```
    scanf ( "%d", &x );

    printf ( "Factorial of %d is %d\n", x, fact(x) );
}
/*****************************************************************************/
/*  fact.c : Function to compute factorial of a function                     */
/*****************************************************************************/

int fact ( int x )
{
    return ( x ? ( x * fact ( x - 1 ) ) : 1 );
}
```

```
###############################################################################
#
# Makefile for the program fact
#
CC = gcc
FLAGS = -g # For debugging

fact:   main.o fact.o
        $(CC) $(FLAGS) -o fact main.o fact.o

main.o: main.c
        $(CC) $(FLAGS) -c main.c

fact.o: fact.c
        $(CC) $(FLAGS) -c fact.c

clean:
        /bin/rm -f fact *.o core
```

- Two important cases in any recursive program
  1. Base case
  2. Recursive case
- Another example – Fibonacci series

Recursion vs. iteration

   - Any variable local to a block (also, default behavior for local variables)
   - Can be explicitly declared as

```
auto int x, y;
```

2. `register`
   - Automatic storage duration
   - Variables created upon entrance to the block, and exist as long as the block is active, and then, destroyed
   - Used to store a highly used variable in the hardware register (as opposed to memory)
   - Defined by

```
register int counter = 1;
```

   - The directive `register` may be ignored by the compiler
   - Can only be used with storage class `auto`
3. `extern`
   - Static storage duration
   - Default for global variables and function names
   - Useful to declare global variables in functions that are compiled in separate files (linkage of variables)
   - May result in unexpected side effects (bad; so avoid it)
   - Could be declared in the header files as

```
extern double foo;
```

4. `static`
   - Static storage duration
   - Local in scope but the memory contents are preserved for the next call to the function (or entrance to the block)
   - Defined by

```
static int fubar;
```

Recursion

- Exemplified by functions that call themselves ina systematic manner
- One of the most complex, most powerful, and most useful paradigms in computer science
- First, an example – computing the factorial of a number
  - The factorial of n is given by

$$n! = n \times (n-1)!$$

```
/*************************************************************************/
/*  main.c : Driver function for factorial function                      */
/*************************************************************************/

#include <stdio.h>

int fact ( int );

main ()
{
    int x;

    printf ( "Please enter a number: " );
```

```
        ${CC} ${DEBUGFLAG} -I${INCLUDES} -c jpeg.c

br_oct.o: br_oct.c br_oct.h
        ${CC} ${DEBUGFLAG} -c br_oct.c

dct.o:  dct.c br_oct.h
        ${CC} ${DEBUGFLAG} -c dct.c

sav_jpg_c.o: sav_jpg_c.c
        ${CC} ${DEBUGFLAG} -c sav_jpg_c.c ${LIBS}

clean :
        /bin/rm -f core *.o
```

- Dependency checking (using above `Makefile` as an example)
  - First target to be built is `jpegc`
  - `make` checks to see if `jpegc` exists in the current directory
  - If it exists, `make` checks `jpeg.o`, `br_oct.o`, `dct.o`, and `sav_jpg_c.o` to see whether any of them are newer than `jpegc`
  - The process in the above step is repeated for each of the `*.o` files as targets, checking their dependencies, using later entries in `Makefile`
  - Only after all the prerequisites have been verified and brought up-to-date, `make` will exit making sure that the file `jpegc` is current
  - Order of checking dependencies is important, and chain of commands must be issued in the correct order
  - It should also be apparent by now why the target name (to the left of the colon) is the same as the filename resulting from the execution of a command
- Minimizing rebuilds
  - If some object files are used in multiple programs, they need to be compiled only once
- Invoking `make`
  - Any target in the `Makefile` can be built by using the command

    ```
    make target
    ```

  - Several targets can be specified in a single invocation of `make`

    ```
    make jpeg.o dct.o
    ```

  - If target is not specified, `make` attempts to make the first available target in the `Makefile`

