Structures

2024 Winter APS 105: Computer Fundamentals

Jon Eyolfson

Lecture 28

1.0.0

You Can Group Variables Within a Structure

You can create your own type with `struct`, its syntax is:

```
struct <name> {
  <variable declarations>
};
```

Where you replace:

`<name>` with the name you'd like to give the group of variables
`<variable declarations>` with as many variable declarations as you wish

You should define a `struct` just below the includes, and not within a function

Let's Calculate the Distance Between Two Points

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

double distance(double x1, double y1, double x2, double y2) {
    return sqrt(pow(x2 - x1, 2.0) + pow(y2 - y1, 2.0));
}

int main(void) {
    double x1 = 1.0;
    double y1 = 2.0;
    double x2 = 4.0;
    double y2 = 6.0;
    double d = distance(x1, y1, x2, y2);
    printf("%.1lf\n", d);
    return EXIT_SUCCESS;
}
```

It’s Not Too Bad to Remember Which Variable is Which

However, we may write more functions, like:

```
void print(double x, double y) {
    printf("point(%.1lf, %.1lf)\n", x, y);
}
```

Now it’s a bit awkward to use, and may lead to errors

We may accidentally write:

```
print(x1, y2);
```

Which would not be one of the two points we intended

We Can Make a Structure That Represents a Point

```
struct point {
    double x;
    double y;
};
```

This creates a new type called: `struct point`

We can create a variable using it with: `struct point p1;`

We can access the inner variables (usually called fields or members) with `.`
- For example, we can use `p1.x` or `p1.y` in this case

You Can Rename Types with `typedef`

The syntax of a `typedef`, is:

```
  typedef <type> <new_name>;
```

Where you replace:

  `<new_name>` by the name of whatever you’d like to name your type
  `<type>` by the type you would like to use when you use `<new_name>`

We Can Use `typedef` Is to Save Us from Writing `struct`

You’re able to create a `struct` without giving it a name, you may write:

```
typedef struct {
    double x;
    double y;
} point_t;
```

Afterwards, you can create a variable with:

```
point_t p1;
```

However, you usually still give the `struct` a name:

```
typedef struct point {
    double x;
    double y;
} point_t;
```

We Could Create Our Two Points and Initialize the Fields

```
point_t p1;
p1.x = 1.0;
p1.y = 2.0;
point_t p2;
p2.x = 4.0;
p2.y = 6.0;
```

We Can Initialize Structures Like Arrays

We can write:

```
point_t p1 = {1.0, 2.0};
point_t p2 = {4.0, 6.0};
```

C sets the values in order within the structure

You may use the field names to set the values in a different order:

```
point_t p1 = {
  .y = 2.0,
  .x = 1.0,
};
```

We Can Re-write the Functions to Use Point Structures

```c
typedef struct point {
    double x;
    double y;
} point_t;

double distance(point_t p1 , point_t p2) {
    return sqrt(pow(p2.x - p1.x, 2.0) + pow(p2.y - p1.y, 2.0));
}
void print(point_t p) {
    printf("point(%lf, %lf)\n", p.x, p.y);
}

int main(void) {
    point_t p1 = {1.0, 2.0};
    point_t p2 = {4.0, 6.0};
    double d = distance(p1, p2);
    printf("%.1lf\n", d);
    return EXIT_SUCCESS;
}
```

Remember, Function Arguments in C are Copy-by-value

Sometimes structures can be very large, and copying them is slow
(also just like variables, we cannot modify the values in the caller)

Normally programmers always use structure arguments as pointers

```
double distance(point_t *p1 , point_t *p2);
```

There’s an Operator to Access Fields Through a Pointer

Assuming we have: `point_t *p1`

We can access the `x` field using: `(*p1).x`

- We have to dereference the pointer to get a `point_t`, then we can access `x`

The `->` operator combines dereferencing and the field access, we can use:

- `p1->x`

Let's Change Our Functions to Use Pointers

```c
double distance(point_t *p1 , point_t *p2) {
    return sqrt(pow(p2->x - p1->x, 2.0) + pow(p2->y - p1->y, 2.0));
}

void print(point_t *p) {
    printf("point(%lf, %lf)\n", p->x, p->y);
}

int main(void) {
    point_t p1 = {1.0, 2.0};
    point_t p2 = {4.0, 6.0};
    double d = distance(&p1, &p2);
    printf("%.1lf\n", d);
    return EXIT_SUCCESS;
}
```

You Cannot Access a Field Using `.` with a Pointer

You'll get a compiler error like the following:

