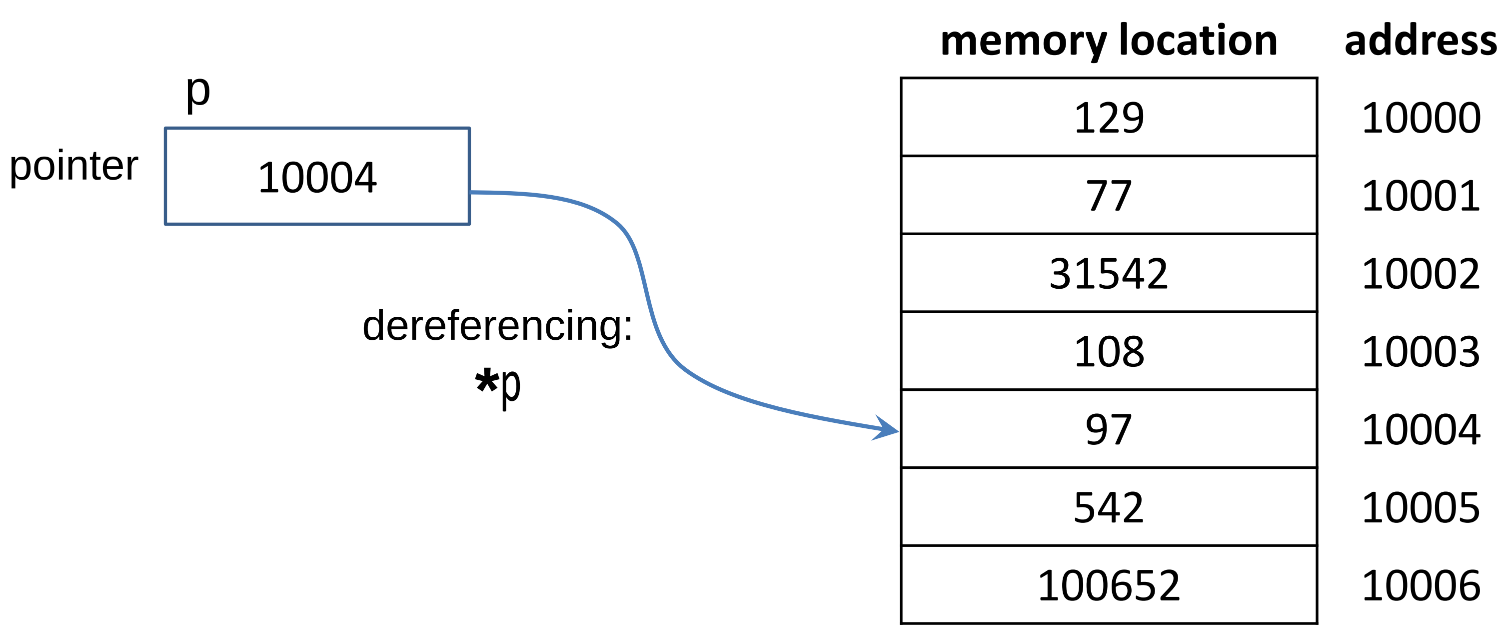
Pointers

- 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 *x* to something of type *T* is declared as

    T *x

    Example: int *p;   // p: pointer to an int

    char **w;   // w: pointer to a pointer to a char

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”)

More arrays

- Consider an array declared as:

  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

hed: /cs/www/classes/cs352/spring10/Code/ex.1.Pointers

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

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…

hed: /cs/www/classes/cs352/spring10/Code/ex.1.Pointers

```
 * 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));
  }
}
% 
```

figure out where the '\0' is

use this to control how many array elements to processes

Example 1…

hed: /cs/www/classes/cs352/spring10/Code/ex.1.Pointers

```
% 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

hed: /cs/www/classes/cs352/spring10/Code/ex.1.Pointers

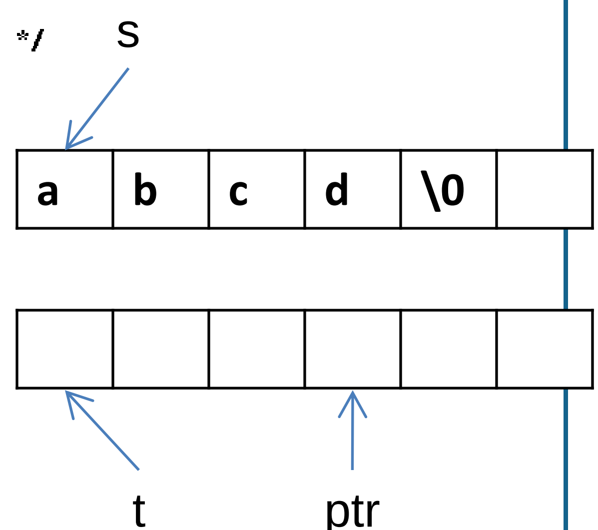
```
% cat str_reverse-1.c
/* File: str_reverse-1.c
 * This program implements a function to reverse a string. */

#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;
}
%
% 
```

array ≈ pointer

Example 2…

hed: /cs/www/classes/cs352/spring10/Code/ex.1.Pointers

```
% 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…

hed: /cs/www/classes/cs352/spring10/Code/ex.1.Pointers

```
% gcc str_reverse-1.c
% ./a.out
abcde
the reverse of  abcde  is  edcba
1234567
the reverse of  1234567  is  7654321
%
```

When 1 = 4

hed: /cs/www/classes/cs352/spring10/Code/ex.1.Pointers

```
% 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 llvar;

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

  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 = llptr; llptr += 1; val2 = llptr;
  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…

hed: /cs/www/classes/cs352/spring10/Code/ex.1.Pointers

