ECE 220 Computer Systems & Programming

Lecture 11: Pointers and Arrays

October 1, 2019

ECE ILLINOIS

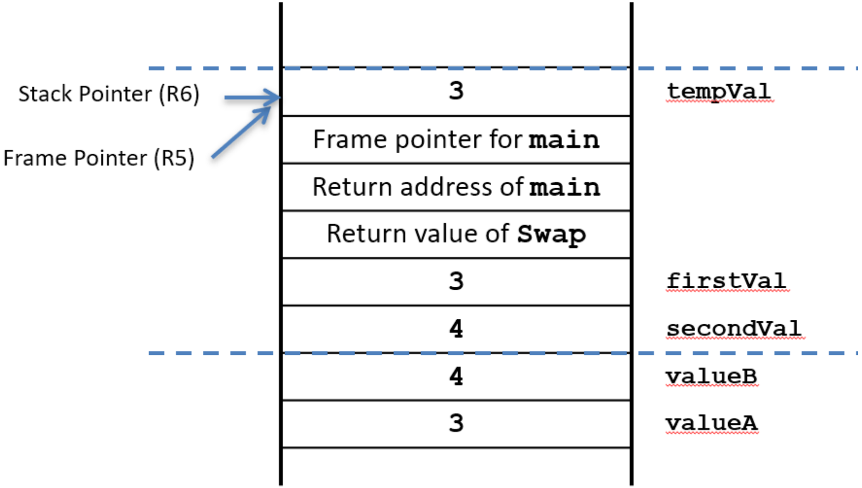
ILLINOIS

Outline

- Chapter 16
- Key concepts
  - Passing by reference with pointers
  - Arrays basics

Function Swap

```
#include<stdio.h>

void Swap(int firstVal, int secondVal);

int main()
{
        int valueA = 3;
        int valueB = 4;

1.      printf("%d %d\n”, valueA, valueB);
2.      Swap(valueA, valueB);
3.      printf("%d %d\n", valueA, valueB);
4.      return 0;
}

void Swap(int firstVal, int secondVal)
{
        int tempVal;

5.      tempVal = firstVal;
6.      firstVal = secondVal;
7.      secondVal = tempVal;
}
```



Stack Pointer (R6) Frame Pointer (R5)	3	tempVal
	Frame pointer for **main**	
	Return address of **main**	
	Return value of **Swap**	
	3	firstVal
	4	secondVal
	4	valueB
	3	valueA

Function **NewSwap**

```
#include<stdio.h>

void NewSwap(int *firstVal, int *secondVal);

int main()
{
        int valueA = 3;
        int valueB = 4;

1.      printf("%d %d\n”, valueA, valueB);
2.      NewSwap(&valueA, &valueB);
3.      printf("%d %d\n", valueA, valueB);
4.      return 0;
}

void NewSwap(int *firstVal, int *secondVal)
{
        int tempVal;

5.      tempVal = *firstVal;
6.      *firstVal = *secondVal;
7.      *secondVal = tempVal;
}
```

Pointers

Declaration

```
int *p;   /* p is a pointer to an int */
```

A pointer in C is always a pointer to a particular data type: `int*`, `double*`, `char*`, etc.

Operators

```
*p
```
-- returns the value pointed to by p

```
&z
```
-- returns the address of variable z

Example

```
int object;
int *ptr;
```

store the value 4 into the memory location associated with “object”

```
object = 4;
ptr = &object;
```

store the address of “object” into the the memory location associated with ptr

```
*ptr = *ptr + 1;
```

read the contents of memory at the address stored in ptr

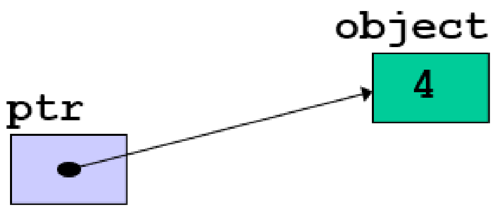
store the result into memory at the address stored in ptr

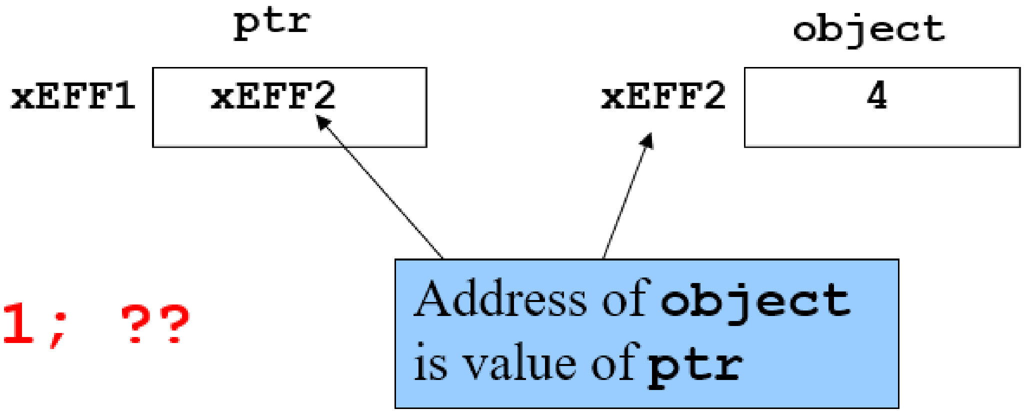
- **&** (address operator)

```
ptr = &object;
```

