Andhra Pradesh State Skill Development Corporation

Programming in C
Pointers

POINTERS

A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address.

Applications of Pointers:

- It is the only way to express some computations.
- It produces compact and efficient code.
- It provides a very powerful tool.

Note:

Pointers are perhaps the most difficult part of C to understand. C's implementation is slightly different in other languages. Pointers in C are difficult and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

Address in C:

Before you get into the concept of pointers, let's first get familiar with address in C.

If you have a variable var in your program, &var will give you its address in the memory, where & is commonly called the reference operator.

You must have seen this notation while using scanf() function. It was used in the function to store the user inputted value in the address of var.

```
scanf("%d", &var);
```

Example:

```
int main()
{
  int var = 5;
  printf("Value: %d\n", var);
  printf("Address: %u", &var);  //Notice, the ampersand(&) before var.
  return 0;
}
```

Output:

```
Value: 5
Address: 2686778
```

Note: You may obtain different value of address while using this code.
In above source code, value 5 is stored in the memory location 2686778. var is just the name given to that location.

Pointer Variables:

If a variable is going to be a pointer, it must be declared as such. A pointer declaration consists of a base type, an *, and the variable name. The general form for declaring a pointer variable is

```
type *name;
```

where type is the base type of the pointer and may be any valid type. The name of the pointer variable is specified by name.

Reference operator (&) and Dereference operator (*):

As discussed, & is called reference operator. It gives you the address of a variable. Likewise, there is another operator that gets you the value from the address, it is called a dereference operator (*).

Below example clearly demonstrates the use of pointers, reference operator and dereference operator.

Note: The * sign when declaring a pointer is not a dereference operator. It is just a similar notation that creates a pointer.

Use of Pointers:

There are a few important operations, which we will do with the help of pointers very frequently.

a. We define a pointer variable,
b. assign the address of a variable to a pointer and
c. finally, access the value at the address available in the pointer variable.

This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

Example

```
#include <stdio.h>

int main () {

   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */

   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

Output:

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

Example:

```
int main(){
   int* pc;
   int c;
   c=22;
   printf("Address of c:%u\n",&c);
   printf("Value of c:%d\n\n",c);
   pc=&c;
   printf("Address of pointer pc:%u\n",pc);
   printf("Content of pointer pc:%d\n\n",*pc);
   c=11;
   printf("Address of pointer pc:%u\n",pc);
   printf("Content of pointer pc:%d\n\n",*pc);
   *pc=2;
   printf("Address of c:%u\n",&c);
   printf("Value of c:%d\n\n",c);
   return 0;
}
```

Output:

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```

About Pointer:

- Normal variable stores the value whereas a pointer variable stores the address of the variable.
- The content of the C pointer always be a whole number i.e. address.

- Always C pointer is initialized to null, i.e. int *p = null.
- The value of null pointer is 0.
- & symbol is used to get the address of the variable.
- * symbol is used to get the value of the variable that the pointer is pointing to.
- If a pointer in C is assigned to NULL, it means it is pointing to nothing.
- Two pointers can be subtracted to know how many elements are available between these two pointers.
- But, Pointer addition, multiplication, division are not allowed.
- The size of any pointer is 2 byte (for a 16 bit compiler).

Types of Pointers:

- NULL Pointer
- Dangling Pointer
- Generic Pointers
- Wild Pointer
- Complex Pointers
- Near Pointer
- Far Pointer
- Huge Pointers

Null Pointer

- NULL Pointer is a pointer which is pointing to nothing.
- The NULL pointer points the base address of the segment.
- In case, if you don't have an address to be assigned to a pointer then you can simply use NULL
- The pointer which is initialized with the NULL value is considered as a NULL pointer.
- NULL is a macro constant defined in following header files –

stdio.h
alloc.h
mem.h
stddef.h

stdlib.h

Defining NULL Value:

#define NULL 0

Example:

