on arrays and pointers
 */
#include <stdio.h>

int f(int X[]) {
    return (int)sizeof(X);
}

int main() {
    int A[20];
    printf("sizeof(int) = %d; sizeof(A) = %d ... f returns %d\n",
           (int)sizeof(int), (int)sizeof(A), f(A));
    return 0;
}

% gcc -Wall sizeof-1.c
% ./a.out
sizeof(int) = 4; sizeof(A) = 80 ... f returns 8
```
Dereferencing + updating pointers

A common C idiom is to use an expression that

– gives the value of what a pointer is pointing at; and
– updates the pointer to point to the next element:

\[ \ast p++ \]

parsed as: \[ \ast ( p++ ) \]

– similarly: \( \ast p-- \), \( *++p \), etc.

evaluates to: value of \( p \) = some address \( a \)
(side effect: \( p \) incremented by ‘++’)

evaluates to: contents of location \( a = \ast p \)
(side effect: \( p \) incremented by ‘++’).
Walking a pointer down an array

dereference the pointer to access memory, then increment the pointer
*p++ vs. (*p)++

After $x = *p++$

After $x = (*p)++$
Two common pointer problems

• Uninitialized pointers
  – the pointer has not been initialized to point to a valid location

• Dangling pointers
  – the pointer points at a memory location that has actually been deallocated
Background: Runtime Memory Organization

Layout of an executing process’s virtual memory:

- **Operating System**: 0xffffffff high addresses
- **Stack** (grows downwards)
- **Memory Mapped Files**
- **Heap** (grows upwards)
- **Global Data**
- **Code**: 0x00000000 low addresses
Background: Runtime Memory Organization

Code:

```java
p(...) {
    ... 
    q(...);
    s(...);
}

q(...) {
    ... 
    r(...);
}

r(...) {
    ... 
}
```

Runtime stack:

- **p’s stack frame**
- **p’s caller’s stack frame**

Stack growth:

- Top of stack
Background: Runtime Memory Organization

Code:

```c
p(...) {
  ...
  q(...);
  s(...);
}

q(...) {
  ...
  r(...);
}

r(...) {
  ...
}
```

Runtime stack:

- stack growth
- p's caller's stack frame
- p's stack frame
- top of stack
Background: Runtime Memory Organization

Code:

```c
p(...) {
    ...
    q(...);
    s(...);
}

q(...) {
    ...
    r(...);
}

r(...) {
    ...
}
```

Runtime stack:

- `p's caller's stack frame`
- `p's stack frame`
- `q's stack frame`

Top of stack

Stack growth
Background: Runtime Memory Organization

Code:
```c
p(...) {
    ...
    q(...);
    s(...);
}

q(...) {
    ...
    r(...);
}

r(...) {
    ...
}
```

Runtime stack:
- "p's caller's stack frame"
- "p's stack frame"
- "q's stack frame"
- "r's stack frame"

- stack growth
- top of stack
Background: Runtime Memory Organization

Code:

```c
p(...) {
    ...
    q(...);
    s(...);
}

q(...) {
    ...
    r(...);
}

r(...) {
    ...
    ... s(...) {
        ...
    }
}
```

Runtime stack:

- **p's caller's stack frame**
- **p's stack frame**
- **q's stack frame**
- **r's stack frame** (deallocated)

Stack growth:
Background: Runtime Memory Organization

Code:

```c
p(...) {
    ...
    q(...);
    s(...);
}

q(...) {
    ...
    r(...);
}

r(...) {
    ...
}

s(...) {
    ...
}
```

Runtime stack:

- Stack growth
- p's caller's stack frame
- p's stack frame
- top of stack
- q's stack frame
- r's stack frame
Background: Runtime Memory Organization

Code:

