Problem Solving with C

Compiled by

M S Anand (anandms@pes.edu)

Text Book(s):

1. "How To Solve It By Computer", R G Dromey, Pearson, 2011.
2. "The C Programming Language", Brian Kernighan, Dennis Ritchie, 2nd Edition, Prentice Hall PTR, 1988.

Reference Book(s):

1. "Expert C Programming; Deep C secrets", Peter van der Linden
2. " The C puzzle Book", Alan R Feuer

Functions

A look at the sizes of typical software packages.

Software	# of lines of code
Windows 10	50 Million lines (approx)
Linux	12 Million (approx)
Facebook	62 Million
Google Chrome (browser)	6.7 Millions
Android OS	12 – 15 Million lines

Can you imagine the effort needed to develop and debug software of such sizes? It certainly cannot be implemented by any one person, it takes a team of programmers to develop such a project.

<u>Introduction</u>

Breaking up a program into reasonably self-contained units is basic to the development of any program of a practical nature. When confronted with a big task, the most sensible thing to do is break it up into manageable chunks. You can then deal with each small chunk fairly easily and you can be reasonably sure that you've done it properly. If you design the chunks of code carefully, you may be able to reuse some of them in other programs.

<u>One of the key ideas in the C language is that every program should be segmented into functions that are relatively short.</u> Even with the examples that you have seen so far that were written as a single main() function, other functions are inevitably involved because you have used a variety of standard library functions for input and output, for mathematical operations …

<u>What is a function?</u>

Making black boxes. Solving problems. Getting results.

A function is a module or block of program code which deals with a particular task. Making functions is a way of isolating one block of code from other independent blocks of code.

Functions serve two purposes. They allow a programmer to say: ‘this piece of code does a specific job which stands by itself and should not be mixed up with anything else’, and they make a block of code reusable since a function can be reused in many different contexts without repeating parts of the program text.

Why functions?

1. Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read, understand, and debug.
2. Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.
3. Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.
4. Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.

Any C program consists of one or more functions, the most important of which is the function main() where execution starts.

The program steps through the statements in sequence in the normal way until it comes across a call to a particular function. At that point, any argument values are transferred to the function and execution moves to the start of that function—that is, the first statement in the body of the function. Execution of the program continues through the function statements until it hits a **return** statement or reaches the closing brace marking the end of the function body. This signals that execution should go back to the point immediately after the point the function was originally called.

The set of functions that make up a program link together through the function calls and their return statements to perform the various tasks necessary for the program to achieve its purpose. In general, each function can be executed many times and can be called from several points within a program.

Variable Scope and Lifetime

In all the programs we have discussed so far, we have declared the variables for the program at the beginning of the block that define the body of the function main(). But you can actually define variables anywhere in the body of a function. Does this make a difference?

YES.

Variables exist only within the block in which they're defined. They're created when they are declared, and they cease to exist at the next closing brace. This is also true of variables that you declare within blocks that are inside other blocks. The variables declared at the beginning of an outer block also exist in the inner block. These variables are freely accessible, as long as there are no other variables with the same name in the inner block.

Variables that are created when they're declared and destroyed at the end of a block are called *automatic variables, because they're automatically created and destroyed. The extent within the program code where a given* variable is visible and can be referenced is called the *variable's scope. When you use a variable within its scope,* everything is okay. But if you try to reference a variable outside its scope, you'll get an error message when you compile the program because the variable doesn't exist outside its scope.

A sample code snippet:

```
{
        int a = 0;              // Create a
        // Reference to a is OK here
        // Reference to b is an error here - it hasn't been created yet
        {
                int b = 10; // Create b
                        // Reference to a and b is OK here
        }                       // b dies here
        // Reference to b is an error here - it has been destroyed
        // Reference to a is OK here
}       // a dies here
```

All the variables that are declared within a block are destroyed and no longer exist after the closing brace of the block. The variable **a** is visible within both the inner and outer blocks because it’s declared in the outer block. The variable **b** is visible only within the inner block because it’s declared within that block.

During program execution, a variable is created and memory is allocated for it when the statement that defines it is executed. For automatic variables the memory that the variable occupies is returned back to the system at the end of the block in which the variable is declared. Of course, while functions called within the block are executing, the variable continues to exist; it is only destroyed when execution reaches the end of the block in which it was created.

The time period during which a variable is in existence is referred to as the *lifetime of the variable*.

```
// Program - scoping out scope
#include <stdio.h>
int main(void)
{
        int count1 = 1; // Declared in outer block
        do
        {
                int count2 = 0; // Declared in inner block
                ++count2;
                printf("count1 = %d count2 = %d\n", count1, count2);
        } while( ++count1 <= 5);
        // count2 no longer exists
        printf("count1 = %d\n", count1);
        return 0;
}
```

```
// Program - More scope in this example
#include <stdio.h>
int main(void)
{
        int count = 0; // Declared in outer block
        do
        {
                int count = 0; // This is another variable called count
                ++count; // this applies to inner count
                printf("count = %d\n", count);
        }
        while( ++count <= 5); // This works with outer count
        printf("count = %d\n", count);
        // Inner count is dead, this is outer count
        return 0;
}
```

It’s not a good idea to do so, but you have used the same variable name, count, at the main() block level and in the loop block. Observe what happens when you compile and run this:

```
count = 1
count = 1
count = 1
count = 1
count = 1
count = 1
count = 6
```

How It Works

You have two variables called count, but inside the loop block the local variable will "hide" the version of count that exists at the main() block level. The compiler will assume that when you use the name count, you mean the one that was declared in the current block. Inside the do-while loop, only the local version of count can be reached, so that is the variable being incremented. The printf() inside the loop block displays the local count value, which is always 1.

