Functions, Arrays & Pointers

#Recursion *#Strings*

Functions

- A function is a block of code that performs a specific task. Functions can be in-built in C (standard library functions) or user-defined as required by a programmer.

- Standard library (C library) functions:
  - ✓ Are in-built to handle tasks such as mathematical computations, input/output (I/O) processing, string handling, etc.
  - ✓ Have their prototypes and data functionalities defined within appropriate header files.
  - ✓ For example, printf(), scanf(), fprintf(), getchar() are standard library functions defined in the stdio.h header file.
  - ✓ All standard library functions are available for use in a program after the appropriate header file(s) are included.
  - ✓ All C programs must have the main() function for proper execution.

- Library functions are advantageous because.
  - ✓ They work! They have been tested rigorously.
  - ✓ They are optimized for performance. They allow you to create the most efficient code optimized for maximum performance.
  - ✓ They save considerable development time. Don't reinvent the wheel by developing a function that performs a task that has already been coded.
  - ✓ The functions are portable. They will work everywhere any time.

User-defined Functions

- User-defined functions are developed/specified by the programmer according to their need.
- Just like standard library functions, user-defined functions are written as a block of code to perform a specific task.
- All functions (standard library and user-defined) are written based on a standard syntax (prototype) that specifies a function's name, parameters and return type.
  - ✧ The function prototype is just the declaration of the function that gives information to the compiler to indicate that the function would be used later on in the program.
  - ✧ Function prototypes follow the format:

    returnType functionName(dataType1 argument1, dataType2 argument 2, ... dataTypeN argumentN);
- All functions must have a function name and may/ may not have a return type and/or defined arguments.
  - ✧ A function name is an identifier and must be unique. After declaring a user-defined function as above, the inner workings of the function itself are defined within the main() under a defined lexical scope for the function.
  - ✧ Since execution of the program starts at the main(), when the compiler encounters a user-defined function within the main, control of the program jumps to the user function.
- User-defined functions are advantageous because:
  - ✓ They are easier to understand, maintain and debug; easily reusable; and allow for a large program to be divided into smaller modules.

Function Returns & Arguments

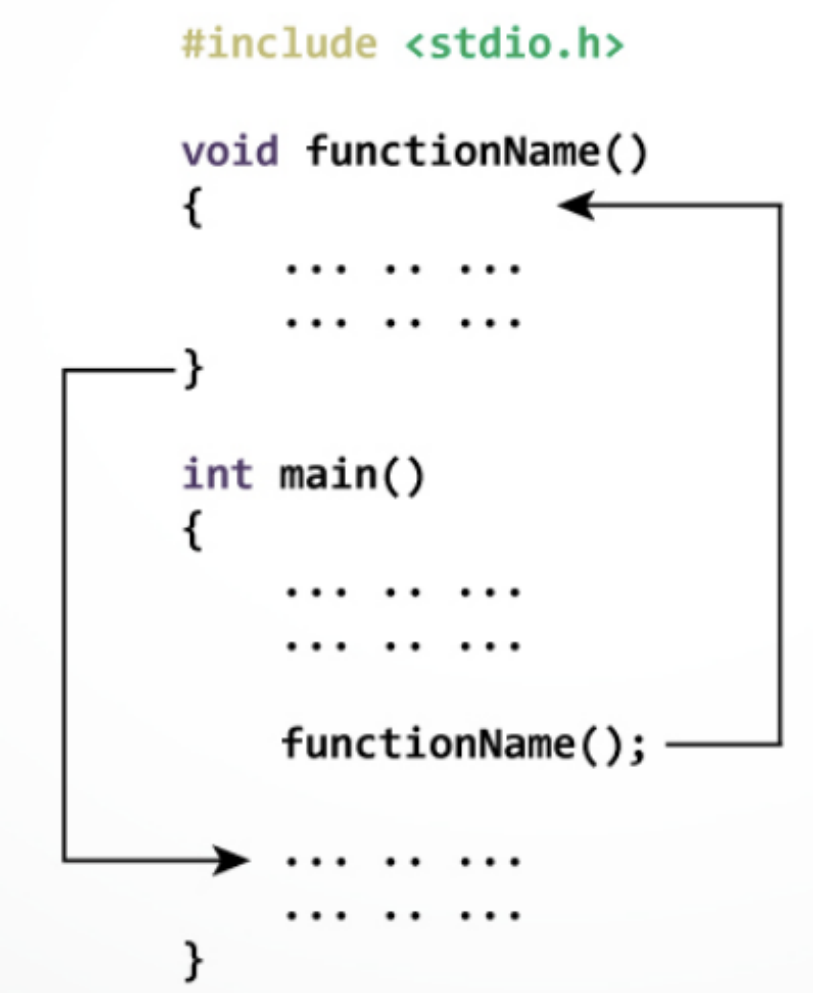
returnType functionName(dataType1 argument1, dataType2 argument 2, ... dataTypeN argumentN);

