

CS 109 S2: Fundamentals of Computer & Programming



Dr. Balu L. Parne
Assistant Professor,
CoED, SVNIT Surat.



Pointers in C-Programming

Introduction to Pointers:

- Pointers are powerful features of C and C++ programming.
- If you have a variable **var** in your program, **&var** will give you its address in the memory.
- We have used address numerous times while using the **scanf()** function.

```
scanf("%d", &var);
```


- Here, the value entered by the user is stored in the **address of var variable**.

Introduction to Pointers:

```
#include <stdio.h>
int main()
{
    int var = 5;
    printf("var: %d\n", var);

    // Notice the use of & before var
    printf("address of var: %u", &var);
    return 0;
}
```

Note: You will probably get a different address when you run the above code.

 C:\Users\balup\Desktop\C Programs\Pointers\Address.exe

```
var: 5
address of var: 6487580
-----
Process exited after 0.6737 seconds with return value 0
Press any key to continue . . .
```

Introduction to Pointers:

- Pointers (pointer variables) are special variables that are used to store addresses rather than values.
- Here is how we can declare pointers. we have declared a **pointer p of int type**.

```
int* p;
```

- Pointer can also be declared as follows:

```
int *p1;  
int * p2;
```

Introduction to Pointers:

- we can declared a pointer p1 and a normal variable p2 as follows:

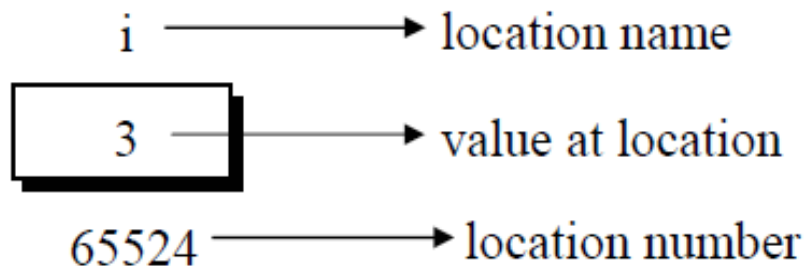
```
int* p1, p2;
```

```
int* pc, c;  
c = 5;  
pc = &c;
```

- Here, 5 is assigned to the c variable. And, the address of c is assigned to the pc pointer.

Pointer Notations:

- Consider the declaration, *int i = 3 ;*
- This declaration tells the C compiler to:
 - Reserve space in memory to hold the integer value.
 - Associate the name **i** with this memory location.
 - Store the value 3 at this location.
- We may represent the **i** location in computer memory as




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.

Pointer Notations:

- We can print this address number through the following program:

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

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.

 C:\Users\balup\Desktop\C Programs\Pointers\PrintingAddress.exe

Address of i = 6487580

Value of i = 3

Process exited after 1.048 seconds with return value 0

Press any key to continue . . .

- 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.

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

Address of i = 6487580

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 ;**
- Here, **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 memory.



- As you can see, **i**'s value is 3 and **j**'s value is **i**'s address. But, we can't use **j** in a program without declaring it.
- **j** is a variable that contains the address of **i**, it is declared as: **int *j ;**

- **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.
- ***** stands for 'value at address'. Thus, **int *j** would mean, the value at the address contained in **j** is an **int**.

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

```
Address of i = 6487580
Address of i = 6487580
Address of j = 6487568
Value of j = 6487580
Value of i = 3
Value of i = 3
Value of i = 3
-----
```

Get Value of Thing Pointed by Pointers:

- To get the value of the thing pointed by the pointers, we use the `*` operator. For example:

```
int* pc, c;  
c = 5;  
pc = &c;  
printf("%d", *pc);    // Output: 5
```

Note: In this example, `pc` is a pointer, not `*pc`. You cannot and should not do something like `*pc = &c`;

`*` is called the dereference operator (when working with pointers). It operates on a pointer and gives the value stored in that pointer.

