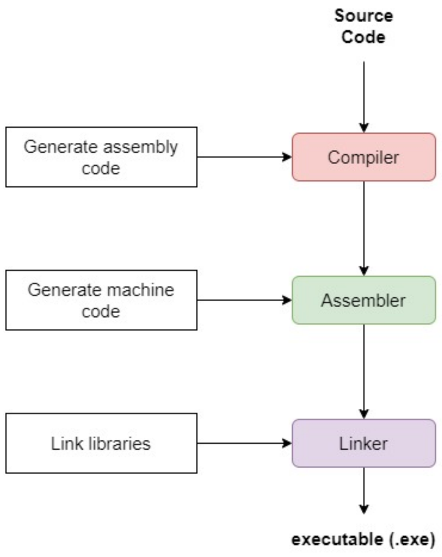
Sardar Patel Institute of Technology
Bhavan's Campus, Munshi Nagar, Andheri (W), Mumbai : 400058, India

(Autonomous College of Affiliated to University of Mumbai)

Static/Dynamic link library

Name: Pranay Singhvi
UID:2021300126
Batch: Comps B(B4)
Experiment No: 2

Aim:	Write a program for creating a static/dynamic link library for number operations.
Theory:	**Introduction** In this tutorial, we'll study the static and dynamic linking processes to generate the final executable from the assembled code for any program. **Generating Executables** Before we start with linking methods, let's first understand the process of generating the executable of any piece of code. **Formally, a computer program is a sequence of statements in a programming language instructing the CPU what to do.** To execute any program, we convert the source code to machine code. We do this by first compiling the code to an intermediate level and then converting it to assembly-level code. After that, we link the assembly code with other libraries or modules it uses. So, **linking is the process of combining external programs with our program to execute them successfully.** After linking, we finally get our executable:  Source Code Generate assembly code Compiler Generate machine code Assembler Link libraries Linker executable (.exe) **Linking** The assembler translates our compiled code to machine code and stores the result in an object file. It's a binary representation of our program. **The assembler gives a memory location to each object and instruction. Some of**

these memory locations are virtual, i.e., they're offsets relative to the base address of the first instruction. That's the case with references to external libraries. The linker resolves them when combining the object file with the libraries.

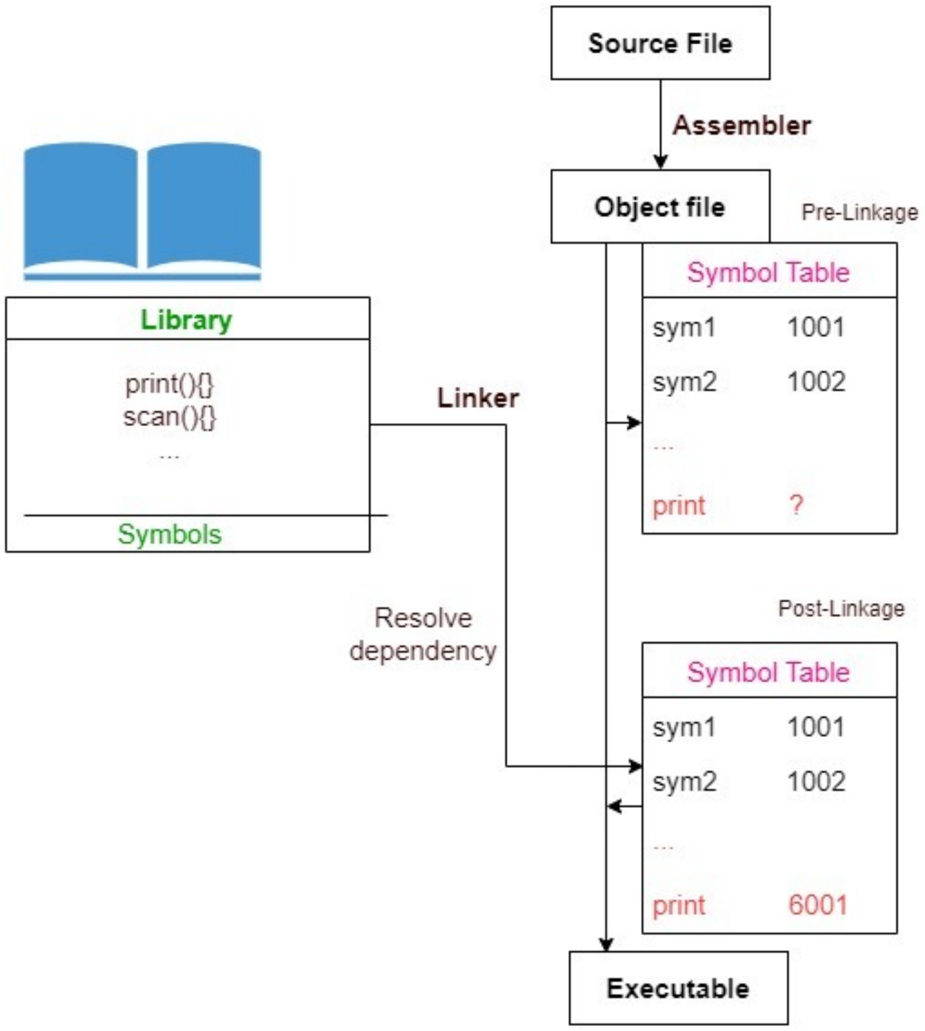
Overall, we have two mechanisms for linking:

