Hack assembly II: Loops and conditionals
COMSM1302 Overview of Computer Architecture

John Lapinskas, University of Bristol

A terror from the past

So far, we can compute simple expressions like (RAM[3] $+$ RAM[4]) & RAM[5], but we don't have a "real computer" yet. Hack assembly as we've covered it so far is more like a calculator attached to a clock.

We need loops and conditionals, but we don't have ifs or while loops.

A terror from the past

So far, we can compute simple expressions like (RAM[3] + RAM[4]) & RAM[5], but we don't have a "real computer" yet. Hack assembly as we've covered it so far is more like a calculator attached to a clock.

We need loops and conditionals, but we don't have ifs or while loops.

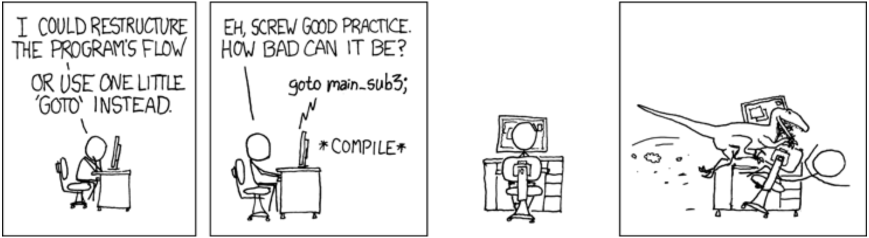
Let's instead discuss something... older. Darker. *Hungrier*.

A terror from the past

So far, we can compute simple expressions like (RAM[3] + RAM[4]) & RAM[5], but we don't have a "real computer" yet. Hack assembly as we've covered it so far is more like a calculator attached to a clock.

We need loops and conditionals, but we don't have ifs or while loops.

Let's instead discuss something... older. Darker. *Hungrier*.



Source: Randall Munroe, xkcd (here)

Gotos in C

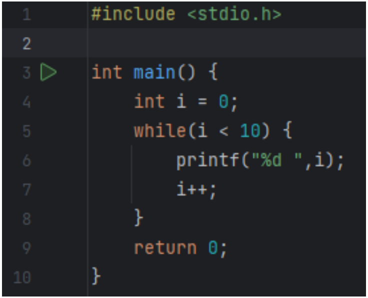
In C, a goto statement allows you to “jump” from **anywhere** in the code to a specific label.

```
1  #include <stdio.h>
2
3  int main() {
4      printf("Hello, World!\n\n");
5      goto skip;
6      printf("Gotos can skip lines of code.\n\n");
7  skip:
8      return 0;
9  }
```

In this code, on executing `goto skip` on line 5, the code skips the print statement on line 6 and resumes after `skip:` on line 8, returning 0.

Using gotos for loops and conditionals

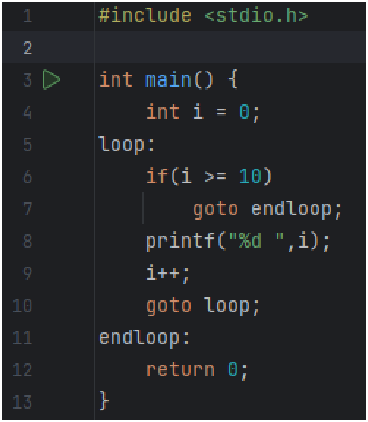
All flow control in C can be expressed as gotos and one-line if statements!



```
#include <stdio.h>

int main() {
    int i = 0;
    while(i < 10) {
        printf("%d ",i);
        i++;
    }
    return 0;
}
```

becomes

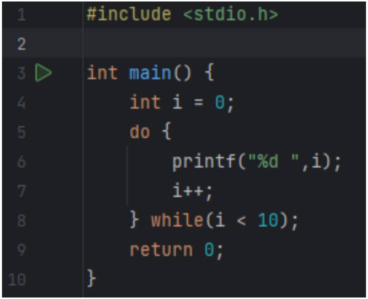


```
#include <stdio.h>

int main() {
    int i = 0;
loop:
    if(i >= 10)
        goto endloop;
    printf("%d ",i);
    i++;
    goto loop;
endloop:
    return 0;
}
```