```
#include
int main()
{
   int  *ptr = NULL;
   printf("The value of ptr is %u",ptr);

   return 0;
}
```

Output:

```
The value of ptr is 0
```

Dangling Pointer

- Dangling pointers arise when an object is deleted or de-allocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the de-allocated memory.
- In short, a pointer pointing to a non-existing memory location is called a dangling pointer.

Generic Pointers / Void pointer

When a variable is declared as being a pointer to type void, it is known as a generic pointer. Since you cannot have a variable of type void, the pointer will not point to any data and therefore cannot be dereferenced. It is still a pointer though, to use it you just have to cast it to another kind of pointer first. Hence the term Generic pointer.

This is very useful when you want a pointer to point to data of different types at different times.

Void pointer is a specific pointer type – void * – a pointer that points to some data location in storage, which doesn't have any specific type. Void refers to the type. Basically the type of data that it points to is can be any. If we assign an address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on. Any pointer type is convertible to a void pointer hence it can point to any value.

Why Void Pointers is important

1. Suppose we have to declare integer pointer, character pointer and float pointer then we need to declare 3 pointer variables.

2. Instead of declaring different types of pointer variable it is feasible to declare single pointer variable which can act as an integer pointer, character pointer.

Declaration of Void Pointer

```
void * pointer_name;
```

Wild Pointer

A Pointer in C that has not been initialized till its first use is known as the Wild pointer. A wild pointer points to some random memory location.

Example of Wild Pointer

```
int main()
{
  int  *ptr;
  /* Ptr is a wild pointer, as it is not initialized Yet */
  printf("%d", *ptr);
}
```

How can we avoid Wild Pointers?

We can initialize a pointer at the point of declaration wither by the address of some object/variable or by NULL;

Near Pointer

- The pointer which can point to only 64KB data segment or segment number 8 is known as a near pointer.
- That is near pointers cannot access beyond the data segment like graphics video memory, text video memory, etc. Size of the near pointer is two byte. With the help of a keyword near, we can make any pointer as a near pointer.

Example

```
#include <stdio.h>
int main()
{
        int x = 25;
        int near *ptr;
        ptr  = &x;
        printf("%d", sizeof ptr);
        return 0;
}
```

Output: 2

Far Pointer

- The pointer which can point or access whole the residence memory of RAM, i.e., which can access all 16 segments is known as far pointer.
- Size of the far pointer is 4 byte or 32 bit.

Example

```
#include <stdio.h>
int main()
{
        int x=10;
        int far *ptr;
        ptr=&x;
        printf("%d",sizeof ptr);
        return 0;
}
```

Output: 4

Huge Pointer:

- The pointer which can point or access whole the residence memory of RAM i.e. which can access all 16 segments is known as **a huge pointer**.
- Size of the far pointer is 4 byte or 32 bit.

Example of Huge Pointer

```
#include<stdio.h>
int main()
{
char huge * far *p;
printf("%d %d %d",sizeof(p),sizeof(*p),sizeof(**p));
return 0;
}
```

Output: 4 4 1

Pointer with Array:

Arrays are closely related to pointers in C programming but the important difference between them is that a pointer variable takes different addresses as value whereas, in case of an array it is fixed.

Example:

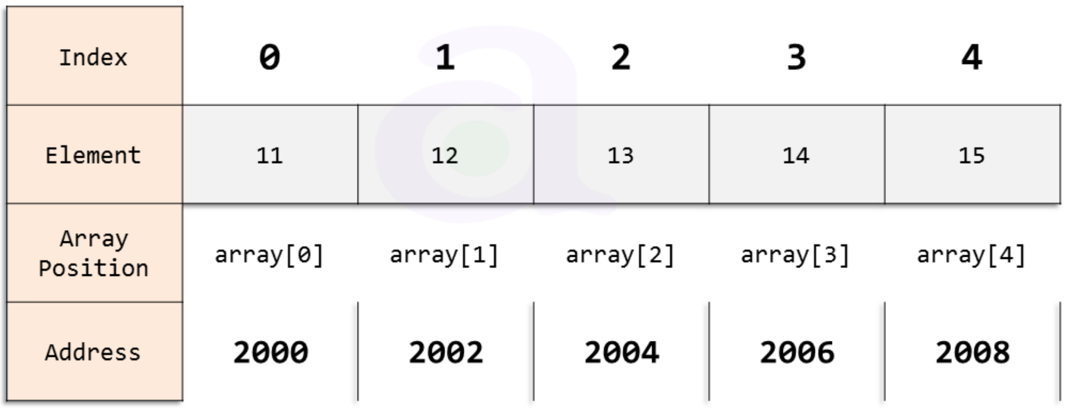
```
#include <stdio.h>
int main()
{
  char charArray[4];
  int i;

