Programming Fundamentals

- Lecturer: Dr. Arnold Meijster
- e-mail: a.meijster@rug.nl
- room: 5161.343 (Bernoulliborg)

More on `printf`: int format specifiers

`%c`	character
`%d`	signed decimal
`%ld`	long decimal
`%u`	unsigned decimal

```
printf("%c %d %u\n", 'e', -17, 17);
```

Output: `e -17 17`

Formatted Output: printf

- **Examples:**

```
printf("%d%d", 123, 456);  // prints: 123456

printf("%d%4d", 123, 456); // prints: 123 456

printf("%04d", 123);       // prints: 0123

printf("pi=%f and e=%f.\n", 3.1415926, 2.718);

printf("%.1f\n", 1.49);    // prints 1.5
```

More on `printf`: float specifiers

Standard precision is 11 positions (including decimal dot):

```
%f    ddd.ddd
%e    d.ddddde{sign}dd
%E    d.dddddE{sign}dd
```

```
printf("%f\n", 1234.56789);
printf("%5.3f\n", 1234.56789);
printf("%12.4e\n", 1234.56789);
```

```
1234.567890
1234.568
  1.2346e+03
```

(at least 5 positions, 3 digits after .)

(at least 12 positions, 4 digits after .)

Coercion: implicit type conversion

Coercion/Widening:

$$\begin{tikzcd}
\text{byte} \arrow[d] & \\
\text{short} \arrow[d] & \\
\text{int} \arrow[r] \arrow[d] & \text{float} \arrow[d] \\
\text{long} \arrow[r] & \text{double}
\end{tikzcd}$$

```
short x;
int y, a;
long b;
float t;
double r;

x = 1234;
y = 5674;
a = x + y;
b = a*x;
t = b/2.25;
r = t + 7;
```

Explicit type conversion: type cast

Coercion/Widening:

$$\begin{tikzcd}
\text{byte} \arrow[d] & \\
\text{short} \arrow[d] & \\
\text{int} \arrow[d] \arrow[r] & \text{float} \arrow[d] \\
\text{long} \arrow[r] & \text{double}
\end{tikzcd}$$

```
short x;
int y, a;
long b;
float t;
double r;

x = 1234;
y = 5674;
x = (short)y - 4*x;
b = y*x;
y = (int)b%y;
t = (float)(b/y);
r = (double)b/y;
y = (int)t;
```

Type conversion: type cast

- Beware! The typecast operator has a very high priority!
  - `(int)3.5*2` yields `6`
  - `(int)(3.5*2)` yields `7`

Example: sum of squares

Exercise: write a program fragment that sums the squares of all natural numbers less than 20.

0*0 + 1*1 + 2*2 + .. + 19*19

```
int sumSquares =
    0*0 + 1*1 + 2*2 + 3*3 + 4*4 + 5*5 +
    6*6 + 7*7 + 8*8 + 9*9 + 10*10 + 11*11 +
    12*12 + 13*13 + 14*14 + 15*15 + 16*16 +
    17*17 + 18*18 + 19*19;
printf("sumSquares = %d\n", sumSquares);
```

Example: sum of squares

```
int sumSquares = 0;
sumSquares +=   1*1;
sumSquares +=   2*2;
sumSquares +=   3*3;
sumSquares +=   4*4;
sumSquares +=   5*5;
sumSquares +=   6*6;
sumSquares +=   7*7;
sumSquares +=   8*8;
sumSquares +=   9*9;
sumSquares += 10*10;
sumSquares += 11*11;
sumSquares += 12*12;
sumSquares += 13*13;
sumSquares += 14*14;
sumSquares += 15*15;
sumSquares += 16*16;
sumSquares += 17*17;
sumSquares += 18*18;
sumSquares += 19*19;
printf("sumSquares = %d\n", sumSquares);
```

Example: sum of squares

For those who like math, it is well-known that

$$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

```
int n = 19;
printf("sumSquares = %d\n", n*(n+1)*(2*n+1)/6);
```

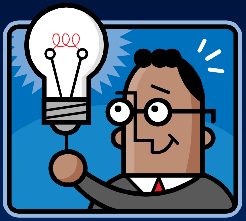
Example: sum of squares

Exercise: write a program fragment that sums the squares of all natural numbers less than 2000

(do not use the formula on the previous slide. It will overflow!)