```plaintext
p(...) {
    ...
    q(...);
    s(...);
}

q(...) {
    ...
    r(...);
}

r(...) {
    ...
    s(...) {
        ...
    }
}
```

Runtime stack:

- **Top of stack**
- **r's stack frame**
- **s's stack frame**
- **q's stack frame**
- **p's stack frame**
- **p's caller's stack frame**

*Stack growth*
Uninitialized pointers: Example

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

char *str;

int main() {  
  scanf("%s", str);  
  printf("String read: %s\n", str);
  printf("Length of string: %d\n", (int)strlen(str));
  return 0;
}

% gcc -Wall uninit-ptr-1.c  
% ./a.out  
abcde  
Segmentation fault
```

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

char *str = array[256];

int main() {  
  scanf("%s", str);
  printf("String read: %s\n", str);
  printf("Length of string: %d\n", (int)strlen(str));
  return 0;
}

% gcc -Wall uninit-ptr-1-fixed.c  
% ./a.out  
abcdefg  
String read: abcdefg
Length of string: 7
```

str was never initialized to point to anything

fix: initialize str
Dangling pointers

What’s wrong with this code?

```c
/* cat dangling-ptr-1.c */
/* File: dangling-ptr-1.c */
/* Purpose: To illustrate dangling pointers */

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

/* read_string(str) -- reads a string into buffer str. Returns str if a string was successfully read, NULL otherwise. */
char *read_string(char *str) {
    int status = scanf("%s", str);
    if (status > 0) {
        return str;
    } else {
        return NULL;
    }
}

/* my_read() -- reads a string into a buffer and returns a pointer to that buffer. */
char *my_read() {
    char buf[128];
    return read_string(buf);
}

int main() {
    char *string = my_read();
    printf(">> string: %s -- length = %d
", string, (int)strlen(string));
    return 0;
}

% gcc  -Wall  dangling-ptr-1.c
% ./a.out
abcdef
>> string: -- length = 1
% */

% gcc  -Wall  dangling-ptr-1.c
% ./a.out
```
Dangling pointers

What’s wrong with this code?

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

// read_string(str) -- reads a string into buffer str. Returns
// str if a string was successfully read, NULL otherwise.
char *read_string(char *str) {
    int status = scanf("%s", str);
    if (status > 0) {
        return str;
    } else {
        return NULL;
    }
}

// my_read() -- reads a string into a buffer and returns a pointer
// to that buffer.
char *my_read() {
    char buf[128];
    return read_string(buf);
}

int main() {
    char *string = my_read();
    printf("> string: %s -- length = %d\n", string, (int)strlen(string));
    return 0;
}
```

```bash
% gcc -Wall dangling-ptr-1.c
% ./a.out
abcdef
> string: -- length = 1
```
Dangling pointers

What’s wrong with this code?

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

// read_string(str) -- reads a string into buffer str. Returns
// str if a string was successfully read, NULL otherwise.
char *read_string(char *str) {
    int status = scanf("%s", str);
    if (status > 0) {
        return str;
    } else {
        return NULL;
    }
}

// my_read() -- reads a string into a buffer and returns a pointer
// to that buffer.
char *my_read() {
    char buf[10];
    return read_string(buf);
}

int main() {
    char *string = my_read();
    printf(">> string: %s -- length = %d\n", string, (int)strlen(string));
    return 0;
}
```

runtime stack

main

my_read

buf

str

c
b
a

string

read_string

read_string

str

buf

36
Dangling pointers

What’s wrong with this code?

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

// read_string(str) -- reads a string into buffer str. Returns
// str if a string was successfully read, NULL otherwise.
char *read_string(char *str) {
    int status = scanf("%s", str);
    if (status > 0) {
        return str;
    } else {
        return NULL;
    }
}

// my_read() -- reads a string into a buffer and returns a pointer
// to that buffer.
char *my_read() {
    char buf[128];
    return read_string(buf);
}

int main() {
    char *string = my_read();
    printf("> string: %s -- length = %d\n", string, (int)strlen(string));
    return 0;
}
```

Dangling pointers are problematic because they can lead to undefined behavior. In this code, the `read_string` function returns a `char *` which is a pointer to a string. However, after the function returns, the pointer is not nullified, and the caller continues to use this pointer to access the string. This can lead to accessing memory that is already on the stack or has been deallocated, leading to undefined behavior such as crashes or data corruption.

