LAB Exam duration: **2:15 PM - 5:00 PM**.

On Tuesday, 8th Sep @ 2:15 PM

B1,B2,B3 (Monday Lab Batch):

Reporting at **Computer Center Lab CC L2**

B4,B5,B6 (Tuesday Lab Batch):

Reporting at **New Core Labs**

On Wednesday, 9th Sep @ 2:15 PM

B7,B8,B9 (Thursday Lab Batch):

Reporting at **Computer Center Lab CC L2**

B10,B11,B12 (Wednesday Lab Batch):

Reporting at **New Core Labs**

Comma– as an operator

- Comma as an operator is a binary operator that takes two expressions as operands.

```
expr1 , expr2
```

- Think of , just like + or – or * or / or = or == etc.. Some examples,
  1. i+2, sum=sum-1;
  2. scanf("%d",&m), sum=0, i=0;
- Execution of expr1 , expr2 proceeds as follows.
- Evaluate expr1, discard its result and then evaluate expr2 and return its value (and type).

Comma Operator execution

- Commas are evaluated from *left to right*. That is,

  scanf("%d",&m), sum=0, i=0;

  is executed as

  (scanf("%d",&m), sum=0), i=0;

- The comma operator has the lowest precedence of all operators in C. So

  a=a+5, sum = sum + a

  is equivalent to

  (a=a+5), (sum = sum + a)

```
int a = 1; int sum = 5;
a=a+5, sum = sum +
a;
```

Assignment operators

- i = i+10; can be shortened to

  i += 10;
- += is a new *assignment operator*.
- Similarly, +=, -=, *=, /=, %=
- expr1 op= expr2 is equivalent to

  expr1 = (expr1) op (expr2).
- Eg. x %= y+1 is x = x%(y+1) .
- Precedence rules are the same as that of = . (Right to left assoc.)

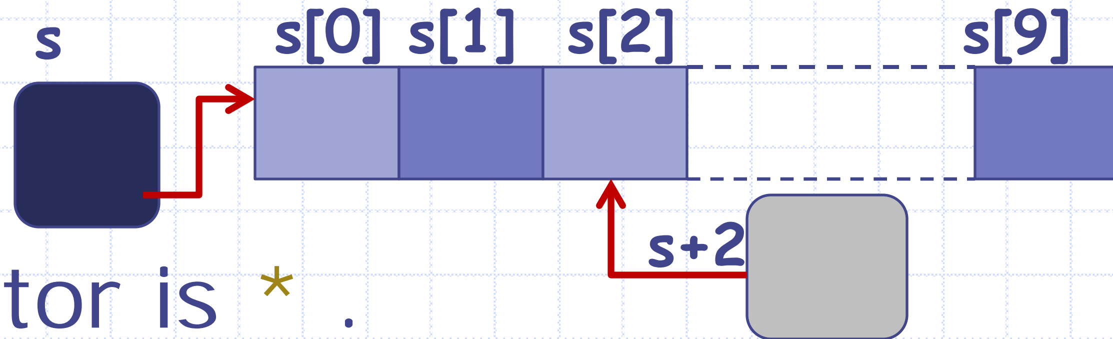
(In)(De)crement operators

- What is the difference between `i++` and `++i` in C ?
- The expression (i++) has the
  - value i
  - side-effect i=i+1
- The expression (++i) has the
  - value i+1
  - side-effect i=i+1
- Eg. (i == ++i) is always FALSE.
- Eg. (i == i++) is always TRUE.

Dereferencing operators

```
int main() {
  int s[10];
  read_into_array(s,10);
  .........
```

- For an array we have seen that [] acts as a dereferencing operator.



- Another such operator is * .
  - ✓ Can act on an array address.
- Eg. s[2] is the same as *(s+2).

Dereferencing operators

Quiz:

Is `3[s] = s[2] + 2;` a valid C expression?

- C Explanation: `3[s] = *(s+3) = s[3]`
- So the above simply updates `s[3]` to `s[2]+2` .

