BLG 102E
Introduction to Scientific Computing and Engineering

SPRING 2025

Ali Çakmak

İTÜ

İSTANBUL TEKNİK ÜNİVERSİTESİ 1773

ISTANBUL TECHNICAL UNIVERSITY

Course Information

Topics

- introduction to programming
- using the C language

- with only a brief introduction to C++ in the last weeks

Weekly Schedule

- introduction
- data types
- data types
- decisions
- repetition
- functions
- functions
- arrays
- arrays and functions
- pointers, strings
- dynamic memory management
- structures, file operations
- preprocessing
- classes

Credits

- 3 hours lecture, 2 hours practice
- practice sessions in lab

Grading

- Labs: 10%
- Two midterms
  - 1st midterm exam: 25%
  - 2nd midterm exam: 25%
- Final exam: 40%

VF Conditions:

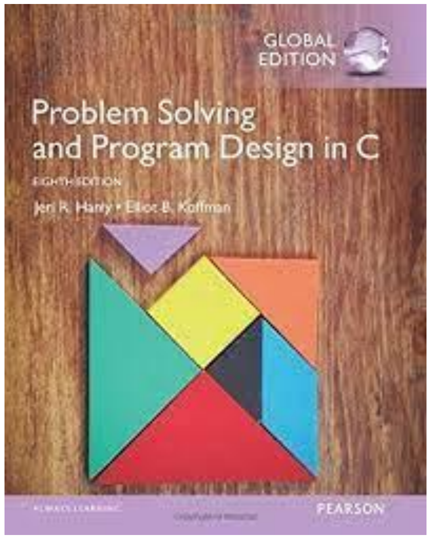
- Midterm exams average < 30/100 **or**
- Attendance to the class < 70% **or**
- Lab Attendance-credit < 8 (out of 12)

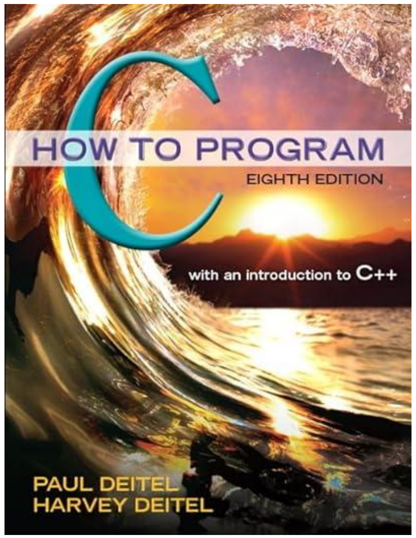
Labs

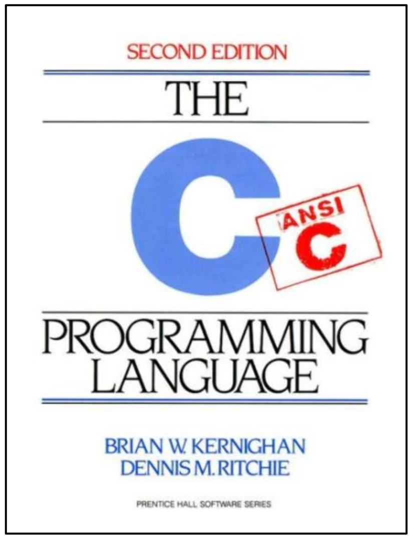
- Lab sessions will be held in computer labs on Thursdays.
  - You may bring your own computer or use the lab computer.
- VF Rule
  - You need to get **attendance credit** for at least 8 labs (out of 12).
  - You will **have to submit proof-of-work to get attendance credit** for the lab sessions.
    - Teaching assistants will guide you on how to do that.
- Otherwise, you will fail with the grade VF and cannot take the final exam.

