THE GEORGE WASHINGTON UNIVERSITY

WASHINGTON, DC

ECE 1120
C Programming for ECE
Spring 2016

Lecturer: Ahsen J. Uppal

Week 1

Teaching Staff

Instructor: Ahsen J. Uppal (auppal@gwu.edu)

Teaching Assistant:
Pradeep Kumar (pradeepk@email.gwu.edu)

This Course

This course provides a hands-on introduction to using C programming to solve engineering problems.

You will learn to analyze, decompose, and translate real-world problems into computational problems.

There will be **heavy** focus on writing actual, working code in a clear and well-structured way.

This Course

This course will be

HARD.

Do not wait to the last minute to start your assignments or ask for help!

Academic Integrity

Students are required to honor the GWU Code of Academic Integrity when completing all assignments, projects, and examinations.

Review the code of Academic Integrity:
http://studentconduct.gwu.edu/

DON'T CHEAT!

>:0

You may not copy code from any other student!

Assignments and Grading

```
Participation (incl lab attendance): 10%
Homework/Labs                      : 25%
Projects (One or Two Large)        : 20%
Midterm                            : 20%
Final Exam                         : 25%
```

There is a 20% **per day** penalty for late submissions.

Grading Disputes

- For materials graded by your TA:
  - Write your dispute in an email to your TA.
  - The TA will decide whether a correction is warrented.
  - If there is still a disagreement, email the details: original report, the TA's response, and the dispute to me.

Online Resources

- We will be using a cloud-based development enviornment called Cloud9 https://c9.io.
  - You will register for a free Cloud9 account (and set permissions to private).
- We will be using a source code distribution tool called git and an online git hosting service called GitHub https://github.com.
  - You will also register for a free GitHub account.

What I expect you know!

You can do basic math with exponents, and logarithms

$x^a * x^b = ?$

$\log(a*b) = ?$

You know what a function is:

f(x) = 3*x^2 + 2

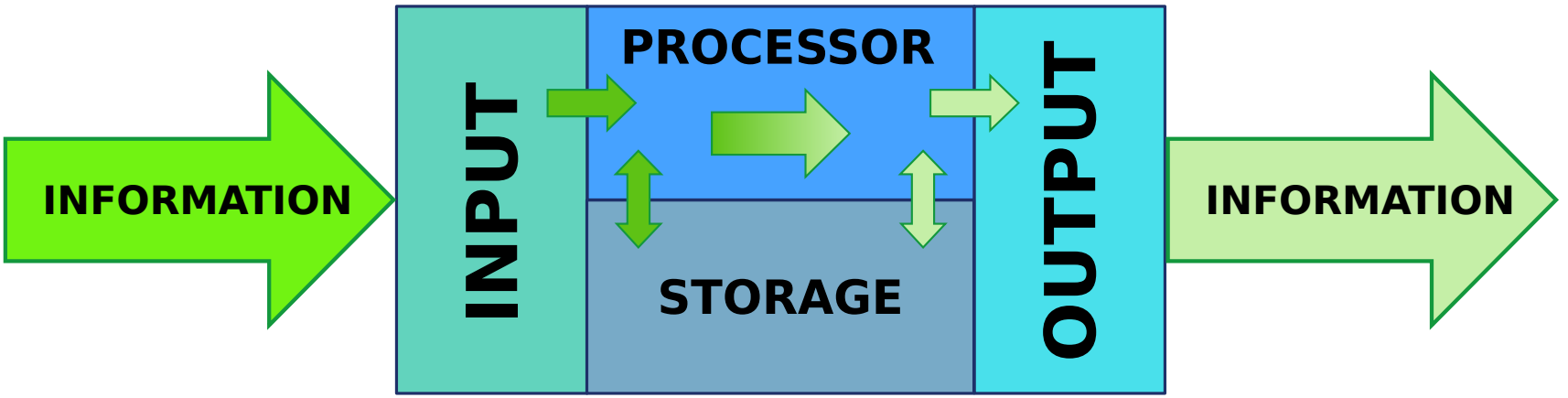
vol(w, h, d) = w * h * d

g(n) = n * g(n - 1) if n > 1 else 1

You know what the domain and range of a function are.

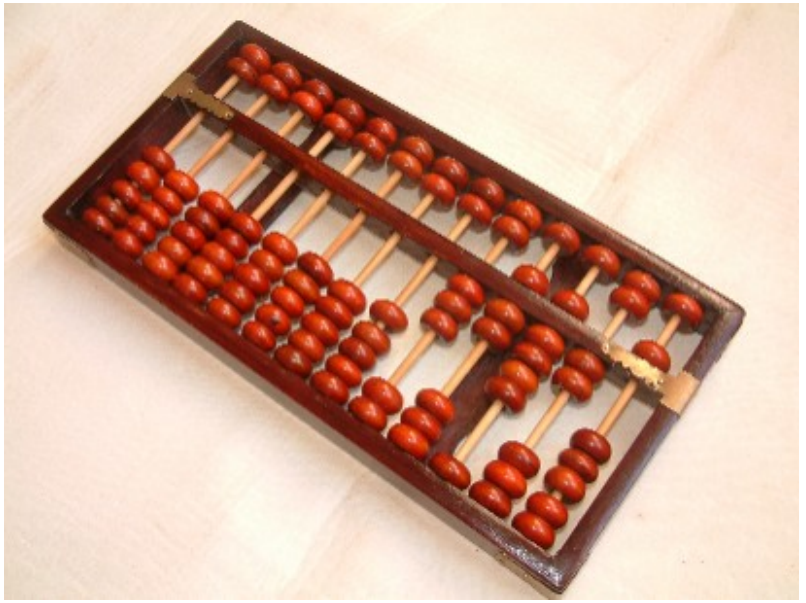
Let's start with what a computer is...

Abstract Information Processing Flow:



Source:
https://en.wikipedia.org/wiki/History_of_computing_hardware
#/media/File:Information_processing_system_(english).svg

Wikipedia has an excellent article on the history of computing hardware.