1. Static
2. Dynamic

Static Linking

In static linking, the system linker copies the dependencies into the final executable. At the time of linking an external library, the linker finds all dependencies that are defined in that library. And it replaces them with the corresponding functions from the library to resolve dependencies in our code. Afterward, the linker generates the final executable file that we can execute on the underlying machine.

For example, let's say our application calls the function *print()* from an external library named *Library.* The assembler generates the object file with all native symbols resolved to their memory addresses. The external reference *print()* cannot be resolved. The linker loads this library and finds the definition of *print()* in it. Then, it maps to *print()* to a memory location and thus resolves the dependency:



So, a statistically linked file contains our program's code as well as the code of all the libraries it invokes. Since we copy complete libraries, we need space on both the disk and in the main memory because the resulting file may be very large.

Benefits of Static Linking

Firstly, static linking ensures the application can run as a standalone binary because we integrate external libraries with it at the compile time.

Secondly, it ensures exclusivity. What does that mean? Each statically linked process gets its copy of the code and data. So, in cases where security is very important (e.g., financial transactions), we use static linking. This is so because it completely isolates one process from another by providing each an independent environment.

Thirdly, for statically linked applications, we bundle everything into our application. Hence, we don't have to

ensure that the client has the right version of the libraries available on their system
Further on, static linking offers faster execution because we copy the entire library content at compile time. Hence, we don't have to run the query for unresolved symbols at runtime. Thus, we can execute a statically linked program faster than a dynamically linked one.

When to Use Static Linking?

We use static linking where we require secure and mutually exclusive processes that don't share any code. We also use static linking for embedded projects where we want a controlled and speedy execution environment with no run-time linkage issues. Most small form factor devices such as video controllers in a mobile handset use static linking in their boot process.

Dynamic Linking

In dynamic linking, we copy the names of the external libraries into our final executable as unresolved symbols. We do the actual linking of these unresolved symbols only at runtime. How?
When encountering an unresolved symbol, we query RAM for it. If the corresponding library isn't loaded, the operating system loads it in the memory. So, **the operating system performs dynamic linking for us by resolving each external symbol on the first muss.** As a result, we load only a single copy of a library in memory and all processes use it.

Benefits of Dynamic Linking

In dynamic linking, we maintain only one copy of a shared library in the memory. Therefore, our program's executable file is smaller as compared to that of a statically linked one. Also, it's more memory efficient because all processes can share the library in RAM instead of using a separate copy. Similarly, all the processes sharing the library benefit from the cache too.
Dynamic linking results in a lower average load time. On average, many programs use a small number of external libraries. So, we load each library only once instead of

