**Day-10(7-11-2024)**

* reusability used in sprint exam -5marks
* Nesting of functions :

Main()

{}

Fun1();

Fun1()

{

Fun2()

}

Fun2{

} => the fun1 calls the fun2

* Recursive function => function calling itself multiple times eg: trees,fibonnaci series
* Nesting is different from recursive functions
* In recursive program may lead to the stack overflow to reduce it we use kill
* Program:

#include<stdio.h>

int fun(int); // declaration

int main()

{

int res=fun(5); // calling

printf("%d",res);

return 0;

}

int fun(int v) // definition

{

if(v==0)

{

return 1;

}

v--;

fun(v);

printf("%d”,v);

return v;

}

}

Output:

V value in fun:0

V value in fun:1

V value in fun:2

V value in fun:3

V value in fun:4

Res=4

* Here is v is destroyed values not the address in recursive functions
* In recursive function the address of the all the functions are same
* In normal function the address of the functions are the destroyed
* **Pointers:**
* Pointers is point to a address not a value
* Declaration :

Datatype \*ptrName;

* Pointer does not belong to the any datatype it special variable points to that datatype
* Int \*ptr

ptr is a Pointer is special variable which points to integer datatype

* 4 types of pointers:

1. Null pointer

* int \*ptr=NULL

1. Void pointer

* Void \*ptr =>generic pointer because it can points to any kind of datatypes

1. Wild pointer

* Float \*ptr it is a pointer without initialization. It is very dangerous
* You should normally initialization the pointer is very important generally we initialize with NULL value.

4)Dangling pointer

* Pointer is pointing to an address where in address reference is destroyed
* Normally every pointer has a same size that is 4 or 8 bytes depending on the bitness of the system.
* Program :

#include<stdio.h>

int fun(int); // declaration

int main()

{

int a=10;

float b =10.2;

void \*ptr=NULL;

printf("address of a:%u and value:%d",&a,a);

printf("address of b:%u and value:%d",&b,b);

printf("address of ptr:%u and value:%d",&ptr,ptr);

printf("size of a:%d",sizeof(a));

printf("size of b:%d",sizeof(b));

printf("size of ptr:%d",sizeof(ptr));

}

* Ptr=&a => ptr have the value of address of a.

ptr=&a;

printf("ptr value :%u",ptr);

* \*ptr => contents of (&a) which is derefencing
* Error : invalid use of void expression
* Generic pointer have difficulty to convert to that particular data type so we explicitly give the type conversion like
* \*(int\*) ptr 🡺 only while using generic pointers

Printf(“value of ptr :%d”, \*(int\*)ptr);

* While defreencing ,there is no need of type conversion when we same type of data type pointer

Int \*ptr=NULL;

Printf(“value of ptr :%d”, \*ptr);

* Float b=10.2;

Int \*ptr1=NULL;

Ptr1=&b;

Printf(“value of ptr1 :%d”, \*(float\*)ptr1); 🡺10.2

Here the the pointer is int but the the ptr1 is pointing to “b” which is float so convert that integer to float in the above line while derefencing (which means \*symbol)

* By using array notation in pointers:

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

ptr=&b[0];

Printf(“address of ptr :%u”, ptr);

Printf(“value of ptr :%d”, ptr[0]); 🡺 11

Printf(“value of ptr :%d”, ptr[1]); 🡺 12

Printf(“value of ptr :%d”, ptr[2]); 🡺 13

Here we are using array format

* By using pointers notation:

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

ptr=&b[0];

Printf(“address of ptr :%u”, ptr);

Printf(“value of ptr :%d”, \*(ptr+0)); 🡺 11

Printf(“value of ptr :%d”, \*(ptr+1)); 🡺 12

Printf(“value of ptr :%d”, (\*ptr+2)); 🡺 13

here we are using derefencing

* **THUMB RULES IN POINTERS:**

1. &\* 🡪 it will Nullify each other
2. Operand[] 🡪 \*operand 🡺 to convert the array to pointer

\*operand 🡪 Operand[] 🡺 to convert the pointer to array

Eg:

Ptr=&\*(b+0);

Ptr=(b+0);

Ptr=b; // these 3 are same operations

here the pointer is same as a array hence proved

eg :

for(int i=0;i<3;i++)

{

Printf(“%d”,\*(ptr+i));

}

b[1]=100;

\*(ptr+1)=100; // for replacing

for(int i=0;i<3;i++)

{

Printf(“%d”,\*(ptr+i));

}

/\*

thumb rules:

1.

