```
int n; int x;
scanf("%d", &n);     // input n

x = 1;                    // [while] initialization
while ( x <= n) {    // [while] cond

    if ((x%3 == 0) || (x%5 == 0)) { // [if] cond
        printf("%d\n", x);
    }

    x = x+1;              // [while] update
}
```

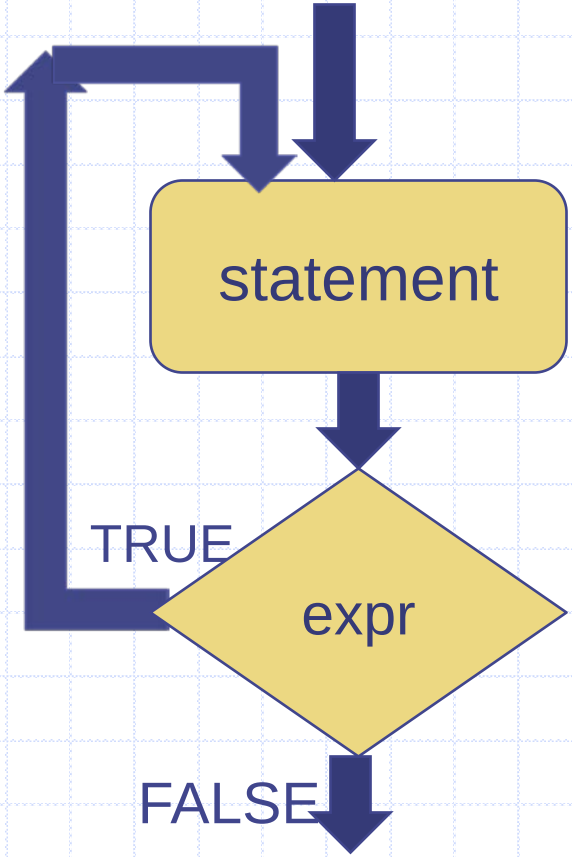
# do-while loops

- do-while statement is a variant of while.

  General form:

```
do
    statement
while (expr);
```

- Execution:
  1. First execute statement.
  2. Then evaluate expr.
  3. If expr is TRUE then go to step 1.
  4. If expr is FALSE then break from loop




- Continuation of loop is tested after the statement.

# Comparing while and do-while

- In a while loop the body of the loop may not get executed even once, whereas, in a do-while loop the body of the loop gets executed at least once.
- In the do-while loop structure, there is a semicolon after the condition of the loop.
- Rest is similar to a while loop.

# Comparative Example

- Problem: Read integers and output each integer until -1 is seen (include -1 in output).
- The program fragments using while and do-while.

Using while

```
int a;/*current int*/

scanf(“%d”,&a);
while (a != -1) {
   printf(“%d\n”, a);
   scanf(“%d”, &a);
}
printf(“%d\n”, a);
```

Using do-while

```
int a; /*current int*/

do {
   scanf(“%d”, &a);
   printf(“%d\n”, a);
} while (a != -1);
```

# Comparative Example

- The while construct and do-while are equally expressive
  - whatever one does, the other can too.
  - but one may be *more readable* than other.

Using do-while

```
int a; /*current int*/

do {
   scanf(“%d”, &a);
   printf(“%d\n”, a);
} while (a != -1);
```

Using while

```
int a;/*current int*/

scanf(“%d”,&a);
while (a != -1) {
   printf(“%d\n”, a);
   scanf(“%d”, &a);
}
printf(“%d\n”, a);
```

# For Loop

- Print the sum of the reciprocals of the first 100 natural numbers.

```
int i; // counter from 1..100
float rsum = 0.0;// the sum

// the for loop
for ( i=1; i<=100; i=i+1 ) {
    rsum = rsum +  (1.0/i);
}
printf(“sum is %f “, rsum);
```

