Function Definition

type function_name ( parameter list ) { declarations statements}

For example

int factorial(int n)
{
    int i, product = 1;
    for (i = 2; I <= n; ++i)
        product *= i;
    return product;
}
void print_course(void)
{
    printf("%s\n%s\n%s\n%s\n%s\n\n",
            "    *************************",
            "    ***Tel Aviv University***",
            "    *** Computer Sciences ***",
            "    *** Programming Proj. ***",
            "    *************************");
}

Usage:
print_course()
The return statement

The `return` statement may or may not include an expression:

```
return expression;       // return
return;                 // return
```

For example:

```
return;
return ++a;
return (a * b);
```

The return statement terminates the execution of the function. There can be more than one return statement in a function.

```c
double abs_value( double x )
{
    if ( x >= 0.0 )
        return x;
    else
        return -x;
}
```
The return statement

Type is converted, if necessary to the type of the function. For example

```c
float f(char a)
{
    int i;
    ...
    return i; /* converted to float */
}
```
Function Prototypes

- Function should be declared before used.
- A function prototype tells the compiler the type of the value returned by the function as well as the number and types of its parameters.

\[ \text{\texttt{type function\_name ( parameter type list )}} \]

For example:
\[ \text{\texttt{double sqrt( double );}} \]

Identifiers are optional, and have no affect on the prototype. So
\[ \text{\texttt{void f(char c, int i);}} \]

is equivalent to:
\[ \text{\texttt{void f(char, int);}} \]
/* create a table of powers */
#include <stdio.h>
#define N 7

/* prototypes */
long power( int, int );
void prn_heading( void );
void prn_tbl_of_powers( int );

int main(void)
{
    prn_heading();
    prn_tbl_of_powers( N );
    return 0;
}
Functions Example

```c
void prn_heading( void ) {
    printf( "\n::: A TABLE OF POWERS :::\n\n" );
}

long power( int m, int n) {
    int i = 0;
    long product = 1;

    for ( i = 1; i <= n; ++i )
        product *= m;
    return product;
}
```
Functions Example

```c
void prn_tbl_of_powers( int n )
{
    int   i = 0, j = 0;
    for ( i = 1; i <= n; ++i )
    {
        for ( j = 1; j <= n; ++j )
        {
            if ( j == 1 )
                printf( "%ld", power( i, j ) );
            else
                printf( "%9ld", power( i, j ) );
        }
        putchar( '\n' );
    }
}
```

:::::  A TABLE OF POWERS  :::::

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>8</td>
<td>16</td>
<td>32</td>
<td>64</td>
<td>128</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>9</td>
<td>27</td>
<td>81</td>
<td>243</td>
<td>729</td>
<td>2187</td>
</tr>
</tbody>
</table>
...

An alternate style for function definition

```
#include <stdio.h>
#define N 7

long power(int m, int n)
{
  ...
}

void prn_heading( void )
{
  ...
}

void prn_tbl_of_powers(int n)
{
  ...

  printf( "%ld", power( i, j ) );
  ...
}

int main(void)
{
  prn_heading();
  prn_tbl_of_powers( N );
  return 0;
}
```
#include <stdio.h>

int compute_sum( int n )

int main(void)
{
    int n = 3, sum = 0;

    printf( "%d\n", n );
    sum = compute_sum( n );
    printf("%d\n", n );
    printf("%d\n", sum );
    return 0;
}

/* sum integers from 1:n */
int compute_sum( int n )
{
    int sum = 0;

    for ( ; n > 0; --n )
    {
        sum += n;
        sum += n;
    }

    return sum;
}
Developing large programs

- main.c
  - include “pgm.h”
- fct.c
  - include “pgm.h”
- prn.c
  - include “pgm.h”

The diagram shows the includes, defines, and prototypes for the files main.c, fct.c, and prn.c, which all include the file pgm.h.
```c
#include "pgm.h"
int main( void ) {
    char ans = 0;
    int i = 0, n = N;
    printf( "%s",
        "This program does not do very much.\n" "Do you want more information? ");
    scanf( " %c", &ans );
    if ( ans == 'y' || ans == 'Y' )
        prn_info( "pgm" );
    for ( i = 0; i < n; ++i )
        fctl1( i );
    printf( "Bye!\n" );
    return 0;
}
```
#include <stdio.h>
#include <stdlib.h>

