* + Function calling itself is known as recursive function.
  + A function which calls another function is known as nested function.

**Code :**

#include<stdio.h>

int f(int);

int main()

{

int res = f(5):

printf("\nRes=%d\n\n",res);

return 0;

}

int f(int v)

{

if(v==0)

return 0;

v--;

f(v);

printf("\nV value in func: %d\n",v);

return v;

}

**Pointers :**

* + pointers does not belong to a datatype.
  + Pointer cannot be initialized.
  + The size of all the pointers will be same. which is either 4 or 8.
  + Pointers can only be used to get the addresses.
  + Code :

#include<stdio.h>

int main()

{

int a = 10;

float b = 20.2;

void \*ptr = NULL;

int \*ptr = NULL;

printf("\nAddress of a=%u and its value=%d",&a,a);

printf("\nAddress of b=%u and its value=%d",&b,b);

printf("\nAddress of ptr=%u and its value=%d",&ptr,ptr);

printf("\nSize of a=%d",sizeof(a));

printf("\nSize of b=%d",sizeof(b));

printf("\nSize of ptr=%d",sizeof(ptr));

ptr = &a;

ptr1=&a;

printf("\nptr value = %u",ptr);

printf("\nValue pointed by ptr = %d",\*(int \*)ptr);

printf("\nValue pointed by ptr1=%d",ptr);

printf("\n");

return 0;

}

ptr = %d,int\*ptr

* + If the ptr is converted to int directly so we need to explicitly convert it.(dereferencing)

ptr = %d, \*(int\*)ptr

* + No need of type conversion if the pointer is pointed to the same type.
  + ptr = base address
  + pointers can be dynamic.
  + pointers are basically arrays but the only difference is that arrays are static.

**Error : Invalid use of void expression**

**CODE :**

#include<stdio.h>

int main()

{

int a = 10;

int b[3] = {11,12,13};

int \*ptr = NULL;

printf("\nAddress of ptr=%u and stored address in ptr=%d",&ptr,ptr);

ptr = &a;

printf("\nAddress of ptr=%u and stored address in ptr=%d",&ptr,ptr);

printf("\nValue stored at %u = %d",ptr,\*ptr);

printf("nBase address of b=%u\n",&b[0]);

ptr = &b[0];

printf("\nValue stored at %u = %d",ptr,\*ptr);

printf("\nb[0] = %d",ptr[0]);

printf("\n");

return 0;

}

**Two thumb rules of pointers :**

1. &\* = Nullify each other
2. If we want to convert an array to a pointer op[]>>\*op

\*op >> op[]

>> ptr = &b[0];

>>ptr = &\*(b+0);

>>ptr=b+0;

>>ptr = b;

* If a value need to be pushed into pointer then the dereferencing is important. If an address needs to be pushed then there is no need to de reference.

**CODE :**

#include<stdio.h>

#include <stdlib.h>

int main()

{

int a = 10;

int \*ptr = NULL;

ptr = &a;

printf("\nValue stored at ptr = %d",\*ptr);

ptr = (int \*)malloc(4);

printf("\naddress of ptr pointing to = %u",ptr);

\*ptr = 101;

printf("\nValue stored at ptr = %d",\*ptr);

printf("\nValue of a = %d",a);

printf("\n");

return 0;

}

**ERROR :**

**Segmentation fault (core dumped)**

>> This happens when a pointer is given a value to be stored without referencing it to any other variable. As the pointer does not contain any address.

* To use malloc "**#include <stdlib.h>**" should be used.
* perror :Catches the error and gives it.
* ptr++ : Movement of address is incremented
* \*ptr : Value is incremented.

Asignment :

Write a program to find a value if it is present in the list.

use dynamic arrays.

Key should be scanned.

List should be 10 - 20 elements.

#include <stdio.h>

#include <stdlib.h>

int toFindKey(int \*arr,int size,int key);

int main() {

int \*arr, size,key,found;

printf("\nEnter size of array : ");

scanf("%d",&size);

arr = (int \*)malloc(size\*sizeof(int));

printf("\nEnter array elements : ");

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

{

scanf("%d",arr+);

}

printf("\nEnter value to be searched : ");

scanf("%d",&key);

found = toFindKey(arr,size,key);

if(found){

printf("\nKey is found in the array");

}else{

printf("\nKey is not found in the array");

}

return 0;

}

int toFindKey(int \*arr,int size,int key){

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

if(\*(arr+i)==key){

return 1;

}

}

return 0;

}

**Dangling pointer situation :**

* Pointer is pointing to the reference but the address is destroyed.
* **CODE :**

#include<stdio.h>

int \*allocMem();

int main()

{

int a = 10;

int \*ptr = NULL;

ptr = allocMem();

printf("\n%d",\*ptr);

printf("\n");

return 0;

}

int \*allocMem()

{

**int a = 10;**

return &a;

}

**ERROR :**

**Segmentation fault (core dumped)**

Because the ptr lost the reference

* Storages in C are : static, extern, register, auto. These are used to overcome the dangling pointer situation.

#include<stdio.h>

int \*allocMem();

int main()

{

int a = 10;

int \*ptr = NULL;

ptr = allocMem();

printf("\n%d",\*ptr);

printf("\n");

return 0;

}

int \*allocMem()

{

**static int a = 10;**

return &a;

}

**Mutable Arrays :**

* Extern keyword is used so that we tell the code that the reference is in another file and not in the same file or code.
* CODE :

#include <stdio.h>

//#define CAP 5

// int CAP=5;

extern int CAP;

int main()

{

int arr[CAP];

int i;

for(i=0;i<CAP;i++)

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

printf("\nList is\n");

for(i=0;i<CAP;i++)

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

printf("\n\n");

return 0;

}

**Pointer to functions :**

* type (\*qptr)

#include <stdio.h>

int add(int,int);

int sub(int, int);

int Calc(int (\*)(),int,int);

int main()

{

int ret;//=Calc(add,10,20);

int ret1=0;//=Calc(sub,20,10);

int (\*func)();

/\*

func = add;

ret = (\*func)(10,20);

printf("\nRet=%d\tRet1=%d\n\n",ret,ret1);

func = sub;

ret1 = (\*func)(100,20);

printf("\nRet=%d\tRet1=%d\n\n",ret,ret1);

\*/

ret1 = Calc(add,110,20);

ret = Calc(sub,30,5);

printf("\nRet=%d\tRet1=%d\n\n",ret,ret1);

return 0;

}

int add(int v1,int v2)

{

return v1+v2;

}

int sub(int v1,int v2)

{

return v1-v2;

}

int Calc(int (\*f)(),int v1,int v2)

{

return (\*f)(v1,v2);

}

Write a program to greeting like using decorators in c program

CODE :

#include<stdio.h>

int main()

{

printChar('\n',1);

}

int printChar(int ch, int n);

{

int i;