

COMPUTER PROGRAMMING - I

Pointers

- Whenever we declare a variable, the system allocates memory location(s) to hold the value of the variable.
- Consider the statement

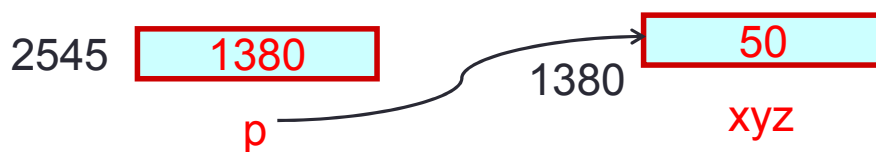
`int xyz = 50;`

- This statement instructs the compiler to allocate a location for the integer variable `xyz`, and put the value `50` in that location.
- Suppose that the address location chosen is `1380`.

xyz	→	variable
50	→	value
1380	→	address

- A pointer is a variable that represents the location or memory address (rather than the value) of a data item.

<u>Variable</u>	<u>Value</u>	<u>Address</u>
xyz	50	1380
p	1380	2545



Declaring Pointer

- The general syntax of declaring pointer variable is

`data_type *ptr_name;`

`int *a;`

`float *b;`

`char *c;`

- The **asterisk**, when used as above in the declaration, tells the compiler that the variable is to be a pointer, and the type of data that the pointer points to, but **NOT** the name of the variable pointed to.

Declaring Pointer

- 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;
```

Contd.

- A pointer variable has to be assigned a valid memory address before it can be used in the program

- ```
int *p, xyz;
:
p = &xyz;
```

  - This is called **pointer initialization**.

## Assignment of Pointer Variables (Cont ..)

- Don't try to assign a specific integer value to a pointer variable since it can be disastrous

```
float *ptr;
ptr = 120;
```

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

## Things to Remember

- Pointer variables must always point to a data item of the *same type*.

```
float x;
int *p;
:
p = &x;
```

➔ will result in erroneous output

- Assigning an absolute address to a pointer variable is prohibited.

```
int *count;
:
count = 1268;
```

# Dereferencing

- *Dereferencing* – Using a pointer variable to access the value stored at the location pointed by the variable
  - Provide indirect access to values and also called *indirection*
- Done by using the *dereferencing operator* *\** in front of a pointer variable
  - Unary operator
  - Highest precedence

# Dereferencing

- `int main( )`
- `{`
- `float data = 50.8;`
- `float *ptr;`
- `ptr = &data;`
- `printf(" value of data = %f \n",data);`
- `printf(" Address of data = %u \n",&data);`
- `printf(" Value of ptr = %u \n",ptr);`
- `printf(" Value of *ptr = %f \n",*ptr);`
- `}`

# Dereferencing

- Once the pointer variable `ptr` has been declared,
- `*ptr` represents the value pointed to by `ptr` (or the value located at the address specified by `ptr`)

## Dereferencing (Cont ..)

- The dereferencing operator `*` can also be used in assignments.

`*ptr = 200;`

- Make sure that `ptr` has been properly initialized

# Accessing a Variable Through its Pointer

- Once a pointer has been assigned the **address** of a variable, the **value** of the variable can be accessed using the **indirection operator** (\*).

```
int a, b;
int *p;
:
p = &a;
b = *p;
```

Equivalent to

b = a

## Example 1

```
#include <stdio.h>
main()
{
 int a, b;
 int c = 5;
 int *p;

 a = 4 * (c + 5);

 p = &c;
 b = 4 * (*p + 5);
 printf ("a=%d b=%d \n", a, b);
}
```

Equivalent

## Example 2

```
#include <stdio.h>
main()
{
 int x, y;
 int *ptr;

 x = 10 ;
 ptr = &x ;
 y = *ptr ;
 printf ("%d is stored in location %u \n", x, &x) ;
 printf ("%d is stored in location %u \n", *&x, &x) ;
 printf ("%d is stored in location %u \n", *ptr, ptr) ;
 printf ("%d is stored in location %u \n", y, *&ptr) ;
 printf ("%u is stored in location %u \n", ptr, &ptr) ;
 printf ("%d is stored in location %u \n", y, &y) ;

 *ptr = 25;
 printf ("\nNow x = %d \n", x);
}
```

$*\&x \Leftrightarrow x$

$ptr = \&x;$   
 $\&x \Leftrightarrow \&*ptr$

### Output:

10 is stored in location 3221224908  
10 is stored in location 3221224908  
10 is stored in location 3221224908  
10 is stored in location 3221224908  
3221224908 is stored in location  
3221224900  
10 is stored in location 3221224904

Now x = 25

Address of x: 3221224908

Address of y: 3221224904

Address of ptr: 3221224900



## Pointer Expressions

- Like other variables, pointer variables can be used in expressions.