Storage classes

- An attribute of an identifier just like the scope
- Helps to determine the identifier's storage duration, scope, and linkage
- Four storage classes available indicated by *storage class specifiers*
  1. `auto`
     - Automatic storage duration
     - Variables created upon entrance to the block, and exist as long as the block is active, and then, destroyed

```
target :   dep1 dep2 dep3 ...   # Dependency line
           cmd                  # Command line
```

1. Dependency lines
   - Also known as *rule lines*
   - Show the dependencies between files
   - Exactly one line long; use of backslash as the last character is allowed to extend the line
   - The dependencies must be satisfied, possibly by building them as targets in other entries, to build the current target
   - A single target may appear on multiple dependency lines; however, only one of them can be associated with commands
2. Command lines
   - Commands to be executed to compile the files
   - Must be on a single line but the use of backslash as the last character of the line allows extended lines
   - Must begin with a tab as the first character on line[1]
   - tabs in the `Makefile` can be checked by the command

     `cat -v -t -e Makefile`
     - `-v` and `-e` options cause all tabs to appear as `^I`
     - `-e` option places a dollar sign at the end of each line to enable you to see any terminating whitespace
   - Multiple commands can be specified on successive lines to be executed in sequence
   - Multiple commands can be specified on same line if they are separated by a semicolon from each other

- Comments can be specified by the # sign and extend to the end of line
- Blank lines
  - Allowed in some places
  - Should not separate a dependency line from its commands
  - Should not separate lists of commands
- Targets without prerequisites
  - Must have a colon
  - Need not be filenames and if so, are always executed
    - `make` treats every non-existent target as out-of-date target
- A simple `Makefile`

```
CC = gcc
INCLUDES = /v3/sanjiv/include
PROJ_LIB = /v3/sanjiv/lib
LIB_OBJS = ${PROJ_LIB}/ras.o ${PROJ_LIB}/round.o
LIBS =  -lm
DEBUGFLAG = -g
#DEBUGFLAG = -O

jpegc:  jpeg.o br_oct.o dct.o sav_jpg_c.o
        ${CC} ${DEBUGFLAG} -o jpegc ${JPEG_OBJS} ${LIB_OBJS} ${LIBS}

jpeg.o: jpeg.c jpeg.h
```

[1] Most important syntax rule of make

- The `main` function appears in only one file, conventionally known as `main.c`
- Advantages
  - Reduction in complexity of organizing the program
  - Reduction in time required for compilation
    - Each file is compiled separately
    - Only those files are recompiled that have been modified
- Compiler creates object files corresponding to each source file
- The object files are linked together to create the final executable
- Compilation time is reduced because linking is much faster than compilation
- Source files are compiled separately under Unix using the `-c` option to the compiler

```
gcc -c main.c
```

- The entire sequence of commands to create an executable can be specified as

```
gcc -c main.c
gcc -c func.c
gcc -o prog main.o func.o
```

The Make Utility

- Utility to aid in the design and maintenance of multiple-file programs
- Relieves the programmer of the burden of remembering which files need recompilation
  - Examines the date of modification of files to see if they have changed
  - Recompiles the files if above is true
- Requires the use of a special file (called description file) in the directory, called `Makefile` or `makefile`
- Examines `Makefile` to determine the dependencies needed to build the executables
  - Sorts out dependency relations among files
  - Automatically recompiles only those files that have been changed after building the executable, and links them into the executable
- Based on non-procedural programming techniques, such as backtracking used in logic programming languages

The Makefile

- Used to make a *target* of the operation, using *prerequisites* or *dependents*; the dependents may have other files as prerequisites
- Sensitive to dependencies such as

$$\begin{array}{lcl} \text{source} & \Rightarrow & \text{object} \\ \text{object} & \Rightarrow & \text{executable} \end{array}$$