# For loop in C

- General form

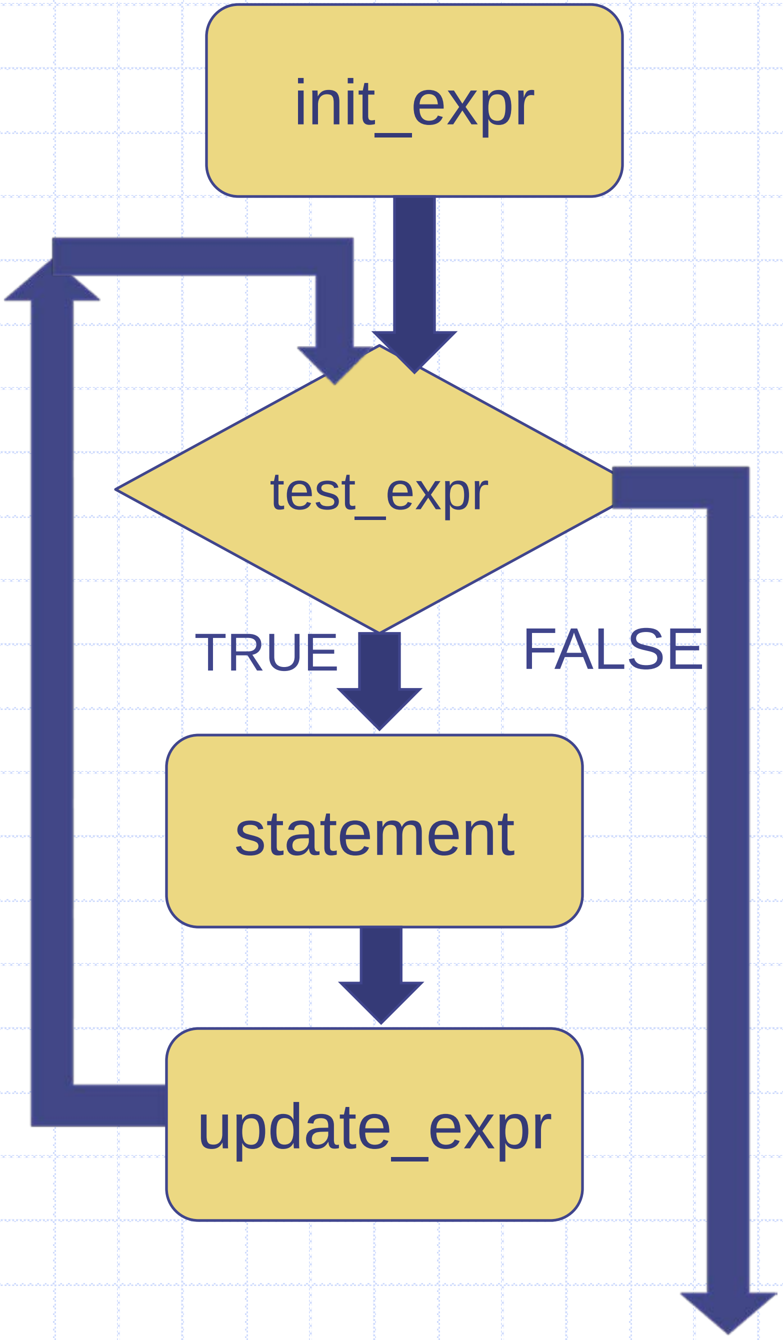
```
for (init_expr; test_expr; update_expr)
   statement;
```

- init_expr is the initialization expression.
- update_expr is the update expression.
- test_expr is the expression that evaluates to either TRUE (non-zero) or FALSE (zero).
- statement is the work to repeat (can be multiple statements in {...} )

# For loop in C

```
for (init_expr; test_expr; update_expr)
        statement;
```

1. First evaluate **init_expr**;
2. Evaluate **test_expr**;
3. If **test_expr** is TRUE then
   a) execute **statement**;
   b) execute **update_expr**;
   c) go to Step 2.
4. if **test_expr** is FALSE then break from the loop



```c
int i;
float rsum = 0.0;

for (i=1; i<=4; i=i+1) {
    rsum = rsum + (1.0/i);
}

printf("sum is %f", rsum);
```

| i | rsum |
| --- | --- |
| 5 | 2.0833333.. |

1. Evaluate init_expr; i.e., i=1;
2. Evaluate test_expr i.e., i<=4 TRUE
3. Enter body of loop and execute.
4. Execute update_expr; i=i+1; i is 2
5. Evaluate test_expr i<=4: TRUE
6. Enter body of loop and execute.
7. Execute i=i+1; i is 3
8. Evaluate test_expr i<=4: TRUE
9. Enter body of loop and execute.
10. Execute i=i+1; i is 4
11. Evaluate test_expr i<=4 TRUE
12. Enter body of loop and execute.
13. Execute i=i+1; i is 5
14. Evaluate test_expr i<=4 FALSE
15. Exit loop & jump to printf

sum is 2.083333

# For loop in terms of while loop

```
for (init_expr; test_expr; update_expr)
    statement;
```

- Execution is (almost) equivalent to

```
init_expr;
while (test_expr) {
    statement;
    update_expr;
}
```

- Almost? Exception if there is a continue; inside statement– this will be covered later.
- Both are equivalent in power.
- Which loop structure to use, depends on the convenience of the programmer.