Abacus first introduced around 2700-2300 BC (Using base 60!)

This is a later Chinese abacus.

What is the input?
What is the processor?
What is the storage?
What is the output?

Source:
https://upload.wikimedia.org/wikipedia/commons/e/ea/Boulier1.JPG

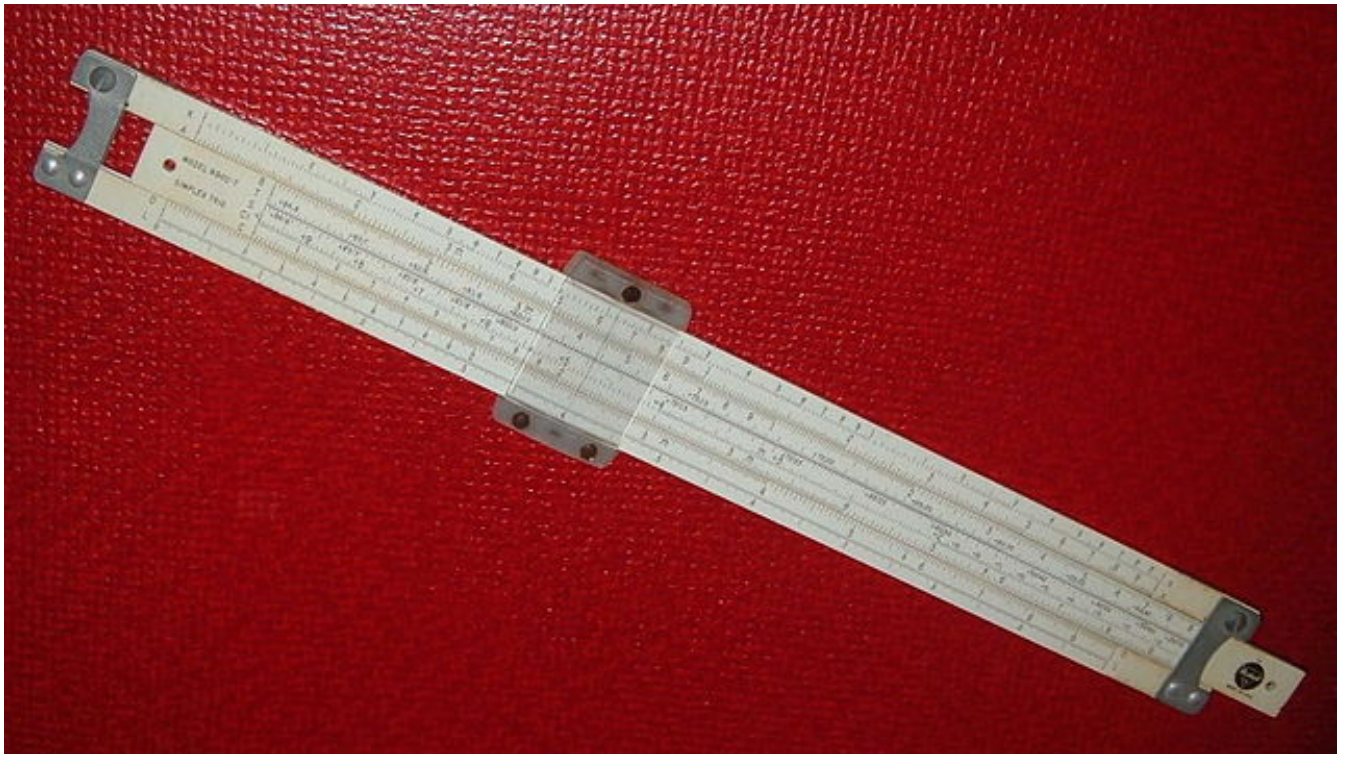
An abacus is great for addition and subtraction, not so good for multiplication and division.

John Napier invented logarithms and a device called “Napier's Bones” to do multiplication and division by using addition and subtraction.

i.e. x * y = log(x) + log(y)

Sources:
https://upload.wikimedia.org/wikipedia/commons/e/e3/John_Napier.jpg
https://upload.wikimedia.org/wikipedia/commons/f/fb/An_18th_century_set_of_Napier's_Bones.JPG

And “Napier's Bones” evolved into slide-rules:



And later, there were even brilliant compact mechanical devices like the Curta.

See: https://www.youtube.com/watch?v=5pD8OztX3Nw

Image Source:
https://en.wikipedia.org/wiki/Slide_rule#/media/File:Sliderule.PickettN902T.agr.jpg

These early examples of computing hardware was just simple mechanical tools to help ***human*** computers.

- The input comes from human mind and fingers.
- The processor is a mechanical device.
- The storage is **one** single value. Storing intermediate results needs human memory or writing.
- The output is visual or tactile.

