LECTURE 21

FUNCTIONS

Functions in C

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

Once a function has been designed and packed, it can be treated as a black box that takes some data from the main program and returns a value. The inner details of the operation are invisible to the rest of the program. All that the program knows about a function is : What goes in and what comes out.

Every C program can be designed using a collection of these black boxes known as functions.

C functions can be classified into two categories :

1. **Library functions : This functions are built-in functions and not required to be written by user.**

   example : sqrt, printf, scanf, exp( ).

2. **Used-Defined functions : It has to be developed by the user at the time of writing a program. However it can later become a part of the C program library.**

   example : main( ).

Need for user-defined functions

- It makes program easier to understand, debug and test.
- It saves both time and space.
- It facilitates top-down modular programming. In this programming style, the high level logic of the overall problem is solved first while the details of each lower-level functions are addressed later.
- The length of a source program can be reduced by using functions at appropriate places.
- It is easy to locate and isolate a faulty function for further investigation.
- A function may be used by many other programs.

LECTURE 22

FORM OF C FUNCTION

In order to make use of a user-defined function, we need to establish three elements that are related to functions :

1. **Function definition**
2. **Function call**
3. **Function declaration**

- **The function definition is an independent program module that is specially written to implement the requirements of the function.**
- **In order to use this function we need to invoke it at a required place in the program. This is known as function call. The program (or a function) that calls the function is referred to as the calling program or calling function.**
- **The calling program should declare any function (like declaration of a variable) that is to be used later in the program. This is known as function declaration or function prototype.**

Function declaration

Like variables, all functions in C program must be declared, before they are invoked. A function declaration also known as function prototype consist of four parts :

1. Function type
2. Function name
3. Parameter list
4. Terminating semicolon

They are coded in the following format :

return-type function-name (parameter list);

For example : int add(int a, int b);

Return type | function name | parameter list | semicolon

Definition of a function

A function definition, also known as function implementation shall include the following elements :

1. **Function name**
2. **Function type**
3. **List of parameters**
4. **Local variable declaration**
5. **Function statements**
6. **A return statement**

All six elements are grouped into two parts :

1. **Function header**
2. **Function body**

```
Function_type function_name(parameter list)
{
 local variable declaration;
 executable statement1;
 executable statement2;
 .....
 .....
 return statement;
}
```

For example :

```
int add(int a, int b)
{
   int c;
   c = a + b;
   printf("%d",c);
   return c;
  }
```

Function call

A function can be called by simply using the function name followed by a list of actual parameters (or arguments), if any, enclosed in parentheses.

```
Void main( )
{
    void add(int a , int b);
    int c;
    printf("Enter the values of a & b:");
    scanf("%d %d" ,&a ,&b);
    c = add(a, b);
    printf("%d" ,c);
  }
```

LECTURE 23

RETURN VALUES AND THEIR TYPES

RETURN VALUES AND THEIR TYPES

A function may or may not send back any value to the calling function.

If it does, it is done through the return statement.

A called function can return only one value per call at the most.

The return statement can take one of the following form:

return;

OR

return(expession);

The plane return does not return any value to the calling function, it acts much as the closing brace of the function.

When a return is encountered the control is immediately passed back to the calling function.

