```c
/*****************************************************************************
 * CS50 Library 6
 * https://manual.cs50.net/library/
 *
 * Based on Eric Roberts' genlib.c and simpio.c.
 *
 * Copyright (c) 2013,
 * Glenn Holloway <holloway@eecs.harvard.edu>
 * David J. Malan <malan@harvard.edu>
 * All rights reserved.
 *
 * BSD 3-Clause License
 * http://www.opensource.org/licenses/BSD-3-Clause
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of CS50 nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
 * IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
 * TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *****************************************************************************/

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

#include "cs50.h"

/**
 * Reads a line of text from standard input and returns the equivalent
```

```
 * char; if text does not represent a char, user is prompted to retry.
 * Leading and trailing whitespace is ignored.  If line can't be read,
 * returns CHAR_MAX.
 */
char GetChar(void)
{
    // try to get a char from user
    while (true)
    {
        // get line of text, returning CHAR_MAX on failure
        string line = GetString();
        if (line == NULL)
        {
            return CHAR_MAX;
        }

        // return a char if only a char (possibly with
        // leading and/or trailing whitespace) was provided
        char c1, c2;
        if (sscanf(line, " %c %c", &c1, &c2) == 1)
        {
            free(line);
            return c1;
        }
        else
        {
            free(line);
            printf("Retry: ");
        }
    }
}

/**
 * Reads a line of text from standard input and returns the equivalent
 * double as precisely as possible; if text does not represent a
 * double, user is prompted to retry.  Leading and trailing whitespace
 * is ignored.  For simplicity, overflow and underflow are not detected.
 * If line can't be read, returns DBL_MAX.
 */
double GetDouble(void)
{
    // try to get a double from user
    while (true)
    {
        // get line of text, returning DBL_MAX on failure
        string line = GetString();
        if (line == NULL)
        {
```

```
 97.             return DBL_MAX;
 98.         }
 99.
100.         // return a double if only a double (possibly with
101.         // leading and/or trailing whitespace) was provided
102.         double d; char c;
103.         if (sscanf(line, " %lf %c", &d, &c) == 1)
104.         {
105.             free(line);
106.             return d;
107.         }
108.         else
109.         {
110.             free(line);
111.             printf("Retry: ");
112.         }
113.     }
114. }
115.
116. /**
117.  * Reads a line of text from standard input and returns the equivalent
118.  * float as precisely as possible; if text does not represent a float,
119.  * user is prompted to retry.  Leading and trailing whitespace is ignored.
120.  * For simplicity, overflow and underflow are not detected.  If line can't
121.  * be read, returns FLT_MAX.
122.  */
123. float GetFloat(void)
124. {
125.     // try to get a float from user
126.     while (true)
127.     {
128.         // get line of text, returning FLT_MAX on failure
129.         string line = GetString();
130.         if (line == NULL)
131.         {
132.             return FLT_MAX;
133.         }
134.
135.         // return a float if only a float (possibly with
136.         // leading and/or trailing whitespace) was provided
137.         char c; float f;
138.         if (sscanf(line, " %f %c", &f, &c) == 1)
139.         {
140.             free(line);
141.             return f;
142.         }
143.         else
144.         {
```

```
            free(line);
            printf("Retry: ");
        }
    }
}

/**
 * Reads a line of text from standard input and returns it as an
 * int in the range of [-2^31 + 1, 2^31 - 2], if possible; if text
 * does not represent such an int, user is prompted to retry.  Leading
 * and trailing whitespace is ignored.  For simplicity, overflow is not
 * detected.  If line can't be read, returns INT_MAX.
 */
int GetInt(void)
{
    // try to get an int from user
    while (true)
    {
        // get line of text, returning INT_MAX on failure
        string line = GetString();
        if (line == NULL)
        {
            return INT_MAX;
        }

        // return an int if only an int (possibly with
        // leading and/or trailing whitespace) was provided
        int n; char c;
        if (sscanf(line, " %i %c", &n, &c) == 1)
        {
            free(line);
            return n;
        }
        else
        {
            free(line);
            printf("Retry: ");
        }
    }
}

/**
 * Reads a line of text from standard input and returns an equivalent
 * long long in the range [-2^63 + 1, 2^63 - 2], if possible; if text
 * does not represent such a long long, user is prompted to retry.
 * Leading and trailing whitespace is ignored.  For simplicity, overflow
 * is not detected.  If line can't be read, returns LLONG_MAX.
