RAMAIAH
Institute of Technology

Unit -5
Pointers

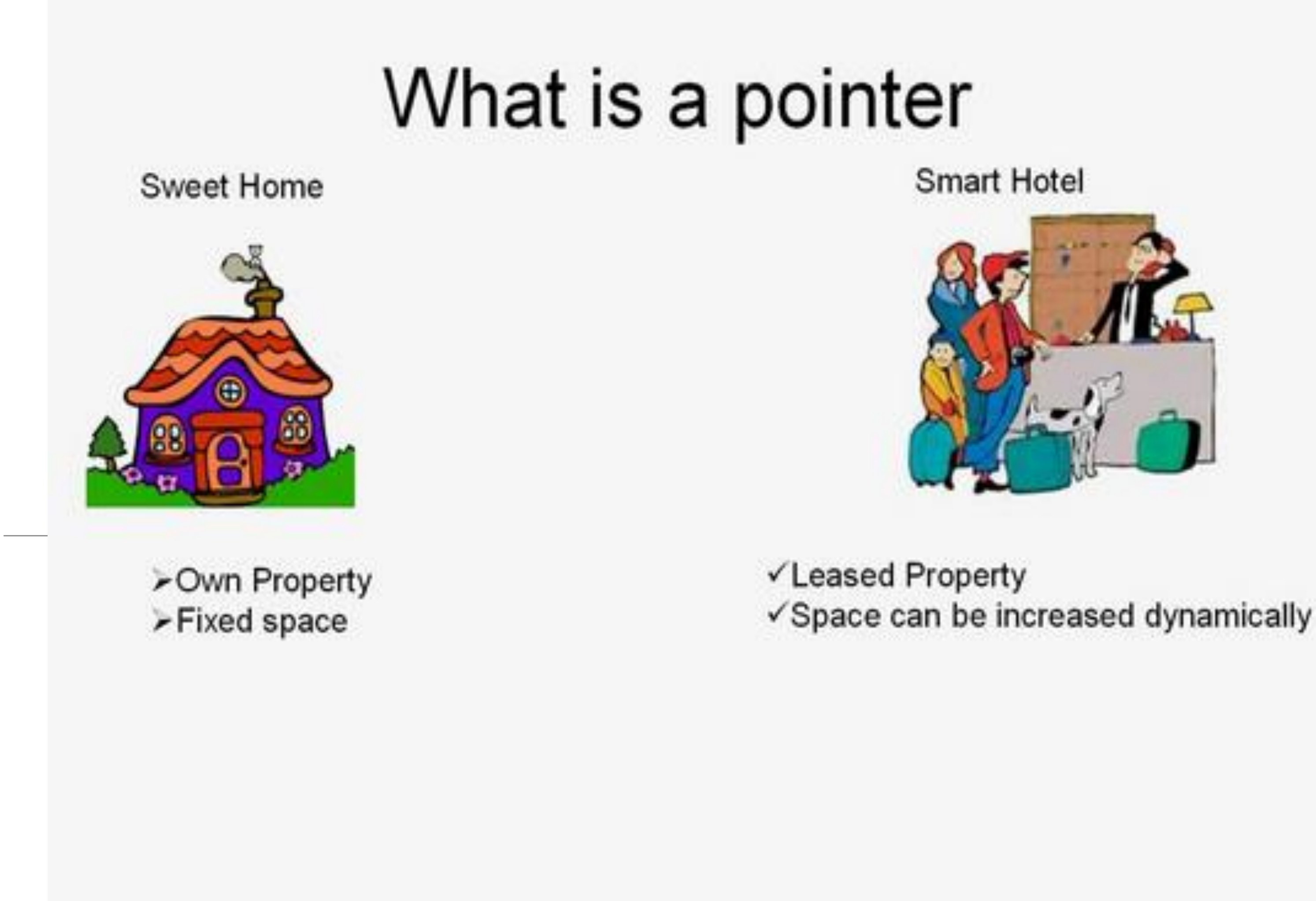
PREPARED BY-

HANUMANTHARAJU R

ASSISTANT PROFESSOR

DEPT.OF CSE, MSRIT

Pointers in C



What is a pointer

Sweet Home

- ➢Own Property
- ➢Fixed space

Smart Hotel

- ✓Leased Property
- ✓Space can be increased dynamically

Why Pointers?

They allow you to **refer to large data structures** in a compact way.

They facilitate **sharing** between different parts of programs.

They make it **possible to get new memory dynamically** as your program is running.

They make it **easy to represent relationships among data** items.

POINTER CAUTION

They are a powerful low-level device.

Undisciplined use can be confusing and thus the source of subtle, hard-to-find bugs.

