Functions
Function

- A program segment that carries out some specific, well-defined task

Example

- A function to add two numbers
- A function to find the largest of n numbers

A function will carry out its intended task whenever it is called or invoked

- Can be called multiple times
Every C program consists of one or more functions

One of these functions must be called `main`

Execution of the program always begins by carrying out the instructions in `main`

Functions call other functions as instructions
void print_banner ()
{
    printf("************\n");
}

void main ()
{
    ... print_banner ();
    ... print_banner ();
}
Calling function (**caller**) may pass information to the called function (**callee**) as parameters/arguments

- For example, the numbers to add

The callee may return a single value to the caller

- Some functions may not return anything
void main()
{
    float cent, fahr;
    scanf("%f", &cent);
    fahr = cent2fahr(cent);
    printf("%fC = %fF\n", cent, fahr);
}

float cent2fahr(float data)
{
    float result;
    result = data * 9/5 + 32;
    return result;
}
float cent2fahr(float data)
{
    float result;
    printf("data = %f\n", data);
    result = data*9/5 + 32;
    return result;
    printf("result = %f\n", result);
}
void main()
{
    float cent, fahr;
    scanf("%f", &cent);
    printf("Input is %f\n", cent);
    fahr = cent2fahr(cent);
    printf("%fC = %fF\n", cent, fahr);
}
int factorial (int m)
{
    int i, temp=1;
    for (i=1; i<=m; i++)
        temp = temp * i;
    return (temp);
}

void main()
{
    int n;
    for (n=1; n<=10; n++)
        printf ("%d! = %d \n", n, factorial (n) );
}

Output

1! = 1
2! = 2
3! = 6 ........ upto 10!
Why Functions?

- Allows one to develop a program in a modular fashion
  - Divide-and-conquer approach
  - Construct a program from small pieces or components
- Use existing functions as building blocks for new programs
- Abstraction: hide internal details (library functions)
Defining a Function

- A function definition has two parts:
  - The first line, called header
  - The body of the function

```plaintext
return-value-type function-name (parameter-list)
{
    declarations and statements
}
```
The first line contains the return-value-type, the function name, and optionally a set of comma-separated arguments enclosed in parentheses.

- Each argument has an associated type declaration.
- The arguments are called formal arguments or formal parameters.

The body of the function is actually a block of statement that defines the action to be taken by the function.
```c
int gcd (int A, int B)
{
    int temp;
    while ((B % A) != 0) {
        temp = B % A;
        B = A;
        A = temp;
    }
    return (A);
}
```
Return value

- A function can return a value
  - Using `return` statement
- Like all values in C, a function return value has a type
- The return value can be assigned to a variable in the caller

```c
int x, y, z;
scanf("%d%d", &x, &y);
z = gcd(x, y);
printf("GCD of %d and %d is %d\n", x, y, z);
```
Function Not Returning Any Value

- **Example**: A function which prints if a number is divisible by 7 or not

```c
void div7 (int n)
{
    if ((n % 7) == 0)
        printf ("%d is divisible by 7", n);
    else
        printf ("%d is not divisible by 7", n);
    return;
}
```

Optional

Return type is void
return statement

- In a value-returning function (result type is not void), `return` does two distinct things
  - specify the value returned by the execution of the function
  - terminate that execution of the callee and transfer control back to the caller
- A function can only return one value
  - The value can be any expression matching the return type
  - but it might contain more than one return statement.
- In a void function
  - return is optional at the end of the function body.
  - return may also be used to terminate execution of the function explicitly.
  - No return value should appear following return.
void compute_and_print_itax ()
{
    float income;
    scanf ("%f", &income);
    if (income < 50000) {
        printf ("Income tax = Nil\n");
        return;
    }
    if (income < 60000) {
        printf ("Income tax = %f\n", 0.1*(income-50000));
        return;
    }
    if (income < 150000) {
        printf ("Income tax = %f\n", 0.2*(income-60000)+1000);
        return;
    }
    printf ("Income tax = %f\n", 0.3*(income-150000)+19000);
}