0*0 + 1*1 + 2*2 + .. + 1999*1999

Example: sum of squares



Clearly, we need some sort of iteration.

C has several loop-constructs. One of them is the for-loop.

Idea: We use a variable `i` to iterate over the range [0..2000) and add `i*i` to `sumSquares`.

```
int i, sumSquares = 0;
for (i=0; i < 2000; i++) {
  sumSquares += i*i;
}
printf("sumSquares = %d\n", sumSquares);
```

Declaration of the loop variable

```
int i, sumSquares = 0;
for (i=0; i < 2000; i++) {
  sumSquares += i*i;
}
printf("sumSquares = %d\n", sumSquares);
```

```
int sumSquares = 0;
for (int i=0; i < 2000; i++) {
  sumSquares += i*i;
}
// At this point the loop variable i does not exist
printf("sumSquares = %d\n", sumSquares);
```

For-statement

The general syntax of the for-statement is:

```
for (initialisation; condition; update) {
    body;
}
```

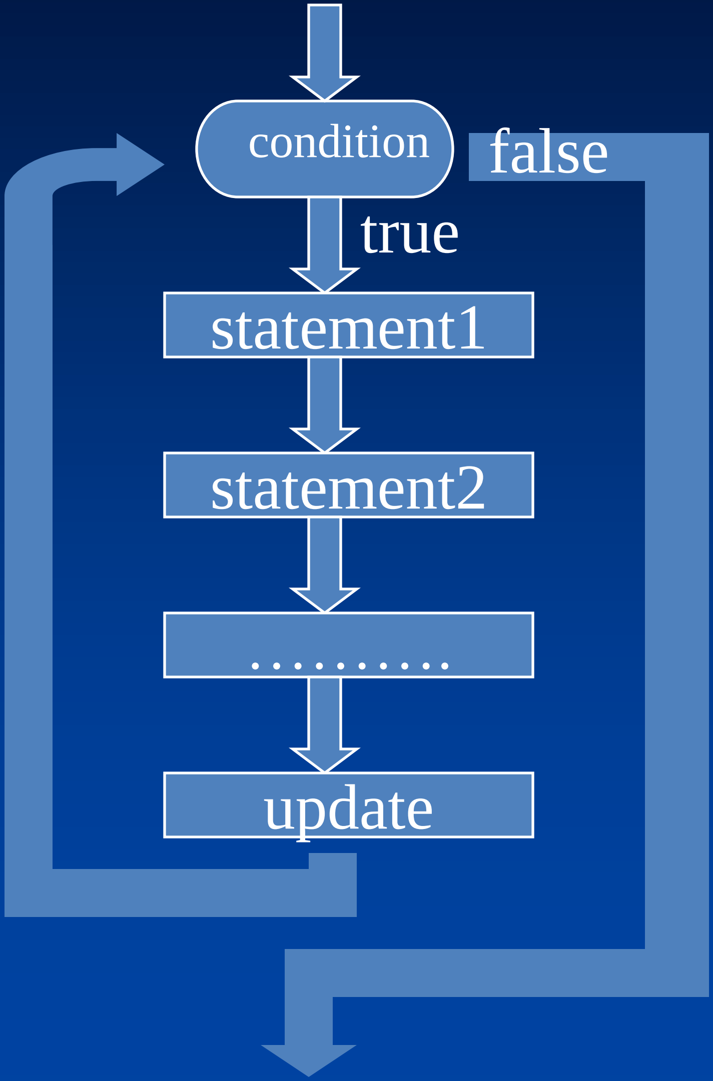
First, the statement **`initialisation`** is executed.

Before each iteration (also the first one!), the boolean **`condition`** is evaluated. This condition is also known as the *guard* of the loop. If it is true (i.e. non-zero value), then the statements of the **`body`** are executed, followed by execution of **`update`**.

If **`condition`** evaluates to **`0`** (i.e. false), then the loop stops and execution of the program continues directly after the for-loop.

For-statement

```
for (initialisation; condition; update) {
    statement1;
    statement2;
    ..
}
```



Example: it’s full of stars

Exercise: write a program fragment that reads a non-negative integer `n` from the input and prints a series of `n` stars (asterisks, *) on the output.

```
int n;
scanf("%d", &n);
for (int i=0; i < n; i++) {
  putchar('*');     // or printf("*");
}
```

Note that, at the beginning of each iteration, the value of `i` is equal to number of printed stars (so far).

