

# Programming with Indexed Variables

## Variable with One Index: 1-Dimensional Array

```
... what(...) {  
    int a[10] ;  
    .....  
}
```

What does the declaration mean?

## Meaning

- It is an **1-dimensional array** of **ten** locations, each of type **int**.
- Compiler generates machine code so that every time the function **what()** is invoked (called), there will be an allocation of **10 consecutive** locations of type **int**. The locations are destroyed when the control returns from the function.

## Meaning

- The total space allocated is  $10 \times \text{sizeof}(\text{int})$ . If the size of an `int` location is 4-bytes, the total allocated space is 40-bytes.
- The locations are indexed by 0 to 9.

## Meaning

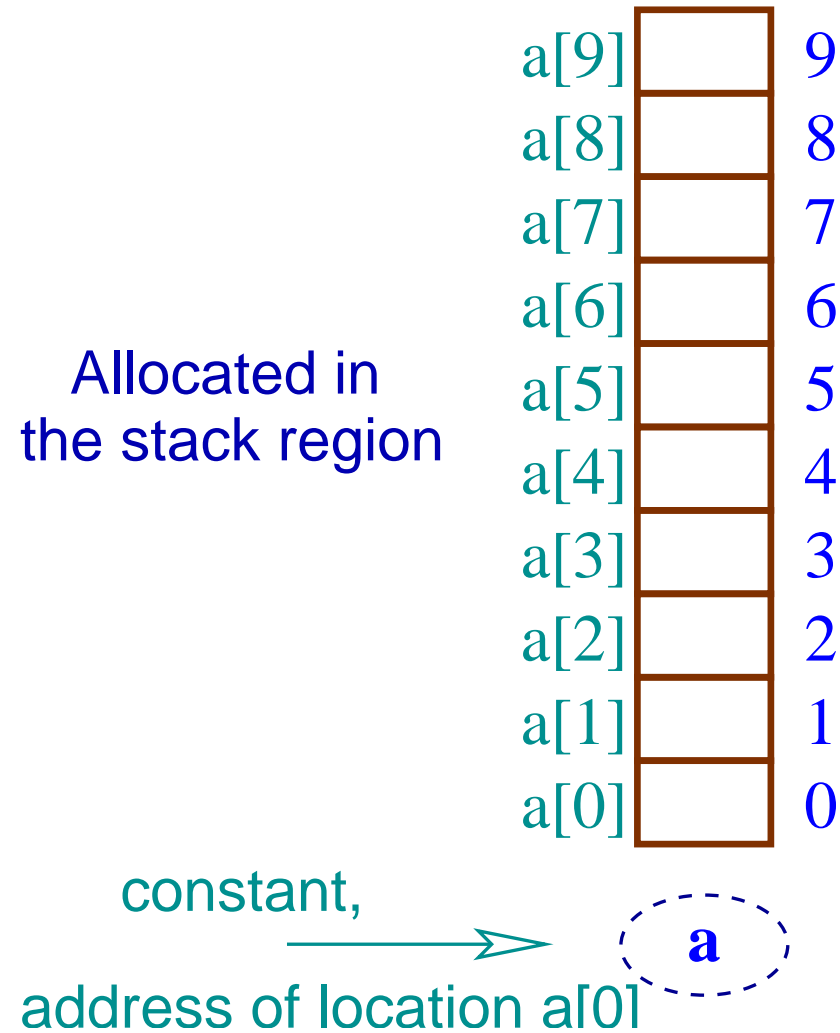
- The name **a** of the array is a **constant expression**, whose value is the address of the  $0^{th}$  location.
- The  $i^{th}$  location may be treated as an indexed variable **a[i]**,  $0 \leq i < 10^a$ .

---

<sup>a</sup>The C compiler does not stop you going beyond the index **9**, but there may be serious **run-time error**.

## Meaning

The array `a[]` is **local** to the function `what()` and its space is allocated in the **stack frame** (activation record) of the function.



## Indexed Variable

Let  $e$  be an integer expression whose value  $v$  is within the range 0 to 9,  $a[e]$  refers to the  $v^{\text{th}}$  location of the array.  $a[e]$  is treated as a variable with its content (r-value) and address (l-value).

```
a[2] = a[1] = 1 ;
```

```
a[3] = 6 - 2*a[1] ;
```

```
a[a[3]] = a[2+a[1]] + 10 ;
```



## Indexed Variable

If  $v$  is not within the range  $[0 \dots 9]$ , an access to  $a[e]$  may give a **run-time error**. But normally a C compiler, unlike Pascal or Python, does not check for array index bound.