- Based on the above function prototype, one can define a function that gives the summation of numbers as:

  int sumNumbers(int number1, int number2, ... int numberN)

- Such a function provides the following information to the compiler:
  - ✓ Function name: *sumNumbers*
  - ✓ Function return type: int
  - ✓ Function arguments: several with the same data type of *int*
- Control of the program is transferred to the user-defined function by calling it. (To be discussed later).
- The operation specifications of a user-defined function are defined within the functions scope.
- Arguments in a function refer to the variable(s) that a function needs to operate on for a specified task.
  - ✓ The arguments passed to a function must match the data types defined in the function declaration, otherwise the compiler throws errors. A function can also be called without passing arguments.
- Once a function has executed, it returns a value to the calling function thus transferring program control back to the calling function.
  - ✓ The type of the value returned to the calling function must match that which was defined for the called (user-defined) function.

Function – Control Flow



- Execution of the program always starts with the ***main()*** function.
- When the compiler encounters ***functionName()*** within the ***main()***, control of the program jumps to ***void functionName().***
- The compiler then starts executing the code as specified under the user-defined function ***void functionName()***.
- Once all the code within the user-defined function has been executed, control then returns to the statement after ***functionName()*** within the main().

No Arguments & No Return Value Functions

```c
#include <stdio.h>

void checkPrimeNumber();

int main()
{
    checkPrimeNumber();     // argument is not passed
    return 0;
}

// return type of the function is void because function
void checkPrimeNumber()
{
    int n, i, flag=0;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0)
        {
            flag = 1;
        }
    }
    if (flag == 1)
        printf("%d is not a prime number.", n);
    else
        printf("%d is a prime number.", n);
}
```

- The return type of the function ***checkPrimeNumber()*** is void because the function does not return anything.
- The empty parenthesis of ***void checkPrimeNumber()*** indicates that the function has no specified arguments and within the ***main()***, the empty parenthesis indicate that no arguments are passed to ***checkPrimeNumber()***.

Return Type & No Arguments

```c
#include <stdio.h>
int getInteger();

int main()
{
     int n, i, flag = 0;

     n = getInteger();    // no argument is passed

     for(i=2; i<=n/2; ++i)
     {
          if(n%i==0){
               flag = 1;
               break;
          }
     }

     if (flag == 1)
          printf("%d is not a prime number.", n);
     else
          printf("%d is a prime number.", n);

     return 0;
}

int getInteger()          // returns integer entered by the user
{
     int n;

     printf("Enter a positive integer: ");
     scanf("%d",&n);

     return n;
}
```

- Again, within the ***main()***, the empty parenthesis on ***getInteger()*** shows that no arguments are passed to the function.
- However, the value returned by the function ***int getInteger()*** is assigned to *n.*

No Return Value

```c
#include <stdio.h>
void checkPrimeAndDisplay(int n);

int main()
{
     int n;

     printf("Enter a positive integer: ");
     scanf("%d",&n);

     // n is passed to the function
     checkPrimeAndDisplay(n);

     return 0;
}

// void indicates that no value is returned from
void checkPrimeAndDisplay(int n)
{
     int i, flag = 0;

     for(i=2; i <= n/2; ++i)
     {
          if(n%i == 0){
               flag = 1;
               break;
          }
     }
     if(flag == 1)
          printf("%d is not a prime number.",n);
     else
          printf("%d is a prime number.", n);
}
```