Example: it’s full of stars (2)

An alternative solution:

```
int n;
scanf("%d", &n);
for (int i=n; i > 0; i--) {
  putchar('*');
}
```

This time, at the beginning of each iteration, the value of `i` is equal to the number of stars that still need to be printed.

Example: it’s full of stars (3)

A 3rd alternative:

```
int n;
scanf("%d", &n);
for (; n > 0; n--) {
  putchar('*');
}
```

Note that the initialization of the for-loop may be empty.

Example: product of odd integers

Exercise: write a program fragment that computes the product of all odd natural numbers less than some value `lim`.

```
int oddProduct = 1; /* note the 1 (not 0) */
for (int i=0; i < lim; i++) {
  if (i%2 == 1) {
    oddProd *= i;
  }
}
```

Example: product of odd integers

A nicer (and more efficient) solution is:

```
int oddProd = 1;
for (int i=3; i < lim; i+=2) {
  oddProd *= i;
}
```

`continue`-statement

Sometimes we want to ‘early’ continue to the next iteration. The continue statement does exactly that. It can be used with any type of loop: for, while, do-while.

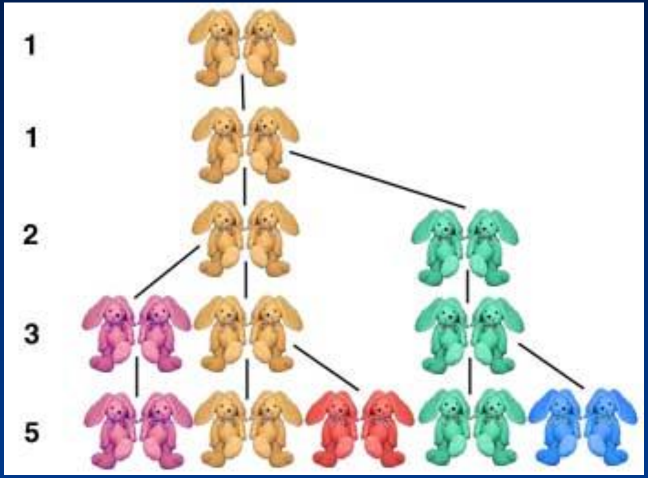
Exercise: write a program fragment that computes the product of all odd natural numbers less than some value `lim` that are not divisible by 7.

```
int oddProd = 1;
for (int i=3; i < lim; i+=2) {
  if (i%7 == 0) {
    continue;
  }
  oddProd *= i;
}
```

Fibonacci: one, one, two, three, five, …

We start with one pair of young rabbits. A rabbit is mature after one year. Each pair of mature rabbits 'produces' one pair of young rabbits:

year	young	mature	total
0	1	0	1
1	0	1	1
2	1	1	2
3	1	2	3
4	2	3	5
5	3	5	8

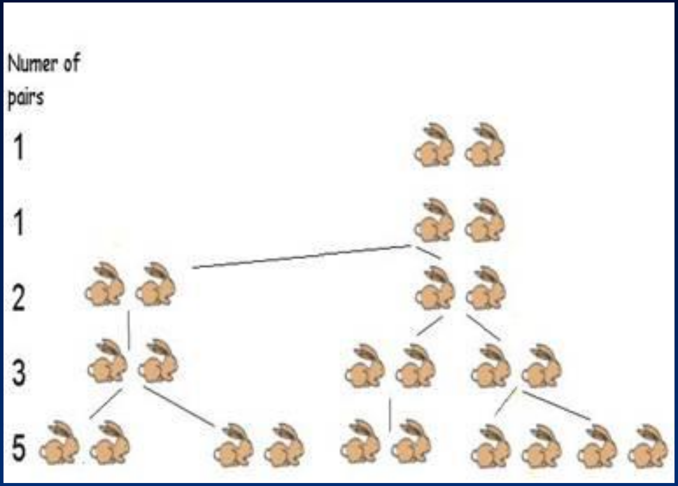


Exercise: write a program fragment that, given a non-negative integer `n`, prints the above table for the years `0` upto `n`:

Note that:

year	young	mature
y	a	b
y + 1	b	a + b

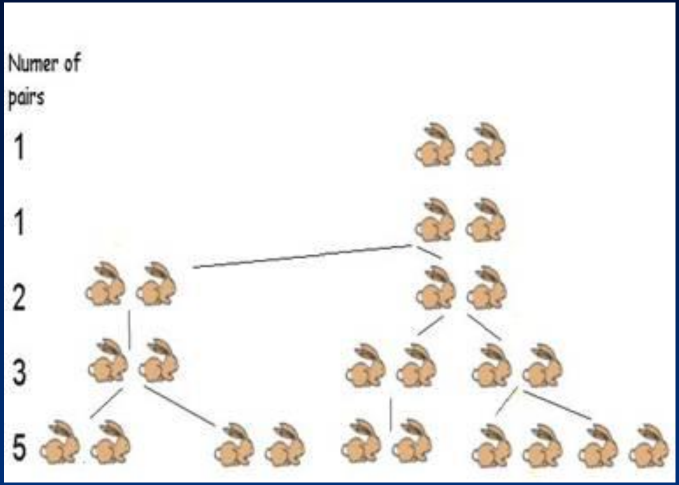
Fibonacci: one, one, two, three, five, …



```
int young = 1;
int mature = 0;

for (int year=0; year < n; year++) {
  printf("%d\t%d\t%d\t%d\n",
         year, young, mature, young + mature);
    /* young == A, mature == B */
    /* young == A, young + mature == A + B */
  mature = young + mature;
    /* young == A, mature == A + B */
    /* mature - young == B, mature == A + B */
  young = mature - young;
    /* young == B, mature == A + B */
}
```

Fibonacci: one, one, two, three, five, …



```
int young = 1;
int mature = 0;

for (int year=0; year < n; year++) {
  printf("%d\t%d\t%d\t%d\n",
         year, young, mature, young + mature);
  mature = young + mature;
  young = mature - young;
}
```

Fibonacci: one, one, two, three, five, …

Variation: In which year do we reach a total number of pairs that is at least (a given) **`n`**?

```
int young = 1;
int mature = 0;
int year = 0;
int n;
scanf("%d", &n);
while (young + mature < n) {
  mature = mature + young;
  young = mature - young;
  year++;
}
/* here: young + mature >= n  (logical negation of guard) */
printf("In year %d we have at least %d pairs.\n", year, n);
```

While-statement

The syntax of the while-statement is:

```
while (condition) {
    body;
}
```

Before each iteration (also the first one!), the boolean `condition` is evaluated. If it is true (i.e. non-zero), then the statements of the `body` are executed.

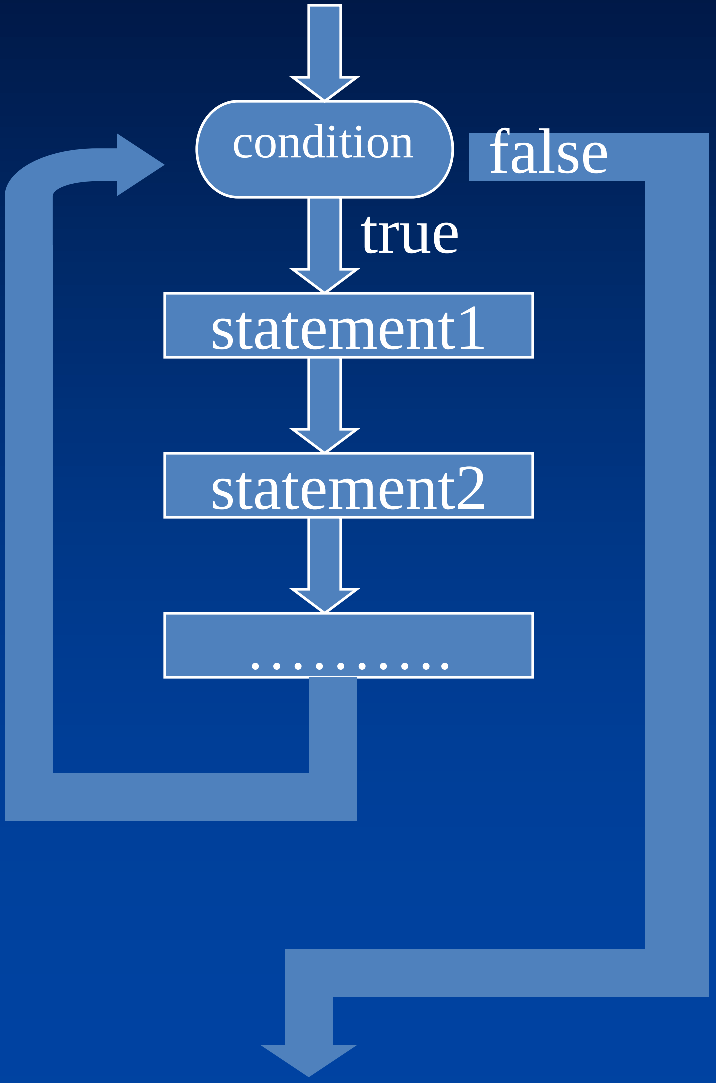
If `condition` evaluates to `0` (i.e. false), then the loop stops and execution of the program continues directly after the while-loop.