## Array Name

The array name **a** is an expression but it is not bound to a **location** so, no value can be assigned to it.

```
int a[10] ;  
.....  
a = ..... // Illegal
```

## Array Name

```
#include <stdio.h>
int main() // arrayName.c
{
    int a[10] ;
    a = (int *)100 ;
    return 0;
}
```

```
$ cc -Wall arrayName.c
```

```
arrayName.c: In function 'main': arrayName.c:9:
error: incompatible types in assignment
```

## Array Name

It was mentioned earlier, that the value of **a** is the **address** of the **0<sup>th</sup>** location i.e.

**a** is equivalent to **&a[0]** and

**\*a** is equivalent to **a[0]**.

## Array and Pointer

The expression  $a+e$  is the address of the location  $a[e]$  i.e.  $\&a[e] \equiv (a+e)$ , and  $*(a+e)$  is same as  $a[e]$ .

Address	Pointer
$a = a+0 \equiv \&a[0]$	$*a \equiv a[0]$
$a+1 \equiv \&a[1]$	$*(a+1) \equiv a[1]$
$a+2 \equiv \&a[2]$	$*(a+2) \equiv a[2]$
...	...

## Array and Pointer

The  $i^{th}$  location of a 1-D array `a[]` of type `int` starts from the address  $(\text{unsigned})a + i * \text{sizeof}(\text{int})^a$ .

---

<sup>a</sup>The `(unsigned)a` makes the address an unsigned integer. We shall not use it explicitly to make the expression look clean.

## Array and Pointer

Location	Starting Address
$0^{th}$	<code>a</code>
$1^{st}$	<code>a + sizeof(int)</code>
$2^{nd}$	<code>a + 2*sizeof(int)</code>
...	...
$i^{th}$	<code>a + i*sizeof(int)</code>

## Pointer Arithmetic

As the value of `sizeof()` depends on data type, the meaning of `a + i` also changes depending on the type of `a[]`.





```
printf("c+10: %p,\t i+10: %p,\t d+10: %p\n", c+10, i+10, d+10);

cP = c, iP = i, dP = d;
printf("\ncP: %p,\t iP: %p,\t dP: %p\n", cP, iP, dP);
printf("cP+1: %p,\t iP+1: %p,\t dP+1: %p\n", cP+1, iP+1, dP+1);
printf("cP+2: %p,\t iP+2: %p,\t dP+2: %p\n", cP+2, iP+2, dP+2);
printf("cP+10: %p,\t iP+10: %p,\t dP+10: %p\n", cP+10, iP+10, dP+10);

cP = (char *)0, iP = (int *)0, dP = (double *)0;
printf("\ncP: %p,\t iP: %p,\t dP: %p\n", cP, iP, dP);
printf("cP+1: %p,\t iP+1: %p,\t dP+1: %p\n", cP+1, iP+1, dP+1);
printf("cP+2: %p,\t iP+2: %p,\t dP+2: %p\n", cP+2, iP+2, dP+2);
printf("cP+10: %p,\t iP+10: %p,\t dP+10: %p\n", cP+10, iP+10, dP+10);
```

```
    return 0;  
}
```

## Array and Pointer

We can write,

$\&(* (a+e)) \equiv \&a[e] \equiv a+e$ , and

$*(\&a[e]) \equiv *(a+e) \equiv a[e]$ .

