Pointer Notation

Consider the declaration,

```
int i = 3 ;
```

This declaration tells the C compiler to:

(a) Reserve space in memory to hold the integer value.

(b) Associate the name **i** with this memory location.

(c) Store the value 3 at this location.

We may represent **i**'s location in memory by the memory map shown in Figure 9.1.

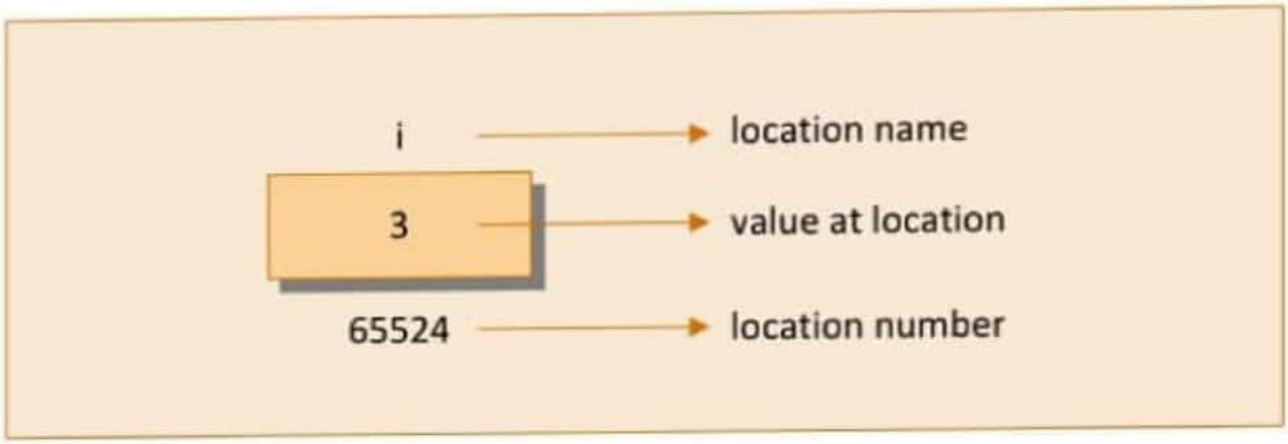


Figure 9.1

We see that the computer has selected memory location 65524 as the place to store the value 3. The location number 65524 is not a number to be relied upon, because some other time the computer may choose a different location for storing the value 3. The important point is, **i**'s address in memory is a number.

We can print this address number through the following program:

```
# include <stdio.h>
int main( )
{
    int i = 3 ;
    printf ( "Address of i = %u\n", &i ) ;
    printf ( "Value of i = %d\n", i ) ;
    return 0 ;
}
```

Scanned with CamScanner

Scanned with CamScanner

```
Address of i = 65524
Value of i = 3
```

Look at the first **printf( )** statement carefully. '&' used in this statement is C's 'address of' operator. The expression **&i** returns the address of the variable **i**, which in this case happens to be 65524. Since 65524 represents an address, there is no question of a sign being associated with it. Hence it is printed out using **%u**, which is a format specifier for printing an unsigned integer. We have been using the '&' operator all the time in the **scanf( )** statement.

The other pointer operator available in C is '*****', called 'value at address' operator. It gives the value stored at a particular address. The 'value at address' operator is also called 'indirection' operator.

Observe carefully the output of the following program:

```
# include <stdio.h>
int main( )
{
    int  i = 3 ;
    printf ( "Address of i = %u\n", &i ) ;
    printf ( "Value of i = %d\n", i ) ;
    printf ( "Value of i = %d\n", *( &i ) ) ;
    return 0 ;
}
```

The output of the above program would be:

```
Address of i = 65524
Value of i = 3
Value of i = 3
```

Note that printing the value of ***(&i)** is same as printing the value of **i**.