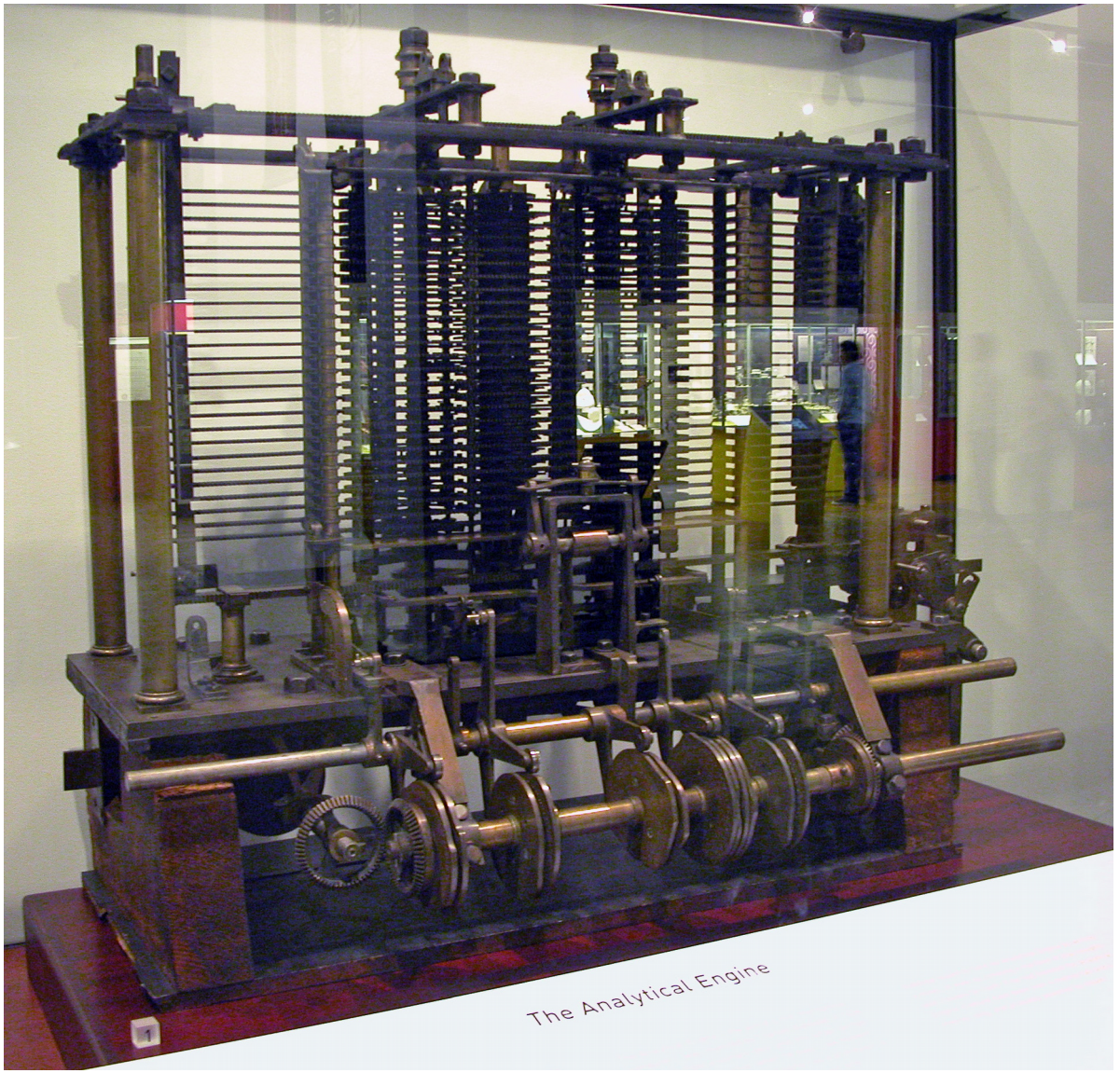
More importantly, all **control instructions** are done by a person.

The key advance was the conception of **programmable** devices.

The key insight is that the machine itself can store a series of instructions to execute as well as data upon which to execute those instructions.

Charles Babbage first conceived the (non-programmable) “Difference Engine” to compute mathematical and astronomical tables, and then later a fully-programmable “Analytical Engine” (1837), which would have used punched cards for input and output.

A model of the Analytical Engine:



Source: https://en.wikipedia.org/wiki/Analytical_Engine#/media/File:AnalyticalMachine_Babbage_London.jpg

Babbage's Analytical Engine was far ahead if its time, it wasn't feasible to build with the machinery of the era. It wasn't really until around the 1940s that the first actual programmable computers were created.

Many of these were still primitive compared to the Analytical Engine because of a critical feature called a **conditional branch**. This lets an instruction execute only if some stored value has a certain value.

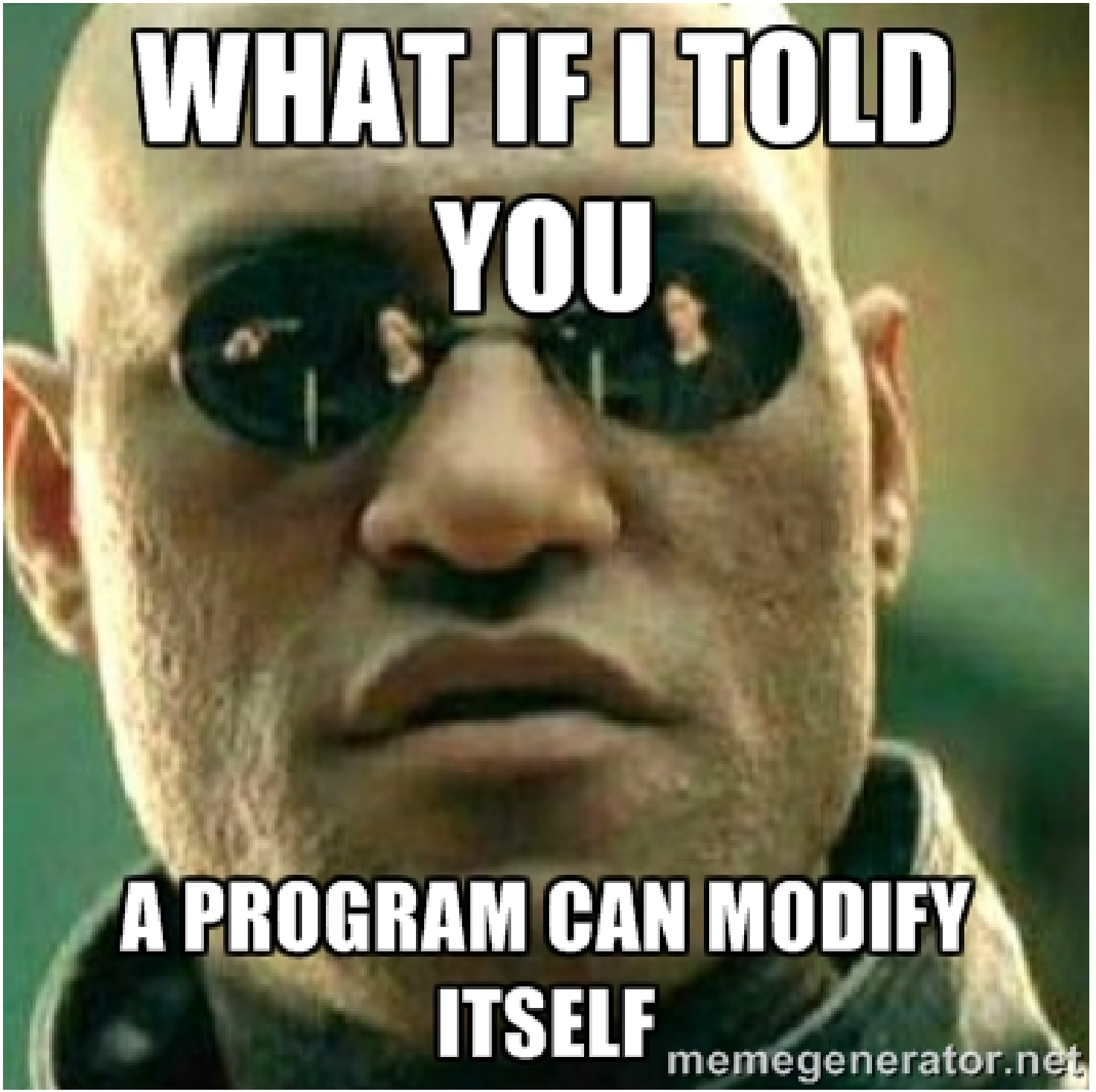
The German Zuse Z4 and American ENIAC were the first to have conditional branches.