 */
```

```
193. long long GetLongLong(void)
194. {
195.     // try to get a long long from user
196.     while (true)
197.     {
198.         // get line of text, returning LLONG_MAX on failure
199.         string line = GetString();
200.         if (line == NULL)
201.         {
202.             return LLONG_MAX;
203.         }
204.
205.         // return a long long if only a long long (possibly with
206.         // leading and/or trailing whitespace) was provided
207.         long long n; char c;
208.         if (sscanf(line, " %lld %c", &n, &c) == 1)
209.         {
210.             free(line);
211.             return n;
212.         }
213.         else
214.         {
215.             free(line);
216.             printf("Retry: ");
217.         }
218.     }
219. }
220.
221. /**
222.  * Reads a line of text from standard input and returns it as a
223.  * string (char*), sans trailing newline character.  (Ergo, if
224.  * user inputs only "\n", returns "" not NULL.)  Returns NULL
225.  * upon error or no input whatsoever (i.e., just EOF).  Leading
226.  * and trailing whitespace is not ignored.  Stores string on heap
227.  * (via malloc); memory must be freed by caller to avoid leak.
228.  */
229. string GetString(void)
230. {
231.     // growable buffer for chars
232.     string buffer = NULL;
233.
234.     // capacity of buffer
235.     unsigned int capacity = 0;
236.
237.     // number of chars actually in buffer
238.     unsigned int n = 0;
239.
240.     // character read or EOF
```

```
    int c;

    // iteratively get chars from standard input
    while ((c = fgetc(stdin)) != '\n' && c != EOF)
    {
        // grow buffer if necessary
        if (n + 1 > capacity)
        {
            // determine new capacity: start at 32 then double
            if (capacity == 0)
            {
                capacity = 32;
            }
            else if (capacity <= (UINT_MAX / 2))
            {
                capacity *= 2;
            }
            else
            {
                free(buffer);
                return NULL;
            }

            // extend buffer's capacity
            string temp = realloc(buffer, capacity * sizeof(char));
            if (temp == NULL)
            {
                free(buffer);
                return NULL;
            }
            buffer = temp;
        }

        // append current character to buffer
        buffer[n++] = c;
    }

    // return NULL if user provided no input
    if (n == 0 && c == EOF)
    {
        return NULL;
    }

    // minimize buffer
    string minimal = malloc((n + 1) * sizeof(char));
    strncpy(minimal, buffer, n);
    free(buffer);

```

```
289.     // terminate string
290.     minimal[n] = '\0';
291.
292.     // return string
293.     return minimal;
294. }
```

```
/*****************************************************************************
 * CS50 Library 6
 * https://manual.cs50.net/library/
 *
 * Based on Eric Roberts' genlib.c and simpio.c.
 *
 * Copyright (c) 2013,
 * Glenn Holloway <holloway@eecs.harvard.edu>
 * David J. Malan <malan@harvard.edu>
 * All rights reserved.
 *
 * BSD 3-Clause License
 * http://www.opensource.org/licenses/BSD-3-Clause
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 * * Redistributions of source code must retain the above copyright notice,
 *   this list of conditions and the following disclaimer.
 * * Redistributions in binary form must reproduce the above copyright
 *   notice, this list of conditions and the following disclaimer in the
 *   documentation and/or other materials provided with the distribution.
 * * Neither the name of CS50 nor the names of its contributors may be used
 *   to endorse or promote products derived from this software without
 *   specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS
 * IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED
 * TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
 * PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
 * HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
 * SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED
 * TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
 * PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
 * SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *****************************************************************************/

#ifndef _CS50_H
#define _CS50_H

#include <float.h>
#include <limits.h>
#include <stdbool.h>
#include <stdlib.h>

```

```
49. /**
50.  * Our own data type for string variables.
51.  */
52. typedef char* string;
53.
54. /**
55.  * Reads a line of text from standard input and returns the equivalent
56.  * char; if text does not represent a char, user is prompted to retry.
57.  * Leading and trailing whitespace is ignored.  If line can't be read,
58.  * returns CHAR_MAX.
59.  */
60. char GetChar(void);
61.
62. /**
63.  * Reads a line of text from standard input and returns the equivalent
64.  * double as precisely as possible; if text does not represent a
65.  * double, user is prompted to retry.  Leading and trailing whitespace
66.  * is ignored.  For simplicity, overflow and underflow are not detected.
67.  * If line can't be read, returns DBL_MAX.
68.  */
69. double GetDouble(void);
70.
71. /**
72.  * Reads a line of text from standard input and returns the equivalent
73.  * float as precisely as possible; if text does not represent a float,
74.  * user is prompted to retry.  Leading and trailing whitespace is ignored.