```
int num1=2, num2= 3, sum=0, mul=0, div=1;
```

```
int *ptr1, *ptr2;
```

```
ptr1 = &num1, ptr2 = &num2;
```

```
sum = *ptr1 + *ptr2;
```

```
mul = sum * *ptr1;
```

```
*ptr2 +=1;
```

## Pointer Expressions

- int main ( )
- {
- int a, b, \*p1,\* p2, x, y, z;
- a = 12;
- b = 4;
- p1 = &a;
- p2 = &b;
- x = \*p1 \* \*p2 -6;
- y = 4\* - \*p2 / \*p1 + 10;
- printf("Address of a = %u\n", p1);
- printf("Address of b = %u\n", p2);
- printf("\n");

## Pointer Expressions

- `printf("a = %d, b = %d\n", a, b);`
- `printf("x = %d, y = %d\n", x, y);`
- `*p2 = *p2 + 3;`
- `*p1 = *p2 - 5;`
- `z = *p1 * *p2 - 6;`
- `printf("\n a = %d, b = %d," , a , b);`
- `printf("\n z = %d\n", z);`
- `}`

## Pointer Expressions

- Address of a = 2686728
- Address of b = 2686724
  
- a = 12, b = 4
- x = 42, y = 9
  
- a = 2, b = 7,
- z = 8

## Null Pointer

- A *null pointer* which is a special pointer value that is known not to point anywhere. This means that a NULL pointer does not point to any valid memory address.

- To declare a null pointer you may use the predefined constant NULL,

```
int *ptr = NULL;
```

- You can always check whether a given pointer variable stores address of some variable or contains a null by writing,

```
if (ptr == NULL)
{
 Statement block;
}
```

## Pass by value

- void f(int i)
- {
- i = 5;
- }
- int main()
- {
- int i = 3;
- f(i);
- printf("%d",i);
- }

## Pass by reference

- void f(int \*i)
- {
- \*i = 5;
- }
- int main()
- {
- int i = 3;
- f(&i);
- printf("%d",i);
- }

## Pass by Value

- void swap(int x,int y);
- void main()
- {
- int a;
- int b;
- printf(" enter two numbers");
- scanf ("%d%d",&a,&b);
- swap(a,b);
- printf("\n a=%d and b=%d",a,b);
- }
- void swap(int x, int y)
- {
- int temp=x;
- x=y;
- y=temp;
- }

## Pass by Reference

- //Pass by reference
- void swap(int \*x,int \*y);
- void main()
- {
- int a;
- int b;
- printf(" enter two numbers");
- scanf ("%d%d",&a,&b);
- swap(&a,&b);
- printf("\n a=%d and b=%d",a,b);
- }

## Pass by Reference

- void swap(int \*x, int \*y)
- {
- int temp=0;
- temp=\*x;
- \*x=\*y;
- \*y=temp;
- }

## Difference between call by value and reference

- **Actual Parameters:** The parameters that are passed while calling the function are called actual parameters.
- **Formal Parameters:** The parameters that hold the values of actual parameters are called formal parameters.
- In call by value method, if any changes are done in formal parameters then actual parameters are not changed.

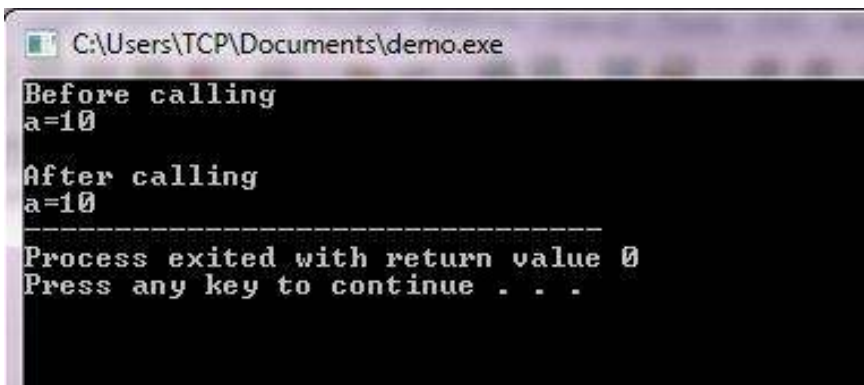
## Difference between call by value and reference

| Call by Value                                                                    | Call by reference                                                                                                 |
|----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------|
| 1. A copy of actual parameters is passed into formal parameters.                 | Reference of actual parameters is passed into formal parameters.                                                  |
| 2. Changes in formal parameters will not result in changes in actual parameters. | Changes in formal parameters will result in changes in actual parameters                                          |
| 3. Separate memory location is allocated for actual and formal parameters.       | Same memory location is allocated for actual and formal parameters.                                               |
| 4. Changes made inside the function are not reflected on other functions         | Changes made inside the function are reflected outside the function as well.                                      |
| 5. Actual arguments remain safe, they cannot be modified accidentally.           | Actual arguments are not safe. They can be accidentally modified. Hence care is required when handling arguments. |
| 6. It works locally.                                                             | It works globally                                                                                                 |

## Example (Call by value)

