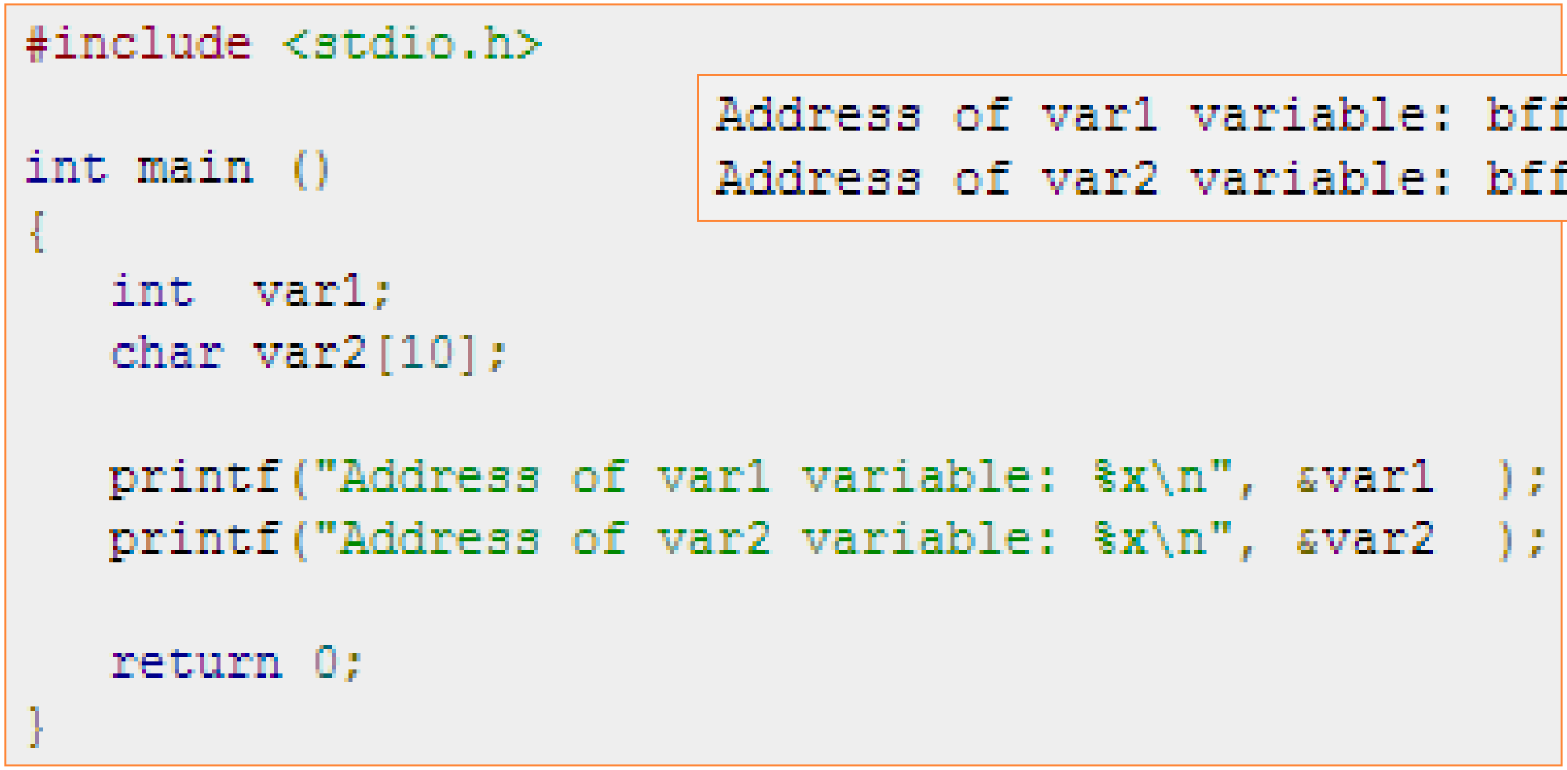
# C – POINTER

Dr. Sheak Rashed Haider Noori
Professor & Associate Head
Department of Computer Science

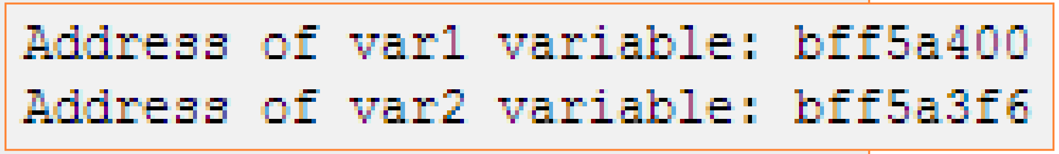
## C – POINTER

- If you want to be proficient in the writing of code in the C programming language, you must have a thorough working knowledge of how to use pointers.
- 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.
- As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which will print the address of the variables defined:

# C – POINTER



```c
#include <stdio.h>

int main ()
{
   int  var1;
   char var2[10];

   printf("Address of var1 variable: %x\n", &var1  );
   printf("Address of var2 variable: %x\n", &var2  );

   return 0;
}
```



```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

So you understood what is memory address and how to access it, so base of the concept is over. Now let us see what is a pointer.

## WHAT IS POINTER?

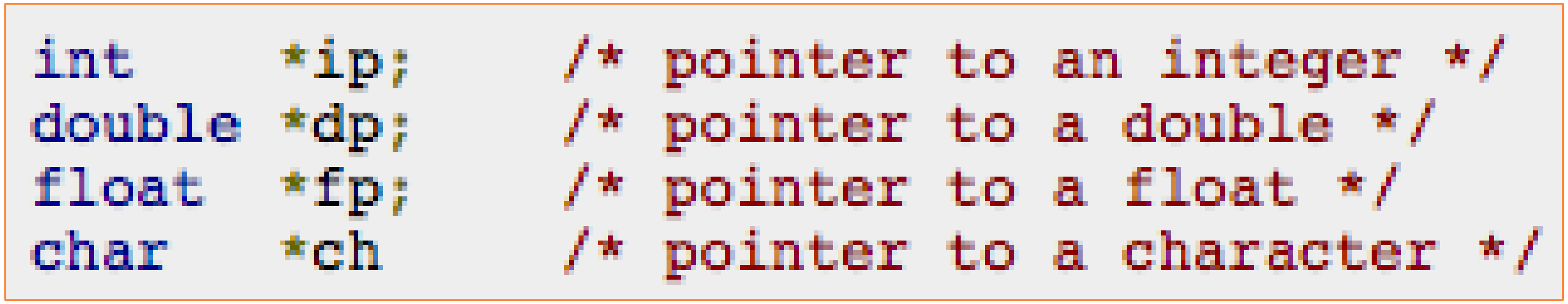
- A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before you can use it to store any variable address.
- The general form of a pointer variable declaration is:

```
dataType  *var_name;
```

- Here,
  - ***dataType*** is the pointer's base type; it must be a valid C data type(i.e., int, float, char etc).
  - **var_name** is the name of the pointer variable.
  - The asterisk * you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer.

## POINTER

- Following are the valid pointer declaration:



```
int    *ip;    /* pointer to an integer */
double *dp;    /* pointer to a double */
float  *fp;    /* pointer to a float */
char   *ch     /* pointer to a character */
```

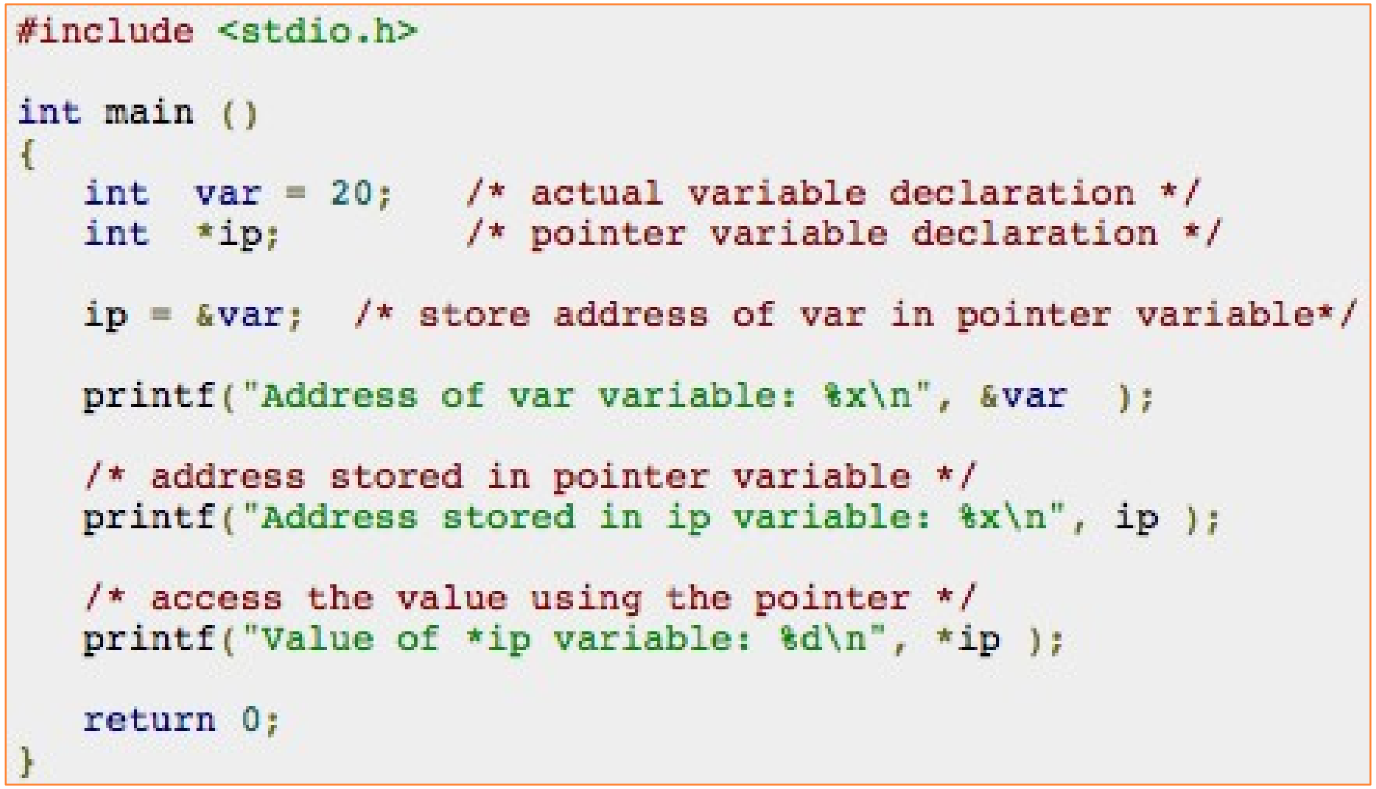
- The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a *long hexadecimal* number that represents a memory address.
- The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

## HOW TO USE POINTERS?

- There are few important operations, which we will do with the help of pointers very frequently.
  1. we define a pointer variable
  2. assign the address of a variable to a pointer and
  3. finally access the value at the address available in the pointer variable by dereferencing.
- Dereferencing is done by using unary operator * that returns the value of the variable located at the address specified by its operand.
- Dereferencing a pointer means getting the value stored in the memory at the address which the pointer "points" to.
- The * is the value-at-address operator, also called the indirection operator. It is used both when declaring a pointer and when dereferencing a pointer.

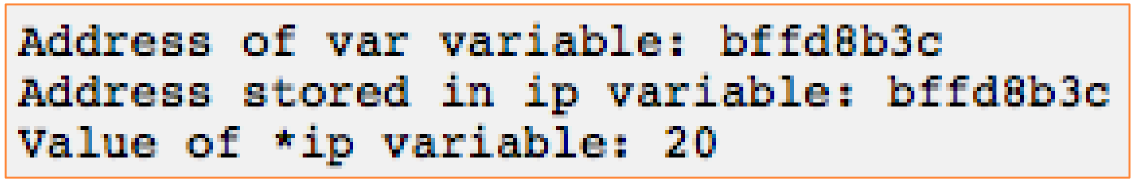
# HOW TO USE POINTERS?

- Following example makes use of these operations:



```c
#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;
}
```

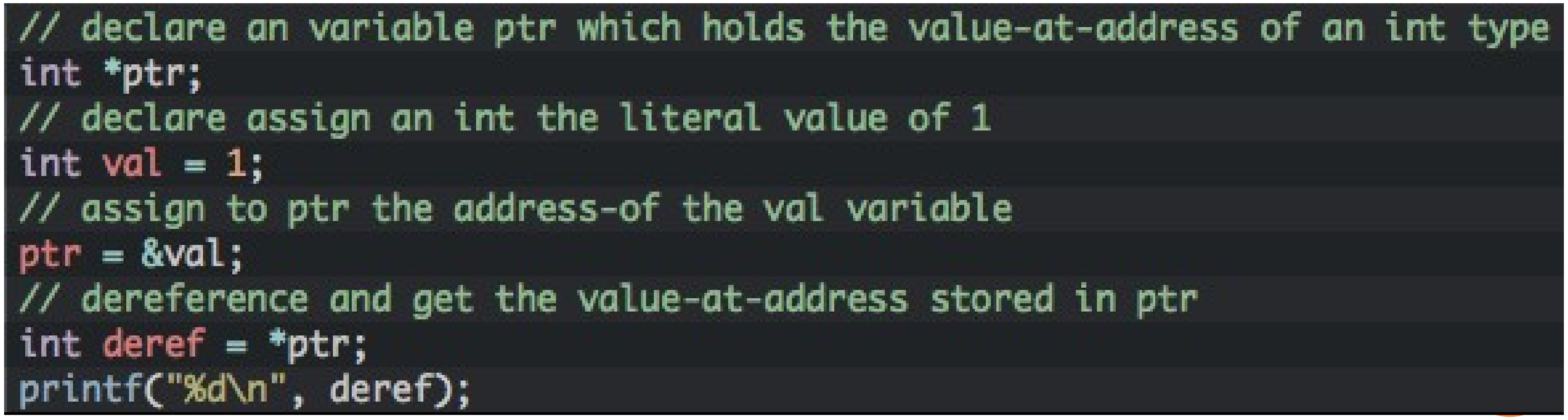
- When the above code is compiled and executed, it produces result something as follows:



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

## HOW TO USE POINTERS?

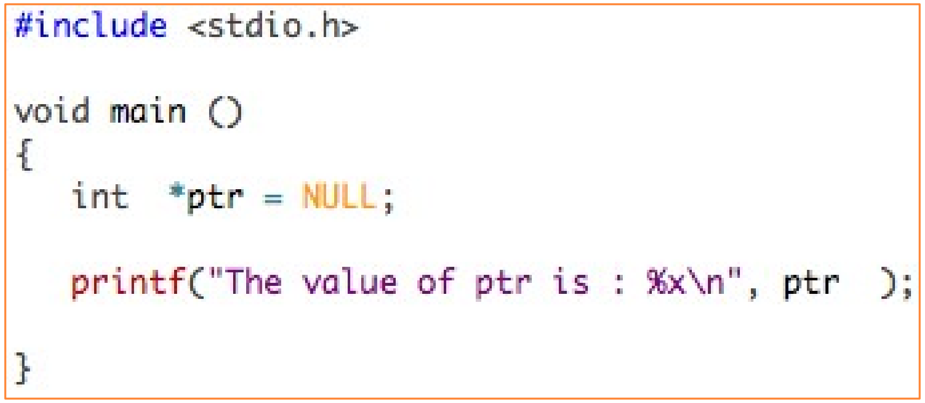
- the & is the address-of operator and is used to reference the memory address of a variable.
- By using the & operator in front of a variable name we can retrieve the memory address-of that variable. It is best to read this operator as address-of operator.
- Following code shows some common notations for the value-at-address (*) and adress-of (&) operators.



