**USER DEFINED FUNCTIONS**

Note: Reusability is important, carries 5 marks

* Nesting of function – one fun calls another fun
* Recursive function – it calls by itself

/\*

Demo on recursive function –

\*/

#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 1;

v--;

f(v);

printf("\nV Value of Function: %d\n",v);

return v;

}

Output:

V Value of Function: 0

V Value of Function: 1

V Value of Function: 2

V Value of Function: 3

V Value of Function: 4

Res=4

* It will generate its own stack frame
* In normal function stack frame variables will have local copies but when it is recursive function it will not have different copies(same address)
* Once function is done it is destroying the values not the address – in recursive function through hash maping
* Addresses are also destroyed in normal fun stack frame
* When all the addresses are destroyed then it returns to main function.
* Recursive function have same addrsses for all copies\*\*\*

If this part is changed like this

int f(int v)

{

printf("\nV Value: %d\n",v);

if(v == 0)

return 1;

then

V Value: 5

V Value: 4

V Value: 3

V Value: 2

V Value: 1

V Value: 0

V Value of Function: 0

V Value of Function: 1

V Value of Function: 2

V Value of Function: 3

V Value of Function: 4

Res=4

**POINTERS**

* It is a special variable which holds address (point to address)
* Pointer does not belong to any type of datatype
* Always read from right to left (ptr is a special variable which is pointing to integer type datatype
* Size of all pointers will be same, either 4 or 8 bytes because it is holding only the address which is of course in this range only

Declaration: dt \*ptrName;

Int \*ptr;

4 types:

1. Null Pointer
2. Void Pointer
3. Wild Pointer
4. Dangling Pointer (it is usually a situation not the name of pointer)

There are near and far, generic pointers as well (void \*ptr) – because this can point to any kind of datatype

Int \*ptr = Null; --- Null Pointer

Float \*ptr; --- Wild Pointer (dangerous as it can be pointed to any value)

/\*

Ponter Demo -

\*/

#include <stdio.h>

int main()

{

int a=10;

float b=20.2;

void \*ptr = NULL;

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

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

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

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

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

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

printf("\n\n");

return 0;

}

O/P

Address of a=1766022732 and its value=10

Address of b=1766022728 and its value=20.200001

Address of ptr=1766022720 and its value=0

Size of a=4

Size of b=4

Size of ptr=8

If changed like this

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

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

ptr = &a;

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

printf("\n\n");

return 0;

then

Address of a=1364800476 and its value=10

Address of b=1364800472 and its value=20.200001

Address of ptr=1364800464 and its value=0

Size of a=4

Size of b=4

Size of ptr=8

ptr value = 1364800476

* Dereferencing can be done by using \*ptr , once after doing this we can get the content or the value stored in that address.

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

Then it results in

error: invalid use of void expression

* This is because generic pointers must be explicitly converted, it cannot do on its own

If void \*ptr = NULL;

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

Then, Value pointed by ptr = 10

Note : The above error occurs only when generic pointers are used

Also if we need conversion between similar datatype then no issues

If

int \*ptr1 = NULL; and printf("\nValue pointed by ptr1 = %d", \*ptr1);

then

Value pointed by ptr1 = 10

NO ERROR

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include <stdio.h>

int main()

{

int a=10;

int \*ptr=NULL;

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

ptr = &a;

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

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

}

O/P

Address of ptr=3217712544 and stored address in ptr=0

Address of ptr=3217712544 and stored address in ptr=3217712556

Value stored at 3217712556=10

// Online C compiler to run C program online

/\*

Ponter Demo -

\*/

#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=%u",&ptr,ptr);

ptr = &a;

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

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

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

ptr = &b[0];

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

}

O/P

Address of ptr=2944258504 and stored address in ptr=0

Address of ptr=2944258504 and stored address in ptr=2944258524

Value stored at 2944258524=10

Address of b=2944258512

Value stored at 2944258512=11

**Pointer behaves similar to array** (array is static but pointers are dynamic)

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

ptr = &b[0];

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

//array notation

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

printf("\nb[1]=%d",ptr[1]);

printf("\nb[2]=%d",ptr[2]);

O/P

Value stored at 4180441984=11

b[0]=11

b[1]=12

b[2]=13

//pointer notation

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

printf("\nb[1]=%d",\*(ptr+1));

printf("\nb[2]=%d",\*(ptr+2));

O/P

b[0]=11

b[1]=12

b[2]=13

// Online C compiler to run C program online

/\*

thumb rules:

1) &\* ==> it will nullify each other

operand[] = \*operand

\*operand = []operand

\*/

#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 base address of b=BAof 1

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

//ptr = (b+0);

ptr = b;

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

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

printf("\n\n");

return 0;

}

O/P

11

12

13

\*(ptr+1) = 100;

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

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

O/P

11

12

13

11

100

13

#include <stdio.h>

#include <stdlib.h>

int main()

{

int a = 10;

int \*ptr = NULL;

ptr = &a;

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

\*ptr = 101;

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

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

printf("\n\n");

return 0;

}

O/P

Value stored at ptr = 10

Value stored at ptr = 101

Value of a = 101

#include <stdio.h>

#include <stdlib.h>

int main()

{

int a = 10;

int \*ptr = NULL;

//ptr = &a;

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

\*ptr = 101;

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

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

printf("\n\n");

return 0;

}

O/P

Segmentation fault

From the above two examples, we get to know that if we know the address of the variable then we can change the value of that variable using pointer or else we cannot.

