【C】Day1

Course	Advanced C
Study Date	@March 15, 2022

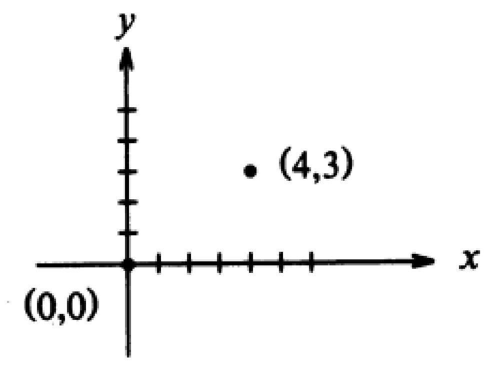
【Ch6】Structures

A structure is a collection of one ore more variables, possibly of many types, grouped together under a single name for convenient handling.

The main change made by the ANSI standard is to define structure assignment-structures may be copied and assigned to, passed to functions, and returned by functions.

6.1 Basics of Structures

Let us create a few structures for graphics. The basic object is a point, which we will assume has an x coordinate an a y coordinate, both integers.



The two components can be placed in a structure declared like this:

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

The keyword `struct` introduces a structure declaration, which is a list of declarations enclosed in brackets.

An optional name called a structure tag may follow the word struct (as with `point` here). The tag names this kind of structure, and can be used subsequently as a shorthand for the part of the declaration in braces.

The variables named in a structure are called members. A structure member or tag and an ordinary (i.e., non-member) variable can have the same name without conflict since they can always be distinguished by context.

Furthermore, the same member names may occur in different structures, although as a matter of style one would normally use the same names only for closely related objects.

A `struct` declaration defines a type. The right brace that terminates the list of members may be followed by a list of variables, just as for any basic type. That is,

```
struct { ...} x, y, z;
```

is syntactically analogous to

```
int x, y, z;
```

in the sense that each statement declares x, y, and z to be variables of the named type and causes space to be set aside for them.

A structure declaration that is not followed by a list of variables reserves no storage; it merely describes a template or the shape of a structure.

If the declaration is tagged, the tag can be used later in definitions of instances of the structure. For example, given the declaration of point above,

```
struct point pt;
```

defines a variable `pt` which is a structure of type `struct point`.

A structure can be initialized by following its definition with a list of initializers, each a constant expression, for the members:

```
struct point maxpt = { 320, 200 };
```

A member of a particular structure is referred to in an expression by a construction of the form

```
structure-name.member
```

The structure member operator `.` connects the structure name and the member name. To print the coordinates of the point pt, for instance,

```
printf("%d, %d", pt.x, pt.y);
```

Structures can be nested. One representation of a rectangle is a pair of points that denote the diagonally opposite corners:

```
struct rect {
  struct point p1;
  struct point pt2;
};
```

If we declare screen as `struct rect screen`, then `screen.pt1.x` refers to the x coordinate of the pt1 member of screen.

6.2 Structures and Functions

The only legal operations on a structure are copying it or assigning to it as a unit, taking its address with &, and accessing its members.

Copying and assignment include passing arguments to functions and returning values from functions as well.

Let us investigate by writing some functions to manipulate points and rectangles. There are at least three possible approaches: pass components separately, pass an entire structure, or pass a pointer to it. Each has its good points and bad points.

The first function, `makepoint`, will take two integers and return a point structure:

```
/* makepoint: make a point from x and y components. */
struct point makepoint(int x, int y) {
  struct point temp;
  temp.x = x;
  temp.y = y;
  return temp;
}
```

`makepoint` can now be used to initialize any structure dynamically, or to provide structre arguments to a function

```
struct rect screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1 = makepiont(0, 0);
screem.pt2 = makepoint(XMAX, YMAX);
middle = makepoint((screen.pt1.x + screen.pt2.x) / 2, (screen.pt1.y + screen.pt2.y) / 2);
```

The next step is a set of functions to do arithmetic on points. For instance,

```
// addpoint: add two points
struct point addpoint(struct point p1, struct point p2) {
  p1.x += p2.x;
  p1.y +=p 2.y;
  return p1;
}
```

Here we incremented the components in `p1` rather than using an explicit temporary variable to emphasize that structure parameters are passed by value like any others.

As another example, the function `ptinrect` tests whether a point is inside a rectangle, where we have adopted the convention that a rectangle includes its left and bottom sides but not its top and right sides:

```
// ptinrect: return 1 if p in r, 0 if not
int ptinrect(struct point p, struct rect r) {
  return p.x >= r.pt1.x && p.x < r.pt2.x && p.y >= r.pt1.y && p.y <= r.pt2.y;
}
```

This function assumes that the rectangle is represented in a standard form where the `pt1` coordinates are less than the `pt2` coordinates.

The following function returns a rectangle guaranteed to be in canonical form:

```
#define max(a, b) ((a) > (b) ? (a) : (b))
#define min(a, b) ((a) < (b) ? (a) : (b))

// canonrect: canonicalize coordinates of rectangle
struct rect canonrect(struct rect r) {
  struct rect temp;

  temp.pt1.x = min(r.pt1.x, r.pt2.x);
  temp.pt1.y = min(r.pt1.y, r.pt2.y);
  temp.pt2.x = max(r.pt1.x, r.pt2.x);
  temp.pt2.y = max(r.pt1,y, r.pt2.y);

  return temp;
}
```

If a large structure is to be passed to a function, it is generally more efficient to pass a pointer than to copy the whole structure. Structure pointers are just like pointers to ordinary variables. The declaration

```
struct point *pp;
```

says that `pp` is a pointer to a structure of type `struct point`. To use pp, we might write:

```
struct point origin, *pp;
pp = &origin;
printf("origin is (%d, %d)\n", (*pp).x, (*pp).y);
```

The parentheses are necessary in `(*pp).x` because the precedence of the structure member operator `.` is higher than `*`.

The expression `*pp.x` means `*(pp.x)`, which is illegal here because `x` is not a pointer.

We can also use the following notation:

```
p -> member-of-structure
```

to refer to a particular member.

The structure operators `.` and `->`, together with `()` for function calls and `[]` for subscripts, are at the top of the precedence hierarchy and thus bind very tightly.