Dynamic memory

EECS 211

Winter 2019

Initial code setup

```
$ cd eecs211
$ curl $URL211/lec/06dynamic.tgz | tar zx
…
$ cd 06dynamic
```

Oops!

I made a mistake. In C, the declaration

```
struct circle read_circle();
```

means that `read_circle` takes any number of arguments.

Oops!

I made a mistake. In C, the declaration

```
struct circle read_circle();
```

means that `read_circle` takes any number of arguments.

In “traditional” C, arguments weren’t checked:

```
double min2();    // declaration

double min3()     // definition
    double x, y, z;
{
    return min2(x, min2(y, z));
}
```

Oops!

I made a mistake. In C, the declaration

```
struct circle read_circle();
```

means that `read_circle` takes any number of arguments.

In “traditional” C, arguments weren’t checked:

```
min2();     // declaration

min3()      // definition
    int x, y, z;
{
    return min2(x, min2(y, z));
}
```

Oops!

I made a mistake. In C, the declaration

```
struct circle read_circle();
```

means that `read_circle` takes any number of arguments.

In “traditional” C, arguments weren’t checked:

```
min2();    // declaration

min3()     // definition
    int x, y, z;
{
    return min2(x, min2(y, z));
}
```

The correct way to say “no arguments” in C is

```
struct circle read_circle(void);
```

And now, strings...

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?

```
#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 –

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.