The diagram illustrates the runtime stack and the main function calling `my_read` which in turn calls `read_string`. The arrow from `read_string` to `my_read` indicates that the pointer returned by `read_string` is used by `my_read` without being nullified, which can result in a dangling pointer issue.

To fix this, you should nullify the pointer passed to `read_string()` after you're done with it, ensuring it's not used after it has been freed or reallocated.
Dangling pointers

What’s wrong with this code?

```c
// File: dangling-ptr-1.c
// Purpose: To illustrate dangling pointers

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

// read_string(str) -- reads a string into buffer str. Returns
// str if a string was successfully read, NULL otherwise.
char *read_string(char *str) {
    int status = scanf("%s", str);
    if (status > 0) {
        return str;
    } else {
        return NULL;
    }
}

// my_read() -- reads a string into a buffer and returns a pointer
// to that buffer.
char *my_read() {
    char buf[128];
    return read_string(buf);
}

int main() {
    char *string = my_read();
    printf("> string: %s -- length = %d\n", string, (int)strlen(string));
    return 0;
}
```

---

dangling pointer!
Dynamic memory allocation

• We can’t always anticipate how much memory to allocate
  – too little ⇒ program doesn’t work
  – too much ⇒ wastes space
• Solution: allocate memory at runtime as necessary
  – malloc(), calloc()
    • allocates memory in the heap area
  – free()
    • deallocates previously allocated heap memory block
Dynamic memory allocation: usage

void * : “generic pointer”

Usage:

```c
int *iptr = malloc(sizeof(int)) // one int

char *str = malloc(64) // an array of 64 chars
// (sizeof(char) = 1 by definition )

int *iarr = calloc(40, sizeof(int)) // a 0-initialized array of 40 ints
```
Dynamic memory allocation: example 1

Always check the return value of any system call that may fail.
Dynamic memory allocation: example 1

```c
#include <stdio.h>

int main(void) {
    int sz;
    scanf("%d", &sz); // read in the size of the vectors (should check for errors)

    // allocate space the vectors
    vec1 = malloc(sz*sizeof(int));
    vec2 = malloc(sz*sizeof(int));

    if (vec1 == NULL || vec2 == NULL) {
        fprintf(stderr, "Out of memory!\n");
        return(1);
    }

    // read in the vectors
    readVec(sz, vec1);
    readVec(sz, vec2);
    // compute and print the dot product
    printf("dot product = %d\n", dotprod(vec1, vec2, sz));

    return 0;
}

// readVec(vec, sz) -- reads in sz integers into the array vec.
// Assumes (does not check) that sz is positive and that vec
// is large enough to hold sz ints.
void readVec(int sz, int vec[]) {
    int i;
    for (i = 0; i < sz; i++) {
        scanf("%d", &vec[i]);
    }
}

// dot product
int dotprod(int vec1[], int vec2[], int sz) {
    int i, dot = 0;
    for (i = 0; i < sz; i++)
        dot += vec1[i] * vec2[i];
    return dot;
}
```

```
gcc -Wall ./dotprod.c
./a.out
1 2 3 4
5 6 7 8
dot product = 70
```
Dynamic memory allocation: example 2

```c
// Program: mystrcat.c
// Function: reads in an integer N, then N strings each of length at
// most 64. Concatenates these strings and prints the result.
// Purpose: Illustrate dynamic memory allocation via malloc().
// NOTE: The code below omits several checks for legality of values
// because the code needs to fit on the classroom screen.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// mystrcat(strs, n) -- strs is an array of n pointers to strings.
// concatenates the strings in strs and returns a pointer to the result.
char * mystrcat(char **strs, int n) {
    int i, len;
    char *buf;
    if (strs == NULL || n <= 0) {
        fprintf(stderr, "ERROR [mystrcat]: invalid argument(s)\n");
        exit(-1);
    }
    for (i = 0, len = 0; i < n; i++) {
        len += strlen(strs[i]); // should check that strs[i] != NULL
    }
    buf = realloc(len + 1, sizeof(char));
    if (buf == NULL) {
        fprintf(stderr, "Out of memory!\n");
        exit(-1);
    }
    for (i = 0; i < n; i++) {
        strcat(buf, strs[i]);
    }
    return buf;
}
int main() {
    --More--(71%)}
```

