Dynamic memory

CS 211

Winter 2020

Initial code setup

The code in this course is available online. To download a copy of this lecture into your Unix shell account:

```
% cd cs211
% curl $URL211/lec/06dynamic.tgz | tar zxvk
⋮
% cd 06dynamic
```

How can we work with strings?

```
bool is_comment(const string*);

// Concatenates array of strings; strips comments.
string strip_concat(const string* begin,
                    const string* end)
{
    string result = "";
    while (begin < end) {
        if (! is_comment(begin))
            result += *begin + "\n";
        ++begin;
    }
    return result;
}
```

How can we work with strings?

```
bool is_comment(const string*);

// Concatenates array of strings; strips comments.
string strip_concat(const string* begin,
                    const string* end)
{
    string result = "";
    while (begin < end) {
        if (! is_comment(begin))
            result += *begin + "\n";
        ++begin;
    }
    return result;
}
```

This is actually C++.

How can we work with strings?

```
bool is_comment(const string*);

// Concatenates array of strings; strips comments.
string strip_concat(const string* begin,
                    const string* end)
{
    string result = "";
    while (begin < end) {
        if (! is_comment(begin))
            result += *begin + "\n";
        ++begin;
    }
    return result;
}
```

This is actually (very inefficient) C++.

Where should strings live?

Solution
in each function’s automatic storage
in one function’s automatic storage
someplace else…

Where should strings live?

Solution	**Problem**
in each function's automatic storage	
in one function's automatic storage	
someplace else…	

Where should strings live?

Solution	**Problem**
in each function's automatic storage	inflexible & inefficient
in one function's automatic storage	
someplace else…	

Where should strings live?

Solution	**Problem**
in each function's automatic storage	inflexible & inefficient
in one function's automatic storage	functions return
someplace else…	

Where should strings live?

Solution	Problem
in each function’s automatic storage	inflexible & inefficient
in one function’s automatic storage	functions return
someplace else…	difficult

A uniform-capacity string

Can be passed, returned, assigned:

```
#define MAXSTRLEN 80

struct string80
{
    char data[MAXSTRLEN + 1];
};

typedef struct string80 string80_t;
```

The easy-but-inflexible solution: all strings have the same capacity

See `src/string80.h`

So we work with `'\0'`-terminated `char*`s

The C string:

```
void copy_string_into(char* dst, const char* src)
{
    while ( (*dst++ = *src++) )
    { }
}
```

This works provided `src` is actually terminated and `dst` has sufficient capacity

See `str/ptr_string.c`

So we work with '\0'-terminated char*s

The C string:

```
void copy_string_into(char* dst, const char* src)
{
    while ( (*dst++ = *src++) )
    { }
}
```

This works provided src is actually terminated and dst has sufficient capacity

See str/ptr_string.c

But how can we ensure that dst has sufficient capacity?

Okay, but where should we store `dst`?

```c
#include "ptr_string.h"
#include <stdio.h>

int main()
{
    // Actually stored in the “static area”:
    const char message[] = "On␣the␣stack!";
    // Stored in main’s stack frame:
    char buf[sizeof message];

    copy_string_into(buf, message);
    printf("%s\n", buf);
    str_toupper_inplace(buf);
    printf("%s\n", buf);
}
```

This function is wrong, and cannot work

```c
#include "ptr_string.h"

char* bad_str_toupper_copy(const char* s)
{
    char result[count_chars(s) + 1];
    str_toupper_into(result, s);
    return result;
}
```

Why?

This function is wrong, and cannot work

```
#include "ptr_string.h"

char* bad_str_toupper_copy(const char* s)
{
    char result[count_chars(s) + 1];
    str_toupper_into(result, s);
    return result;
}
```

Why? The result points to an object that is destroyed when `bad_str_toupper_copy` returns.

Dynamic memory allocation: The basics

- Function `void* malloc(size_t size)` requests `size` bytes of memory from the system.

Dynamic memory allocation: The basics

- Function `void* malloc(size_t size)` requests `size` bytes of memory from the system.
- `malloc()` either returns a pointer to a new object of the requested size, or indicates failure by returning special "pointer-to-nowhere" `NULL`.

(Type `void*` literally means "pointer to nothing," but better to think of it as a pointer to *uninitialized memory of unknown size*.)

Dynamic memory allocation: The basics

- Function `void* malloc(size_t size)` requests `size` bytes of memory from the system.
- `malloc()` either returns a pointer to a new object of the requested size, or indicates failure by returning special "pointer-to-nowhere" `NULL`.
- Function `void free(void* ptr)` releases memory back to the system.

