#15 Application of Pointers

By Saurabh Shukla | 2017©mysirg.com

Application of Pointers

Pointers are very useful concept as it does miraculous things that we can never imagine without pointers. Pointers are memory references and capable to make changes in the memory location.

Recall the concept of functions where one function cannot access variables of the calling function. Though, we can pass values of the variables of 'calling function' to the 'called function' and called function now can use values received in formal arguments but cannot make any change in actual arguments.

If we pass address of the variables of calling functions in place of values of the variables, then called function can receive variable addresses in pointers and can access actual arguments via pointers.

Another concept was, we cannot return more than one value from a function.

When we have addresses of variables of calling function, we can assign any value (or result of some calculation) to actual arguments via pointers, which resolve the need of returning multiple values.

Call by Reference

When a function is called by passing addresses of variables, it is known as call by reference. (Note that, in C++ call by reference is something else)

Function can only access its own memory and cannot access variables of other functions but if we pass address of variables during function call, we actually give power to the 'called function' to access variables of 'calling function' via addresses of variables

Try to find issue in the following program:

```
1   void swap();
2   int main()
3   {
4       int a=10,b=20;
5       swap();
6       printf("a=%d and b=%d",a,b);
7       getch();
8   }
9   void swap()
10  {
11      int t;
12      t=a;
13      a=b;
14      b=t;
15  }
```

In swap function, variable t is declared but a and b are not declared. If you are thinking they belong to the main function, then you are wrong. Variables declared in the body of a function are called local variables. They are accessible only within the function body. Variables a and b are declared in main function. They cannot be accessed from outside the function body.

Now observe another program:

```
1   void swap(int,int);
2   int main()
3   {
4       int a=10,b=20;
5       swap(a,b);
6       printf("a=%d and b=%d",a,b);
7       getch();
8   }
9   void swap(int x,int y)
10  {
11      int t;
12      t=x;
13      x=y;
14      y=t;
15  }
```

यदि आप यह सोच रहे हैं की swap function में a और b की value पास करने से काम बन जायेगा तो आप गलत हैं| a और b main function के variables हैं | x और y swap function के variables हैं | a की value x में कॉपी हो गयी है एवं b की value y में कॉपी

हो गयी है | x और y में data इंटरचेंज करने से बदलाव सिर्फ x और y में होंगे | a और b में कोई changes नहीं होंगे |

The question is how to swap values of a and b variables from swap function. The answer is by passing addresses of variable a and b in swap function. We know that pointer variables are required to store addresses. Thus formal arguments must be pointer variables.

We know that the pointer variable can access variable whose address it contains.

The correct program is

```
1   void swap(int*,int*);
2   int main()
3   {
4       int a=10,b=20;
5       swap(&a,&b);
6       printf("a=%d and b=%d",a,b);
7       getch();
8   }
9   void swap(int *x,int *y)
10  {
11      int t;
12      t=*x;
13      *x=*y;
14      *y=t;
15  }
```

In the above program pointers x and y are pointing to the variables a and b. Writing *x is as good as writing a. Similarly, writing *y is as good as writing b. With the help of pointers we can make changes in the variables of main function.

swap(&a,&b); is known as function call by passing address of simply call by reference. Reference means address.

Notice the declaration of the function swap. Since the formal arguments are pointers, you must specify by using asterisk (*) symbol in argument types in declaring function.

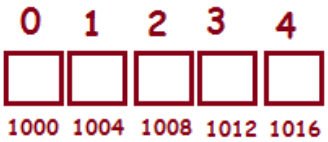
void swap(int*, int*);

Pointers and Arrays

Remember two things about pointers and arrays:

1) Whatever may the size of an array it always gets stored in contiguous memory locations.
2) When pointer is incremented it always points to immediately next location of its own type

```
main()
{
 int a[5];
 ...
}
```

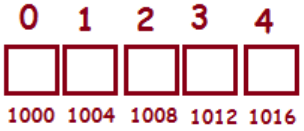


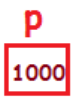
a means 1000

In the above example, we have declared an array with name a of size 5 of type int. All five variables in the array occupy memory in a sequence. Assume the address of first variable in the array as 1000. Subsequent addresses should be 1004, 1008, 1012 and 1016 as each int block consumes 4 bytes in memory.

Anywhere in the program writing a means 1000. The name of the array is representation of address of the first block of array.