```
% 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
./ptr-arith.c:18: warning: assignment makes integer from pointer without a
./ptr-arith.c:22: warning: assignment makes integer from pointer without a
./ptr-arith.c:22: warning: assignment makes integer from pointer without a
./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()

lectura.cs.arizona.edu - PuTTY

```
eanson@lectura:~/cs352/fall16/slides/programs$ cat sizeof.c
/*
 * File: 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;
}

eanson@lectura:~/cs352/fall16/slides/programs$
```

sizeof() invoked with a type name

sizeof() invoked with a variable name

sizeof() invoked with a pointer dereference

Figuring out sizes: sizeof()

lectura.cs.arizona.edu - PuTTY

```
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
-----       ----   --------   --------   -------
 char:        1         1          1         8
 int:         4         4          4         8
 long long:   8         8          8         8
```

More sizeof()

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

hed: /cs/www/classes/cs352/spring10/Code/ex.1.Pointers

```
% 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
%
```

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:

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

*p++

parsed as: $\boxed{*\ (\boxed{\text{p++}})}$

- similarly: *p--, *++p, etc.

evaluates to: contents of location a = *p
(side effect: p incremented by '++')

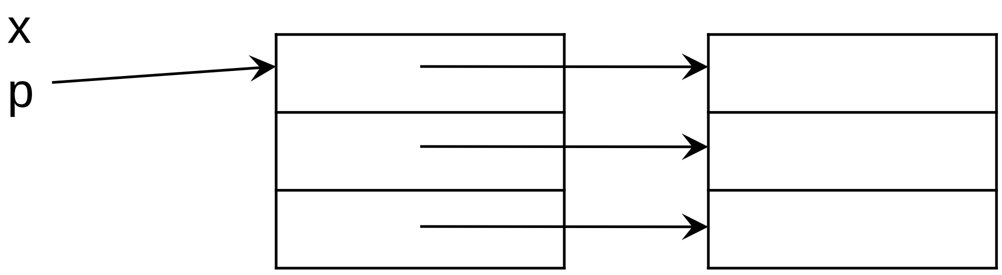
Walking a pointer down an array

hed: /cs/www/classes/cs352/spring10/Code/ex.1.Pointers

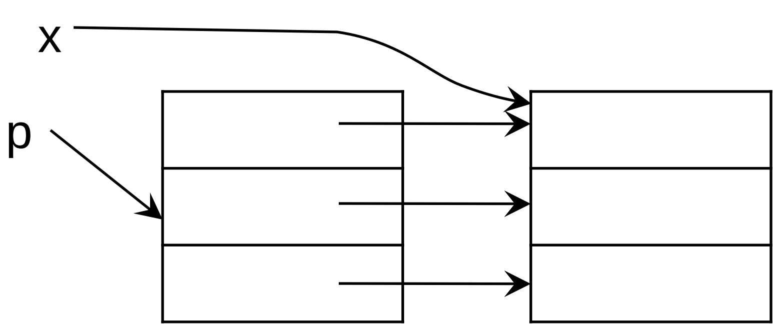
```
% cat array-walk.c
/*
 * File: array-walk.c
 * Purpose: Illustrate walking down an array with a pointer
 */
#include <stdio.h>

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;
}
% gcc -Wall array-walk.c
% ./a.out
1
2
3
4
0
sum = 10
%
```