- Program crashes
- Memory leaks
- Unpredictable results

Pointer

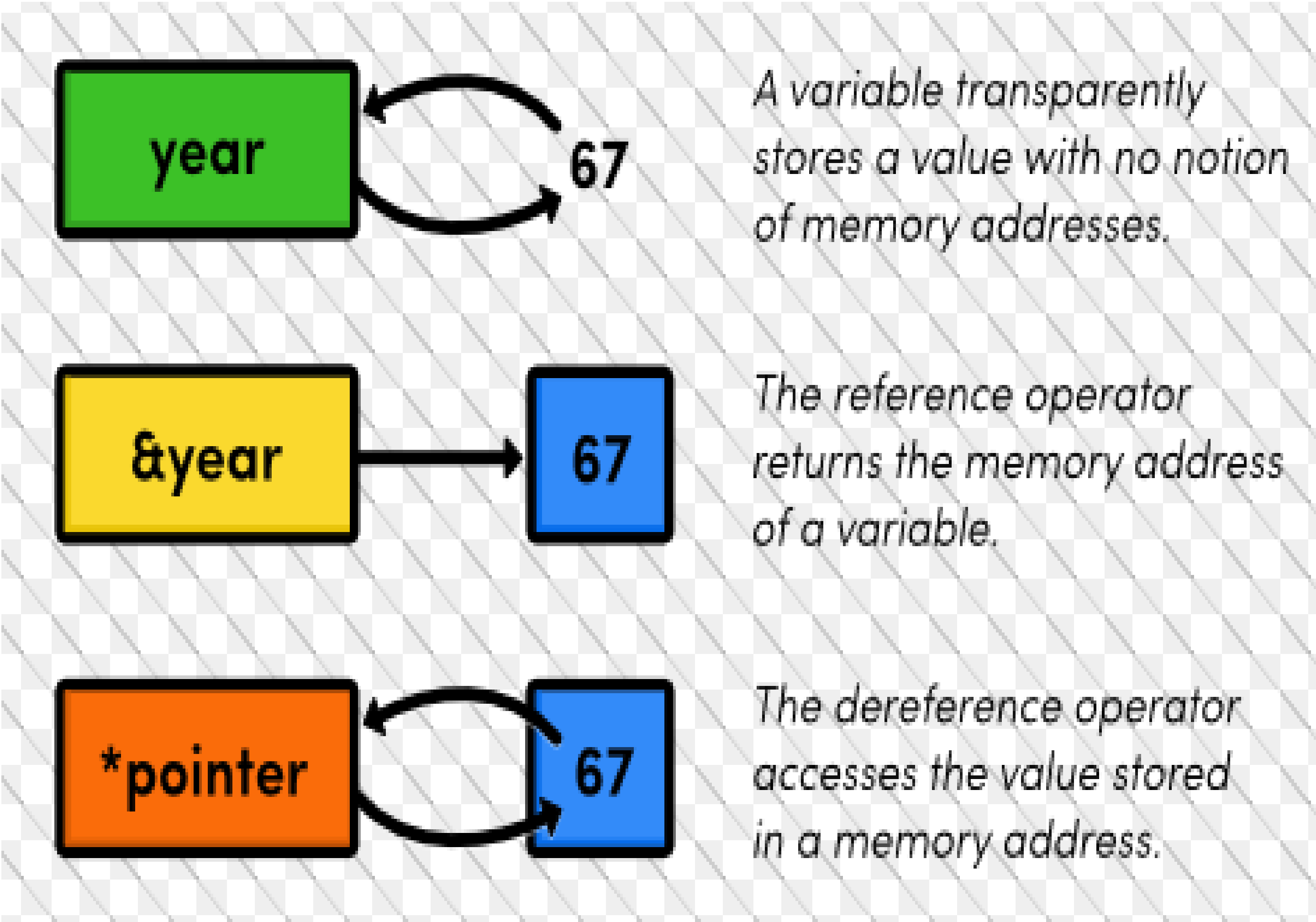
- A pointer is a variable which contains the address in memory of another variable.
- The unary operator & gives the —address of a variable".
- The indirection or dereference operator * gives the —contents of an object pointed to by a pointer".
- **IMPORTANT: When a pointer is declared it does not point anywhere. You must set it to point somewhere before you use it. So,**
  - int *ip;
  - *ip = 100;
    - will generate an **error (program crash!!).**

C POINTER VARIABLES

- To declare a pointer variable, we must do two things
  - Use the *** (star)** character to indicate that the variable being defined is a pointer type.
  - Indicate the type of variable to which the pointer will point (the pointee).