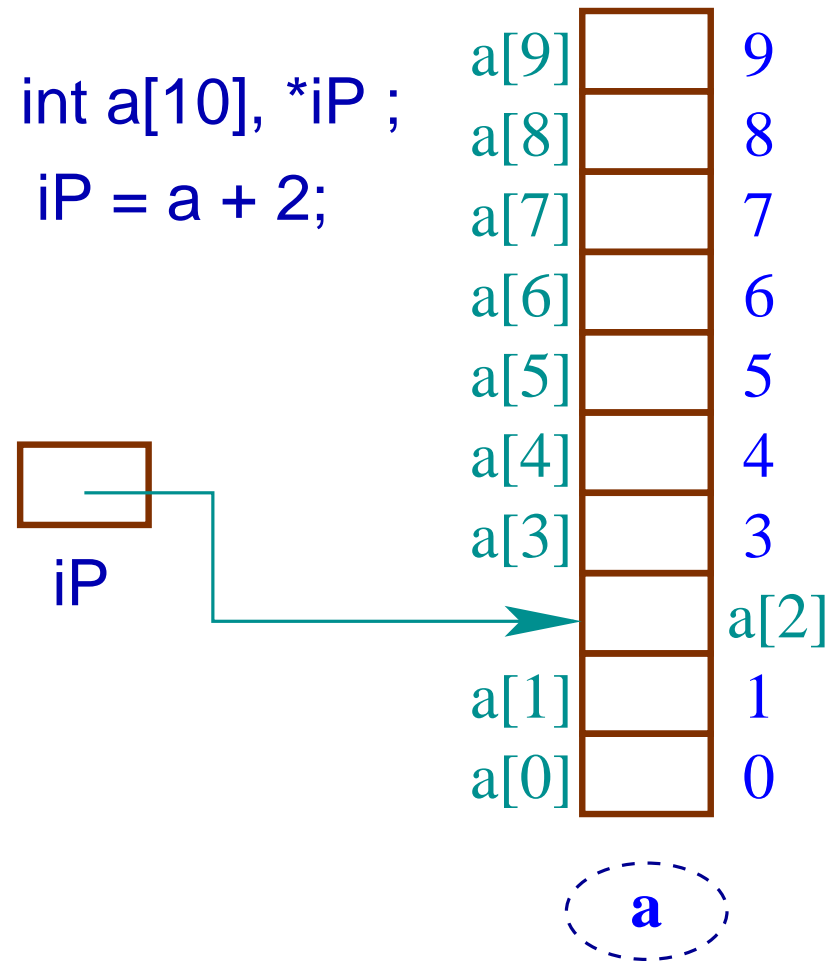
The  $*$  and  $\&$  operators are inverse to each other.

## Pointer Arithmetic

The address of an element of a 1-D array can be assigned to a pointer variable of appropriate type and the array elements can be accessed using the pointer variable. This in general is

not a good programming practice

.



## Pointer Arithmetic

```
#include <stdio.h>
int main() // ptrArith2.c
{
    int a[10] = {0, 10, 20, 30, 40, 50,
                60, 70, 80, 90}, *iP ;

    iP = a + 2;
    printf("a[2]: %d\t*iP: %d\t\tiP[0]: %d\n", a[2], *iP,
    printf("a[5]: %d\t*(iP+3): %d\t\tiP[3]: %d\n", a[5],
    return 0;
}
```

## Example

Write a C program that

1. reads a positive integer  $n$  ( $n \leq \text{MAXSIZE}$ );
2. reads  $n$  integers in an array of type `int` starting from the index `0`;
3. prints the data present in the array from the index `0`;
4. reverses the data positions in the array  
 $\text{data}[i] \leftrightarrow \text{data}[n-1-i]$ ,
5. again prints the data present in the array from the index `0`.



C Program

```
#include <stdio.h>
#define MAXSIZE 100
int main()
{ // revArray.c
    int noOfData, data[MAXSIZE], i, halfNo ;

    printf("Enter the No. of Data (<= %d): ",
           MAXSIZE);

    scanf("%d", &noOfData) ;
    printf("\nEnter the Data\n") ;
    for(i = 0; i < noOfData; ++i)
        scanf("%d", &data[i]) ; // data+i

    printf("%d data present are\n", noOfData) ;
```

```
for(i = 0; i < noOfData; ++i)
    printf("%d ", data[i]) ; // *(data+i)
halfNo = (noOfData - 1)/2 ;
for(i = 0; i <= halfNo; ++i) {
    int temp ;

    temp = data[i] ;
    data[i] = data[noOfData-1-i] ;
    data[noOfData-1-i] = temp ;
}
printf("\nData After Reversal\n") ;
for(i = 0; i < noOfData; ++i)
    printf("%d ", data[i]) ;
```

```
printf("\n") ;  
return 0 ;  
}
```

