

Structs Notes

In C, a structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling.

There are many situations in which we want to process data about a certain entity or object but the data consists of items of various types.

For example, the data for a student (the student record) may consist of several fields such as a name, address, and telephone number (all of type string); number of courses taken (integer); fees payable (floating-point); names of courses (string); grades obtained (character); and so on.

The data for a car may consist of manufacturer, model, and registration number (string); seating capacity and fuel capacity (integer); and mileage and price (floating-point).

For a book, we may want to store author and title (string); price (floating-point); number of pages (integer); type of binding: hardcover, paperback, spiral (string); and number of copies in stock (integer).

Suppose we want to store data for 100 students in a program. One approach is to have a separate array for each field and use subscripts to link the fields together. Thus, `name[i]`, `address[i]`, `fees[i]`, and so on, refer to the data for the *i*th student.

The problem with this approach is that if there are many fields, the handling of several parallel arrays becomes clumsy and unwieldy.

For example, suppose we want to pass a student's data to a function via the parameter list.

This will involve the passing of several arrays.

Also, if we are sorting the students by name, say, each time two names are interchanged, we have to write statements to interchange the data in the other arrays as well.

In such situations, C structures are convenient to use.

Consider the problem of storing a date in a program. A date consists of three parts: the day, the month, and the year.

Each of these parts can be represented by an integer.

For example, the date "September 14, 2006" can be represented by the day, 14; the month, 9; and the year, 2006.

We say that a date consists of three fields, each of which is an integer.

If we want, we can also represent a date by using the name of the month, rather than its number.

In this case, a date consists of three fields, one of which is a string and the other two are integers.

In C, we can declare a date type as a structure using the **keyword struct**. Consider this declaration:

```
struct date {int day, month, year};
```

It consists of the keyword struct followed by some name we choose to give to the structure (date, in the example); this is followed by the declarations of the fields enclosed in left and right braces.

Note the semicolon at the end of the declaration just before the right brace; this is the usual case of a semicolon ending a declaration.

The right brace is followed by a semicolon, ending the struct declaration.

We could also have written the declaration as follows, where each field is declared individually:

```
struct date
{
    int day;
    int month;
    int year;
};
```

The style above is preferred for its readability.

Given the struct declaration, we can declare variables

of type struct date, as follows:

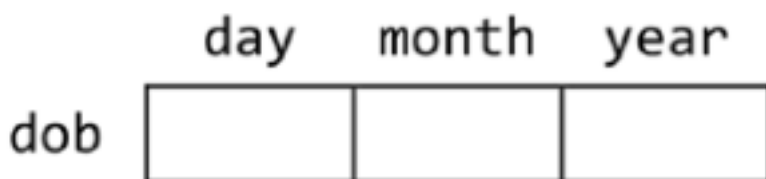
```
struct date dob; //dob is the variable of type struct date
```

To hold a "date of birth".

This declares dob as a “structure variable” of type date.

It has three fields called day, month, and year.

This can be pictured as follows:



We refer to the day field as `dob.day`, the month field as `dob.month`, and the year field as `dob.year`.

In C, the period (`.`), as used here, is referred to as the structure member operator.

In general, a field is specified by the structure variable name, followed by a period, followed by the field name.

We could declare more than one variable at a time, as follows:

```
struct date borrowed, returned;
```

Each of these variables has three fields: day, month, and year.

The fields of borrowed are referred to by **borrowed.day**, **borrowed.month**, and **borrowed.year**.

The fields of returned are referred to by **returned.day**, **returned.month**, and **returned.year**.

In this example, **each field is an int** and can be used in any context in which an int variable can be used.

For example, to assign the date “November 14, 2015” to dob, we can use this:

```
dob.day = 14;  
dob.month = 11;  
dob.year = 2015;
```

	day	month	year
dob	14	11	2015

We can also read values for day, month, and year with the following:

the following:

```
scanf("%d %d %d", &dob.day, &dob.month, &dob.year);
```

Suppose today was declared as follows:

```
struct date today;
```

Assuming we had stored a value in today, we could then assign all the fields of today to dob with the following:

```
dob = today;
```

This one statement is equivalent to the following:

```
dob.day = today.day;  
dob.month = today.month;  
dob.year = today.year;
```

We can print the “value” of dob with this:

```
printf("The party is on %d%d%d\n", dob.day, dob.month, dob.year);
```

For this example, the following will be printed:

The party is on 14/11/2015

Note that each field has to be printed individually.