```
• #include<stdio.h>
•
• void fun(int x)
• {
• x=x+5;
• }
•
• int main()
• {
• int a=10;
• printf("Before calling\n a=%d",a);
• fun(a);
• printf("\n\nAfter calling\n a=%d",a);
•
• return 0;
• }
```

## Example (Call by value)



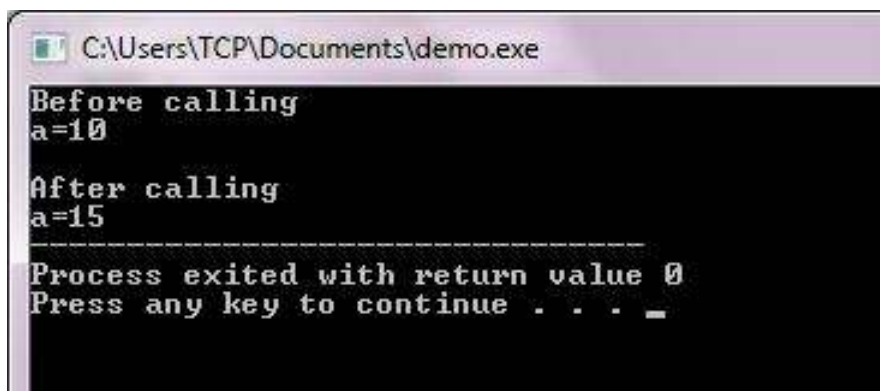
```
C:\Users\TCP\Documents\demo.exe
Before calling
a=10
After calling
a=10

Process exited with return value 0
Press any key to continue . . .
```

## Example (Call by Reference)

```
• #include<stdio.h>
•
• void fun(int *x)
• {
• *x=(*x)+5;
• }
•
• int main()
• {
• int a=10;
• printf("Before calling\na=%d",a);
• fun(&a);
• printf("\n\nAfter calling\na=%d",a);
•
• return 0;
• }
```

## Example (Call by Reference)



