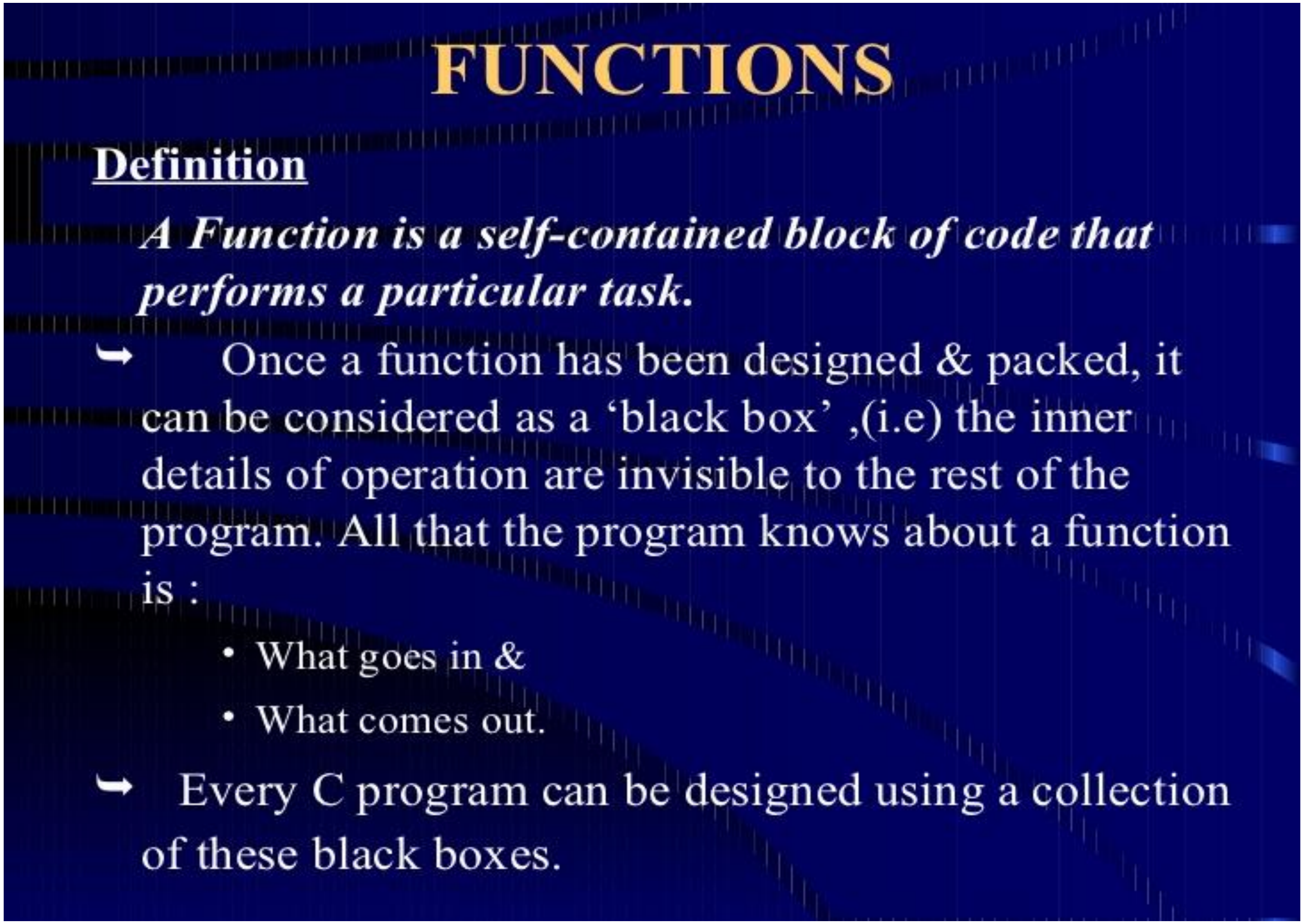


FUNCTIONS

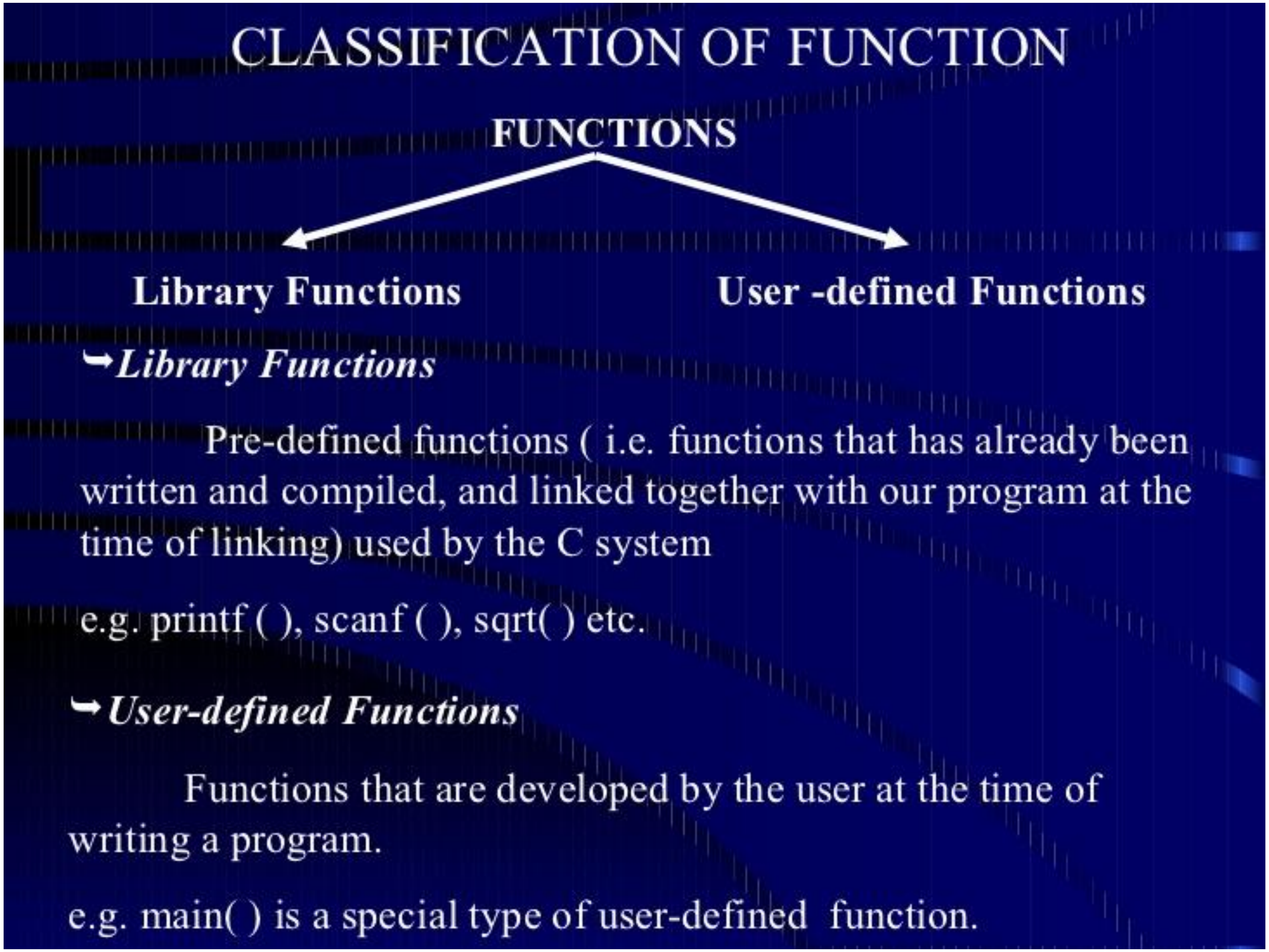
Definition

A Function is a self-contained block of code that performs a particular task.

➥ Once a function has been designed & packed, it can be considered as a 'black box' ,(i.e) the inner details of operation are invisible to the rest of the program. All that the program knows about a function is :

- What goes in &
- What comes out.

➥ Every C program can be designed using a collection of these black boxes.



CLASSIFICATION OF FUNCTION

$$\begin{tikzcd}
 & \textbf{FUNCTIONS} \arrow[dl] \arrow[dr] & \\
\textbf{Library Functions} & & \textbf{User -defined Functions}
\end{tikzcd}$$

➥***Library Functions***

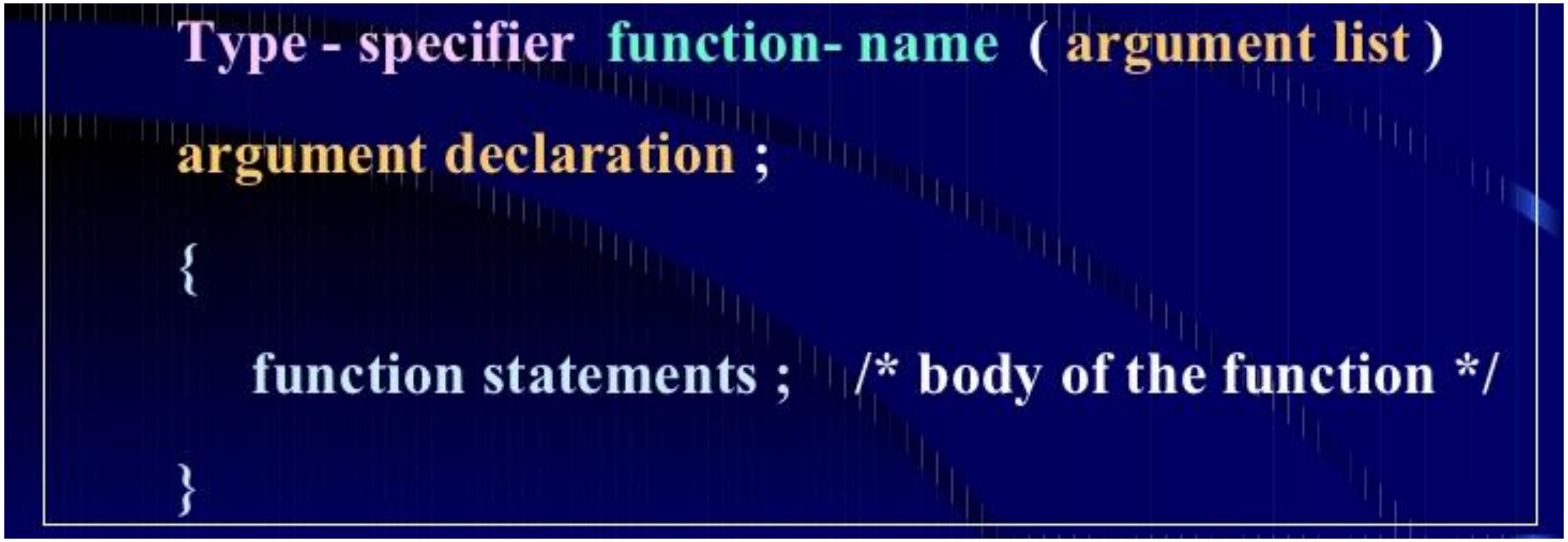
Pre-defined functions (i.e. functions that has already been written and compiled, and linked together with our program at the time of linking) used by the C system

e.g. printf ( ), scanf ( ), sqrt( ) etc.