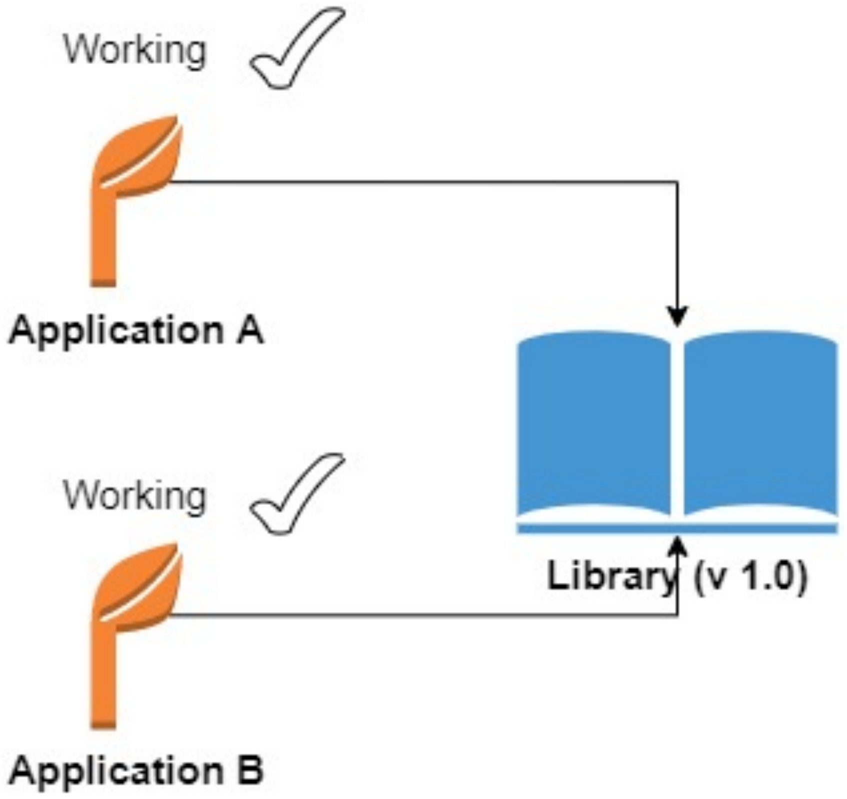
multiple times. Thus, all the calls after the first one will take less load time. That's because only the missing unresolved symbols will be loaded in memory, but after the first loading, they'll all be there for subsequent calls.
From the deployment and maintenance perspective, dynamic linking offers us easier updating and deployment. We can update and recompile the external libraries to offer the latest changes to our programs. After recompilation, we reload new versions.
Dynamic linking promotes modularity. We use it to develop large programs that require multiple language versions having multiple modules in them. For example, the *Zoho* accounting software has many modules to implement various accounting features such as income tax, corporate tax, sales tax, etc. Each of these modules is dynamically loaded at runtime as per the user's request.

When to Use Dynamic Linking?

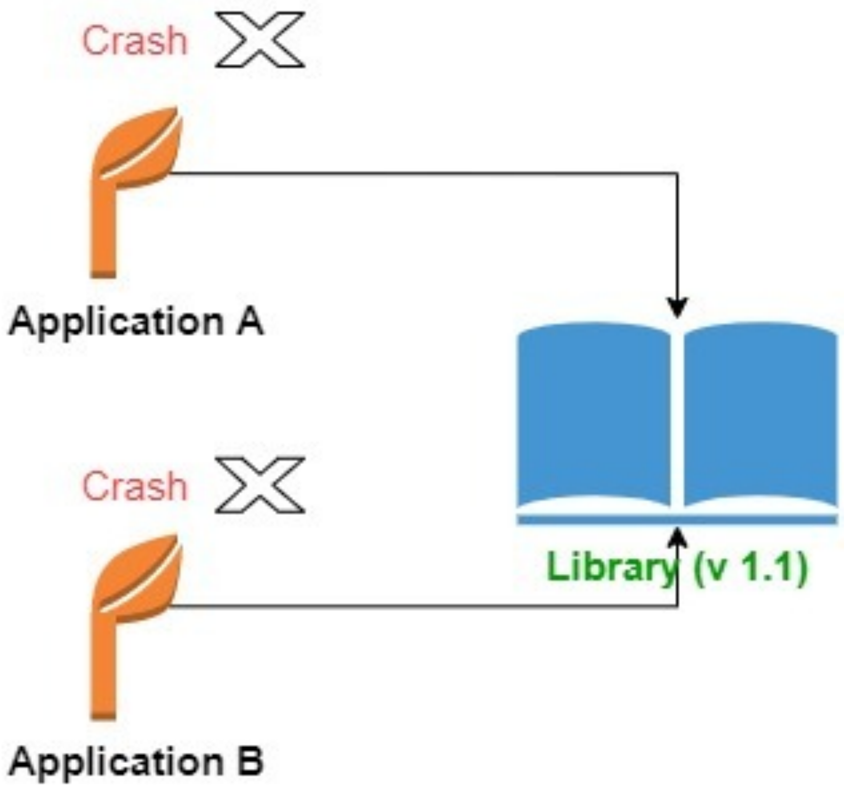
We use dynamic linking when we have many applications using a common set of libraries.
For example, let's suppose we have a set of microservices with each using the queue service (e.g., RabbitMQ). Then, we can optimize resources using dynamic linking.

