#14 Pointers

By Saurabh Shukla | 2017©mysirg.com

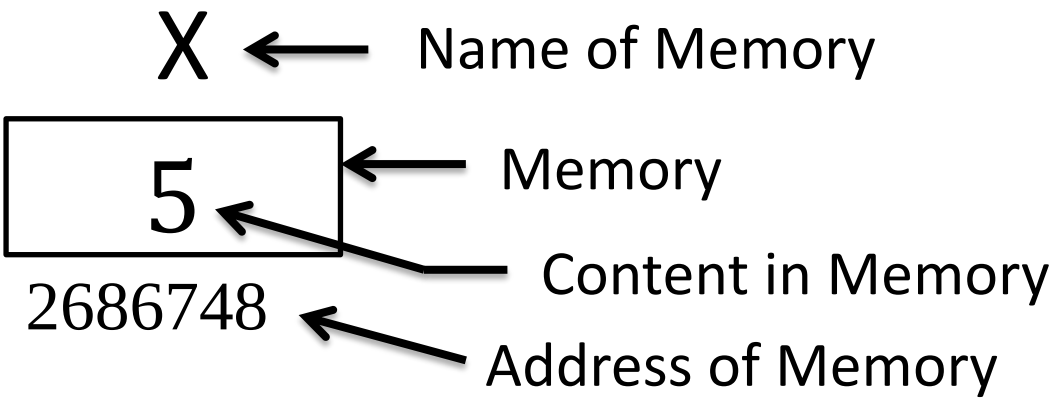
Pointers

Pointer is a variable which is used to hold address of another variable.

Consider the following statement

int x=5;

The above declaration statement tells compiler about the name of memory block (variable x), amount of memory occupied by the block (4 bytes) and type of content (integer constant 5).



Every byte has a logical reference number known as address of variable. In the above example address of x is 2686748. Though this address number could be anything and unpredictable to us, it is always in the range 0 to 4294967295.

Whatever would be the address of our variable following things are important:

- Addresses are always in the range 0 to 4294967295. (Target Platform dependent)
- Every byte will have some address number
- Address number is always an integer, even if the variable is not of type int.
- We cannot decide address number of any variable.
- Addresses are also known as references.

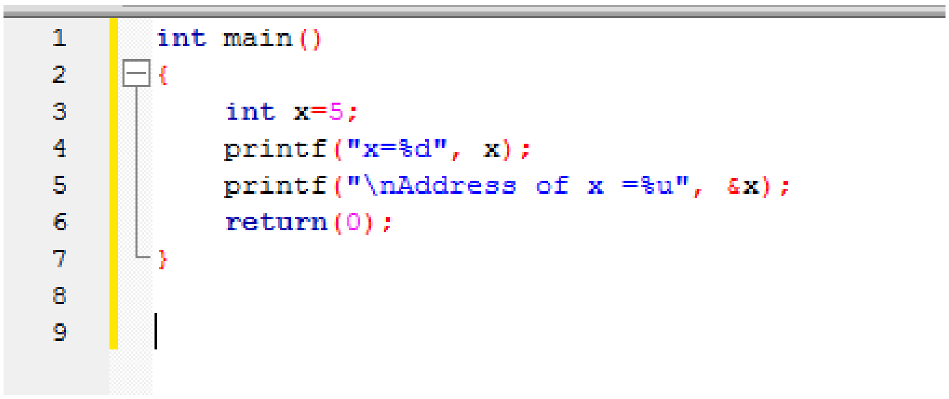
Address Operator

You are already use to of address of operator as you have been using it in scanf() function. Now let us examine its characteristics:

- & is called address of operator, also known as referencing operator.

- It is a unary operator, thus requires only one operand.
- Operand must be a variable.
- & returns the reference number of variable specified as the operand.

