Intro - GCD

Week1

GCD

- An algorithm to find the greatest common divisor of two positive integers m and n, m ≥ n.

GCD

- An algorithm to find the greatest common divisor of two positive integers m and n, m ≥ n.
- A naïve solution – Described *informally* as follows.

GCD

- An algorithm to find the greatest common divisor of two positive integers m and n, m ≥ n.
- A naïve solution – Described *informally* as follows.
  1. Take the smaller number n.

GCD

- **An algorithm to find the greatest common divisor of two positive integers m and n, m ≥ n.**
- **A naïve solution – Described *informally* as follows.**
  1. **Take the smaller number n.**
  2. **For each number k, n ≥k≥1, in descending order, do the following.**
     1. If k divides m and n, then k is the gcd of m and n

GCD

- **An algorithm to find the greatest common divisor of two positive integers m and n, m ≥ n.**
- **A naïve solution – Described *informally* as follows.**
    1. **Take the smaller number n.**
    2. **For each number k, n ≥k≥1, in descending order, do the following.**
        1. If k divides m and n, then k is the gcd of m and n

- **This will compute gcd correctly, but is VERY slow (think about large numbers m and n).**

GCD

- An algorithm to find the greatest common divisor of two positive integers m and n, m ≥ n.
- A naïve solution – Described *informally* as follows.
  1. Take the smaller number n.
  2. For each number k, n ≥k≥1, in descending order, do the following.
     1. If k divides m and n, then k is the gcd of m and n

- This will compute gcd correctly, but is VERY slow (think about large numbers m and n).
- There is a faster way…

GCD Algorithm - Intuition

GCD Algorithm - Intuition

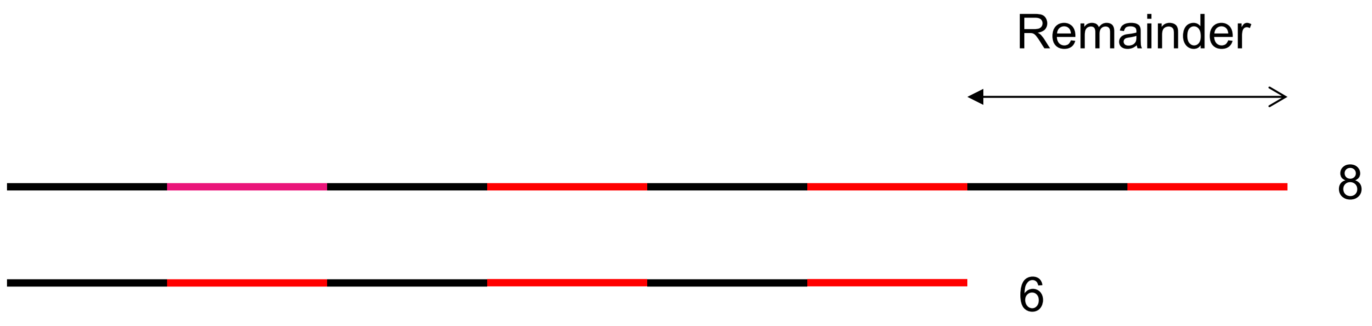
To find gcd of 8 and 6. Consider rods of length 8 and 6. Measure the longer with the shorter. Take the remainder if any. Repeat the process until the longer can be exactly measured as an integer multiple of the shorter.

GCD Algorithm - Intuition

To find gcd of 8 and 6. Consider rods of length 8 and 6. Measure the longer with the shorter. Take the remainder if any. Repeat the process until the longer can be exactly measured as an integer multiple of the shorter.

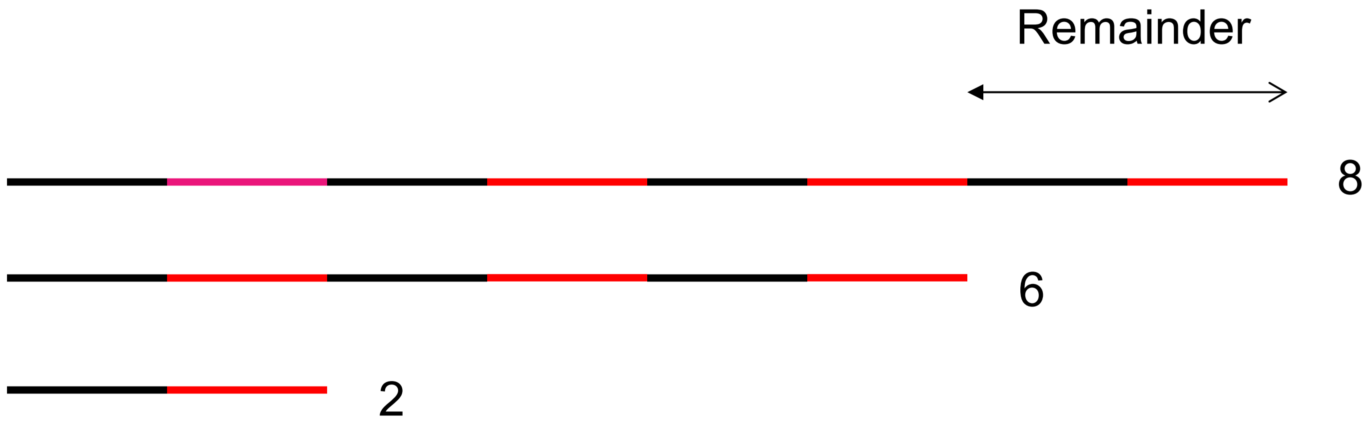
GCD Algorithm - Intuition

To find gcd of 8 and 6. Consider rods of length 8 and 6. Measure the longer with the shorter. Take the remainder if any. Repeat the process until the longer can be exactly measured as an integer multiple of the shorter.



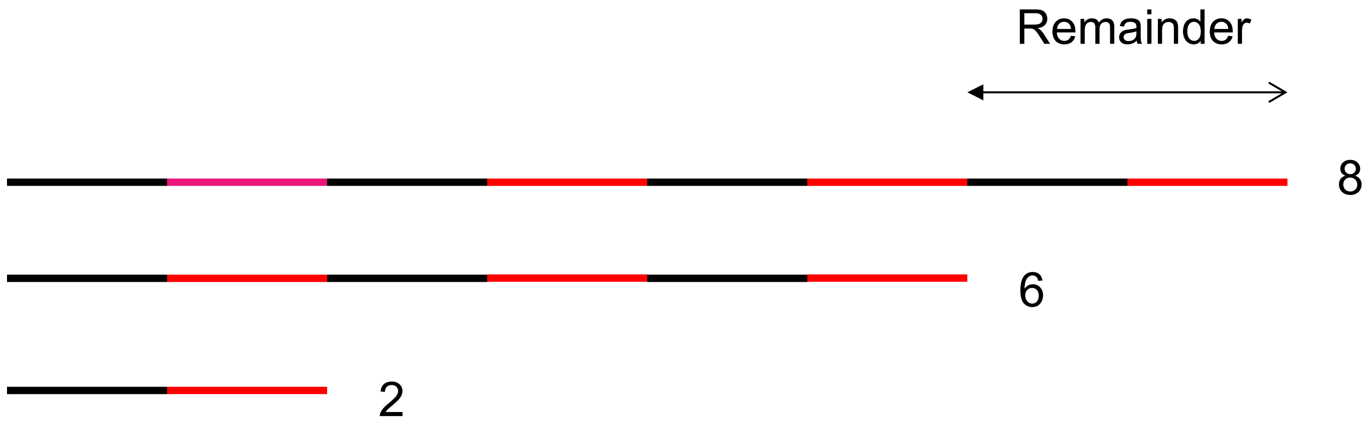
GCD Algorithm - Intuition

To find gcd of 8 and 6. Consider rods of length 8 and 6. Measure the longer with the shorter. Take the remainder if any. Repeat the process until the longer can be exactly measured as an integer multiple of the shorter.



GCD Algorithm - Intuition

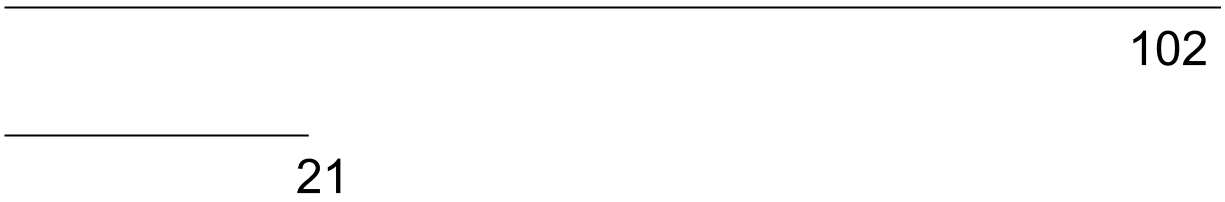
To find gcd of 8 and 6. Consider rods of length 8 and 6. Measure the longer with the shorter. Take the remainder if any. Repeat the process until the longer can be exactly measured as an integer multiple of the shorter.



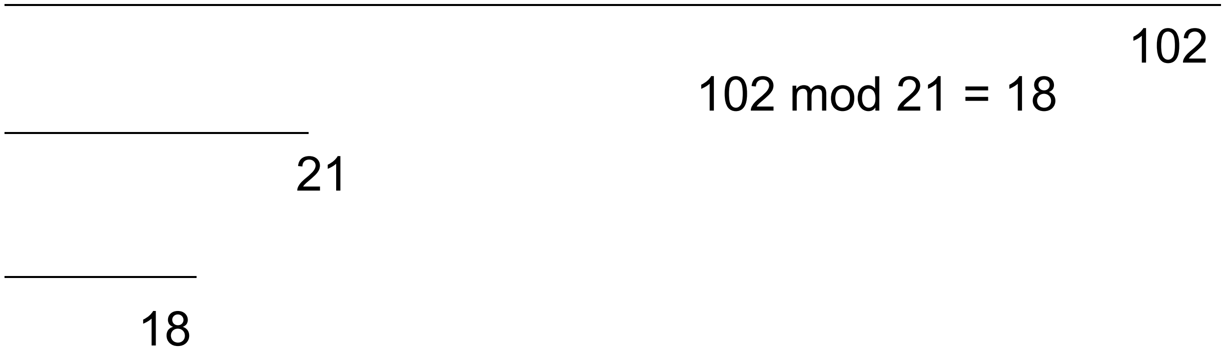
Gcd(8, 6) = 2.

GCD Algorithm - Intuition

GCD Algorithm - Intuition



GCD Algorithm - Intuition



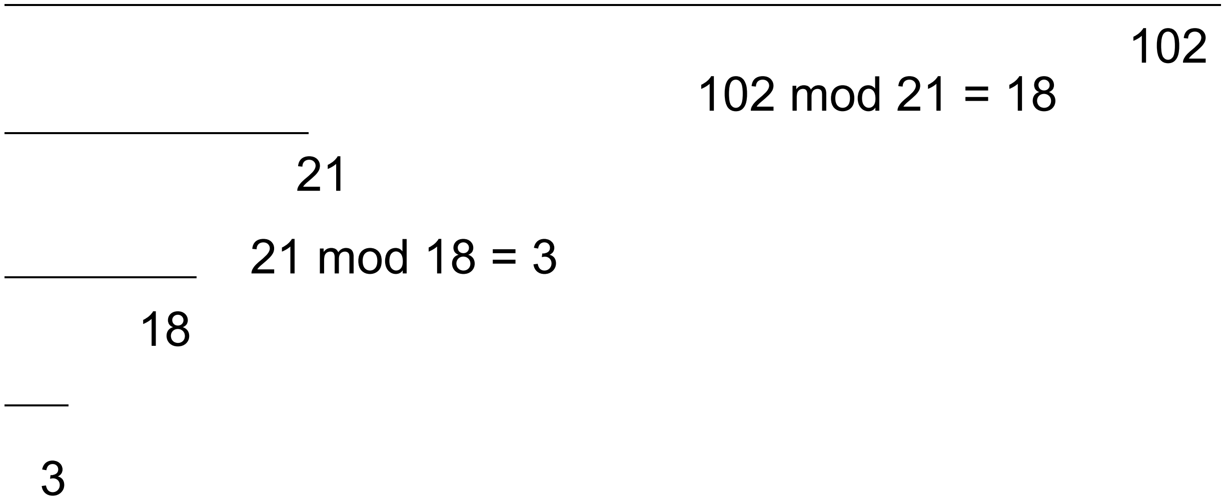
102

102 mod 21 = 18

21

18

GCD Algorithm - Intuition



102

102 mod 21 = 18

21

21 mod 18 = 3

18

3

Gcd (102, 21) = 3

Euclid’s method for gcd

Euclid’s algorithm (step-by-step method for calculating gcd) is based on the following simple fact.

Suppose a > b. Then the gcd of a and b is the same as the gcd of b and the remainder of a when divided by b.

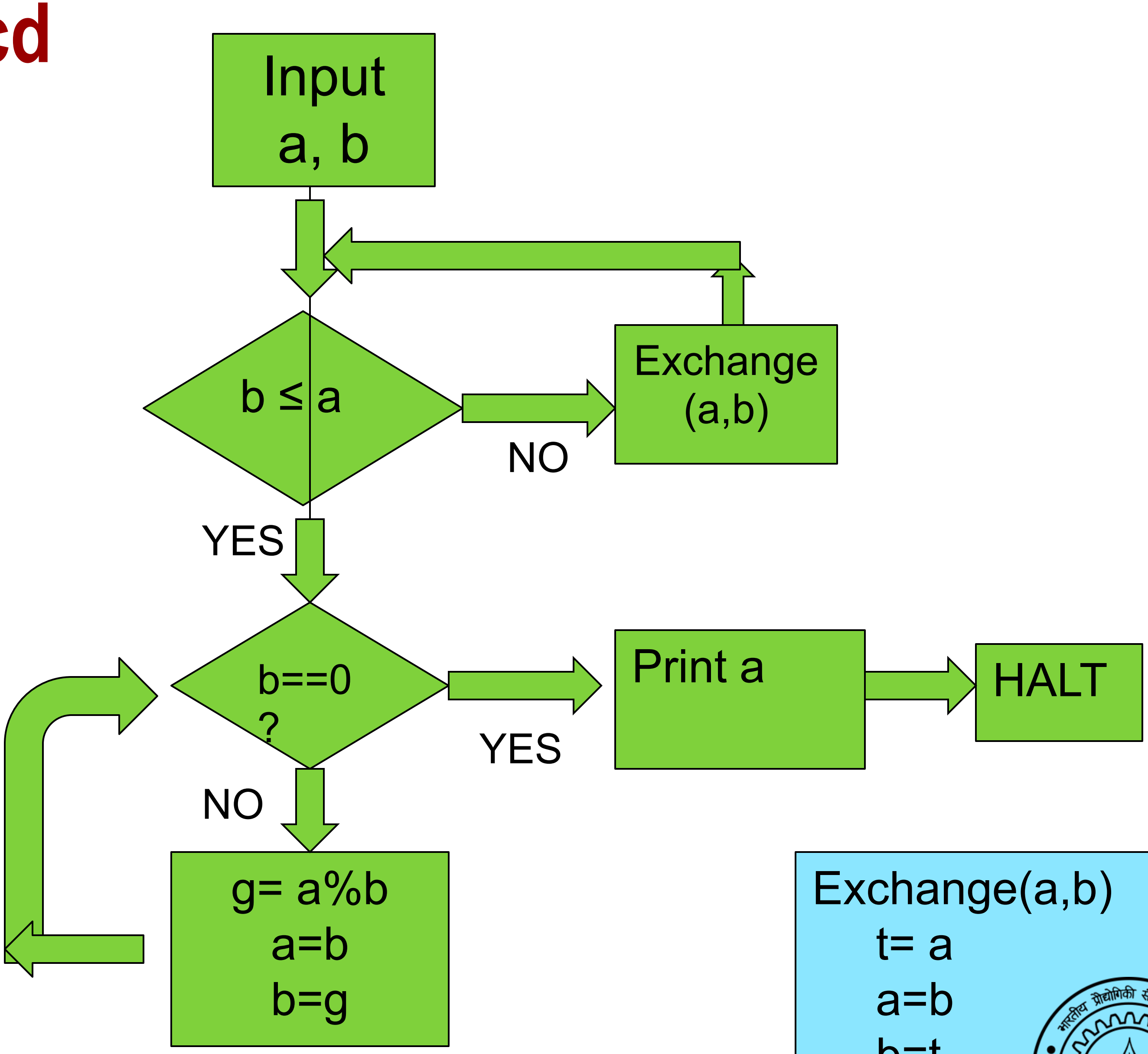
gcd(a,b) = gcd (b, a % b)

To see this consider division of a by b

$a = bq + r$

Euclid's gcd

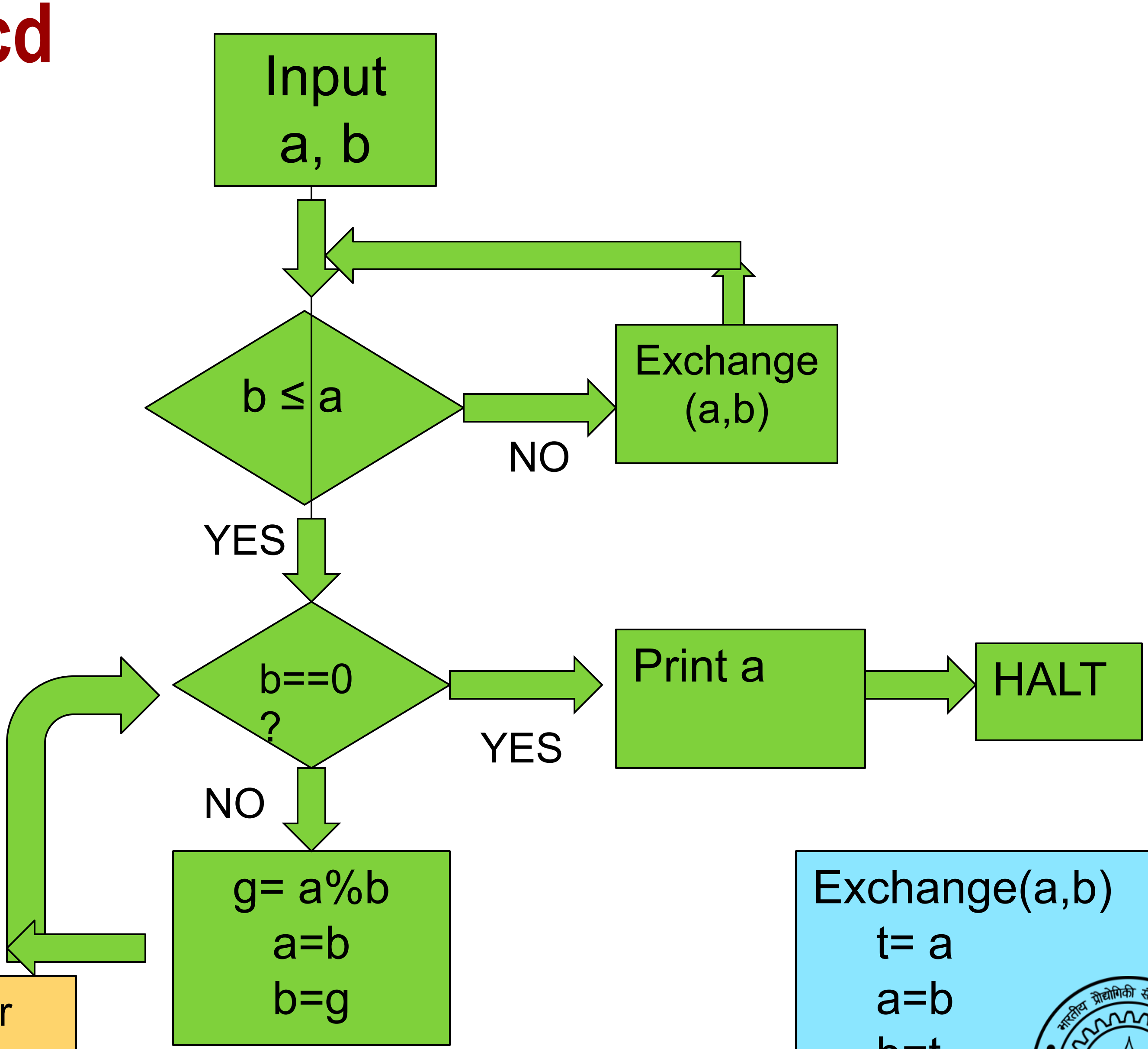
a,b,g are variables. Variables "store" exactly one value at a time.



```
Exchange(a,b)
    t= a
    a=b
    b=t
```

Euclid’s gcd

a,b,g are variables. Variables “store” exactly one value at a time.