As soon as you exit the loop, the outer count variable becomes visible, and the last printf() displays its final value from the loop as 6.

<u>**Variable Scope and Functions**</u>

The body of every function is a block (which may contain other blocks, of course). <u>As a result, the automatic variables you declare within a function are local to the function and don’t exist elsewhere. Therefore, the variables declared within one function are quite independent of those declared in another function or in a nested block.</u>

There’s nothing to prevent you from using the same name for variables in different functions; they will remain quite separate. Indeed, this is an advantage. It would be very hard to keep coming up with distinct names for all of the variables in a large program. It’s handy to be able to use the same name such as count in different functions. It’s still a good idea to avoid any unnecessary or misleading overlapping of variable names in your various functions, and, of course, you should try to use names that are meaningful to make your programs easy to follow.

Defining a Function

When you create a function, you specify the function header as the first line of the function definition, followed by the executable code for the function enclosed between braces. The block of code between braces following the function header is called the *function body.*

• The *function header defines the name of the function, the function parameters (which specify* the number and types of values that are passed to the function when it’s called), and the type for the value that the function returns.

• The *function body contains the statements that are executed when the function is called, and* these have access to any values that are passed as arguments to the function.

The general form of a function:

```
Return_type Function_name( Parameters - separated by commas )
{
// Statements...
}
```

The statements in the function body can be absent, but the braces must be present. If there are no statements in the body of a function, the return type must be void, and the function will have no effect. void means “absence of any type,” and here it means that the function doesn’t return a value.
A function that has statements in the function body but does not return a value must also have the return type specified as void.
Conversely, for a function that does not have a void return type, **every return statement** in the function body must return a value of the specified return type.

Defining a function with an almost content-free body is often useful during the testing phase of a complicated program, for example, a function defined so that is just contains a return statement, perhaps returning a default value. This allows you to run the program with only selected functions actually doing something; you can then add the detail for the function bodies step by step, testing at each stage, until the whole thing is implemented and fully tested.

Parameters and Arguments

The term *parameter refers to a* ***placeholder in a function definition*** *that specifies the type of value that should be* passed to the function when it is called. The value passed to a function corresponding to a parameter is referred to as an *argument.*

A function parameter consists of the type followed by the parameter name that is used within the body of the function to refer to the corresponding argument value that is transferred when the function is called.

Note

The statements in the body of a function can contain nested blocks of statements. However**, you can't define a function inside the body of another function**.

The general form for calling a function :
Function_name(List of Arguments - separated by commas)

Naming a Function

The name of a function can be any legal name in C that isn't a reserved word (such as int, double, sizeof, and so on) and isn't the same as the name of another function in your program. **You should ensure that you do not use the same names as any of the standard library functions because this would not only prevent you from using the library function, but would also be very confusing**. Of course, if you do use a library function name and include the header file for the function into your source file, your program will not compile.

A legal name has the same form as that of a variable: a sequence of letters and digits, the first of which must be a letter. As with variable names, the underline character counts as a letter. Other than that, the name of a function can be anything you like, but ideally the name that you choose should give some clue as to what the function does and should not be too long.

The return Statement

The return statement provides the means of exiting from a function and resuming execution of the calling function at the point from which the call occurred. In its simplest form, the return statement is just this:

```
return;
```

This form of the return statement is used exclusively in a function where the return type has been declared as void. It doesn't return a value. The more general form of the return statement is:

```
return expression;
```

This form of return statement must be used when the return value type for the function has been declared as some type other than void. The value that's returned to the calling program is the value that results when expression is evaluated, and this should be of the return type specified for the function.

The Pass-By-Value Mechanism

When you pass an argument to a function, the argument value, whatever it is, is not passed directly to the function.

A copy of the argument value is made first and stored on the stack, and it is this copy that is made available to the function, not the original value.

Function declaration

A *function declaration, also called a function prototype, is a statement that defines the essential characteristics of* a function. It defines its name, its return value type, and the type of each of its parameters. You can write a prototype for a function exactly the same as the function header and just add a semicolon at the end.

A function declaration is referred to as a function prototype because it provides all the external specifications for the function. A function prototype enables the compiler to generate the appropriate instructions at each point where you call the function and to check that you use it correctly in each case. When you include a standard header file in a program, the header file adds the function prototypes for library functions to the program. For example, the header file stdio.h contains function prototypes for printf() and scanf(), among others.

