



# ICS-103

## Computer Programming in C

Lectures 2~3

### Chapter 1:

## Overview of Computers & Programming

Dr. Tarek Ahmed Helmy El-Basuny

## What we have discussed last time?

### □ We introduced the followings:

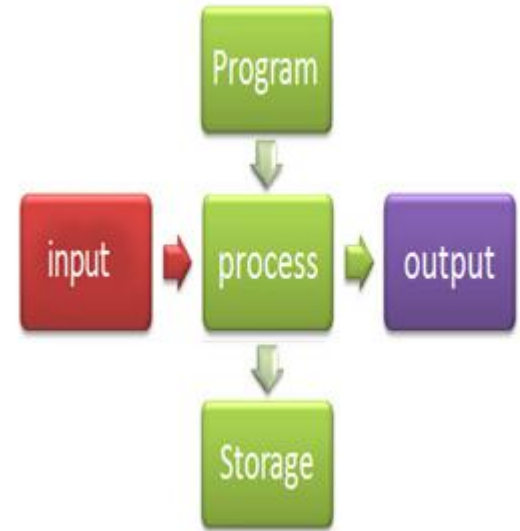
- ⇒ Contact data and setting the office hours,
- ⇒ Course Information,
- ⇒ References of the Course,
- ⇒ Notes for Class Attendance,
- ⇒ Course Outline: Topics to be covered during this course,
- ⇒ Course objective and learning outcomes,
- ⇒ Grading Policies,
- ⇒ Advices for achieving high grade in the course,
- ⇒ Quick Review of:
  - What is a computer program?
  - Imperative programming
  - Declarative programming
  - Similarity between a text book and a computer program

# Topics to be Discussed in this Chapter

- ❑ Overview of Computer System and its Components:
  - Hardware
  - Software
- ❑ Computer's Programming Languages
  - High level Programming Language
  - Assembly Programming Language
  - Low Level (Machine) Programming Languages
- ❑ C Program Development Environment
  - Edit, Compile, Link, Load
- ❑ Software Development Steps
  - Analyze the problem.
  - Design the algorithm to solve the problem.
  - Implement the algorithm (write a program) using a certain language.
  - Test and verify the program.
  - Maintain and update the program if necessary
- ❑ Algorithm Representation
  - Pseudo Code
  - Flowcharts

# Computers

- ❑ A **computer** is a device that accepts information (in the form of digitalized data) and process/manipulates it based on a program (how the data is to be processed) then outputs the result.
- ❑ **Computers are useless without programming.**
- ❑ Computer can deal with input data of different types (i.e. numbers, letters, images, graphics, and sound).
- ❑ **Programming languages (i.e. C)** allow us to write programs that tell the computer what to do and to provide a way to communicate with computers.
- ❑ **Programs and the input data** are then converted to machine instructions so the computer can understand them.



The computer receives input, stores & processes it and then outputs result.

# Hardware & Software

- ❑ Any computing system has a combination of **Hardware** and **Software** .
- ❑ **Hardware** is the tangible equipments used to perform the necessary computations.
  - i.e. **Processor**, **Memory**, **Storage Devices**, **Monitor**, **Keyboard**, **Mouse**, **Printer**, etc.
- ❑ **Software** consists of the programs that enable us to solve problems with a computer by providing it with a list of instructions to follow.
  - **Word Processing** (i.e. **Word**, etc.), **Internet Browsers** (i.e. **Chrome**, etc.), **Operating systems** (i.e. **Windows** , etc.)

# Computer's Hardware (HW)

## ❑ Main Memory

### ➤ RAM: Random Access Memory:

- Memory that can be read or written in any order (as opposed to sequential access memory),
- **Volatile**, its content will be lost once the power is disconnected.
- Used to store currently executing programs.

### ➤ ROM: Read Only Memory:

- Memory that cannot be written to, **stores the BIOS routines**.
- **Non-volatile**, its content will not be lost with power disconnection.