Input a, b

b ≤ a

NO

Exchange (a,b)

YES

b==0 ?

YES

Print a

HALT

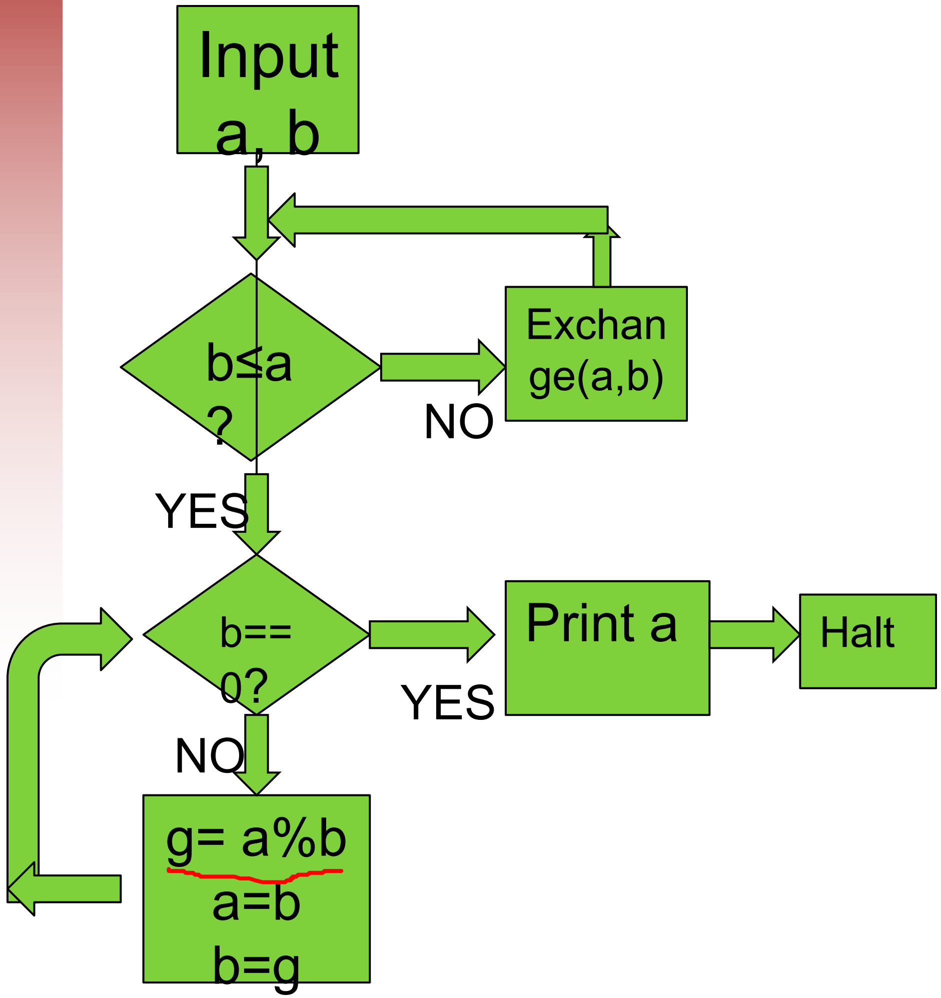
NO

g= a%b
a=b
b=g

a%b is the remainder when a is divided by b.
Eg. 8%3 is 2

```
Exchange(a,b)
    t= a
    a=b
    b=t
```

Variables and Assigning them



Input a, b

b≤a ?

NO

Exchange(a,b)

YES

b== 0?

YES

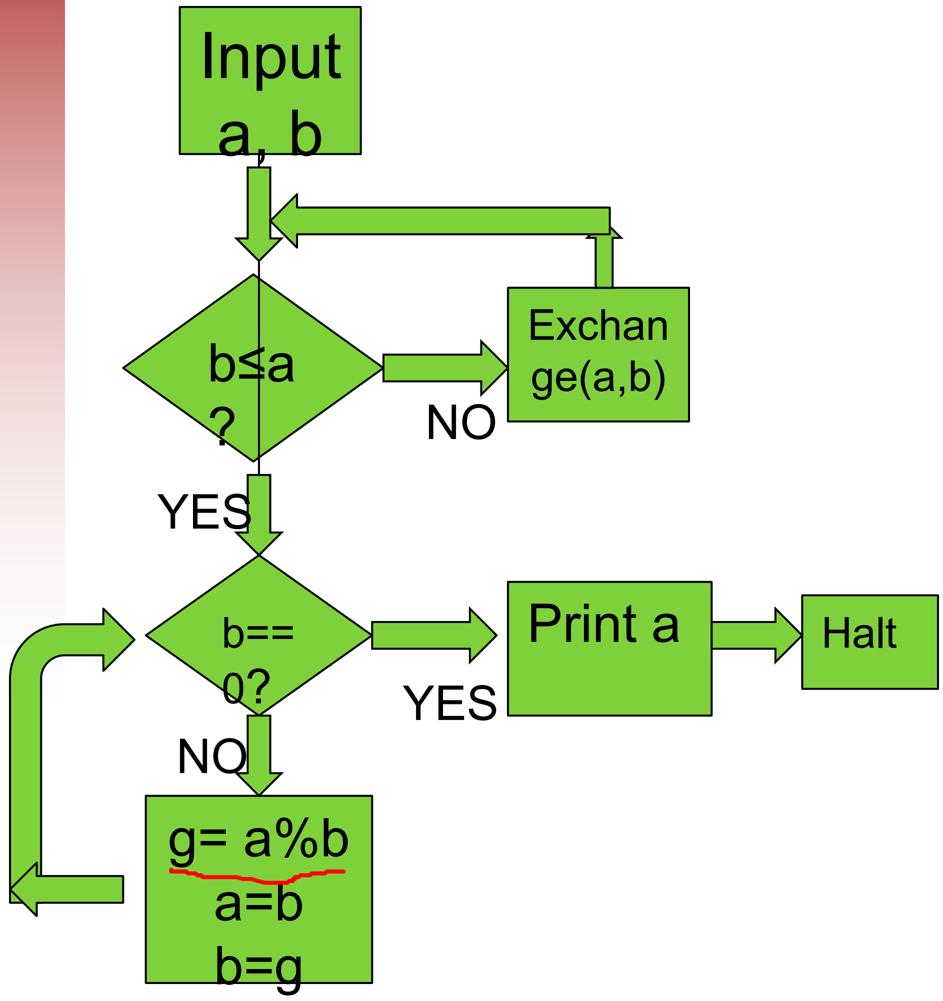
Print a

Halt

NO

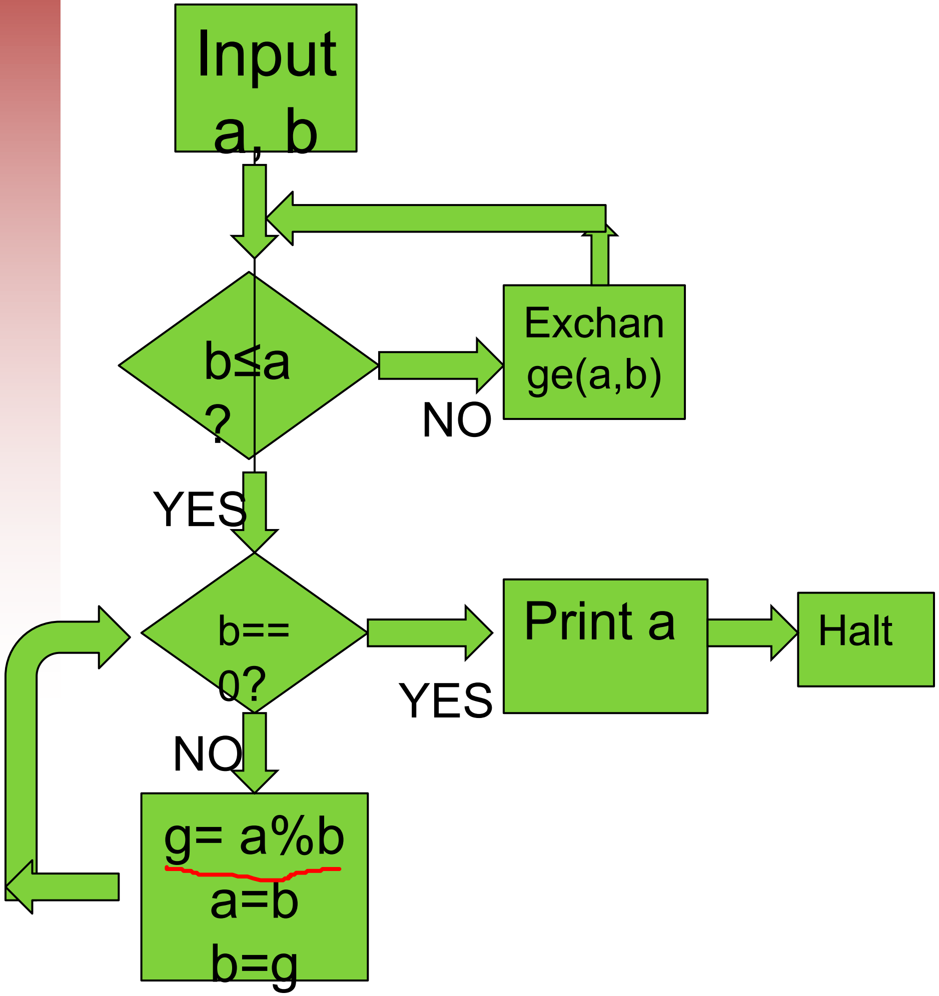
g= a%b
a=b
b=g

Variables and Assigning them

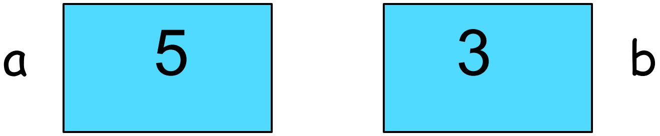


- Concept of variable: a name for a box.

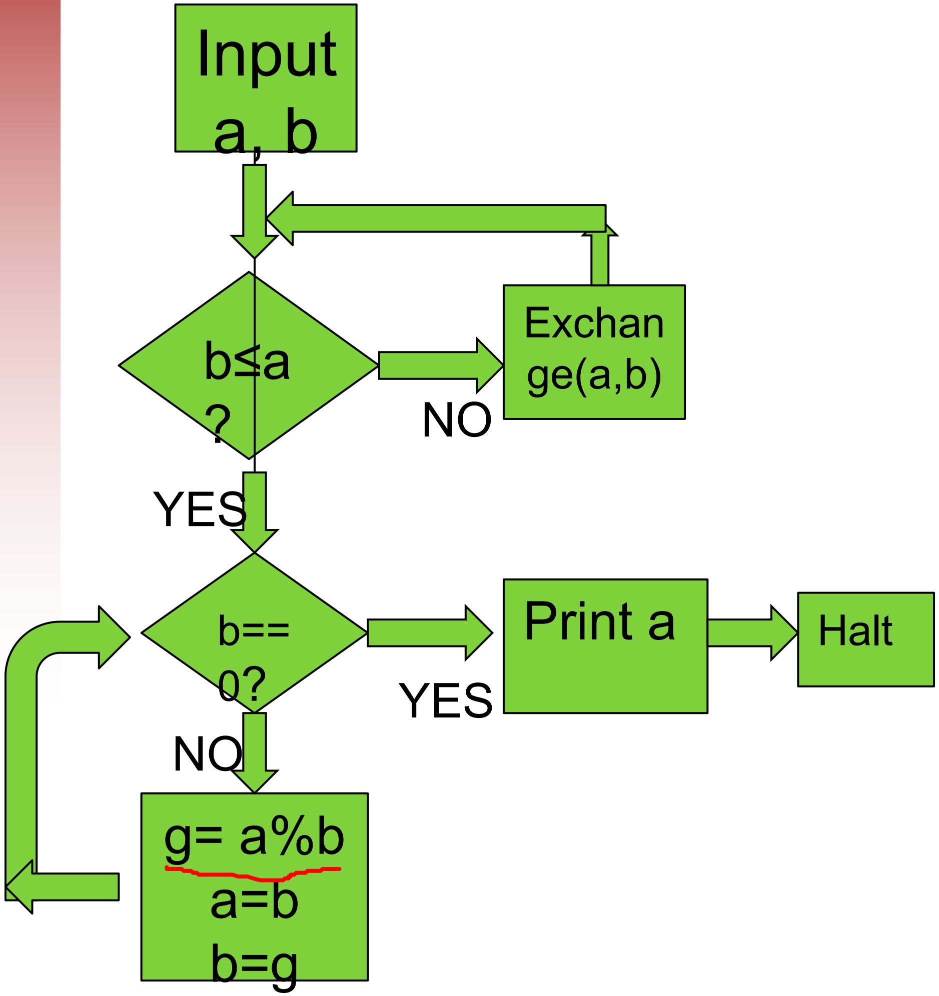
Variables and Assigning them



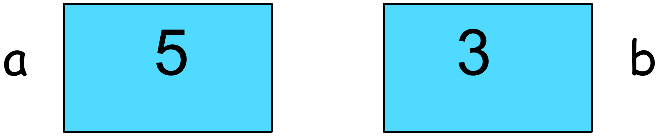
- Concept of variable: a name for a box.
- a,b,g are variables that are names for integer boxes.



Variables and Assigning them

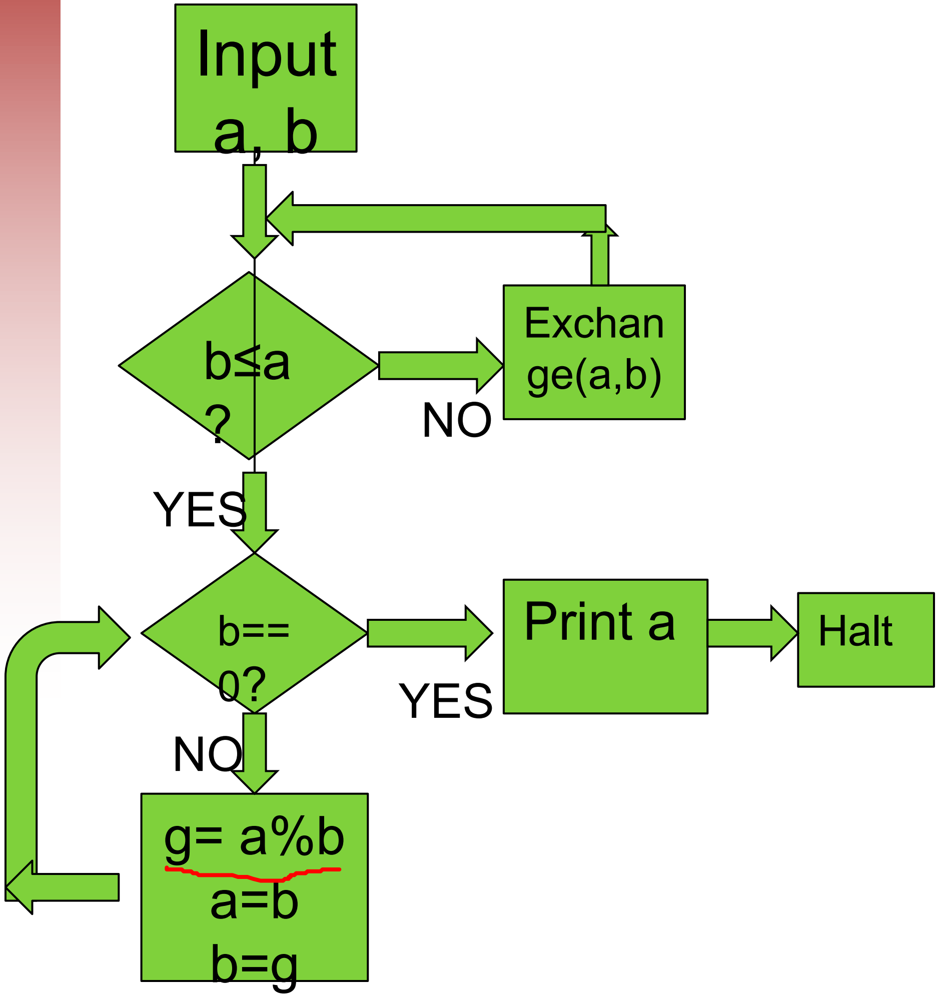


- Concept of variable: a name for a box.
- a,b,g are variables that are names for integer boxes.

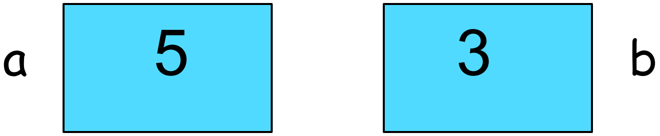


- Assignment a = b replaces whatever is stored in a by what is stored in b.

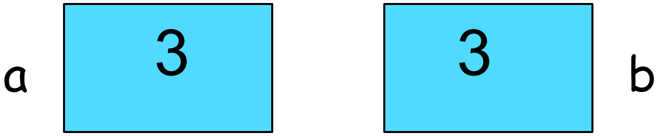
Variables and Assigning them



- Concept of variable: a name for a box.
- a,b,g are variables that are names for integer boxes.



