Advanced Topics in C

Variable-length argument lists

- Possible to have functions that take variable number of arguments
- Best exemplified by `printf`
  - `printf` must receive a string as its first argument, but can receive any number of additional arguments
  - The prototype for `printf` is specified as
    ```c
    int printf ( const char *, ... );
    ```
- The ellipsis in the function `printf` indicates that the function receives a variable number of arguments of any type
- The ellipsis cannot be placed in the middle of the parameter argument list (it must be at the end of the list)

Using command line arguments or program parameters

- Defined by specifying parameters with the function `main`
- Useful in specifying command-line parameters for programs
- Possible to send any number of parameters to the program through the use of just two parameters
  1. `argc`
     - Number of arguments (argument count)
     - Declared as `int`
  2. `argv[]`
     - Entire command line (argument vector)
     - Declared as `char *` (character array)
     - `argv[0]` points to the command itself
     - `argv[1]` points to the first argument (program parameter)
     - `argv[2]` points to the second argument (program parameter)
     - `...`
      
      Notes:
      * Each array element `argv[i]` is a pointer to a character
      * Name of character constant and string arrays are also pointers to characters
      * An array of pointers can be regarded as a pointer to the first array element, which itself is a pointer
      * The name `argv` is a pointer to a pointer and can also be written as `**argv`

- Program to compute factorial

```c
#include <stdio.h>

int fact ( int ); /* Function prototype */

int main ( int argc, char *argv[] )
{
    int x; /* Variable to hold parameter */
    if ( argc != 2 )
    {
        printf ( "Usage: %s <number>\n", argv[0] );
        exit ( 1 );
    }
    x = atoi ( argv[1] );
    printf ( "Factorial of %d is %d\n", x, fact(x) );
}
```
return ( 0 );
}

/******************************************************************************/
/* fact.c */
int fact ( int x )
|
| return ( x ? ( x * fact ( x - 1 ) ) : 1 );
|

/******************************************************************************/

Automatic variables

- Variables local or internal to a program block
- So called since the memory is automatically allocated upon entrance to the block and released upon exit
- No meaning outside the block in which they are declared
- Valid only within the scope of the variable
- Can be declared either at the beginning of a function or at the beginning of a block
- Allow for efficient use of memory
- Local scope prevents the inadvertent change of value in other functions
- Can be declared with the keyword register for fast access
  - Declared as
    register int n;
    register char c;
  - May be assigned to internal CPU registers
  - Only automatic variables and formal parameters can be assigned as registers; no pointers (registers have no address)

External variables and program structure

- External to the module in which they are used
- Allows for permanent allocation of memory

/******************************************************************************/
#include <stdio.h>
extern int counter;  /* Declaration of counter */
extern void inc_counter ( void );
main()
|
| 
| int index;
| for ( index = 0; index < 10; index++ )
| inc_counter();
| printf ( "Counter is: %d\n", counter );
|

/******************************************************************************/
int counter;  /* Definition of counter */
void inc_counter ( void );
|
| 
| counter++;
Advanced Topics

• External variable has only one definition but can have several declarations
  – Declaration specifies the attributes of a variable
  – Definition specifies the attributes of a variable as well as allocates memory
    definition = declaration + memory allocation

• External variables are defined outside the functions (contrast with automatic variables)
• The keyword `extern` in function declarations can be omitted without causing any problems

Pointers to functions

• Functions stored in memory much the same way as data
  – Functions have addresses that can be assigned to pointers
  – Notation for these pointers is cryptic and we have to be extremely careful with the parentheses and the unary operator *

• If `p` is a pointer to a function that has no parameters, a call to that function can be written as

  \((\ast p)()\)

• The parentheses in \((\ast p)\) cannot be omitted since in \(\ast p()\), the parentheses have higher precedence than \(\ast\) and the function is read as

  \(\ast(p())\)

  which is incorrect and denotes the object pointed to by the value returned by \(p\), assuming that \(p\) is a function returning a pointer

• The parentheses surrounding \(\ast p\) also occur in declarations and prototypes

• Example of usage

```c
/* pfunc.c : Illustrating a pointer to a function */
#include <stdio.h>
#include <ctype.h>
#include <math.h>
main()
{
    char ch;
    double (*p)( double );
    printf ("Enter c, s, or t to select one of the functions cos, sin, tan: ");
    switch ( ch = tolower ( getchar () ) )
    {
        case 'c': p = cos; break;
        case 's': p = sin; break;
        case 't': p = tan; break;
        default : printf ( "Wrong character\n" ); exit ( 1 );
    }
    printf ( "Argument Function value\n" );
    printf ( "%8.3f %12.8f\n", 0.1, (*p)(0.1) );
    printf ( "%8.3f %12.8f\n", 0.3, (*p)(0.3) );
    printf ( "%8.3f %12.8f\n", 1.6, (*p)(1.6) );
}
```
Useful in passing the function name as a parameter to other functions

- In numerical analysis, a general integration routine is called with the interval boundaries and the function to be integrated as arguments

Example: Compute the function

\[ \text{sum} = f(1) + f\left(\frac{1}{2}\right) + f\left(\frac{1}{3}\right) + \ldots + f\left(\frac{1}{n}\right) \]

```c
#include <stdio.h>

#define fn_sum ( int n, double (*fn)() )

main()
{
    printf ( "Sum of squares: %f\n", fn_sum( 5, square ) );
    printf ( "Sum of cubes : %f\n", fn_sum( 5, cube ) );
}
```

```c
double fn_sum ( int n, double (*fn)() )
{
    double s = 0;
    int i;
    for ( i =1; i <= n; s += (*fn)(1.0/i++) );
    return ( s );
}
```

```c
/* square.c */
double square ( double x )
{
    return ( x * x );
}
```

```c
/* cube.c */
double cube ( double x )
{
    return ( x * x * x );
}
```

In-memory format conversion

- Allows the conversion from internal binary format to display (decimal) format without putting it onto screen or file
• Resulting character sequence appears as a string variable
• Achieved by the function *sprintf*
• The following two statements are equivalent, using the declaration `int i = 123; char str[8];`

\[
\text{sprintf ( str, ”%4d”, 2*i );}
= \text{strcpy ( str, ” 246” );}
\]

• The target string variable should have enough space to accommodate the output string
• The target variable (`str` in above example) could be a pointer in the middle of character array (string)
• *sscanf* function
  – Same as *scanf* with in-memory scanning of string
  – Better substitute for *scanf* when used in conjunction with *fgets*