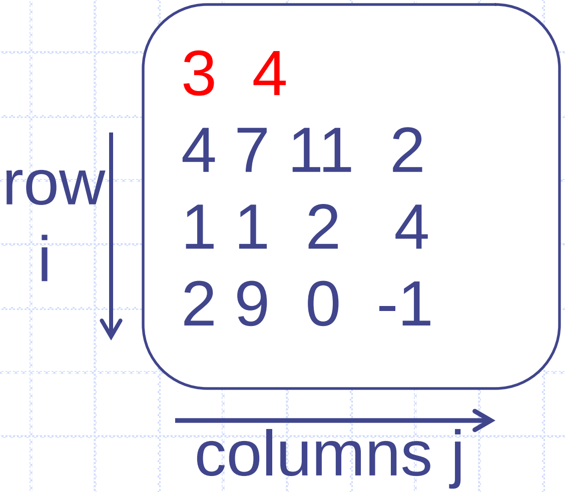
ESC101: Introduction to Computing

Recap & More

Problem 1

- The first line of the input consists of two positive integers m and n.
- This line is followed by m lines, each containing n integers, signifying an m X n matrix A. Calculate

$$\Sigma_i(\Sigma_j A_{ij})^2 \ , 1 \le i < m \ , 1 \le j < n \ .$$



```
3  4
4 7 11  2
1 1  2   4
2 9  0  -1
```

Desired output
$(4+7+11+2)^2 + (1+1+2+4)^2 + (2+9+0+ (-1))^2$

e.g. $A_{20} = 2$, $A_{12} = 2$

Double loops

- Need something of a double loop here (loop inside a loop).
- One loop to do the row sum of each row.
- Once a row is finished, we square the row sum.
- Another (outer) loop to add the squares of row sum over all rows that have been fully read.

Inner loop: Row sum

- Easy part first: assume we are at the beginning of a row (have not read any numbers yet) and write a loop to calculate the row sum.

```
int a;  /* the current integer */
int colindex; /* index of current column */
int rowsum;   /* sum of row entries read so far */
int rowsumsq;  /* square of the sum of row entries */
rowsum = 0;
colindex = 0;
while (colindex < n) {  /* not finished reading n cols*/
  scanf(“%d”, &a);   /* read next number  */
  rowsum = rowsum + a;    /* add to rowsum */
  colindex = colindex + 1;   /* increment colindex */
}
rowsumsq = rowsum * rowsum;    /*square rowsum */
```

Outer Loop Structure

- We have a code that reads the next n integers from the terminal and sums them.
- Modify it so that it reads the next m integers from the output of the previous code, specifically the value of rowsumsq and sums them.

- Task: Modify code below so that it reads the next m integers from the output of the previous code, specifically the value of rowsumsq and sums them.

```
int a;  /* the current integer */
int colindex; /* index of current column */
int rowsum;   /* sum of row entries read so far */
int rowsumsq;  /* square of the sum of row entries */
rowsum = 0;
colindex = 0;
while (colindex < n) {  /* not finished reading n cols*/
  scanf(“%d”, &a);    /* read next number  */
  rowsum = rowsum + a;    /* add to rowsum */
  colindex = colindex + 1;    /* increment colindex */
}
rowsumsq = rowsum * rowsum;    /*square rowsum */
```

- Previous code modified to read the next m integers from the output of the previous code, specifically the value of rowsumsq and sums them.

Outer Loop: Still in Design Phase: **incomplete and informal**

```
int rowindex;    /* index of current row being read */
int sqsum;       /* sum of col entries read so far */
sqsum = 0;
rowindex = 0;
while (rowindex < m) {   /* not finished reading m rows*/
   sqsum = sqsum + ``rowsumsq'';    /* add to sqsum */
   rowindex = rowindex + 1;    /* increment rowindex */
}
printf("%d ",sqsum);
```

rowsumsq comes from previous code. Let's insert that code here.

