مبانی برنامه نویسی

Structured program development

Dr. Mehran Safayani

safayani@iut.ac.ir

safayani.iut.ac.ir

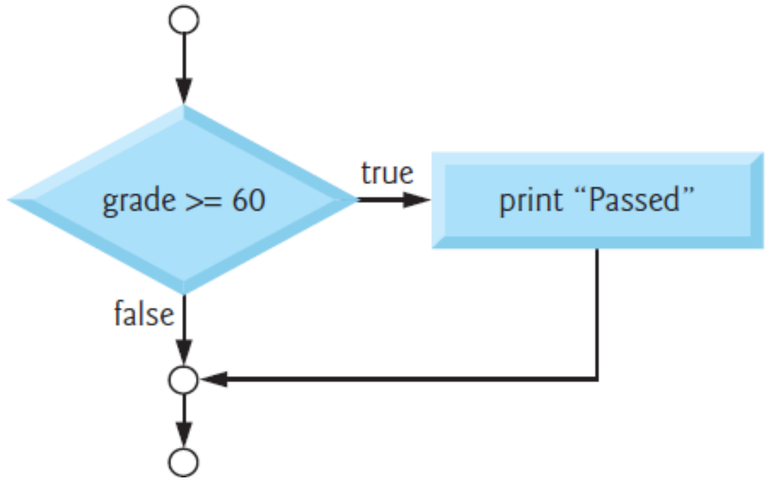
https://www.aparat.com/mehran.safayani

https://github.com/safayani/Programming_Basics_course

Department of Electrical and computer engineering, Isfahan university of technology, Isfahan, Iran

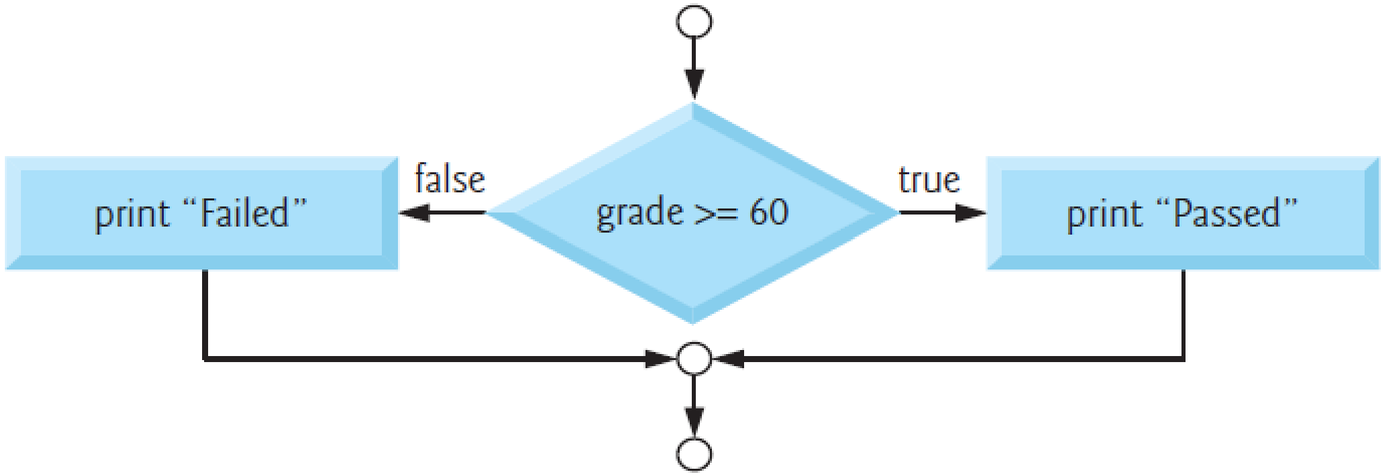
The if Selection Statement

```
if (grade >= 60) {
   puts("Passed");
} // end if
```



The if…else Selection Statement

```
if (grade >= 60) {
   puts("Passed");
} // end if
else {
   puts("Failed");
} // end else
```



conditional operator (?:)

```
puts((grade >= 60) ? "Passed" : "Failed");
```

- The **conditional operator** ?: is C's only ternary operator, taking three operands.
- Syntax: condition ? value_if_true : value_if_false
- It returns a value and can be used directly in expressions and function arguments.
- Best practice: Ensure the second and third operands are of the **same type** to avoid errors.
- It provides a compact alternative to simple if...else statements.