We could also write a **function printDate**, say, which prints a date given as an argument.

The following program shows how **printDate** can be written and used.


```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  struct date
5  {
6      int day;
7      int month;
8      int year;
9  };
10
11 int main()
12 {
13     struct date dob;
14     void printDate(struct date);
15     dob.day = 14;
16     dob.month = 11;
17     dob.year = 2015;
18     printDate(dob);
19
20 }
21
22 void printDate(struct date d)
23 {
24     printf("%d-%d-%d\n", d.day, d.month, d.year);
25 }
```

```
C:\Users\HP-EliteBook\Downloads\C\PrintDateLearnC2\bin\Debug\PrintDateLearnC2.exe
14-11-2015
Process returned 0 (0x0)   execution time : 1.903 s
Press any key to continue.
```

We note, in passing, that C provides a date and time structure, `tm`, in the standard library. In addition to the date, it provides, among other things, the time to the nearest second.

To use it, your program must be preceded by the following:

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



The **construct struct date** is a bit cumbersome to use, compared to single word types such `int` or `double`.

Fortunately, C provides us with **typedef** to make working with structures a little more convenient.

We can use `typedef` to give a name to some existing type, and this name can then be used to declare variables of that type.

We can also use `typedef` to construct shorter or more meaningful names for predefined C types or for user-declared types, such as structures.

declared types, such as structures.

For example, the following statement declares a new type-name `Whole`, which is synonymous with the predefined type `int`:

```
typedef int Whole;
```

Note that `Whole` appears in the same position as a variable would, not right after the word `typedef`. We can then declare variables of type `Whole`, as follows:

```
Whole amount, numCopies;
```

This is exactly equivalent to:

```
int amount, numCopies;
```

For those accustomed to the term `real` of languages like Pascal or FORTRAN, the following statement allows them to declare variables of type `Real`:

```
typedef float Real;
```

In this book, we use at least one uppercase letter to distinguish type names declared using typedef.

We could give a short, meaningful name, Date, to the date structure shown earlier with the following declaration.

```
typedef struct date
{
    int day;
    int month;
    int year;
} Date;
```

Recall that C distinguishes between uppercase and lowercase letters so that date is different from Date.

We could, if we wanted, have used any other identifier, such as DateType, instead of Date.

We could now declare “structure variables” of type Date, such as the following:

```
Date dob, borrowed, returned;
```

Notice how much shorter and neater this is compared to the following:

```
struct date dob, borrowed, returned;
```

Since there is hardly any reason to use this second form, we could omit date from the declaration above and write this:

```
typedef struct
{
    int day;
    int month;
    int year;
} Date;
```

Thereafter, we can use Date whenever the struct is required.

For example, we can rewrite printDate as follows:

```
void printDate(Date d)
{
    printf("%d%d%d\n", d.day, d.month, d.year);
}
```

To pursue the date example, suppose we want to store the “short” name – the first three letters, for example Aug – of the month.

We will need to use a declaration such as this:

```
typedef struct
{
    int day
    char month[4]; //one position for \0 to end string
    int year;
} Date;
```

We can represent the date “November 14, 2015” in a Date variable dob with the following:

```
dob.day = 14;
strcpy(dob.month, "Nov"); //remember to include string.h
dob.year = 2015;
```

And we can write printDate as follows:

```
void printDate(Date d)
{
    printf("%d%d%d\n", d.month, d.day, d.year);
}
```

The call, printDate(dob), will print this: Nov 14, 2015.

Suppose we want to store information about students.

For each student, we want to store their name, age, and gender (male or female).