```
// declare an variable ptr which holds the value-at-address of an int type
int *ptr;
// declare assign an int the literal value of 1
int val = 1;
// assign to ptr the address-of the val variable
ptr = &val;
// dereference and get the value-at-address stored in ptr
int deref = *ptr;
printf("%d\n", deref);
```

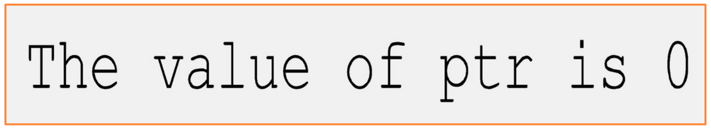
## NULL POINTERS IN C

- It is always a good practice to assign a NULL value to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.
- The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

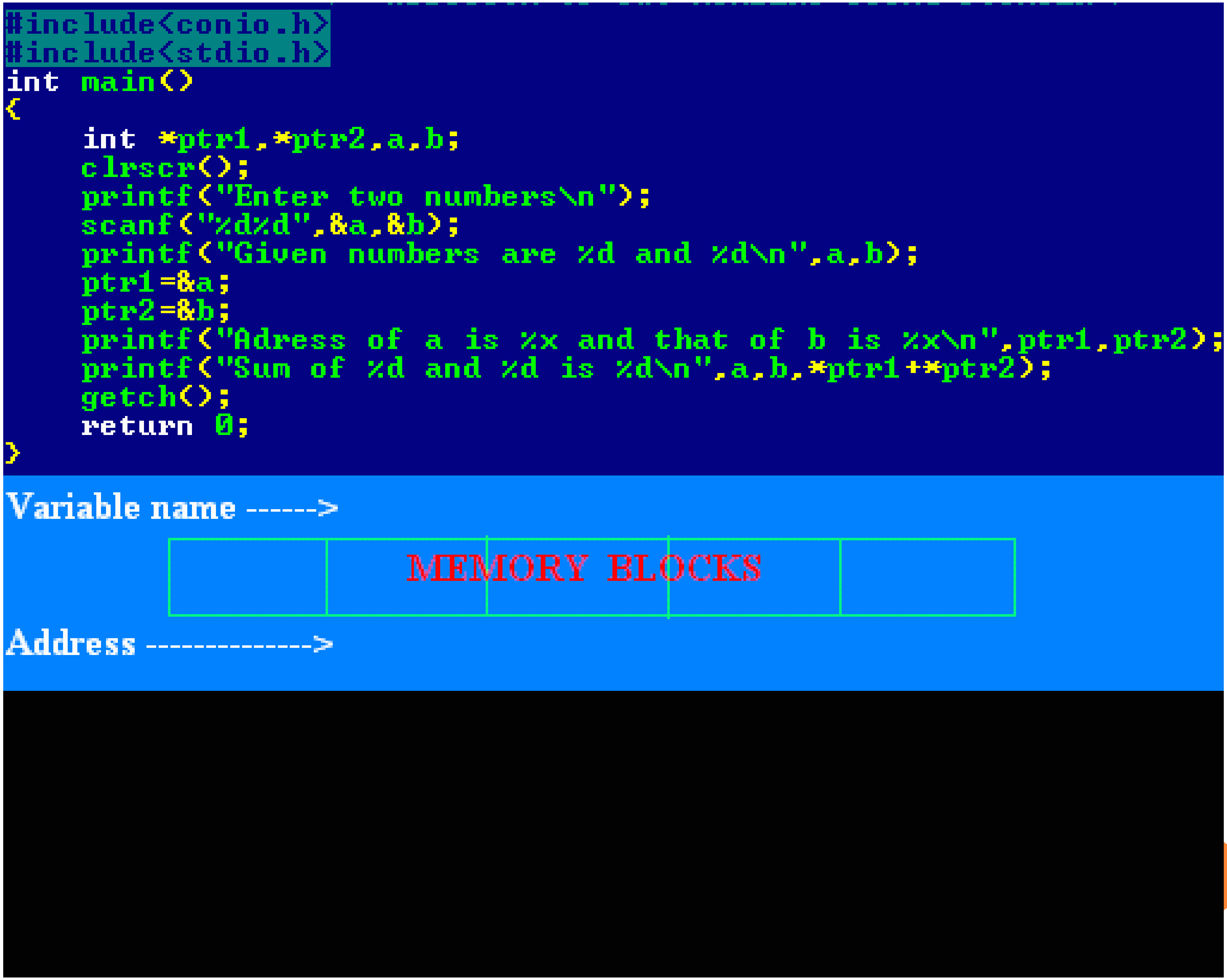


```
#include <stdio.h>

void main ()
{
   int  *ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

}
```

- When the above code is compiled and executed, it produces the following result:



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



```
#include<conio.h>
#include<stdio.h>
int main()
{
     int *ptr1,*ptr2,a,b;
     clrscr();
     printf("Enter two numbers\n");
     scanf("%d%d",&a,&b);
     printf("Given numbers are %d and %d\n",a,b);
     ptr1=&a;
     ptr2=&b;
     printf("Adress of a is %x and that of b is %x\n",ptr1,ptr2);
     printf("Sum of %d and %d is %d\n",a,b,*ptr1+*ptr2);
     getch();
     return 0;
}
```

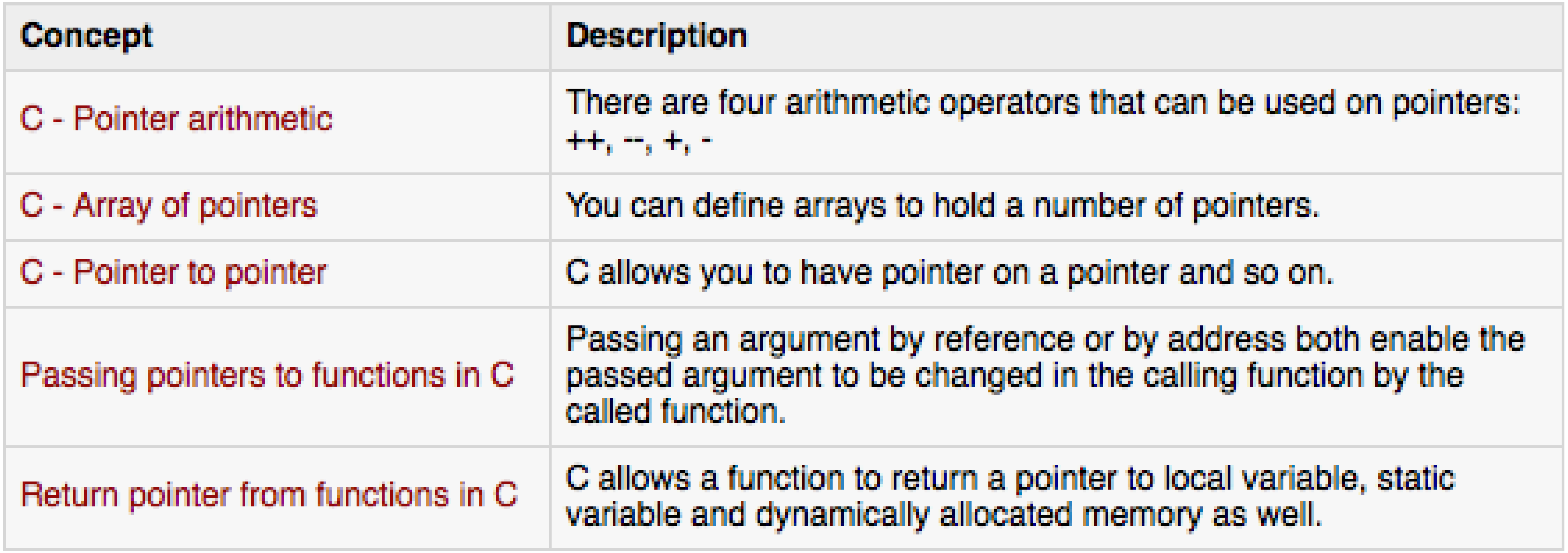
## NULL POINTERS IN C

- On most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.
- To check for a null pointer you can use an if statement as follows:

```
if(ptr)     /* succeeds if p is not null */
if(!ptr)    /* succeeds if p is null */
```

## C POINTERS IN DETAIL

- Pointers have many but easy concepts and they are very important to C programming. There are following few important pointer concepts which should be clear to a C programmer:



| Concept | Description |
|---|---|
| C - Pointer arithmetic | There are four arithmetic operators that can be used on pointers: ++, --, +, - |
| C - Array of pointers | You can define arrays to hold a number of pointers. |
| C - Pointer to pointer | C allows you to have pointer on a pointer and so on. |
| Passing pointers to functions in C | Passing an argument by reference or by address both enable the passed argument to be changed in the calling function by the called function. |
| Return pointer from functions in C | C allows a function to return a pointer to local variable, static variable and dynamically allocated memory as well. |

# C - POINTER ARITHMETIC

- C pointer is an address, which is a numeric value. Therefore, you can perform arithmetic operations on a pointer just as you can a numeric value.
- There are four arithmetic operators that can be used on pointers: ++, --, +, and -
- To understand pointer arithmetic, let us consider that **ptr** is an integer pointer which points to the address 1000. Assuming 32-bit integers, let us perform the following arithmetic operation on the pointer:

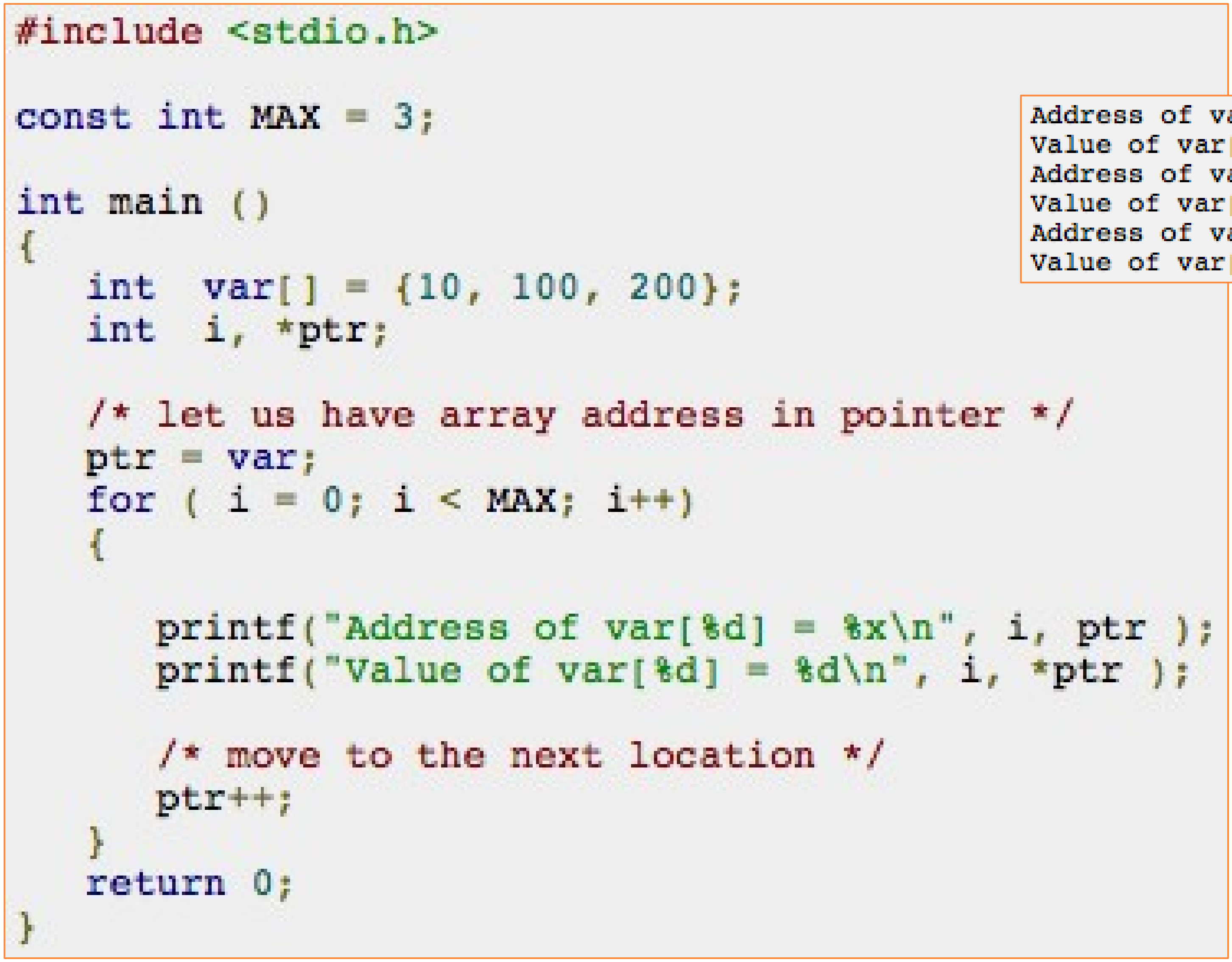
**++ptr**

- Now, after the above operation, the **ptr** will point to the location 1004 because each time ptr is incremented, it will point to the next integer location which is 4 bytes next to the current location.
- If **ptr** points to a character whose address is 1000, then above operation will point to the location 1001 because next character will be available at 1001.

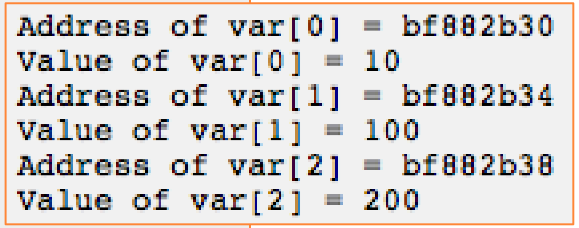
## INCREMENTING A POINTER

- We prefer using a pointer in our program instead of an array because the variable pointer can be incremented, unlike the array name which cannot be incremented because it is a constant pointer.
- The following program increments the variable pointer to access each succeeding element of the array (see next slide).

## INCREMENTING A POINTER



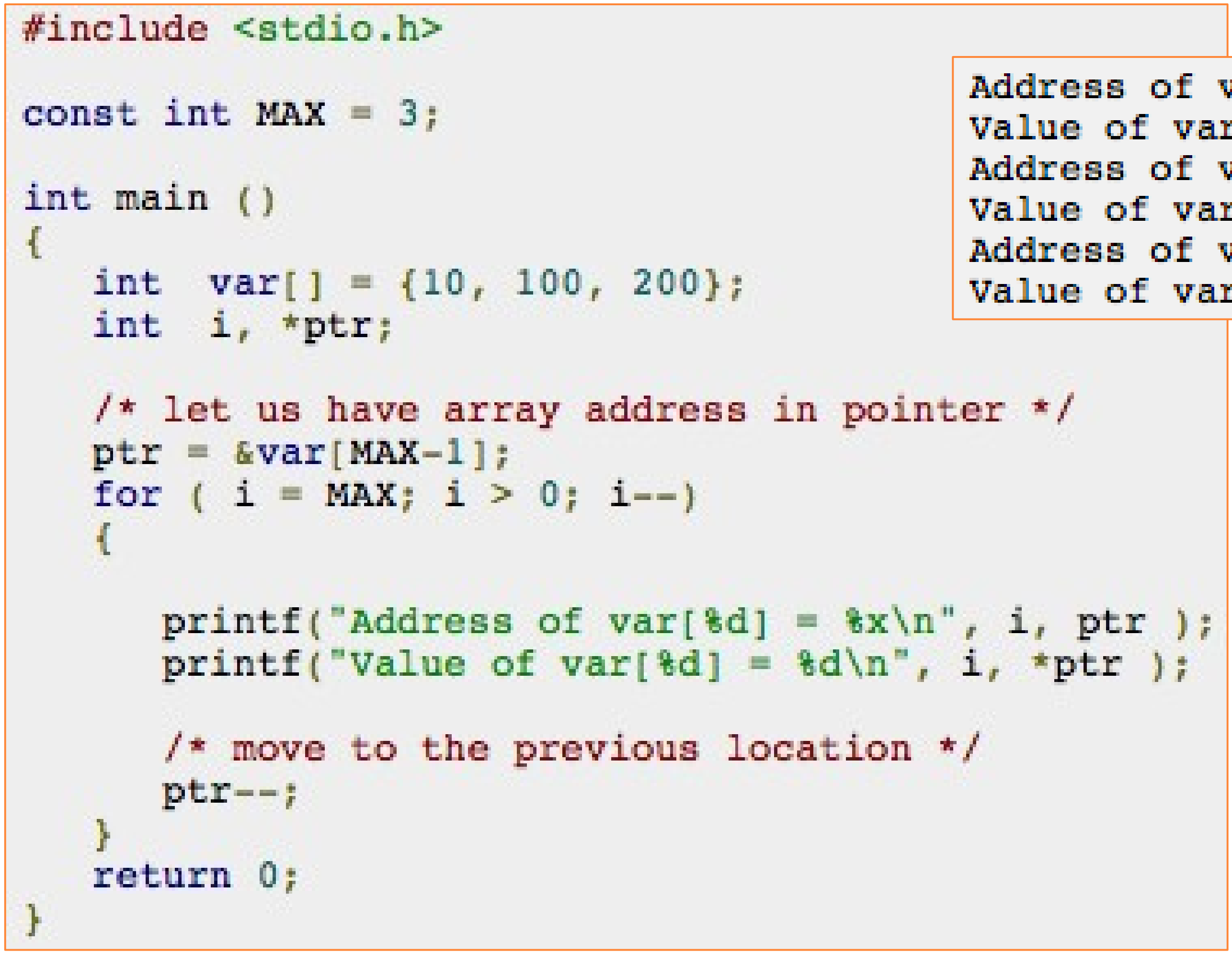
```c
#include <stdio.h>

const int MAX = 3;

int main ()
{
   int  var[] = {10, 100, 200};
   int  i, *ptr;