- figure out the total size of the concatenated strings
- allocate space
- concatenate the strings into buf
Dynamic memory allocation: example 2

```c
int main() {
    int n, i;
    char **strs, buf[65];

    scanf("%d", &n); // should check that n > 0 etc.

    strs = malloc(n * sizeof(char *));
    if (strs == NULL) {
        fprintf(stderr, "Out of memory!\n")
        exit(-1);
    }

    for (i = 0; i < n; i++) {
        scanf("%s", buf); // should check that something was read in
        strs[i] = strdup(buf);
    }

    printf(">> Concatenated string: %s\n", mystrcat(strs, n));
    return 0;
}
```

```
gcc -Wall ./mystrcat.c
c./a.out
123
abc
456
def
789
>> Concatenated string: 123abc456def789
```
Reading a line at a time

• scanf is nice. It parses the input and converts it to integers, floating points, etc.
• Sometimes we want to read a line at a time from the input. scanf is not so great for that.
• Fortunately there are functions that do read in a line at a time from the input. Two of them are:
  • getline()
  • fgets()
getline

GETLINE(3) Linux Programmer's Manual GETLINE(3)

NAME
getline, getdelim - delimited string input

SYNOPSIS
#include <stdio.h>

ssize_t getline(char **lineptr, size_t *n, FILE *stream);

ssize_t getdelim(char **lineptr, size_t *n, int delim, FILE *stream);

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

getline(), getdelim():
Since glibc 2.10:
   _POSIX_C_SOURCE >= 200809L || _XOPEN_SOURCE >= 700
Before glibc 2.10:
   _GNU_SOURCE

DESCRIPTION
getline() reads an entire line from stream, storing the address of the
buffer containing the text into *lineptr. The buffer is null-termin"
**DESCRIPTION**

getline() reads an entire line from stream, storing the address of the buffer containing the text into *lineptr. The buffer is null-terminated and includes the newline character, if one was found.

If *lineptr is NULL, then getline() will allocate a buffer for storing the line, which should be freed by the user program. (In this case, the value in *n is ignored.)

Alternatively, before calling getline(), *lineptr can contain a pointer to a malloc(3)-allocated buffer *n bytes in size. If the buffer is not large enough to hold the line, getline() resizes it with realloc(3), updating *lineptr and *n as necessary.

In either case, on a successful call, *lineptr and *n will be updated to reflect the buffer address and allocated size respectively.

getdelim() works like getline(), except that a line delimiter other than newline can be specified as the delimiter argument. As with getline(), a delimiter character is not added if one was not present in the input before end of file was reached.
The nice thing about getline is that it allocates the memory you need for you. You don’t have to worry about the input being too big to fit in the memory you’ve allocated.