Terminate function execution before reaching the end
Calling a function

- Called by specifying the function name and parameters in an instruction in the calling function
- When a function is called from some other function, the corresponding arguments in the function call are called actual arguments or actual parameters
  - The function call must include a matching actual parameter for each formal parameter
  - Position of an actual parameters in the parameter list in the call must match the position of the corresponding formal parameter in the function definition
  - The formal and actual arguments must match in their data types
void main ()
{
    double x, y, z;
    char op;
    ...
    z = operate (x, y, op);
    ...  
}  

double operate (double x, double y, char op) 
{
    switch (op) {
    case ‘+’ : return x+y+0.5 ;
    case ‘~’  : if (x>y)
               return x-y + 0.5;
               return y-x+0.5;
    case ‘x’ : return x*y + 0.5;
    default : return –1;
    }
}
When the function is executed, the value of the actual parameter is copied to the formal parameter.
Another Example

/* Compute the GCD of four numbers */
void main()
{
    int n1, n2, n3, n4, result;
    scanf("%d %d %d %d", &n1, &n2, &n3, &n4);
    result = gcd( gcd (n1, n2), gcd (n3, n4) );
    printf ("The GCD of %d, %d, %d and %d is %d \n", n1, n2, n3, n4, result);
}
void main()
{
    int numb, flag, j=3;
    scanf("%d",&numb);
    while (j <= numb)
    {
        flag = prime(j);
        if (flag==0)
            printf("%d is prime
",j);
        j++;
    }
}

int prime(int x)
{
    int i, test;
    i=2, test =0;
    while ((i <= sqrt(x)) && (test ==0))
    {
        if (x%i==0) test = 1;
        i++;
    }
    return test;
}
void main()
{
    int numb, flag, j=3;
    scanf("%d",&numb);
    printf("numb = %d\n",numb);
    while (j <= numb)
    {
        printf("Main, j = %d\n",j);
        flag = prime(j);
        printf("Main, flag = %d\n",flag);
        if (flag == 0)
            printf("%d is prime\n",j);
        j++;
    }
}

int prime(int x)
{
    int i, test;
    i = 2; test = 0;
    printf("In function, x = %d \n",x);
    while ((i <= sqrt(x)) && (test == 0))
    {
        if (x%i == 0) test = 1;
        i++;
    }
    printf("Returning, test = %d \n",test);
    return test;
}
The output

5
numb = 5
Main, j = 3
In function, x = 3
Returning, test = 0
Main, flag = 0
3 is prime
Main, j = 4
In function, x = 4

Returning, test = 1
Main, flag = 1
Main, j = 5
In function, x = 5
Returning, test = 0
Main, flag = 0
5 is prime
Points to note

- The identifiers used as formal parameters are “local”.
  - Not recognized outside the function
  - Names of formal and actual arguments may differ

- A value-returning function is called by including it in an expression
  - A function with return type T (≠ void) can be used anywhere an expression of type T can be used
Returning control back to the caller

- If nothing returned
  - return;
  - or, until reaches the last right brace ending the function body

- If something returned
  - return expression;
Function Prototypes

- Usually, a function is defined before it is called
  - `main()` is the last function in the program written
  - Easy for the compiler to identify function definitions in a single scan through the file

- However, many programmers prefer a top-down approach, where the functions are written after `main()`
  - Must be some way to tell the compiler
  - Function prototypes are used for this purpose
    - Only needed if function definition comes after use
Function prototypes are usually written at the beginning of a program, ahead of any functions (including `main()`)

Prototypes can specify parameter names or just types (more common)

Examples:

```c
int gcd (int , int );
void div7 (int number);
```

- Note the semicolon at the end of the line.
- The parameter name, if specified, can be anything; but it is a good practice to use the same names as in the function definition
Some more points

- A function cannot be defined within another function
  - All function definitions must be disjoint
- Nested function calls are allowed
  - A calls B, B calls C, C calls D, etc.
  - The function called last will be the first to return
- A function can also call itself, either directly or in a cycle
  - A calls B, B calls C, C calls back A.
  - Called recursive call or recursion
Example: main calls ncr, ncr calls fact

```c
int ncr (int n, int r);
int fact (int n);

void main()
{
    int i, m, n, sum=0;
    scanf ("%d %d", &m, &n);
    for (i=1; i<=m; i+=2)
        sum = sum + ncr (n, i);
    printf ("Result: %d \n", sum);
}

int  ncr (int n, int r)
{
    return (fact(n) / fact(r) / fact(n-r));
}

int  fact (int n)
{
    int i, temp=1;
    for (i=1; i<=n; i++)
        temp *= i;
    return (temp);
}
```
Local variables