  for(i = 0; i < 4; ++i)
  {
    printf("Address of charArray[%d] = %u\n", i, &charArray[i]);
  }

  return 0;
}
```

Output:

```
Address of charArr[0] = 37814048
Address of charArr[1] = 37814049
Address of charArr[2] = 37814050
Address of charArr[3] = 37814051
```

Declaration of array :

int array[5]={ 11, 12, 13, 14, 15 };



Index	0	1	2	3	4
Element	11	12	13	14	15
Array Position	array[0]	array[1]	array[2]	array[3]	array[4]
Address	2000	2002	2004	2006	2008

Here variable arrayName will give the base address, which is a constant pointer pointing to the element, arrayName[0]. Therefore arrayName is containing the address of arrayName[0] i.e 1000. In short, arrayName has two purpose - it is the name of an array and it acts as a pointer pointing towards the first element in the array.

a. arrayName *is equal to* &arrayName[0] //by default

We can declare a pointer of type int to point to the array arrayName.

```
int *p;
p = arrayName;
or p = &arrayName[0];
```

String as Pointer:

In the previous tutorial, we learn that the abstract idea of a string is implemented with just an array of characters. For example, here is a string:

```
char arrayName[] = "Single";
```

What this array looks like in memory is the following:

S	i	n	g	l	e	\0

where the beginning of the array is at some location in computer memory, for example, location 1000.

Note: Don't forget that one character is needed to store the nul character (\0), which indicates the end of the string.

A character array can have more characters than the abstract string held in it, as below:
char arrayName[10] = "Single";

giving an array that looks like:

S	i	n	g	l	e	\0			

(where 3 array elements are currently unused).
Since these strings are really just arrays, we can access each character in the array using subscript notation, as in:

```
printf("Third char is: %c\n", arrayName[2]);
```

which prints out the third character, n.
A disadvantage of creating strings using the character array syntax is that you must say ahead of time how many characters the array may hold. For example, in the following array definitions, we state the number of characters (either implicitly or explicitly) to be allocated for the array.

```
char arrayName[] = "Single";  /* 7 characters */
char arrayName[10] = "Single";
```

Thus, you must specify the maximum number of characters you will ever need to store in an array. This type of array allocation, where the size of the array is determined at compile-time, is called static allocation.

Strings as pointers

Another way of accessing a contiguous chunk of memory, instead of with an array, is with a pointer.
Since we are talking about strings, which are made up of characters, we'll be using pointers to characters, or rather, char *'s.

However, pointers only hold an address, they cannot hold all the characters in a character array. This means that when we use a char * to keep track of a string, the character array containing the string must already exist (having been either statically- or dynamically-allocated).

Below is how you might use a character pointer to keep track of a string.