# Example: Geometric Progression

- Given positive real numbers $r$ and $a$, and a positive integer, $n$, the $n^{th}$ term of the geometric progression with $a$ as the first term and $r$ as the common ratio is $ar^{n-1}$.
- Write a program that given $r$, $a$, and $n$, displays the first $n$ terms of the corresponding geometric progression.

```
#include<stdio.h>
int main(){
    int n, i;      float r, a, term;

    // Reading inputs from the user
    scanf("%f", &r);
    scanf("%f", &a);
    scanf("%d", &n);
    term = a;
    for (i=1; i<=n; i=i+1) {
        printf("%f\n", term); // Displaying i^th term
        term = term * r;        // Computing (i+1)^th term
    }
    return 0;
}
```

```
#include<stdio.h>
int main(){
    int n, i;     float r, a, term;

    // Reading inputs from the u
    scanf("%f", &r);
    scanf("%f", &a);
    scanf("%d", &n);
    term = a;
    for (i=1; i<=n; i=i+1) {
        term = term * r;          // Computing $(i+1)^{th}$ term
        printf("%f\n", term); // Displaying $(i+1)^{th}$ term
    }
    return 0;
}
```

**Careful**: Changing the order of statements changes the meaning of the program. Computation of $a, ar, \ldots, ar^{n-1}$ vs. $ar, ar^2, \ldots, ar^n$

# Overflow

- The types like int, char, long can hold bounded values.
- A complex expression that produces a final value within bound might produce intermediate values that go beyond the bounds
  - Overflow
  - May result in incorrect final value
- Some tricks or simplification may be needed to get correct value

# Avoiding Overflow: Examples

- Permutation: ${}^{n}P_r = n!/(n-r)!$
- Computation of ${}^{100}P_2 = \frac{100!}{98!}$
  - If factorials are computed explicitly, may produce wrong result
    - 100! and 98! Are too big to be stored in long
  - But the result can be computed easily as 100*99

# Avoiding Overflow: Examples

- Terms in the series: $(x+1)^{2k}/(2k+1)!$
- Direct computation of $n^{th}$ term
  - May not "fit" in the data types
  - But the result can be computed precisely using the relation:

$$T_n = T_{n-1} * R$$

where

$$R = \frac{(x+1)^2}{2n \ * (2n+1)}$$

  - $T_n, R$ will fit in memory for very large $n$

# Nested Loops

- Loop with in a loop
- Many iterations of inner loop ⇒ One iteration of outer loop

# Example

- Write a program that displays the following pattern

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 2 | 4 | 6 | 8 | 10 |
| 3 | 6 | 9 | 12 | 15 |
| 4 | 8 | 12 | 16 | 20 |
| 5 | 10 | 15 | 20 | 25 |
| 6 | 12 | 18 | 24 | 30 |
| 7 | 14 | 21 | 28 | 35 |
| 8 | 16 | 24 | 32 | 40 |

integers are printed in 5 columns each

```
#include<stdio.h>
int main(){
    int i, j;

    for (i=1; i<=8; i=i+1) {
        for (j=1; j<=5; j=j+1) {
            printf("%4d", i*j); // Displaying i, 2i, ..., 5i
        }
        printf("\n"); // Move to the next line
    }

    return 0;
}
```

# Displaying a pattern

```
#include <stdio.h>
int main(){
  int i,j;
  for (i=1; i<=5; i=i+1){
    for (j=i; j<2*i; j=j+1){
      printf("%d ",j);
    }
    printf("\n");
  }
  return 0;
}
```

- Output?

1

2 3

3 4 5

4 5 6 7

5 6 7 8 9

# increment/decrement operator

- Two very common actions in C

```
i = i + 1;
i = i – 1;
```

- These can be written in short as:

```
i++      // increment
i--      // decrement
```

- Complete semantics are bit involved
  - Not covered in this course
  - Advise: Do not use them other than:
    - in update_expr of for/while loops
    - Standalone statements: i++;

# Continue and Break