DLL Hell Problem

Here, we describe the key weakness of the dynamic linking approach. That's the DLL Hell problem, where DLL stands for a dynamically linked library.
The DLL Hell problem occurs when a DLL that the operating system loads is different from the version our application expects. As a result, we get unresolved symbols. For example, that can happen if some functions have different signatures in the DLL's newer version.
Let's say applications *A* and *B* use version 1.o Of a DLL library:



Now, let’s suppose we update this DLL to version 1.1. If we change any APIs the applications use but forget to update the codes of A and B to use the new APIs, both applications will crash at runtime since they call the old APIs:

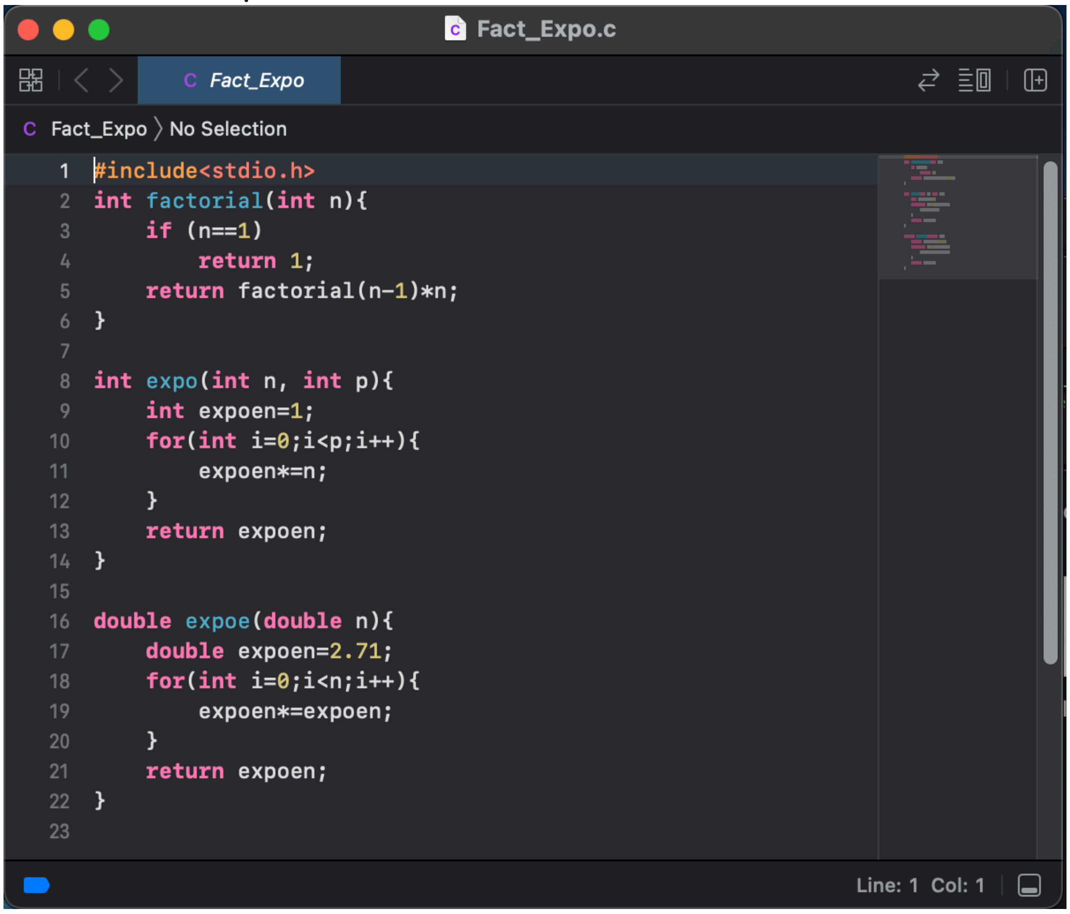


With DLLs, we have no built-in mechanism to check for backward compatibility. Thus, even minor changes to a DLL can cause problems.