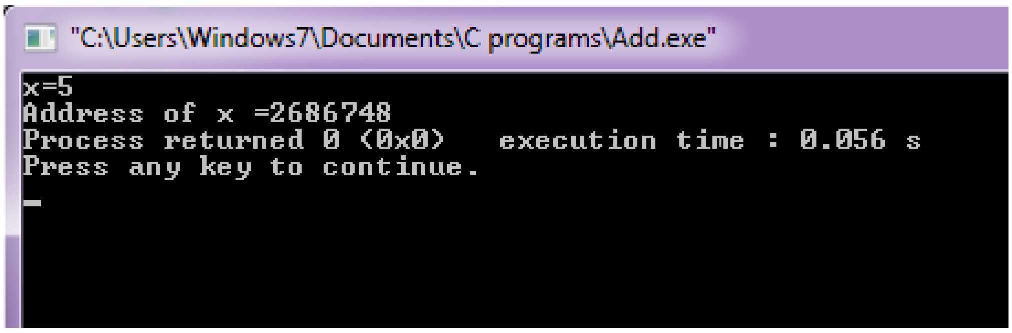
Example



```
int main()
{
    int x=5;
    printf("x=%d", x);
    printf("\nAddress of x =%u", &x);
    return(0);
}

```

Output of the above program



"C:\Users\Windows7\Documents\C programs\Add.exe"

```
x=5
Address of x =2686748
Process returned 0 (0x0)   execution time : 0.056 s
Press any key to continue.
```

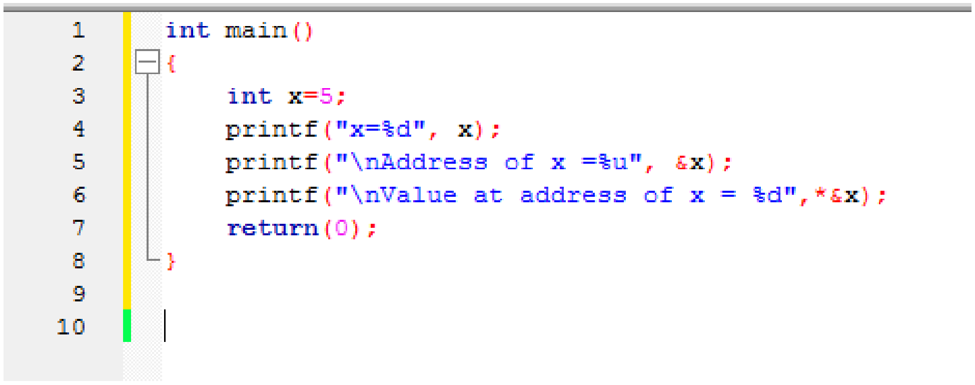
We can print address of a variable by using address of operator. First printf() is a usual statement printing value of the variable. Second printf() is used to print address of variable x. Since %d can print integers comes in the range -2,147,483,648 to 2,147,483,647, so we are using %u to match the range. %u can display integers in the range from 0 to 4294967295.

Indirection Operator

Another important operator is indirection operator (*).

- Indirection operator is also known as 'value at' operator or 'dereferencing operator'.
- It is also a unary operator that is single operand is needed.
- Operand must be a variable address.
- It gives a representation to the block whose address is specified in the operand.

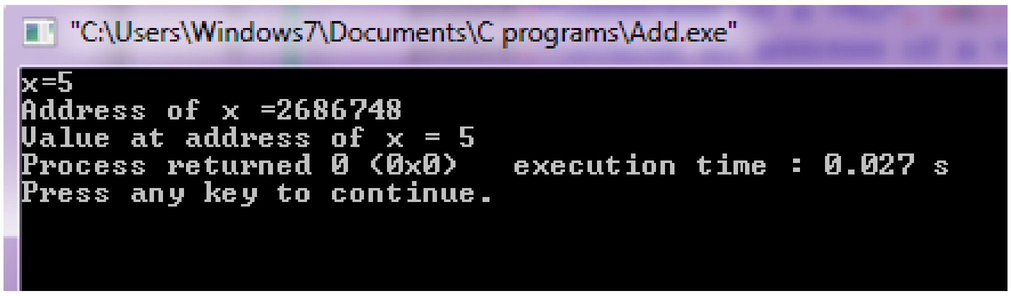
Example



```
int main()
{
    int x=5;
    printf("x=%d", x);
    printf("\nAddress of x =%u", &x);
    printf("\nValue at address of x = %d",*&x);
    return(0);
}

```

The output of the above program is:



"C:\Users\Windows7\Documents\C programs\Add.exe"

```
x=5
Address of x =2686748
Value at address of x = 5
Process returned 0 (0x0)   execution time : 0.027 s
Press any key to continue.
```

In the last printf() value at 2686748 gets printed. Since &x represents address of x that is 2686748, so the operand of indirection operator (*) is 2686748. As a result *&x represents block x.