&\* = Nullify each other

2.

op[] => \*op

\*op => op[]

\*/

#include <stdio.h>

int main()

{

int a=10;

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

int i;

int \*ptr=NULL;

//ptr = &b[0]; //=>ptr is pointing to the BA of b=BAof 1st element

//ptr = &\*(b+0)

//ptr =(b+0)

ptr = b;

for(i=0;i<3;i++)

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

//b[1] = 100;

\*(ptr+1) = 100;

for(i=0;i<3;i++)

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

printf("\n\n");

return 0;

}

* **Dynamic Pointers:**

Program1:

#include<stdio.h>

int fun(int); // declaration

int main()

{

int a=10;

int \*ptr=NULL;

ptr=&a;

printf("value stored at ptr :%d\n",\*ptr);

\*ptr=101;

printf("value stored at ptr :%d\n",\*ptr);

printf("value stored at a :%d\n",a);

return 0;

}

o/p: ptr=10

ptr=101

a=101

program2:

#include<stdio.h>

int fun(int); // declaration

int main()

{

int a=10;

int \*ptr=NULL;

\*ptr=101;

printf("value stored at ptr :%d\n",\*ptr);

printf("value stored at a :%d\n",a);

return 0;

}

o/p:

it gives segmentation fault or core dumped because we did not give any address and we did not point to any address to store the value(101) in the runtime

* Dynamic allocation🡺 malloc,calloc,realloc 🡺memory allocated in heap memory🡺dynamically enter the memory you want in the runtime
* Malloc:
* We have to #include<stdlib.h>
* These are unnamed address allocation.
* Named address means the memory allocated by the compiler
* Malloc allocates the size bytes and returns the base address to the allocated memory
* If malloc failed it returns NULL value
* free() is used when the malloc is successful
* malloc(nmemb\*size);

nmemb-no of values to allocated

* **man malloc**
* we should allocate the memory before we used it
* void \*malloc(size)

program for malloc:

#include<stdio.h>

int fun(int); // declaration

int main()

{

int a=10;

int \*ptr=NULL;

ptr=&a;

printf("value stored at ptr :%d\n",\*ptr);

ptr=(int\*) malloc(1\*sizeof(int));

printf("address of the ptr:%u\n",&ptr);

\*ptr=101;

printf("value stored at ptr :%d\n",\*ptr);

printf("value stored at a :%d\n",a);

return 0;

}

Program for malloc using arrays:

#include<stdio.h>

int fun(int); // declaration

int main()

{

int a=10;

int \*ptr=NULL;

ptr=&a;

printf("value stored at ptr :%d\n",\*ptr);

ptr=(int\*) malloc(3\*sizeof(int));

printf("address of the ptr:%u\n",&ptr);

ptr[0]=101;

ptr[1]=102;

ptr[2]=103;

printf("value stored at ptr :%d\n",\*ptr);

printf("value stored at a :%d\n",a);

for(int i=0;i<3;i++)

{

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

}

return 0;

}

o/p: 101 102 103

if in this case in malloc is like ptr=(int\*) malloc(1\*sizeof(int)); => it can only give the value to the reserved memory (101) but we donot have guarantee that 102 and 103 will print it will print the garbage values

* program for malloc using pointer:

#include <stdio.h>

#include <stdlib.h>

int main()

{

int a=10;

int i;

int \*ptr = NULL;

ptr = &a;

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

ptr = (int \*)malloc(3\*sizeof(int));

if(ptr == NULL)

{

perror("malloc: ");

exit(0);

}

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

\*(ptr+0) = 101;

\*(ptr+1) = 102;