Conditional Branch Example.
Add the absolute value of one number to another:

```
x = x * 12
y = y * 2 + 3

If x > 0:
    y = y + x
Else:
    y = y – x
```

One other critical feature (but easy to overlook feature) is the ability to modify arbitrary memory. It's easier to understand this when we're storing multiple results….


WHAT IF I TOLD YOU
A PROGRAM CAN MODIFY ITSELF
memegenerator.net

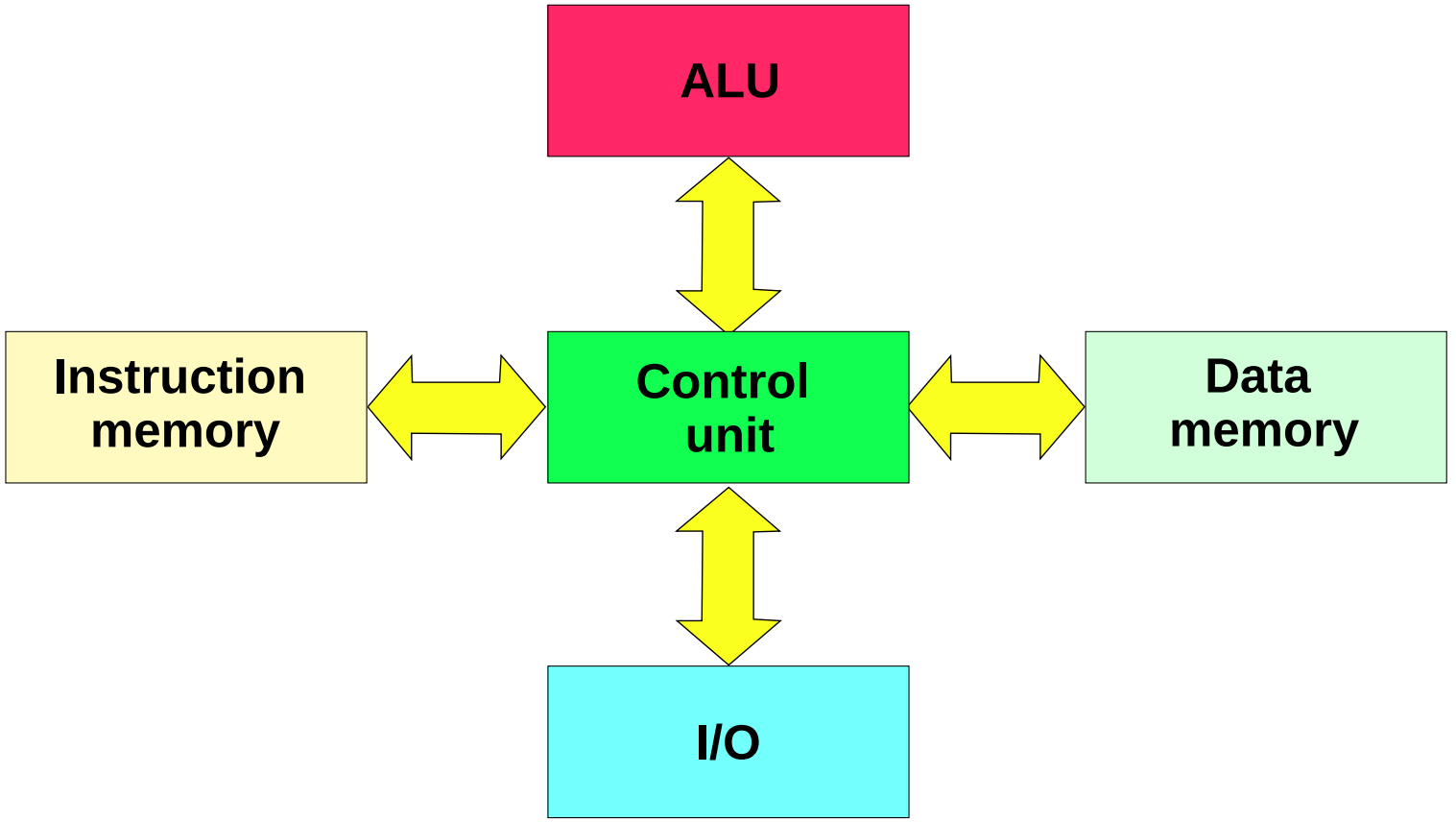
Conditional branches and the ability to modify arbitrary memory allow a computer to have a certain elegant property known as “Turing Completeness”.