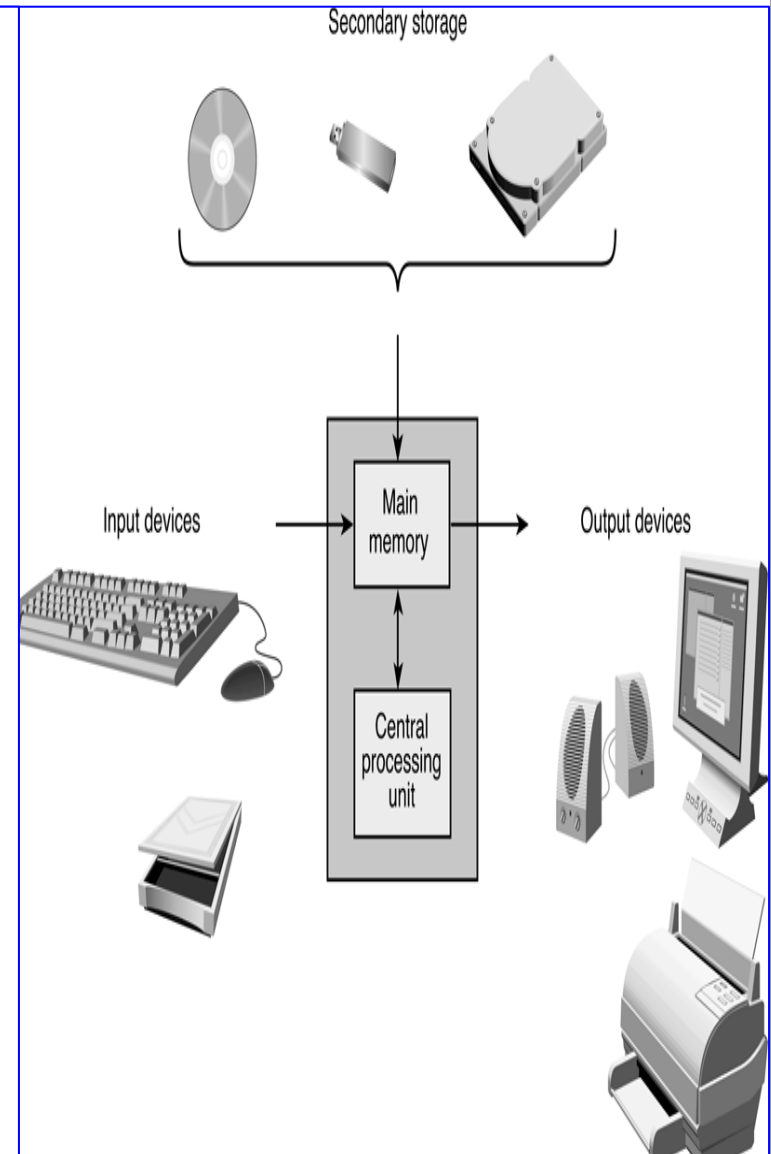
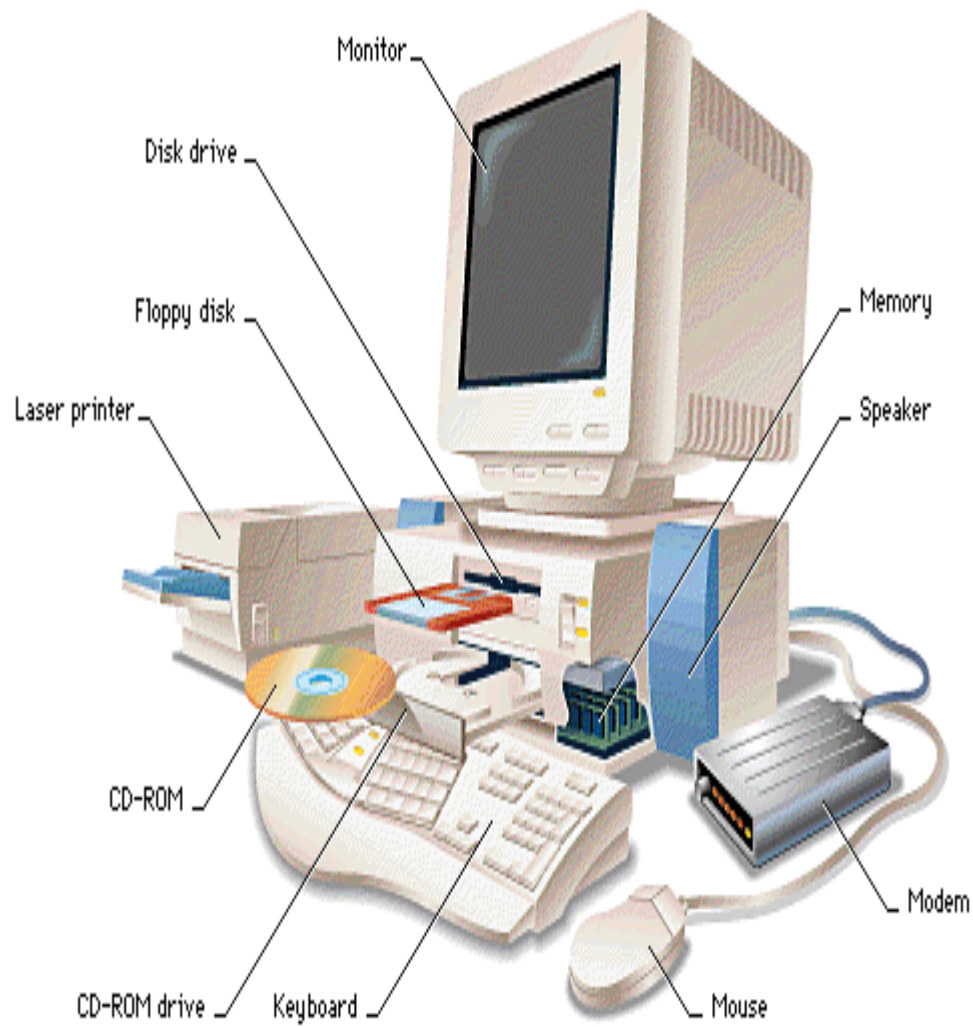
❑ **Secondary Memory**: Magnetic hard disks, Flash (**solid state**) disks, Optical disks (CDs and DVDs).

❑ **Central Processing Unit**: Executes all computer operations and performs arithmetic and logical operations.

❑ **Input and Output Devices**: Keyboard, Mouse, Scanner, Monitor, Printer, Microphone, and Speakers.

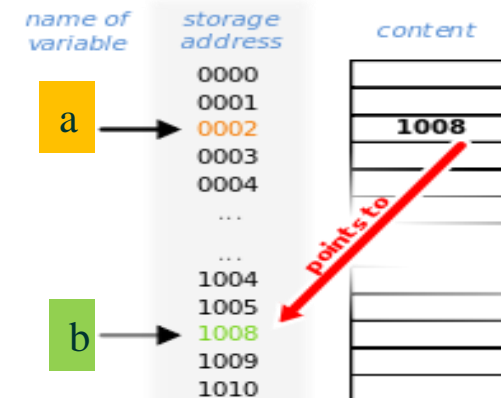
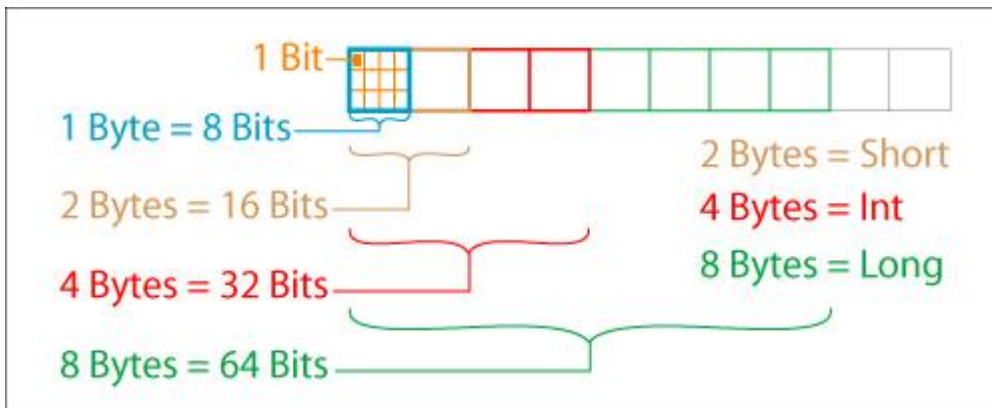
❑ **Networking Devices**: Devices that allow computers to be linked and communicate with each other to form computer networks.