  - Returns address of operand

```
int object = 4;
int *ptr;
ptr = &object; //ptr gets address of object
ptr “points to” object
```





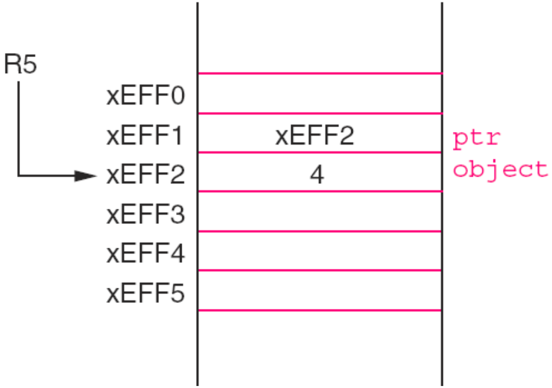
Address of **object** is value of **ptr**

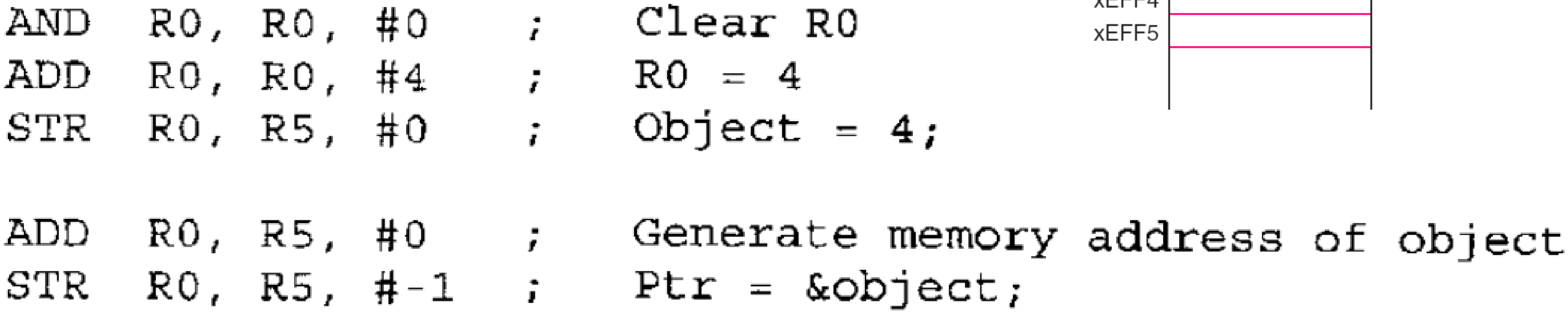
```
*ptr = *ptr + 1; ??
```

Pointers in LC3

- The indirection operator ‘*’

```
int object = 4;
int *ptr;
ptr = &object;
```

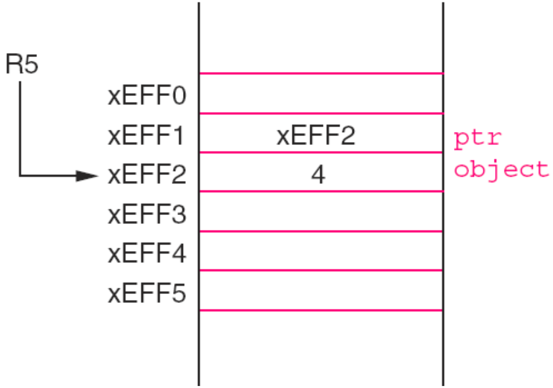




```
AND   R0, R0, #0    ;   Clear R0
ADD   R0, R0, #4    ;   R0 = 4
STR   R0, R5, #0    ;   Object = 4;

ADD   R0, R5, #0    ;   Generate memory address of object
STR   R0, R5, #-1   ;   Ptr = &object;
```

