

Intro+Single Character Input And Output

Let us answer few questions before starting with Unit 2.

What is Text Processing?

The term text processing refers to the Analysis, manipulation, and generation of text. Text usually refers to all the alphanumeric characters specified on the keyboard .

What is String Matching?

String matching is the Classical and existing problem of searching a given " Pattern " within a well defined string or “Text”

Real world Applications of String matching.

- ✓ Spell Checkers
- ✓ Spam Filters
- ✓ Intrusion Detection System
- ✓ Search Engines
- ✓ Plagiarism Detection
- ✓ Bioinformatics
- ✓ Digital Forensics
- ✓ Information Retrieval Systems

List of few Text processing and String Matching Problems

- ✓ Given a text, list the stop words
- ✓ Building search Engines
- ✓ Develop a tool for Plagiarism Detection in any reports
- ✓ Design a spell checkers for different editors
- ✓ Given a set of values, display all the repeated words
- ✓ Given a text, display all the questions asked in the text.
- ✓ Given a data set, list all the words starting with a given character and list name and phone numbers of the persons who registered for the event.

- ✓ Finding the particular pattern in a given text using a Basic Brute Force or Naive String-Matching Algorithm

Introduction

A Character refers to a single input . It Occupies a single byte. We code English alphabets A as 65, B as 66, Z as 90, a as 97 and so on in a coding scheme called ASCII.

Single Character Input-Output Functions

- 1) scanf and printf – Discussed in Unit 1

```
printf("%c", x);  
scanf("%c", &x);
```

- 2) getchar and putchar – Discussed in Unit 1

```
c=getchar();  
putchar(c);
```

- 3) getc and putc

- 4) getch , getche and putch

getc and putc functions :

The simplest file I/O functions are getc and putc. These are analogous to getchar and putchar functions and handles one character at a time.

Assume that a file is opened with mode w and the file pointer fp1.

Syntax:

```
int putc(int char, FILE *fp)
```

Example:

```
putc(c,fp1);
```

putc writes the character contained in the variable c to the file associated with the FILE pointer fp1.

getc is used to read a character from a file that has been opened in read mode.

Syntax:

```
int getc(FILE *fp)
```

Example:

```
c=getc(fp2);
```

getc reads a character from the file whose file pointer is fp2.

The file pointer moves by one character position for every operation of getc and putc .The getc will return an end of file marker EOF ,when end of file has been reached.Therefore the reading should be terminated when EOF is encountered.

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Screen

Example Program:

Program 1:

```
#include<stdio.h>

int main ()
{
    char c;
    printf("Enter character: ");
    c = getc(stdin);
    printf("Character entered: ");
    putc(c, stdout);
}
```

```
    return 0;  
}
```

Program 2:

```
#include <stdio.h>  
  
int main ()  
{  
    char ch;  
    while((ch = getc(stdin)) != EOF)  
    {  
        putc(ch, stdout);  
    }  
}
```

In the above program the `getc()` function reads character from the standard input file (`stdin`). And the `putc()` function displays to the standard output file (`stdout`).

We will be looking at these functions again in File processing topic under Unit IV.

Non-Standard Input Output Functions

`getc()` , `getche()` and `putc()` functions are called as **non-standard functions** since they are not part of standard library and ISO C(certified c programming language).

`getc()` and `getche()`:

`getc()` and `getche()` reads a character from the keyboard and copies it into memory area which is identified by the variable `ch`. No arguments are required for this function. `getc` and `getche()` function doesn't wait for the user to press enter/return key.

In the `getc()` function the typed character will not be echoed(displayed) on the screen and in `getche()` function the typed character will be echoed(displayed) on the screen.

Syntax:`ch=getch()``or``ch=getche();`**putch():**

putch function outputs a character stored in the memory using a variable on the output screen.

Syntax:`putch(ch);`**Program 1:**

```
int main()
{
    char ch;
    ch = getch();
    putch(ch);
    return 0;
}
```

Program 2

```
int main()
{
    char ch;
    ch = getche();
    putch(ch);
    return 0;
}
```

Chapter 3: Functions

Functions break large computing tasks into smaller ones and enable people to build on what others have done instead of starting from scratch. In programming, Function is a subprogram to carry out a specific task.

A function is a self-contained block of code that performs a particular task. Once the function is designed, it can be treated as a black box. The inner details of operation are invisible to rest of the program.

Benefits of using functions.

1. Reduced Coding Time – Code Reusability
2. Modularity.
3. Reduced Debugging Time
4. Sharing
5. Maintainability

3.1 Types

C functions can be broadly classified into two categories:

- **Library Functions**
 - Functions which are defined by C library. Examples include printf(), scanf(), strcat() etc. We just need to include appropriate header files to use these functions. These are already declared and defined in C libraries.
- **User-defined Functions**
 - Functions which are defined by the developer at the time of writing program.

- Developer can make changes in the implementation as and when he/she wants.

In order to make use of user defined functions, we need to understand three key terms associated with functions: **Function Definition, Function call and Function Declaration.**

3.2 Function Definition

Each function definition has the form

```
return_type function_name(parameters) // parameters optional
{
    // declarations and statements
}
```

A function definition should have a return type. The return type must match with what that function returns at the end of the function execution. If the function is not designed to return anything, then the type must be mentioned as void. Like variables, **function name is also an identifier** and hence **must follow the rules of an identifier**. Parameters list is optional. If more than one parameter, it must be comma separated. Each parameter is declared with its type and parameters receive the data sent during the function call. If function has parameters or not, but the parentheses is must. Block of statements inside the function or body of the function must be inside { and }. There is no indentation requirement as far as the syntax of C is considered. For readability purpose, it is a good practice to indent the body of the function.

Function definitions can occur in any order in the source file and the source program can be split into multiple files.

```
int sum(int a, int b)
{
```

```
return a+b;
}

int decrement(int y)
{
return y-1;
}

void disp_hello()    // This function doesn't return anything
{
printf("Hello Friends\n");
}

double use_pow(int x)
{
return pow(x,3);      // using pow() function is not good practice.
}

int fun1(int a, int)  // Error in function definition.
{
}

int fun2(int a, b)    // Invalid Again.
{}
```

3.3 Function Call

Function-name(list of arguments); // semicolon compulsory
// Arguments depends on the number of parameters in the function definition

A function can be called by using function name followed by list of arguments (if any) enclosed in parentheses. **The function which calls other function is known to be Caller and the function which is getting called is known**

to be the Callee. The arguments must match the parameters in the function definition in its type, order and number. Multiple arguments must be separated by comma. Arguments can be any expression in C. Need not be always just variables. A function call is an expression. When a function is called, Activation record is created. **Activation record is another name for Stack Frame.** It is composed of:

- Local variables of the callee
- Return address to the caller
- Location to store return
- valueParameters of the
- callee Temporary variables

The order in which the arguments are evaluated in a function call is not defined and is determined by the calling convention(out of the scope of this notes) used by the compiler. It is left to the compiler Writer. Always arguments are copied to the corresponding parameters. Then the control is transferred to the called function. Body of the function gets executed. When all the statements are executed, callee returns to the caller. OR when there is return statement, the expression of return is evaluated and then callee returns to the caller. If the return type is void, function must not have return statement inside the function.

```
#include<stdio.h>int main()
{
int x = 100;

int y = 10;
```

```
int answer = sum(x,y);
printf("sum is %d\n",answer);
answer = decrement(x);
printf("decremented value is %d\n",answer);
disp_hello();
double ans = use_pow(x);
printf("ans is %lf\n",ans);
answer = sum(x+6,y);
printf("answer is %d\n", answer);
printf("power : %lf\n", use_power(5));
return 0;
}
```

3.4 Function Declaration/ Prototype

All functions must be declared before they are invoked.

The function declaration is as follows.

```
return_type Function_name (parameters list); // semicolon compulsory
```

The parameters list must be separated by commas. The parameter names do not need to be the same in declaration and the function definition. The types must match the type of parameters in the function definition in number and order. Use of identifiers in the declaration is optional. When the declared types do not match with the types in the function definition, compiler will produce an error.

Interface and Implementation

The **function declaration** is the interface. The **function Definition** is the implementation.

Interface tells us what the function expects. It does not tell us how the function works.

If we change the function definition, the user or the client of the function is not affected.

PES University

3.1 Parameter Passing in C

Parameter passing is always by Value in C

Argument is copied to the corresponding parameter. The parameter is not copied back to the argument. It is possible to copy back the parameter to argument only if the argument is l-value. Argument is not affected if we change the parameter inside a function.

Version 1:

```
void fun1(int a1); // declaration
```

```
int main(){
```

```
    int a1 = 100;
```

```
    printf("before function call a1 is %d\n", a1); // a1 is 100
```

```
    fun1(a1); // call
```

```
    printf("after function call a1 is %d\n", a1); // a1 is 100
```

```
    return 0;
```

```
}
```

```
void fun1(int a1)
```

```
{
```

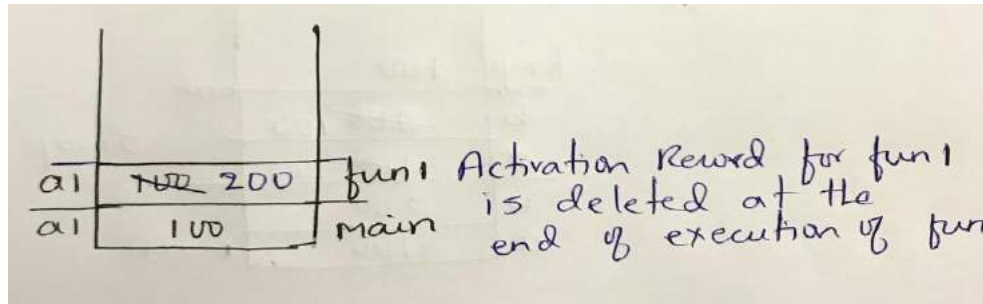
```
    printf("a1 in fun1 before changing %d\n", a1); //100
```

```
    a1 = 200;
```

```
    printf("a1 in fun1 after changing %d\n", a1); //200
```

```
}
```

a1 has not changed in main function. The parameter a1 in fun1 is a copy of a1 from main function. Refer to the below diagram to understand activation record creation and deletion to know this program output in detail.



Think about this Question: **Is it impossible to change the argument by calling a function?**

If yes, then how library functions work? Example: scanf, strcpy, strncpy and so on.

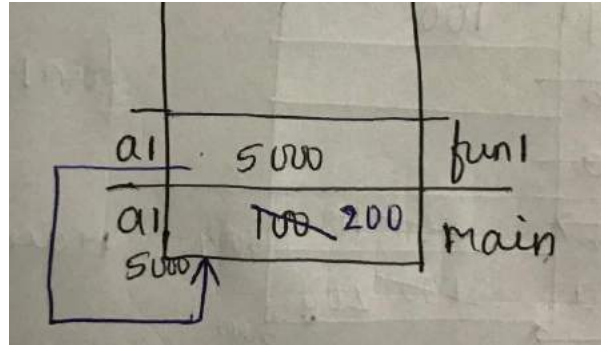
Answer is No. It is possible to change the argument if we pass l-value

Version 2:

```
void fun1(int *a1);

int main(){
    int a1 = 100;
    printf("before function call a1 is %d\n", a1);    // 100
    fun1(&a1);    // call
    printf("after function call a1 is %d\n", a1);    // 200
}

void fun1(int *a1)
{
    printf("*a1 in fun1 before changing %d\n", *a1); //100
    *a1 = 200;
    printf("*a1 in fun1 after changing %d\n", *a1);    //200
}
```

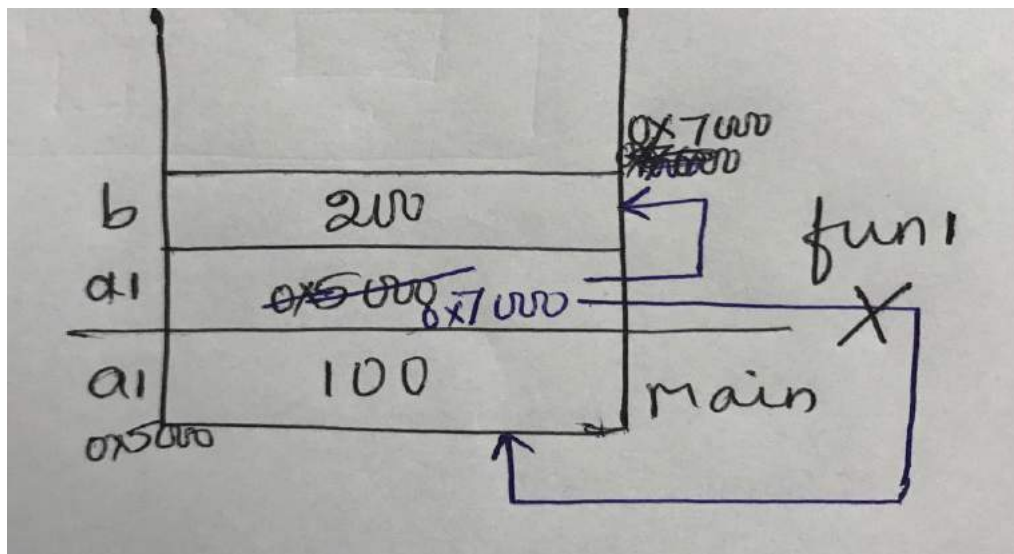


Version 3:

```
void fun1(int *a1);

int main(){
    int a1 = 100;
    printf("before function call a1 is %d\n", a1);    // 100
    fun1(&a1);    // call
    printf("after function call a1 is %d\n", a1);    // 100
}

void fun1(int *a1)
{
    int b = 200;
    printf("*a1 in fun1 before changing %d\n", *a1); //100
    a1 = &b;
    printf("*a1 in fun1 after changing %d\n", *a1);  //200
}
```

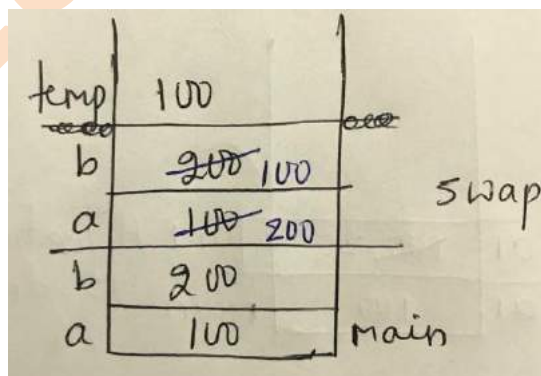


Shall we write the function to swap two numbers and test this function?

