Dynamic Memory

CS 211

Initial code setup

The code in this course is available in your Unix shell account. You can get your own copy like this:

```
% cd cs211
% tar -xvkf ~cs211/lec/06_dynamic.tgz
⋮
% cd 06_dynamic
```

Road map

Where to store strings

- Uniform-capacity strings
- Storing C-style strings

Road map

- Where to store strings
  - Uniform-capacity strings
  - Storing C-style strings
- Dynamic memory allocation
  - The mechanics
  - Examples

Road map

- Where to store strings
  - Uniform-capacity strings
  - Storing C-style strings
- Dynamic memory allocation
  - The mechanics
  - Examples
- Appendix: The Nulls

How can we work with strings?

```
bool is_comment(string);

// Concatenates sequence of strings, stripping comments.
string strip_concat(vector<string> lines)
{
    string result = "";

    for (string line : lines) {
        if (! is_comment(line)) {
            result += line + "\n";
        }
    }

    return result;
}
```

How can we work with strings?

```
bool is_comment(string);

// Concatenates sequence of strings, stripping comments.
string strip_concat(vector<string> lines)
{
    string result = "";

    for (string line : lines) {
        if (! is_comment(line)) {
            result += line + "\n";
        }
    }

    return result;
}
```

This is actually C++.

How can we work with strings?

```
bool is_comment(string);

// Concatenates sequence of strings, stripping comments.
string strip_concat(vector<string> lines)
{
    string result = "";

    for (string line : lines) {
        if (! is_comment(line)) {
            result += line + "\n";
        }
    }

    return result;
}
```

This is actually (very inefficient) C++.

Up next

Where to store strings

- Uniform-capacity strings
- Storing C-style strings

Dynamic memory allocation

- The mechanics
- Examples

Appendix: The Nulls

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

Up next

A uniform-capacity string

Can be passed, returned, assigned:

src/string80.c

```
struct string80
{
    char data[81];
};

typedef struct string80 string80_t;
```

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

Up next

So C uses pointers to 0-terminated `char` arrays

src/ptr_string.c

```c
// Copies characters from 0-terminated
// string src to dst.
void our_strcpy(char* dst, char const* src)
{
    while ( (*dst++ = *src++) )
    { }
}
```

So C uses pointers to 0-terminated `char` arrays

`src/ptr_string.c`

```
// Copies characters from 0-terminated
// string src to dst.
//
// PRECONDITION (unchecked): dst points to an array whose
// size is at least strlen(src) + 1.
void our_strcpy(char* dst, char const* src)
{
    while ( (*dst++ = *src++) )
    { }
}
```

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

So C uses pointers to 0-terminated `char` arrays

`src/ptr_string.c`

```
// Copies characters from 0-terminated
// string src to dst.
//
// PRECONDITION (unchecked): dst points to an array whose
// size is at least strlen(src) + 1.
void our_strcpy(char* dst, char const* src)
{
    while ( (*dst++ = *src++) )
    { }
}
```

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

But how can we ensure that `dst` has sufficient capacity?

Solution 1: Reuse existing memory

src/ptr_string.c

```
#include <ctype.h>

// Converts s to uppercase in place
void string_toupper_inplace(char* s)
{
    for (; *s; ++s) *s = toupper(*s);
}
```

Solution 1: Reuse existing memory

src/ptr_string.c

```c
#include <stdio.h>
#include <ctype.h>

// Converts s to uppercase in place
void string_toupper_inplace(char* s)
{
    for (; *s; ++s) *s = toupper(*s);
}

int main(void)
{
    char hello[] = "Hello, malloc";
    string_toupper_inplace(hello);
    puts(hello);   // HELLO, MALLOC
}
```

Solution 2: Kick the can (to the caller)

src/ptr_string.c

```
// Uppercases src into dst.
void strcpy_toupper(char* dst, char const* src)
{
    while (*src) *dst++ = toupper(*src++);
}
```

Solution 2: Kick the can (to the caller)

src/ptr_string.c

```
// Uppercases src into dst.
void strcpy_toupper(char* dst, char const* src)
{
    while (*src) *dst++ = toupper(*src++);
}

int main(void)
{
    char const hello[] = "Hello, malloc";
    char upper_hello[sizeof hello];

    strcpy_toupper(upper_hello, hello);

    puts(upper_hello);   // HELLO, MALLOC
}
```

Solution 3: Find some new memory?

src/stack_string.c

```c
// Returns an uppercase copy of src.
char* bad_string_clone_toupper(char const* src)
{
    char result[strlen(src) + 1];
    strcpy_toupper(result, src);
    return result;
}
```