(Type `void*` literally means "pointer to nothing," but better to think of it as a pointer to *uninitialized memory of unknown size*.)

Dynamic memory allocation: The rules

1. Every pointer returned by `malloc()` must be `NULL`-checked (because dereferencing `NULL` is UB)

Dynamic memory allocation: The rules

1. Every pointer returned by `malloc()` must be `NULL`-checked (because dereferencing `NULL` is UB)
2. Every *object* returned by `malloc()` must have its address passed to `free()` *exactly* once (because otherwise you leak memory)

Dynamic memory allocation: The rules

1. Every pointer returned by `malloc()` must be `NULL`-checked (because dereferencing `NULL` is UB)
2. Every *object* returned by `malloc()` must have its address passed to `free()` *exactly* once (because otherwise you leak memory)
3. After an object is freed, it must not be accessed (read or written) or freed again (or else UB)

Dynamic memory allocation: The rules

1. Every pointer returned by `malloc()` must be `NULL`-checked (because dereferencing `NULL` is UB)
2. Every *object* returned by `malloc()` must have its address passed to `free()` *exactly* once (because otherwise you leak memory)
3. After an object is freed, it must not be accessed (read or written) or freed again (or else UB)
4. A object that was not obtained from `malloc()` must not be freed (or else nasal demons)

Dynamic memory allocation: The rules

1. Every pointer returned by `malloc()` must be `NULL`-checked (because dereferencing `NULL` is UB)
2. Every *object* returned by `malloc()` must have its address passed to `free()` *exactly* once (because otherwise you leak memory)
3. After an object is freed, it must not be accessed (read or written) or freed again (or else UB)
4. A object that was not obtained from `malloc()` must not be freed (or else nasal demons)
5. Except: `free(NULL)` is just fine

Heap allocation example

```c
#include "ptr_string.h"
#include <stdlib.h>

char* string_clone(const char* s)
{
    char* result = malloc(count_chars(s) + 1);
    if (result) copy_string_into(result, s);
    return result;
}

char* str_toupper_clone(const char* s)
{
    char* result = malloc(count_chars(s) + 1);
    if (result) str_toupper_into(result, s);
    return result;
}
```

Concatenating two strings, result in the heap

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

char* string_concat(const char* s, const char* t)
{
    size_t s_len = strlen(s);  // count_chars
    size_t t_len = strlen(t);

    char* result = malloc(s_len + t_len + 1);
    if (result == NULL) return NULL;

    strcpy(result, s);         // copy_string_into
    strcpy(result + s_len, t);

    return result;
}
```

Our initial example

```c
char* strip_concat(char** lines, size_t count)
{
    size_t total_len = 0;

    for (size_t i = 0; i < count; ++i)
        if (! is_comment(lines[i]))
            total_len += strlen(lines[i]) + 1;

    char* result = malloc(total_len + 1);
    if (result == NULL) return NULL;

    char* fill = result;

    for (size_t i = 0; i < count; ++i) {
        if (! is_comment(lines[i])) {
            fill = stpcpy(fill, lines[i]);
            *fill++ = '\n';
        }
    }

    *fill = '\0';

    return result;
}
```

See `src/string_fun.c` and `test/test_string_fun.c`.

– Next: Linked data structures –

Extras

- Arrays vs. Pointers
- Arrays vs. Strings
- The Nulls

Arrays vs. Pointers

Arrays *decay* to pointers

```
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);             // ⇒ 0x7ffee5c6e2f0
put_ptr(a);
put_int(a[0]);
put_int(*a);
```

Arrays *decay* to pointers

```
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);             // ⇒ 0x7ffee5c6e2f0
put_ptr(a);                 // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);
put_int(*a);
```

Arrays *decay* to pointers

```
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);              // ⇒ 0x7ffee5c6e2f0
put_ptr(a);                  // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);               // ⇒ 2
put_int(*a);
```

Arrays *decay* to pointers

```
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);             // ⇒ 0x7ffee5c6e2f0
put_ptr(a);                 // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);              // ⇒ 2
put_int(*a);                // ⇒ 2
```

Arrays *decay* to pointers

```
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);            // ⇒ 0x7ffee5c6e2f0
put_ptr(a);                // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);             // ⇒ 2
put_int(*a);               // ⇒ 2

put_ptr(&a[1]);
put_ptr(a + 1);
put_int(a[1]);
put_int(*(a + 1));
```

Arrays *decay* to pointers

