Imperative Programming

Week 2

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

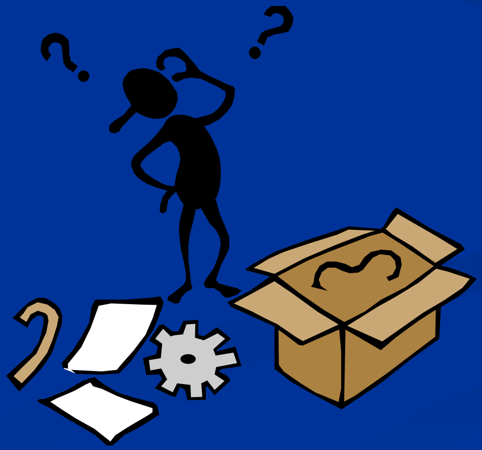
Example: sum of squares

```
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);
```

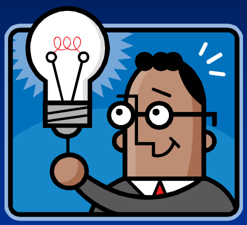
Example: sum of squares

Exercise: write a program fragment that sums the squares of all natural numbers less than 200.

0*0 + 1*1 + 2*2 + .. + 199*199



Example: sum of squares



It is much easier to use 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..200) and add `i*i` to `sumSquares`.

```
int i, sumSquares = 0;
for (i=0; i < 200; i++) {
  sumSquares += i*i;
}
printf("sumSquares = %d\n", sumSquares);
```

For-statement

The general syntax of the for-statement is:

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

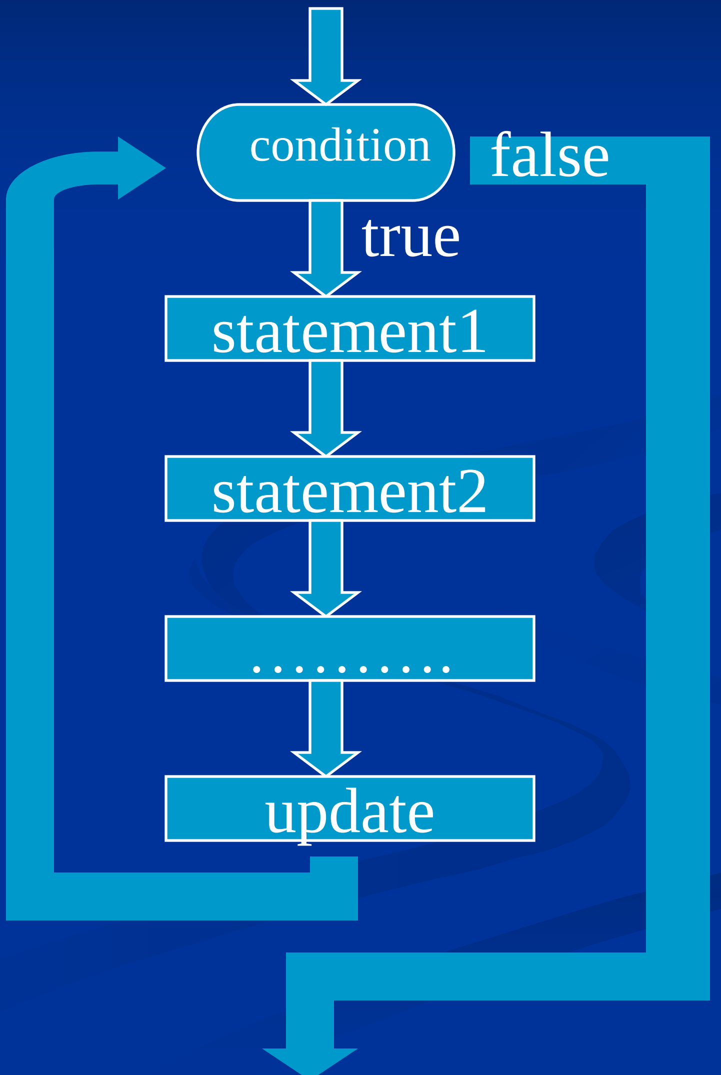
First, the statement **`initialisation`** is executed.

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('*');
}
putchar('\n');
```

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('*');
}
putchar('\n');
```

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('*');
}
putchar('\n');
```

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 **`limit`**.

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

Example: product of odd integers

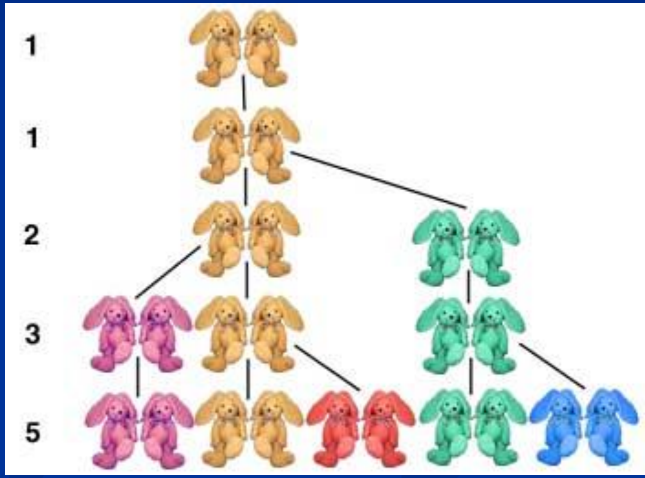
A nicer (and more efficient) solution is:

```
int oddProd = 1;
for (int i=3; i < limit; i+=2) {
  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