On the other side, we find that the statically linked libraries don't suffer from this problem. This is so because the library version that our application uses internally is included with it in the same bundle. Hence, there will never be any version mismatch.

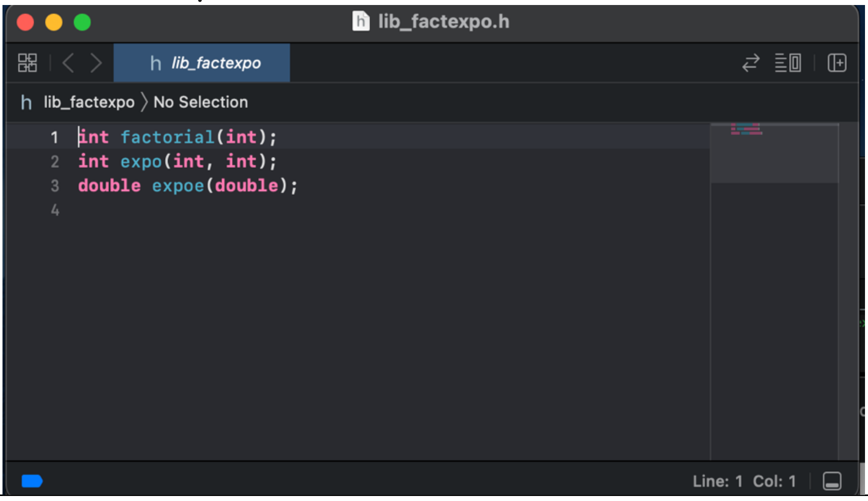
Procedure (Along with code & output):

1. **Static Linking:**
   a. Create a C file for performing Factorial and Exponential Function



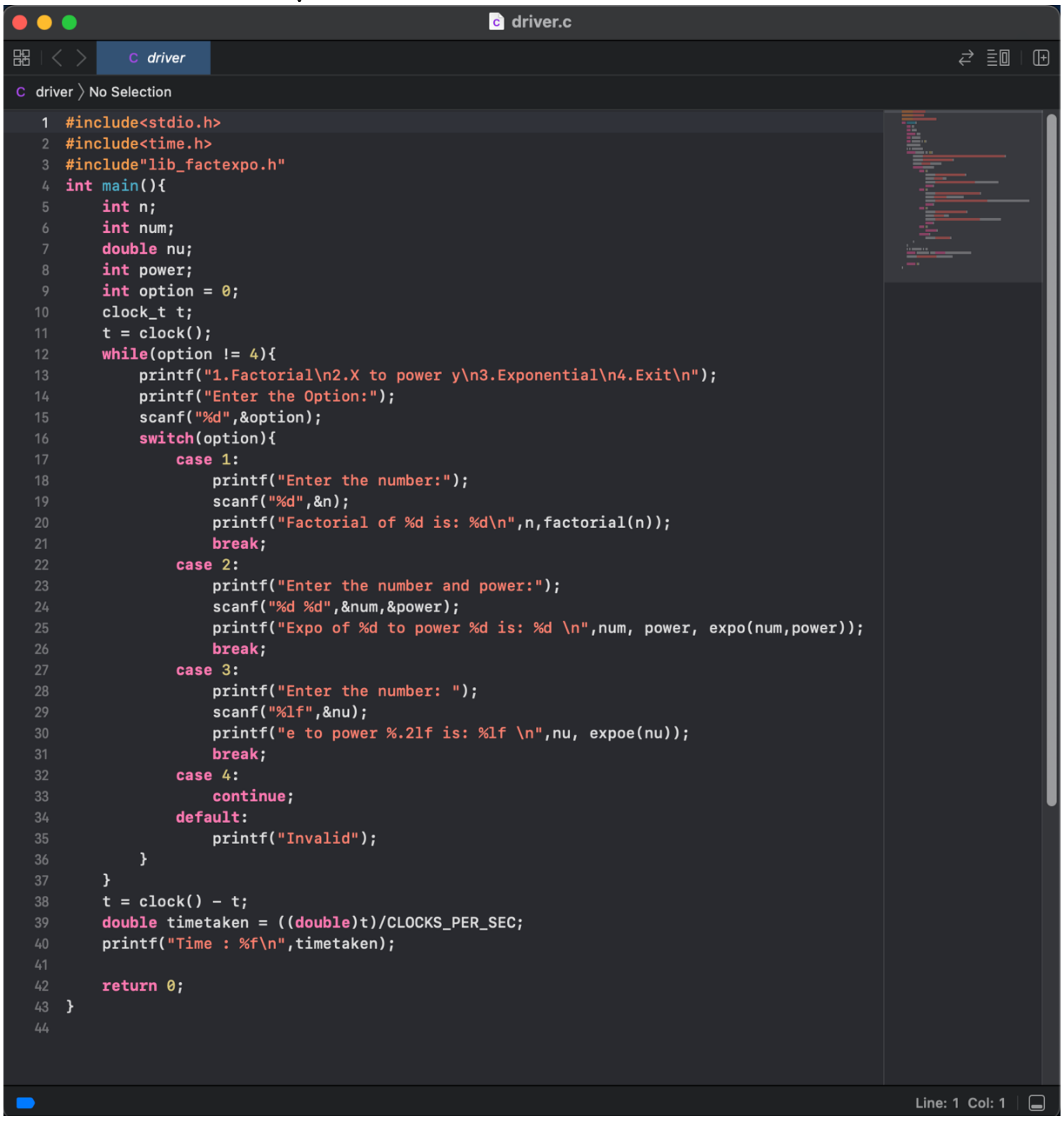
```c
#include<stdio.h>
int factorial(int n){
    if (n==1)
        return 1;
    return factorial(n-1)*n;
}

int expo(int n, int p){
    int expoen=1;
    for(int i=0;i<p;i++){
        expoen*=n;
    }
    return expoen;
}

double expoe(double n){
    double expoen=2.71;
    for(int i=0;i<n;i++){
        expoen*=expoen;
    }
    return expoen;
}
```