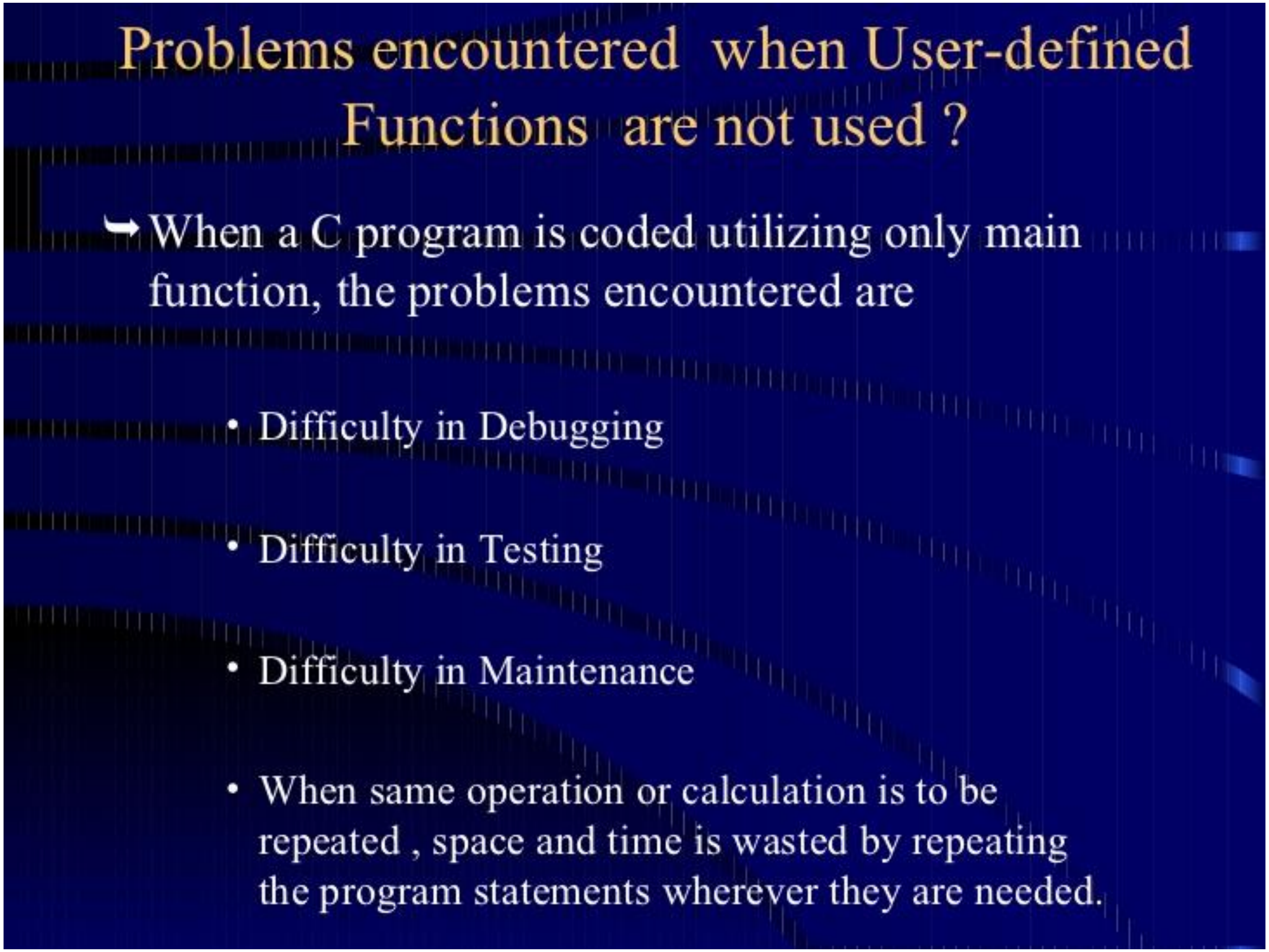
➥***User-defined Functions***

Functions that are developed by the user at the time of writing a program.

e.g. main( ) is a special type of user-defined function.



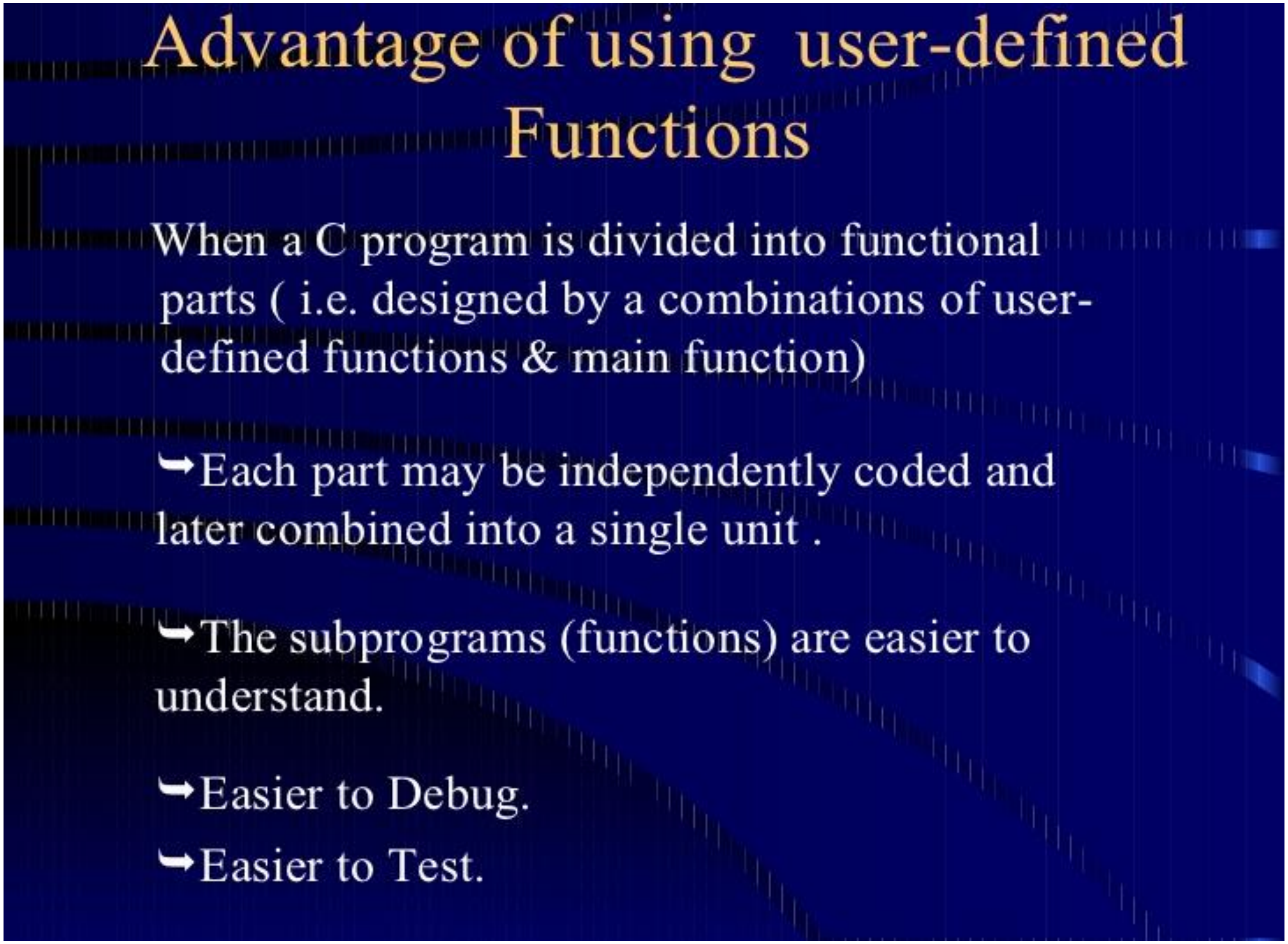
```
Type - specifier  function- name  ( argument list )
argument declaration ;
{
    function statements ;    /* body of the function */
}
```



Problems encountered when User-defined Functions are not used ?

➥ When a C program is coded utilizing only main function, the problems encountered are

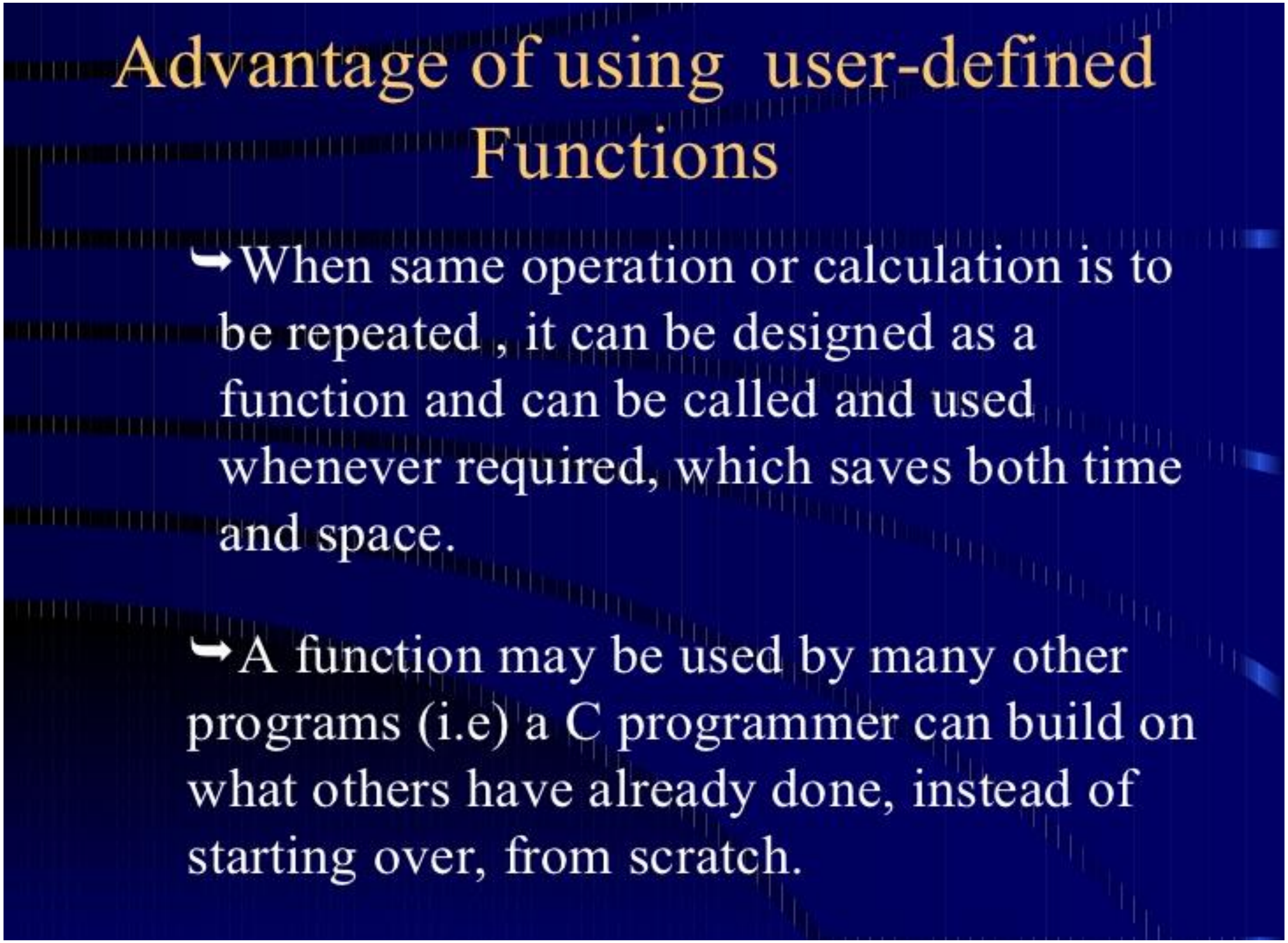
- Difficulty in Debugging
- Difficulty in Testing
- Difficulty in Maintenance
- When same operation or calculation is to be repeated , space and time is wasted by repeating the program statements wherever they are needed.



Advantage of using user-defined Functions

When a C program is divided into functional parts (i.e. designed by a combinations of user-defined functions & main function)

- Each part may be independently coded and later combined into a single unit .
- The subprograms (functions) are easier to understand.
- Easier to Debug.
- Easier to Test.



Advantage of using user-defined Functions

- When same operation or calculation is to be repeated , it can be designed as a function and can be called and used whenever required, which saves both time and space.

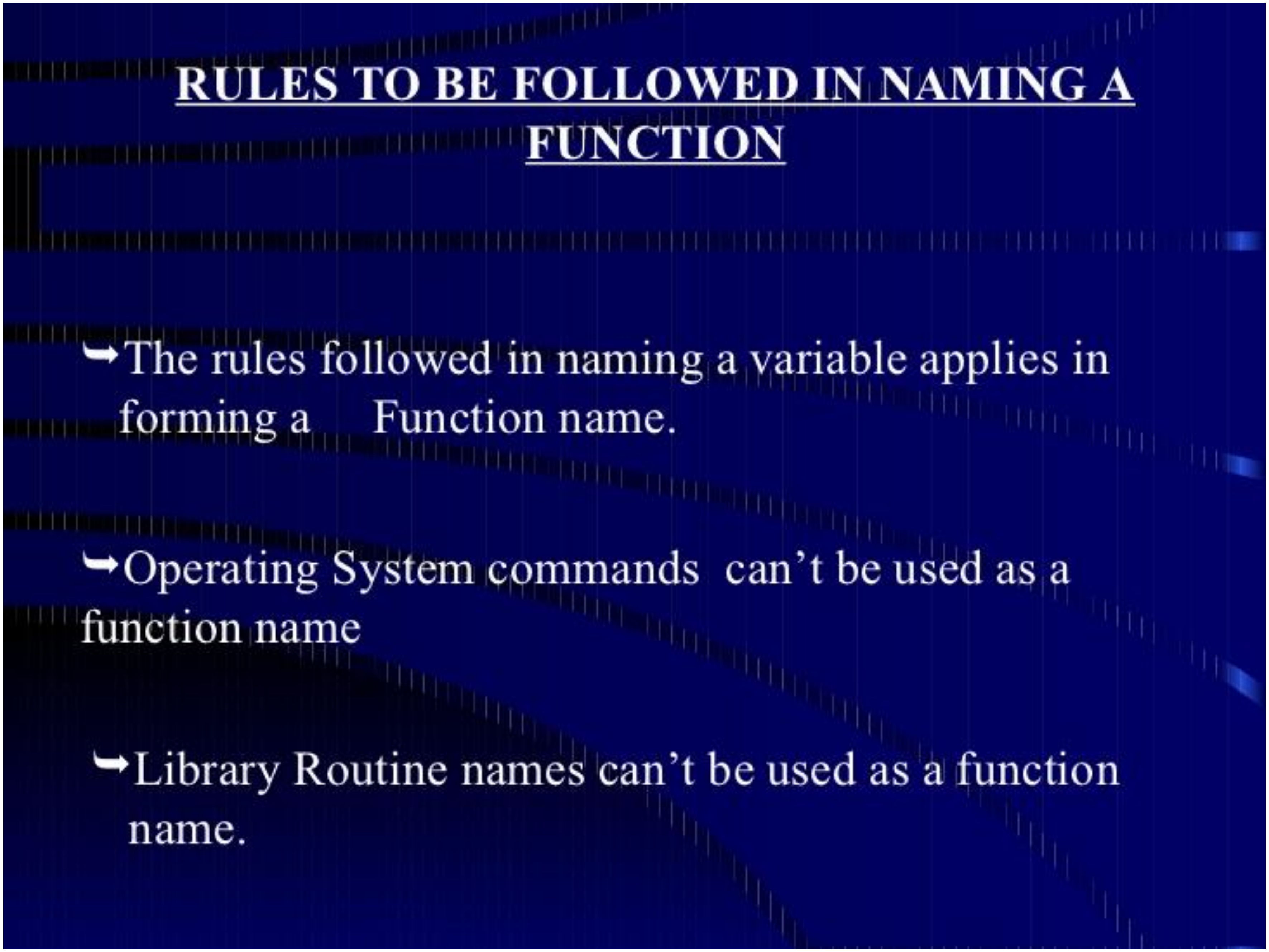
- A function may be used by many other programs (i.e) a C programmer can build on what others have already done, instead of starting over, from scratch.

```
void read_arr(int a[10][10],int row,int col)
{
    int i,j;
    for(i=1;i<=row;i++)
    {
    for(j=1;j<=col;j++)
    {
        printf(\"Enter Element %d %d : \",i,j);
        scanf(\"%d\",&a[i][j]);
            }
    }
}
```

```
void print_arr(int m[10][10],int row,int col)
{
    int i,j;
    for(i=1;i<=row;i++)
        {
        for(j=1;j<=col;j++)
        {
            printf(\"%d \",m[i][j]);
         }
        printf(\"\\n\");
        }
}
```

