Pointers

Chapter 9

Recall from function

Call by value & Call by reference (Or Pass by value & Pass by reference)

Passing arguments to functions:

An argument is a data passed from a program to the function. In function, we can pass a variable by two ways.

1. Pass by value (or call by value)

2. Pass by reference (or call by reference)

Functions call by value

In this the value of actual parameter is passed to formal parameter when we call the function. But actual parameters are not changed. For eg:

```
#include<stdio.h>

void swap(int, int) ; /*function prototype*/

void main( )      {

int x, y ;

clrscr( ) ;

x = 10 ;

swap (x,y) ; /*function call by value */

printf (“x = %d \n”, x) ;

printf (“y=%d\n”, y)

}
```

```
void swap (int a, int b) /*function
definition */
{
        int t ;
        t = a ;
        a = b ;
        b = t ;
}
```

Output x = 10
y = 20

Function call by reference:

In this type of function call, the address of variable or argument is passed to the

function as argument instead of actual value of variable. For this, pointer is necessary, we will study in this chapter.

Pointer

A pointer is a variable that stores a memory address of a variable. Pointer can have any name that is legal for other variable and it is declared in the same fashion like other variables but always preceded /xy '*' (asterik) operator. Thus pointer variables are defined as

int *a ;

float *b ;

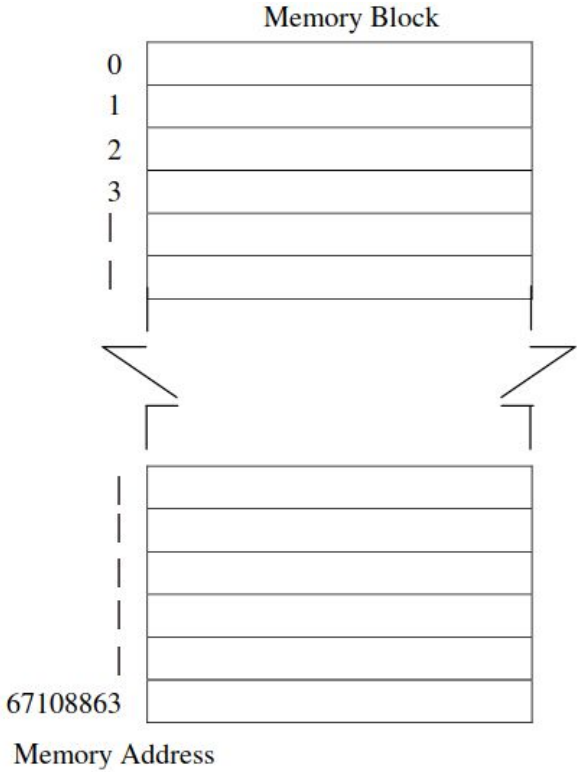
char *c ;

where a, b, c are pointer variable which stores address of integer, float and char variable.

Thus the following assignments are valid.

```
int x ;

float y ;

char c ;

a = &x ; /* the address of x is assigned to pointer variable a */

b = &y ; /* the address of y is stored to pointer variable b */

c = &c ; /* the address of c is stored to pointer variable c */
```

Before studying pointers, it is important to understand how memory is organized in a computer. The memory in a computer is made up of bytes arranged in a sequential manner. Each byte has an index number which is called address of that byte. The address of these bytes start from zero and the address of last byte is one less than the size of memory. Suppose we have 64MB of RAM (Random Access Memory), then memory will consist of $64 \times 2^{20} = 67108864$. the address of these bytes will be from 0 to 67108863.



We have studied that it is necessary to declare a variable before using it, since compiler has to reserve space for it. The data type of the variable also has to be mentioned so that the compiler knows how much space need to be reserved. For example:

int age;

The compiler reserves 2 consecutive bytes from memory for this variable and associates the name age with it. The address of first byte from the two allocated bytes is k\a the address of variable age.

Suppose compiler has reserved bytes numbered 65524 and 65525 for the storage of variable age, then the address of variable age will e 65524. Let us assign some value to this variable.

age = 20;

Now this value will be stored in these 2 bytes in form of binary representation. The number of bytes allocated will depend on the data type of variable. For example, 4 bytes would have been allocated for a float variable, and the address of first byte would be called the address of variable.