- Assignment a = b replaces whatever is stored in a by what is stored in b.
- After a = b



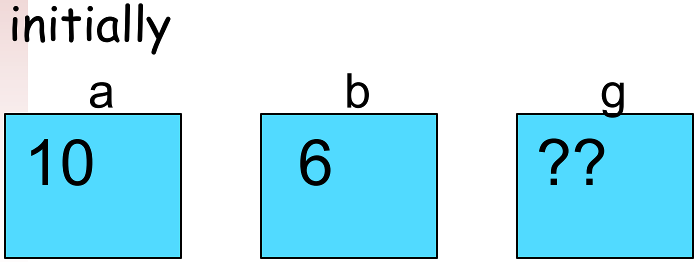
Sequential assignments

```
g = a%b;
a = b;
b = g;
```

- Semi-colons give a sequential order in which to apply the statements.

Sequential assignments

```
g = a%b;
 a = b;
 b = g;
```



initially

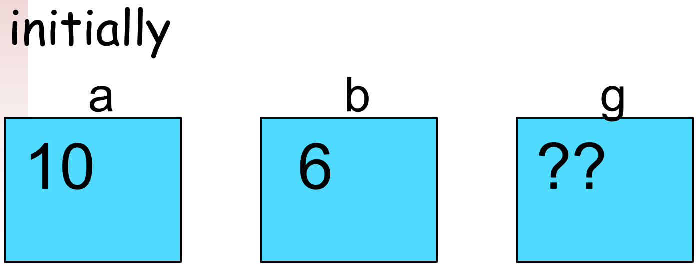
a b g

10 6 ??

- Semi-colons give a sequential order in which to apply the statements.
- Variables are boxes to which a name is given.
- We have 3 variables: a, b, g. This gives us three boxes. Initially, a is 10, b is 6 and g is undefined.

Sequential assignments

```
g = a%b;
a = b;
b = g;
```



initially

a

10

b

6

g

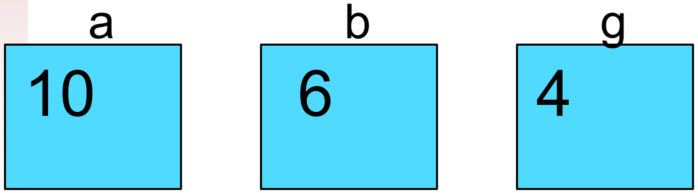
??

- Semi-colons give a sequential order in which to apply the statements.
- Variables are boxes to which a name is given.
- We have 3 variables: a, b, g. This gives us three boxes. Initially, a is 10, b is 6 and g is undefined.
- Run statements in sequence.
- Next statement to run

Sequential assignments

```
    g = a%b;
🏃     a = b;
       b = g;
```

After g = a %b



a	b	g
10	6	4

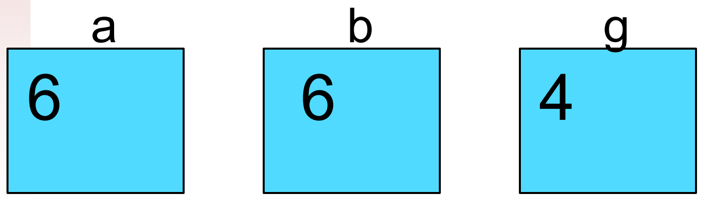
- Semi-colons give a sequential order in which to apply the statements.
- Variables are boxes to which a name is given.
- We have 3 variables: a, b, g. This gives us three boxes. Initially, a is 10, b is 6 and g is undefined.
- Run statements in sequence.
- Next statement to run

🏃

Sequential assignments

```
g = a%b;
a = b;
b = g;
```

After a = b

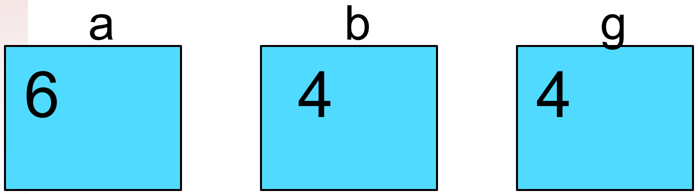


- Semi-colons give a sequential order in which to apply the statements.
- Variables are boxes to which a name is given.
- We have 3 variables: a, b, g. This gives us three boxes. Initially, a is 10, b is 6 and g is undefined.
- Run statements in sequence.
- Next statement to run

Sequential assignments

```
g = a%b;
a = b;
b = g;
```

After b = g



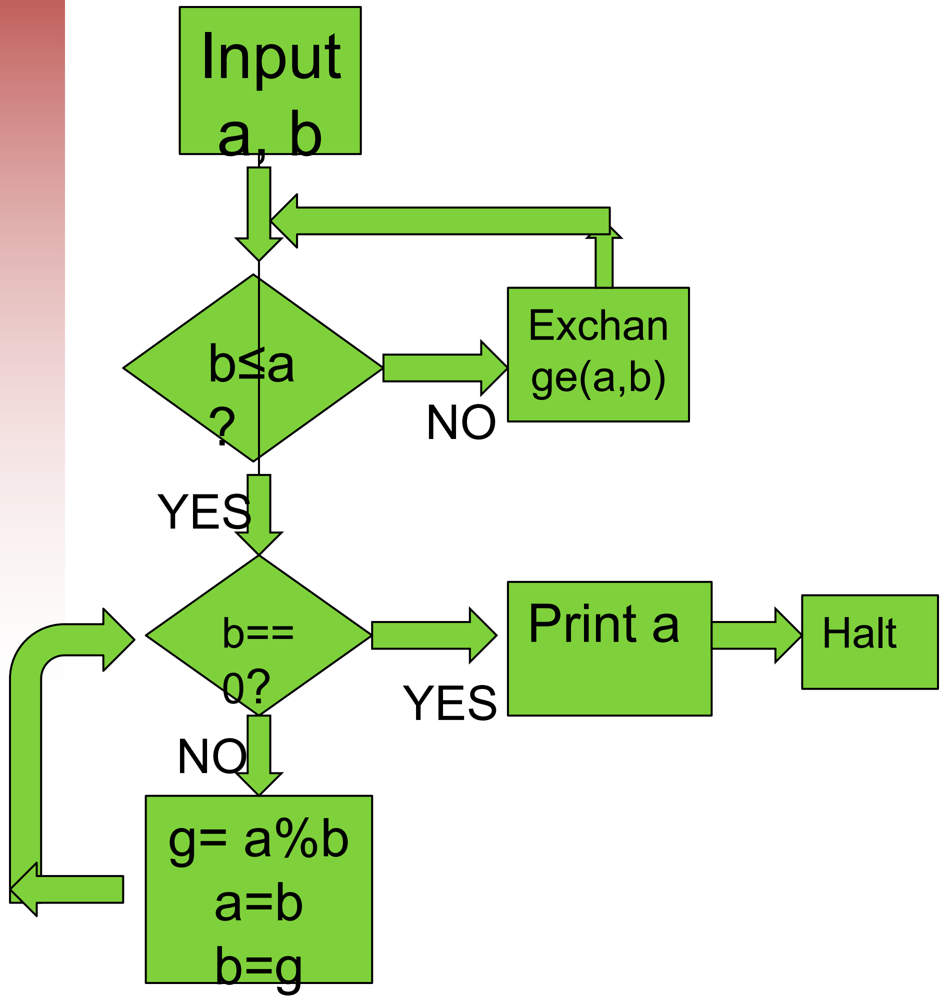
a
6

b
4

g
4

- Semi-colons give a sequential order in which to apply the statements.
- Variables are boxes to which a name is given.
- We have 3 variables: a, b, g. This gives us three boxes. Initially, a is 10, b is 6 and g is undefined.
- Run statements in sequence.
- Next statement to run

Running the program



Input a, b

b≤a ?

NO

Exchange(a,b)

YES

b== 0?

YES

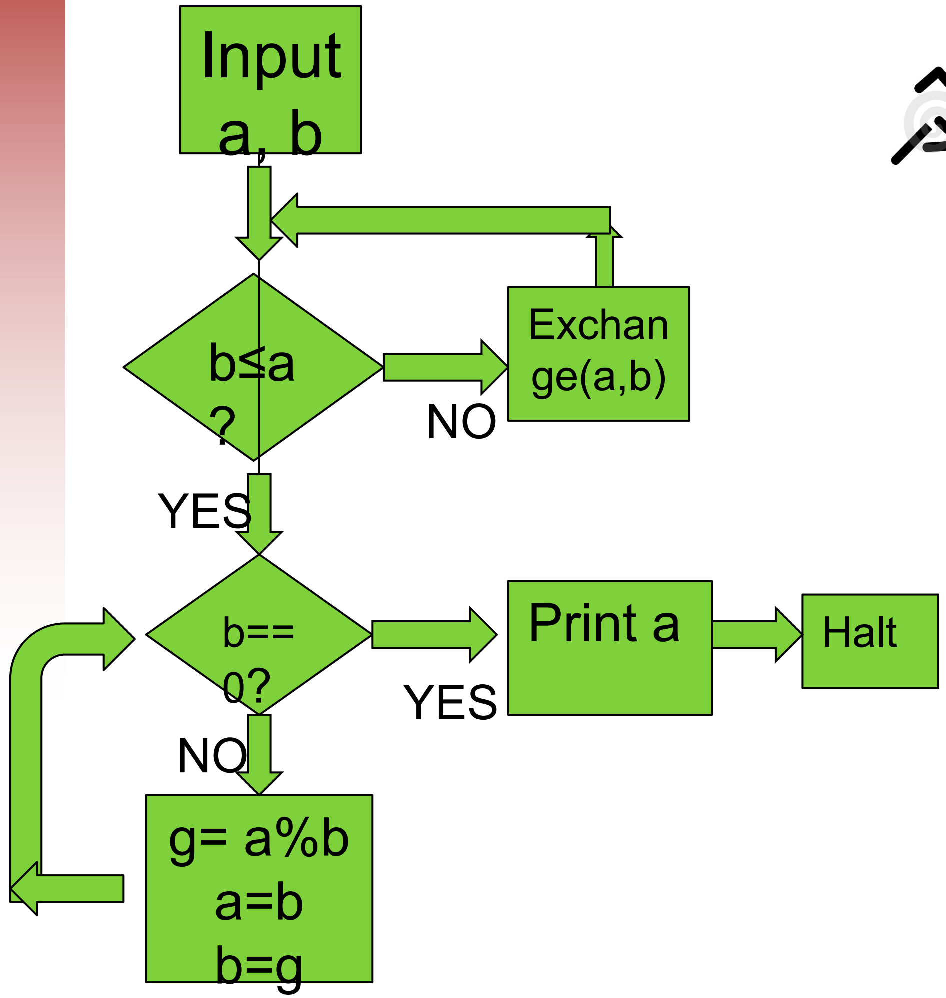
Print a

Halt

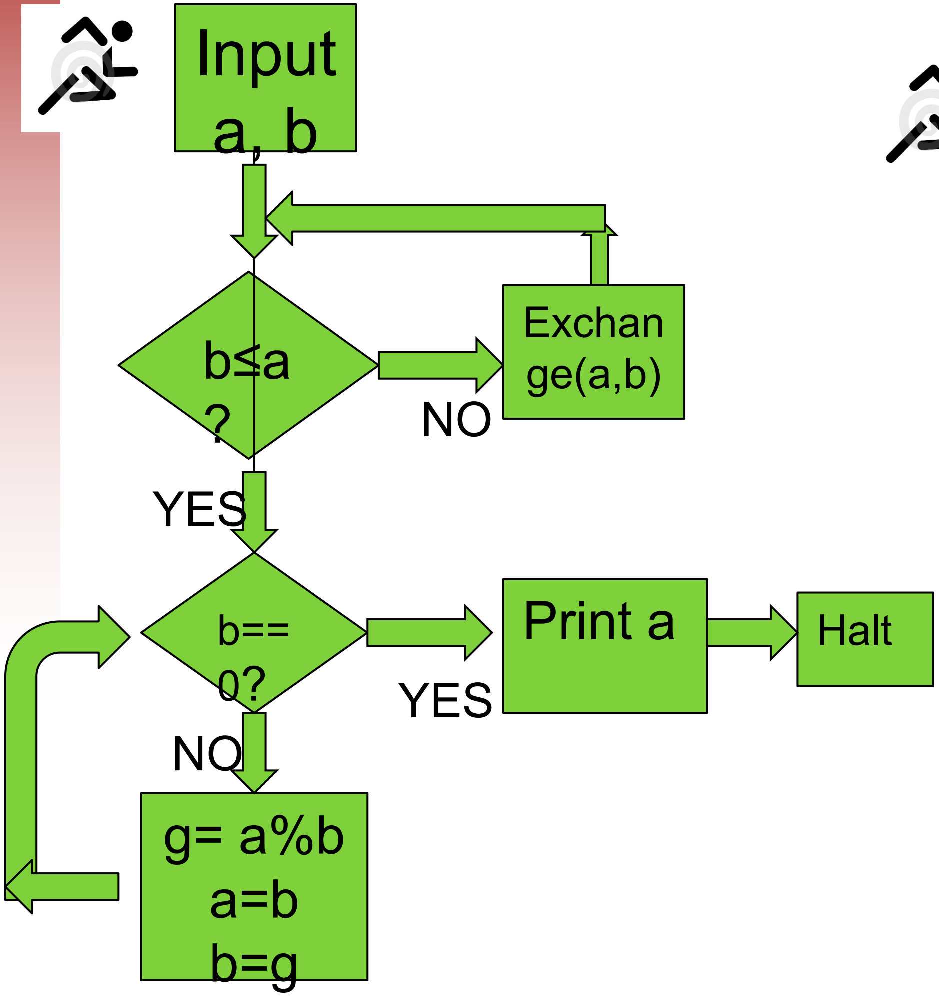
NO

g= a%b
a=b
b=g

Running the program



Running the program



Input a, b

b≤a ?

NO

Exchange(a,b)

YES

b==0?

YES

Print a

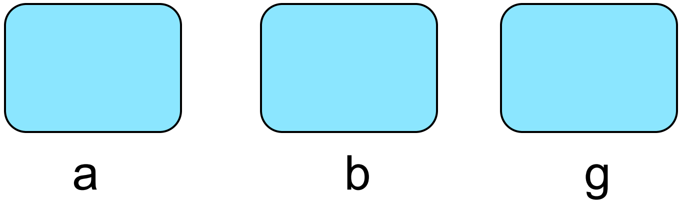
Halt

NO

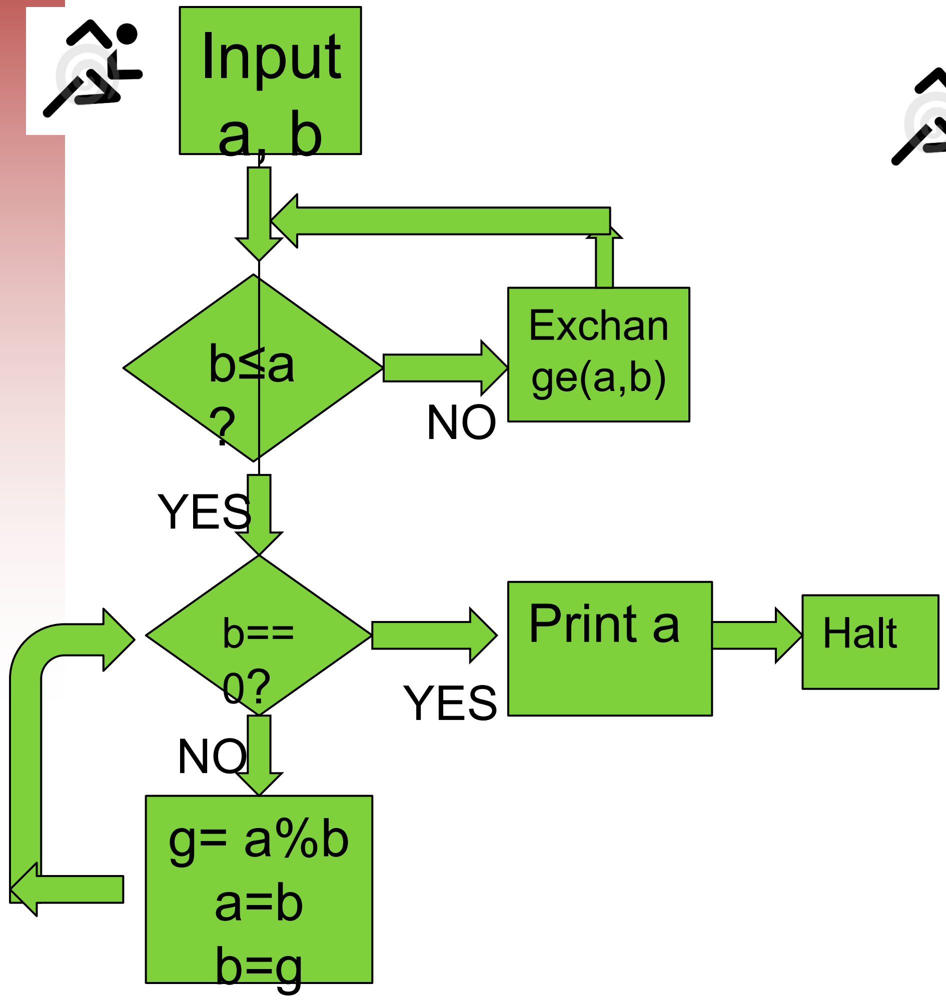
g= a%b
a=b
b=g

Program counter. At the next step to be executed. Initially at beginning.

State of the program is variables : boxes with names.