Function prototypes generally appear at the beginning of a source file prior to the definitions of any of the functions or in a header file. The function prototypes are then external to all of the functions in the source file, and their scope extends to the end of the source file, thereby allowing any of the functions in the file to call any function regardless of where you've placed the definitions of the functions.

Note that the parameter names do not have to be the same as those used in the function definition. It is not even required to include the names of parameters in a function prototype.

Two categories of functions:

Standard library functions – getchar(), putchar(), printf(), etc

User-defined functions – The user designs and implements these.

Some simple programs

1. Write a function to which we pass an integer and the function returns the number of bits set to 1 in that integer.
2. Write a function to reverse the digits in a given integer
3. Write a program to draw a box (using the '*' character given the width and height.

Interfaces and implementations
A module comes in two parts, its **interface** and its **implementation**.

The interface specifies *what a module does. It declares the identifiers,* types, and routines that are available to code that uses the module. An implementation specifies *how a module accomplishes the* purpose advertised by its interface. For a given module, there is usually one interface, but there might be many implementations that provide the facilities specified by the interface. Each implementation might use different algorithms and data structures, but they all must meet the specification given by the interface.

A *client is a piece of code that uses a module. Clients import interfaces;* implementations *export them. Clients need to see only the interface.* **Indeed, they may have only the object code for an implementation**.

Clients share interfaces and implementations, thus avoiding unnecessary code duplication. This methodology also helps avoid bugs — interfaces and implementations are written and debugged once, but used often.

Interfaces

An interface specifies only those identifiers that clients may use, hiding irrelevant representation details and algorithms as much as possible. This helps clients avoid dependencies on the specifics of particular implementations. This kind of dependency between a client and an implementation — *coupling* — *causes bugs when an implementation* changes; these bugs can be particularly hard to fix when the dependencies are buried in hidden or implicit assumptions about an implementation. A well-designed and precisely specified interface reduces coupling.

C has only minimal support for separating interfaces from implementations, but simple conventions can yield most of the benefits of the interface/implementation methodology. In C, an interface is specified by a header file, which usually has a .h file extension. This header file declares the macros, types, data structures, variables, and routines that clients may use.

A client imports an interface with the C pre-processor #include directive.

Implementations

An implementation exports an interface. It defines the variables and functions necessary to provide the facilities specified by the interface. An implementation reveals the representation details and algorithms of its particular rendition of the interface, but, ideally, clients never need to see these details. Clients share object code for implementations, usually by loading them from libraries.

An interface can have more than one implementation. As long as the implementation adheres to the interface, it can be changed without affecting clients. A different implementation might provide better performance, for example. Well-designed interfaces avoid machine dependencies, but may force implementations to be machine-dependent, so different implementations or parts of implementations might be needed for each machine on which the interface is used.

What is Recursion?
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.

<u>Recursive functions</u>
The C programming language supports recursion, i.e., a function to call itself. While using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

What is base condition in recursion?

In the recursive program, the solution to the base case is provided and the solution of the bigger problem is expressed in terms of smaller problems.

```
int fact(int n)
{
        if (n < = 1) // base case
                return 1;
        else
                return n*fact(n-1);
}
```

In the above example, base case for n < = 1 is defined and larger value of number can be solved by converting to smaller one till base case is reached.

How is a particular problem solved using recursion?
The idea is to represent a problem in terms of one or more smaller problems, and add one or more base conditions that stop the recursion. For example, we compute factorial n if we know factorial of (n-1). The base case for factorial would be n = 0. We return 1 when n = 0.

Why does "Stack Overflow" error occur in recursion?
If the base case is not reached or not defined, then the stack overflow problem may arise. Let us take an example to understand this.

```
int fact(int n)
{
        // wrong base case (it may cause stack overflow).
        if (n == 100)
                return 1;
        else
                return n*fact(n-1);
}
```

If fact(10) is called, it will call fact(9), fact(8), fact(7) and so on but the number will never reach 100. So, the base case is not reached. If the memory is exhausted by these functions on the stack, it will cause a stack overflow error.

Direct and indirect recursion

A function **fun** is called direct recursive if it calls the same function **fun**. A function **fun** is called indirect recursive if it calls another function say **fun_new** and **fun_new** calls **fun** directly or indirectly.

What are the disadvantages of recursive programming over iterative programming?
Note that both recursive and iterative programs have the same problem-solving powers, i.e., every recursive program can be written iteratively and vice versa is also true. The recursive program has greater space requirements than iterative program as all functions will remain in the stack until the base case is reached. It also has greater time requirements because of function calls and returns overhead.

What are the advantages of recursive programming over iterative programming?
Recursion provides a clean and simple way to write code. Some problems are inherently recursive like tree traversals, Tower of Hanoi, etc. For such problems, it is preferred to write recursive code. We can write such codes also iteratively with the help of a stack data structure.

