



# **CS10003:** **Programming & Data Structures**

**Dept. of Computer Science & Engineering**  
**Indian Institute of Technology Kharagpur**

*Autumn 2020*



# Array Pointers



# Pointer Expressions

Like other variables, pointer variables can appear in expressions

What are allowed in C?

- Add an integer to a pointer

- Subtract an integer from a pointer

- Subtract one pointer from another (related)

  - If  $p1$  and  $p2$  are both pointers to the same array, then  $p2 - p1$  gives the number of elements between  $p1$  and  $p2$

# Pointer Expressions

What are not allowed?

Adding two pointers.

```
p1 = p1 + p2;
```

Multiply / divide a pointer in an expression

```
p1 = p2 / 5;
```

```
p1 = p1 - p2 * 10;
```

# Scale Factor

We have seen that an integer value can be added to or subtracted from a pointer variable

```
int *p1, *p2;  
int i, j;  
:  
p1 = p1 + 1;  
p2 = p1 + j;  
p2++;  
p2 = p2 - (i + j);
```

In reality, it is not the integer value which is added/subtracted, but rather the **scale factor** times the value

# Scale Factor

<u>Data Type</u>	<u>Scale Factor</u>
char	1
int	4
float	4
double	8

If `p1` is an integer pointer, then

`p1++`

will increment the value of `p1` by 4



# Scale Factor

The scale factor indicates the number of bytes used to store a value of that type

So the address of the next element of that type can only be at the (current pointer value + size of data)

The exact scale factor may vary from one machine to another

Can be found out using the `sizeof`

Gives the size of that data type

Syntax:

`sizeof (data_type)`

# Pointer arithmetic and element size

```
double * p, *q ;
```

The expression `p+1` yields the correct machine address for the next variable of that type.

Other valid pointer expressions:

```
p+i
```

```
++p
```

```
p+=i
```

```
p-q /* Num of array elements between p and q */
```



# Pointer Arithmetic

Since a pointer is just a mem address, we can add to it to traverse an array.

$p+1$  returns a ptr to the next array element.

$(*p)+1$  vs  $*p++$  vs  $*(p+1)$  vs  $*(p)++$  ?

$x = *p++ \Rightarrow x = *p ; p = p + 1 ;$

$x = (*p)++ \Rightarrow x = *p ; *p = *p + 1 ;$

What if we have an array of large structs (objects)?

C takes care of it: In reality,  $p+1$  doesn't add 1 to the memory address, it adds the size of the array element.

# Pointer Arithmetic

We can use pointer arithmetic to “walk” through memory:

```
void copy(int *from, int *to, int n) {
    int i;
    for (i=0; i<n; i++) {
        *to++ = *from++;
    }
}
```

- C automatically adjusts the pointer by the right amount each time (i.e., 1 byte for a `char`, 4 bytes for an `int`, etc.)

# Pointer Arithmetic

C knows the size of the thing a pointer points to – every addition or subtraction moves that many bytes.

So the following are equivalent:

```
int get(int array[], int n)
{
    return (array[n]);
    /* OR */
    return *(array + n);
}
```

# Pointer Arithmetic

Array size  $n$ ; want to access from 0 to  $n-1$

test for exit by comparing to address one element past the array

```
int ar[10], *p, *q, sum = 0;
...
p = ar; q = &(ar[10]);
while (p != q)
    /* sum = sum + *p; p = p + 1; */
    sum += *p++;
```

Is this legal?

C defines that one element past end of array **must be a valid address**, i.e., not cause an bus error or address error

# Example

```
int main()
{
    printf ("No. of bytes in int is %u \n",    sizeof(int));
    printf ("No. of bytes in float is %u \n",  sizeof(float));
    printf ("No. of bytes in double is %u \n", sizeof(double));
    printf ("No. of bytes in char is %u \n",   sizeof(char));

    printf ("No. of bytes in int * is %u \n",  sizeof(int *));
    printf ("No. of bytes in float * is %u \n", sizeof(float *));
    printf ("No. of bytes in double * is %u \n", sizeof(double *));
    printf ("No. of bytes in char * is %u \n",  sizeof(char *));
    return 0;
}
```

## Output on a PC

```
No. of bytes in int is 4
No. of bytes in float is 4
No. of bytes in double is 8
No. of bytes in char is 1
No. of bytes in int * is 4
No. of bytes in float * is 4
No. of bytes in double * is 4
No. of bytes in char * is 4
```

Note that pointer takes 4 bytes to store, independent of the type it points to

However, this can vary between machines

Output of the same program on a server

**No. of bytes in int is 4**

**No. of bytes in float is 4**

**No. of bytes in double is 8**

**No. of bytes in char is 1**

**No. of bytes in int \* is 8**

**No. of bytes in float \* is 8**

**No. of bytes in double \* is 8**

**No. of bytes in char \* is 8**

Always use sizeof() to get the correct size`

Should also print pointers using **%p** (instead of %u as we have used so far for easy comparison)

# Example

```
int main()
{
    int A[5], i;

    printf("The addresses of the array elements are:\n");
    for (i=0; i<5; i++)
        printf("&A[%d]: Using %p = %p, Using %u = %u", i, &A[i], &A[i]);
    return 0;
}
```