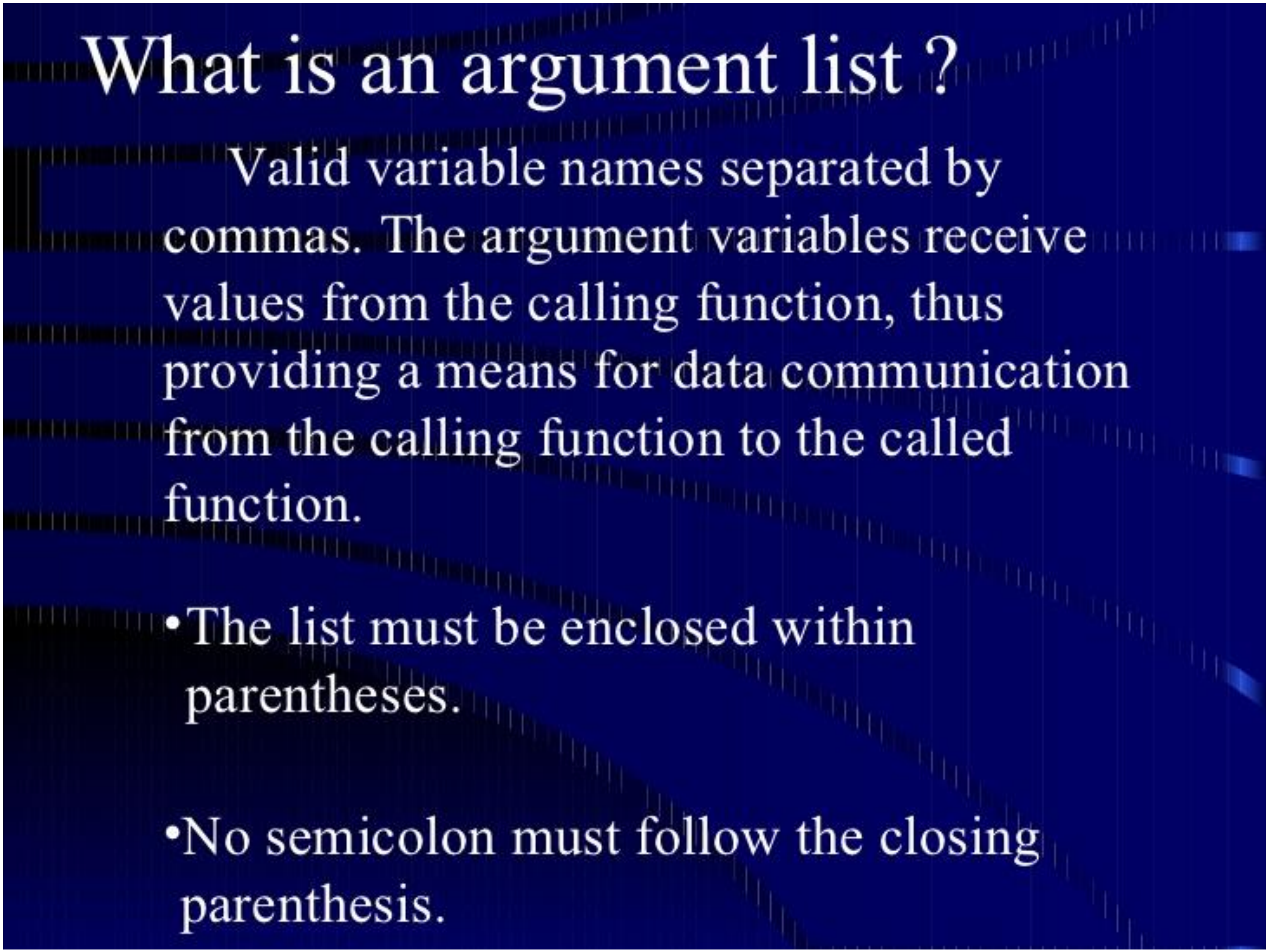
```
void add_arr(int m1[10][10],int m2[10][10],int m3[10][10],int row,int col)
{
   int i,j;
   for(i=1;i<=row;i++)
   {
   for(j=1;j<=col;j++)
   {
   m3[i][j] =  (m1[i][j] + m2[i][j]);
   }
   }
}
```

```
main()
{
   int m1[10][10],m2[10][10],m3[10][10],row,col
   clrscr();
   printf(\"Enter number of rows :\");
   scanf(\"%d\",&row);
   printf(\"Enter number of colomns :\");
   scanf(\"%d\",&col);
   read_arr(m1,row,col);
   read_arr(m2,row,col);
   add_arr(m1,m2,m3,row,col);
   print_arr(m3,row,col);
   }
```



RULES TO BE FOLLOWED IN NAMING A FUNCTION

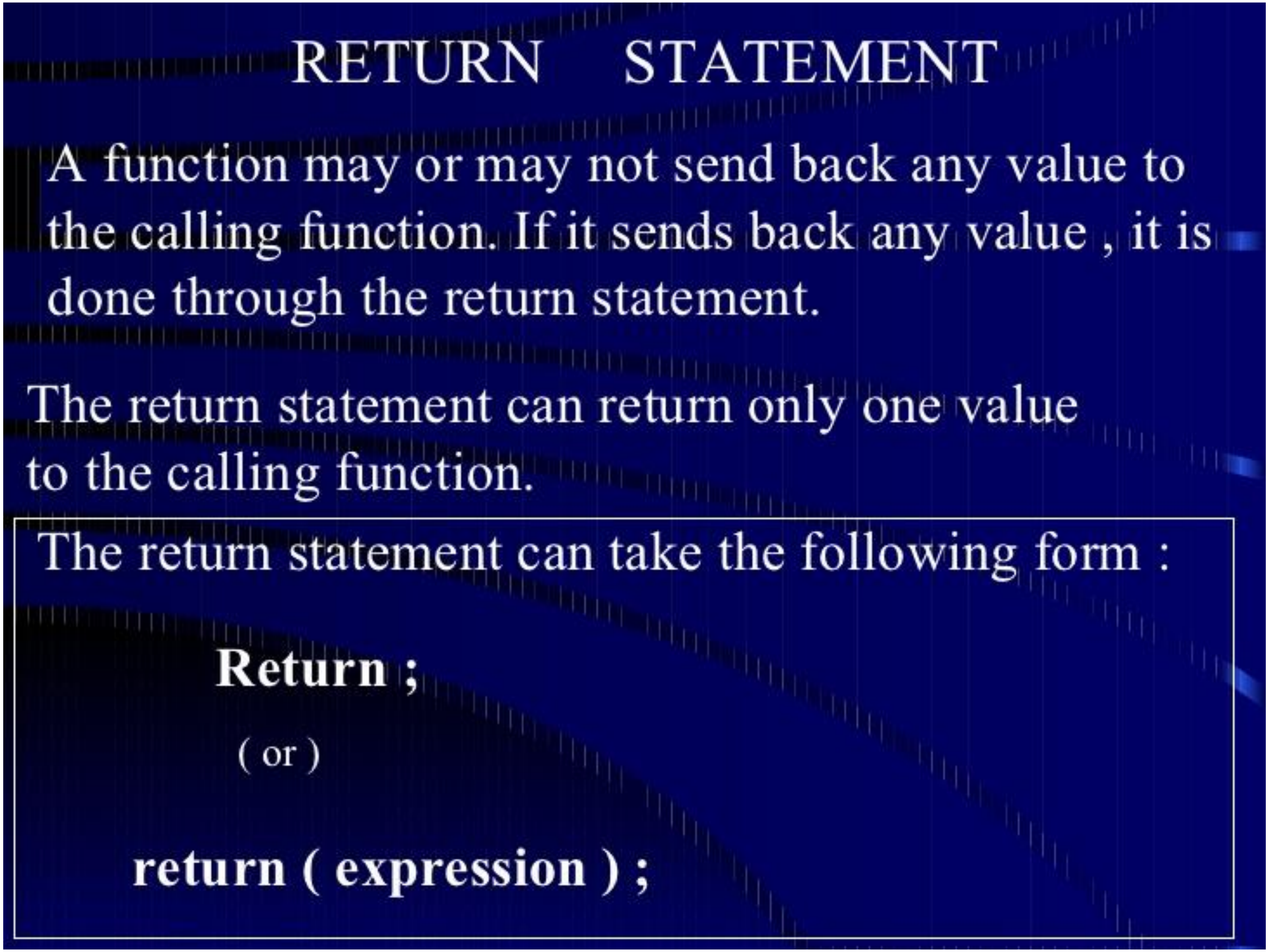
- The rules followed in naming a variable applies in forming a Function name.
- Operating System commands can't be used as a function name
- Library Routine names can't be used as a function name.



What is an argument list ?

Valid variable names separated by commas. The argument variables receive values from the calling function, thus providing a means for data communication from the calling function to the called function.

- The list must be enclosed within parentheses.
- No semicolon must follow the closing parenthesis.



RETURN STATEMENT

A function may or may not send back any value to the calling function. If it sends back any value , it is done through the return statement.

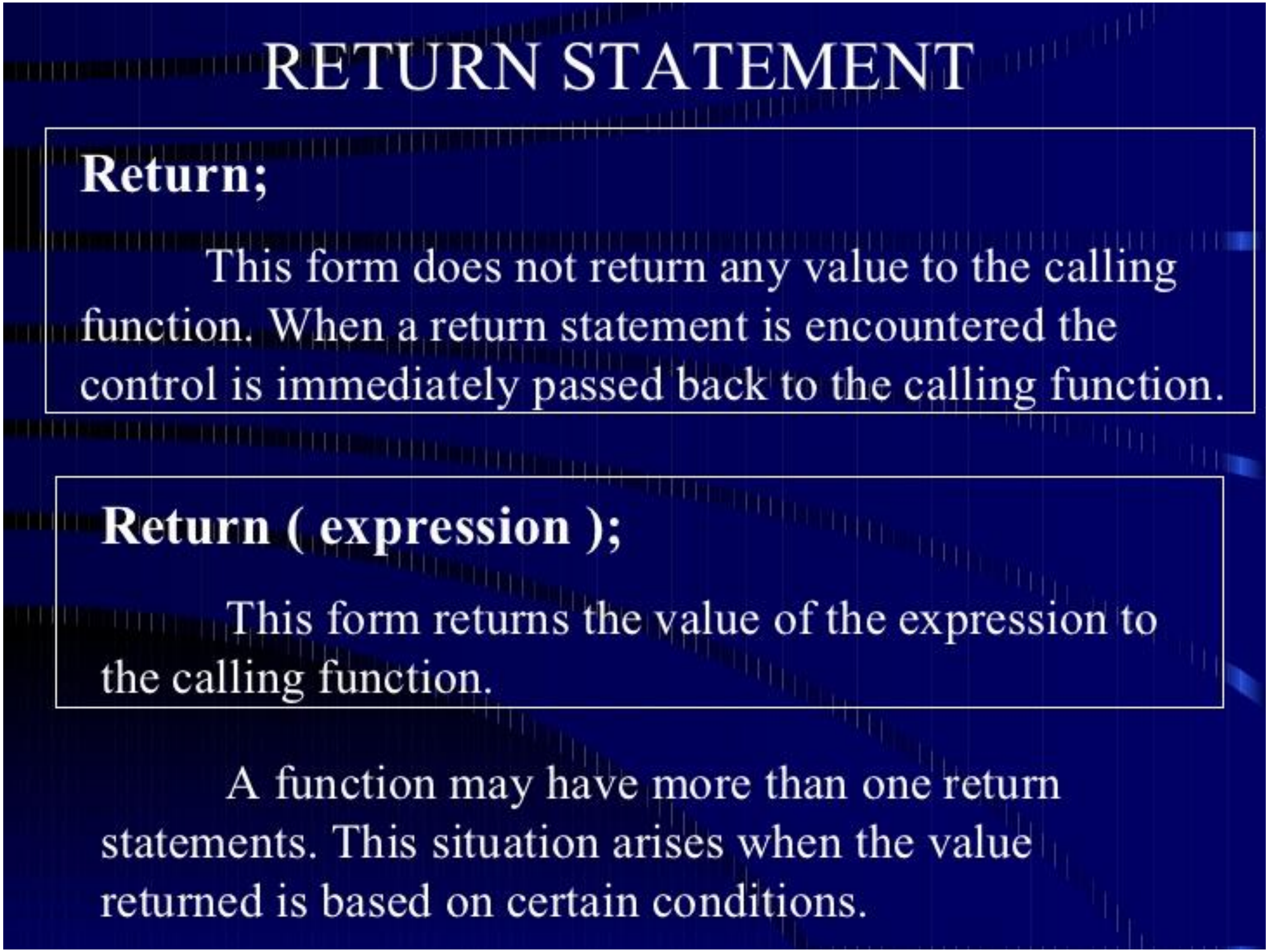
The return statement can return only one value to the calling function.