- Consists of two main types of lines described in the following general syntax

- Used to change the *seed* of the random number generator
- Initially seeded with a value 1 but can be changed at any time to a new seed value
- Program with a seed

```
#include   <stdio.h>

main()
{
    int i, j;

    srand ( 120101 );                 /* Seed the random number */

    for ( j = 0; j < 10; j++ )
    {
        i = ( rand() % 6 ) + 1;
        printf ( "The random number is: %d\n", i );
    }

    exit ( 0 );
}
```

- Still problem – Cannot change seed every time
- Use the `time` function
  - `time` returns the time since 00:00:00 GMT, January 1, 1970, measured in seconds

```
#include   <stdio.h>

main()
{
    int i, j;

    srand ( time ( NULL ) );                 /* Seed the random number */

    for ( j = 0; j < 10; j++ )
    {
        i = ( rand() % 6 ) + 1;
        printf ( "The random number is: %d\n", i );
    }

    exit ( 0 );
}
```

Makefiles

Independent compilation

- Large programs are difficult to maintain
- Problem solved by breaking the program into separate files
- Different functions placed in different files

```
#include   <stdio.h>

main()
{
    int i;

    i = rand();

    printf ( "The random number is: %d\n", i );

    exit ( 0 );
}
```

- Problem – Not truly random; Same number every time
- Pseudo random numbers
- Let us try a sequence of random numbers by modifying the program

```
#include   <stdio.h>

main()
{
    int i, j;

    for ( j = 0; j < 10; j++ )
    {
        i = rand();
        printf ( "The random number is: %d\n", i );
    }

    exit ( 0 );
}
```

- Scaling – Generating a random number within a specified range
- Generating a random number between 1 and 6 (rolling a dice)

```
#include   <stdio.h>

main()
{
    int i, j;

    for ( j = 0; j < 10; j++ )
    {
        i = ( rand() % 6 ) + 1;
        printf ( "The random number is: %d\n", i );
    }

    exit ( 0 );
}
```

- The `srand` function

Data type	printf	scanf
long double	%Lf	%Lf
double	%f	%lf
float	%f	%f
unsigned long int	%lu	%lu
long int	%ld	%ld
unsigned int	%u	%u
int	%d	%d
short	%hd	%hd
char	%c	%c

Header files

- Standard header files provided by the system
- How to find those
- Custom designed header files
- All header files end with a `.h` extension
- User-defined header files can be included with a pair of double quotes

```
#include "functions.h"
```

Calling functions: Call by value and Call by reference

- Protocols to pass parameters to the functions
- Dictate whether the parameter values get modified or not
- Call by value
  - The function creates its own local variables and copies the values in the passed parameters to those variables
  - Changes in the local variables do not affect the contents of the variables that are passed
  - It is useful as it prevents the accidental *side effects*
- Call by reference
  - Instead of creating the copy (or extra variable), the function takes the memory address of each parameter and assigns those as the address of the local variables
  - Any changes made to the value of the variable are reflected back into the calling functions
  - May lead to side effects that make the job of debugging harder
- C only allows you to pass parameters using call by value, but provides a mechanism to simulate call by reference

Random number generation – Using the C library functions

- First attempt: Copying the one from the book

- What is the return value type of `main()`?
- Parameter list
  - List of identifiers to describe the parameters received by the function
  - Use the keyword `void` in parameter list in the function definition if no parameters to be passed
  - Every identifier must have its type stated before it
  - Omission of type makes the compiler assume an `int` for the type of the parameter
  - Declaration like `float x, y` must be avoided
    - If both `x` and `y` are expected to be `float` types, change the above to `float x, float y`
    - As it stands, the declaration means `float x, int y`
  - The parameter list must not be terminated by a semicolon
  - A variable in the parameter list must not be redefined in the body of the function
- Function body
  - Defined by the compound statement within the curly braces
  - C does not allow the definition of a function within another function
  - **Most important rule of thumb**
    - **A function should be no longer than one page (about 20 lines of code)**
  - Try to fit the function header into one line
- The function prototype, function header, and function calls should all agree in the number, type, and order of arguments and parameters, and in the type of return value