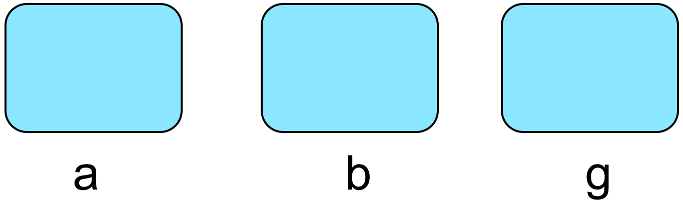


Running the program



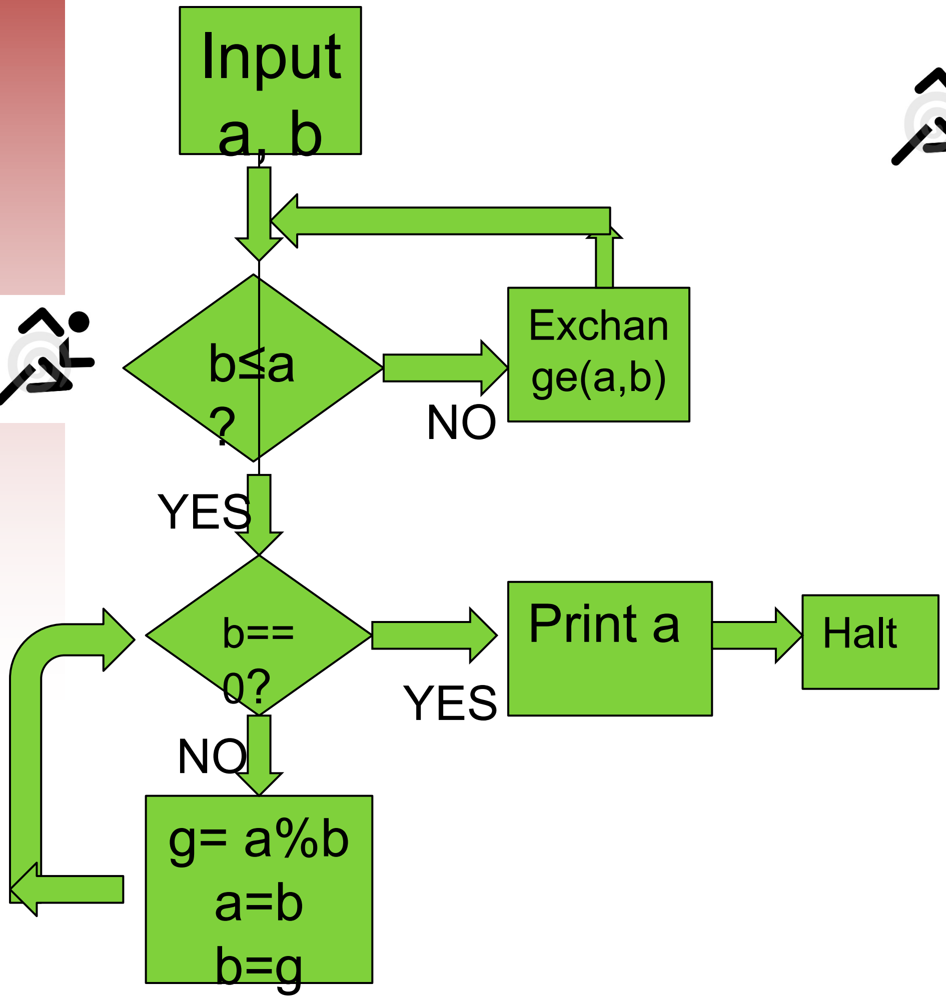
Program counter. At the next step to be executed. Initially at beginning.

State of the program is variables : boxes with names.



Now let us start running the flowchart. One step at a time.

Running the program



Program counter. At the next step to be executed. Initially at beginning.

State of the program is variables : boxes with names.

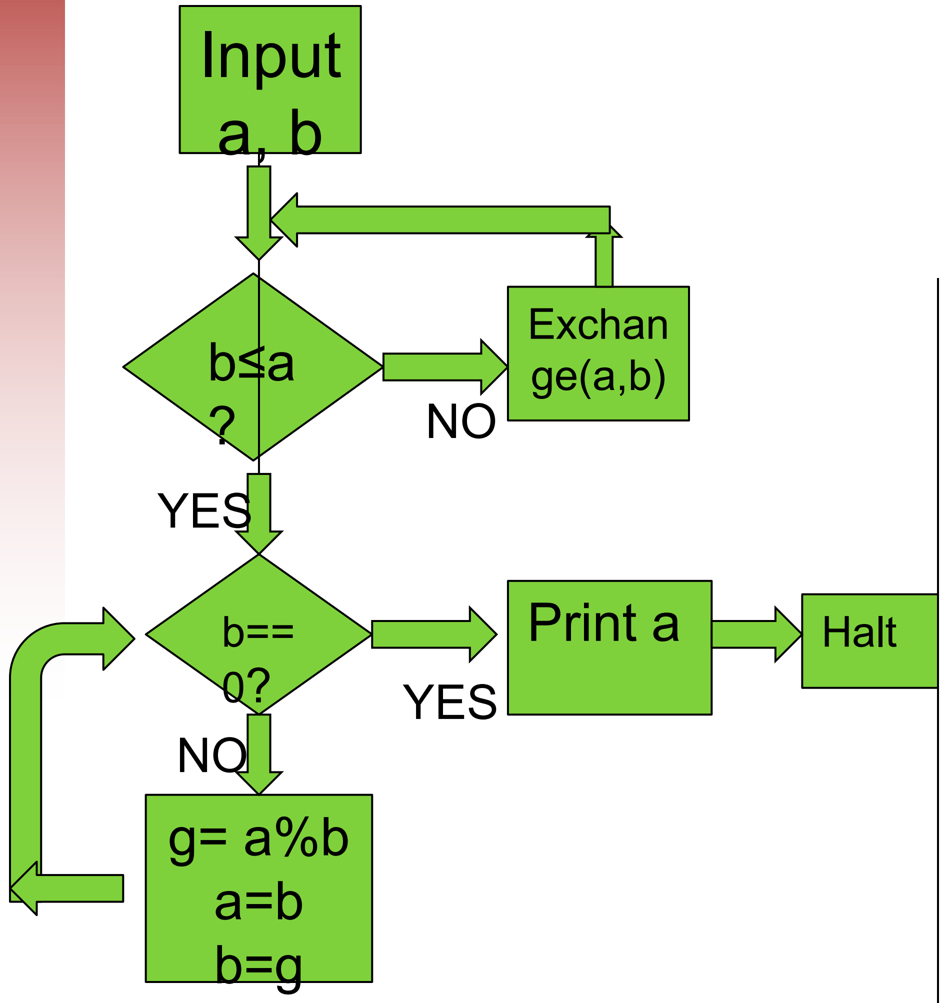
a	b	g

Now let us start running the flowchart. One step at a time.

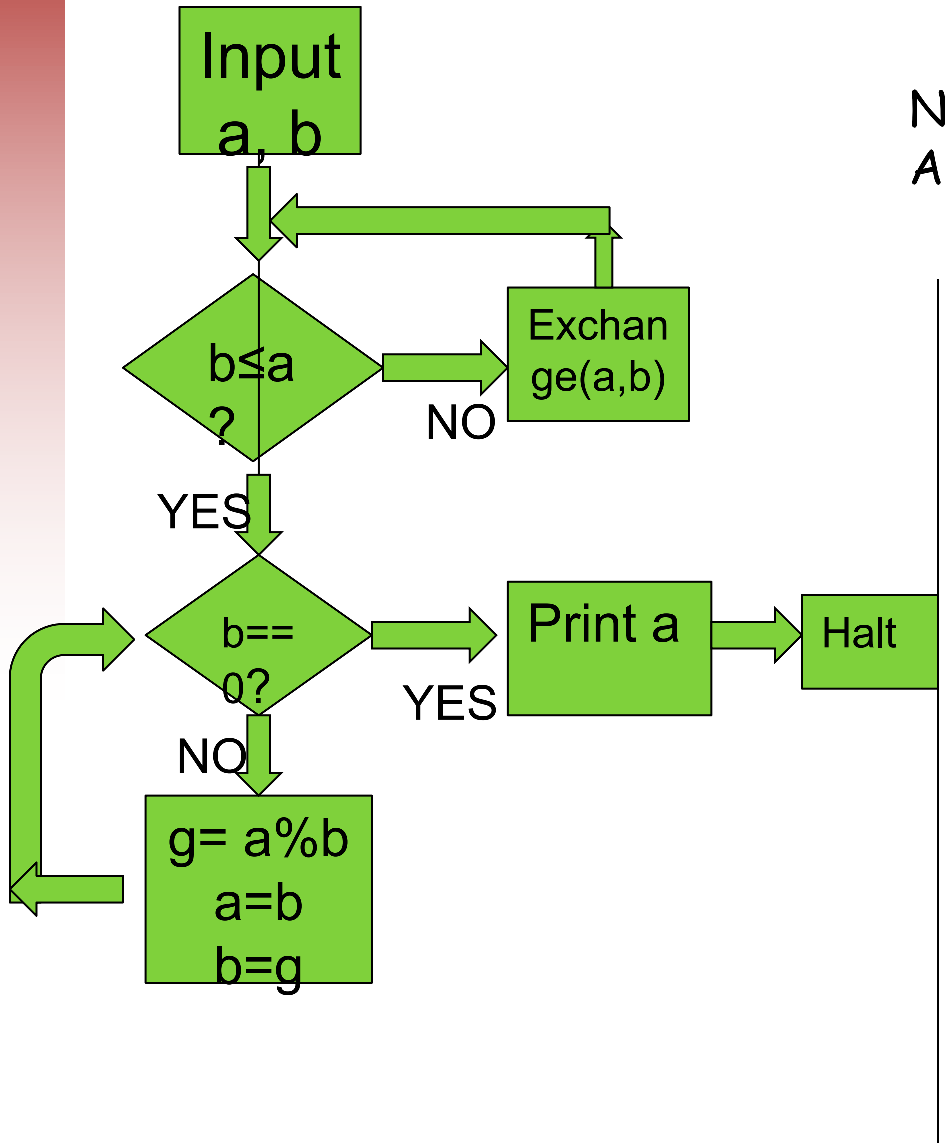
1. After input step:

a	b	g
8	6	??

Tracing the execution

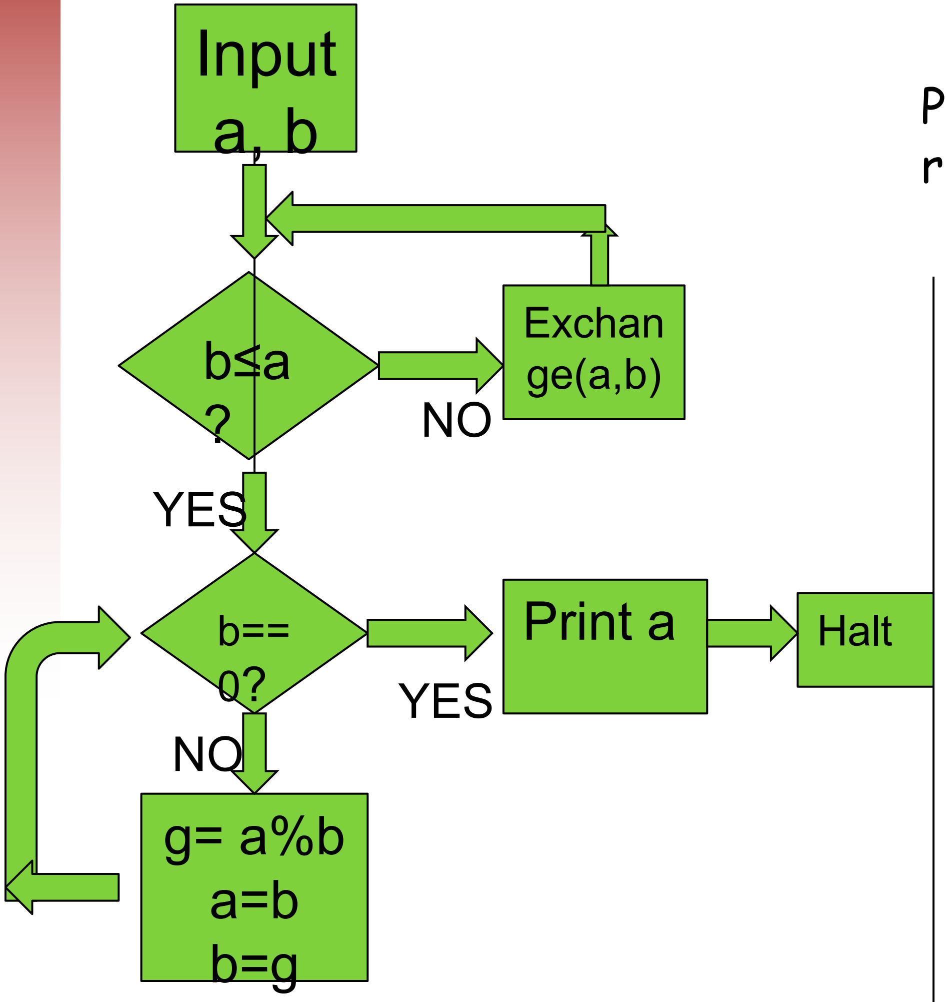


Tracing the execution



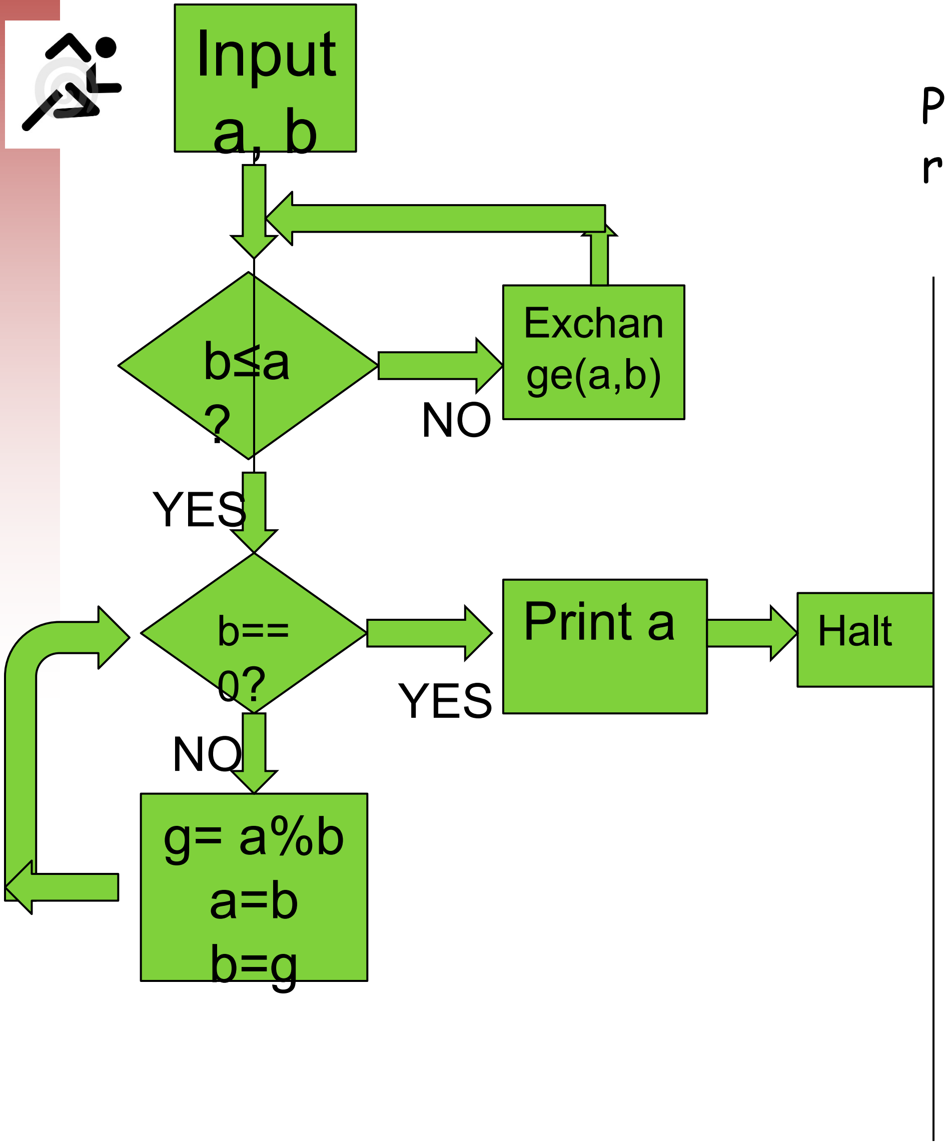
Now let us start running the flowchart.
Always one box at a time.

Tracing the execution



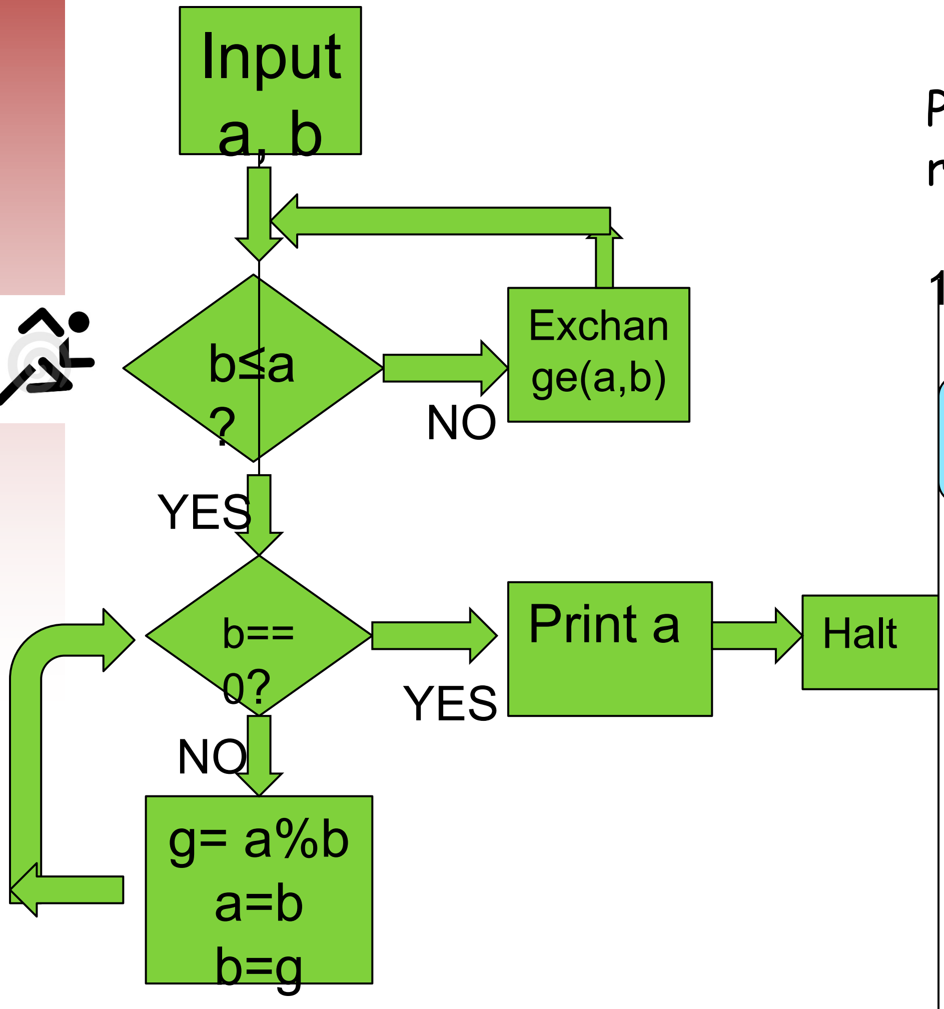
Program Counter is at the next step to be run

