Struct

Nov 25, 2019
struct

- **struct**: A collection of variables, possibly of different types, grouped under a single name for common reference as a unit.

- **Example**: To represent a point in a two-dimensional graph, we can declare a point structure:

  ```
  struct point { //point is the struct name, optional
      int x; // member x
      int y; // member y
  }
  ```
A struct declaration defines a type

The closing brace may be followed by a list of variables, just as for any basic type:

```c
struct point {
    int x;
    int y;
} pt1, pt2, pt3;
```

This syntax is similar to

```c
int i, j, k;
```

Both define variables of the named type and set space aside for them

We can define variables using our struct:

```c
struct point p4, p5, p6;
```
And we can initialize the members when we define it:

```c
struct point p7 = {5, 10};
```

We can assign values to members after definition using the structure member operator, or dot operator

```c
p7.x = 5;
p7.y = 10;
```
We can also nest structs

For example, we can use our point structure to define a rectangle structure:

```c
struct rect {
    struct point pt1; // lower left corner
    struct point pt2; // upper right corner
};
```

This function calculates the area of a rectangle:

```c
int rectArea(struct rect r) {
    return (r.pt2.x - r.pt1.x) * (r.pt2.y - r.pt1.y);
}
```
Two Ways to Define Struct Variables

- Declare a point structure, then define two point variables

```c
struct point {
    int x; // member
    int y; // member
}; //note the semicolon here
struct point pt1, pt2;
```

- Define a point structure and two point variables in the same statement

```c
struct point {
    int x; // member
    int y; // member
} pt1, pt2;
```
What We Can Do with a Struct

- Reference members
  \[
  \text{box.pt2.x} = \text{box.pt1.x} + \text{width};
  \]

- Assign as a unit
  \[
  \text{pt2} = \text{pt1};
  \]

- Take its address
  \[
  \text{struct point *ppt1;}
  \]
  \[
  \text{ppt1} = &\text{pt1};
  \]

- Note: We CANNOT compare structures directly:
  \[
  \text{if (pt1 == pt2)} \ldots \text{WRONG!}
  \]

- We have to compare members one by one:
  \[
  \text{if (pt1.x == pt2.x && pt1.y == pt2.y)} \ldots
  \]
Struct Pointers

- If pp is a pointer to pt1
- *pp is the struct
- (*pp).x and (*pp).y are the members
- The parentheses are necessary because . (dot) has higher precedence than * (dereference)
Pointers to structures are used often.

There is an alternative notation to access a member via a pointer to a structure:

```c
pp->x  //this is preferred
(*pp).x //same as pp->x
```

With this notation we are dereferencing the pointer and accessing a member of the struct it points to.
sizeof is a compile-time unary operator that produces the number of bytes used to store its operand.

Note that sizeof produces a number of type size_t, which is an unsigned integer type.

The t means type – it is a typedef created for ints describing size.

sizeof can be called on a variable or a type:

- sizeof variable_name
- sizeof (type_name)

Parentheses with sizeof:
- variable name: optional
- type name: required
A Function Can Return a Struct

```c
struct point makePoint(int x, int y) {
    struct point *temp;

    temp = (struct point *) malloc(sizeof(struct point));
    temp->x = x; //there is no name conflict here
    temp->y = y;
    return *temp;
}
```

//avoid this style in K&R
```c
struct point makePoint(int x, int y) {
    struct point temp;

    temp.x = x;
    temp.y = y;
    return temp;
}
```
We Can Pass a Struct to a Function

- Three ways to pass a struct to a function:
  - Pass members separately
  - Pass the entire struct
  - Pass a pointer to the struct
Example: Test if a Point is Inside a Rectangle

- If point p is inside rect r, return 1, else return 0

  // Pass members: ll lower left, ur upper right
  int ptInRect (int point_x, int point_y,
               int llx, int lly, int urx, int ury) {
    return x >= llx && x <= urx && y >= lly && y <= ury;
  }

  // Pass struct
  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;
  }

  // Pass struct pointer
  int ptInRect (struct point *pp, struct rect *rp) {
    return pp->x >= rp->pt1.x &&
           pp->x <= rp->pt2.x &&
           pp->y >= rp->pt1.y &&
           pp->y <= rp->pt2.y;
  }
Note on Precedence

- Review the precedence chart
- . and \( \rightarrow \) are at the top of the precedence chart
- If we have \( ++pp->x \), it reads as \( ++(pp->x) \)
- So member \( x \) is pre-incremented
struct string {
    int len;
    char *cp;
} *p;

struct string xp;

p = &xp;
struct string a = {8, "yellow"};
struct string b = {9, "black"};
struct string *ptr_a = &a;
p = &b;
## Example: Precedence

<table>
<thead>
<tr>
<th>expression</th>
<th>same as</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>++p-&gt;len</td>
<td>++(p-&gt;len)</td>
<td>preincrements len evaluates to incremented value of len</td>
</tr>
<tr>
<td>*p-&gt;cp</td>
<td>*(p-&gt;cp)</td>
<td>value is a char</td>
</tr>
<tr>
<td>*p-&gt;cp++</td>
<td>*((p-&gt;cp)++)</td>
<td>value is a char, post-increments cp</td>
</tr>
<tr>
<td>ptr_a-&gt;len</td>
<td>(*a).len</td>
<td>8</td>
</tr>
<tr>
<td>ptr_a-&gt;cp</td>
<td>(*a).cp</td>
<td>'y'</td>
</tr>
<tr>
<td>ptr_b-&gt;len</td>
<td>(*b).len</td>
<td>9</td>
</tr>
<tr>
<td>ptr_b-&gt;cp</td>
<td>(*b).cp</td>
<td>'b'</td>
</tr>
</tbody>
</table>
What if we want to write a program to count the number of times each C reserved word is seen in some code?

/* number of reserved keywords */
#define NKEYS 32

Parallel arrays (what we’d do until now):

/* array of keyword strings */
char *keyword[NKEYS];
/* corresponding array of counts */
int keyCount[NKEYS] // corresponding counts

Array of structs: (put the word and count in one data structure)

struct key {
    char *word; // pointer to a string literal
    int count; // corresponding count
} keyTable[NKEYS]; // array of struct key elements
struct key {
    char *word; // pointer to a string literal
    int count; // corresponding count
} keyTable[] = { "auto", 0, "break", 0,
    ...
    "volatile", 0, "while", 0
};
struct key {
    char *word; // pointer to a string literal
    int count; // corresponding count
} keyTable[] = {
    {"auto", 0},
    {"break", 0},
    ...
    {"volatile", 0},
    {"while", 0}
};

// we can use sizeof to figure out NKEYS
#define NKEYS (sizeof(keytab) / sizeof(struct key))