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)
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 \textit{x} to something of type \textit{T} is declared as
    \begin{align*}
    T \; *x
    \end{align*}

Example:
\begin{itemize}
\item int \*p; \quad \text{\texttt{// p: pointer to an int}}
\item char **w; \quad \text{\texttt{// w: pointer to a pointer to a char}}
\end{itemize}
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 *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")
• 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)
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:

  ```
  int x;
  scanf("%d", &x); // &x = address of x
  ```

• Reading in a string:

  ```
  char str[...];
  scanf("%s", str); // str = address of the array str
  ```
Example 1

```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;
    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));
    }
}
```

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...

```bash
% 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>

/* strRev() returns a string that is the reverse of the argument string s. */
char *strRev(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, strRev(s));
    }

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

```c
% cat str_reverse-1.c
/* File: str_reverse-1.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, *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>

char cvar;
int ivar;
long long llvar;

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

    long long val1, val2;
    val1 = cptr; cptr += 1; val2 = cptr;
    printf("\%ld\n", val1, val2, val2-val1);

    val1 = iptr; iptr += 1; val2 = iptr;
    printf("\%ld\n", val1, val2, val2-val1);

    val1 = llptr; llptr += 1; val2 = llptr;
    printf("\%ld\n", val1, val2, val2-val1);

    return 0;
}
```

pointers of different types

pointer arithmetic: add 1 to pointers of different types
- **When 1 = 4…**

```
% gcc ./ptr-arith.c -o ptr-arith
./ptr-arith.c: In function `main':
./ptr-arith.c:18: warning: assignment makes integer from pointer without a cast
./ptr-arith.c:18: warning: assignment makes integer from pointer without a cast
./ptr-arith.c:22: warning: assignment makes integer from pointer without a cast
./ptr-arith.c:22: warning: assignment makes integer from pointer without a cast
./ptr-arith.c:26: warning: assignment makes integer from pointer without a cast
./ptr-arith.c:26: warning: assignment makes integer from pointer without a cast

% ../ptr-arith
>> [char *]: old = 6294068; new = 6294069 ... difference = 1
>> [int *]: old = 6294064; new = 6294068 ... difference = 4
>> [long long *]: old = 6294056; new = 6294064 ... difference = 8
```

- **-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   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 *l1lptr = &llvar;

    printf("Sizes: type variable *pointer pointer\n");
    printf("----- ---- ------- ------ ------
");
    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(*l1lptr), sizeof(l1lptr));

    return 0;
}
```

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

eanson@lectura:~/cs352/fall16/slides/programs$ a.out

<table>
<thead>
<tr>
<th>Sizes:</th>
<th>type</th>
<th>variable</th>
<th>*pointer</th>
<th>pointer</th>
</tr>
</thead>
<tbody>
<tr>
<td>char:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>int:</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>long long:</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</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
#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", sizeof(int), (int)sizeof(A), f(A));
    return 0;
}
```

what is passed to f() is a pointer, not the whole array
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

```c
int main() {
    int iarray[100], n, num, status, *iptr, sum;
    // read a bunch of numbers, stop when a 0 is read.
    for (iptr = iarray, n = 0; n < 100; n++) {
        status = scanf("%d", &num);
        if (status == 0 || num == 0) {
            break;
        }
        *iptr++ = num;
    }
    // now add the numbers
    for (iptr = iarray, sum = 0; n > 0; n--) {
        sum += *iptr++;
    }
    printf("sum = %d\n", sum);
    return 0;
}
```

% cat array-walk.c
/*
 * File: array-walk.c
 * Purpose: Illustrate walking down an array with a pointer
 */
#include <stdio.h>

% gcc -Wall array-walk.c
% ./a.out
1 2 3 4 0
sum = 10
*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**
- **Stack** (grows downwards)
- **Memory mapped files**
- **Heap** (grows upwards)
- **Global data**
- **Code**

addresses:
- High addresses: 0xffffffff
- Low addresses: 0x00000000
Background: Runtime Memory Organization

Runtime stack:

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

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

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

```
Background: Runtime Memory Organization

Code:

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

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

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

Runtime stack:

- **p's caller's stack frame**
- **p's stack frame**
- Stack growth

Top of stack
Background: Runtime Memory Organization

Code:

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

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

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

Runtime stack:

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

Stack growth:

- Top of stack

Diagram showing the memory organization with stack frames for `p`, `q`, and `r`. The stack grows downwards with each function call.
Code:

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

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

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

Runtime stack:

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

**Top of stack**

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

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

r(...) {
    ...
}
```
Background: Runtime Memory Organization

Code:

```
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**

Stack growth from top of stack to bottom.
Background: Runtime Memory Organization

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

Runtime stack:
```
p’s caller’s stack frame

p’s stack frame

q’s stack frame

s’s stack frame

r’s stack frame

stack growth

top of stack
```
Uninitialized pointers: Example

```
#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;
}
```

str was never initialized to point to anything

---

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

char *str;
char array[256];

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

fix: initialize str
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;
}
```

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

char *read_string(char *str) {  
    int status = scanf("%s", str);  
    if (status > 0) {  
        return str;  
    }  
    else {  
        return NULL;  
    }  
}

char *my_read() {  
    char buf[128];  
    return read_string(buf);  
}

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

```sh
$ 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[128];
   return read_string(buf);
}

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

What's wrong with this code?