dereference the pointer to access memory, then increment the pointer

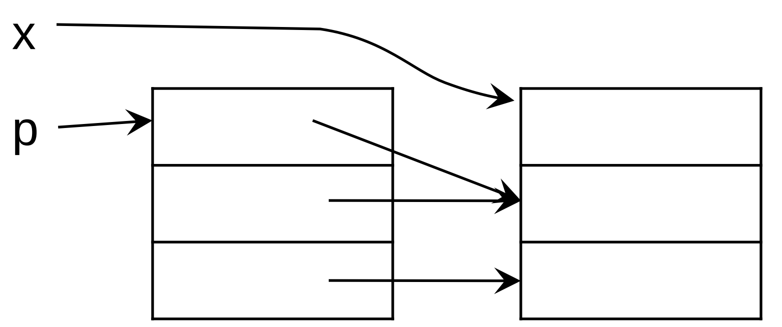
*p++ vs. (*p)++



after x = *p++



after x = (*p)++

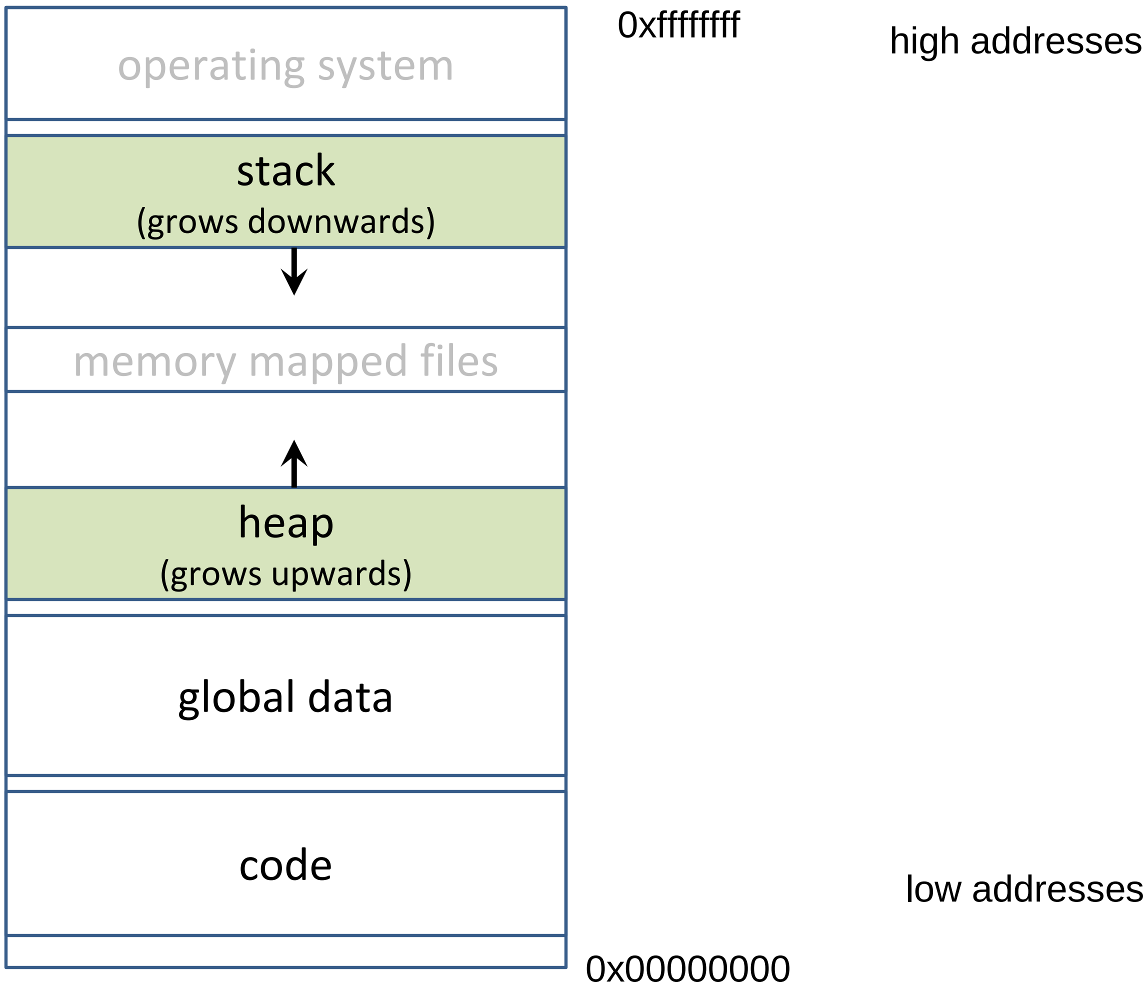


Two common pointer problems

- 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:

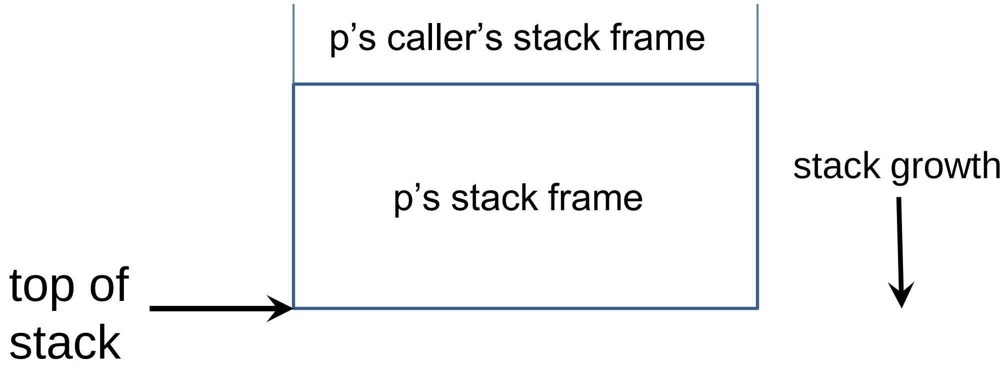


Background: Runtime Memory Organization

Code:

```
p(…) {
  …
  q(…);
  s(…);
}

q(…) {
  …
  r(…);
}

s(…) {
  …
}

r(…)
{
  …
}
```

Runtime stack:

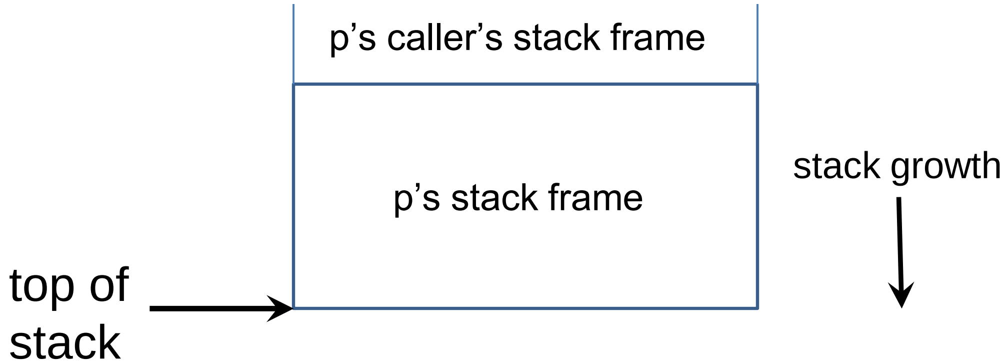


Background: Runtime Memory Organization

Code:

```
p(…) {
  …
  q(…);
  s(…);
}

q(…) {
  …
  r(…);
}

r(…)
{
  …
}

s(…) {
  …
}
```

Runtime stack:

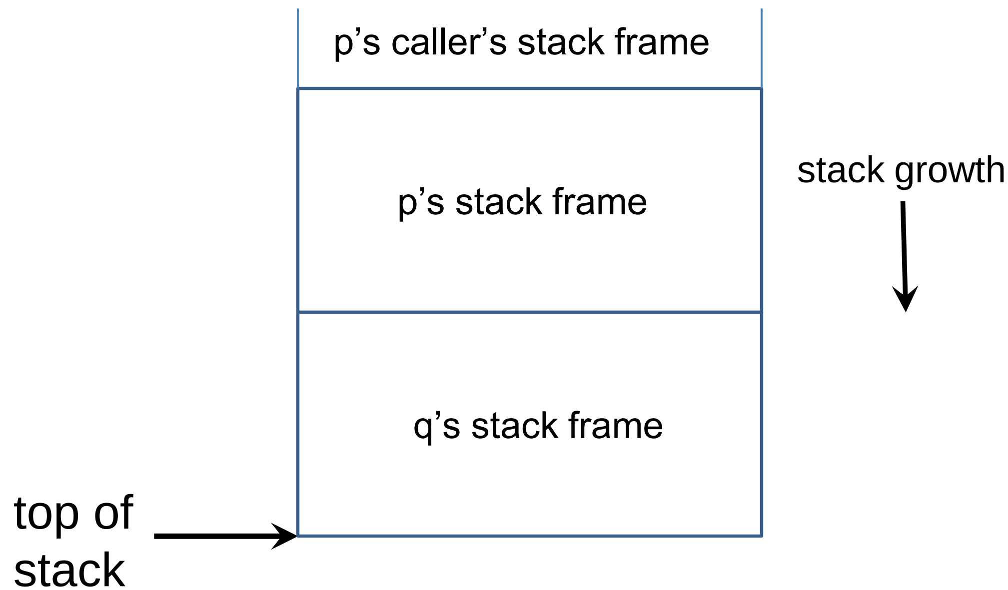


Background: Runtime Memory Organization

Code:

```
p(…) {
  …
  q(…);
  s(…);
}

q(…) {
  …
  r(…);
}

s(…) {
  …
}

r(…)
{
  …
}
```

Runtime stack:

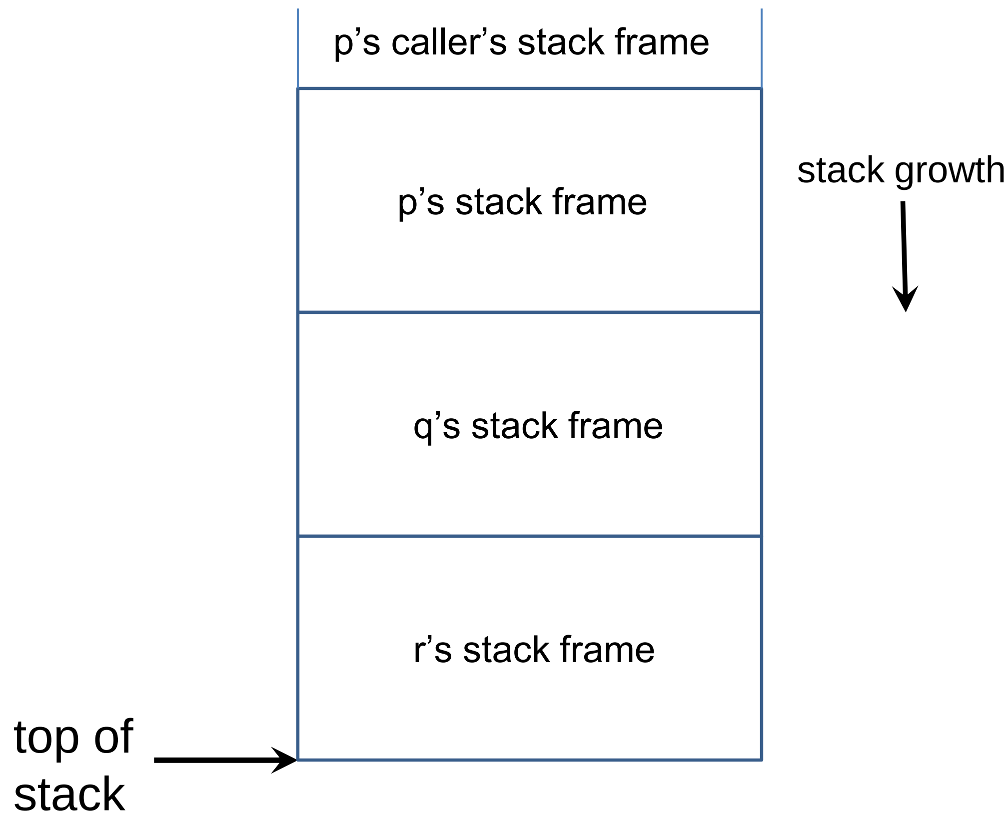


Background: Runtime Memory Organization

Code:

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

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

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

Runtime stack:

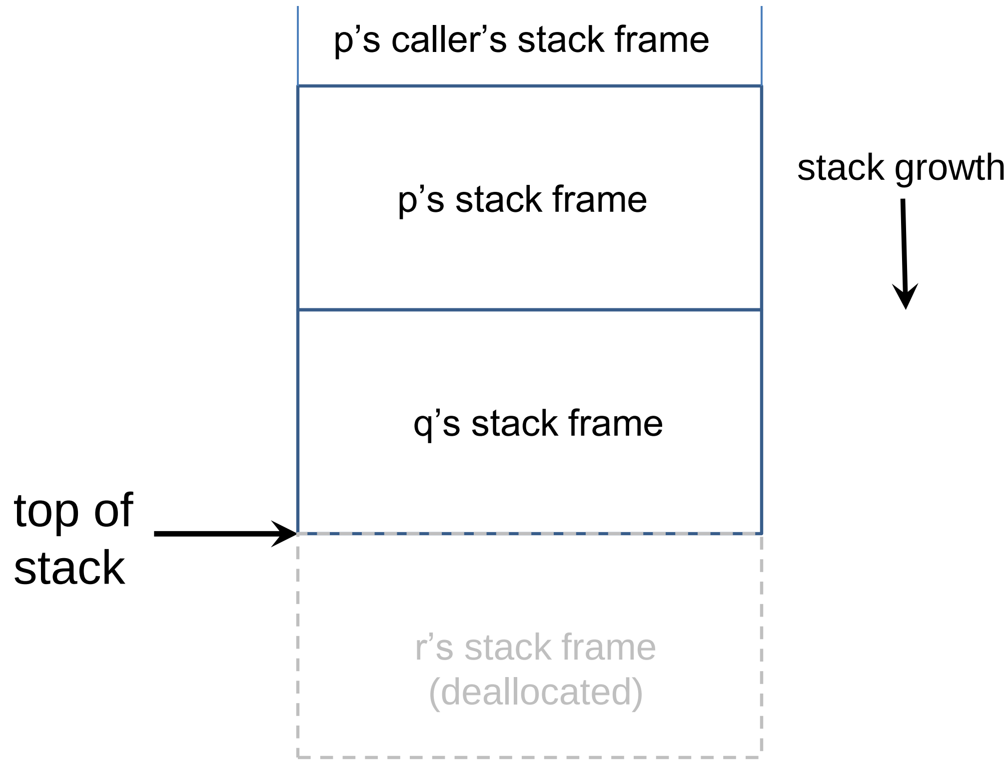


Background: Runtime Memory Organization

Code:

```
p(…) {
  …
  q(…);
  s(…);
}

q(…) {            s(…) {
  …                 …
  r(…);           }
}

r(…)
{
  …
}
```

