Computer Organization and Architecture

C Structures and Unions

Utah Tech University

Spring 2025

Structures

Data structures are made up of

- *primitive types* like `int`, `double`, `char *`, etc., and
- *composite types* that combine them.

The primary composite types include arrays (lists of elements of the same type) and *structures*, also called *records*, which collect members of different types into a single object. Other more complex data structures are built up from these pieces.

```
struct Date {
    short month;
    short day;
    short year;
};
```

A `struct` must have at least one member. The *tag* is optional.

Defining structure types

The members of a `struct` may have any complete type, including previously-defined structure types. They must not be variable-length arrays or pointers to such arrays.

```
struct Song {
    char title[64];
    char artist[32];
    char composer[32];
    short duration;
    struct Date published;
};
```

Each member is embedded in the `struct`, and an instance of a `struct` includes its members laid out in memory in the order they are declared in the type.

A `struct` type cannot contain itself as a member (what would the `sizeof` such a type be?), however it can contain a pointer to its own type:

```
struct Cell {
    struct Song song;
    struct Cell *next;
};
```

Structures and typedefs

You can declare objects of `struct` types:

```
struct Song song1, song2, *pSong = &song1;
```

The type name is `struct Song`, not `Song`. You can use `typedef` to define a one-word name for a `struct` type:

```
typedef struct Song Song_t;
Song_t song1, song2, *pSong = &song1;
```

You can also define a `struct` without a tag. This is most commonly combined with `typedef`:

```
typedef struct {
    struct Cell *pFirst;
    struct Cell *pLast;
} SongList_t;
```

`struct` definitions are often placed in a header file so the type can be used across multiple source files. The definition is also commonly combined with the prototypes of functions that act on the `struct`.

Accessing structure members

The *dot operator* (`.`) lets you access members of a `struct` when the left operand is a structure object:

```
Song_t song1, song2, *pSong = &song1;

// passing an array to a function yields a pointer
strcpy(song1.title, "Flaming Wreck");
strcpy(song1.composer, "Joe Pernice");
song1.duration = 335;

// song1.published is a struct
song1.published.year = 2001;

// pSong is not a song object, but *pSong is
if ((*pSong).duration > 180)
    printf("The song %s is more than 3 minutes long\n", (*pSong).title);
```

Accessing structure members

If you have a pointer to a structure you can use the *arrow operator* (`->`) to access the structure's members instead of combining the indirection and dot operators (`*` and `.`), i.e., `p->m` is equivalent to `(*p).m`:

```
if (pSong->duration > 180)
    printf("The song %s is more than 3 minutes long\n", pSong->title);
pSong->published.year = 2001;
```

You can use assignment to copy the entire contents of a structure object to another object of the same type:

```
song2 = song1;
```

This copies the entire region of memory including

- The contents of embedded `struct` members
- The contents of arrays
- Pointer values, but **not** the values they point to

Similarly, passing a `struct` by value to a function (or returning one) copies the entire object. This is fine when objects are small and memory sharing is not desired, but it is common to share larger objects (or any objects when sharing is called for) using pointers to the objects.

Unions

A *union* is declared and defined like a structure, but instead of laying out members in memory one after another, the members of a union occupy the same storage, letting you select one of several alternative values for a single object.

```
union tree {
    struct leaf leaf_node;
    struct branch branch_node;
};

union tree t;
t.branch_node.key = 536;
```

The size of a `union` is the size of its largest member. A `union` must be initialized and used consistently through only one of its members (say, `branch_node`). It may be re-initialized as a different member (say `leaf_node`) and then used consistently through that member.

Discriminated unions

Unions are usually combined with structures to implement *discriminated unions*, a kind of *sum type*, where a single value may take on one of several shapes.

```
enum message_type { CREATE, READ, UPDATE, DELETE };
struct message {
    enum message_type type;
    union {
        struct create_message create;
        struct read_message read;
        struct update_message update;
        struct delete_message delete;
    } body;
};

struct message msg;
/* ... */
switch (msg.type) {
case CREATE:
    printf("create message with id %d\n", msg.body.create.id);
    break;
// ...
}
```