C++

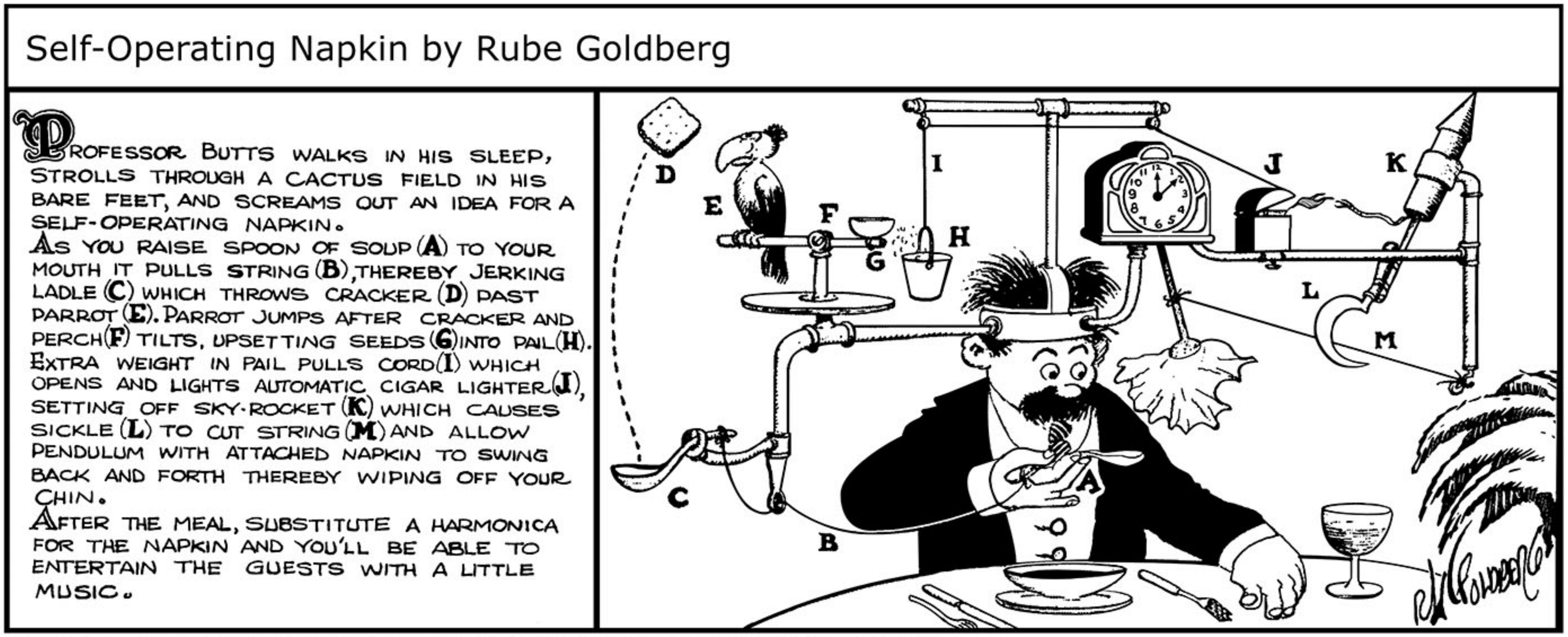
CS 003A Fundamentals of Computer Science II

SEKHAR RAVINUTALA

CONTROLLING COMPLEXITY

"Make things as simple as possible, but no simpler."