Nested if…else Statements

```
If student's grade is greater than or equal to 90
    Print "A"
else
    If student's grade is greater than or equal to 80
        Print "B"
    else
        If student's grade is greater than or equal to 70
            Print "C"
        else
            If student's grade is greater than or equal to 60
                Print "D"
            else
                Print "F"
```

```
if (grade >= 90) {
   puts("A");
} // end if
else {
   if (grade >= 80) {
      puts("B");
   } // end if
   else {
      if (grade >= 70) {
         puts("C");
      } // end if
      else {
         if (grade >= 60) {
            puts("D");
         } // end if
         else {
            puts("F");
         } // end else
      } // end else
   } // end else
} // end else
```

```
if (grade >= 90) {
   puts("A");
} // end if
else if (grade >= 80) {
   puts("B");
} // end else if
else if (grade >= 70) {
   puts("C");
} // end else if
else if (grade >= 60) {
   puts("D");
} // end else if
else {
   puts("F");
} // end else
```

Blocks and Compound Statements

- Always use **braces** { } for the bodies of if and else statements, even for **single statements**.
- This prevents logic errors, such as the **"dangling else"** problem, where an else is mistakenly associated with the wrong if.
- Using braces ensures the code's structure is clear and the logic is executed as intended.

-

Empty Statement

```
if (grade >= 60);
```

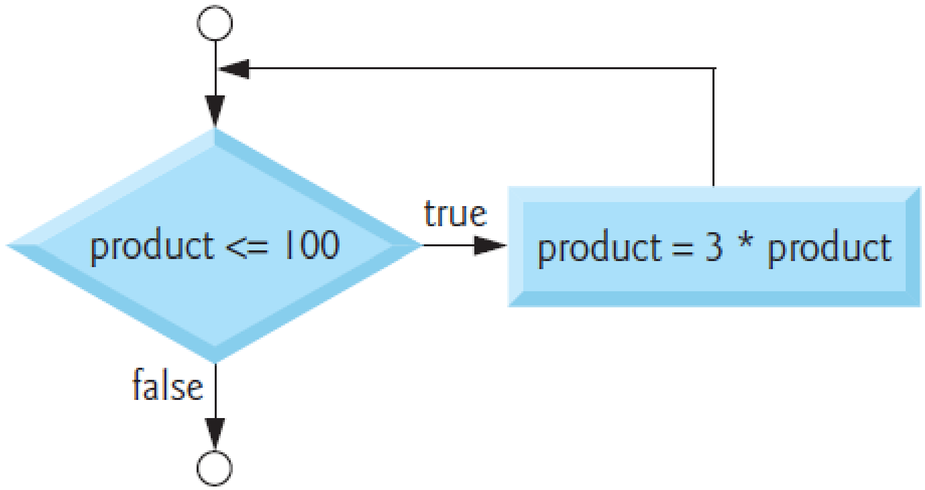
- leads to a logic error in single-selection if statements and a syntax error in double selection and nested if…else statements.

The while Iteration Statement

- **Calculating the First Power of 3 Greater Than 100**

```
int product = 3;

while (product <= 100) {
    product = 3 * product;
}
```



The while Iteration Statement

```
int product = 3;

while (product <= 100) {
    product = 3 * product;
}
```

- **Calculating the First Power of 3 Greater Than 100**
- The while **loop** repeats as long as its condition remains true.
- The loop's **body** (a single or compound statement) must modify a variable so the condition can eventually become false.
- If the condition never becomes false, an **infinite loop** occurs, which is a logic error.
- When the condition evaluates to false, the loop terminates, and the program continues with the next statement.

Counter- Controlled Iteration

- *A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.*

```c
// fig03_02.c
// Class average program with counter-controlled iteration.
#include <stdio.h>

// function main begins program execution
int main(void) {
   // initialization phase
   int total = 0; // initialize total of grades to 0
   int counter = 1; // number of the grade to be entered next

   // processing phase
   while (counter <= 10) { // loop 10 times
      printf("%s", "Enter grade: "); // prompt for input
      int grade = 0; // grade value
      scanf("%d", &grade); // read grade from user
      total = total + grade; // add grade to total
      counter = counter + 1; // increment counter
   } // end while

   // termination phase
   int average = total / 10; // integer division
   printf("Class average is %d\n", average); // display result
} // end function main
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Some refinements: **Sentinel-Controlled Iteration**

- *Develop a class-averaging program that will process an* arbitrary *number of grades each time the program is run.*
- **Sentinel Values:** a **sentinel value** to indicate “end of data entry.”
- Proper message in case of zero grade

```c
int main(void) {
   // initialization phase
   int total = 0; // initialize total
   int counter = 0; // initialize loop counter

   // processing phase
   // get first grade from user
   printf("%s", "Enter grade, -1 to end: "); // prompt for input
   int grade = 0; // grade value
   scanf("%d", &grade); // read grade from user

   // loop while sentinel value not yet read from user
   while (grade != -1) {
      total = total + grade; // add grade to total
      counter = counter + 1; // increment counter

      // get next grade from user
      printf("%s", "Enter grade, -1 to end: "); // prompt for input
      scanf("%d", &grade); // read next grade
   } // end while

   // termination phase
   // if user entered at least one grade
   if (counter != 0) {

      // calculate average of all grades entered
      double average = (double) total / counter; // avoid truncation

      // display average with two digits of precision
      printf("Class average is %.2f\n", average);
   } // end if
   else { // if no grades were entered, output message
      puts("No grades were entered");
   } // end else
} // end function main
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

```
Enter grade, -1 to end: -1
No grades were entered
```

Always Use Braces in a while Statement

```
while (grade != -1)
   total = total + grade; // add grade to total
counter = counter + 1; // increment counter

// get next grade from user
printf("%s", "Enter grade, -1 to end: "); // prompt for input
scanf("%d", &grade); // read next grade
```

ERR ⊗ This would cause an infinite loop if the user did not input -1 as the first grade.