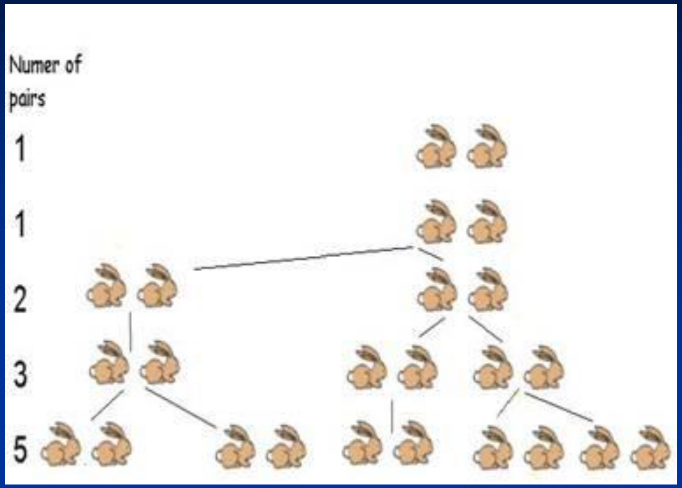


Exercise: write a program fragment that, given a non-negative integer `n`, prints the above table for the years `0` upto `n`:

Note that:

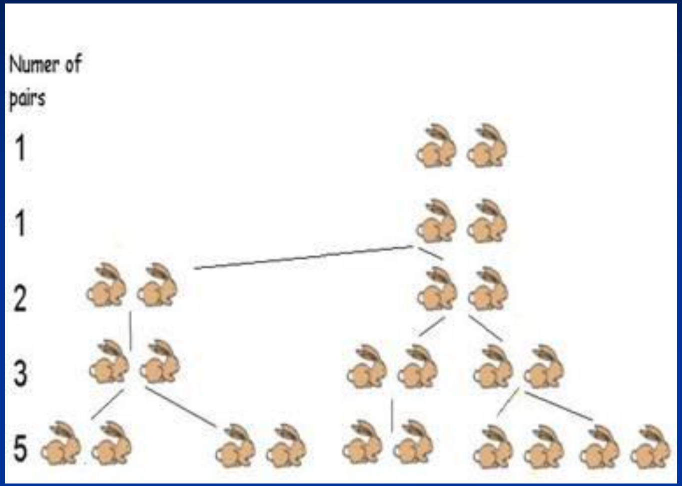
year	young	mature
y	a	b
y + 1	b	a + b

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 */
  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;

while (young + mature < n) {
  mature = mature + young;
  young = mature - young;
  year++;
}
/* here: young + mature >= n  (negation of the 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.

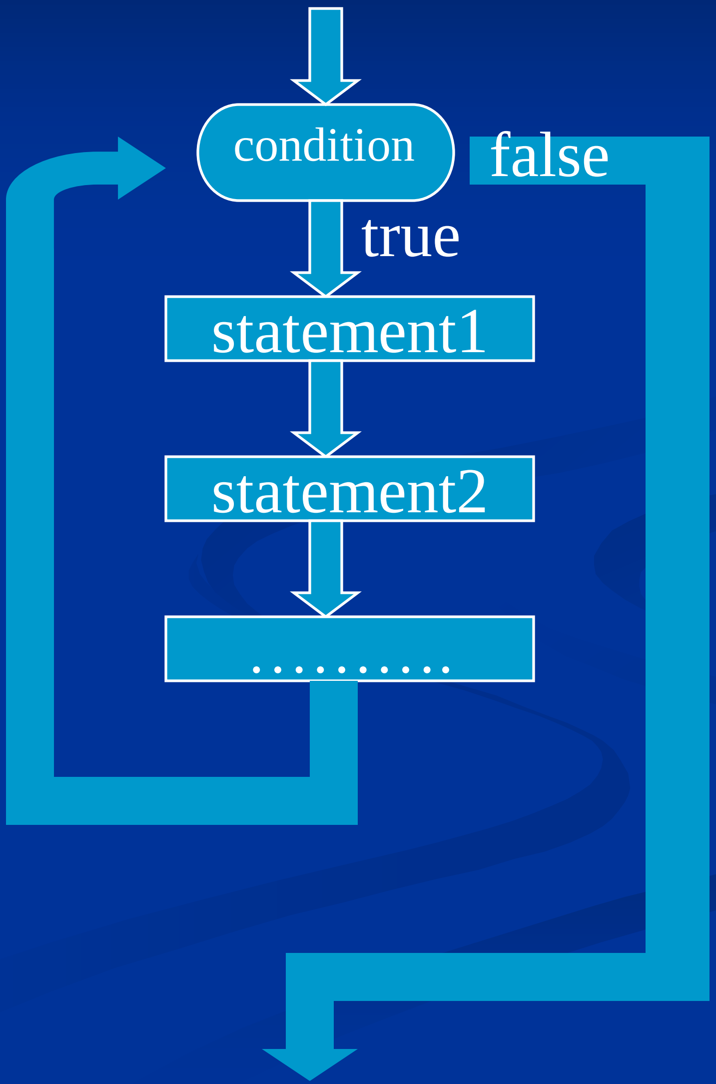
If `condition` evaluates to `0` (i.e. false), then the loop stops and execution of the program continues directly after the while-loop.

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.