Solution 3: Find some new memory?

src/stack_string.c

```
// Returns an uppercase copy of src.
char* bad_string_clone_toupper(char const* src)
{
    char result[strlen(src) + 1];
    strcpy_toupper(result, src);
    return result;
}

int main(void)
{
    char const* hello = "Hello, malloc";
    char* upper_hello = bad_string_clone_toupper(hello);
    puts(upper_hello);   // OH SHIT
}
```

Solution 3: Find some new memory?

src/stack_string.c

```c
// Returns an uppercase copy of src.
char* bad_string_clone_toupper(char const* src)
{
    char result[strlen(src) + 1];
    strcpy_toupper(result, src);
    return result;
}

int main(void)
{
    char const* hello = "Hello, malloc";
    char* upper_hello = bad_string_clone_toupper(hello);
    puts(upper_hello);   // OH SHIT
}
```

`bad_string_clone_toupper()` is wrong, and cannot work

You can't return a pointer to a stack (local) variable

Here's an example with `int`s:

```
int* faulty_inc(int z)
{
    int result = z + 1;
    return &result;
}
```

You can't return a pointer to a stack (local) variable

Here's an example with `int`s:

```
int* faulty_inc(int z)
{
    int result = z + 1;
    return &result;
}

int main(void)
{
    int* p =   faulty_inc(5);
    printf("%d\n", *p);
}
```

You can't return a pointer to a stack (local) variable

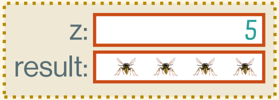
Here's an example with `int`s:

```
int* faulty_inc(int z)
{
    int result = z + 1;
    return &result;
}

int main(void)
{
    int* p = ▶ faulty_inc(5);
    printf("%d\n", *p);
}
```

You can't return a pointer to a stack (local) variable

Here's an example with `int`s:

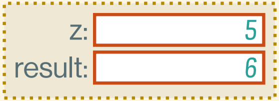
```
int* faulty_inc(int z)
{
    int result = z + 1;
    return &result;
}

int main(void)
{
    int* p = faulty_inc(5);
    printf("%d\n", *p);
}
```



You can't return a pointer to a stack (local) variable

Here's an example with `int`s:

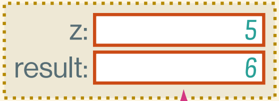
```c
int* faulty_inc(int z)
{
    int result = z + 1;
    return &result;
}

int main(void)
{
    int* p = faulty_inc(5);
    printf("%d\n", *p);
}
```



You can't return a pointer to a stack (local) variable

Here's an example with `int`s:

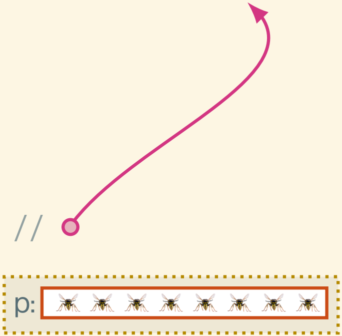
```
int* faulty_inc(int z)
{
    int result = z + 1;
    return &result; //
}

int main(void)
{
    int* p = faulty_inc(5);
    printf("%d\n", *p);
}
```



You can’t return a pointer to a stack (local) variable

Here’s an example with `int`s:

```
int* faulty_inc(int z)
{
    int result = z + 1;
    return &result;
}

int main(void)
{
    int* p =   faulty_inc(5); //
    printf("%d\n", *p);
}
```

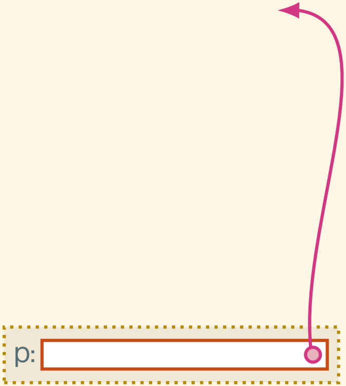


The result points to an object that is destroyed when `faulty_inc` returns!

You can’t return a pointer to a stack (local) variable

Here’s an example with `int`s:

```
int* faulty_inc(int z)
{
    int result = z + 1;
    return &result;
}

int main(void)
{
    int* p =   faulty_inc(5);
    printf("%d\n", *p);
}
```

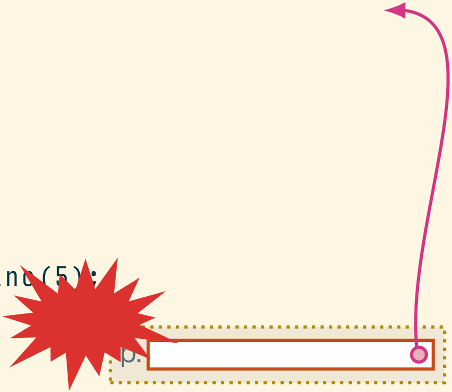


The result points to an object that is destroyed when `faulty_inc` returns!