   b. Create a header file (lib_factexpo.h) for the library.



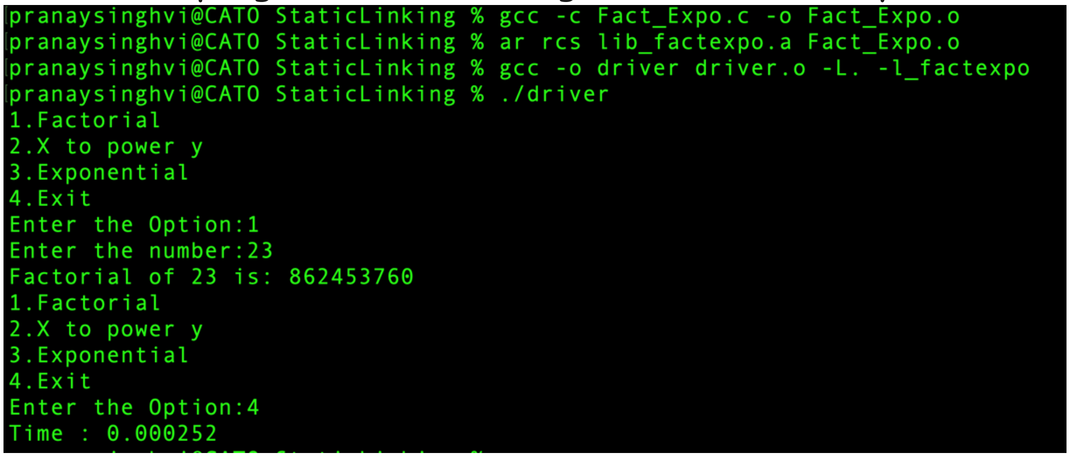
```c
int factorial(int);
int expo(int, int);
double expoe(double);
```

c. Create a driver program that uses the created library.



```c
#include<stdio.h>
#include<time.h>
#include"lib_factexpo.h"
int main(){
    int n;
    int num;
    double nu;
    int power;
    int option = 0;
    clock_t t;
    t = clock();
    while(option != 4){
        printf("1.Factorial\n2.X to power y\n3.Exponential\n4.Exit\n");
        printf("Enter the Option:");
        scanf("%d",&option);
        switch(option){
            case 1:
                printf("Enter the number:");
                scanf("%d",&n);
                printf("Factorial of %d is: %d\n",n,factorial(n));
                break;
            case 2:
                printf("Enter the number and power:");
                scanf("%d %d",&num,&power);
                printf("Expo of %d to power %d is: %d \n",num, power, expo(num,power));
                break;
            case 3:
                printf("Enter the number: ");
                scanf("%lf",&nu);
                printf("e to power %.2lf is: %lf \n",nu, expoe(nu));
                break;
            case 4:
                continue;
            default:
                printf("Invalid");
        }
    }
    t = clock() - t;
    double timetaken = ((double)t)/CLOCKS_PER_SEC;
    printf("Time : %f\n",timetaken);

    return 0;
}
```

