Functions in C

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

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)

Function - Introduction

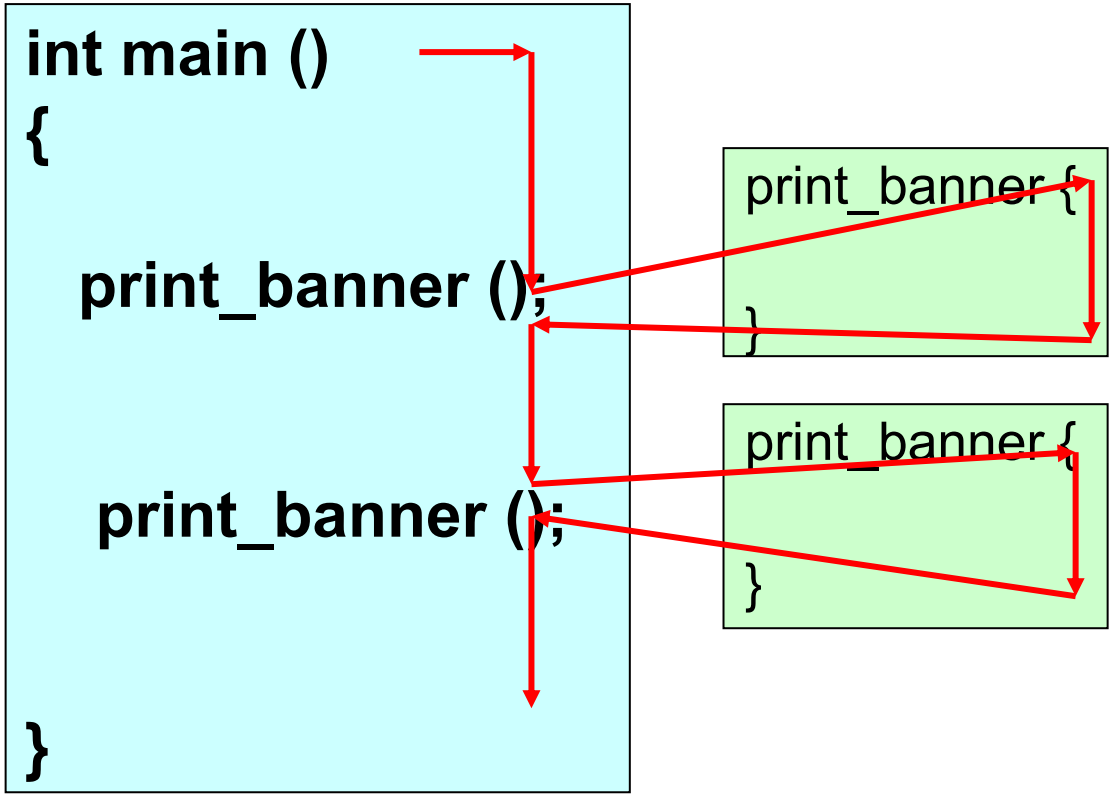
- Every C program consists of one or more functions
- One of these functions must be called main
- Execution of the program always begins by carrying out the instructions in main
- Functions call other functions as instructions

Function Control Flow

```
void print_banner ()
{
    printf("************\n");
}
```

```
int main ()
{
    . . .
    print_banner ();
    . . .
    print_banner ();

}
```



Function Control Flow

- Calling function (calling) may pass information to the called function (called) as parameters/arguments
  - For example, the numbers to add
- The called may return a single value to the caller
  - Some functions may not return anything

Calling function (Calling)

Called function (Called)

parameter

```
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;
}
```

Parameter passed

Returning value

Calling/Invoking the cent2fahr function

Defining a Function

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

```
return-value-type  function-name  ( parameter-list )
{
    declarations and statements
}
```

Defining a Function

- 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

Parameter passing

- When the function is executed, the value of the actual parameter is copied to the formal parameter

parameter passing

```
int main ()
{       . . .
        double circum;
        . . .
        area1 = area(circum);
        . . .
}
```

```
double area (double r)
{
        return (3.14*r*r);
}
```

Example of function definition

Return-value type

Formal parameters

```
int  gcd  (int A, int B)
{
    int  temp;
    while ((B % A) != 0)  {
        temp = B % A;
        B = A;
        A = temp;
    }
    return (A);
}
```

BODY

Value returned

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

```
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

```
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;
}
```

Return type is void

Optional

return statement

- In a value-returning function (return 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 function (Caller)

Called function (Callee)

parameter

```
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;
}
```

Parameter passed

Returning value

Calling/Invoking the cent2fahr function

How it runs

```
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);
}
```

Output

```
$ ./a.out
32
Input is 32.000000
data = 32.000000
32.000000C = 89.599998F

$./a.out
-45.6
Input is -45.599998
data = -45.599998
-45.599998C = -50.079998F
$
```

Function Prototypes

- 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:

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

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

Another Example

Function declaration (prototype)

```
int  factorial (int m);
```

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

Function call

```
int  factorial (int m)
{
    int i, temp=1;
    for (i=1; i<=m; i++)
          temp = temp * i;
    return (temp);
}
```

Function definition

Output

```
1! = 1
2! = 2
3! = 6  …….. upto 10!
```

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

Example

Formal parameters

```
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;
        }
}
```

Actual parameters

Local variables

Calling function (Calling)

Called function (Called)

parameter

```
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;
}
```

Parameter passed

Returning value

Calling/Invoking the cent2fahr function

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

Local variables

```
/* 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);
}
```

parameter

local variables

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

Points to note

- 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;

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

```
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);
}
```

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

Variable Scope

```
#include  <stdio.h>
int A = 1;
void main()
{
   myProc();
   printf ( "A = %d\n", A);
}

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

Global variable

Hides the global A

Output:

A = 3

A = 1

Compute GCD of two numbers

```
int main() {
    int  A, B, temp;
    scanf ("%d %d", &A, &B);
    if  (A > B)  {
        temp = A;  A = B;  B = temp;
    }
    while ((B % A) != 0)  {
        temp = B % A;
        B = A;
        A = temp;
    }
    printf ("The GCD is %d", A);
}
```

```
12 )  45  (  3
      36
       9  )  12  (  1
             9
             3  )  9  (  3
                   9
                   0
```

Initial: ***A=12, B=45***
Iteration 1: temp=9, B=12,A=9
Iteration 2: temp=3, B=9, A=3
B % A = 0 ➔ GCD is 3

Compute GCD of two numbers (with function)

```
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);
```

Return-value type

Formal parameters

```
int  gcd  (int A, int B)
{
    int  temp;
    while ((B % A) != 0)  {
        temp = B % A;
        B = A;
        A = temp;
    }
    return (A);
}
```

Value returned

BODY