Program Control Structures

Miguel A. Figueroa-Villanueva

Department of Electrical and Computer Engineering
University of Puerto Rico - Mayagüez Campus

September 2, 2009

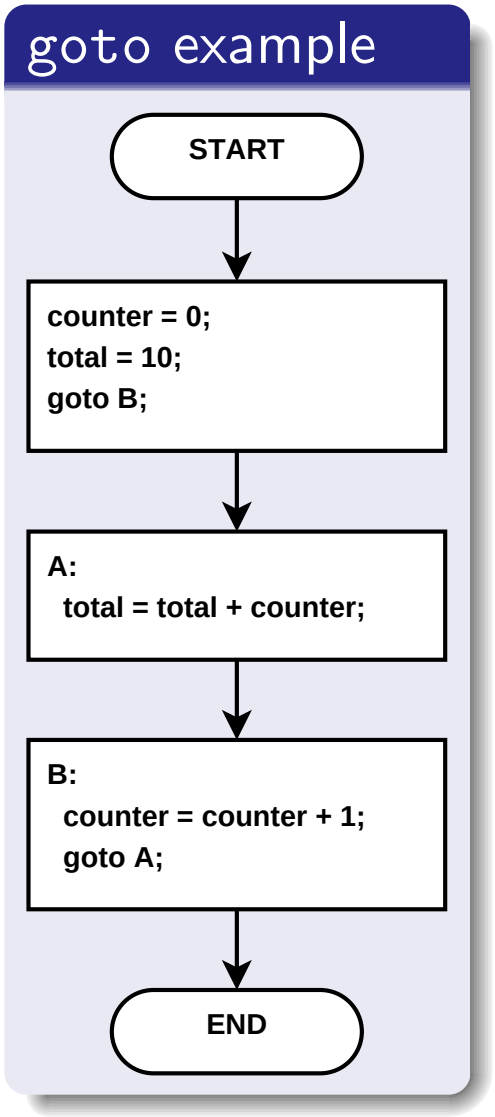
Outline

Program Control Structures

- *Bohm and Jacopini (1966)* work led to demonstrate that all algorithms can be be expressed in a structured program with only three **control structures**:
  - the sequence structure,
  - the selection structure,
  - and the repetition structure.
- These structures will be represented using flowcharts with two connectors; one entry and one exit connector.
- Programs will be built by creating multiple combinations of these **single-entry/single-exit control statements**.
- These modules can be combined in two forms:
  - **control-statement stacking**,
  - and **control-statement nesting**.
- That is, all programs will be built using only three types of control statements combined in only two ways.

Sequence Structure

- Built into structured languages, such as C.
  - That is, no special keyword required.
- Program instructions are executed sequentially, in the order they appear. This is important for program clarity.
- The *infamous* goto statement led to many problems in the past due to its arbitrary **transfer of control**.
- What value will the variable total have at the end of the execution of the program in the Figure?



Selection Structure

- The selection structure is used to choose from alternative courses of action.
- Also called conditional or branching structure.
- Selection structures can be classified into three categories:
  - single-selection statement
  - double-selection statement
  - multiple-selection statement
- C has three types: `if` , `if...else` , and `switch`.