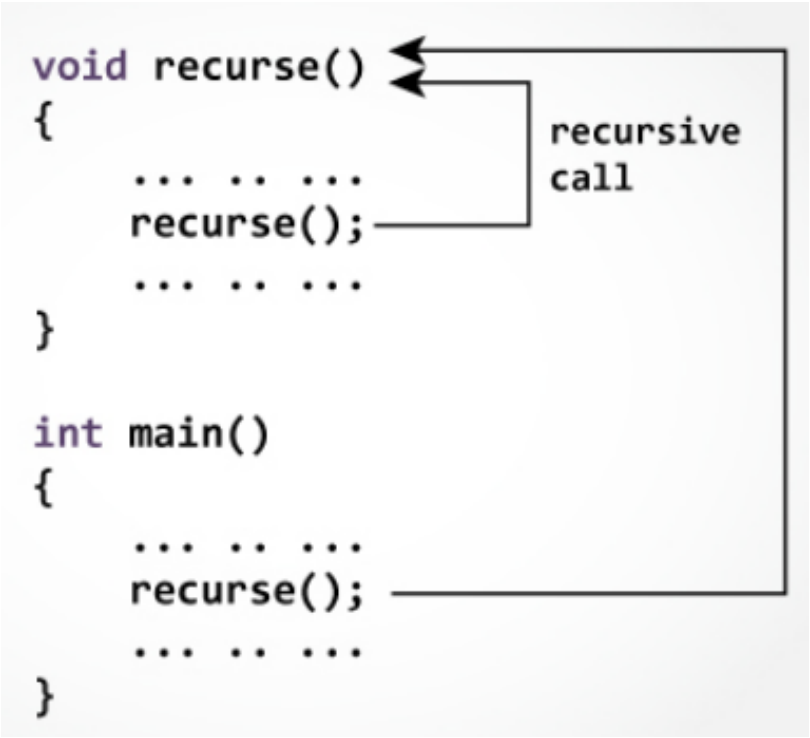
Return Value & Arguments

```c
#include <stdio.h>
int checkPrimeNumber(int n);

int main()
{
    int n, flag;

    printf("Enter a positive integer: ");
    scanf("%d",&n);

    // n is passed to the checkPrimeNumber() function
    // the value returned from the function is assigned
    flag = checkPrimeNumber(n);

    if(flag == 1)
        printf("%d is not a prime number",n);
    else
        printf("%d is a prime number",n);

    return 0;
}

// integer is returned from the function
int checkPrimeNumber(int n)
{
    int i;

    for(i=2; i <= n/2; ++i)
    {
        if(n%i == 0)
            return 1;
    }

    return 0;
}
```

- The input from the user is passed to ***checkPrimeNumber()*** and the function checks whether the number passed is a prime or not.
- If the passed argument is a prime number, the user-defined function returns a 0.
- If the passed argument is not a prime, the user-defined function returns a 1.
- The return value is then assigned to the *flag* variable.

Recursion

- A function that calls itself is called a recursive function, while the process is referred to as recursion.
- The layout below illustrates recursion concept using a function called *recurse()*.



- In this layout, when the *main()* calls *recurse()*, control of the program jumps to the function definition at *void recurse()*.
- The instructions specified under *recurse()* are executed until such a time when function calls itself at *recurse()*.
- Again, control of the program is returned to the function *void recurse()*.

Infinite Recursion

- Recursion can continue endlessly unless some condition to prevent it is set and met.
- To prevent infinite recursions, the *if . . . . . else* statement, (or any other decision making) can be used where a branch makes a recursive call and the other doesn't.

```c
#include <stdio.h>
int sum(int n);

int main()
{
     int number, result;

     printf("Enter a positive integer: ");
     scanf("%d", &number);

     result = sum(number);

     printf("sum = %d", result);
     return 0;
}

int sum(int num)
{
     if (num!=0)
          return num + sum(num-1); // sum()
     else
          return num;
}
```

