

C – Structs and Dynamic Memory Allocation

Karthik Dantu

Ethan Blanton

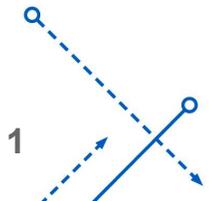
Computer Science and Engineering

University at Buffalo

`kdantu@buffalo.edu`

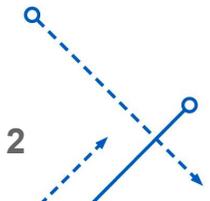
Portions of this lecture are borrowed from the U-W CSE 333 course slides

Karthik Dantu



Administrivia

- Piazza has a search bar – use it!
- Corollary – name your posts descriptively so others can find them!
- GitHub – commit regularly
- Git – learn features such as tagging
- Don't push `.o` and executable files or other build products



Memory Allocation

So far, we have seen two kinds of memory allocation:

```
int counter = 0;    // global var

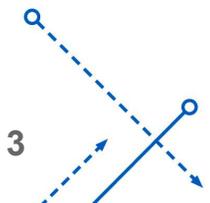
int main(int argc, char** argv) {
    counter++;
    printf("count = %d\n", counter);
    return EXIT_SUCCESS;
}
```

- `counter` is **statically**-allocated
 - Allocated when program is loaded
 - Deallocated when process gets reaped

```
int foo(int a) {
    int x = a + 1;    // local var
    return x;
}

int main(int argc, char** argv) {
    int y = foo(10); // local var
    printf("y = %d\n", y);
    return EXIT_SUCCESS;
}
```

- `a`, `x`, `y` are **automatically**-allocated
 - Allocated when function is called
 - Deallocated when function returns

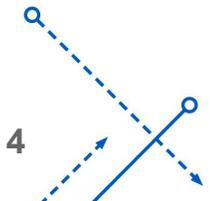


Dynamic Allocation

- Situations where static and automatic allocation aren't sufficient:
 - We need memory that persists across multiple function calls but not the whole lifetime of the program
 - We need more memory than can fit on the Stack
 - We need memory whose size is not known in advance to the caller

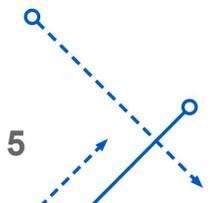
```
// this is pseudo-C code  
char* ReadFile(char* filename) {  
    int size = GetFileSize(filename);  
    char* buffer = AllocateMem(size);  
  
    ReadFileIntoBuffer(filename, buffer);  
    return buffer;  
}
```

Karthik Dantu



Dynamic Memory Allocation

- What we want is **dynamically**-allocated memory
Your program explicitly requests a new block of memory
The language allocates it at runtime, perhaps with help from OS
Dynamically-allocated memory persists until either:
Your code explicitly deallocated it (manual memory management)
A garbage collector collects it (automatic memory management)
- C requires you to manually manage memory
Gives you more control, but causes headaches

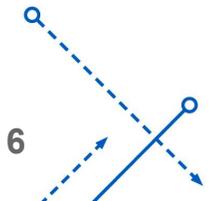


Aside: NULL

- **NULL** is a memory location that is **guaranteed to be invalid**
 - In C on Linux, **NULL** is `0x0` and an attempt to dereference **NULL** *causes a segmentation fault*
- Useful as an indicator of an uninitialized (or currently unused) pointer or allocation error
 - It's better to cause a segfault than to allow the corruption of memory!

```
int main(int argc, char** argv) {  
    int* p = NULL;  
    *p = 1; // causes a segmentation fault  
    return EXIT_SUCCESS;  
}
```

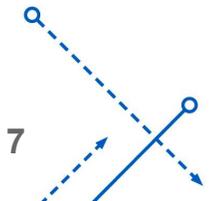
Karthik Dantu



malloc()

- General usage: `var = (type*) malloc(size in bytes)`
- **malloc** allocates a block of memory of the requested size
Returns a pointer to the first byte of that memory
And **returns NULL** if the memory allocation failed!
You should assume that the memory initially contains garbage
You'll typically use `sizeof` to calculate the size you need

```
// allocate a 10-float array  
float* arr = (float*) malloc(10*sizeof(float));  
if (arr == NULL) {  
    return errcode;  
}  
... // do stuff with arr
```



calloc()