\*(ptr+2) = 103;

for(i=0;i<3;i++)

printf("\n%d element addres = %u",i,&ptr[i]);

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

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

printf("\nArray Elements are\n");

for(i=0;i<3;i++,ptr++){

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

}

printf("\n\n");

return 0;

}

* if the malloc failed it returns null value then check

ptr=(int\*) malloc(100000000000);

if(ptr==NULL)

{

perror(“malloc:”);

exit(0);

}

o/p : it checks error i.e cannot allocate memory

* if you want to specify only base address and assign the values to all the locations by incrementing the pointer to that address

#include <stdio.h>

#include <stdlib.h>

int main()

{

int a=10;

int i;

int \*ptr = NULL;

ptr = (int \*)malloc(3\*sizeof(int));

if(ptr == NULL)

{

perror("malloc: ");

exit(0);

}

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

for(i=0;i<3;i++)

printf("\n%d element addres = %u",i,&ptr[i]);

\*ptr = 101;

printf("\n%d is stored at %u",\*ptr,ptr);

ptr++;

\*ptr = 102;

printf("\n%d is stored at %u",\*ptr,ptr);

ptr++;

\*ptr = 103;

printf("\n%d is stored at %u",\*ptr,ptr);

ptr--;

ptr--;

printf("\nArray Elements are\n");

for(i=0;i<3;i++,ptr++){

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

}

printf("\n\n");

return 0;

}

* In the just above example we did decrement the ptr value because we did not get the values for that address

Or

we can do by using temporary pointer which is temp ,this temp is fixed to the base address

#include <stdio.h>

#include <stdlib.h>

int main()

{

int a=10;

int i;

int \*ptr = NULL;

int \*temp = NULL;

ptr = (int \*)malloc(3\*sizeof(int));

if(ptr == NULL)

{

perror("malloc: ");

exit(0);

}

temp = ptr;

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

for(i=0;i<3;i++)

printf("\n%d element addres = %u",i,&ptr[i]);

\*ptr = 101;

printf("\n%d is stored at %u",\*ptr,ptr);

ptr++;

\*ptr = 102;

printf("\n%d is stored at %u",\*ptr,ptr);

ptr++;

\*ptr = 103;

printf("\n%d is stored at %u",\*ptr,ptr);

printf("\nArray Elements are\n");

for(i=0, ptr = temp;i<3;i++,ptr++){

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

}

printf("\n\n");

return 0;

}

question )write a program to find a value if an elements in array or list

ans) A screenshot of a computer program

Description automatically generated

A white background with black text

Description automatically generated

* There are 4 storage classes in c:

1. **Static**

* to avoid dangling situations .this static is exist till the end of our process.

Program :

#include <stdio.h>

int \*allocMem();

int main()

{

int a=10;

int \*ptr=NULL;

ptr=allocMem();

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

return 0;

}

int \*allocMem()

{

static int a=10;

return &a;

}

o/p: 10

* Static scope variable exists until the process
* Static is declared only once
* Static scope is global
* Application: for banking
* In dangling it destroys the address and stack also but static is declare that particular address cannot be destroyed

1. **Extern**

* Extern is a mutable array
* Extern keyword is used
* Extern storage class simply tells us that the variable is defined elsewhere and not within the same block where it is used.
* The main purpose of using extern variables is that they can be accessed between two different files which are part of a large program.

1. **Auto**

* This is the default storage class for all the variables declared inside a function or a block.
* Auto variables can be only accessed within the block/function they have been declared and not outside them (which defines their scope). Of course, these can be accessed within nested blocks within the parent block/function in which the auto variable was declared.

1. **Register**

* keyword is register. the time taken by the cpu to access the variable stored in the register is taken less time than the static variable
* register is cost do not use to store all the values, only use special variables
* in software field mainly used extern and dynamic memory
* **Function pointers:**
* This function pointer Can hold the address of an another function
* Declaration: Type(\*fptr)();
* It can only point to one address of the function at a time
* Using function pointers we can give the one function as a input arguments to the another function.
* Program:

A screenshot of a computer program

Description automatically generated

A white background with black text

Description automatically generated