# Components of a Computer System



# Memory

- ❑ **Memory Cell (MC)**: An individual storage location in memory.
- ❑ **Address of a MC**: The relative position of a memory cell in the main memory.
- ❑ **Content of a MC**: Information stored in the memory cell (i.e. instruction or data).
  - Every memory cell has content, whether we know it or not.
- ❑ **Bit**: The name comes from binary digit, either 0 or 1.
- ❑ **Byte**: A memory cell is actually a grouping of smaller units called bytes.
  - A byte is made up of 8 bits.
  - It is the amount of storage required to store a single character (i.e. letter H).





# Understanding Memory

- Each Memory Cell has an **address** and a **value**.
  - The address is an integer number.
  - The value can be an integer, a real number, or a character.
- One Byte = 8 bits.
- One Kilobyte (KB) =  $2^{10} = 1024$  Bytes
- One Megabyte (MB) =  $2^{20} > 10^6$  Bytes
- One Gigabyte (GB) =  $2^{30} > 10^9$  Bytes
- One Terabyte (TB) =  $2^{40} > 10^{12}$  Bytes
- One Petabyte (PB) =  $2^{50} > 10^{15}$  Bytes
- One Exabyte (EB) =  $2^{60} > 10^{18}$  Bytes
- One Zettabyte (ZE) =  $2^{70} > 10^{21}$  Bytes
- One Yottabyte (YB) =  $2^{80} > 10^{24}$  Bytes

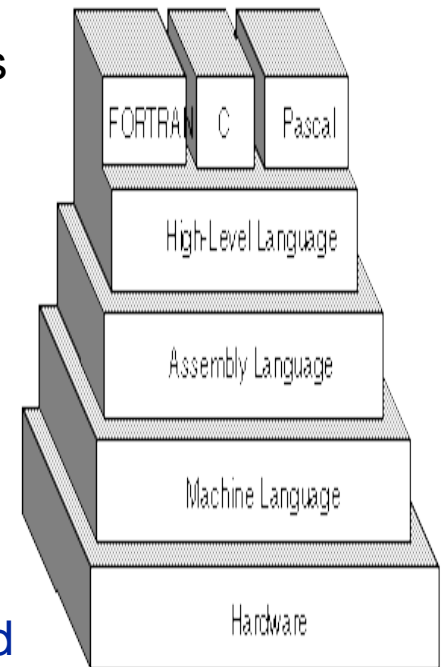
Memory	
Address	Contents
0	-27.2
1	354
2	0.005
3	-26
4	H
.	.
.	.
.	.
998	X
999	75.62

## Computer's Software (SW)

- ❑ **Software** is a general term for the various kinds of programs used to operate **computers** and related devices.
- ❑ Without software, computers would be useless.
- ❑ Based on the goal, computer's software can be divided into:
  - **Operating System**: controls the interaction between machine and user  
(**Examples**: Windows, Mac, Linux, ios, Android, etc.)
    - Manages the input and output devices.
    - Communicates with computer's user.
    - Manages memory and processor time.
    - Manages Storage Disks.
  - **Application Software**: developed to assist a computer's user in accomplishing specific tasks. Example: **Word, Excel, PPT, Internet Explorer, etc.**