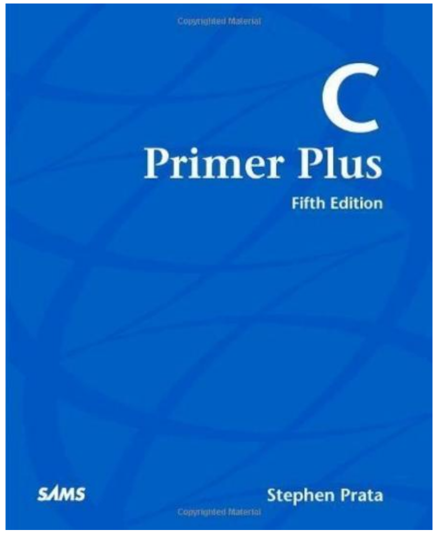
Textbooks

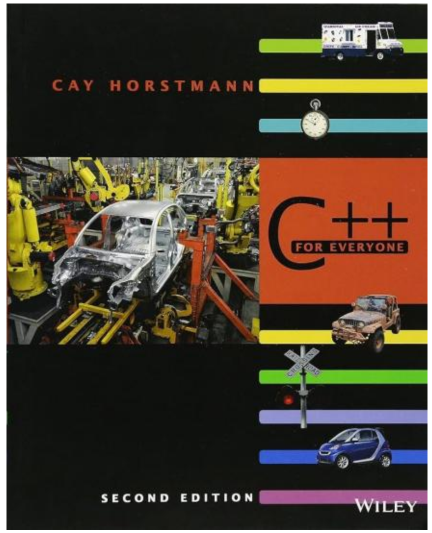
- Problem Solving and Program Design in C, 8th ed. by Jeri Hanly, Elliot Koffman
- C How to Program, 8th Edition by Paul Deitel, Harvey Deitel
- C Primer Plus, 5th Edition, by Stephen Prata
- C Programming Language, 2nd Edition, by Brian W. Kernighan, Dennis M. Ritchie
- C++ for Everyone, 2nd Edition, by Cay S. Horstmann











Logistics

- own computer: any Linux installation
- gcc compiler suite (c99)
- any text editor

Ninova

http://ninova.itu.edu.tr/

- resources and slides
- announcements
- grades
- attendance sheet

- check Ninova regularly
- check your ITU e-mail account regularly
- "İTÜ Mobil" app on mobile devices

BLG 102E
Introduction to Scientific Computing and Engineering

WEEK 1

ISTANBUL TECHNICAL UNIVERSITY

Programming

- computer program: sequence of instructions to the computer

- takes inputs
- produces outputs

- programming: designing and implementing programs

Machine Code

- every processor has its own instruction set
- instructions are encoded as numbers

- running programs must be in this machine code

Machine Code Example

- decide whether someone is over 18 years or not:
    - get the value in memory address 0x0000002F (current year)
    - subtract the value in memory address 0x0000003E (birth year) from that
    - compare the result (age) with 18 (0x12)

```
A12F0000002B053E00000083F812
```

High-Level Languages

- it's very difficult to write programs in machine code
- write program in a "high-level language": source code
  - more abstract statements instead of primitive instructions
  - names instead of memory addresses
- use a program to convert source code to machine code

