Chapter 13: Structs

Structures provide a way to group a set of related variables of diverse types into a single unit of memory. The structure as a whole can be referenced by a single name or pointer; the structure members can be accessed individually too. Structures can be passed to functions and returned from functions. They are defined using the keyword `struct`.

Section 13.1: Flexible Array Members

Version ≥ C99

Type Declaration

A structure *with at least one member* may additionally contain a single array member of unspecified length at the end of the structure. This is called a flexible array member:

```
struct ex1
{
    size_t foo;
    int flex[];
};

struct ex2_header
{
    int foo;
    char bar;
};

struct ex2
{
    struct ex2_header hdr;
    int flex[];
};

/* Merged ex2_header and ex2 structures. */
struct ex3
{
    int foo;
    char bar;
    int flex[];
};
```

Effects on Size and Padding

A flexible array member is treated as having no size when calculating the size of a structure, though padding between that member and the previous member of the structure may still exist:

```
/* Prints "8,8" on my machine, so there is no padding. */
printf("%zu,%zu\n", sizeof(size_t), sizeof(struct ex1));

/* Also prints "8,8" on my machine, so there is no padding in the ex2 structure itself. */
printf("%zu,%zu\n", sizeof(struct ex2_header), sizeof(struct ex2));

/* Prints "5,8" on my machine, so there are 3 bytes of padding. */
printf("%zu,%zu\n", sizeof(int) + sizeof(char), sizeof(struct ex3));
```

The flexible array member is considered to have an incomplete array type, so its size cannot be calculated using `sizeof`.

Usage

You can declare and initialize an object with a structure type containing a flexible array member, but you must not attempt to initialize the flexible array member since it is treated as if it does not exist. It is forbidden to try to do this, and compile errors will result.

Similarly, you should not attempt to assign a value to any element of a flexible array member when declaring a structure in this way since there may not be enough padding at the end of the structure to allow for any objects required by the flexible array member. The compiler will not necessarily prevent you from doing this, however, so this can lead to undefined behavior.

```
/* invalid: cannot initialize flexible array member */
struct ex1 e1 = {1, {2, 3}};
/* invalid: hdr={foo=1, bar=2} OK, but cannot initialize flexible array member */
struct ex2 e2 = {{1, 2}, {3}};
/* valid: initialize foo=1, bar=2 members */
struct ex3 e3 = {1, 2};

e1.flex[0] = 3; /* undefined behavior, in my case */
e3.flex[0] = 2; /* undefined behavior again */
e2.flex[0] = e3.flex[0]; /* undefined behavior */
```

You may instead choose to use `malloc`, `calloc`, or `realloc` to allocate the structure with extra storage and later free it, which allows you to use the flexible array member as you wish:

```
/* valid: allocate an object of structure type `ex1` along with an array of 2 ints */
struct ex1 *pe1 = malloc(sizeof(*pe1) + 2 * sizeof(pe1->flex[0]));

/* valid: allocate an object of structure type ex2 along with an array of 4 ints */
struct ex2 *pe2 = malloc(sizeof(struct ex2) + sizeof(int[4]));

/* valid: allocate 5 structure type ex3 objects along with an array of 3 ints per object */
struct ex3 *pe3 = malloc(5 * (sizeof(*pe3) + sizeof(int[3])));

pe1->flex[0] = 3; /* valid */
pe3[0]->flex[0] = pe1->flex[0]; /* valid */
```

`Version < C99`

The 'struct hack'

Flexible array members did not exist prior to C99 and are treated as errors. A common workaround is to declare an array of length 1, a technique called the 'struct hack':

```
struct ex1
{
    size_t foo;
    int flex[1];
};
```

This will affect the size of the structure, however, unlike a true flexible array member:

```
/* Prints "8,4,16" on my machine, signifying that there are 4 bytes of padding. */
printf("%d,%d,%d\n", (int)sizeof(size_t), (int)sizeof(int[1]), (int)sizeof(struct ex1));
```

To use the `flex` member as a flexible array member, you'd allocate it with `malloc` as shown above, except that `sizeof(*pe1)` (or the equivalent `sizeof(struct ex1)`) would be replaced with `offsetof(struct ex1, flex)` or the longer, type-agnostic expression `sizeof(*pe1)-sizeof(pe1->flex)`. Alternatively, you might subtract 1 from the desired length of the "flexible" array since it's already included in the structure size, assuming the desired length is

greater than 0. The same logic may be applied to the other usage examples.