Assuming that a name is no longer than 30 characters, we could use the following declaration:

```
typedef struct
{
    char name[31];
    int age;
    char gender;
} Student;
```

We can now declare variables of type Student, as follows:

```
Student stud1, stud2;
```

Each of stud1 and stud2 will have its own fields: name, age, and gender.

We can refer to these fields as follows:

```
stud1.name stud1.age stud1.gender  
stud2.name stud2.age stud2.gender
```

As usual, we can assign values to these fields or read values into them. And, if we want, we can assign all the fields of stud1 to stud2 with one statement:

```
stud2 = stud1;
```

ARRAY OF STRUCTURES

Suppose we want to store data on 100 students.

We will need an array of size 100, and each element of the array will hold the data for one student.

Thus, each element will have to be a structure – we need an “array of structures.”

We can declare the array with the following, similar to how we say “`int pupil[100]`” to declare an integer array of size 100:

```
Student pupil[100];
```

This allocates storage for `pupil[0]`, `pupil[1]`, `pupil[2]`, up to `pupil[99]`.

Each element `pupil[i]` consists of three fields that can be referred to as follows:

```
pupil[i].name    pupil[i].age    pupil[i].gender
```

First we will need to store some data in the array. Assume we have data in the following format (name, age, gender):

```
"Jones, John" 24 M  
"Moringer, Lisa" 33 F  
"Sifa, Sandy" 29 F  
"Layne, Dennis" 49 M  
"END"
```

Suppose the data are stored in a file `input.txt` and `in` is declared as follows:

```
FILE *in = fopen("input.txt", "r");
```

If `str` is a character array, assume we can call the function:

```
getString(in, str)
```

to store the next data string in quotes in str without the quotes.

Also assume that readChar(in) will read the data and return the next non-whitespace character.

Exercise: Write the functions getString and readChar. We can read the data into the array pupil with the following code:

```
int n = 0;
char temp[31];
getString(in, temp);
while(strcmp(temp, "END") != 0)
{
    strcpy(pupil[n].name , temp);
    fscanf(in, "%d", &pupil[n].age);
    pupil[n].gender = readChar(in);
    n++;
    getString(in,temp);
}
```

At the end, n contains the number of students stored, and pupil[0] to pupil[n-1] contain the data for those students.

To ensure that we do not attempt to store more data than we have room for in the array, we should check that n is within the bounds of the array.

Assuming that MaxItems has the value 100, this can be done by changing the while condition to the following:

```
while(n < MaxItems && strcmp(temp, "END") != 0)
```

or by inserting the following just after the statement `n++`; inside the loop:

```
if(n == MaxItems) break;
```

SEARCHING AN ARRAY OF STRUCTURES

With the data stored in the array, we can manipulate it in various ways.

For instance, we can write a function to search for a given name.

Assuming the data is stored in no particular order, we can use a sequential search as follows:

```
//SEQUENTIAL SEARCHING AN ARRAY OF STRUCTURE
```

```
int search(char key[], Student list[], int n)
{
    //search for key in list[0] to list[n-1]
    //if found, return the location; if not found, return -1
    for (int h = 0; h < n; h++)
        if(strcmp(key, list[h].name) == 0) return h;
    return -1;
} //end search
```

Given the previous data, the call `search("Singh, Sandy", pupil, 4)` will return 2, and the following call will return -1: `search("Layne, Sandy", pupil, 4)`.

NESTED STRUCTURES

C allows us to use a structure as part of the definition of another structure – a structure within a structure, called a **nested structure**.

Consider the Student structure. Suppose that, instead of age, we want to store the student's date of birth.

This might be a better choice since a student's date of birth is fixed, whereas his age changes, and the field would have to be updated every year.

We could use the following declaration:

```
typedef struct
```

```
{  
    char name[31];  
    Date dob;  
    char gender;  
}  
Student;
```

If `mary` is a variable of type `Student`, then `mary.dob` refers to her date of birth. But `mary.dob` is itself a `Date` structure.

If necessary, we can refer to its fields with `mary.dob.day`, `mary.dob.month`, and `mary.dob.year`.