For example :

```
mul(x,y)
int x,y;
  int p;
  p=x*y;
  return(p);
}
```

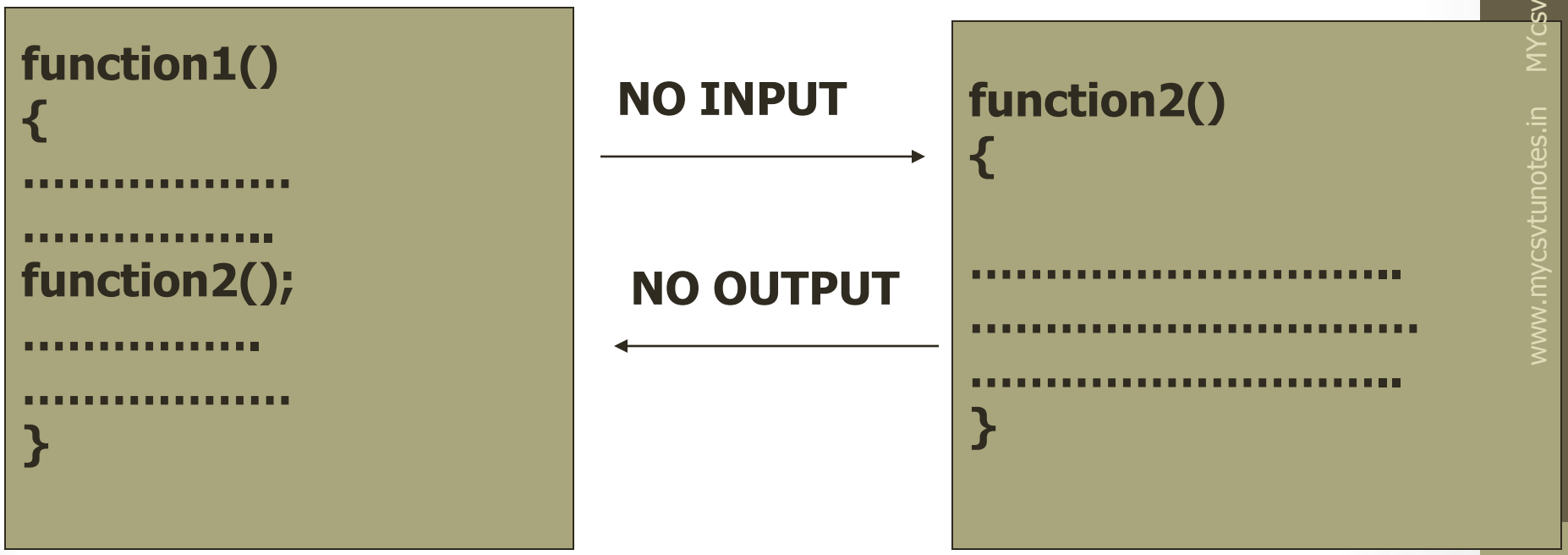
If we do not specify any return type then by default compiler assumes it as int.

