Subsections

- Structures
  - Defining New Data Types
- Unions
- Coercion or Type-Casting
- Enumerated Types
- Static Variables
- Exercises

Further Data Types

This Chapter discusses how more advanced data types and structures can be created and used in a C program.

Structures

Structures in C are similar to records in Pascal. For example:

```
  struct gun
                                  {
                                  char name[50];
                                  int magazinesize;
                                  float calibre;
                                  };

                struct gun arnies;
```

defines a new structure `gun and makes arnies an instance of it.`

NOTE: that gun is a ***tag*** for the structure that serves as shorthand for future declarations. We now only need to say struct gun and the body of the structure is implied as we do to make the arnies variable. The tag is ***optional***.

Variables can also be declared between the } and ; of a struct declaration, ***i.e.***:

```
  struct gun
                                  {
                                  char name[50];
                                  int magazinesize;
                                  float calibre;
                                  } arnies;
```

struct's can be pre-initialised at declaration:

```
  struct gun arnies={"Uzi",30,7};
```

which gives arnie a 7mm. Uzi with 30 rounds of ammunition.

To access a member (or field) of a struct, C provides the . operator. For example, to give arnie more rounds of ammunition:

```
   arnies.magazineSize=100;
```

Defining New Data Types

`typedef` can also be used with structures. The following creates a new type `agun` which is of type `struct gun` and can be initialised as usual:

```
typedef struct gun
                                {
                                char name[50];
                                int magazinesize;
                                float calibre;
                                } agun;

              agun arnies={"Uzi",30,7};
```

Here `gun still acts as a` ***tag*** `to the struct and is optional. Indeed since we have defined a new data type it is not really of much use,`

`agun is the new data type. arnies is a variable of type agun which is a structure.`

`C also allows arrays of structures:`

```
typedef struct gun
                                {
                                char name[50];
                                int magazinesize;
                                float calibre;
                                } agun;

             agun arniesguns[1000];
```

`This gives arniesguns a 1000 guns. This may be used in the following way:`

```
        arniesguns[50].calibre=100;
```

`gives Arnie's gun number 50 a calibre of 100mm, and:`

```
        itscalibre=arniesguns[0].calibre;
```

`assigns the calibre of Arnie's first gun to itscalibre.`

Unions

A union is a variable which may hold (at different times) objects of different sizes and types. C uses the `union` statement to create unions, for example:

```
        union number
                                {
                                short shortnumber;
                                long longnumber;
                                double floatnumber;
                                } anumber
```

defines a union called `number and an instance of it called anumber. number is a union` ***tag*** `and acts in the same way as a tag for a structure.`

`Members can be accessed in the following way:`

```
printf("%ld\n",anumber.longnumber);
```

This clearly displays the value of longnumber.

When the C compiler is allocating memory for unions it will always reserve enough room for the largest member (in the above example this is 8 bytes for the double).

In order that the program can keep track of the type of union variable being used at a given time it is common to have a structure (with union embedded in it) and a variable which flags the union type:

An example is:

```
typedef struct
                                { int maxpassengers;
                                } jet;

              typedef struct
                                { int liftcapacity;
                                } helicopter;

              typedef struct
                                { int maxpayload;
                                } cargoplane;

               typedef                  union
                                { jet jetu;
                                                helicopter helicopteru;
                                                cargoplane cargoplaneu;
                                } aircraft;

              typedef                   struct
                                { aircrafttype kind;
                                                int speed;
                                                aircraft description;
                                } an_aircraft;
```

This example defines a base union aircraft which may either be jet, helicopter, or
cargoplane.

In the an_aircraft structure there is a kind member which indicates which structure is being held at the time.

Coercion or Type-Casting

C is one of the few languages to allow ***coercion***, that is forcing one variable of one type to be another type. C allows this using the cast operator `()`. So:

```
int integernumber;
              float floatnumber=9.87;

                                integernumber=(int)floatnumber;
```

assigns 9 (the fractional part is thrown away) to `integernumber`.

And:

```
int integernumber=10;
                float floatnumber;

                                    floatnumber=(float)integernumber;
```

assigns 10.0 to floatnumber.

Coercion can be used with any of the simple data types including char, so:

```
          int integernumber;
                char letter='A';

                                    integernumber=(int)letter;
```

assigns 65 (the ASCII code for `A') to integernumber.

Some typecasting is done automatically -- this is mainly with integer compatibility.

A good rule to follow is: **If in doubt cast.**

Another use is the make sure division behaves as requested: If we have two integers internumber and anotherint and we want the answer to be a float then :

```
e.g.
 floatnumber =
                (float) internumber / (float) anotherint;
```

ensures floating point division.

Enumerated Types

Enumerated types contain a list of constants that can be addressed in integer values.

We can declare types and variables as follows.

```
enum days {mon, tues, ..., sun} week;

enum days week1, week2;
```

NOTE: As with arrays first enumerated name has index value 0. So `mon` has value 0, `tues` 1, and so on.

`week1` and `week2` are variables.

We can define other values:

```
enum escapes { bell = `\a',
```

```
backspace = `\b',  tab = `\t',

newline = `\n', vtab = `\v',

return = `\r'};
```

We can also override the 0 start value:

```
enum months {jan = 1, feb, mar, ......, dec};
```

Here it is implied that feb = 2 ***etc.***

Static Variables

A **`static`** variable is local to particular function. However, it is only initialised once (on the first call to function).

Also the value of the variable on leaving the function remains **intact**. On the next call to the function the the `static` variable has the same value as on leaving.

To define a `static` variable simply prefix the variable declaration with the `static` keyword. For example:

```
void stat(); /* prototype fn */

               main()
                { int i;

                              for (i=0;i<5;++i)
                                             stat();
                }


stat()
                {             int auto_var = 0;
                               static int static_var = 0;

                               printf( ``auto = %d, static = %d \n'',

                                             auto_var, static_var);
                               ++auto_var;
                               ++static_var;
                }
```

Output is:

```
auto_var = 0, static_var= 0
              auto_var = 0, static_var = 1
              auto_var = 0, static_var = 2
              auto_var = 0, static_var = 3
              auto_var = 0, static_var = 4
```

Clearly the auto_var variable is created each time. The static_var is created once and remembers its value.

Exercises

Exercise 12386

Write program using enumerated types which when given today's date will print out tomorrow's date in the for 31st January, for example.

Exercise 12387

Write a simple database program that will store a persons details such as age, date of birth, address ***etc.***

Dave Marshall
1/5/1999