Type Conversion for Accurate Division

- **Problem:** Integer division (int / int) truncates the fractional part.
- **Solution:** Use an **explicit cast** to convert operands:
  - (double) total creates a temporary floating-point value
- **Result:**
  - (double) total / counter forces **floating-point division**
  - The compiler performs **implicit conversion** on counter
- **Key Point:** Cast operator is a unary operator with high precedence, ensuring accurate arithmetic results. This precedence is one level higher than that of the multiplicative operators *, / and %.

Formatting and Considerations for Floating-Point Numbers

- **Precision Control:** Use %.nf in printf to specify decimal places (e.g., %.2f for two digits).
- **Rounding:** The displayed value is rounded, but the value in memory remains unchanged.
- **Inherent Imprecision:** Floating-point numbers ar approximations due to fixed memory allocation.
- **Key Implications:**
  - Avoid using == to compare floating-point numbers for equality.
  - They are suitable for most applications where exact precision is not critical.

```c
#include <math.h>
#define TOLERANCE 0.00001

if (fabs(number1 - number2) < TOLERANCE) {
```

- *A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, 10 of the students who completed this course took the licensing examination. Naturally, the college wants to know how well its students did on the exam. You've been asked to write a program to summarize the results. You've been given a list of these 10 students. Next to each name is a 1 if the student passed the exam or a 2 if the student failed. Your program should analyze the results of the exam as follows:*

- *1. Input each test result (i.e., a 1 or a 2). Display the prompting message "Enter result" each time the program requests another test result.*
- *2. Count the number of test results of each type.*
- *3. Display a summary of the test results indicating the number of students who passed and the number who failed.*
- *4. If more than eight students passed the exam, print the message "Bonus to instructor!"*

Pseudocode for examination-results problem

```
1  Initialize passes to zero
2  Initialize failures to zero
3  Initialize student to one
4
5  While student counter is less than or equal to ten
6      Input the next exam result
7
8      If the student passed
9          Add one to passes
10     else
11         Add one to failures
12
13     Add one to student counter
14
15 Print the number of passes
16 Print the number of failures
17 If more than eight students passed
18     Print "Bonus to instructor!"
```

```c
int main(void) {
   // initialize variables in definitions
   int passes = 0;
   int failures = 0;
   int student = 1;

   // process 10 students using counter-controlled loop
   while (student <= 10) {
      // prompt user for input and obtain value from user
      printf("%s", "Enter result (1=pass,2=fail): ");
      int result = 0; // one exam result
      scanf("%d", &result);

      // if result 1, increment passes
      if (result == 1) {
         passes = passes + 1;
      } // end if
      else { // otherwise, increment failures
         failures = failures + 1;
      } // end else

      student = student + 1; // increment student counter
   } // end while

   // termination phase; display number of passes and fail
   printf("Passed %d\n", passes);
   printf("Failed %d\n", failures);

   // if more than eight students passed, print "Bonus to
   if (passes > 8) {
      puts("Bonus to instructor!");
   } // end if
} // end function main
```

```
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 2
Enter Result (1=pass, 2=fail): 2
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 2
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 2
Passed 6
Failed 4
```

```
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 2
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Enter Result (1=pass, 2=fail): 1
Passed 9
Failed 1
Bonus to instructor!
```

Fig. 3.6 | Analysis of examination results. (Part 2 of 2.)

Assignment Operators

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Increment and Decrement Operators

Operator	Sample expression	Explanation
++	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

```c
int main(void) {
   // demonstrate postincrement
   int c = 5; // assign 5 to c
   printf("%d\n", c); // print 5
   printf("%d\n", c++); // print 5 then postincrement
   printf("%d\n\n", c); // print 6

   // demonstrate preincrement
   c = 5; // assign 5 to c
   printf("%d\n", c); // print 5
   printf("%d\n", ++c); // preincrement then print 6
   printf("%d\n", c); // print 6
} // end function main
```

```
5
5
6

5
6
6
```

Fig. 3.7 | Preincrementing and postincrementing. (Part 2 of 2.)

Operators	Grouping	Type
++ *(postfix)* -- *(postfix)*	right to left	postfix
+ - *(type)* ++ *(prefix)* -- *(prefix)*	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

- a = b = c = 5;
- num = 5; a = num++;
- x=1;y=2; x += y += 5; // x=8 y=7