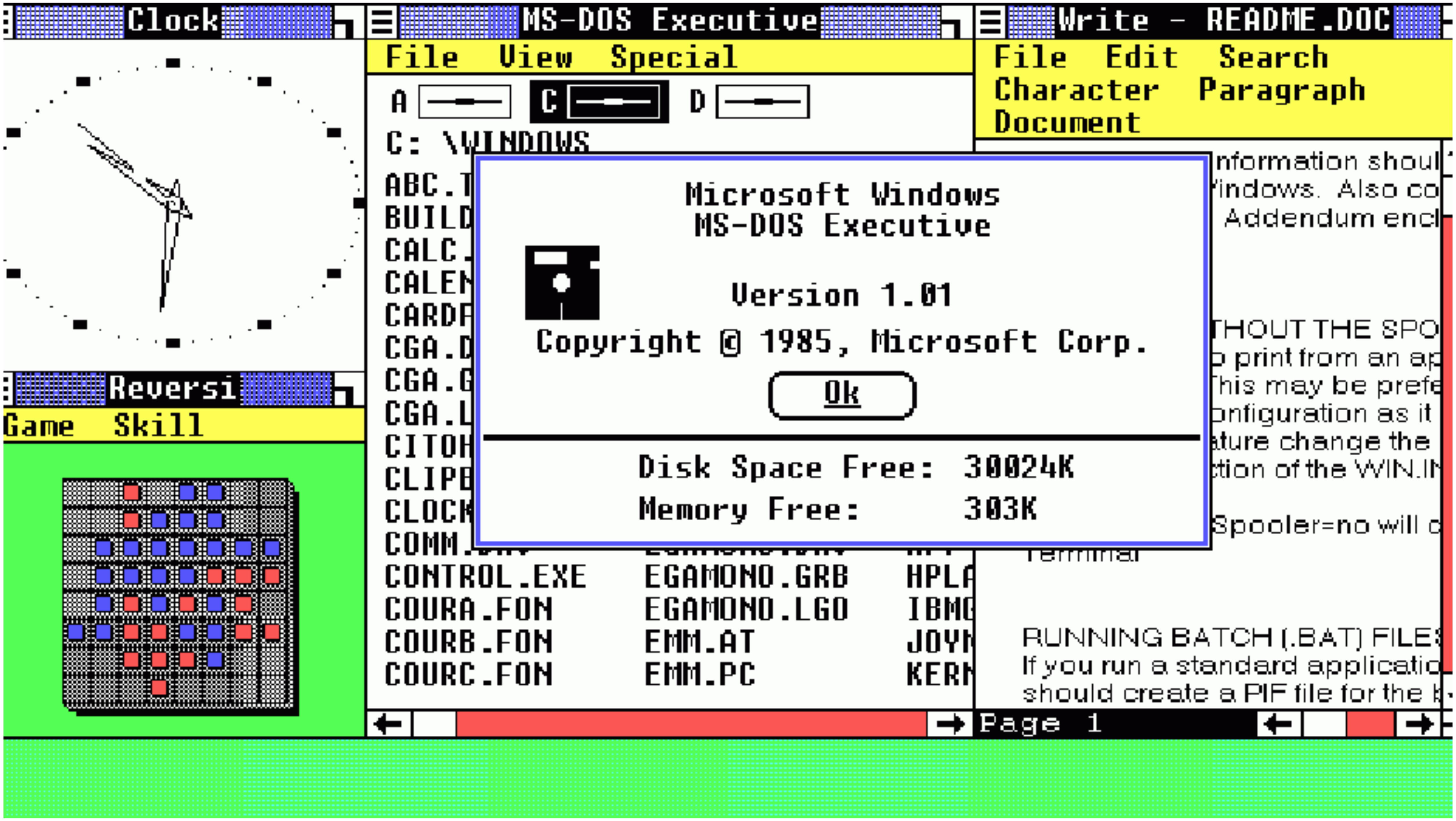
-Albert Einstein



Self-Operating Napkin by Rube Goldberg

PROFESSOR BUTTS WALKS IN HIS SLEEP, STROLLS THROUGH A CACTUS FIELD IN HIS BARE FEET, AND SCREAMS OUT AN IDEA FOR A SELF-OPERATING NAPKIN.
AS YOU RAISE SPOON OF SOUP (A) TO YOUR MOUTH IT PULLS STRING (B), THEREBY JERKING LADLE (C) WHICH THROWS CRACKER (D) PAST PARROT (E). PARROT JUMPS AFTER CRACKER AND PERCH (F) TILTS, UPSETTING SEEDS (G) INTO PAIL (H). EXTRA WEIGHT IN PAIL PULLS CORD (I) WHICH OPENS AND LIGHTS AUTOMATIC CIGAR LIGHTER (J), SETTING OFF SKY-ROCKET (K) WHICH CAUSES SICKLE (L) TO CUT STRING (M) AND ALLOW PENDULUM WITH ATTACHED NAPKIN TO SWING BACK AND FORTH THEREBY WIPING OFF YOUR CHIN.
AFTER THE MEAL, SUBSTITUTE A HARMONICA FOR THE NAPKIN AND YOU'LL BE ABLE TO ENTERTAIN THE GUESTS WITH A LITTLE MUSIC.

MICROSOFT WINDOWS – THE BEGINNINGS



```
C: \WINDOWS
ABC.T
BUILD
CALC.
CALEN
CARDF
CGA.D
CGA.G
CGA.L
CITOH
CLIPB
CLOCK
COMM.
CONTROL.EXE     EGAMONO.GRB     HPLA
COURA.FON       EGAMONO.LGO     IBMG
COURB.FON       EMM.AT          JOYM
COURC.FON       EMM.PC          KERN
```

Microsoft Windows
MS-DOS Executive

Version 1.01

Copyright © 1985, Microsoft Corp.

Disk Space Free: 30024K
Memory Free: 303K

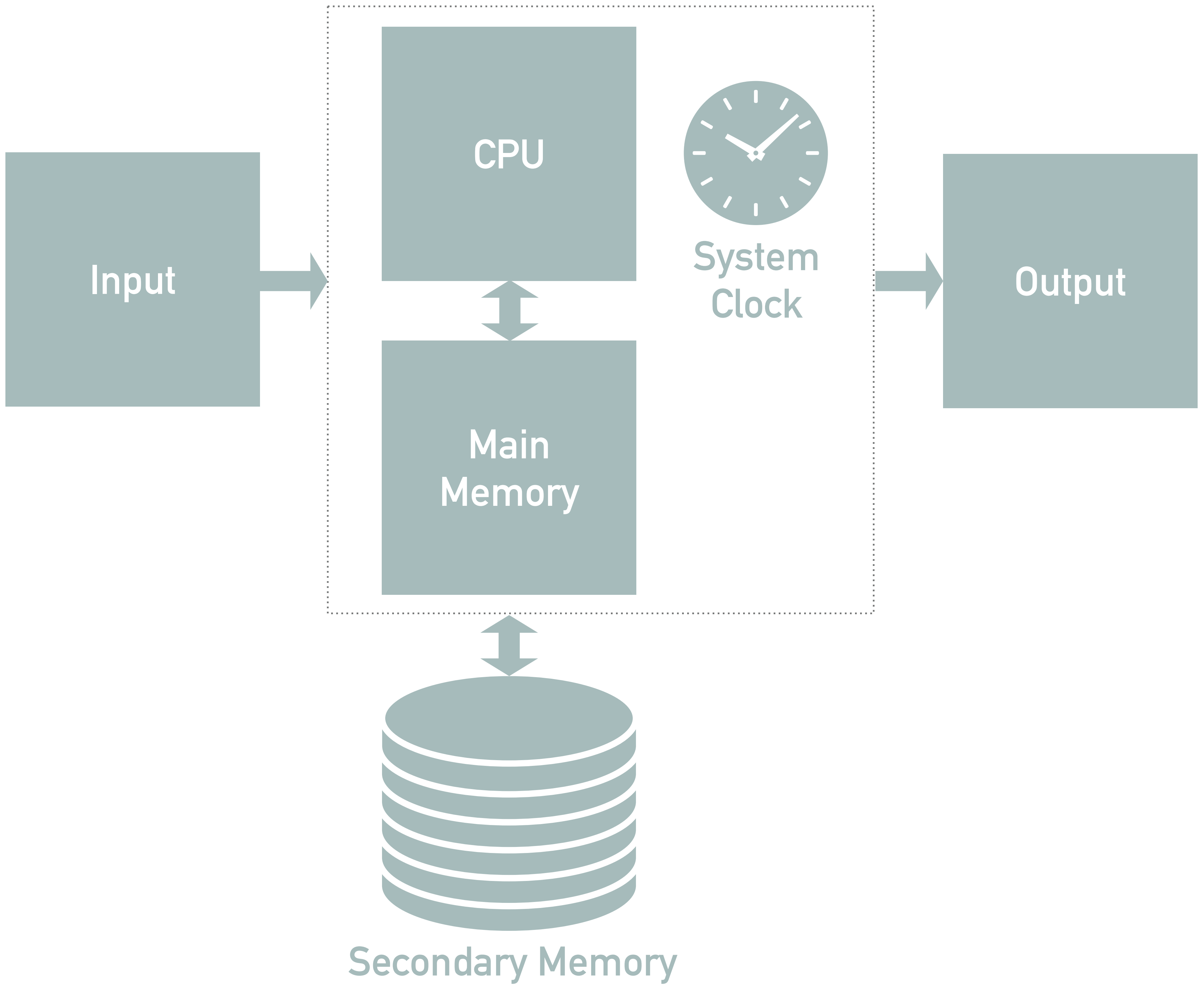
information shoul
Windows. Also co
Addendum encl

