**Array**

Array is somewhat similar to list in Python. There are also differences.

Array has the following characteristics.

* has # of elements
* all the elements should be of the same type – homogeneous
* size of the array is specified at the point of creation
  + size is given as a constant at compile time
  + size could be specified at runtime in C99 VLA : variable length array
* array cannot grow or shrink
* elements are selected based on index or subscript
* index is an integer; starting value is 0 – zero based indexing
* supports random access
  + time required to access the element does not depend on its position
  + is always same
* Array at compile time is an array. It knows everything about itself.
* Array at runtime degenerates to a pointer.
  + At runtime, the size of the array is not stored.
* Index out of bounds – accessing elements beyond the size of the array cannot be checked.
  + Checking would require the size of the array at runtime – extra memory
  + checking would incur some runtime cost.
  + This if done would be penalizing the right programmer
* So, ‘C’ follows the philosophy. The programmer gets what he deserves!
* C array at runtime is a constant pointer
* So, arrays cannot be assigned one to another. Arrays are not assignment compatible.

You may read the following notes along with the file 1\_array.c.

This is the way to declare an array of 5 elements and initialize.

int a[5] = {11, 22, 33, 44, 55};

//printf("what : %d\n", a[5]); // undefined

In this array, the elements are in the position 0 to 4.

We can access the element as a[2]. This can be used as r-value or l-value.

What happens if we access outside the array bounds?

What happens if we access a[5]?

There is no checking at runtime. This is an undefined behaviour. The program may crash if you are lucky. You may get anything in the world. You get what you deserve.

**Array initialization:**

int a[] = {10, 20, 30, 40, 50};

The compiler will count the # of elements in the initializer and make the array have so many elements. This array will have 5 elements.

int b[5];

This array will have 5 uninitialized elements when declared within a block.

int c[5] = {11, 22};

When the array is partially initialized, the remaining elements become 0.

**Array Traversal:**

**a) using indexing**

int a[] = {10, 20, 30, 40, 50}; int n = 5;

for(int i = 0; i < n; ++i)

{

printf("%d ", a[i]);

}

printf("\n");

**b) Using pointer: version 1**

As the array is a constant pointer, we can initialize a pointer variable with an array or we can assign an array to a pointer variable.

int a[] = {10, 20, 30, 40, 50}; int n = 5;

int \*p = a;

or

int \*p; p = a;

We can then use p wherever we use the array name a.

for(int i = 0; i < n; ++i)

{

printf("%d ", p[i]);

}

printf("\n");

**c) Using pointer: version 2**

We can also use pointer arithmetic to access the elements.

These are the valid pointer arithmetic.

* Add an int to a pointer
* Subtract an int from a pointer
* take difference of two pointers when they point to the same array.

No other operations are allowed. We should never treat pointer as an integer.

Adding an integer to a pointer advances the pointer to the ith integer from there.

int a[] = {10, 20, 30, 40, 50}; int n = 5;

int \*p = a;

for(int i = 0; i < n; ++i)

{

printf("%d ", \*(p + i));

// we can also use \*(a + i)

}

**d) Using pointer: version 3**

The pointer can be incremented; array cannot. Array is a constant pointer.

There is a difference between these two expressions.

(\*p)++ and \*p++.

In the first case, \*p is incremented. The value of the expression is the old value of \*p.

In the second case, p is incremented. The old value of p is stored in temporary and then dereferenced.

We can use the following code for displaying the array.

int a[] = {10, 20, 30, 40, 50}; int n = 5;

int \*p = a;

for(int i = 0; i < n; ++i)

{

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

}

**d) Using pointer: version 4**

As the array and pointer have the same value, can we replace p by a?

int a[] = {10, 20, 30, 40, 50}; int n = 5;

int \*p = a;

for(int i = 0; i < n; ++i)

{

printf("%d ", \*a++);

}

The answer is NO. Array cannot be changed.

**Passing array as an argument to a function :**

When an array is an argument, the corresponding parameter is a pointer. The parameter gets the value of the array.

The parameter is not an array. It can never be an array.

We can ask the parameter whether it refers to an array. There is no way to find the number of elements in the array through the pointer.

These are the possible ways to send array as argument.

a) array and its size

b) pointer to the beginning and to the end of the array

c) have a logical element to mark the end of the array – this concept is called sentinel.

In this example, we will pass the array and its size.

void read\_array(int x[], int n);

void disp\_array(int \*x, int n);

int x[] and int\* x are same.

These are the routines to read and display the arrays.

void read\_array(int x[], int n)

{

for(int i = 0; i < n; ++i)

{

scanf("%d", x++); // &x[i] x + i x++

}

}

void disp\_array(int \*x, int n)

{

for(int i = 0; i < n; ++i)

{

printf("%d ", \*x++); // x[i] \*(x + i)

}

printf("\n");

}

check these files.

2\_array.h 2\_util.c 2\_util.h makefile