#define N 3

void fct1(int k);
void fct2(void);
void prn_info(char *);
#include "pgm.h"

void fct1( int n )
{
    int i = 0;
    printf("Hello from fct1()\n");
    for ( i = 0; i < n; ++i )
        fct2();
}

void fct2( void )
{
    printf( "Hello from fct2()\n" );
}
```c
#include "pgm.h"
void prn_info( char *pgm_name )
{
    printf( "Usage: %s\n\n", pgm_name );
    printf( "%s\n",
        "This program illustrates how one can write a program\n" "in more than one file. In this example, we have a\n" "single .h file that gets included at the top of our\n" "three .c files. Thus the .h file acts as the "glue"\n" "that binds the program together.\n" "\n"
        "Note that the functions fct1() and fct2() when called\n" "only say "hello." Writing a serious program, the\n" "programmer sometimes does this in a first working\n" "version of the code.\n" );
}
```
Compiling and linking

```bash
gcc -c main.c fct.c wrt.c
gcc -o pgm.exe main.o fct.o wrt.o
```

We will learn in the sequel to handle such projects using the `make` utility.
Assertions

A macro that helps to ensure that the value of an expression is what it is supposed to be.

```c
#include <assert.h>
#include <stdio.h>
int f( int a, int b );

int main( void )
{
    int a = 0, b = 0, c = 0;
    ....
    scanf( "%d%d", &a, &b );
    ....
    c = f( a,b );
    assert( c > 0 ); /* an assertion */
    ....
}
```
int f( int a, int b )
{
    ..... 
    assert( a == 1 || a == -1 );
    assert( b >= 7 && b <= 11 );
    ..... 
}
Scope Rules

• Each block introduces its own identifiers.
• An outer block definition is valid unless an inner block redefines it.
• Inner blocks may be nested to arbitrary depths, limited by system limitations.

```c
{  
  int a = 2, b = 2, c = 3;
  printf( "%d\n", a );  /* 2 */
  {
    int a = 5;
    printf("%d\n", a);  /* 5 */
  }
  printf("%d\n", ++a);  /* 3 */
}
Scope Rules - example

```c
{
    int a = 1, b = 2, c = 3;
    printf ( "%3d%3d%3d\n", a, b, c );
{
    int b = 4;
    float c = 5.0;
    printf( "%3d%3d%5.1f\n", a, b, c );
    a = b;
    {
        int c;
        c = b;
        printf ( "%3d%3d%3d\n", a, b, c );
    }
    printf( "%3d%3d%5.1f\n", a, b, c );
}
    printf( "%3d%3d%3d\n", a, b, c );
}
```
Storage Classes

- auto
- extern
- register
- static
This is the default storage class. Can be made explicit (but seldom done so):

```c
auto int a, b, c;
auto float f;
```
External variables are used for transmitting information across blocks and functions.

When a variable is declared outside a function, storage is permanently assigned to it, and its storage class is `extern`.

Variables defined outside a function have external storage type even if not explicitly stated as `extern`.

Some old C compilers complain when `extern` is used.
#include <stdio.h>
int a = 1, b = 2, c = 3; /* global variables */
int f(void);

int main(void)
{
    printf( "%3d\n", f() );
    printf( "%3d%3d%3d\n", a, b, c );
    return 0;
}

int f( void )
{
    int b = 0, c = 0; /* b and c are local!! */
    /* global b and c are masked */
    a = b = c = 4;
    return (a + b + c);
}
extern - example - cont.

file1.c
#include <stdio.h>
int a = 1, b = 2, c = 3; /* global variables */
int f(void);

int main(void)
{
    printf( "%3d\n", f() );
    printf( "%3d%3d%3d\n", a, b, c );
    return 0;
}

file2.c
int f( void )
{
    extern int a; /* look for it elsewhere */
    int b = 0, c = 0; /* b and c are local!! */
    /* global b and c are masked */
    a = b = c = 4;
    return (a + b + c); }

Storage class `register` tells the compiler that the associated variables are to be stored in high speed memory registers.

```c
{
    register int i;
    for ( i = 0; i < LIMIT; ++i ) {
        ...
    }
} /* block exit will free the register */
```
The main use of `static` is to allow a local variable to retain its previous value when the block is reentered.

```c
void f(void)
{
    static int cnt = 0;

    ++cnt;
    if ( cnt % 2 == 0 )
        .... /* do something */
    else
        .... /* do something else */
}
```
Static external variables

This is a more subtle use of `static`. It allows privacy mechanism for external variables. Static external variables are scope-restricted external variables: Their scope is restricted ONLY to the remainder of the file in which they are defined.