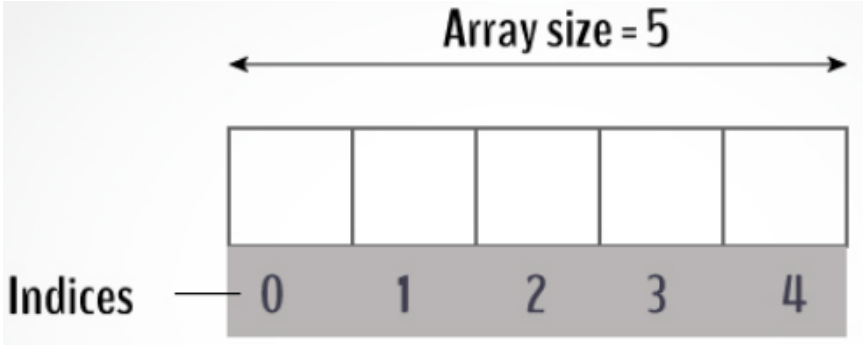
Output

```
Enter a positive integer:3
sum = 6
```

ARRAYS

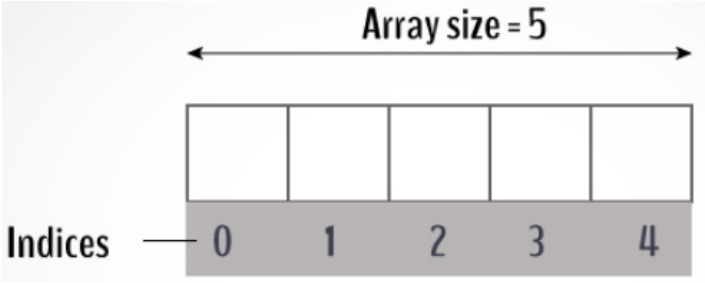
- An array is a collection of a fixed number of values of the same type.
- For example, if you wanted to store 10 names for different students who complained about their end of semester results, an array would hold the student's names in sequence.
- At declaration, an array must be specified with a data type and name. for example: *data_type array_name[array_size];*

  char names[10];
- The above is a declaration of an array called names with 10 elements all of the *char* data type.
- The size and type of an array cannot be changed after declaration.



Types of Arrays

- There are 2 types of arrays:
  1. One-dimensional arrays
  2. Multi-dimensional arrays



- Elements of an array are accessed using indices.
- Array elements are indexed starting from 0 to n-1, where n is the array size.
- In the example of an array of student names, *names[0]* would be the first student name while *names[9]* is the 10th student name.
- Suppose the starting address of *names[0]* is 2020a, then the next address location of 2052a would be for the second element *names[1]* and the next address of 2084a would be for *names[2]*, and so on. Every element of the array is of type char that occupies 32 bytes.
- When you attempt to access array elements out of their index bounds, your program may return very unacceptable results.

Arrays - Example

```c
// Program to find the average of n (n < 10) numbers

#include <stdio.h>
int main()
{
      int marks[10], i, n, sum = 0, average;
      printf("Enter n: ");
      scanf("%d", &n);
      for(i=0; i<n; ++i)
      {
            printf("Enter number%d: ",i+1);
            scanf("%d", &marks[i]);
            sum += marks[i];
      }
      average = sum/n;

      printf("Average = %d", average);

      return 0;
}
```

Output

```
Enter n: 5
Enter number1: 45
Enter number2: 35
Enter number3: 38
Enter number4: 31
Enter number5: 49
Average = 39
```

Passing Arrays

- You can pass a single element of an array or an entire array (be it one-dimensional or multi-dimensional) to a function.
- Passing a single element of an array is similar to passing a variable to a function.

```c
#include <stdio.h>
void display(int age)
{
    printf("%d", age);
}

int main()
{
    int ageArray[] = {2, 3, 4};
    display(ageArray[2]); //Passing array element ageArray[2]
    return 0;
}
```