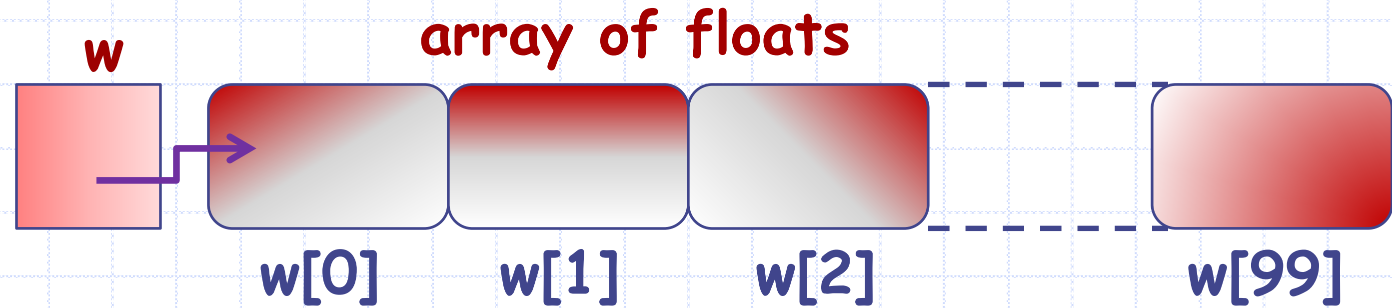
ESC101: Introduction to Computing

Strings

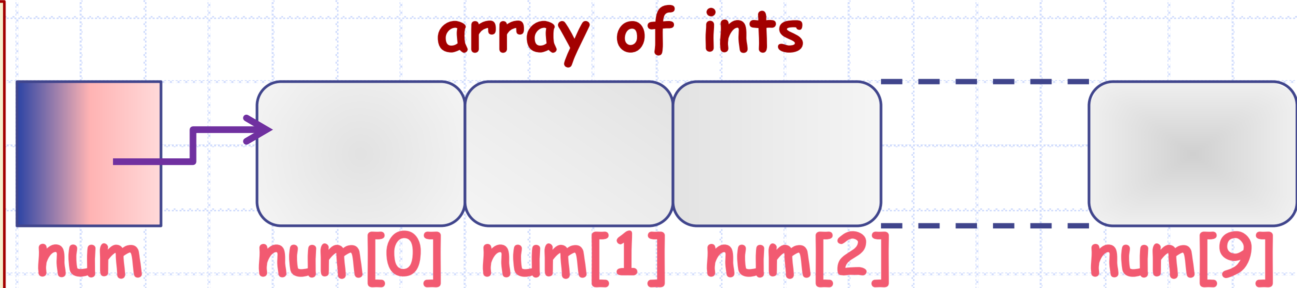
Recap about arrays

Basics: Arrays are defined as follows.

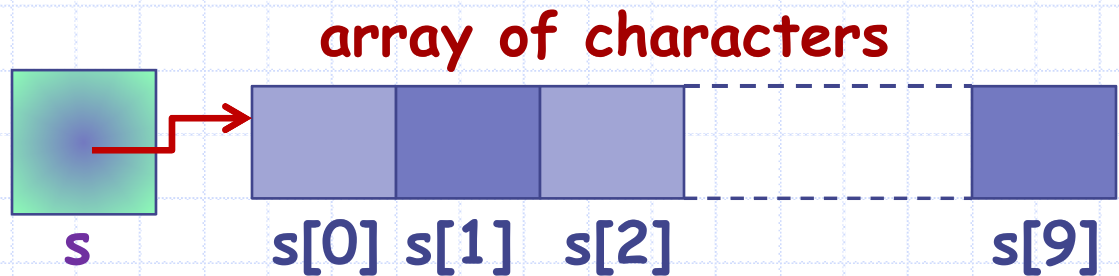
```
float w[100];
int num[10];
char s[10];
….
```



float w[100] defines 100 variables of type float. Their names are indexed: w[0],w[2],…w[99]



It also defines a variable called w which stores the address of w[0].



How can we create an int array num[] and initialize it to:

Method 1

```
int  num[] = {-2,3,5,-7,19, 103, 11};
```

1. Initial values are placed within curly braces separated by commas.
2. The size of the array **need not be specified**. It is set to the number of initial values provided.
3. Array elements are assigned in sequence in the index order. First constant is assigned to array element [0], second constant to [1], etc..

Method 2

```
int  num[10] = {-2,3,5, -7, 19, 103, 11};
```

Specify the array size. **size must be at least equal to the number of initialized values.** Array elements assigned in index order. Remaining elements are set to 0.

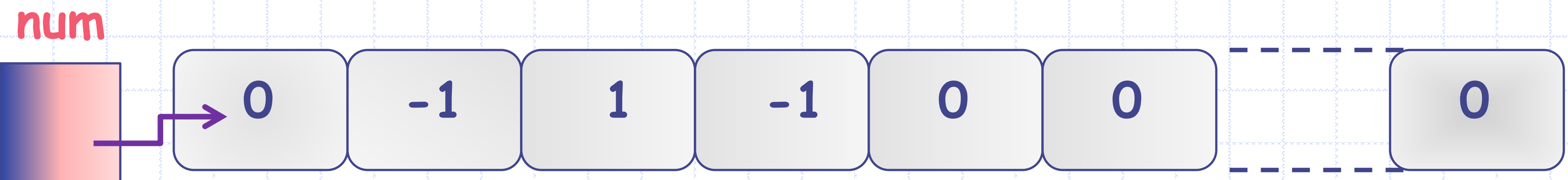
Recommended method: array size determined from the number of initialization values.

```
int  num[] = {-2,3,5,-7,19,103,11};
```

Is this correct?

```
int num[100] ={0,-1,1,-1};
```

YES! Creates num as an array of size 100. First 4 entries are initialized as given. num[4] … num[99] are set to 0.