General declaration of a pointer

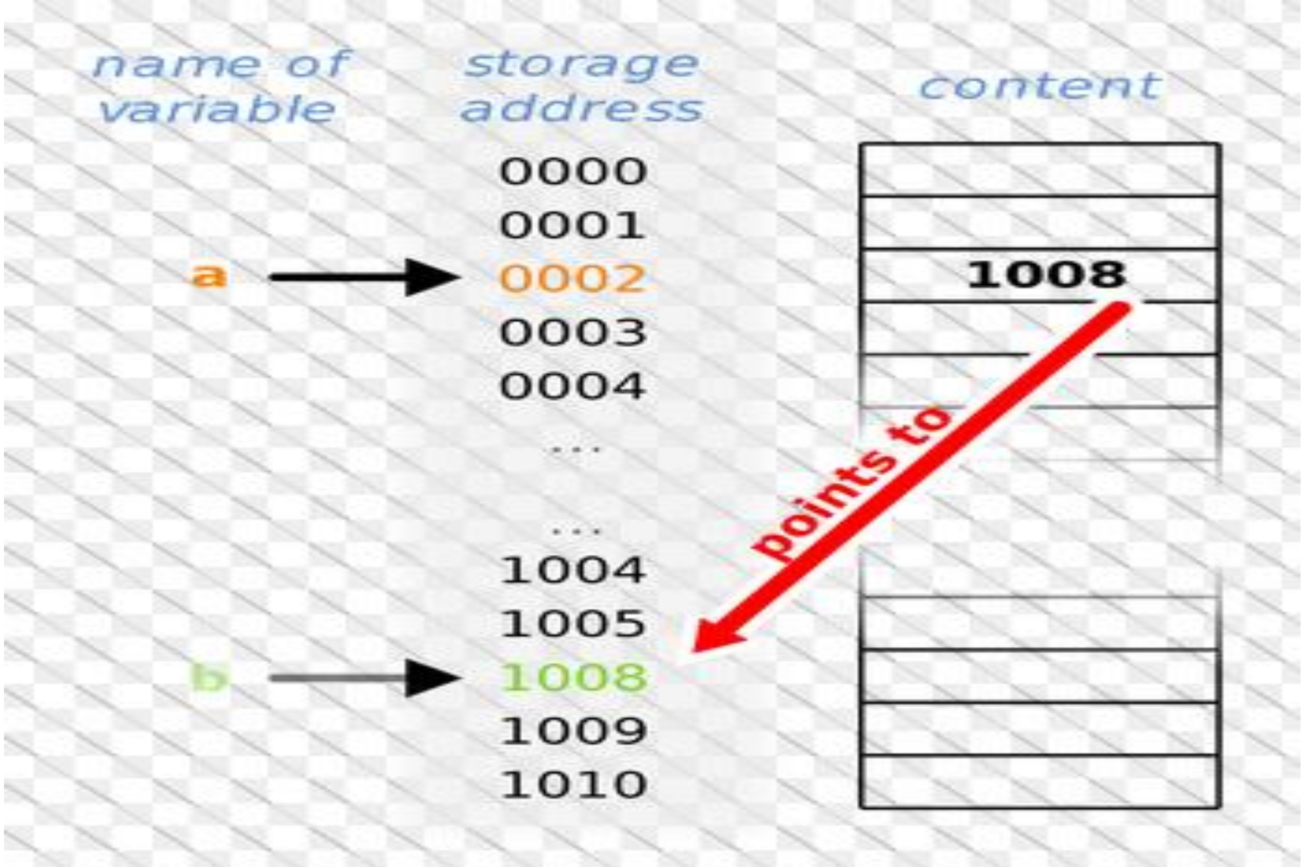
```
type *nameOfPointer;
```

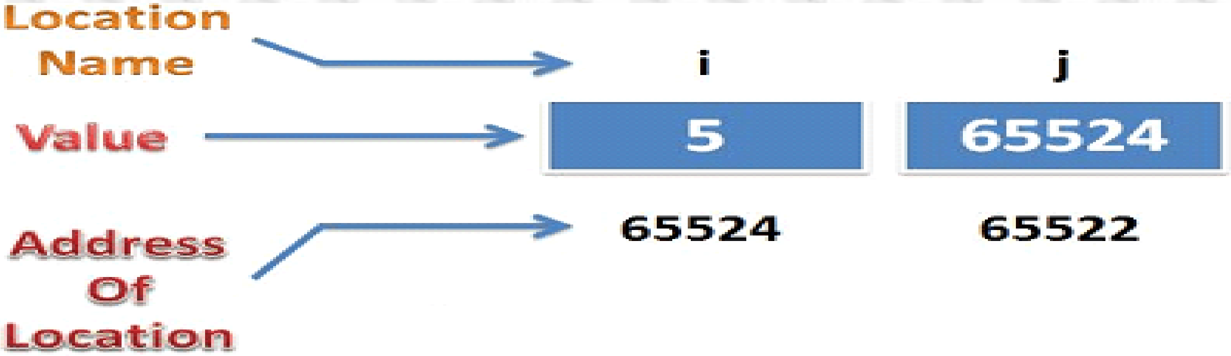


A variable transparently stores a value with no notion of memory addresses.

The reference operator returns the memory address of a variable.

The dereference operator accesses the value stored in a memory address.





Location Name

i

j

Value

5

65524

Address Of Location

65524

65522

POINTER DECLARATION

- The declaration

```
int *intPtr;
```

defines the variable `intPtr` to be a pointer to a variable of type `int`.

Read this declaration as

- "`intPtr` is a pointer to an `int`", or equivalently
- "`*intPtr` is an `int`"

Caution -- Be careful when defining multiple variables on the same line. In this definition

```
int *intPtr, intPtr2;
```

`intPtr` is a pointer to an `int`, but `intPtr2` is not!

Declaration of Pointer Variables (Cont ..)

Whitespace doesn’t matter and each of the following will declare **ptr** as a pointer (to a **float**) variable and **data** as a **float** variable

```
float *ptr, data;
 float* ptr, data;
 float (*ptr), data;
 float data, *ptr;
```

ADDRESSING CONCEPT

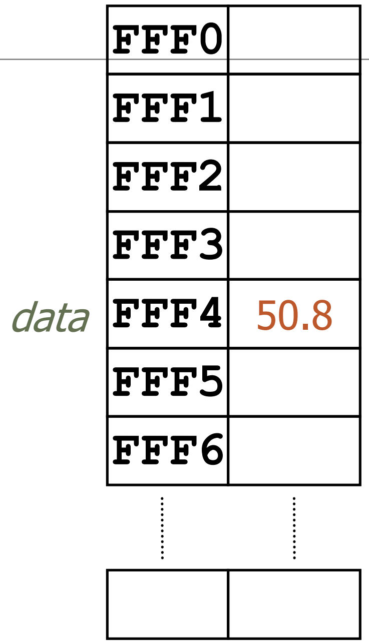
Pointer stores the **address** of another entity

It **refers** to a memory location

```
int i = 5;
int *ptr;                  /* declare a pointer variable */
ptr = &i;                  /* store address-of i to ptr */
printf(“*ptr = %d\n”, *ptr);  /* refer to referee of ptr */
```

Assignment of Pointer Variables (Cont ..)

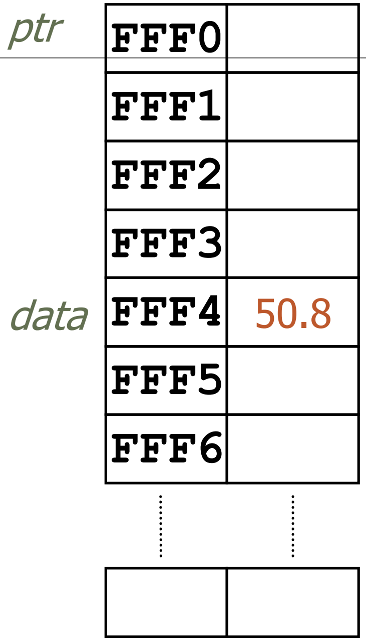
```
float data = 50.8;
float *ptr;
ptr = &data;
```



	FFF0	
	FFF1	
	FFF2	
	FFF3	
data	FFF4	50.8
	FFF5	
	FFF6	
	⋮	⋮

Assignment of Pointer Variables (Cont ..)

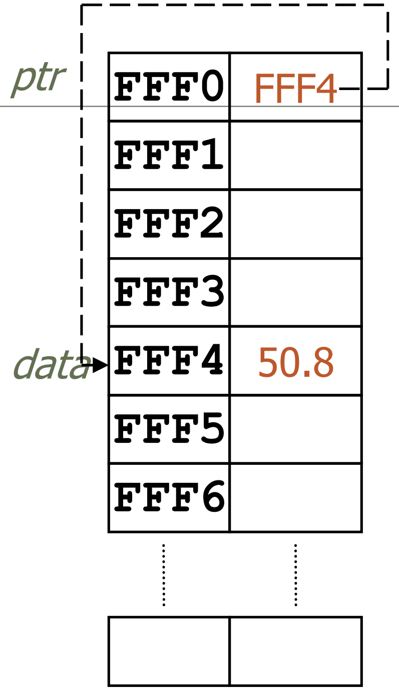
```
float data = 50.8;
float *ptr;
ptr = &data;
```



ptr	FFF0	
	FFF1	
	FFF2	
	FFF3	
data	FFF4	50.8
	FFF5	
	FFF6	

Assignment of Pointer Variables (Cont ..)

```
float data = 50.8;
float *ptr;
ptr = &data;
```



Assignment of Pointer Variables

- Don’t try to assign a specific integer value to a pointer variable since it can be disastrous
  - `float *ptr;`
  - `ptr = 120;`

- You cannot assign the address of one type of variable to a pointer variable of another type even though they are both integrals

```
int data = 50;
float *ptr;
ptr = &data;
```

POINTER EXAMPLES

```
int x = 1, y = 2, z[10];

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

ip = &x;     /* ip points to (contains the memory address of) x */

y = *ip;     /* y is now 1, indirectly copied from x using ip */

*ip = 0;     /* x is now 0 */

ip = &z[5]; /* ip now points to z[5] */
```

NULL

NULL is a special value which may be assigned to a pointer

NULL indicates that this pointer does not point to any variable (there is no pointee)

Often used when pointers are declared

```
int *pInt = NULL;
```

Often used as the return type of functions that return a pointer to indicate function failure

```
int *myPtr;
myPtr = myFunction( );
if (myPtr == NULL){
  /* something bad happened */
}
```