Compatibility

If compatibility with compilers that do not support flexible array members is desired, you may use a macro defined like `FLEXMEMB_SIZE` below:

```
#if __STDC_VERSION__ < 199901L
#define FLEXMEMB_SIZE 1
#else
#define FLEXMEMB_SIZE /* nothing */
#endif

struct ex1
{
    size_t foo;
    int flex[FLEXMEMB_SIZE];
};
```

When allocating objects, you should use the `offsetof(struct ex1, flex)` form to refer to the structure size (excluding the flexible array member) since it is the only expression that will remain consistent between compilers that support flexible array members and compilers that do not:

```
struct ex1 *pe10 = malloc(offsetof(struct ex1, flex) + n * sizeof(pe10->flex[0]));
```

The alternative is to use the preprocessor to conditionally subtract 1 from the specified length. Due to the increased potential for inconsistency and general human error in this form, I moved the logic into a separate function:

```
struct ex1 *ex1_alloc(size_t n)
{
    struct ex1 tmp;
#if __STDC_VERSION__ < 199901L
    if (n != 0)
        n--;
#endif
    return malloc(sizeof(tmp) + n * sizeof(tmp.flex[0]));
}
...

/* allocate an ex1 object with "flex" array of length 3 */
struct ex1 *pe1 = ex1_alloc(3);
```

Section 13.2: Typedef Structs

Combining `typedef` with `struct` can make code clearer. For example:

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

as opposed to:

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

could be declared as:

```
Point point;
```

instead of:

```
struct Point point;
```

Even better is to use the following

```
typedef struct Point Point;

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

to have advantage of both possible definitions of `point`. Such a declaration is most convenient if you learned C++ first, where you may omit the `struct` keyword if the name is not ambiguous.

`typedef` names for structs could be in conflict with other identifiers of other parts of the program. Some consider this a disadvantage, but for most people having a `struct` and another identifier the same is quite disturbing. Notorious is e.g POSIX' `stat`

```
int stat(const char *pathname, struct stat *buf);
```

where you see a function `stat` that has one argument that is `struct stat`.

`typedef`'d structs without a tag name always impose that the whole `struct` declaration is visible to code that uses it. The entire `struct` declaration must then be placed in a header file.

Consider:

```
#include "bar.h"

struct foo
{
    bar *aBar;
};
```

So with a `typedef`d `struct` that has no tag name, the `bar.h` file always has to include the whole definition of `bar`. If we use

```
typedef struct bar bar;
```

in `bar.h`, the details of the `bar` structure can be hidden.

See Typedef

Section 13.3: Pointers to structs

When you have a variable containing a `struct`, you can access its fields using the dot operator (`.`). However, if you have a pointer to a `struct`, this will not work. You have to use the arrow operator (`->`) to access its fields. Here's an example of a terribly simple (some might say "terrible and simple") implementation of a stack that uses pointers to `struct`s and demonstrates the arrow operator.

```c
#include <stdlib.h>
#include <stdio.h>

/* structs */
struct stack
{
    struct node *top;
    int size;
};

struct node
{
    int data;
    struct node *next;
};

/* function declarations */
int push(int, struct stack*);
int pop(struct stack*);
void destroy(struct stack*);

int main(void)
{
    int result = EXIT_SUCCESS;

    size_t i;

    /* allocate memory for a struct stack and record its pointer */
    struct stack *stack = malloc(sizeof *stack);
    if (NULL == stack)
    {
        perror("malloc() failed");
        return EXIT_FAILURE;
    }

    /* initialize stack */
    stack->top = NULL;
    stack->size = 0;

    /* push 10 ints */
    {
        int data = 0;
        for(i = 0; i < 10; i++)
        {
            printf("Pushing: %d\n", data);
            if (-1 == push(data, stack))
            {
                perror("push() failed");
                result = EXIT_FAILURE;
                break;
            }

            ++data;
        }
    }

    if (EXIT_SUCCESS == result)
    {
        /* pop 5 ints */
        for(i = 0; i < 5; i++)
        {
            printf("Popped: %i\n", pop(stack));
```

```
        }
    }

    /* destroy stack */
    destroy(stack);

    return result;
}

/* Push a value onto the stack. */
/* Returns 0 on success and -1 on failure. */
int push(int data, struct stack *stack)
{
    int result = 0;

    /* allocate memory for new node */
    struct node *new_node = malloc(sizeof *new_node);
    if (NULL == new_node)
    {
        result = -1;
    }
    else
    {
        new_node->data = data;
        new_node->next = stack->top;
        stack->top = new_node;
        stack->size++;
    }

    return result;
}

/* Pop a value off of the stack. */
/* Returns the value popped off the stack */
int pop(struct stack *stack)
{
    struct node *top = stack->top;
    int data = top->data;
    stack->top = top->next;
    stack->size--;
    free(top);
    return data;
}