The expression **&i** gives the address of the variable **i**. This address can be collected in a variable, by saying,

```
j = &i ;
```

But remember that **j** is not an ordinary variable like any other integer variable. It is a variable that contains the address of other variable (**i** in this case). Since **j** is a variable, the compiler must provide it space in the

Scanned with CamScanner

Scanned with CamScanner

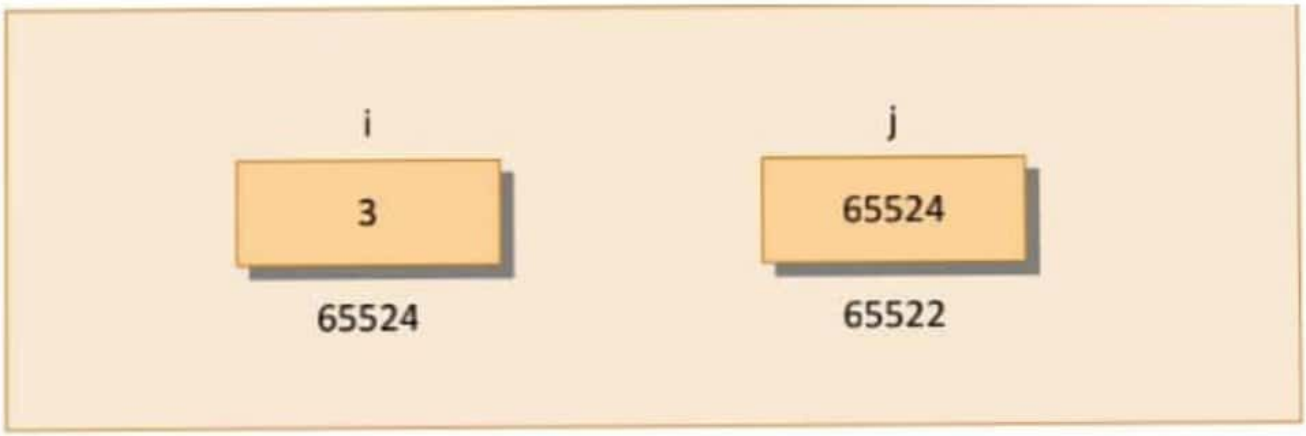


Figure 9.2

As you can see, **i**'s value is 3 and **j**'s value is **i**'s address.

But wait, we can't use **j** in a program without declaring it. And since **j** is a variable that contains the address of **i**, it is declared as,

```
int *j ;
```

This declaration tells the compiler that **j** will be used to store the address of an integer value. In other words, **j** points to an integer. How do we justify the usage of * in the declaration,

```
int *j ;
```

Let us go by the meaning of *. It stands for 'value at address'. Thus, **int *j** would mean, the value at the address contained in **j** is an **int**.

Here is a program that demonstrates the relationships we have been discussing.

```
# include <stdio.h>
int main( )
{
    int i = 3 ;
    int *j ;

    j = &i ;
    printf ( "Address of i = %u\n", &i ) ;
    printf ( "Address of i = %u\n", j ) ;
    printf ( "Address of j = %u\n", &j ) ;
    printf ( "Value of j = %u\n", j ) ;
    printf ( "Value of i = %d\n", i ) ;
```

Scanned with CamScanner

Scanned with CamScanner

```
    printf ( "Value of i = %d\n", *( &i ) ) ;
    printf ( "Value of i = %d\n", *j ) ;
    return 0 ;
}
```

The output of the above program would be:

```
Address of i = 65524
Address of i = 65524
Address of j = 65522
Value of j = 65524
Value of i = 3
Value of i = 3
Value of i = 3
```

Work through the above program carefully, taking help of the memory locations of **i** and **j** shown earlier. This program summarizes everything that we have discussed so far. If you don't understand the program's output, or the meanings of **&i**, **&j**, ***j** and ***(&i)**, re-read the last few pages. Everything we say about pointers from here onwards will depend on your understanding these expressions thoroughly.

Look at the following declarations:

```
int  *alpha ;
char  *ch ;
float  *s ;
```

Here, **alpha, ch** and **s** are declared as pointer variables, i.e., variables capable of holding addresses. Remember that, addresses (location nos.) are always going to be whole numbers, therefore pointers always contain whole numbers. Now we can put these two facts together and say—pointers are variables that contain addresses, and since addresses are always whole numbers, pointers would always contain whole numbers.

