Lecture 8: Custom Types and Data Structures in C

COMP26020 Part 1 (C) Lecture Notes

Pierre Olivier

These notes summarise the important points mentioned in the lectures. They are supposed to be a help for revising and not a way to avoid attending the live lectures and watching the videos. In other words, live lectures and videos may include examinable content that is not present in these notes.

The slides for this lecture are available here:
https://olivierpierre.github.io/comp26020-lectures/08-c-custom-types.

Videos and recordings of live sessions can be found on the video portal: https://video.manchester.ac.uk/lectures.

Here we cover how to create and use custom data types and structures in C.

Custom Types: `typedef`

To create a custom alias for a type, use the `typedef` keyword as follows:

```c
typedef long long unsigned int my_int;  // 'my_int' is now equivalent to 'long long unsigned int'
int main(int argc, char **argv) {
    my_int x = 12;
    printf("x is: %llu\n", x);
    return 0;
}
```

`typedef`, followed by the type to alias, here `long long unsigned int`, followed by the name of the alias, here `my_int`. Following that, one can use `my_int` to refer to `long long unsigned int`: it is much shorter.

Custom Data Structures: `struct`

Custom data structures are extremely useful in C. They aggregate several fields of various types. They are created and manipulated with the `struct` keyword. On this example we define a `struct person`:

```c
#include <stdio.h>
#include <string.h> // needed for strcpy

// Definition of the struct:
struct person {
    char name[10];
    float size_in_meters;
    int weight_in_grams;
};

void print_person(struct person p) {
    /* Access fields with '.' */
    printf("%s has a size of %f meters and "
        "weights %d grams\n", p.name,
        p.size_in_meters, p.weight_in_grams);
}
```

```c
int main(int argc, char **argv) {
    // declare a variable of the struct type:
    struct person p1;
    // sets the variable field
    p1.size_in_meters = 1.6;
    p1.weight_in_grams = 60000;
    strcpy(p1.name, "Julie");
    struct person p2 = {"George", 1.8, 70000};
    print_person(p1);
    print_person(p2);
    return 0;
}
```

It has a `name`, which is a character array of length 10. A `size_in_meter` which is a `float`. And a `weight` in grams which is an `int`. In the `main` function we declare a `struct person` variable `p1`.

We can set values in the fields with the `.` operator. We do not detail `strcpy` here, just know that it sets `Julie` in the `name` field of `p1`. We can also do static initialisation with braces followed by the fields value in order. We have a `print_person` function that takes a `struct person` as parameter and prints its field values.

In memory, the fields of a struct are placed contiguously. For our example, on x86-64 a `char` is one byte, so the array is 10 bytes in total. Followed by a `float` which is 4 bytes. Followed by an `int` which is also 4 bytes. So one instance of our `struct` should be 18 bytes. Note that the compiler may insert some padding for performance reasons.

Custom Data Structures and `typedef`

To avoid using the struct keyword each time one refer to a structure, `typedef` can be used:

```c
struct s_person {
    /* fields declaration ... */
};
typedef struct s_person person;
void print_person(person p) { /* ... */}
int main(int argc, char **argv) {
    person p1;
    person p2 = {"George", 1.8, 70000};
    /* ... */
}
```

After the `struct` declaration, use `typedef` followed by the name of the struct, here `s_person`, followed by the name of the alias, here `person`. And now one can just use `person`. A faster method to do so is to `typedef` during the struct declaration:

```c
typedef struct {
    /* fields declaration ... */
} person;
```

Enumerations

Enumerations are textual names that are mapped by the compiler to integer constants under the hood. They are useful in situation where integers are required, for example a `switch`, but when textual names are more meaningful. Here is an example, we define a color `enum` with 3 values: `RED`, `BLUE`, and `GREEN`:

```c
enum color {
    RED,
    BLUE,
    GREEN
};

int main(int argc, char **argv) {
```

```
    enum color c1 = BLUE;
    switch(c1) {
        case RED:
            printf("c1 is red\n"); break;
        case BLUE:
            printf("c1 is blue\n"); break;
        case GREEN:
            printf("c1 is green\n"); break;
        default:
            printf("Unknown color\n");
    }
    return 0;
}
```

We declare an `enum color` variable `c1` and sets it to `BLUE`. Then we can use it as the condition of a `switch`. Note that by convention constants are often in capital letters in C.

Same as for structs, enums can be used with typedef to avoid using the `enum` keyword So for example we simply use color to declare `c2`:

```
enum {
    BLUE,
    /* ,,, */
} e_color;
enum e_color c1 = BLUE; // without typedef
typedef enum e_color color;
color c2 = RED;
```

Or, in a shorter way:

```
typedef enum {
    BLUE,
    /* ... */
} color;
```