



# **Computer Programming & Problem Solving**

**CS100**

**Mrs Sanga G. Chaki**

**Department of Computer Science and Engineering  
National Institute of Technology, Goa**

**May, 2023**



# Topics

**1. Pointers revisited**

**2. Arrays and Pointers**

**3. Passing array elements/whole arrays to  
functions**

# Recap of Pointers

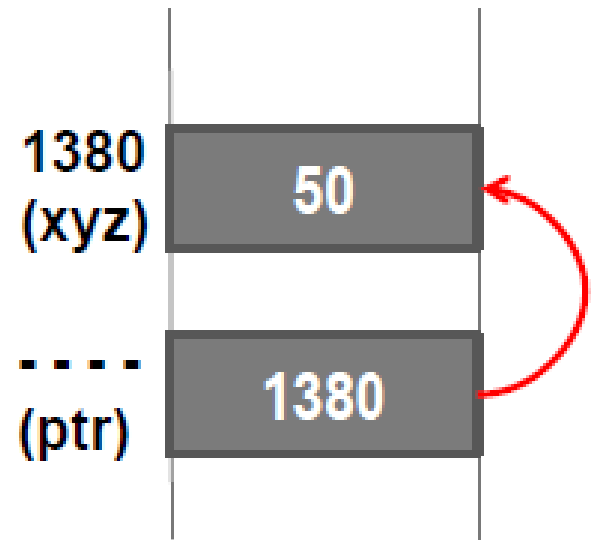
1. A pointer is a C variable whose value is the address of another variable.

2. Example:

```
int xyz = 50;
```

```
int *ptr; // Here ptr is a pointer to an integer
```

```
ptr = &xyz;
```



# Pointer to an Array Element

```
1  #include <stdio.h>
2  int main() {
3      int a[10] = {10,9,8,7,6,5,4,3,2,1};
4      int *i, *j;
5      i = &a[3]; //i is pointer to a[3]
6      j = &a[7]; //j is pointer to a[3]
7      printf("\nAt idx 3, value = %d, address = %u", a[3], &a[3]);
8      printf("\nAt idx 3, value = %d, address = %u", a[3], i);
9      printf("\nAt idx 7, value = %d, address = %u", a[7], &a[7]);
10     printf("\nAt idx 7, value = %d, address = %u", a[7], j);
11     return 0;
12 }
```

1. What is the address of an array element? Say `a[3]`?
2. So how will I declare a pointer to this array element?



# Base Address of an Array

1. The address of the 1<sup>st</sup> element of any array, or the address of the element with index = 0.
2. Can be specified using the & operator : eg. &a[0]
3. Also just by mentioning the array name.
4. If we have the base address, we can navigate the whole array.

```
1  #include<stdio.h>
2  int main()
3  {
4  int arr[5] = { 1, 2, 3, 4, 5 };
5  int *ptr = arr; //base address of array arr
6  int *ptr1 = &arr[0]; //base address of array arr
7  printf("%p\n", ptr);
8  printf("%p\n", ptr1);
9  return 0;
10 }
```

Output

```
/tmp/SLMyczD13H.o
0x7fff1ba07c60
0x7fff1ba07c60
```



# Recap of Pointers – Illegal usages

## 1. Pointing at constant.

a) `&235`

## 2. Pointing at array name

a) `int arr[20];`

b) `&arr;`

## 3. Pointing at expression

a) `&(a+b)`



# Pointer Arithmetic

1. Before we learn about pointers and arrays, we need to learn pointer arithmetic.



# Pointer Operations

- 1. Addition of a number to a pointer**
- 2. Subtraction of a number from a pointer**
- 3. Subtraction of one pointer from another**
- 4. Comparison of two pointer variables**



# What happens here?



```
main( )
{
    int i = 3, *x ;
    float j = 1.5, *y ;
    char k = 'c', *z ;

    printf ( "\nValue of i = %d", i ) ;
    printf ( "\nValue of j = %f", j ) ;
    printf ( "\nValue of k = %c", k ) ;
    x = &i ;
    y = &j ;
    z = &k ;
    printf ( "\nOriginal address in x = %u", x ) ;
    printf ( "\nOriginal address in y = %u", y ) ;
    printf ( "\nOriginal address in z = %u", z ) ;
    x++ ;
    y++ ;
    z++ ;
    printf ( "\nNew address in x = %u", x ) ;
    printf ( "\nNew address in y = %u", y ) ;
    printf ( "\nNew address in z = %u", z ) ;
}
```