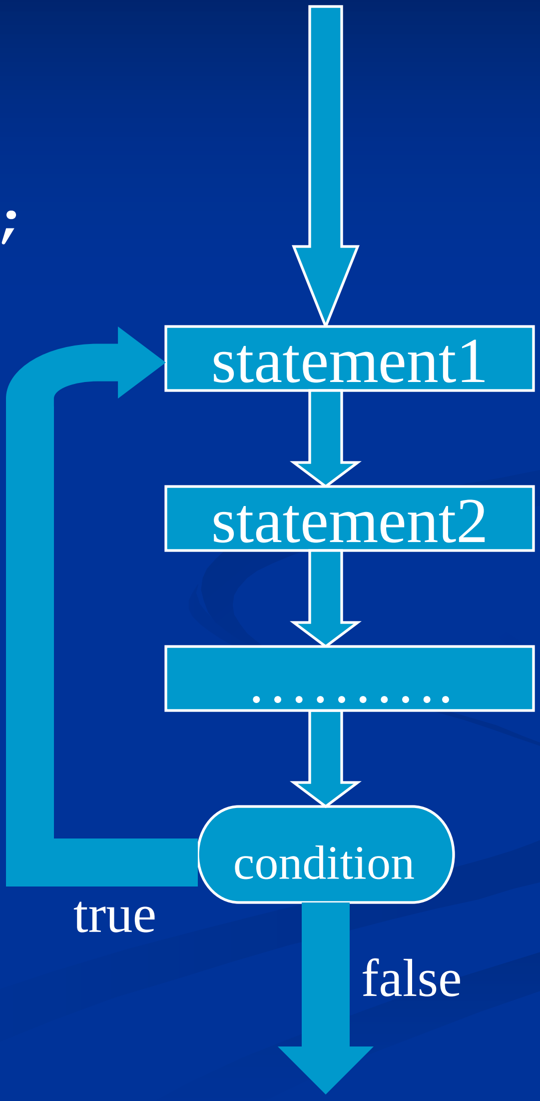
If `condition` evaluates to `0` (i.e. false), then the loop stops and execution of the program continues directly after the do-while-loop.

