Structures

This lecture deals with Structures.

Chapter 6 K & R

Structures

A structure puts together one or more variables of possibly different types under a single name.

Structures permit a group of related variables to be treated as a unit. So structures help to organize complex data.

Structures can be built in a hierarchical manner

e.g. A point can be defined as a pair of coordinates,
a rectangle can be defined as a pair of points etc.

Structure assignment permits structures to be copied.
Structures can be passed to functions and returned by functions.

Structures - Syntax

e.g.

```
/* decalres the structure of type called point */
struct  point {
        int    x;
        int    y;
};

/* defines a variable pt of type struct point */
struct point pt;

/*  both can be combined as follows */
struct  point {
        int    x;
        int    y;
}pt;
```

Structure Initialization

Structure Initialization
eg.

```
struct point origin = {0, 0};
```

Note that the initializers are constant expressions.

An automatic structure may also be initialized by assignment or by calling a function that returns a structure of the right type.

". " - The Dot Operator

The member of a particular structure is referred to in an expression by the form

structure-name.member

The structure membr operator " . " connects the structure name and the member name.

e.g.

```
struct point pt;

printf ("%d, %d", pt.x, pt.y);

pt.x = pt.x + pt.y;
```

Nested Structures

Structures can be nested , i.e., can be built into a hierarchy.

e.g. /* defines a type called rect */

```
struct rect {
        struct point pt1;
        struct point pt2;
};

struct rect screen;   /* declares variable screen */

screen.pt1.x          /* refers to the x coordinate of the
                         pt1 member of screen */
```

Structure Operations

Legal operations on structures:

- copying it or assigning to it as a unit
- taking its address with &
- accessing its members

Copy and assigment include passing arguments to functions and returning values from functions as well

Structures may be initialized by a list of constant member values, an automatic structure may also be initialized by an assignment.

No other operation is permitted in particular structures may not be compared

Structures and Functions

```
/*  function makepoint takes two integers and returns
    a point structure */

struct  point makepoint (int  x, int  y)
{
      struct point temp;

      temp.x  =  x;
      temp.y  =  y;
      return temp;
}
```

Note that x (as well as y) is used both as argument name and member name without any problem.

Structures and Functions

```
/*  function makepoint can be used to initialize any structure
      dynamically */

struct  rect  screen;
struct point middle;
struct point makepoint(int, int);

screen.pt1  =  makepoint (0, 0);
screen.pt2  =  makepoint (XMAX,  YMAX);
middle  = makepoint ((screen.pt1.x + screen.pt2.x) / 2,
                                (screen.pt1.y + screen.pt2.y) / 2);
```

Structures and Functions

```
/* addpoint:  add two points */

struct point addpoint (struct point p1, struct point p2)
{
      p1.x  +=  p2.x ;
      p1.y  +=  p2.y;
      return  p1;
}
```

Structure parameters are passed by value like any others.

Pointers to Structures

If a large structure is to be passed to a function, it is often more efficient to pass a pointer than to copy the whole structure.

Structure pointers are like pointers to ordinary variables.

```
struct   point  *pp;   /* declares pp as a pointer */
```

If pp points to a point structure (as in the above declaration),

*pp is a structure
(*pp).x   and (*pp).y are the members.

Pointers to Structures

```
struct   point  origin, *pp;

pp  =  &origin;
printf ("origin is (%d, %d)\n",  (*pp).x , (*pp).y);
```

Note: The parentheses are necessary in (*pp).x because the precedence of the structure member operator . is higher than *.

The expression *pp.x means *(pp.x) which is illegal as x is not a pointer.

"->" - The ' Arrow ' Operator

If p is a pointer to a structure, then

p -> member-of-structure
refers to a particular member.

```
struct   point  origin, *pp;

pp  =  &origin;
printf ("origin is (%d, %d)\n",  pp ->x ,  pp ->y);
```

"->" - The ' Arrow ' Operator

Both . and -> associate left to right.
struct rect r, *rp = &r;

the following expressions are equivalent

```
r.pt1.x
rp -> pt1.x
(r.pt1).x
(rp ->pt1).x
```

The Structure Operators

The structure operators . and -> together with ( ) for function calls and [ ] for subscripts are at the top of the precedence hierarchy and thus bind very tightly.

```
struct {
    int  len;
    char *str;
} *p;

++p ->len      /* increments len, not p */

(++p) ->len    /* increments  p before accessing len */

(p++) ->len    /* increments  p after accessing len */
```

The Structure Operators

```
struct {
        int    len;
        char *str;
} *p;

*p ->str      /*  same as  *(p ->str),  fetches what str points to*/

*p ->str++      /* increments str after accessing whatever it
                   points to (just like *s++)  */

(*p->str)++      /* increments  whatever str points to */

*p++->str      /* increments  p after whatever str points to */
```

Arrays of Structures

```
struct  key{
        char  *word;
        int   count;
} keytab[NKEYS];
```

The above declares a structure type key and defines an array keytab of structres and sets aside storage for them. Each element of the array keytab is a structure.