Value of i = 3

Value of j = 1.500000

Value of k = c

Original address in x = 65524

Original address in y = 65520

Original address in z = 65519

New address in x = 65526

New address in y = 65524

New address in z = 65520

# Addition/Subtraction of number to a pointer



```
int i = 4, *j, *k;
```

```
j = &i;
```

```
j = j + 1;
```

```
j = j + 9;
```

```
k = j + 3;
```

```
j = j - 2;
```

```
j = j - 5;
```

```
k = j - 6;
```

1. Every time a pointer is incremented/decremented it points to the immediately next/previous location of its type.

# Addition/Subtraction of number to a pointer

```
1  #include <stdio.h>
2  int main() {
3      int i=10;
4      int * j;
5      j = &i;
6      printf("\n j = %u", j);
7      j++;
8      printf("\n j = %u", j);
9      j++;
10     printf("\n j = %u", j);
11     j--;
12     printf("\n j = %u", j);
13     j=j+9;
14     printf("\n j = %u", j);
15     j=j-9;
16     printf("\n j = %u", j);
17     return 0;
18 }
```

## Output

```
/tmp/kUq8tyZUJS.o
j = 3056920588
j = 3056920592
j = 3056920596
j = 3056920592
j = 3056920628
j = 3056920592
```

# Subtraction of one pointer from another

```
main()  
{  
    int arr[] = { 10, 20, 30, 45, 67, 56, 74 };  
    int *i, *j;  
  
    i = &arr[1];  
    j = &arr[5];  
    printf ( "%d %d", j - i, *j - *i );  
}
```

**When possible?**

**One pointer variable can be subtracted from another provided both variables point to elements of the same array**

1.  $j - i ?? \rightarrow 4$ . why?
2. Because  $j$  and  $i$  are pointing to locations that are 4 integers apart.
3.  $*j - *i ?? \rightarrow 36$ .
4. Since  $*j$  and  $*i$  return the values present at addresses contained in the pointers  $j$  and  $i$

# Comparison of two Pointer Variables

```
main( )
{
    int arr[ ] = { 10, 20, 36, 72, 45, 36 };
    int *j, *k ;

    j = &arr [ 4 ] ;
    k = ( arr + 4 ) ;

    if ( j == k )
        printf ( "The two pointers point to the same location" ) ;
    else
        printf ( "The two pointers do not point to the same location" ) ;
}
```

1. **Pointer variables can be compared provided both variables point to objects of the same data type.**

# Pointer Operations – Not Allowed



1. Addition of two pointers
2. Multiplication of a pointer with a constant
3. Division of a pointer with a constant

**Try and see what errors you get!**

# How to Access Array Elements

## – Method 1



```
main()  
{  
    int num[] = { 24, 34, 12, 44, 56, 17 };  
    int i;  
  
    for ( i = 0 ; i <= 5 ; i++ )  
    {  
        printf ( "\naddress = %u ", &num[i] ) ;  
        printf ( "element = %d", num[i] ) ;  
    }  
}
```

### 1. Using subscripted variables

# How to Access Array Elements

## – Method 2



```
1  #include <stdio.h>
2  int main() {
3      int A[]={1,2,3,4};
4      int i, *j;
5      j=&A[0];
6      for(i=0;i<6;i++){
7          printf("\nIndex = %d",i);
8          printf("\nAddress = %u",j);
9          printf("\nValue = %d",*j);
10         j++;
11     }
12     return 0;
13 }
```

### 1. Using Pointers



# Base Address of an Array

```
1  #include <stdio.h>
2  int main() {
3      int A[]={1,2,3,4};
4      int *i, *j, *k; //Pointers to base address
5      //Different ways to access base address of an array
6      i = &A[0];
7      j = A;
8      k = (A+0);
9      printf("\n Base Address = %u",i);
10     printf("\nBase Address = %u",j);
11     printf("\nBase Address = %u",k);
12     int *l, *m; //Pointers to address of element at index 2
13     //Different ways to access address of element at index 2
14     l = &A[2];
15     m = (A+2);
16     printf("\nAddress of element at index 2 = %u",l);
17     printf("\nAddress of element at index 2 = %u",m);
18     return 0;
19 }
```