Fun fact: The most important discoveries about what any computer can and cannot do were made in the 1930s by Alan Turing and others when there were NO actual computers. These theorems rely on an hypothetical machine called a “Turing Machine”.

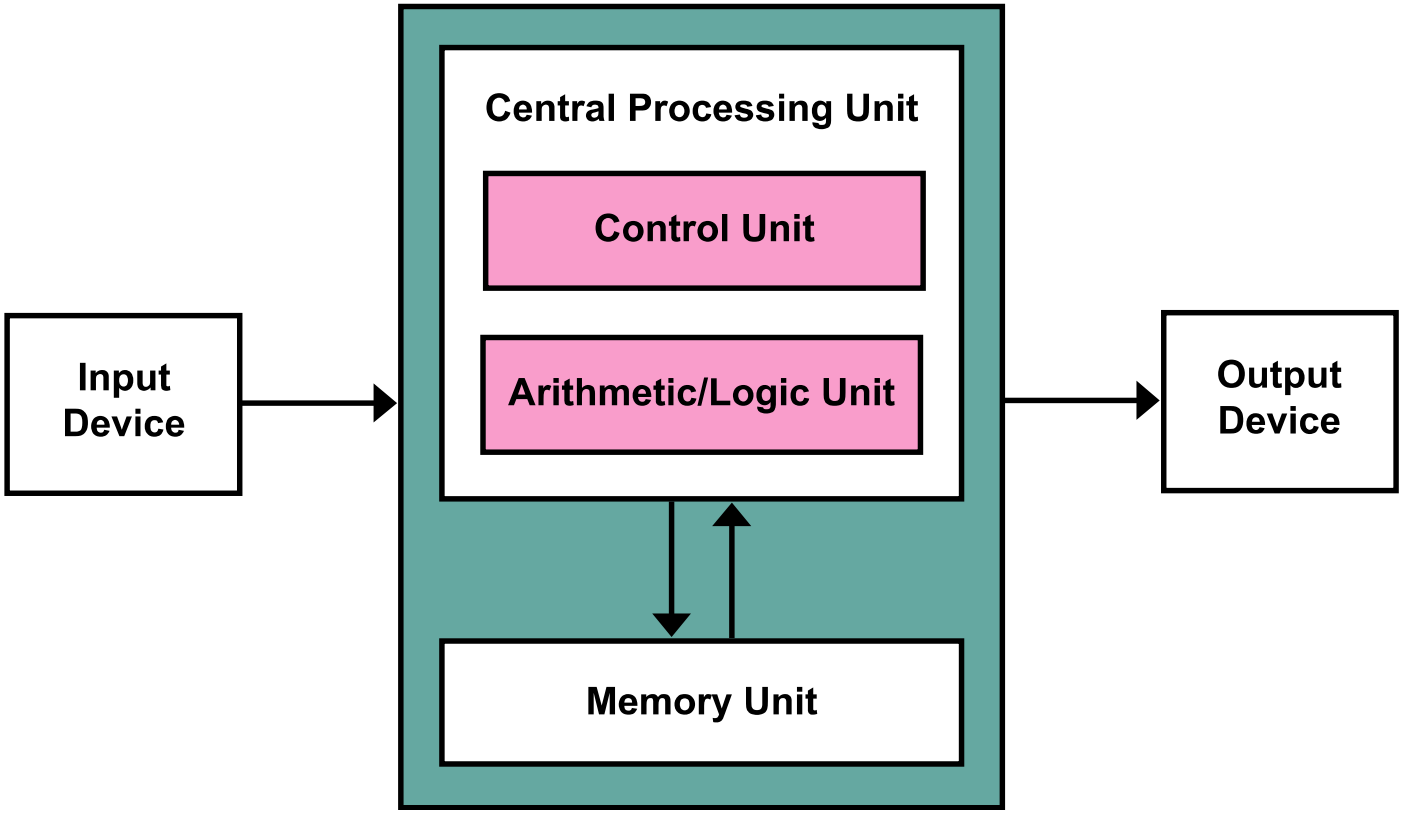
To summarize, the key features that make modern computing machines possible:

- Storage for multiple values
- Stored program
- Conditional branches
- Arbitrary ability to write memory (self-modifying code possible)

A machine design where instructions and data are in separate memories are known as a **Harvard architecture**.



A computer design where instructions and data reside in the same memory is known as a **von Neumann architecture**.



An ALU is an arithmetic logic unit, basically a glorified 4-function calculator. Takes two values in and calculates a result.

A control unit looks at a location in memory and decodes what instruction is there.

An input/output (IO) device would might something like a keyboard, mouse, monitor, printer, etc.

A memory unit stores values. In modern computers, this is a combination of DRAM (dynamic random access memory) and disk (a hard drive or SSD).

How it works

How does a computer execute a program? (example programs: a computer game, a word processor, etc)

The instructions that comprise the program are copied from a storage device (disk) into the memory

After the instructions are loaded, the CPU starts executing the program.

For each instruction, the instruction is retrieved from memory, decoded to figure out what it represents, and the appropriate action carried out. (the *fetch- execute cycle)*

Then the next instruction is fetched, decoded and executed.