Platform Dependence

- machine code depends on the processor
- and also on the operating system

- but source code doesn't have to

- convertor generates machine code for a particular platform

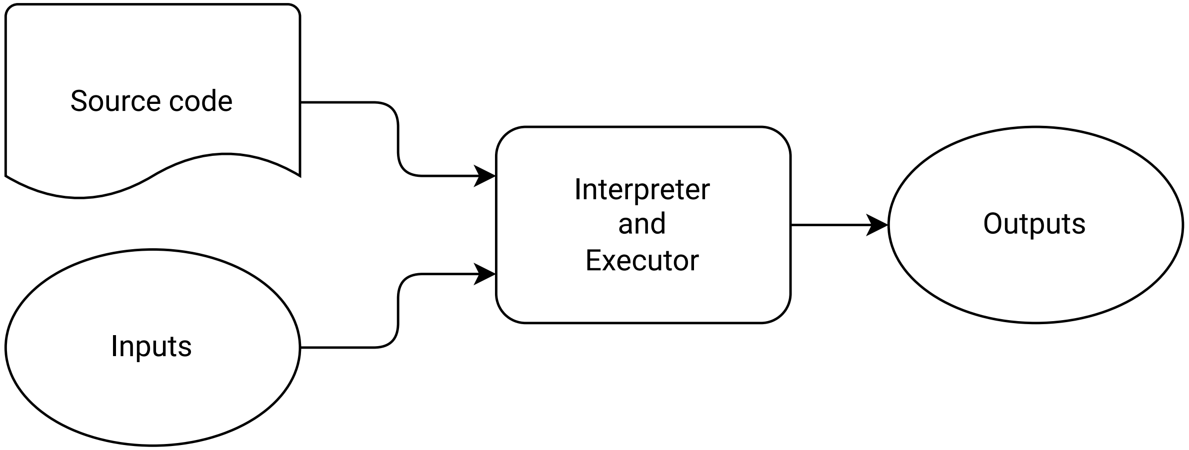
Portability

- same source code
- different convertors for different platforms

Conversion Methods

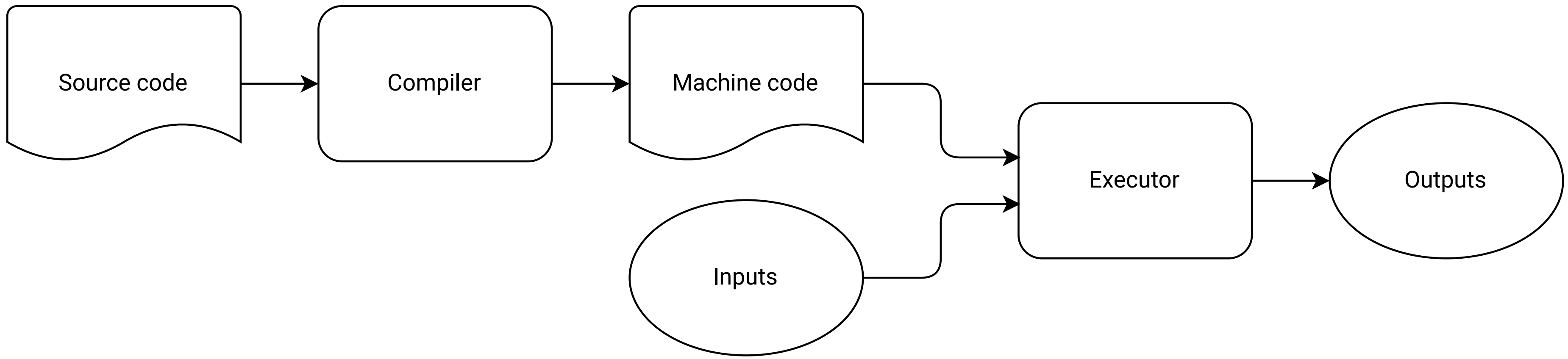
- interpreting
  - convert first statement
  - execute it
  - convert next statement
  - execute it
  - ...
- compiling
  - convert all statements
  - execute first statement
  - execute next statement
  - ...

Interpreting



- conversion during runtime

Compiling



- conversion during compile time

Interpreting vs Compiling

- compiled programs run faster
- compiled programs use less memory

- interpreted languages are more flexible
- development is easier in interpreted languages

Programmer's Workflow

- interpreted
  1. edit source code
  2. run the program and test it
  3. if incorrect behavior,

     go to step 1

- compiled
  1. edit source code
  2. compile to machine code
  3. if compilation errors,

     go to step 1
  4. run the program and test it
  5. if incorrect behavior,

     go to step 1

Interactive Coding

- REPL: Read, Eval, Print, Loop
- ask a question, get an answer

- show prompt, wait for input
- evaluate input
- print result

- show prompt, wait for input
- ...

Minimal Program

- a program that does nothing:

```
int main() {
    return 0;
}
```

Starting Point

```
int main() {
    return 0;
}
```

- program starts at `main`: entry point
- every program must have one and exactly one

Function

```
int main() {
    return 0;
}
```

- `main` is a function
- functions consist of statements
- statements enclosed in curly braces

Statement

```
int main() {
    return 0;
}
```

- statements end with a semicolon

Function Result

```
int main() {
    return 0;
}
```

- functions report their results using **`return`**

Program Result

```
int main() {
    return 0;
}
```

- result of `main` is the exit status:

  success (0), failure (1)
- `int` is the type of the result (integer)

Keywords

- some words in the language have special meaning: keywords
- `int`, `return`
- their use is restricted

Hello, world!