Address operator (&):

C provides an address operator ‘&’, which returns the address of a variable when placed before it. This operator can be read as “the address of”, so ‘**&sn**’ means address of sn, similarly ‘**&price**’ means address of price. The following program prints the address of variables using address operator

/* Program to print address of variable using & */

```
#include<stdio.h>

main( )

{

int sn = 30;

float price = 150.50;

printf(“value of sn=%d, Address of sn=%u\n”, sn, &sn);

printf(“value of price=%f, Address of price=%u\n”, price, &price);

}
```

output:

value of sn = 30, Address of sn = 65524

value of price = 150.500000 Address of price = 65520

A pointer is a variable that contains a memory address of data or another variable. [In other word, a pointer is a variable that stores memory address]. Like all other variables it also has a name to be declared and occupies some space in memory. It is called pointer because it points to a particular location in memory by storing the address of that location.

Declaration and assigning (initializing) of pointer:

Like any other variable, pointer variable should also be declared before being used. The general syntax is

data_type *pname;

where pname is the name of pointer variable, which should be a valid C identifier. The asterisk '*' preceding this name informs the compiler that the variable is declared as apointer. data_type is basic data type. For example:

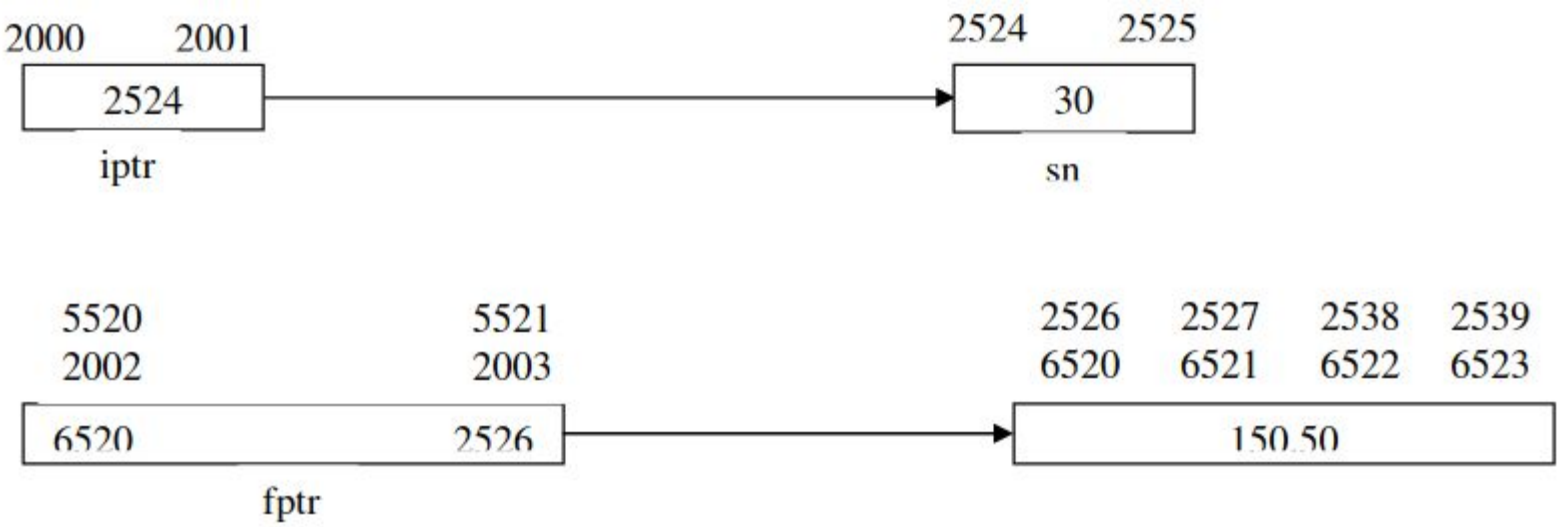
int *iptr,sn=30;

float *fptr,price=150.50;

iptr=&sn; *assigning of pointer*\

fptr=&price;

Now, iptr contains the address of variable sn i.e. it points to variable sn, similarly fptr points to variable sn, similarly fptr points variable price.



Pointer are also variables so compiler will reserve space for them and they will also have some address. All pointers irrespective of their base type will occupy same space in memory since all of them contain address only. Generally 2 bytes are used to store an address (may vary in different computers), so the compiler allocates 2 bytes for a pointer variable.