Tracing the execution



Program Counter is at the next step to be run

Tracing the execution

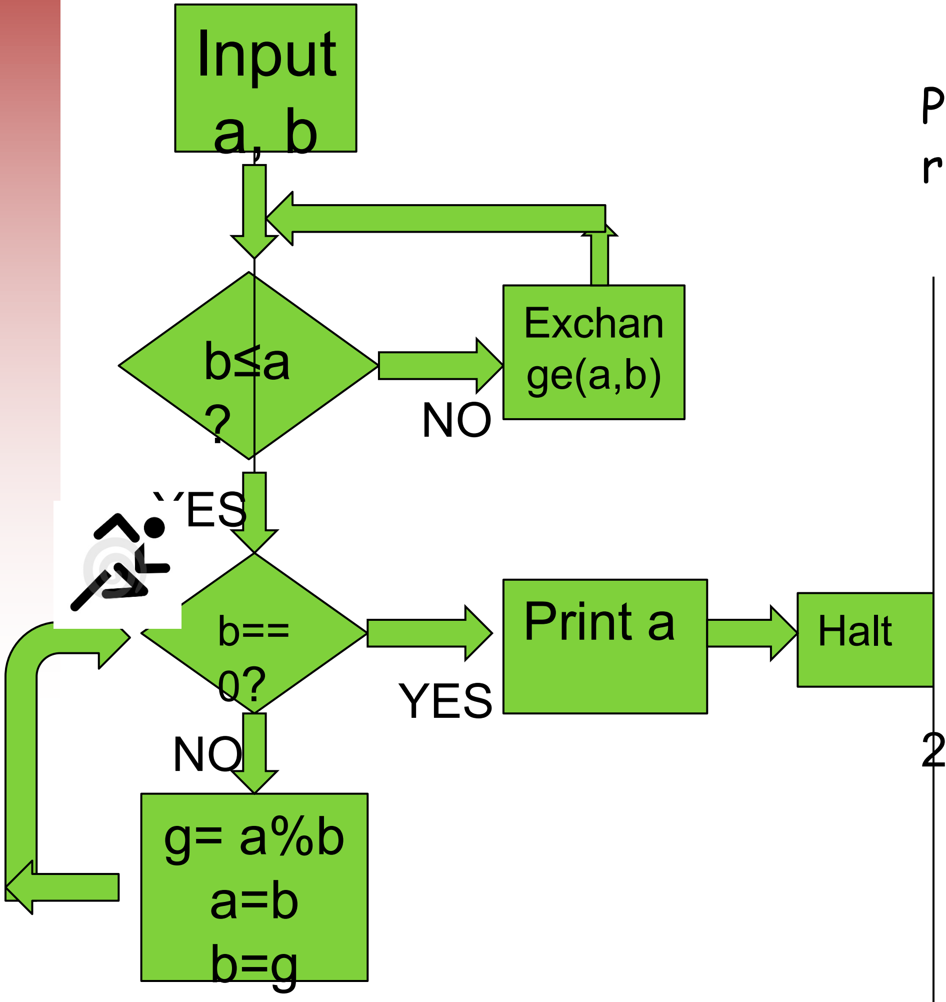


Program Counter is at the next step to be run

1. After input step:

8	6	??
a	b	g

Tracing the execution

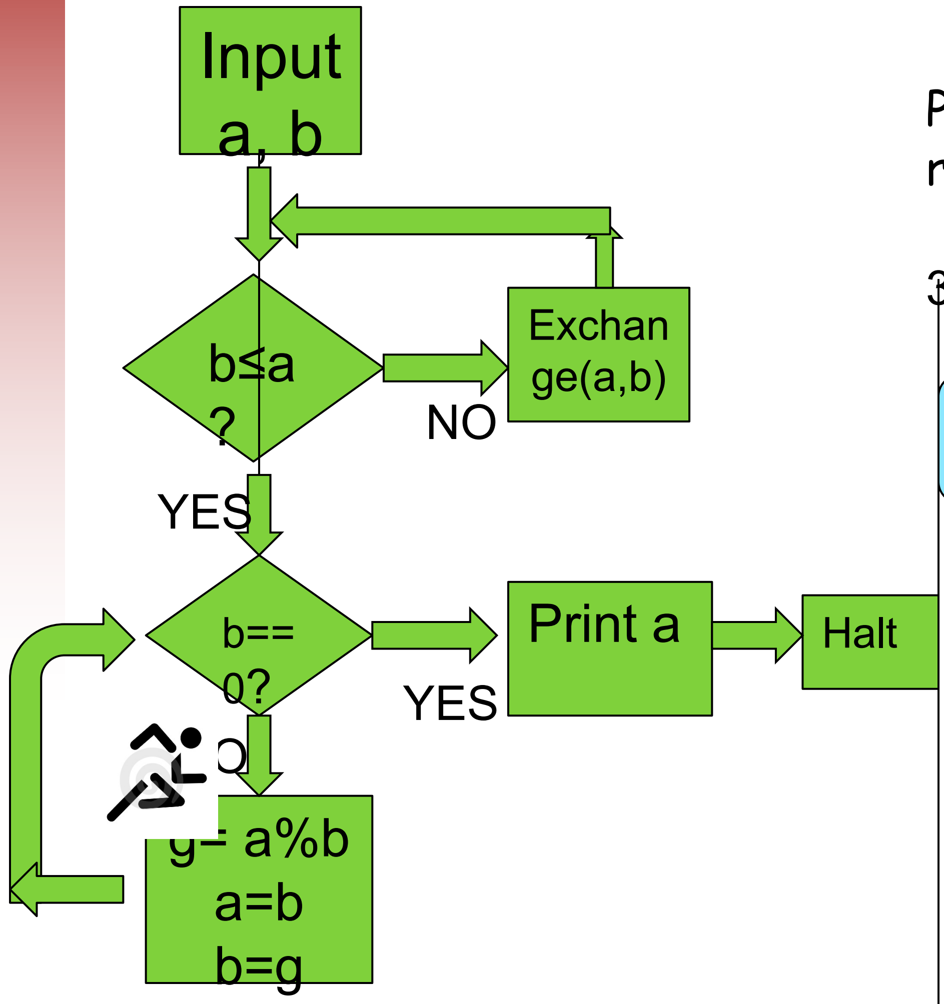


Program Counter is at the next step to be run

2. Test b < a? YES

8	6	??
a	b	g

Tracing the execution

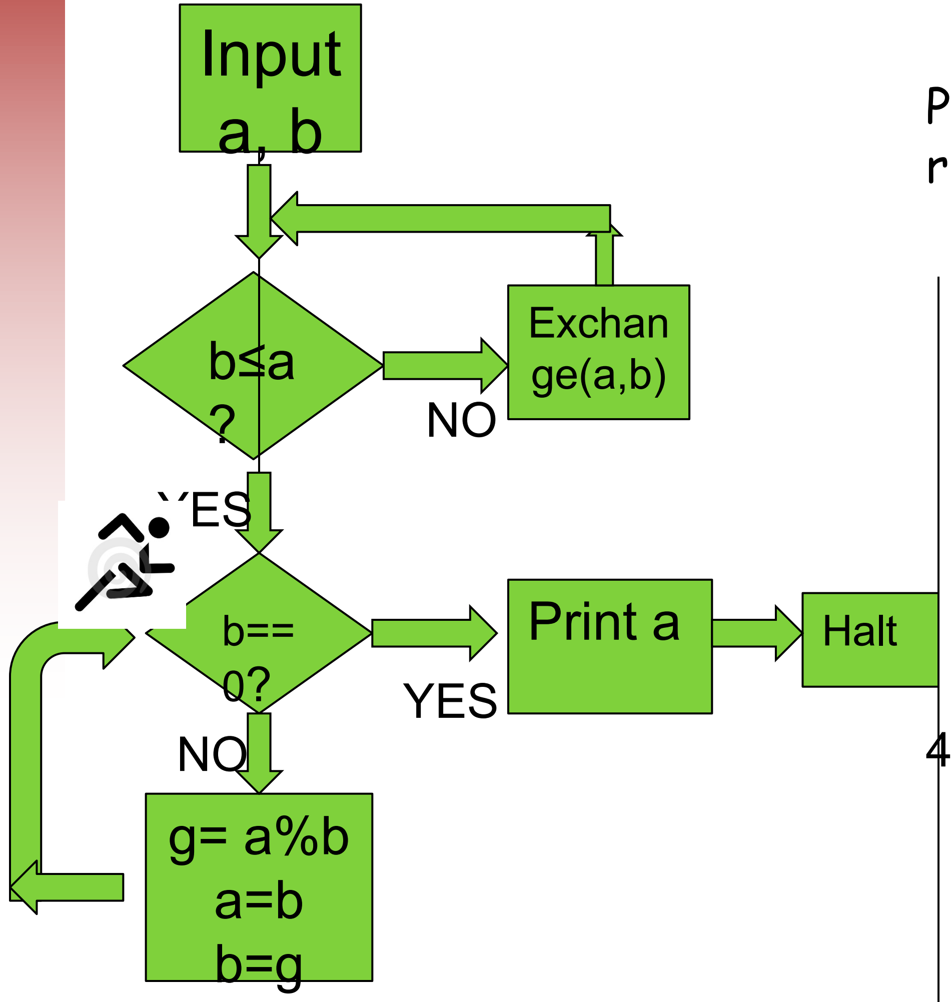


Program Counter is at the next step to be run

3. Test b==0? NO

8	6	??
a	b	g

Tracing the execution

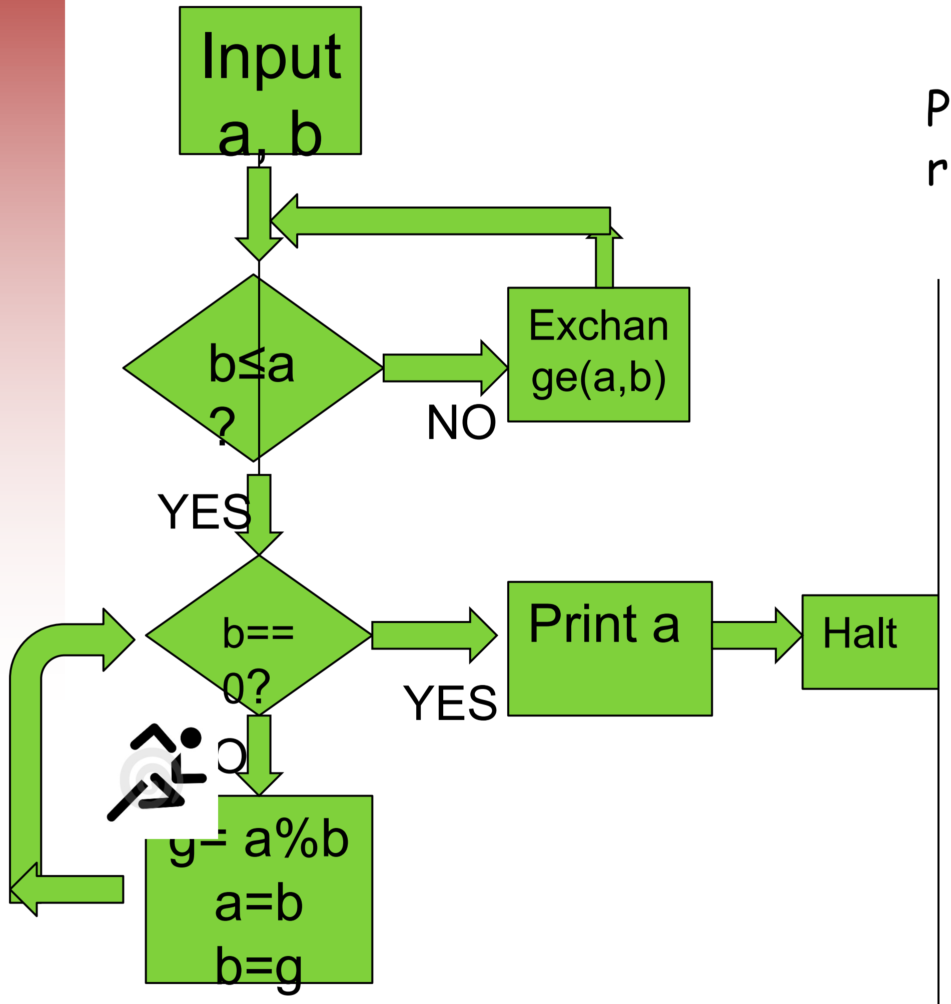


Program Counter is at the next step to be run

4. g=a%b; a=b; b=g;

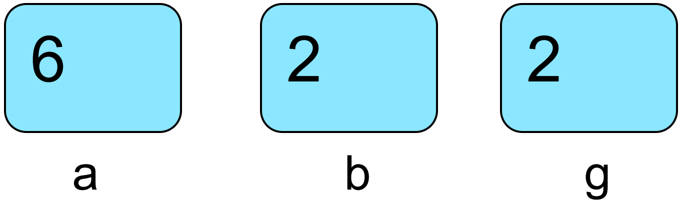
a	b	g
6	2	2

Tracing the execution



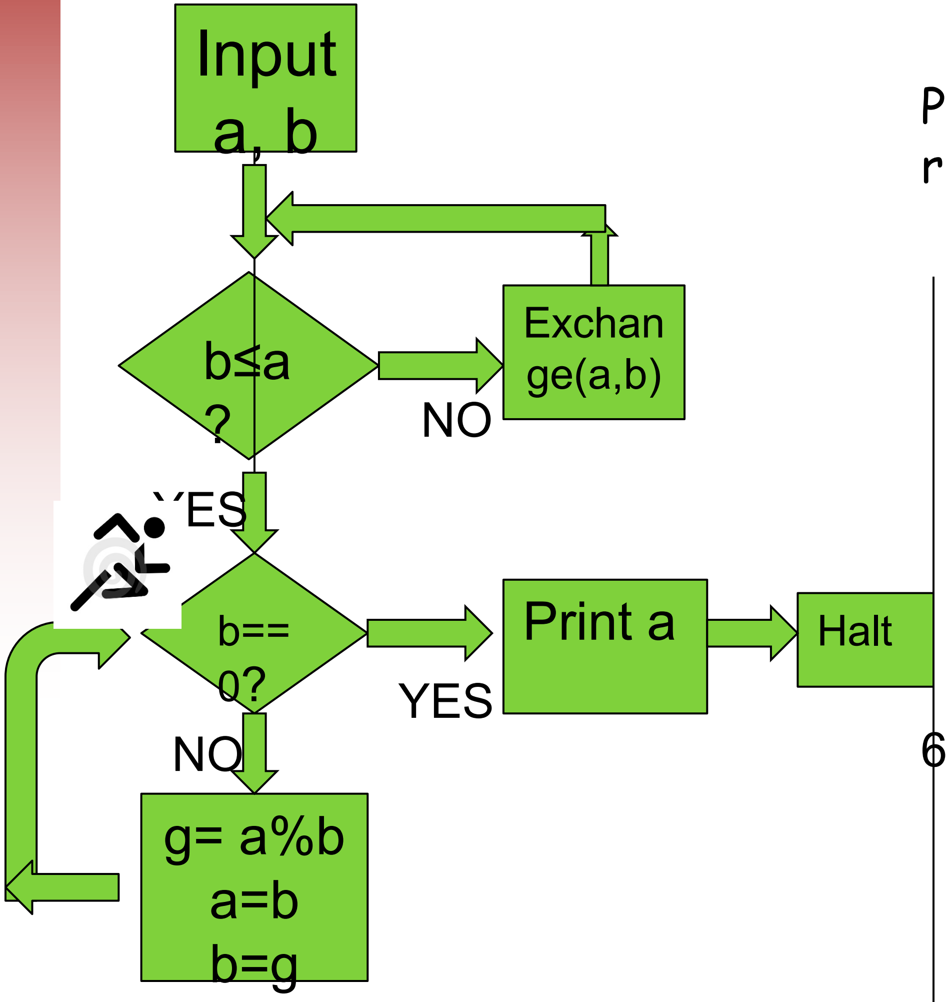
Program Counter is at the next step to be run

5. Test b==0? NO



a	b	g
6	2	2

Tracing the execution

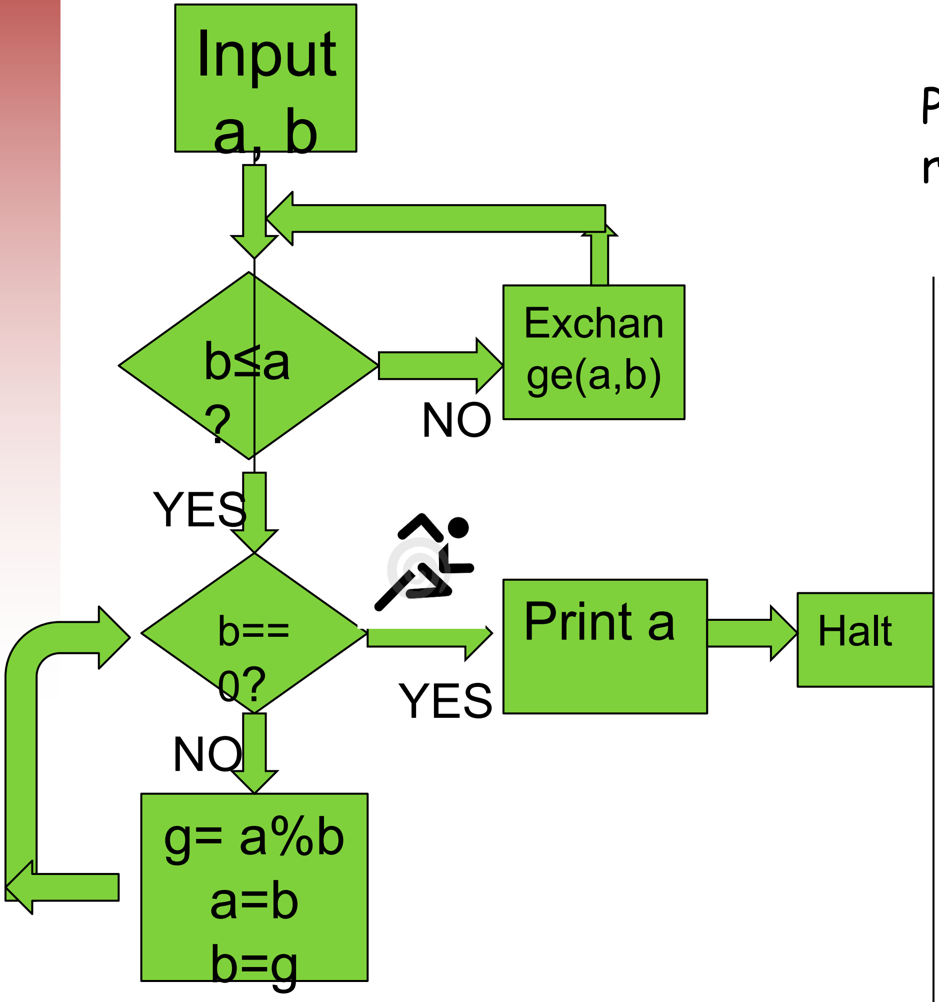


Program Counter is at the next step to be run

6. g=a%b; a=b; b=g;