Machine level programming

- Example: suppose we want the computer to add two numbers, and if the preliminary result is less than 10, then add 10 to the result

- The instructions that the CPU carries out might be :

```
[INSTR1]  Load into ALU the number from mem location 15
[INSTR2]  Load into ALU  the number from mem location 7
[INSTR3]  Add the two numbers in the ALU
[INSTR4]  If result is bigger than 10 jump to [INSTR6]
[INSTR5]  Add 10 to the number in the ALU
[INSTR6]  Store the result from ALU into mem location 3
```

- The **processors instruction set**: all basic operations that can be carried out by a certain type of processor

Machine level programming

- the **instructions and operands** are represented in *binary* notation (sequences of 0s and 1s).
  - Why binary ? Because computer hardware relies on electric/electronic circuits that have/can switch between 2 states
  - **bit** (binary digit)
  - Byte: 8 bits
- The program carried out by the CPU, on a hypothetical processor type, could be:

  1010 1111

  1011 0111

  0111

  …
- This way had to be programmed the first computers !
- The job of the first programmers was to code directly in machine language and to enter their programs using switches

Higher level languages

- Assembly language
  - First step from machine language
  - Uses ***symbolic names*** for operations

  – Example: a hypothetical assembly language program sequence:

```
1010 1111
1011 0111
0111
0011 1010
0010 1100
0110 1010
...
...
```

```
LD1  15
LD2   7
ADD
CMP  10
JGE  12
ADD  10
```

Higher level languages

- Assembly language (cont)
  - Translation of assembly language into machine language: in the beginning done manually, later done by a special computer program – the *assembler*
  - Disadvantages: Low-level language:
    - programmer must learn the instruction set of the particular processor
    - Program must be rewritten in order to run on a different processor type – program is not *portable*

Higher level languages

- High level languages
  - Using ***more abstract instructions***
  - **Portable** programs result

  - Example: a hypothetical program sequence:

```
DEFVAR a,b,c;
BEGIN
        READ a
        READ b
        READ c
        c := a+b
        IF (c <10) THEN c:=c+10
        PRINT c
END                              …
```

- High level languages
  - Writing portable programs, using more abstract instructions
  - A high level instruction (*statement*) is translated into many machine instructions
  - Translation of high level language into machine instructions: done by special computer programs – *compilers* or *interpreters*

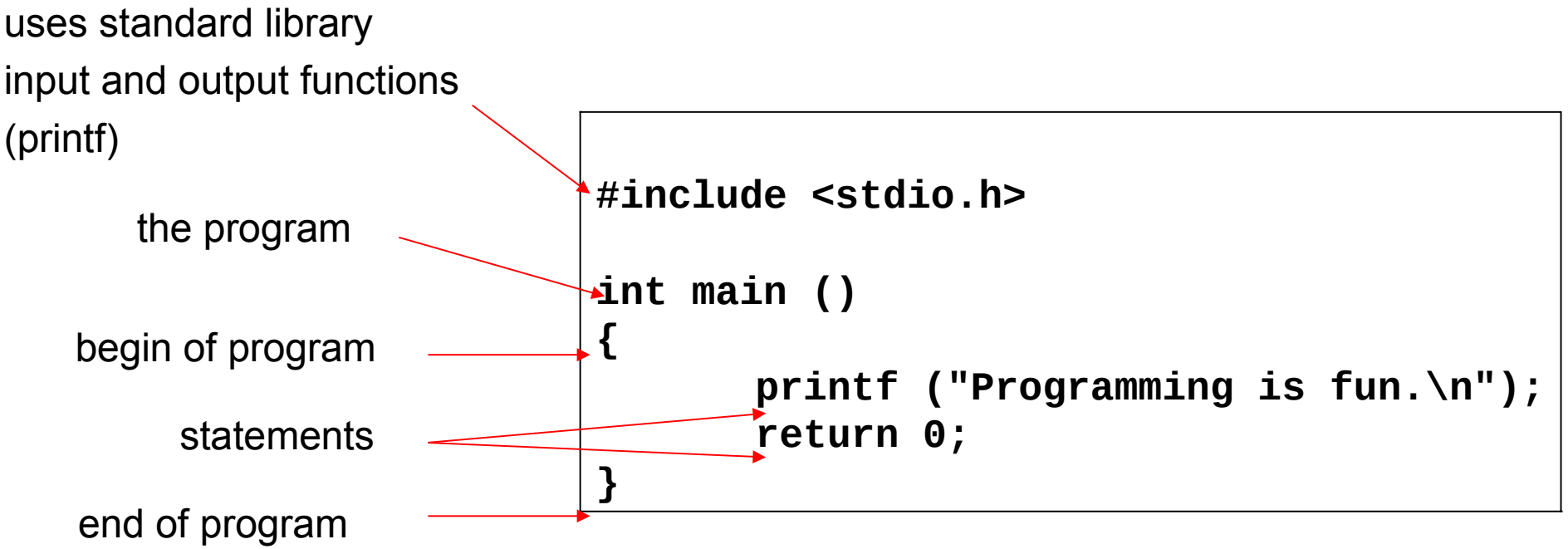
Operating Systems

- Operating system: a program that controls the entire operation of a computer system:
  - Handles all input and output (I/O) operations that are performed on a computer
  - manages the computer system’s resources
  - handles the execution of programs (including multitasking or multiuser facilities)
- Most famous OS families:
  - Windows
  - Unix

The C Programming Language

- Developed by Dennis Ritchie at AT&T Bell Laboratories in the early 1970s
- Growth of C tightly coupled with growth of Unix: Unix was written mostly in C
- Success of PCs: need of porting C on MS-DOS
- Many providers of C compilers for many different platforms => need for standardization of the C language
- 1990: ANSI C (American National Standards Institute)
- International Standard Organization: ISO/IEC 9899:1990
- 1999: standard updated: C99, or ISO/IEC 9899:1999

The first C program



main: a special name that indicates where the program must begin execution. It is a special *function*.