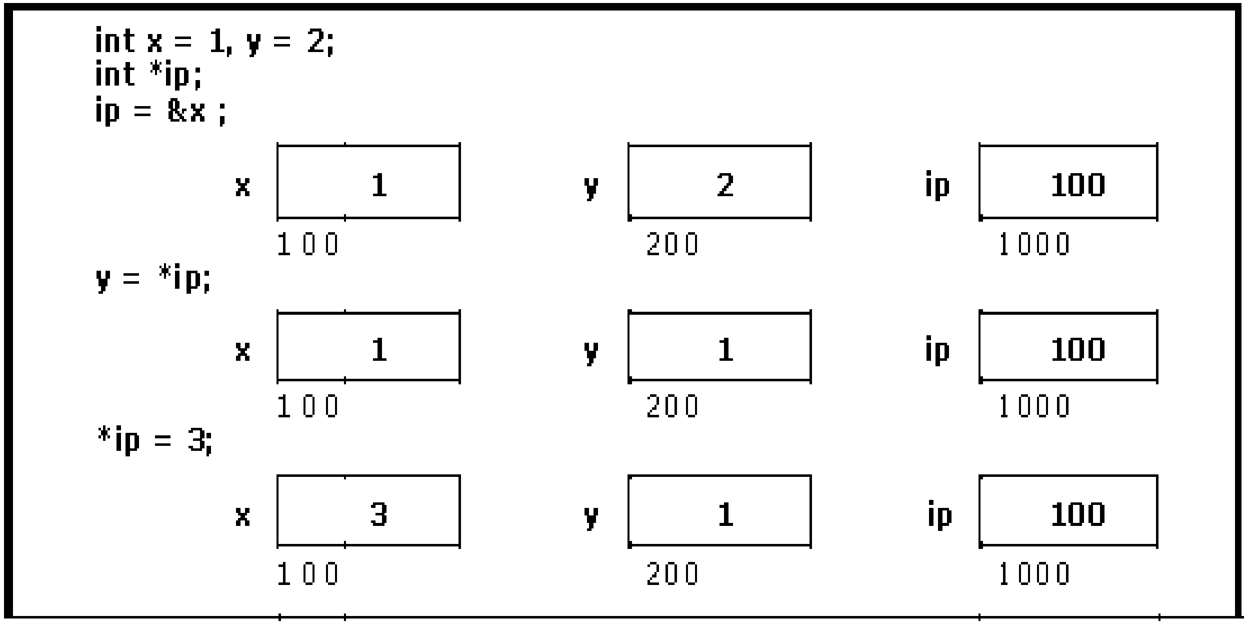
Dereferencing a pointer whose value is NULL will result in program termination.

Pointers Example

```
int i = 5, j = 10;
int *ptr;
int **pptr;
ptr = &i;
pptr = &ptr;
*ptr = 3;
**pptr = 7;
ptr = &j;
**pptr = 9;
*pptr = &i;
*ptr = -2;
```

Pointer example

```
#include <stdio.h>
main()
{
int x = 1, y = 2;
int *ip;
ip = &x;
y = *ip;
*ip = 3;
}
```

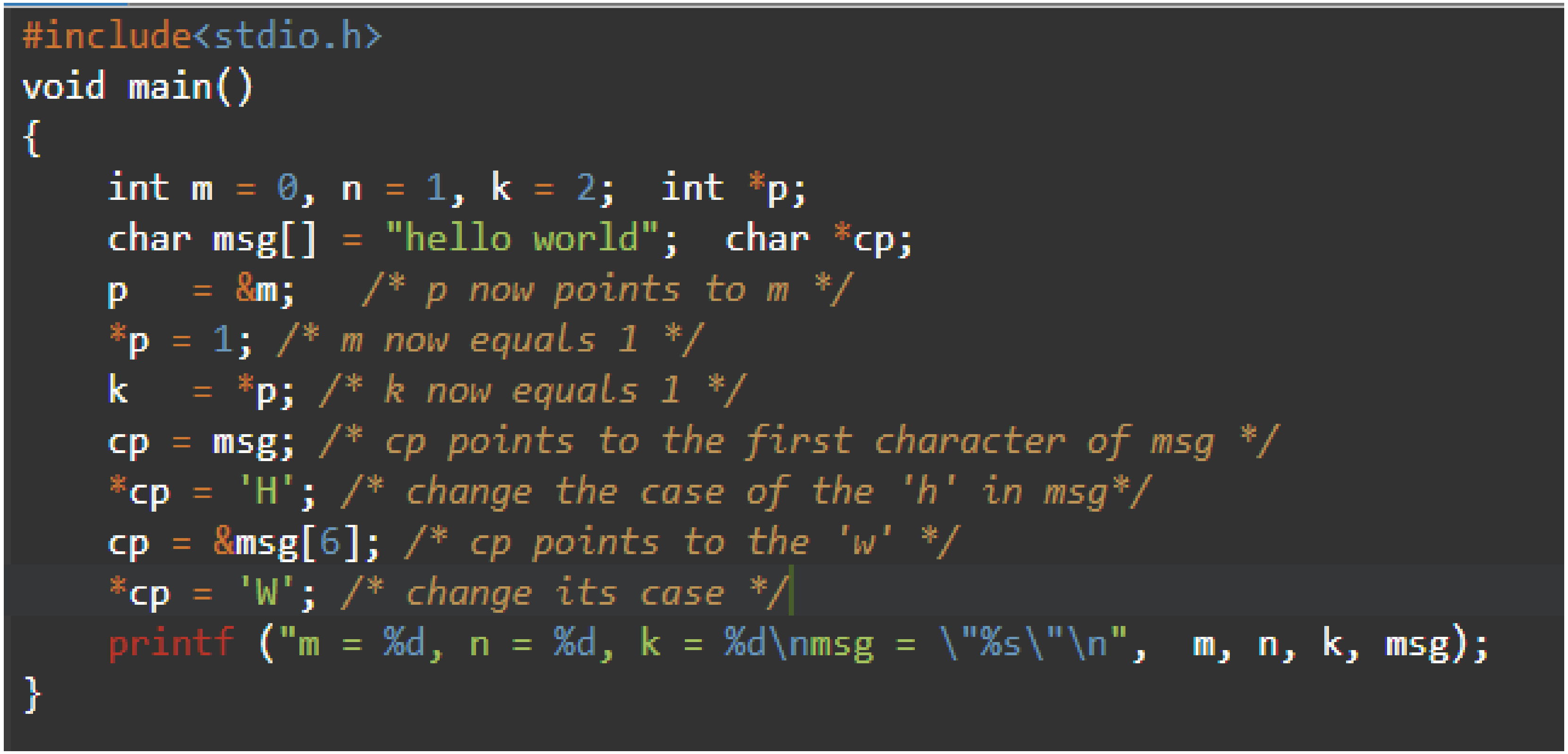


```
int x = 1, y = 2;
int *ip;
ip = &x ;
```