Pointers in LC3

***ptr = *ptr + 1; ??**



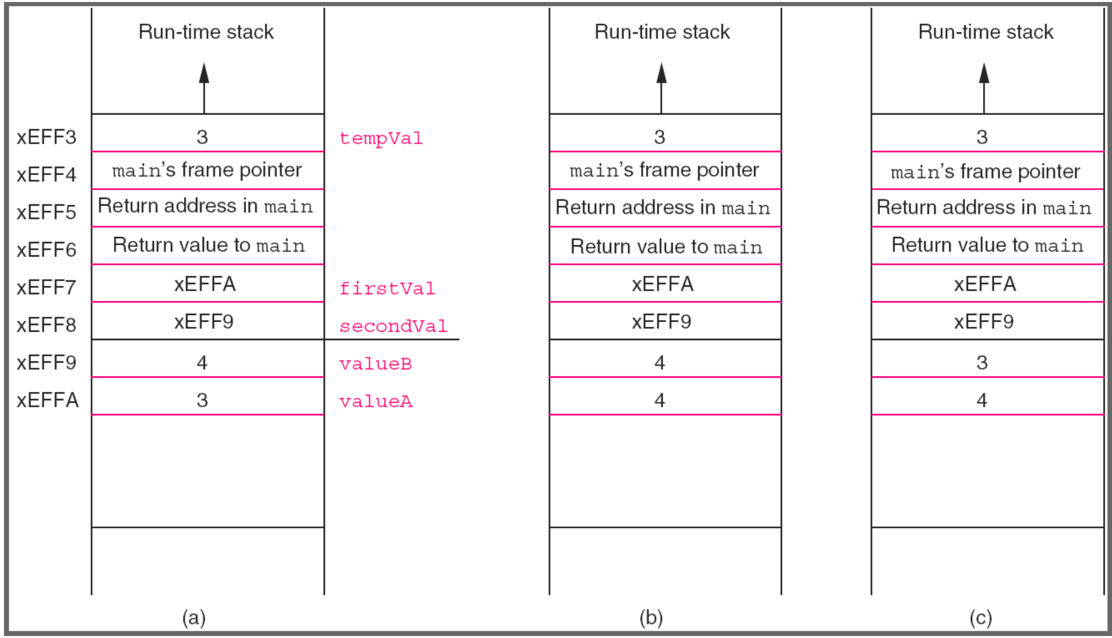
```
LDR   R0, R5, #-1    ;   R0 contains the value of ptr
LDR   R1, R0, #0     ;   R1 <- *ptr
ADD   R1, R1, #1     ;   *ptr + 1
STR   R1, R0, #0     ;   *ptr = *ptr + 1;
```

```
#include<stdio.h>

void NewSwap(int *firstVal, int *secondVal);

int main()
{
        int valueA = 3;
        int valueB = 4;

1.      printf("%d %d\n”, valueA, valueB);
2.      NewSwap(&valueA, &valueB);
3.      printf("%d %d\n", valueA, valueB);
4.      return 0;
}

void NewSwap(int *firstVal, int *secondVal)
{
        int tempVal;

5.      tempVal = *firstVal;
6.      *firstVal = *secondVal;
7.      *secondVal = tempVal;
}
```