The return statement can take the following form :

Return ;

(or)

return (expression) ;



RETURN STATEMENT

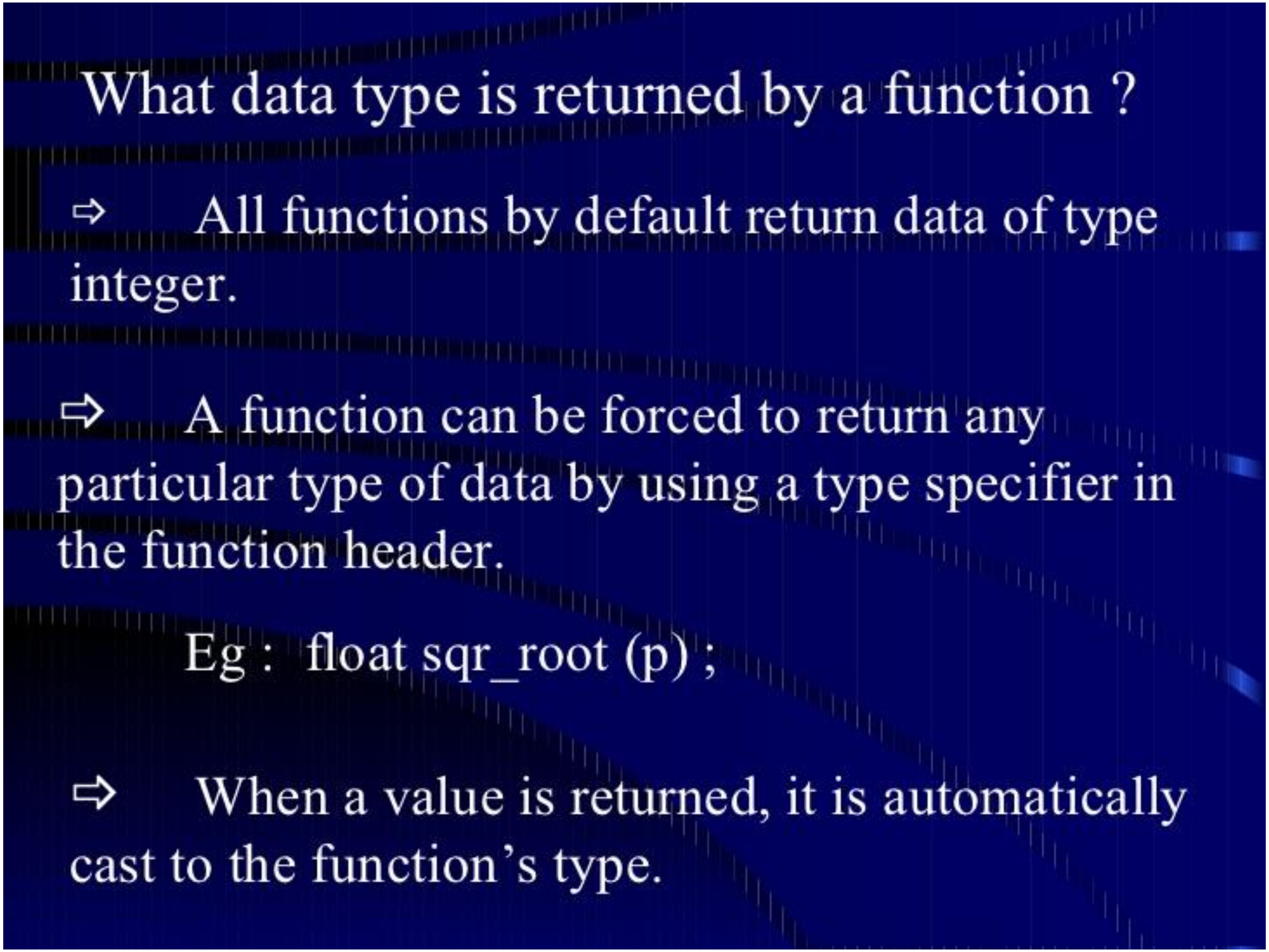
Return;

This form does not return any value to the calling function. When a return statement is encountered the control is immediately passed back to the calling function.

Return (expression);

This form returns the value of the expression to the calling function.

A function may have more than one return statements. This situation arises when the value returned is based on certain conditions.



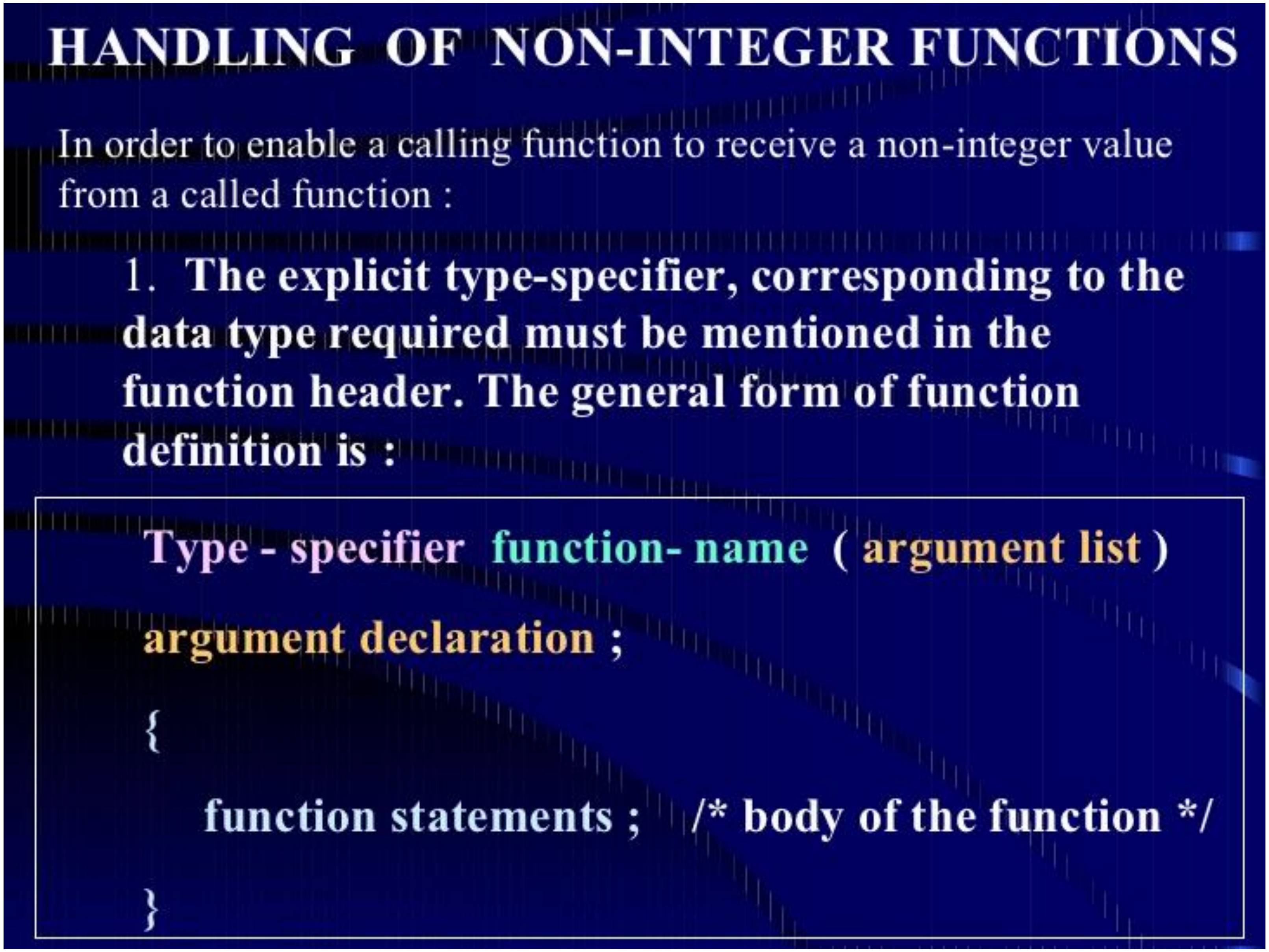
What data type is returned by a function ?

⇨ All functions by default return data of type integer.

⇨ A function can be forced to return any particular type of data by using a type specifier in the function header.

Eg : float sqr_root (p) ;

⇨ When a value is returned, it is automatically cast to the function’s type.

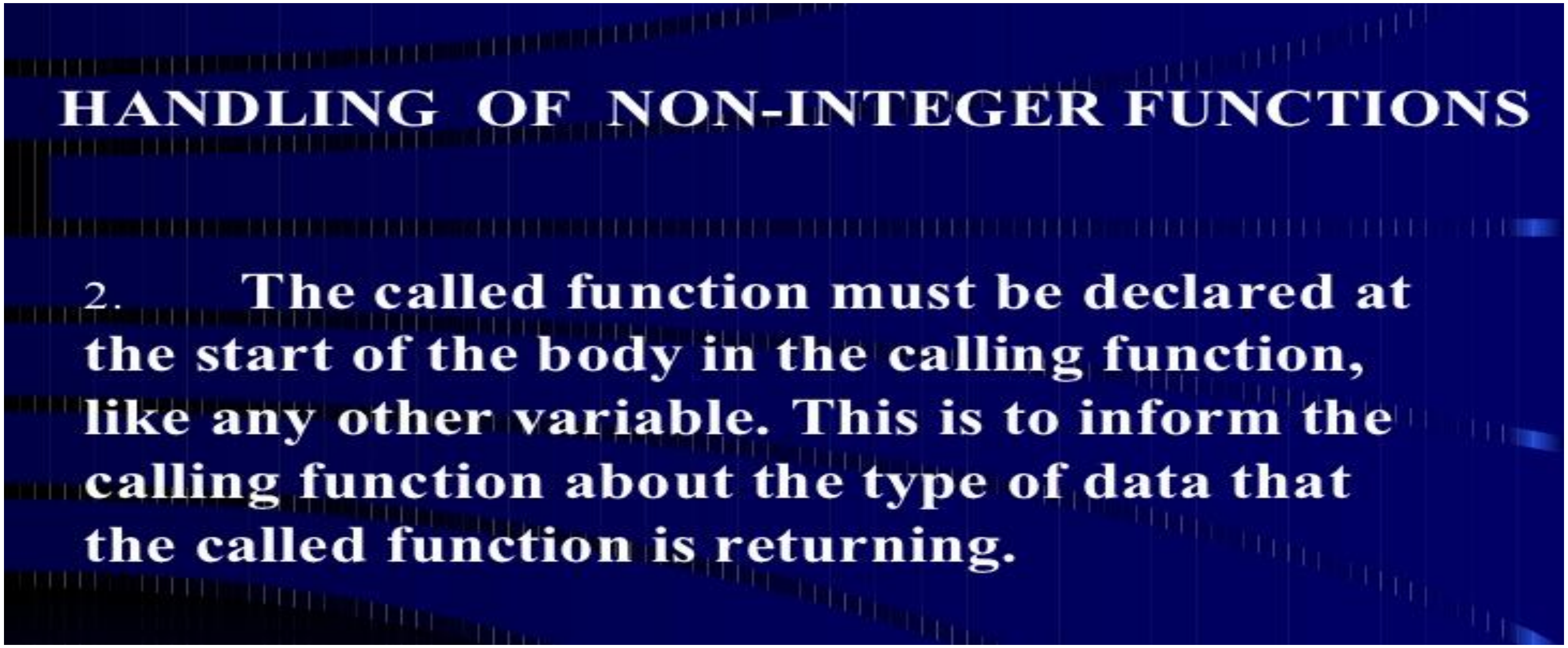


HANDLING OF NON-INTEGER FUNCTIONS

In order to enable a calling function to receive a non-integer value from a called function :

