8 Structures

We should start with a warning: if you see yourself using complex C structures in your code you are probably using the wrong programming language. Most likely a language with *object oriented* features, such as Java or C++, would do the job in a much cleaner and efficient way.

C structures offer the programmer a way to create variables that aggregate data of different types under a single name. The facility is mostly a convinience and little support is provided in the way of ensuring integrity of the data. Unlike arrays, which also holds more than one piece of data of the same type (say, integers, floats, doubles, characters, strings), the components of a structure may be of different types.

8.1 Structure declaration and initialization

As an illustration, let's develop a database of the planets in our solar system. Each planet will be described by its name, diameter, number of moons, orbit time, and rotation time. For example, for Jupiter we have:

Name:	Jupiter
Diameter:	$142,800$ km
Moons:	4
Orbit time:	11.9 yr
Rotation time:	9.925 hr

and similarly for the other 8 planets (Mercury, Venus, Earth, Mars, Saturn, Uranus, Neptune, and Pluto).

We would like to define a data structure that can hold the information of each planet. This structure should contain five components (fields), one for each data item. We must specify the name of each component and its type. For example, the planet's name should be stored in a component that is an array of characters.

In C, we do that by creating a `struct`:

```
struct Planet_ {
  char name[20];
  double diameter;
  int moons;
```

```
    double orbit_time;
    double rotation_time;
  };
```

which is a set of variable declarations that we aggreagate under the name `Planet_`. The above statement however is simply a declaration of the new `struct` type `Planet_`. This mean that `Planet_` is not a regular variable, and no memory is allocated by the `struct` declaration. Once a `struct` has been declared you can now create regular variables which can actually hold data.

For example, the following statement:

```
  struct Planet_ mars, jupiter = { "Jupiter", 142800, 4, 11.9, 9.925 };
```

will now create and initialize a variable named `jupiter` that can hold the data specified in the declaration of the `struct Planet_`. The variable `mars` is also created but left uninitialized.

Note that the keyword `struct` must be present in the declaration of the variable `jupiter`. In order to streamline programming, it is common practice to use the keywork `typedef` associated with `structs`. Using `typedef` we can manipulate a `struct` or other complicated data types as if they were native C types[7]. The following statement

```
  typedef struct Planet_ Planet;
```

allows one to refer to `struct Planet_` simply as `Planet`. So the variable jupiter can now be defined using

```
  Planet jupiter = { "Jupiter", 142800, 4, 11.9, 9.925 };
```

where `Planet` is used as a native C type.

One can even use `typedef` at the time of declaration of a `struct`. For example:

```
  typedef struct Planet_ {
    char name[20];
    double diameter;
    int moons;
    double orbit_time;
    double rotation_time;
  } Planet;
```

simultaneously defines the `struct Planet_` and the new type `Planet`. This usage is so common that C allows one to ommit the name of the `struct` and go straight to the new type as in:

[7] See Section 7.7 for another example that uses `typedef` with a pointer to function.

```
typedef struct {
  char name[20];
  double diameter;
  int moons;
  double orbit_time;
  double rotation_time;
} Planet;
```

There are however disadvantages of `typedef`'s. First, you may end up with lots of names used for types that can potentially clash with variable names in your program. Second, one might need to look harder to find and understand code where a `typedef` is used instead of the original `struct`.

8.2 Accessing fields of a structure

We can refer to a component (field) of a structure by using the selection operator (`.`). For example, the following statements set the data for the planet `mars` which was previously declared but left uninitialized.

```
mars.diameter = 6792.4;
mars.orbit_time = 1.88;
mars.rotation_time = 24.65;
mars.moons = 2;
```

Using the operator '`.`' you can access any element of a structure in order you wish.

One can create pointers to a `struct` variable much in the same way as creating a pointer any other type of variable. The following statements the same thing:

```
Planet mars, *pptr;

mars.diameter = 6792.4;

pptr = &mars;
(*pptr).diameter = 6792.4;
```

Note that we were forced to use parenthesis around `*pptr` due to operator precedence rules. Indeed, `*pptr.name` is dereferencing the contents of the field `(*pptr).name`, which happens to be a pointer to `char`, while `(*pptr).name` is dereferencing the pointer `pptr` and accesing the field.

Because of this small nuisance, C provides the dedicated *arrow operator* (`->`), also know as direct component selection operator. Using this operator the statements below do the same thing:

```
Planet mars, *pptr = &mars;

mars.diameter = 6792.4;
(*pptr).diameter = 6792.4;
pptr -> diameter = 6792.4;
```

For example, one could print the diameter of a planet with the following statement:

```
Planet jupiter = { "Jupiter", 142800, 4, 11.9, 9.925 };
Planet *pl;

pl = &jupiter;
printf("%s's diameter is %.2f km.\n",
       pl -> name,
       pl -> diameter);
```

Note that in our example above we have been careful not to assign any value to the field `name`. Here is where C oddities with respect to array manipulation and memory management can quickly become problematic. Consider for example the following inocent piece of code:

```
1  typedef struct {
2    char name[21];
3    double diameter;
4    int moons;
5    double orbit_time;
6    double rotation_time;
7  } Planet;
8
9  void set_planet(Planet * pl, char * name,
10                 int moons, double diameter, double orbit_time,
11                 double rotation_time) {
12
13   pl -> name = name;
14   pl -> moons = moons;
15   pl -> diameter = diameter;
16   pl -> orbit_time = orbit_time;
17   pl -> rotation_time = rotation_time;
18
19 }
20
21 main() {
22
23   Planet mars;
24   set_planet( &mars, "Mars", 2, 6792.4, 1.88, 24.65, 2);
25
26 }
```

Luckily, the above code is invalid in most modern good compilers! The potential disaster here is the assignment in line 13 which could overwrite `mars.name` with a pointer to a temporary character string `"Mars"`. A good compiler will pick this up and throw an error but some compilers might just ignore it.