75.  * For simplicity, overflow and underflow are not detected.  If line can't
76.  * be read, returns FLT_MAX.
77.  */
78. float GetFloat(void);
79.
80. /**
81.  * Reads a line of text from standard input and returns it as an
82.  * int in the range of [-2^31 + 1, 2^31 - 2], if possible; if text
83.  * does not represent such an int, user is prompted to retry.  Leading
84.  * and trailing whitespace is ignored.  For simplicity, overflow is not
85.  * detected.  If line can't be read, returns INT_MAX.
86.  */
87. int GetInt(void);
88.
89. /**
90.  * Reads a line of text from standard input and returns an equivalent
91.  * long long in the range [-2^63 + 1, 2^63 - 2], if possible; if text
92.  * does not represent such a long long, user is prompted to retry.
93.  * Leading and trailing whitespace is ignored.  For simplicity, overflow
94.  * is not detected.  If line can't be read, returns LLONG_MAX.
95.  */
96. long long GetLongLong(void);
```

```
 97.
 98. /**
 99.  * Reads a line of text from standard input and returns it as a
100.  * string (char *), sans trailing newline character.  (Ergo, if
101.  * user inputs only "\n", returns "" not NULL.)  Returns NULL
102.  * upon error or no input whatsoever (i.e., just EOF).  Leading
103.  * and trailing whitespace is not ignored.  Stores string on heap
104.  * (via malloc); memory must be freed by caller to avoid leak.
105.  */
106. string GetString(void);
107.
108. #endif
```

```c
/**
 * list-0.c
 *
 * David J. Malan
 * malan@harvard.edu
 *
 * Demonstrates a linked list for numbers.
 */

#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "list-0.h"

// linked list
node* first = NULL;

// prototypes
void delete(void);
void insert(void);
void search(void);
void traverse(void);

int main(void)
{
    int c;
    do
    {
        // print instructions
        printf("\nMENU\n\n"
            "1 - delete\n"
            "2 - insert\n"
            "3 - search \n"
            "4 - traverse\n"
            "0 - quit\n\n");

        // get command
        printf("Command: ");
        c = GetInt();

        // try to execute command
        switch (c)
        {
            case 1: delete(); break;
            case 2: insert(); break;
            case 3: search(); break;
```

```
            case 4: traverse(); break;
        }
    }
    while (c != 0);

    // free list before quitting
    node* ptr = first;
    while (ptr != NULL)
    {
        node* predptr = ptr;
        ptr = ptr->next;
        free(predptr);
    }
}

/**
 * Tries to delete a number.
 */
void delete(void)
{
    // prompt user for number
    printf("Number to delete: ");
    int n = GetInt();

    // get list's first node
    node* ptr = first;

    // try to delete number from list
    node* predptr = NULL;
    while (ptr != NULL)
    {
        // check for number
        if (ptr->n == n)
        {
            // delete from head
            if (ptr == first)
            {
                first = ptr->next;
                free(ptr);
            }

            // delete from middle or tail
            else
            {
                predptr->next = ptr->next;
                free(ptr);
            }

```

```
 97.             // all done
 98.             break;
 99.         }
100.         else
101.         {
102.             predptr = ptr;
103.             ptr = ptr->next;
104.         }
105.     }
106.
107.     // traverse list
108.     traverse();
109. }
110.
111. /**
112.  * Tries to insert a number into list.
113.  */
114. void insert(void)
115. {
116.     // try to instantiate node for number
117.     node* newptr = malloc(sizeof(node));
118.     if (newptr == NULL)
119.     {
120.         return;
121.     }
122.
123.     // initialize node
124.     printf("Number to insert: ");
125.     newptr->n = GetInt();
126.     newptr->next = NULL;
127.
128.     // check for empty list
129.     if (first == NULL)
130.     {
131.         first = newptr;
132.     }
133.
134.     // else check if number belongs at list's head
135.     else if (newptr->n < first->n)
136.     {
137.         newptr->next = first;
138.         first = newptr;
139.     }
140.
141.     // else try to insert number in middle or tail
142.     else
143.     {
144.         node* predptr = first;
```

```
        while (true)
        {
            // avoid duplicates
            if (predptr->n == newptr->n)
            {
                free(newptr);
                break;
            }

            // check for insertion at tail
            else if (predptr->next == NULL)
            {
                predptr->next = newptr;
                break;
            }

            // check for insertion in middle
            else if (predptr->next->n > newptr->n)
            {
                newptr->next = predptr->next;
                predptr->next = newptr;
                break;
            }

            // update pointer
            predptr = predptr->next;
        }
    }

    // traverse list
    traverse();
}

/**
 * Searches for a number in list.