Version – 1: Is this right version?

```
void swap(int a, int b)
{
    int temp = a; a = b; b = temp;
}

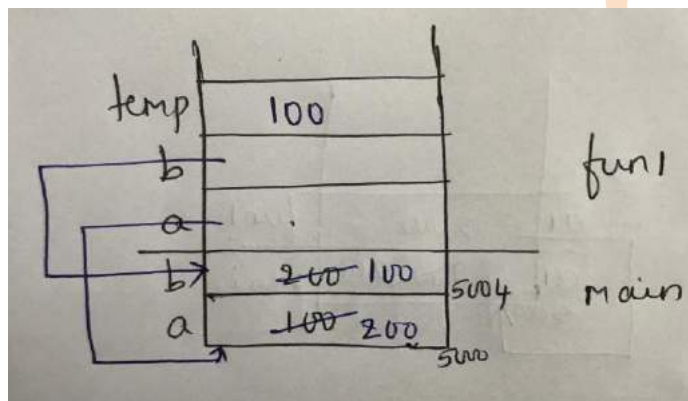
int main()
{
    int a = 100; int b = 200;
    printf("before call a is %d and b is %d\n", a, b);    // a is 100 and b is
    200
    swap(a, b);
    printf("after call a is %d and b is %d\n", a, b);    // a is 100 and b is 200
}
```



Version 2:

```
void swap(int *a, int *b)
{
    int temp = *a; *a = *b; *b = temp; }

int main()
{
    int a = 100; int b = 200;
    printf("before call a is %d and b is %d\n", a, b); // a is 100 and b is
    200
    swap(&a, &b);
    printf("after call a is %d and b is %d\n", a, b); // a is 100 and b is 200
}
```



3.2 return keyword in C

Function must do what it is supposed to do. It should not do something extra. ForExample, refer to the below code.

```
int add(int a, int b)
{
return a+b;
}
```

The function add() is used to add two numbers. It should not print the sum inside the

function. We cannot use this kind of functions repeatedly if it does something extra than what is required. Here comes the usage of return keyword inside a function.

Syntax: **return expression;**

In 'C', function returns a single value. The expression of return is evaluated and copied to the temporary location by the called function. This temporary location does not have any name. **If the return type of the function is not same as the expression of return in the function definition, the expression is cast to the return type before copying to the temporary location.** The calling function will pick up the value from this temporary location and use it.

```
void f1(int);
void f2(int*);
void f3(int);
void f4(int*);
int* f5(int* );
int* f6();
int main()
{   int x =
```

```
100;f1(x);
printf("x is %d\n", x);          // 100
double y = 6.5;
f1(y);
// observe what happens when double value is passed as
//argument to integer parameter?

printf("y is %lf\n", y);
int *p = &x; // pointer variable
f2(p);
printf("x is %d and *p is %d\n", x, *p); // 6.500000 // 100 100
f3(*p);
printf("x is %d and *p is %d\n", x, *p);

// 100 100

f4(p);
printf("x is %d and *p is %d\n", x, *p, p);

int z= 10;
p =f 5(&z);
printf("z is %d and %d\n", *p, z); // 10 10
p = f6();
printf("*p is %d \n", *p);

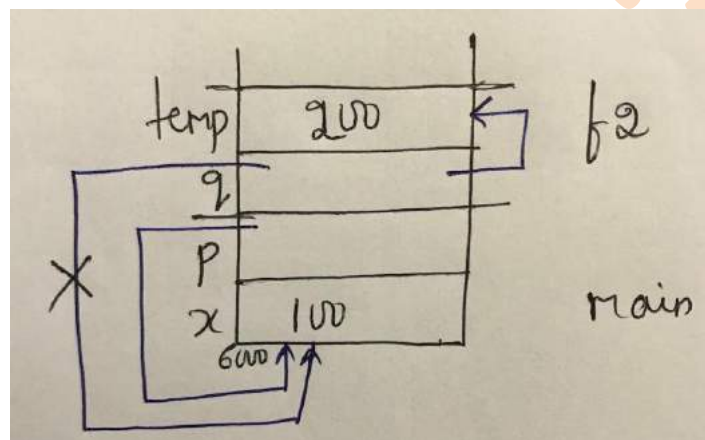
}
void f1(int x)
{
```

```

    x = 20;
}
void f2(int* q)
{
    int temp = 200;

    q = &temp;
}

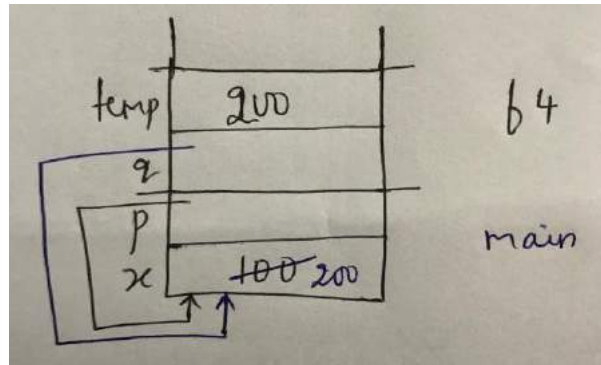
```



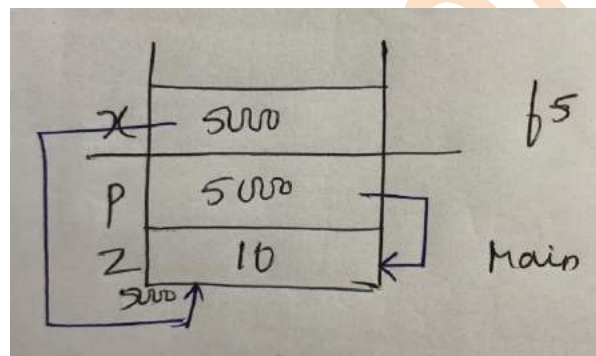
```

void f3(int t)
{
    t = 200;
}
void f4(int* q)
{
    int temp = 200;
    *q = temp;
}

```

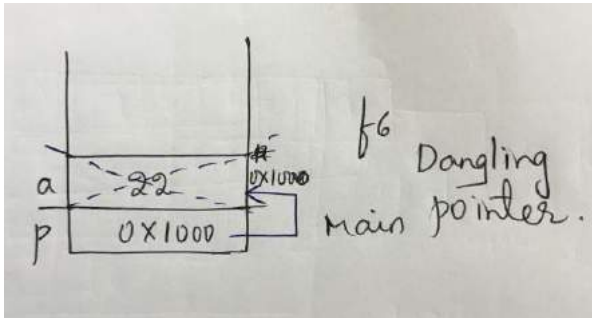


```
int* f5(int* x)
{
    return x;
}
```



```
int* f6()
{
    int a = 22;
    return &a;
}
```

// We should never return a pointer to a local variable



When there is function call to f6, Activation Record gets created for that and deleted when the function returns. p points to a variable which is not available after the function execution. This problem is known as **Dangling Pointer**. The pointer is available. But the location it is not available which is pointed by pointer. Applying **dereferencing operator(*)** on a **dangling pointer** is **always** undefined behaviour.

Recursion

A function may call itself directly or indirectly. The function calling itself with the termination/stop/base condition is known as recursion. Recursion is used to solve various problems by dividing it into smaller problems. We can opt if and recursion instead of looping statements. In recursion, we try to express the solution to a problem in terms of the problem itself, but of a smaller size.

Consider the below code.

```
int main()
{
    printf("Hello everyone\n");
    main();
    return 0;
}
```

Execution starts from main() function. Hello everyone gets printed. Then call to main() function. Again, Hello everyone gets printed and these repeats. we have not defined any condition for the program to exit, results in Infinite Recursion. In order to prevent infinite recursive calls, we need to define proper base condition in a recursive function.

Let us write Iterative and Recursive functions to find the Factorial of a given number. client.cis given as below.

```
int fact(int n);

int main()
{
    int n = 6;
    printf("factorial of %d is %d\n",fact(n));
}
```

Iterative Implementation:

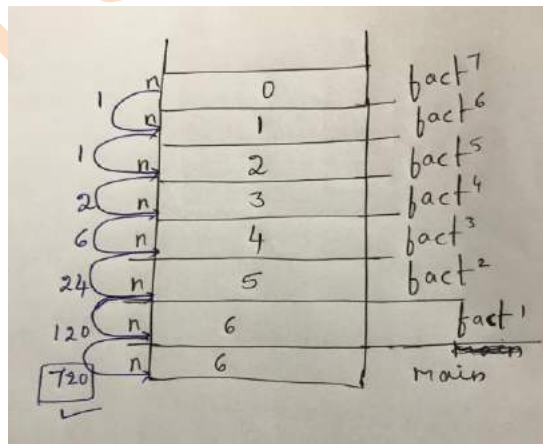
```
int fact(int n)
{
    int result = 1;
    int i;
    for (i = 1; i <= n; i++)
    {
        result = result * i;
    }
    return result;
}
```

Recursive Implementation:

Logic: If n is 0 or n is 1, result is 1

Else, result is $n * (n-1)!$

```
int fact(int n)
{
    if (n == 0)
        return 1;
    else
    {
        return n * fact(n-1);
    }
}
```

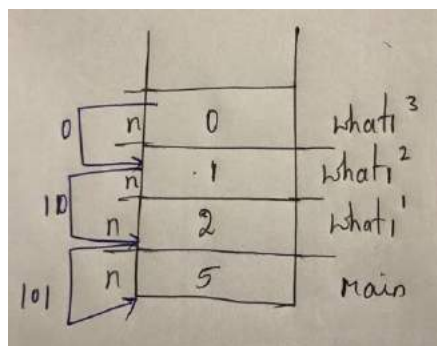


Let us start tracing the given recursive function

Example 1: What this function does?

```
int what1(int n){
    if (n == 0) {    return 0; }
    else
    {    return (n % 2) + 10 * what1(n / 2); }
}
```

// if n is 5, refer to the diagram.



This code prints the given number in binary.

Example 2:

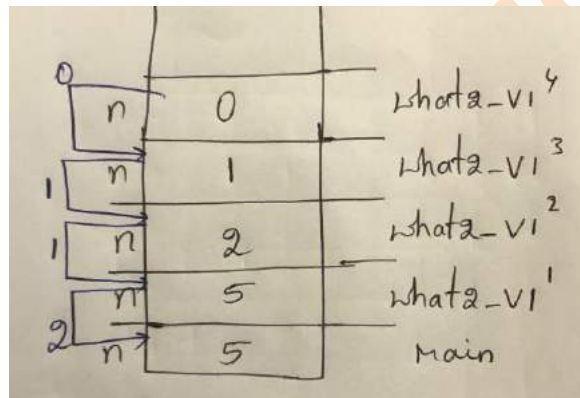
Version 1:

```
int what2_v1(int n)
{
    if (n == 0)
        return 0;
    else
        return (n & 1) + what2_v1(n >> 1);
}
```

//Finds the number of 1s in the binary representation of the number

Version 2:

```
int what2_v2(int n)
{
    if(!n)
    {
        return 0;
    }
    else if(!(n % 2))
    {
        return what2_v2(n / 2);
    }
    else
    {
        return 1+what2_v2(n / 2);
    }
}
```



Try to trace below with diagram.

Example 3:

Version 1:

```
int what3_v1(int b,int p)
{
    int result=1;
    if(p==0)
    {
        return result;
    }
    result=b*(what3_v1(b,p-1));
}
```

Version 2:

```
int what3_v2(int b, int p)
{
    if(!p)
        return 1;
    else if(!(p % 2))
        return what3_v2(b*b, p/2);
    else
        return b*what3_v2(b*b,p/2);
}
```

Example 3 Finds b to the power p

Example 4:

```
int what4(int n)
{
    if (n>0)
        return n + what4(n - 1);
    else if (n==0)
        return 0;
    else
        return -1;
}
```

If the given number is n, Finds $n+(n-1)+(n-2)+ \dots +0$

If n is -ve, returns -1.

PES University

Interface and Implementation

Interface means Function declaration. Implementation means Function definition. Interface tells us what the function requires while calling. It does not tell us anything about how that function works. Implementation always refers to the body of the function. If any change in the body of the function, it does not affect outside. The one who is using the function, need not worry about the change in implementation.

Let us say, requirement is to check whether a given number is a palindrome or not.

Version 1:

```
#include<stdio.h>
#include "palin.h"
int main()
{   int n;

    printf("Enter the number\n");
    scanf("%d", &n);

    int rev = 0;
    int number = n;
    while(n>0)    // can be just while(n). When n is 0, loop terminates.
    {   rev = rev * 10 + n % 10;
        n /= 10;
    }

    if(number == rev)
        printf("%d is palindrome\n",number);
    else
        printf("%d is not palindrome\n",number);
}
```

Version 2:

Now we will write the function to check whether the given number is

palindrome or not. #include<stdio.h>

```
int is_palin(int);           // interface
```

```
int main()
```

```
{    int n;
```

```
    printf("Enter the number\n");
```

```
    scanf("%d", &n);
```

```
    if(is_palin(n)) //implementation in next page
```

```
    {
```

```
        printf("%d is a palindrome\n", n);
```

```
    }
```

```
else
```

```
{
```

```
    printf("%d is not a palindrome\n", n);
```

```
}
```

```
}
```

```
int is_palin(int n)                // implementation
{
    int rev = 0;
    int number = n;

    while(n>0)
    {
        rev = rev * 10 + n %
        10;n /= 10;
    }
    return number == rev;
}
```

You might want to write reverse function which reverses the given number and the returnvalue of that can be used for checking inside the is_palin function. Try it!

Version 3:

Differentiate between client code, server code and the header files.

Declarations of functions in palin.h

```
int is_palin(int n);
```

Then we write the client code [where the execution starts – main() in C] as below. client.c contains the below code.

```
#include<stdio.h>
#include "palin.h"        // interface

int main()
{
    int n;
    printf("Enter the number\n");
```

```
scanf("%d", &n);

if(is_palin(n)) //defition in palin.c ( next page )
{
    printf("%dis a palindrome\n", n);
}
else
{
    printf("%d is not a palindrome\n", n);
}
}
```

Definitions of User defined functions are saved in palin.c

```
int is_palin(int n)           // implementation
{
    int rev = 0;
    int number = n;
    while(n>0)
    {
        rev = rev * 10 + n % 10;
        n /= 10;
    }
    return number == rev;
}
```

Commands to execute above codes is as below.

```
gcc -c client.c           // this creates client.o
gcc -c palin.c            // this creates palin.o
gcc client.o palin.o      // this is for linking. We get the loadable image
a.exe or a.out
a.exe or ./a.out         // press enter to see the result or output
```

If I make some changes in the implementation file i.e., palin.c, I need to compile only that. Then link palin.o and client.o to get the executable.