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 3 grades **`(x, y, z)`** from the keyboard 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 ("Grade: %4.1f\n", grade);
}
```

Which loop to use?

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

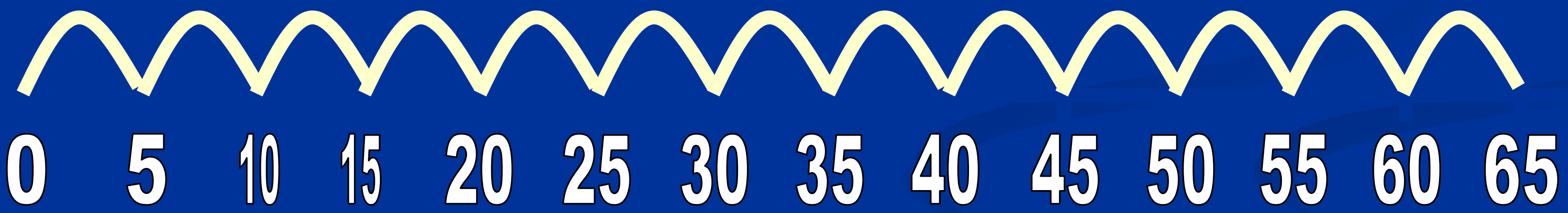
- Still, there is a preference based on style arguments:
  - **for-statement:** use it when we know the number of iterations in advance.

  - **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.

Integer division without division

We want to compute the integer division 65/5 without using the division operator (/).



So, 65/5 = 1 + (65-5)/5 = 2 + (60-5)/5 = … = 13 + 0/5 = 13

Integer division without division

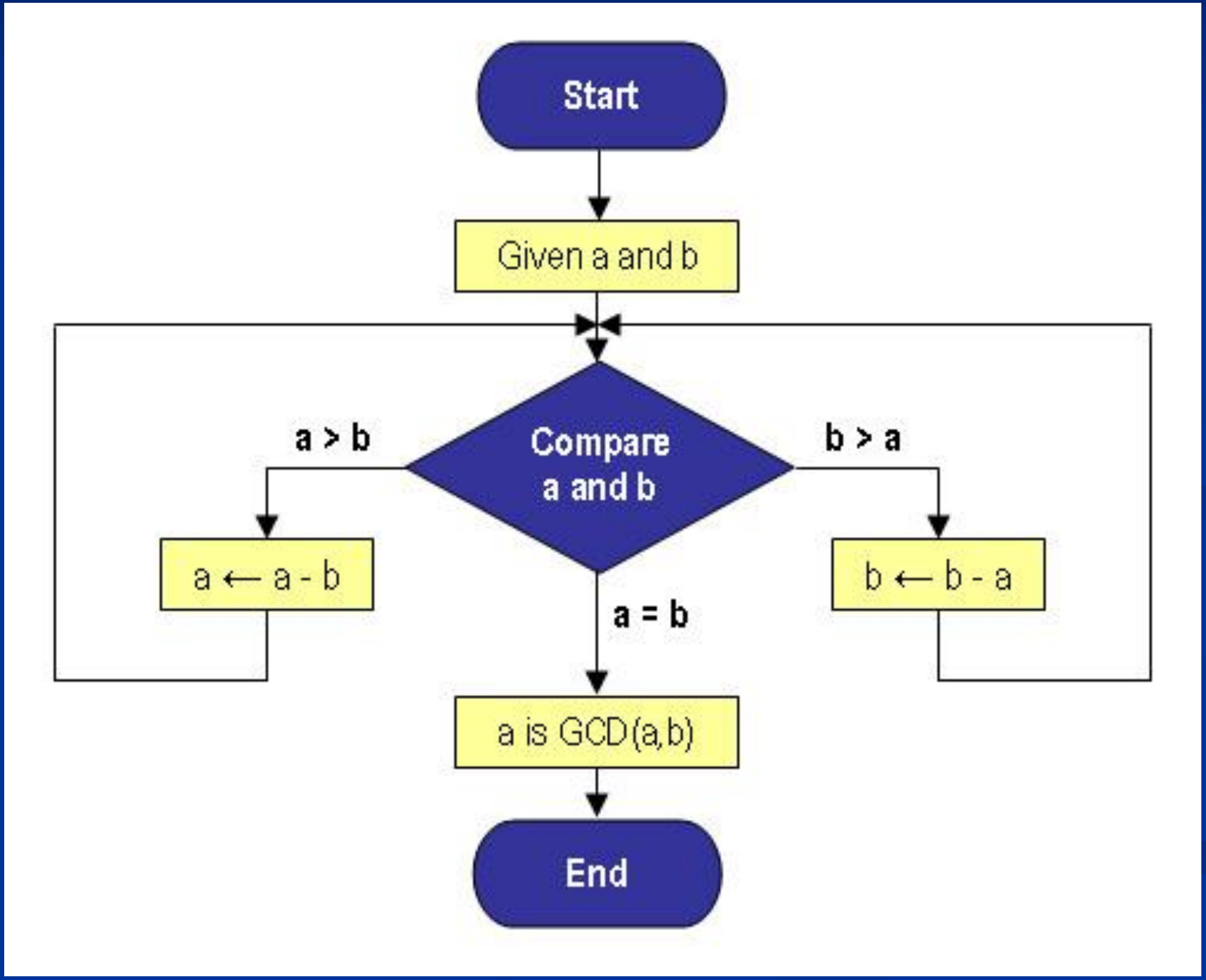
```
/* a==A, b==B */
n = 0;
while (a >= b) {
   /* a>=b && n + a/b == A/B */
   /* n + 1 + (a-b)/b == A/B */
   a = a - b;
   /* n + 1 + a/b == A/B */
   n++;
   /* n + a/b == A/B */
}
/* a < b and n + a/b == A/B */
/* n == A/B */
```

GCD: Greatest Common Divisor

- Let a and b be non-negative integers.
- GCD(a,b) is the greatest integer that divides a and b without a remainder.
  - In other words, it is the largest common factor of a and b.

Euclid of Alexandria was a Greek mathematician from the 3rd century before Christ. He is the inventor of the Euclidean method for finding the greatest common divisor of two non-negative integers.

GCD: Euclid’s algorithm



GCD: Euclid's algorithm

- For natural numbers a, b (where a > b) the following theorem holds:

  $$\gcd(a , b) = \gcd(a-b , b)$$

  Example: gcd(34 , 8) = gcd(26 , 8) = gcd(18 , 8) = gcd(10 , 8) = gcd(2 , 8)
  gcd(8,2) = gcd(6 , 2) = gcd(4 , 2) = gcd(2 , 2) = gcd(0 , 2)=2

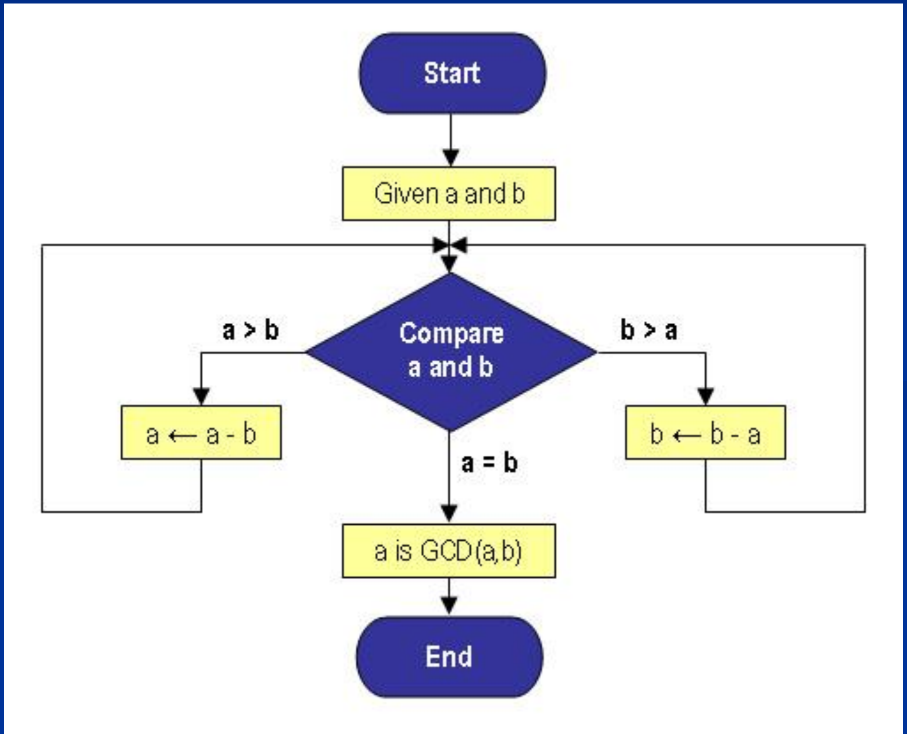
- Of course, an example is not a proof. Here is the proof:

- Let a>b and gcd(a , b) = c. First we show that c is a divider of a-b and b.
  - gcd(a , b) = c, so c is a divider of a and b.
  - Hence, there exist natural numbers m and n such that: a = m*c and b = n*c.
  - So, a - b = (m - n)*c, and c is a divider of a – b.

- Next, we need to prove that c is the greatest divider of b and a - b.
  - Let d>c be a divider of b and a - b.
  - Hence, there exist natural numbers p and q such that: a - b = p*d and b = q*d.
  - So, a = (p + q)*d.
  - Hence, d is a divider of a and b.
  - But, this contradicts that gcd(a , b) = c, since this would mean that d is greater than gcd(a,b).
  - So, we conclude that c is the greatest divider of b and a - b.

GCD: Euclid’s algorithm

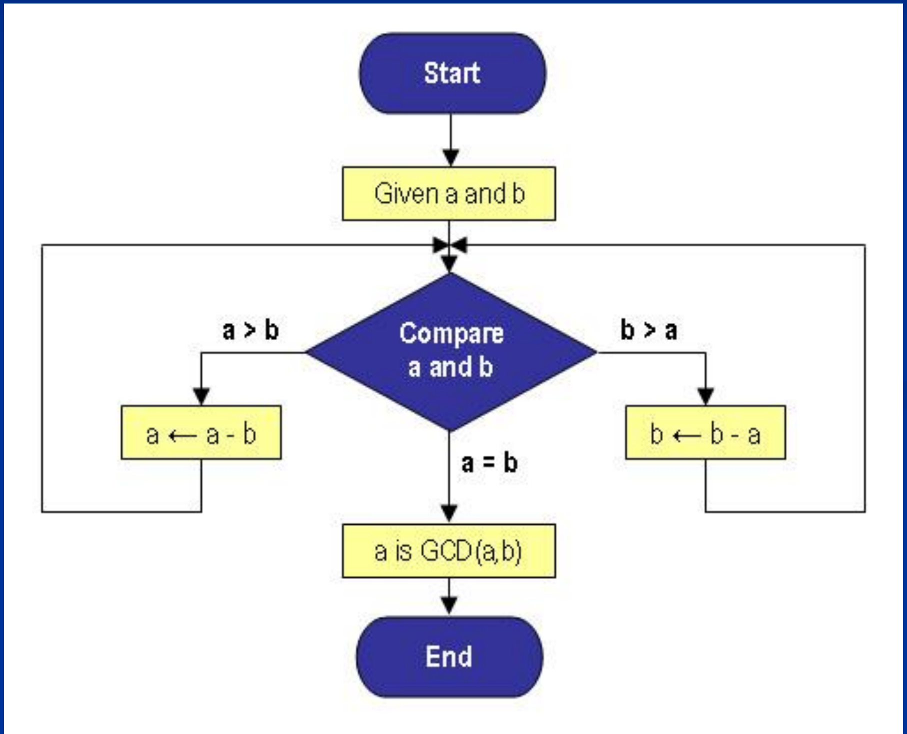
```
/* a==A, b==B */
if ((a == 0) || (b == 0)) {
  a = a + b;
} else {
  while (a != b) {
    if (a > b) {
      a = a- b;
    } else {
      b = b - a;
    }
  }
  /* a == b == gcd(A,B) */
}
/* a == gcd(A,B) */
```



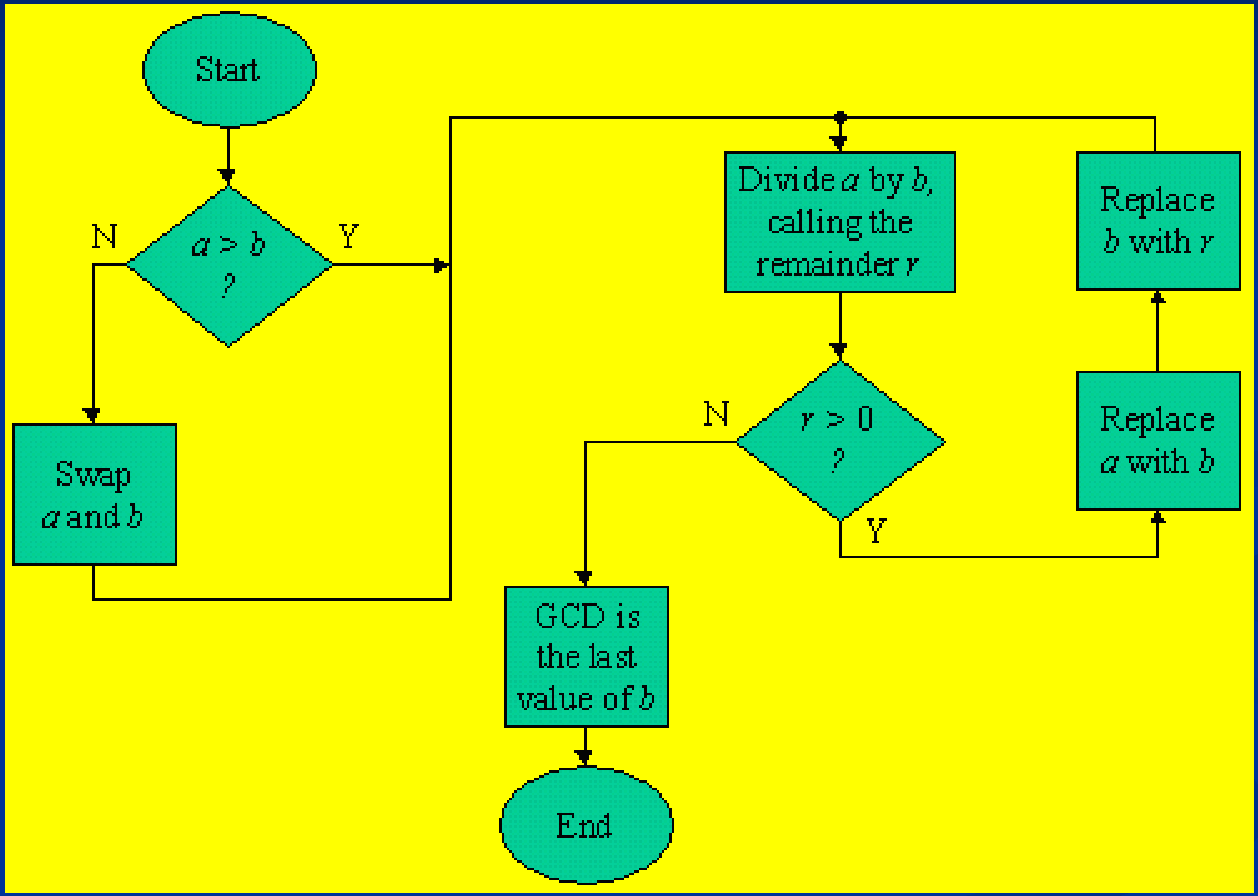
Note that this process does not terminate if initially one number is zero, while the other is non-zero.

GCD: Euclid’s algorithm