Output

```
4
```

Passing Entire Arrays

```c
// Program to calculate average by passing an array to a function

#include <stdio.h>
float average(float age[]);

int main()
{
        float avg, age[] = {23.4, 55, 22.6, 3, 40.5, 18};
        avg = average(age); // Only name of an array is passed as an argument
        printf("Average age = %.2f", avg);
        return 0;
}

float average(float age[])
{
        int i;
        float avg, sum = 0.0;
        for (i = 0; i < 6; ++i) {
                sum += age[i];
        }
        avg = (sum / 6);
        return avg;
}
```

Output

```
Average age = 27.08
```

Passing Multi-dimensional Arrays

```c
#include <stdio.h>
void displayNumbers(int num[2][2]);
int main()
{
    int num[2][2], i, j;
    printf("Enter 4 numbers:\n");
    for (i = 0; i < 2; ++i)
        for (j = 0; j < 2; ++j)
            scanf("%d", &num[i][j]);

    // passing multi-dimensional array to a function
    displayNumbers(num);
    return 0;
}

void displayNumbers(int num[2][2])
{
    int i, j;
    printf("Displaying:\n");
    for (i = 0; i < 2; ++i)
        for (j = 0; j < 2; ++j)
            printf("%d\n", num[i][j]);
}
```

Output

```
Enter 4 numbers:
2
3
4
5
Displaying:
2
3
4
5
```

Strings as Arrays

- In C programming, a string is an array of characters terminated by a null character, *\0*.
- When the compiler encounters a sequence of characters enclosed in quotation marks, it appends a null character at the end of the string.
- For example, the string *"Student Names"* becomes *Student Names\0*.
- Since strings are by definition an array of characters, they are similarly declared the same way arrays are declared.
- For example, char c[5] could be a declaration for a variable *c* of type *char* with *5* elements.
- Here are some of the different ways for initializing strings:

```
char c[] = "abcd";

char c[50] = "abcd";

char c[] = {'a', 'b', 'c', 'd', '\0'};

char c[5] = {'a', 'b', 'c', 'd', '\0'};
```

String Input & Output

- *scanf()* and *printf()* can be used to handle string inputs and outputs.
- However, *scanf()* will read a sequence of characters until it encounters a white space (space, new line, tab, etc) after which everything else is discarded.

```c
#include <stdio.h>
int main()
{
    char name[20];
    printf("Enter name: ");
    scanf("%s", name);
    printf("Your name is %s.", name);
    return 0;
}
```

Output

```
Enter name: Dennis Ritchie
Your name is Dennis.
```

- Dennis Ritchie was the name entered but *scanf()* only input Dennis and discarded everything else after.

gets(), puts() & Passing Strings

- *gets()* instead of *scanf()* can be used to read strings while *puts()* can be used to display the string.
- Other commonly used string functions include: *strlen()* calculates the length of strings; *strcpy()* copies a string to another; *strcmp()* compares 2 strings; *strcat()* concatenates 2 strings.

```c
#include <stdio.h>
void displayString(char str[]);

int main()
{
    char str[50];
    printf("Enter string: ");
    gets(str);
    displayString(str);       // Passing
    return 0;
}
void displayString(char str[])
{
    printf("String Output: ");
    puts(str);
}
```

```c
#include <stdio.h>
int main()
{
    char name[30];
    printf("Enter name: ");
    gets(name);       // read string
    printf("Name: ");
    puts(name);       // display string
    return 0;
}
```

POINTERS (*)

- Pointers are used in C to access memory and manipulate addresses.
- This feature of C distinguishes from other popular languages such as java and python that provide no capability for such memory access and manipulation.
- As you know already, every variable has associated memory assigned to it to hold the value(s) of the variable.
- For example, for a variable *sum*, the use of *&sum* would give would give the address in memory for *sum*, where & is commonly referred to as the reference operator. You have applied this operator severally when using *scanf()*.

Pointers - Example

```c
#include <stdio.h>
int main()
{
   int var = 5;
   printf("Value: %d\n", var);
   printf("Address: %u", &var);
   return 0;
}
```

Output

```
Value: 5
Address: 2686778
```

Question:

Would you expect this program to return the same value for Address when run on your device?