# Computer Programming Languages

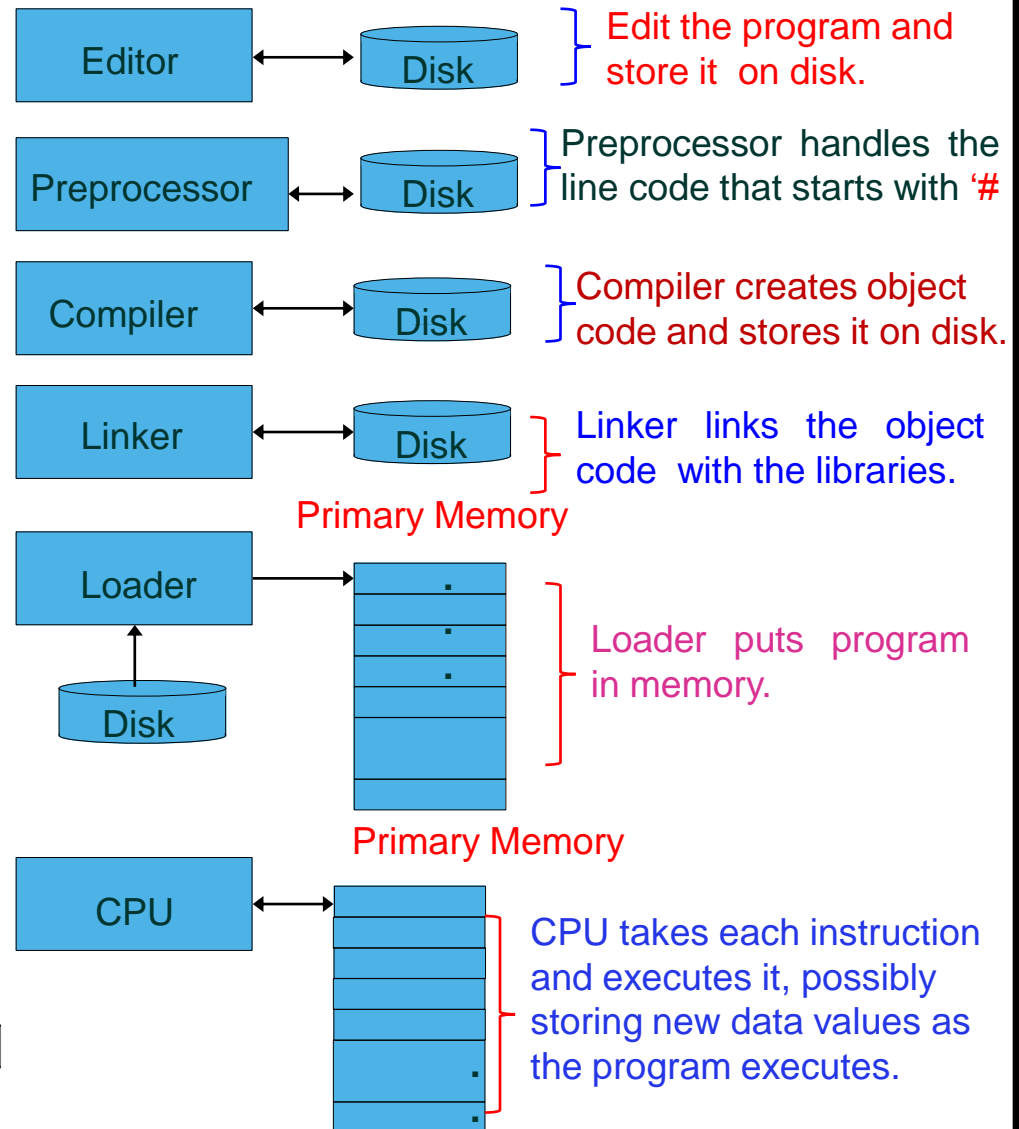
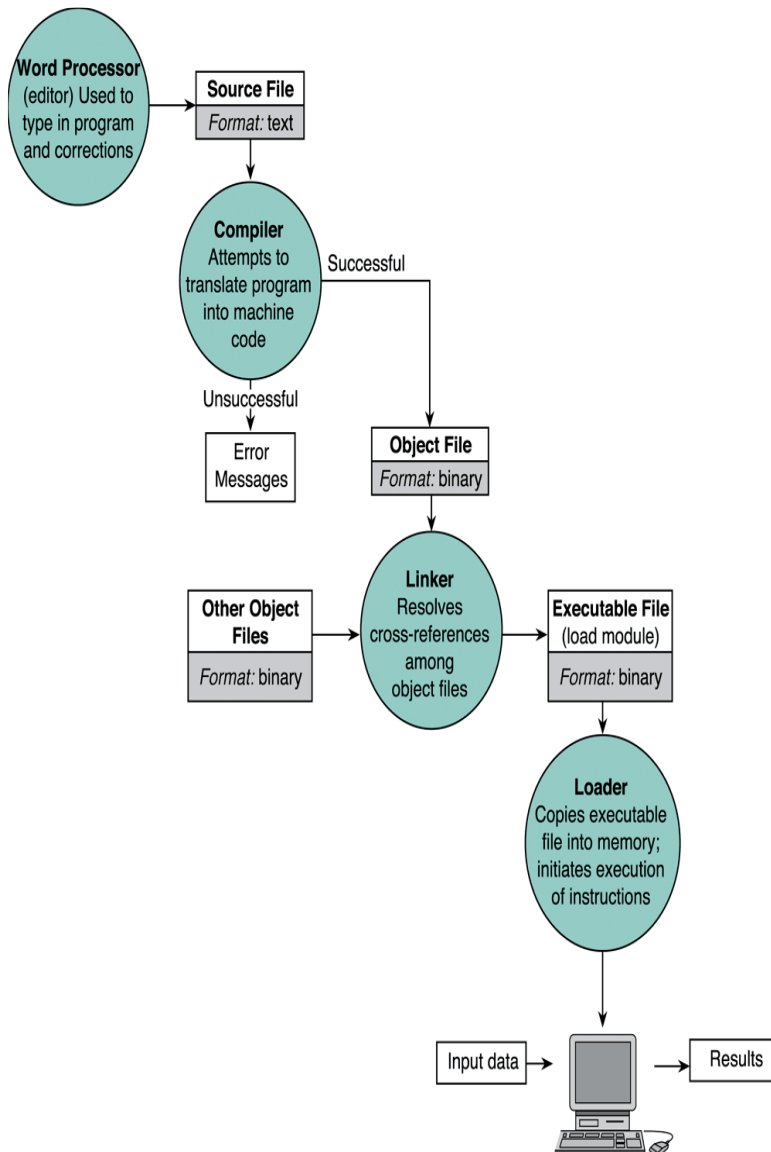
- ❑ **High-level Language:** is a programming language designed to simplify computer programming.
  - it combines algebraic expressions and high-level commands that can be read by the human.
  - **Examples:** Fortran, C, C++, Prolog, Perl, and Java.
- ❑ **Assembly Language:** is programming **language** that uses symbols (**called mnemonics**) that correspond to machine language instructions.
  - Very close to the actual machine language.
  - An assembly program written for one type of CPU won't run on another.
- ❑ **Machine Language:** Programming language that can be directly understood by a computer without conversion (**compilation**).
  - It is the native binary language and is difficult to be read and understood by humans.
  - There is a different machine language for every processor family.