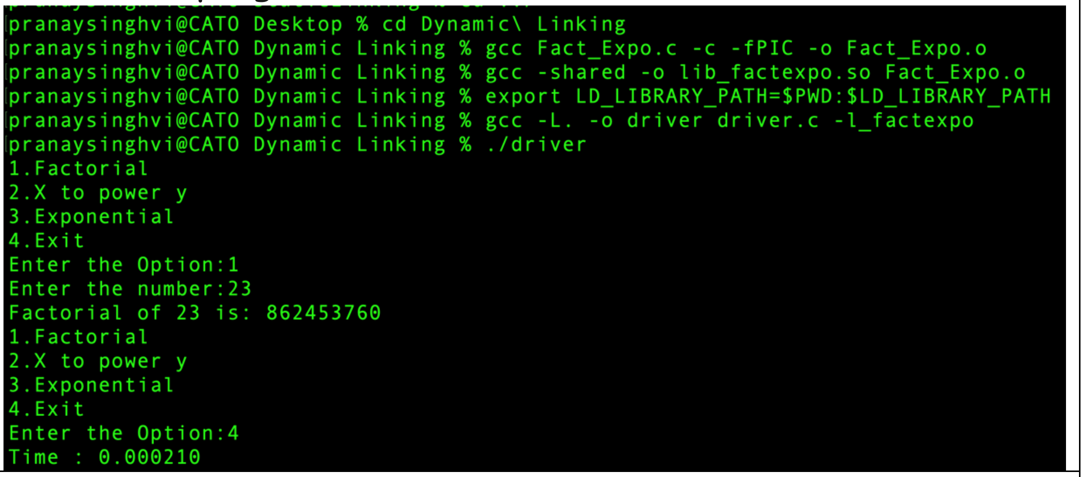
d. Creating the static library. Compiling the driver program and including the static library in it.



```
pranaysinghvi@CATO StaticLinking % gcc -c Fact_Expo.c -o Fact_Expo.o
pranaysinghvi@CATO StaticLinking % ar rcs lib_factexpo.a Fact_Expo.o
pranaysinghvi@CATO StaticLinking % gcc -o driver driver.o -L. -l_factexpo
pranaysinghvi@CATO StaticLinking % ./driver
1.Factorial
2.X to power y
3.Exponential
4.Exit
Enter the Option:1
Enter the number:23
Factorial of 23 is: 862453760
1.Factorial
2.X to power y
3.Exponential
4.Exit
Enter the Option:4
Time : 0.000252
```

2. Dynamic Linking

a. After creating the same C, header, and driver files as above, compile the C files. Creating a dynamic library and compiling the driver program using the dynamic library. Executing the driver program.



```
pranaysinghvi@CATO Desktop % cd Dynamic\ Linking
pranaysinghvi@CATO Dynamic Linking % gcc Fact_Expo.c -c -fPIC -o Fact_Expo.o
pranaysinghvi@CATO Dynamic Linking % gcc -shared -o lib_factexpo.so Fact_Expo.o
pranaysinghvi@CATO Dynamic Linking % export LD_LIBRARY_PATH=$PWD:$LD_LIBRARY_PATH
pranaysinghvi@CATO Dynamic Linking % gcc -L. -o driver driver.c -l_factexpo
pranaysinghvi@CATO Dynamic Linking % ./driver
1.Factorial
2.X to power y
3.Exponential
4.Exit
Enter the Option:1
Enter the number:23
Factorial of 23 is: 862453760
1.Factorial
2.X to power y
3.Exponential
4.Exit
Enter the Option:4
Time : 0.000210
```

Conclusion:

I learned about static linking and dynamic linking using C language. Using different commands like ar rcs for creation of static linking.