# Which method to use?

- 1. Accessing array elements by pointers is always faster than accessing them by subscripts.**
- 2. Convenience matters:**
  - a) Array elements should be accessed using pointers if the elements are to be accessed in a fixed order**
  - b) Access the elements using a subscript if there is no fixed logic in accessing the elements**

# TILL Here





# Passing Array Elements to a Function

- 1. Array elements can be passed to a function by calling the function by value, or by reference.**
- 2. Call by value: pass values of array elements to the function**
- 3. Call by reference: pass addresses of array elements to the function – what is the address of an array element? – Similar to variable addresses.**

# Passing Array Elements to a Function

```
main()  
{  
    int i;  
    int marks[] = { 55, 65, 75, 56, 78, 78, 90 };  
  
    for ( i = 0 ; i <= 6 ; i++ )  
        display ( marks[i] );  
  
    for ( i = 0 ; i <= 6 ; i++ )  
        disp ( &marks[i] );  
}
```

```
display ( int m )  
{  
    printf ( "%d ", m );  
}
```

```
disp ( int *n )  
{  
    printf ( "%d ", *n );  
}
```



# Passing an Entire Array to a Function

```
display ( &num[0], 6 ) ;  
display ( num, 6 ) ;
```

1. Just pass the address of the first (base) element of the array
2. And the number of elements in the array

# Passing an Entire Array to a Function

main.c

```
1 // Online C compiler to run C program online
2 #include <stdio.h>
3 void display ( int *j, int n ) {
4     int i ;
5     for ( i = 0 ; i <= n - 1 ; i++ ) {
6         printf ( "\nelement %d = %d",i, *j ) ;
7         j++ ; /* increment pointer to point to next element */ }
8 }
9 int main(){
10     int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
11     display (&num[0], 6 ) ;
12     return 0;
13 }
```

# Passing an Entire Array to a Function

```
1 // Online C compiler to run C program online
2 #include <stdio.h>
3 void display ( int *j, int n ) {
4     int i ;
5     for ( i = 0 ; i <= n - 1 ; i++ ) {
6         printf ( "\nelement %d = %d",i, *j ) ;
7         j++ ; /* increment pointer to point to next element */ }
8 }
9 int main(){
10     int num[ ] = { 24, 34, 12, 44, 56, 17 } ;
11     display (num, 6 ) ;
12     return 0;
13 }
14
```



# Passing an Entire Array to a Function

```
1  #include <stdio.h>
2  int search(int array[], int n, int x) {
3      for (int i = 0; i < n; i++)
4          if (array[i] == x)
5              return i;
6      return -1;
7  }
8  int main() {
9      int array[] = {2, 4, 0, 1, 9};
10     int x = 1;
11     int n = sizeof(array) / sizeof(array[0]);
12     int result = search(array, n, x);
13     (result == -1) ? printf("Element not found") : printf("Element found at
        index: %d", result);
14     return 0;
15 }
```

# Things to remember



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

1. Is this correct?

# Things to remember



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

1. Is this correct?

# Things to remember



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

1. Is this correct?
2. No - Assigning an absolute address to a pointer variable is prohibited



# Pointer Expressions

1. If p1 and p2 are two pointers, the following statements are valid:

a)  $\text{sum} = (*p1) + (*p2)$ ; BUT NOT  $\text{sum} = p1 + p2$

b)  $\text{prod} = (*p1) * (*p2)$ ;

c)  $*p1 = *p1 + 2$ ;

d)  $x = *p1 / *p2 + 5$ ;

# What is happening here?

```
int x[ 5 ] = { 10, 20, 30, 40, 50 };  
int *p;
```

```
p = &x[1];  
printf( "%d", *p);
```

```
p++;  
printf( "%d", *p);
```

```
p = p + 2;  
printf( "%d", *p);
```

# Last Slide!

```
int num[] = { 24, 34, 12, 44, 56, 17 } ;
```

Base address of an array is given by the name of the array.

So what is **\*num**?

**\*num** and **\*( num + 0 )** both refer to 24.

So what is **\*( num + 1 )**

When we say, **num[i]**, the C compiler internally converts it to **\*( num + i )**. So following all are same:

```
num[i]  
*( num + i )  
*( i + num )  
i[num]
```