The bad thing about getline is that it is a GNU extension and is not available for all systems. (It is available on lectura though)
fgets

GETS(3)

NAME

fgets, fgetc, getc, getchar, gets, ungetc - input of characters and strings

SYNOPSIS

#include <stdio.h>

int fgetc(FILE *stream);

char *fgets(char *s, int size, FILE *stream);

int getc(FILE *stream);

int getchar(void);

char *gets(char *s);

int ungetc(int c, FILE *stream);

DESCRIPTION

fgetc() reads the next character from stream and returns it as an
fgets

getchar() is equivalent to getc(stdin).

gets() reads a line from stdin into the buffer pointed to by s until either a terminating newline or EOF, which it replaces with a null byte ('\0'). No check for buffer overrun is performed (see BUGS below).

fgets() reads in at most one less than size characters from stream and stores them into the buffer pointed to by s. Reading stops after an EOF or a newline. If a newline is read, it is stored into the buffer. A terminating null byte ('\0') is stored after the last character in the buffer.

ungetc() pushes c back to stream, cast to unsigned char, where it is available for subsequent read operations. Pushed-back characters will be returned in reverse order; only one pushback is guaranteed.

Calls to the functions described here can be mixed with each other and with calls to other input functions from the stdio library for the same input stream.

For nonlocking counterparts, see unlocked_stdio(3).
fgets

- The nice thing about fgets() is that it’s a standard c function and should be available on all systems.
- The bad thing is that you have to allocate the memory for them to store the input lines into.
- Notice that one of the parameters to fgets is one less than the maximum number of characters to be read in. (See how this is slightly different than scanf?)

- There is a fprintf() and printf() if I want to just print to stdout. Likewise there is a fscanf() to read from any stream and scanf() to just read from stdin. fgets() lets me read from any stream, can I use gets() when I just want to read from stdin?
Don’t use gets()! Even the manual page says not to.

char cast to an int or EOF on end of file or error.

gets() and fgets() return s on success, and NULL on error or when end of file occurs while no characters have been read.

ungetc() returns c on success, or EOF on error.

CONFORMING TO

So why is it in the library?

BUGS

Never use gets(). Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer, it is extremely dangerous to use. It has been used to break computer security. Use fgets() instead.

It is not advisable to mix calls to input functions from the stdio library with low-level calls to read(2) for the file descriptor associated with the input stream; the results will be undefined and very probably not what you want.

Manual page fgets(3) line 55 (press h for help or q to quit)
• A **struct** is
  – an *aggregate* data structure, i.e., a collection of other data;
  – can contain components ("fields") of different types
    • by contrast, arrays contain components of the same type
  – fields are accessed by name
    • by contrast, array elements are accessed by position

• Unlike Java classes, a **struct** can only contain data, not code.
Declaring structs

- A node for a linked list of integers:

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

optional “structure tag” – used to refer to the structure
Accessing structure fields

• Given
  – a struct `s` containing a field `f`
  to access `f`, we write `s.f`

  **Example:**
  ```
  struct foo {
    int count, bar[10];
  } x, y;
  x.count = y.bar[3];
  ```

• Given
  – a pointer `p` to a struct `s` containing a field `f`
  to access `f` we write `p->f` // eqvt. to: `(*p).f`

  **Example:**
  ```
  struct foo {
    int count, bar[10];
  } *p, *q;
  p->count = q->bar[3];
  ```
Example: sorting a linked list of strings

```c
/*
 * File: sort_strings.c
 * Purpose: read in a number of strings from stdin until EOF is encountered;
 * sort the strings in alphabetical order, then print out the result.
 * Illustrates the use of structs, dynamic data structures.
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct s {
    char *str;
    struct s *next;
};
struct s *list_hd = NULL;
```

declares `list_hd` as “pointer to something of type `struct s`”

```
struct s
  str
  next
```
points to a string

points to another list node
Example: sorting a linked list of strings

```c
struct s {
    char *str;
    struct s *next;
};
struct s *list_hd = NULL;

/*
  * read_string() -- reads in a string from stdin and adds it to the list.
  * Returns a pointer to the linked-list node for that string, if one was
  * created; NULL otherwise.
  */
struct s *read_string() {
    struct s *tnpnode;
    char buf[64];
    int status;

    status = scanf("%s", buf);
    if (status == EOF) {
        return NULL;
    }
    tnpnode = malloc(sizeof(struct s));
    if (tnpnode == NULL) {
        fprintf(stderr, "Out of memory!\n");
        exit(1);
    }
    tnpnode->str = strdup(buf);
    tnpnode->next = list_hd;
    list_hd = tnpnode;
    return tnpnode;
}
```