Simple examples

Calculating the factorial of a given number

```
#include <stdio.h>
unsigned long long int factorial(unsigned int num)
{
   if(num <= 1)
      return 1;
   return num * factorial(num - 1);
}

int main(void)
{
   int num;
   printf("key in a positive number");
   scanf ("%d", &num);
   printf("Factorial of %d is %ld\n", num, factorial(num));
   return 0;
}
```

The Fibonacci sequence

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

```
#include <stdio.h>
int fibonacci(int i)
{
   if(i == 0)
      return 0;
   if(i == 1)
      return 1;
   return fibonacci(i-1) + fibonacci(i-2);
}

int main(void)
{
   int i;
   for (i = 0; i < 10; i++)
      printf("%d\t\n", fibonacci(i));
   return 0;
}
```

Euclid’s Algorithm: gcd(m,n) = gcd(n, m mod n) and gcd(m, 0) = m

GCD using recursion

```
#include <stdio.h>
int hcf(int n1, int n2);
int main(void)
{
    int n1, n2;
    printf("Enter two positive integers: ");
    scanf("%d %d", &n1, &n2);
    printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1, n2));
    return 0;
}
int hcf(int n1, int n2)
{
    if (n2 != 0)
        return hcf(n2, n1 % n2);
    else
        return n1;
}
```

Tail recursion

A special form of recursion where the last operation of a function is a recursive call

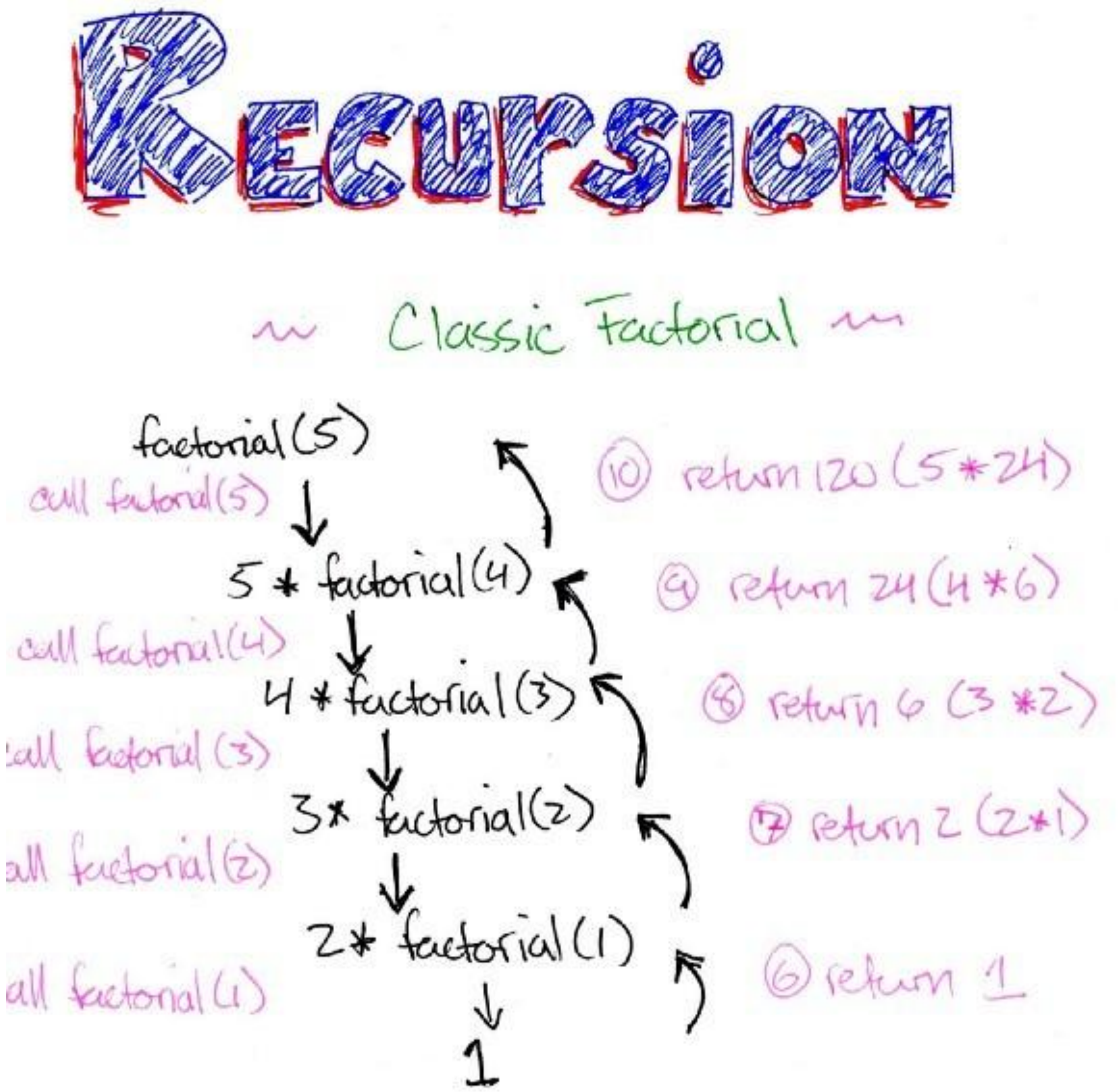
```
// An example of tail recursive function
void print(int n)
{
    if (n < 0)
        return;
    printf (“%d “, n);
    // The last executed statement is recursive call
    print(n-1);
}
```