- General usage:

```
var = (type*) calloc(num, bytes per element)
```

- Like **malloc**, but also zeros out the block of memory

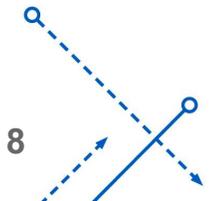
Helpful when zero-initialization wanted (but don't use it to mask bugs – fix those)

Slightly slower; but useful for non-performance-critical code

malloc and **calloc** are found in `stdlib.h`

```
// allocate a 10-double array  
double* arr = (double*) calloc(10, sizeof(double));  
if (arr == NULL) {  
    return errcode;  
}  
... // do stuff with arr
```

Karthik Dantu

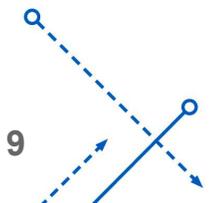


free()

- Usage: `free(pointer);`
- Deallocates the memory pointed-to by the pointer
Pointer *must* point to the first byte of heap-allocated memory
(*i.e.* something previously returned by **malloc** or **calloc**)
Freed memory becomes eligible for future allocation
Pointer is unaffected by call to free
Defensive programming: can set pointer to `NULL` after freeing it

```
float* arr = (float*) malloc(10*sizeof(float));  
if (arr == NULL)  
    return errcode;  
...           // do stuff with arr  
free(arr);  
arr = NULL;   // OPTIONAL
```

Karthik Dantu





Practice

- Which lines have errors?

A. Line 1

B. Line 2

C. Line 4

D. Line 6

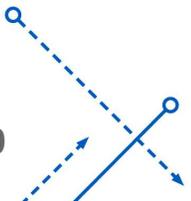
E. We're lost...

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

1   a[2] = 5;
2   b[0] += 2;
3   c = b+3;
4   free(&(a[0]));
5   free(b);
6   free(b);
7   b[0] = 5;

    return EXIT_SUCCESS;
}
```





Practice

- Which line below is first *guaranteed* to cause an error?

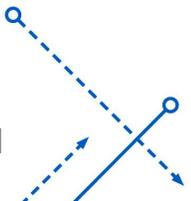
- A. Line 1
- B. Line 4
- C. Line 6
- D. Line 7
- E. We're lost...

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

1   a[2] = 5;
2   b[0] += 2;
3   c = b+3;
4   free(&(a[0]));
5   free(b);
6   free(b);
7   b[0] = 5;

    return EXIT_SUCCESS;
}
```



Memory Corruption

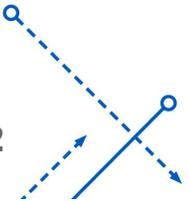
- There are all sorts of ways to corrupt memory in C

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int a[2];
    int* b = malloc(2*sizeof(int));
    int* c;

    a[2] = 5;    // assign past the end of an array
    b[0] += 2;   // assume malloc zeros out memory
    c = b+3;    // mess up your pointer arithmetic
    free(&(a[0])); // free something not malloc'ed
    free(b);
    free(b);    // double-free the same block
    b[0] = 5;   // use a freed pointer

    // any many more!
    return EXIT_SUCCESS;
}
```



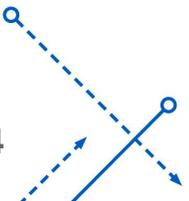
Memory Leak

- A **memory leak** occurs when code fails to deallocate dynamically-allocated memory that is no longer used
e.g. forget to **free** malloc-ed block, lose/change pointer to malloc-ed block
- What happens: program's VM footprint will keep growing
This might be OK for *short-lived* program, since all memory is deallocated when program ends
Usually has bad repercussions for *long-lived* programs
 - Might slow down over time (*e.g.* lead to VM thrashing)
 - Might exhaust all available memory and crash
 - Other programs might get starved of memory



Derived Data Types

- Arrays require all elements to be of the same data type.
- Many times, we want to group items of different types in a structure
- E.g., grade roster = {Name (char *), UBID (int) , Active (bool) , Lab1 (float), PA0 (float), ..}
- `struct`: Derived data type composed of members that are basic or other derived data types



Structured Data

- A **struct** is a C datatype that contains a set of fields
Similar to a Java class, but with no methods or constructors
Useful for defining new structured types of data
Behave similarly to primitive variables

- Generic declaration:

```
struct tagname {  
    type1 name1;  
    ...  
    typeN nameN;  
};
```

```
// the following defines a new  
// structured datatype called  
// a "struct Point"  
struct Point {  
    float x, y;  
};  
  
// declare and initialize a  
// struct Point variable  
struct Point origin = {0.0,0.0};
```

Declaring structs

Just specify the struct
(no space reserved)

```
// the following defines a new  
// structured datatype called  
// a "struct Point"  
struct Point {  
    float x, y;  
};
```

specify the struct and
declare a variable
(space reserved)

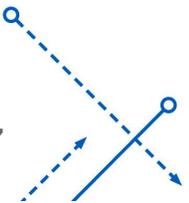
```
// the following defines a new  
// structured datatype called  
// a "struct Point" and declares  
// a variable "origin" of type  
// struct Point  
struct Point {  
    float x, y;  
} origin;
```



Using structs

- Use “.” to refer to a field in a struct
 - Use “->” to refer to a field from a struct pointer
- Dereferences pointer first, then accesses field

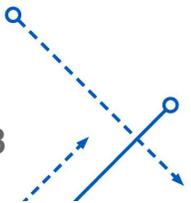
```
struct Point {  
    float x, y;  
};  
  
int main(int argc, char** argv) {  
    struct Point p1 = {0.0, 0.0}; // p1 is stack allocated  
    struct Point* p1_ptr = &p1;  
  
    p1.x = 1.0;  
    p1_ptr->y = 2.0; // equivalent to (*p1_ptr).y = 2.0;  
    return EXIT_SUCCESS;  
}
```



Copy by Assignment

- You can assign the value of a struct from a struct of the same type – *this copies the entire contents!*

```
struct Point {  
    float x, y;  
};  
  
int main(int argc, char** argv) {  
    struct Point p1 = {0.0, 2.0};  
    struct Point p2 = {4.0, 6.0};  
  
    printf("p1: {%f,%f} p2: {%f,%f}\n", p1.x, p1.y, p2.x, p2.y);  
    p2 = p1;  
    printf("p1: {%f,%f} p2: {%f,%f}\n", p1.x, p1.y, p2.x, p2.y);  
    return EXIT_SUCCESS;  
}
```



typedef

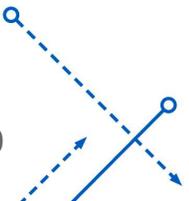
- Generic format: `typedef type name;`
- Allows you to define new data type *names/synonyms*
Both `type` and `name` are usable and refer to the same type
Be careful with pointers – `*` before `name` is part of `type`!

```
// make "superlong" a synonym for "unsigned long long"
typedef unsigned long long superlong;

// make "str" a synonym for "char*"
typedef char *str;

// make "Point" a synonym for "struct point_st { ... }"
// make "PointPtr" a synonym for "struct point_st*"
typedef struct point_st {
    superlong x;
    superlong y;
} Point, *PointPtr; // similar syntax to "int n, *p;"

Point origin = {0, 0};
```



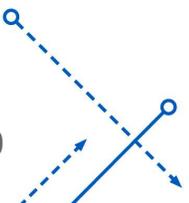
Dynamically-allocated Structs

- You can **malloc** and **free** structs, just like other data type
`sizeof` is particularly helpful here

```
// a complex number is a + bi
typedef struct complex_st {
    double real;    // real component
    double imag;   // imaginary component
} Complex, *ComplexPtr;

// note that ComplexPtr is equivalent to Complex*
ComplexPtr AllocComplex(double real, double imag) {
    Complex* retval = (Complex*) malloc(sizeof(Complex));
    if (retval != NULL) {
        retval->real = real;
        retval->imag = imag;
    }
    return retval;
}
```

Karthik Dantu



Aside: Arguments in C

- In most languages, arguments can be
 - Passed by value
 - Passed by reference
- C uses pass-by-value
- Example