- Here, the address of `c` is assigned to the `pc` pointer. To get the value stored in that address, we used `*pc`.

Changing Value Pointed by Pointers:

- Consider example:

```
int* pc, c;  
c = 5;  
pc = &c;  
c = 1;  
printf("%d", c);    // Output: 1  
printf("%d", *pc);  // Ouptut: 1
```

- We have assigned the **address of c** to the **pc pointer**.
- Then, we **changed the value of c to 1**. Since pc and the address of c is the same, *pc gives us 1.

Changing Value Pointed by Pointers:

- Consider another example:

```
int* pc, c;  
c = 5;  
pc = &c;  
*pc = 1;  
printf("%d", *pc); // Output: 1  
printf("%d", c);   // Output: 1
```

- We have assigned **the address of c to the pc pointer**.
- Then, we changed ***pc to 1** using `*pc = 1;`. Since pc and the address of c is the same, c will be equal to 1.

Changing Value Pointed by Pointers:

- Consider another example:

```
int* pc, c, d;  
c = 5;  
d = -15;  
  
pc = &c; printf("%d", *pc); // Output: 5  
pc = &d; printf("%d", *pc); // Output: -15
```

- Initially, the address of c is assigned to the pc pointer using `pc = &c;`. Since c is 5, `*pc` gives us 5.
- Then, the address of d is assigned to the pc pointer using `pc = &d;`. Since d is -15, `*pc` gives us -15.

```

#include <stdio.h>
int main()
{
    int* pc, c;
    c = 22;
    printf("Address of c: %u\n", &c);
    printf("Value of c: %d\n\n", c);

    pc = &c;
    printf("Address of pointer pc: %u\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc);

    c = 11;
    printf("Address of pointer pc: %u\n", pc);
    printf("Content of pointer pc: %d\n\n", *pc);

    *pc = 2;
    printf("Address of c: %u\n", &c);
    printf("Value of c: %d\n\n", c);
    return 0;
}

```

Address of c: 6487572

Value of c: 22

Address of pointer pc: 6487572

Content of pointer pc: 22

Address of pointer pc: 6487572

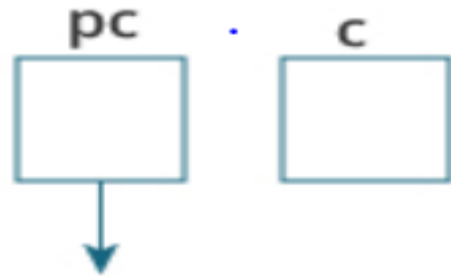
Content of pointer pc: 11

Address of c: 6487572

Value of c: 2

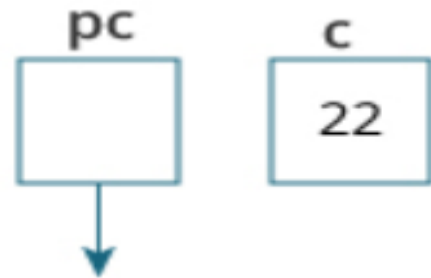
Explanation of the program:

```
int* pc, c;
```



Here, a **pointer `pc`** and a **normal variable `c`**, both of **type `int`**, is created. Since **`pc` and `c` are not initialized** at initially, pointer `pc` points to either no address or a random address. And, variable `c` has an address but contains random garbage value.

```
c = 22;
```



This **assigns 22 to the variable `c`**. That is, 22 is stored in the memory location of variable `c`.

