Overview

• C Review
• Void pointers
• Allocation
• structs
void*

(Void Pointers)
void*

- Like any other pointer, it refers to some memory address
- However, it has no associated type, and cannot be dereferenced directly
- Question: why can’t it be dereferenced?
No Dereferencing

```c
void* p = 2;
*p; // get what’s at p
```

<table>
<thead>
<tr>
<th>Value</th>
<th>0x21</th>
<th>0x00</th>
<th>0x01</th>
<th>0x52</th>
<th>0xF0</th>
<th>0xAB</th>
<th>0x2C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

- `void*` is a value without context
- Without context, there is no way to know how to interpret the value (`int?` `char?` `double?`)
How to Use a `void*`

- A `void*` cannot be dereferenced directly
- However, it is possible to cast a `void*` to another type

```c
char* str = "moo";
void* p = str;
printf( "%s\n", (char*)p );
```
How to Use a *void*

- A *void* also coerces into other pointer types

```c
char* str = "moo";
void* p = str;
char* str2 = p; // no errors
```
Caveat

• A `void*` also coerces into other pointer types

• The compiler will trust you blindly

```c
char* str = "moo";
void* p = str;

// no compiler errors, but
// this is probably not what
// is desired
int* nums = p;
```
Why a `void*`?

- Allows for generic data structures
  - A list of `ints` looks a lot like a list of `chars`
- Can refer to some block of memory without context
- Up next: why anyone would want to do that
Dynamic Memory Allocation
Motivation

• We want to read in a dictionary of words

• Before reading it in:
  • We don’t know how many words there are
  • We don’t know how big each word is

apple    banana    <<empty>>    aardvark
pear
Possible Solution

• Allocate the maximum amount you could ever need

• Question: why is this generally not a good solution? (2 reasons)

// 1000 words max with
// 100 characters max per word
char dictionary[1000][100];
Problems

• Most things do not have a good “maximum” you can get a grasp of

• Your program always needs the maximum amount of memory, and usually the vast majority is completely wasted
What is Desired

- A way to tell the computer to give a certain amount of memory to a program as it runs
- Only what is explicitly requested is allocated
Dynamic Memory Allocation

- Dynamic: as the program runs
- Memory allocation: set aside memory
malloc

• The most generic way to allocate memory
• Takes the number of bytes to allocate
• Returns a void* to the block of memory allocated

// size_t is an integral defined elsewhere
void* malloc( size_t numBytes );
Using `malloc`

- The `sizeof` operator comes in handy
- Returns an integral size as a `size_t`
- For example: allocate room for 50 integers dynamically:

```c
// dynamically
int* nums1;
nums1 = malloc( sizeof( int ) * 50 );
```

```c
int nums2[ 50 ]; // statically
```
Question

- Why did we `malloc with sizeof(int)` instead of `sizeof(int*)`?
- We assigned it to an `int*`, after all

```c
int* nums1;
nums1 = malloc( sizeof( int ) * 50 );
```
Answer

- We wanted room for 50 integers, not integer pointers

```c
int* nums1;
nums1 = malloc( sizeof( int ) * 50 );
```
Importance

• Static allocation can only be done with constants

• Dynamic allocation can be done with variables

```c
int numToAllocate;
scanf( "%i", &numToAllocate );
int* nums = malloc(sizeof( int ) * numToAllocate);
int nums2[ numToAllocate ]; // ERROR
```
Memory Contents

• The contents of the memory allocated by malloc is undefined

• You will need to initialize it yourself with a loop (or by using the `memset` function)
free

• Once we are done using a block of memory, call free on it

• If a block is never freed, it is called a memory leak

• Memory is still allocated but wasted

```c
int* nums;
nums = malloc( sizeof( int ) * 50 );
...
// done with nums
free( nums );
```
malloc1.c,
malloc2.c
On Calling `free`

- With static allocation, the compiler handles deallocation for you.
- With dynamic allocation, you must call `free` yourself.
- The simple act of knowing when to call `free` can be hard.
- In general, mathematically unsolvable!
Memory-Related Bugs