NOTE:

var is just the name given to the address location 2686778

Pointer Variables

- A variable that stores the address (and not a value) is referred to as a *pointer variable* in C, or simply *pointer*.
- The syntax for creating a pointer is:

  *data_type *pointer_variable_name;*

  *int *var;*

- In the above statement, *var* is a pointer variable of type *int*.
- As previously mentioned, the & operator gives you the address of a variable, and is thus referred to as a reference operator.
- The * operator on the other hand, gets you the value from the address, and is thus referred to as the dereference operator.

Pointer - Example

```c
#include <stdio.h>
int main()
{
    int* pc, c;

    c = 22;
    printf("Address of c: %u\n", &c);
    printf("Value of c: %d\n\n", c);

    pc = &c;
    printf("Address of pointer pc: %u\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc);

    c = 11;
    printf("Address of pointer pc: %u\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc);

    *pc = 2;
    printf("Address of c: %u\n", &c);
    printf("Value of c: %d\n\n", c);
    return 0;
}
```

Output

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```

Pointers Explained

- From the previous pointer example,

```
int *pc, c;
```

- Declares a pointer *pc* and a regular variable *c*, all of type *int*.
- Since *pc* and *c* are not initialized, pointer pc points to either no address or a random address, while variable c has an address holding random data.
- The statement *c = 22* assigns 22 to the variable c and as such, the address of *c* now holds the value 22.
- The use of *%u* for addresses is because they are usually expressed as unsigned integers.

Pointers & Arrays

- Consider the program below and its output:

```c
#include <stdio.h>
int main()
{
    int x[4];
    int i;

    for(i = 0; i < 4; ++i)
    {
        printf("&x[%d] = %u\n", i, &x[i]);
    }

    printf("Address of array x: %u", x);

    return 0;
}
```

```
&x[0] = 1450734448
&x[1] = 1450734452
&x[2] = 1450734456
&x[3] = 1450734460
Address of array x: 1450734448
```

- There is a difference of 4 bytes between consecutive elements of the array because the int data type occupies 4 bytes.
- Note that *&x[0]* and *x* return the same result hence *x[0]* is equivalent to **x*.
- Similarly, *&x[1]* is equivalent to *x+1* hence *x[1]* is equivalent to **(x+1)*; *&x[2]* is equivalent to *x+2* hence *x[2]* is equivalent to **(x+2)* and so on.

Pointers and Arrays cont'd

- Basically, *&x[i]* is equivalent to *x+i* hence *x[i]* is equivalent to **(x+i)*.
- In most cases, array names decay (are converted) to pointers and that explains the ability to use the same pointer name as the name of the array to manipulate the elements of an array.
- Fundamentally however, pointers and arrays are not the same thing.
- There are cases where array names don't decay to pointers.

Strings & Pointers

- Just like arrays, strings also decay to pointers hence one can use the same string name for a pointer to manipulate the elements of a string.

```c
#include <stdio.h>

int main(void) {
   char name[] = "Harry Potter";

   printf("%c", *name);        // Output: H
   printf("%c", *(name+1));    // Output: a
   printf("%c", *(name+7));    // Output: o

   char *namePtr;

   namePtr = name;
   printf("%c", *namePtr);       // Output: H
   printf("%c", *(namePtr+1));   // Output: a
   printf("%c", *(namePtr+7));   // Output: o
}
```

Call by Reference

- Just as it is possible to pass variable values to a function, addresses too (which are also some kind of values) can also be passed to a function.
- Pointers support this capability within the function definition.
- Consider the example below and its output.

```
num1 = 10
num2 = 5
```

```c
#include <stdio.h>
void swap(int *n1, int *n2);

int main()
{
    int num1 = 5, num2 = 10;

    // address of num1 and num2 is passed
    swap( &num1, &num2);

    printf("num1 = %d\n", num1);
    printf("num2 = %d", num2);
    return 0;
}

// pointer n1 and n2 stores the address of
void swap(int* n1, int* n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

- ✓ The addresses of num1 and num2 are passed to swap() and pointers n1 and n2 store them.
- ✓ When the values and n1 and n2 are changed, num1 and num2 are also changed respectively.
- ✓ This technique is referred to as call by reference since addresses are passed to a function instead of variable values.