Using gotos for loops and conditionals

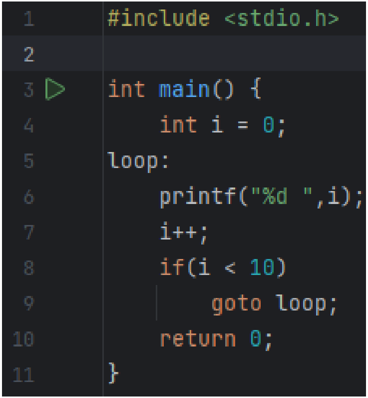
All flow control in C can be expressed as gotos and one-line if statements!



```c
#include <stdio.h>

int main() {
    int i = 0;
    do {
        printf("%d ",i);
        i++;
    } while(i < 10);
    return 0;
}
```

becomes

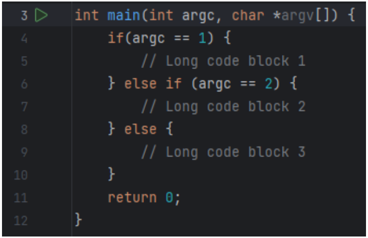


```c
#include <stdio.h>

int main() {
    int i = 0;
loop:
    printf("%d ",i);
    i++;
    if(i < 10)
        goto loop;
    return 0;
}
```

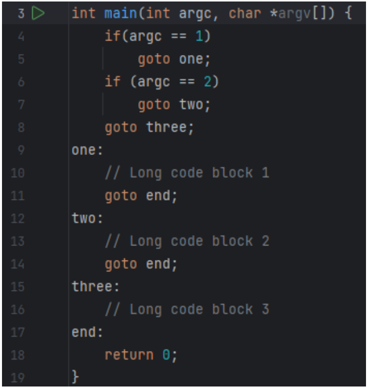
Using gotos for loops and conditionals

All flow control in C can be expressed as gotos and one-line if statements!



```c
int main(int argc, char *argv[]) {
    if(argc == 1) {
        // Long code block 1
    } else if (argc == 2) {
        // Long code block 2
    } else {
        // Long code block 3
    }
    return 0;
}
```

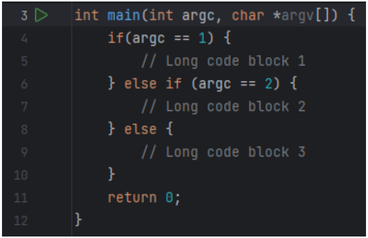
becomes



```c
int main(int argc, char *argv[]) {
    if(argc == 1)
        goto one;
    if (argc == 2)
        goto two;
    goto three;
one:
    // Long code block 1
    goto end;
two:
    // Long code block 2
    goto end;
three:
    // Long code block 3
end:
    return 0;
}
```

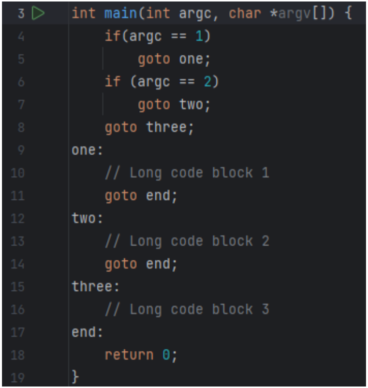
Using gotos for loops and conditionals

All flow control in C can be expressed as gotos and one-line if statements!



```c
int main(int argc, char *argv[]) {
    if(argc == 1) {
        // Long code block 1
    } else if (argc == 2) {
        // Long code block 2
    } else {
        // Long code block 3
    }
    return 0;
}
```

becomes



```c
int main(int argc, char *argv[]) {
    if(argc == 1)
        goto one;
    if (argc == 2)
        goto two;
    goto three;
one:
    // Long code block 1
    goto end;
two:
    // Long code block 2
    goto end;
three:
    // Long code block 3
end:
    return 0;
}
```

So why use whiles and elses instead of gotos?

Go to statement considered harmful

Gotos are completely unrestricted (within a function). You can use them to simulate loops this way, but you can goto one label from 20 different places in a 10,000-line function.

If you see a loop or an if statement in someone's C code, you know exactly what it will do to the control flow. But if you see a label, the code could jump to that label from *anywhere*.