```
C:\Users\TCP\Documents\demo.exe
Before calling
a=10

After calling
a=15

Process exited with return value 0
Press any key to continue . . . _
```



# Pointers

- `int main()`
- `{`
- `int x, *y, z, *q;`
- `x = 3;`
- `y = &x;`
- `printf("%d\n", x);`
- `printf("%d\n", y);`
- `printf("%d\n", *y);`
- `printf("%d\n", *y+1);`
- `printf("%d\n", *(y+1));`
- `z = *(&x);`
- `q = &*y;`
- `}`

- 3
- 2686736
- 3
- 4
- 8
- .

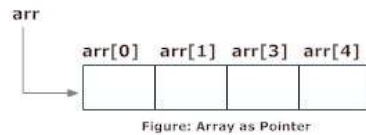
## Pointers and arrays

- In C programming, name of the array always points to the first element of an array.
- Here, address of first element of an array is `&arr[0]`. Also, *arr* represents the address of the pointer where it is pointing.
- Hence, `&arr[0]` is equivalent to *arr*.
- Also, value inside the address `&arr[0]` and address *arr* are equal.
- Value in address `&arr[0]` is `arr[0]`
- value in address *arr* is `*arr`.
- Hence, `arr[0]` is equivalent to `*arr`.

## Pointers and arrays

- `a[1]` is equivalent to `(a+1)` AND,
- `a[1]` is equivalent to `*(a+1)`.
- `&a[2]` is equivalent to `(a+2)` AND,
- `a[2]` is equivalent to `*(a+2)`.
- `&a[3]` is equivalent to `(a+3)` AND,
- `a[3]` is equivalent to `*(a+3)`. . .
- `&a[i]` is equivalent to `(a+i)` AND,
- `a[i]` is equivalent to `*(a+i)`.

- Consider an array
- `int arr[4];`



## Pointers and Arrays

- `int main(){`
- `char c[4];`
- `int i;`
- `for(i=0;i<4;++i){`
- `printf("Address of c[%d]=%x\n",i,&c[i]);`
- `}`
- `return 0;`
- `}`

## Pointer and arrays

- Address of c[0]=28ff18
- Address of c[1]=28ff19
- Address of c[2]=28ff1a
- Address of c[3]=28ff1b

## Pointers and arrays

- int main()
- {
- int a[10],n,i;
- printf("enter number values");
- scanf("%d",&n);
- printf("enter the values");
- for( i=0;i<n;i++)
- {
- scanf("%d",&a[i]);
- }
- int total;
- total = totalArray(a,n);
- printf("Total of array elements is %d",total);
- }

## Pointers and arrays

- `int totalArray(int *A, int N)`
- `{`
- `int i,total = 0;`
- `for (i = 0; i < N; i++)`
- `total += A[i];`
- `return total;`
- `}`

## Passing Arrays into function

- `int main()`
- `{`
- `int a[10],n,i;`
- `printf("enter number values");`
- `scanf("%d",&n);`
- `printf("enter the values");`
- `for( i=0;i<n;i++)`
- `{`
- `scanf("%d",&a[i]);`
- `}`
- `int total;`
- `total = totalArray(a,n);`
- `printf("Total of array elements is %d",total);`
- `}`

## Passing Arrays into function

- `int totalArray(int A[ ], int N)`
- `{`
- `int i,total = 0;`
- `for (i = 0; i < N; i++)`
- `total += A[i];`
- `return total;`
- `}`

## Pointers and arrays

- enter number values5
- enter the values1
- 2
- 3
- 4
- 5
- Total of array elements is 15

## Pointers and arrays

- Write a program to search an element within an array using concept of pointers.
- `int main()`
- `{`
- `int a[10],m,i, n;`
- `printf("enter number of values");`
- `scanf("%d",&n);`
- `printf("enter the values");`
- `for( i=0;i<n;i++)`
- `{ printf("\n");`
- `scanf("%d",&a[i]);`
- `}`
- `printf("enter the value to search");`
- `scanf("%d",&m);`
- `search(a,m,n);`
- `}`

## Pointers and arrays

- `void search(int x[],int s,int n)`
- `{`
- `int i, count=0;`
- `for( i=0;i<n;i++)`
- `{`
- `if(*x==s)`
- `{`
- `printf("\nthe number%d is found at %d:",s,(i+1));`
- `count++;`
- `}`
- `x++;`
- `}`
- `printf(" \ncount:%d ",count);`
- `}`

## Structure with pointer

- `*C program to demonstrate example of structure pointer (structure with pointer)*`
- `#include <stdio.h>`
- `struct item`
- `{`
- `char itemName[30];`
- `int qty;`
- `float price;`
- `float amount;`
- `};`

## Structure with pointer

- `int main()`
- `{`
- `struct item itm; /*declare variable of structure item*/`
- `struct item *pltem; /*declare pointer of structure item*/`
- `pltem = &itm; /*pointer assignment - assigning address of itm to pltem*/`
- `/*read values using pointer*/`
- `printf("Enter product name: ");`
- `gets(pltem->itemName);`
- `printf("Enter price:");`
- `scanf("%f",&pltem->price);`
- `printf("Enter quantity: ");`
- `scanf("%d",&pltem->qty);`



## Structure with pointer

- `/*calculate total amount of all quantity*/`
- `pltem->amount =(float)pltem->qty * pltem->price;`
- `/*print item details*/`
- `printf("\nName: %s",pltem->itemName);`
- `printf("\nPrice: %f",pltem->price);`
- `printf("\nQuantity: %d",pltem->qty);`
- `printf("\nTotal Amount: %f",pltem->amount);`
- `return 0;`
- `}`