## Output on a server machine

```
&A[0]: Using %p = 0x7fffb2ad5930, Using %u = 2997705008
&A[1]: Using %p = 0x7fffb2ad5934, Using %u = 2997705012
&A[2]: Using %p = 0x7fffb2ad5938, Using %u = 2997705016
&A[3]: Using %p = 0x7fffb2ad593c, Using %u = 2997705020
&A[4]: Using %p = 0x7fffb2ad5940, Using %u = 2997705024
```

**0x7fffb2ad5930 = 140736191093040** in decimal (**NOT 2997705008**)  
so print with %u prints a wrong value (4 bytes of unsigned int cannot hold 8 bytes for the pointer value)



# Pointers and Arrays

When an array is declared,

The compiler allocates sufficient amount of storage to contain all the elements of the array in contiguous memory locations

The **base address** is the location of the first element (**index 0**) of the array

The compiler also defines the array name as a **constant pointer** to the first element



# Example

Consider the declaration:

```
int x[5] = {1, 2, 3, 4, 5};
```

Suppose that each integer requires 4 bytes

Compiler allocates a contiguous storage of size  $5 \times 4 = 20$  bytes

Suppose the starting address of that storage is 2500

<u>Element</u>	<u>Value</u>	<u>Address</u>
x[0]	1	2500
x[1]	2	2504
x[2]	3	2508
x[3]	4	2512
x[4]	5	2516

# Example

The array name `x` is the starting address of the array

Both `x` and `&x[0]` have the value `2500`

`x` is a constant pointer, so cannot be changed

`X = 3400`, `x++`, `x += 2` are all illegal

If `int *p` is declared, then

`p = x;` and `p = &x[0];` are equivalent

We can access successive values of `x` by using `p++` or `p--` to move from one element to another

Relationship between  $p$  and  $x$ :

$p = \&x[0] = 2500$

$p+1 = \&x[1] = 2504$

$p+2 = \&x[2] = 2508$

$p+3 = \&x[3] = 2512$

$p+4 = \&x[4] = 2516$

**In general,  $*(p+i)$  gives the value of  $x[i]$**

C knows the type of each element in array  $x$ , so knows how many bytes to move the pointer to get to the next element



# Important !!

**Pitfall:** An array in C does not know its own length, & bounds not checked!

Consequence: While traversing the elements of an array (either using [ ] or pointer arithmetic), we can accidentally access off the end of an array (access more elements than what is there in the array)

Consequence: We must pass the array and its size to a function which is going to traverse it, or there should be some way of knowing the end based on the values (Ex., a -ve value ending a string of +ve values)

Accessing arrays out of bound can cause **segmentation faults**

Hard to debug (already seen in lab)

Always be careful when traversing arrays in programs



# Example

Pointers can be defined for any type, including user defined types

Example

```
struct name {  
    char first[20];  
    char last[20];  
};  
struct name *p;
```

p is a pointer which can store the address of a **struct name** type variable




# Pointers to Structures

Pointer variables can be defined to store the address of structure variables

Example:

```
struct student {  
    int roll;  
    char dept_code[25];  
    float cgpa;  
};  
struct student *p;
```

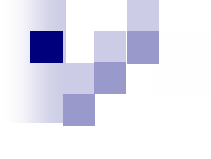


Just like other pointers, p does not point to anything by itself after declaration

Need to assign the address of a structure to p  
Can use & operator on a struct student type variable

Example:

```
struct student x, *p;  
scanf("%d%s%f", &x.roll, x.dept_code, &x.cgpa);  
p = &x;
```



Once `p` points to a structure variable, the members can be accessed in one of two ways:

`(*p).roll, (*p).dept_code, (*p).cgpa`

Note the `( )` around `*p`

`p -> roll, p -> dept_code, p -> cgpa`

The symbol `->` is called the **arrow** operator

**Example:**

```
printf("Roll = %d, Dept.= %s, CGPA = %f\n", (*p).roll,  
      (*p).dept_code, (*p).cgpa);
```

```
printf("Roll = %d, Dept.= %s, CGPA = %f\n", p->roll,  
      p->dept_code, p->cgpa);
```






# Pointers and Array of Structures

Recall that the name of an array is the address of its 0-th element

Also true for the names of arrays of structure variables.

Consider the declaration:

```
struct student class[100], *ptr ;
```



The name `class` represents the address of the 0-th element of the structure array

`ptr` is a pointer to data objects of the type `struct student`

The assignment

```
ptr = class;
```

will assign the address of `class[0]` to `ptr`

Now `ptr->roll` is the same as `class[0].roll`. Same for other members

When the pointer `ptr` is incremented by one (`ptr++`) :

The value of `ptr` is actually increased by `sizeof(struct student)`

It is made to point to the next record

Note that `sizeof` operator can be applied on any data type

# A Warning

When using structure pointers, be careful of operator precedence

Member operator “.” has higher precedence than “\*”

`ptr -> roll` and `(*ptr).roll` mean the same thing

`*ptr.roll` will lead to error

The operator “->” enjoys the highest priority among operators

`++ptr -> roll` will increment `ptr->roll`, not `ptr`

`(++ptr) -> roll` will access `(ptr + 1)->roll` (for example, if you want to print the roll no. of all elements of the class array)

# Arrays within Structures

C allows the use of arrays as structure members.

Example:

```
struct stud {  
    int    roll;  
    char  dept_code[25];  
    int   marks[6];  
    float cgpa;  
};  
struct  stud  class[100];
```

To access individual marks of students:

```
class[35].marks[4]
```

```
class[i].marks[j]
```



**Thank You!**