Go to statement considered harmful

Gotos are completely unrestricted (within a function). You can use them to simulate loops this way, but you can goto one label from 20 different places in a 10,000-line function.

If you see a loop or an if statement in someone's C code, you know exactly what it will do to the control flow. But if you see a label, the code could jump to that label from *anywhere*.

The use of ifs and whiles and function calls to control program flow is known as **structured programming** and rose to prominence in the 1960s. Before that, none of them existed and all flow control was with gotos. Now, it's the foundation of all software engineering as a discipline.

Go to statement considered harmful

Gotos are completely unrestricted (within a function). You can use them to simulate loops this way, but you can goto one label from 20 different places in a 10,000-line function.

If you see a loop or an if statement in someone's C code, you know exactly what it will do to the control flow. But if you see a label, the code could jump to that label from *anywhere*.

The use of ifs and whiles and function calls to control program flow is known as **structured programming** and rose to prominence in the 1960s. Before that, none of them existed and all flow control was with gotos. Now, it's the foundation of all software engineering as a discipline.

There are a *very few* situations in which gotos are still somewhat reasonable to use in C, but it's best avoided unless you know what you're doing and can fight off a velociraptor or two.

Go to statement considered harmful

Gotos are completely unrestricted (within a function). You can use them to simulate loops this way, but you can goto one label from 20 different places in a 10,000-line function.

If you see a loop or an if statement in someone's C code, you know exactly what it will do to the control flow. But if you see a label, the code could jump to that label from *anywhere*.

The use of ifs and whiles and function calls to control program flow is known as **structured programming** and rose to prominence in the 1960s. Before that, none of them existed and all flow control was with gotos. Now, it's the foundation of all software engineering as a discipline.

There are a *very few* situations in which gotos are still somewhat reasonable to use in C, but it's best avoided unless you know what you're doing and can fight off a velociraptor or two.

(The slide title comes from Edsger W. Dijkstra's seminal 1968 article, which coined the term "structured programming" and set the movement going.)

The horrible truth

Gotos in Hack: Jumps

In assembly, gotos with simple if statements are usually the only form of flow control we have.

We call them **jumps** or **branches** to make it clear that each one comes from a single machine code instruction.

Any instruction in Hack assembly not starting with `@` can be followed by a semicolon and one of seven jump instructions. This reads as "if [result of instruction] satisfies [condition], goto the ROM address contained in A".

For example, `M=A+D;JGT` stores $A + D$ in M, then jumps to the address contained in A if $A + D > 0$.

You can also omit the left-hand side of the instruction to jump without assigning any values, e.g. `A+D;JGT` is valid assembly that jumps to the address contained in A if $A + D > 0$.

List of jump conditions in Hack

Condition	Mnemonic	Jumps if
`JMP`	**J**u**MP**	Always
`JGT`	**J**ump if **G**reater **T**han	[result] > 0
`JEQ`	**J**ump if **EQ**ual	[result] $= 0$
`JLT`	**J**ump if **L**ess **T**han	[result] < 0
`JGE`	**J**ump if **G**reater than or **E**qual	[result] ≥ 0
`JNE`	**J**ump if **N**ot **E**qual	[result] $\neq 0$
`JLE`	**J**ump if **L**ess than or **E**qual	[result] ≤ 0

Don't try to memorise this table — just refer back as needed!

Remember, all jumps are to the address stored in A.

Warning: The PC is updated at the same time as A, at the start of the next clock cycle! An instruction like `A=A+D;JMP` has undefined behaviour.

Help from the assembler: Labels

Each (non-comment) line of assembly is one line of machine code. So to unconditionally jump to line 100 of (non-comment/whitespace) assembly, stored in ROM[99], we would use `@99` followed by e.g. `0;JMP`.

(We normally write `0;JMP` for a jump with no calculation attached, but this is a convention, not a requirement.)

Help from the assembler: Labels

Each (non-comment) line of assembly is one line of machine code. So to unconditionally jump to line 100 of (non-comment/whitespace) assembly, stored in ROM[99], we would use `@99` followed by e.g. `0;JMP`.

(We normally write `0;JMP` for a jump with no calculation attached, but this is a convention, not a requirement.)