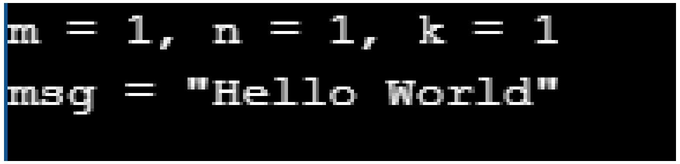
```
y = *ip;
```

```
*ip = 3;
```

Pointer Example

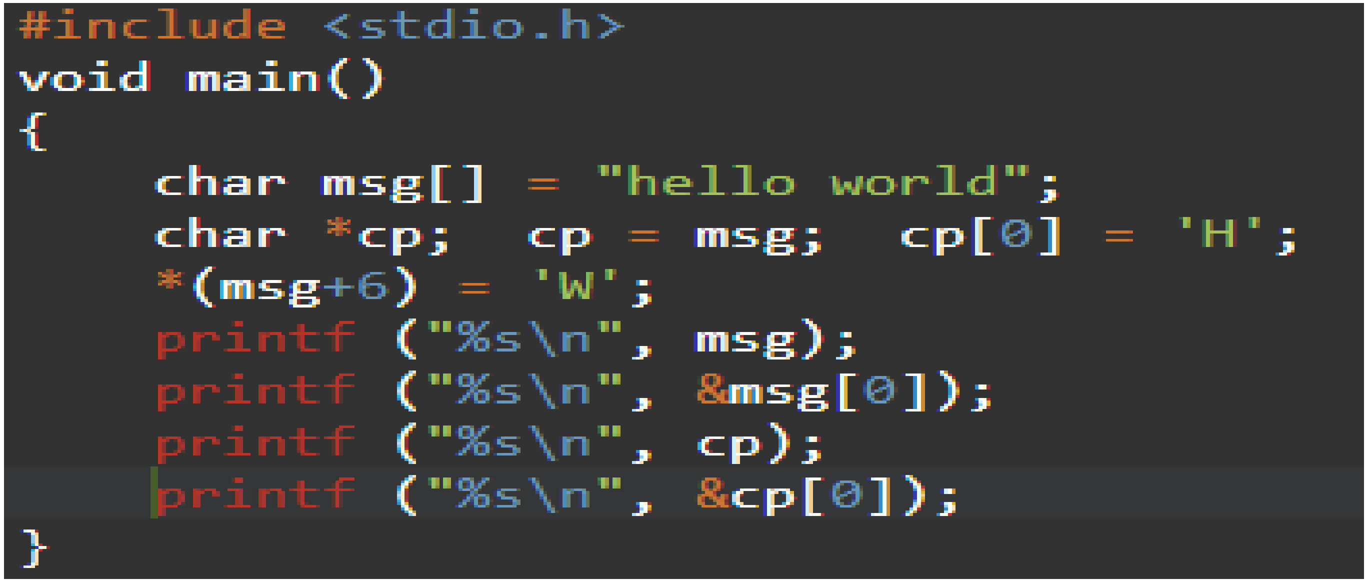


```
#include<stdio.h>
void main()
{
    int m = 0, n = 1, k = 2;  int *p;
    char msg[] = "hello world";  char *cp;
    p   = &m;   /* p now points to m */
    *p = 1; /* m now equals 1 */
    k   = *p; /* k now equals 1 */
    cp = msg; /* cp points to the first character of msg */
    *cp = 'H'; /* change the case of the 'h' in msg*/
    cp = &msg[6]; /* cp points to the 'w' */
    *cp = 'W'; /* change its case */
    printf ("m = %d, n = %d, k = %d\nmsg = \"%s\"\n",  m, n, k, msg);
}
```

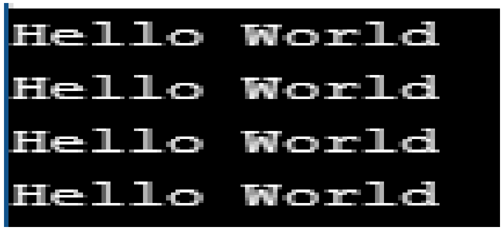


```
m = 1, n = 1, k = 1
msg = "Hello World"
```

Pointer example



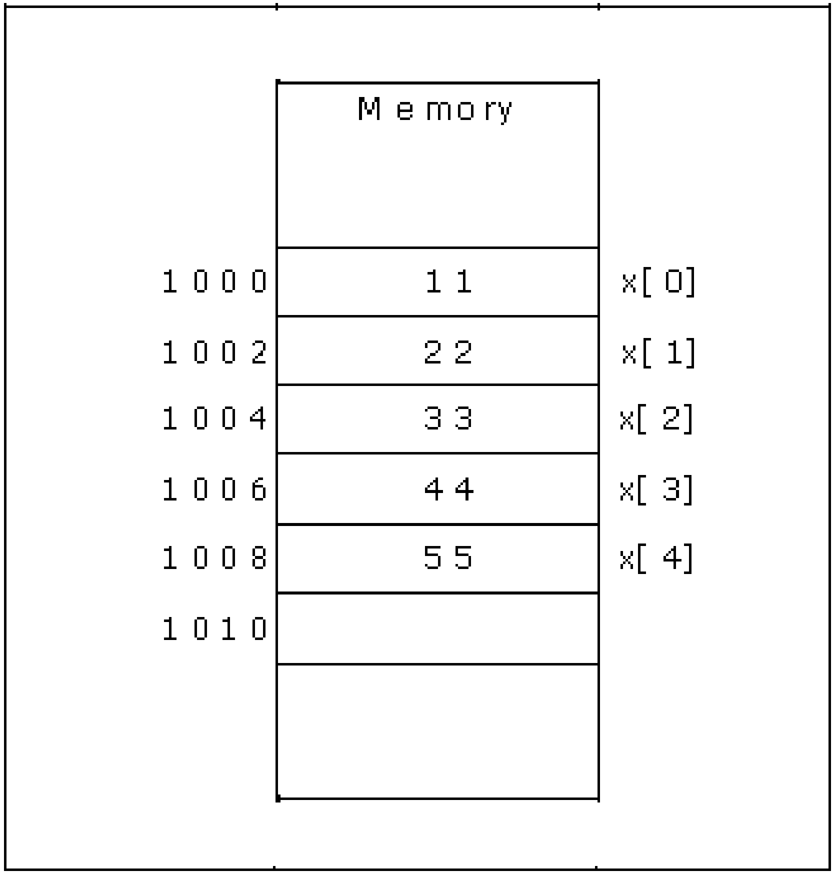
```
#include <stdio.h>
void main()
{
    char msg[] = "hello world";
    char *cp;  cp = msg;  cp[0] = 'H';
    *(msg+6) = 'W';
    printf ("%s\n", msg);
    printf ("%s\n", &msg[0]);
    printf ("%s\n", cp);
    printf ("%s\n", &cp[0]);
}
```



```
Hello World
Hello World
Hello World
Hello World
```