/* destroy the stack */
void destroy(struct stack *stack)
{
    /* free all pointers */
    while(stack->top != NULL)
    {
        pop(stack);
    }
}
```

Section 13.4: Passing structs to functions

In C, all arguments are passed to functions by value, including structs. For small structs, this is a good thing as it means there is no overhead from accessing the data through a pointer. However, it also makes it very easy to accidentally pass a huge struct resulting in poor performance, particularly if the programmer is used to other languages where arguments are passed by reference.

```c
struct coordinates
{
    int x;
    int y;
    int z;
};

// Passing and returning a small struct by value, very fast
struct coordinates move(struct coordinates position, struct coordinates movement)
{
    position.x += movement.x;
    position.y += movement.y;
    position.z += movement.z;
    return position;
}

// A very big struct
struct lotsOfData
{
    int param1;
    char param2[80000];
};

// Passing and returning a large struct by value, very slow!
// Given the large size of the struct this could even cause stack overflow
struct lotsOfData doubleParam1(struct lotsOfData value)
{
    value.param1 *= 2;
    return value;
}

// Passing the large struct by pointer instead, fairly fast
void doubleParam1ByPtr(struct lotsOfData *value)
{
    value->param1 *= 2;
}
```

Section 13.5: Object-based programming using structs

Structs may be used to implement code in an object oriented manner. A struct is similar to a class, but is missing the functions which normally also form part of a class, we can add these as function pointer member variables. To stay with our coordinates example:

```c
/* coordinates.h */

typedef struct coordinate_s
{
    /* Pointers to method functions */
    void (*setx)(coordinate *this, int x);
    void (*sety)(coordinate *this, int y);
    void (*print)(coordinate *this);
    /* Data */
    int x;
    int y;
} coordinate;

/* Constructor */
coordinate *coordinate_create(void);
/* Destructor */
void coordinate_destroy(coordinate *this);
```

And now the implementing C file:

```c
/* coordinates.c */

#include "coordinates.h"
#include <stdio.h>
#include <stdlib.h>

/* Constructor */
coordinate *coordinate_create(void)
{
    coordinate *c = malloc(sizeof(*c));
    if (c != 0)
    {
        c->setx = &coordinate_setx;
        c->sety = &coordinate_sety;
        c->print = &coordinate_print;
        c->x = 0;
        c->y = 0;
    }
    return c;
}

/* Destructor */
void coordinate_destroy(coordinate *this)
{
    if (this != NULL)
    {
        free(this);
    }
}

/* Methods */
static void coordinate_setx(coordinate *this, int x)
{
    if (this != NULL)
    {
        this->x = x;
    }
}

static void coordinate_sety(coordinate *this, int y)
{
    if (this != NULL)
    {
        this->y = y;
    }
}

static void coordinate_print(coordinate *this)
{
    if (this != NULL)
    {
        printf("Coordinate: (%i, %i)\n", this->x, this->y);
    }
    else
    {
        printf("NULL pointer exception!\n");
    }
}
```

An example usage of our coordinate class would be:

```
/* main.c */

#include "coordinates.h"
#include <stddef.h>

int main(void)
{
    /* Create and initialize pointers to coordinate objects */
    coordinate *c1 = coordinate_create();
    coordinate *c2 = coordinate_create();

    /* Now we can use our objects using our methods and passing the object as parameter */
    c1->setx(c1, 1);
    c1->sety(c1, 2);

    c2->setx(c2, 3);
    c2->sety(c2, 4);

    c1->print(c1);
    c2->print(c2);

    /* After using our objects we destroy them using our "destructor" function */
    coordinate_destroy(c1);
    c1 = NULL;
    coordinate_destroy(c2);
    c2 = NULL;

    return 0;
}
```

Section 13.6: Simple data structures

Structure data types are useful way to package related data and have them behave like a single variable.

Declaring a simple `struct` that holds two `int` members:

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

`x` and `y` are called the *members* (or *fields*) of `point` struct.

Defining and using structs:

```
struct point p;    // declare p as a point struct
p.x = 5;           // assign p member variables
p.y = 3;
```

Structs can be initialized at definition. The above is equivalent to:

```
struct point p = {5, 3};
```

Structs may also be initialized using designated initializers.

Accessing fields is also done using the `.` operator

```
printf("point is (x = %d, y = %d)", p.x, p.y);
```