Another C Program

```
#include <stdio.h>
#define MAXSIZE 100
int main() // revArray2.c
{
    int noOfData, data[MAXSIZE], i, j ;

    printf("Enter the No. of Data (<= %d): ",
           MAXSIZE);

    scanf("%d", &noOfData) ;
    printf("\nEnter the Data\n") ;
    for(i = 0; i < noOfData; ++i)
        scanf("%d", &data[i]) ; // data+i

    printf("%d data present are\n", noOfData) ;
```

```
for(i = 0; i < noOfData; ++i)
    printf("%d ", data[i]) ; // *(data+i)
for(i = 0, j=noOfData-1; i < j; ++i, --j) {
    int temp ;

    temp = data[i] ;
    data[i] = data[j] ;
    data[j] = temp ;
}
printf("\nData After Reversal\n") ;
for(i = 0; i < noOfData; ++i)
    printf("%d ", data[i]) ;
printf("\n") ;
```

```
    return 0 ;  
}
```



## Example

Solve the previous problem by writing a function to reverse the data in the array.

## Function Interface

- `void reverseData(int [], int) ;`
- The `first parameter` is the starting address of the array. This is equivalent to writing `int *`. The `second parameter` is the number of data present in the array.
- This function does not return any value, so the return type is `void`.

## Command Abstraction

The purpose of this function is to change the content of different locations of the array. The job is similar to that of a sequence of **statements** or **commands** and not like an **expression** (does not compute and return a value).

## Command Abstraction

This type of object is called a **procedure** or a **subprogram** in programming languages like Pascal or FORTRAN. But in C it also is called a function. Here the function is an abstraction of a sequence of commands.

## Actual Parameters for an 1-D Array?

- It is necessary to access the array elements, `a[e]` within a called function.
- An array element can be accessed if its address is known.
- The compiler can generate code to compute the address of `a[e]` if it gets the starting address of the array, the value of `e`, and the size of each array element.

Address of an Array Element:  $a[e]$

$$a + v \times s$$

- $a$  is the starting address of the array,
- $v$  is the value of the expression  $e$ .
- $s$  is the size of each element of the array

An Example

```
#include <stdio.h>
int main() // arrayAddr.c
{
    int a[10] ;
    printf("sizeof(int) = %u\n", sizeof(int));
    printf("Address of a[0]=%u:%u\n",
           (unsigned)a, (unsigned)&a[0]);
    printf("Address of a[1]=%u:%u\n",
           (unsigned)a+1*sizeof(int), (unsigned)&a[1]);
    printf("Address of a[2]=%u:%u\n",
           (unsigned)a+2*sizeof(int), (unsigned)&a[1]);
    printf("Address of a[7]=%u:%u\n",
           (unsigned)a+7*sizeof(int), (unsigned)&a[7]);
    return 0 ; }
```



## A Run

```
$ ./a.out
```

```
sizeof(int) = 4
```

```
Address of a[0]=3220264176:3220264176
```

```
Address of a[1]=3220264180:3220264180
```

```
Address of a[2]=3220264184:3220264180
```

```
Address of a[7]=3220264204:3220264204
```

```
$
```

## Address of an Array Element: $a[e]$

- The value of  $e$  is computed (compiler generates code for that).
- The size of an array element depends on its type, the programming language, compiler and the machine. But all these information are known *a priori*.

## Address of an Array Element: $a[e]$

In case of an 1-D array, the only unknown within a **called function** (callee) is the **starting address** of the array which has been declared in the **caller** or even at a higher level.

So the **only actual parameter passed** in this case is the **starting address** of the array.