a	b	g
2	0	0

Tracing the execution

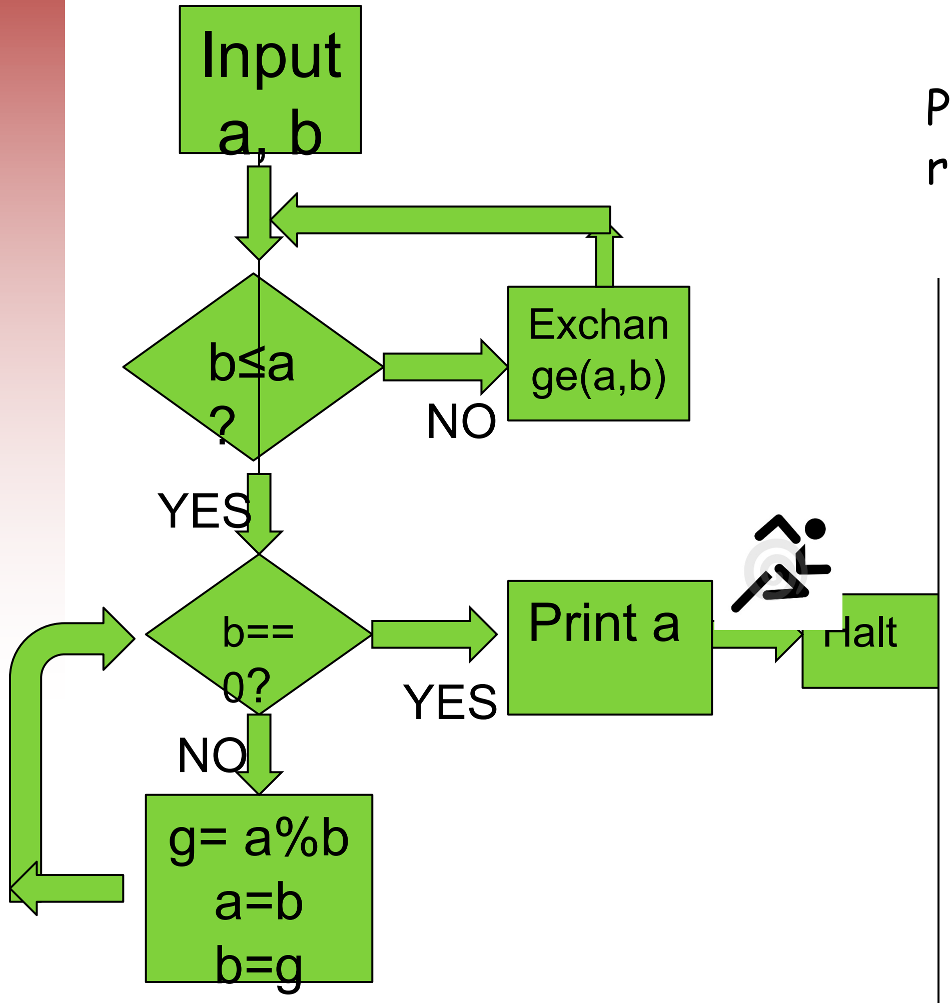


Program Counter is at the next step to be run

7. Test b==0? YES

a	b	g
2	0	0

Tracing the execution

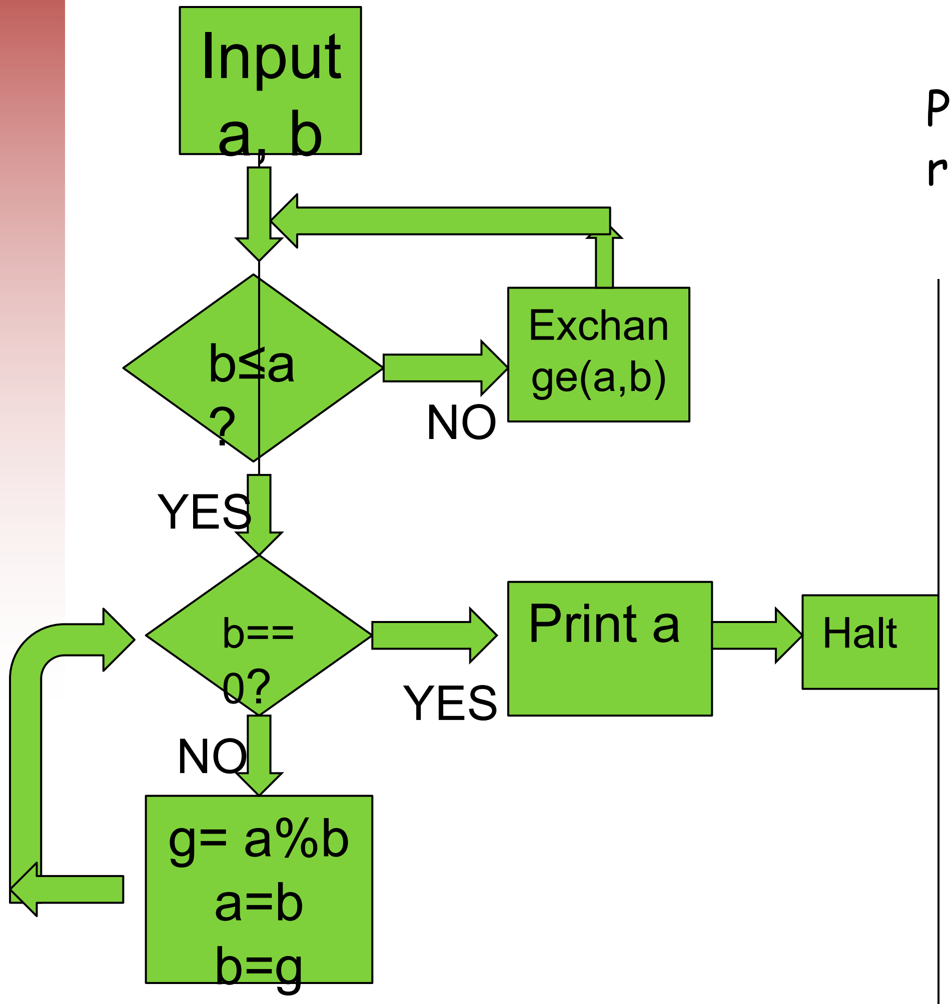


Program Counter is at the next step to be run

8. Print a

2

Tracing the execution



Program Counter is at the next step to be run

8. Print a

2

Multiple solutions and comparing them

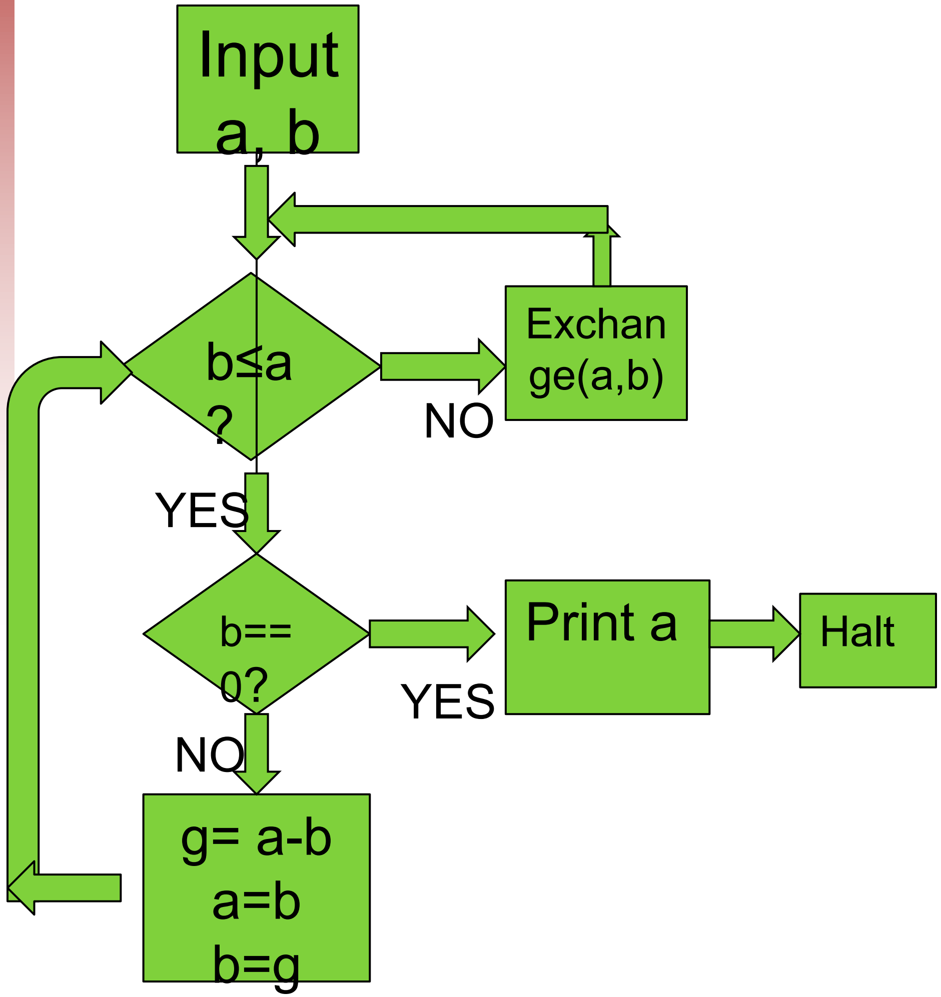
- How many times did we run in the loop? The fewer the better.

Multiple solutions and comparing them

- Multiple solutions are possible for the same problem.

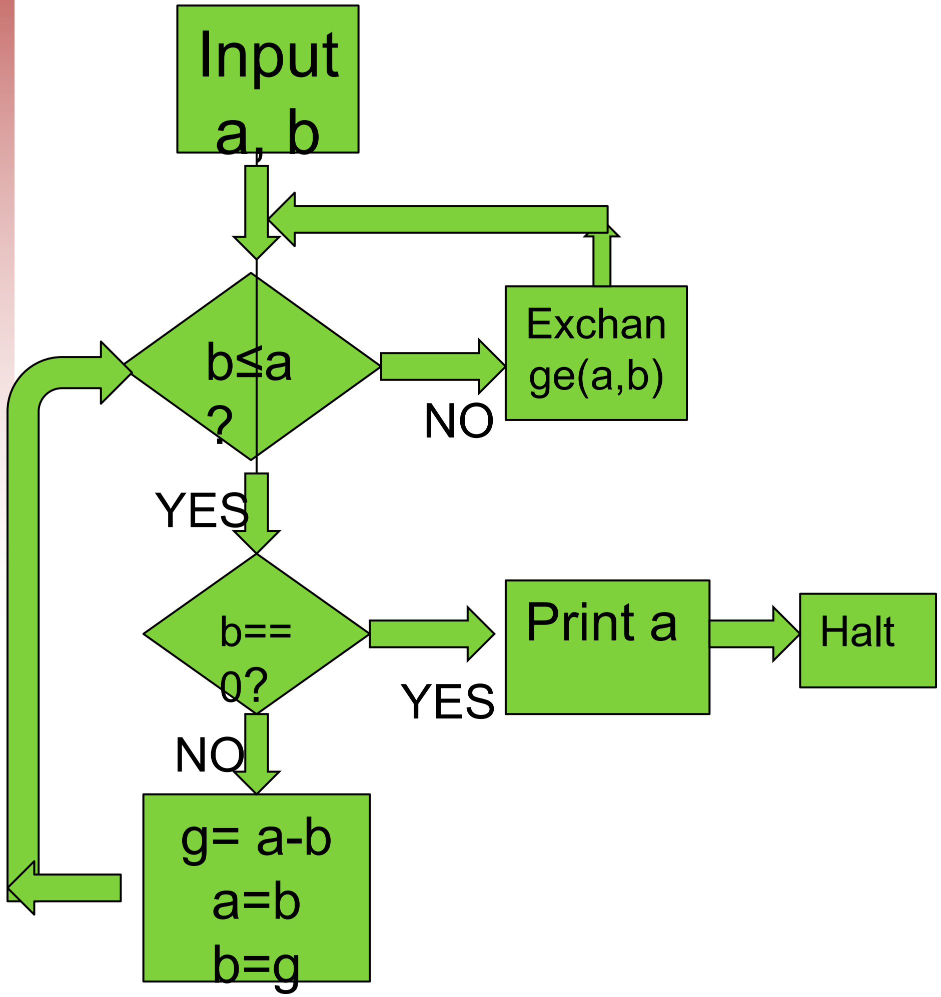
- How many times did we run in the loop? The fewer the better.

Multiple solutions and comparing them



- Multiple solutions are possible for the same problem.
- Is the adjacent flowchart correct for gcd?
- How many times did we run in the loop? The fewer the better.

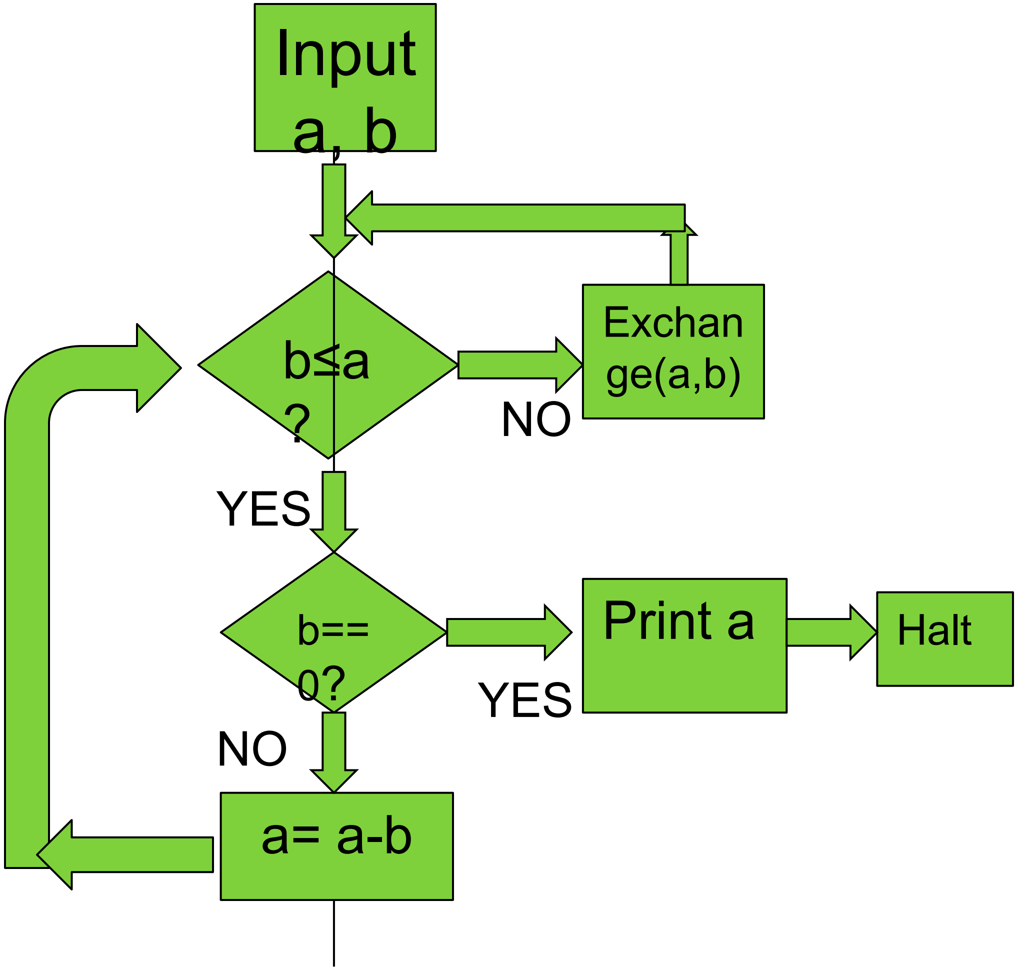
Multiple solutions and comparing them



- Multiple solutions are possible for the same problem.
- Is the adjacent flowchart correct for gcd?
- How many times did we run in the loop? The fewer the better.
- Is it seriously more: by an order of magnitude? Notion of complexity.

Another solution

- A (slower) alternative. How many times does the loop iterate?



Acknowledgments: This lecture slide is based on the material prepared by Prof. Sumit Ganguly, CSE, IIT Kanpur. The slide design is based on a template by Prof. Krithika Venkataramani.

Intro - Programming Cycle

Week1

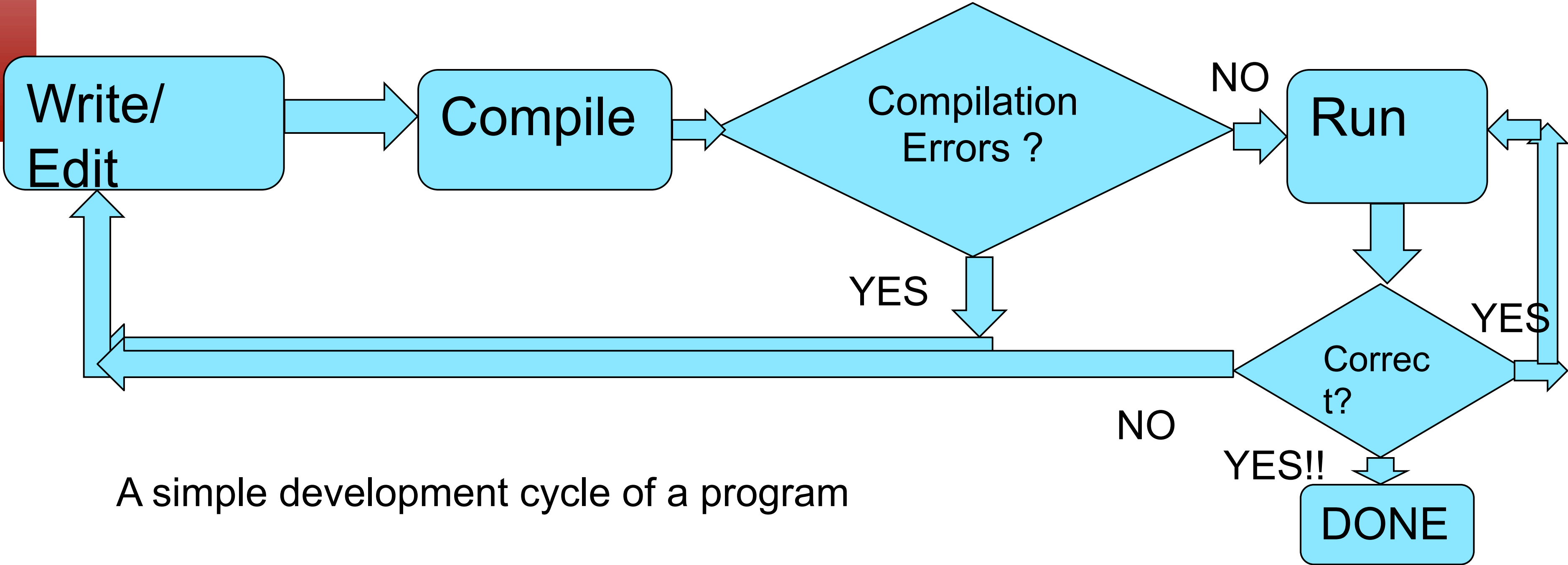
The Programming Cycle

The Programming Cycle

1. Write your program or edit (i.e., change or modify) your program.
2. Compile your program. If compilation fails, return to editing step.
3. Run your program. If output is not correct, return to editing step.

