Functions and Program Structure

This lecture deals with functions and program structure.

Chapter 4 K & R

Functions and Program Structure

The idea of a function is a key structuring principle in the construction of any C program.

Any computation of a C program is broken into several functions and each function performs a simple well-defined task.

Functions are often shared across programs, a function designed in the context of a particular program finds use in another program.

Role of Functions

The idea of a function is a key structuring principle in the construction of any C program.

Any computation of a C program is broken into several functions and each function performs a simple well-defined task.

Functions are often shared across programs, a function designed in the context of a particular program finds use in another program.

Function as an Abstraction

Function is also a unit of abstraction, it is unnecessary to know how it works in order to use it.

C promotes the view that a program is built from several small functions rather than a few large functions.

So functions not only keep the complexity of the program under control (program easier to modify, maintain, debug use) but also promote 'sharing' (functions built by one user can be used by another for a different purpose)

Function Definition - An Example

```
/* function to calculate greatest common divisor
of two nonnegative integers */

int gcd  (int u, int v)      /*  u and v  formal parameters  */
{
    int tmp;                  /*    tmp  automatic variable      */

    while  (v  !=  0)  {
            tmp  =  u  %  v;
            u  =  v;
            v  =  tmp;
   }

    return  (u) ;
}
```

Function Definition Syntax

```
return-type function-name ( argument declarations)
{
  declarations and statements
}
```

Various parts may be absent; a minimal function is

```
dummy()  {}
```

which does nothing and returns nothing.

If the return type is omitted, int is assumed.

In C all function arguments are passed by value.
Function definitions can appear in any order, and in one source file or several , although no function can be split between files.

Return statement

The return statement is a mechanism for returning a value from the called function to its caller.

Any expression can follow return:

```
  return expression;
```

The expression will be converted to the return type of the function if necessary.

The called function is free to ignore the returned value. Also there is no need for any expression after return.

Control also returns to the caller with no value when the execution "falls off the end" of the function by reaching the closing right brace.

Function - Example (K&R page 24)

```
#include  <stdio.h>

int   power(int m, int n);  /* function declaration */

/* to test power function   */
main()
{
        int   i;
        for (i  =  0;  i  <  10;  ++i)
                printf("%d  %d\n", i, power(2, i));
        return 0;
}
```

Function - Example (K&R page 25)

```
/*  power:  raise  base to n-th power;  n  >=  0  */

int  power(int  base, int n);
{
     int   i,  p;

     p  =  1;
     for (i  =  0;  i  <  n;  ++i)
          p  =  p  *  base;
     return p;
}
```

Automatic Variables

Variables that are private or local to a function are called automatic variables.

No other function can have direct access to them.

Each local variable comes into existence when the function is called and disappears when the function is exited.

So they do not retain their values from one function call to the next.

e.g. in the function power(), the variables i and p are automatic.

External Variables

External variables are defined outside of any function.

External variables are globally accessable.

Functions are always external because in C functions cannot be defined inside other funtions.

External variables and functions can be defined to be visible only within a single file.

External variables are permanent and they retain their values from one function invocation to the next.

External Variables - Examples

```
/*  in file main.c  */
int     i  =  5;

main()
{
        printf  ("%d  ", i);
        foo  ();
        printf  ("%d\n", i);
}
-----------------------------------------
/*  in file foo.c  */
external    int   i;
foo()
{
        i = 100;
}
```

Static vs External

If a variable is defined outside of any function, qualifier static restricts the visibilty of the variable to the file containing its definition.

Static vs External

```
/*  in file mod1.c   */

double x;
static double result;

static void dosquare  ()

{
        double square ();
        x  =  2.0;
        result  =  square ();
}

main()
{
        dosquare();
        printf  ("%f\n",  result);
}
```

```
/*  in file mod2.c  */

extern   double x;


double  square  ()
{

        return  (x   *   x);

}
```

Header Files

The definitions and declarations shared among the files may be centralized as much as possible so that there is only one copy to get right and keep right as the program evolves.

This common information is kept in a header that ends in .h

For programs of moderate size it is better to have one header file that contains everything that is needed to be shared between any twoparts in the parts in the program.

The header file is included using the   #include <filename.h>

C Preprocessor

Key features of the C preprocessor are

#include - to include a file during compilation
#define - to replace a token by an arbitrary sequence of characters.

#include "filename" vs #include <filename>

#include is a preferred way to tie the declarations together for a large program. It guarentees that all the source files will be supplied with the same definitions and variable declarations.

When an included file is changed, all the files that depend on it must be recompiled.

Files that are included by #include directive may themselves contain #include directives.

C Preprocessor

#define provides a simple macro substitution facility.
It is commonly used to parametrize a program

```
#define  MAXLENGTH   100
...
char linebuffer [MAXLENGTH];
...
for (i  =  0;  i   <   MAXLENGTH;  i++)   {
...
}
...
```

Macros may be parameterized.

```
#define   MAX  (a,  b)     (a  >  b?  a: b)
...
       i  = MAX  (j,  20);
```

will expand to

```
       i  =  (j  >  20?  j:  20);
```

Recursion

Recursive function is defined in terms of itself.

Functions in C may be used recursively i.e, a function may call itself either directly or indirectly.

It is a convenient and elegant programming technique when used with care.

Recursion - Example

```
#include <stdio.h>

/*  printd:  print n in decimal  */

void printd  (int  n)
{

        if  (n  /  10)
                printd  ( n  /  10);
        putchar (n  % 10 +  '0');
}
```

Note that each invocation of printd gets a fresh set of all automatic variables