Pointers and Arrays

```
int x[5] = {11, 22, 33, 44, 55};
int *p = x;
```



	Memory	
1000	11	x[0]
1002	22	x[1]
1004	33	x[2]
1006	44	x[3]
1008	55	x[4]
1010		

Pointers and Arrays

int x[5] = {11, 12, 13, 14, 15};

int *p = x;

- ***p++ :** The increment ++ operator has a higher priority than the indirection operator * .
  - Therefore p is increment first. The new value in p is then 1002 and the content at this address is 20.
- ***(p++):** is same as *p++.
- **(*p)++:** *p which is content at address 1000 (i.e. 10) is incremented. **Therefore (*p)++ is 12.**

Pointers and Arrays example

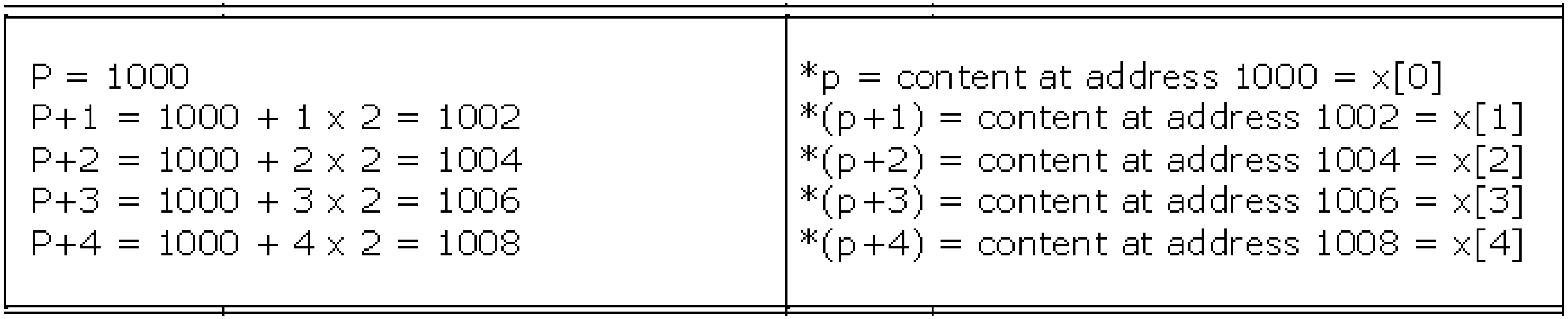
```
#include <stdio.h>
main()
{
int x[5] = {11, 22, 33, 44, 55};
int *p = x, i;                    /* p=&x[0] = address of the first element */
for (i = 0; i < 5; i++)
{
printf (“\n x[%d] = %d”, i, *p); /* increment the address*/
p++;
}
}
```

Output:

```
x [0] = 11 x [1] = 22 x [2] = 33 x [3] =
44 x [4] = 55
```

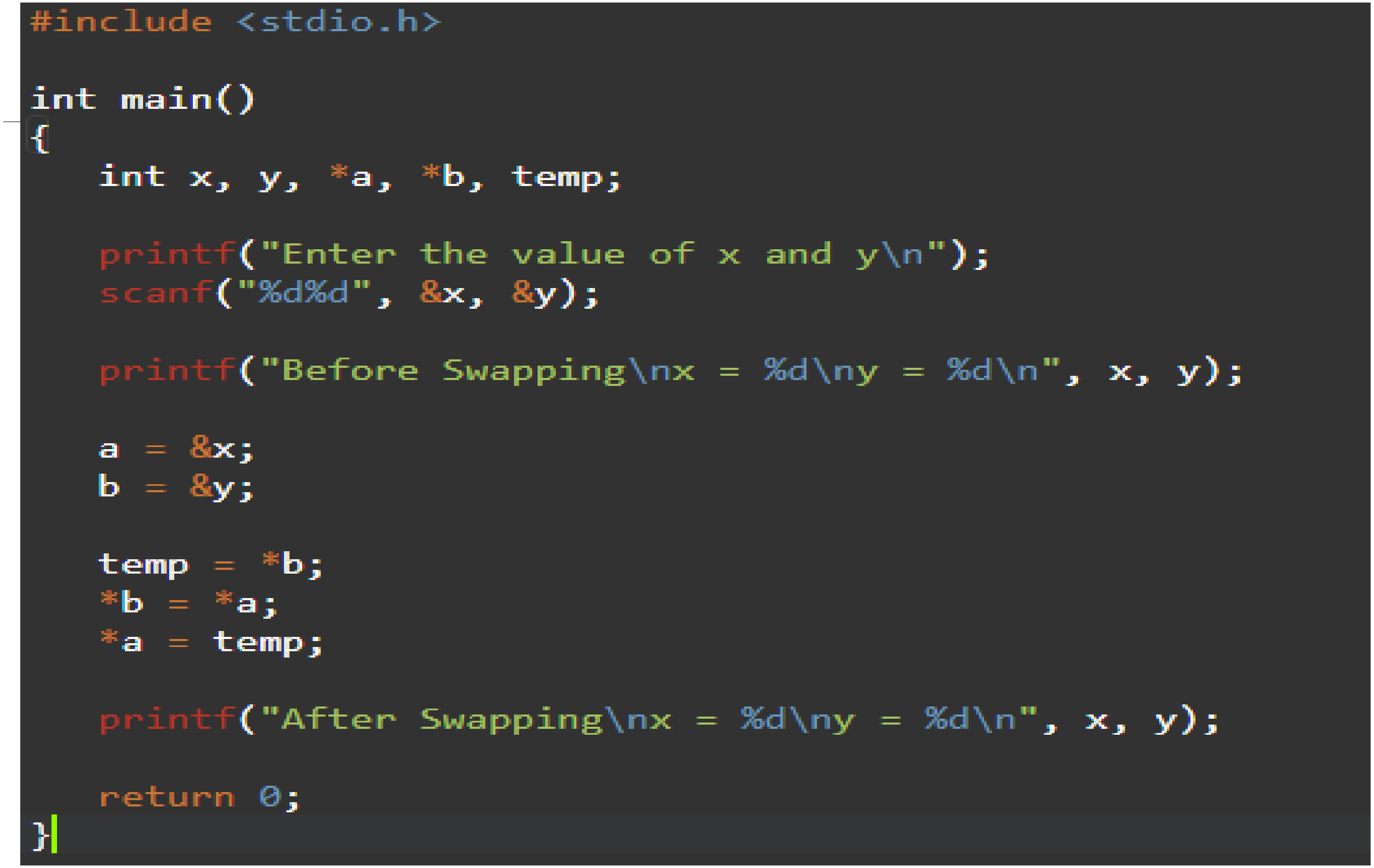
Pointers and Arrays example

```
int x[5] = {11, 22, 33, 44, 55};
int *p = x, i;
```