If there are one or two implementation files, the changed files can be recompiled and relinked easily. But, if you have many implementation files and you have made modifications to some of these, you need to remember which all files you need to compile again. Or Compile all files and link all object files again. This is waste of time. Rather, we have a facility called make which completes this requirement in an easier way.

Usage of make command

make is Unix utility that is designed to start execution of a makefile. A makefile is a special file, containing shell commands, that you create and name it with .mk extension preferably. While in the directory containing this makefile, you will type make and the commands in the makefile will be executed.

A makefile that works well in one shell may not execute properly in another shell. Because it must be written for the shell which will process the makefile.

The makefile contains a list of rules. These rules tell the system what commands you want to be executed. These rules are commands to compile (or recompile) a series of files. **The rules, which must begin in column 1, are specified in two types of lines.** The first line is called a **Dependency line** and the subsequent line(s) are called **Action lines**. **The action line(s) must be indented with a tab.**

The dependency line is made of two parts. The first part (before the colon) are **Target files** and the second part (after the colon) are called **Source files**. It is called a dependency line because the **first part depends on the second part**. Multiple target files must be separated by a space. Multiple source files must also be separated by a space.

Make reads the makefile and creates a dependency tree and takes whatever action is necessary. It will not necessarily do all the rules in the makefile as all dependencies may not need updated. It will rebuild target files if they are missing or older than the dependency files. It keeps track of the recent time files (normally object files) were updated and only updates those files which are required (ones containing changes) to keep the source file up-to-date. If you have a large program with many source and/or header files, when you change a file on which others depend, you must recompile all the dependent files. Without a makefile, this is an

extremely time-consuming task.

Command to execute on **Ubuntu**:

```
make -f filename.mk // -f to read file as a make file
```

Windows: you need to download this utility.

If you have installed gcc using mingw package, go to mingw/bin folder in command prompt and type as below.

```
mingw-get install mingw32-make // press enter
```

Command to execute on **Windows**

```
mingw32-make -f filename.mk // -f to read file as a make file
```

Corresponding Make file for palindrome checking code is as below.

```
a.out : client.o palin.o
gcc client.o palin.o client.o : client.c palin.h
gcc -c client.c palin.o : palin.c palin.h
gcc -c palin.c
```

3.1 Passing Array to a Function

When array is passed as an argument to a function, arguments are copied to parameters of the function and parameters are always pointers. Array degenerates to a pointer at runtime. All the operations that are valid for pointer will be applicable for array too in the body of the function. Function call happens always at run time.

```
#include<stdio.h>

int main()
{
    int arr[100]; int n;
    printf("Enter the number of elements you want to
store\n");scanf("%d",&n);
    printf("enter %d elements\n",n);
    read(arr,n);
    printf("entered elements are\n");
    display(arr,n);
}

void read(int arr[],int n)    // arr is an array. But it becomes pointer at runtime
{
    // check the sizeof(arr). It is same as size of pointer

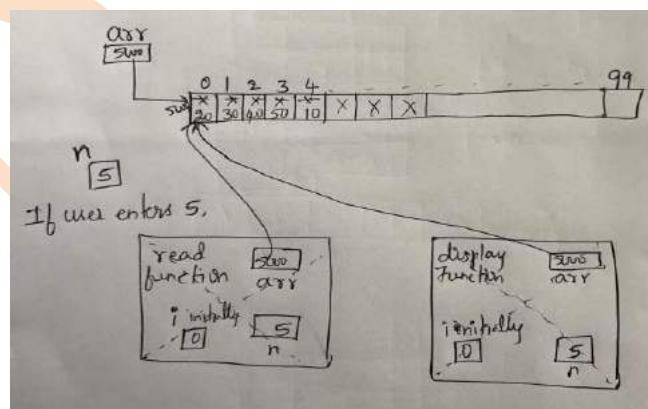
    int i = 0;
    while(i<n)
    {
        scanf("%d",&arr[i]); i++;
    }
}

void display(int arr[],int n)
{
    int i = 0;
    while(i<n)
    {
        printf("%d",arr[i]);
        i++;
    }
}
```

So, use **pointer variable** in the parameters of the function definition when **array is passed as the argument**.

```
void read(int *arr , int n)
{
    int i = 0;
    while(i<n)
    {
        scanf("%d",&arr[i]);
        i++;
    }
}

void display(int *arr , int n)
{
    int i = 0;
    while(i<n)
    {
        printf("%d",arr[i]);
        i++;
    }
}
```



You can write this code using client and server having a clear line between interface and implementation. Use make files and make command for the same.

Let us try to pass array as an argument and create one more array using this array in the mainfunction

```
//int a[] read(int arr[],int n);      //return type cannot be
arrayint* read(int arr[],int n);
void display(int *arr, int
n);int main()
{
    int arr[100];
    int n;

    printf("Enter the number of elements you want to store\n");
    scanf("%d",&n);
    printf("enter %d elements\n",n);
    // int b[] = read(arr,n);// this throws compile-time Error. Cannot return an
    array

    // So change b to pointer and return type of function to pointer.
    int *b = read(arr,n);
    printf("entered elements in arr\n");
    display(arr,n);
    printf("Elements in b\n");
    display(arr,n);
    return 0;
}

int* read(int *arr, int n)          // observe return type int*
{
    int i = 0;
    while(i<n)
    {
        scanf("%d",&arr[i]); i++;
    }
    return arr;
}
```

```
void display(int *arr, int n)
{
    int i = 0;
    while(i<n)
    {
        printf("%d\t",arr[i]); i++;
    }
}
```

PES University

PES University

Arrays, Initialization and Traversal

When solving problems, it is important to visualize the data related to the problem. Sometimes the data consist of just a single number (as we discussed in unit-1, primitive types). At other times, the data may be a coordinate in a plane that can be represented as a pair of numbers, with one number representing the x-coordinate and the other number representing the y-coordinate. There are also times when we want to work with a set of similar data values, but we do not want to give each value a separate name. For example, we want to read marks of 1000 students and perform several computations. To store marks of thousand students, we do not want to use 1000 locations with 1000 names. To store group of values using a single identifier, we use a data structure called an Array.

“An array is a linear data structure, which is a finite collection of similar data items stored in successive or consecutive memory locations.”

Characteristics/Properties of arrays:

- Non-primary data or secondary data type
- Memory allocation is contiguous in nature
- Elements need not be unique.
- Demands same /homogenous types of elements
- Random access of elements in array is possible
- Elements are accessed using index/subscript
- Index or subscript starts from 0
- Memory is allocated at compile time.
- Size of the array is fixed at compile time. Returns the number of bytes occupied by the array.
- Cannot change the size at runtime.
- Arrays are assignment incompatible.
- Accessing elements of the array outside the bound can have undefined behaviour at runtime.

Types of Array:

Broadly classified into two category as

Category 1:

1. Fixed Length Array: Size of the array is fixed at compile time
2. Variable Length Array - Not supported by older few C standards. We are not discussing as a part of this unit.

Category 2:

1. One Dimensional Array
2. Muti-Dimensional Array-Discussed in unit-4

One dimensional array:

- One dimensional array is also called as linear array(also called as 1-D array).
- The one dimensional array stores the data elements in a single row or column.

Array Declaration and Initialization:**Declaration:****Syntax:**

Data_type Array_name[Size]; // Declaration syntax

Example: int marks [5];

- Type : Specifies the type of the element that will be contained in the array
- Size : Indicates the maximum number of elements that can be stored inside the array
- Array_name: Identifier to identify a variable.
- Subscripts in array can be integer constant or integer variable or expression that yields integer
- C performs no bound checking – care should be taken to ensure that the array indices are within the declared limits

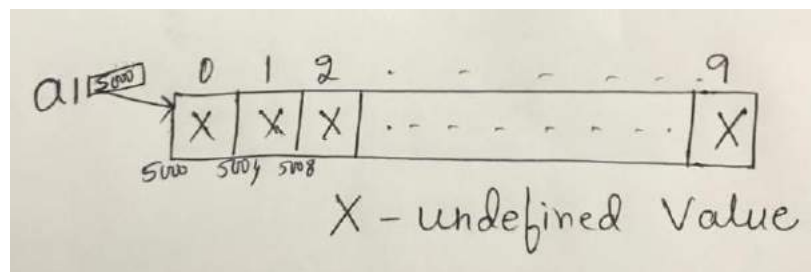
Example: float average [6];

- can contain 6 floating point elements, 0 to 5 are valid array indices
char name[20];
- can contain 20 char elements, 0 to 19 are valid array indices
double x[15];
- Can contain 15 elements of type double, 0 to 14 are valid array indices.

Consider int a1[10];

Declaration allocates number of bytes in a contiguous manner based on the size specified. All these memory locations are filled with undefined values. If the starting address is 0x5000 and if the size of integer is 4 bytes, refer the diagrams below to understand this declaration clearly. Number of bytes allocated = size specified * size of integer i.e, 10 * 4.

```
printf("size of array is %d\n", sizeof(a1)); // 40 bytes
```



Address of the first element is called the Base address of the array. Address of ith element of the array can be found using formula:

$$\text{Address of } i^{\text{th}} \text{ element} = \text{Base address} + (\text{size of each element} * i)$$

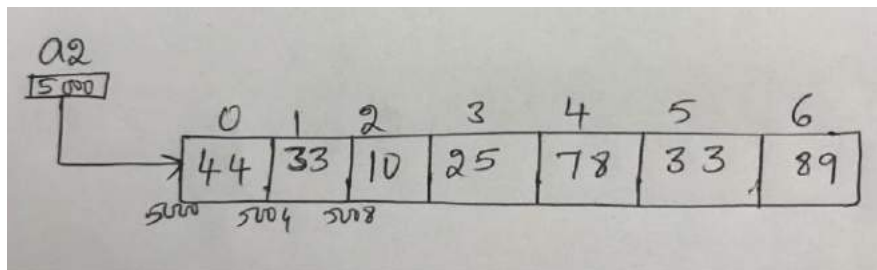
Initialization of an Array

- After an array is declared it must be initialized.
- An Uninitialized array will contain undefined values.
- An array can be initialized at either compile time or at runtime

Consider int a2[] = {44,33,10,25,78,33,89};

Above initialization does not specify the size of the array. Size of the array is based on the number of elements stored in it and the size of type of elements stored. So, the above array occupies $7 \times 4 = 28$ bytes of memory in a contiguous manner.

```
printf("size of array is %d\n", sizeof(a2)); //28 bytes
```



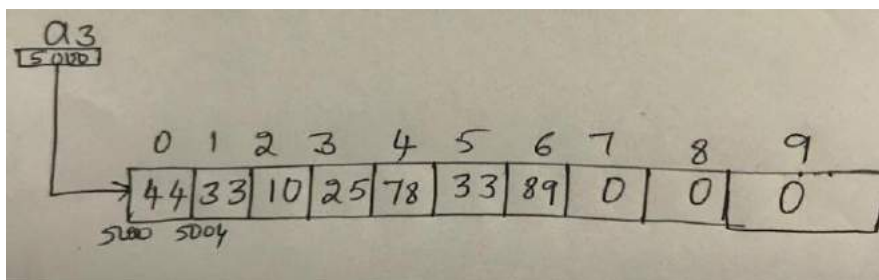
Partial Initialization:

Consider `int a3[10] = {44,33,10,25,78,33,89};`

Size of the array is specified. But only few elements are stored. Now, the size of the array is $10 \times 4 = 40$ bytes of memory in a contiguous manner. **All uninitialized memory locations in the array are filled with default value 0.**

```
printf("size of array is %d\n", sizeof(a3)); //40 bytes
```

```
printf("%p %p\n", a3, &a3); // Must be different. But shows same address
```



Compile time Array initialization:

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

Syntax: 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]={67,87,56,77,5} //undefined behaviour
```

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] = {0,0,29,0,0,0,0,0,7,0,0,0,48};
```

- 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.
- 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};
```

Runtime initialization:

- Using a loop and input function in c

```
Example: int a[5];
```

```
for(int i=0;i<5;i++)
```

```
{
```

```
    scanf("%d",&a[i]);
```

```
}
```

Traversal of the Array:

Accessing each element of the array is known as traversing the array.

Consider `int a1[10];`

How do you access the 5th element of the array? **a1[4]**

How do you display each element of the array?

```
int i;
```

```
for(i = 0; i<10;i++)
```

```
    printf("%d\t",a1[i]); //Using the index operator [ ].
```

Above code prints undefined values as a1 is just declared.

Now Consider

```
int a2[ ] = {12,22,44,14,77,911};
```

```
int i;
```

```
for(i = 0; i<10;i++)
```

```
    printf("%d\t",a2[i]); //Using the index operator [ ].
```

Above code prints initialized values when i is between 0 to 5. When i becomes 6, a2[6] is outside bound as the size of a2 is fixed at compile time and it is 6*4(size of int is implementation specific) = 24 bytes. **Anytime accessing elements outside the array bound is an undefined behaviour.**

Example Code 1:

Let us consider the program to read 10 elements from the user and display those 10 elements.

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

```
int main()
```

```
{
```

```
    int arr[100]; int n;
```

```
    printf("enter the number of elements\n");
```

```
    scanf("%d",&n);
```

```
    printf("enter %d elements\n",n);
```

```
    int i = 0;
```

```
    while(i<n)
```

```
    {
```

```
        scanf("%d",&arr[i]);  
        i++;  
    }  
    printf("entered elements are\n");  
    i = 0;  
    while(i<n)  
    {  
        printf("%d\n",arr[i]);  
        i++;  
    }  
    return 0;  
}
```

Example Code 2:

Let us consider the program to find the sum of the elements of an array.