Runtime stack:

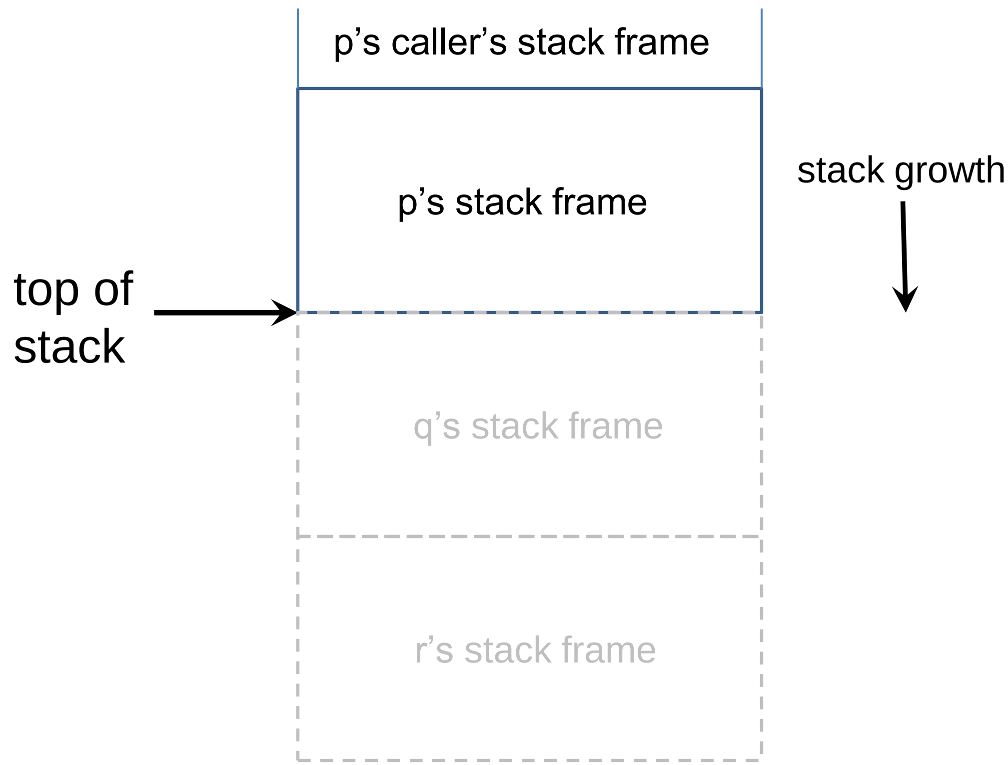


Background: Runtime Memory Organization

Code:

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

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

r(...)
{
  ...
}

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

Runtime stack:

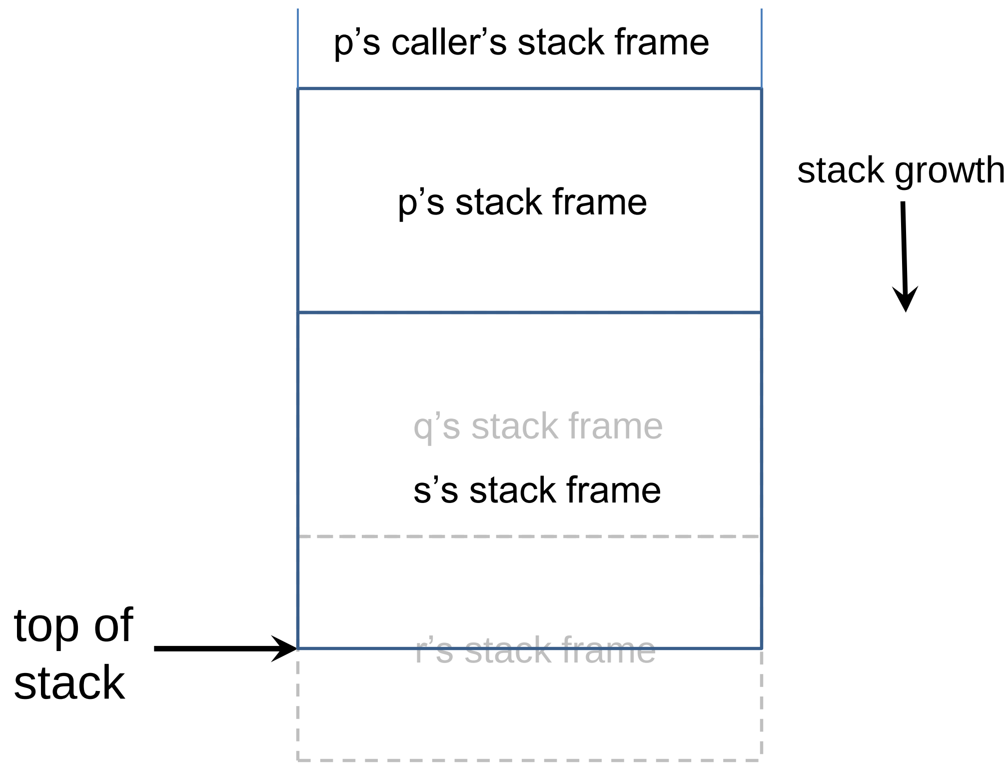


Background: Runtime Memory Organization

Code:

```
p(…) {
  …
  q(…);
  s(…);
}

q(…) {          s(…) {
  …               …
  r(…);         }
}

r(…)
{
  …
}
```