   /* let us have array address in pointer */
   ptr = var;
   for ( i = 0; i < MAX; i++)
   {

      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );

      /* move to the next location */
      ptr++;
   }
   return 0;
}
```



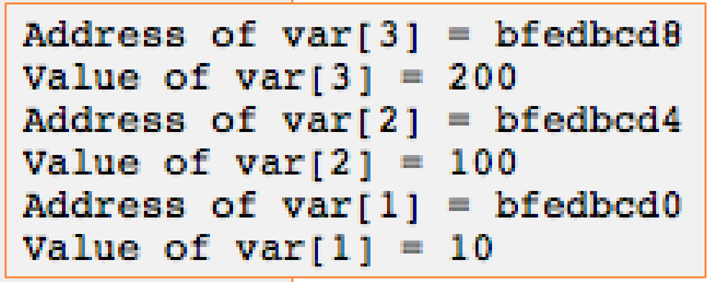
```
Address of var[0] = bf882b30
Value of var[0] = 10
Address of var[1] = bf882b34
Value of var[1] = 100
Address of var[2] = bf882b38
Value of var[2] = 200
```

# DECREMENTING A POINTER

- The same considerations apply to decrementing a pointer, which decreases its value by the number of bytes of its data type as shown below:

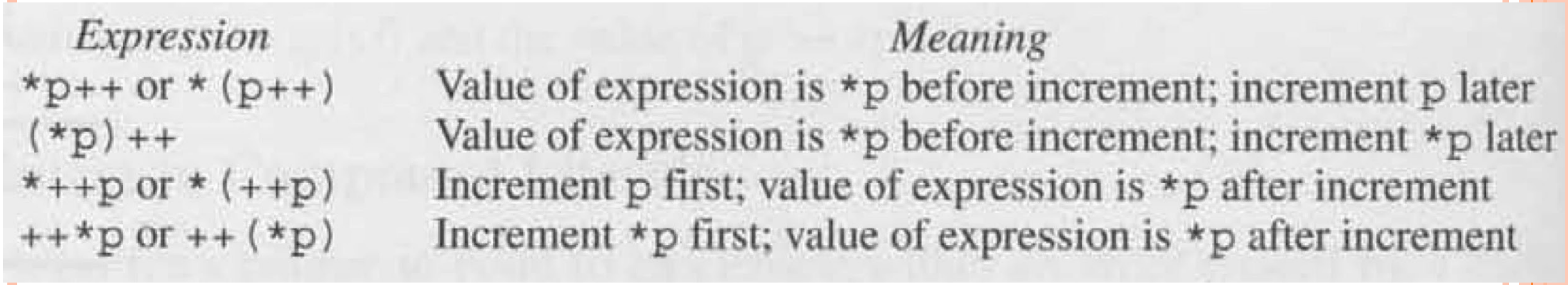


```
#include <stdio.h>

const int MAX = 3;

int main ()
{
   int  var[] = {10, 100, 200};
   int  i, *ptr;

   /* let us have array address in pointer */
   ptr = &var[MAX-1];
   for ( i = MAX; i > 0; i--)
   {

      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );

      /* move to the previous location */
      ptr--;
   }
   return 0;
}
```



```
Address of var[3] = bfedbcd8
Value of var[3] = 200
Address of var[2] = bfedbcd4
Value of var[2] = 100
Address of var[1] = bfedbcd0
Value of var[1] = 10
```

## POINTER * AND ++

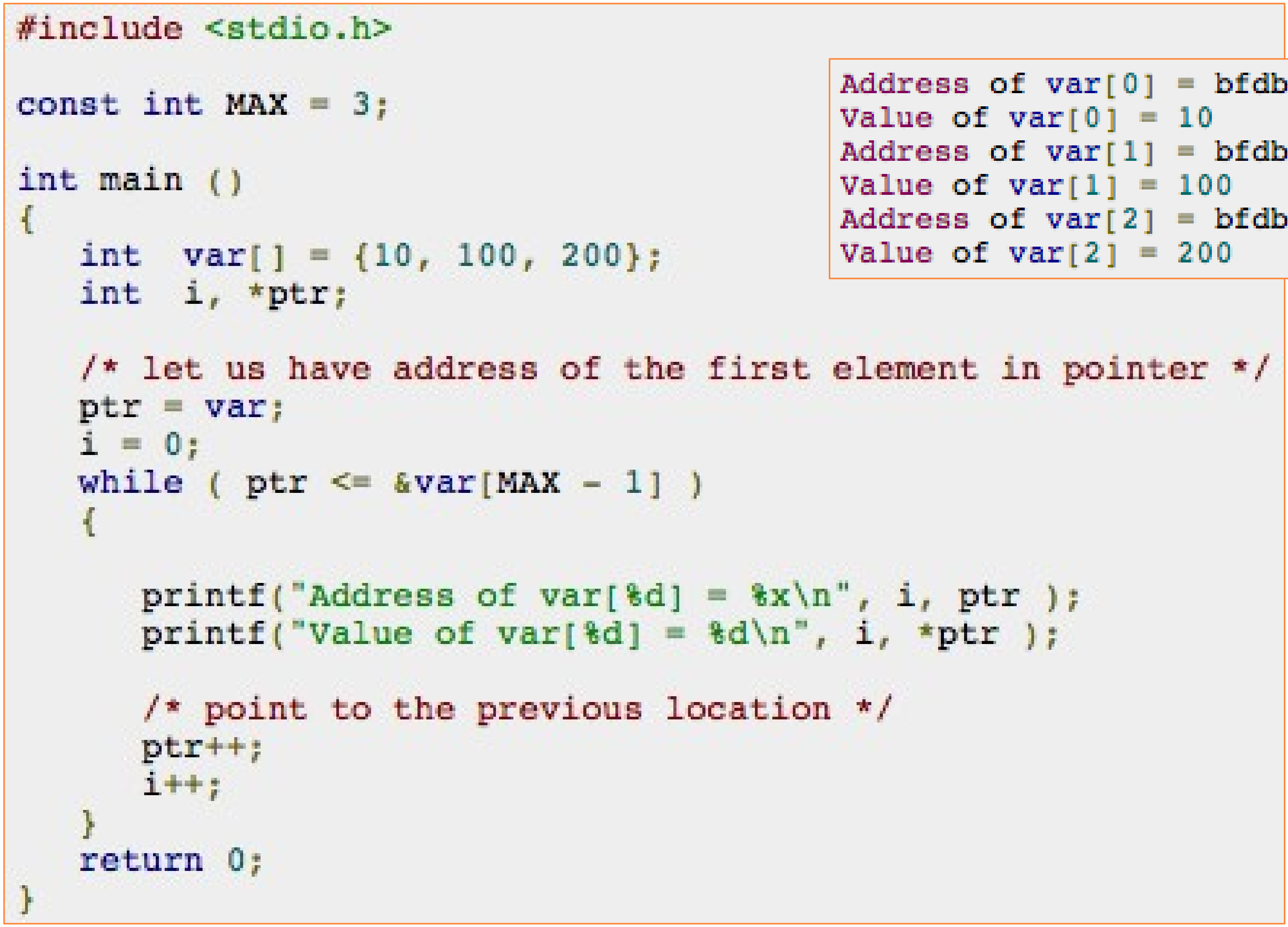


| *Expression* | *Meaning* |
|---|---|
| `*p++` or `*(p++)` | Value of expression is `*p` before increment; increment `p` later |
| `(*p)++` | Value of expression is `*p` before increment; increment `*p` later |
| `*++p` or `*(++p)` | Increment `p` first; value of expression is `*p` after increment |
| `++*p` or `++(*p)` | Increment `*p` first; value of expression is `*p` after increment |

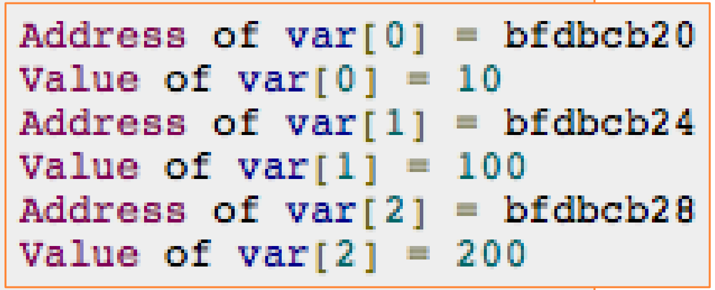
## POINTER COMPARISONS

- Pointers may be compared by using relational operators, such as ==, <, and >.
- If p1 and p2 point to variables that are related to each other, such as elements of the same array, then p1 and p2 can be meaningfully compared.
- The following program modifies the previous example one by incrementing the variable pointer as long as the address to which it points is either less than or equal to the address of the last element of the array, which is &var[MAX - 1] (see next slide).

# POINTER COMPARISONS



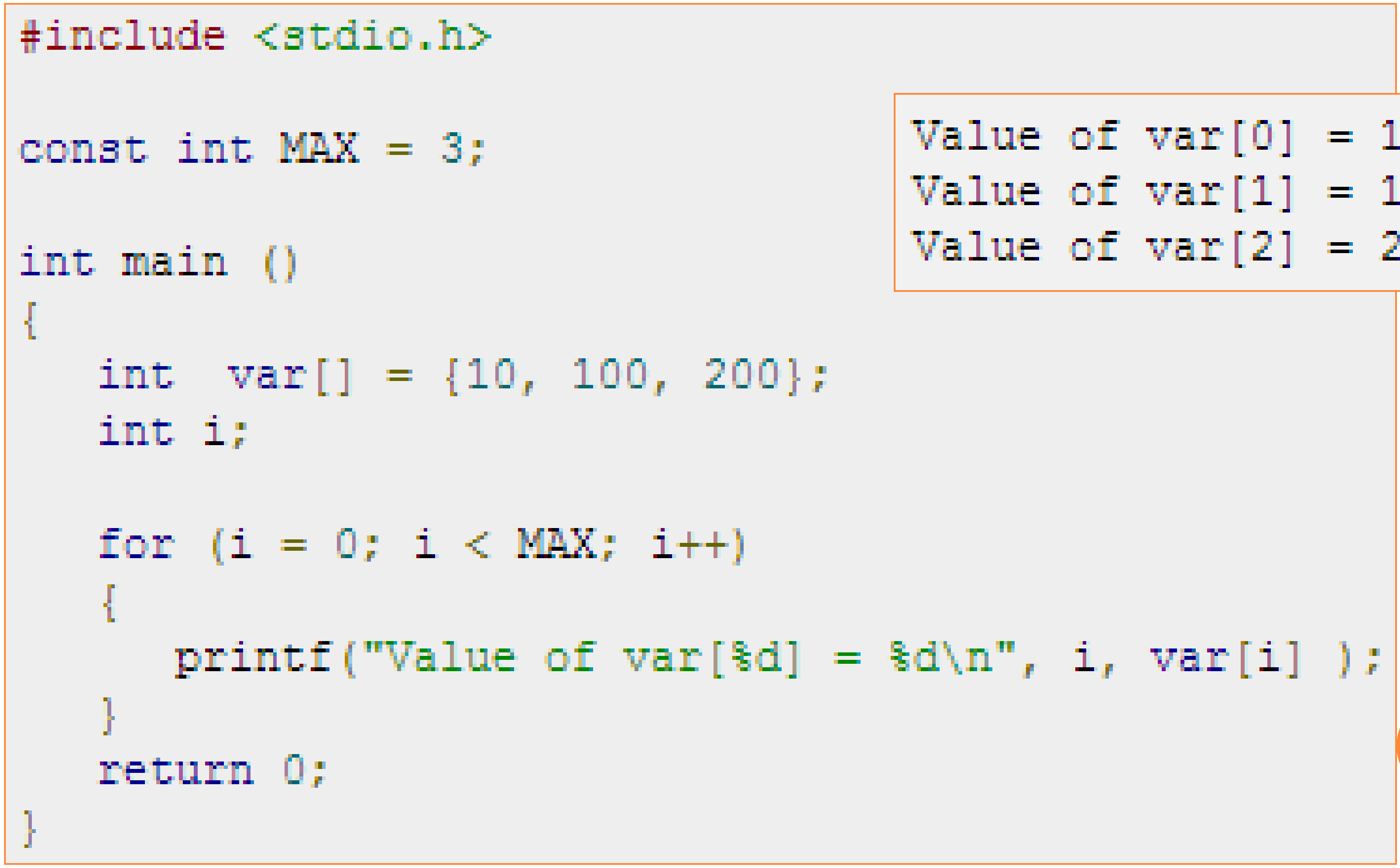
```c
#include <stdio.h>