Problem: This is *awful* to maintain. What if we need to delete one of lines 1–99? Or add another line? We need to update every jump in the file!

Help from the assembler: Labels

Each (non-comment) line of assembly is one line of machine code. So to unconditionally jump to line 100 of (non-comment/whitespace) assembly, stored in ROM[99], we would use `@99` followed by e.g. `0;JMP`.

(We normally write `0;JMP` for a jump with no calculation attached, but this is a convention, not a requirement.)

Problem: This is *awful* to maintain. What if we need to delete one of lines 1–99? Or add another line? We need to update every jump in the file!

Solution: Like with C, the assembler provides **labels**.

A line of the form `(Label)` doesn't correspond to any machine code. Instead, if the next line would appear at e.g. ROM position 100, then it tells the assembler to replace all instances of `@Label` with `@100`.

Example: Computing a sum

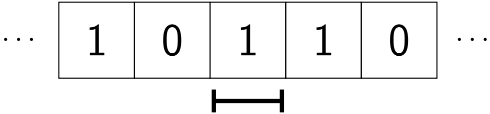
Sum.asm outputs to RAM[1] a sum of all the integers from 0 to RAM[0]:

RAM[0] (input)	RAM[1] (output)
0	0
1	$0+1=1$
2	$0+1+2=3$
3	$0+1+2+3=6$
$\vdots$	$\vdots$

[See video for live coding and explanation.]

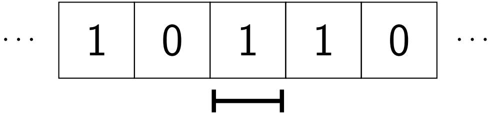
What is a “real” computer anyway?

A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



What is a “real” computer anyway?

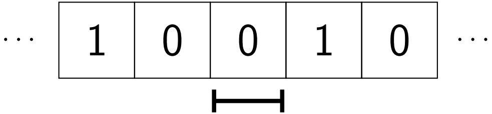
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 3, head reads 1 $\longrightarrow$ Write 0, move right, enter state 3.

What is a "real" computer anyway?

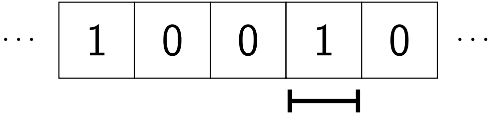
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 3, head reads 1 ⟶ Write 0, move right, enter state 3.

What is a "real" computer anyway?

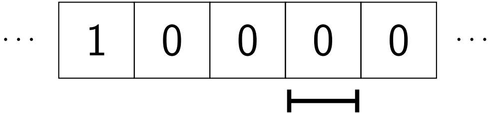
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 3, head reads 1 $\longrightarrow$ Write 0, move right, enter state 3.

What is a “real” computer anyway?

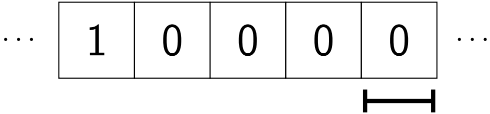
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 3, head reads 1 ⟶ Write 0, move right, enter state 3.

What is a “real” computer anyway?

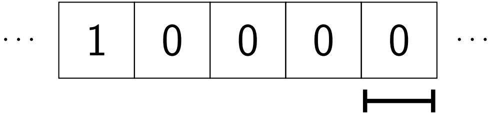
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 3, head reads 1 ⟶ Write 0, move right, enter state 3.

What is a “real” computer anyway?

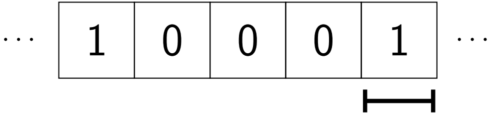
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 3, head reads 0 $\longrightarrow$ Write 1, move left, enter state 7.

What is a “real” computer anyway?

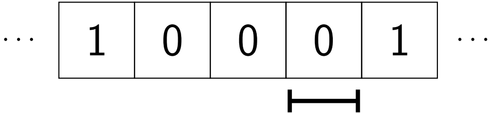
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 3, head reads 0 ⟶ Write 1, move left, enter state 7.

What is a “real” computer anyway?