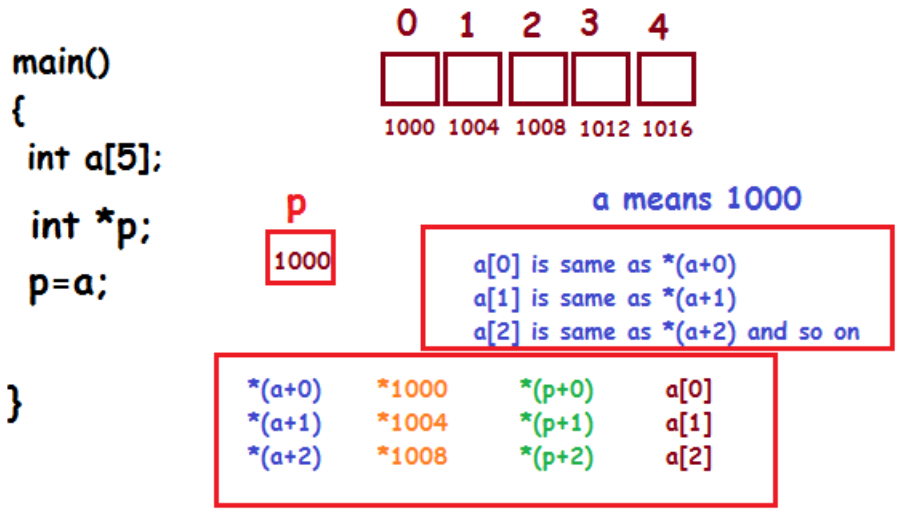
```
main()
{
 int a[5];
 int *p;
 p=a;

}
```





a means 1000

Variable p is a pointer variable of type int, which means p can contain address of some int block. In the above code, we have assigned a in p, that is 1000 is assigned in p. Now p is a pointer which points to the very first int block of the array.



Internally, a[0] is converted into *(a+0). [ ] is an operator called subscript operator. It requires two operands. In a[0], a is first operand and 0 is second operand. Subscript operator add both the operands first (1000+0 is equal to 1000). Then apply dereferencing operator on the result of addition (*1000). Final result becomes the representation of variable whose address is 1000.

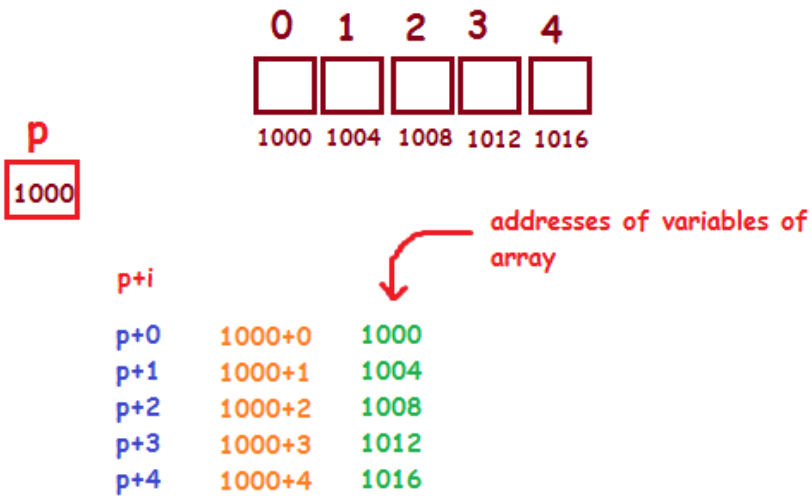
Pointer variable contains 1000 as an address, so the result of p+0 is same as the result of a+0. Thus you can access any array with the pointer which points to the first block of the array.

Following program illustrate the usage of pointer to access an array.

```c
#include<stdio.h>
#include<conio.h>
void input(int *);
void display(int *);
int main()
{
    int a[5];
    input(&a[0]);
    display(&a[0]);
    getch();
    return(0);
}
void input(int *p)
{
    int i;
    printf("Enter five numbers");
    for(i=0; i<=4; i++)
       scanf("%d", p+i );
}
void display(int *p)
{
    int i;
    printf("Five numbers are: ");

    for(i=0; i<=4; i++)
       printf("%d ", *(p+i));
}
```

In the above program, we have created an array in main function of size 5. To fill this array with user data, we call a function input(). Since function input cannot use variables declared in main function, we pass address of the very first block of the array. Input function received this address in pointer p.

Writing p+i in scanf function in definition of input function is same as writing addresses of array variables one by one.



scanf function needs address of the variable where input data from keyboard has to store. We have to pass addresses of array variables one by one. p+i represents address of array variables.

In display function, we need to pass values of array variables in printf, so pass *(p+i) which is as good as writing a[i].

