# 1 Structures in C

Very often, when you need to store data, you have several pieces of related information that could be stored together. The pedestrian way of doing it is creating a variable for each value that needs to be stored, and try to process them all properly. However, more often than not such techniques are totally inadequate for the task at hand.

Let us take an example that should be very familiar. Suppose we want to store information about a student in the ES112 course. The first thing we need to decide is the information we need to store. Here is the list of information we want to store:

- The first name of the student.
- The last name of the student.
- The ID number of the student.
- The two midterm grades of the student.
- The ten problem set grades of the student.
- The final grade of the student.
- The overall average of the student.
- The letter grade of the student.

Now, only if we had a data type that could store all this information in one piece, we could do really cool stuff with it. Well, such a data type is obviously not built-in, but we can make one for ourselves. In order to do this, we will need to define a `struct` first, as follows:

```
struct student {
    char first_name[40];
    char last_name[40];
    char id[15];
    int midterm[2];
    int homework[10];
    int final;
    float average;
    char letter_grade[3];
};
```

This is, for all practical purposes, the definition of a data type called "`struct student`". So, it should normally be placed directly within the file, above any functions.

So, now that we have defined such a `struct`, how do we define a variable of this type? The answer is, just like any other variable, as follows:

```
struct student mehmet;
```

Now, `mehmet` is a variable of type `struct student`, which holds all the information above in it. But how can assign values to the different fields of this structure, or read them out? For that purpose, the C language contains the dot operator ".". Look at the following example:

```
#include <stdio.h>
#include <string.h>

/* Definition of struct student should go here */

int main(void)
{
    struct student mehmet;
    int i;

    strcpy(mehmet.first_name, "Mehmet");
    strcpy(mehmet.last_name, "Ozturk");
    strcpy(mehmet.id, "9800820");
    mehmet.midterm[0] = 88;
    mehmet.midterm[1] = -1;

    for(i=0; i<10; i++) {
        mehmet.homework[i] = -1;
    }

    mehmet.homework[2] = 100;
    mehmet.homework[3] = 98;

    mehmet.final = -1;

    mehmet.average = -1.0;

    strcpy(mehmet.letter_grade, "NG");

}
```

Thus, the general way of accessing a field of a structure variable is:

```
<struct-variable>.<field-name>
```

## 2 Pointers to Structures

The real power of structures can only be realized by utilizing pointers to them. A pointer to a structure is defined just as a pointer to any other type of variable is defined:

```
struct student *ptr;
```

There is one more operator you need to know before we can start having fun with operators. That is the arrow "->" operator. Now, suppose you have a pointer to a `struct student`, and you want to access the name field in there. How would you do that? First, you have to dereference the pointer, and the use the "." operator, as follows:

```
(*ptr).first_name
```

You need to use a pair of parentheses because the dot operator has greater priority than the dereference operator. However, there is an equivalent (yet easier to type and read) way of doing this using the arrow operator as follows:

```
ptr->first_name
```

In other words, if you have a pointer rather than the structure itself, you use the arrow operator rather than the dot operator.

# 3 A Note on Using Structures as Function Parameters

Since a structure is really a user-defined data type, you can also pass it as a function argument. Using the `struct student` data type, we could for example, define a function that has the following prototype:

```
void print_student_information(struct student st);
```

Recall that the function calling conventions in C dictate that all parameters are passed *by value*. This means that a copy of the variable being passed is made, which is local to the function called. The same applies to structures passed as parameters. If you call the above function, the following happens:

1. A new `struct student` is created in memory.
2. Contents of the parameter are copied to this new `struct student`.

What is the problem here? Well, it is two-fold. First, you are creating a structure which is about 100 bytes long, and using extra memory. Second, you are wasting time copying the contents of the whole structure. This is usually not desirable. The solution is usually to pass a *pointer* to the structure rather than the structure itself. So, the prototype of the above function would become:

```
void print_student_information(struct student *st);
```

In this way, only a pointer is passed to the function, and no large memory allocation or copying takes place. The rule of thumb is, unless you *really* want a private copy of the structure in the function, pass a pointer to the structure instead of the structure to the function.

# 4 Example: Complex Numbers

Without further ado, we jump directly to examples making use of structures. Here is an example showing how to implement complex number operations using structures:

```
#include <stdio.h>
#include <math.h>

/* z = x + iy */

struct complex {
  float x;
```

```
  float y;
};

float length(struct complex *z);
float argument(struct complex *z);
void add(struct complex *result, struct complex *a, struct complex *b);
void subtract(struct complex *result, struct complex *a, struct complex *b);
void negate(struct complex *z);

int main(void)
{
  struct complex a, b, c, d;

  a.x = 4;
  a.y = 3;

  b.x = 5;
  b.y = 12;

  printf("Length of %2.2f + %2.2fi = %2.2f\n", a.x, a.y, length(&a));
  printf("Length of %2.2f + %2.2fi = %2.2f\n", b.x, b.y, length(&b));

  add(&c, &a, &b);

  printf("Sum of these numbers: %2.2f + %2.2fi\n", c.x, c.y);

  subtract(&d, &a, &b);

  printf("Difference of these numbers: %2.2f + %2.2fi\n", d.x, d.y);

  printf("Argument of the difference: %2.3f\n", argument(&d));

  return 0;
}

float length(struct complex *z)
{
  return sqrt(z->x*z->x + z->y*z->y);
}

float argument(struct complex *z)
{
  return atan2(z->y, z->x);
}

void add(struct complex *result, struct complex *a, struct complex *b)
{
  result->x = a->x + b->x;
  result->y = a->y + b->y;
}
```

```
void subtract(struct complex *result, struct complex *a, struct complex *b)
{
  result->x = a->x - b->x;
  result->y = a->y - b->y;
}

void negate(struct complex *z)
{
  z->x = -z->x;
  z->y = -z->y;
}
```