Another way of doing this is:

```
struct  key{
        char  *word;
        int   count;
};
struct key keytab[NKEYS];
```

Self-referential Structures

```
struct tnode{                    /* the tree node */
    char  *word;
    int   count;
    struct tnode *left;          /* left child */
    struct tnode *right;         /* right child */
};
```

The above is a recursive definition of tnode

It is illegal for a structure to contain an instance of itself, but

```
struct tnode *left;
```

declares left to be a pointer to tnode, not a tnode itself.

Typedef

Typedef facility can be used to create new data type names.
It only creates a synonym and not a new type.

```
typedef  int Length;   /* makes name Length a synonym for int */

Length  len, maxlen;
Length *lengths[ ];

typedef  struct tnode  *Treeptr;

typedef struct  tnode{                         /* the tree node */
            char  *word;
            int   count;
            Treeptr   left;          /* left child */
            Treeptr   right;         /* right child */
      }Treenode;
```

Typedef

```
Treeptr  talloc(void)
{
        return (Treeptr) malloc (sizeof(Treenode));
}
```

Unions

A union is a variable that may hold (at different times) objects of different types and sizes

```
union u_tag {
        int     ival;
        float fval;
        char *sval;
} u;
```

The variable u is large enough to hold the largest of the three types.

Members of union are accessed as

    union-name.member
        or
    union-pointer ->member

Command Line Arguments

```
 #include <stdio.h>
/* echo command-line arguments - version 1- page 115 K&R*/
main(int argc, char *argv[])
{
        int i;

        for( i  = 1;  i  <  argc;  i++)
                printf("%s  " , argv[i] );
        printf ("\n");
        return  0;
}
```

The program myecho.c compiled as

gcc myecho.c -o myecho

will produce the executable file myecho.

$ myecho hello, world

produces the output

hello, world

Command Line Arguments

argc and argv are a C mechanism for getting program arguments from the command line.

argc and argv parameters are passed automatically to the main () function by the operating system.

argc is the number of command line words, including the command name

if the program is executed with the command

    myecho hello, world

then argc is 3.

Command Line Arguments

The declaration

```
char   *argv[]
```

indicates that argv is an array of pointers to characters, i.e., argv is an array of character strings.

The number of entries of argv is argc.
The first one, argv[0], is the program name.
The rest, argv[1] ... argv[argc-1], are the command line arguments.

In the command

```
myecho  hello, world
```

argv[0] - "myecho"
argv[1] - "hello,"
argv[2] - "world"

Command Line Arguments

argc is the number of command-line arguments the program is invoked with

argv is a pointer to an array of character strings that contain the argument,one per string

argv[0] is the name by which the program is invoked

argc is at least 1
argv[1] is the first operational argument
argv[argc-1] is the last operational argument
argv[argc] is a null pointer

Command Line Arguments

```
#include <stdio.h>
/* echo command-line arguments - version  2- page 115
K&R*/
main(int argc, char  *argv[])
{
    while(--argc  >  0)
        printf ("%s " ,  *++argv);
    printf( "\n");
    return  0;
}
```

$ myecho hello, world

produces the output

hello, world

Command Line Arguments

argc is the number of command-line arguments the program is invoked with

argv is a pointer to an array of character strings that contain the argument,one per string

argv[0] is the name by which the program is invoked

argc is at least 1
argv[1] is the first operational argument
argv[argc-1] is the last operational argument
argv[argc] is a null pointer

Command Line Arguments

```
#include <stdio.h>

/* echo command-line arguments - version 3*/
main(int argc, char **argv)
{
    while(--argc > 0)
        printf ("%s %s" , *++argv, (i < argc-1) ? " " : "");
    printf( "\n");
    return 0;
}
```

$ myecho hello, world
produces the output
hello, world

Command Line Arguments

Note that each element of the array argv [ ] is a string. This aspect is brought out by the program below

```
#include <stdio.h>
/* power program with command line arguments*/
main(int  argc,  char  *argv[ ])
{
        double  x;
        int       n;

        if (argc < 3) {
                printf (" missing arguments,  power [base] [index]\n");
                exit(1);
        }
        else {
                x = atof (argv [1]);
                n = atoi (argv [2]);
                printf ("%f \n", power(x, n));
                exit(0);
        }
}
```