Typical commercial systems are very large. When the size of a codebase increases the difficulties in avoiding errors, making changes and so on also increase significantly. By segmenting the program into clearly defined functions, we can address these problems, as well as improve reusability of code.

What do we mean by a well-defined function? "Do one task and not do similar but unrelated tasks" is one good maxim. We will note other guidelines as we go along.

Functions

A function is a piece of code that performs a particular task. Instead of writing this block of code inside a larger chunk of code, we give it a *good* name and put it into a function.

In other words, functions help us to split up a long program into named sections so that the section can be reused –in the same program as well as in other programs.

main()

Every C program consists of one special function called `main()`. Execution will always start from `main()`.

Categories Of Functions

C functions can be classified into two categories,

Built-in Functions These functions are also called as 'library functions'. Library functions are those functions which are defined by C library, or external ('third party') libraries, example `printf()`, `scanf()`, `strcat()` etc. You just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.

For example, all standard I/O functions are available in <*stdio.h* >. Mathematical functions are in <*math.h* >. String handling functions are in <*string.h* >.

Once again:

- whenever you want to use any library function in your program include the corresponding header file before `main()` and
- indicate the library required. For example, `c99 -lm compute.c`, compiles and links a program `compute.c` that uses math library functions.

User-defined Functions User-defined functions are those functions which are defined by the user at the time of writing the program. Such functions are made for code reusability and for saving time and space.

Declaration

Every function we use must be declared before it is used – as we have noted, the header files handle the declaration for built-in functions. The actual body of the function can be defined separately.

For functions we write, we can either add a declaration line at the top of the program or put the declaration in a header file of our own and include that header file. The second method is appropriate when we are writing a suite of related functions, the first is sufficient for most of our purposes in the early stages.

General syntax of function declaration is:

```
return_type function_name(parameter-list);
```

A function declaration is similar to a variable declaration. It tells the compiler about the function: its name, and how to call it. It also helps the compiler to detect wrongly written invocations.

A function declaration consists of 4 parts.

1. return-type
2. function name
3. parameter list
4. terminating semicolon

Definition

General syntax of function definition is:

```
return_type function_name (parameter-list) {
    function-body;
}
```

The first line known as function header and the statement(s) within braces are called *function body*.

return-type return type specifies the type of value (*int, float, char, double*) that the function is expected to return to the program calling the function. Some functions perform the desired operations without returning a value. In this case, the return type is the keyword void[1].

function-name function name specifies the name of the function. The function name is any valid C identifier and therefore must follow the same rules of naming as other variables in C. The function name and the parameter list together constitute the *function signature*.

parameter-list The parameter list refers to the type, order, and number of the parameters of a function. The parameter list declares the variables that will receive the data sent by calling program. They are often referred to as *formal parameters*. These are separated by commas.

function-body The function body contains the declarations and the statements necessary for performing the required task. The body is enclosed within curly braces { } and consists of three parts.

- local variable declaration,
- statements that perform the tasks of the function, and
- a **return** statement that returns the value evaluated by the function. (Or a empty **return** for a procedure).

Calling

Control of the program is transferred to user-defined function by *calling* it (invoking). Arguments are the values specified during the function call, for which the formal parameters are declared in the function. The number of arguments must match the number of formal parameters and must be of the same (or equivalent) data type.

Example

```
1 #include <stdio.h>
2 int sum(int, int);
3 int display();
```

[1]Technically such functions are known as *procedures*

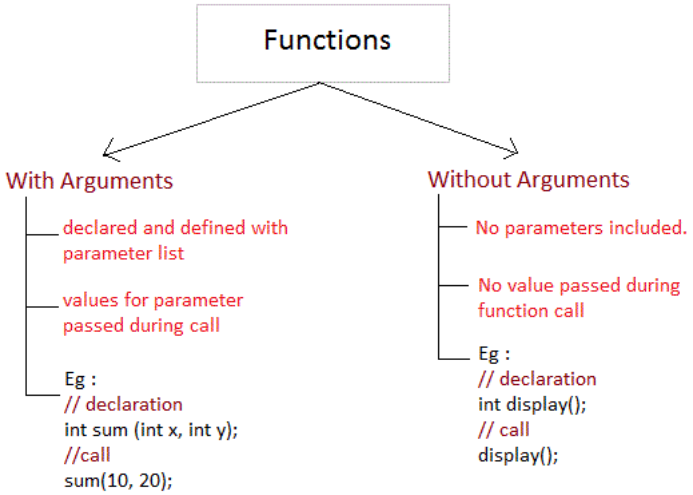

Figure 1: Invocation: With and Without Arguments

```
4
5  int sum(int x, int y) {
6      int add = x + y;
7      return add;
8  }
9
10 int display() {
11     int result = sum(10, 20);
12     return result;
13 }
14
15 int main() {
16     int total = display();
17     printf("sum of two numbers is %d\n", total);
18     return 0;
19 }
```



```
$
$ c99 Program-08-1.c
$ ./a.out
sum of two numbers is 30
$
```

Figure 2: Using Function

Benefits of using functions

- User defined functions help to decompose the large program into small segments. This makes programs easy to understand, maintain and debug.
- Easy code Reuseability. You just have to call the function by its name to use it.
- It provides modularity to the program.
- It reduces the size of a program and easy to understand the actual logic of a program.

Code Reading

Primes in Range

A program to Display Prime Numbers in a given range.

The following three step approach should be used a general guideline to think and plan tour programs.

1. Identify functions needed;
   - describe their purpose
   - determine what information they need for that
   - choose a name
2. Implement the functions
3. Put the functions together to do the whole task

In this case, we can see that we need a function to check if a given number is prime, we can simply run a loop over the range and call that function.

We are looking for an output like this:

```
$
$ c99 Program-08-2.c
$ ./a.out
Enter start of range: 15
Enter end  of range: 50
    17    19    23    29    31    37    41    43    47
Thats 9 primes between 15 and 50
$
```

Figure 3: Primes In Range

Now let us actually write the code, adding a main which accepts the range and calls the function.

```
#include <stdio.h>
#include <stdbool.h>

bool isPrime(int);
int PrimesInRange(int, int);

int main() {
    int start;
    int end;
    printf("Enter start of range: ");
    scanf("%d", &start);
    printf("Enter end  of range: ");
    scanf("%d", &end);
    int n = PrimesInRange(start, end);
    printf("\nThats %d primes between %d and %d\n", n, start, end);
    return 0;
}

int PrimesInRange(int a, int b) {
    int primeCount = 0;
    for (int n = a; n <= b; n++) {
        if (isPrime(n)) {
```



```
23                 primeCount++;
24                 printf("%6d", n);
25             }
26         }
27         return primeCount;
28     }
29
30     bool isPrime(int n) {
31         for (int i = 2; i < n; i++) {
32             if (n % i == 0) {
33                 return false;
34             }
35         }
36         return true;
37     }
```