Function

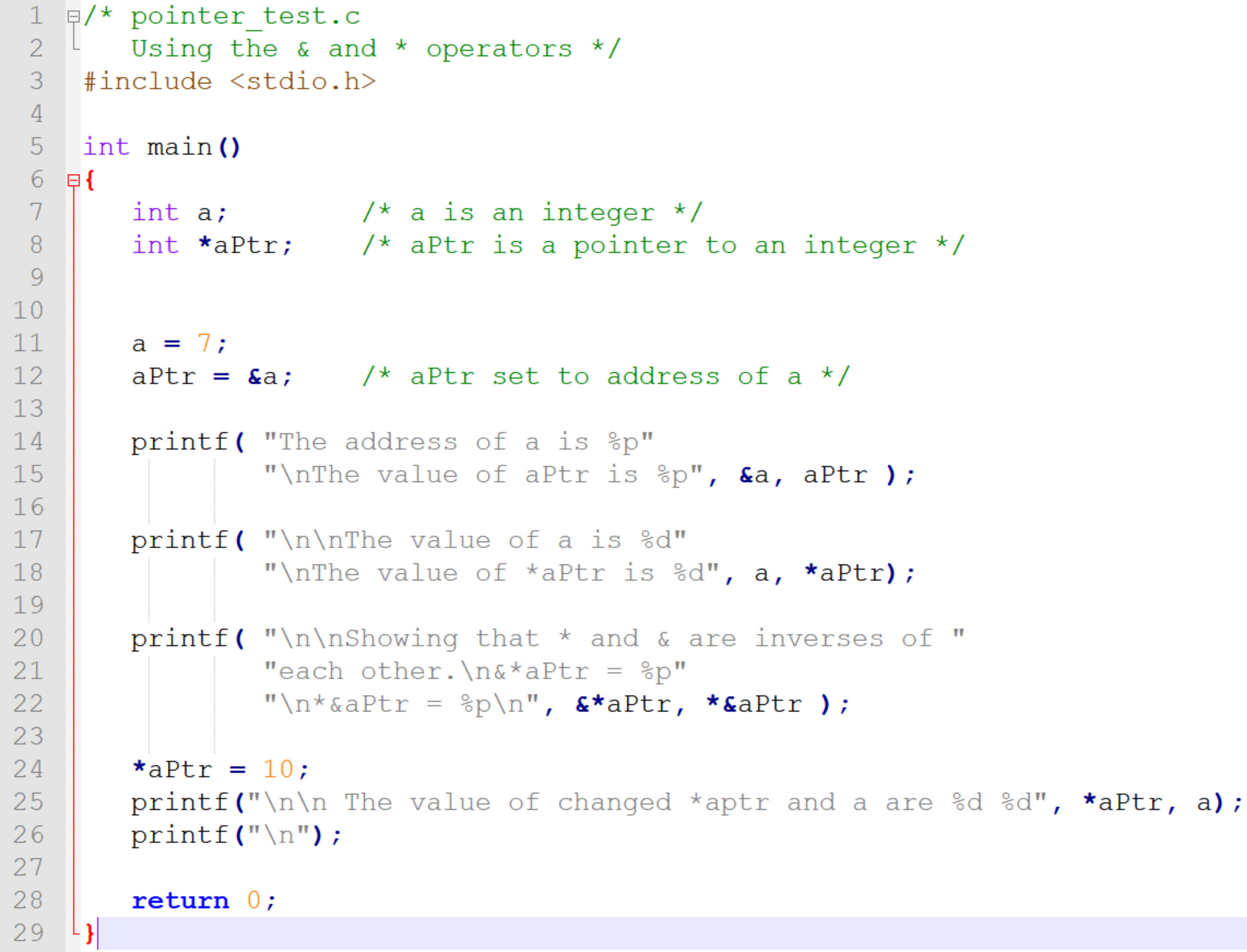
NewSwap

More Examples on Github: **CallByValue, CalByReference**



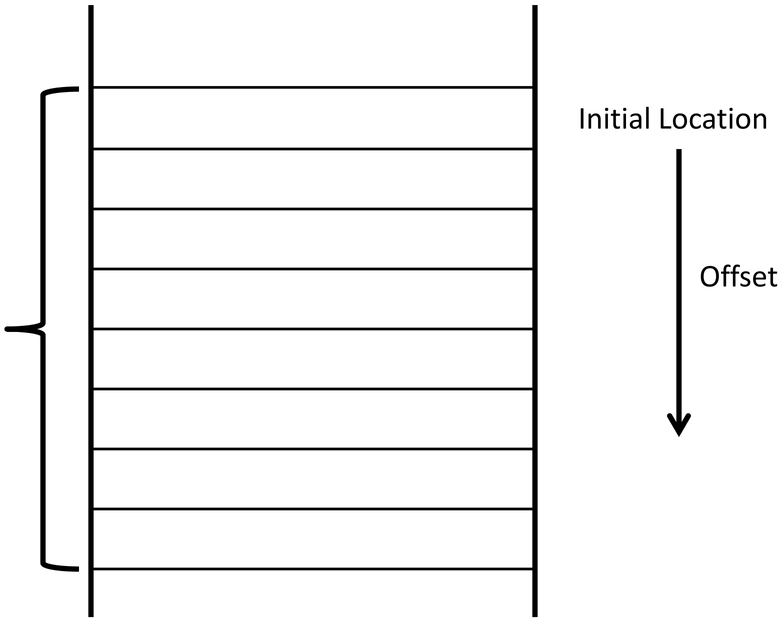
	Run-time stack (a)		Run-time stack (b)	Run-time stack (c)
xEFF3	3	tempVal	3	3
xEFF4	main’s frame pointer		main’s frame pointer	main’s frame pointer
xEFF5	Return address in main		Return address in main	Return address in main
xEFF6	Return value to main		Return value to main	Return value to main
xEFF7	xEFFA	firstVal	xEFFA	xEFFA
xEFF8	xEFF9	secondVal	xEFF9	xEFF9
xEFF9	4	valueB	4	3
xEFFA	3	valueA	4	4

Exercise:



```c
/* pointer_test.c
   Using the & and * operators */
#include <stdio.h>

int main()
{
   int a;          /* a is an integer */
   int *aPtr;      /* aPtr is a pointer to an integer */


   a = 7;
   aPtr = &a;      /* aPtr set to address of a */

   printf( "The address of a is %p"
           "\nThe value of aPtr is %p", &a, aPtr );

   printf( "\n\nThe value of a is %d"
           "\nThe value of *aPtr is %d", a, *aPtr);

   printf( "\n\nShowing that * and & are inverses of "
           "each other.\n&*aPtr = %p"
           "\n*&aPtr = %p\n", &*aPtr, *&aPtr );

   *aPtr = 10;
   printf("\n\n The value of changed *aptr and a are %d %d", *aPtr, a);
   printf("\n");

   return 0;
}
```

Arrays: Basic Concept



Sequentially arranged data of **same type**

Arrays: Basic Concept

How do we allocate a group of memory locations?