```
char label[] = "atnyla";
char arrayName1[10] = "welcome";
char *Ptr;
```

```
Ptr = arrayName;
```

We would have something like the following in memory (e.g., supposing that the array arrayName started at memory address 2000, etc.):

arrayName - starting memory address @2000

a	t	n	y	l	a	\0

arrayName1 - starting memory address @3000

w	e	l	c	o	m	e	\0		

Ptr - memory address @4000

```
--------
| 2000 |
--------
```

Example:

```
int main()
{
   char strng[100];
   char *ptr;

   printf("Enter a string: ");
   gets(strng);

   //assign address of str to ptr
   ptr=strng;

   printf("Entered string is: ");
   while(*ptr!='\0')
      printf("%c",*ptr++);

   return 0;
}
```

```
Enter a string: Andra Pradesh
Entered string is: Andhra Pradesh
```

Pointers to Function:

A function has a physical location in memory that can be assigned to a pointer. This address is the entry point of the function and it is the address used when the function is called. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions.

Pointer as a function parameter list is used to hold the address of argument passed during the function call. This is also known as call by reference. When a function is called by reference any change made to the reference variable will affect the original variable.

Example:

```
void swap(int *a, int *b); // function prototype

int main()
{
   int p=10, q=20;
   printf("Before Swapping:\n\n");
   printf("p = %d\n",p);
   printf("q = %d\n\n",q);

   swap(&p,&q); //passing address of p and q to the swap function
   printf("After Swapping:\n\n");
   printf("p = %d\n",p);
   printf("q = %d\n",q);
   return 0;
}

//pointer a and b holds and points to the address of p and q
void swap(int *a, int *b)
{
   int temp;
   temp = *a;
   *a = *b;
   *b = temp;
}
```

Output:

```
Before Swapping:

p = 10
q = 20

After Swapping:
```

```
p = 20
q = 10
```

Another example:

```
#include<stdio.h>

int add(int x, int y)
{
 return x+y;
}

int main( )
{
 int (*functionPtr)(int, int);
 int s;
 functionPtr = add;   //
 s = functionPtr(20, 45);
 printf("Sum is %d",s);
 getch();
 return 0;
}
```

Output:

```
Sum is 65
```

Function returning Pointer:

A function can also return a pointer to the calling function. In this case you must be careful, because local variables of function don't live outside the function. They have scope only till inside the function. Hence if you return a pointer connected to a local variable, that pointer will point to nothing when function ends.

Example

```
#include <stdio.h>
int* checklarger(int*, int*);
void main()
{
 int num1 ;
 int num2;
 int *ptr;

 printf("Enter Two number: \n");
 scanf("%d %d",&num1,&num2);

 ptr = checklarger(&num1, &num2);
 printf("%d is larger \n",*ptr);
}
```

```
int* checklarger(int *m, int *n)
{
        if(*m > *n)
        return m;
        else
        return n;
}
```

Output:

```
Enter Two number:
546
1213
1213 is larger
```

Address of the Function:

We can fetch the address of an array by the array name, without indexes, Similarly We can fetch the address of a function by using the function's name without any parentheses or arguments. To see how this is done, read the following program, which compares two strings entered by the user. Pay close attention to the declarations of checkString( ) and the function pointer p, inside main( ).

Example:

```
#include
#include
void checkString(char *a, char *b,
int (*cmp)(const char *, const char *));

int main(void)
{
        char strng1[80], strng2[80];
        int (*ptr)(const char *, const char *); /* function pointer */
        ptr = strcmp; /* assign address of strcmp to ptr */
        printf("Enter two strings.\n");
        gets(strng1);
        gets(strng2);
        checkString(strng1,strng2,ptr); /* pass address of strcmp via ptr */
        return 0;
}

void checkString(char *m, char *n,
int (*cmp) (const char *, const char *))
{
        printf("Testing for equality.\n");

        if(!(*cmp)(m, n)){ printf("Equal \n");
```

```
            }
            else{
             printf("Not Equal \n");
            }
}
```

Output :

```
Enter two strings.
atnyla
atnyla
Testing for equality.
Equal
```

Output 1:

```
Enter two strings.
atnyla
atnlla
Testing for equality.
Not Equal
```