1. **The explicit type-specifier, corresponding to the data type required must be mentioned in the function header. The general form of function definition is :**

```
Type - specifier  function- name  ( argument list )
argument declaration ;
{
    function statements ;    /* body of the function */
}
```

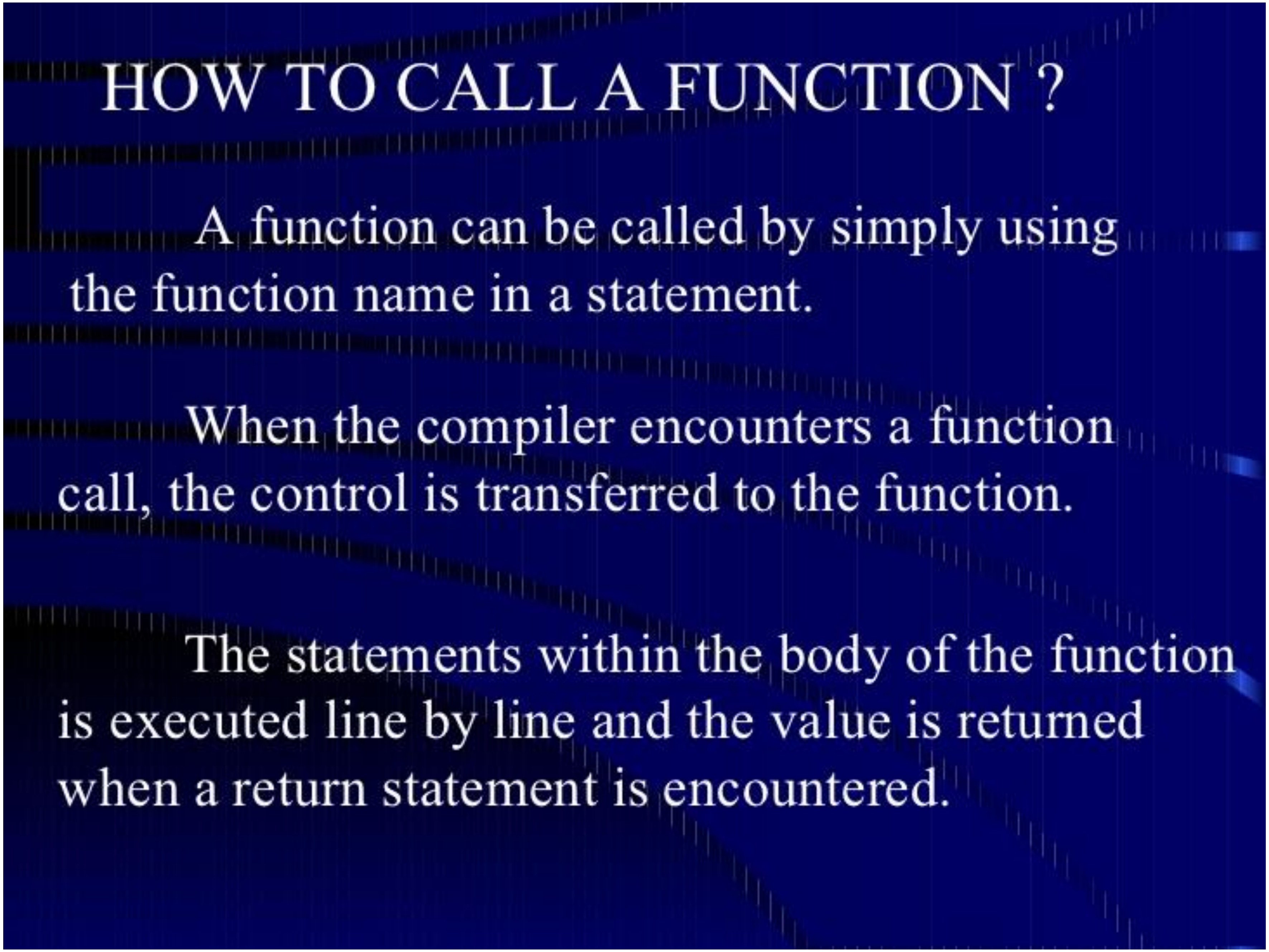

HANDLING OF NON-INTEGER FUNCTIONS

2. The called function must be declared at the start of the body in the calling function, like any other variable. This is to inform the calling function about the type of data that the called function is returning.

```
void add_arr(int m1[10][10],int m2[10][10],int m3[10][10],int row,int col);

void read_arr(int a[10][10],int row,int col);

void print_arr(int m[10][10],int row,int col)
```

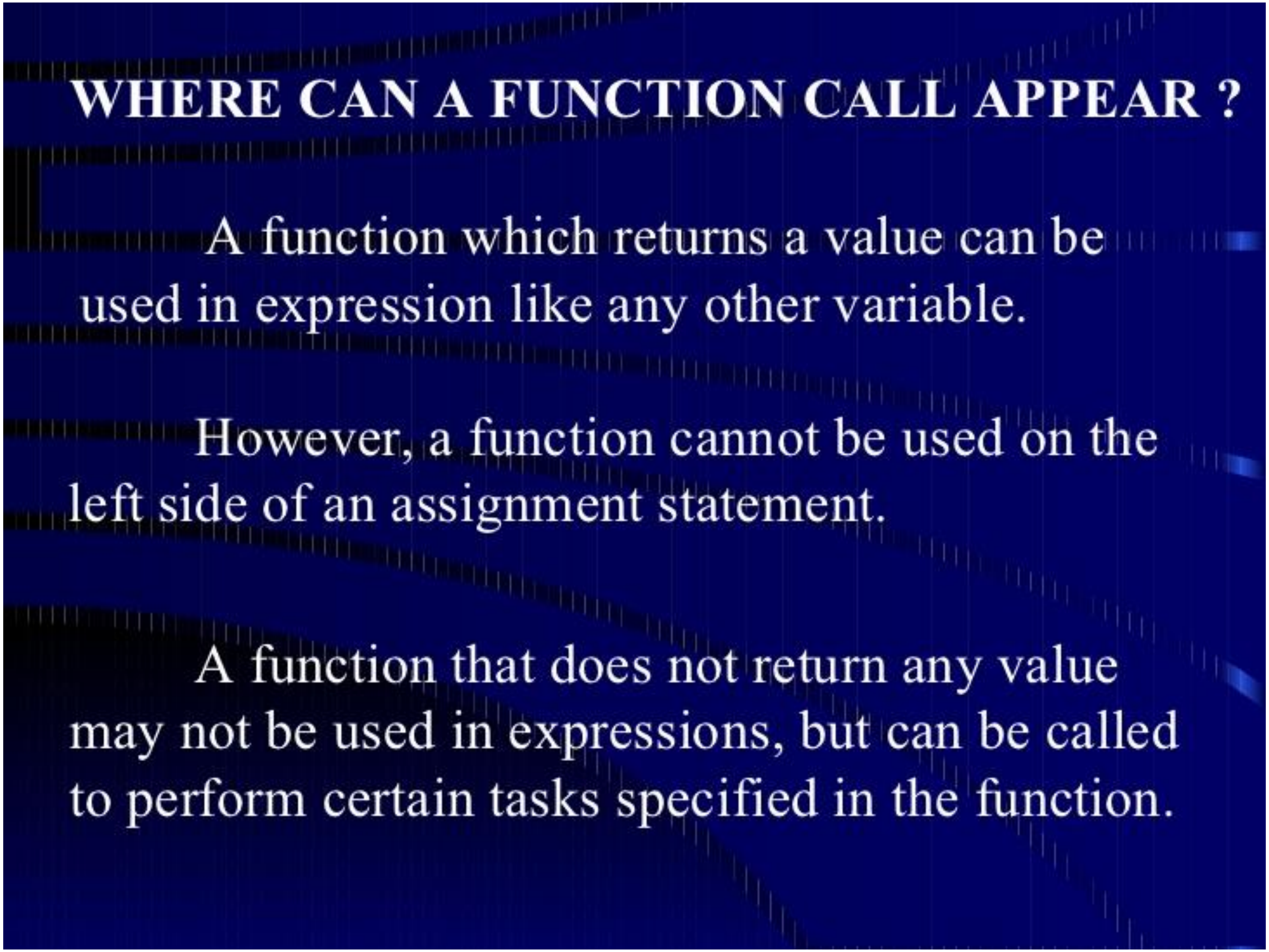


HOW TO CALL A FUNCTION ?

A function can be called by simply using the function name in a statement.

When the compiler encounters a function call, the control is transferred to the function.

The statements within the body of the function is executed line by line and the value is returned when a return statement is encountered.



WHERE CAN A FUNCTION CALL APPEAR ?

A function which returns a value can be used in expression like any other variable.

However, a function cannot be used on the left side of an assignment statement.

A function that does not return any value may not be used in expressions, but can be called to perform certain tasks specified in the function.

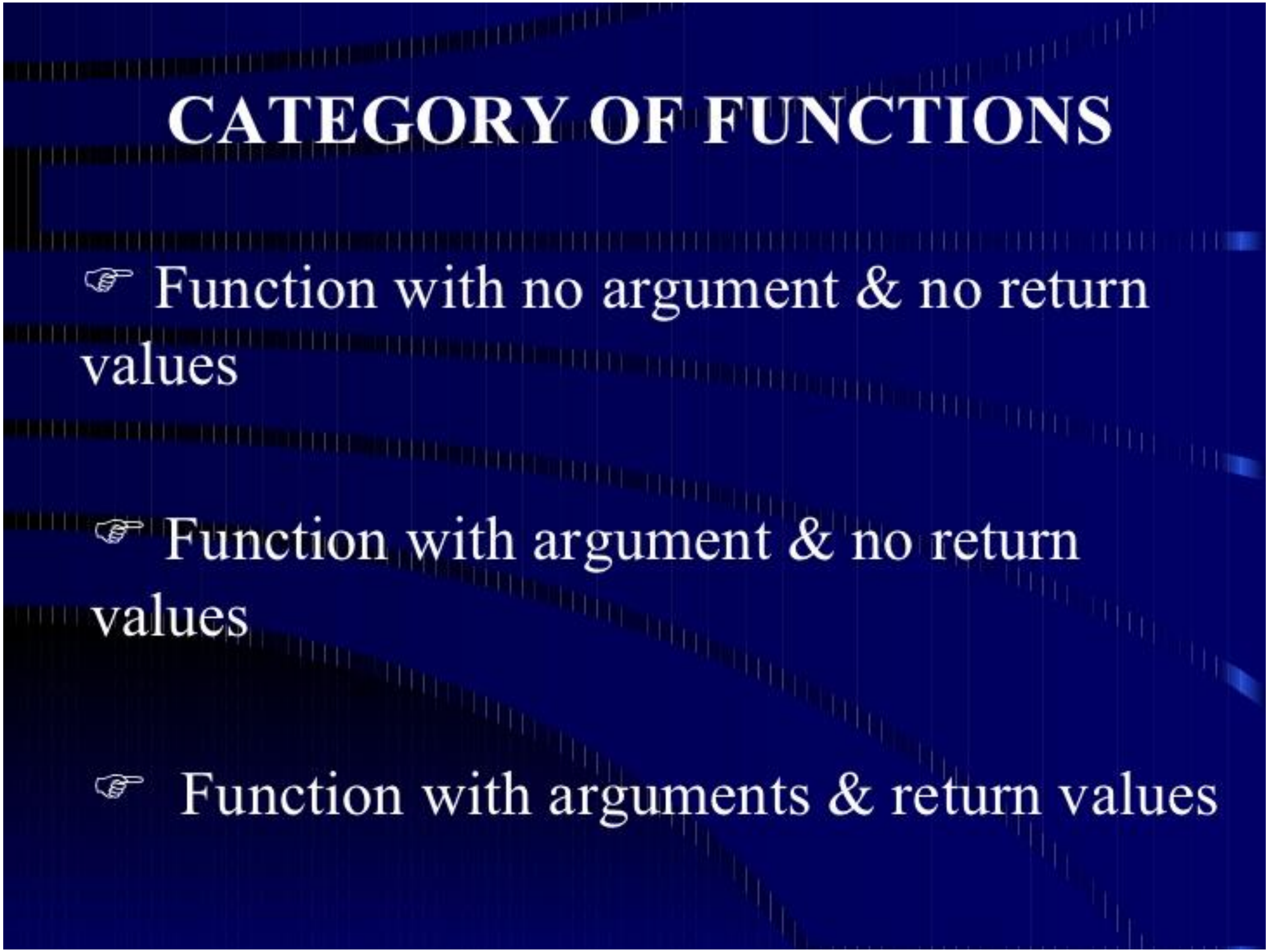
```
int add(int, int);  //or int add(int a, int b);

int main()
{
int a=10, b=20,sum=0;

sum=add(a,b);

printf("%d", sum);
}

int add(int x,int y)
{
 return(x+y);
}
```

