Data Aggregates

Overview

- Structures
- Unions
- Typedef
- Bitfields

2

Data Aggregates

This chapter shows some uses of types available in C which use the earlier mentioned fundamental data types. They have a higher level of definition.

Intro to Structures

- Aggregate of other types, much like a database record
- Structure creates new type

  struct <name> { [members] };
- Example a new type representing a point

```
struct point
{
  int x;
  int y;
};
```

3

Structs

A struct is an example of an aggregate. It consists of a number of items called members. A structure is a type definition of something that consists of more than one value. Creating variables of a structure creates variables consisting of other variables like a record in a database. In a database a record is an element of a database table and each record consists of several data elements. For example a record in an address database will typically consist of a name, address, city, phone number. Each record in the database will consists of these aggregate parts.

The struct is declared using the keyword *struct*:

```
struct Record
{
    char name[30];      /* Name of max 29 characters */
    char city[20];      /* City of max 19 characters */
    char street[30];    /* Street of max 29 characters */
    int number;         /* Street number */
    char phone[11];     /* Phone number max 10 characters */
};
```

The struct declaration is nothing more than a type definition. The members defined in the structure become concrete members when we create a variable of this struct type.

Using Structure

- Use structure as a new built-in type

```
struct point pt;
struct point pt2 = {10, 11};
```

- Access members of struct using the dot (.)

```
pt.x = 10
pt.y = 20
```

- Can assign (copy) struct variables

```
pt = pt2;
```

4

Using the Structure

If we want to use the structure we need to create variables of the struct type we just created. The variables are created like normal variables of a fundamental type using the type and a variable name, but we have to use the keyword *struct*.

```
void main()
{
    int int_var;        /* Normal int variable */
    struct point pt;    /* Variable of the struct type point */
    struct point pt2;   /* Another variable of type point */
}
```

After creating the point we can access its members by using a period (.) and the name of the member.

```
void main()
{
    int int_var;        /* Normal int variable */
    struct point pt;    /* Variable of the struct type point */
    struct point pt2;   /* Another variable of type point */

    /* Initialise the members of the two points */

    pt.x = 10.0;
    pt.y = 20.0;
    pt2.x = 30.0;
    pt2.y = 40.0;

    /* Assign two structs */
    pt = pt2;
}
```

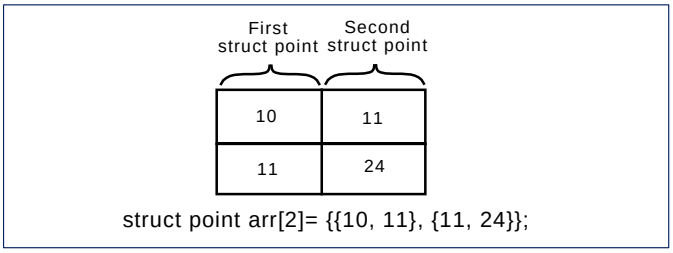
It is possible to assign two variables of the same structure. When we assign them all members of the struct variable will be copied.

```
/* Program to show the use of structs.
   (C) Datasim BV 1995 */

#include <stdio.h>

struct point
{
   int x;                     // The x-co-ordinate
   int y;                     // The y-co-ordinate
};

void main()
{

    struct point my_point = { 1, -2 };

    printf("The size of a point struct is: %d\n", sizeof(struct point));
    printf("x: %d, y: %d\n", my_point.x, my_point.y );
}
```

Arrays of Structures

- Create an array of a structure
- Each element is a variable of that structure
- Can initialise the elements upon declaring using {}

```
struct point arr[2] = {{10, 11}, {11, 24}};
```



5

Arrays of Structs

In the previous modules we created arrays of numbers and arrays of characters (strings). It is also possible to create arrays of structures. If we create such an array then each element in the array is a structure with its own fields. Declaring such an array is rather straight forward:

```
void main()
{
    struct point arr[4];
}
```

The variable *arr* is an array of four point structures. If we want to initialise their values we first have to index in the array to get the structure and then use the period to access the elements of the structure.

```
void main()
{
    struct point arr[4];

    arr[0].x = 10;      /* The first structure field x */
    arr[0].y = 11;      /* The first structure field y */
    arr[1].x = 12;      /* The second structure field x */
    arr[1].y = 13;      /* The second structure field y */
}
```

Another possibility is to initialise the array upon creation by specifying the value of each element in the array between {} brackets. In this case each element is a structure which can also be initialised by {} brackets.

```
void main()
{
    struct point arr[2] = {{10, 11}, {11, 24}};
}
```

The following example shows how to create an array of objects with each object being itself a struct.

```
/* Program to show the use of structs (arrays of structs).
   (C) Datasim BV 1995 */

#include <stdio.h>

struct point
{
  int x;  // The x-co-ordinate
  int y;  // The y-co-ordinate
};

void main()
{
  struct point pl[4] = { {0, 0}, {1, 0}, {1, 1}, {0, 1}};
  int cnt;

  // Now print the points of the polyline
  for (cnt = 0; cnt < 4; cnt++)
  {
    printf("Point number %d: (%d, %d)\n", (cnt+1), pl[cnt].x, pl[cnt].y);
  }
}
```

Structures and Pointers

- Access members of structures using dot (.)

```
struct point pt;
pt.x = 10.0;
```

- When using pointers to structures use a -> to access struct members

```
struct point* pp = &pt;
pt->x=20.0;   /* Use -> to access members */
(*pt).y=30.0; /* Use dereference and . */
```

6

Structures and Pointers

Normally we access the elements of a structure using a period. This period gives us access to a member of a struct variable. However using pointers to structures is a bit more complicated.

```
struct point pt;
struct point* pp;
pp = &pt;
```

The pointer *pp* points to the variable *pt* which is a *point* struct. Accessing its members through the pointer is a bit more difficult. First we need to get the structure the pointer points to by dereferencing:

```
pp          /* pointer to structure */
(*pp)       /* structure that pp points to */
```

Next we can access the members using the period:

```
(*pp).x = 10.0;
(*pp).y = 20.0;
```

Another way of accessing the fields when using a pointer to a structure is with the symbol '->' (minus greater-than):

```
pp->x = 10.0;
pp->y = 20.0;
```

Passing Structures

- When passing structures to functions all members are copied
- Use pointers to structures for passing structures

```
void print(struct point* pt);
```

7

Passing Structures

Normally when you pass a value to a function the value you pass is copied to the argument of the function. The function uses a copy of the original value. This is called "call by value".

```
void print(int a)
{
    printf("%d\n", a);
}

void main()
{
    int value = 10;
    print(value); /* value is copied to a */
}
```

When we pass a structure to a function, a copy is created as well. Copying a structure results in copying all the fields of that structure. Bigger structures will result in a big overhead in copying. To solve this, the best way is to always use a pointer when passing structures to functions:

```
void print(struct point* pt)
{
    printf("(%d, %d)\n", pt->x, pt->y);
}

void main()
{
    struct point point_1;

    point_1.x = 10.0;
    point_1.y = 20.0;
    print(&point_1);
}
```

Unions

- Or relation: variables contain value of one of the union field types
- Same syntax as struct only type union

```
union u_tag
{
  int ival;
   float fval;
};
```

- Size equal to size of largest member

8

Unions

The union has a resemblance to a struct. The declaration of a union is almost the same as that of a structure. Only instead of the keyword *struct* we use the keyword *union*. The syntax:

```
union <name> {<Fields>};
```

The operations are the same for unions as that for structs. The difference between the struct and the union is that the struct is an AND relation and the union is an OR relation. When we create a variable of a predefined structure then that variable will have all the fields that are declared in the structure. If we create a variable of a union type, that variable will contain a value which is one of the fields declared in the union.

```
union u_tag
{
    int ival;
    float fval;
};

void main()
{
    union u_tag var;
}
```

The variable fields of the variable *var* can be accessed using the period and the field name. If we assign a value to such a field the variable *var* will contain the value assigned to that field. The other fields should not be used because all the fields occupy the same memory. The variable *var* should contain or an *int* value assigned via the field *ival* or a *float* value assigned via the field *fval*:

```
void main()
{
    union u_tag var;
    var.ival = 10;      /* Variable var contains an int value */
    var.fval = 20.0;    /* Variable var contains a float value */
}
```

If we assign for example a value to the field *ival*, we can access the *float* value *fval* but its value will be unpredictable because it is filled using the type *int*. Using variable of type union can be very cumbersome. If we want to access its value we have to know what value was placed in that union. That is why mostly unions are part of a structure with a field specifying what kind of value is stored in the union.

```
struct value
{
    union u_tag var;
    int type;    /* 0 = int, 1 = float */
};

void print(struct value* val);

void main()
{
    struct value variable;
    variable.type = 0;
    variable.var.ival = 10;
    print(&variable);

    variable.type = 1;
    variable.var.fval = 10.10;
    print(&variable);
}

void print(struct value* val)
{
    if (val->type == 0)
    {
          printf("An integer value : %d\n", val->var.ival);
    }
    else
    {
          printf("A float value : %f\n", val->var.fval);
    }
}
```

Typedef

- Assign another name to existing type
- Use typedef for substitution

```
typedef long int INT32;
```

- Do not forget the semi-colon

type def is a compiler construct, not a pre processor construct

9

Typdef

Using *typedef* it is possible to assign another name to an already existing type. Instead of the already existing type name, the *typedef* can be used as well.

```
typedef long int INT32;

void main()
{
    INT32 number;
    INT32 number2 = 10L;
}
```

The *typedef* is often used for portability reasons. Suppose all number used in a source file should be a 32 bit value. On most platforms the *int* type 32 bit. But on other platforms the *int* type might be 16 bit. If we port our code to these platforms, we have to change all the references to the *int* type into a *long int*. By using the *typedef* we only have to change the *typedef* and all reference to the *typedef* are automatically changed.

Bitfields

- Used for the fields of a structure
- We can specify the number of bits for each field
- Only possible for *int*

```
struct member
{
  unsigned int age          : 4;
  unsigned int golfclub     : 1;
  unsigned int footbalclub  : 1;
};
```

10

Bit Fields

Bit fields are integral types. They are highly implementation dependent and should be used with care, especially when attempting to write portable software.
The syntax of a bit-field is:

<identifier> : <constant_expression>

The <constant_expression> tells us how many bits must be allocated for <identifier>. For example:

```
struct my_bit
{
  int b1 : 24;
  int b2 : 16;
};
```

We summarise the implementation dependencies of bit-fields:

- The integral type can be *int*, *signed* or *unsigned*.
- Assignment of fields can be from left to right or from right to left.
- How a plain bit-field is interpreted depends on the underlying hardware.

Using bit-fields does not necessarily imply greater speed or efficiency.