Inner, Outer Loops Implemented

```
int rowindex=0;    /* index of current row being read */
int sqsum=0;       /* sum of col entries read so far */

while (rowindex < m) { /* not finished reading m rows*/
 int rowsum=0;    /* sum of row entries read so far */
 int a;  /* the current integer */
 int colindex=0; /* index of current column */
 int rowsumsq;  /* square of the sum of row entries */
 while (colindex < n) {  /* not finished reading n cols*/
  scanf(“%d”, &a);    /* read next number  */
  rowsum = rowsum + a;     /* add to rowsum */
  colindex++;   /* increment colindex */
 }
 rowsumsq = rowsum * rowsum;    /*square rowsum */

 sqsum = sqsum + rowsumsq;    /* add to sqsum */
 rowindex++;   /* increment rowindex */
}
printf(“%d “,sqsum);
```

Problem 2

- Read n, assume n >=2. Read n integers, and print triplets of consecutively positive input integers that are Pythogorean, skipping negative ints. For input

```
8 1 -1 3 -3 4 -4 -5 5
```

- Output should be

```
3 4 5
```

- Need a single loop, but *several* counters.

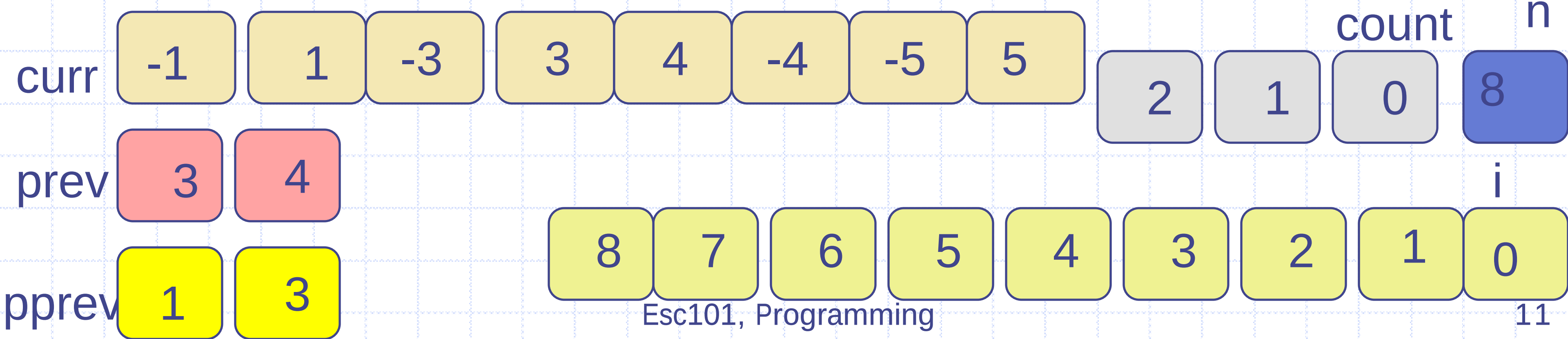
```
int curr, prev, pprev;/* current, prev, pprev positive nos.*/
int n;                /*  number of integers */
int i;                /* for loop counter */
int count = 0;        /* no. of  positive ints  seen yet */
scanf(“%d”, &n);
for (i=0; i < n ; i++)   {
   scanf(“%d”, &curr);
   if (curr <= 0) continue;  /* skip non-positive nos. */
   if (count == 0) { pprev = curr; count =1; }
   else {
       if (count == 1) { prev = curr; count =2; }
       else{                /* count  is 2 and will  remain 2 */
           if (pprev*pprev + prev*prev == curr*curr){
                                   /* Pythagorean triple found */
              printf(“%d  %d  %d\n”, pprev, prev, curr);}
              pprev = prev;
               prev = curr;
           }
       }
   }
} // end for loop
```

```
int curr, prev, pprev,n, i, count = 0;  scanf(“%d”, &n);
for (i=0;        i < n ;        i  = i+1)   {
    scanf(“%d”, &curr);
    if  (curr <= 0) {      continue; }
    if (count == 0) {      pprev = curr; count =1; }
              else { if (count == 1) {  prev = curr; count =2;
                     else  {  /* count  is 2 and will  remain 2 */
                         if (pprev*pprev + prev*prev == curr*curr){
                                printf(“%d  %d  %d\n”, pprev, prev, curr);}
                         pprev = prev;
                         prev = curr;
                     }
              }
}    }
```

Output

```
3  4   5
```

8 -1 1 -3 3 4 -4 -5 5



A general principle of program development

1. **Break up your task into smaller sub-tasks, and those sub-tasks into still smaller sub-tasks and so on until each sub-task is easily solvable in a function/block.**

2. **Write a function for each of the sub-tasks.**

3. **Design your program from the top-down, big task to the small tasks.**

   I. **Debug/test your program bottom-up.**
   II. **Debug functions that perform elementary tasks, and then move on to testing more complex functions. (Commonly, printf is used.)**

Example 3

What is printed by the program?

```
int f (int a, int b) {
    return a+b;
}
```

```
main () {
    int a = 1, b = 2;
    a = f(f(a,b),b);
    printf(“%d  %d”, a,b);
}
```

1. Execution proceeds similar to evaluating mathematical function expression.
2. Care needed to handle expr with side effects.