If we want to store a name in a more flexible way – for example, first name, middle initial, and last name, we could use a structure like this:

```
typedef struct
```

```
{  
    char first[21];  
    char middle;  
    char last[21];  
}  
Name;
```

The `Student` structure now becomes the following, which contains two structures, `Name` and `Date`:

```
typedef struct
```

```
{  
    Name name; //assumes Name has already been declared  
    Date dob; //assumes Date has already been declared  
    char gender;  
} Student;
```

If **st** is a variable of type **Student**, **st.name** refers to a structure of the type **Name**; **st.name.first** refers to the student's first name;

And **st.name.last[0]** refers to the first letter of her last name.

Now, if we want to sort the array **pupil** by last name, the while condition in the function **sort** becomes this:

```
while (k >= 0 && strcmp(temp.name.last, pupil[k].name.last) < 0)
```

A structure may be nested as deeply as you want.

The dot (.) operator associates from left to right. If **a**, **b**, and **c** are structures, the construct.

`a.b.c.d` is interpreted as `((a.b).c).d`

WORKING WITH FRACTIONS

Consider the problem of working with fractions, where a fraction is represented by two integer values:

one for the numerator and the other for the denominator.

For example, 5/9 is represented by the two numbers 5 and 9.

We will use the following structure to represent a fraction:

```
typedef struct
{
    int num;
    int den;
} Fraction;
```

If `f` is variable of type `Fraction`, we can store 5/9 in `f` with this:

```
f.num = 5;  
f.den = 9;
```

	num	den
f	5	9

We can also read two values representing a fraction and store them in `f` with a statement such as this:

```
scanf("%d %d", &f.num, &f.den);
```

We can write a function, `printFraction`, to print a fraction. It is shown in the following program.

```
#include <stdio.h>

typedef struct
{
    int num;
    int den;
} Fraction;

int main()
{
    void printFraction(Fraction)
    Fraction f;
    f.num = 5;
    f.den = 9;
    printFraction(f);
}

void printFraction(Fraction f)
{
    printf("%d%d", f.num, f.den);
}
```

When run, the program will print:

5/9

MANIPULATE FRACTIONS

We can write functions to perform various operations on fractions. For instance, since:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

we can write a function to add two fractions as follows:

```
Fraction addFraction(Fraction a, Fraction b)
{
    Fraction c;
    c.num = a.num * b.den + a.den * b.num;
    c.den = a.den * b.den;
    return c;
} //end addFraction
```

Similarly, we can write functions to subtract:


```
Fraction subFraction(Fraction a, Fraction b)
{
    Fraction c;
    c.num = a.num * b.den - a.den * b.num;
    c.den = a.den * b.den;
    return c;
} //end subFraction
```

Multiply:

```
Fraction mulFraction(Fraction a, Fraction b)
{
    Fraction c;
    c.num = a.num * b.num;
    c.den = a.den * b.den;
    return c;
} //end mulFraction
```

Divide:

```
Fraction divFraction(Fraction a, Fraction b)
{
    Fraction c;
    c.num = a.num * b.den;
    c.den = a.den * b.num;
    return c;
} //end divFraction
```

To illustrate their use, suppose we want to find $\frac{2}{5}$ of $\{\frac{3}{7} + \frac{5}{8}\}$:

We can do this with the following statements:

```
Fraction a, b, c, sum, ans;  
a.num = 2; a.den = 5;  
b.num = 3; b.den = 7;  
c.num = 5; c.den = 8;  
sum = addFraction(b,c);  
ans = mulFraction(a, sum);  
printFraction(ans);
```

Strictly speaking, the variables `sum` and `ans` are not necessary, but we've used them to simplify the explanation.

Since an argument to a function can be an expression, we could get the same result with this:

```
printFraction(mulFraction(a, addFraction(b, c)));
```

When run, this code will print the following, which is the correct answer:

118/280

However, if you want, you can write a function to reduce a fraction to its lowest terms.

This can be done by finding the highest common factor (HCF) of the numerator and denominator.

You then divide the numerator and denominator by their HCF.

For example, the HCF of 118 and 280 is 2 so $118/280$ reduces to $59/140$.

Writing this function is left as an exercise.