Runtime stack:



Uninitialized pointers: Example

hed: /cs/www/classes/cs352/spring10/Code/ex.2.Pointers

```
% cat uninit-ptr-1.c
/*
 * File: uninit-ptr-1.c
 * Purpose: illustrate uninitialized pointers
 */

#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
String read: (null)
Segmentation fault
%
% 
```

str was never initialized to point to anything

hed: /cs/www/classes/cs352/spring10/Cod

```
% cat uninit-ptr-1-fixed.c
/*
 * File: uninit-ptr-1-fixed.c
 * Purpose: fixes the problem with uninitialized pointers in
 *       file uninit-ptr-1.c
 */

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

fix: initialize str

Dangling pointers

What’s wrong with this code?

hed: /cs/www/classes/cs352/spring10/Code/ex.2.Pointers

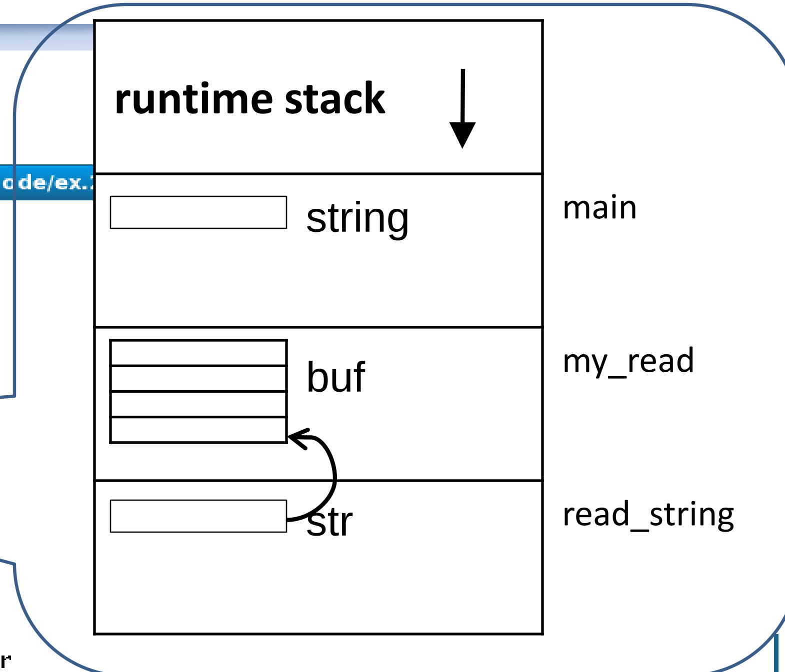
```
% cat dangling-ptr-1.c
// File: dangling-ptr-1.c
// Purpose: To illustrate dangling pointers

#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?

hed: /cs/www/classes/cs352/spring10/Code/ex.

```
% 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
%
```

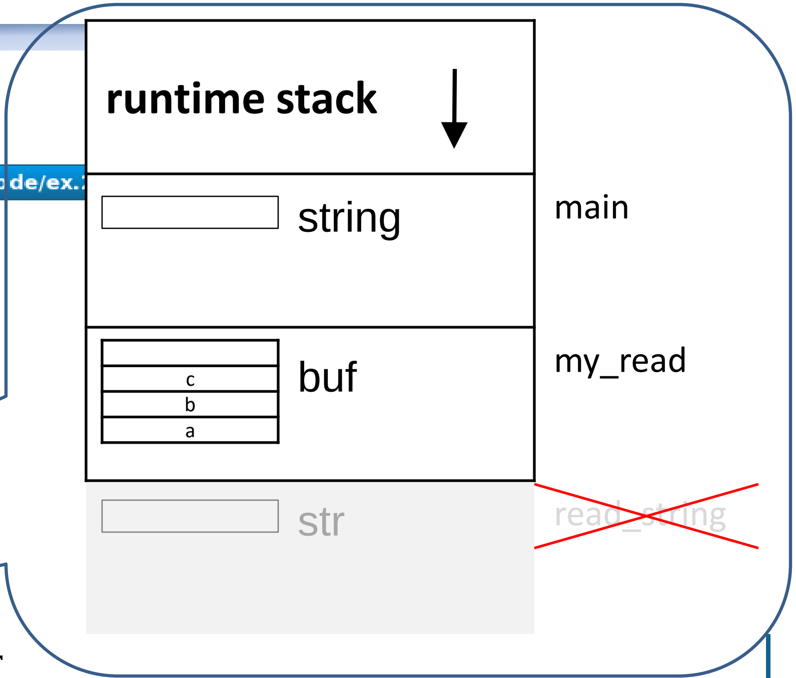


Dangling pointers

What's wrong with this code?

hed: /cs/www/classes/cs352/spring10/Code/ex.

```
% 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
%
```