Important Note:

- We cannot assign anything to block by just writing address to the left of assignment operator.
  &x=25; //Error
- &x is not a variable. It is a way to represent address of variable x.
- &x in itself is a constant and we cannot assign anything to constant.

Declaration of Pointer variable

However we cannot assign anything to &x, we can assign address of x (&x) into some other variable. Let’s say variable is j then we can write

j =&x; // Valid statement

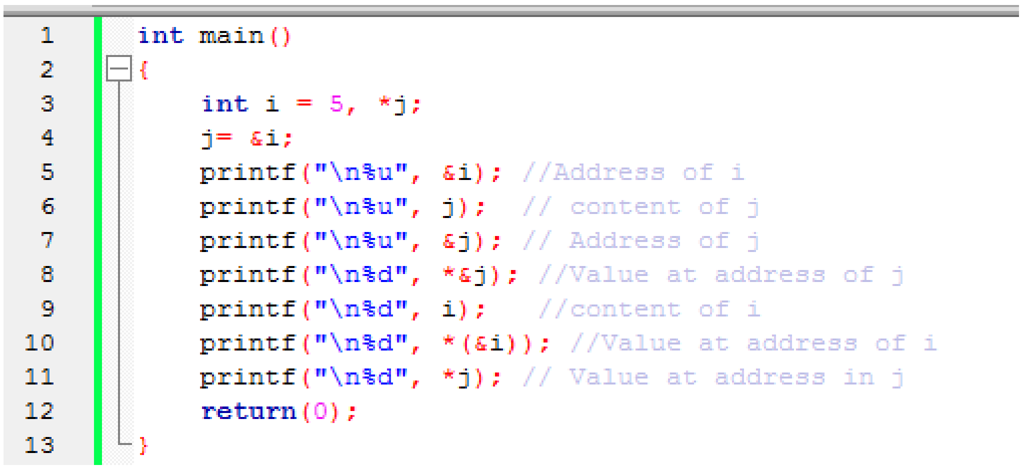
This is the correct way as variable is in the left side of assignment operator.

Here j is a special variable, it contains address of another variable, it must be declared in a special way.

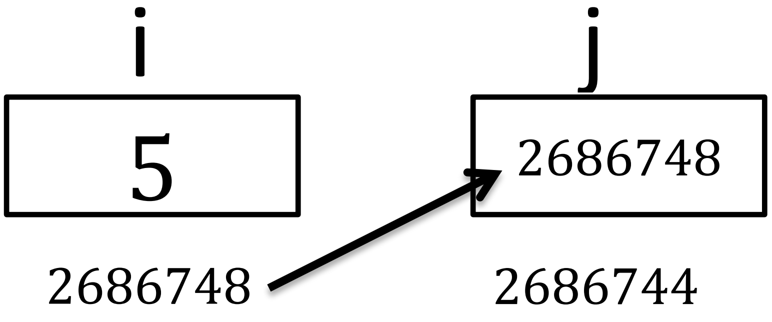
int *j;

Asterisk (*) symbol before j in the declaration statement tells the compiler that this is a special variable meant only to store address of another variable. This special variable is called pointer.

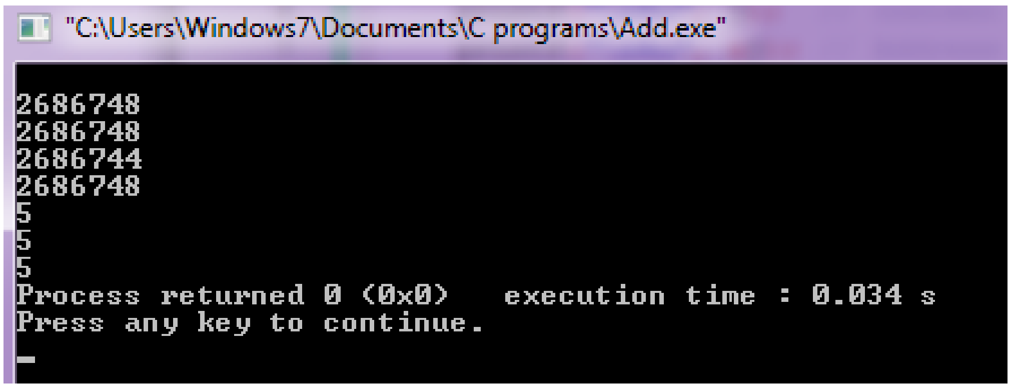
Example



```
int main()
{
    int i = 5, *j;
    j= &i;
    printf("\n%u", &i); //Address of i
    printf("\n%u", j);  // content of j
    printf("\n%u", &j); // Address of j
    printf("\n%d", *&j); //Value at address of j
    printf("\n%d", i);   //content of i
    printf("\n%d", *(&i)); //Value at address of i
    printf("\n%d", *j); // Value at address in j
    return(0);
}
```