- **character string**
- **table of numbers**

How about this?

```
int num0;
int num1;
int num2;
int num3;
```

Not too bad, but…

- **what if there are 100 numbers?**
- **how do we write a loop to process each number?**

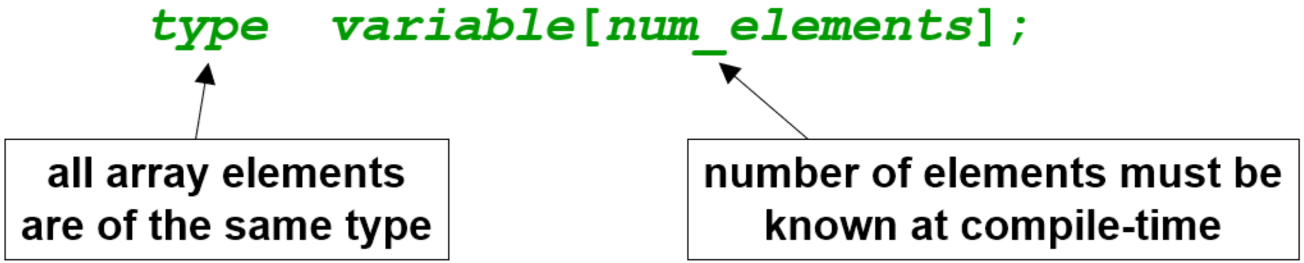
Fortunately, C gives us a better way -- the *array*.

```
int num[4];
```

Declares a sequence of four integers, referenced by:
`num[0]`, `num[1]`, `num[2]`, `num[3]`.

Arrays: Syntax

Declaration

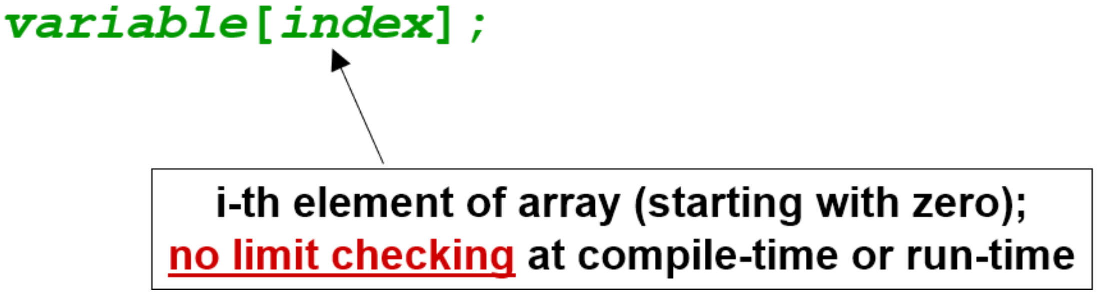


```
type   variable[num_elements];
```