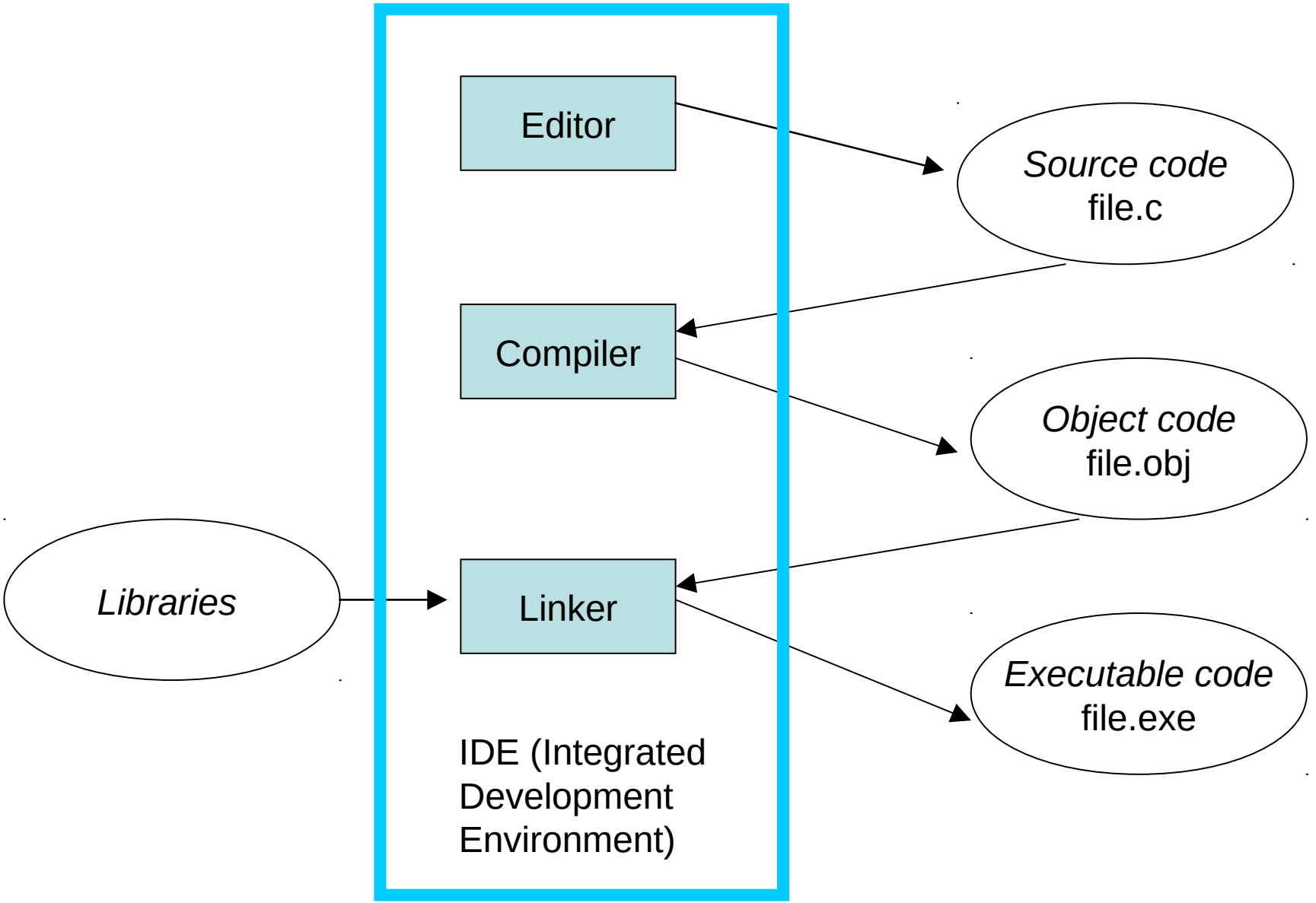
first statement: calls a routine named printf, with argument the string of characters “Programming is fun \n”

last statement: finishes execution of main and returns to the system a status value of 0 (conventional value for OK)

The format in C

- Statements are terminated with semicolons
- Indentation is nice to be used for increased readability.
- Free format: white spaces and indentation is ignored by compiler
- **C is case sensitive** – pay attention to lower and upper case letters when typing !
  - All C keywords and standard functions are lower case
  - Typing INT, Int, etc instead of int is a compiler error
- Strings are placed in double quotes
- New line is represented by \n (Escape sequence)