Handling two dimensional array via pointers

There are two ways to create pointers that will be used to store address of two dimensional arrays. First is simply create a pointer and second is pointer to an array

Example of handling two dimensional arrays using int pointer:

```
1  main()
2  {
3     int i;
4     int a[2][3]={3,5,1,6,7,8};
5     int *p;
6     p=&a[0][0];
7     for(i=0;i<=6;i++)
8       printf("%d",*(p+i));
9  }
```

In the above example, a pointer of type int is created to point very first int block of two dimensional arrays. Irrespective of the dimension size of 2d arrays, pointer p is added with value of the variable i, as a result it points to the next element of the array in iteration.

Array elements are always stored in contiguous fashion, irrespective of size of the dimensions. Pointer p is an int pointer that is it points to an int block. Adding 1 to the address stored in p results the address of the immediately next block.

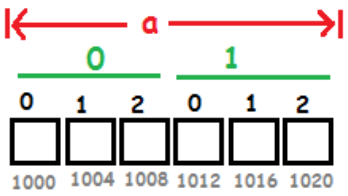
Second way of handling two dimensional arrays using pointers is illustrated with the following example:

```
1   main()
2   {
3       int i,j;
4       int a[2][3]={3,5,1,6,7,8};
5       int (*p)[3];
6       p=a;
7       for(i=0;i<=1;i++)
8       {
9             for(j=0;j<=2;j++)
10            printf("%d ",*( *(p+i)+j));
11            printf("\n");
12      }
13  }
14
```

Observe the way we created pointer variable.

int (*p)[3];

p is a pointer to an array of size 3 of type int. this pointer is different from int pointer (in previous example). Int pointer points to the block of type int, but here p is a pointer to an array of size 3 of type int. when 1 is added to variable p, the resulting address is the address of next array.



p+0	1000	address of first 1d array
p+1	1012	address of second 1d array
*(p+0)	1000	address of first int block of first 1d array
*(p+1)	1012	address of first int block of second 1d array
*(p+0)+1	1004	address of second int block of first 1d array

Finally, we can conclude that writing *(*(p+i)+j) is same as a[i][j]

Pointers and Strings

Just like an int array, char array can also be managed with pointers. We can store address of first char block of a char array in char pointer. Now this pointer can be used for any string manipulation task:

```
1  int main()
2  {
3      char str[ ]="Welcome";
4      char *p;
5      p=str;
6      printf("%s",p);
7      return(0);
8  }
9
```

In the above example, p is a char pointer. Remember that the name of the array represents address of the first block of array.

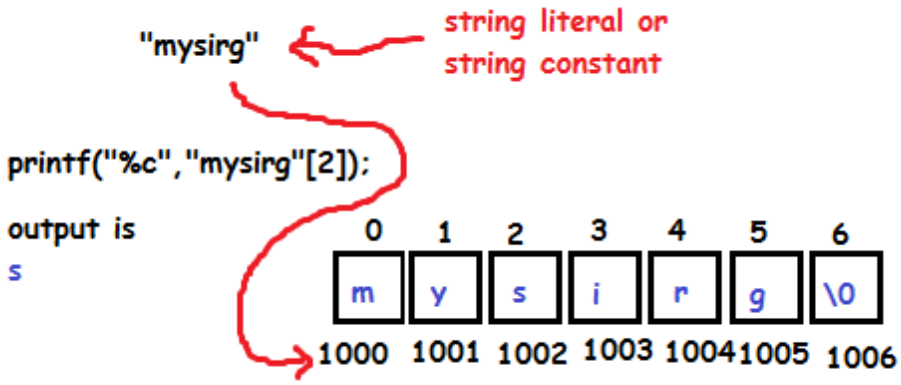
p=str;

p is assigned with the address of the first char block of array.

In printf("%s",p), value of p (address of first char block of array) is passed as an argument. This is required to print a string with format specifier %s.

Variable p and array name str both can be used to represent address of first block of array, but there is a difference between them, p is a variable and str is not. You can change value of p as it is a variable but str is like a constant, it always represents address of first block of array.

String literal or string constant always represents base address of string.



Name of the array is also a representative of base address and string constant too. In the above example, in printf, %c format specifier is used. String constant ("mysirg") is as good as 1000 (base address) and 2 is index number. Therefore it represents value stored at index 2, which is s.

Example: Define a function to calculate length of string

```
int stringlength(char *p)
{
   int i=0;
   while(*(p+i)!='\0')
     i++;
   return(i);
}
```

Array of pointers

Array of pointers is a group of pointers. We create many pointers to point similar lists; we can store addresses of these lists in an array of pointers.