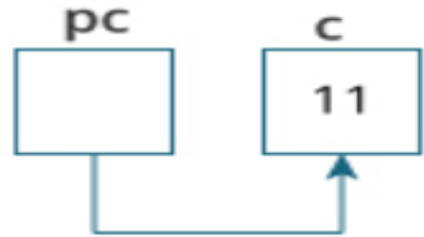
Explanation of the program:

```
pc = &c;
```



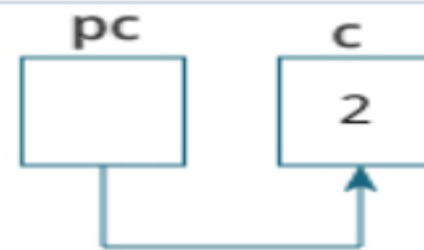
This assigns the address of variable `c` to the pointer `pc`.

```
c = 11;
```



This assigns 11 to variable `c`.

```
*pc = 2;
```



This change the value at the memory location pointed by the pointer `pc` to 2.

Common mistakes when working with pointers:

- Suppose, you want pointer pc to point to the address of c. Then,

```
int c, *pc;  
  
// pc is address but c is not  
pc = c; // Error  
  
// &c is address but *pc is not  
*pc = &c; // Error  
  
// both &c and pc are addresses  
pc = &c;  
  
// both c and *pc values  
*pc = c;
```

```
#include <stdio.h>
int main()
{
    int c = 5;
    int *p = &c;
    printf("%d", *p);
    return 0;
}
```

```
int *p = &c;
```

Is equivalent to

```
int *p;
p = &c;
```

In both cases, we are creating a pointer p (not *p) and assigning &c to it.

To avoid this confusion, we can use the statement like this:

```
int* p = &c;
```

```
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**.

```

#include<stdio.h>
int main( )
{
    int i = 3, *j, **k;
    j = &i;
    k = &j;
    printf("\nAddress of i = %u", &i);
    printf("\nAddress of i = %u", j);
    printf("\nAddress of i = %u", *k);
    printf("\nAddress of j = %u", &j);
    printf("\nAddress of j = %u", k);
    printf("\nAddress of k = %u", &k);
    printf("\nValue of j = %u", j);
    printf("\nValue of k = %u", k);
    printf("\nValue of i = %d", i);
    printf("\nValue of i = %d", *(&i));
    printf("\nValue of i = %d", *j);
    printf("\nValue of i = %d", **k);
    return 0;
}

```

```

Address of i = 6487580
Address of i = 6487580
Address of i = 6487580
Address of j = 6487568
Address of j = 6487568
Address of k = 6487560
Value of j = 6487580
Value of k = 6487568
Value of i = 3
Value of i = 3
Value of i = 3
Value of i = 3
-----

```

```
#include<stdio.h>
int main()
{
    //int a[] = {6,2,1,5,3};
    int b =10;
    int *p;
    p=&b;
    printf("%d \n", b);
    printf("%d \n", *p);
    printf("%p \n", &b);
    printf("%p \n", p);
    return 0;
}
```

 C:\Users\balup\Desktop\Data Structures

10

10

000000000062FE14

000000000062FE14

```

int main()
{
    int a[] = {6,2,1, 5, 3}; int *q;
    q = a;
    printf("%p \n", a);
    printf("%d \n", q);
    q++;
    printf("%d \n", q);
    printf("%p \n", &a);
    printf("%p \n", &q);
    printf("%d \n", a[2]);
    printf("%d \n", *(a+2));
    printf("%d \n", *(2+a));
    printf("%d \n", 2[a]);
    printf("%d \n", *(q+1));
    printf("%p \n", a+1);
    printf("%p \n", &a+1);
    printf("%p \n", &a[0]+1);
    printf("%d \n", *(a+1));
    printf("%d \n", *a+1);
    return 0;
}

```

```

000000000062FE00
6487552
6487556
000000000062FE00
000000000062FDF8
1
1
1
1
1
000000000062FE04
000000000062FE14
000000000062FE04
2
7

```