How is memory allocated to different function calls in recursion?
When any function is called from main(), the memory is allocated to it on the stack. A recursive function calls itself, the memory for a called function is allocated on top of memory allocated to calling function and different copies of local variables are created for each function call. When the base case is reached, the function returns its value to the function by whom it is called and memory is de-allocated and the process continues.

```
#include <stdio.h>
void printFun(int test)
{
    if (test < 1)
        return;
    else
    {
        printf (“%d “, test);
        printFun(test-1);    // statement 2
        printf (“%d “, test);
        return;
    }
}

int main()
{
    int test = 3;
    printFun(test);
}
```

```
long factorial(int n)
{
  if (n <= 1) // Base case
    return 1;
  else
    return (n*factorial(n-1));
}
```


Recursion
Classic Factorial
factorial(5)
call factorial(5)
5 * factorial(4)
call factorial(4)
4 * factorial(3)
call factorial(3)
3* factorial(2)
call factorial(2)
2* factorial(1)
call factorial(1)
1
⑩ return 120 (5*24)
⑨ return 24 (4*6)
⑧ return 6 (3*2)
⑦ return 2 (2*1)
⑥ return 1

Arrays

An array is a fixed number of data items that **are all of the same type**. The data items in an array are referred to as elements. The elements in an array are all of type int, or of type long, or all of any type you choose.

```
char   c_array [10];
short    s_array [20];
```

Access the elements through the index which starts from 0.

An array is defined as **finite ordered collection of homogenous** data, stored in contiguous memory locations.
finite *means* data range must be defined.
ordered *means* data must be stored in continuous memory addresses.
homogenous *means* data must be of similar data type.

Initialization of an Array

After an array is declared it must be initialized. Otherwise, it will contain **garbage** value(any random value). An array can be initialized at either **compile time** or at **runtime**.

Compile time Array initialization

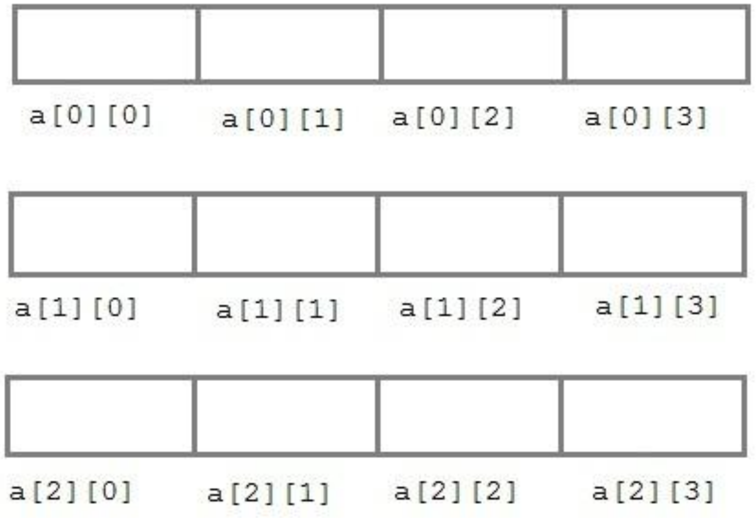
Compile time initialization of array elements is same as ordinary variable initialization. The general form of initialization of array is,

```
data-type array-name[size] = { list of values };
int marks[4]={ 67, 87, 56, 77 }; // integer array initialization
float area[5]={ 23.4, 6.8, 5.5 }; // float array initialization
int arr[] = {2, 3, 4};
int marks[4]={ 67, 87, 56, 77, 59 }; // Compile time error
```

Runtime initialization
Use scanf to get the input into the array;

Multi-dimensional arrays

data-type array-name[row-size][column-size]
/* Example */
int a[3][4];



Declaring and initializing an array together

```
int arr[][3] = {
            {0,0,0},
            {1,1,1}
            };
```

Note: We have not assigned any row value to our array in the above example. It means we can initialize any number of rows. But, we must always specify number of columns, else it will give a compile time error.

Runtime initialization of a two dimensional Array

```
#include<stdio.h>
int main(void)
{
        int arr[3][4];
        int i, j;
        printf("Enter array elements");
        for(i = 0; i < 3;i++)
                for(j = 0; j < 4; j++)
                        scanf("%d", &arr[i][j]);

        for(i = 0; i < 3; i++)
                for(j = 0; j < 4; j++)
                        printf("%d", arr[i][j]);
        return 0;
}
```

Row-major and column-major methods of storing

In computing, **row-major order** and **column-major order** are methods for storing multidimensional arrays in linear storage such as random access memory.

The difference between the orders lies in which elements of an array are contiguous in memory. In a row-major order, the consecutive elements of a row reside next to each other, whereas the same holds true for consecutive elements of a column in a column-major order.