const int MAX = 3;

int main ()
{
   int  var[] = {10, 100, 200};
   int  i, *ptr;

   /* let us have address of the first element in pointer */
   ptr = var;
   i = 0;
   while ( ptr <= &var[MAX - 1] )
   {

      printf("Address of var[%d] = %x\n", i, ptr );
      printf("Value of var[%d] = %d\n", i, *ptr );

      /* point to the previous location */
      ptr++;
      i++;
   }
   return 0;
}
```



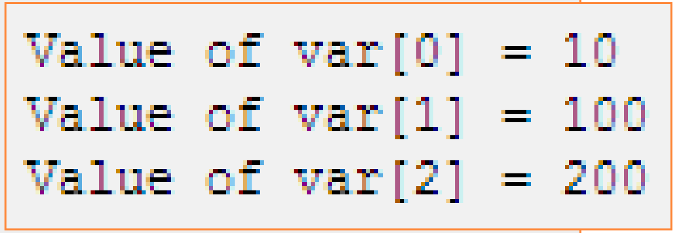
```
Address of var[0] = bfdbcb20
Value of var[0] = 10
Address of var[1] = bfdbcb24
Value of var[1] = 100
Address of var[2] = bfdbcb28
Value of var[2] = 200
```

# ARRAY OF POINTERS

- Before we understand the concept of arrays of pointers, let us consider the following example, which makes use of an array of 3 integers:



```c
#include <stdio.h>

const int MAX = 3;

int main ()
{
   int  var[] = {10, 100, 200};
   int i;

   for (i = 0; i < MAX; i++)
   {
      printf("Value of var[%d] = %d\n", i, var[i] );
   }
   return 0;
}
```