```
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);              // ⇒ 0x7ffee5c6e2f0
put_ptr(a);                  // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);               // ⇒ 2
put_int(*a);                 // ⇒ 2

put_ptr(&a[1]);              // ⇒ 0x7ffee5c6e2f4
put_ptr(a + 1);
put_int(a[1]);
put_int(*(a + 1));
```

Arrays *decay* to pointers

```
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);              // ⇒ 0x7ffee5c6e2f0
put_ptr(a);                  // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);               // ⇒ 2
put_int(*a);                 // ⇒ 2

put_ptr(&a[1]);              // ⇒ 0x7ffee5c6e2f4
put_ptr(a + 1);
put_int(a[1]);
put_int(*(a + 1));
```

Arrays *decay* to pointers

```
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);             // ⇒ 0x7ffee5c6e2f0
put_ptr(a);                 // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);              // ⇒ 2
put_int(*a);                // ⇒ 2

put_ptr(&a[1]);             // ⇒ 0x7ffee5c6e2f4
put_ptr(a + 1);             // ⇒ 0x7ffee5c6e2f4
put_int(a[1]);
put_int(*(a + 1));
```

Arrays *decay* to pointers

```
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);           // ⇒ 0x7ffee5c6e2f0
put_ptr(a);               // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);            // ⇒ 2
put_int(*a);              // ⇒ 2

put_ptr(&a[1]);           // ⇒ 0x7ffee5c6e2f4
put_ptr(a + 1);           // ⇒ 0x7ffee5c6e2f4
put_int(a[1]);            // ⇒ 3
put_int(*(a + 1));
```

Arrays *decay* to pointers

```
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);             // ⇒ 0x7ffee5c6e2f0
put_ptr(a);                 // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);              // ⇒ 2
put_int(*a);                // ⇒ 2

put_ptr(&a[1]);             // ⇒ 0x7ffee5c6e2f4
put_ptr(a + 1);             // ⇒ 0x7ffee5c6e2f4
put_int(a[1]);              // ⇒ 3
put_int(*(a + 1));          // ⇒ 3
```

Arrays *decay* to pointers

```
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);             // ⇒ 0x7ffee5c6e2f0
put_ptr(a);                 // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);              // ⇒ 2
put_int(*a);                // ⇒ 2

put_ptr(&a[1]);             // ⇒ 0x7ffee5c6e2f4
put_ptr(a + 1);             // ⇒ 0x7ffee5c6e2f4
put_int(a[1]);              // ⇒ 3
put_int(*(a + 1));          // ⇒ 3

put_size(sizeof a);
put_size(sizeof (a + 0));
```

Arrays *decay* to pointers

```
int a[] = { 2, 3, 4, 5, 6 };

put_ptr(&a[0]);               // ⇒ 0x7ffee5c6e2f0
put_ptr(a);                   // ⇒ 0x7ffee5c6e2f0
put_int(a[0]);                // ⇒ 2
put_int(*a);                  // ⇒ 2

put_ptr(&a[1]);               // ⇒ 0x7ffee5c6e2f4
put_ptr(a + 1);               // ⇒ 0x7ffee5c6e2f4
put_int(a[1]);                // ⇒ 3
put_int(*(a + 1));            // ⇒ 3

put_size(sizeof a);           // ⇒ 20
put_size(sizeof (a + 0)); // ⇒ 8
```

Array indexing is pointer arithmetic

$\langle \mathit{aexpr} \rangle$ [$\langle \mathit{iexpr} \rangle$]   *means*   *($\langle \mathit{aexpr} \rangle$ + $\langle \mathit{iexpr} \rangle$)

Array indexing is pointer arithmetic

$$\langle aexpr \rangle \texttt{[} \langle iexpr \rangle \texttt{]} \quad \textit{means} \quad \texttt{*(} \langle aexpr \rangle \texttt{ + } \langle iexpr \rangle \texttt{)}$$

$$\texttt{\&} \langle aexpr \rangle \texttt{[} \langle iexpr \rangle \texttt{]} \quad \textit{means} \quad \langle aexpr \rangle \texttt{ + } \langle iexpr \rangle$$

Arrays vs. Strings

Strings are arrays of chars