all array elements are of the same type

number of elements must be known at compile-time

Array Reference

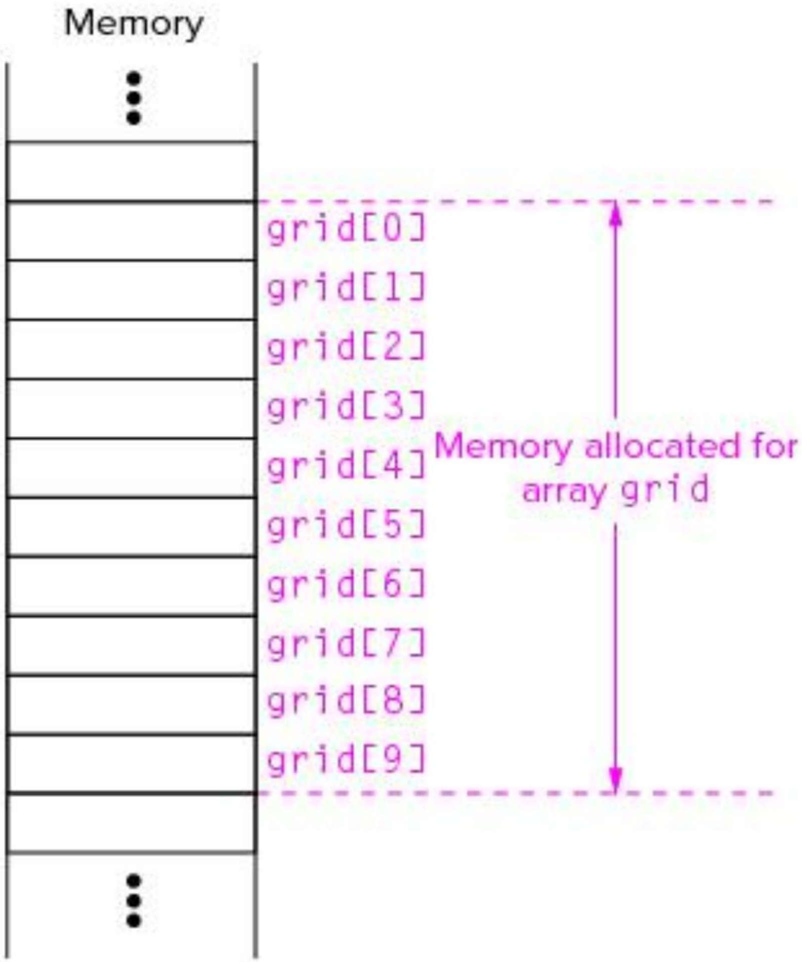


```
variable[index];
```

i-th element of array (starting with zero); no limit checking at compile-time or run-time

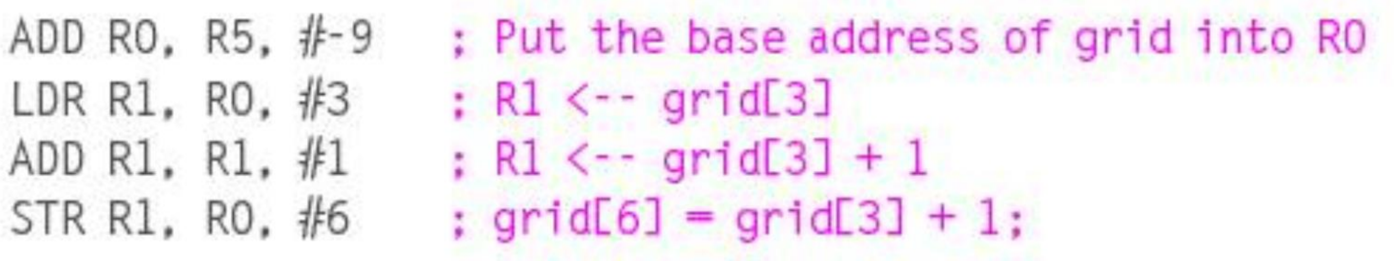
Memory allocation of Int grid [10]

grid[0] is allocated at the lowest memory address



The array grid allocated in memory.

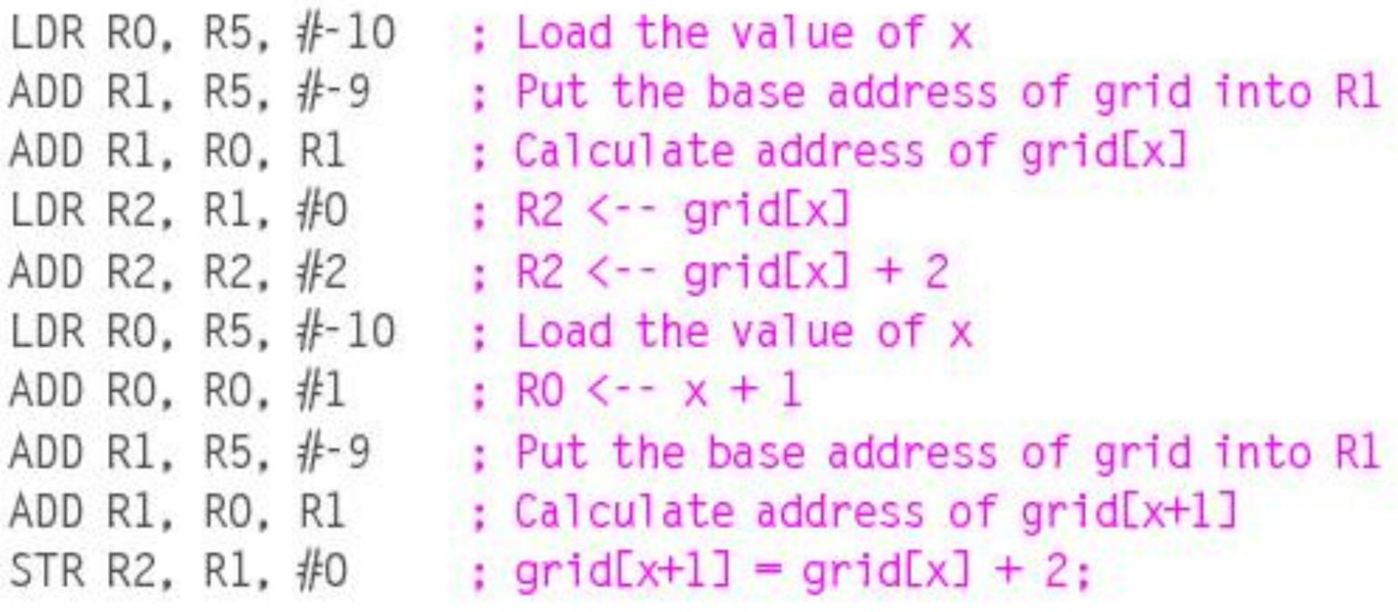
Assume **grid** is local variable. **grid[6] = grid[3]+1;**



```
ADD R0, R5, #-9    ; Put the base address of grid into R0
LDR R1, R0, #3     ; R1 <-- grid[3]
ADD R1, R1, #1     ; R1 <-- grid[3] + 1
STR R1, R0, #6     ; grid[6] = grid[3] + 1;
```