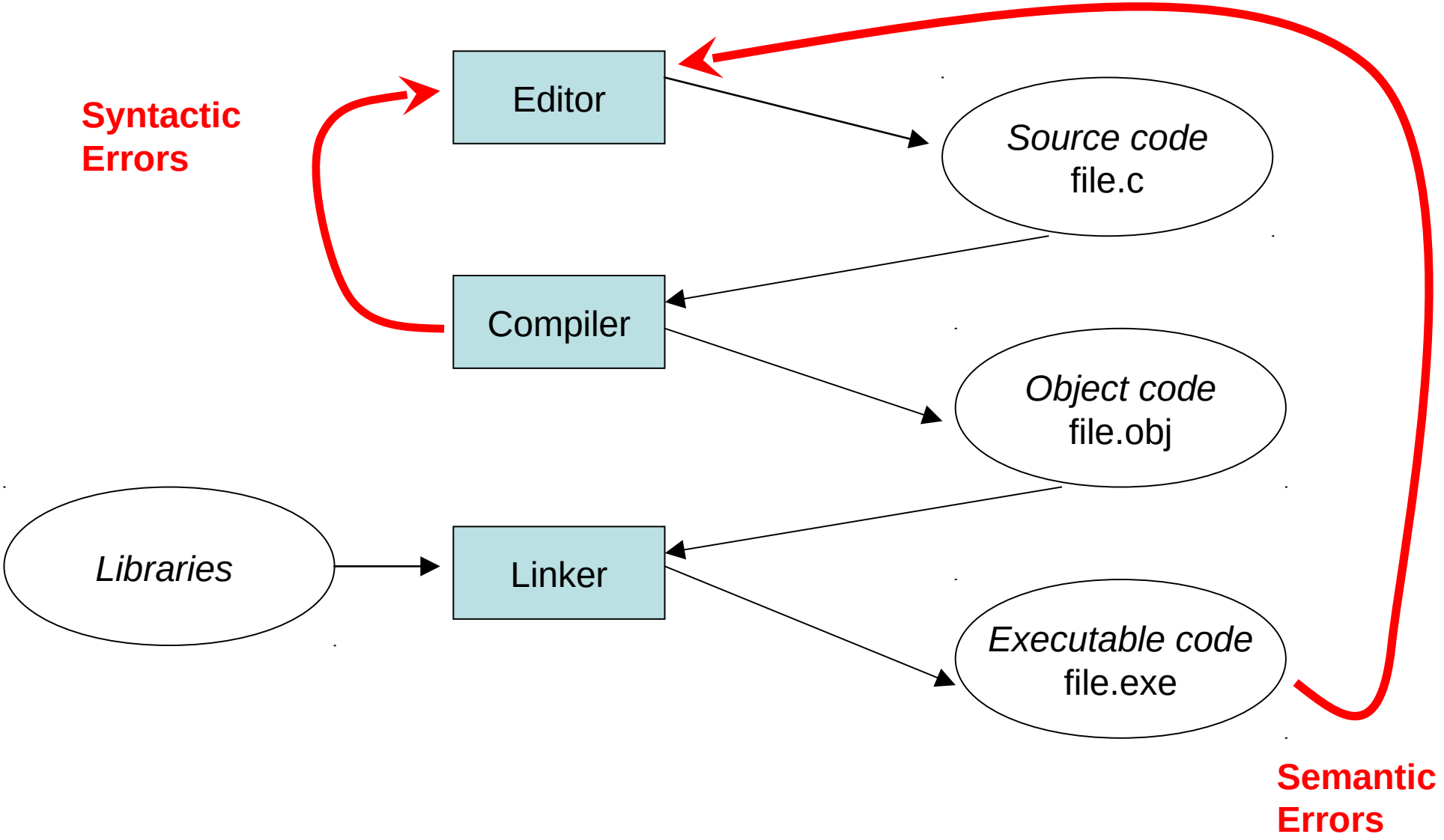
Compiling and running C programs



C Compilers and IDE's

- One can:
  - use a text editor to edit source code, and then use independent command-line compilers and linkers
  - use an IDE (like Cloud9): everything together + facilities to debug, develop and organize large projects
- There are several C compilers and IDE's that support various C compilers
- Lab: Dev-C++ IDE for C and C++, Free Software (under the GNU General Public License)
  - Works with gcc (GNU C Compiler)
    - supports the C99 standard
    - available on Windows and Unix
  - The GNU Project (http://www.gnu.org/): launched in 1984 in order to develop a complete Unix-like operating system which is free software - the GNU system.

Debugging program errors



Syntax and Semantics

- Syntax errors: violation of programming language rules (grammar)
  - "*Me speak English good*."
  - Use valid C symbols in wrong places
  - Detected by the compiler
- Semantics errors: errors in meaning:
  - "*This sentence is excellent Italian*."
  - Programs are syntactically correct but don’t produce the expected output
  - User observes output of running program

Second program

```
#include <stdio.h>
int main (void)
{
    printf ("Programming is fun.\n");
    printf ("And programming in C is even more fun.\n");
    return 0;
}
```

Displaying multiple lines of text

```
#include <stdio.h>
int main (void)
{
   printf ("Testing...\n..1\n...2\n....3\n");
   return 0;
}
```

It is not necessary to make a separate call to printf for each line of output !

Output:

```
Testing...
..1
...2
....3
```

Variables

- Programs can use symbolic names for storing computation data and results
- Variable: a symbolic name for a memory location
  - programmer doesn’t has to worry about specifying (or even knowing) the value of the location’s address
- In C, variables have to be ***declared*** before they are used

Using and Displaying Variables

```
#include <stdio.h>
int main (void)
{
        int sum;
        sum = 50 + 25;
        printf ("The sum of 50 and 25 is %i\n", sum);
        return 0;
}
```

Variable sum **declared** of type int

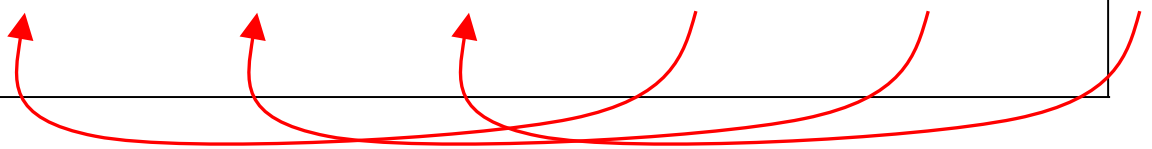
Variable sum **assigned** expression 50+25

Value of variable sum is **printed** in place of %i

The printf routine call has now 2 arguments: first argument a string containing also a format specifier (%i), that holds place for an integer value to be inserted here

Displaying multiple values

```
#include <stdio.h>
int main (void)
{
  int value1, value2, sum;
  value1 = 50;
  value2 = 25;
  sum = value1 + value2;
  printf ("The sum of %i and %i is %i\n",value1, value2, sum);
  return 0;
}
```



The format string must contain as many placeholders as expressions to be printed

Using comments in a program

- Comment statements are used in a program to document it and to enhance its readability.
- Useful for human readers of the program – compiler ignores comments
- Ways to insert comments in C:
  - When comments span several lines: start marked with /*, end marked with */
  - Comments at the end of a line: start marked with //

Using comments in a program

```
/* This program adds two integer values
and displays the results */

#include <stdio.h>
int main (void)
{
   // Declare variables
   int value1, value2, sum;
   // Assign values and calculate their sum
   value1 = 50;
   value2 = 25;
   sum = value1 + value2;
   // Display the result
   printf ("The sum of %i and %i is %i\n",
                                    value1, value2, sum);
   return 0;
}
```