Formal Parameter `int x[]` or `int *x`

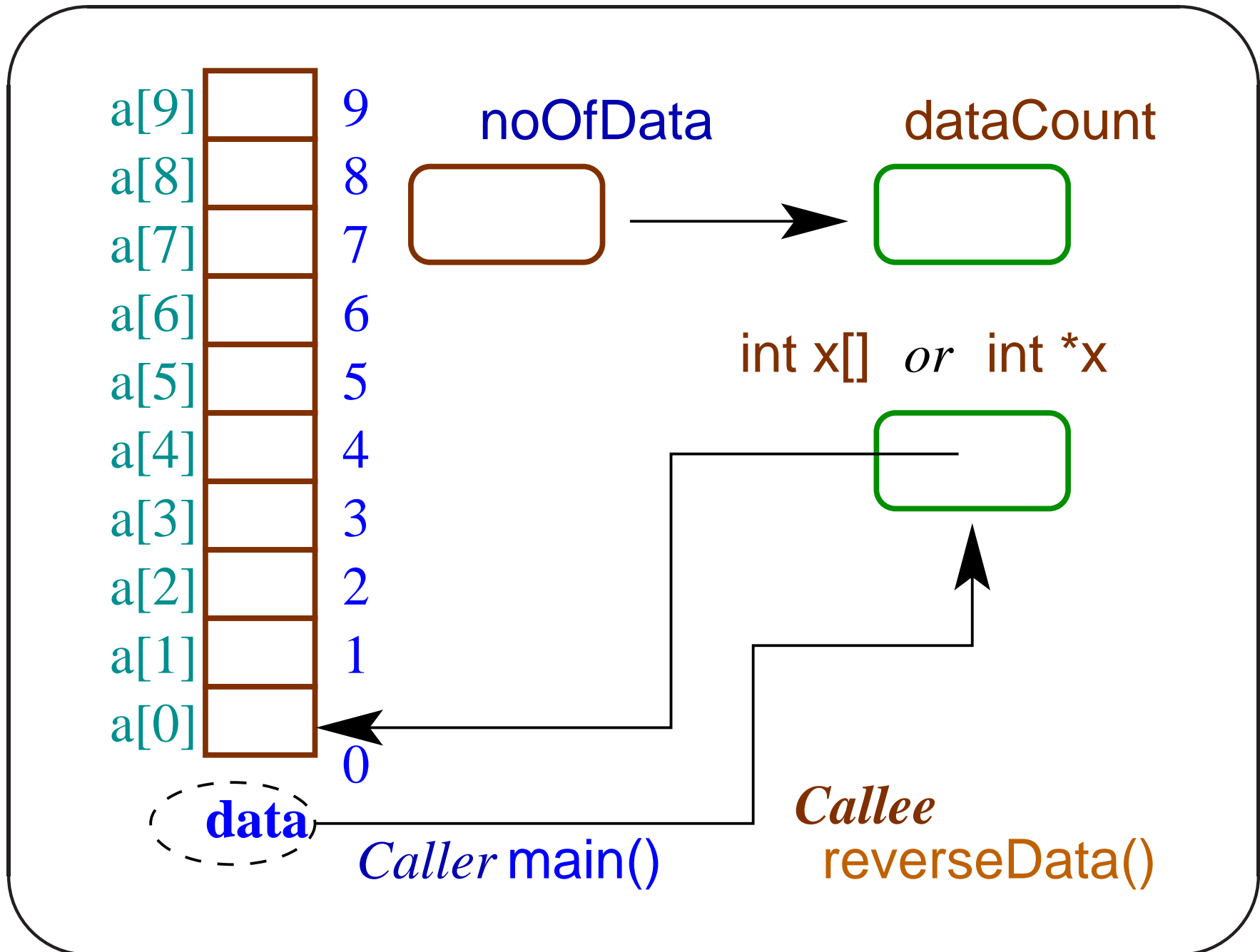
The formal parameter `x` receives the address of an `int` location. It is usually treated as the starting address of an 1-D array. But it is essentially a pointer of type `int`.

Formal Parameter `int x[]` or `int *x`

The language does not stop a programmer to pass **any address** as the actual parameter, but the result may be memory access violation (segmentation fault) or incorrect value.

## Passing an 1-D Array

```
#define MAXSIZE 100
void reverseData(int [], int); // Interface
int main()
{
    int .... data[MAXSIZE], noOfData ;
    .....
    ..... reverseData(data, noOfData)
}
void reverseData(int x[], int dataCount) {
    .... x[e] .....
}
```



C Program



```
#include <stdio.h>
#define MAXSIZE 100
void reverseData(int [], int);
int main() // revArray1.c
{
    int noOfData, data[MAXSIZE], i ;
    printf("Enter the No. of Data (<= %d): ",
           MAXSIZE);

    scanf("%d", &noOfData) ;
    printf("\nEnter the Data\n") ;
    for(i=0; i<noOfData; ++i) scanf("%d", &data[i]);
    printf("%d data present are\n", noOfData) ;
    for(i=0; i<noOfData; ++i) printf("%d ", data[i]);
    reverseData(data, noOfData) ;
```

```
printf("\nData After Reversal\n") ;
for(i = 0; i < noOfData; ++i)
    printf("%d ", data[i]) ;
printf("\n") ;
return 0;
}
void reverseData(int x[], int dataCount) {
    int halfNo, i ;

    halfNo = (dataCount - 1)/2 ;
    for(i = 0; i <= halfNo; ++i) {
        int temp ;

        temp = x[i] ;
```

```
        x[i] = x[dataCount-1-i] ;  
        x[dataCount-1-i] = temp ;  
    }  
}
```

## Array and Pointer

What will happen if the function `reverseData()` is called as `reverseData(a+2, noOfData-2)` ?