Is this correct?

NO! it won't compile!

```
int  num[6] = {-2,3,5,-7,19,103,11};
```

Why?

1. num is declared to be an int array of size 6 but 7 values have been initialized.
2. Number of initial values must be less than equal to the size specified.

Initialization values could be constants or **constant expressions**. Constant expressions are expressions built out of constants.

```
int num[] = { 109, 7+3, 25*1023 };
```

Type of each initialization constant should be promotable/demote-able to array element type.

E.g.,

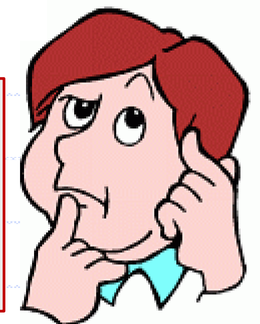
```
int num[] =  { 1.09, 'A', 25.05};
```

Float constants 1.09 and 25.05 downgraded to int

Would this work?

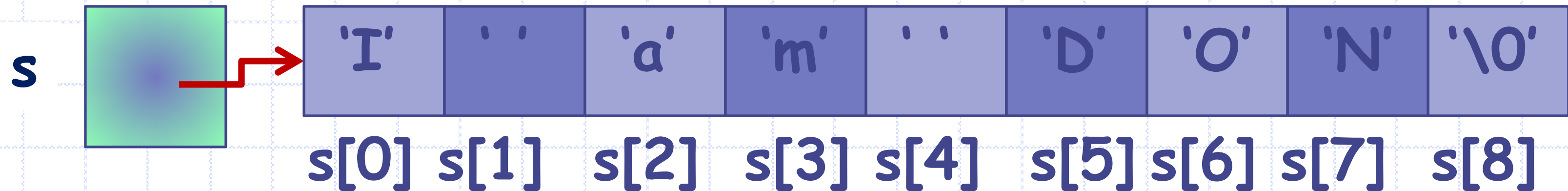
```
int curr = 5;
int num[] = { 2, curr*curr+5};
```

YES! ANSI C allows constant expressions **AND simple expressions** for initialization values. "Simple" is compiler dependent.



Character array initialization

Character arrays may be initialized like arrays of any other type. Suppose we want the following char array.



We can write:

```
s[]={'I',' ','a',' m',' ','D','O','N','\0' };
```

BUT! C allows us to define **string constants**. We can also write:

```
s[] = "I am DON";
```

1. "I am DON" is a **string** constant. Strings constants in C are specified by enclosing in double quotes.
2. It is equivalent to a character array *ending* with '\0'.
3. The '\0' character (also called NULL char) is automatically added to the end.

Printing strings

We have used string constants many times. Can you recall?

printf and scanf: the first argument is always a string.

1. `printf("The value is %d\n", value);`
2. `scanf("%d", &value);`

Strings are printed using %s option.

E.g. 1

```
printf("%s", "I am DON");
```

Output

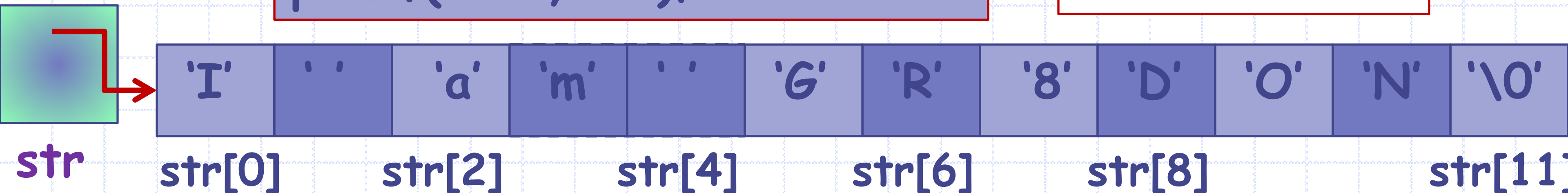
I am DON

E.g. 2

```
char str[]="I am GR8DON";
printf("%s", str);
```

Output

I am GR8DON



State of memory after definition of str in E.g. 2. Note the NULL char added in the end.

This NULL char is not printed.