CS 240: Programming in C

Lecture 6: Structures,
Declaration vs. Definition,
String Functions

PURDUE UNIVERSITY

Announcements

- Homework 3 is out
- Homework 2 is due Wednesday at 9:00 pm
- Remember the code standard!

Homework 2 & 3

- Strings in C end with a special character, ‘\0’
  - Known as the NUL terminator
- scanf will add ‘\0’ to any strings it reads in
- You must make sure your buffers are large enough to contain it!

Homework 2 & 3

```
#define MAX_NAME_LEN (40)

char buf[MAX_NAME_LEN];
fscanf(in_fp, “%40[^\n]”, buf);
```

- This will store 41 characters into buf
  - It reads 40, then adds ‘\0’
- Causes buffer overflow

Homework 2 & 3

```
#define MAX_NAME_LEN (40)

char buf[MAX_NAME_LEN];
fscanf(in_fp, “%39[^\n]”, buf);
```

or

```
#define MAX_NAME_LEN (40)

char buf[MAX_NAME_LEN + 1];
fscanf(in_fp, “%40[^\n]”, buf);
```

typedef

- Standard C has a set of built-in types
  - int, char, float, etc.
- You can create your own types
  - As long as they’re not already C keywords
- Examples:

```
typedef int my_number;
typedef double my_array[3];

my_number n = 5;
my_array arr = { 1.5, 2.9, 3.7 };
```

When to use typedef

- Use typedef when you have a variable type that is used a lot and...
  - ...has a really long, cumbersome description, e.g.:

```
typedef double * const *my_ptr;
```

  - ...has a parameter type that you might change sometime later, e.g.:

```
typedef double my_array[3];
```

  - ...it is a structure (more on this today)
  - ...it is so confusing that you really can't deal with it without creating a typedef (more later this semester)

Getting the syntax of typedef…

- The syntax of typedef might seem backwards to you. Here is how to always get it right…
  - Pretend that you're defining a variable of the type you want to see
  - Let the variable name be the name of the new type
  - Add 'typedef' to the beginning of the definition

typedef syntax cont.

- For instance, if you want a type called uint5 that can be used to define an array of five unsigned integers,
  - Pretend you're defining a variable:

```
unsigned int uint5[5];
```

  - Make it a type:

```
typedef unsigned int uint5[5];
```

  - Use it:

```
uint5 arr = { 1, 2, 3, 4, 5 };
```

Introduction to structures

- Large programs usually have many pieces of data
- Instead of creating a separate variable for each one, it is helpful to group them together to better organize them
- A structure is the thing that allows you to use one name to refer to many variables

What does a structure look like?

- Type declaration:

```
struct my_data {
  int age;
  float height;
};
```


Note the semicolon!

- Storage definition and initialization:

```
struct my_data my_var = { 19, 5.3 };
```

Declaration / definition together

```
struct my_data {
  int age;
  float height;
} my_var = { 19, 5.3 };
```

Accessing elements

- Once a structure variable has been defined, you can access its internal elements with the dot (.) operator:

```
my_var.height = 6.1;
int x = my_var.age;

printf("Age: %d, Height: %f\n",
       my_var.age, my_var.height);
```

Properties of structures

- Anything that can be defined can also be defined inside of a struct { ... }
  - **Except functions**
  - You can put arrays in structures
  - You can put other structures in structures
  - You can put them in any order:

```
struct stuff {
  double d_var;
  int    i_arr[100];
  float  f_arr[20];
};
```

More properties of structures

- There's no limit to the number of elements in a structure
  - Each must have a unique name, though
- Structures can be passed to functions and returned from functions:

```
struct my_data grow(struct my_data start) {
  struct my_data finish;

  finish.age = start.age + 1;
  finish.height = start.height * 1.1;
  return finish;
}
```

Even more properties of structures

- You can assign one structure variable to another:

```
int main() {
  struct my_data small_kid = { 1, 2.5 };
  struct my_data big_kid = { 3, 4.1 };

  small_kid = big_kid;
}
```

- Note: this is an *assignment*. Not an *initialization*.

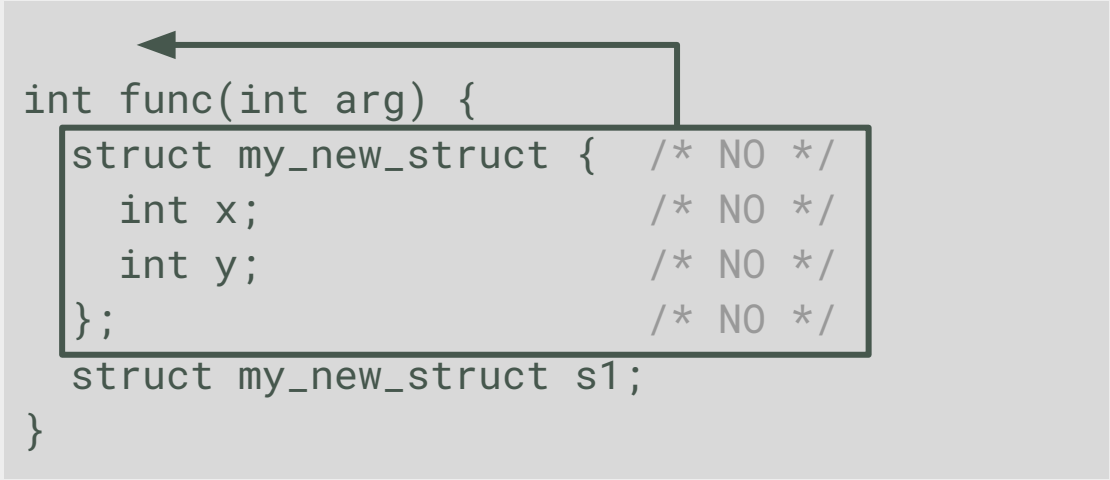
```
small_kid = { 2, 3.2 }; /* nope! */
```