Programming languages or their standard libraries that support multi-dimensional arrays typically have a native row-major or column-major storage order for these arrays.

Row-major order is used in C/C++/Objective-C (for C-style arrays), etc
Column-major order is used in Fortran, MATLAB, etc

String and Character Array

String is a sequence of characters that is treated as a single data item and terminated by null character '\0'. Remember that C language does not support strings as a data type.

A **string** is actually one-dimensional array of characters in C language. These are often used to create meaningful and readable programs.

For example: The string "hello world" contains 12 characters including '\0' character which is automatically added by the compiler at the end of the string.

Declaring and Initializing a string variables

```
char name[13] = "StudyTonight"; // valid character array initialization
char name[10] = {'L','e','s','s','o','n','s','\0'}; // valid initialization
```

Remember that when you initialize a character array by listing all of its characters separately then you must supply the '\0' character explicitly.

Some examples of illegal initialization of character array are,

```
char ch[3] = "hell"; // Illegal
char str[4];
str = "hell"; // Illegal
```

String Input and Output

Input function scanf() can be used with **%s** format specifier to read a string input from the terminal. But there is one problem with scanf() function, it terminates its input on the first white space it encounters. Therefore if you try to read an input string "Hello World“ using scanf() function, it will only read **Hello** and terminate after encountering white spaces.

However, C supports a format specification known as the **edit set conversion code %[..]** that can be used to read a line containing a variety of characters, including white spaces.

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

int main(void)
{
        char str[20];
        printf("Enter a string");
        scanf("%[^\n]", str);
//scanning the whole string, including the white spaces
        printf("%s", str);
        return 0;
}
```

Write functions to perform the following:

1. Concatenating two strings
2. Finding the length of a string
3. Reversing a string
4. Copying one string to another
5. Comparing two strings

Hands-on session

Programs

Write a program in C to simulate a calculator with the following operations: +, -, * and /.

Write a program in C to print the tables from 1 to 12 in a format shown below:

Write a program in C to convert a given string (for ex “1234”) to the corresponding integer.

Write a program to convert a given integer value to a string format.

Write a program to convert a given integer into a string which has the binary equivalent of the number.
Example: Assuming that an integer is 32-bits long and for the number 5, the resultant string should be "00000000000000000000000000000101"

Variable Length Arrays (C99)

We can define arrays where the dimensions are determined at runtime, when you execute the program.

```
int size = 0;
printf("Enter the number of elements you want to store: ");
scanf("%d", &size);
float values[size];
```

You can also define arrays with two or more dimensions with any or all of the dimensions determined at execution time. For example:

```
int rows = 0;
int columns = 0;
printf("Enter the number of rows you want to store: ");
scanf("%d", &rows);
printf("Enter the number of columns in a row: ");
scanf("%d", &columns);
float beans[rows][columns];
```

<u>A C11-conforming compiler does not have to implement support for variable length arrays because it is an optional feature</u>. If it does not, the symbol __STDC_NO_VLA__ must be defined as 1. You can check for support for variable length arrays using this code:

```
#ifdef __STDC_NO_VLA__
printf("Variable length arrays are not supported.\n");
exit(1);
#endif
```

String

A *string constant is a sequence of characters or symbols* between a pair of double-quote characters. Anything between a pair of double quotes is interpreted by the compiler as a string, including any special characters and embedded spaces.

Variables that store Strings

C has no specific provision within its syntax for variables that store strings, and because there are no string variable types, there are no special operators in the language for processing strings. This is not a problem though, because the standard library provides an extensive range of functions to handle strings.

We use an array of type char to hold strings. This is the simplest form of string variable. You can declare an array variable like this:

```
char saying[20];
```

This variable can accommodate a string that contains up to 19 characters, because you must allow one element for the termination character. Of course, you could also use this array to store 20 characters that are not a string.

Caution

Remember that when you specify the dimension of an array that you intend to use to store a string, it must be at least one greater than the number of characters in the string that you want to store. The compiler automatically adds \0 to the end of every string constant.

You can initialize a string variable when you declare it:

```
char saying[] = "This is a string.";
```

Initializing a char array and declaring it as constant is a good way of handling standard messages:

```
const char message[] = "The end of the world is nigh.";
```

Because you declare message as const, it’s protected from being modified explicitly within the program. Any attempt to do so will result in an error message from the compiler. This technique for defining standard messages is particularly useful if they’re used in many places within a program. It prevents accidental modification of such constants in other parts of the program. Of course, if you do need to be able to change the message, then you shouldn’t specify the array as const.

Arrays of Strings

You can use a two-dimensional array of elements of type char to store strings, where each row holds a separate string. In this way you can store a whole bunch of strings and refer to any of them through a single variable name, as in this example:

```
char sayings[3][32] = {
"Manners maketh man.",
"Many hands make light work.",
"Too many cooks spoil the broth."
};
```

This definition creates an array of three rows of 32 characters. The strings between the braces will be assigned in sequence to the three rows of the array, sayings[0], sayings[1], and sayings[2]. Note that you don’t need to put braces around each string. The compiler deduces that each string is intended to initialize one row of the array. The first dimension specifies the number of strings that the array can store. The second dimension is specified as 32, which is just sufficient to accommodate the longest string, including its terminating \0 character.

<u>**The Idea of a Pointer**</u>

C provides a remarkably useful type of variable called a pointer. <u>A *pointer* is a variable that stores an address—that is, its value is the address of another location in memory that can contain a value</u>.

You already used an address when you used the scanf() function. A pointer variable with the name pNumber is defined by the second of the following two statements:

```
int Number = 25;
int *pNumber = &Number;
```

You declare a variable, Number, with the value 25, and a pointer, pNumber, which contains the address of Number.

You can now use the variable pNumber in the expression *pNumber to obtain the value contained in Number. The * is the dereference operator, and its effect is <u>to access the data stored at the address specified by a pointer</u>.

Operations with Strings

The standard library provides a number of functions for processing strings. To use these, you must include the string.h header file into your source file. String functions introduced in the C11 standard are safer and more robust than the traditional functions you may be used to using. They offer greater protection against errors such as buffer overflow. However, that protection is dependent on careful and correct coding. These string processing functions do bounds checking for arrays. The names of these functions end with _s.

How do you know whether your compiler supports these functions?

```
#include <stdio.h>
int main(void)
{
#if defined __STDC_LIB_EXT1__
printf("Optional functions are defined.\n");
#else
printf("Optional functions are not defined.\n");
#endif
return 0;
}
```

The nine most commonly used functions in the string library are:

strcat - concatenate two strings
strchr - string scanning operation
strcmp - compare two strings
strcpy - copy a string
strlen - get string length
strncat - concatenate one string with part of another
strncmp - compare parts of two strings
strncpy - copy part of a string
strrchr - a pointer to the **last** occurrence of c within s instead of the first.

The strcat function

```
char *strcat(char * s1, const char * s2);
```

The strcat() function appends a copy of the string pointed to by s2 (including the terminating null byte) to the end of the string pointed to by s1. The initial byte of s2 overwrites the null byte at the end of s1. If copying takes place between objects that overlap, the behavior is undefined. The function returns s1.

This function is used to attach one string to the end of another string.

The strchr function

char *strchr(const char *s, int c);

The strchr() function shall locate the first occurrence of c (converted to a char) in the string pointed to by s. The terminating null byte is considered to be part of the string.

<u>The function returns the location of the found character, or a null pointer if the character was not found</u>.

This function is used to find certain characters in strings.

The strcmp function

int strcmp(const char *s1, const char *s2);

The strcmp() function shall compare the string pointed to by s1 to the string pointed to by s2. The sign of a non-zero return value shall be determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type unsigned char) that differ in the strings being compared. Upon completion, strcmp() shall return an integer greater than, equal to, or less than 0, if the string pointed to by s1 is greater than, equal to, or less than the string pointed to by s2, respectively.

The strcpy function

```
char *strcpy(char *s1, const char *s2);
```

The strcpy() function shall copy the C string pointed to by s2 (including the terminating null byte) into the array pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined. The function returns s1. There is no value used to indicate an error: if the arguments to strcpy() are correct, and the destination buffer is large enough, the function will never fail.

Important: You must ensure that the destination buffer (s1) is able to contain all the characters in the source array, including the terminating null byte. Otherwise, strcpy() will overwrite memory past the end of the buffer, causing a buffer overflow, which can cause the program to crash, or can be exploited by hackers to compromise the security of the computer.

The strlen function

size_t strlen(const char *s);

The strlen() function shall compute the number of bytes in the string to which s points, not including the terminating null byte. It returns the number of bytes in the string. No value is used to indicate an error.

The strncat function

char *strncat(char *s1, const char *s2, size_t n);

The strncat() function shall append not more than n bytes (a null byte and bytes that follow it are not appended) from the array pointed to by s2 to the end of the string pointed to by s1. The initial byte of s2 overwrites the null byte at the end of s1. A terminating null byte is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined. The function returns s1

The strncmp function

int strncmp(const char *s1, const char *s2, size_t n);
The strncmp() function shall compare not more than n bytes (bytes that follow a null byte are not compared) from the array pointed to by s1 to the array pointed to by s2. The sign of a non-zero return value is determined by the sign of the difference between the values of the first pair of bytes (both interpreted as type unsigned char) that differ in the strings being compared. See strcmp for an explanation of the return value.

The strncpy function

char *strncpy(char *s1, const char *s2, size_t n);

The strncpy() function shall copy not more than n bytes (bytes that follow a null byte are not copied) from the array pointed to by s2 to the array pointed to by s1. If copying takes place between objects that overlap, the behavior is undefined. If the array pointed to by s2 is a string that is shorter than n bytes, null bytes shall be appended to the copy in the array pointed to by s1, until n bytes in all are written. The function shall return s1; no return value is reserved to indicate an error.
It is possible that the function will **not** return a null-terminated string, which happens if the s2 string is longer than n bytes.

The strrchr function

char *strrchr(const char *s, int c);

The strrchr function is similar to the strchr function, except that strrchr returns a pointer to the **last** occurrence of c within s instead of the first.

The strrchr() function shall locate the last occurrence of c (converted to a char) in the string pointed to by s. The terminating null byte is considered to be part of the string. Its return value is similar to strchr's return value.

Summary

