C Programming Lesson: Introduction to struct

Marcus Birkenkrahe

2025-04-30

Lesson Overview

This 50-minute session introduces the use of `struct` in C for organizing and manipulating related data.

Learning Objectives

By the end of the session, you will:

- Declare and initialize a `struct`
- Use dot (`.`) and arrow (`->`) notation
- Pass structs to functions (by value and by pointer)
- Write a simple data model with a `struct` and function interaction

Why would you group Data?

- What kind of data naturally belong together? For example, when playing a **game**, when drawing a **graph**, or when looking for a **date**?
  - **Game:** A player's `name`, `health`, and `level` in the game.
  - **Graph**: A `point` with `(x,y)` coordinates.
  - **Date**: A `day`, `month`, and `year`.

Basic `struct Point`

- We define a `struct`, a `Point`, defined by pair of coordinates `(x,y)`:

```
struct Point {
  int x;
  int y;
};
```

- To declare a `Point`, create an **instance** of the `struct`:

```
struct Point pt; // declare pt to be a Point
```

- To access the `x` and `y` inside of a `Point`, use the `.` (dot) operator:

```
pt.x = 3;
pt.y = 4;
```

- Or you can initialize the point when declaring it:

```
struct Point pt = {3,4}; // define pt to be a Point at (3,4)
```

- Code along:

```
#include <stdio.h> // include I/O


/* main program */
int main() {

  return 0;
}
```

- Solution:

```
#include <stdio.h> // include I/O

// Declare a Point (x,y)
struct Point {
  int x; // x coordinate
  int y; // y coordinate
```

```
};

/* main program */
int main() {
  // define a Point pt with coordinates (3,4)
  struct Point pt = {3, 4};
  // print the point coordinates
  printf("Point is at (%d, %d)\n", pt.x, pt.y);

  return 0;
}
```

- Why use a `struct` instead of two separate variables?

  Improves code organization, readability and maintainability - like grouping two parts of a picture so that you can move it around more easily.

A `struct` of 1000 points

- What if we wanted to store a thousand points?

  We would use an array of `struct Point`. This scales much better than managing two separate arrays for `x` and `y`.

- Let's initialize 1000 points & print the last point. Code along: (onecompiler.com/c/43ggz83f6)

```
#include <stdio.h>

// declare a Point in the (x,y) plane

/* main program */
int main (void)
{
  // declare an array of 1000 points

  // initialize all points with random values

  // print the last point
```

```
  return 0;
}
```

- Solution:

```
#include <stdio.h>

// declare a Point in the (x,y) plane
struct Point {
  int x;
  int y;
};

/* main program */
int main (void)
{
  // declare an array of 1000 points
  struct Point points[1000];

  // initialize all points with random values
  for (int i = 0; i < 1000; i++) {
    points[i].x = i;
    points[i].y = i * i;
  }

  // print the last point
  printf("The last point is (%d,%d)\n",
         points[999].x, points[999].y);
  printf("999^2 = %d\n", 999 * 999);

  return 0;
}
```

Memory layout

- What does `struct Point p1 = {3,4};` do in memory?

  It allocates space for two integers (4 bytes each) in contiguous memory and assigns `x=3` and `y=4`.

```
struct Point
+--------+--------+
|   x=3  |   y=4  |
+--------+--------+
```

- In systems programming (e.g. for Internet connections between computers), or embedded C (on a microcontroller), a `struct` is mapped directly onto addresses.
- Knowing memory layout aids portability of your code. Portability is the key to any computer doing what you want solely based on the program and the data that you feed to it.

Pointers and structs

- Passing a `struct` to a function creates a copy - this can be expensive for large data sets.
- Better: Use pointers! To access a pointer structure element use the arrow operator `->` : `ptr->x` is the same as `(*ptr).x`

```
// Declare a Point (x,y)
struct Point {
  int x;
  int y;
};

struct Point p = {10,20}; // Point p at (10,20)
struct Point *ptr = &p;   // ptr points at point p
// PRINT "x = , y = " + pointer to x, pointer to y
printf(" x = %d, y = %d\n",
       ptr->x, ptr->y);
printf(" x = %d, y = %d\n",
       (*ptr).x, (*ptr).y);
```

- Example: code along at onecompiler.com/c/43gh38ut7

```
// Input/Output

// declare 2D (float) Point (x,y)
```

```
// declare function to move a Point by (dx,dy)

/* main program */
// BEGIN

// define Point at (1,2)

// PRINT "Original: (x,y)"

// Move point by (0.5,-1.5)

// PRINT "Moved: (x,y)"

// END

// move_point: move Point by (dx,dy)

// BEGIN
// pointer to x + dx
// pointer to y + dy
// END
```

- Solution:

```
#include <stdio.h>

// declare 2D (float) Point (x,y)
struct Point {
  float x;
  float y;
};
// declare function to move a Point by (dx,dy)
void move_point(struct Point *, float, float);

/* main program */
int main(void)
{ // BEGIN

  // define Point at (1,2)
```

```
  struct Point p = {1.0,2.0};
  // PRINT "Original: (x,y)"
  printf("Original: (%.2f,%.2f)\n", p.x, p.y);
  // Move point by (0.5,-1.5)
  move_point(&p, 0.5, -1.5);
  // PRINT "Moved: (x,y)"
  printf("Moved:    (%.2f,%.2f)\n", p.x, p.y);

  return 0;
} // END

// move_point: move Point by (dx,dy)
void move_point(struct Point *p, float dx, float dy)
{ // BEGIN
  p->x += dx; // pointer to x + dx
  p->y += dy; // pointer to y + dy
} // END
```