```
error: member reference type 'point_t *' is a pointer; did you mean to use '->'?
```

Luckily, this is easy to fix, and may be a common message you see

We Can Create a Structure Dynamically

Usually, we create our own dedicated function for this:

```
point_t *point_create(double x, double y) {
    point_t *point = malloc(sizeof(point_t));
    point->x = x;
    point->y = y;
    return point;
}
```

This function lets us create a point dynamically, and initialize it in one step:

```
  point_t *p1 = point_create(1.0, 2.0);
```

As always, we need to remember to `free` p1 after we're done with it

Usually, We Always Prefix All Functions that Use a Structure

In the case of our point, all our functions should start with: `point_`

We should define our other functions like:

```
double point_distance(point_t *p1 , point_t *p2) {
    return sqrt(pow(p2->x - p1->x, 2.0) + pow(p2->y - p1->y, 2.0));
}

void point_print(point_t *point) {
    printf("point(%.1lf, %.1lf)\n", point->x, point->y);
}
```

The following is more advanced usage of C programming
you may not be able to use them for this course

Usually, You Divide Code Between Different Files

So, I may create a file called `point.c` that contains the defintion of:

- `point_create`, `point_distance`, and `point_print`

`point.c` could also be the only file that defines the `struct` itself (more later)

We then create a file called `point.h` that contains the function prototypes

- That way we can use these functions in other `.c` files

Recall, we call a `.h` file a header file

An Example Header File for Our Point

```c
#ifndef POINT_H
#define POINT_H

typedef struct point point_t;

point_t *point_create(double x, double y);
double point_distance(point_t *p1 , point_t *p2);
void point_print(point_t *point);

#endif
```

This Header File is Included Once and Hides the `struct`

The lines that start with a `#` are preprocessor commands
- They make sure that we only read this file once when we compile a file

We can also do a forward declaration of the `struct`
- We tell the compiler we’re going to define this later

We can create pointers to the `struct` without knowing its fields

Now We Can Just Use the Header File in Our Program

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

#include "point.h"

int main(void) {
    point_t *p1 = point_create(1.0, 2.0);
    point_print(p1);
    point_t *p2 = point_create(4.0, 6.0);
    point_print(p2);
    printf("distance(p1, p2) = %.1lf\n", point_distance(p1, p2));
    free(p1);
    free(p2);
    return EXIT_SUCCESS;
}
```

Our Program Cannot Access Fields Itself

It doesn't know the definition of the `struct`
- It's more flexible if we can hide the details

We should create more functions to access and set the fields
- Typically, you create a getter to get the current value,
  - and a setter to modify the value

Our Complete Header File

```c
#ifndef POINT_H
#define POINT_H

typedef struct point point_t;

point_t *point_create(double x, double y);
double point_distance(point_t *p1 , point_t *p2);
double point_getX(point_t *p);
void point_setX(point_t *p, double x);
double point_getY(point_t *p);
void point_setY(point_t *p, double y);
void point_print(point_t *point);

#endif
```

Our Complete Header File

It doesn't know the definition of the `struct`
- It's more flexible if we can hide the details

The Start of Our `point.c` Code

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

#include "point.h"

typedef struct point {
    double x;
    double y;
} point_t;

point_t *point_create(double x, double y) {
    point_t *point = malloc(sizeof(point_t));
    point->x = x;
    point->y = y;
    return point;
}
```

The Rest of Our `point.c` Code

```c
double point_distance(point_t *p1 , point_t *p2) {
    return sqrt(pow(p2->x - p1->x, 2.0) + pow(p2->y - p1->y, 2.0));
}

double point_getX(point_t *p) { return p->x; }
void point_setX(point_t *p, double x) { p->x = x; }

double point_getY(point_t *p) { return p->y; }
void point_setY(point_t *p, double y) { p->y = y; }

void point_print(point_t *point) {
    printf("point(%.1lf, %.1lf)\n", point->x, point->y);
}
```

We When Compile, We Need to Use Both Files

We could put our `main` function in a file named `main.c`

We need to compile both files at the same time with:

```
gcc point.c main.c -o main -lm
```

Afterwards, we can run `main` as before