The declaration **float *s** does not mean that **s** is going to contain a floating-point value. What it means is, **s** is going to contain the address of a floating-point value. Similarly, **char *ch** means that **ch** is going to contain the address of a char value. Or in other words, the value at address stored in **ch** is going to be a **char**.

The concept of pointers can be further extended. Pointer, we know is a variable that contains address of another variable. Now this variable

Scanned with CamScanner

Scanned with CamScanner

itself might be another pointer. Thus, we now have a pointer that contains another pointer's address. The following example should make this point clear:

```
# include <stdio.h>
int main( )
{
    int  i = 3, *j, **k ;

    j = &i ;
    k = &j ;
    printf ( "Address of i = %u\n", &i ) ;
    printf ( "Address of i = %u\n ", j ) ;
    printf ( "Address of i = %u\n ", *k ) ;
    printf ( "Address of j = %u\n ", &j ) ;
    printf ( "Address of j = %u\n ", k ) ;
    printf ( "Address of k = %u\n ", &k ) ;
    printf ( "Value of j  = %u\n ", j ) ;
    printf ( "Value of k  = %u\n ", k ) ;
    printf ( "Value of i  = %d\n ", i ) ;
    printf ( "Value of i  = %d\n ", * ( &i ) ) ;
    printf ( "Value of i  = %d\n ", *j ) ;
    printf ( "Value of i  = %d\n ", **k ) ;
    return 0 ;
}
```

The output of the above program would be:

```
Address of i = 65524
Address of i = 65524
Address of i = 65524
Address of j = 65522
Address of j = 65522
Address of k = 65520
Value of j  = 65524
Value of k  = 65522
Value of i  = 3
Value of i  = 3
Value of i  = 3
Value of i  = 3
```

Scanned with CamScanner

Scanned with CamScanner

Figure 9.3 would help you in tracing out how the program prints the above output.

Remember that when you run this program, the addresses that get printed might turn out to be something different than the ones shown in Figure 9.3. However, with these addresses too, the relationship between **i**, **j** and **k** can be easily established.

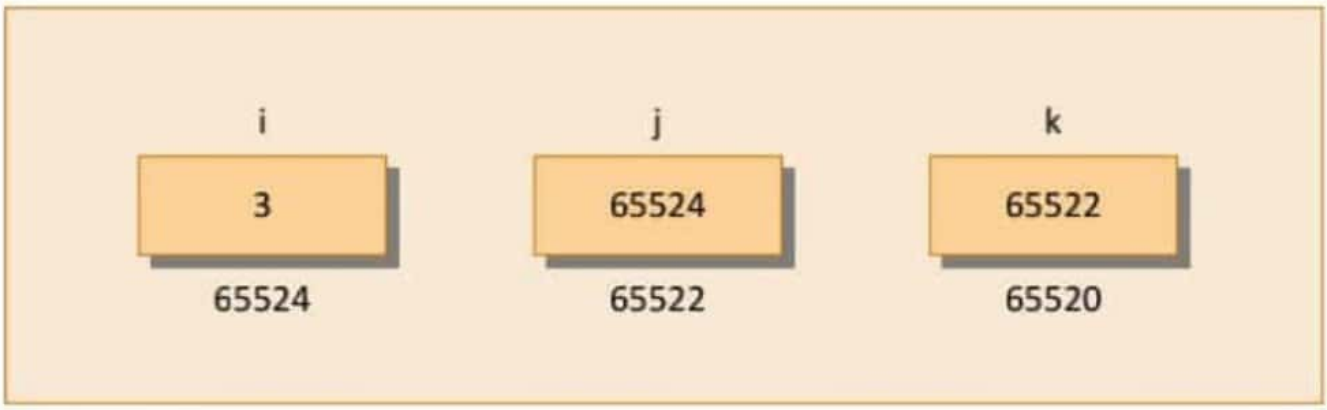


Figure 9.3

Observe how the variables **j** and **k** have been declared,

```
int i, *j, **k ;
```

Scanned with CamScanner

Scanned with CamScanner

```
# include <stdio.h>
void swapv ( int  x, int  y ) ;
int main( )
{
    int  a = 10, b = 20 ;
    swapv ( a, b ) ;
    printf ( "a = %d b = %d\n", a, b ) ;
    return 0 ;
}
void swapv ( int  x, int  y )
{
    int  t ;
    t = x ;
    x = y ;
    y = t ;
    printf ( "x = %d y = %d\n", x, y ) ;
}
```

The output of the above program would be:

```
x = 20 y = 10
a = 10 b = 20
```

Note that values of **a** and **b** remain unchanged even after exchanging the values of **x** and **y**.