`if` - single-selection statement

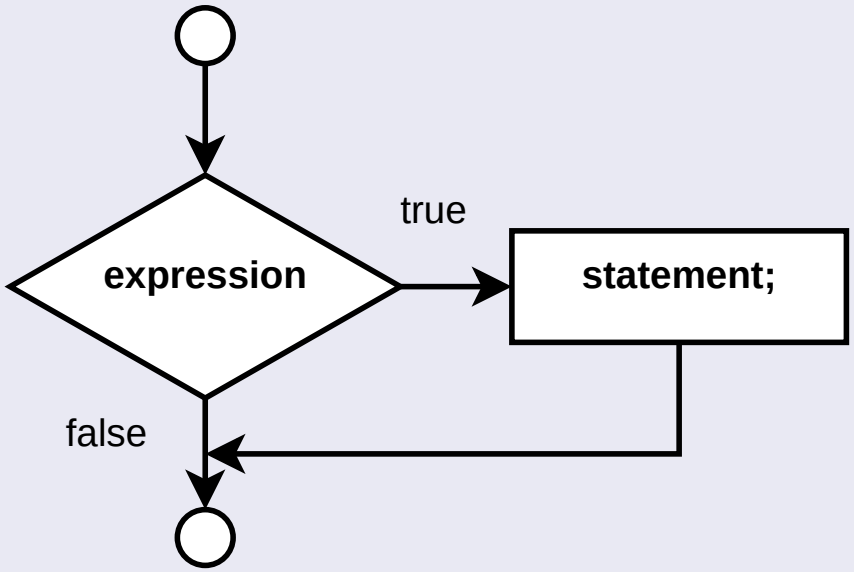
definition

Structure that will execute the eclosed *statement* or *set of statements* if and only if *expression* is `true`.

pseudocode

```
if (expression)
{
  statement;
}
```

flowchart



`if` - single-selection statement

example

problem

Print *PASSED* if grade is greater or equal to 65.

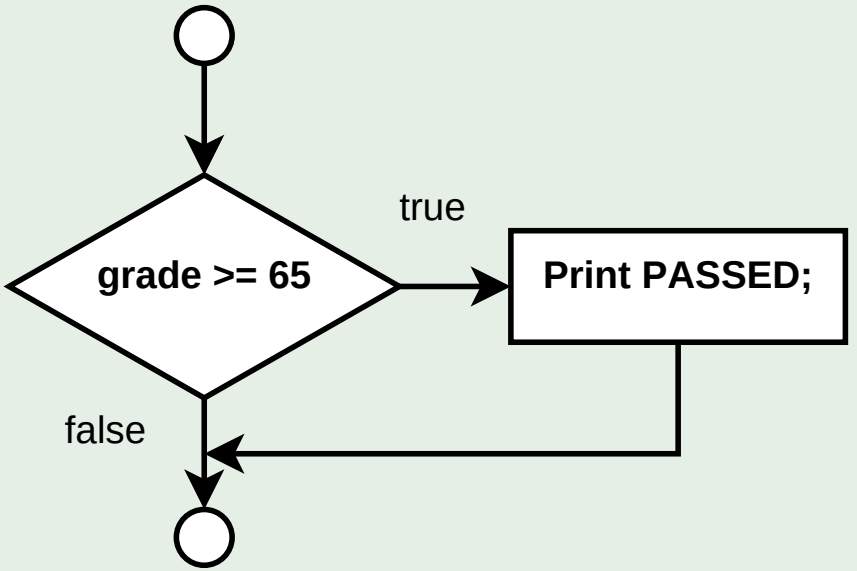
if - single-selection statement

solution

pseudocode

```
if (grade >= 65)
{
   printf("PASSED");
}
```

flowchart



`if..else` - double-selection statement

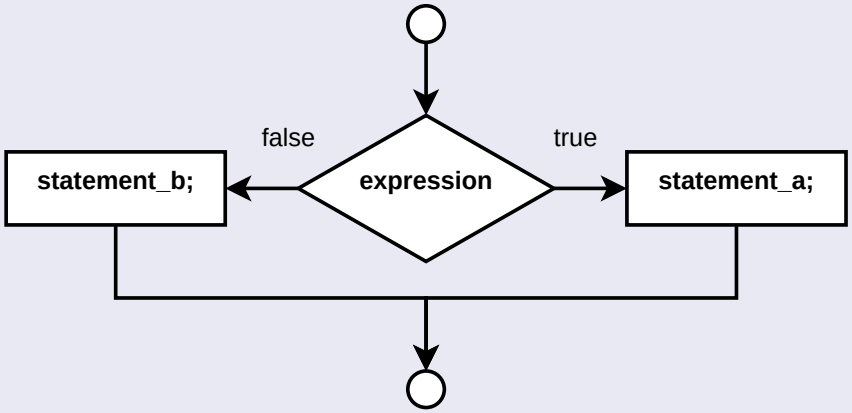
definition

Structure that will execute the eclosed *statement_a* if *expression* is `true`, otherwise it will execute the eclosed *statement_b*.

pseudocode

```
if (expression)
{
  statement_a;
}
else
{
  statement_b;
}
```

flowchart



`if..else` - double-selection statement

example

problem

Print *PASSED* if grade is greater or equal to 65, otherwise print *FAILED*.