Function prototypes

- Declaration of the function (not definition) to assist the compiler in making sure that the function is being used as intended
- The prototypes for the C standard library are also required and given to the compiler by including the header files with the `#include` preprocessor directive
- You can also design your own header files (a good idea)
- Parameter names are not required in the prototype declarations but parameter types are required
- If parameter names are mentioned in the prototype declarations, they are ignored by the compiler
- A semicolon is required at the end of the prototype declaration
- Scope of the function prototype declaration
- The argument types that may not match are *coerced*
  - Coercion is not the same as promotion but does follow promotion rules
  - Changing `double` to `int`
- Promotion hierarchy for data types

```
#include  <stdio.h>

int read_add ( int a, int b )
{
    int x;              /* Value of the variable */
    char line[80];      /* Buffer for input      */

    printf ( "Enter the next value of x (0 to terminate): " );
    fgets ( line, sizeof(line), stdin );

    if ( ( sscanf(line, "%d", &x) ) )
        return ( x ? ( a * x + b ) : 0 );
    else
    {
        printf ( "You entered a non-integer value; program being terminated\n");
        return ( 0 );
    }
}
```

- `void` functions
  - Equivalent to subroutines or procedures
  - Do not return a value
  - Function to print an error message

```
void pr_error ( int err_no )
{
    switch ( err_no )
    {
    case 1 :
        printf ( "Attempt to divide by zero.\n" );
        break;
    case 2 :
        printf ( "Attempt to read non-integer when integer expected.\n" );
        break;
    case 3 :
        printf ( "Attempt to evaluate square root of a negative number.\n" );
        break;
    default :
        printf ( "Unknown error occurred\n" );
        break;
    }
    printf ( "Program terminated\n" );
}
```

- Cautions
  - *Omitting the return-value-type in a function definition causes a syntax error if the function prototype specifies a return value type other than* `int`
  - *Forgetting to return a value from a function that is supposed to return a value can lead to unexpected errors. The ANSI standard states that the result of this omission is undefined.*
  - *Returning a value from a function whose return type has been declared* `void` *causes a syntax error.*

  - Akin to the `STOP` statement in Fortran subroutines
  - Can also be written as `return x` but preferable to write it with parentheses

- Function prototype
  - Declaration to be used by the compiler (not definition of function)
- Functions with parameters
  - Evaluate the function $\sum(ax + b)$ for given values of x when a and b are given as integers. The program terminates when a zero is given for the value of x

```
/*****************************************************************************/
/*                                                                           */
/* Program to compute the value of sum ( ax + b ) for some given values      */
/*     of x, when a and b are fixed                                          */
/*                                                                           */
/*****************************************************************************/

/* main.c                                                                    */

#include <stdio.h>

#define A 5      /* First coefficient of the polynomial */
#define B 7      /* Second coefficient of the polynomial */

int read_add ( int, int );

main()
{
    int i,          /* for temporary result       */
        x,          /* to get the first value     */
        s;          /* to keep summation          */
    char line[80];  /* Buffer for line            */

    printf ( "Enter a value of x (0 to terminate): " );
    fgets ( line, sizeof (line), stdin );
    sscanf ( line, "%d", &x );

    if ( !x )
        return;     /* Stop program */

    s = A * x + B;  /* Initialize s with first value of x */

    while ( i = ( read_add ( A, B ) ) )
        s += i;

    printf ( "The polynomial is evaluated to %d\n", s );
}

/*****************************************************************************/
/*                                                                           */
/* read_add.c                                                                */
/*                                                                           */
/*****************************************************************************/
```

  - Other functions are directly or indirectly called from `main`

- Syntax of user-defined function definitions

```
type fn_name ( parameters ) { ... }
```