```
#include <stdio.h>

int main()
{
    char mystery[] = {
      71, 111, 32, 39, 67, 97, 116, 115, 33, 0
    };

    printf("%s\n", mystery);
}
```

Strings are arrays of `char`s

```c
#include <stdio.h>

int main()
{
    char mystery[] = {
      71, 'o', 32, 39, 67, 97, 116, 115, 33, 0
    };

    printf("%s\n", mystery);
}
```

Strings are arrays of `char`s

```
#include <stdio.h>

int main()
{
    char mystery[] = {
      71, 'o', 32, 39, 67, 'a', 116, 115, 33, 0
    };

    printf("%s\n", mystery);
}
```

Strings are arrays of `char`s

```c
#include <stdio.h>

int main()
{
    char mystery[] = {
      71, 'o', 32, 39, 67, 'a', 't', 115, 33, 0
    };

    printf("%s\n", mystery);
}
```

Strings are arrays of chars

```c
#include <stdio.h>

int main()
{
    char mystery[] = {
      71, 'o', 32, 39, 67, 'a', 't', 's', 33, 0
    };

    printf("%s\n", mystery);
}
```

Strings are arrays of `char`s

```c
#include <stdio.h>

int main()
{
    char mystery[] = {
      71, 'o', 32, 39, 67, 'a', 't', 's', '!', 0
    };

    printf("%s\n", mystery);
}
```

Strings are arrays of chars

```c
#include <stdio.h>

int main()
{
    char mystery[] = {
      71, 'o', 32, 39, 67, 'a', 't', 's', '!', '\0'
    };

    printf("%s\n", mystery);
}
```

Strings are arrays of `char`s

```c
#include <stdio.h>

int main()
{
    char mystery[] = {
      71, 'o', 32, '\'', 67, 'a', 't', 's', '!', '\0'
    };

    printf("%s\n", mystery);
}
```

How long is a C string?

```
int main()
{
    const char* cptr = "12345";
}
```

How long is a C string?

```
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);           // ⇒ ?









}
```

How long is a C string?

```
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);              // ⇒ 8
}
```

How long is a C string?

```
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);            // ⇒ 8
    printf("%zu\n", sizeof *cptr);           // ⇒ ?
}
```

How long is a C string?

```
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);          // ⇒ 8
    printf("%zu\n", sizeof *cptr);         // ⇒ 1











}
```

How long is a C string?

```
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);             // ⇒ 8
    printf("%zu\n", sizeof *cptr);            // ⇒ 1
    printf("%zu\n", sizeof(const char*));     // ⇒ 8
    printf("%zu\n", sizeof(const char));      // ⇒ 1
}
```

How long is a C string?

```
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);                // ⇒ 8
    printf("%zu\n", sizeof *cptr);               // ⇒ 1
    printf("%zu\n", sizeof(const char*));        // ⇒ 8
    printf("%zu\n", sizeof(const char));         // ⇒ 1

    const char carray[] = "12345";
    printf("%zu\n", sizeof carray);              // ⇒ ?





}
```

How long is a C string?

```
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);              // ⇒ 8
    printf("%zu\n", sizeof *cptr);             // ⇒ 1
    printf("%zu\n", sizeof(const char*));      // ⇒ 8
    printf("%zu\n", sizeof(const char));       // ⇒ 1

    const char carray[] = "12345";
    printf("%zu\n", sizeof carray);            // ⇒ 6
}
```

How long is a C string?

```
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);               // ⇒ 8
    printf("%zu\n", sizeof *cptr);              // ⇒ 1
    printf("%zu\n", sizeof(const char*));       // ⇒ 8
    printf("%zu\n", sizeof(const char));        // ⇒ 1

    const char carray[] = "12345";
    printf("%zu\n", sizeof carray);             // ⇒ 6
    printf("%zu\n", sizeof(const char[6]));     // ⇒ 6




}
```

How long is a C string?

```
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);              // ⇒ 8
    printf("%zu\n", sizeof *cptr);             // ⇒ 1
    printf("%zu\n", sizeof(const char*));      // ⇒ 8
    printf("%zu\n", sizeof(const char));       // ⇒ 1

    const char carray[] = "12345";
    printf("%zu\n", sizeof carray);            // ⇒ 6
    printf("%zu\n", sizeof(const char[6]));    // ⇒ 6

    for (size_t i = 0; i < sizeof carray; ++i)
        printf("%d␣", (int) carray[i]);
    // ⇒ ?
}
```

How long is a C string?

