**Pointers**

Datatype \*ptrName;

Pointer does not belong to any kind of datatype. It is just a special variable that holds the address

**Int 🡨 \* ptr** (read from right to left)

The ptr special variable is pointing to integer datatype. Pointers should always be initialized.

The size of the pointers is 4bytes/8bytes. The size of the pointers is same for all data types.

The value of the pointer is always positive so we use %u

4types of Pointer

1. Null pointer
2. Void pointer
3. Wild pointer
4. Dangling pointer (situation)

Void \*ptr 🡺void pointer/generic pointer. It can point to any kind of datatype

Int \*ptr = NULL; 🡺null pointer. It points to nothing

Int \*ptr 🡺wild pointer. As it is not initialized. It can point to any kind of data

ptr 🡺 gives the address of the variable pointing to

\*ptr 🡺gives the value(dereferencing using \*)

&ptr 🡺gives the address of pointer

Error! : Invalid use of void expression

When we use void ptr and it is holding to an int datatype and if we want to print the value, it gives this error

We need to do type casting as the pointer do not know to which bytes of data it should point.

Basically, if we are using int(any)pointer for int(any) datatype no need of type casting

Int a=10;

Void \*ptr=NULL;

Ptr= &a;

Printf (“%d value of a is:”, \*(int \*) ptr); // First \* is for dereferencing int \* is to convert into int

**Arrays and Pointers:** Arrays are static pointers can be dynamic.

int a[3] ={1,2,3};

int \*ptr= NULL;

ptr= &a[0];

printf (“%d”, ptr[0]); // array notation

printf (“%d”, ptr[1]);

printf (“%d”, ptr[2]);

printf (“%d”, \*(ptr+0)); // pointer notation

printf (“%d”, \*(ptr+1));

printf (“%d”, \*(ptr+2));

1. &\* 🡺nullify each other

2. Op[ ] 🡺 \*op

\*op 🡺op[ ]

We can directly give the name of array to pointer

ptr = &b[0] ==== ptr =b;

**Dynamic Pointers**

To point to an unnamed address

Read man malloc

Malloc 🡪allocates size bytes and returns a pointer to allocated memory the base address

If size is 0, the malloc return Null or as unique pointer that can later successfully passed to free

void \*malloc(size\_t,size)

We can do free only when malloc is successful. The static memory cannot be freed by us but dynamic memory has to be free else there will be a memory leak in heap

int \*ptr =NULL;

ptr =(int \*)malloc(4); //generally malloc return void data type, so we need to do type casting

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

\*ptr=1001;

## Array dynamically

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

ptr [0] =10; or \*(ptr+0)=10;

ptr [1] =20; or \*(ptr+1)=20;

ptr [2] = 30; or \*(ptr+2)=30;

ptr [3] = 40; or \*(ptr+3)=40;

perror(“”); 🡪capture the error and prints the error

\*ptr= 101;

Printf (“Address: %u,ptr);

Ptr++;

\*ptr= 102;

Printf (“Address: %u,ptr);

Ptr++;

\*ptr= 103;

Printf (“Address: %u,ptr);

Ptr++;

Ptr--; // to get back to base address we can also use ptr=temp

Ptr--;

For (int i=0; i<4;i++)

{

Printf (“%d”, \*(ptr+i));

Ptr++;

}

ptr++ it points to the next address

we need to store the ptr value in temp

WAP to search a value in list using dynamic arrays

// Online C compiler to run C program online

#include <stdio.h>

#include<stdlib.h>

int search(int \*, int \*,int);

int main() {

int i,key,res;

int \*ptr=NULL;

int \*temp=NULL;

ptr= (int \*)malloc(9\*sizeof(int));

temp= ptr+(7\*sizeof(int));

printf("Address : %u\n",ptr);

ptr++;

\*ptr=11;

ptr++;

\*ptr=17;

ptr++;

\*ptr=18;

ptr++;

\*ptr=19;

ptr++;

\*ptr=20;

res= search(ptr,temp,12);

if(res==1)

printf("Found");

if(res==0)

printf("Not found");

return 0;

}

int search(int \*a,int \*n,int key)

{

for(a;a<=n;a++){

if(\*a==key)

return 1;

}

return 0;

}

**Dangling Pointer Situation:** Pointer is pointing a address wherein the address is destroyed

We can use the keyword static before the local variable

int \*allocMem();

int \*allocMem();

int main() {

// Write C code here

printf("Try programiz.pro");

int \*ptr=NULL;

ptr=allocMem();

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

return 0;

}

int \*allocMem()

{

static int a=10;

return &a;

}

**Static:**

The static storage class the variable is present till the process terminates even the function is done. But the static variable cannot be used in other functions.

Only once the memory is allocated to that variable

**Auto:**

**Register:**

When we use the register key word the variable is stored in CPU memory instead of stack memory and it is easy and fast to access the register variable. The CPU registers are costly, so we use them only for the variables that are used more frequently.

**Extern**

Using the extern keyword, we can use to establish a reference from another file. Say there is a variable in another file and it should be used in this file in global declaration we give extern variable name. This is what happens in Mutable Array. Eg: extern int Cap;.

#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;

}

In Python there are decorators. Decorators are functions that call another functions. The inner functionality is done by function pointers

**\*\*\*Function Pointers**

A function like a variable has an location in memory. So it is possible to declare a pointer

***Type (\*fptr)( );***

The fptr is a ptr to fn which return type value

The parenthesis are necessary for fptr because if not used then it takes as a type \* a pointer

#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=Calc(sub,20,10);

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);

}

WAP as a decorator to print a greeting