- The function `my_read()` does not handle the case where the string is not read successfully.
- The variable `buf` is not initialized, which can lead to undefined behavior.
- The string is not null-terminated, which can lead to issues when using it.

To fix these issues:

1. Initialize `buf` before using it.
2. Check if the string was successfully read.
3. Ensure the string is null-terminated.
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;
}
```

```
% 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\n", string, (int)strlen(string));
    return 0;
}
% 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[128];
    return read_string(buf);
}

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

runtime stack

string

buf

str

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
```

DESCRIPTION

calloc() allocates memory and returns a pointer to the allocated memory. If the requested size is 0, then the function returns NULL. If there is insufficient memory, nothing is allocated.

malloc() allocates memory and returns a pointer to the allocated memory. Although the returned pointer is unique, it is not necessarily in a known state, so it is not suitable for use as a null pointer.

free() frees the memory that had been allocated by a previous call to malloc(), provided that the memory has not already been deallocated.

realloc() changes the memory allocated by malloc(). The contents will not be guaranteed to be compatible with the reallocated memory if malloc(size) is called. If realloc() is called with a size of 0, the memory is freed.

RETURN VALUE

For calloc() and malloc(), the value returned is a pointer to the allocated memory, which is suitably aligned for any kind of variable, or NULL if the request fails.
Dynamic memory allocation: example 1

```c
/* File: dotprod.c
 * Purpose: read in an integer N, then two vectors of N ints each.
 * Print out the dot product of the two vectors.
 * Illustrates the use of dynamic memory allocation using malloc */
#include <stdio.h>
#include <stdlib.h>

void readVec(int sz, int vec[]);

// dotprod(vec1, vec2, sz) -- computes the dot product of two
// integer vectors vec1 and vec2, each of size sz.
int dotprod(int *vec1, int *vec2, int sz) {
    int i, dp;
    for (i = 0, dp = 0; i < sz; i++) {
        dp += vec1[i] * vec2[i];
    }
    return dp;
}

int main() {
    int *vec1, *vec2, sz;
    scanf("%d", &sz); // read in the size of the vectors (s)
    // 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;
}
```

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

```c
#include <stdio.h>

int main()
{
    int sz, dotprod, i;

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

    // allocate space for 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]);
    }
}
```

```bash
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 = calloc(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;
}
```

- figure out the total size of the concatenated strings
- allocate space
- concatenate the strings into buf
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
./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)):

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

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

• 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) Linux Programmer's Manual 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
fgets() 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

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.

So why is it in the library?
Structs

• 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;
}
```

"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:

```c
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:

```c
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` points to a string
- `next` 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 *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;
}
```
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%)
}
```

traverse the list

compare strings by lexicographic ordering

idiomatic C:
“iterate as long as ptr ≠ NULL”
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;
}
```

```text
input strings

sorted output
```

```text
abc
abbott
aardvark
AMPERSAND
mnop
------ list contents ------
AMPERSAND
aardvark
abbott
abc
mnop
pqr
uvwxyz
zzzzz
```
typedefs

• Allow us to define *aliases* for types

• Syntax:

  ```
  typedef old_type_name new_type_name;
  ```

  • `new_type_name` becomes an alias for `old_type_name`

• Example:

  – typedef int BasePay;
  – typedef struct node {
    int value;
    struct node *next;
  } node;
defines “wcnode” as an alias for “struct wc”

we can use “wcnode” in place of “struct wc”

but not here, since “wcnode” has not yet been defined
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
    \[ ] ( ) -> \cdot ++_{\text{post}} --_{\text{post}}
  – unary expressions
    ++_{\text{pre}} --_{\text{pre}} \& *_{\text{deref}} + - ~ ! \text{sizeof}
  – type cast
  – arithmetic
  – ...

How are these parsed?
• \(*p++\)
  \(\text{++ binds tighter than *:}\)
  \(*(p++)\) \text{not:} \((p)++\)

• \(*p->q\)
  \(\rightarrow \text{binds tighter than *:}\)
  \(*(p->q)\) \text{not:} \((p)->q\)

• \(*A[10]\)
  \([\ ] \text{binds tighter than *:}\)
  \(*(A[10])\) \text{not:} \((A)[10]\)

• \(*p->q++\)
  \(\rightarrow \text{and ++ left-associative:}\)
  \(*((p->q)++))\)