In the second method (call by reference), the addresses of actual arguments in the calling function are copied into the formal arguments of the called function. This means that, using these addresses, we would have an access to the actual arguments and hence we would be able to manipulate them. The following program illustrates this fact:

```
# include <stdio.h>
void swapr ( int  *, int  * ) ;
int main( )
{
    int  a = 10, b = 20 ;
    swapr ( &a, &b ) ;
```

Scanned with CamScanner

Scanned with CamScanner

```
    printf ( "a = %d b = %d\n", a, b ) ;
    return 0 ;
}
void swapr ( int  *x, int  *y )
{
    int  t ;

    t = *x ;
    *x = *y ;
    *y = t ;
}
```

The output of the above program would be:

```
a = 20 b = 10
```

Note that this program manages to exchange the values of **a** and **b** using their addresses stored in **x** and **y**.

Usually, in C programming, we make a call by value. This means that, in general, you cannot alter the actual arguments. But if desired, it can always be achieved through a call by reference.

Using a call by reference intelligently, we can make a function return more than one value at a time, which is not possible ordinarily. This is shown in the program given below.

```
# include <stdio.h>
void areaperi ( int, float  *, float  * ) ;
int main( )
{
    int  radius ;
    float  area, perimeter ;

    printf ( "Enter radius of a circle " ) ;
    scanf ( "%d", &radius ) ;
    areaperi ( radius, &area, &perimeter ) ;

    printf ( "Area = %f\n", area ) ;
    printf ( "Perimeter = %f\n", perimeter ) ;
    return 0 ;
}
void areaperi ( int  r, float  *a, float  *p )
```

Scanned with CamScanner

Scanned with CamScanner

```
{
    *a = 3.14 * r * r ;
    *p = 2 * 3.14 * r ;
}
```

And here is the output...

```
Enter radius of a circle 5
Area = 78.500000
Perimeter = 31.400000
```

Here, we are making a mixed call, in the sense, we are passing the value of **radius** but, addresses of **area** and **perimeter**. And since we are passing the addresses, any change that we make in values stored at addresses contained in the variables **a** and **p**, would make the change effective in **main( )**. That is why, when the control returns from the function **areaperi( )**, we are able to output the values of **area** and **perimeter**.

Thus, we have been able to indirectly return two values from a called function, and hence, have overcome the limitation of the **return** statement, which can return only one value from a function at a time.

Conclusions

From the programs that we discussed here, we can draw the following conclusions:

(a) If we want that the value of an actual argument should not get changed in the function being called, pass the actual argument by value.

(b) If we want that the value of an actual argument should get changed in the function being called, pass the actual argument by reference.

(c) If a function is to be made to return more than one value at a time, then return these values indirectly by using a call by reference.

Summary

(a) Pointers are variables which hold addresses of other variables.

(b) A pointer to a pointer is a variable that holds address of a pointer variable.

(c) The & operator fetches the address of the variable in memory.

Scanned with CamScanner

Scanned with CamScanner

```
main( )
{
    int i = 3, *x ;
    float j = 1.5, *y ;
    char k = 'c', *z ;

    printf ( "\nValue of i = %d", i ) ;
    printf ( "\nValue of j = %f", j ) ;
    printf ( "\nValue of k = %c", k ) ;
    x = &i ;
    y = &j ;
    z = &k ;
    printf ( "\nOriginal address in x = %u", x ) ;
    printf ( "\nOriginal address in y = %u", y ) ;
    printf ( "\nOriginal address in z = %u", z ) ;
    x++ ;
    y++ ;
    z++ ;
    printf ( "\nNew address in x = %u", x ) ;
    printf ( "\nNew address in y = %u", y ) ;
    printf ( "\nNew address in z = %u", z ) ;
}
```

Suppose **i**, **j** and **k** are stored in memory at addresses 1002, 2004 and 5006, the output would be...