Example

```
int main()
{
   int a=3,b=4,c=5,i;
   int *p[3];
   p[0]=&a;
   p[1]=&b;
   p[2]=&c;
   for(i=0;i<=2;i++)
     printf("%d ",*p[i]);
   return(0);
 }
```

Output:

3 4 5

In the above example we have three pointers, p[0], p[1] and p[2]. These pointers are used to contain addresses of a, b and c. Thus pointer array means sequence of pointers

Dangling Pointers

A pointer becomes dangling when memory of the block is released whose address is still reside in pointer. It is much like wild pointer. The only difference in wild pointer and dangling pointer is, wild pointers are un-initialized pointers thus containing invalid address. Wild pointer points to unknown location which can be dangerous from programming point of view.

On the other hand dangling pointers are initialized perfectly but due to programmers carelessness it is still pointing to the memory area which was released.

```
int main()
{
   int *p;
   {
     int x=4;
     p=&x;
    }
    *p=5;
   return(0);
}
```

See the scope of variable x is limited to the inner block. Pointer p is containing address of variable x. As soon as control comes out from inner block, memory of variable x is released, but the pointer p still contains address of x. Here p becomes dangling

const pointer and pointer to const

Const pointer is a pointer whose value cannot be altered.

Pointer to a const is a pointer that stores an address of const variable thus we cannot modify variable data using pointer.

Declaration of pointer to a const:

```
const int  x = 4;
const int *p; //or int const *p;
p=&x;
```

The above code tells that the pointer p is a pointer to a const. This means we cannot modify variable x using pointer p.

For example:

```
*p=6; //error: cannot modify const variable
 p++; //valid:
```

Another example:

```
int x=4;
const int *p; //or int const *p;
p=&x;

x++; //valid
*p=6;//error: cannot modify const variable
p++; //valid
```

Another example

```
const int x=4;
int *p;
p=&x;
x++; //error: cannot modify const variable
*p=3; //undefined behavior, should be an error
p++; //valid
```

Example for const pointer

```
int x=5;
int *const p=&x;
p++; //error: cannot modify const variable p
x++; //valid
*p=7; //valid
```

Another example:

```
int x=5;
int *const p;
p=&x; //error cannot modify const variable p.
```

Remember that const variables must be initialized during declaration.

References:

YouTube video links

- Lecture 15 Application of pointers in C part-1
  - https://youtu.be/bcAn3s-GltY?list=PL7ersPsTyYt2Q-SqZxTA1D-melSfqBRMW

- Lecture 15 Application of pointers in C part-2
  - https://youtu.be/VwHO8jHiygQ?list=PL7ersPsTyYt2Q-SqZxTA1D-melSfqBRMW
- Lecture 15 Application of pointers in C part-3
  - https://youtu.be/yGhdI6kQXDc?list=PL7ersPsTyYt2Q-SqZxTA1D-melSfqBRMW
- Lecture 15 Application of pointers in C part-4
  - https://youtu.be/cRKINlLFrGY?list=PL7ersPsTyYt2Q-SqZxTA1D-melSfqBRMW
- Lecture 15 Application of pointers in C part-5
  - https://youtu.be/OxlYhsVBvHM?list=PL7ersPsTyYt2Q-SqZxTA1D-melSfqBRMW

Exercise

1) Write a function to calculate area and circumference of a circle. Function is of takes something returns nothing nature. The result of area and circumference should be printed in calling function.
2) Write functions using call by reference concept with given prototype
   a. void sort(int *p, int size); // p is a pointer to first int block of an array and size is length of the array.
   b. void rotate(int *p,int size, int d); //p is a pointer to first int block of an array and size is the length of array. You have to shift elements d times in clock wise direction.
3) Write a recursive function to print all permutations of a given string.
4) Write functions with given prototypes:
   a. int isAlphaNumeric(char*p); //return 1 if the string is alphanumeric otherwise return 0
   b. char* reverse(char*p); //return address of the reverse of the string pointed to by p
   c. int isPalindrome(char *p); // return 1 if string is palindrome otherwise return 0
   d. int compareCaseIgnore(char *p, char*q); //returns 1 if string pointed by p comes later in dictionary than string pointed to by q, otherwise -1. If strings are same returns 0. Comparison should be case insensitive.
   e. int countWords(char *p); //return number of words in a string pointed to by pointer p
   f. char* capitalize(char*p); //make first letter of each word of the string pointed to by pointer p capital and return the address of the string.