The while-loop and the for-loop are very similar. In fact, the general form above can be written as a for-loop with an empty initialisation and an empty update (not very stylish):

```
for (; condition; ) {
    body;
}
```

While-statement

```
while (condition) {
      statement1;
      statement2;
      ..
   }
```



Summing the input

Exercise: write a program fragment that reads a series of integers from the input, and outputs the sum of the numbers. The series is terminated by a zero.

```
int number, sum = 0;

scanf("%d", &number);
while (number != 0) {
  sum += number;
  scanf("%d", &number);
}
printf("sum=%d\n", sum);
```

```
int number, sum = 0;

scanf("%d", &number);
while (number) {
  sum += number;
  scanf("%d", &number);
}
printf("sum=%d\n", sum);
```

Do-While-statement

The syntax of the do-while-statement is:

```
do {
   body;
} while (condition);
```

First, the `body` is executed (without testing the `condition`).
Note that the body of the loop is executed at least once!

At the end of each iteration, the boolean `condition` is evaluated. If it is true (i.e. non-zero), then the statements of the `body` are executed again.

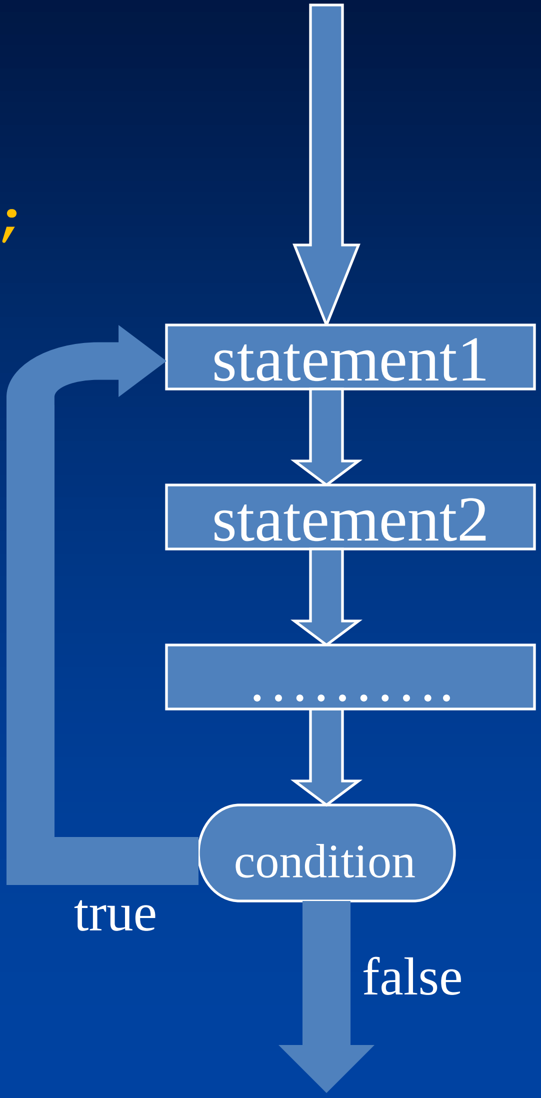
If `condition` evaluates to `0` (i.e. false), then the loop stops and execution of the program continues directly after the do-while-loop.