```
int arr[] = {-1,4,3,1,7};  
  
int i;  
  
int sum = 0;  
  
int n = sizeof(arr)/sizeof(arr[0]);  
  
for(i = 0; i<n; i++)  
{    sum += arr[i];  
}  
  
printf("sum of elements of the array is %d\n", sum);
```

// Think about writing a function to find the sum of elements of the array.

Few questions to think about above codes?

- Should I use while only? for and while are same in C except for syntax.
- Is there any clear demarcation between interface and implementation? No. Everything is in the same file i.e. client . Try to separate these two
- Can I use arr[n] while declaration? Variable length array
- Can we write read and display user defined functions to do the same? Yes. To do this, we should understand functions and how to pass array to a function.



Unit II: Arrays, Initialization and Traversal

2021

Pointers

- Pointer is a variable which contains the address. This address is the location of another object in the memory.
- Pointers can be used to access and manipulate data stored in memory.
- Pointer of particular type can point to address any value of that particular type.
- Size of pointer of any type is same/constant in that system.
- Not all pointers actually contain an address

Example: NULL pointer // Value of NULL pointer is 0.

- Pointer can have three kinds of contents in it
 1. The address of an object, which can be dereferenced.
 2. A NULL pointer
 3. Undefined value // If p is a pointer to integer, then – int *p;

Pointer Declaration:

Syntax:

Data-type *name;

Example: int *p;

- Compiler assumes that any address that it holds points to an integer type.

p= ∑ // Memory address of sum variable is stored into p.

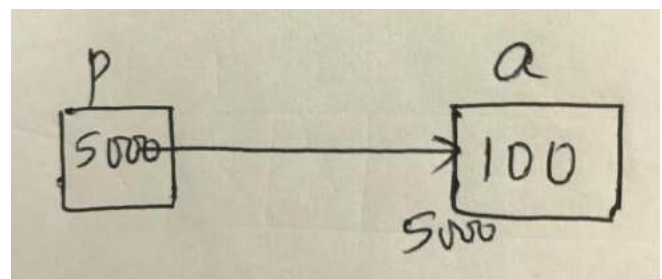
Consider,

int *p; // p can point to anything where integer is stored. int* is the type. Not just int.

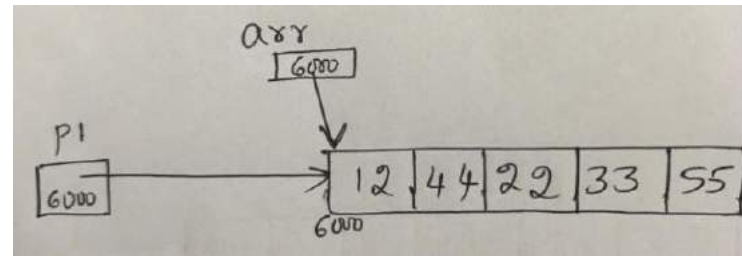
int a = 100;

p=&a;

printf("a is %d and *p is %d", a,*p);



```
Now, int arr[] = {12,44,22,33,55};
int *p1 = arr; // same as int *p1; p1 = arr;
           // same as int *p1; p1 = &arr[0];
```



```
int arr2[10];
```

```
arra2 = arr; // Arrays are assignment incompatible. Compile time Error
```

Pointer Arithmetic:

Below arithmetic operations are allowed on pointers

- Add an int to a pointer
- Subtract an int from a pointer
- Difference of two pointers when they point to the same array.

Integer is not same as pointer. We get warning when we try to compile the code where in integer is stored in variable of int* type.

```
int arr[ ] = {12,33,44};
```

```
int *p2 = arr;
```

```
printf("before increment %p %d\n",p2, *p2);    // 12
```

```
p2++; //same as p2 = p2+1 // This means 5000+sizeof(every element)*1 if 500 is the base address
```

```
//increment the pointer by 1. p2 is now pointing to next location.
```

```
printf("after increment %p %d\n",p2, *p2); // 33
```

Example on Pointer Arithmetic :

```
int *p, x = 20;

p = &x;

printf("p    = %p\n", p);
printf("p+1 = %p\n", (int*)p+1);
printf("p+1 = %p\n", (char*)p+1);
printf("p+1 = %p\n", (float*)p+1);
printf("p+1 = %p\n", (double*)p+1);
```

Sample output:

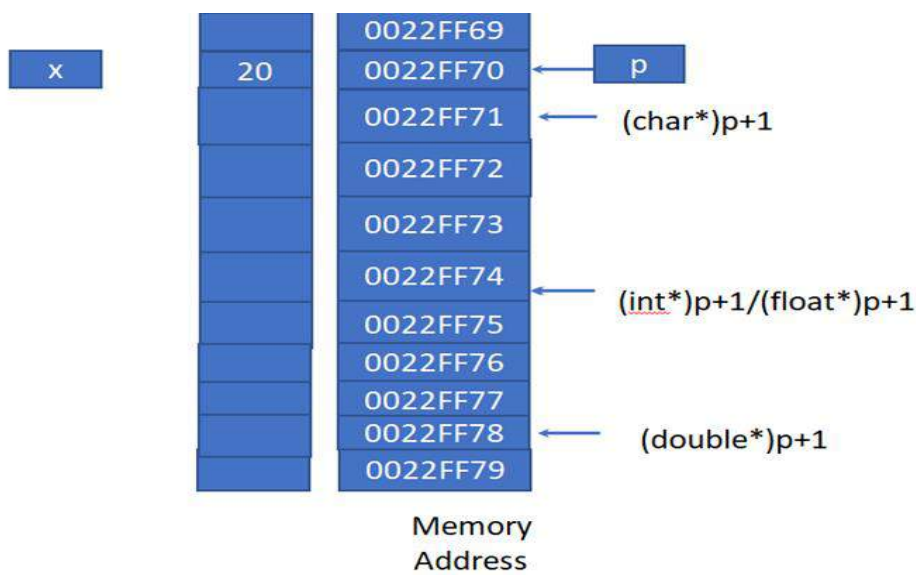
p = 0022FF70

p+1 = 0022FF74

p+1 = 0022FF71

p+1 = 0022FF74

p+1 = 0022FF78



Example:

```
int main()
```

```
{
```

```
    int *p;
```

```
    int a = 10;
```

```
    p = &a;
```

```
    printf("%d\n",(*p)+1); // 11 ,p is not changed
```

```
    printf("before *p++ %p\n",p); //address of p
```

```
    printf("%d\n",*p++); // same as *p and then p++ i.e 10
```

```
    printf("after *p++ %p\n",p); //address incremented by the size of type of value stored
```

```
in it
```

```
    return(0);
```

```
}
```

```
int main()
```

```
{
```

```
    int *p;
```

```
    int a = 10;
```

```
    p = &a;
```

```
    printf("%d\n",*p); //10
```

```
    printf("%d\n",(*p)++); // 10 value of p is used and then value of p is incremented
```

```
    printf("%d\n",*p); // 11
```

```
    return 0;
```

```
}
```

Array Traversal using pointers:

```
int arr[] = {12,44,22,33,55};
```

```
int *p3 = arr;
```

```
int i;
```

Version 1: Index operator can be applied on pointer. Array notation

```
for(i = 0; i < 5; i++)
```

```
    printf("%d \t", p3[i]); // 12 44 22 33 55
```

```
    // every iteration added i to p3 . p3 not modified
```

Version 2: Using pointer notation

```
for(i = 0; i < 5; i++)
```

```
    printf("%d \t", *(p3+i)); // 12 44 22 33 55
```

```
    // every iteration i value is added to p3 and content at that  
    address is printed. p3 not modified
```

Version 3:

```
for(i = 0; i < 5; i++)
```

```
    printf("%d \t", *p3++); // 12 44 22 33 55
```

```
    // Use p3, then increment, every iteration p3 is incremented.
```

Version 4: undefined behaviour if you try to access outside bound

```
for(i = 0; i < 5; i++)
```

```
    printf("%d \t", *++p3); // 44 22 33 55 undefined value
```

```
    // every iteration p3 is incremented.
```

Version 5:

```
for(i = 0;i<5;i++)  
    printf("%d \t",(*p3)++); // 12 13 14 15 16  
// every iteration value at p3 is used and then incremented.
```

Version 6:

```
for(i = 0;i<5;i++,p3++)  
    printf("%d \t",*p3); // 12 44 22 33 55  
// every iteration value at p3 is used and then p3 is incremented.
```

Version 7: p3 and arr has same base address of the array stored in it. But array is a constant pointer. It cannot point to anything in the world.

```
for(i = 0;i<5;i++)  
    printf("%d \t", *arr++); // Compile Time Error
```

Arrays and Pointers:

- An array during compile time is an actual array but degenerates to a constant pointer during run time.
- Size of the array returns the number of bytes occupied by the array. But the size of pointer is always constant in that particular system.

Examples: int *p1; float *f1 ; char *c1;

```
printf("%d%d%d ",sizeof(p1),sizeof(f1),sizeof(c1)); // Same value for all
```

```
int a[] = {22,11,44,5};
```

```
int *p = a;
```

```
a++; // Error constant pointer
```

```
p++; // Fine
```

```
p[1] = 222; // allowed
```

```
a[1] = 222 ; // Fine
```

```
int *q={23,56,7,8,8}; //warning. Not a good idea to initialize a set of elements to a pointer
```

```
q[1]=500 ; //undefined behaviour
```

If variable i is used in loop for the traversal, a[i], *(a+i), p[i], *(p+i), i[a], i[p] are all same.

Differences:

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

```
int a[10];
```

```
int *p;
```

```
p=a; //allowed
```

```
a=p; //not allowed
```

5. An arithmetic operation on pointer variable is allowed.

```
int a[10];
```

```
int *p;
```

```
p++; /*allowed*/
```

```
a++; /*not allowed*/
```


Passing Array to a Function:

When array is passed as an argument to a function, arguments are copied to parameters of the function and parameters are always pointers. Array degenerates to a pointer at runtime. All the operations that are valid for pointer will be applicable for array too in the body of the function. Function call happens always at run time.

Array as a formal parameter and actual parameter

- Array being formal parameter - Indicated using empty brackets in the parameter list.

```
void myfun(int a[],int size);
```

- Array being actual parameter – Indicated using the name of the array

```
Array a is declared as int a[5];
```

```
Then myfun is called as myfun(a,n);
```

Example: Program to read and display

```
void read(int arr[],int); //function prototype
```

```
void display(int arr[],int); //function prototype
```

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

```
int main()
```

```
{
```

```
    int arr[100]; int n;
```

```
    printf("Enter the number of elements you want to store\n");
```

```
    scanf("%d",&n);
```

```
    printf("enter %d elements\n",n);
```

```
    read(arr,n);
```

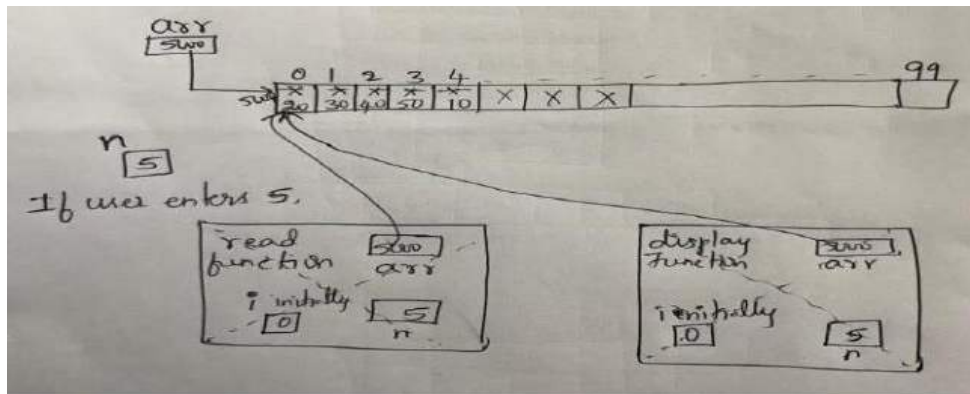
```
    printf("entered elements are\n");
```

```
        display(arr,n);
        return(0);
    }
//function definition
void read(int arr[],int n)
{
    int i = 0;
    while(i<n)
    {
        scanf("%d",&arr[i]);
        i++;
    }
}
//function definition
void display(int arr[],int n)
{
    int i = 0;
    while(i<n)
    {
        printf("%d",arr[i]);
        i++;
    }
}
```

So, use **pointer variable in the parameters of the function definition when array is passed as the argument.**

```
void read(int *arr , int n)
{
    int i = 0;
    while(i<n)
    {
        scanf("%d",&arr[i]); i++;
    }
}

void display(int *arr , int n)
{
    int i = 0;
    while(i<n)
    {
        printf("%d",arr[i]);
        i++;
    }
}
```



you can write this code using client and server having a clear line between interface and implementation. Use make files and make command for the same.

Let us try to pass array as an argument and create one more array using this array in the main function

```
//int a[] read(int arr[],int n);    //return type cannot be array

int* read(int arr[],int n);

void display(int *arr, int n);

int main()
{
    int arr[100];

    int n;

    printf("Enter the number of elements you want to store\n");
    scanf("%d",&n);

    printf("enter %d elements\n",n);

    //int b[] = read(arr,n); // this throws compile-time Error. Cannot return an array
```

```
//So change b to pointer and return type of function to pointer.

int *b = read(arr,n);

printf("entered elements in arr\n");

display(arr,n);

printf("Elements in b\n");

display(arr,n);

return 0;

}

int* read(int *arr, int n) // observe return type int*
{
    int i = 0;
    while(i<n)
    {
        scanf("%d",&arr[i]);
        i++;
    }
    return arr;
}

void display(int *arr, int n)
{
```

```
int i = 0;

while(i<n)

{

    printf("%d\t",arr[i]);

    i++;

}

}
```

Passing individual array elements to a function:

- Indexed variables can be arguments to functions.
- Program contains these declarations as

```
int a[10];
```

```
int i;
```

```
myfunc(int n);
```

- Variables a[0] through a[9] are of type int, making below calls is legal

```
myfunc(a[0]); //passing first element
```

```
myfunc(a[3]); //passing second element
```

```
myfunc(a[i]);
```

Example:

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

```
void display(int age1, int age2)
```

```
{
```

```
    printf("%d\n", age1);
```

```
printf("%d\n", age2);  
}
```

```
int main()  
{  
    int ageArray[] = {2, 8, 4, 12};  
    display(ageArray[1], ageArray[2]); // Passing second and third elements to display()  
    return 0;  
}
```

Const Modifier

- Using the const modifier on a variable or array tells the compiler that the contents will not be changed by the program.

Example: int main()

```
{  
    const int i=10;  
    const int j=i+10; //fine  
    i++; //error  
    return(0);  
}
```

- Array parameters allow a function to change the values stored in the array argument.
- If a function should not change the values of the array argument, use the modifier **const**
- An array parameter modified with const is a **constant array parameter**.

Example:

```
void display(const int a[ ], int size);
```

Program to display an constant array

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