- A function can define its own local variables
- The locals have meaning only within the function
  - Each execution of the function uses a new set of locals
  - Local variables cease to exist when the function returns
- Parameters are also local
/* Find the area of a circle with diameter d */
double circle_area (double d) {
    double radius, area;
    radius = d/2.0;
    area = 3.14*radius*radius;
    return (area);
}
Revisiting $nCr$

```c
int fact(int x)
{
    int i, fact = 1;
    for (i = 2; i <= x; ++i) fact = fact * i;
    return fact;
}

int ncr(int x, int y)
{
    int p, q, r;
    p = fact(x);
    q = fact(y);
    r = fact(x - y);
    return p / (q * r);
}
```

```c
void main()
{
    int n, r;
    scanf("%d%d", &n, &r);
    printf("n=%d, r=%d, nCr=%d\n", n, r, ncr(n, r));
}
```

The variable $x$ in function `fact` and $x$ in function `ncr` are different.

The values computed from the arguments at the point of call are copied on to the corresponding parameters of the called function before it starts execution.
Scope of a variable

- Part of the program from which the value of the variable can be used (seen)
- Scope of a variable - Within the block in which the variable is defined
  - Block = group of statements enclosed within { }
- Local variable – scope is usually the function in which it is defined
  - So two local variables of two functions can have the same name, but they are different variables
- Global variables – declared outside all functions (even main)
  - scope is entire program by default, but can be hidden in a block if local variable of same name defined
#include <stdio.h>
int A = 1;
void main()
{
    myProc();
    printf ( "A = %d
", A);
}

void myProc()
{
    int A = 2;
    if ( A==2 )
    {
        int A = 3;
        printf ( "A = %d
", A);
    }
    printf ( "A = %d
", A);
}

Output:
A = 3
A = 2
A = 1
Parameter Passing: by Value and by Reference

- Used when invoking functions
- **Call by value**
  - Passes the value of the argument to the function
  - Execution of the function does not change the actual parameters
    - All changes to a parameter done inside the function are done on a copy of the actual parameter
    - The copy is removed when the function returns to the caller
    - The value of the actual parameter in the caller is not affected
  - Avoids accidental changes
Call by reference

- Passes the **address** to the original argument.
- Execution of the function may affect the original
- Not directly supported in C except for arrays
### Parameter passing & return: 1

```c
void main()
{
    int a=10, b;
    printf ("Initially a = %d\n", a);
    b = change (a);
    printf ("a = %d, b = %d\n", a, b);
}

int change (int x)
{
    printf ("Before x = %d\n",x);
    x = x / 2;
    printf ("After x = %d\n", x);
    return (x);
}
```

**Output**

Initially a = 10
Before x = 10
After x = 5

a = 10, b = 5
```c
void main()
{
    int x=10, b;
    printf (“M: Initially x = %d\n”, x);
    b = change (x);
    printf (“M: x = %d, b = %d\n”, x, b);
}

int change (int x)
{
    printf (“F: Before x = %d\n”,x);
    x = x / 2;
    printf (“F: After x = %d\n”, x);
    return (x);
}
```

**Output**

M: Initially x = 10
F: Before x = 10
F: After x = 5
M: x = 10, b = 5
void main()
{
    int x=10, b;
    printf ("M: Initially x = %d\n", x);
    x = change (x);
    printf ("M: x = %d, b = %d\n", x, x);
}

int change (int x)
{
    printf ("F: Before x = %d\n", x);
    x = x / 2;
    printf ("F: After x = %d\n", x);
    return (x);
}

Output

M: Initially x = 10
F: Before x = 10
F: After x = 5
M: x = 5, b = 5
void main()
{
    int x=10, y=5;
    printf ("M1:  x = %d, y = %d\n", x, y);
    interchange (x, y);
    printf ("M2:  x = %d, y = %d\n", x, y);
}

void interchange (int x, int y)
{
    int temp;
    printf ("F1:  x = %d, y = %d\n", x, y);
    temp= x; x = y; y = temp;
    printf ("F2:  x = %d, y = %d\n", x, y);
}

Output

M1:  x = 10, y = 5
F1:  x = 10, y = 5
F2:  x = 5, y = 10
M2:  x = 10, y = 5

How do we write an interchange function? (will see later)
Passing Arrays to Function