Assume, x is allocated on top of the grid.

grid[x+1] = grid[x]+2



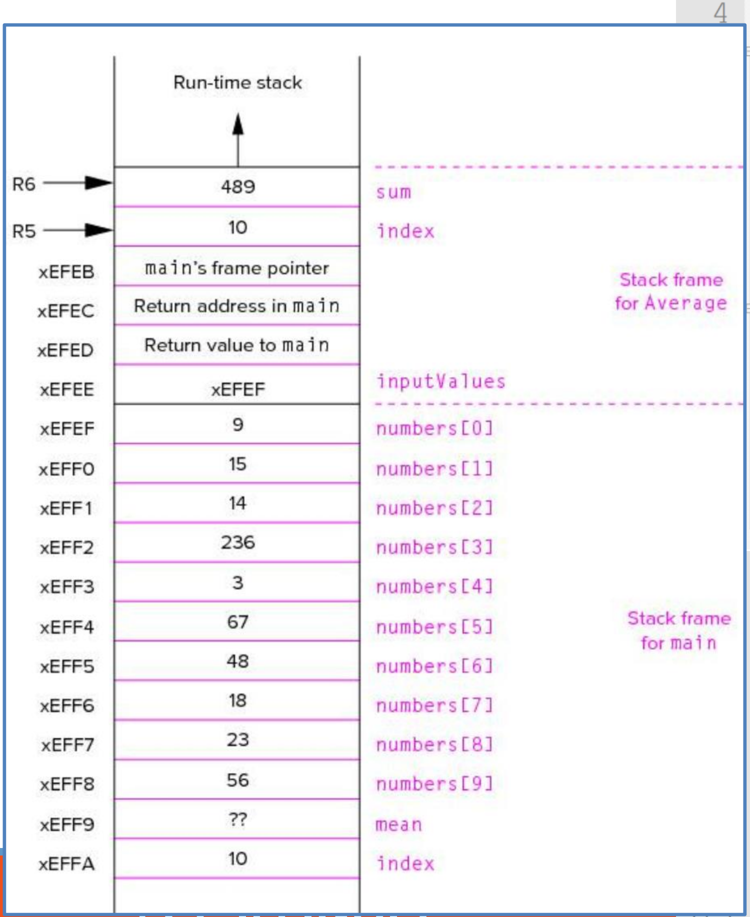
```
LDR R0, R5, #-10   ; Load the value of x
ADD R1, R5, #-9    ; Put the base address of grid into R1
ADD R1, R0, R1     ; Calculate address of grid[x]
LDR R2, R1, #0     ; R2 <-- grid[x]
ADD R2, R2, #2     ; R2 <-- grid[x] + 2
LDR R0, R5, #-10   ; Load the value of x
ADD R0, R0, #1     ; R0 <-- x + 1
ADD R1, R5, #-9    ; Put the base address of grid into R1
ADD R1, R0, R1     ; Calculate address of grid[x+1]
STR R2, R1, #0     ; grid[x+1] = grid[x] + 2;
```

Arrays Example

- **Declaring and using Arrays**

```
Int grid[10]= {5,7,8,9,10,11,12,2,3,1};
int grid1[10] = {0,1,2,3,4,5,6,7,8,9};
grid1[6] = grid1[3] + 1;
int i;
for(i=0;i<10;i++)
{
   grid[i] = grid[i]+grid1[i];
}
```