```
void display(const int *,int);//prototype
```

```
int main()
```

```
{
```

```
    int number[]={22,55,66,77,88};
```

```
    display(number,5);
```

```
    return(0);
```

```
}
```

```
void display(const int *num, int size)
```

```
{
```

```
    for(int i=0;i<size;i++)
```

```
    {
```

```
        printf("%d\t",num[i]);
```

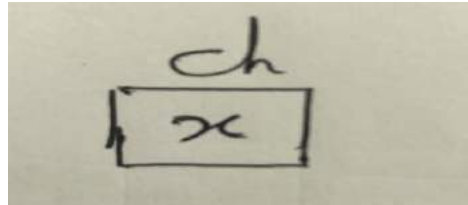


```
}  
num[1]=100 ;//compiler error  
}
```

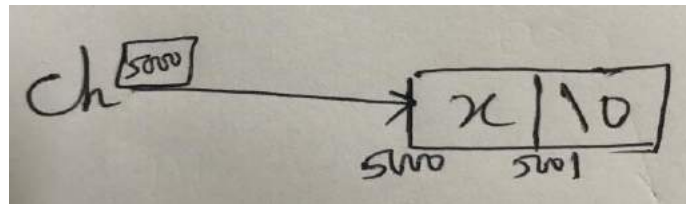
String in C

A string in C is an array of characters and terminated with a special character '\0' or NULL. ASCII value of NULL character is 0. Each character occupies 1 byte of memory. There is **NO** datatype available to represent a string in C.

`char ch = 'x' ;` // character constant in single quote. always of 1 byte



`char ch[] = "x" ;` // String constant. always in double quotes. Terminated with '\0'. Always 1 byte more when specified between “ and ” in initialization.



Declaration and Initialization of String

Declaration:

`char variable_name[size];`

Size of the array must be specified compulsorily.

Initialization:

If the size is not specified, the compiler counts the number of elements in the array and allocates those many bytes to an array.

If the size is specified, it allocates those many bytes and unused memory locations are initialized with default value '\0'. This is partial initialization.

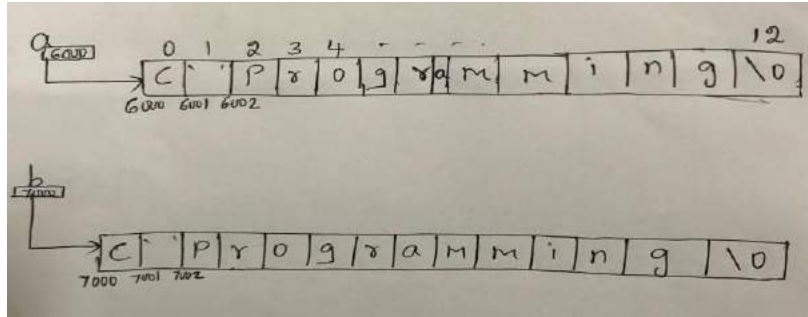
If the string is hard coded, it is programmer's responsibility to end the string with '\0' character.

Accessing outside the bound is an undefined behaviour in both the cases

```
char a[] = { 'C', ' ', 'P', 'r', 'o', 'g', 'r', 'a', 'm', 'm', 'i', 'n', 'g', '\0' };
```

```
char b[] = "C Programming" ;
```

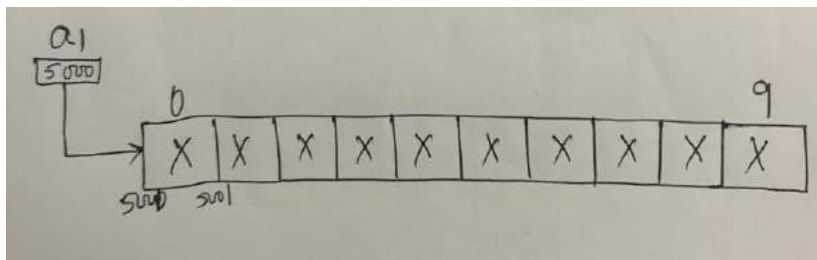
C standard gives shorthand notation to write initializer's list. b is a shorthand available only for storing strings in C.



Let us consider below code segments.

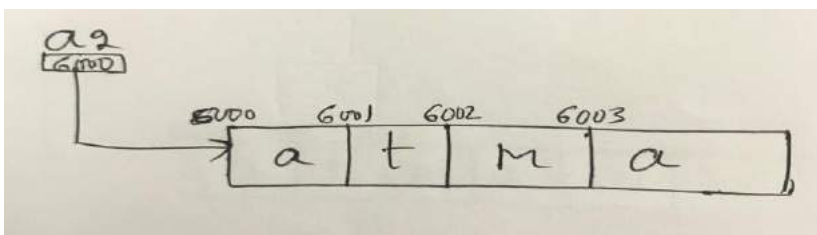
```
char a1[10]; // Declaration: Memory locations are filled with undefined values
```

```
printf("sizeof a1 is %d",sizeof(a1)); // 10
```



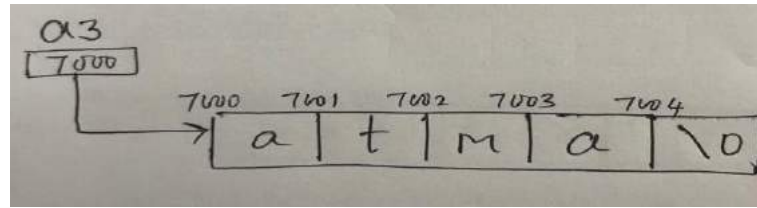
```
char a2[ ] = {'a','t','m','a'}; // Initialization
```

```
printf("sizeof a2 is %d",sizeof(a2)); // 4but cannot assure about a2 while printing
```



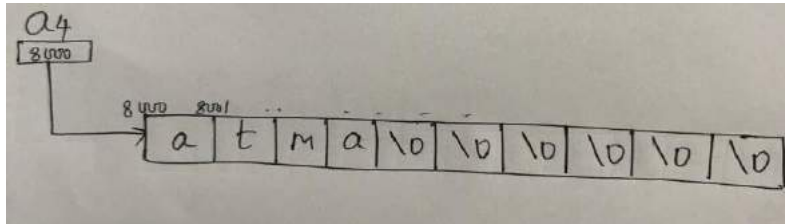
```
char a3[ ] = "atma" ;
```

```
printf("sizeof a3 is %d",sizeof(a3)); // 5sure about a3 while printing
```



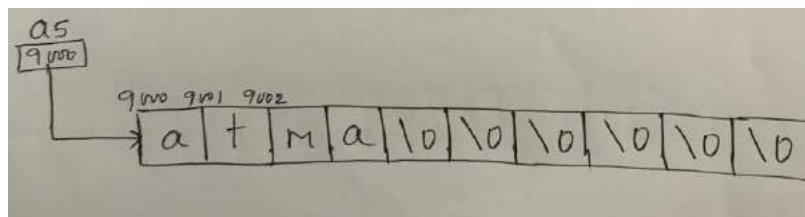
```
char a4[10] = {'a','t','m','a'} // Partial Initialization
```

```
printf("sizeof a4 is %d",sizeof(a4)); // 10 sure about a4 while printing
```



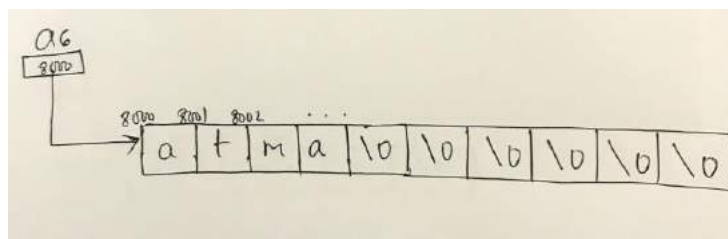
```
char a5[10] = "atma" ;
```

```
printf("sizeof a5 is %d",sizeof(a5)); // 10 sure about a5 while printing
```



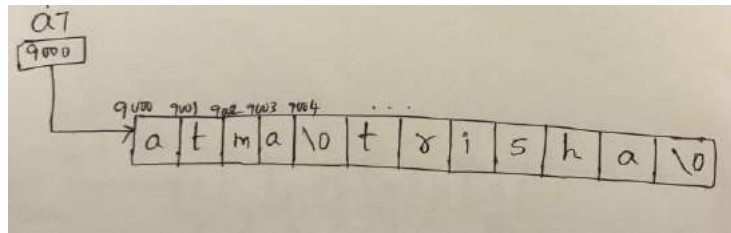
```
char a6[10] = {'a','t','m','a', '\0'};
```

```
printf("sizeof a6 is %d",sizeof(a6)); // 10 sure about a6 while printing
```



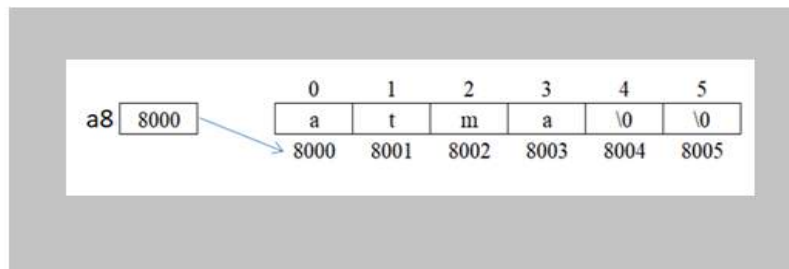
```
char a7[] = {'a', 't', 'm', 'a', '\0', 't', 'r', 'i', 's', 'h', 'a', '\0'};
```

```
printf("sizeof a7 is %d",sizeof(a7)); // 12 a7 will be printed only till first '\0'
```



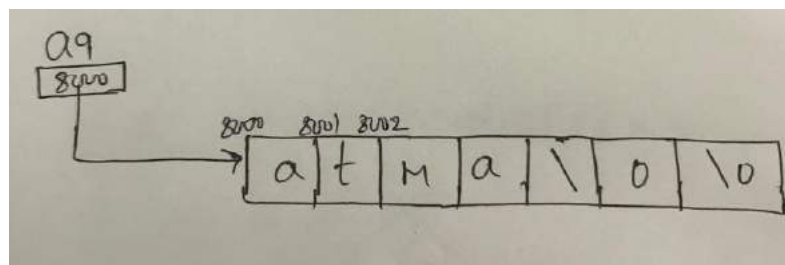
```
char a8[ ] = "atma\0" ;
```

```
printf("sizeof a8 is %d",sizeof(a8));// 6 sure about a8 while printing
```



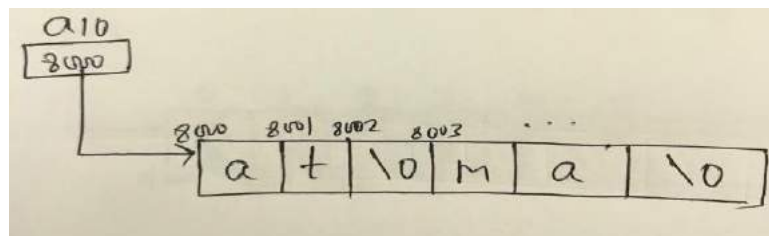
```
char a9[ ] = "atma\\0" ;
```

```
printf("sizeof a9 is %d",sizeof(a9));// 7 sure about a9 while printing
```



```
char a10[ ] = "at\0ma" ;
```

```
printf("sizeof a10 is %d",sizeof(a10));//6 a10 will be printed only till first '\0'
```



Read and Display Strings in C

As usual, the formatted functions: scanf and printf is used to read and display string. %s is the format specifier for string .Consider the below example code.

```
char str1[] = {'a', 't', 'm', 'a', 't', 'r', 'i', 's', 'h', 'a', '\0' };
```

One way of printing all the characters is using putchar in a loop.

```
int i;
for (i = 0; i < sizeof(str1); i++)
    printf("%c", str1[i]); // each element of string is a character. So %c.
```

Think about using while(str1[i] != '\0') rather than sizeof ?

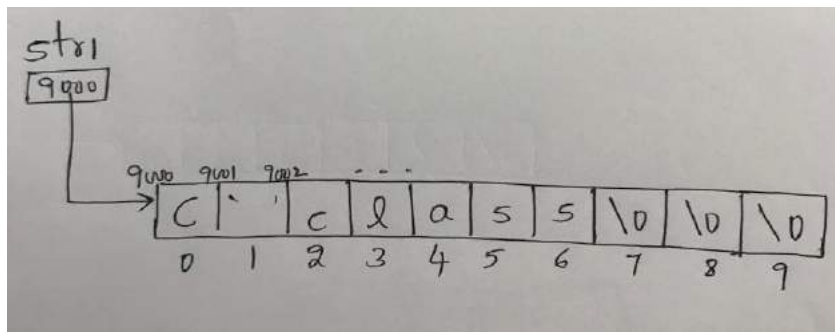
```
while(str1[i] != '\0')
{
    printf("%c", str1[i]);
    i++;
}
```

Also taking each character from the user using getchar() as we did in Unit_1 Last problem solution. Looks like a tedious task. So, use %s format specifier and No loops required.

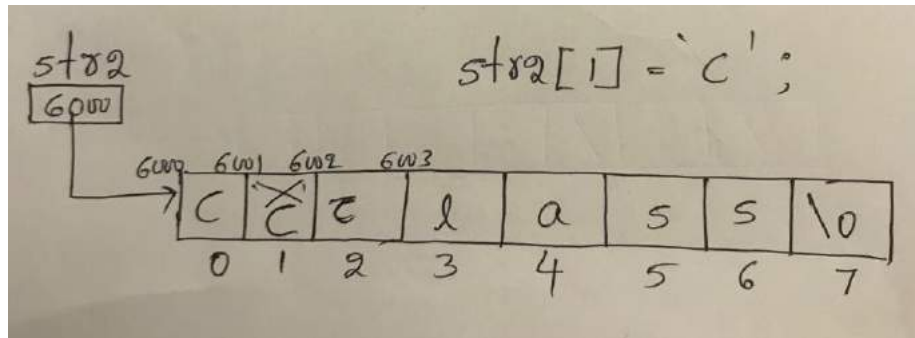
scanf with %s will introduce '\0' character at the end of the string. printf with %s requires the address and will display the characters until '\0' character is encountered.

Consider,

```
char str1[10] = "C class" ;
printf("%s", str1); // C class
```



```
char str2[ ] = "C class" ;
printf("%s\n", str2); // C class
str2[1] = 'C' ;
printf("%s\n", str2); // CCclass
```



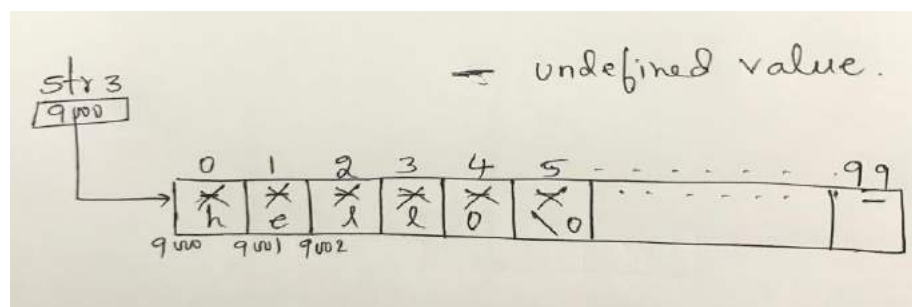
Let us deal with how to take the input from the user for a string variable.