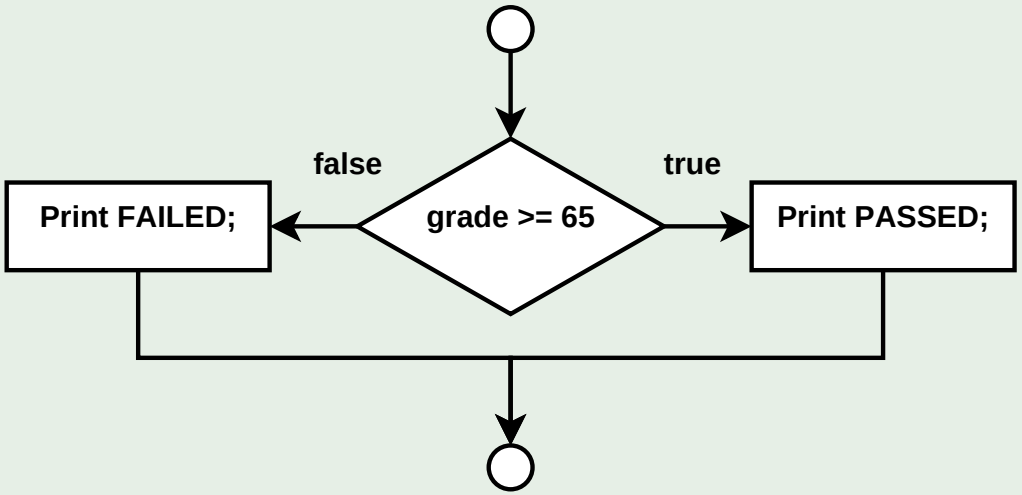
`if..else` - double-selection statement

solution

pseudocode

```
if (grade >= 65)
{
   printf("PASSED");
}
else
{
   printf("FAILED");
}
```