THOUT THE SPO
o print from an ap
This may be prefe
onfiguration as it
ature change the
ction of the WIN.IN

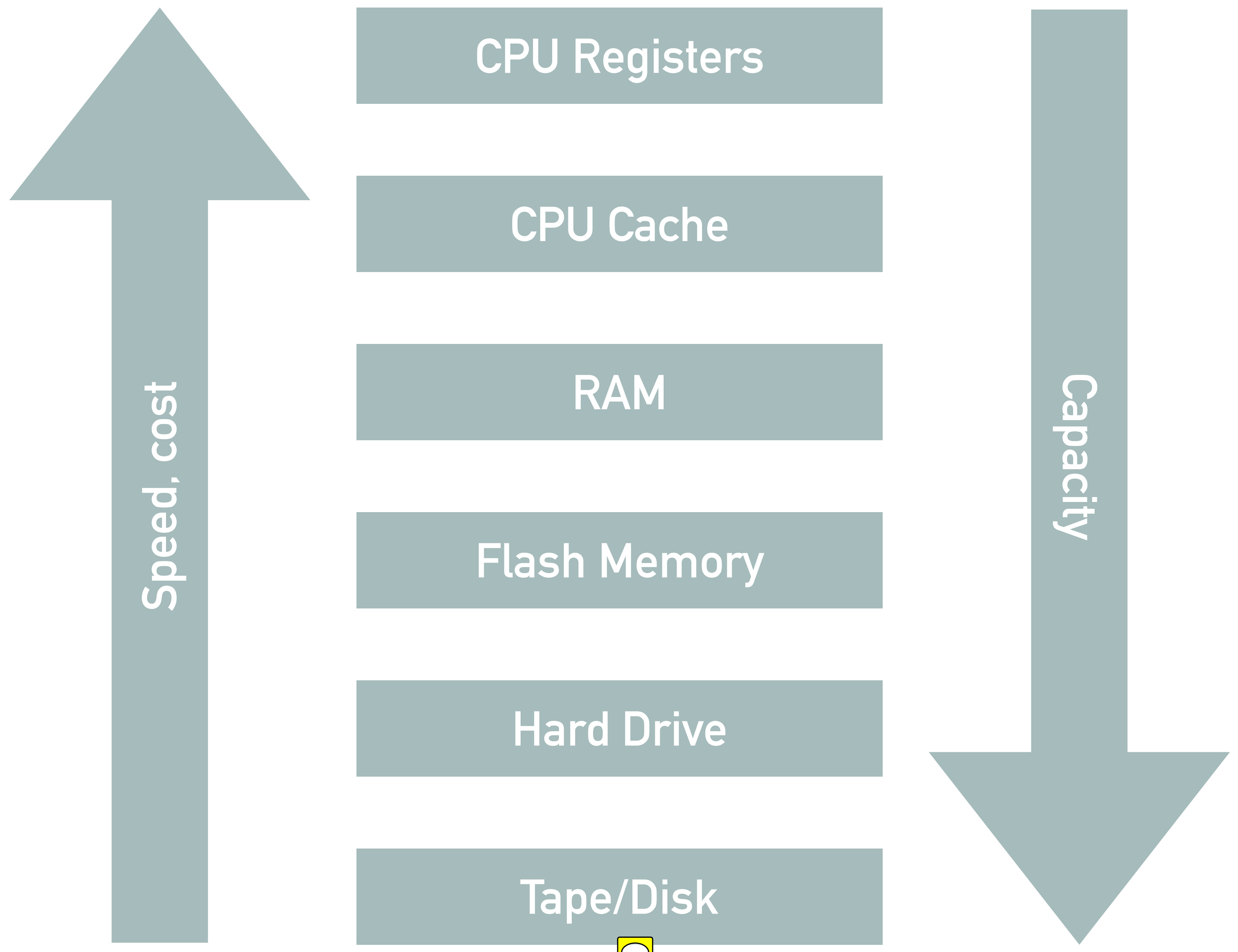
Spooler=no will c
Terminal

RUNNING BATCH (.BAT) FILES
If you run a standard applicatio
should create a PIF file for the b