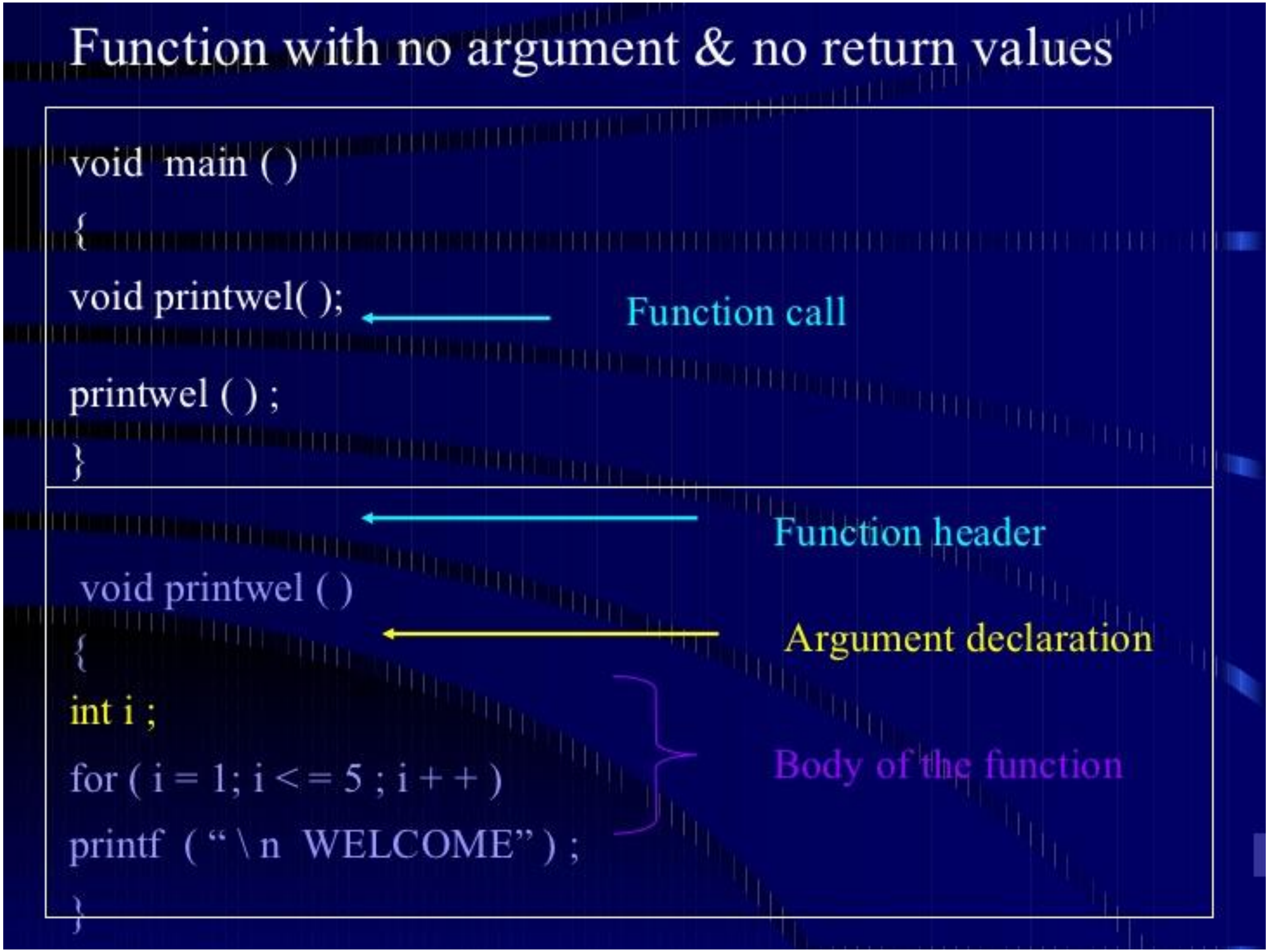


CATEGORY OF FUNCTIONS

☞ Function with no argument & no return values

☞ Function with argument & no return values

☞ Function with arguments & return values



Function with no argument & no return values

```
void  main ( )
{
void printwel( );

printwel ( ) ;
}
```

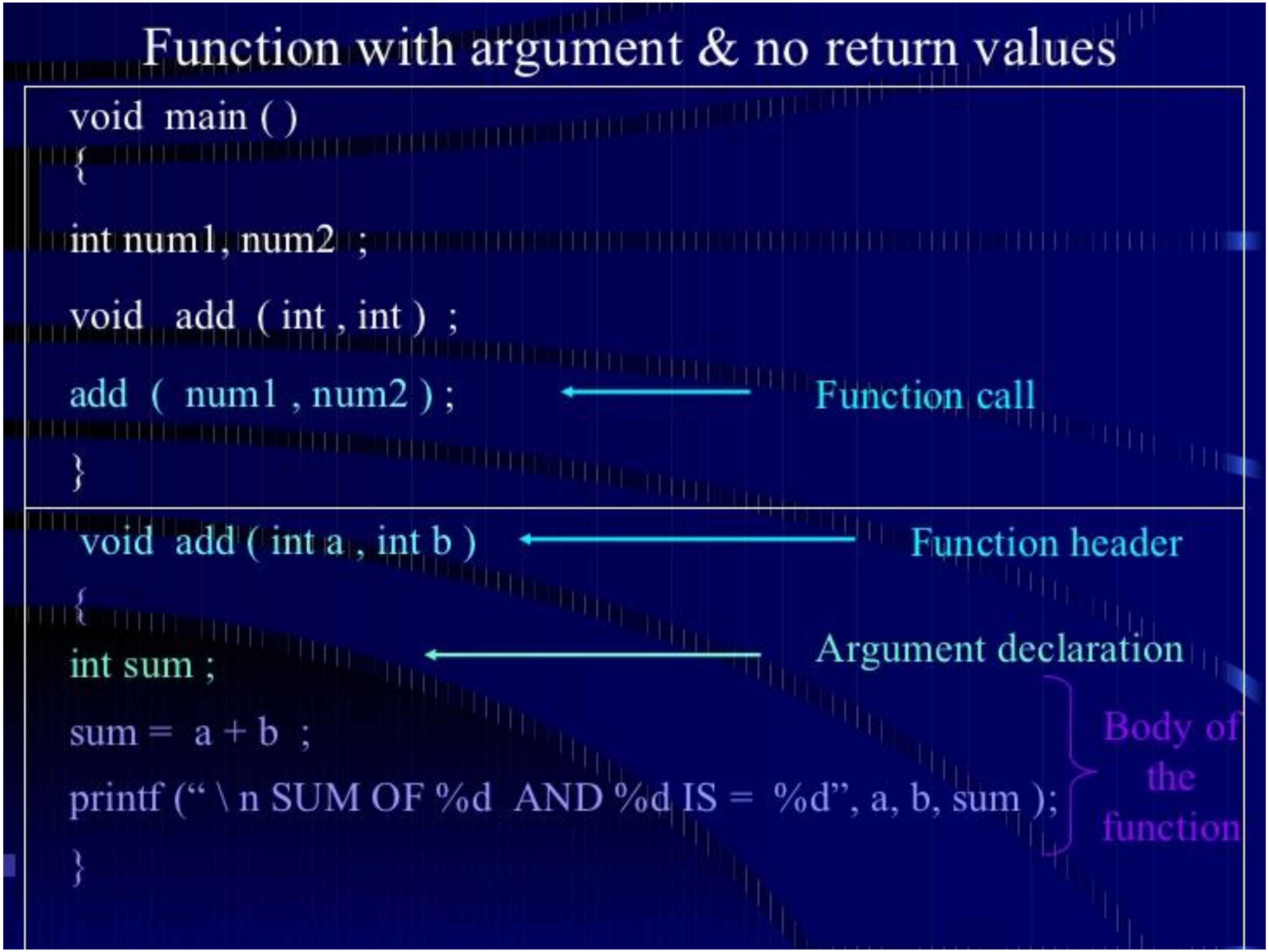
Function call

Function header

```
 void printwel ( )
{
int i ;
for ( i = 1; i < = 5 ; i + + )
printf  ( “ \ n  WELCOME” ) ;
}
```

Argument declaration

Body of the function



Function with argument & no return values

```
void  main ( )
{

int num1, num2  ;

void   add  ( int , int )  ;

add  (  num1 , num2 ) ;
}
```

```
 void  add ( int a , int b )
{
int sum ;
sum =   a + b  ;
printf (“ \ n SUM OF %d  AND %d IS =  %d”, a, b, sum );
}
```

Function call
Function header
Argument declaration
Body of the function

Function with arguments & return values

```
void  main ( )
{

int num1 ;

float num2 , sum , add ( ) ;
sum  =  add  (  num1 , num2 ) ;
printf ( " %f " , sum ) ;
}
```