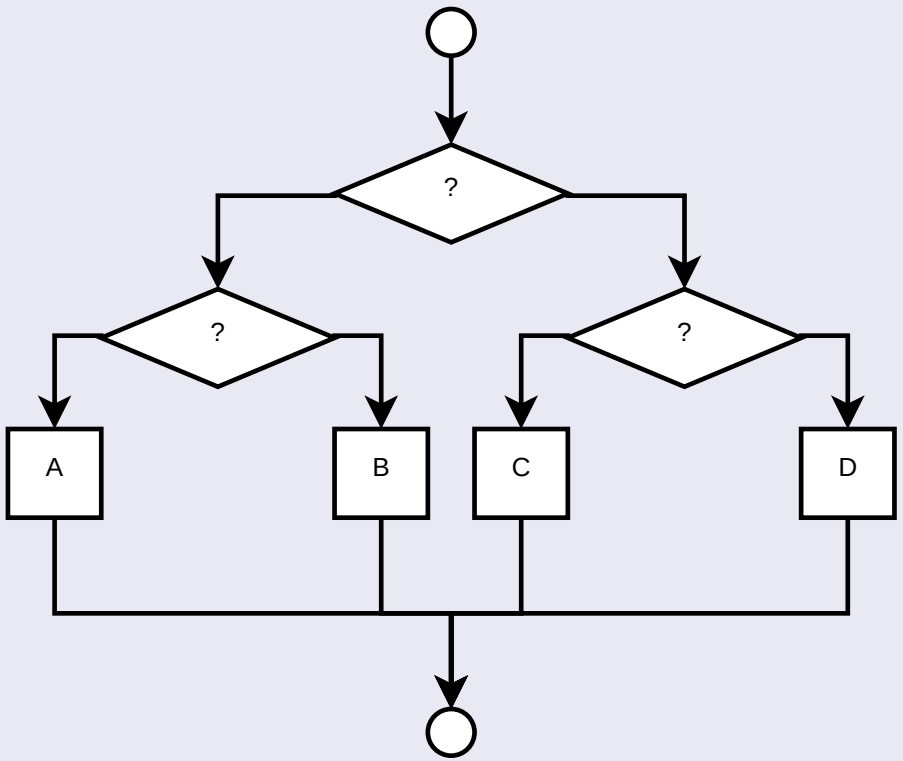
flowchart



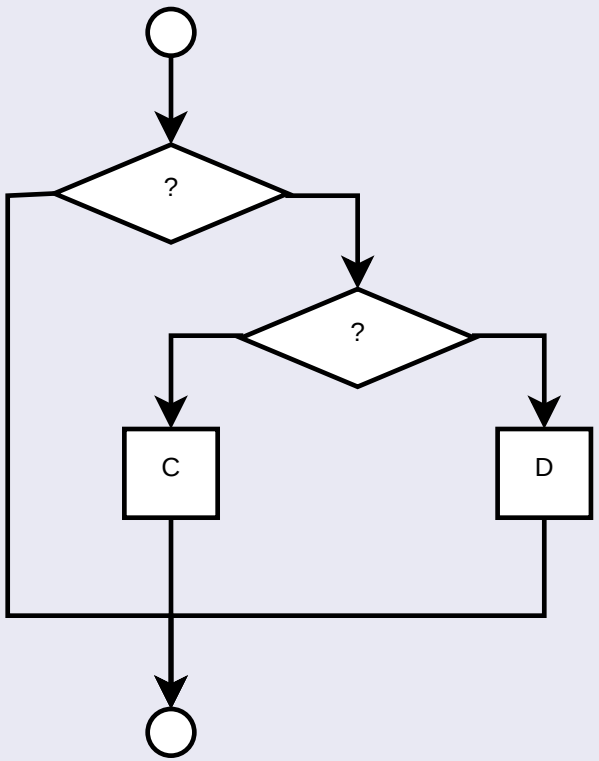
`if..else` - multiple-selection statement

- It is also possible to create a multiple-selection structure using **nested** `if..else` structures.

example



example



`if..else` - multiple-selection statement

example

problem

Print the grade letter of a student based on the point percentage.