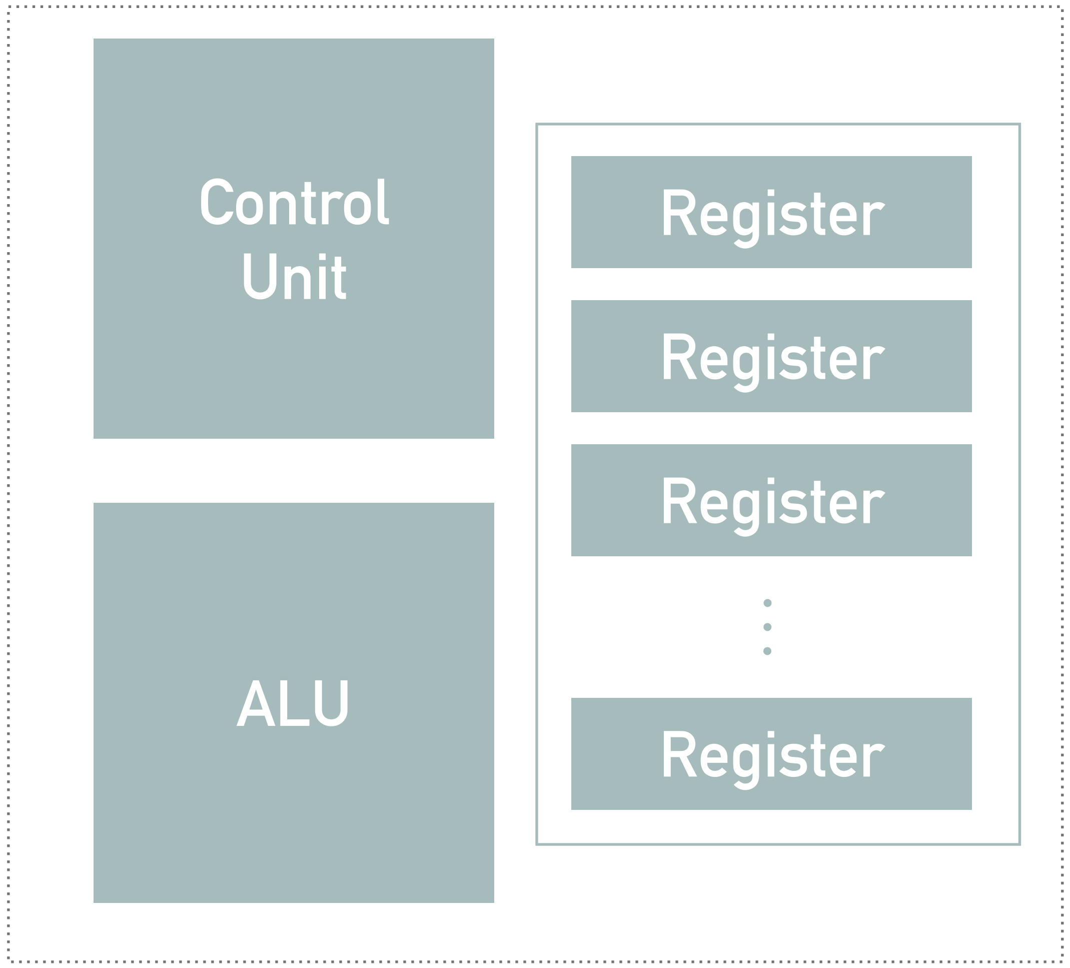
COMPUTER SYSTEM



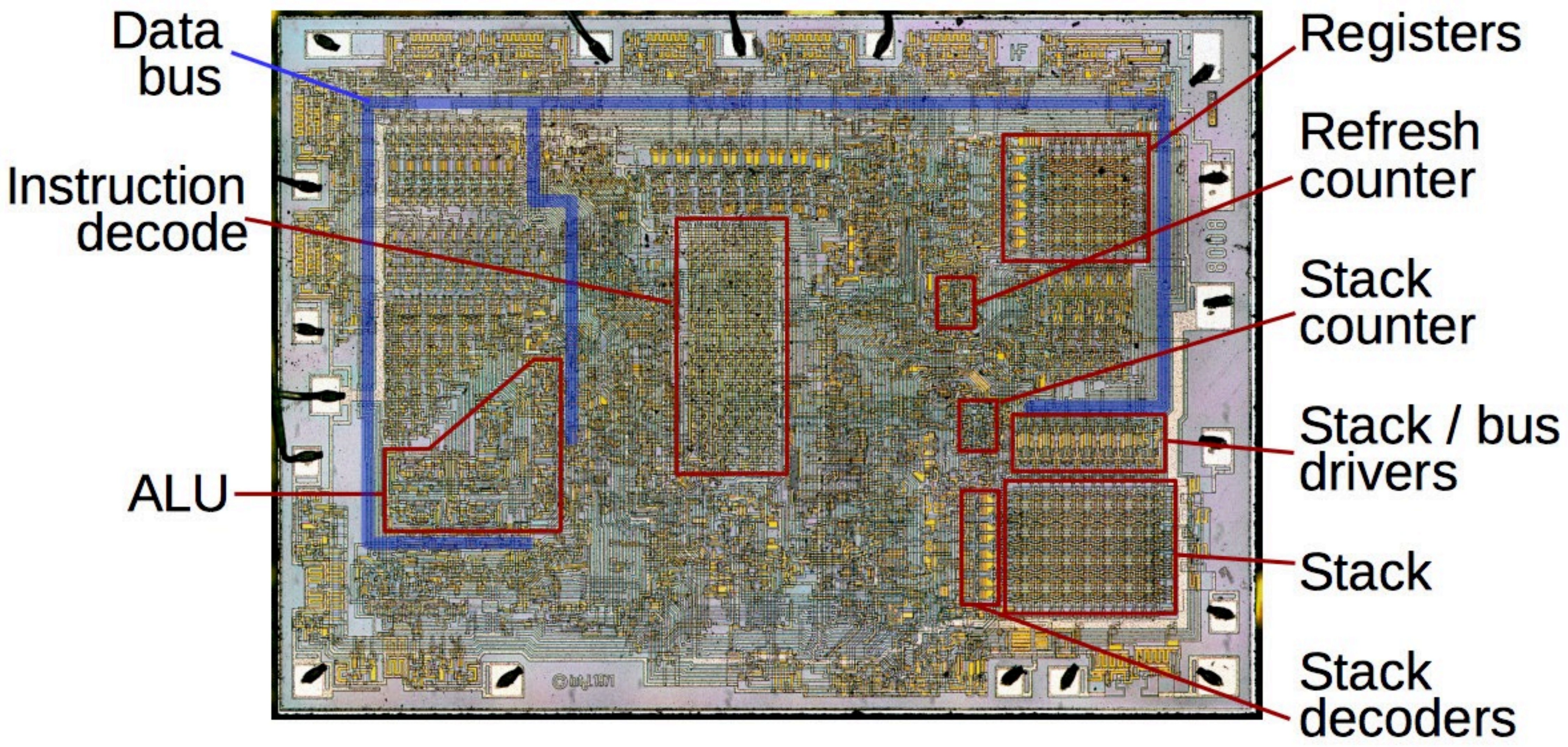
MEMORY HIERARCHY



CPU COMPONENTS



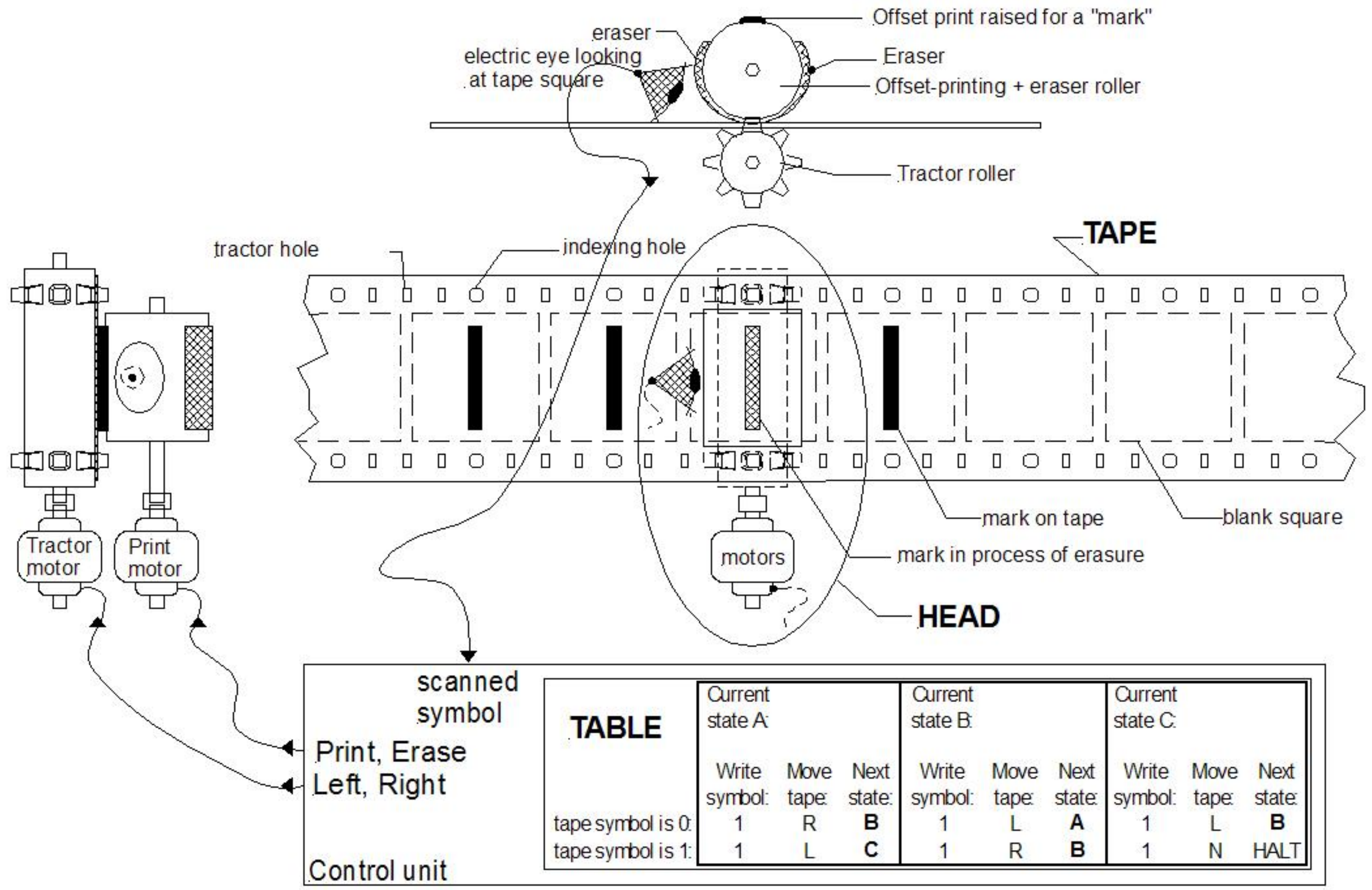
CPU COMPONENTS - INTEL 8008



DATA REPRESENTATION

- Storage unit
  - Bit - "Low" or "High" state representing 0/1
  - Byte - Sequence of 8 bits
  - Word - Sequence of bytes, varies (e.g., 8 bytes in a 64 bit system)
- Notation
  - Binary - 0, 1
  - Octal - 0, 1, 2, 3, 4, 5, 6, 7
  - Hexadecimal - 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

TURING MACHINE



	Current state A:			Current state B:			Current state C:		
	Write symbol:	Move tape:	Next state:	Write symbol:	Move tape:	Next state:	Write symbol:	Move tape:	Next state:
tape symbol is 0:	1	R	**B**	1	L	**A**	1	L	**B**
tape symbol is 1:	1	L	**C**	1	R	**B**	1	N	HALT

A fanciful mechanical Turing machine's TAPE and HEAD. The TABLE instructions might be on another "read only" tape, or perhaps on punch-cards. Usually a "finite state machine" is the model for the TABLE.

TURING MACHINE

CODING THE COMPUTER - ASSEMBLY/MACHINE CODE

https://schweigi.github.io/assembler-simulator/

```
; Simple example
; Writes Hello World to the output

        JMP start
hello: DB "Hello World!"                ; Variable
        DB 0                            ; String terminator

start:
        MOV C, hello                    ; Point to var
        MOV D, 232                      ; Point to output
        CALL print
        HLT                             ; Stop execution

print:                                  ; print(C:*from, D:*to)
        PUSH A
        PUSH B
        MOV B, 0
.loop:
        MOV A, [C]                      ; Get char from var
        MOV [D], A                      ; Write to output
        INC C
        INC D
        CMP B, [C]                      ; Check if end
        JNZ .loop                       ; jump if not

        POP B
        POP A
        RET
```

CPU AND APPLICATIONS

- **Multitasking**: When **blocked** by a task, CPU makes a **context switch** to another task