allocate memory for a list node

amount allocated = size of the struct (not the pointer to the struct)
Example: sorting a linked list of strings

```c
struct s {
    char *str;
    struct s *next;
};
struct s *list_hd = NULL;

/*
 * read_string() -- reads in a string from stdin and adds it to the list.
 * Returns a pointer to the linked-list node for that string, if one was
 * created; NULL otherwise.
 */
struct s *read_string() {
    struct s *tmpnode;
    char buf[64];
    int status;

    status = scanf("%s", buf);
    if (status == EOF) {
        return NULL;
    }

    tmpnode = malloc(sizeof(struct s));
    if (tmpnode == NULL) {
        fprintf(stderr, "Out of memory!\n");
        exit(1);
    }

    tmpnode->str = strdup(buf);
    tmpnode->next = list_hd;
    list Hd = tmpnode;

    return tmpnode;
}
```

- fill in the fields of the newly allocated struct
- add it to the head of the linked list

**tmpnode, buf** will get deallocated

does this cause any problems?
Example: sorting a linked list of strings

```c
/*
 * sort_list() -- sorts the list of strings in alphabetical order, so that
 * list_hd points to the first string in this order. This function uses
 * a straightforward bubble sort algorithm.
 */
void sort_list() {
    struct s *ptr1, *ptr2;
    char *tmp;
    for (ptr1 = list_hd; ptr1 != NULL; ptr1 = ptr1->next) {
        for (ptr2 = ptr1->next; ptr2 != NULL; ptr2 = ptr2->next) {
            if (strcmp(ptr1->str, ptr2->str) > 0) {
                // ptr1->str is "greater than" ptr2->str -- swap them
                tmp = ptr1->str;
                ptr1->str = ptr2->str;
                ptr2->str = tmp;
            }
        }
    }
}

/*
 * print_list() -- prints out the strings in the list one per line
 */
void print_list() {
    struct s *ptr;
    printf("----- list contents -----\n");
    for (ptr = list_hd; ptr; ptr = ptr->next) {
        printf("%s\n", ptr->str);
    }
}

int main() {
    while (read_string() != NULL) {
        // loop, repeatedly calling read_string(), until it encounters EOF and returns NULL
    }
    --More--(97%)
```
Example: sorting a linked list of strings

```c
int main() {
    while (read_string() != NULL) {
        // loop, repeatedly calling read_string(), until it encounters EOF and returns NULL
    }
    sort_list();
    print_list();
    return 0;
}
```

Operator Precedence and Associativity

• Operator precedence and associativity define how an expression is parsed and evaluated
  – The text (King, *C Programming: A Modern Approach*), Appendix A has a full list of all C operator precedences

• Some highlights: in decreasing order of precedence:
  – postfix expressions ( [ ] ( ) -> . ++_postfix  --_postfix )
  – unary expressions ( ++_prefix  --_prefix  &  *  +  -  ~  !  sizeof )
  – type cast
  – arithmetic: multiplicative  ▶ additive  ▶ bit-shift
  – relational (not all of the same precedence)
  – bitwise operators (not all of the same precedence)
Operator Precedence Examples

- Decreasing order of precedence:
  - postfix expressions
    - [ ] ( ) -> · ++_post --_post
  - unary expressions
    - ++_pre --_pre & *_deref + - ~ ! sizeof
  - type cast
  - arithmetic
  - ...

How are these parsed?

- *p++
  - ++ binds tighter than *:
    - *(p++) not: (*p)++

- *p->q
  - -> binds tighter than *:
    - *(p->q) not: (*p)->q

- *A[10]
  - [ ] binds tighter than *:
    - *(A[10]) not: (*A)[10]

- *p->q++
  - -> and ++ left-associative:
    - *( (p->q) ++ )