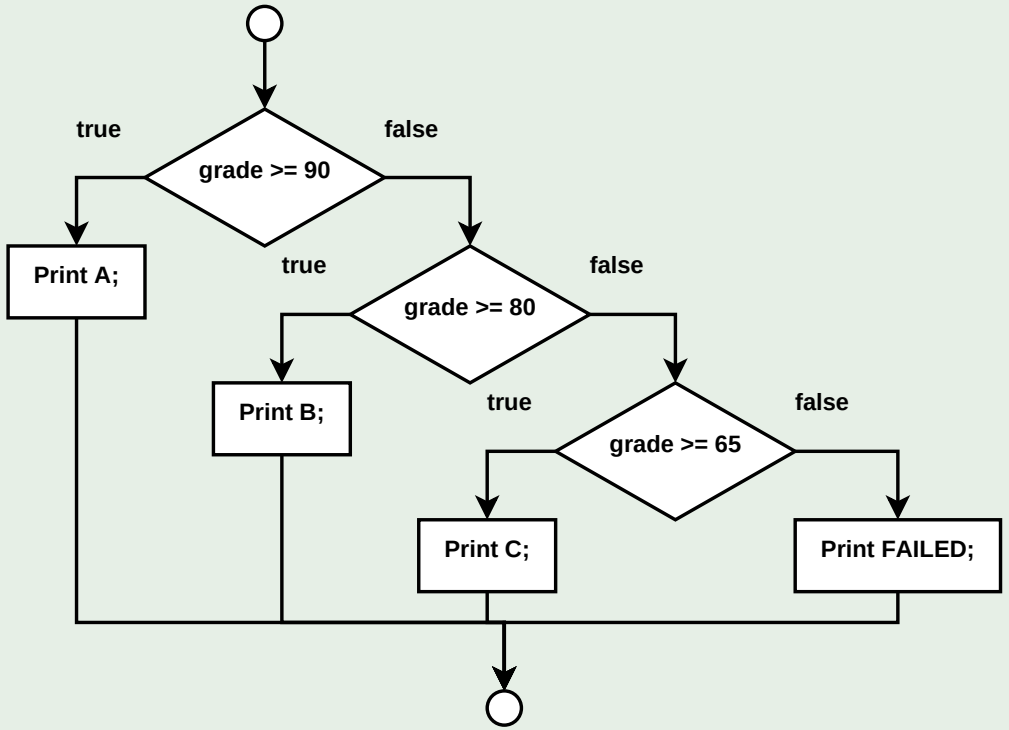
if..else - multiple-selection statement

solution

pseudocode

```
if (grade >= 90)
{
   printf("A");
}
else
{
   if (grade >= 80)
   {
      printf("B");
   }
   else
   {
      if (grade >= 65)
      {
         printf("C");
      }
      else
      {
         printf("F");
      }
   }
}
```

flowchart



if..else - multiple-selection statement

- The following structure can be used to organize the multiple-selection structure in the previous example.
- These are equivalent for the compiler, however it is more readable and compact.

solution (preferred)

```
if (grade >= 90)
{
   printf("A");
}
else if (grade >= 80)
{
   printf("B");
}
else if (grade >= 65)
{
   printf("C");
}
else
{
   printf("F");
}
```

Repetition Structure

- The repetition structure is used to repeat a series of instructions multiple times.
- Also called iterative structure or loops.
- There are two common iterative structures:
  - counter-controlled repetition
  - sentinel-controlled repetition
- C has three types: `while` , `do...while` , and `for`.

example

```
while (there are more items in my shopping list)
{
  purchase next item and cross it off my list;
}
```

`while` - repetition statement

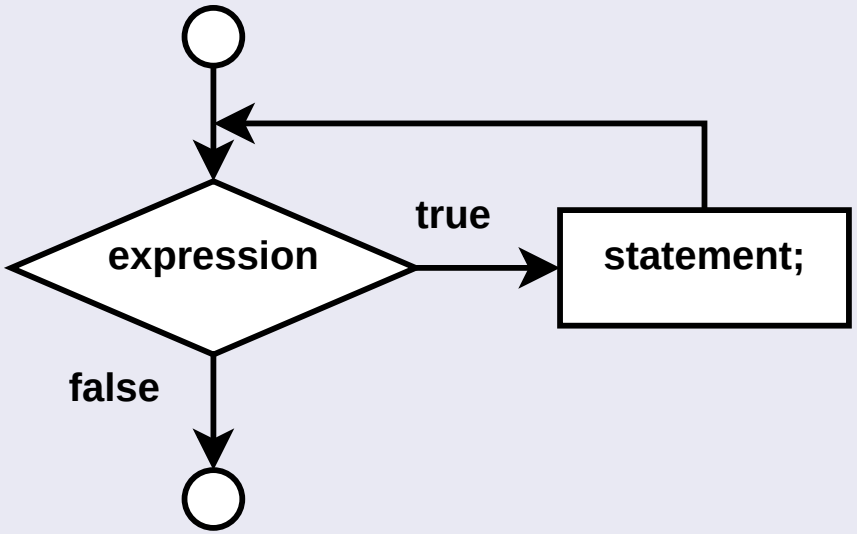
definition

Structure that will execute the eclosed *statement* **while** *expression* is `true`. When the *expression* is false it will stop the execution of the enclosed *statement*.