Output is



"C:\Users\Windows7\Documents\C programs\Add.exe"

```
2686748
2686748
2686744
2686748
5
5
5
Process returned 0 (0x0)   execution time : 0.034 s
Press any key to continue.
```

Size of pointer

Since pointer contains only address of another variable, they consume 4 bytes in memory. Whatever is the type of pointer, it consumes 4 bytes in memory.

Wild Pointer

When pointer is declared without initialization, it contains garbage value. This garbage is interpreted as address because it is stored in pointer variable. Obviously this garbage represents invalid address as we haven't made any reservation to that memory location. Such pointers are called un-initialized pointers or **wild pointers**.

An important point to understand

```
float *s;
```

- Above declaration does not mean that it is going to store floating point value. It actually means 's' is going to contain address of floating point value.
- 's' will store an address of some float block which contains floating point value.
- We can call such pointer as float pointer.

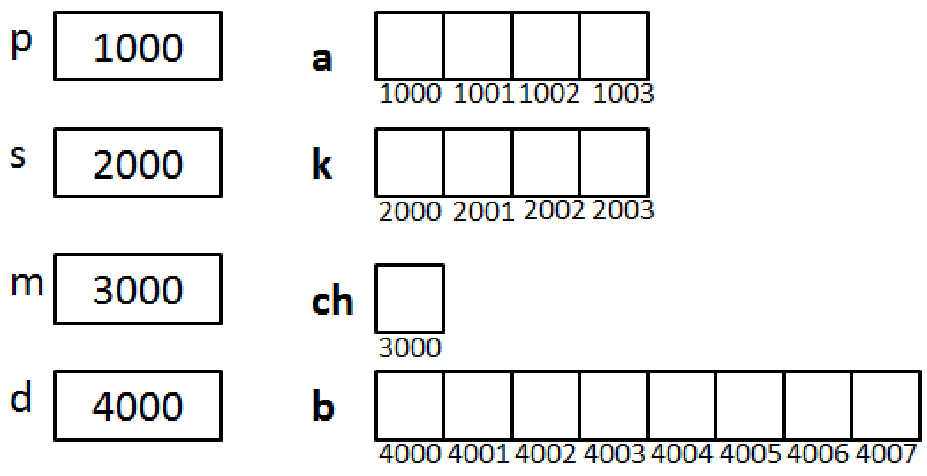
Concept of base address

Address of the first byte of any memory block is known as base address. Size of variable depends on its type. Each byte will get address. It is possible to have multiple address of a variable, as variable can be of multiple bytes.

Consider the following declarations:

```
int *p, a;
float *s, k;
char *m, ch;
double *d, b;
p=&a;
s=&k;
m=&ch;
d=&b;
```

Now we have four pointers, each of different type. Each of the pointers consumes 4 bytes in memory. p contains address of int block 'a', s contains address of float block 'k, m contains address of char block 'ch' and d contains address of double block 'b'.



It is important to note that operating system allocates address to each byte. It is nothing but position number of bytes in program's memory. In our example 'a' is a variable of type int whose address is 1000 (assume), but this is the address of first byte of int block. 1001, 1002 and 1003 are the addresses of subsequent bytes. Similarly, 2000 is the address of first byte of float block. Since float block is of 4 bytes, each byte has different address in a sequential manner.