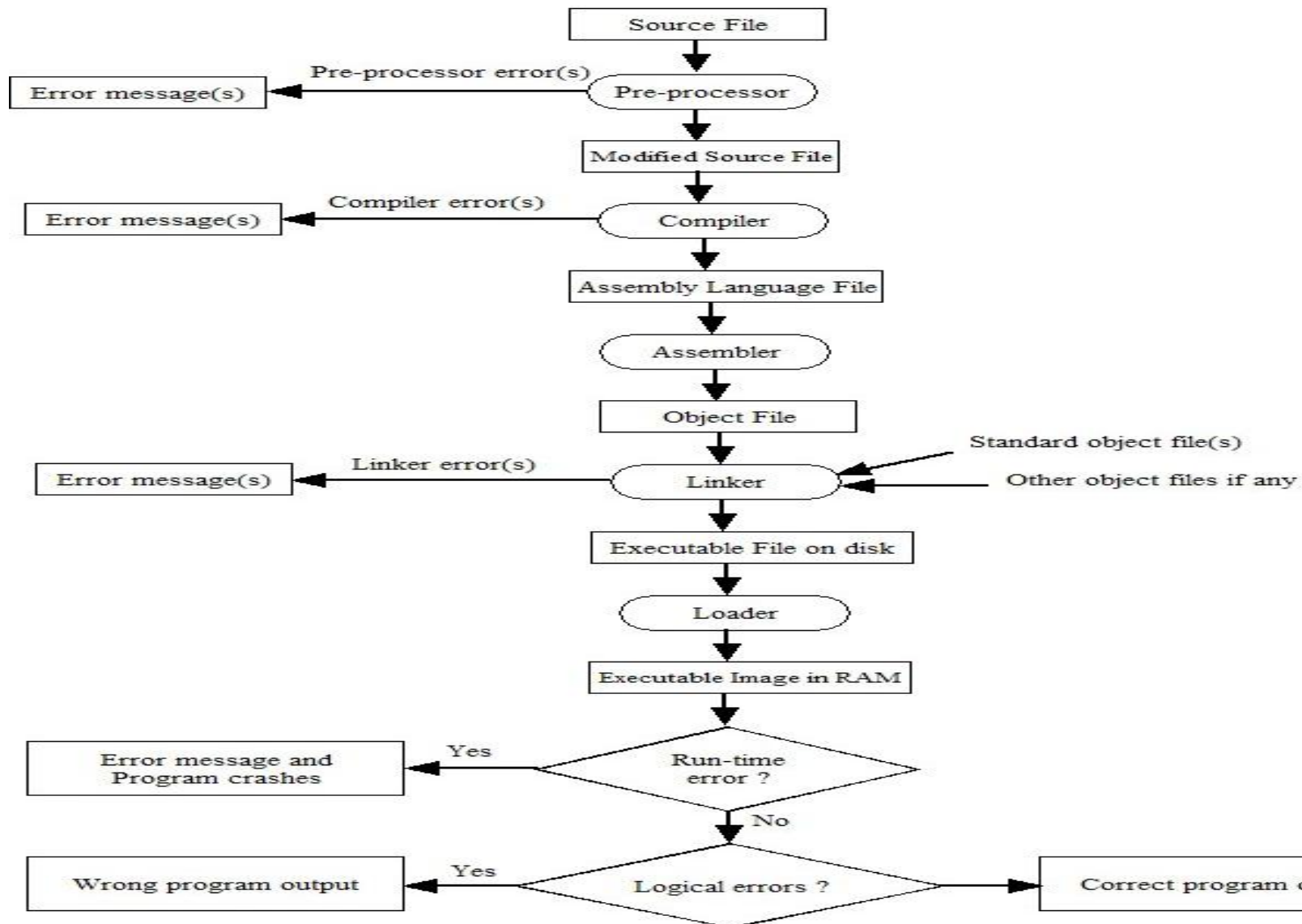
# The Compiler

- ❑ Compilation is the process of translating the source code (**high-level**) into executable code (**machine level**).
- ❑ **Source file**: contains the original program code.
  - A Compiler turns the **Source File** into an **Object File**
- ❑ **Object file**: contains low level instructions which can be understood by the CPU.
  - A Linker turns the **Object File** into an **Executable**
- ❑ **Executable object file**: It contains machine code that can be directly loaded into memory and then executed.
  - The linker links together a number of object files to produce a binary file which can be executed.
- ❑ **Integrated Development Environment (IDE)**: a SW package that combines simple text editor with a compiler, linker, loader, and debugger tool.
  - For example, GCC, Borland C, Eclipse or Visual Studio, etc.

# C Program Development Environment



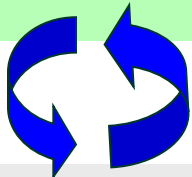
# Compilation Process of a C Program



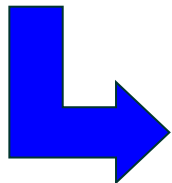
# Writing and Running C Programs

```
#include <stdio.h>

/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello world\n");
    return 0;
}
```



```
$ gcc -Wall -g my_program.c -o my_program
tt.c: In function `main':
tt.c:6: parse error before `x'
tt.c:5: parm types given both in parmlist and separately
tt.c:8: `x' undeclared (first use in this function)
tt.c:8: (Each undeclared identifier is reported only once
tt.c:8: for each function it appears in.)
tt.c:10: warning: control reaches end of non-void function
tt.c: At top level:
tt.c:11: parse error before `return'
```



my\_program

- Write the source code program using an editor such as **emacs** and save it as file e.g. **my\_program.c**

- The compiler converts the program from source to an “**executable**” or “**binary**”:

- If the **Compiler** gives any errors and/or warnings; then re-edit the source file again to fix it, and re-compile it again.

- Resolves external references to produce the executable program through **Linking** process.

- Run it and see if it works

# A Quick Digression About the Compiler

```
#include <stdio.h>
/* The simplest C Program */
int main(int argc, char **argv)
{
    printf("Hello world\n");
    return 0;
}
```

Preprocess

Compilation occurs in two steps:  
“Preprocessing” and “Compiling”

Why ?

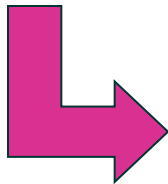
```
__extension__ typedef unsigned long long int __dev_t;
__extension__ typedef unsigned int __uid_t;
__extension__ typedef unsigned int __gid_t;
__extension__ typedef unsigned long int __ino_t;
__extension__ typedef unsigned long long int __ino64_t;
__extension__ typedef unsigned int __nlink_t;
__extension__ typedef long int __off_t;
__extension__ typedef long long int __off64_t;
extern void flockfile (FILE *__stream) ;
extern int ftrylockfile (FILE *__stream) ;
extern void funlockfile (FILE *__stream) ;
int main(int argc, char **argv)
{
    printf("Hello world\n");
    return 0;
}
```

In **Preprocessing**, source code is “expanded” into a larger form that is simpler for the compiler to understand. Any line that starts with ‘#’ is a line that is interpreted by the **Preprocessor**.

- Include files are “pasted in” (**#include**)
- Macros are “expanded” (**#define**)
- Comments are stripped out ( **/\* \*/ , //** )
- Continued lines are joined ( **\** )

\ ?

The compiler then converts the resulting text into **binary** code so that the CPU can run it.