• What is wrong with this code?

```c
int* foo() {
    int x = 7;
    return &x;
}

void bar() {
    int* p = foo();
    *p = 8;
}
```
Memory-Related Bugs

• What is wrong with this code?

```c
int* foo() {
    int x = 7;
    return &7;
}
```

```c
void bar() {
    int* p = foo();
    *p = 8;
}
```

Who knows what `p` points to? (undefined)

Space for `x` is deallocated when `foo` returns

Called a “dangling pointer”
Memory-Related Bugs

- What is wrong with this code?

```c
void foo() {
    int* p = (int*)malloc(sizeof(int));
    *p = 7;
    free(p);
    *p = 8;
}
```
Memory-Related Bugs

• What is wrong with this code?

```c
void foo() {
    int* p = (int*)malloc(sizeof(int));
    *p = 7;
    free(p);
    *p = 8;
}
```

*p* is deallocated, then used. Called a “use after free”
Memory-Related Bugs

- What is wrong with this code?

```c
void foo() {
    int* p = (int*)malloc(sizeof(int));
    *p = 7;
}
```
Memory-Related Bugs

• What is wrong with this code?

```c
void foo() {
    int* p = (int*)malloc(sizeof(int));
    *p = 7;
}
```

`p` is allocated, but never deallocated. This is a memory leak.
structs
Question

• What is a struct?
Basic Idea

• A way to group a **fixed** number of items, of potentially **different** types

• Arrays: multiple items of the same type

• A way to create whole new datatypes
Example

// defining
typedef struct _person {
  char* name;
  char* address;
  int phone;
} person;

...

// using
struct _person p1;
person p2, p3;
Questions

typedef struct _person {
    char* name;
    char* address;
    int phone;
} person;

... 

person p;

• How do I access p’s phone field?
• How do I update p’s name field?
typedef struct _person {
    char* name;
    char* address;
    int phone;
} person;

...  

person p;

- p.phone
- p.name = NULL
Passing structs

• structs are copied when passed to functions

struct blah { int x; };

void foo(struct blah b) { b.x = 7; }

int main() {
    struct blah p;
    p.x = 1;
    foo(p);
    printf("%d", p.x); // prints what?
    return 0;
}
Passing structs

• structs are copied when passed to functions

```c
struct blah { int x; };

void foo(struct blah b) { b.x = 7; }

int main() {
    struct blah p;
    p.x = 1;
    foo(p);
    printf("%d", p.x); // prints 1
    return 0;
}
```
Passing structs

- Often passed via pointer, since they tend to be at least of moderate size
- Avoids copying
Dealing with pointers to structs can get obnoxious because of parentheses

```c
struct blah { int x; };

void foo(struct blah* b) {
    (*b).x = 7;
}
```
Pointers to structs

- Can alleviate this with the equivalent arrow operator

```c
struct blah { int x; };

void foo(struct blah* b) {
  (*b).x = 7;
  b->x = 8;
}
```
Question

• How might we allocate a struct?
Answer

• How might we allocate a `struct`?

```c
struct blah { int x; };
...
struct blah* b = malloc(sizeof(struct blah));
```
Question

• What about an array of size \( n \) of structs?

```c
struct blah { int x; };
```
Answer

- What about an array of size $n$ of structs?

```c
struct blah { int x; };
...

struct blah* arr = malloc(sizeof(struct blah) * n);
arr[3].x = 7; // n > 3
```
Putting it All Together (If Time Allows)
Problem Description

• We have a file in the following format:

  3
  Apple
  Giraffe
  Hover

• First line is the number of words, and subsequent lines are words

• Each word is 20 characters or less
Problem

- Read it into an array of type `char**` (an array of strings)
- Dynamic allocation must be used
Related Problem

• Read it into an array of type `char*` (a single string)
• Dynamic allocation must be used
• How do we access individual strings?