You can't return a pointer to a stack (local) variable

Here's an example with `int`s:

```
int* faulty_inc(int z)
{
    int result = z + 1;
    return &result;
}

int main(void)
{
    int* p =   faulty_inc(5);
    printf("%d\n", *p);
}
```



The result points to an object that is destroyed when `faulty_inc` returns!

Up next

Up next

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 (undefined behavior))

Dynamic memory allocation: The rules

1. Every pointer returned by `malloc()` must be `NULL`-checked (because dereferencing `NULL` is UB (undefined behavior))
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 (undefined behavior))
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 (undefined behavior)
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 (undefined behavior)
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, a/k/a UB)

Dynamic memory allocation: The rules

1. Every pointer returned by `malloc()` must be `NULL`-checked (because dereferencing `NULL` is UB (undefined behavior)
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, a/k/a UB)
5. Except: `free(NULL)` is just fine

Up next

Heap allocation example

src/heap_string.c

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

// Makes an uppercase copy of `s`.
char* string_clone_toupper(char const* s)
{
    char* result = malloc(our_strlen(s) + 1);
    if (!result) return NULL;

    strcpy_toupper(result, s);
    return result;
}
```

Example of using str_toupper_clone

```
char* string_clone_toupper(char const*);

int main(void)
{
    char const *hello = "Hello, malloc";

    char* upper_hello = string_clone_toupper(hello);
    if (! upper_hello) {
        perror(NULL);
        return 1;
    }

    puts(upper_hello);  // HELLO, MALLOC
    free(upper_hello);
}
```

Concatenating two strings, result in the heap

src/string_fun.c

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

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

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

    return result;
}
```

Concatenating two strings, result in the heap, v. 2

Using `snprintf(3)`:

```
char* string_concat(char const* s, char const* t)
{
    char c;
    size_t size = snprintf(&c, 1, "%s%s", s, t);

    char* result = malloc(size);
    if (! result) return NULL;

    snprintf(result, size, "%s%s", s, t);
    return result;
}
```

Our initial example

src/string_fun.c

```
char* strip_concat(char const* const lines[], size_t n)
{
    size_t total_len = 0;


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

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

    char* fill = result;

    for (size_t i = 0; i < n; ++i) {
        if (is_comment(lines[i])) continue;
        strcpy(fill, lines[i]);
        fill += strlen(fill);
        *fill++ = '\n';
    }

    *fill = 0;

    return result;
}
```

Our initial example

```
char* strip_concat(char const* const lines[], size_t n)
{
    size_t total_len = 0;


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

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

    char* fill = result;

    for (size_t i = 0; i < n; ++i) {
        if (is_comment(lines[i])) continue;
        strcpy(fill, lines[i]);
        fill += strlen(fill);
        *fill++ = '\n';
    }

    *fill = 0;

    return result;
}
```

src/string_fun.c

test/test_string_fun.c

Our initial example

```c
char* strip_concat(char const* const lines[], size_t n)
{
    size_t total_len = 0;


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

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

    char* fill = result;

    for (size_t i = 0; i < n; ++i) {
        if (is_comment(lines[i])) continue;
        strcpy(fill, lines[i]);
        fill += strlen(fill);
        *fill++ = '\n';
    }

    *fill = 0;

    return result;
}
```

src/string_fun.c

test/test_string_fun.c

src/strip_concat_main.c

– Next time: Undefined Behavior –

```
struct page* intentionally_blank = NULL;
```

Up next

`NULL` versus null versus NUL

Thing	Type of Thing	Purpose of Thing

`NULL` versus null versus NUL

Thing	Type of Thing	Purpose of Thing
"[a] null [pointer]"	*T** (for any *T*)	stands for a missing object

`NULL` versus null versus NUL

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 null versus NUL

Thing	Type of Thing	Purpose of Thing
"[a] null [pointer]"	*T*`*` (for any *T*)	stands for a missing object
`NULL`	`void*`	null pointer constant
`(char)0`	`char`	string terminator value (a/k/a NUL)

`NULL` versus null versus NUL

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

`NULL` versus null versus NUL

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

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