```
/* a==A, b==B */
if ((a == 0) || (b == 0)) {
  a = a + b;
} else {
  while (a != b) {
    while (a > b) {
      a = a- b;
    }
    while (b > a) {
      b = b - a;
    }
  }
  /* a == b == gcd(A,B) */
}
/* a == gcd(A,B) */
```

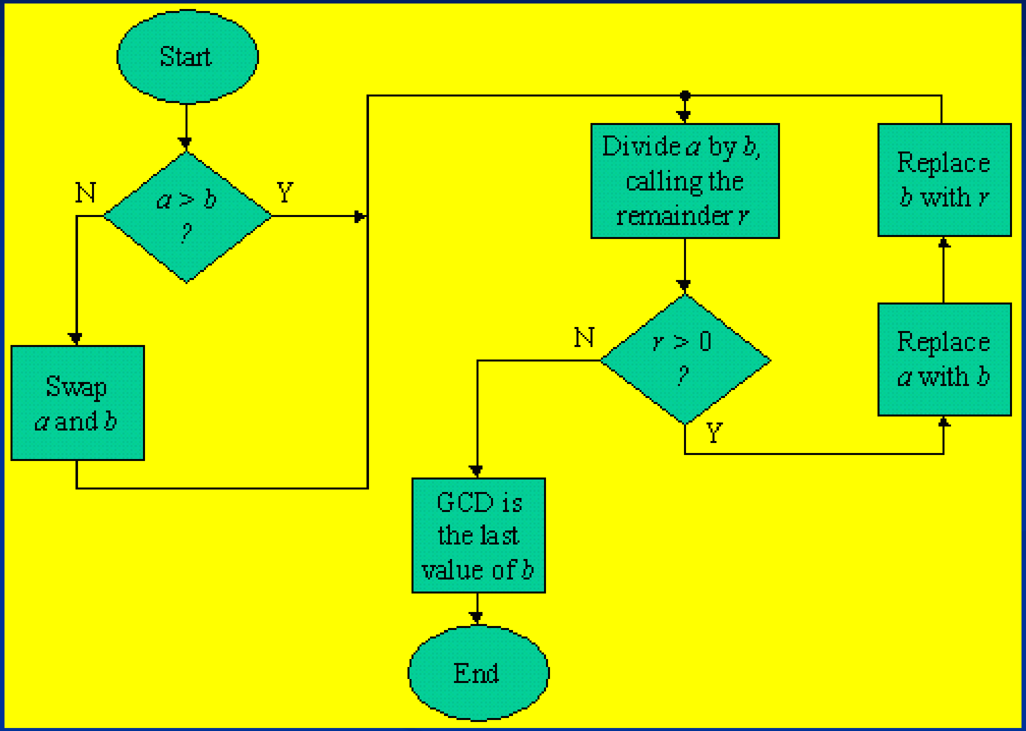


GCD: a bit smarter



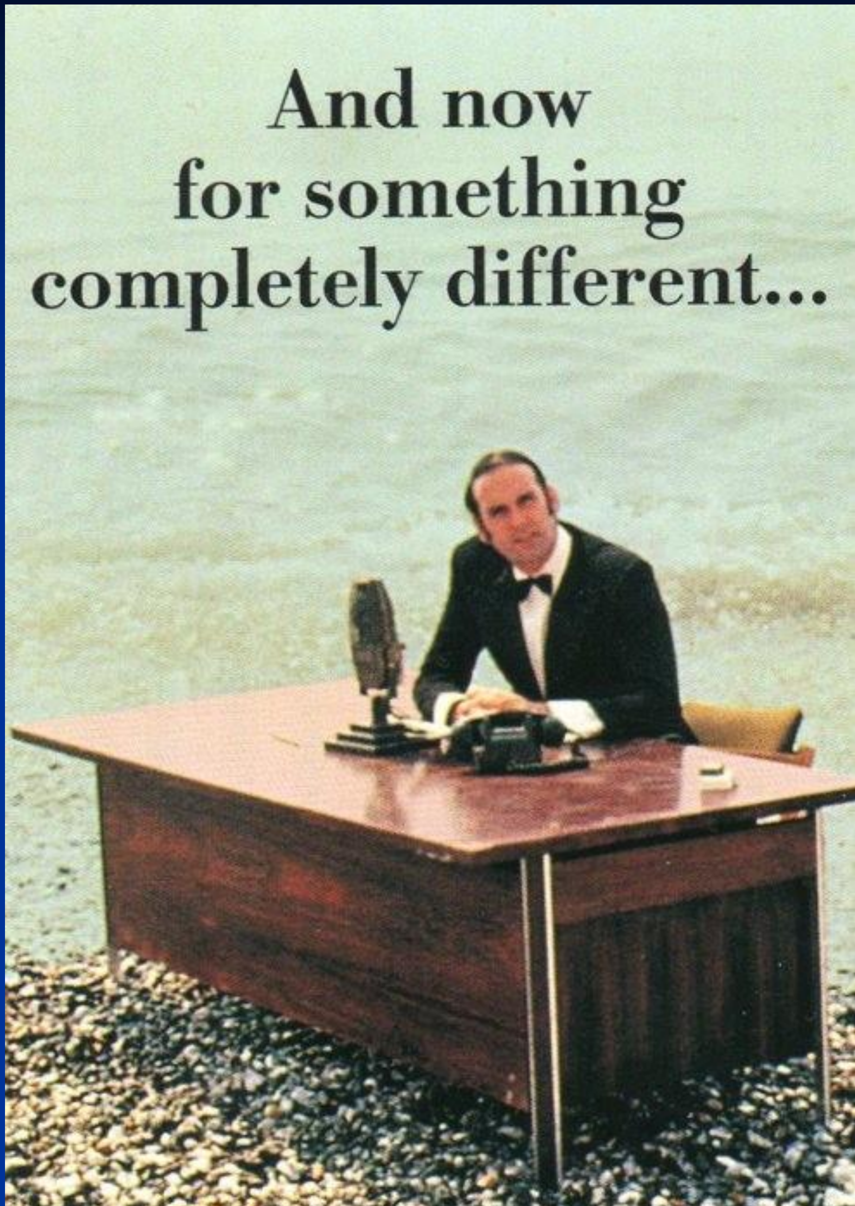
GCD: a bit smarter