Gives error as ENOMEM

So, dynamically memory can be allocated using malloc, calloc and realloc

* Malloc – generic (it will allocate a memory(heap) and sends base address) If it fails to allocate a memory it returns NULL
* It can free the memory as it is dynamically allocated(when malloc is successful, it cannot be done if it is statically allocated
* If pointer is null it cannot free
* Syntax : malloc(nmemb\*size);
* Ex : for 10 members – malloc(10\*size);

Using unnamed address

// Online C compiler to run C program online

#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); // rather then \*malloc this is used because ptr is of type int

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\n");

return 0;

}

O/P

Address of ptr pointing to = 6132384

Value stored at ptr = 101

Value of a = 10

//Highlighted is the base address

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

O/P

Address of ptr pointing to = 29094560

Value stored at ptr = 101

Value of a = 10

When the standard library functions are giving errors then we need to use perror to check it

if(ptr == NULL)

{

perror(“malloc: “);

Exit (0);

}

O/P

Cannot allocate memory

Pointers in Arithmetic Operations

// Online C compiler to run C program online

#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)); // rather then \*malloc this is used because ptr is of type 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 address = %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("\nValue stored at ptr = %d",\*ptr);

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

\*/

ptr--;

ptr--;

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

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

{

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

ptr++;

}

printf("\n\n");

return 0;

}

O/P

Address of ptr pointing to = 28938912

0 element address = 28938912

1 element address = 28938916

2 element address = 28938920

101 is stored at 28938912

102 is stored at 28938916

103 is stored at 28938920

Array Elements are

101

102

103

Whenever malloc occurs, the address is stored in head and temp. So if ptr = temp is done we can get back the base address

Assignment

Write a program to find a value if it is present in a list(array) – Use dynamic array

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*arr, size, value, i, found = 0;

// Asking the user for the size of the array

printf("Enter the number of elements in the array: ");

scanf("%d", &size);

// Dynamically allocating memory for the array

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

if (arr == NULL) {

printf("Memory allocation failed!\n");

return 1; // Exit if memory allocation fails

}

// Taking input for the array elements

printf("Enter the elements of the array:\n");

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

printf("Element %d: ", i + 1);

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

}

// Asking for the value to search in the array

printf("Enter the value to search for: ");

scanf("%d", &value);

// Searching for the value in the array

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

if (arr[i] == value) {

found = 1; // Value found

break;

}

}

// Displaying the result

if (found) {

printf("Value %d is present at index %d.\n", value, i);

} else {

printf("Value %d is not present in the array.\n", value);

}

// Freeing the dynamically allocated memory

free(arr);

return 0;

}

O/P

/tmp/FaRA3Hj4dt.o

Enter the number of elements in the array: 10

Enter the elements of the array:

Element 1:

2

Element 2: 3

Element 3: 4

Element 4: 55

Element 5: 6

Element 6: 67

Element 7: 8

Element 8: 543

Element 9: 2

Element 10: 4

Enter the value to search for: 87

Value 87 is not present in the array

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* OR

#include <stdio.h>

#include <stdlib.h>

int main() {

int \*arr, s, key, i, found = 0;

printf("Enter the number of elements in the array: ");

scanf("%d", &s);

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

printf("Enter the elements of the array:\n");

for (i = 0; i < s; i++) {

printf("Element %d: ", i + 1);

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

}

printf("Enter the value to search for: ");

scanf("%d", &key);

for (i = 0; i < s; i++) {

if (arr[i] == key) {

found = 1;

break;

}

}

if (found) {

printf("\n Value %d is present at index .\n", key, i);

} else {

printf("\n Value %d is not present in the array.\n", key);

}

return 0;

}

Dangling pointer situation – pointer is pointing to reference (address) whrein the address has been destroyed

// Online C compiler to run C program online

#include <stdio.h>

int \*allocMem();

int main()

{

int a=10;

int \*ptr=NULL;

//ptr=&a;

ptr=allocMem();

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

return 0;

}

int \*allocMem()

{

int a=10;

return &a;

}

O/P

/tmp/kzWextk3A1.c: In function 'allocMem':

/tmp/kzWextk3A1.c:21:12: warning: function returns address of local variable [-Wreturn-local-addr]

21 | return &a;

| ^~

/tmp/kzWextk3A1.o

Segmentation fault

=== Code Exited With Errors ===

4 storage classes in C

* static
* extern
* register
* auto

// Online C compiler to run C program online

#include <stdio.h>

int \*allocMem();

int main()

{

int a=10;

int \*ptr=NULL;

//ptr=&a;

ptr=allocMem();

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

return 0;

}

int \*allocMem()

{

static int a=10;

return &a;

}

O/P

/tmp/SaRsFHNK3m.o

10

=== Code Execution Successful ===

#include <stdio.h>

int fun();

int main()

{

int ret = 0;

ret = fun();

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

printf("\n\n");

return 0;

}

int fun()

{

int a = 10;

a=a+2;

return a;

}

O/P

Ret = 12

int ret = 0;

ret = fun();

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

ret = fun();

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

printf("\n\n");

return 0;

O/P

Ret = 12

Ret = 12

int fun()

{

static int a = 10;

a=a+2;

return a;

}

O/P

Ret = 12

Ret = 14

**Static**

When a variable is declared with static, its scope is until the program exists – it makes it a global variable

**Auto**

By default all variables are auto – rarely we use it in C

**Register**

It is stored in CPU memory, so it is much faster to access from stack memory

**Extern**

The variable might be present in some other file but not in the same line

At link time we are getting that file picked up for compilation.

To compile an extern variable contained fun –  *gcc -o app p10.o config.c*

Function Pointers :

It can hold the address of an another function

Syntax: Type(\*fptr)(); returns the type value