Scanned with CamScanner

Scanned with CamScanner

(d) The * operator lets us access the value present at an address in memory with an intension of reading it or modifying it.

(e) A function can be called either by value or by reference.

(f) Pointers can be used to make a function return more than one value simultaneously in an indirect manner.

Scanned with CamScanner

Scanned with CamScanner

Pointers :-

A pointer is a variable that represents the location rather than the value of a data item, such as variable or an array element.

Now, int i = 3;

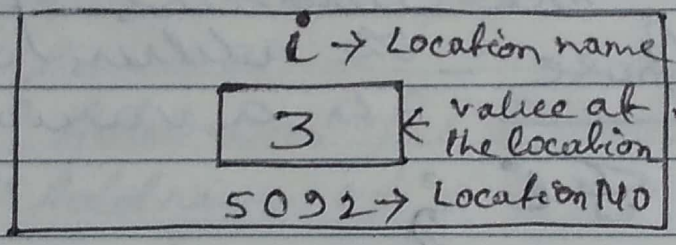
The above declaration tells the compiler to :

(a) Reserve space in memory to hold the integer value

(b) Associate the name i with this memory location.

(c) Store the value 03 at this location.

September • Tuesday



← Memory Map of the above declaration (int i=

```
main()
{ int i = 10 ;
printf("\n Address of
        i = %u", &i);
printf("\n value of i = %d", i);
}
```

pointer variable declaration

(*) → indirection operator

value at address

O/t → 6485

3

Scanned with CamScanner

④
```
main()
{
  int i=3;
printf("Address of i = %u", &i);
"   ("Value of i = %d", i);
"   ( Value of i = %d, *(&i));
}
```

O/P → 6485
3
3

Printing the value of *(&i) is same as printing the value of i.

08

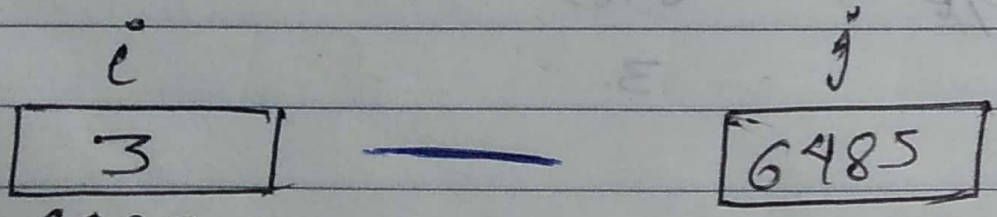
Thursday • September

'&i' gives the address of the variable

Suppose :- The address can be stored in a variable. say 'j'

```
j = &i;
```

'j' is not an ordinary variable. Special type of variable as it can store the address of another variable.



Scanned with CamScanner

i's value is 3 and j's value is i's address.

j can be declared as:

int *j;

j is called a pointer.

declaration tells compiler that j will be used to store the address of an integer value.

i.e, j can

j points to an integer variable

```
main()
{
  int i=3, *j;
  j = &i;
  printf("Address of i = %u", &i);
  printf("Address of i = %u", j);
  printf("Address of j = %u", &j);
  printf("value of j = %u", j);
  printf("value of i = %d", i);
  printf("value of i = %d", *(&i));
  printf("value of i = %d", *j);
}
```

Address of i = 6485

= 6485

6485

6485

6485

10 September • Saturday
Sunday 11
Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo Tu We Th Fr Sa Su Mo OCT 2011
Scanned with CamScanner

int *alpha;
char *ch;
float *s;

pointer variables always hold an address which must be a whole number.

means → alpha points to an integer value

alpha or

alpha contain the address of a floating point value.

printf("%d %d %d", sizeof(alpha), sizeof(*ch), sizeof(*s))

ch → ch is going to contain the address of a char value. Can point only a char value

O/P
222

S → contain addr of a floating point value

Tuesday • September 13

Pointers :- It is a variable which contains address of another variable.

Now, this variable itself might be another pointer,

i.e,

A pointer can contains the address of another pointer as int, char, float.

i	j	k
3	6485	3276
6485	3276	7234

Scanned with CamScanner