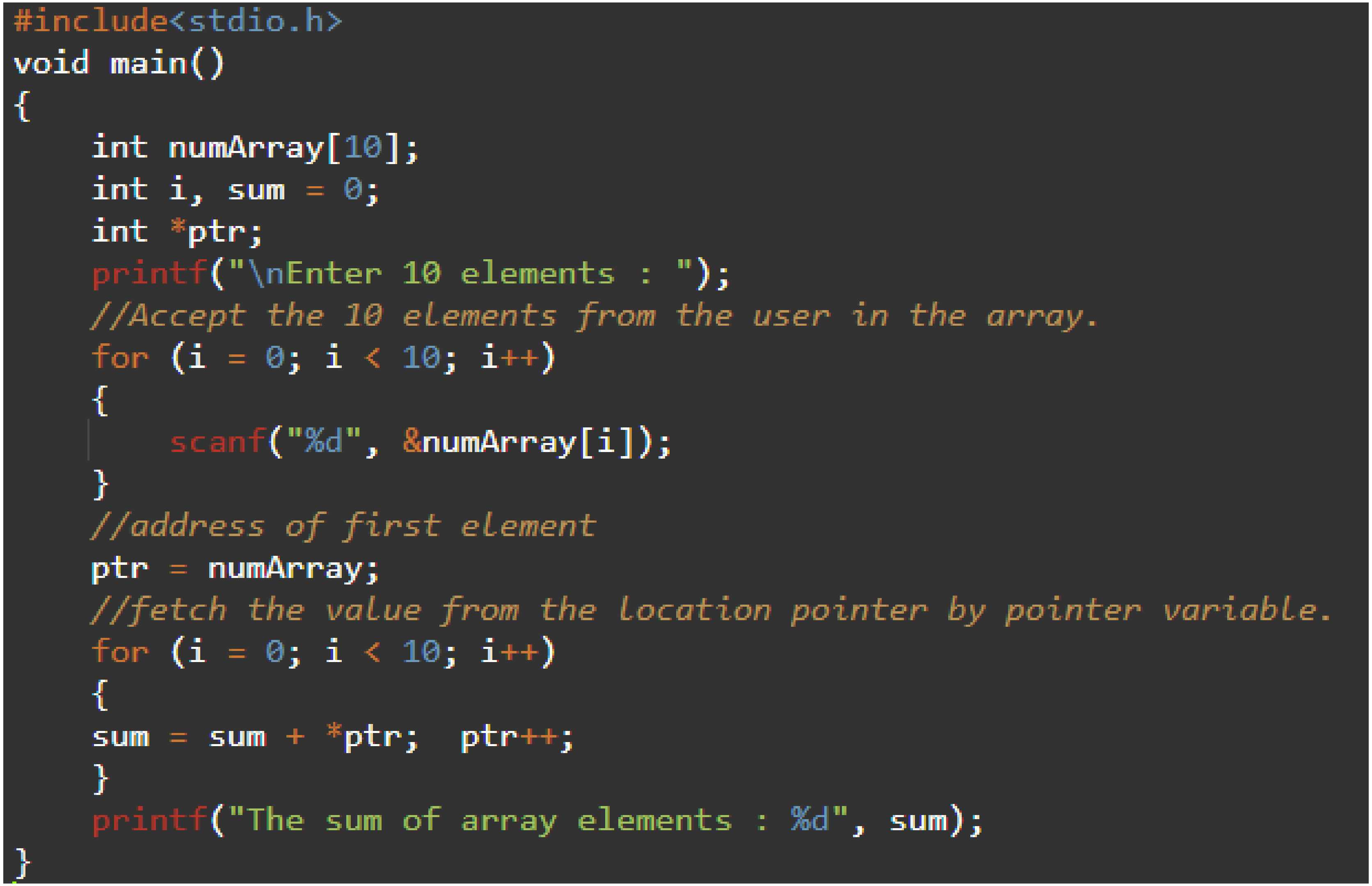
P = 1000	*p = content at address 1000 = x[0]
P+1 = 1000 + 1 x 2 = 1002	*(p+1) = content at address 1002 = x[1]
P+2 = 1000 + 2 x 2 = 1004	*(p+2) = content at address 1004 = x[2]
P+3 = 1000 + 3 x 2 = 1006	*(p+3) = content at address 1006 = x[3]
P+4 = 1000 + 4 x 2 = 1008	*(p+4) = content at address 1008 = x[4]

Pointers-Program to swap two numbers



```c
#include <stdio.h>

int main()
{
   int x, y, *a, *b, temp;

   printf("Enter the value of x and y\n");
   scanf("%d%d", &x, &y);

   printf("Before Swapping\nx = %d\ny = %d\n", x, y);

   a = &x;
   b = &y;

   temp = *b;
   *b = *a;
   *a = temp;

   printf("After Swapping\nx = %d\ny = %d\n", x, y);

   return 0;
}
```