## Array Initialization

```
/*
 * arrayInit1.c
 */
#include <stdio.h>
#define MAXSIZE 5
int main()
{
    int a[MAXSIZE], b[MAXSIZE] = {0, 1, 2, 3, 4},
        c[MAXSIZE] = {10}, i ;
    float x[MAXSIZE], z[MAXSIZE] = {10.0},
        y[MAXSIZE] = {0, 10.1, 20, 30, 40};
    for(i = 0; i < MAXSIZE; ++i)
```

```
    printf("a[%d]=%d, \t\tb[%d]=%d, \tc[%d]=%d\n",  
           i, a[i], i, b[i], i, c[i]) ;  
  
    printf("\n") ;  
    for(i = 0; i < MAXSIZE; ++i)  
        printf("x[%d]=%f, \t\ty[%d]=%f, \tz[%d]=%f\n",  
               i, x[i], i, y[i], i, z[i]) ;  
  
    return 0 ;  
}
```

## Size is Implicit

```
#include <stdio.h>
int main() // arrayInit2.c
{
    int c[] = {100, 200}, b[]={10, 20} ;
    int i;

    for(i = 0; i < 5; ++i)
        printf("b[%d] = %d, \tc[%d] = %d\n",
               i, b[i], i, c[i]) ;

    printf("\n") ;
    return 0;
}
```

## Interesting Output

```
$ ./a.out
```

```
b[0] = 10, c[0] = 100
```

```
b[1] = 20, c[1] = 200
```

```
b[2] = 100, c[2] = 2
```

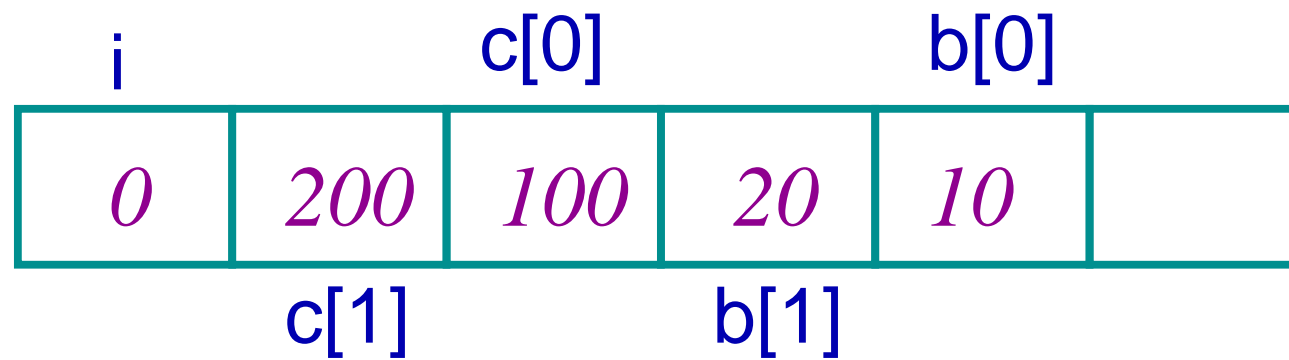
```
b[3] = 200, c[3] = 134513840
```

```
b[4] = 4, c[4] = 0
```

```
$
```



## Memory Allocation



## Space Allocation

- Two locations of type `int` are allocated and initialized to `10, 20` for `b[ ]`.
- Two more locations are allocated and initialized for `c[ ]` with `100, 200`.
- One location is allocated for `i`.

## Space Allocation

- The compiler does not prohibit access to `b[2]`, `b[3]` or `c[2]`, `c[3]`.
- `b[4]` and `2[2]` overlaps with `i`!

## C Compiler Does Not Check for the Array Limit

- Beyond the limit you get meaningless data.
- There may be memory protection violation.

.....

b[368] = 809330281, c[368] = 778121006

b[369] = 892549937, c[369] = 7632239

b[370] = 1029636154, c[370] = 0

Segmentation fault (core dumped)

\$