LECTURE 24

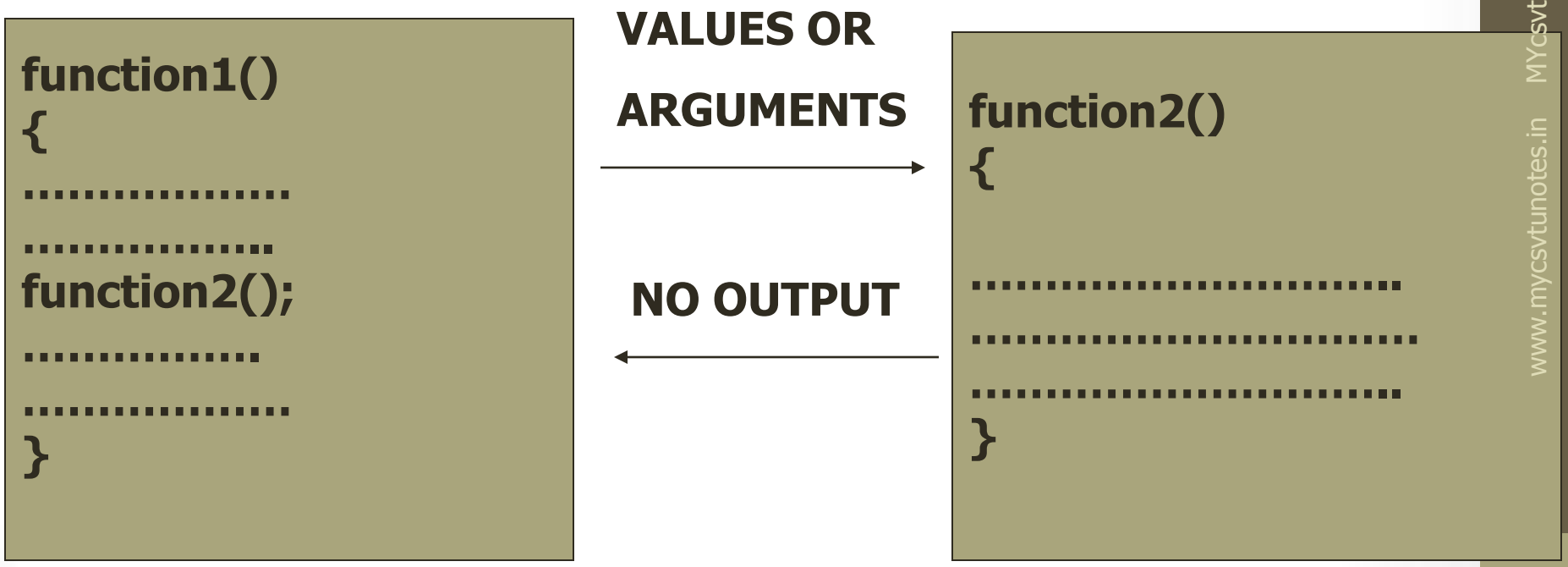
CATEGORY OF FUNCTIONS

CATEGORIES OF FUNCTIONS

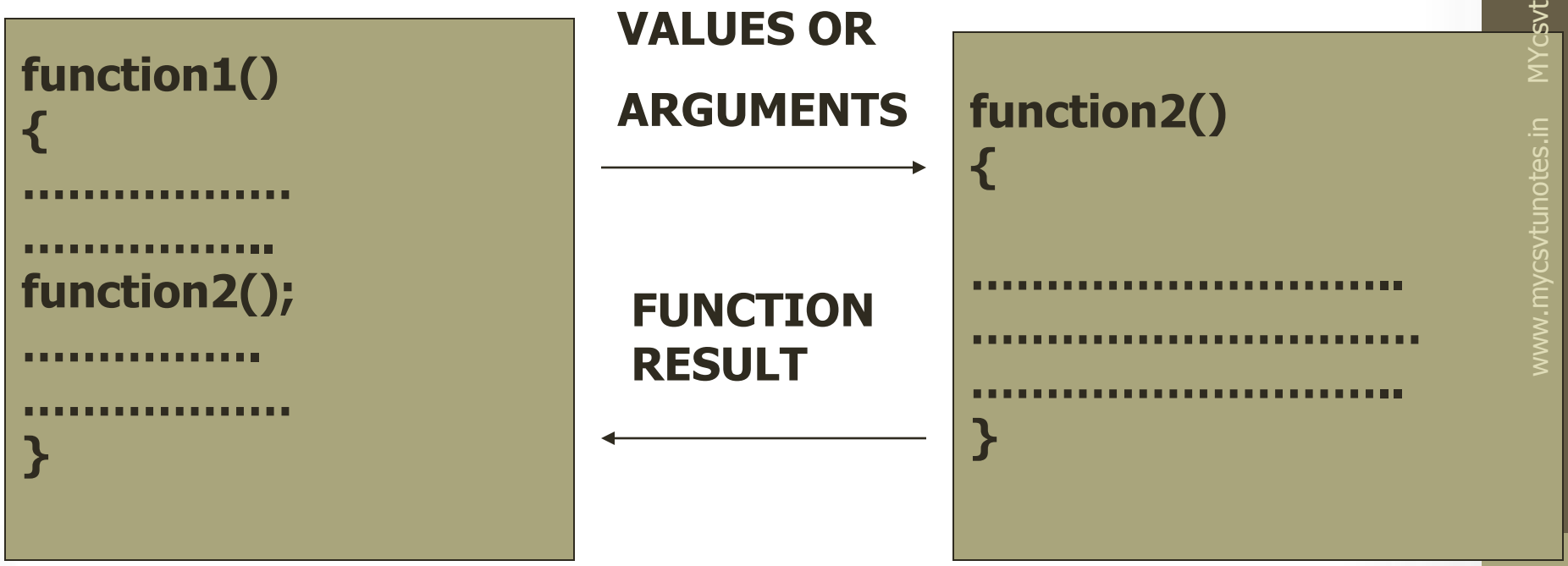
1. Function with no argument and no return value:



2. Function with arguments but no return value:



3. Function with arguments and with return value:



Calling Functions: Call by Value and Call by Reference

- Used when invoking functions
- Call by value
  - Copy of argument passed to function
  - Changes in function do not effect original
  - Use when function does not need to modify argument
    - Avoids accidental changes
- Call by reference
  - Passes original argument
  - Changes in function effect original
  - Only used with trusted functions
- For now, we focus on call by value

Pointers as function arguments

We can pass the address of a variable as an argument to a function.

When we pass addresses to a function, the parameters receiving the addresses should be pointers.

The process of calling a function using pointers to pass the addresses of variables is known as call by reference.

The function which is called by reference can change the value of the variable used in the call.

This mechanism is also known as call by address or pass by pointers.

1. **The function parameters are declared as pointers.**
2. **The dereferencing pointers are used in the function body.**
3. **When the function is called, the addresses are passed as actual arguments.**

Functions returning pointers

A function can return a single value by its name or return multiple values through pointer parameters.

Since pointers are a data type in C, we can also force a function to return a pointer to the calling function.

LECTURE 25

NESTING OF FUNCTION

Nesting of functions

C permits nesting of functions freely. main can call function1, which calls function2, which calls function3,…… and so on. There is in principle no limit as to how deeply functions can be nested.

STORAGE CLASSES

In C not only all variables have a data type, they also have storage class that provides information about their location and visibility.

The storage class decides the portion of the program within which the variables are recognized.

The following storage classes are most relevant to functions :

1. Automatic variables
2. External variables
3. Static variables
4. Register variables

Scope : It determines over what region of the program a variable is actually available for use (‘active’).

Longevity : It refers to the period during which a variable retains a given value during execution of a program (alive).

Visibility : It refers to the accessibility of a variable fro the memory.

Automatic variables

Automatic variables are declared inside a function in which they are to be utilized.

They are created when the function us called and destroyed automatically when the function is exited, hence the name automatic.

Automatic variables are therefore private(or local) to the function in which they are declared.

Because of this property, automatic variables are also reffered to as local or internal variables.

A variable declared inside a function without storage class specification is, by default, an automatic variable.

We may also use the keyword auto to declare automatic variables explicitly.

We may declare and use the same variable name in different functions in the same program without causing any confusion to the compiler.

EXTERNAL VARIABLES

Variables that are both alive and active throughout the entire program are known as external variables.

They are also known as global variables.

Unlike local variables, global variables can be accessed by any function in the program.

External variables are declared outside a function.

```
    int number;
    float length =7.5;
   void main( )
  {
     ……..
    ……..
  }
  function1( )
 {
    ……..
    ……
  }
  function2( )
{
   ……
   ……
}
```

In case a local variable and a global variable have the same name, the local variable will have precedence over the global one in the function where it is declared.

```
int count;
  void  main( )
  {
    count=10;
    ……..
    ……..
  }
  function()
  {
    int count=0;
    count=count+1;
    ……..
  }
```

Once a variable has been declared as global, any function can use it and change its value.

Then subsequent functions can reference only that new value.

One another aspect of a global variable is that it is available only from the point of declaration to the end of the program.

For example :

```
 void main( )
 {
    y=5;
    ……
    ……..
 }
 int y;
func1( )
 {
  y=y+1;
 }
```

In the program above, the main cannot access the variable as it has been declared after the main function.

This problem can be solved by declaring the variable with the storage class extern.

```
void main( )
{
   extern int y;
  ……..
  ……
}
func1( )
{
  extern int y;
……
}
int y;
```

Although the variable y has been defined after both the functions, the external declaration of y inside the functions informs the compiler that y is an integer type defined somewhere else in the program.

The extern declaration does not allocate storage space for variables.

In case of array the definition should include their size as well.

An extern within a function provides the type information to just that one function.

We can provide type information to all functions within a file by placing external declaration before any of them.

The distinction between definition and declaration also applies to functions.

A function is defined when its parameters and function body are specified.

This tells the compiler to allocate space for the function code and provides type information for the parameters.

Since the functions are external by default, we declare them without the qualifier extern.

We have been assumed that all the functions are defined in one file.

However in real-life programming environment, we may use more than one source files which may be compiled separately and linked later to form an executable object code.

This approach is very useful because any change in one file does not affect other files thus eliminating the need for recompilation of the entire program.

Multiple source files can share a variable provided it is declared as an external variable appropriately.

Variables that are shared by two or more files are global variables and therefore we must declare them accordingly in one file and then explicitly declare them with extern in other files.

The extern specifier tells the compiler that the following variable types and names have already been declared elsewhere and no need to create storage space for them.

It is the responsibility of the linker to resolve the reference problem.

Multifile global variables are declared without extern in one of the files.

The extern declaration is done in places where secondary references are made.

When a function is defined in one file and accessed in another, the later file must include a function declaration.

The declaration identifies the function as an external function whose definition appears elsewhere.

We usually place such declarations at the beginning of the file, before all functions.

Static variables

The value of static variables persists until the end of the program.

A variable can be declared static using the keyword static.

```
static int y;

static float x;
```

A static variable may be either an internal type or an external type depending on the place of declaration.

Internal static variables are those which are declared inside a function.

The scope of internal variable extend up to the end of the function in which they are defined. Therefore internal static variables are similar to auto variables except hat they remain in existence throughout the remainder of the program.

Therefore internal static variables can be used to retain values between function calls.

A static variable is initialized only once, when the program is compiled.

It is never initialized again.

Register variables

We can tell the compiler that a variable should be kept in one of the machine's register, instead of keeping in the memory (where normal variables are stored).

Since a register is much faster than a memory access, keeping the frequently accessed variables (e.g. loop control variables) in the register will lead to faster execution of programs.

This is done as follows :

```
register int count;
```

LECTURE 26

RECURSION

RECURSION

When a called function in turn calls another function a process of chaining occurs.

<u>Recursion</u> is a special case of this process where a function calls itself.

For example :

```
void main( )
{
  printf("this is an example of recursion\n");

   main( );
}
```

Recursion

- Recursive functions
  - Functions that call themselves
  - Can only solve a base case
  - Divide a problem up into
    - What it can do
    - What it cannot do
      - What it cannot do resembles original problem
      - The function launches a new copy of itself (recursion step) to solve what it cannot do
  - Eventually base case gets solved
    - Gets plugged in, works its way up and solves whole problem

Recursive functions can be effectively used to solve problems where solution is expressed in terms of successive applying the same solution to subsets of the problem.

When we write recursive functions, we must have an if statement to force the function to return without the recursive call being executed. Otherwise, the function will never return.

Recursion

- Example: factorials
  - `5! = 5 * 4 * 3 * 2 * 1`
  - Notice that
    - `5! = 5 * 4!`
    - `4! = 4 * 3!` …
  - Can compute factorials recursively
  - Solve base case (`1! = 0! = 1`) then plug in
    - `2! = 2 * 1! = 2 * 1 = 2;`
    - `3! = 3 * 2! = 3 * 2 = 6;`

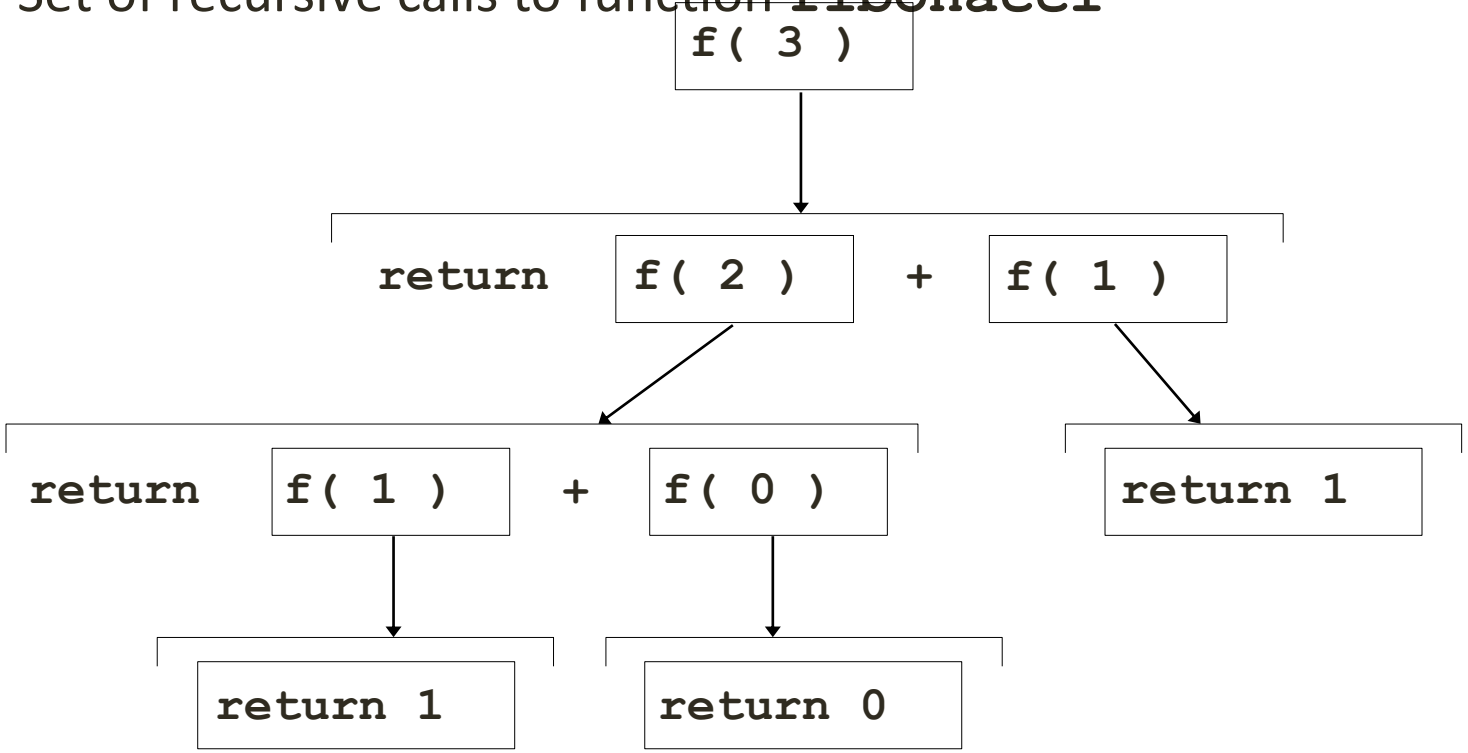
Example Using Recursion: The Fibonacci Series

- Fibonacci series: 0, 1, 1, 2, 3, 5, 8...
  - Each number is the sum of the previous two
  - Can be solved recursively:
    - **`fib( n ) = fib( n - 1 ) + fib( n - 2 )`**
  - Code for the **`fibaonacci`** function

```
long fibonacci( long n )
{
  if (n == 0 || n == 1)  // base case
     return n;
  else
     return fibonacci( n - 1) +
        fibonacci( n - 2 );
}
```

5.14 Example Using Recursion: The Fibonacci Series

- Set of recursive calls to function **fibonacci**



```c
/* Fig. 5.15: fig05_15.c
   Recursive fibonacci function */
#include <stdio.h>

long fibonacci( long );

int main()
{
   long result, number;

   printf( "Enter an integer: " );
   scanf( "%ld", &number );
   result = fibonacci( number );
   printf( "Fibonacci( %ld ) = %ld\n", number, result );
   return 0;
}

/* Recursive definition of function fibonacci */
long fibonacci( long n )
{
   if ( n == 0 || n == 1 )
      return n;
   else
      return fibonacci( n - 1 ) + fibonacci( n - 2 );
}
```

```
Enter an integer: 0
Fibonacci(0) = 0

Enter an integer: 1
Fibonacci(1) = 1
```

1. Function prototype

1.1 Initialize variables

2. Input an integer

2.1 Call function `fibonacci`

2.2 Output results.

3. Define `fibonacci` recursively

```
Enter an integer: 2
Fibonacci(2) = 1

Enter an integer: 3
Fibonacci(3) = 2

Enter an integer: 4
Fibonacci(4) = 3

Enter an integer: 5
Fibonacci(5) = 5

Enter an integer: 6
Fibonacci(6) = 8

Enter an integer: 10
Fibonacci(10) = 55

Enter an integer: 20
Fibonacci(20) = 6765

Enter an integer: 30
Fibonacci(30) = 832040

Enter an integer: 35
Fibonacci(35) = 9227465
```

Program Output

Recursion vs. Iteration

- Repetition
  - Iteration: explicit loop
  - Recursion: repeated function calls
- Termination
  - Iteration: loop condition fails
  - Recursion: base case recognized
- Both can have infinite loops
- Balance
  - Choice between performance (iteration) and good software engineering (recursion)

LECTURE 27

FUNCTIONS WITH ARRAY

Passing arrays to a function

One dimensional arrays :

To pass a one-dimensional array to a called function, it is sufficient to list the name of the array, without any subscripts and the size of the array as arguments.

For example :

Declaration : void largest (int [ ] , int);

Calling : largest (a , n)

Definition : void largest (int a[ ] , int n)

{

}

Three rules to pass an array to a funcion

1. The function must be called by passing only the name of the array.

   for example : largest (a);

2. In the function definition, the formal parameter must be an array type, the size of the array does not need to be specified.

   for example : void largest (int a[ ])

   {

   }

3. The function prototype must show that the argument is an array.

   for example : void largest (int [ ]);

Two dimensional arrays

The rules are :

1. **The function must be called by passing only the array name.**

   for example : average(matrix,m,n);

2. **In the function definition, we must indicate that the array has two dimensions by including two sets of brackets.**

   for example : void average(int matrix[ ][n],int m,int n)

   {

   }

3. **The size of the second dimension must be specified.**

4. **The prototype declaration should be similar to function header.**

   for example : void average(int matrix[ ][n],int,int);

Passing strings to functions

1. **The string to be passed must be declared as a formal argument of the function when it is defined.**

   for example : void display (char str[ ])

   {

   }

2. **The function prototype must show that the argument is a string**

   void display(char str[ ]);

3. **A call to the function must have a string array name without subscripts as its actual argument.**

   display(str);

LECTURE 28

STRUCTURE DEFINITION

STRUCTURES

A structure is a collection of data items of different types.

A structure is a convenient tool for handling a group of logically related data items.

For example it can be used to represent a set of attributes, such as student_name, roll_number and marks.

Defining a structure

Structures must be defined first for their format that may be used later to declare structure variable.