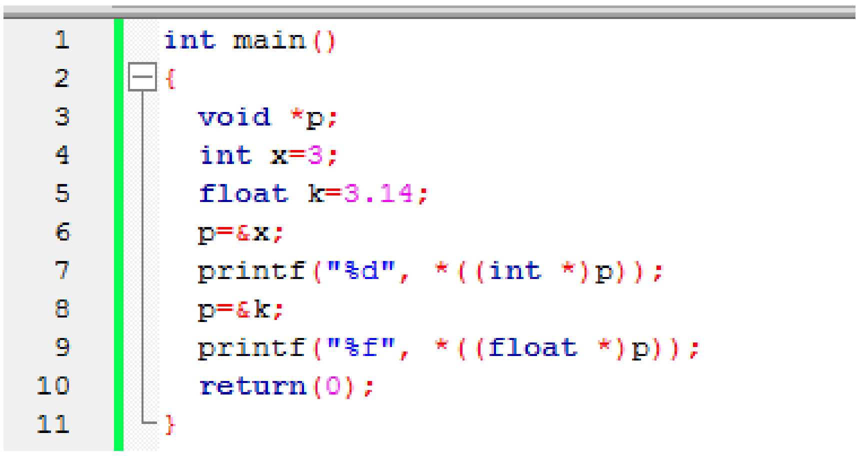
Address of the first byte of the block of any type is known as base address of that block.

Pointer variable can store one address at a time and it is always base address. To access any block through pointer requires address of remaining bytes of the block. This is possible because of two reasons. First, addresses of bytes of a block are always in a sequence. Second, the type of pointer, which stores base address.

If the pointer is declared as float pointer (consider 's' in above example), it knows about 3 more addresses to be accessed along with base address. Similarly, if the pointer is declared as double pointer (consider 'd' in above example), it is clear that 7 more addresses to be accessed along with base address.

Void pointer

Pointers declared with keyword void are void pointers or generic pointers. Since type of pointer is void, pointer cannot access block, whose address it holds, by just de-referencing pointer. As we studied in the above section, pointers only hold base address and type of pointer decides how many more bytes need to be accessed. If the type of pointer is void then we cannot know how many more bytes need to be accessed. Every time when we access block through pointer, typecasting is needful.

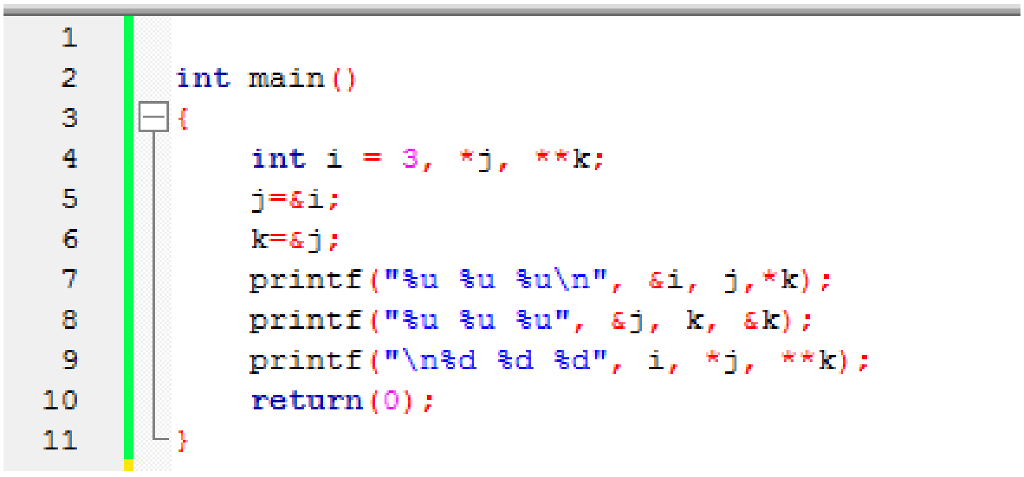


```c
int main()
{
   void *p;
   int x=3;
   float k=3.14;
   p=&x;
   printf("%d", *((int *)p));
   p=&k;
   printf("%f", *((float *)p));
   return(0);
}
```

Extended Concept of pointers

Pointer is a variable which contains address of another variable. Now this variable itself could be another pointer. Thus we now have a pointer, which contains another pointer address