 */
void search(void)
{
    // prompt user for number
    printf("Number to search for: ");
    int n = GetInt();

    // get list's first node
    node* ptr = first;

    // search for number
    while (ptr != NULL)
    {
```

```c
        if (ptr->n == n)
        {
            printf("\nFound %i!\n", n);
            sleep(1);
            break;
        }
        ptr = ptr->next;
    }
}

/**
 * Traverses list, printing its numbers.
 */
void traverse(void)
{
    // traverse list
    printf("\nLIST IS NOW: ");
    node* ptr = first;
    while (ptr != NULL)
    {
        printf("%i ", ptr->n);
        ptr = ptr->next;
    }

    // flush standard output since we haven't outputted any newlines yet
    fflush(stdout);

    // pause before continuing
    sleep(1);
    printf("\n\n");
}
```

```c
/**
 * list-0.h
 *
 * David J. Malan
 * malan@harvard.edu
 *
 * Defines a node for a linked list of integers.
 */

typedef struct node
{
    int n;
    struct node* next;
}
node;
```

```c
/**
 * list-1.c
 *
 * David J. Malan
 * malan@harvard.edu
 *
 * Demonstrates a linked list for students.
 */

#include <cs50.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include "list-1.h"

// linked list
node* first = NULL;

// prototypes
void delete(void);
void insert(void);
void search(void);
void traverse(void);

int main(void)
{
    int c;
    do
    {
        // print instructions
        printf("\nMENU\n\n"
            "1 - delete\n"
            "2 - insert\n"
            "3 - search\n"
            "4 - traverse\n"
            "0 - quit\n\n");

        // get command
        printf("Command: ");
        c = GetInt();

        // try to execute command
        switch (c)
        {
            case 1: delete(); break;
            case 2: insert(); break;
            case 3: search(); break;
```

```
            case 4: traverse(); break;
        }
    }
    while (c != 0);

    // free list before quitting
    node* ptr = first;
    while (ptr != NULL)
    {
        node* predptr = ptr;
        ptr = ptr->next;
        if (predptr->student != NULL)
        {
            if (predptr->student->name != NULL)
            {
                free(predptr->student->name);
            }
            if (predptr->student->house != NULL)
            {
                free(predptr->student->house);
            }
            free(predptr->student);
        }
        free(predptr);
    }
}

/**
 * Tries to delete a student.
 */
void delete(void)
{
    // prompt user for ID
    printf("ID to delete: ");
    int n = GetInt();

    // get list's first node
    node* ptr = first;

    // try to delete student from list
    node* predptr = NULL;
    while (ptr != NULL)
    {
        // check for ID
        if (ptr->student->id == n)
        {
            // delete from head
            if (ptr == first)
```

```
 97.             {
 98.                 first = ptr->next;
 99.                 free(ptr->student->name);
100.                 free(ptr->student->house);
101.                 free(ptr->student);
102.                 free(ptr);
103.             }
104.
105.             // delete from middle or tail
106.             else
107.             {
108.                 predptr->next = ptr->next;
109.                 if (ptr->student->name != NULL)
110.                 {
111.                     free(ptr->student->name);
112.                 }
113.                 if (ptr->student->house != NULL)
114.                 {
115.                     free(ptr->student->house);
116.                 }
117.                 free(ptr->student);
118.                 free(ptr);
119.             }
120.
121.             // all done
122.             break;
123.         }
124.         else
125.         {
126.             predptr = ptr;
127.             ptr = ptr->next;
128.         }
129.     }
130.
131.     // traverse list
132.     traverse();
133. }
134.
135. /**
136.  * Tries to insert a student into list.
137.  */
138. void insert(void)
139. {
140.     // try to instantiate node for student
141.     node* newptr = malloc(sizeof(node));
142.     if (newptr == NULL)
143.     {
144.         return;
```

```
145.     }
146.
147.     // initialize node
148.     newptr->next = NULL;
149.
150.     // try to instantiate student
151.     newptr->student = malloc(sizeof(student));
152.     if (newptr->student == NULL)
153.     {
154.         free(newptr);
155.         return;
156.     }
157.