- a program that prints a message:

```
#include <stdio.h>


int main() {

    printf("Hello, world!\n");
    return 0;
}
```

Output

- use the `printf` function to print a message on the screen

```
#include <stdio.h>


int main() {

    printf("Hello, world!\n");
    return 0;
}
```

Libraries

- implementation for `printf` is not contained in our code
- commonly used functions are collected into libraries
- `printf` is part of the standard library

Header Files

- to use a function, include its header file

```
#include <stdio.h>


int main() {

    printf("Hello, world!\n");

    return 0;

}
```

Newline

- the **\n** character moves the cursor to the next line

```
#include <stdio.h>


int main() {


    printf("Hello, world!\n");

    return 0;

}
```

Comments

- it's helpful to explain the code
- for people who will read the code

- comments are ignored by language processors
- no effect during runtime

Line Comments

- anything from // to the end of the line

```
#include <stdio.h>  // needed for printf

int main() {

    printf("Hello, world!\n");
    return 0;
}
```

Multiline Comments

- anything between `/*` and `*/`
- can span over multiple lines

```
/* (C) H. Turgut Uyar
   Prints a message on the screen. */


#include <stdio.h>  // needed for printf
...
```

Code Style

- programmers follow style conventions
    - lowercase or uppercase letters in names
    - spaces for visual separation

- not mandatory rules
- make code easier to read

Different Styles

- different programmers have different preferences
- in a team, members should agree on style

Line Length

- lines shouldn't be too long
- requires horizontal scrolling

- popular value: 80
- can be increased in large monitors

Whitespace

- whitespace is insignificant:

```
int main(){printf("Hello, world!\n");return 0;}
```

- but this is not readable

Indentation

- statements should start with leading space
  - how much space?
  - which character to use: space or tab?

- use spaces, not tabs
- 4 spaces

Indentation Example

- statements are indented 4 spaces within function

```
int main() {
    printf("Hello, world!\n");



    return 0;
}
```

Indentation Bad Example

- statements are not indented

```
int main() {
printf("Hello, world!\n");



return 0;
}
```

Indentation Worse Example

- statements are inconsistently indented

```
int main() {
   printf("Hello, world!\n");



 return 0;
}
```

Function Braces

- where to put curly braces around function statements?

- opening brace
  - on the same line
  - on the next line
  - on the next line, indented
- closing brace
  - on the same line
  - on the next line
  - on the next line, dedented

Brace Style Example

- opening brace on the same line
- closing brace on the next line, dedented

```
int main() {
    printf("Hello, world!\\n");


    return 0;
}
```

Brace Style Alternative Example

- opening brace on the next line
- closing brace on the next line, dedented

```
int main()
{
    printf("Hello, world!\n");

    return 0;
}
```

Function Parentheses

- whether to put space around parentheses after function name

Function Parentheses Example

- no space before, one space after

```
int main() {
    printf("Hello, world!\n");



    return 0;
}
```

Function Parentheses Alternative Examples

- space before

```
int main () {
    ...
}
```

- no space after

```
int main(){
    ...
}
```

- these styles are not as popular

Blank Lines

- how many blank lines to separate components?

Blank Lines Example

- one blank line after **`#include`** lines

```
#include <stdio.h>


int main() {


    printf("Hello, world!\n");

    return 0;

}
```