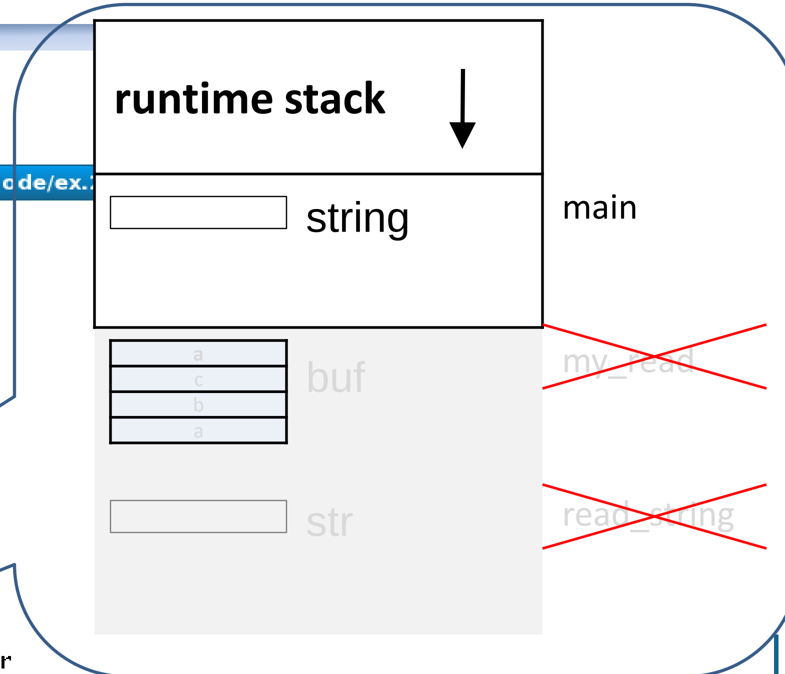


Dangling pointers

What’s wrong with this code?

hed: /cs/www/classes/cs352/spring10/Code/ex.

```
% 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
%
```

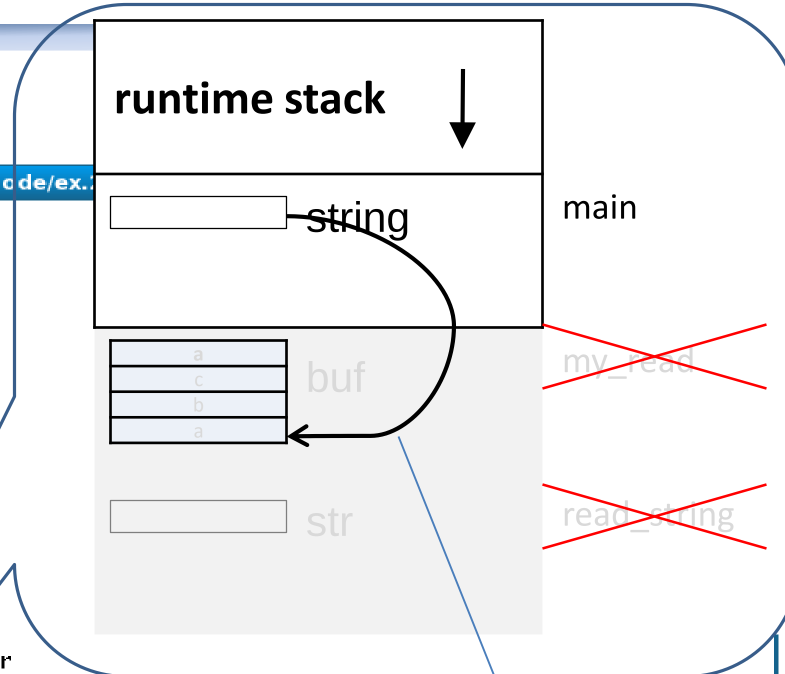


Dangling pointers

What's wrong with this code?

hed: /cs/www/classes/cs352/spring10/Code/ex.

```
% 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
% 
```

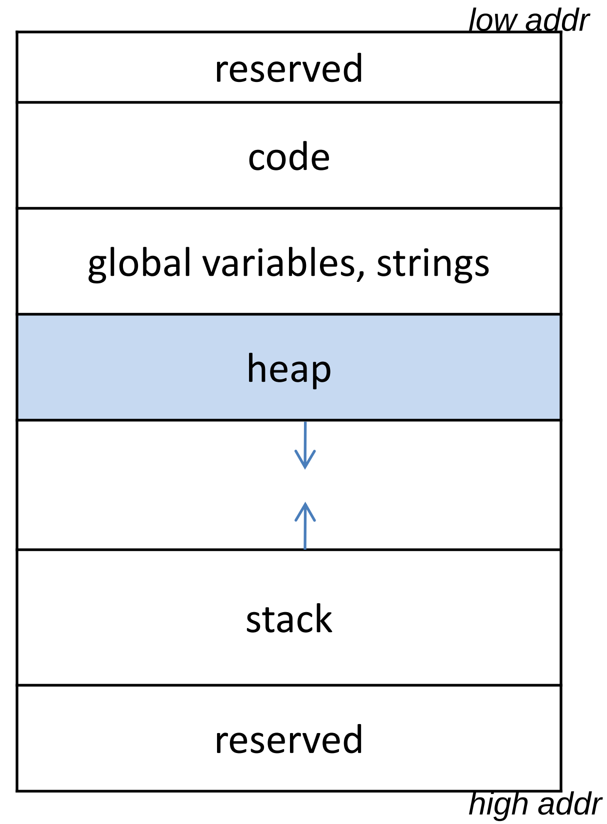


dangling pointer!

Dynamic memory allocation

- We can't always anticipate how much memory to allocate
  - too little $\Rightarrow$ program doesn't work
  - too much $\Rightarrow$ wastes space
- Solution: allocate memory at runtime as necessary
  - malloc(), calloc()
    - allocates memory in the heap area
  - free()
    - deallocates previously allocated heap memory block

program memory layout



Dynamic memory allocation: usage

```
MALLOC(3)                  Linux Programmer's Manual                 MALLOC(3)

NAME
       calloc, malloc, free, realloc - Allocate and free dynamic memory

SYNOPSIS
       #include <stdlib.h>

       void *calloc(size_t nmemb, size_t size);
       void *malloc(size_t size);
       void free(void *ptr);
       void *realloc(void *ptr, size_t size);

DESCRIPTION
       calloc()  allocate
       returns a pointer
       size  is  0,  then
       later be successfu

       malloc() allocates
       memory  is  not  c
       unique pointer val

       free() frees the m
       previous  call  to
       already been calle
       tion is performed.

       realloc()  changes
       The contents will
       cated memory will
       loc(size); if size
       ptr is NULL, it mu
       realloc().  If the

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.

:
```

void * : “generic pointer”