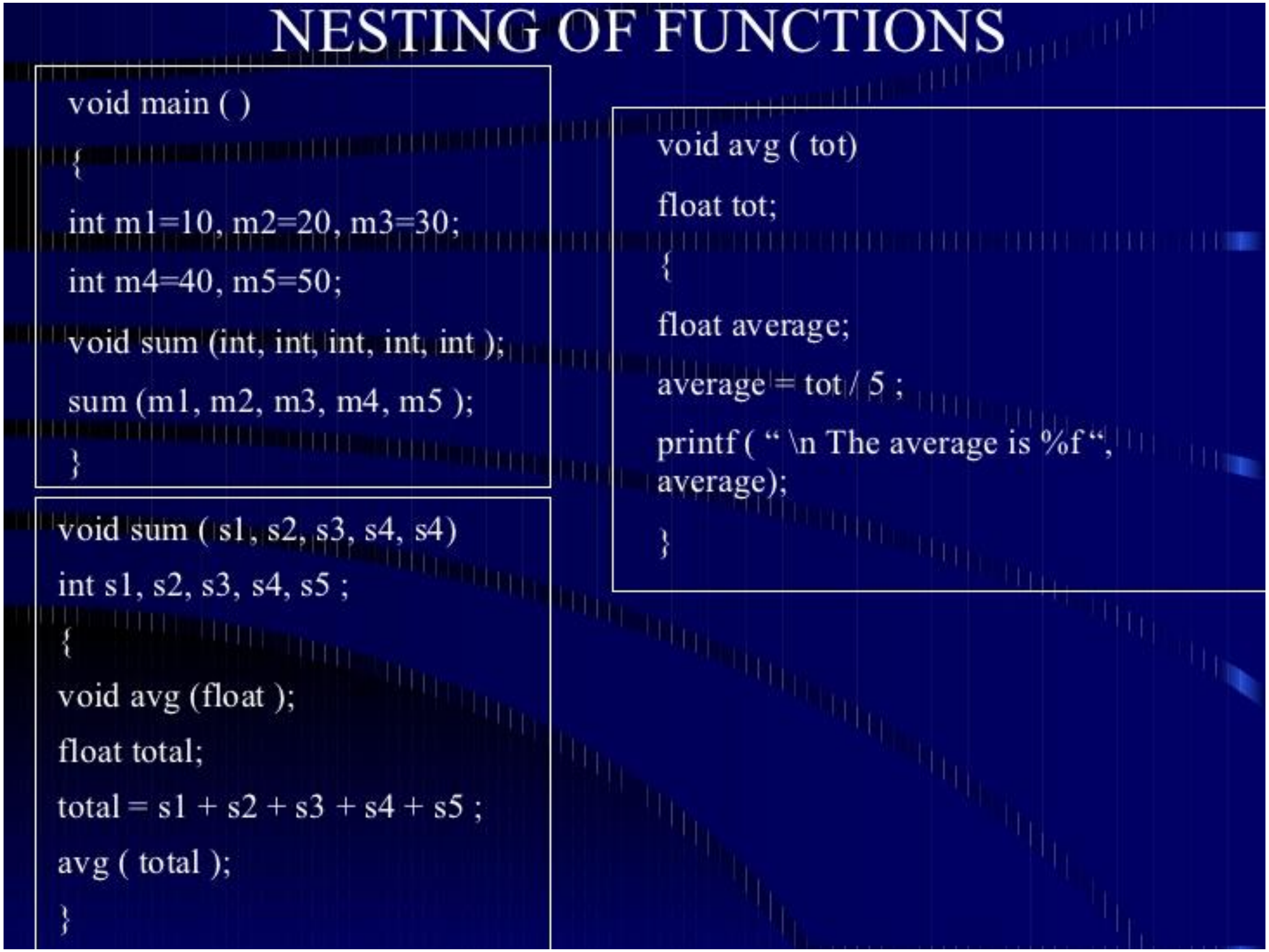
Function call

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

Function header

Argument declaration

Body of the function

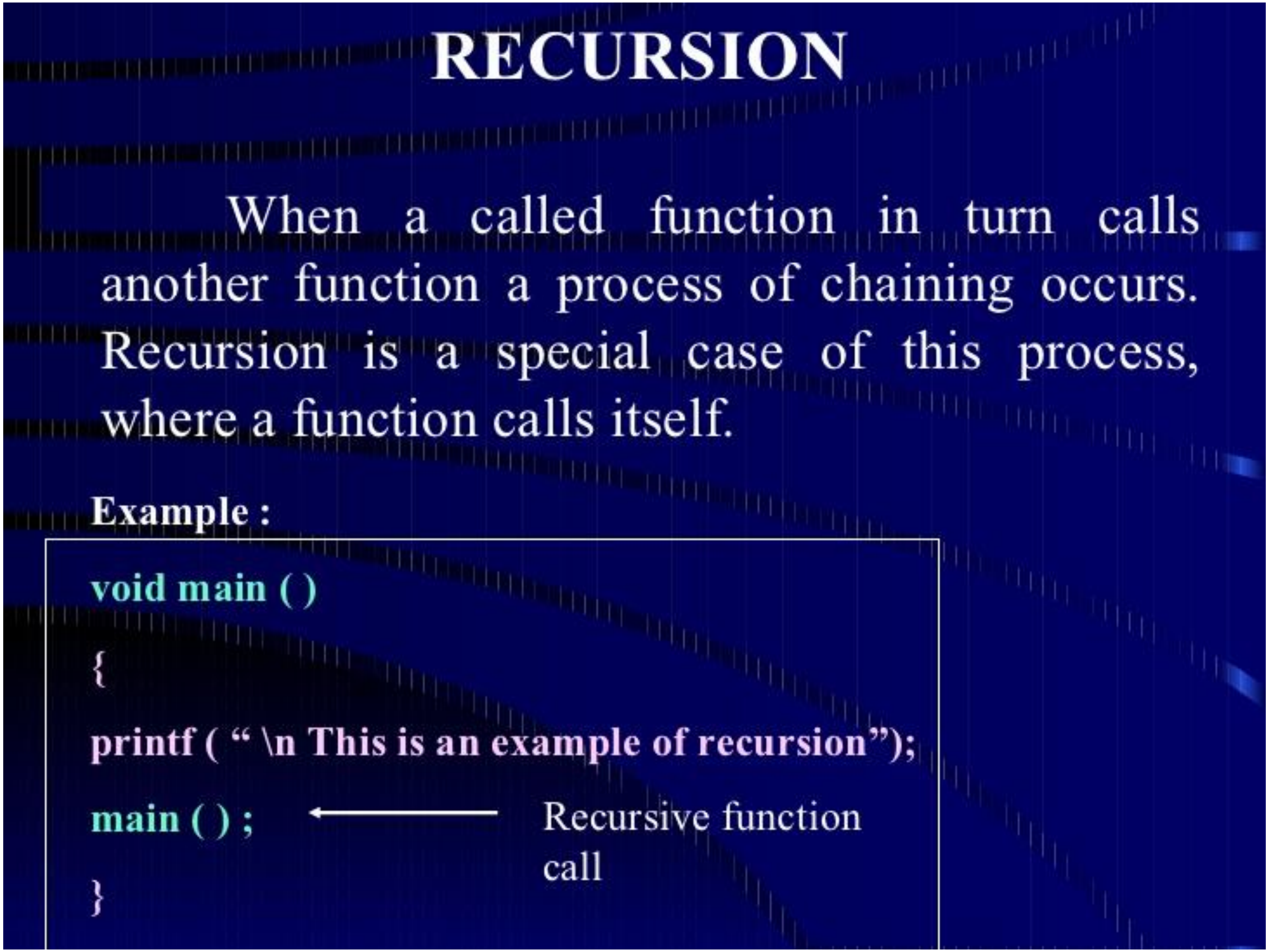


NESTING OF FUNCTIONS

```
void main ( )
{
int m1=10, m2=20, m3=30;
int m4=40, m5=50;
void sum (int, int, int, int, int );
sum (m1, m2, m3, m4, m5 );
}
```

```
void sum ( s1, s2, s3, s4, s4)
int s1, s2, s3, s4, s5 ;
{
void avg (float );
float total;
total = s1 + s2 + s3 + s4 + s5 ;
avg ( total );
}
```

```
void avg ( tot)
float tot;
{
float average;
average = tot / 5 ;
printf ( “ \n The average is %f “,
average);
}
```

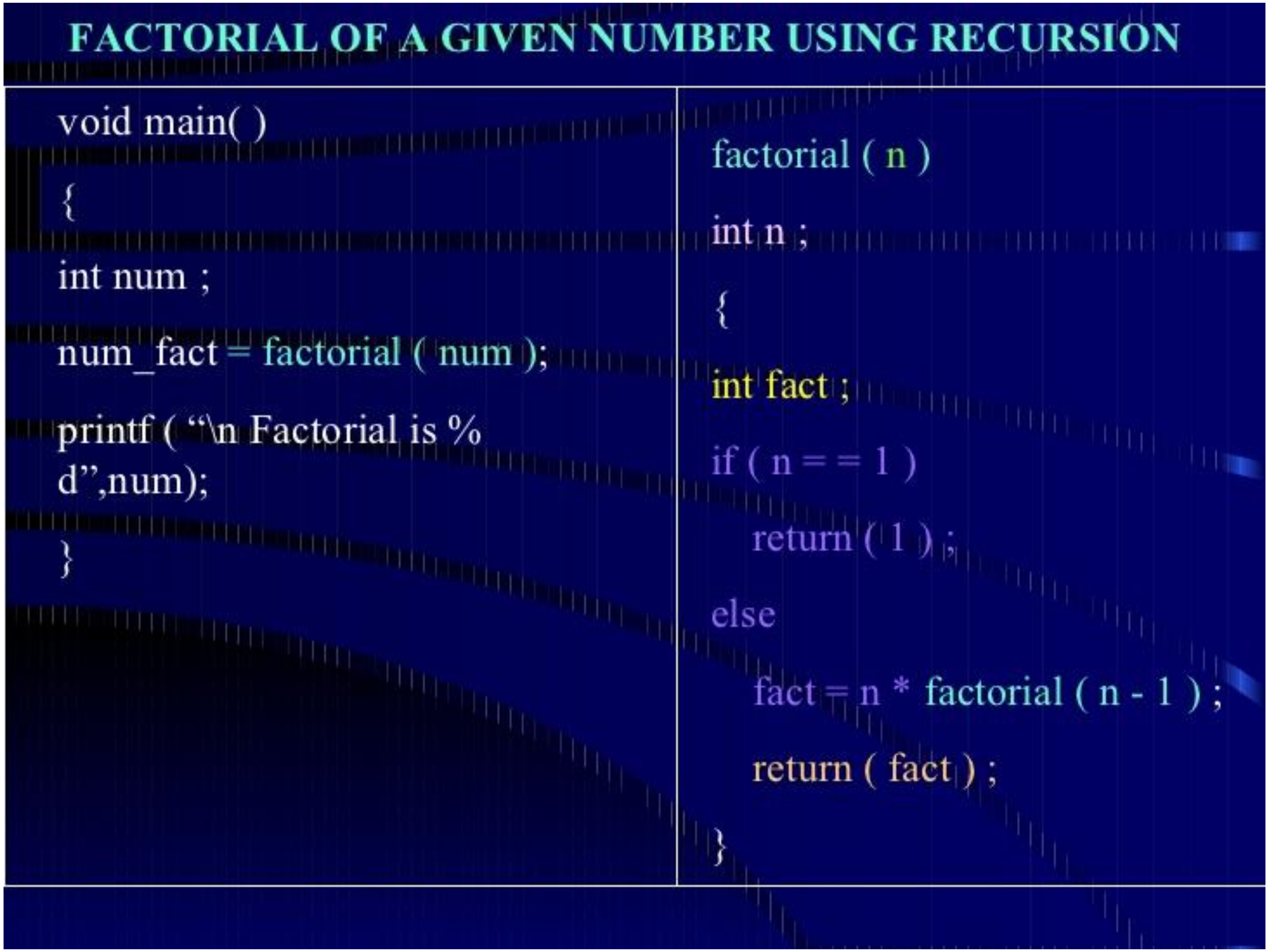


RECURSION

When a called function in turn calls another function a process of chaining occurs. Recursion is a special case of this process, where a function calls itself.

Example :

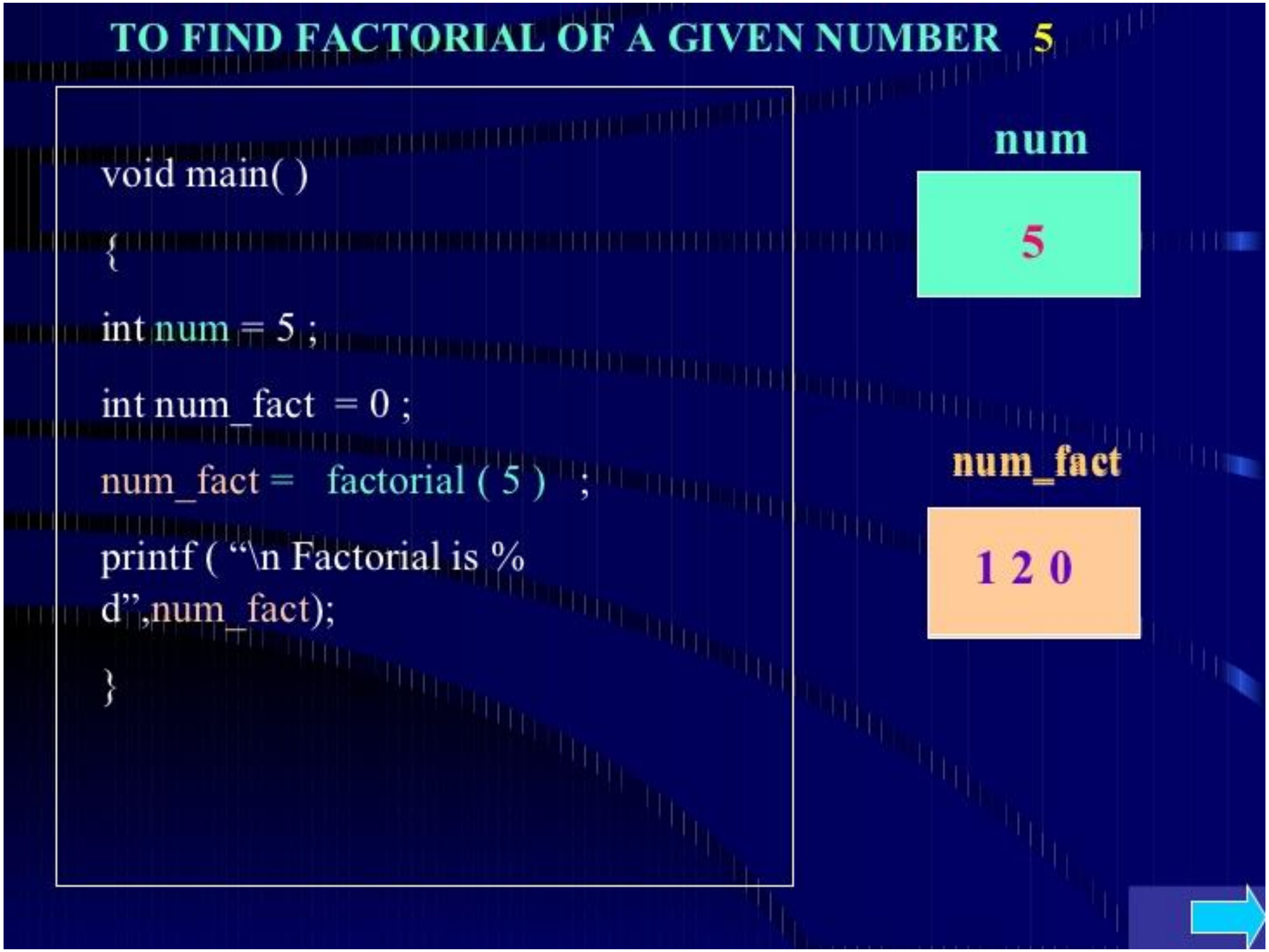
```
void main ( )
{
printf ( " \n This is an example of recursion");
main ( ) ;      ←――――――     Recursive function
                             call
}
```



FACTORIAL OF A GIVEN NUMBER USING RECURSION

```
void main( )
{
int num ;
num_fact = factorial ( num );
printf ( "\n Factorial is %
d",num);
}
```