- **Multiprogramming**: Based on multitasking, CPU switches between different programs when blocked
- **Multithreading**: Switches happen between different "threads" of control within a program
- **Multiprocessing**: Pieces of a single program execute simultaneously in different processors
- **SIMD** (Single Instruction, Multiple Data): CPU processes multiple blocks of data simultaneously

OPERATING SYSTEM

- Core functions
  - Virtual machine: Hide the underlying hardware/software details to provide simplified access through **system calls**
  - Resource manager: Manage and orchestrate the different components for efficiency
- Popular operating systems
  - Microsoft Windows
  - Unix: Many variants like Linux and BSD Unix
  - Apple MacOS: Also based on Unix

TYPES OF LANGUAGES

- Level of abstraction
  - Low level: Assembly/Machine
  - High level: E.g., C/C++, Python
- Code generation
  - Interpreted: E.g., Python, JavaScript
  - Compiled: E.g., C/C++ (binary), Java (byte code)
- Nature of typing
  - Static vs. dynamic typing
  - Strong vs. weak typing

SOFTWARE DEVELOPMENT

- Modularity
  - High **cohesion**, low **coupling**
  - Can be independently developed, tested, and maintained
  - **Top-down** (with **stubs**) vs. **Bottom-up** (with **drivers**) development
- **Algorithms**
- Grading criteria
  - Operation: Correctness, addressing requirements
  - Organization: Separation into logical modules and functions, especially for testability
  - Code quality: Readability, robustness, efficiency

ORIGINS OF C++

- The "C" programming language
  - Created by **Dennis Ritchie**
  - Used to build the **Unix** operating system
  - Largely compatible with C++ and can be mixed freely
- The "C++" language
  - Created by **Bjarne Stroustrup**
  - Meant to improve on C, like adding **classes** for **object oriented** design to improve cohesion and modularity
  - Recent standards: **C++11** (largest set of changes), **C++14,** and **C++17**