Equivalent with:

```
body;
while(condition){
  body;
}
```

Do-While-statement

```
do {
      statement1;
      statement2;
      ..
} while (condition) ;
```



Summing the input (II)

Exercise: write a program fragment that reads a series of integers from the input, and outputs the sum of the numbers. The series is terminated by a zero.

```
int number, sum = 0;

scanf("%d", &number);
while (number) {
   sum += number;
   scanf("%d", &number);
}
printf("sum=%d\n", sum);
```

```
int number, sum = 0;

do {
   scanf("%d", &number);
   sum += number;
} while (number);
printf("sum=%d\n", sum);
```

Example: read input

Ask the user to type in a positive integer ≥ 1:

```
int number;

do {
  printf ("Please, type an integer >=1: ");
  scanf("%d", &number);
  if (number < 1) {
    printf("Number is not >= 1, try again.\n");
  }
} while (number < 1);
```

`break`-statement

Sometimes we want to ‘break’ out of a loop.

In C you can do this with the `break`-statement.
It can be used with any type of loop: for, while, do-while.

It is almost always used in combination with an if-statement.

A `break`-statement has the same effect as normal loop termination: the loop stops, and execution of the program continues directly after the loop.

Example: grade calculator

For some course, a teacher has a list of integer grades. There are three grades per student. To help him calculate the final grades, we write a program (fragment) that repeatedly reads three integer valued grades **`(x, y, z)`** from the input and then prints the weighted average using the factors 0.2, 0.3 and 0.5. The program should stop if the entered value for **`x`** is zero.

```
int x;
float grade;
while (1) {  /* infinite loop, alternative is for(;;) */
  printf("1st grade: ");
  scanf("%d", &x);
  if (x == 0) {  /* alternative test: !x */
    break;
  }
  grade = 0.2*x;
  printf("2nd grade: ");
  scanf("%d", &x);
  grade += 0.3*x;
  printf("3rd grade: ");
  scanf("%d", &x);
  grade += 0.5*x;
  printf ("Weighted average: %2.1f\n", grade);
}
```

Which loop to use?

This is a matter of style! Each of the three loop-constructs can be expressed as the other two. So, you could say that they are equivalent.

Still, there is a preference based on style arguments:

- **for-statement:** use it when we know the number of iterations in advance.
- **while-statement**: use it when we do not know the number of iterations in advance.
- **do-while-statement**: use it when we do not know the number of iterations in advance and the body of the loop must be performed at least once.

Collatz: 3n+1 problem

The Collatz conjecture is an unsolved conjecture in mathematics named after Lothar Collatz, who first proposed it in 1937 at Syracuse University. The conjecture is also known as the $3n + 1$ problem, or the Syracuse problem.

Collatz conjecture

The Collatz conjecture says that the following algorithm will eventually terminate for any non-negative integer *a*.

```
1. Print $a$
2. If $a = 1$ goto step 4
3. If $a$ is even then
      divide $a$ by two,
   otherwise
      set $a \leftarrow 3a + 1$
   Go to Step 1.
4. Print "Done"
```

Collatz conjecture

```
1. Print $a$
2. If $a = 1$ goto step 4
3. If $a$ is even then
      divide $a$ by two,
   otherwise
      set $a \leftarrow 3a + 1$
   Print $a$
     Go to Step 1.
4. Print “Done”
```

```
int main(int argc, char *argv[]) {
   int a;
  scanf("%d", &a);
  printf("%d", a);
  while (a != 1) {
    a = (a%2 ? 3*a + 1 : a/2);
    printf(", %d", a);
  }
  printf("\nDone\n");
  return 0;
}
```

Collatz conjecture

Mathematicians aren't so interested in the actual sequences, but rather the number of terms in the sequence until you get to 1

For example,

```
27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484,
242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466,
233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445,
1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425,
1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858,
2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077,
9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976,
488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40,
20, 10, 5, 16, 8, 4, 2, 1    [112 iterations]

171, 514, 257, 772, 386, 193, 580, 290, 145, 436, 218, 109, 328,
164, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484,
242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466,
233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445,
1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425,
1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858,
2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077,
9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976,
488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40,
20, 10, 5, 16, 8, 4, 2, 1   [125 iterations]
```

Collatz conjecture: number of steps

```
int main(int argc, char *argv[]) {
  int a;
  scanf("%d", &a);
  int steps = 0;
  while (a != 1) {
    a = (a%2 ? 3*a + 1 : a/2);
    steps++;
  }
  printf("%d steps\n", steps);
  return 0;
}
```

Prime number?