- Functions must be defined (or declared) before they are used
- The name of the function can be any valid C identifier
- Example program

```
/* Sum the list of integer numbers typed in by user     */
/* Limitation: Cannot include 0 in the list             */

#include <stdio.h>

int read_int_no ();

main()
{
    int i, s = 0;
    printf ( "Please type some integers (not zero), followed by a CTRL-D on a \
line by itself\n" );
    while ( i = read_int_no() )      /* Call the function */
        s += i;
    printf ( "The sum is %d\n", s );
}

int read_int_no ( void )
{
    int x;
    char ch;
    char line[80];
    if ( ( ch = getchar() ) == EOF )
        return ( 0 );
    else
    {
        ungetc ( ch, stdin );
        fgets ( line, sizeof(line), stdin );
        sscanf ( line, "%d", &x );
        return ( x );
    }
}
```

  - Two functions: `main` and `read_int_no`
  - `read_int_no`
    * Called by `main` (not *used* by `main`)
    * Returns an integer value
    * Indicated by writing the keyword `int` before the function name
  - `return` statement
    * Explicitly declares a value to be returned
    * Logically the last statement to be executed in the function

Scope of variables

- Area of the program where the variable has a valid existence
- Global variable
  - Valid everywhere in the program
  - Not preferable to have too many of them
  - Permanent in nature
    - Created and initialized before the program is started
    - Remain in existence until the program terminates
- Local variable
  - Limited scope within the block enclosed by the `{ }`
  - Variable not valid outside the block
  - Temporary in nature
    - Allocated from the *stack*
    - Allocation of too many local variables may result in a "stack overflow" or "out of memory" error
    - Initialized upon entrance to the block
    - Also known as automatic variables because the space for them is allocated automatically
  - Can be made permanent by declaring them as `static`
    - Even then, the scope of the variables remain unchanged
    - The effect is that the `static` variables are not reinitialized upon entrance to the block
- What if a variable is declared twice, once as global and once as local to a block
  - The local definition takes precedence over the global value
- Example

Statement	$\texttt{global}_i$	`local`	`very_local`	$\texttt{global}_c$
`int global;  /* Global variable */`	√	⊘	⊘	⊘
`main()`	√	⊘	⊘	⊘
`{`	√	⊘	⊘	⊘
`  int local;  /* Local variable */`	√	√	⊘	⊘
`  global = 1;`	√	√	⊘	⊘
`  local = 2;`	√	√	⊘	⊘
`  {`	√	√	⊘	⊘
`  int very_local;`	√	√	√	⊘
`  char global;`	⊙	√	√	√
`  very_local = global + local;`	⊙	√	√	√
`  }`	√	√	⊘	⊘
`}`	√	⊘	⊘	⊘

Function definition

- The function `main`
  - Essential part of every C program
  - Point where the program starts the execution

Functions

Divide and conquer

- Problems in developing large programs
- Break the large program into manageable modules
- Useful to design, implement, and maintain large programs

Program modules in C

- Known as *functions*
- You have already used some prepackaged functions from the C standard library to develop programs
- ANSI C standard library to make programs portable
- Programmer-defined functions
  - Written by a programmer and used many times over
  - The statements in the function are hidden from other functions, or are private to the function
- Functions *invoked* by a *function call*
  - Function call specifies function name and a set of arguments
  - The function returns a value to the calling function
- Software reusability

Math library functions

- Available in the standard C library
- Any program using such functions should include the file `math.h` by using the statement

```
#include <math.h>
```

- When compiling, the command will specify the math library functions as

```
gcc -o foo foo.c -lm
```

Functions

- Abstraction and software reusability
- Packaging code as a function allows us to
  - Avoid repeating code in a program
  - Recycle the code
  - Execute certain code by simply calling the function from different points in the code
  - Minimize errors in the code
  - Develop code faster by putting multiple persons in the development of software, and giving them precise subtasks