If pointers are declare after the variable like

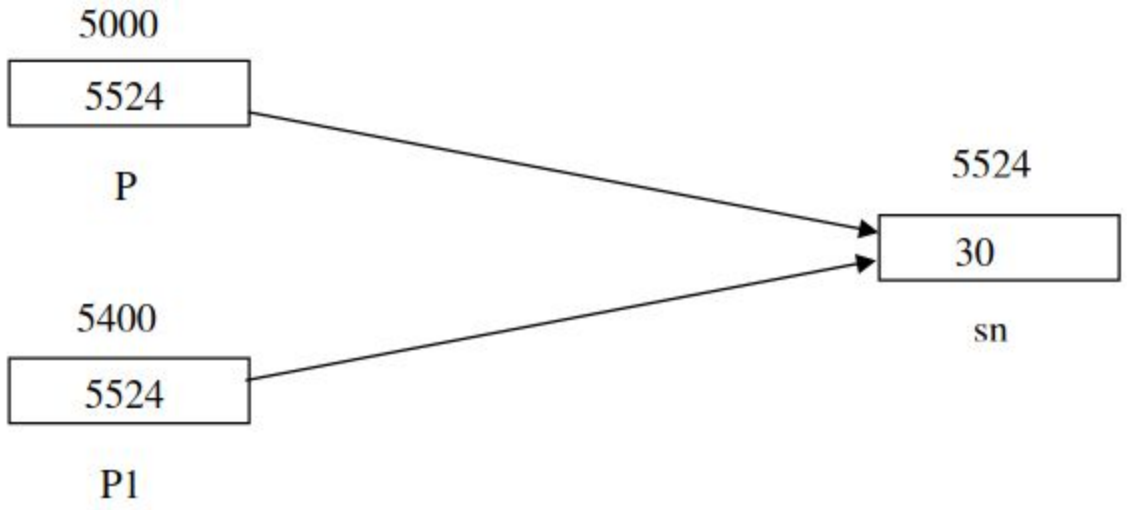
int sn=30,*p=sn;

float price=150.50,*q=&price;

It is also possible to assign the value of one poin ter variable to the other provided their base type is some.

P1=P;

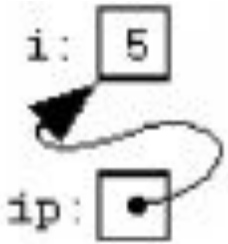
Now, both pointer variable P and P1 contains the address of variable sn and points the same variable



Pointers (that is, pointer values) are generated with the ``address-of'' operator&, which we can also think of as the``pointer-to'' operator. We demonstrate this by declaring (and initializing) anintvariablei, and then settingiptopoint to it:

int i = 5; ip = &i;

The assignment expression **ip = &i**; contains both parts of the “two-step process'': **&i** generates a pointer to **i**, andthe assignment operator assigns the new pointer to (that is, places it ``in'') the variable **ip**. Now **ip**”points to''**i**,which we can illustrate with this picture



Application of pointer:

Some uses of pointers are

1) Accessing array elements

2) Returning more than one value from a function

3) Accessing dynamically allocated memory

4) Implementing data structure like linked lists, trees, and graphs.

5) Increasing the execution speed as they refer address.

Indirection or Deference Operator:

The operator '*', used in front of a variable, is called pointer or indirection or deference operator. Normal variable provides direct access to their own values where as a pointer provides indirect access to the values of the variable whose address it stores. The indirection operator (*) is used in two distinct ways with pointers, declaration and deference. When the pointer is declared, the star indicates that it is a pointer, not a normal variable. When the pointer is deferenced, the indirection operator indicates the value at that memory location stored in the pointer. Let us take an example:

int a = 87;

float b = 4.5;

int*P1 = &a;

float*p2 = &b;

In above program, if we place '*' before P1 when we can access the variable whose address is stored in P1. Since P1 contains the address of variable a, we can access the variable a by waiting *P1. Similarly we can access variable b by writing *P2. So we can use *P1 & *P2 in place of variable names a and b anywhere in our program. Let us see some other examples:

*P1=9; is equivalent to a=9;

(*P1)++; is equivalent to a++;

x=*P2+10; is equivalent to x=b+10;

printf("%d %f",*P1,*P2); is equivalent to printf("%d%f",a,b);

scanf("%d%f",P1,P2); is equivalent to scanf("%d%f",&a,&b);

```
#include<stdio.h>

main( ){

int a=50;

float b=4.5;

int *P1=&a;

float *P2=&b;

printf(“value of P1 = address of a = %u\n”,P1);

printf(“value of P2 = address of b = %u\n”, P2);

printf(“Address of P1 = %u\n”,&P1);

printf(“Address of P2 = %u\n’, &P2);

printf(“value of a= %d%d%d\n”,a,*P1,*(&a));

printf(“value of b = %f%f%f\n”,b,*P2,*(&b);

}
```