Blank Lines Bad Example

- no blank line after **`#include`** lines

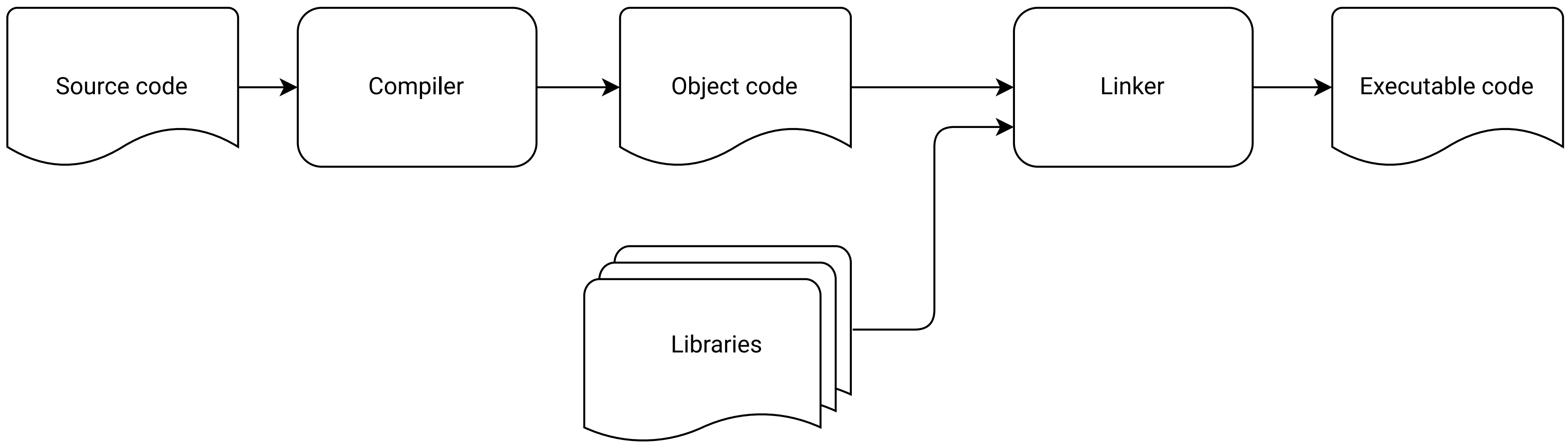
```
#include <stdio.h>
int main() {
    printf("Hello, world!\n");

    return 0;
}
```

Building Executables

- executables get built in two stages
  - compiling
  - linking

Build Stages



Build Problems

- error: no executable gets built

- warning: executable gets built
- but possibly won't work as intended

- don't ignore warnings

Syntax Errors

- violating the syntax rules of the language
  - forgetting semicolons
  - not closing parentheses
  - not closing quotes
- error at compile-time

Forgotten Semicolon

```
#include <stdio.h>

int main() {
    printf("Hello, world!\n")
    return 0;
}
```

- no semicolon at end of printing statement

Name Errors

- using undefined names
- warning at compile-time, error at link-time

Undefined Name

```
#include <stdio.h>


int main() {
    print("Hello, world!\n");
    return 0;
}
```

- `print` instead of `printf`

Case Sensitivity

- uppercase and lowercase are not the same
- can cause name issues

Incorrect Case

```
#include <stdio.h>

int Main() {
    printf("Hello, world!\n");
    return 0;
}
```

- `Main` instead of `main`
- no problem at compile-time,

  error at link-time

Algorithm

- algorithm: step by step description of a solution
- like a recipe
- independent from programming language

Algorithm Properties

- algorithm must be unambiguous
- precise instructions for each step

- algorithm must not run forever
- either find a solution, or report failure

Square Root

- finding the square root of a number

- start with an initial guess
- repeatedly improve guess
- until the guess is good enough

Variables

- number: x
- guess: g
- improved guess: g'

Improving Guess

- improved guess:

$$g' = \frac{g + \frac{x}{g}}{2}$$

Termination

- when is guess good enough?

$$g^2 \approx x$$

- must be precise:

$$|g^2 - x| < 10^{-3}$$

Square Root Algorithm

- initial guess: 1

1. $g = 1$
2. if $|g^2 - x| < 10^{-3}$ then g is the result, stop
3. $g' = \frac{g + \frac{x}{g}}{2}$
4. replace g with g' and go to step 2