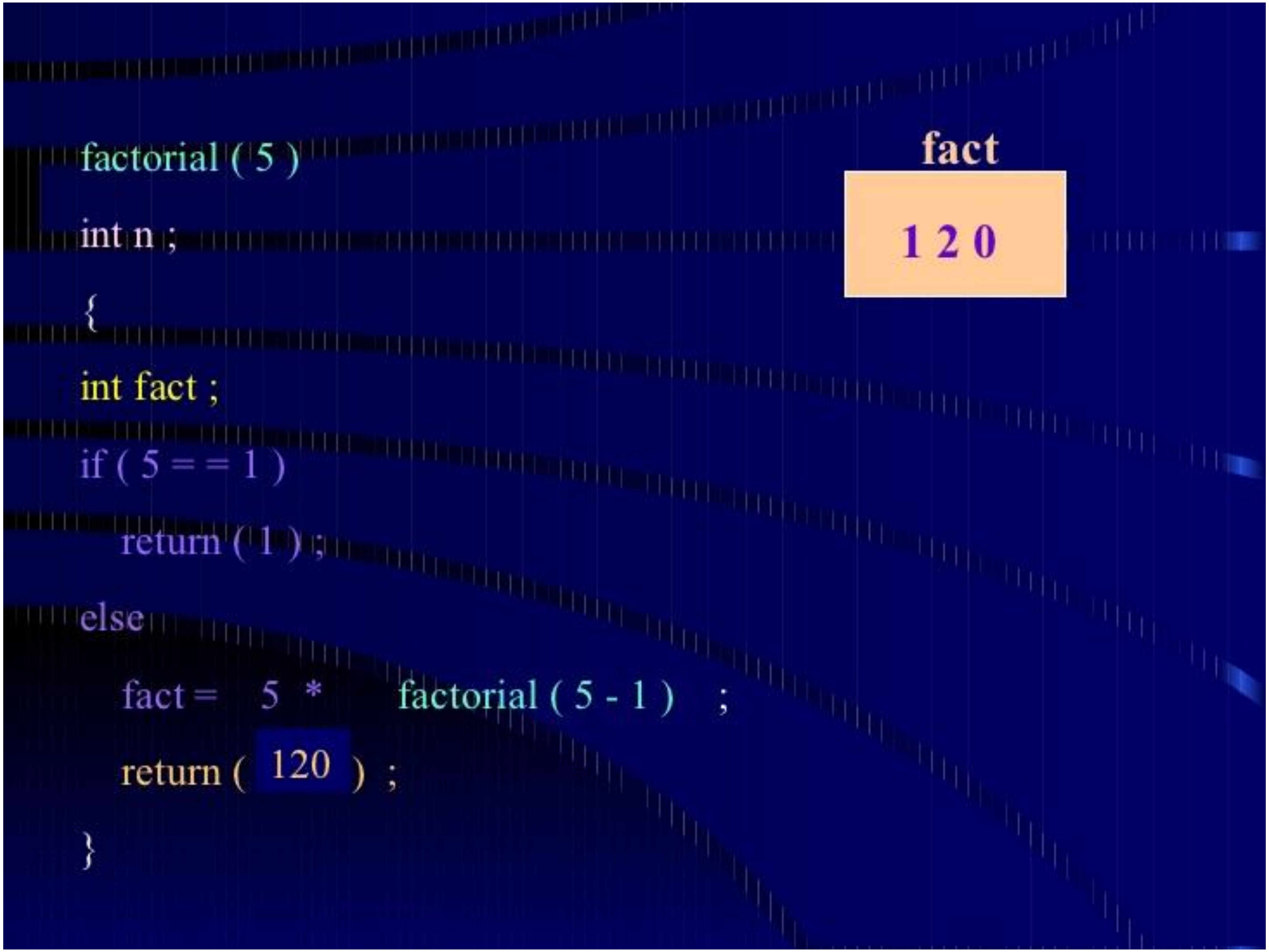
```
factorial ( n )
int n ;
{
int fact ;
if ( n = = 1 )
   return ( 1 ) ;
else
   fact = n * factorial ( n - 1 ) ;
   return ( fact ) ;
}
```



TO FIND FACTORIAL OF A GIVEN NUMBER 5

```
void main( )
{
int num = 5 ;
int num_fact  = 0 ;
num_fact =   factorial ( 5 )   ;
printf ( "\n Factorial is %
d",num_fact);
}
```

num
5
num_fact
1 2 0



```
factorial ( 5 )
int n ;
{
int fact ;
if ( 5 = = 1 )
    return ( 1 ) ;
else
    fact =    5  *      factorial ( 5 - 1 )    ;
    return ( 120 )  ;
}
```

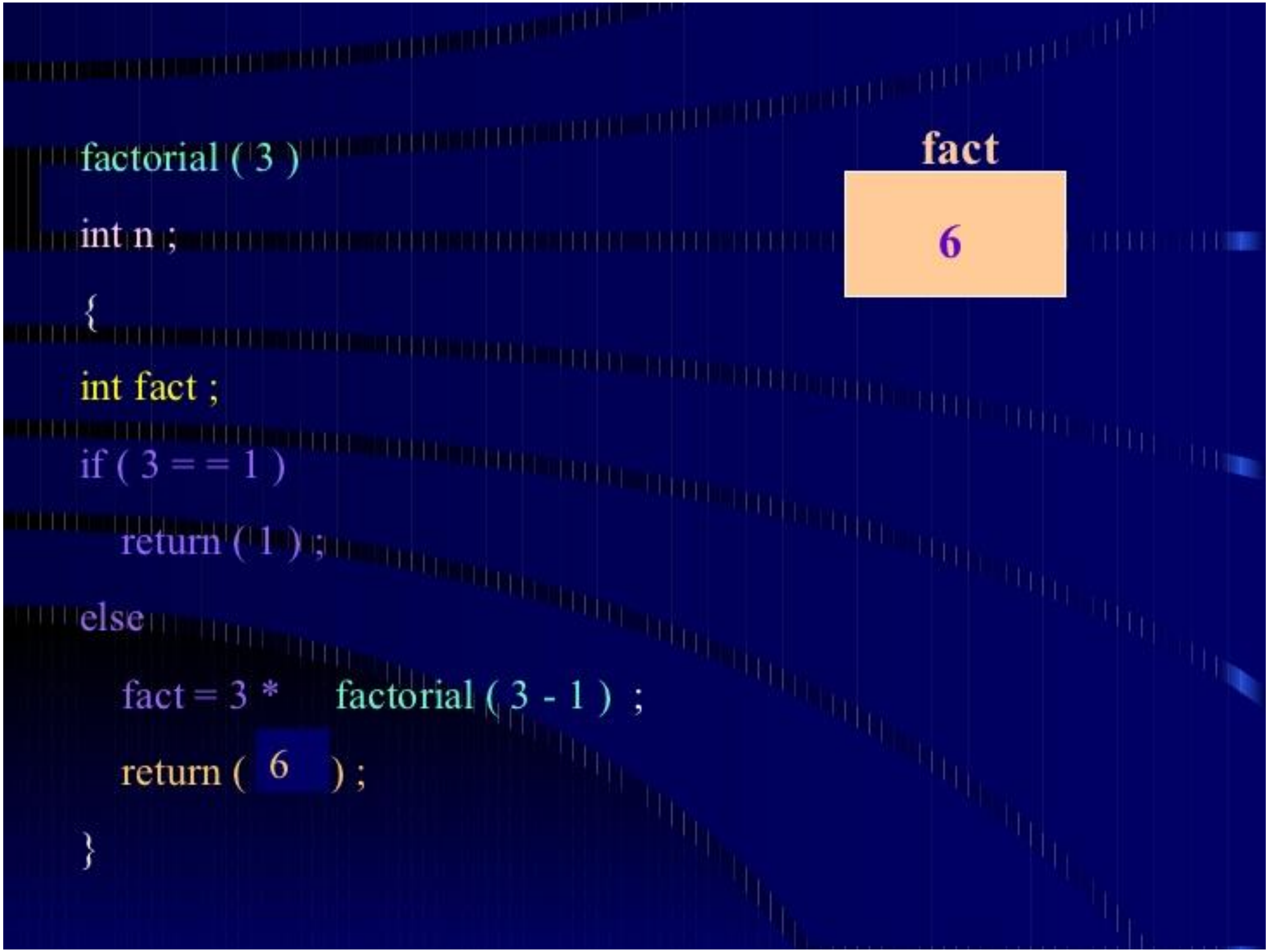
fact

1 2 0

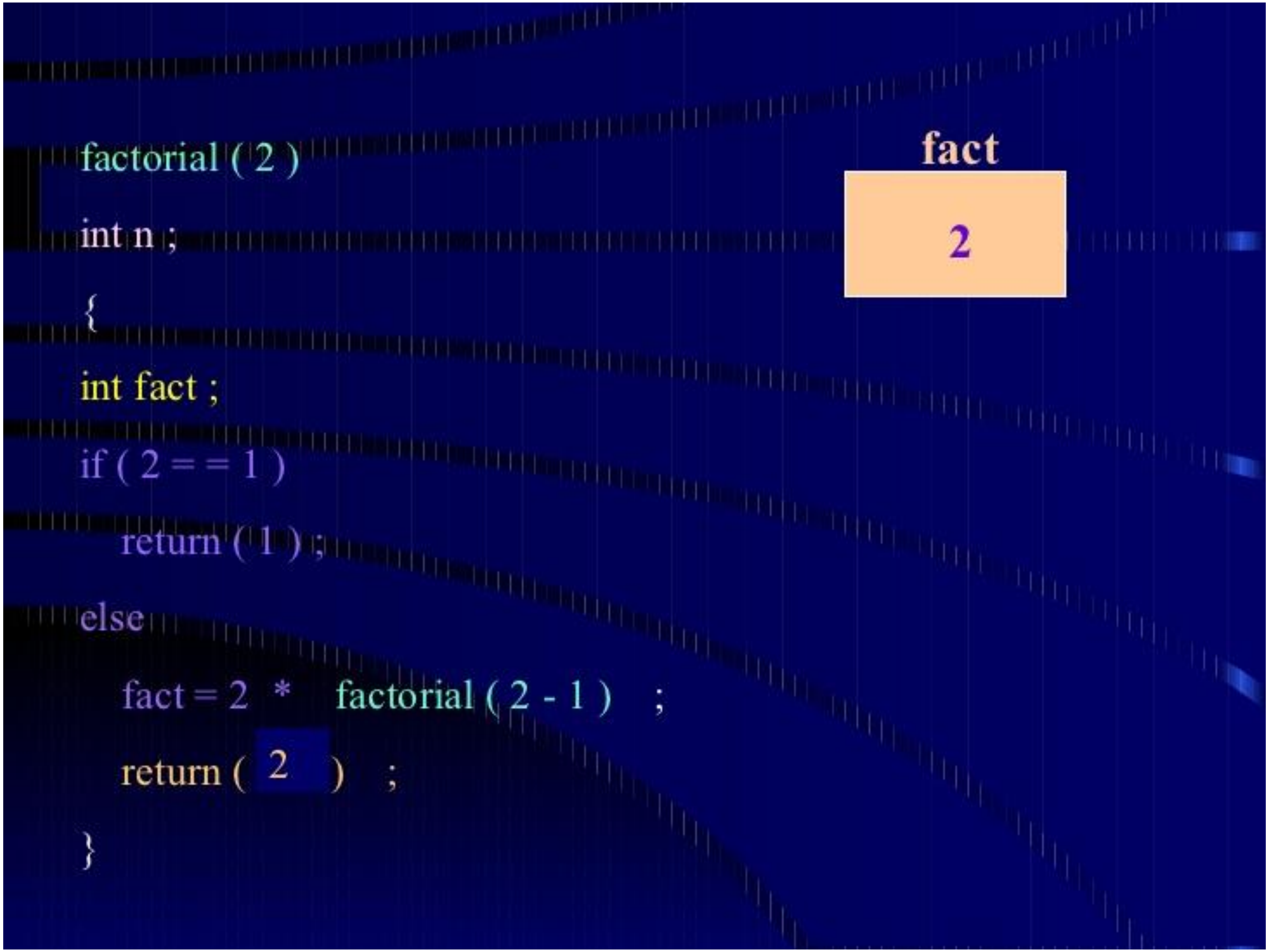
```
factorial ( 4 )
int n ;
{
int fact ;
if ( 4 = = 1 )
   return ( 1 ) ;
else
   fact =  4  *    factorial ( 4 - 1 )    ;
   return ( 24 ) ;
}
```

fact

2 4



```
factorial ( 3 )
int n ;
{
int fact ;
if ( 3 = = 1 )
   return ( 1 ) ;
else
   fact = 3 *     factorial ( 3 - 1 )  ;
   return (  6   ) ;
}
```



```
factorial ( 2 )
int n ;
{
int fact ;
if ( 2 = = 1 )
    return ( 1 ) ;
else
    fact = 2  *    factorial ( 2 - 1 )    ;
    return (  2   )    ;
}
```

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

fact

1

Call by Reference

```
#include<stdio.h>


int main()
{
int a=10, b=20;
printf("a= %d b=%d\n", a,b);

add(&a,&b);
printf("\na= %d b=%d\n", a,b);

}

void add(int *a,int *b)
{
printf("\na= %d b=%d\n", *a,*b);
*a=30;
*b=60;
printf("\na= %d b=%d\n", *a,*b);

}
```

Call by Value

```
#include<stdio.h>
void add(int a,int b);

int main()
{
int a=10, b=20;
printf("a= %d b=%d\n", a,b);

add(a,b);
printf("\na= %d b=%d\n", a,b);

}

void add(int a,int b)
{
printf("\na= %d b=%d\n", a,b);
 a=40; b=60;

 printf("\na= %d b=%d\n", a,b);
}
```