OUTPUT:
Value of P1 = Address of a = 65524
Value of P2 = Address of b = 65520
Address of P1 = 65518
Address of P2 = 65516
Value of a = 50 50 50
Value of b = 4.500000 4.500000 4.500000

Passing pointers to a function

A pointer can be passed to a function as an argument. Passing a pointer means passing address of a variable instead of value of the variable. As address is passed in this case, this mechanism is also k\a call by address or call by reference. When pointer is passed to a function, while function calling, the formal argument of the function must be compatible with the passing pointer i.e. if integer pointer is being passed, the formal argument in function must be pointer of the type integer and so on. As address of variable is passed in this mechanism, if value in the passed address is changed within function, the value of actual variable also changed.

```
* Program to illustrate the use of passing pointer to a function */
#include<stdio.h>
void addGraceMarks(int *m){
        *m = *m+10;
}
void main( )
{
int marks;
printf(“Enter actual marks: It”);
scanf(“%d”,&marks);
addGraceMarks(&marks); \* Passing address *\
printf(“\n The graced marks is: \t%d”, marks);
}
```

```
\* Program to convert upper case letter into lower and vice versa using passing pointer to a function *\
#include<stdio.h>
void conversion(char*); \ function prototype *\
main( ) {
char input;
printf(“Enter character of our choice:\n”);
scanf(“%c”,&input);
conversion(&input);
printf(“\n The corresponding character is:\t%c”,input);
}
void conversion(char*c) {
        if(*c>=97 && *c<=122)
        *c=*c-32;
        else if c *c>=65 && *c<=90)
        *c=*c+32;                }
```

Relationship between array and pointer

There is a close association between pointer and array. Array name is pointer to itself.

For example:

int a[5],*P;

P=&a[0];

After statement p=&a[0]; p point the array i.e. the p contains the address of a[0] (first address of array)

P++;

After this statement P points to a[1] element.

```
#include<stdio.h>
void main( )
{
int a[5] = {5, 6, 7, 8, 9};
int *P, i;
P = &a[0];
for(i=0; i<=4; i++)
printf("%d\n",*(P+i));
}
```

```
#include <stdio.h>

int* larger(int*, int*);

void main()
{
    int a = 15;
    int b = 92;
    int *p;
    p = larger(&a, &b);
    printf("%d is larger",*p);
}

int* larger(int *x, int *y)
{
    if(*x > *y)
        return x;
    else
        return y;
}
```

Exercises

- WAP to enter a variable and display its address space.
- WAP to define an array, input some elements and find their sum using pointer.
- WAP using pointers swap two variables using function
- WAP using pointers to determine the largest of two numbers using function.

Dynamic Memory Allocation

As you know, an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.

Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.

To allocate memory dynamically, library functions are `malloc()`, `calloc()`, `realloc()` and `free()` are used. These functions are defined in the `<stdlib.h>` header file.

C malloc()

The name "malloc" stands for memory allocation.

The `malloc()` function reserves a block of memory of the specified number of bytes. And, it returns a pointer of `void` which can be casted into pointers of any form.

Syntax: `ptr = (castType*) malloc(size);`

```
ptr = (float*) malloc(100 * sizeof(float));
```

C calloc()

The name "calloc" stands for contiguous allocation.

The `malloc()` function allocates memory and leaves the memory uninitialized, whereas the `calloc()` function allocates memory and initializes all bits to zero.

Syntax of calloc()

```
ptr = (castType*)calloc(n, size);
```

Example:

```
ptr = (float*) calloc(25, sizeof(float));
```

The void pointer in C is **a pointer that is not associated with any data types**. It points to some data location in the storage. This means that it points to the address of variables. It is also called the general purpose pointer. In C, malloc() and calloc() functions return void * or generic pointers.

C free()

Dynamically allocated memory created with either `calloc()` or `malloc()` doesn't get freed on their own. You must explicitly use `free()` to release the space.

Syntax of free()

```
free(ptr);
```

C realloc()

If the dynamically allocated memory is insufficient or more than required, you can change the size of previously allocated memory using the `realloc()` function.