Passing Array as Arguments

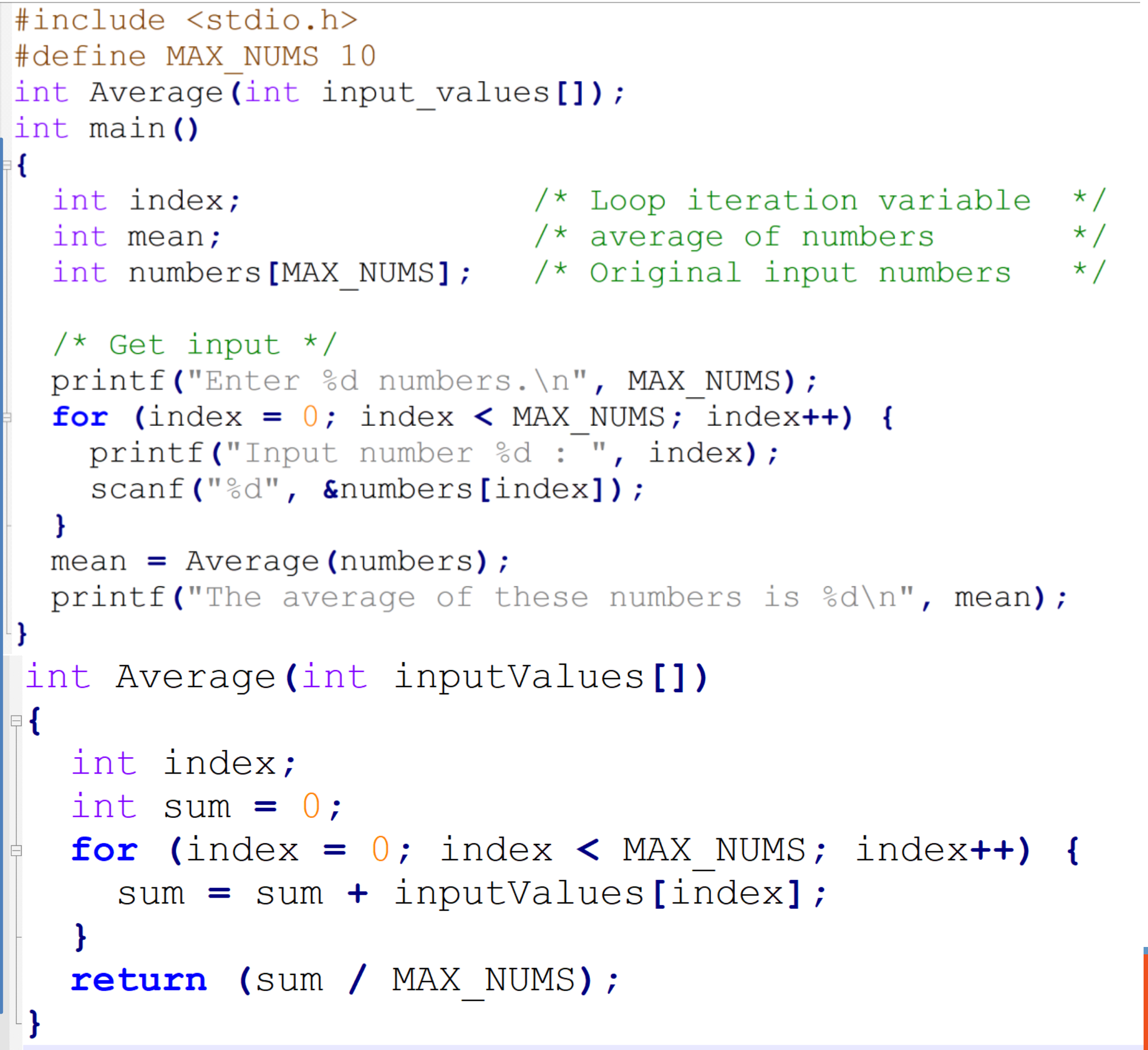


Run-time stack

R6	489	sum
R5	10	index
xEFEB	main's frame pointer	
xEFEC	Return address in main	
xEFED	Return value to main	
xEFEE	xEFEF	inputValues
xEFEF	9	numbers[0]
xEFF0	15	numbers[1]
xEFF1	14	numbers[2]
xEFF2	236	numbers[3]
xEFF3	3	numbers[4]
xEFF4	67	numbers[5]
xEFF5	48	numbers[6]
xEFF6	18	numbers[7]
xEFF7	23	numbers[8]
xEFF8	56	numbers[9]
xEFF9	??	mean
xEFFA	10	index

Stack frame for Average

Stack frame for main



```c
#include <stdio.h>
#define MAX_NUMS 10
int Average(int input_values[]);
int main()
{
  int index;                     /* Loop iteration variable  */
  int mean;                      /* average of numbers       */
  int numbers[MAX_NUMS];         /* Original input numbers   */

  /* Get input */
  printf("Enter %d numbers.\n", MAX_NUMS);
  for (index = 0; index < MAX_NUMS; index++) {
    printf("Input number %d : ", index);
    scanf("%d", &numbers[index]);
  }
  mean = Average(numbers);
  printf("The average of these numbers is %d\n", mean);
}
int Average(int inputValues[])
{
  int index;
  int sum = 0;
  for (index = 0; index < MAX_NUMS; index++) {
    sum = sum + inputValues[index];
  }
  return (sum / MAX_NUMS);
}
```