```
char *strcat(char * s1, const char * s2);
char *strchr(const char *s, int c);
int strcmp(const char *s1, const char *s2);
char *strcpy(char *s1, const char *s2);
int strlen(const char *s);
char *strncat(char *s1, const char *s2, int n);
int strncmp(const char *s1, const char *s2, int n);
char *strncpy(char *s1, const char *s2, int n);
char *strrchr(const char *s, int c);
```

Designated initializers (C99)

It is often the case that relatively few elements of an array need to be initialized explicitly; the other elements can be given default values:

int a[15] = {0,0,29,0,0,0,0,0,0,7,0,0,0,0,48};

So, we want element 2 to be 29, 9 to be 7 and 14 to be 48 and the other values to be zeros. For large arrays, writing an initializer in this fashion is tedious.

C99's **designated initializers**

int a[15] = {[2] = 29, [9] = 7, [14] = 48};

Each number in the brackets is said to be a designator.

Besides being shorter and easier to read, designated initializers have another advantage: the order in which the elements are listed no longer matters. The previous expression can be written as:

int a[15] = {[14] = 48, [9] = 7, [2] = 29};

Designators must be constant expressions. If the array being initialized has length n, each designator must be between 0 and n-1. However, if the length of the array is omitted, a designator can be any non-negative integer. In that case, the compiler will deduce the length of the array by looking at the largest designator.

```
int b[] = {[2] = 6, [23]=87};
```

Because 23 has appeared as a designator, the compiler will decide the length of this array to be 24.

An initializer can use both the older technique and the later technique.

```
int c[10] = {5, 1, 9, [4] = 3, 7,2,[8]=6};
```

The initializer specifies that the array's first three elements will be 5,1 and 9, Element 4 will have value 3. The two elements after element 4 will be 7 and 2. Finally, element 8 will have the value 6. All elements for which no value is specified will default is zero.

Problem

Using arrays, write a program to check whether a given number has repeated digits.

Ex: 456754 (has repeated digits)

3456 (Does not have)

```
#include <stdio.h>
#include <stdbool.h>      // C99 only

int main (void)
{
            long num;
            int digit;
            bool digit_seen [10] = {false};
            printf (“Enter a number:”);
            scanf (“%d”, &num);

            while (num > 0)
            {
                        digit = num%10;
                        if (digit_seen [digit])
                                    break;
                        digit_seen [digit] = true;
                        num /= 10;
            }
            if (num > 0)
                        printf (“Digits are repeated\n”);
            else
                        printf (“No repitition\m”);
            return 0;
}
```

Using the sizeof operator with arrays

The sizeof operator executed on an array gives the size of the array in bytes:

```
char arr [20];
sizeof (arr) – results in 20 bytes
```

What about the following:

```
int iarr [20];
sizeof (iarr)   -  ??

long larr [40];
sizeof (larr)   -   ??
```

We can use sizeof to measure the size of an array element:

```
    char carr [20];
    sizeof (carr) – gives 20
    sizeof (carr[0]) --  ??
```

What is the result of the following:

```
    sizeof (carr) / sizeof (carr[0])

    char m_array [20][30];
    sizeof (m_array)  ---   ??
```

Constant arrays

```
const char hex_array[] = {‘0’, ‘1’, ‘2’, ‘3’, ‘4’, ‘5’, ‘6’, ‘7’, ‘8’, ‘9’,
                          ‘A’, ‘B’, ‘C’, ‘D’, ‘E’, ‘F’};
```

The compiler will detect any attempt to modify the values and inform the user.

<u>Arrays and Pointers – Differences</u>

1. the sizeof operator
   a. sizeof(array) returns the amount of memory used by all elements in array
   b. sizeof(pointer) only returns the amount of memory used by the pointer variable itself
2. the & operator
   a. &array is an alias for &array[0] and returns the address of the first element in array
   b. &pointer returns the address of pointer
3. a string literal initialization of a character array
   a. char array[] = “abc” sets the first four elements in array to ‘a’, ‘b’, ‘c’, and ‘\0’
   b. char *pointer = “abc” sets pointer to the address of the “abc” string (which may be stored in read-only memory and thus unchangeable)
4. Pointer variable can be assigned a value whereas array variable cannot be.
5. Arithmetic on pointer variable is allowed.

Pointers

Note

char* is a **mutable** pointer to a **mutable** character/string.

const char* is a **mutable** pointer to an **immutable** character/string. You cannot change the contents of the location(s) this pointer points to. Also, compilers are required to give error messages when you try to do so. For the same reason, conversion from const char * to char* is deprecated.

char* const is an **immutable** pointer (it cannot point to any other location) **but** the contents of location at which it points are **mutable**.

const char* const is an **immutable** pointer to an **immutable** character/string.