```
char str3[100];
```

```
printf("Enter the string");
```

```
scanf("%s", str3); // User entered Hello Strings and pressed enter key
```

```
printf("%s\n", str3); // Hello. Reason: scanf terminates when white space in the user input
```



```
char str4[100]; char str5[100];
```

```
printf("Enter the string");
```

```
scanf("%s %s", str4, str5); // User entered Hello Strings and pressed enter key
```

```
printf("%s %s", str4, str5); // Hello Strings
```

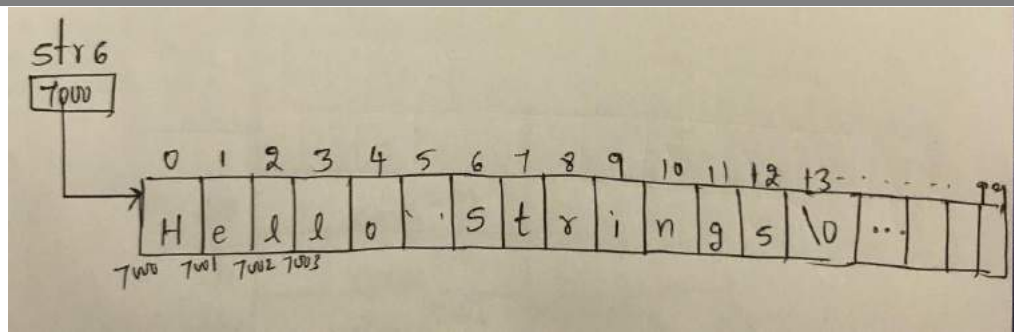
If you want to store the entire input from the user until user presses new line in one character array variable, use `^[^n]` with `%s`. Till `\n` encountered, everything has to be stored in one variable.

```
char str6[100];
```

```
printf("Enter the string");
```

```
scanf("%[^\n]s", str6); // User entered Hello Strings and pressed enter key
```

```
printf("%s", str6); // Hello Strings
```



Pointers v/s String

As array and pointers are related, strings and pointers go hand in hand.

Let us consider few examples.

Example 1:

```
char *p1 = "pesu";
```

//p1 is stored at stack. "pesu" is stored at code segment of memory. It is read only.

```
printf("size is %d\n", sizeof(p1)); // size of pointer
```

// This statement assigns to p variable a pointer to the character array

```
printf("p1 is %s", p1) ; //pesu p1 is an address. %s prints until \0 is encountered
```

```
p++; // Pointer may be modified to point elsewhere.
```

```
printf("p1 is %s", p1) ; //esu
```

```
p1[1] = 'S' ; // No compile time Error
```

```
printf("p1 is %s", p1) ; //Behaviour is undefined if you try to modify the string contents
```

Example 2:

```
char p2[] = "pesu"; // Stored in the Stack segment of memory
```

```
printf("size is %d\n", sizeof(p2)); // 5 = number of elements in array+1 for '\0'
```

```
printf("p2 is %s", p2) ; //pesu
```

```
p2[1] = 'E'; //Individual characters within the array may be modified.
```

```
printf("p2 is %s", p2) ; //pEsu
```

```
p2++; // Compiletime Error
```

// Array always refers to the same storage. So array is a Constant pointer

Built-in String Functions

Even if the string is not a primitive type in C, there are few string related functions available in string.h header file. Include this header file if you are using any functions from this. Result is undefined if '\0' is not available at the end of character array which is passed to builtin string functions. These string functions expect '\0'.

Here are few which are available.

strlen(a) – Expects string as an argument and returns the length of the string, excluding the NULL character

strcpy(a,b) – Expects two strings and copies the value of b to a.

strcat(a,b) – Expects two strings and concatenated result is copied back to a.

strchr(a,ch) – Expects string and a character to be found in string. Returns the address of the matched character in a given string if found. Else, NULL is returned.

strcmp(a,b) – Compares whether content of array a is same as array b. If a==b, returns 0. Returns +ve, if array a has lexicographically higher value than b. Else, -ve.

Few more to think about: strncmp, strcmpi, strncat, strrev, strlwr,strupr, strchr, strstr, strrstr, strset, strnset

Refer to below code segments to understand few builtin string functions.

String Length and String copy

```
char str1 = "pes";
printf("string length is %d\n",strlen(str1));// 3
char a[10]="abcd";
char b[20],c[10];
//b=a;
//we are trying to equate two addresses. Array Assignment incompatible
strcpy(b,a);// copy a to b.
printf("a is %s and b is %s\n",a,b);
printf("length of a is %d and length of b is %d\n",strlen(a),strlen(b));
```

Comparing two strings:

```
char a[10]="abcd";
char b[10]="abcd";
char c[10]="AbCD";
char d[10]="abD";
int i=strcmp(a,b);
int j=strcmpi(a,c); // ignore case
int k=strncmp(a,d,2); // compare only n characters , here 2
printf("first %d--cmp\t%d--cmpi\t%d--ncmp\n",i,j,k);
i=strcmp(a,d);
j=strcmpi(a,d);
k=strncmp(a,d,3);
printf("later %d--cmp\t%d--cmpi\t%d--ncmp\n",i,j,k);
```

String concatenation

```
char a[100]="abcd";
char b[100]="pqr";
char c[100]="whatsapp";
strcat(a,b);// Make sure a is big enough to hold a and b both.
printf("a is %s and b is %s\n",a,b);
printf("length of a is %d and length of b is %d\n",strlen(a),strlen(b));
// make sure size of a is big enough to hold b as well
strncat(a,c,2);
printf("a is %s and size is %d\n",a,strlen(a));
```

Finding character in a given string:

```
char ch[] = "This is a test";
printf("present at %p\n",strchr (buf, 't')); // returns address
```

String operations using User defined functions

Let us write user defined functions to understand few string operations in detail.

Given the client code,

```
char mystr[ ] = "pes";
```

```
printf("Length is %d\n", my_strlen(mystr));
```

Different versions of my_strlen() implementation are as below.

Version 1:

Iterate through every character of the string using a variable count. Stop iteration when NULL character is encountered. Return the value of i.

```
int my_strlen(char str[])  
{  
  
    int i = 0;  
    while(str[i] != '\0')  
    {  
        ++i;  
    }  
    return i;  
}
```

Version 2:

Run the loop till *s becomes '\0' character. Increment the pointer and the counter when *s is not NULL.

```
int my_strlen(char *str)  
{  
    int i=0;  
    while(*str)  
    {  
        // str[i] != '\0'  
        // *(str+i) != '\0'---> all these are same  
        i++;  
        str++;  
    }  
}
```

```
    return i;
}
```

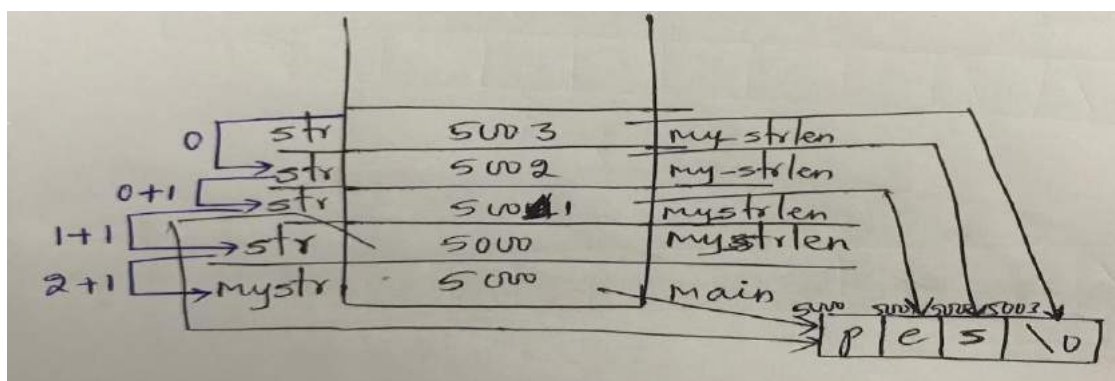
Version 3: Using recursion and pre-increment

```
int my_strlen(char *str)
{
    if (!(*str))
        return 0;
    else
        return 1+my_strlen(++str);
}
```

Version 4: Using recursion and no change in the pointer in the function call

```
int my_strlen(char *str)
{
    if (!(*str))
        return 0;
    else
        return 1+my_strlen(str+1);
}
```

Version 3 and 4 . Same diagram



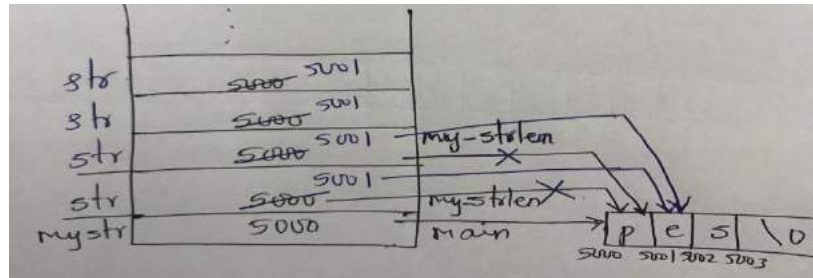
Version 5: Using recursion and post-increment

```
int my_strlen(char *str)
{
    if (!(*str))
        return 0;
```

else

return 1+my_strlen(str++);

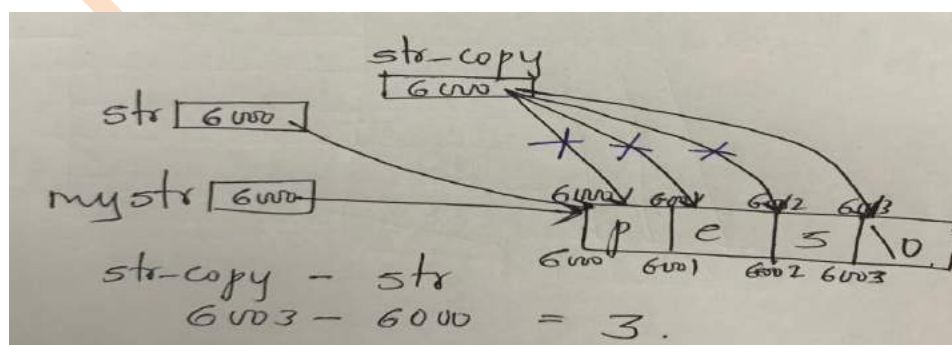
}



Version 6: Using Pointer Arithmetic

Use a local pointer which points to the first character of the string. Keep incrementing this till '\0' is found. Then subtract this from the pointer specified in the parameter which points to the beginning of the string. This finds the length of the string.

```
int my_strlen(char *str)
{
    char *str_copy = str;
    while(str_copy)
    {
        str_copy++;
    }
    return str_copy - str;
}
```



Given the client code,

```
char mystr1[] = "pes university";
char mystr2[100];
printf("%s\n",mystr1);
my_strcpy(mystr2, mystr2);
printf("%s\n",mystr2);
```

Different versions of my_strcpy() implementation are as below.

Version 1:

```
void my_strcpy(char *b, char *a)
{
    // copy a to b
    int i =0;
    while(a[i] != '\0')
    {
        b[i]=a[i];
        i++;
    }
    b[i] = '\0'; // append '\0' at the end
}
```

Version 2: with pointers

```
void my_strcpy(char *b, char *a)
{
    // copy a to b
    while(*a != '\0')
    {
        *b = *a;
        b++;
        a++;
    }
    *b = '\0'; // append '\0' at the end
}
```

Version 3:

```
void my_strcpy(char *b, char *a)
{
    while((*b = *a) != '\0')
```

```

    {
        b++;
        a++;
    }
}

```

Version 4:

```

void my_strcpy(char *b, char *a)
{
    while(*b++ = *a++);    // same as for(;*b++ = *a++;);
}

```

Given the client code,

```

char str1[100];
char str2[100];
printf("enter the first string\n");
scanf("%s",str1);
printf("enter the second string\n");
scanf("%s",str2);
int res = my_strcmp(str1,str2);
printf("result is %d\n",res);
if(!res)
    printf("%s and %s are equal\n",str1,str2);
else if(res > 0)
    printf("%s is higher than %s\n",str1,str2);
else
    printf("%s is lower than %s\n",str1,str2);

```

Different versions of my_strcmp() implementation are as below.

Version 1:

Returns <0 if a<b, 0 if a==b , >0 if a>b

```

int my_strcmp(char *a, char *b)
{

```

```

int i;

for(i = 0; b[i] != '\0' && a[i] != '\0' && b[i] == a[i]; i++);

return a[i]-b[i];

}

```

Version 2: pointer version

```

int my_strcmp(char *a, char *b)
{
    for(;*b && *a && *b == *a; a++,b++);
    return *a - *b;
}

```

Given the client code,

```

char str1[100];
printf("enter the string\n");
scanf("%s",str1);
char ch = getchar();
char *p = my_strchr(str, ch);
printf("present in this address %d",p);
if(p)
    printf("present in %d position\n", p - a);
else
    printf("character not present\n");

```

Different versions of my_strchr() implementation are as below for the logic:

my_strchr returns a pointer to the first occurrence of c that is converted to a character in string.
The function returns NULL if the specified character is not found.

Version 1:

```

char* mystchr(char *a, char c)
{
    char *p = NULL;
    char *s = a;

```



```
while(*s != '\0' && p==NULL)
{
    if(*s == c)
        p = s;
    s++;
}
return p;
}
```

Version 2:

```
char *mystrchr(char *a,char c)
```

```
{
    while(*a && *a != c)
    {
        a++;
    }
    if (!(*a)){
        return NULL;
    } // Can replace if else with return !(*a)? NULL: a;
    else
    {
        return a;
    }
}
```

Given the client code,

```
char str1[100]; char str2[100];
printf("enter first string\n");
scanf("%s",str1);
printf("enter second string\n");
scanf("%s",str2);
strcat(str1,str2);
printf("str1 is %s and str2 is %s\n",str1,str2);
```

Different versions of my_strcat() implementation are as below for the logic: my_strcat appends the entire second string to the first

Version 1:

```
void my_strcat(char *a,char *b)
{
    int i = 0;
    while(a[i]!='\0')
    {
        i++;
    }
    // while can be replaced with strlen()
    int j;
    for(j = 0;b[j] != '\0';j++,i++)
        a[i] = b[j];
    a[i] = '\0';
}
```

Version 2:

```
void my_strcat(char *a,char *b)
{
    while(*a)
    {
        a++;
    }; // increment a till NULL.
    while(*b)
    {
        *a = *b;
        a++;
        b++;
    }
    // Once NULL, copy each character from b to a and increment a and b both till b becomes '\0'
    *a = '\0'; // assign '\0' character to a's end
}
```

Common Programming errors related to Strings in C

1. To store the string in a variable, it can never be `char* name[10]`. It is always **char name[10]**.