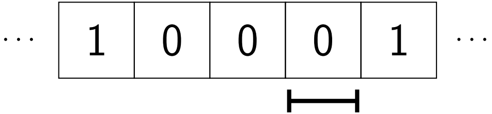
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 3, head reads 0 ⟶ Write 1, move left, enter state 7.

What is a "real" computer anyway?

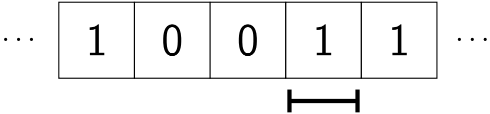
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 7, head reads 0 $\longrightarrow$ Write 1, move left, enter state 1.

What is a "real" computer anyway?

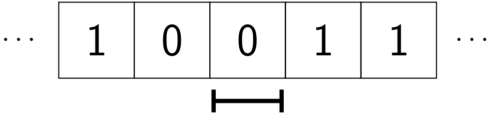
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 7, head reads 0 ⟶ Write 1, move left, enter state 1.

What is a “real” computer anyway?

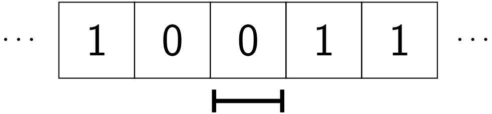
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 7, head reads 0 ⟶ Write 1, move left, enter state 1.

What is a “real” computer anyway?

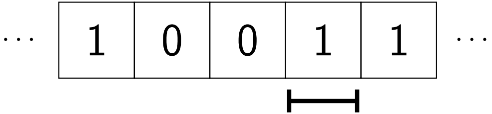
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 1, head reads 0 ⟶ Write 0, move right, enter state 10.

What is a “real” computer anyway?

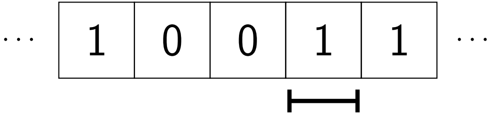
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 1, head reads 0 $\longrightarrow$ Write 0, move right, enter state 10.

What is a “real” computer anyway?

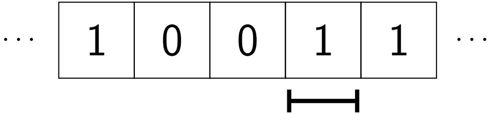
A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 10, head reads 1 ⟶ Halt.

What is a “real” computer anyway?

A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



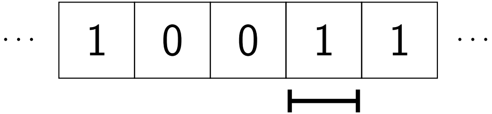
Machine in state 10, head reads 1 ⟶ Halt.

Note: The above definition is non-examinable! So don’t worry if you didn’t follow it exactly.

(The rest of the slide will be examinable, though.)

What is a “real” computer anyway?

A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



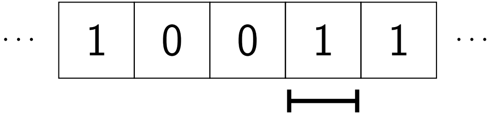
Machine in state 10, head reads 1 ⟶ Halt.

Why care? Because the **Church-Turing thesis** says that if a computing problem is solvable, then a well-chosen Turing machine can solve it. Write the input on the tape, run it until it halts, read the tape for the output.

So if we can simulate any Turing machine, we have a “real” computer! We say a computer model which can do this is **Turing-complete**.

What is a “real” computer anyway?

A **Turing machine** is a two-sided infinite string of tape divided into cells containing binary values, plus a tape head and a collection of (finitely many) possible states. At each time step, based on the current cell and its internal state, the tape head writes a 1 or 0 and moves left or right along the tape, and then the Turing machine changes state or halts. E.g.:



Machine in state 10, head reads 1 ⟶ Halt.

Why care? Because the **Church-Turing thesis** says that if a computing problem is solvable, then a well-chosen Turing machine can solve it. Write the input on the tape, run it until it halts, read the tape for the output.

So if we can simulate any Turing machine, we have a “real” computer! We say a computer model which can do this is **Turing-complete**.

Note: Not all computing problems are solvable! It is impossible to tell whether an arbitrary Turing machine ever halts (the **Halting Problem**).