The Programming Cycle

1. Write your program or edit (i.e., change or modify) your program.
2. Compile your program. If compilation fails, return to editing step.
3. Run your program. If output is not correct, return to editing step.



A simple development cycle of a program

Editing

- Open an editor. An editor is a system program that lets you type in text, modify and update it.

- Create your program. Type in your program in an editor. For example use the program gedit or Notepad++. Save what you type into a file called sample.c.

Step 2: Compile

Step 2: Compile

- After editing, you have to COMPILE the program.

Step 2: Compile

- After editing, you have to COMPILE the program.
- The computer cannot execute a C program or the individual statements of a C program directly.

Step 2: Compile

- After editing, you have to COMPILE the program.
- The computer cannot execute a C program or the individual statements of a C program directly.
- For example, in C you can write

```
g = a %b
```

Step 2: Compile

- After editing, you have to COMPILE the program.
- The computer cannot execute a C program or the individual statements of a C program directly.
- For example, in C you can write

```
g = a %b
```

- The microprocessor cannot execute this statement. It translates it into an equivalent piece of code consisting of even more basic statements. For example

Step 2: Compile

- After editing, you have to COMPILE the program.
- The computer cannot execute a C program or the individual statements of a C program directly.
- For example, in C you can write

```
g = a %b
```

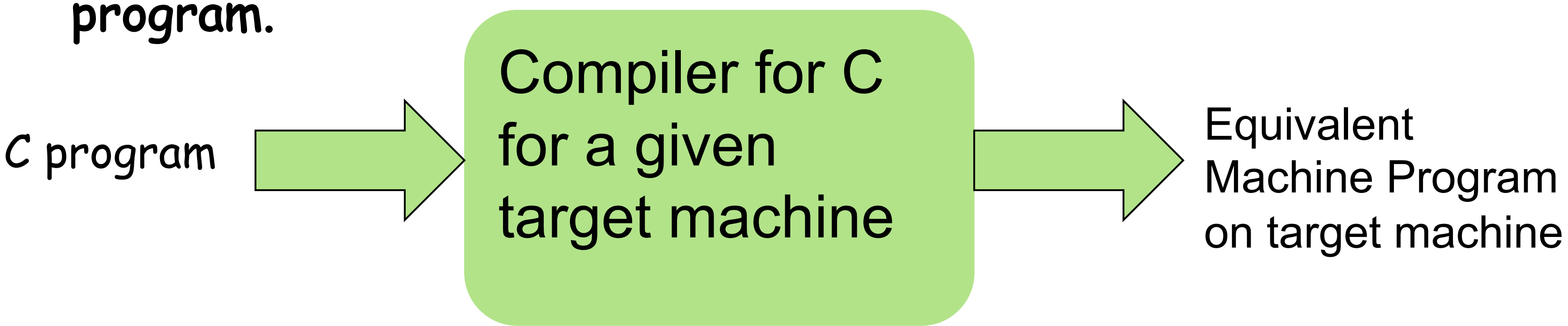
- The microprocessor cannot execute this statement. It translates it into an equivalent piece of code consisting of even more basic statements. For example
  - Load from memory location 0xF04 into register R1
  - Load from memory location 0xF08 into register R2
  - Integer divide contents of R1 by contents of R2 and keep remainder in register R3
  - Store contents of R3 into memory location 0xF12.

Why program in high level languages like C

- Writing programs in machine language is long, tedious and error-prone.

- They are also not portable—meaning program written for one machine may not work on another machine.

- Compilers work as a bridge.
- Take as input a C program and produce an equivalent machine program.


C program
Compiler for C for a given target machine
Equivalent Machine Program on target machine

How do you compile?

- On Unix/Linux systems you can **COMPILE** the program using the gcc command.

```
% gcc sample.c
%
```

- If there are no errors, then the system silently shows the prompt (%).
- If there are errors, the system will list the errors and line numbers. Then you can edit (change) your file, fix the errors and recompile.
- As long as there are compilation errors, the **EXECUTABLE** file is not created.

Compilation

- We will use the compiler *gcc* . The command is

```
% gcc  yourfilename.c
```

- gcc stands for Gnu C compiler.
- If there are no errors then gcc places the machine program in an executable format for your machine and calls it a.out.
- The file a.out is placed in your current working directory. More on directories in a little bit!

Simple! Program

sample.c: The program prints the message “Welcome to C”

Simple! Program

- **We will see some of the simplest C programs.**

sample.c: The program prints the message “Welcome to C”

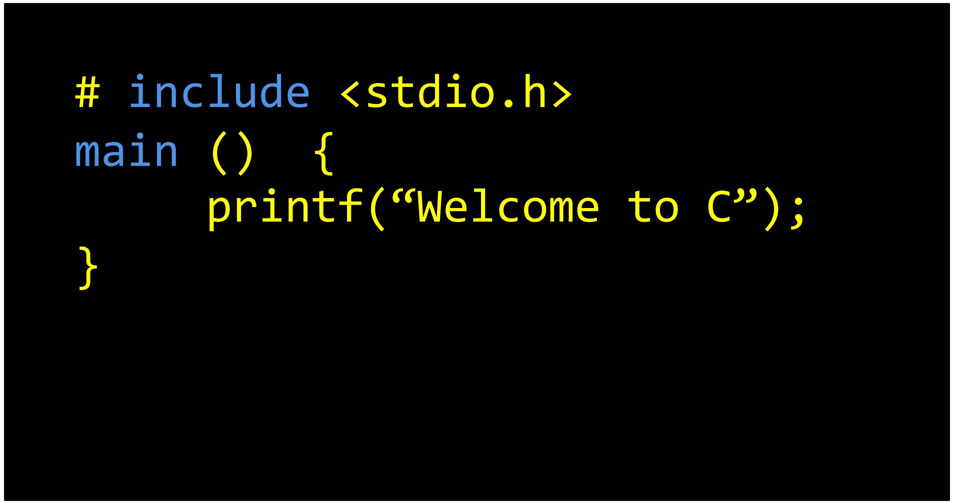
Simple! Program

- We will see some of the simplest C programs.
- Open an editor and type in the following lines. Save the program as sample.c

sample.c: The program prints the message “Welcome to C”

Simple! Program

- We will see some of the simplest C programs.
- Open an editor and type in the following lines. Save the program as sample.c



```
# include <stdio.h>
main ()  {
     printf(“Welcome to C”);
}
```

sample.c: The program prints the message “Welcome to C”

Compile and Run

Compile and Run

- Now compile the program. System compiles without errors.

```
% gcc sample.c
%
```

- Compilation creates the executable file a.out by default.

Compile and Run

- Now compile the program. System compiles without errors.

```
% gcc sample.c
%
```

- Compilation creates the executable file a.out by default.
- Now run the program. The screen looks like this:

Compile and Run

- Now compile the program. System compiles without errors.

```
% gcc sample.c
%
```

- Compilation creates the executable file a.out by default.
- Now run the program. The screen looks like this:

```
% ./a.out
Welcome to C%
```

Program statements

Program statements

```
# include <stdio.h>

main ()
{
   printf(“Welcome to C”);
}
```

Program statements

```
# include <stdio.h>

main ()
{
    printf(“Welcome to C”);
}
```

1. This tells the C compiler to include the standard input output library.

2. Include this line routinely as the first line of your C file.

Program statements

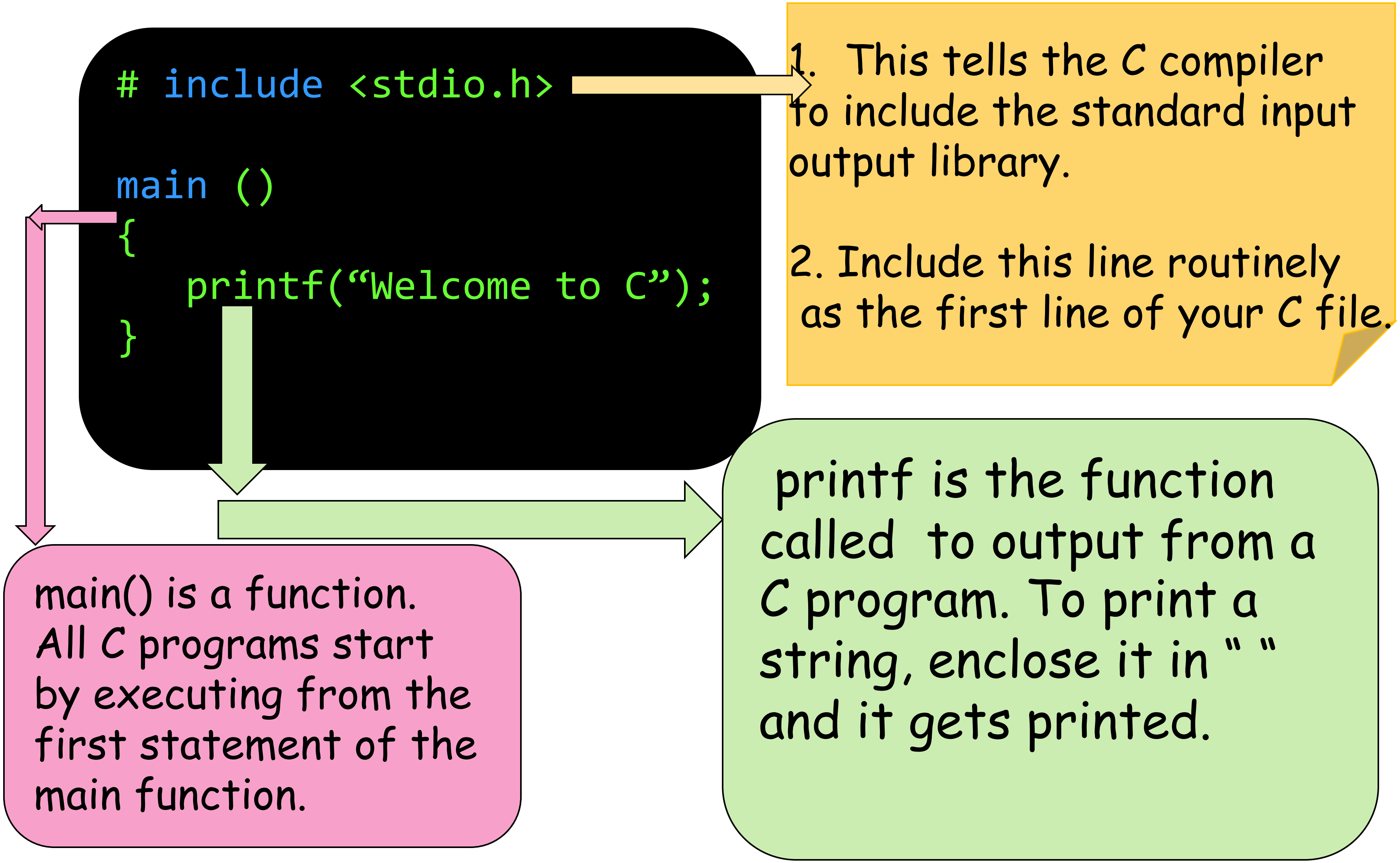
```
# include <stdio.h>

main ()
{
   printf("Welcome to C");
}
```

1. This tells the C compiler to include the standard input output library.

2. Include this line routinely as the first line of your C file.

main() is a function.
All C programs start by executing from the first statement of the main function.

Program statements



```
# include <stdio.h>

main ()
{
    printf(“Welcome to C”);
}
```

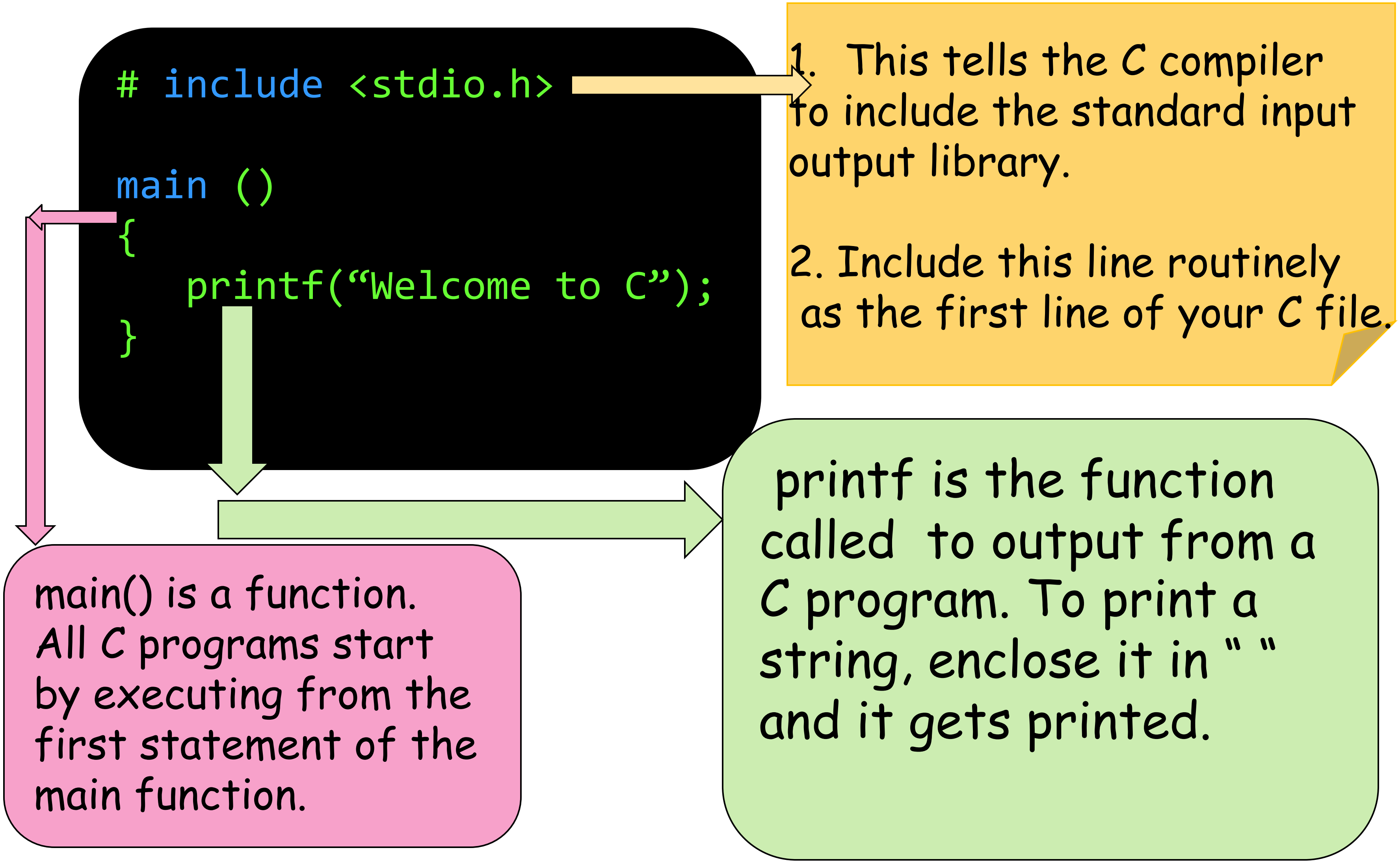
1. This tells the C compiler to include the standard input output library.

2. Include this line routinely as the first line of your C file.

main() is a function. All C programs start by executing from the first statement of the main function.

printf is the function called to output from a C program. To print a string, enclose it in “ “ and it gets printed.

Program statements



```
# include <stdio.h>

main ()
{
    printf(“Welcome to C”);
}
```

1. This tells the C compiler to include the standard input output library.

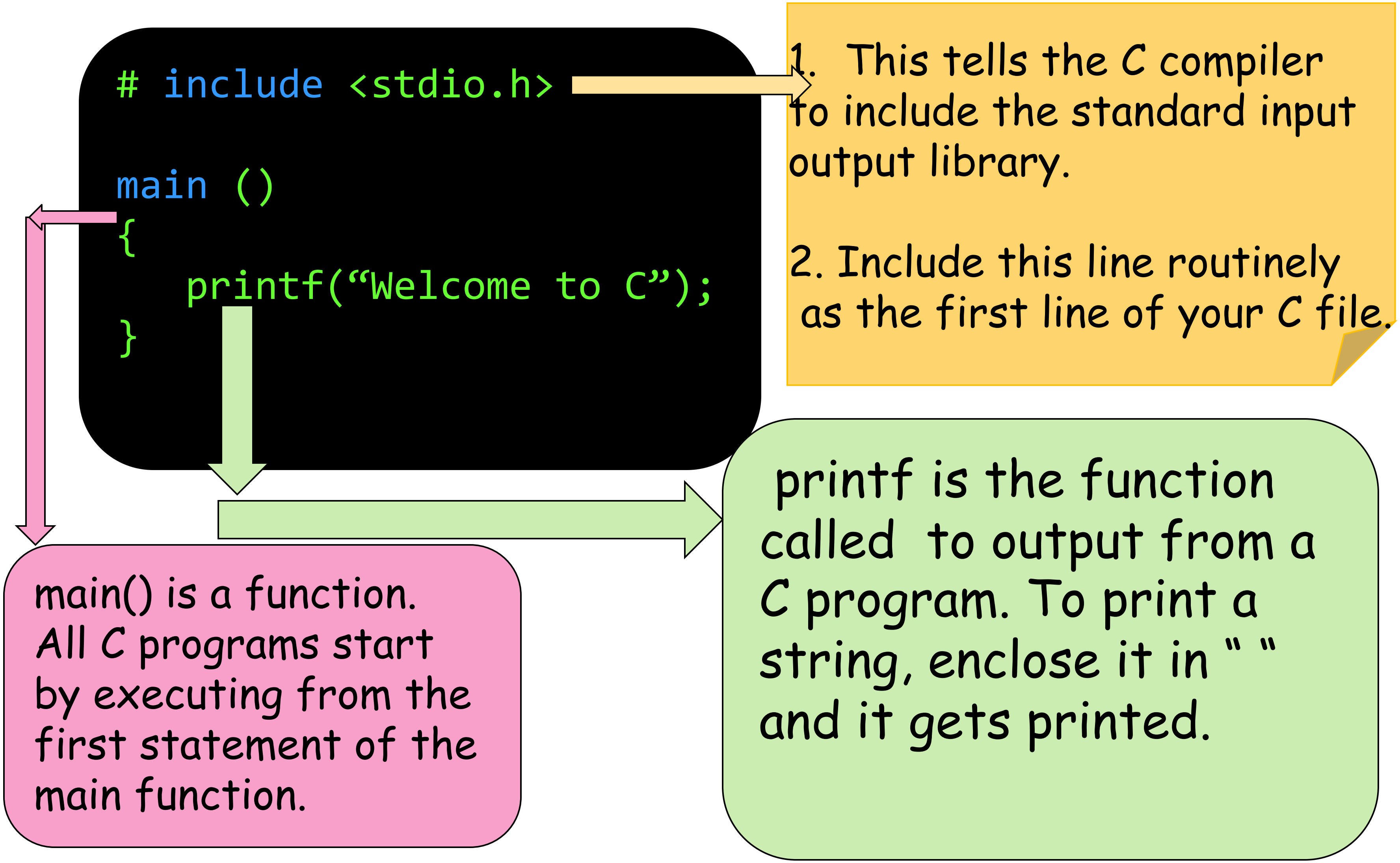
2. Include this line routinely as the first line of your C file.

main() is a function. All C programs start by executing from the first statement of the main function.

printf is the function called to output from a C program. To print a string, enclose it in “ “ and it gets printed.

```
printf(“Welcome to C”);
```

Program statements



```
# include <stdio.h>

main ()
{
    printf(“Welcome to C”);
}
```

1. This tells the C compiler to include the standard input output library.

2. Include this line routinely as the first line of your C file.

main() is a function. All C programs start by executing from the first statement of the main function.

printf is the function called to output from a C program. To print a string, enclose it in “ “ and it gets printed.