Evaluation of f(f(a,b),b)

1. First evaluate inner f(a,b) for a = 1, b=2.
2. This is 3.
3. So f(f(a,b),b) becomes f(3,2).
4. This is 5.
5. So output is 5 2

Pure expressions do not change the state of the program, e.g.,

1. a- b *c/d
2. f(f(a,b), f(f(a,b),a))

Expressions with side-effects change the state of the program for example,

1. a = a +1
2. f(a=b+1,b=a+1)

Example 4

What is printed by the program?

```
int f (int a, int b) {
        return b-a;
}
```

```
main () {
    int a = 2, b = 1;

    a = f( a=b+1,    b=a+1);

   printf("%d  %d", a,b);
}
```

Evaluate f(a=b+1,b=a+1).

How should we evaluate it?

BUT!

Rule: All arguments are evaluated before function call is made.

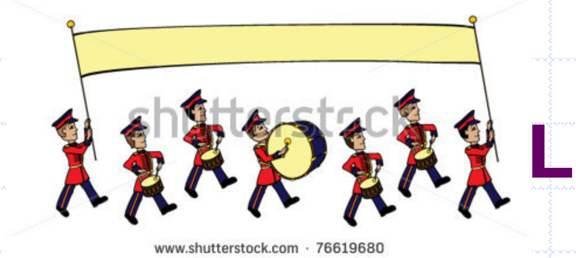
C doesn't specify order, in which arguments are evaluated. This is left to the compiler.

Let us evaluate function arguments in left to right order.

main()			
	a	2	2
	b	1	3

Expected Output

1 3



Left-right OR right-left

```
int f (int a, int b) {
          return b-a;
}
```

```
main () {
      int a = 2, b = 1;

      a = f( a=b+1,    b=a+1);

      printf("%d    %d", a,b);
}
```

a	2	4	-1
b	1	3	

We used left to right evaluation. Expected output: **1 3**

Let us compile and run on a CC machine, output is: **-1 3**

What happened? The compiler evaluated right to left.

Output is **-1 3**

What was the mistake?

- Actually, C does not specify the order in which the arguments of a function should be evaluated.
- It leaves it to the compiler. Compilers may evaluate arguments in different orders.
- Both answers are consistent with C language!! What should we do?

Write your arguments to functions so that the result is not dependent on the order in which they are evaluated. Better still, write them so that the operand expressions are side-effect free.

For example

```
int f (int a, int b) {
       return b-a;
}
```

```
main () {
    int a = 2, b = 1;
    a=b+1;
    b=a+1;

    a = f( a, b ); /* operands do not have
                      side effects */

   printf(“%d  %d”, a,b);
}
```

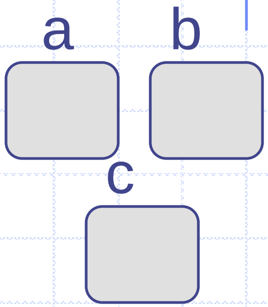
Comma– as a separator

- C allows multiple variables of the **same** type to be defined as one statement, separated by commas.

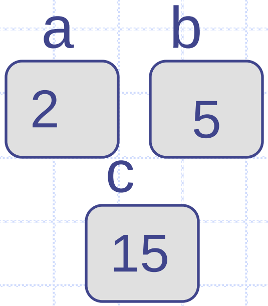
Examples (independent definitions)

```
int a, b, c; ✓
int a = 2, b = 5, c=15; ✓
float x = 3.59, y = 4.5; ✓
int x = 5, float y = 10.0; ✗
```

Defines three integer variables named a,b and c.

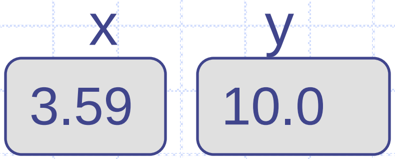


Defines three integer variables named a,b and c. Initializes a to 2, b to 5 and c to 15.



Compilation error!

Defines two float variables named x and y. Initializes x to 3.59 and y to 10.0.



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).