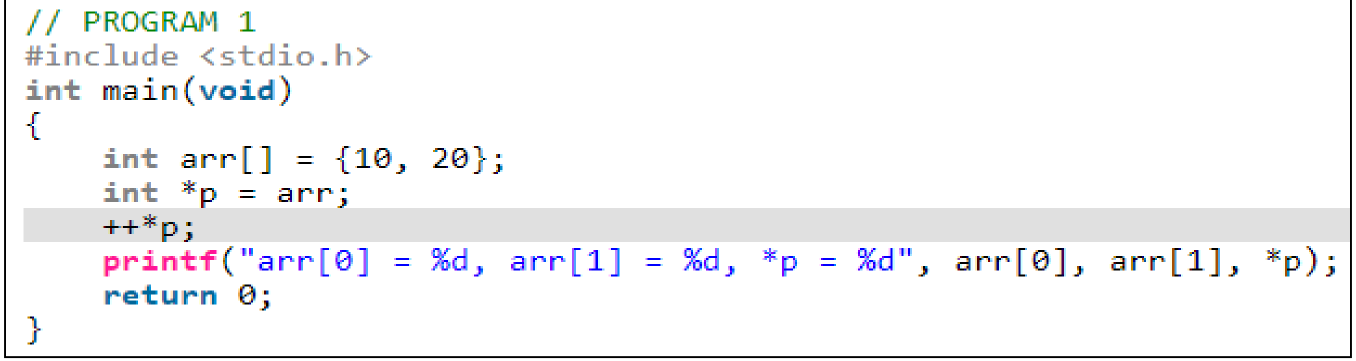
Pointers- Sum of all the elements in an array



```c
#include<stdio.h>
void main()
{
    int numArray[10];
    int i, sum = 0;
    int *ptr;
    printf("\nEnter 10 elements : ");
    //Accept the 10 elements from the user in the array.
    for (i = 0; i < 10; i++)
    {
        scanf("%d", &numArray[i]);
    }
    //address of first element
    ptr = numArray;
    //fetch the value from the location pointer by pointer variable.
    for (i = 0; i < 10; i++)
    {
    sum = sum + *ptr;  ptr++;
    }
    printf("The sum of array elements : %d", sum);
}
```

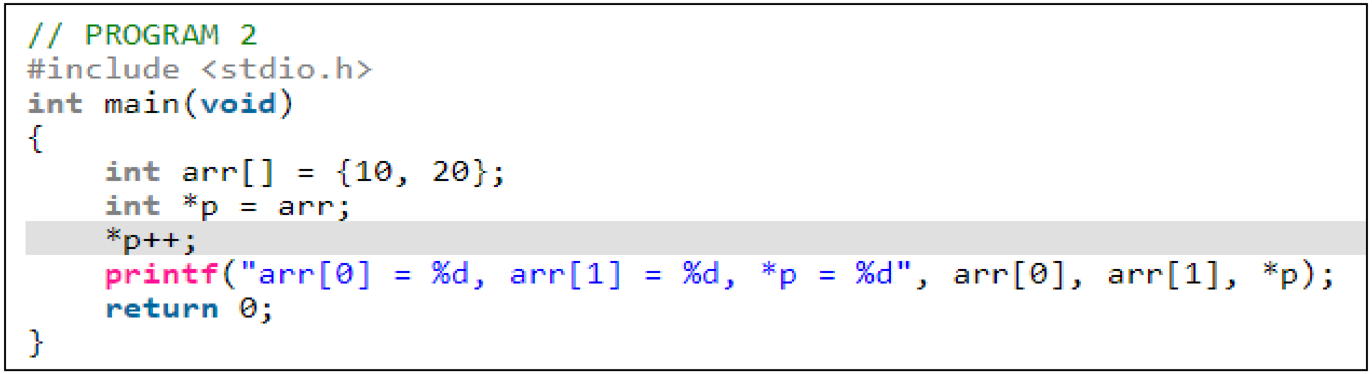
Difference between *p++, ++*p, *++p

1. Precedence of prefix ++ and * is same. Associativity of both is right to left.
2. Precedence of postfix ++ is higher than both * and prefix ++. Associativity of postfix ++ is left to right.
3. The expression **++*p** has two operators of same precedence, so compiler looks for assoiativity. Associativity of operators is right to left. Therefore the expression is treated as ***++(*p)***.
4. The expression ***p++** is treated as ***(p++)*** as the precedence of postfix ++ is higher than *.
5. The expression ***++p** has two operators of same precedence, so compiler looks for assoiativity. Associativity of operators is right to left. Therefore the expression is treated as ***(++p)***.



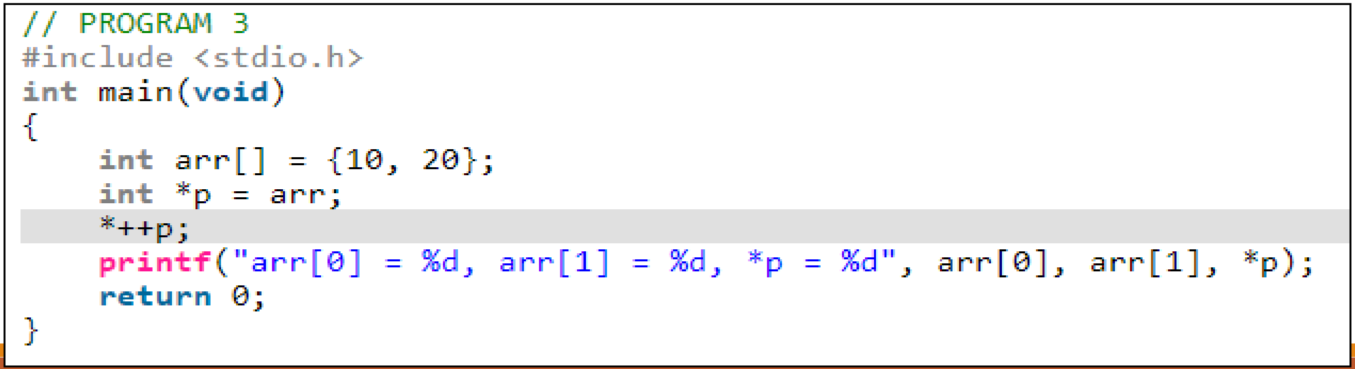
```c
// PROGRAM 1
#include <stdio.h>
int main(void)
{
    int arr[] = {10, 20};
    int *p = arr;
    ++*p;
    printf("arr[0] = %d, arr[1] = %d, *p = %d", arr[0], arr[1], *p);
    return 0;
}
```

“arr[0] = 11,
arr[1] = 20,
**p = 11“*



```c
// PROGRAM 2
#include <stdio.h>
int main(void)
{
    int arr[] = {10, 20};
    int *p = arr;
    *p++;
    printf("arr[0] = %d, arr[1] = %d, *p = %d", arr[0], arr[1], *p);
    return 0;
}
```

“arr[0] = 10,
arr[1] = 20,
**p = 20“*



```c
// PROGRAM 3
#include <stdio.h>
int main(void)
{
    int arr[] = {10, 20};
    int *p = arr;
    *++p;
    printf("arr[0] = %d, arr[1] = %d, *p = %d", arr[0], arr[1], *p);
    return 0;
}
```

“arr[0] = 10,
arr[1] = 20,
**p = 20“*

Thank you