Syntax of realloc()

```
ptr = realloc(ptr, x);
```

```
// Program to calculate the sum of n numbers entered by the user

#include <stdio.h>
#include <stdlib.h>

int main() {
  int n, i, *ptr, sum = 0;

  printf("Enter number of elements: ");
  scanf("%d", &n);

  ptr = (int*) malloc(n * sizeof(int));

  // if memory cannot be allocated
  if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
  }

  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }

  printf("Sum = %d", sum);

  // deallocating the memory
  free(ptr);

  return 0;
}
```

// Program to calculate the sum of n numbers entered by the user

```
#include <stdio.h>
#include <stdlib.h>

int main() {
  int n, i, *ptr, sum = 0;
  printf("Enter number of elements: ");
  scanf("%d", &n);

  ptr = (int*) calloc(n, sizeof(int));
  if(ptr == NULL) {
    printf("Error! memory not allocated.");
    exit(0);
  }
  printf("Enter elements: ");
  for(i = 0; i < n; ++i) {
    scanf("%d", ptr + i);
    sum += *(ptr + i);
  }

  printf("Sum = %d", sum);
  free(ptr);
  return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
  int *ptr, i , n1, n2;
  printf("Enter size: ");
  scanf("%d", &n1);

  ptr = (int*) malloc(n1 * sizeof(int));

  printf("Addresses of previously allocated memory:\n");
  for(i = 0; i < n1; ++i)
    printf("%p\n",ptr + i);

  printf("\nEnter the new size: ");
  scanf("%d", &n2);

  // reallocating the memory
  ptr = realloc(ptr, n2 * sizeof(int));

  printf("Addresses of newly allocated memory:\n");
  for(i = 0; i < n2; ++i)
    printf("%p\n", ptr + i);

  free(ptr);

  return 0;
}
```

Double Pointers

A pointer is used to store the address of variables.

So, when we define a pointer to pointer, the first pointer is used to store the address of the second pointer. Thus it is known as double pointers.

```
int main() {
   int v = 76;
   int *p1;
   int **p2;
   p1 = &v;
   p2 = &p1;
   printf("Value of v = %d\n", v);
   printf("Value of v using single pointer = %d\n", *p1 );
   printf("Value of v using double pointer = %d\n", **p2);
   return 0;
}
```

Program to add numbers in an array using pointer

```
#include <stdio.h>
int add(int *s, int n)
{
    int sum=0;
    for(int i=0;i<n;i++)
    sum+= *(s+i);
    return sum;
}
int main()
{
    int a[10],i,n,as=0;
    printf("Enter a number:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter elements of an array [%d]",i+1);
        scanf("%d",&a[i]);
    }
        as+=add(a,n);
    printf("The total sum is %d",as);
    return 0;          }
```

WAP to find the largest in an array using pointer.

```
int main()
{
    int *a,i,n,largest;
    printf("Enter a number:");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("Enter elements of an array [%d]",i+1);
        scanf("%d",(a+i));
    }
    largest=*(a);
    for(i=0;i<n;i++)
    {
        if(largest<*(a+i))
        {
            largest=*(a+i);
        }
    }
    printf("The total sum is %d",largest);
    return 0;
}
```

WAP to find the length of string using pointer and function

```
#include <stdio.h>
void count(char *s, int n)
{
    int count=0;
    while(*s != '\0')
    {
        count++;
        s++;
    }
    printf("The total count is %d",count);
}
int main()
{
    char s[20];
    int n;
    printf("Enter the string:");
    gets(s);
    n=strlen(s);

    count(s,n);
    return 0;          }
```

WAP to find the factorial and square using pointers and functions

```
#include <stdio.h>
void factorial(int *s)
{
    int fact=1;
    for(int i=1;i<=*s;i++)
    {
        fact*=i;
    }
    printf("The factorial is %d",fact);
}
void sqr(int *s)
{
    printf("The square is %d",*s * *s);
}
```

```
int main()
{
    int n,f;
    printf("Enter the string:");
    scanf("%d",&n);
    factorial(&n);
    printf("The factorial is %d",f);
    sqr(&n);

    return 0;
}
```

```
#include <stdio.h>
void length(char *p, char *q){
    int count=0;
    while(*p!='\0'){
        p++;
    }
    while(*q!='\0'){
        *p=*q;
        q++;
        p++;
    }
    *p='\0';
}
```

```
int main()
{
    char a[50],b[50];
    printf("Enter a string:");
    gets(a);
    gets(b);
    length(a,b);
    printf("The length of the string is %s",a);

    return 0;
}
```