```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

## ARRAY OF POINTERS

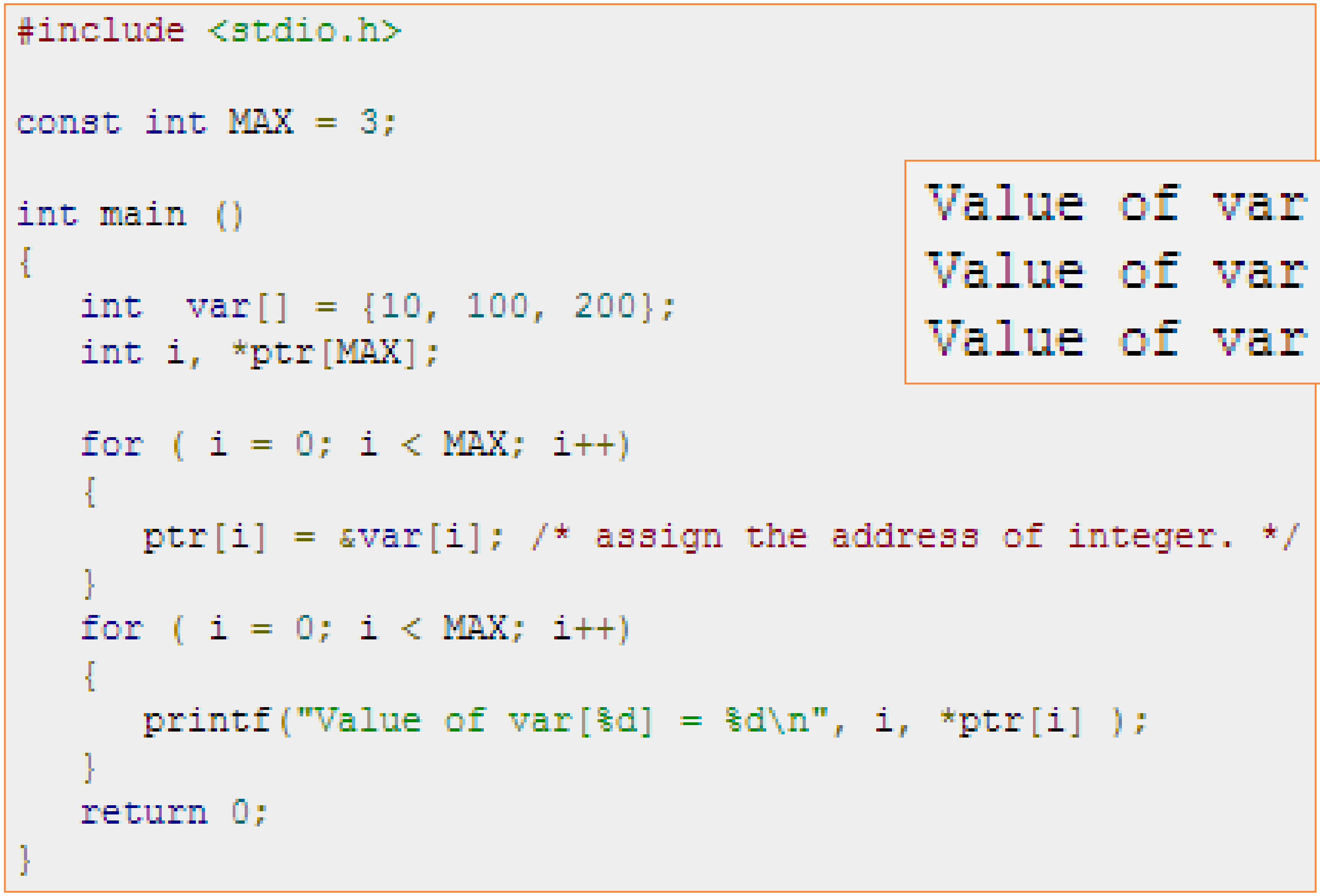
- There may be a situation when we want to maintain an array, which can store pointers to an *int* or *char* or any other data type available. Following is the declaration of an array of pointers to an integer:

*int* *ptr[ ];

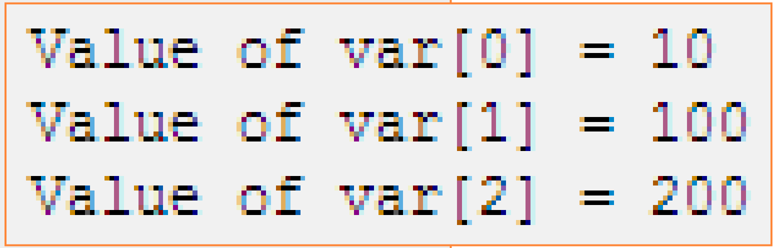
- This declares **ptr** as an array of integer pointers. Thus, each element in ptr, now holds a pointer to an int value. Following example makes use of three integers, which will be stored in an array of pointers as follows:

## ARRAY OF POINTERS

- Following example makes use of three integers, which will be stored in an array of pointers as follows:



```c
#include <stdio.h>

const int MAX = 3;

int main ()
{
   int  var[] = {10, 100, 200};
   int i, *ptr[MAX];

   for ( i = 0; i < MAX; i++)
   {
      ptr[i] = &var[i]; /* assign the address of integer. */
   }
   for ( i = 0; i < MAX; i++)
   {
      printf("Value of var[%d] = %d\n", i, *ptr[i] );
   }
   return 0;
}
```



```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

## ARRAY OF POINTERS

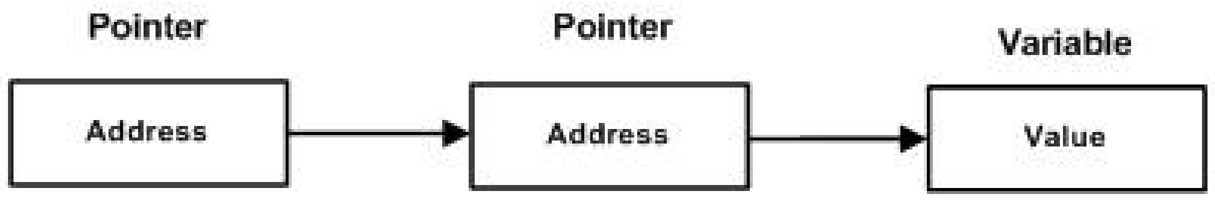
- You can also use an array of pointers to character to store a list of strings as follows:

```
#include <stdio.h>

const int MAX = 4;

int main ()
{
   char *names[] = {
                   "Zara Ali",
                   "Hina Ali",
                   "Nuha Ali",
                   "Sara Ali",
   };
   int i = 0;

   for ( i = 0; i < MAX; i++)
   {
      printf("Value of names[%d] = %s\n", i, names[i] );
   }
   return 0;
}
```