```
before swap a = 1
before swap b = 2
after swap a = 1
after swap b = 2
```

```
void swap(int a, int b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int main() {
    int a = 1;
    int b = 2;

    printf("a before swap=%d\n", a);
    printf("b before swap=%d\n", b);
    swap(a, b);
    printf("a after swap=%d\n", a);
    printf("b after swap=%d\n", b);

    return 0;
}
```

<https://denniskubes.com/2012/08/20/is-c-pass-by-value-or-reference/>

Karthik Dantu

Aside: Arguments in C

- **FIX:** pass a pointer to the variables

```
before swap a = 1
before swap b = 2
after swap a = 2
after swap b = 1
```

```
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main() {
    int a = 1;
    int b = 2;

    printf("a before swap=%d\n", a);
    printf("b before swap=%d\n", b);
    swap(&a, &b);
    printf("a after swap=%d\n", a);
    printf("b after swap=%d\n", b);

    return 0;
}
```

<https://denniskubes.com/2012/08/20/is-c-pass-by-value-or-reference/>

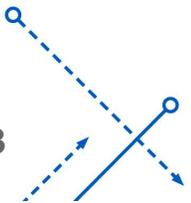
Karthik Dantu

Structs as Arguments

- Structs are passed by value, like everything else in C
Entire struct is copied
To manipulate a struct argument, pass a pointer instead

```
typedef struct point_st {  
    int x, y;  
} Point, *PointPtr;  
  
void DoubleXBroken(Point p)    { p.x *= 2; }  
  
void DoubleXWorks(PointPtr p) { p->x *= 2; }  
  
int main(int argc, char** argv) {  
    Point a = {1,1};  
    DoubleXBroken(a);  
    printf("( %d, %d )\n", a.x, a.y);    // prints: ( 1, 1 )  
    DoubleXWorks(&a);  
    printf("( %d, %d )\n", a.x, a.y);    // prints: ( 2, 1 )  
    return EXIT_SUCCESS;  
}
```

Karthik Dantu



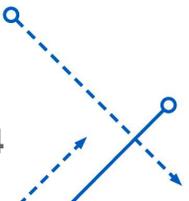
Returning Structs

- Exact method of return depends on calling conventions
Often returned in memory for larger structs

```
// a complex number is a + bi
typedef struct complex_st {
    double real;    // real component
    double imag;   // imaginary component
} Complex, *ComplexPtr;

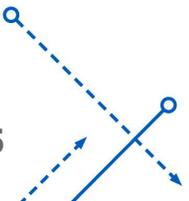
Complex MultiplyComplex(Complex x, Complex y) {
    Complex retval;

    retval.real = (x.real * y.real) - (x.imag * y.imag);
    retval.imag = (x.imag * y.real) - (x.real * y.imag);
    return retval; // returns a copy of retval
}
```



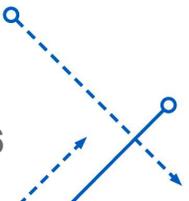
Pass Copy of Struct or Pointer?

- Value passed: passing a pointer is cheaper and takes less space unless struct is small
- Field access: indirect accesses through pointers are a bit more expensive and can be harder for compiler to optimize
- For small structs (like `struct complex_st`), passing a copy of the struct can be faster and often preferred if function only reads data; for large structs use pointers



Exercise #1

- Write a program that defines:
 - A new structured type Point
 - Represent it with `floats` for the x and y coordinates
 - A new structured type Rectangle
 - Assume its sides are parallel to the x-axis and y-axis
 - Represent it with the bottom-left and top-right Points
 - A function that computes and returns the area of a Rectangle
 - A function that tests whether a Point is inside of a Rectangle



Extra: Exercise #2

- Implement `AllocSet()` and `FreeSet()`

`AllocSet()` needs to use `malloc` twice: once to allocate a new `ComplexSet` and once to allocate the “points” field inside it
`FreeSet()` needs to use `free` twice

```
typedef struct complex_st {
    double real;    // real component
    double imag;   // imaginary component
} Complex;

typedef struct complex_set_st {
    double    num_points_in_set;
    Complex* points;    // an array of Complex
} ComplexSet;

ComplexSet* AllocSet(Complex c_arr[], int size);
void FreeSet(ComplexSet* set);
```