Usage:

```
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

hed: /cs/www/classes/cs352/spring10/Code/ex.2.Pointers

```
/* 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;
}
--More--(79%)
```

ALWAYS check the return value of any system call that may fail

Dynamic memory allocation: example 1

hed: /cs/www/classes/cs352/spring10/Code/ex.2.Pointers

```
  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]));
  }
}
%
%
%
%
%
% gcc -Wall ./dotprod.c
% ./a.out
4
1  2  3  4
5  6  7  8
dot product = 70
%
```

Dynamic memory allocation: example 2

hed: /cs/www/classes/cs352/spring10/Code/ex.2.Pointers

```
// 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;
}

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

hed: /cs/www/classes/cs352/spring10/Code/ex.2.Pointers

```
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
5
123
abc
456
def
789
>> Concatenated string: 123abc456def789
%
```

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

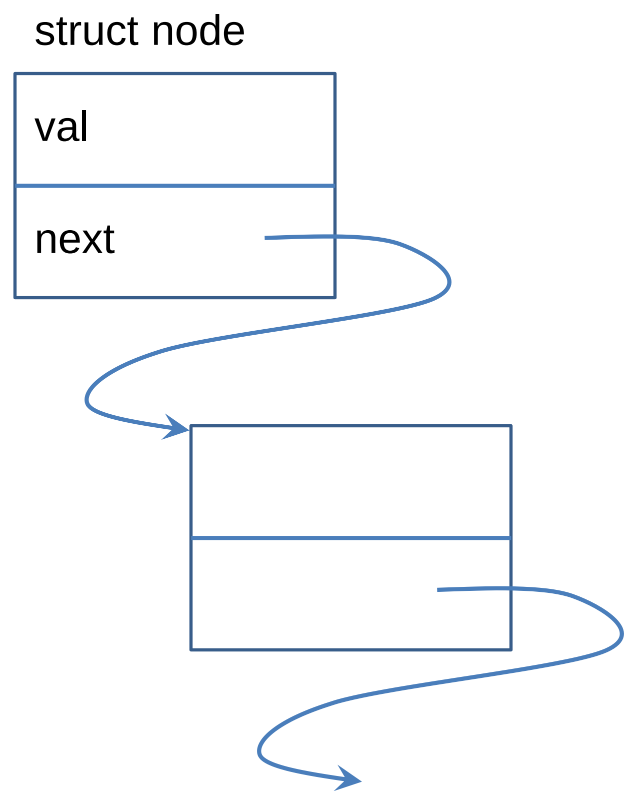
- Unlike Java classes, a **struct** can only contain data, not code.

Declaring structs

- A node for a linked list of integers:

```
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:

declares x, y to be variables of type "struct foo"

```
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**

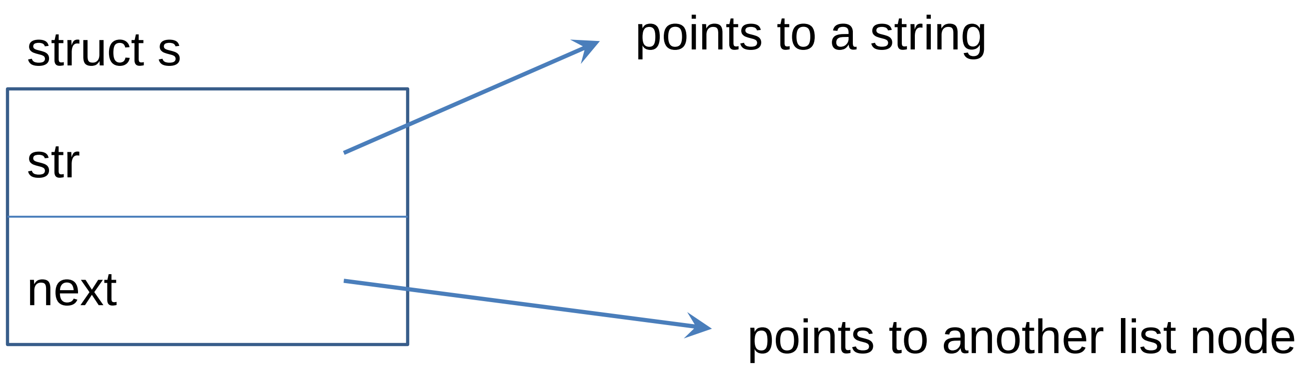
Example:

```
struct foo {
    int count, bar[10];
} *p, *q;

p->count = q->bar[3];
```

Example: sorting a linked list of strings

```
/*
 * 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**"



Example: sorting a linked list of strings

```
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;
}
```

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

```
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

hed: /cs/www/classes/cs352/spring10/Code/ex.4.Pointers

```
/*
 * 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 straightforard 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-> str1 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

hed: /cs/www/classes/cs352/spring10/Code/ex.4.Pointers

```
int main() {
  while (read_string() != NULL) {
    // loop, repeatedly calling read_string(), until it encounters EOF and returns NULL
  }

  sort_list();

  print_list();

  return 0;
}
hed: 233 %
hed: 233 %
hed: 233 % gcc -Wall sort-strings.c
hed: 234 % ./a.out
pqr
uvwxyz
zzzzz
abc
abbott
aardvark
AMPERSAND
lmnop
----- list contents -----
AMPERSAND
aardvark
abbott
abc
lmnop
pqr
uvwxyz
zzzzz
hed: 235 %
hed: 235 %
hed: 235 %
hed: 235 %
hed: 235 %
hed: 235 %
hed: 235 %
```

input strings

sorted output

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 ([ ] ( ) -> . $++_{\text{postfix}}$ $--_{\text{postfix}}$)
  - unary expressions ($++_{\text{prefix}}$ $--_{\text{prefix}}$ & * + - ~ ! 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) ++)