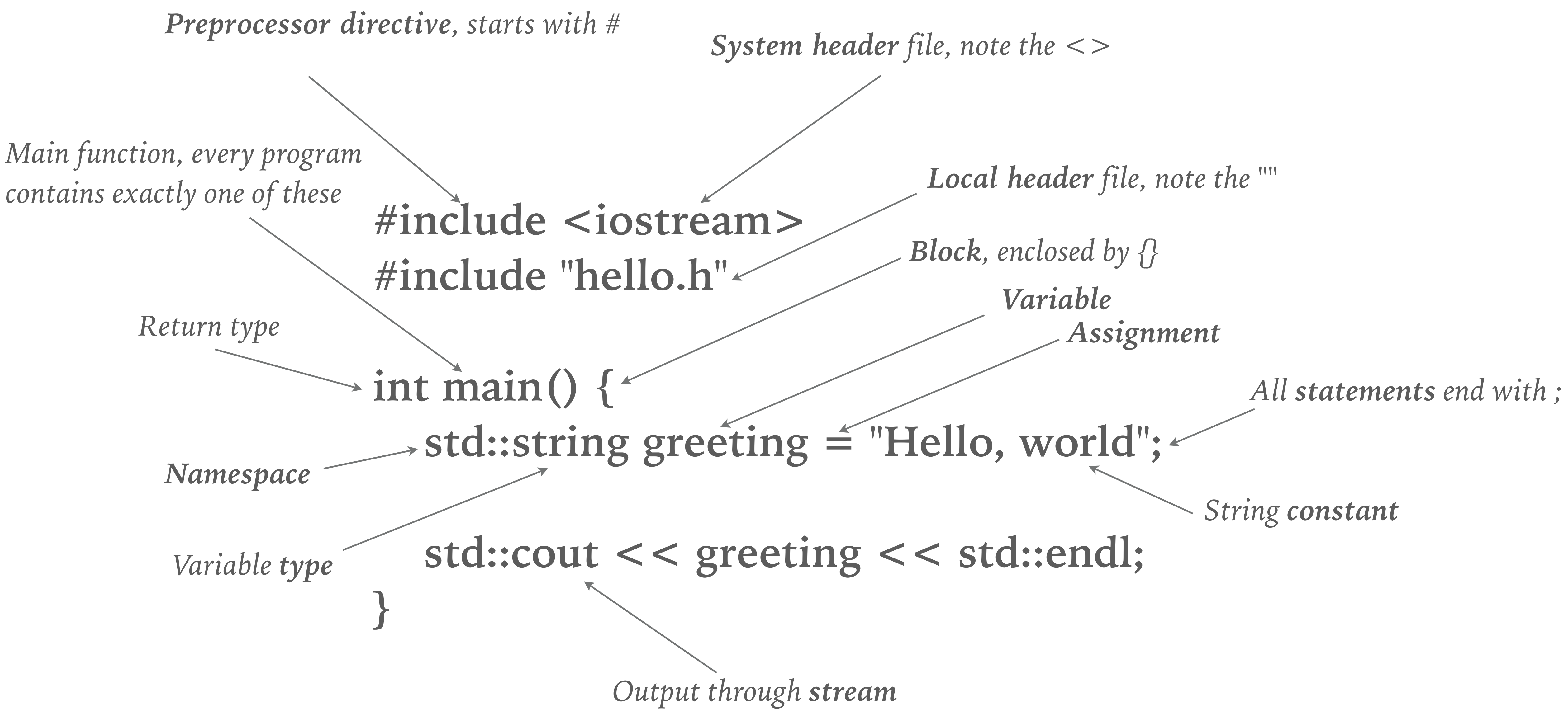
C++ DEVELOPMENT

- Write **source code**
  - **Definitions** in source files (.cpp)
  - **Declarations** in **header** files (.h)
- **Compile** the source code into **object** files (.o)
- **Link** the object files with **libraries** to create an **executable** file (no extension on Linux/Mac, .exe on Windows)
- **Standard libraries** are linked automatically along with compilation, so the compile and link stages are one

ERRORS

- **Compile time** errors
  - Incorrect use of language syntax, illegal access of undeclared functions, etc.
  - Code won't even compile, so they're easy to catch and fix
- **Run time** errors
  - Mistakes in logic (AKA **bugs**), illegal access of memory leading to **exceptions**, etc.
  - May not always occur and can be difficult to track down
  - **Unit testing** lets you test individual modules before they become part of the full application
  - **Debugging** is the process of catching the bugs using **breakpoints** and step/step run through the code

SIMPLE C++ PROGRAM



```
#include <iostream>
#include "hello.h"

int main() {
  std::string greeting = "Hello, world";

  std::cout << greeting << std::endl;
}
```

Preprocessor directive, starts with #
System header file, note the <>
Main function, every program contains exactly one of these
Local header file, note the ""
Block, enclosed by {}
Variable
Return type
Assignment
All statements end with ;
Namespace
String constant
Variable type
Output through stream

INPUT/OUTPUT

- Input
  - **cin** for input stream**, in <iostream>**, namespace **std**
  - Read to variable "hours" with **cin >> hours**
  - Can be chained like **cin >> hours >> minutes**
- Output
  - **cout** for output stream, also in **<iostream>**
  - Write variable "hours" with **cout << hours**
  - Can also be chained with **cout << hours << endl**

COMMENTS

- Type 1
  - // This is a comment
  - Single line only
  - Use sparingly, for short explanations of code that is not readily understood; do NOT parrot the code
- Type 2
  - /* This is a comment */
  - Single or multi line
  - Use at top of file, function, etc. to explain what is happening inside those entities

TYPES

- **Primitive** types
  - Core, built-in types
  - Common types: **bool**, **char**, **short**, **int**, **float**, **double**, **long double** in increasing width in bytes
- **Non primitive** types
  - Built from primitive and other non primitive types
  - Common types: **std::string**, fixed width integers (e.g., **uintmax_t**)
- Keyword **auto** may be used in place of a type (after C++11)
- Use **decltype** to get type of another variable, like **decltype(foo) bar** to declare **bar** to be of same type as **foo**
- **Cast** one type to another with **static_cast** (e.g., static_cast<float>(10))
- Can get width in byes with **sizeof()**

VARIABLES

- Need to be **declared** by specifying a **type**
- Use **identifier** (name) that follows the **camelBack** naming convention (e.g., timeOfDay)
- Must be named to reflect their function (e.g., use **studentName** rather than **sn**, **n**, etc.)
- Made read-only by using keyword **const**, in which case they should follow upper **snake** naming convention (e.g., TIME_OF_DAY)
- Can be **initialized** at the time of declaration or **assigned** a value later

VARIABLE MANIPULATIONS

- Arithmetic operations
  - +, -, *, / for basic add, subtract, multiply, and divide
  - % for modulo (remainder after dividing, like 11 % 3 = 2)
  - The result depends on the operands: e.g., 11 / 2 = 5, but 11 / 2.0 = 5.5
  - ++ increments and -- decrements, before or after expression is evaluated depending where it is placed (e.g., i + j-- decrements after)
  - Some assignments can be simplified (e.g., a = a + 10 as a += 10)
- Boolean operations
  - && for boolean AND (e.g., true && false = false)
  - || for boolean OR (e.g., true || false = true)
  - ! for boolean NOT (e.g., !true = false)
  - Use parenthesis to force evaluation order (e.g., (a || !b) && c)