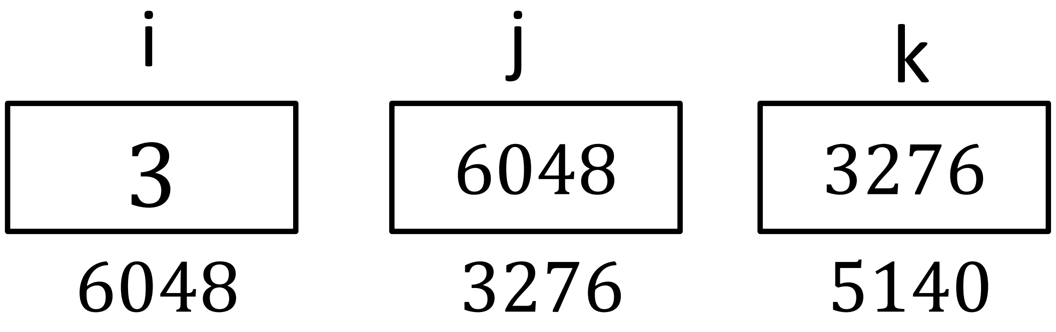
Consider the following example



```c

int main()
{
     int i = 3, *j, **k;
     j=&i;
     k=&j;
     printf("%u %u %u\n", &i, j,*k);
     printf("%u %u %u", &j, k, &k);
     printf("\n%d %d %d", i, *j, **k);
     return(0);
}
```

Assume the following memory addresses for simplicity



Output should be:

```
6048 6048 6048

3276 3276 5140

3 3 3
```

Pointer Jargons

```
int i;      // i is an int

int *j;    // j is a pointer to an int

int **k;  // k is a pointer to a pointer to an int
```

Pointer’s Arithmetic

Although C language is not very strict about compatibility issues between various type of data, but programmer has to take special care about type conversions. It is also needfull in forthcoming programming languages.

Rule 1: Take care for the compatibility

Example

```
int main()
{
        int i=3;
        char *j;
        j = &i;    //wrong as incompatible assignment
        printf(“%d %u”, i, &i);
}
```

(Error message depends on compiler most of the C compilers do not show any error)

Example

```
int main()
{
        long int i=3;
        int  *j;
        j = &i;    //Wrong as incompatible assignment
        printf(“%d %u”, i, &i);
        return(0);
}
```

Example

```
int main()
{
        int k, i=3, *j;
        j = &i;
        k = j;      //Wrong as incompatible assignment
        printf("%u", k);
        return(0);
}
```

Example

```
int main()
{
        int k, i=3, *j;
        j = &i;
        k = *j;
        printf("%d", k);
        return(0);
}
```

The output is

```
3
```

Example

```
int main()
{
        int k, i=3, *j;
        j = &i;
        k = *j + *j;
        printf("%d", k);
        return(0);
}
```

The output is

```
6
```

In this program *j would means ‘value at address contained in j’ and it is 3.

Rule 2: We cannot add two addresses.

Example

```
int main()
{
        int *k, i=3, *j;
        j = &i;
        k = j + j;
        printf("%d", k);
        return(0);
}
```

Error: Invalid pointer addition.

Similarly, we cannot multiply, divide two addresses.

Rule 3: We cannot multiply scalar value to an address. Similarly we cannot divide an address with scalar value

```
int main()
{
        int *k, i=3, *j;
        j = &i;
        k = j * 5;  //Error, cannot multiply address and scalar value
        printf("%d", k);
        return(0);
}
```

Rule 4: Adding 1 to the pointer gives address of the immediately next block of the same type. Similarly subtracting 1 from the pointer gives address of the previous block of the same type

```
int main()
{
        int *k, i=3, *j;  //assume address of i is 1000
        j = &i;  // the value stored in j is 1000
        k = j + 1; // j+1 is adding 1 to the address stored in j that is 1000, the result is 1004
        printf("%d", k);  //Output is 1004
        return(0);
}
```

References:

YouTube video links

- Lecture 14 Pointers in C part-1
    - https://youtu.be/d26HpQ2DKUo?list=PL7ersPsTyYt2Q-SqZxTA1D-melSfqBRMW
- Lecture 14 Pointers in C part-2
    - https://youtu.be/8aBNPSIziuw?list=PL7ersPsTyYt2Q-SqZxTA1D-melSfqBRMW

Exercise

1) Write a program to print address of variables declared in the program.
2) Write a program to print the size of the pointer variable of any type.
3) Can we store address of int variable in char pointer?
4) Can we add two addresses?
5) Can we subtract two addresses?
6) What will be the value stored in double pointer after adding 3 to it? Assume initial value stored in double pointer is 1000.