- But with C99...

```
small_kid = (struct my_data) { 2, 3.2 };
```

Where to put declarations

- Structure variables can be defined inside or outside of a function
  - Keep the declaration outside



How about a struct typedef

- It works the same as before
  - Pretend you're defining a struct variable:

```
struct my_data md;
```

  - Change it to a typedef:

```
typedef struct my_data md;
```

  - Use it as a type:

```
md my_var = { 12, 5.1 };
```

Declaring a struct typedef

- Often we declare the typedef when we declare the struct

```
typedef struct my_new_struct {
  int x;
  float y;
} new_struct_type;

new_struct_type var = { 1, 3.2 };
```

Complete example

```
struct coord {
  float x;
  float y;
  float z;
};

typedef struct coord coord_t;
```

A function that uses it

```
coord_t add_coord(coord_t a, coord_t b) {
  coord_t sum = { 0.0, 0.0, 0.0 };
  sum.x = a.x + b.x;
  sum.y = a.y + b.y;
  sum.z = a.z + b.z;

  return sum;
}
```

Another function that uses it

```
#include <stdio.h>

void print_coord(coord_t coord) {
  printf("(%f, %f, %f)", coord.x, coord.y, coord.z);

  return;
}
```

Do something with it

```
int main() {
  coord_t one = { 2, 4, 6 };
  coord_t two = { 1, 2, 3 };

  print_coord(add_coord(one, two));

  return 0;
}
```

Definitions vs. declarations

- Definition: **allocates storage** for a variable (or function)
- Declaration: **announces** the properties of a variable (or function)

What's this?

```
struct hey {
  int zap;
  float zing;
};
```

And this?

```
struct point {
  int x;
  int y;
} var;
```

More examples

- A **function prototype** is a forward declaration:

```
double some_function(int, float);
```

- The creation of that function is a definition:

```
double some_function(int age, float height) {
  return age * height;
}
```

- What's a typedef? It's a declaration...
  - It turns a variable definition into a type declaration

Structures inside structures

- You can define structure variables inside other structures:

```
struct segment {
  struct coord one;
  struct coord two;
};
```

- When you initialize, you need extra braces:

```
struct segment my_seg = { {0, 0, 0},
                          {0, 0, 0} };
```

Arrays in structures

- You can define array variables in structure declarations:

```
struct person {
  char name[40];
  char rank[15];
  int  codes[4];
};
```

- Definition and initialization:

```
struct person cap = { “Kirk”,
                      “Captain”,
                      {10, 20, 25, 9} };
```

Assignment

- Take another definition and initialization:

```
struct person sp = { “”, “”, {0, 0, 0, 0} };
```

- Assigning elements of the arrays is usually done individually:

```
#include <string.h>
strncpy(sp.name, “Spock”, 40);
strncpy(sp.rank, “Lieutenant Commander”, 15);
sp.codes[0] = 5;
sp.codes[1] = 10;
sp.codes[2] = 15;
sp.codes[3] = 20;
```

Compound literals

- But with C99, it doesn't have to be

```
sp = (struct person) { “Who”, “Wat”, {1, 2, 3, 4} };
```

Complete example

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

struct person {
  char name[40];
  char rank[15];
  int codes[4];
};
typedef struct person person_t;

void print_person(person_t);
```

Complete example

```
int main() {
  person_t cap = { “Kirk”, “Captain”,
                   {10, 20, 25, 9} };
  person_t sp = { “”, “”, {0, 0, 0, 0} };

  strncpy(sp.name, “Spock”, 40);
  strncpy(sp.rank, “Lieutenant Commander”, 15);
  sp.codes[0] = -1;
  sp.codes[1] = 10;
  sp.codes[2] = 15;
  sp.codes[3] = 20;
```

Complete example

```
  print_person(cap);
  print_person(sp);
  return 0;
}

void print_person(person_t pt) {
  printf("Name:  %s\n", pt.name);
  printf("Rank:  %s\n", pt.rank);
  printf("Codes: %d, %d, %d, %d\n\n",
         pt.codes[0], pt.codes[1],
         pt.codes[2], pt.codes[3]);
  return;
}
```

Complete example (output)

```
$ vi ex1.c
$ gcc -std=c99 -g -o ex1 ex1.c
$ ./ex1
Name:  Kirk
Rank:  Captain
Codes: 10, 20, 25, 9

Name:  Spock
Rank:  Lieutenant Comm□□□□
Codes: -1, 10, 15, 20

$
```

Avoid global variables

- A global definition is one that exists outside of any function. Avoid where possible.

```
int my_count = 0;

int do_count() {
  int index = 0;
  for (index = 0; index < 100; index++) {
    my_count = my_count + index;
  }
  return my_count;
}
```

String comparison functions

- You can compare two strings with strcmp():

```
int result;

result = strcmp(one, two);
if (result == 0) {
  printf(“The strings are equal.\n”);
}
else if (result < 0) {
  printf(“%s comes before %s\n”, one, two);
}
else {
  printf(“%s comes before %s\n”, two, one);
}
```

General string operations

- Determine the length of a string with strlen():

```
char name[20] = “Chris”;
printf(“Length of string %s is %d\n”, name, strlen(name));
```

  - How many bytes do you need to store it?
- When using string functions, always:

```
#include <string,h>
```

- We can explain strings more when we study pointers

For next lecture

- If this stuff is confusing, review K&R 6.1-6.3, 6.7-6.9
  - ...and/or Beej Chapter 8
- Practice the examples!!
- Finish homework 2
- Start on homework 3

Slides

- Slides are heavily based on Prof. Turkstra's material from previous semesters.