C Programming/Advanced data types

In the chapter Variables we looked at the primitive data types. However *advanced* data types allow us greater flexibility in managing data in our program.

Structs

Structs are data types made of variables of other data types (possibly including other structs). They are used to group pieces of information into meaningful units, and also permit some constructs not possible otherwise. The variables declared in a struct are called "members". One defines a struct using the `struct` keyword. For example:

```
struct mystruct {
    int int_member;
    double double_member;
    char string_member[25];
} struct_var;
```

`struct_var` is a variable of type `struct mystruct`, which we declared along with the definition of the new `struct mystruct` data type. More commonly, struct variables are declared after the definition of the struct, using the form:

```
struct mystruct struct_var;
```

It is often common practice to make a *type synonym* so we don't have to type "struct mystruct" all the time. C allows us the possibility to do so using a `typedef` statement, which aliases a type:

```
typedef struct {
    // ...
} Mystruct;
```

The `struct` itself is an *incomplete* type (by the absence of a name on the first line), but it is aliased as `Mystruct`. Then the following may be used:

```
Mystruct struct_var;
```

The members of a struct variable may be accessed using the member access operator `.` (a dot) or the indirect member access operator `->` (an arrow) if the struct variable is a pointer:

```
struct_var.int_member = 0;
struct_var->int_number = 0; // this statement is equivalent to: (*struct_var).int_number = 0;
```

(Pointers will be explained in the next chapter.) Structs may contain not only their own variables but may also contain variables pointing to other structs. This allows a recursive definition, which is very powerful when used with pointers:

```
struct restaurant_order {
    char description[100];
    double price;
```

```
    struct restaurant_order *next_order;
};
```

This is an implementation of the linked list data structure. Each node (a restaurant order) is pointing to one other node. The linked list is terminated on the last node (in our example, this would be the last order) whose `next_order` variable would be assigned to `NULL`.

A recursive struct definition can be tricky when used with `typedef`. It is not possible to declare a struct variable inside its own type by using its aliased definition, since the aliased definition by `typedef` does not exist before the `typedef` statement is evaluated:

```
typedef struct Mystruct {
    // ...
    struct Mystruct *pointer; // Mystruct *pointer; would cause a compile-time error
} Mystruct;
```

The size of a struct type is at least the sum of the sizes of all its members. But a compiler is free to insert padding bytes between the struct members to align the members to certain constraints. For example, a struct containing of a char and a float will occupy 8 bytes on many 32bit architectures.

Unions

The definition of a union is similar to that of a struct. The difference between the two is that in a struct, the members occupy different areas of memory, but in a union, the members occupy the same area of memory. Thus, in the following type, for example:

```
union {
    int i;
    double d;
} u;
```

The programmer can access either `u.i` or `u.d`, but not both at the same time. Since `u.i` and `u.d` occupy the same area of memory, modifying one modifies the value of the other, sometimes in unpredictable ways. This is also the main reason that unions are rarely seen in practice.

The size of a union is the size of its largest member.

Enumerations

Enumerations are artificial data types representing associations between labels and integers. Unlike structs or unions, they are not composed of other data types. An example declaration:

```
enum color {
    red,
    orange,
    yellow,
    green,
    cyan,
    blue,
    purple,
} crayon_color;
```

In the example above, red equals 0, orange equals 1, ... and so on. It is possible to assign values to labels within the integer range, but they must be a literal.

Similar declaration syntax that applies for structs and unions also applies for enums. Also, one *normally* doesn't need to be concerned with the integers that labels represent:

```
enum weather weather_outside = rain;
```

This peculiar property makes enums especially convenient in switch-case statements:

```
enum weather {
    sunny,
    windy,
    cloudy,
    rain,
} weather_outside;

// ...

switch (weather_outside) {
case sunny:
    wear_sunglasses();
    break;
case windy:
    wear_windbreaker();
    break;
case cloudy:
    get_umbrella();
    break;
case rain:
    get_umbrella();
    wear_raincoat();
    break;
}
```

Enums are a simplified way to emulate associative arrays in C.

Retrieved from "https://en.wikibooks.org/w/index.php?title=C_Programming/Advanced_data_types&oldid=4198104"

This page was last edited on 29 October 2022, at 02:37.

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy.