1. Code
2. Recursion

Recursion is defining something in the form of itself.

When a function calls itself, it is said to be a recursive function. The most important criterion for a recursive function is that it must have an ending point or base condition so that it can return back to the firstly calling function else stack overflow will occur as it will go into an infinite loop.

Example:

void f1(int x)

{

if(x>0) /\*Base condition\*/

{

f1(x-1); /\*Recursive call\*/

}

printf(“%d”, x);

}

A new activation record is created every time the function is called and is placed at the top of the stack. If its scope is over, it is popped out. So the variables of one function call do not interfere with that of the others.

Stack frame or activation record is a space in memory which has the details of an instance of a function. It will at least have the return address (where to return to- the calling function). Furthermore, it has the local variables and their values (if they exist). All the activation records are stored in a stack which follows the LIFO (Last In First Out) mechanism. The activation record on top of the stack is the current function. Note that it doesn’t have static variables in the AR.

Tracing the program

main

returns 129 return sum(4)\*4+5

returns 31 return sum(3)\*3+4

returns 9 return sum(2)\*2+3

returns 3 return sum(1)\*1+2

returns 1 return sum(0)\*0+1

sum

n=5

n=2

n=4

n=3

sum

sum

sum

sum

n=1

sum

sum

n=0

Output

129

7. Array

An array is a collection of elements of the same data type which are stored in contiguous memory locations. Each element is stored in a separate memory location. Each element has a different index or subscript in sequential order by which they are referenced. An array can have any number of dimensions(compiler dependent).

Eg. int A[20]; /\*1-Dimensional Integer array\*/

int A[10][10]; /\*2\_Dimensional Integer array-Matrix\*/

char s[20];/\*Character array(1-Dimensional) or string\*/

8.

1000 1004 1008 1012 1016 1020 1024 1028 1032

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

a[0][0] a[0][1] a[0][2] a[1][0] a[1][1] a[1][2] a[2][0] a[2][1] a[2][2]

p[0][0] p[0][1] p[0][2]

|  |  |  |
| --- | --- | --- |
| 1000 | 1012 | 1024 |

a[0] a[1] a[2]

\*-dereferences the pointer

output

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 1 | 1 | 1 |
| 2 | 4 | 2 | 4 |
| 3 | 7 | 3 | 7 |
| 4 | 2 | 4 | 2 |
| 5 | 5 | 5 | 5 |
| 6 | 8 | 6 | 8 |
| 7 | 3 | 7 | 3 |
| 8 | 6 | 8 | 6 |
| 9 | 9 | 9 | 9 |

9. Pointers

A pointer is a variable that stores memory address(either address of another variable or a particular address in memory). It stores the starting address of any data and stores only one address. A pointer points to a particular location in memory.

Syntax:

data type \*pointer\_name;

The \* indicates that it is a pointer.

int \*p; /\*Pointer to integer\*/

All pointers are allocated the same size(word size- processor dependant).

When a pointer is not assigned a value, it points to garbage. If it is assigned NULL, it points to nothing.

Pointer uses 2 operators

1. &- referencing or address operator
2. \* -deferencing operator

\*p denotes value pointed to by p

p denotes value of p(address which it is storing)

&p denotes address of the pointer itself

& and \* cancel each other.

Arrays with pointers

The name of an array is actually a constant pointer (value cannot be changed). We refer to all the elements of an array using base address which is pointed to by name of array.

Eg. int A[10];

A is a constant pointer

int \*p=A;/\*Assigns the base address of array to p\*/

p++;/\*Goes to next location according to size of variable (here 4) (pointer addition)\*/

\*p in RHS- value

\*p in LHS- address

Strings with pointers

The name of string is also a constant pointer which points to the first element of the string. It is used to manipulate values of its elements.

Eg. char s[20];/\*s is a char pointer\*/

The problems with arrays and strings is that they have fixed size which leads to wastage of memory. Sometimes we might use too less and sometimes we may require more. So, this is where dynamic memory allocation comes into play. Memory can be allocated to and deallocated from a pointer dynamically from the heap section. It uses functions such as malloc(), calloc(), realloc() and free().

malloc() and calloc() are used to allocate memory

p= (data type\*) malloc(no of bytes);

p= (data type\*) calloc(n, size of element);

free() is used to remove or deallocate the memory

free(p);

realloc() is used to reallocate or increase the amount of memory allocated to the pointer

p=realloc(p, new\_size);

