CSC / CPE 357

Systems Programming

Chapter 6 in The C
Programming Language

Structures

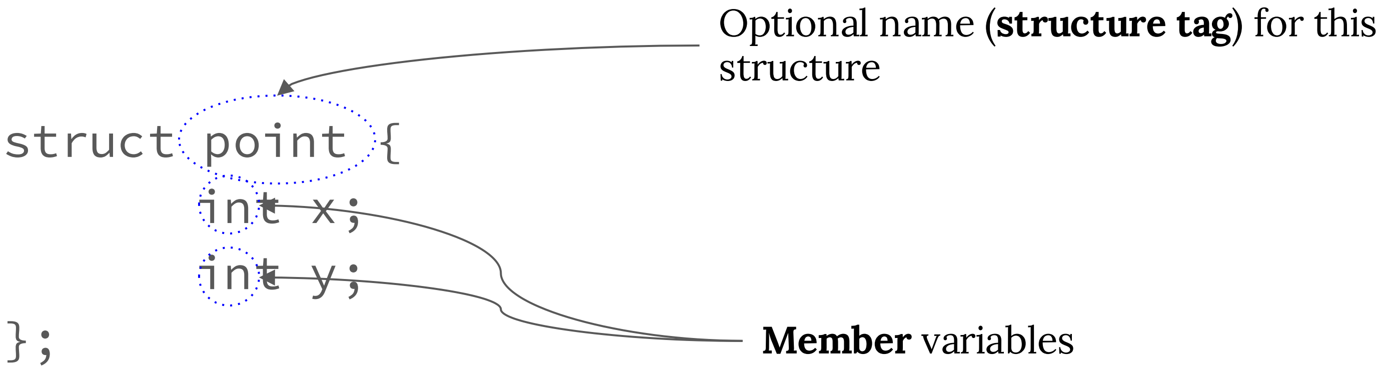
- A **structure** is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.
  - Called "records" in some other languages
  - Not equivalent to classes (no behavior/methods)

- Structures help to organize complex data

- Group of related variables treated as a unit instead of as separate entities

Example Structure

Declare a structure named point with x and y coordinates



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

Optional name (**structure tag**) for this structure

Member variables

Struct Syntax

```
struct point pt;  // define a variable pt of type point

struct point maxpt = { 200, 100 };  // define and initialize



// use the structure member operator (.) to access members

printf("%d, %d\n", maxpt.x, maxpt.y);
```

Nested Structures

C allows nested structures:

```
struct rect {
       struct point pt1;
       struct point pt2;
};
```

Structures and Functions

- Valid operations on structures:
  - Copy or assign as a unit
  - Find address of structure with &
  - Access individual members

Structure sizeof()

The `sizeof()` operator can be applied to structures. May return surprising results, due to machine alignment requirements.

```
struct s {
    char c;
    int i;
};


sizeof(struct s);  // sizeof(char) + sizeof(int)  ??
```

Pointers to Structures

A pointer to a structure can be defined as:

```
struct point *pp;  // pp is a pointer to a point structure
```

Dereference using * operator to access member variables:

```
(*pp).x;
```

Note that the parentheses are required here due to operator precedence.

`*pp.x` means `*(pp.x)` -- an invalid operation (x is an int, not a pointer)

Pointers to Structures

Pointers to structures are commonly used, dereferencing is cumbersome:

```
(*pp).x
```

C supports shorthand notation for combination of dereference and member access:

```
pp->member-of-structure
```

`pp->x` is equivalent to `(*pp).x`

Pointers to Structures (Operator Precendece)

Structure operators (`.` and `->`) have high precedence.

```
++pp->x;  // increments x not pp, equivalent to: ++(pp->x)
(++pp)->x;  // increments pointer before accessing member
```

Arrays of Structures

An array of `point` structures may be defined as:

```
    struct point points[10];
```

Each element in the array is of type `struct point`. Storage required is:

```
    <array length> * sizeof(struct point)
```

Access member x of an array element 4:

```
    points[4].x;
```

Self-Referential Structures

To implement common data structures such as linked lists or trees, it is often useful to define self-referential structures:

```
struct list_element {
    int val;
    struct list_element *next;
}
```

We will revisit this during discussions of dynamic memory allocation.

Typedef

To create new data type names, C provides the `typedef` keyword:

```
typedef char *String;  // String is now a synonym for char *
String s;   // equivalent to char *s;



typedef struct list_element ListEl;
ListEl l;
```

Note that a `typedef` declaration does not create a new type; it creates a synonym only. Two common uses:

1. Handle machine-dependent differences (for example `size_t` may be an `unsigned int` on some machines, `unsigned long` on others)
2. Program clarity / understandability

typedef versus #define

typedef obeys scoping rules just like variables, whereas #define is valid from the point where it appears in a file

Other differences:

```
typedef int *int_p1;
int_p1 a, b, c;  // a, b, c are all int pointers
```

```
#define int_p2 int*
int_p2 a, b, c;  // only a is a pointer
```

```
typedef char c10[10];
c10 x, y, z;  // create three 10-char arrays
```

Unions

- A **union** is a variable that may hold (at different times) objects of different types and sizes, with the compiler keeping track of size and alignment requirements.

- A way to manipulate different kinds of data in a single area of storage, without embedding any machine-dependent information in the program.

Union Example

```
union int_float_or_string {
   int ival;
   float fval;
   char *sval;
} u;
```

- The variable `u` will be large enough to hold the largest of the three types
- Any of these types (`int`, `char *`, or `float`) may be assigned to `u`
- Type retrieved must be the type most recently stored
  - Programmer's responsibility to keep track
- Same operations permitted on unions as on structures
  - Copying as a unit, address of (&), member access with `.` or `->`