- Array element can be passed to functions as ordinary arguments
  - IsFactor (x[i], x[0])
  - sin (x[5])
Passing Entire Array to a Function

- An array name can be used as an argument to a function
  - Permits the entire array to be passed to the function
  - The way it is passed differs from that for ordinary variables

- Rules:
  - The array name must appear by itself as argument, without brackets or subscripts
  - The corresponding formal argument is written in the same manner
    - Declared by writing the array name with a pair of empty brackets
const int ASIZE = 5;
float average (int B[]) 
{
    int i, total=0;
    for (i=0; i<ASIZE; i++)
        total = total + B[i];
    return ((float) total / (float) ASIZE);
}

void main ( ) {
    int x[ASIZE] ; float x_avg;
    x = {10, 20, 30, 40, 50};
    x_avg = average (x) ;
}
void main()
{
    int  n;
    float  list[100], avg;
    
    avg  =  average (n, list);
    
}

float  average  (int a, float x[])
{
    
    sum = sum + x[i];

}
void VectorSum (int a[ ], int b[ ], int vsum[ ], int length) {
    int i;
    for (i=0; i<length; i=i+1)
        vsum[i] = a[i] + b[i] ;
}

void PrintVector (int a[ ], int length) {
    int i;
    for (i=0; i<length; i++) printf ("%d ", a[i]);
}

void main () {
    int x[3] = {1,2,3}, y[3] = {4,5,6}, z[3];
    VectorSum (x, y, z, 3) ;
    PrintVector (z, 3) ;
}
The Actual Mechanism

- When an array is passed to a function, the values of the array elements are **not passed** to the function
  - The array name is interpreted as the *address* of the first array element
  - The formal argument therefore becomes a *pointer* to the first array element
  - When an array element is accessed inside the function, the address is calculated using the formula stated before
  - Changes made inside the function are thus also reflected in the calling program
Passing parameters in this way is called **call-by-reference**.

Normally parameters are passed in C using **call-by-value**.

Basically what it means?

- If a function changes the values of array elements, then these changes will be made to the original array that is passed to the function.
- This does not apply when an individual element is passed on as argument.
Passing 2-d Arrays as Parameters

- Similar to that for 1-D arrays
  - The array contents are not copied into the function
  - Rather, the address of the first element is passed
- For calculating the address of an element in a 2-d array, we need:
  - The starting address of the array in memory
  - Number of bytes per element
  - Number of columns in the array
- The above three pieces of information must be known to the function
Example Usage

```c
int main()
{
    int a[15][25], b[15][25];
    :
    :
    add (a, b, 15, 25);
    :
}
```

We can also write

```c
int x[15][25], y[15][25];
```

But at least 2\textsuperscript{nd} dimension must be given
Library Functions
Library Functions

- Set of functions already written for you, and bundled in a “library”
- Example: printf, scanf, getchar,
- C library provides a large number of functions for many things
- We look at functions for mathematical use
Math Library Functions

- Math library functions
  - perform common mathematical calculations
  - Must include a special header file
    ```c
    #include <math.h>
    ```

- Example
  ```c
  printf ("%f", sqrt(900.0));
  ```
  - Calls function `sqrt`, which returns the square root of its argument

- Return values of math functions can be float/double/long double

- Arguments may be constants, variables, or expressions
Math Library Functions

double acos(double x) – Compute arc cosine of x.
double asin(double x) – Compute arc sine of x.
double atan(double x) – Compute arc tangent of x.
double atan2(double y, double x) – Compute arc tangent of y/x.
double cos(double x) – Compute cosine of angle in radians.
double cosh(double x) – Compute the hyperbolic cosine of x.
double sin(double x) – Compute sine of angle in radians.
double sinh(double x) – Compute the hyperbolic sine of x.
double tan(double x) – Compute tangent of angle in radians.
double tanh(double x) – Compute the hyperbolic tangent of x.
Math Library Functions

double ceil(double x) – Get smallest integral value that exceeds x.
double floor(double x) – Get largest integral value less than x.
double exp(double x) – Compute exponential of x.
double fabs (double x) – Compute absolute value of x.
double log(double x) – Compute log to the base e of x.
double log10 (double x) – Compute log to the base 10 of x.
double pow (double x, double y) – Compute x raised to the power y.
double sqrt(double x) – Compute the square root of x.