The general format of a structure definition is as follows :

```
struct tag_name
{
    data_type member1;
    data_type member2;
     ……….
     ……….
};
```

For example :

```
struct student
{
    int roll_no.
    char name[20];
    int m1, m2, m3;
   float per;
};
```

In defining a structure you may note the following syntax :

1. **The template is terminated with semicolon.**
2. **While the entire definition is considered as a statement, each member is declared independently for its name and type in a separate statement inside the template.**
3. **The tag_name can be used to declare structure variables of its type, later in program.**

Declaring structure variables

After defining a structure format we can declare variables of that type.

A structure variable declaration is similar to the declaration of variables of any other data types.

It includes the following elements :

1. **The keyword struct.**
2. **The structure tag name.**
3. **List of variable names separated by commas.**
4. **A terminating semicolon.**

For example : struct student s1,s2;

The members of a structure themselves are not variables. They do not occupy any memory until they are associated with the structure variables such as student.

When the complier comes across a declaration statement, it reserves memory space for the structure variables.

It is also allowed to combine both the structure definition and variables in one statement.

Accessing structure members

The members themselves are not variables, they should be linked to the structure variables in order to make them meaningful members.

The link between a member and a variable is established using member operator ‘ . ‘, which is also known as dot operator or period operator.

For example : s1.roll_no;

LECTURE 29

INITIALIZATION AND COMPARISION OF STRUCTURE

Initializing Structures

- Initializer lists
  - Example:

```
card oneCard = { "Three", "Hearts" };
```

- Assignment statements
  - Example:

```
card threeHearts = oneCard;
```

  - Could also declare and initialize **threeHearts** as follows:

```
card threeHearts;
threeHearts.face = “Three”;
threeHearts.suit = “Hearts”;
```

STRUCTURE COMPARISION

We can compare values of members of same structure just like as ordinary variables.

For example :

```
struct student
{
   int roll_no;
   char name[20];
   float per;
};
struct student s1, s2;
```

We can compare percentage as :

```
if(s1.per>=s2.per)
{
  ……….
  ……….
}
```

LECTURE 30

ARRAY OF STRUCTURES

ARRAYS OF STRUCTURES

Suppose we have to maintain details for all the students of a class.

In such cases we may declare an array of structures, each element of the array representing a structure variable.

For example :

struct student std[50];

Defines an array called std that consist of 50 elements.

Each element is defined to be of type struct list.

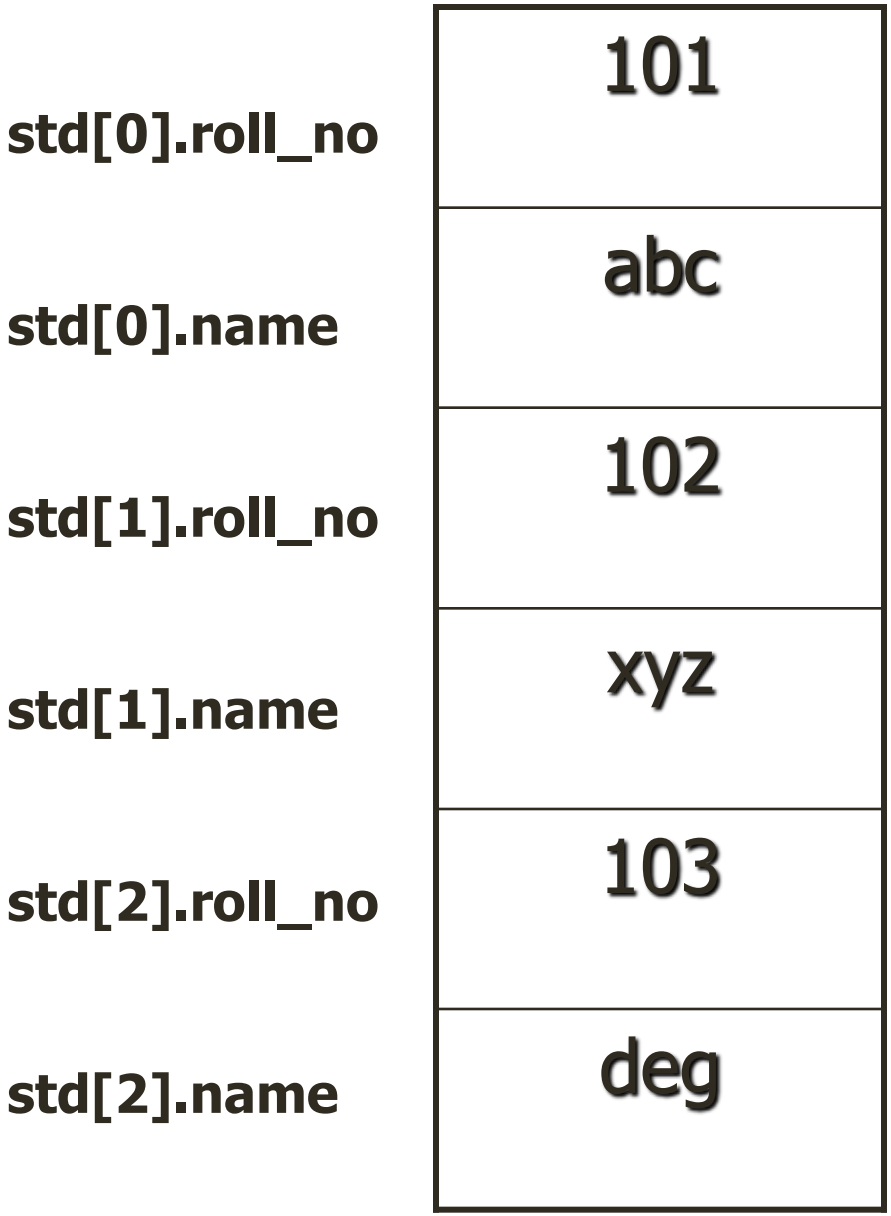
Consider the following declaration:

```
struct student
{
    int roll_no;
   char name[20];
};
struct student std[3]={{101,”abc”},{102,”xyz”},{103,”deg”}};
```