pseudocode

```
while (expression)
{
  statement;
}
```

flowchart



`while` - repetition statement

example

problem

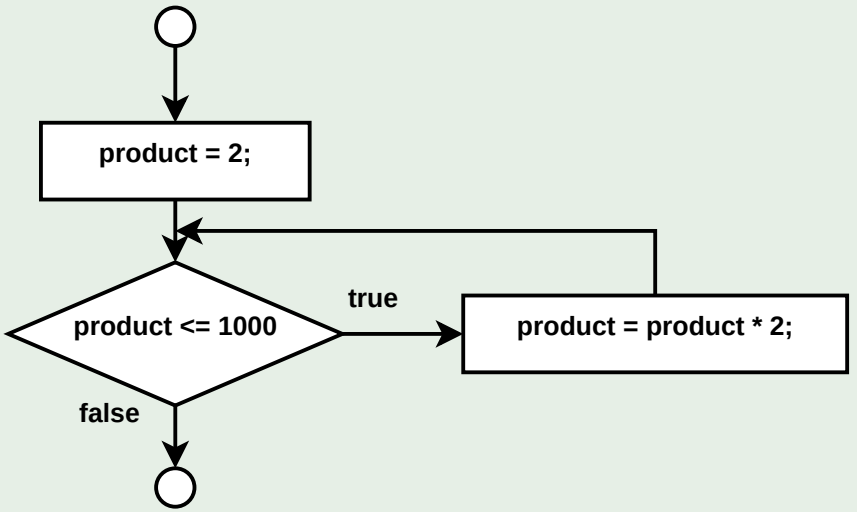
Determine the first power of 2 larger than 1000.

while - repetition statement
solution

pseudocode

```
product = 2;
while (product <= 1000)
{
   product = product * 2;
}
```

flowchart



`while` - counter-controlled repetition

- Loop repeated until a `counter` variable reaches a certain value.
- If we need n iterations, the `counter` will count n times.

`while` - counter-controlled repetition

example

problem

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.

while - counter-controlled repetition

solution

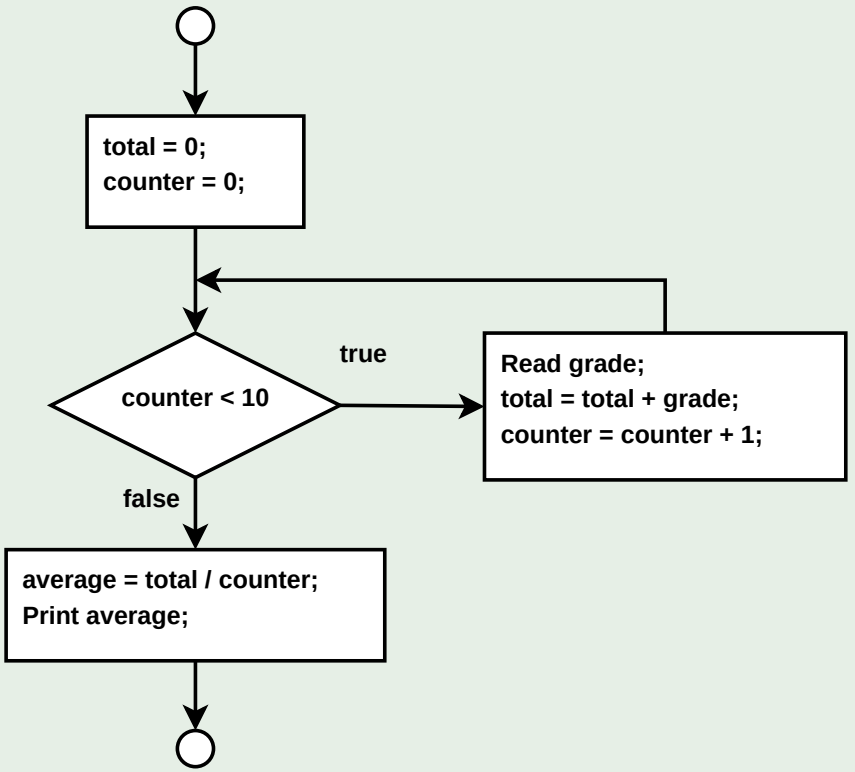
pseudocode

```
total = 0;
counter = 0;

while (counter < 10)
{
  Read grade;
  total = total + grade;
  counter = counter + 1;
}

average = total / counter;
Print average;
```

flowchart



while - counter-controlled repetition

manual variable inspection

Create a table to inspect the algorithm functionality for a given instance. This will help verify whether your algorithm is correct.