158.     // try to initialize student
159.     printf("Student's ID: ");
160.     newptr->student->id = GetInt();
161.     printf("Student's name: ");
162.     newptr->student->name = GetString();
163.     printf("Student's house: ");
164.     newptr->student->house = GetString();
165.     if (newptr->student->name == NULL || newptr->student->house == NULL)
166.     {
167.         if (newptr->student->name != NULL)
168.         {
169.             free(newptr->student->name);
170.         }
171.         if (newptr->student->house != NULL)
172.         {
173.             free(newptr->student->house);
174.         }
175.         free(newptr->student);
176.         free(newptr);
177.         return;
178.     }
179.
180.     // check for empty list
181.     if (first == NULL)
182.     {
183.         first = newptr;
184.     }
185.
186.     // else check if student belongs at list's head
187.     else if (newptr->student->id < first->student->id)
188.     {
189.         newptr->next = first;
190.         first = newptr;
191.     }
192.
```

```c
    // else try to insert student in middle or tail
    else
    {
        node* predptr = first;
        while (true)
        {
            // avoid duplicates
            if (predptr->student->id == newptr->student->id)
            {
                free(newptr->student->name);
                free(newptr->student->house);
                free(newptr->student);
                free(newptr);
                break;
            }

            // check for insertion at tail
            else if (predptr->next == NULL)
            {
                predptr->next = newptr;
                break;
            }

            // check for insertion in middle
            else if (predptr->next->student->id > newptr->student->id)
            {
                newptr->next = predptr->next;
                predptr->next = newptr;
                break;
            }

            // update pointer
            predptr = predptr->next;
        }
    }

    // traverse list
    traverse();
}


/**
 * Searches for student in list via student's ID.
 */
void search(void)
{
    // prompt user for ID
    printf("ID to search for: ");
```

```c
    int id = GetInt();

    // get list's first node
    node* ptr = first;

    // search for student
    while (ptr != NULL)
    {
        if (ptr->student->id == id)
        {
            printf("\nFound %s of %s (%i)!\n",
                ptr->student->name, ptr->student->house, id);
            sleep(1);
            break;
        }
        ptr = ptr->next;
    }
}

/**
 * Traverses list, printing its numbers.
 */
void traverse(void)
{
    // traverse list
    printf("\nLIST IS NOW: ");
    node* ptr = first;
    while (ptr != NULL)
    {
        printf("%s of %s (%i)  ",
            ptr->student->name, ptr->student->house, ptr->student->id);
        ptr = ptr->next;
    }

    // flush standard output since we haven't outputted any newlines yet
    fflush(stdout);

    // pause before continuing
    sleep(1);
    printf("\n\n");
}
```

```c
/**
 * list-1.h
 *
 * David J. Malan
 * malan@harvard.edu
 *
 * Defines structures for students and linked lists thereof.
 */

typedef struct
{
    int id;
    char* name;
    char* house;
}
student;

typedef struct node
{
    student* student;
    struct node* next;
}
node;
```

```c
/**
 * memory.c
 *
 * David J. Malan
 * malan@harvard.edu
 *
 * Demonstrates memory-related errors.
 *
 * problem 1: heap block overrun
 * problem 2: memory leak -- x not freed
 *
 * Adapted from
 * http://valgrind.org/docs/manual/quick-start.html#quick-start.prepare.
 */

#include <stdlib.h>

void f(void)
{
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;
}

int main(void)
{
    f();
    return 0;
}
```

```c
/**
 * scanf-0.c
 *
 * David J. Malan
 * malan@harvard.edu
 *
 * Reads a number from the user into an int.
 *
 * Demonstrates scanf and address-of operator.
 */

#include <stdio.h>

int main(void)
{
    int x;
    printf("Number please: ");
    scanf("%i", &x);
    printf("Thanks for the %i!\n", x);
}
```

```c
/**
 * scanf-1.c
 *
 * David J. Malan
 * malan@harvard.edu
 *
 * Reads a string from the user into memory it shouldn't.
 *
 * Demonstrates possible attack!
 */

#include <stdio.h>

int main(void)
{
    char* buffer;
    printf("String please: ");
    scanf("%s", buffer);
    printf("Thanks for the %s!\n", buffer);
}
```

```c
/**
 * scanf-2.c
 *
 * David J. Malan
 * malan@harvard.edu
 *
 * Reads a string from the user into an array (dangerously).
 *
 * Demonstrates potential buffer overflow!
 */

#include <stdio.h>

int main(void)
{
    char buffer[16];
    printf("String please: ");
    scanf("%s", buffer);
    printf("Thanks for the %s!\n", buffer);
}
```