Square Root Example

- find: $\sqrt{3}$
- guesses:

$$1$$

$$\frac{1+\frac{3}{1}}{2} = 2$$

$$\frac{2+\frac{3}{2}}{2} = 1.75$$

$$\frac{1.75+\frac{3}{1.75}}{2} \approx 1.732$$

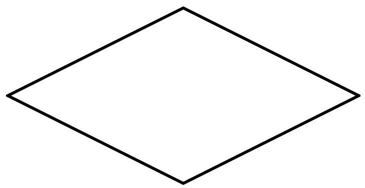
Flowchart

- algorithm diagram: flowchart

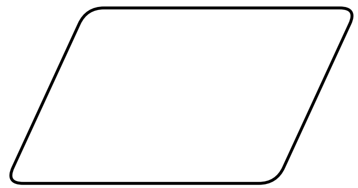
Shapes

- statement

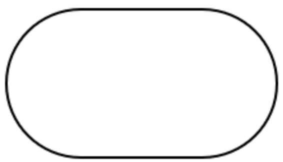
- decision



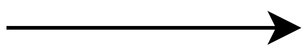
- input/output



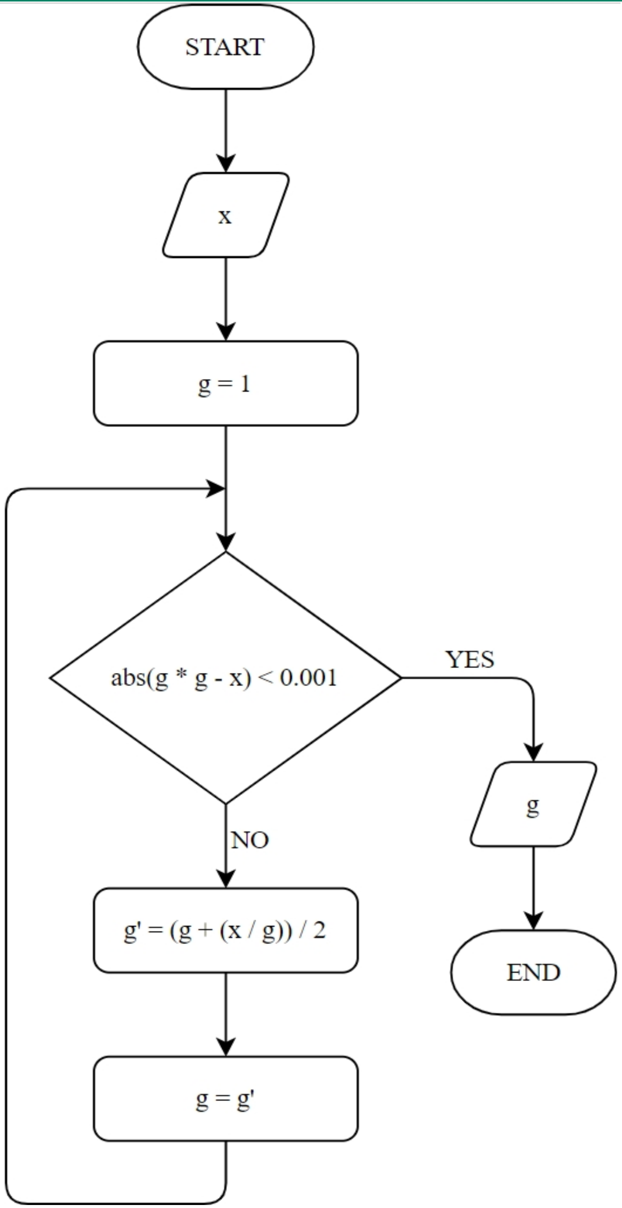
- start/end



- connector



Square Root Flowchart



- using a code-like notation

Example1 : Calculating Grade

Phase 1: Define the Problem

- Write a program to calculate a student’s passing grade and print on screen.