FLOW CONTROL - IF

- Used for **branching**
- Employs a **boolean expression** and a **block**
- Always create explicit blocks using **braces**
- Combine with **else** to chain

```
if (colors > 6) {                                          // colors > 6
    std::cout << "Too many colors" << std::endl;
} else if (colors < 3) {                                   // colors < 3
    std::cout << "Too few colors" << std::endl;
} else {                                                   // colors = 3, 4, 5, or 6
    std::cout << "Perfect!" << std::endl;
}
std::string size = colors > 4 ? "Large" : "Small";         // Conditional assignment
```

FLOW CONTROL - SWITCH

- Used for **branching** based on a single expression value
- Value should be **int** or **enum**
- Don't forget the **break**

```
switch (code) {
        case 101:
                printf("Fire\n");
                break;
        case 220:
                printf("Earthquake\n");
                break;
        case 311:
                printf("Flood\n");
                break;
        default:
                printf("Unknown\n");
                break;
}
```

FLOW CONTROL - WHILE

- Used for **looping**, checks first
- Employs a **boolean expression** and a **block**
- Always create explicit blocks using **braces**
- Use **break** to quit midway, **continue** to skip

```
int count = 10;
while (count > 0) {                                   // Prints 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 (NOT 0)
	std::cout << "Count: " << count-- << std::endl;
}
count = 10;
while (count > 0) {                                   // Prints 9, 8, 7, 6, 5, 4 (note the start/end values)
	std::cout << "Count: " << --count << std::endl;
	if (count < 5) {
		break;
	}
}
```

FLOW CONTROL - DO-WHILE

- Used for **looping**, checks after one loop
- Employs a **boolean expression** and a **block**
- Always create explicit blocks using **braces**
- Use **break** to quit midway, **continue** to skip

```
int count = 0;
do {                                                        // Prints 0, even though condition is false
        std::cout << "Count: " << count-- << std::endl;
} while (count > 0);
count = 10;
do {                                                        // Prints 9, 8, 7, 6 (note the start/end values)
        std::cout << "Count: " << --count << std::endl;
        if (count < 6) {
                break;
        }
} while (count > 6);
```

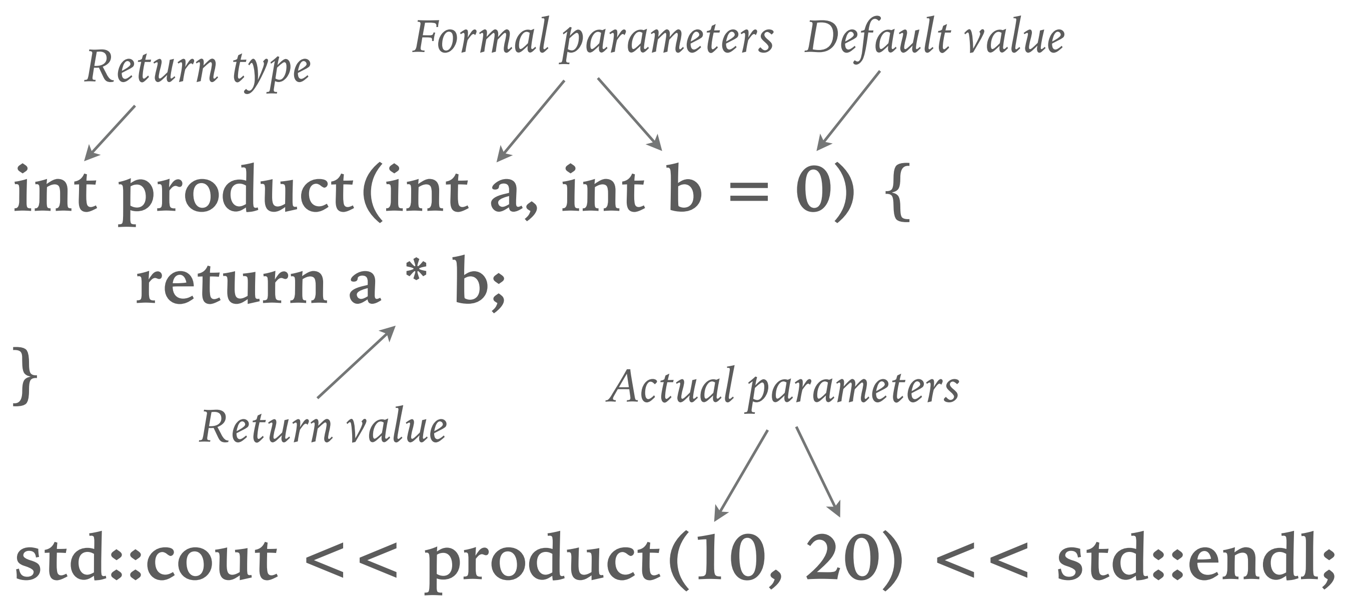
FLOW CONTROL - FOR

- Used for **looping**, by far the most popular
- Employs an **initialization**, a **boolean expression**, a post-loop operation like **increment**, and a **block**
- Always create explicit blocks using **braces**
- Use **break** to quit midway, **continue** to skip

```
for (int i = 0; i < 5; ++i) {                         // Prints 0, 1, 2, 3, 4 (NOT 5)
        std::cout << i << std::endl;
}
for (int i = 0; i < 5; ++i) {                         // Prints 0, 1, 3, 4 (NOT 2)
        if (i == 2) {
                continue;
        }
        std::cout << i << std::endl;
}
```

FUNCTIONS

- Isolate related logic into a **cohesive** entity
- Use when:
  - The same logic is run repeatedly (e.g., sort numbers)
  - Isolating it makes the code easier to read and understand
- **Declared** in the header file and **defined** in the source file



```
int product(int a, int b = 0) {
    return a * b;
}

std::cout << product(10, 20) << std::endl;
```

FUNCTIONS - CONTINUED

```
int product(int a, int b) {
	return a * b;
}

/**
 * This overloads the above product() by using different formal
 * parameters and types. Parameter "c" is passed by reference.
 */
void product(int a, int b, int &c) {
	c = a * b;
}

int main() {
	std::cout << product(10, 20) << std::endl;

	int result;
	product(10, 20, result);
	std::cout << result << std::endl;
}
```

ARRAYS - SINGLE DIMENSION

```
#include <cstdio>

int main() {
   // Space allocated automatically
   int numbers[] = {10, -200, 3};

   // Prints the 3 numbers above
   for (int i = 0; i < 3; ++i) {
      printf("%d\n", numbers[i]);
   }

   // Also iterate this way
   for (int number : numbers) {
      printf("%d\n", number);
   }
}
```