```
int main()
{
    const char* cptr = "12345";
    printf("%zu\n", sizeof cptr);              // ⇒ 8
    printf("%zu\n", sizeof *cptr);             // ⇒ 1
    printf("%zu\n", sizeof(const char*));      // ⇒ 8
    printf("%zu\n", sizeof(const char));       // ⇒ 1

    const char carray[] = "12345";
    printf("%zu\n", sizeof carray);            // ⇒ 6
    printf("%zu\n", sizeof(const char[6])); // ⇒ 6

    for (size_t i = 0; i < sizeof carray; ++i)
        printf("%d␣", (int) carray[i]);
    // ⇒ 49 50 51 52 53 0
}
```

A string algorithm

```
size_t count_chars(const char* s)
{
    size_t result = 0;
    while (*s++) ++result;
    return result;
}
```

A string algorithm

```
size_t count_chars(const char* s)
{
    size_t result = 0;
    while (*s++) ++result;
    return result;
}
```

```
size_t count_chars(const char* s)
{
    size_t i = 0;
    while (s[i] != '\0') ++i;
    return i;
}
```

A string algorithm

```
size_t count_chars(const char* s)
{
    size_t result = 0;
    while (*s++) ++result;
    return result;
}
```

```
size_t count_chars(const char* s)
{
    const char* t = s;
    while (*t) ++t;
    return t - s;
}
```

Counting characters

```
int main()
{
    const char carray[] = "12345",
               *cptr    = "12345";

    printf("%zu\n", count_chars(carray));  // ⇒ ?
    printf("%zu\n", count_chars(cptr));    // ⇒ ?
}
```

Counting characters

```
int main()
{
    const char carray[] = "12345",
               *cptr    = "12345";

    printf("%zu\n", count_chars(carray));  // ⇒ 5
    printf("%zu\n", count_chars(cptr));    // ⇒ 5







}
```

Counting characters

```
int main()
{
    const char carray[] = "12345",
               *cptr    = "12345";

    printf("%zu\n", count_chars(carray));  // ⇒ 5
    printf("%zu\n", count_chars(cptr));    // ⇒ 5

    char buf[800] = {'a'};
    printf("%zu\n", sizeof buf);           // ⇒ ?
    printf("%zu\n", count_chars(buf));     // ⇒ ?




}
```

Counting characters

```
int main()
{
    const char carray[] = "12345",
               *cptr    = "12345";

    printf("%zu\n", count_chars(carray));  // ⇒ 5
    printf("%zu\n", count_chars(cptr));    // ⇒ 5

    char buf[800] = {'a'};
    printf("%zu\n", sizeof buf);           // ⇒ 800
    printf("%zu\n", count_chars(buf));     // ⇒ 1




}
```

Counting characters

```
int main()
{
    const char carray[] = "12345",
               *cptr    = "12345";

    printf("%zu\n", count_chars(carray));  // ⇒ 5
    printf("%zu\n", count_chars(cptr));    // ⇒ 5

    char buf[800] = {'a'};
    printf("%zu\n", sizeof buf);           // ⇒ 800
    printf("%zu\n", count_chars(buf));     // ⇒ 1

    buf[1] = buf[2] = buf[4] = buf[5] = 'b';
    printf("%zu\n", count_chars(buf));     // ⇒ ?
    printf("%s\n", buf);                   // ⇒ ?
}
```

Counting characters

```
int main()
{
    const char carray[] = "12345",
               *cptr    = "12345";

    printf("%zu\n", count_chars(carray));  // ⇒ 5
    printf("%zu\n", count_chars(cptr));    // ⇒ 5

    char buf[800] = {'a'};
    printf("%zu\n", sizeof buf);           // ⇒ 800
    printf("%zu\n", count_chars(buf));     // ⇒ 1

    buf[1] = buf[2] = buf[4] = buf[5] = 'b';
    printf("%zu\n", count_chars(buf));     // ⇒ 3
    printf("%s\n", buf);                   // ⇒ abb
}
```

The Nulls

`NULL` versus nul versus null

Thing	Type of Thing	Purpose of Thing

`NULL` versus nul versus null

Thing	**Type of Thing**	**Purpose of Thing**
“[a] null [pointer]”	$T*$ for any T	stands for a missing object

`NULL` versus nul versus null

Thing	**Type of Thing**	**Purpose of Thing**
"[a] null [pointer]"	$T*$ for any T	stands for a missing object
`NULL`	`void`$*$	null pointer constant

NULL versus nul versus null

Thing	Type of Thing	Purpose of Thing
“[a] null [pointer]”	$T*$ for any T	stands for a missing object
NULL	void*	null pointer constant
'\0' (a/k/a nul)	int	0 with character connotation

NULL versus nul versus null

Thing	Type of Thing	Purpose of Thing
"[a] null [pointer]"	$T*$ for any T	stands for a missing object
`NULL`	`void*`	null pointer constant
`'\0'` (a/k/a nul)	`int`	`0` with character connotation

So `NULL` is null, but nul is something completely different.