```
Value of names[0] = Zara Ali
Value of names[1] = Hina Ali
Value of names[2] = Nuha Ali
Value of names[3] = Sara Ali
```

# POINTER TO POINTER

- A pointer to a pointer is a form of multiple indirection, or a chain of pointers. Normally, a pointer contains the address of a variable. When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

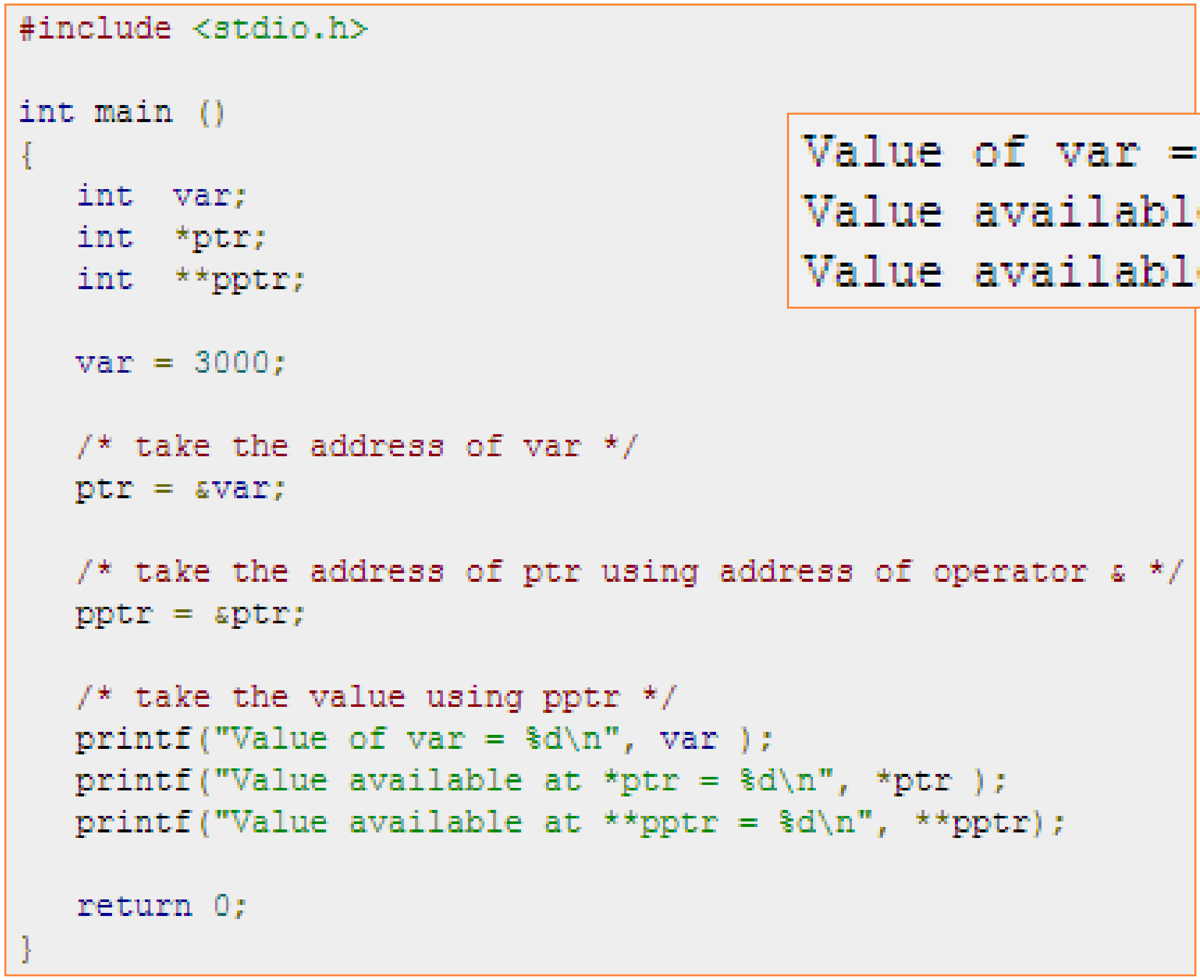


- A variable that is a pointer to a pointer must be declared as such. This is done by placing an additional asterisk in front of its name. For example, following is the declaration to declare a pointer to a pointer of type *int*:

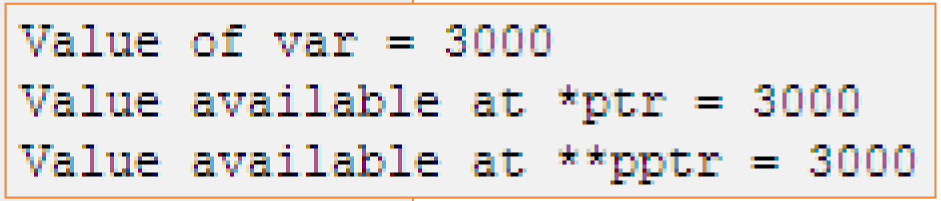
```
int **var;
```

# POINTER TO POINTER

- When a target value is indirectly pointed to by a pointer to a pointer, accessing that value requires that the asterisk operator be applied twice, as is shown below in the example:



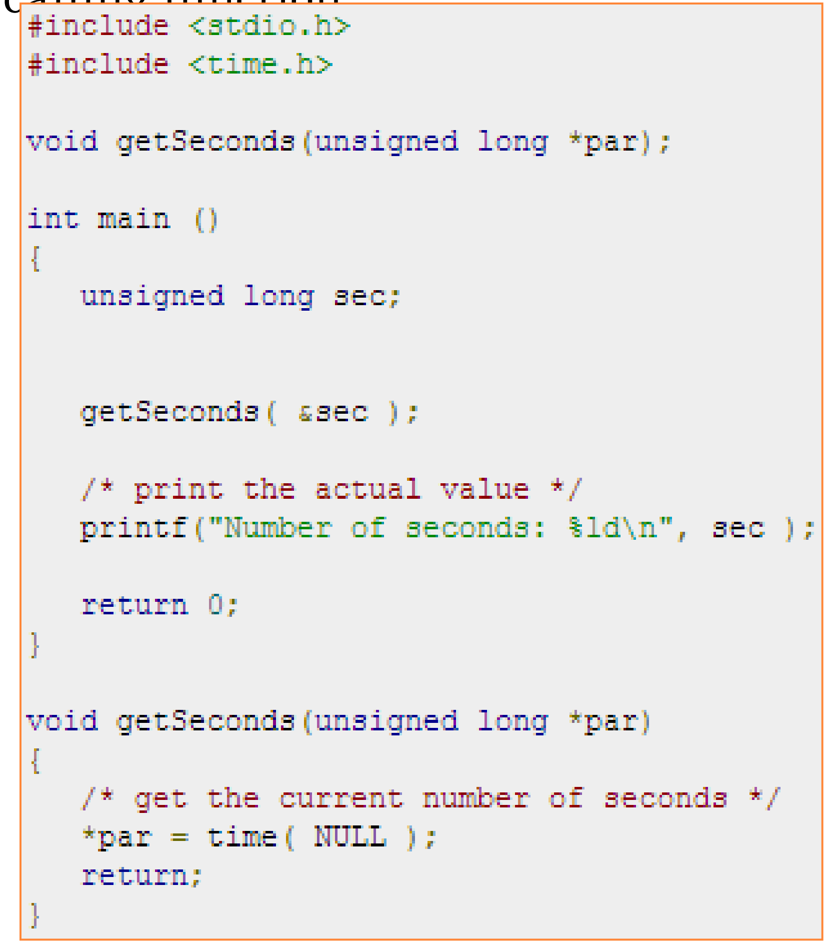
```c
#include <stdio.h>

int main ()
{
   int  var;
   int  *ptr;
   int  **pptr;

   var = 3000;

   /* take the address of var */
   ptr = &var;

   /* take the address of ptr using address of operator & */
   pptr = &ptr;

   /* take the value using pptr */
   printf("Value of var = %d\n", var );
   printf("Value available at *ptr = %d\n", *ptr );
   printf("Value available at **pptr = %d\n", **pptr);

   return 0;
}
```



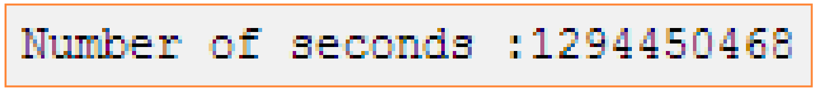
```
Value of var = 3000
Value available at *ptr = 3000
Value available at **pptr = 3000
```

# PASSING POINTERS TO FUNCTIONS

- C programming language allows you to pass a pointer to a function. To do so, simply declare the function parameter as a pointer type.
- Following is a simple example where we pass an unsigned long pointer to a function and change the value inside the function which reflects back in the calling function:



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

void getSeconds(unsigned long *par);

int main ()
{
   unsigned long sec;


   getSeconds( &sec );

   /* print the actual value */
   printf("Number of seconds: %ld\n", sec );

   return 0;
}

void getSeconds(unsigned long *par)
{
   /* get the current number of seconds */
   *par = time( NULL );
   return;
}
```



```
Number of seconds :1294450468
```

## PASSING POINTERS TO FUNCTIONS

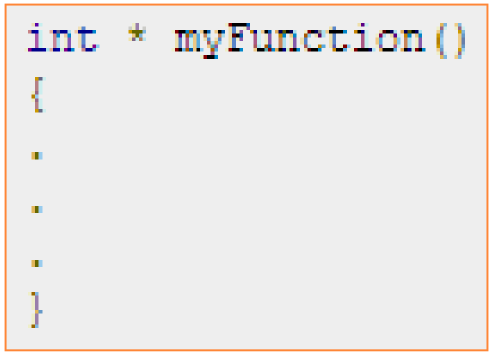
- The function, which can accept a pointer, can also accept an array as shown in the following example:

```
Average value is: 214.40000
```

```
#include <stdio.h>

/* function declaration */
double getAverage(int *arr, int size);

int main ()
{
   /* an int array with 5 elements */
   int balance[5] = {1000, 2, 3, 17, 50};
   double avg;

   /* pass pointer to the array as an argument */
   avg = getAverage( balance, 5 ) ;

   /* output the returned value  */
   printf("Average value is: %f\n", avg );

   return 0;
}

double getAverage(int *arr, int size)
{
  int    i, sum = 0;
  double avg;

  for (i = 0; i < size; ++i)
  {
    sum += arr[i];
  }

  avg = (double)sum / size;

  return avg;
}
```

## RETURN POINTER FROM FUNCTIONS

- C allows us to return a pointer from a function. To do so, you would have to declare a function returning a pointer as in the following example:



```
int * myFunction()
{
.
.
.
}
```

- Second point to remember is that, it is not good idea to return the address of a local variable to outside of the function so you would have to define the local variable as **static** variable.

# RETURN POINTER FROM FUNCTIONS

- Consider the following function which will generate 10 random numbers and return them using an array name which represents a pointer i.e., address of first array element.

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

/* function to generate and retrun random numbers. */
int * getRandom( )
{
   static int  r[10];
   int i;

   /* set the seed */
   srand( (unsigned)time( NULL ) );
   for ( i = 0; i < 10; ++i)
   {
      r[i] = rand();
      printf("%d\n", r[i] );
   }

   return r;
}

/* main function to call above defined function */
int main ()
{
   /* a pointer to an int */
   int *p;
   int i;

   p = getRandom();
   for ( i = 0; i < 10; i++ )
   {
       printf("*(p + [%d]) : %d\n", i, *(p + i) );
   }

   return 0;
}
```

```
1523198053
1187214107
1108300978
430494959
1421301276
930971084
123250484
106932140
1604461820
149169022
*(p + [0]) : 1523198053
*(p + [1]) : 1187214107
*(p + [2]) : 1108300978
*(p + [3]) : 430494959
*(p + [4]) : 1421301276
*(p + [5]) : 930971084
*(p + [6]) : 123250484
*(p + [7]) : 106932140
*(p + [8]) : 1604461820
*(p + [9]) : 149169022
```