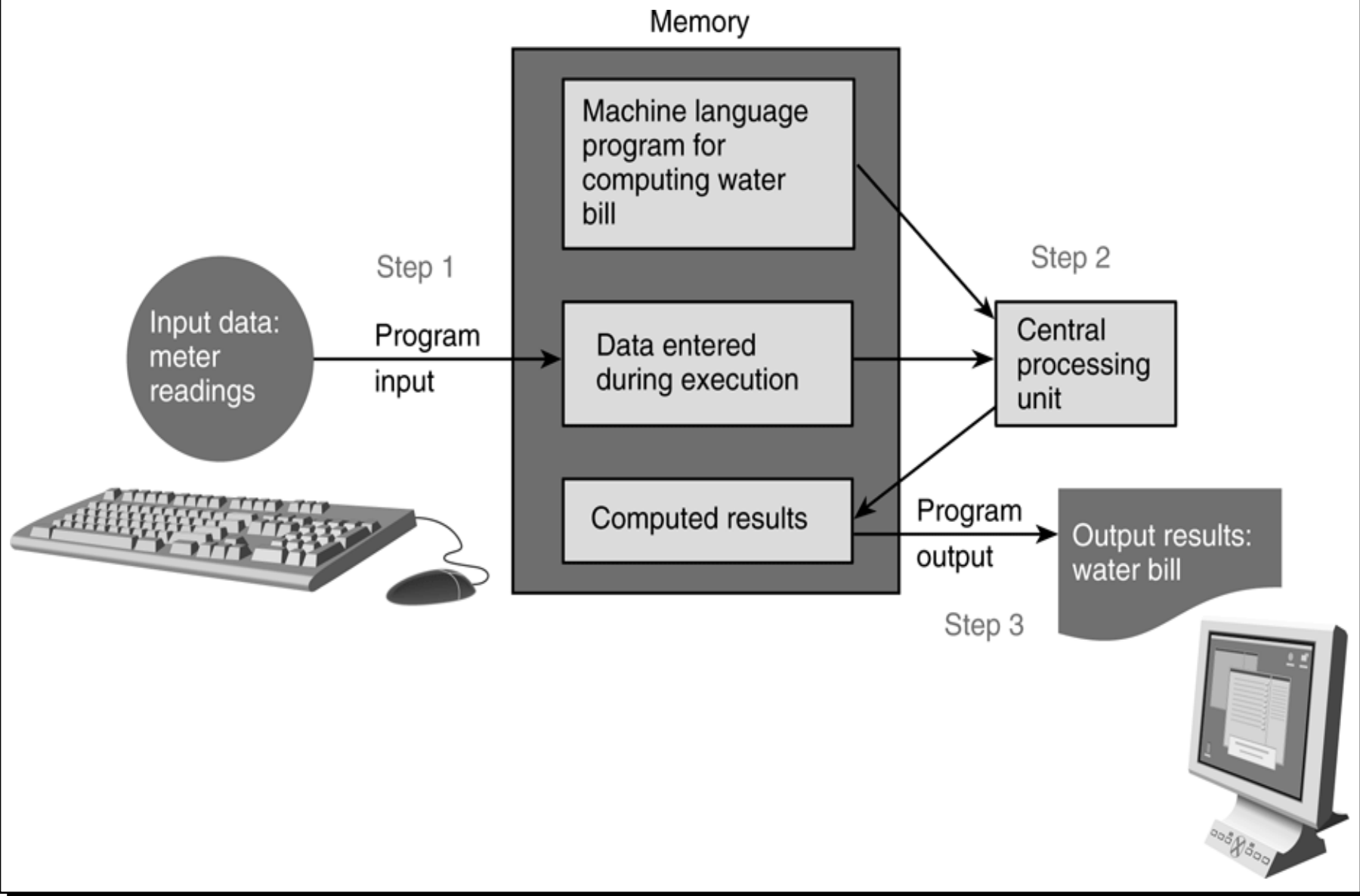


Compile

my\_program



# Flow of Information During Program Execution



## Steps of Handling a Problem

1. **Problem identification**: Specifies the problem that should be solved on the computer.
2. **Analysis**: Understand the problem and identify the **required inputs**, **outputs**, and **computation**.
3. **Design**: Design and develop the list of steps **called algorithm** to solve the problem.
4. **Implementation**: Write the algorithm as a program using a certain programming language.
5. **Testing**: Test and verify that the program actually works as desired.
6. **Maintenance**: Maintain any undetected errors and keep it up-to-date.

## Example: Converting Miles to Kilometers

### □ Problem example:

- You have been asked to convert a list of **miles** to kilometers.
- Since you like programming, you decide to write a C program to do the job.

2. **Analysis:** Outline the problem and its requirements, **What is the information that we need to process in order to find the solution?** i.e., in the above problem:

- We need to receive miles as **input**
- We need to **output** kilometers
- We know that **1 mile = 1.609 kilometers**

3. **Design:** set the algorithm to solve the problem.

1. Get distance in miles as **input**
2. Convert it to kilometers using a **certain formula**.
3. Display kilometers as **output**

## 4. Implementation in C Language

```
1.  /*
2.   * Converts distance in miles to kilometers.
3.   */
4.  #include <stdio.h>           /* printf, scanf definitions */
5.  #define KMS_PER_MILE 1.609   /* conversion constant      */
6.
7.  int
8.  main(void)
9.  {
10.     double miles, /* input - distance in miles.      */
11.            kms;    /* output - distance in kilometers */
12.
13.     /* Get the distance in miles. */
14.     printf("Enter the distance in miles> ");
15.     scanf("%lf", &miles);
16.
17.     /* Convert the distance to kilometers. */
18.     kms = KMS_PER_MILE * miles;
19.
20.     /* Display the distance in kilometers. */
21.     printf("That equals %f kilometers.\n", kms);
22.
23.     return (0);
24. }
```

### Sample Run

```
Enter the distance in miles> 10.00
That equals 16.090000 kilometers.
```

### 5. Test

- ➡ We need to test the previous program to make sure it works.
- ➡ We run our program and enter different values and make sure the output is correct.

### 6. Maintenance

- ➡ Next time, your boss gets a contract with NASA, so he wants you to add support for converting to Astronomical Unit (AU), where AU is the Earth's average distance from the Sun.
  - 1 Astronomical Unit = 149,597,871 Kilometers

## Representing Algorithms

❑ **Algorithm:** A list of ordered steps for solving a problem in a specified time.