The fix here is not to throw away the original allocated space but to copy the name into that space. The following modified function `set_planet` would do a better job:

```
void set_planet(Planet * pl, char * name,
                int moons, double diameter, double orbit_time,
                double rotation_time) {

  strncpy(pl -> name, name, 20);
  pl -> name[20] = '\0';

  pl -> moons = moons;
  pl -> diameter = diameter;
  pl -> orbit_time = orbit_time;
  pl -> rotation_time = rotation_time;

}
```

Note the use of the function `strncpy` to copy at most 20 characters to the previously allocated 21 character array `pl -> name` and the explicit inclusion of a string terminating at the end of the assignment. Just as it happend before with `scanf`, `gets` and other functions we studied previously, the use of the simpler function `strcpy` is discouraged. This function will copy one character string into another until it finds a string terminating character. The functions `strncpy` and `strcpy` are part of `string.h`.

Note that an alternative here would be to declare the structure `Planet` to contain a simple `char` pointer as in:

```
typedef struct {
  char *name;
  double diameter;
  int moons;
  double orbit_time;
  double rotation_time;
} Planet;
```

The price to be paid though would be that the programmer would be fully responsible for performing the memory management.

8.3 Structures as function arguments and return parameters

Structures can be used as arguments and return parameters of functions. Here again, be extra cautious with the limitations of C! Everything will be copied, not referenced, as C enforces a *pass by-value* policy for both arguments and return parameters. However, recall that if pointers are involved, only the pointer is copied, not what it points to!

8.4 Nested structures

Be warned one more time: if you're getting close to use this, you may be better working with another language!

One can define nested structures in C. That is, structure whose components are themselves structures. For example, the following definition of a structure type includes a component that is an array of structures planets:

```
typedef struct {
  char name[21];
  double diameter;
  int number_of_planets;
  Planet planets[50];
} star;
```

Note that we have tacitly assumed that a star can have no more than 50 planets by chosing a `Planet` array of fixed size 50 :). As commented at the end of the previous section, an alternative would be to declare the field `planets` as a simple pointer to `Planet` and let the user be responsible to perform dynamic allocation and memory management as briefly discussed in Section 7.6.

8.5 Complex arithmetic using structures

A complex number x is a pair of real numbers a and b. If we let i denote the $\sqrt{-1}$, than

$$x = a + ib, \qquad\qquad i = \sqrt{-1}$$

The real number a is the real part of x; the real number b is the imaginary part of x.

One can then use a C structure to represent complex numbers. For instance

```c
typedef struct{
  double real;
  double imag;
} Complex;
```

represents a complex number using `double` as the base floating type. Using the above structure one can define a variable to hold the complex number $x = 1 - 2i$ in a C program as

```c
Complex x = {1, -2};
```

The above construction declares and initializes the variable `x` to have in its fields `real = 1` and `imag = -2`.

If $x = a + ib$ and $y = c + id$ are two complex numbers, then we have that complex number operations:

$$x + y = (a + c) + i(b + d),$$
$$xy = (ac - bd) + i(ad + bc).$$

We can define functions to compute such operations using structures. The functions:

```c
Complex
complex_sum(const Complex x, const Complex y) {
  /* z = x + y */
  Complex z = {x.real + y.real, x.imag + y.imag};
  return z;
}

Complex
complex_mult(const Complex x, const Complex y) {
  /* z = x * y */
  Complex z = { x.real * y.real - x.imag * y.imag,
                x.real * y.imag + x.imag * x.real };
  return z;
}
```

implement the above operations. Note that the functions *return* a `struct`, which is in this case performs fine as the `Complex` has only primitive types which are finely copied by the compiler. The following program shows an application of the above functions:

```c
/* Program w10-11.c */
#include <stdio.h>

typedef struct {
```

```
  double real;
  double imag;
} Complex;

Complex
complex_sum(const Complex x, const Complex y) {
  /* z = x + y */
  Complex z = {x.real + y.real, x.imag + y.imag};
  return z;
}

Complex
complex_mult(const Complex x, const Complex y) {
  /* z = x * y */
  Complex z = { x.real * y.real - x.imag * y.imag,
                x.real * y.imag + x.imag * x.real };
  return z;
}

void
complex_print(const Complex x) {
  if ( x.imag > 0 )
    printf("%g + %g i", x.real, x.imag);
  else
    printf("%g - %g i", x.real, -x.imag);
}

main() {

  Complex x, y, z1, z2, z3;
  printf("Add and multiply two complex numbers.\n");

  printf("Enter x: real and imaginary parts: ");
  scanf("%lf %lf", &x.real, &x.imag);
  printf("x = ");
  complex_print(x);
  printf("\n");

  printf("Enter y: real and imaginary parts: ");
  scanf("%lf %lf", &y.real, &y.imag);
  printf("y = ");
  complex_print(y);
  printf("\n");

  /* z1 = x + y */
  z1 = complex_sum(x, y);
  /* z2 = x * y  */
```

```
  z2 = complex_mult(x, y);
  /* z3 = x * (x + y) * y  */
  z3 = complex_mult( complex_mult(x, complex_sum(x,y)), y);

  printf("x + y      = ");
  complex_print(z1);
  printf("\n");

  printf("x * y      = ");
  complex_print(z2);
  printf("\n");

  printf("x*(x+y)*y = ");
  complex_print(z3);
  printf("\n");

}
```