```

#include<stdio.h>
int main()
{
    int a[5], i;
    int *q;
    q = a;
    printf("Enter the elements of an array \n");
    for (i=0; i<5; i++)
    {
        scanf("%d", &a[i]);
        //scanf("%d", &q[i]);
    }
    for(i=0; i<5;i++)
    {
        printf("%d ", a[i]);
        printf("%d ", *(q+i));
        printf("%d ", *(a+i));
        printf("%d ", i[a]);
        printf("%d ", i[q]);
    }
    return 0;
}

```

C:\Users\balup\Desktop\Data Structures Program\ArrayPointer3.exe

Enter the elements of an array

```

6
7
8
9
4
6 6 6 6 6 7 7 7 7 7 8 8 8 8 8 9 9 9 9 9 4 4 4 4 4

```

Function Call:

- There are two types of function calls—**call by value** and **call by reference**.
- Arguments can generally be passed to functions in one of the two ways:
 - sending the values of the arguments
 - sending the addresses of the arguments

Call by Value:

In the first method the 'value' of each of the actual arguments in the calling function is copied into corresponding formal arguments of the called function. With this method the changes made to the formal arguments in the called function have no effect on the values of actual arguments in the calling function. The following program illustrates the 'Call by Value'.

Call by Value:

```
#include <stdio.h>
void swap(int n1, int n2);
int main()
{
    int num1 = 5, num2 = 10;
    // address of num1 and num2 is passed
    swap(num1, num2);
    printf("num1 = %d\n", num1);
    printf("num2 = %d", num2);
    return 0;
}

void swap(int n1, int n2)
{
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}
```

```
num1 = 5
num2 = 10
-----
```

Call by Reference:

- In C programming, it is also possible to pass addresses as arguments to functions.
- To accept these addresses in the function definition, we can use pointers. **It's because pointers are used to store addresses.**

In the second method (call by reference) the addresses of actual arguments in the calling function are copied into 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.

Call by Reference:

- When we call a function by passing the addresses of actual parameters then this way of calling the function is known as call by reference.
- In call by reference, the operation performed on formal parameters, affects the value of actual parameters because all the operations performed on the value stored in the address of actual parameters

```
#include <stdio.h>
void increment(int *var)
{
    /* Although we are performing the increment on variable
    * var, however the var is a pointer that holds the address
    * of variable num, which means the increment is actually done
    * on the address where value of num is stored.
    */
    *var = *var+1;
}
int main()
{
    int num=20;
    /* This way of calling the function is known as call by
    * reference. Instead of passing the variable num, we are
    * passing the address of variable num
    */
    increment(&num);
    printf("Value of num is: %d", num);
    return 0;
}
```

C:\Users\balup\Desktop\C Programs\Poi

Value of num is: 21

Call by Reference:

```
#include <stdio.h>
void swap(int *n1, int *n2);
int main()
{
    int num1 = 5, num2 = 10;
    // address of num1 and num2 is passed
    swap( &num1, &num2);
    printf("num1 = %d\n", num1);
    printf("num2 = %d", num2);
    return 0;
}

void swap(int* n1, int* n2)
{
    int temp;
    temp = *n1;
    *n1 = *n2;
    *n2 = temp;
}
```

```
num1 = 10
num2 = 5
-----
```


Passing Pointers to Functions:

```
#include <stdio.h>
void addOne(int* ptr)
{
    (*ptr)++; // adding 1 to *ptr
}
int main()
{
    int* p, i = 10;
    p = &i;
    addOne(p);
    printf("%d", *p);
    return 0;
}
```

11

```
#include<stdio.h>
int main()
{
    int radius;
    float area, perimeter;
    printf("\nEnter radius of a circle ");
    scanf("%d", &radius);
    areaperi(radius, &area, &perimeter);
    printf("Area = %f", area);
    printf("\nPerimeter = %f", perimeter);
}

areaperi(int r, float *a, float *p)
{
    *a = 3.14 * r * r ;
    *p = 2 * 3.14 * r ;
}
```

```
Enter radius of a circle 4
Area = 50.240002
Perimeter = 25.120001
-----
```

Summary:

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



Thank You...!!! 😊