- An algorithm can be represented using:

- ❑ Formulas

- ❑ English description of the algorithm

- ❑ Pseudo-code

- ❑ Flowcharts

- ❑ High-level programming language

More easily  
expressed

More  
precise

## Pseudo Code & Flowchart

- ❑ **Pseudo Code:** A combination of English phrases and language constructs to describe the algorithm steps.
- ❑ **Flowchart:** A diagram that shows the step-by-step execution of a program.

# Why use Pseudo Code?

- ❑ **Pseudo code** is an artificial and informal language that helps programmers develop algorithms.
- ❑ **Pseudo Code:** Written as a combination of **English and programming constructs**
  - Based on selection (**if, then, else, switch**) and iteration (**for, while, repeat**) constructs in high-level programming languages.
- ❑ **The benefits of pseudo code are:**
  - It enables the programmer to concentrate on the algorithm without worrying about all the syntactic details of a particular programming language.
  - You can write pseudo code without even knowing what programming language you will use for the final implementation.
  - Pseudo code cannot be compiled or executed,
  - It does not follow syntax rules.
  - It is simply an important step in producing the final code.
- ❑ **Pseudo Code** Example:  
Input Miles  
 $\text{Kilometers} = \text{Miles} * 1.609$   
Output Kilometers



## Example: Pseudo Code

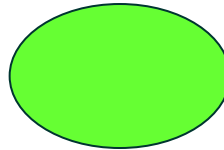
- ❑ **Problem:** Calculate your final grade for ICS 103.
- ❑ **Understand the problem:**
  - Get different grades as inputs and then compute the final grade as an output.
- ❑ **Analyze the problem:**
  - We need to input grades for exams, labs, quizzes.
  - We need to know the percentage each part counts for.
  - We need to use a formula for calculating the final grade.
  - Then we need to output the final grade.
- ❑ **Design the algorithm**
  1. Get the grades: exams, quizzes, assignments, and labs.
  2.  $\text{Grade} = 0.25 * \text{Midterm Exam} + 0.3 * \text{Final Exam} + 0.2 * \text{Quizzes} + 0.5 * \text{Assignments} + 0.2 * \text{Lab}$
  3. Output the Grade
- ❑ **Implement and Test:**
  - Learn how to program in C,
  - Write the program, then Input some test values,
  - Calculate and check the final grade.

# Flowcharts

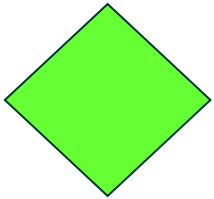
- ❑ A Flowchart is graphical representation of an algorithm.
- ❑ Written as a combination of the following graphical notations:



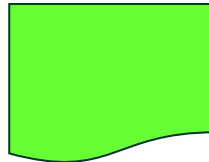
Processing



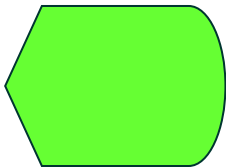
Start or Terminal



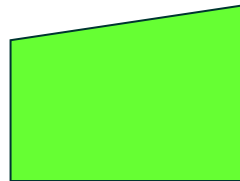
Decision



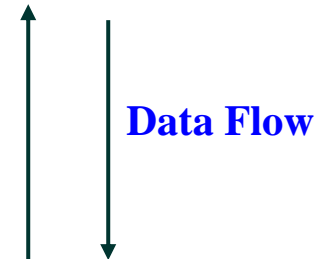
Document



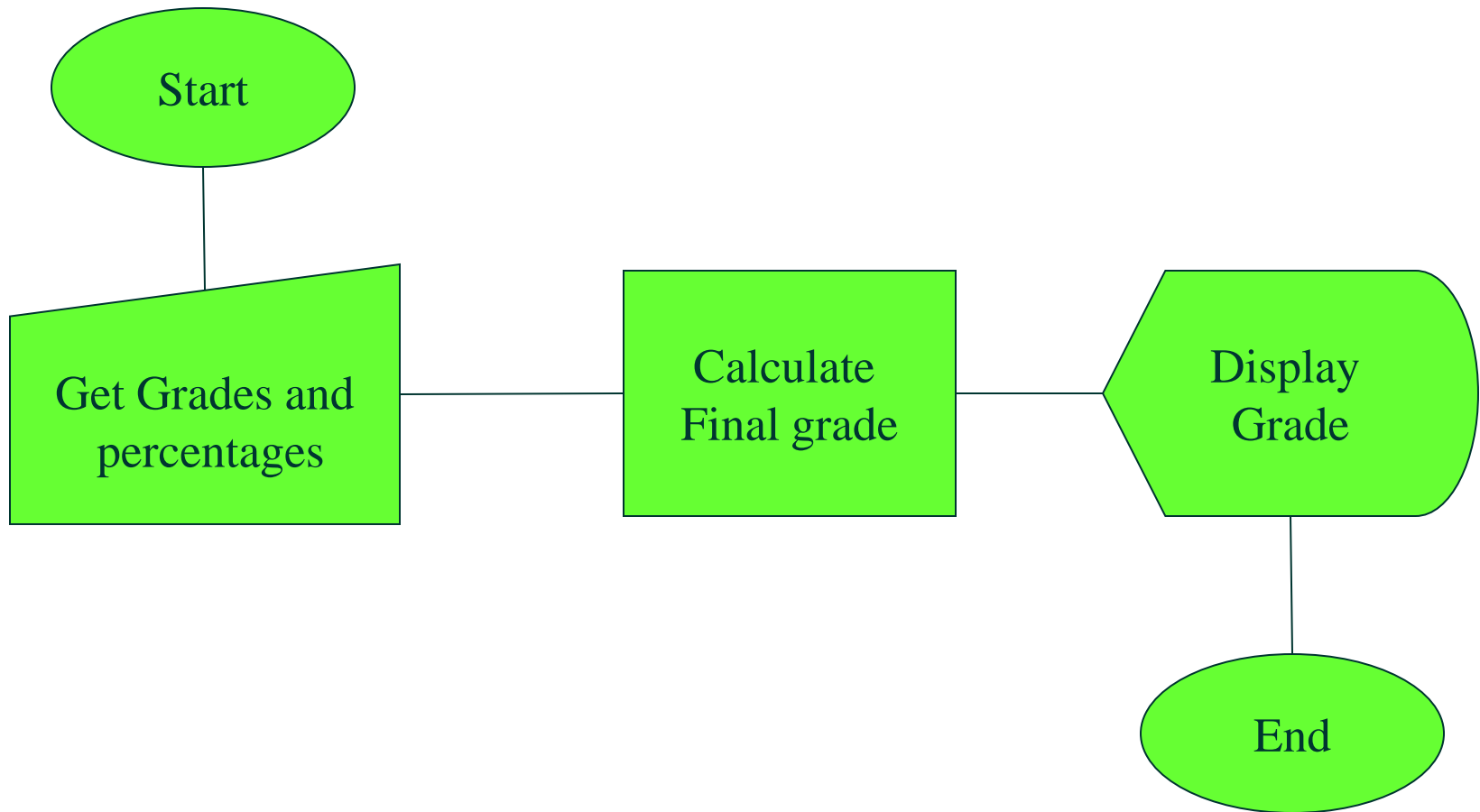
Display



Manual Input

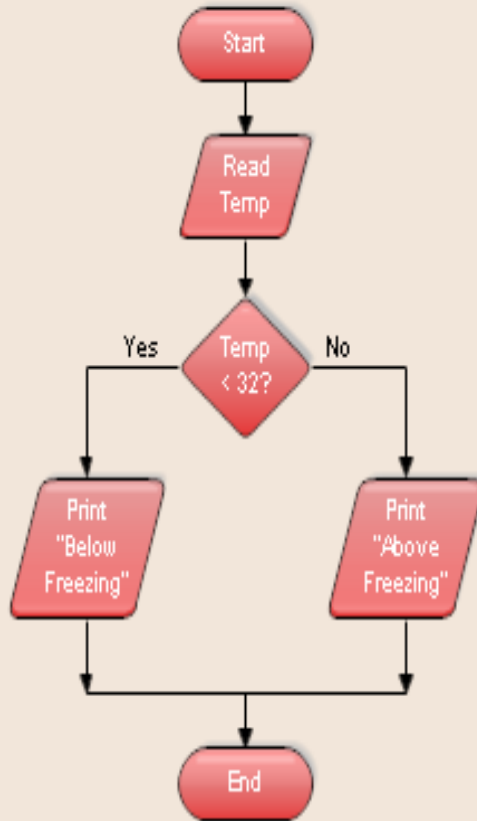


## Example of a Flowchart

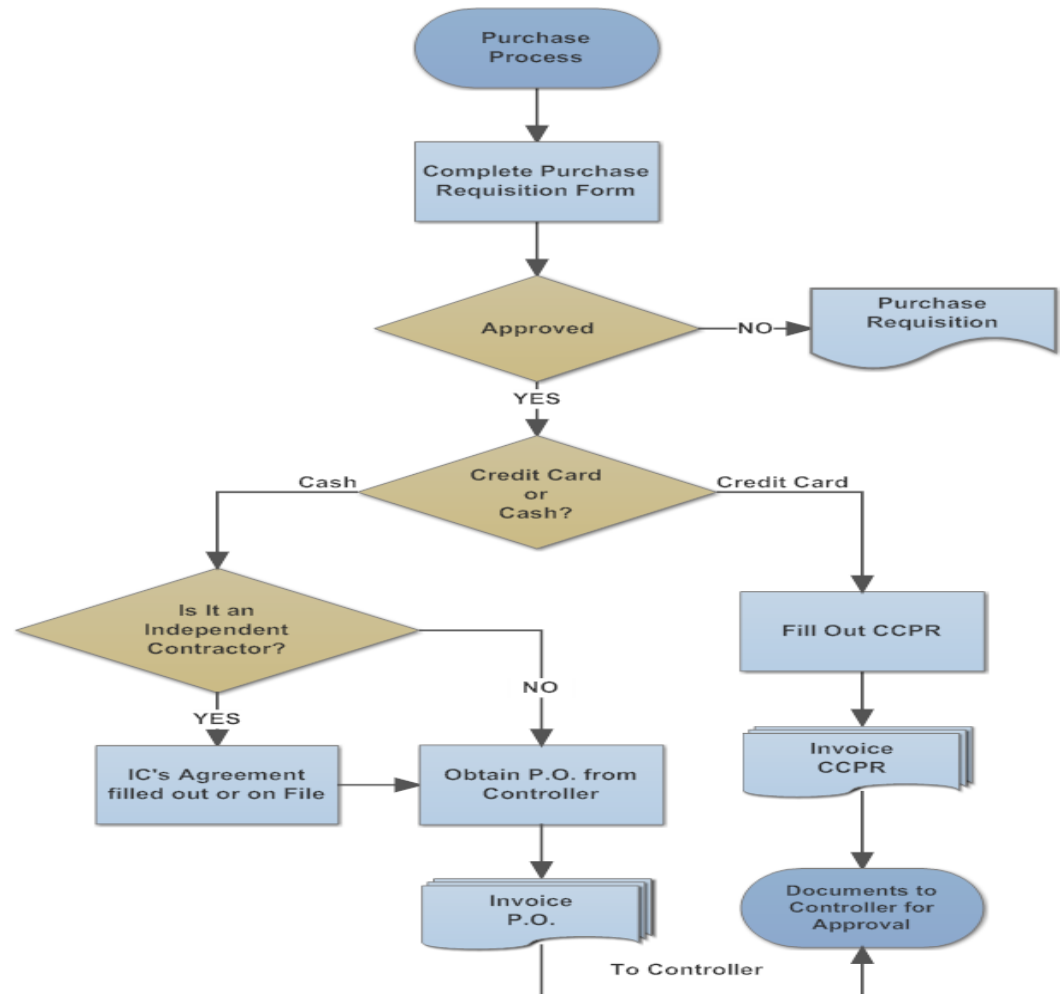


# Flow Chart Representation: Examples

Flowchart



PURCHASING PROCESS





**The End!!**

**Thank you**

**Any Questions?**