```
char * name[10]; //declaring an array of 10 pointer pointing each one to a char
```

```
char name[10]; //String declaration
```

2. Cannot assign a string by using **the operator =**.

```
name = "choco"; // Invalid
```

```
char name[10] = "choco" OR use strcpy() // valid
```

3. To print a string,

```
char a[]="hello";
```

```
printf("%s",a);//valid
```

```
printf("%s",*a);//Invalid // *a is the first character of the string not an address
```

Provide only the address of the first element of the string

4. void my_strcat(char *a,char *b)

```
{
    int i = 0;
    while(a[i] != '\0')
    {
        i++;
    }
    int j;
    for(j = 0; b[j] != '\0'; j++, i++)
        a[i] = b[j];

    // here we have missed the
    statement a[i] = '\0'; OR a[i] =
    b[j]; at the end of the function
    definition
}

void my_strcpy(char *b, char *a)
{
    // copy a to b
    while(*a != '\0')
    {
        *b = *a;
        b++;
        a++;
    }
    Missing statement is *b = '\0' OR *b = *a
}
```

If a string lacks the terminating null character, the program may be tricked into reading or writing data outside the bounds of the array. This is referred to as a **null-termination errors** .

Problem Solving: Text Processing

Let us write a function to find the position of First occurrence of the input character in a given string. We will use this function to find the positions of all occurrences of the input character in the string.

It is a very good practice to separate the interface and implementation. We will stick on to this.

First of all, let us start with writing the **function definition** to find the position of the first occurrence of the character. If the character not available in the string, return -1, else return the position of the character starting the index from 0.

// 1.c

// find the pos of ch in src;

// return the pos if found

// -1 if not found

//Version 1:

```
int find_left(const char *src, char ch)
{
    int pos = 0;
    while(src[pos] != '\0' && src[pos] != ch)
        ++pos;
    return src[pos] ? pos : -1;
}
```

//Version2:

```
int find_left(const char * src, char ch)
{
    int p = -1;    int i;
    int check = 0;
    for(i = 0; i < strlen(src); ++i)
    {
        if(src[i] == ch && check == 0)
        {
            p = i; check = 1;
        }
    }
}
```

```

    }
}
return p;
}

```

// Version 3:

```

int find_first(const char *src,char ch)

{
    int i = 0;

    while(*src && *src != ch )

    {
        src++;

        i++;

    }

    return (*src) ? i : -1;

}

```

// Think about changing implementation of find_first function to find the last occurrence of a character in a given string

// 1.h

```
int find_left(const char*,char);
```

Client code is as below.

```
//1_client.c:
```

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

```
#include "1.h"
```

```
int main()
```

```
{
```

```
    char str[100];
```

```
printf("enter the string\n");
scanf("%[^\n]s",str);
fflush(stdin);
char ch;
printf("enter the character\n");
ch = getchar();
int pos = find_left(str, ch);
if(pos != -1)
{
    printf("%c found at pos %d\n", ch, pos);
}
else
{
    printf("%c not found in %s\n", ch, x);
}
return 0;
}
```

If the function specified in 1.c have to be used for finding all the occurrences of the character, we should not be touching that function definition or function implementation. As per the requirement I am changing my client code as below. Highlighted code is the modified part in the client.

```
#include <stdio.h>
#include "1.h"
int main()
{
    char str[100];

    printf("enter the string\n");

    scanf("%[^\n]s",str);

    fflush(stdin);
```

```

char ch;

printf("enter the character\n");

ch = getchar();

// find all occurrences

int pos = -1;

int i;

while( (i = find_left(str + pos + 1, ch)) != -1)
{
    pos = pos + i + 1;

    printf("%d ", pos);
}

printf("\n");

return 0;
}

```

If I have to find the count of given character in a given string, what is the small modification required in the above client code? Maintain the counter starting from 0. Once the character is found, increment the counter. Print the counter outside the loop.

```

int count = 0;

while( (i = find_left(str + pos + 1, ch)) != -1)
{
    pos = pos + i + 1;

    count++;
}

printf("The character %c is present in string %s, %d times\n",ch, str,count);

```

Now, let us consider the next problem to be solved.

Given a string, Create a text with only alphabets.

Here is the solution.

```
#include<stdio.h>

void text_alphabets(char*, char*);

int main()
{
    char str[100];

    char str_new [100];

    printf("enter the string\n");

    scanf("%[^\n]s",str);

    text_alphabets(str,str_new);

    printf("text with only alphabets is %s",str_new);

    return 0;
}

void text_alphabets(char *a, char* b)
{
    while(*a)
    {
        if((*a >= 'a' && *a <= 'z') || (*a >= 'A' && *a <= 'Z'))
        {
            *b = *a;

            b++;
        }
    }
}
```



```
        a++;  
  
    }  
  
    *b = *a;  
  
}
```

Small modifications are required in the function definition, to display only the digits from a given string

```
void only_digits(char *a, char* b)
```

```
{  
  
    while(*a)  
    {  
        if(*a >= '0' && *a <= '9')  
        {  
            *b = *a;  
            b++;  
        }  
        a++;  
    }  
    *b = *a;  
}
```

Problem Solving: String Matching

The problem of finding occurrence(s) of a pattern string within another string or body of text is known as String Matching. Also known as exact string matching, string searching, text searching.

There are **different algorithms** for efficient string matching.

- Brute Force Algorithm/ Naïve String-search Algorithm
- Horspool Algorithm
- Boyer Moore Algorithm
- Rabin – Karp Algorithm
- Knuth – Morris- Pratt Algorithm

Let us define the problem to be solved.

Given a text `txt[0..n-1]` and a pattern `pat[0..m-1]`, write a function `pattern_match(char txt[], int n, char pattern[], int m)` that returns the position of the character in the text string when the first occurrence of pattern is matched with the text string.

Where,

n is the length of the text string and m is the length of the pattern string. i is the loop variable to traverse through the text string. j is the loop variable to traverse through the pattern. Whenever characters do not match in text and pattern, increment only i . Then again i th character from text must be compared with j th character of Pattern. Whenever characters in both strings match, increment i and j both

Requirements to solve this problem are listed here.

- Read a Text string and Pattern string
- Usage of loop variables to traverse across two strings
- Action to be taken when characters do not match in text and pattern
- Action to be taken whenever characters in both strings match

Demonstration of the C Solution:

Let us separate the interface and implementation.

```
// client.c
#include<stdio.h>
#include<string.h>
#include"1.h"
int main()
{
    char text[100];
    char pat[100];
    printf("enter the string\n");
    scanf("%[^\n]s",text);
    fflush(stdin);
    printf("enter the pattern\n");
    scanf("%[^\n]s",pat);
    int n = strlen(text);
    int m = strlen(pat);
    int pos = pattern_match(text,n,pat,m);
    if(pos == -1)
        printf("pattern not found\n");
    else
        printf("pattern is found at %d\n",pos);
    return 0;
}
```

```
// 1.c
```

//version 1: Having multiple return statement in the same function is not a good programmer's habit.

```
int pattern_match(char text[],int n,char pat[],int m)
{
    int i; int j;
    for(i = 0;i<=n-m;i++)
    {
        for(j = 0; j<m && text[i+j] ==pat[j];j++);
        if(j==m)
            return i;
    }
    return -1;
}
```

Version 2: modified the usage of return twice.

```
int pattern_match(char text[],int n,char pat[],int m)
{
    int i; int j;
    int res = -1; // added in this version
    for(i = 0;res == -1 && i<=n-m;i++)
    {
        for(j = 0; j<m && text[i+j] ==pat[j];j++);
        if(j==m)
            res = i; // modified in this version of code
    }
    return res; //modified.
}
```

If res is initialized to -1 and res == -1 in the outer for loop, what happens? Are these two statements required? If yes, Why? Think!

Also Think about the solution for Pattern matching using Recursion.

INTRODUCTION TO DEBUGGING WITH GDB

gdb stands for “**GNU Debugger**”. GDB allows you to inspect the C programs while they are executing and also allows you to see what exactly happens when your program crashes. It is supported by various languages such as Ada, Assembly, C, C++, D, Fortran, Go, Objective-C, OpenCL, Modula-2, Pascal, Rust. GDB can run on UNIX and Microsoft Windows variants and Mac OS X. Richard Stallman was the original author of **gdb**, and of many other **gnu** programs. Many others have also contributed to its development. The Latest version of GDB is 10.2 which was released on Apr 25th, 2021.

Tasks Supported by GDB

gdb can do four main kinds of things (plus other things in support of these) to help you catch bugs

- ✓ Start your program, specifying anything that might affect its behavior.
- ✓ Make your program stop on specified conditions.
- ✓ Examine what has happened, when your program has stopped.
- ✓ Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another.

Getting In and Out of GDB

1) **Starting the GDB:** type ‘**gdb**’ and press enter key



```
Microsoft Windows [Version 10.0.19041.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\DELL>gdb
GNU gdb (GDB) 7.6.1
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "mingw32".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb)
```

Gdb open prompt lets you know that it is ready for commands.

2) Exiting the GDB: type 'quit' or q and press enter key (press Ctrl-d)

```
C:\Users\DELL>gdb
GNU gdb (GDB) 7.6.1
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "mingw32".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb) q

C:\Users\DELL>
```

Invoking gdb

- ✓ The gdb can be run with a variety of arguments and options, to specify the debugging environment
- ✓ The most usual way to start gdb is with one argument, specifying an executable program:

gdb program

Ex: gdb a.exe

- ✓ It can also be started by specifying both an executable program and core file :

gdb program core

Ex: gdb a.exe core

- ✓ To debug a running process, specify a process ID as a second argument or option -p

gdb program 1234

Ex:gdb a.exe 1234

gdb -p 1234

This would attach gdb to process 1234. With option -p you can omit the program filename.

- ✓ gdb can be run without printing the front material by specifying --silent (or -q/--quiet):

gdb --silent OR gdb -q OR gdb --quiet

```
C:\Users\DELL>gdb --silent  
(gdb)
```

What gdb Does During Startup ?

1. Performs minimal setup required to initialize basic internal state.
2. Reads commands from the early initialization file in your home directory. Only a restricted set of commands can be placed into an early initialization file.
3. Executes commands and command files specified by the ‘-eiex’ and ‘-eix’ command line options in their specified order. Only a restricted set of commands can be used with ‘-eiex’ and ‘eix’.
4. Sets up the command interpreter as specified by the command line .
5. Reads the system wide initialization file and the files from the system wide initialization directory.
6. Reads the initialization file (if any) in your home directory and executes all the commands in that file.
7. Executes commands and command files specified by the ‘-iex’ and ‘-ix’ options in their specified order. The options ‘-ex’ and ‘-x’ can also be used for the same purpose .This way the settings can be applied before gdb init files get executed and before inferior gets loaded.
8. Processes command line options and operands.
9. Reads and executes the commands from the initialization file (if any) in the current working directory as long as ‘set auto-load local-gdbinit’ is set to ‘on’ .This is only done if the current directory is different from your home directory. Thus, we can have more than one init file, one generic in your home directory, and another, specific to the program you are debugging, in the directory where you invoke gdb.
10. Executes commands and command files specified by the ‘-ex’ and ‘-x’ options in their specified order.
11. Reads the command history recorded in the history file.

Usage of GDB on C code

Problem to be solved:

Program to add two numbers that shows undefined behavior

gdb1.c

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

```
int main()
```

```
{
```

```
    int c;
```

```
    int a;
```

```
    int b;
```

```
    c=a+b;
```

```
    printf("%d\n",c);
```

```
    return 0;
```

```
}
```

Function to calculate the sum of all the factors of a given number and client must display the result.

1_find_factors_sum.c

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

```
int sum_factors(int n);
```

```
int main()
```

```
{
```

```
    int number;int sum;
```



```
scanf("%d",&number);

sum=sum_factors(number);

printf("Sum of factors of a number %d is %d\n",number,sum);

return 0;

}

int sum_factors(int n)
{
    int sum=0;
    for(int i=1;i<=n;++i)
    {
        if(n%i==0)
        {
            sum = sum+i;
        }
    }
    return sum;
}
```

Compile the code with -g option:

-g: Produces debugging Information in the Operating system's native format.

Invoke gdb

`gdb a.exe`

```
C:\Users\DELL\Desktop>gcc -g -c gdb1.c
C:\Users\DELL\Desktop>gcc gdb1.o -o 1.exe
C:\Users\DELL\Desktop>gcc -g -c 1_find_factors_sum.c
C:\Users\DELL\Desktop>gcc 1_find_factors_sum.o -o 2.exe

C:\Users\DELL\Desktop>gdb
GNU gdb (GDB) 7.6.1
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "mingw32".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb) run
Starting program:
No executable specified, use `target exec'.
(gdb) target exec 1.exe
(gdb) run
Starting program: C:\Users\DELL\Desktop\1.exe
[New Thread 7068.0x3388]
[New Thread 7068.0x46b4]
8388864
[Inferior 1 (process 7068) exited normally]
(gdb) file 2.exe
Reading symbols from C:\Users\DELL\Desktop\2.exe...done.
(gdb) run
```

GDB Commands

help or h: Provides the list of classes of commands.

help class: Using one of the general help classes as an argument, a list of the individual commands in that class can be obtained

Example: help breakpoint

help command: With a command name as help argument, gdb displays a short paragraph on how to use that command.

Example: help run

Using **help all** details of all commands in all the classes can be obtained

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Command name abbreviations are allowed if unambiguous.
(gdb) help breakpoints
Making program stop at certain points.

List of commands:
```

run OR r: Starts the program under gdb. Program begins to execute immediately.