Write a program fragment that reads a non-negative integer n from the keyboard and outputs YES if n is a prime number, and NO otherwise.

Prime number? A naive solution

```
int n;
scanf("%d", &n);
int a = 2;
while (a < n) {
   if (n%a == 0) {
      break; // n is not a prime
   }
   a++;
}
printf((n >= 2) && (a == n) ? "YES\n" : "NO\n");
```

The maximum 32 bit signed integer is $2^{31} - 1 = 2147483647$, and happens to be a prime. This means that for this number, the above program takes 2,147,483,647 iterations!

Prime number?

Problem analysis:

1. if there exists integers $a > 1$ and $b > 1$ such that $a \cdot b = n$ then n is not a prime number, otherwise it is.
2. We may assume that $a \leq b$, otherwise we swap the roles of a and b.
3. This means that $a^2 \leq a \cdot b$.
4. Hence, if we search for a suitable a using a while-loop, then we can stop searching once $\mathrm{a}^2 > n$.
5. The logical negation of $a^2 > n$ is $a^2 \leq n$, so that serves as the test of the loop.

Prime number? A better solution

```
int n;
scanf("%d", &n);
int a = 2;
while (a*a <= n) {
  if (n%a == 0) {
    break; // n is not a prime
  }
  a++;
}
printf((n >= 2) && (a*a > n) ? "YES\n" : "NO\n");
```

Since $\sqrt{2^{31}-1} \approx 46341$, it will now require 46341 steps to determine that $2^{31}-1 = 2147483647$ is prime. In other words, this program is expected to be more than 40000 times faster for this particular input!

Prime number? Minor improvements

Note that 2 is the only even prime. So, instead of `a++` we could start with `a=3` and use `a+=2`. This halves the number of iterations.

Moreover, in each iteration we compute `a*a` in the condition `a*a <= n`. On most CPUs multiplication takes more time (cpu clock times) than addition. Moreover, multiplication times 2^n is a very cheap operation (it only requires shifting the bit representation of the number by n positions to the left and extending it with zeros).

Now, realize that: $(a+2)^2 = a^2 + 4 \cdot a + 4$. So, if we maintain in an extra variable `sq` the value of `a*a`, then we can replace the condition by sq<=n, and update `sq` using `sq += 4*a + 4` followed by `a += 2;`

Prime number? Final version

```
int n;
scanf("%d", &n);
int a = 3, sq = 9;
while (sq <= n) {
  if (n%a == 0) {
    break; // n is not a prime
  }
  sq += 4*a + 4;
  a += 2;
}
printf((n==2) || ((n >= 2) && (n%2) && (sq > n))
       ? "YES\n" : "NO\n");
```

Exponentiation: c $= a^b$

```
/* compute a^b, call this value M */
/* a^b==M */
int c = 1;
/* c*a^b==M */
while (b != 0) {
  /* c*a^b==M, b>0 */
  /* c*a*a^(b-1)==M */
  c = c*a;
  /* c*a^(b-1)==M */
  b = b - 1;
  /* c*a^b==M */
}
/* b==0, c*a^b==M */
/* c==M */
```

Exponentiation: c $= a^b$

```
int c = 1;
while (b != 0) {
  c = c*a;
  b = b - 1;
}
```

Exponentiation: c = a^b (faster version)

```
int c = 1;
/* c*a^b==M */
while (b != 0) {
  /* c*a^b==M, b>0 */
  if (b%2 == 0) {
    /* c*a^b==M, b>0, b even */
    /* c*(a^2)^(b/2)==M */
    a = a*a;
    /* c*a^(b/2)==M */
    b = b/2;
    /* c*a^b==M */
  } else { // do what we did before
    /* c*a*a^(b-1)==M */
    c = c*a;
    b = b - 1;
    /* c*a^b==M */
  }
  /* c*a^b==M */
}
/* c==M */
```

Exponentiation: c $= a^b$ (faster version)

```
int c = 1;
/* c*a^b==M */
while (b != 0) {
  if (b%2 == 0) {
    a = a*a;
    b = b/2;
  } else {
    c = c*a;
    b = b - 1;
  }
}
/* c==M */
```