```
/* a==A, b==B>0 */
do {
    r = a%b;
    if (r > 0) {
      a = b;
      b = r;
    }
} while (r != 0);
/* b==gcd(A,B) */
```



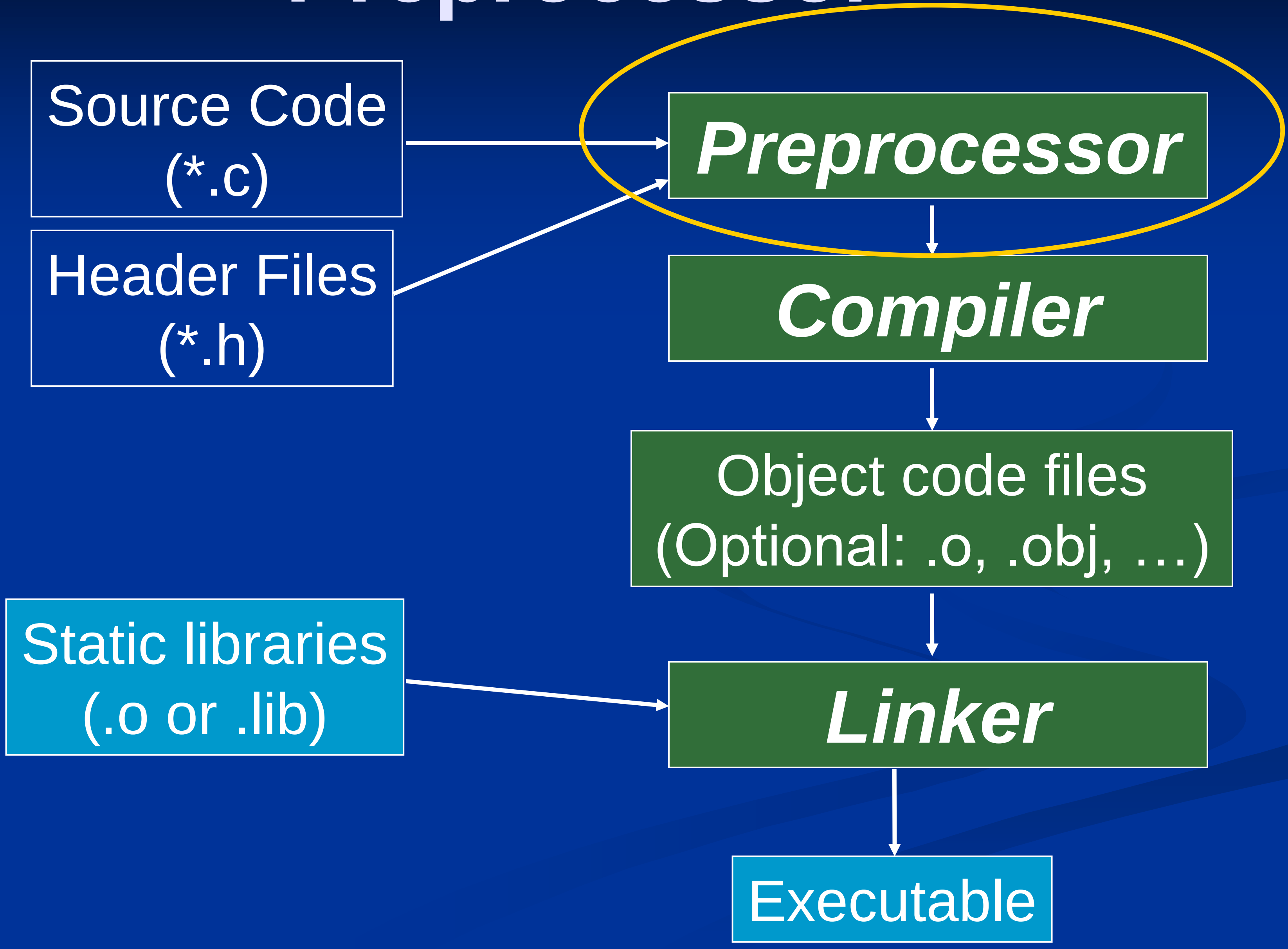
Even nicer is:

```
/* a==A, b==B */
while (a != 0) {
   r = b%a;
   b = a;
   a = r;
}
/* b==gcd(A,B) */
```



The C pre-processor

Preprocessor



Pre-processor

- Pre-processing
  - before compilation
  - in-place inclusion of (include-)files
  - in-place substitution of constants and macros
  - *Conditional compilation* of code

- A line with a pre-processor directive starts with the symbol #

`#include` directive

- Includes the contents of a file *in place*.

- **`#include <filename>`**
  - Searches file in *standard include path* (usually **`/usr/include`**).

`#include "filename"`

  - Searches first in current directory, followed by a search in the standard include path.
  - Is used for user-defined include files.

`#define` directive: symbolic constants

- `#define`
  - Preprocessor directive used for defining *symbolic constants* and *macros*.
  - During compilation, the compiler substitutes all occurrences of the symbolic constants and macros in place (symbolically!).
  - Convention: UPPERCASE LETTERS

- Examples:
  - `#define PI 3.14159`
    - **Beware. This is wrong:** `#define PI = 3.14159`
  - `#define MAX(a,b) ((a) > (b) ? (a) : (b))`

Macros

- Use parentheses (abundantly)!!! If you omit parentheses (inspired by normal operator precedence rules), many macros will go wrong!

- Example: area of a circle with radius r is pi*r*r

```
#define PI 3.14159265
#define CIRC(r)  PI*r*r

float a = CIRC(5);
float b = CIRC(3+2);
```

After pre-processing:

```
float a = 3.14159265*5*5;
float b = 3.14159265*3+2*3+2;
```

- A safe (and therefore better) macro is:

```
#define CIRC(r)  ((PI)*(r)*(r))
```

After pre-processing:

```
float a = ((3.14159265)*(5)*(5));
float b = ((3.14159265)*(3+2)*(3+2));
```

Macros

- Be very careful if you use macros in combination with `++` or `--`

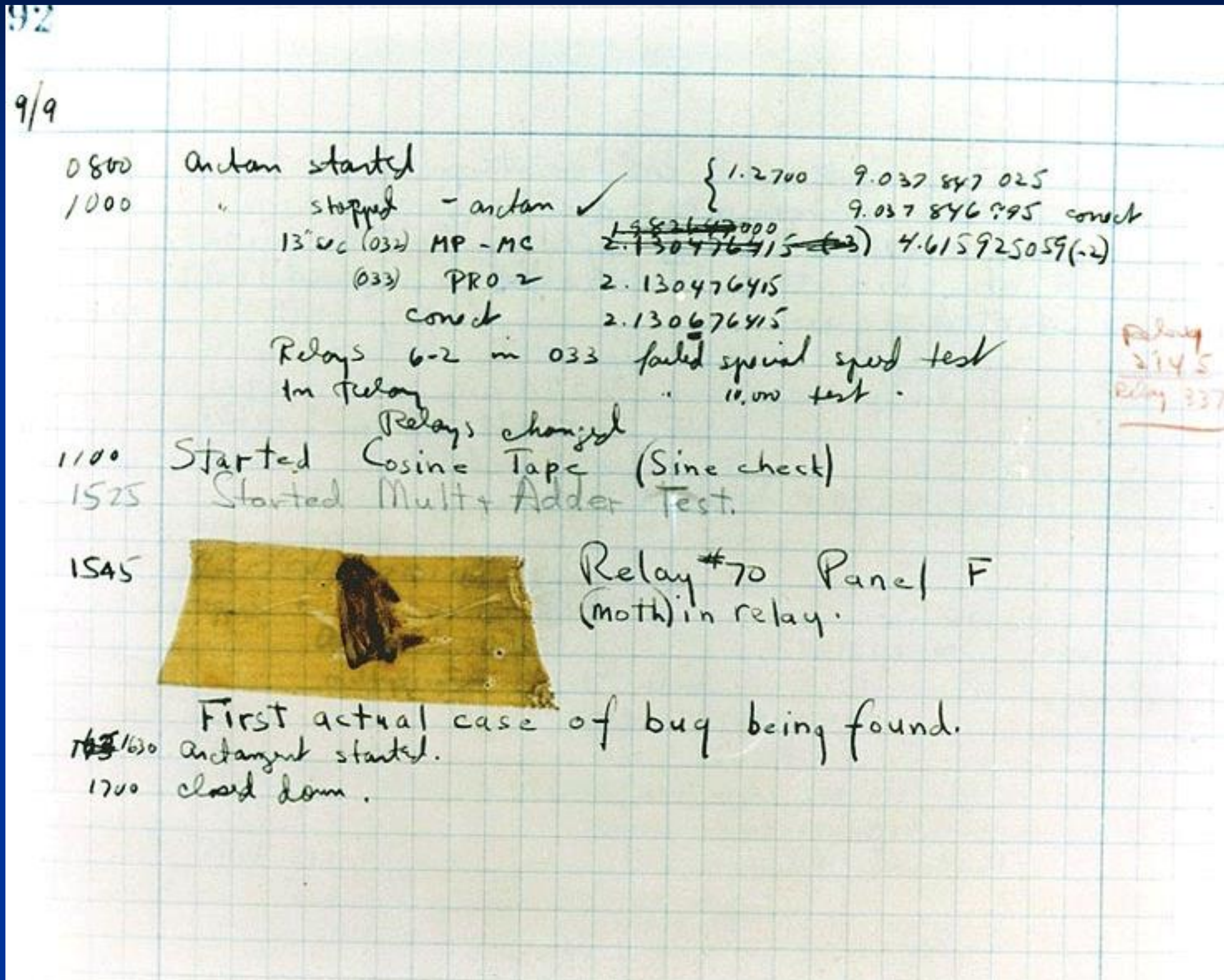
```
#define square(a) ((a)*(a))

int a = 5;
int b = square(a++);

printf("a=%d, b=%d\n", a, b);
```

The output of the program fragment: `a=7, b=30`

Result of pre-processing is: `int b = ((a++)*(a++));`

So, `a` is incremented twice!

Debugging



Page from the logbook of the Mark II -computer (Harvard university, 1947)

Debugging

You can use auxiliary printf-statements to track the values of variables (and hopefully find a bug):

```
float sum = 0;
int n;
for (n=100; n >= 0; n--) {
   printf("n = %d\n", n);
   sum += 1.0/n;
}
```

This program crashes after printing `n = 0`

`#if` : Conditional Compilation

- A kind of `if`-statement that is evaluated at compile-time!
  - Allows you to turn on/off code regions.

```
float sum = 0;
int n;
for (n=100; n >= 0; n--) {
  #if 1
    printf("n = %d\n", n);
  #endif
  sum += 1.0/n;
}
```

- Each `#if` must be matched with an `#endif`

Conditional Compilation

There is also an **`#else`**.

```
float sum = 0;
int n;
for (n=100; n >= 0; n--) {
  #if 1
   printf("n = %d\n", n);
   sum += 1.0/n;
  #else
   sum += 1.0/n;
  #endif
}
```

Conditional Compilation: debugging

```
#define DEBUG 1

 float sum = 0;
 int n;
 for (n=100; n >= 0; n--) {
   #if DEBUG
    printf("n = %d\n", n);
   #endif
   sum += 1.0/n;
 }
```

Assertions

- **`#include <assert.h>`**

- An assert can be used to test the validity of some boolean predicate.
  - If 0 (false), then an error message is printed and the program aborts.

```
#include <assert.h>

float sum = 0;
int n;
for (n=100; n >= 0; n--) {
  assert(n!=0);
  sum += 1.0/n;
}
```

Assertions

- If **`NDEBUG`** is defined, then all **`assert`** statements are ignored (at compile time).

```
#include <assert.h>

#define NDEBUG

float sum = 0;
int n;
for (n=100; n >= 0; n--) {
  assert(n!=0);
  sum += 1.0/n;
}
```

```
float sum = 0;
int n;
for (n=100; n >= 0; n--) {
  sum += 1.0/n;
}
```

End week 2