`printf(“Welcome to C”);` is a statement in C. Statements in C end in semicolon ;

Errors

Errors

- Let us systematically enumerate a few common errors.

Errors

- Let us systematically enumerate a few common errors.

1. Forgetting to include stdio.h.

Errors

- Let us systematically enumerate a few common errors.

1. Forgetting to include stdio.h.
2. Forgetting main function

Errors

- Let us systematically enumerate a few common errors.

1. Forgetting to include stdio.h.
2. Forgetting main function
3. Forgetting semicolon

Errors

- Let us systematically enumerate a few common errors.

1. Forgetting to include stdio.h.
2. Forgetting main function
3. Forgetting semicolon
4. Forgetting open or close brace ({ or }).

Errors

- Let us systematically enumerate a few common errors.

1. Forgetting to include stdio.h.
2. Forgetting main function
3. Forgetting semicolon
4. Forgetting open or close brace ({ or }).
5. Forgetting to close the double quote.

Errors

- Let us systematically enumerate a few common errors.

1. Forgetting to include stdio.h.
2. Forgetting main function
3. Forgetting semicolon
4. Forgetting open or close brace ({ or }).
5. Forgetting to close the double quote.

Try deliberately making these mistakes in your code. Save them and try to compile. Study the error messages for each.

Familiarity with error messages will help you find coding errors later.

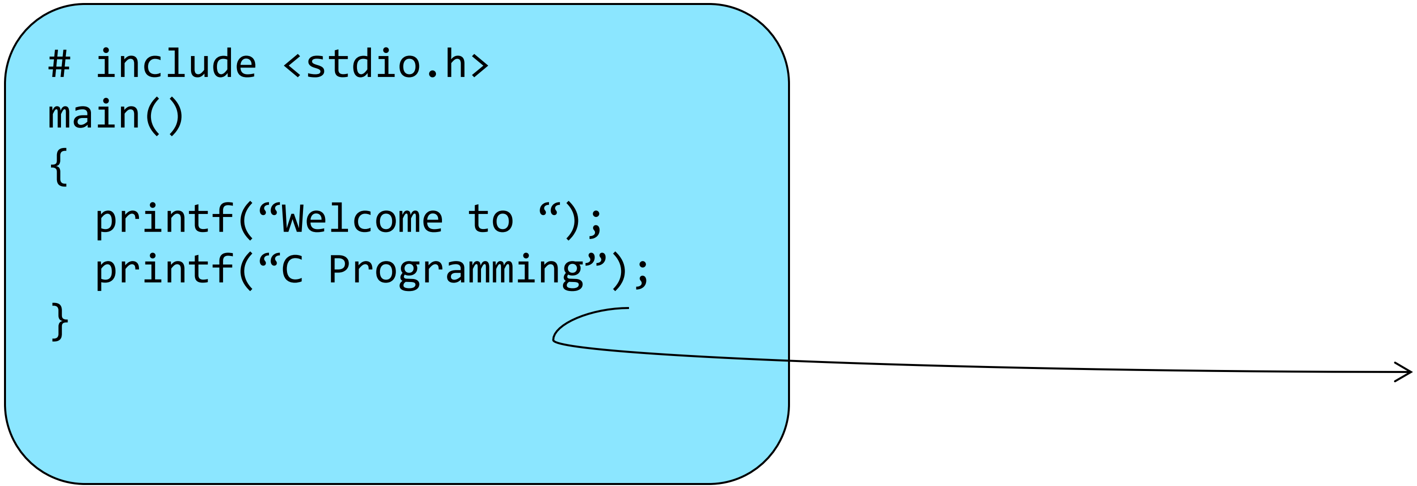
Acknowledgments: This lecture slide is based on the material prepared by Prof. Sumit Ganguly, CSE, IIT Kanpur. The slide design is based on a template by Prof. Krithika Venkataramani.

Intro - Tracing a simple program

Week1

Another simple program

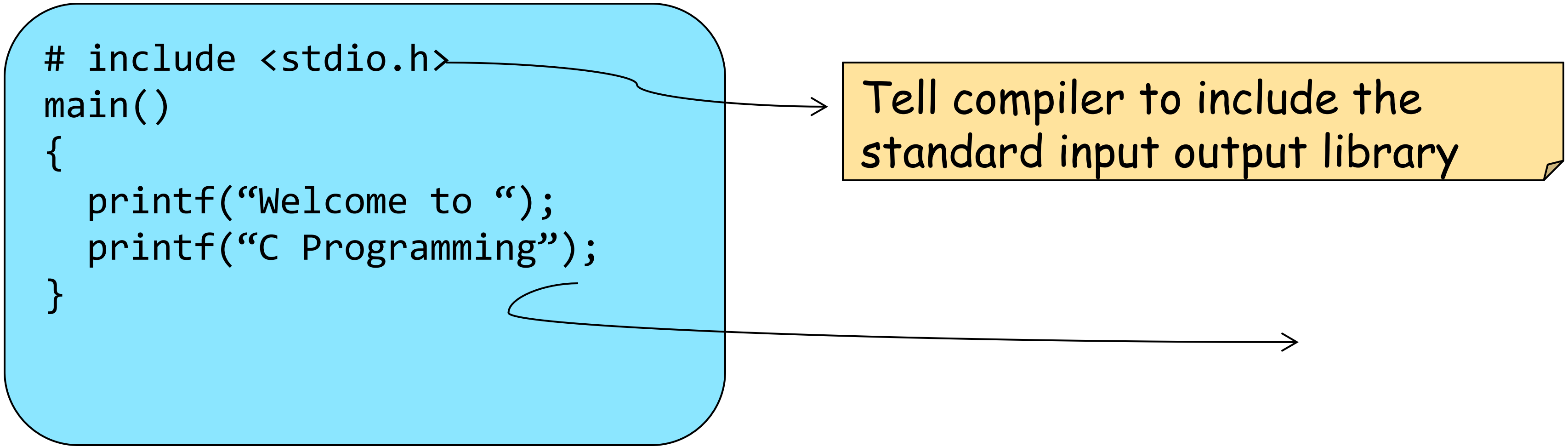
Another simple program



```
# include <stdio.h>
main()
{
  printf("Welcome to ");
  printf("C Programming");
}
```

sample.c

Another simple program



```
# include <stdio.h>
main()
{
   printf(“Welcome to “);
   printf(“C Programming”);
}
```

Tell compiler to include the standard input output library

sample.c

Another simple program

```
# include <stdio.h>
main()
{
   printf(“Welcome to “);
   printf(“C Programming”);
}
```

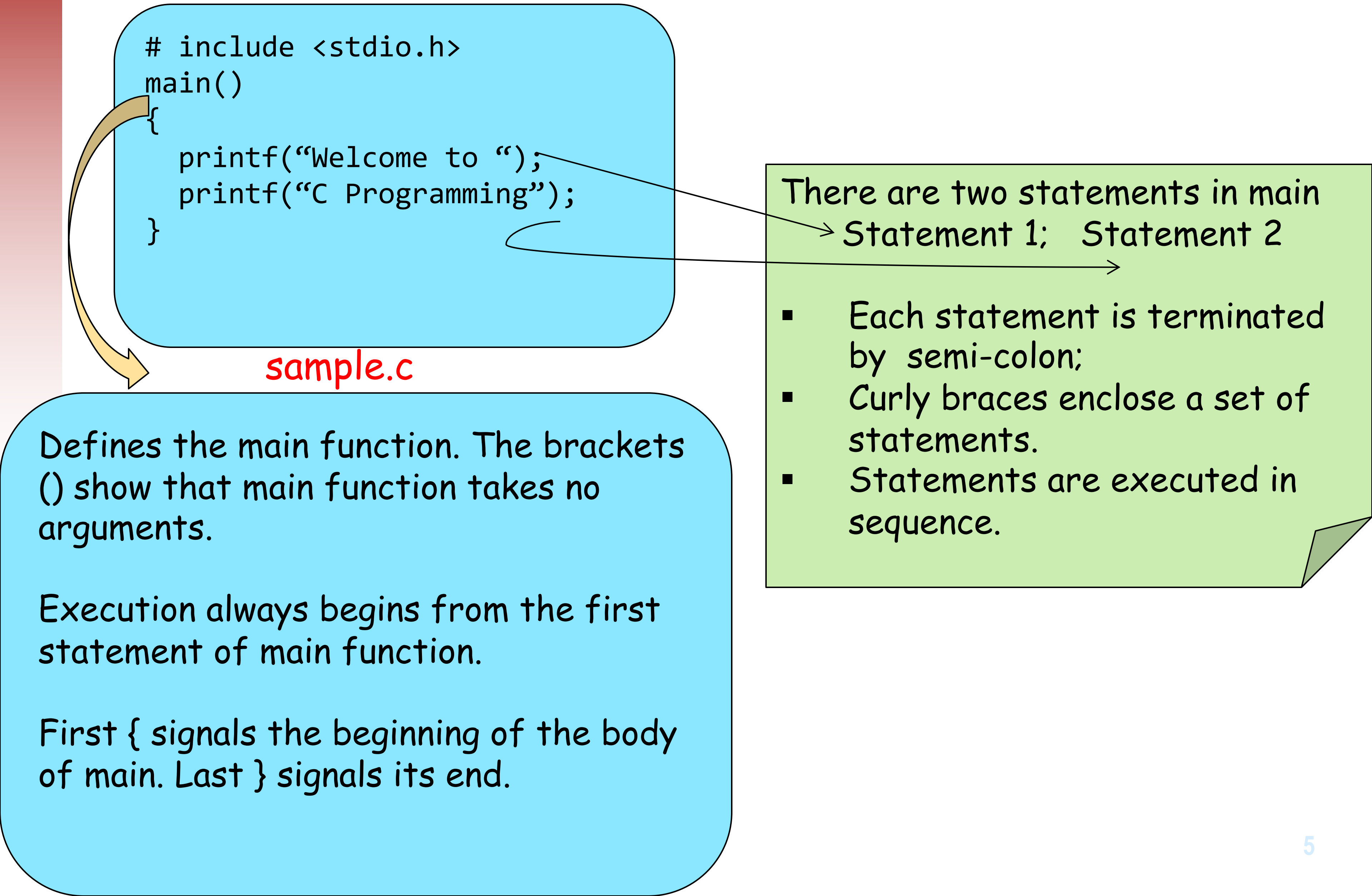
sample.c

Defines the main function. The brackets () show that main function takes no arguments.

Execution always begins from the first statement of main function.

First { signals the beginning of the body of main. Last } signals its end.

Another simple program



```
# include <stdio.h>
main()
{
   printf(“Welcome to “);
   printf(“C Programming”);
}
```

sample.c

There are two statements in main
Statement 1;   Statement 2

- Each statement is terminated by semi-colon;
- Curly braces enclose a set of statements.
- Statements are executed in sequence.

Defines the main function. The brackets () show that main function takes no arguments.

Execution always begins from the first statement of main function.

First { signals the beginning of the body of main. Last } signals its end.

Another simple program

```
# include <stdio.h>
main()
{
  printf(“Welcome to “);
  printf(“C Programming”);
}
```

sample.c

There are two statements in main

Statement 1; Statement 2

- Each statement is terminated by semi-colon;
- Curly braces enclose a set of statements.
- Statements are executed in sequence.

Defines the main function. The brackets () show that main function takes no arguments.

Execution always begins from the first statement of main function.

First { signals the beginning of the body of main. Last } signals its end.

Compile and Run

```
%gcc sample.c
%./a.out
Welcome to C Programming%
```

Tracing the Execution

Tracing the Execution

```
# include <stdio.h>
main()
{
        printf(“Welcome to “);
        printf(“C Programming”);
}
```

Tracing the Execution

```
# include <stdio.h>
main()
{
   🏃 printf(“Welcome to “);
      printf(“C Programming”);
}
```

- Program counter 🏃 starts at the first executable statement of main.

Tracing the Execution

```
Line
No.
1   # include <stdio.h>
2   main()
3   {
4          printf(“Welcome to “);
5          printf(“C Programming”);
6   }
```

- Program counter starts at the first executable statement of main.
- Line numbers of C program are given for clarity.

Tracing the Execution

```
Line
No.
  1  # include <stdio.h>
  2  main()
  3  {
  4          printf(“Welcome to “);
  5          printf(“C Programming”);
  6  }
```

- Program counter starts at the first executable statement of main.
- Line numbers of C program are given for clarity.
- Let us run the program, one step at a time.

Tracing the Execution

```
Line
No.
1  # include <stdio.h>
2  main()
3  {
4          printf(“Welcome to “);
5          printf(“C Programming”);
6  }
```

Output:

- Program counter starts at the first executable statement of main.
- Line numbers of C program are given for clarity.
- Let us run the program, one step at a time.
- Program terminates gracefully when main ``returns”.

Tracing the Execution

Line No.

```
1  # include <stdio.h>
2  main()
3  {
4        printf(“Welcome to “);
5        printf(“C Programming”);
6  }
```

Output: After lines 3,4

```
Welcome to
```

- Program counter starts at the first executable statement of main.
- Line numbers of C program are given for clarity.
- Let us run the program, one step at a time.
- Program terminates gracefully when main ``returns”.

Tracing the Execution

```
Line
No.
1  # include <stdio.h>
2  main()
3  {
4         printf(“Welcome to “);
5         printf(“C Programming”);
6  }
```

Output:

After lines 5,6

```
Welcome to C Programming%
```

- Program counter starts at the first executable statement of main.
- Line numbers of C program are given for clarity.
- Let us run the program, one step at a time.
- Program terminates gracefully when main ``returns”.

Tracing the Execution

Line No.

```
1  # include <stdio.h>
2  main()
3  {
4          printf(“Welcome to “);
5          printf(“C Programming”);
6  }
```

Output: After lines 5,6

```
Welcome to C Programming%
```

- Program counter starts at the first executable statement of main.
- Line numbers of C program are given for clarity.
- Let us run the program, one step at a time.
- Program terminates gracefully when main ``returns”.

Program Comments

Program Comments

```
# include <stdio.h>
/* a simple C program */
main()
{
        printf(“Welcome to “);      /* first print */
        printf(“C Programming”);    /* second print */
}
```

Program Comments

```
# include <stdio.h>
/* a simple C program */
main()
{
        printf(“Welcome to “);      /* first print */
        printf(“C Programming”);    /* second print */
}
```

- These are called COMMENTS.
- Any text between successive /* and */ is a comment and will be ignored by the compiler.
- Comments are NOT part of the program.
- They are written for us to understand or explain the program better.
- Comments can be short or long. Any number of comments may be included.
- It is a very good idea to comment your programs. For larger programs, industry, this is a must. Will help you and other developers understand and maintain programs.

Notes*

- Just as main() is a function, printf("…") is also a function. printf is a library function from the standard input output library, which is why we inserted the statement

```
# include <stdio.h>
```

- printf takes as arguments a sequence of characters in double quotes, like "Welcome to". A sequence of characters in double quotes is called a *string constant.*

- We "call" functions that we define or from the libraries.

Printing in different lines
The newline character

Printing in different lines
The newline character

- All letters, digits, comma, underscore are called characters. There are 256 characters in C.

‘a’…’z’ ‘A’ .. ‘Z’ ‘0’..’9’ ‘@’ ‘.’ ’,’ ‘!’ ‘’’ ‘%’
‘^’ ‘&’ etc..

Printing in different lines
The newline character

- All letters, digits, comma, underscore are called characters. There are 256 characters in C.

‘a’…’z’ ‘A’ .. ‘Z’ ‘0’..’9’ ‘@’ ‘.’ ’,’ ‘!’ ‘’’ ‘%’
‘^’ ‘&’ etc..

- There is a special character called newline. In C it is denoted as ‘\n’

Printing in different lines
The newline character

- All letters, digits, comma, underscore are called characters. There are 256 characters in C.

‘a’…’z’ ‘A’ .. ‘Z’ ‘0’..’9’ ‘@’ ‘.’ ’,’ ‘!’ ‘’’ ‘%’
‘^’ ‘&’ etc..

- There is a special character called newline. In C it is denoted as ‘\n’

- When used in printf, it causes the current output line to end and printing will start at the next line.

The newline character

- Newline character ‘\n’ is like any other letter and can be used multiple times in a line
- “…\nC…” is treated as …’\n’ followed by ‘C’.

```
#include <stdio.h>
main()
{
    printf(“Welcome to \n”);
    printf(“C programming\n”);
}
```

When we compile and execute,

```
$./a.out
Welcome to
C programming
$
```

Last on newlines

- To repeat, newline character '\n' is like any other character. It can be used multiple times. Another example.

```
#include <stdio.h>

main()
{
    printf(“Welcome to\n\nC\n”);
}
```

- When we compile and execute, we have the following.

```
$./a.out
Welcome to

C
$
```

Acknowledgments: This lecture slide is based on the material prepared by Prof. Sumit Ganguly, CSE, IIT Kanpur. The slide design is based on a template by Prof. Krithika Venkataramani.