ARRAYS - MULTIPLE DIMENSIONS

```
#include <cstdio>

int main() {
    // Note the "3" for second dimension
    int matrix[][3] = {
        {-2, 44, 200},
        {10, 1, 123}
    };

    // Prints all the numbers above.
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 3; ++j) {
            printf("numbers[%d][%d] = %d\n", i, j, matrix[i][j]);
        }
    }
}
```

ARRAYS - PASS TO FUNCTIONS

```
// Pass size because a[] doesn't disclose it
void processSingle(int a[], int size) {
}

// Other dimensions must be defined
void processMulti(int a[][5][3], int size) {
}

// Prevent elements being changed with const
void readSingle(const int a[], int size) {
}

int main() {
    int single[3];
    int multi[2][5][3];

    processSingle(single, 3);
    processMulti(multi, 2);
}
```

ARRAYS - LIMITATION OF FIXED SIZE

```
#include <cstdio>

int main() {
   // Space allocated for 3
   int numbers[3];

   // Add to pre-allocated space
   numbers[0] = 100;
   numbers[1] = 200;
   numbers[2] = 300;

   // Print the 3 numbers above.
   for (int number : numbers) {
      printf("%d\n", number);
   }
}
```

FILE STREAMS

```
#include <fstream>                              // Stream declarations
#include <cstdlib>                              // For exit()

int main() {
    std::string buffer;                         // Buffer for the copy
    std::ifstream ifs("original.txt");          // Input stream initialized
    std::ofstream ofs;                          // Uninitialized
    ofs.open("output/copy.txt");                // Opened explicitly, note path

    if (ifs.fail() || ofs.fail()) {             // Call fail() to see if successful
        exit(1);                                // Exit with error or use assert() to confirm
    }

    while (std::getline(ifs, buffer)) {         // Reads line, ends on EOF
        ofs << buffer << std::endl;             // Copy each line to output file
    }

    ifs.close();                                // Always close!
    ofs.close();

    exit(0);                                    // Successful exit
}
```

FILE STREAMS - CONTINUED

```
#include <fstream>                              // Stream declarations
#include <cstdlib>                              // For exit()
#include <vector>

struct Name {
   std::string first;
   std::string last;
};

int main() {
   std::ifstream ifs("names.txt");              // List of names
   std::vector<Name> names;                     // Vector to store the names
   Name name;                                   // Name struct as buffer

   while (ifs >> name.first >> name.last) {     // Read and also check for EOF
      names.push_back(name);                    // Add name to vector
   }

   ifs.close();                                 // Always close!

   exit(0);                                     // Successful exit
}
```

ENUM

```
#include <iostream>

enum ColorA {
   AR, AW, AB                              // AR = 0, AW = 1, AB = 2
};

enum ColorB {
   BR = 10, BW = 20, BB = 10               // OK for both BR and BB to be 10
};

enum ColorC {
   CR, CW = 20, CB                         // CR = 0, CW = 20, CB = 21
};

int main() {
   ColorA c1 = ColorA::AR;                 // In enum's namespace
   ColorA c2 = AB;                         // Also global, so must be unique

   std::cout << c2 << std::endl;           // Prints as int: 2
}
```

ENUM CLASS

```
#include <iostream>

enum class ColorA {
   R, W, B                                       // R = 0, W = 1, B = 2
};

enum class ColorB {
   R = 10, W = 20, B = 10                        // OK for both R and B to be 10
};

enum class ColorC {
   R, W = 20, B                                  // R = 0, W = 20, B = 21
};

int main() {
   ColorA c1 = ColorA::R;                        // Must access through namespace
   ColorC c2 = ColorC::B;

   std::cout << (int) c2 << std::endl;           // Not int, needs to be type cast
}
```

C STRINGS

```
#include <cstring>

int main() {
    char name[] = "Jose";              // Length = 5, has '\0' at the end
    char school[] = {'P', 'C', 'C'};   // Length = 3, a char array, not C string
    char buffer[6];                    // Can hold up to a 5-char string

    name[0] = 'R';                     // name = "Rose"
    name[6] = 'A';                     // Out of bounds, will corrupt adjacent data

    strcpy(buffer, "Hi");              // buffer = "Hi"
    int len = strlen(buffer);          // Counts non-null values, so 2 (NOT 3)
    int res = strcmp(buffer, "Hi");    // Equal, so 0

    strcat(buffer, ".");               // buffer = "Hi."
}
```

STRING CLASS

```
#include <string>
#include <iostream>
#include <cstring>

using namespace std;
int main() {
   // All are equivalent and initialize using a C string
   string a = "Hello";
   string b("Hello");
   string c = {"Hello"};
   string d {"Hello"};

   cout << a + ", world" << endl;              // Concatenate with + operator
   a[0] = 'F';                                 // Access directly by indexing

   cout << a.length() << endl;                 // Length (5 here)
   cout << strlen(a.c_str()) << endl;          // Convert to C string and check length (also 5)
   cout << a.empty() << endl;                  // Check if empty (false here)
   cout << a.substr(1, 2) << endl;             // Get substring (el here)
   cout << a.find("o", 2) << endl;             // Find index of substring (4 here)
   cout << (a == b) << endl;                    // Check equality with == operator (false here)
   cout << (a < b) << endl;                    // Check ordering with < operator (true here)
}
```

STRUCT

- Useful for holding multiple variables, especially of different type together as a single entity
- All **struct** entities are really special kind of **classes**

Keyword

Type name

```
#include <iostream>

struct Student {
      int id;
      std::string name;
      bool fullTime;
};

int main() {
      Student student = {23431, "Ace Coder", true};

      printf("%d\n", student.id);
}
```

Members

Note the semicolon

Used like any other type

Initialization similar to array

STRUCT - ACCESS MEMBERS DIRECTLY

```
#include <iostream>

struct Student {
	int id;
	std::string name;
};

void getStudent(Student &student) {
	std::cin >> student.id >> student.name;
}

int main() {
	Student student;

	getStudent(student);
	std::cout << student.id << "\t" << student.name << std::endl;
}
```

STRUCT – ACCESS MEMBERS THROUGH POINTER

```
#include <iostream>

struct Student {
      int id;
      std::string name;
};

void getStudent(Student &student) {
      std::cin >> student.id >> student.name;                          // Access by "."
}

void setName(Student *studentp, std::string name) {
      studentp->name = name;                                           // Access by "->"
}

int main() {
      Student student;

      getStudent(student);
      setName(&student, "Johnny Sokko");
      std::cout << student.id << "\t" << student.name << std::endl;
}
```