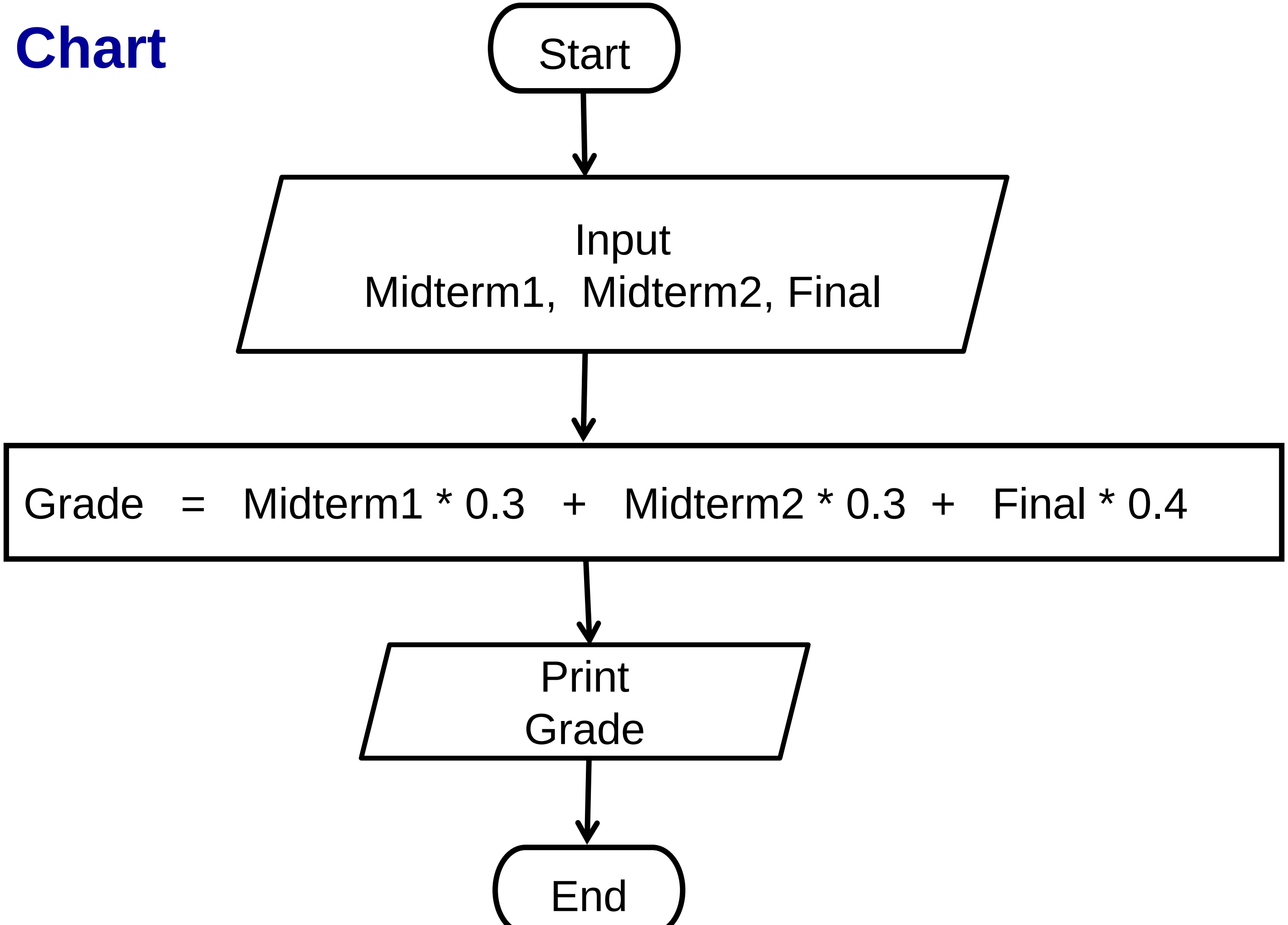
- **<u>INPUTS:</u>** Inputs are the numeric values:
  Midterm exam1, Midterm exam2, Final exam

- **<u>OUTPUT:</u>** Output is the numeric value of Grade.

- **<u>PROCESSING:</u>** Grade should be calculated with the following weights:
  - 30% of midterm1
  - 30% of midterm2
  - 40% of final

Phase 2: Design the Program

Flow Chart



Example2 : Calculating Factorial

Phase 1: Define the Problem

- Write a program to calculate the factorial of a number.

- **INPUT:** An integer number N.

- **OUTPUT:** Factorial of the N.

- **PROCESSING:** Factorial is computed as the following.

  N ! = 1 * 2 * 3 * 4 * ….. * N

Phase 2: Design the Program

Variables:

N : Number (loop limit)

i : Loop counter

fact : Factorial result