Differentiate

1. Array and pointer

|  |  |
| --- | --- |
| Array | Pointer |
| 1. An array can only hold fixed size of data(static). | 1. A pointer can be allocated memory dynamically as well |
| 1. The name of the array is a constant pointer and it points to the base address of the array. It cannot be modified. | 2. Pointers are variables and they can point to any one memory at a time during the life of the program. It can be modified. They can hold the address of another variable or they can point to a space in memory by dynamic allocation. |
| 1. Eg. int A[10];   An integer array which can store 10 elements. | 3. Eg. int \*p;  An integer pointer which points to one memory address but can point to any no of elements depending on usage.  int a;  p=&a; /\*address of an integer\*/  p=(int \*)malloc(12);/\*points to 12 bytes of data of integer type, i.e., 3 integers\*/ |

1. Array and structure

|  |  |
| --- | --- |
| Array | Structure |
| 1. An array holds elements of the same type in contiguous memory locations. | 1. A structure can hold elements of different data types. The elements of a structure are held in contiguous memory locations. |
| 1. The array is a set of multiple elements of a primitive data type like int, float, char etc. or even structures. | 1. A structure can itself be used as a data type |
| 1. An array of structures is possible. Eg. struct student stu[20]; | 1. A structure can have an array as its element or even a struct variable as its element. Eg.   struct student  {  char name[20];  struct address add; /\*previously defined\*/  } |
| 1. An array can be multi- dimensional.   Eg. int A[20]; | 4. Eg. typedef  {  int d\_no;  }address; |

1. Structure and Union

|  |  |
| --- | --- |
| Structure | Union |
| 1. A structure is allocated memory higher than a union because it allocates memory for each and every element. | 1. A union allocates lesser memory because it allocates the memory that the largest element requires to the whole union and all the elements share it. |
| 1. Any element can be accessed at any time, i.e., all elements are active as each of them have separate memories allocated to them. | 1. At one point of time, only one element is active, i.e., only one element when printed or used gives the correct value. That element is the one which was assigned value recently.   Eg. stu.a=5;  stu.b=10;  Now b is the active element. |
| 1. Eg.   struct stu  {  int a;  int b;  }  Allocated 8 bytes | 1. Eg.   union stu  {  int a;  int b;  }  Allocated 4 bytes |

1. Macro and function

|  |  |
| --- | --- |
| Macro | Function |
| 1. A macro is single lined | 1. A function is multi lined |
| 1. A macro does not have an explicit declaration, definition and body like function. | 2. A function has clear definitions and body enclosed in curly braces |
|  | 3. A function may or may not return a value. |
| 1. A macro replaces the code during preprocessing itself at the particular location making the code of the program longer. | 4. A function is called from another function during runtime. |
| 1. Since the macro is copied to location, execution time is smaller, but macros are preferred only for making small functions. | 5. The execution time for function call is longer as control needs to be shifted to the function and its involves creation of activation record and pushing it into stack and popping it when the function is over and return control to calling function etc. |
| 1. Eg.   #define A 10  #define B(x) x\*x  void main()  {  int a, b;  a=A;  b=B(a);/\*100\*/  }  After preprocessing  void main()  {  int a, b;  a=10;  b=a\*a;/\*100\*/  } | 1. Eg.   void sum(int a, int b)  {  printf(“%d”, a+b);  }  void main()  {  int a=10, b=20;  sum(a, b);  } |

1. String and character array

|  |  |
| --- | --- |
| String | Character array |
| 1. A string is actually a type in other languages but in C it is implemented using a character array | 1. A character array is an array of char type and it can be used like any array. |
| 1. A string is terminated using \0 (null character) | 2. A character array is not terminated by \0 unless given explicitly during initialization. |
| 1. All string functions can be used and will be compatible with it. | 3. String functions cannot always used with character array (depends on \0). |
| 1. The name of string refers to the whole string | 4. The name of an array refers to the first char of array(base address) |
| 1. scanf(“%s”, string); | 5. scanf(“%c”, c[0]);/\*first element\*/  (Or)  scanf(“%c”, c+1);/\*second element\*/ |
| 1. Eg. char string[]=”hello”;   Here \0 is present implicitly at end of string.  Size=6 | 6. Eg. char ar[]={‘h’, ‘e’, ‘l’, ‘l’, ‘o’};  Here it is not terminated by \0 character unless given explicitly  Size=5 |

x----------------------------x-----------------------------x-----------------------x----------------------x