- ✓ Specify the executable with an argument to gdb
- ✓ Use target exec command OR use file command

list OR l: Used to print lines from a source file. By default, ten lines are printed

- ✓ **list line_num** : Print lines centered around that specified line number
- ✓ **list func**: Print lines centered around the beginning of function func.
- ✓ **list**: Print more lines.
- ✓ **list -**: Print lines just before the lines last printed.
- ✓ **list first, last**: print lines from first to last
- ✓ **list ,last**: Print lines ending with last
- ✓ **listsize** : Display the number of lines that list prints.
- ✓ **set listsize unlimited**: Make the list command display count source lines (unless the list argument explicitly specifies some other number). Setting count to unlimited or 0 means there's no limit.

```
(gdb) file 2.exe
Reading symbols from C:\Users\DELL\Desktop\2.exe...done.
(gdb) list
1      #include<stdio.h>
2      int sum_factors(int n);
3      int main()
4      {
5          int number;int sum;
6          scanf("%d",&number);
7          sum=sum_factors(number);
8          printf("Sum of factors of a number %d is %d\n",number,sum);
9          return 0;
10     }
(gdb) list -
(gdb) list 1,5
1      #include<stdio.h>
2      int sum_factors(int n);
3      int main()
4      {
5          int number;int sum;
```

```
(gdb) file 2.exe
Reading symbols from C:\Users\DELL\Desktop\2.exe...done.
(gdb) show listsize
Number of source lines gdb will list by default is 10.
(gdb) list
1      #include<stdio.h>
2      int sum_factors(int n);
3      int main()
4      {
5          int number;int sum;
6          scanf("%d",&number);
7          sum=sum_factors(number);
8          printf("Sum of factors of a number %d is %d\n",number,sum);
9          return 0;
10     }
(gdb) set listsize 15
(gdb) show listsize
Number of source lines gdb will list by default is 15.
(gdb) list
11     int sum_factors(int n)
12     {
13         int sum=0;
14         for(int i=1;i<=n;++i)
15         {
16             if(n%i==0)
17             {
18                 //printf("%d\t",i);
19                 sum = sum+i;
20             }
21         }
22         return sum;
23     }
24 }
```

print OR p: It evaluates and prints the value of an expression of the language your program is written in.

- ✓ **p expression:** prints the value of a given expression.
- ✓ **p variable:** prints the value of a given variable.
- ✓ **p function_name :: variable:** To specify a static variable in a particular function or file colon-colon (::) notation is used.

```
(gdb) p 1+2
$1 = 3
(gdb) p 1==1
$2 = 1
(gdb) p 1==2
$3 = 0
(gdb) p a
No symbol "a" in current context.
(gdb)
```

GDB Commands - Stopping and Continuing in GDB

breakpoint: Makes the program stop whenever a certain point in the program is reached

- ✓ **break line_num:** breaks at start of code for that line.
- ✓ **break function_name :** break at start of code for that function.
- ✓ **break address_of_instruction:** break at that exact address.
- ✓ **break if cond:** Set a breakpoint with condition cond. It evaluates the expression cond each time the breakpoint is reached, and stop only if the value is nonzero.
- ✓ **break with no arguments:** Sets the breakpoint at the next instruction to be executed in the selected stack frame.

```
(gdb) break 8
Breakpoint 3 at 0x401492: file 1_find_factors_sum.c, line 8.
(gdb) break 15
Breakpoint 4 at 0x4014cb: file 1_find_factors_sum.c, line 15.
(gdb) info b
Num      Type             Disp Enb Address            What
3        breakpoint        keep y  0x00401492 in main at 1_find_factors_sum.c:8
4        breakpoint        keep y  0x004014cb in sum_factors at 1_find_factors_sum.c:15
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
error return ../../gdb-7.6.1/gdb/windows-nat.c:1275 was 5
Starting program: C:\Users\DELL\Desktop\2.exe
[New Thread 2616.0x1830]
[New Thread 2616.0x36a0]
12
Breakpoint 4, sum_factors (n=12) at 1_find_factors_sum.c:16
16      if(n%i==0)
(gdb) continue
Continuing.
Breakpoint 4, sum_factors (n=12) at 1_find_factors_sum.c:16
16      if(n%i==0)
(gdb)
```

watchpoint: A special breakpoint that stops the program when the value of an expression changes, without having to predict a particular place where this may happen. When expression is modified, the program will interrupt and print out the old and new values. The expression may be as simple as the value of a single variable, or as complex as many variables combined by operators

watch expression

```
(gdb) n
14          for(int i=1;i<=n;++i)
(gdb) watch i
Hardware watchpoint 5: i
(gdb) n
Hardware watchpoint 5: i

Old value = 1
New value = 2
0x004014e2 in sum_factors (n=12) at 1_find_factors_sum.c:14
14          for(int i=1;i<=n;++i)
(gdb) n

Breakpoint 4, sum_factors (n=12) at 1_find_factors_sum.c:16
16          if(n%i==0)
(gdb) n
19              sum = sum+i;
(gdb) n
14          for(int i=1;i<=n;++i)
(gdb) n
Hardware watchpoint 5: i

Old value = 2
New value = 3
0x004014e2 in sum_factors (n=12) at 1_find_factors_sum.c:14
14          for(int i=1;i<=n;++i)
```

catchpoint: A special breakpoint that stops the program when a certain kind of event occurs such as exceptions.

delete: Can individually delete the above using their numbers.

```
(gdb) info b
Num    Type           Disp Enb Address      What
3      breakpoint      keep y   0x00401492 in main at 1_find_factors_sum.c:8
4      breakpoint      keep y   0x004014cb in sum_factors at 1_find_factors_sum.c:15
5      hw watchpoint   keep y   i
breakpoint already hit 2 times
(gdb) delete 3
(gdb) info b
Num    Type           Disp Enb Address      What
4      breakpoint      keep y   0x004014cb in sum_factors at 1_find_factors_sum.c:15
5      hw watchpoint   keep y   i
breakpoint already hit 2 times
(gdb) delete
Delete all breakpoints? (y or n) y
(gdb) info b
No breakpoints or watchpoints.
```

clear: Delete any breakpoints set at the specified location .

- ✓ **clear line_num:** all breakpoints in that line are cleared.
- ✓ **clear file_name:** breakpoints at beginning of function are cleared
- ✓ **clear:** Delete any breakpoints at the next instruction to be executed in the selected stack frame.

```
(gdb) b 8
Breakpoint 10 at 0x401492: file 1_find_factors_sum.c, line 8.
(gdb) b 15
Breakpoint 11 at 0x4014cb: file 1_find_factors_sum.c, line 15.
(gdb) info b
Num      Type      Disp Enb Address      What
10       breakpoint  keep y  0x00401492  in main at 1_find_factors_sum.c:8
11       breakpoint  keep y  0x004014cb  in sum_factors at 1_find_factors_sum.c:15
(gdb) cclear 10
No breakpoint at 10.
(gdb) cclear 8
Deleted breakpoint 10
(gdb) cclear
Deleted breakpoint 11
(gdb)
```

disable: Makes the breakpoint inoperative, but remembers the information on the breakpoint so that it can be enabled later using enable.

```
(gdb) b 8
Breakpoint 1 at 0x401492: file 1_find_factors_sum.c, line 8.
(gdb) b 15
Breakpoint 2 at 0x4014cb: file 1_find_factors_sum.c, line 15.
(gdb) info b
Num      Type      Disp Enb Address      What
1        breakpoint  keep y  0x00401492  in main at 1_find_factors_sum.c:8
2        breakpoint  keep y  0x004014cb  in sum_factors at 1_find_factors_sum.c:15
(gdb) disable b 1
(gdb) info b
Num      Type      Disp Enb Address      What
1        breakpoint  keep n  0x00401492  in main at 1_find_factors_sum.c:8
2        breakpoint  keep y  0x004014cb  in sum_factors at 1_find_factors_sum.c:15
(gdb)
```

To re-enable the recent disabled breakpoint. Type enable b.

```
(gdb) b 8
Breakpoint 1 at 0x401492: file 1_find_factors_sum.c, line 8.
(gdb) b 15
Breakpoint 2 at 0x4014cb: file 1_find_factors_sum.c, line 15.
(gdb) info b
Num      Type      Disp Enb Address      What
1        breakpoint  keep y  0x00401492  in main at 1_find_factors_sum.c:8
2        breakpoint  keep y  0x004014cb  in sum_factors at 1_find_factors_sum.c:15
(gdb) disable 1
(gdb) info b
Num      Type      Disp Enb Address      What
1        breakpoint  keep n  0x00401492  in main at 1_find_factors_sum.c:8
2        breakpoint  keep y  0x004014cb  in sum_factors at 1_find_factors_sum.c:15
(gdb) enable 1
(gdb) info b
Num      Type      Disp Enb Address      What
1        breakpoint  keep y  0x00401492  in main at 1_find_factors_sum.c:8
2        breakpoint  keep y  0x004014cb  in sum_factors at 1_find_factors_sum.c:15
(gdb)
```

Info: To print a list of all breakpoints, watchpoints, and catchpoints

GDB Commands

file: To change to a different file during a gdb session file command is used . It reads the symbols for getting the contents of pure memory, and it is the program executed when you use the `run' command.

```
C:\Users\DELL\Desktop>gdb
GNU gdb (GDB) 7.6.1
Copyright (C) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "mingw32".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb) run
Starting program:
No executable specified, use `target exec'.
(gdb) target exec 2.exe
(gdb) run
Starting program: C:\Users\DELL\Desktop\2.exe
[New Thread 16176.0x3d0c]
[New Thread 16176.0x226c]
12
Sum of factors of a number 12 is 28
[Inferior 1 (process 16176) exited normally]
(gdb) file 1.exe
Reading symbols from C:\Users\DELL\Desktop\1.exe...done.
(gdb) run
Starting program: C:\Users\DELL\Desktop\1.exe
[New Thread 8888.0x13c8]
[New Thread 8888.0x7f4]
7954560
[Inferior 1 (process 8888) exited normally]
(gdb)
```

next OR n: Continue to the next source line in the current stack frame. Function calls that appear within the line of code are executed without stopping

```
Breakpoint 4, sum_factors (n=12) at 1_find_factors_sum.c:16
16      if(n%i==0)
(gdb) next
19      sum = sum+i;
(gdb) next
14      for(int i=1;i<=n;++i)
(gdb) next
Breakpoint 4, sum_factors (n=12) at 1_find_factors_sum.c:16
16      if(n%i==0)
(gdb) next
19      sum = sum+i;
(gdb) p i
$4 = 2
(gdb) p n
$5 = 12
(gdb) _
```

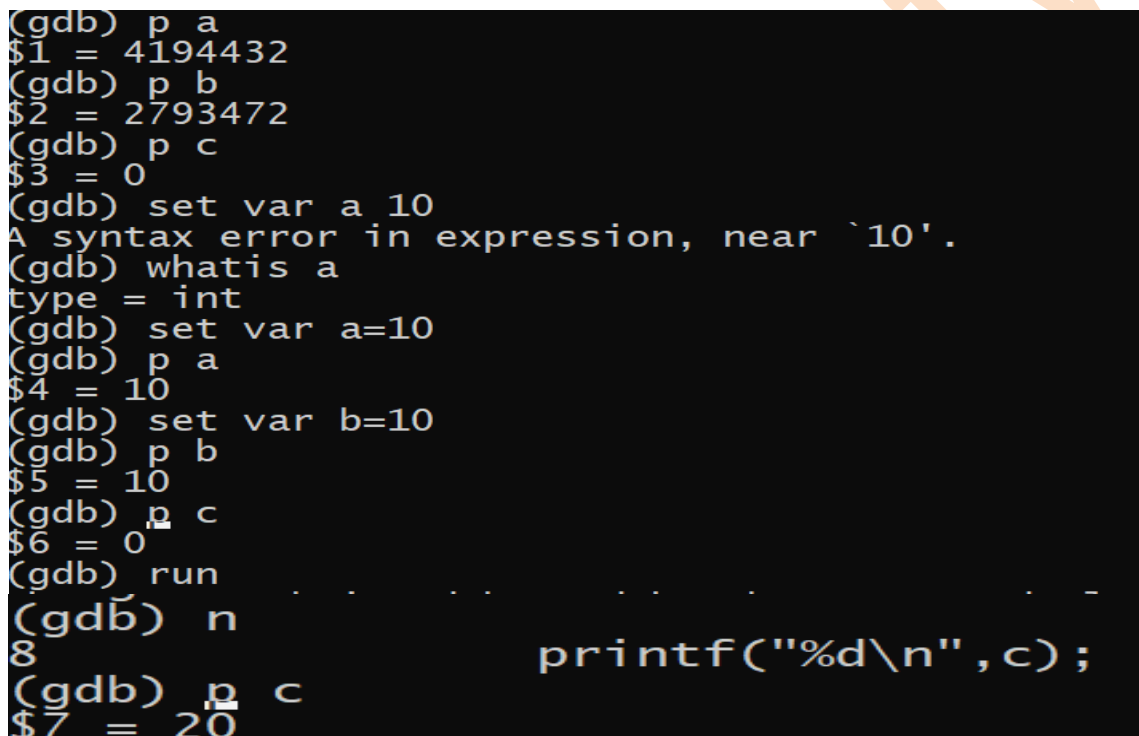
step OR s: Same as next but steps inside any functions called within the line.

continue OR c: Resume program execution at the address where your program last stopped. Any breakpoints set at that address are bypassed.

whatis: Prints the data type of arg, which can be either an expression or a name of a data type. With no argument, print the data type of \$, the last value in the value history.

set var: Modify the run-time behavior of readline by altering the values of variables.

set var a =10



```
(gdb) p a
$1 = 4194432
(gdb) p b
$2 = 2793472
(gdb) p c
$3 = 0
(gdb) set var a 10
A syntax error in expression, near `10'.
(gdb) whatis a
type = int
(gdb) set var a=10
(gdb) p a
$4 = 10
(gdb) set var b=10
(gdb) p b
$5 = 10
(gdb) p c
$6 = 0
(gdb) run
(gdb) n
8 printf("%d\n",c);
(gdb) p c
$7 = 20
```

backtrace - produces a stack trace of the function calls that lead to a seg fault (should remind you of Java exceptions) .

where - same as backtrace.It works even when you're still in the middle of the program .

finish - runs until the current function is finished .