This declares std as an array of three elements std[0], std[1], std[2].

And initializes their roll_no as:

```
    std[0].roll_no=101
    std[1].roll_no=102
    std[2].roll_no=103
```


std[0].roll_no
101
std[0].name
abc
std[1].roll_no
102
std[1].name
xyz
std[2].roll_no
103
std[2].name
deg

LECTURE 31

UNION

Union

Unions are a concept borrowed from structures and therefore follow the same syntax as structure. However, there is a major difference between them in terms of storage. In structures, each member has its own storage location, whereas all the members of the union use the same location. This implies that although a union may contain many members of different types, it can handle only one member at a time.

Like structures, a union can be declared using the keyword union as follows :

```
union item
{
   int m;
   float x;
  char c;
} code;
```

Unions

- **`union`**
  - Memory that contains a variety of objects over time
  - Only contains one data member at a time
  - Members of a **`union`** share space
  - Conserves storage
  - Only the last data member defined can be accessed
- **`union`** declarations
  - Same as struct

```
union Number {
  int x;
  float y;
};
union Number value;
```

Unions

- Valid **`union`** operations
  - Assignment to **`union`** of same type: **`=`**
  - Taking address: **`&`**
  - Accessing union members: **`.`**
  - Accessing members using pointers: **`->`**

```c
/* Fig. 10.5: fig10_05.c
   An example of a union */
#include <stdio.h>

union number {
   int x;
   double y;
};

int main()
{
   union number value;

   value.x = 100;
   printf( "%s\n%s\n%s%d\n%s%f\n\n",
           "Put a value in the integer member",
           "and print both members.",
           "int:   ", value.x,
           "double:\n", value.y );

   value.y = 100.0;
   printf( "%s\n%s\n%s%d\n%s%f\n",
           "Put a value in the floating member",
           "and print both members.",
           "int:   ", value.x,
           "double:\n", value.y );
   return 0;
}
```

1. Define union

1.1 Initialize variables

2. Set variables

3. Print

Program Output

```
Put a value in the integer member
and print both members.
int:   100
double:
-9255959211743313600000000000000000000000000000000000000000000000000000.00000

Put a value in the floating member
and print both members.
int:   0
double:
100.000000
```

LECTURE 32

POINTER

POINTERS

A pointer is a derived data type in C. it is built from one of the fundamental data types available in C.

A pointer variable is a variable that contains an address which is the location of another variable in memory.

Since these memory addresses are the locations in the computer memory where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.

Benefits of using pointers :

1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function arguments.
3. Pointers permit references to functions and thereby facilitating passing of function as arguments to other functions.
4. The use of pointer arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management.
6. Pointers provide an efficient tool for manipulating dynamic data structures such as linked list, queues, stacks and trees.
7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduces the program execution time.

LECTURE 33

Accessing the address of a variable

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately.

Then how can we determine the address of a variable?

This can be done with the help of the operator & (address of) available in C.

The operator & immediately preceding a variable returns the address of the variable associated with it.

For example the statement :

P=&quantity;

would assign the address of quantity to variable p.