```c
void f(void)
{
    ... /* v is not available here */
}
static int v;

void g(void)
{
    ....... /* v is available here */
}
```
#define INITIAL_SEED     17
#define MULTIPLIER     25173
#define INCREMENT     13849
#define MODULUS      65536
#define FLOATING_MODULUS  65536.0

static unsigned seed = INITIAL_SEED; /*external, 
      but private to this file */

unsigned random( void )
{
    seed = (MULTIPLIER * seed + INCREMENT) % MODULUS;
    return seed;
}

double probability(void)
{
    seed = ( MULTIPLIER * seed + INCREMENT ) % MODULUS;
    return ( seed / FLOATING_MODULUS );
}
Recursion

A function is said to be *recursive* if it calls itself, directly or indirectly. In C all functions can be used recursively.

```c
int factorial ( int n ) /* recursive version */ {
    if ( n <= 1 )
        return 1;
    else
        return ( n * factorial( n - 1 ) );
}

int factorial( int n ) /* iterative version */ {
    int product = 1;
    for( ; n > 1; --n )
        product *= n;
    return product;
}
```
write a line backwards

```c
#include <stdio.h>
void wrt_it(void);
int main(void)
{
    printf("Input a line:   ");
    wrt_it();
    printf("\n\n");
    return 0;
}

void wrt_it(void)
{
    int c = 0;
    if ( ( c = getchar() ) != '\n' )
        wrt_it();
    putchar( c );
}
```
double power( double val, unsigned int pow )
{
    if ( pow == 0 )
        return( 1.0 ); /* pow(x,0) returns 1 */
    else
        return( power( val, pow - 1 ) * val );
}
int fib(int n) {
    if (n <= 1)
        return n;
    else
        return (fib(n - 1) + fib(n - 2));
}

<table>
<thead>
<tr>
<th>Value of n</th>
<th>Number of invocations of fib</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>67</td>
</tr>
<tr>
<td>16</td>
<td>3193</td>
</tr>
<tr>
<td>32</td>
<td>7049155</td>
</tr>
</tbody>
</table>
Iterative (Non Recursive) Solution

```c
int fib(int n) {
    int i, fn, fn1 = 0, fn2 = 1;
    if( n == 0 | n == 1)
        return n;
    else {
        for ( i = 2; i <= n; ++i ) {
            fn = fn1 + fn2;
            fn1 = fn2;
            fn2 = fn;
        }
        return fn
    }
}
```
Top Down Recursion

/* Outputs an integer comma separated */
void int_comma( int num )
{
    if ( num < 1000 )
        printf( "%3d", num );
    else
    {
        int_comma( num / 1000 );
        printf( ",%03d", num % 1000 );
    }
}
Every instance of a function execution (call) creates an Activation Record, (frame) for the function. Activation records hold required execution information for functions:

- Parameters
- Return memory address, (calling instruction address)
- Dynamic link - pointer to activation record of calling function
- Return value for the function
- Local variables
Activation records are created and stored in an area of memory termed the runtime stack.
Storage Organization

- The program stack grows each time a function call is made.
- Infinite recursion results in a collision between the runtime stack and the heap termed a runtime stack overflow error.
- Illegal pointer de-references (garbage, dangling-references) often result in memory references outside the operating system allocated partition, (segment) for the C program resulting in a segmentation error (GPF - access violation) and core dump.
```c
int x; /* static storage */
void main() {
    int y; /* dynamic stack storage */
    char *str; /* dynamic stack storage */
    str = malloc(10); /* dynamic heap storage */
    y = foo(23);
    free(str); /* deallocates heap storage */
}
/* y and str deallocated when stack frame is popped */
int foo(int z) {
    /* z is dynamic stack storage */
    char ch[100]; /* ch is dynamic stack storage */
    if (z == 23) foo(7);
    return 3;
}
```
Last example: Hanoi Towers

Move the disks from peg A to peg C, so that no disk lies on a smaller disk
Hanoi Towers - cont.

```c
#include "hanoi.h" /* has all includes, externs and function prototypes*/
int    cnt = 0; /* count the number of moves */
int main(void)
{
    int    n; /* number of disks */
    scanf("%d", &n);
    assert(n > 0);
    move(n, 'A', 'B', 'C'); /* recursive fct */
    return 0;
}
```
Hanoi Towers - cont.

```c
#include "hanoi.h"
void move(int n, char a, char b, char c) {
    if (n == 1) {
        ++cnt;
        printf("%5d: %s%d%s%c%s%c.\n", cnt,
                "Move disk ",1, " from tower ",a, " to tower ",c);
    } else {
        move(n - 1, a, c, b);
        ++cnt;
        printf("%5d: %s%d%s%c%s%c.\n", cnt,
                "Move disk ",n, " from tower ", a, " to tower ", c);
        move(n - 1, b, a, c);
    }
}
```