pseudocode

```
total = 0;
counter = 0;

while (counter < 10)
{
  Read grade;
  total = total + grade;
  counter = counter + 1;
}

average = total / counter;
Print average;
```

table

For the instance where we have the following grades: 98, 76, 71, 87, 83, 90, 57, 79, 82, 94.

iteration	counter	grade	total
0	0	-	0
1	1	98	98
2	2	76	174
3	3	71	245
4	4	87	332
5	5	83	415
6	6	90	505
7	7	57	562
8	8	79	641
9	9	82	723
10	10	94	**817**

average = 817 / 10 = 81.7

`while` - sentinel-controlled repetition

- Loop repeated until a variable is given a **sentinel value**.
- The sentinel value is a special value that indicates the *end of data entry*.
- Also known as, **signal value** or **flag value**.
- The sentinel-controlled repetition is used when we need to iterate for an indefinite amount of times (i.e., **the number of iterations are not known before the loop begins execution**).

`while` - sentinel-controlled repetition

example

problem

Develop a class averaging program that will process an arbitrary number of grades each time the program is run.

while - sentinel-controlled repetition

solution

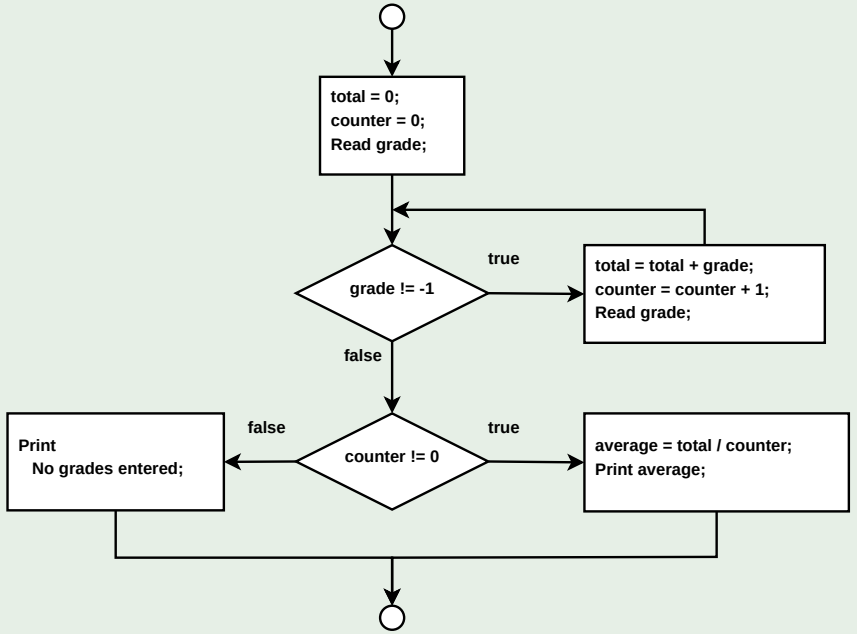
pseudocode

```
total = 0;
counter = 0;

Read grade;
while (grade not equal -1)
{
   total = total + grade;
   counter = counter + 1;
   Read grade;
}

if (counter not equal 0)
{
   average = total / counter;
   Print average;
else
{
   Print "No grades entered."
}
```

flowchart



while - sentinel-controlled repetition

manual variable inspection

pseudocode

```
total = 0;
counter = 0;

Read grade;
while (grade not equal -1)
{
   total = total + grade;
   counter = counter + 1;
   Read grade;
}

if (counter not equal 0)
{
   average = total / counter;
   Print average;
else
{
   Print "No grades entered."
}
```

table

For the instance where we have the following grades: 98, 76, 71, 87.

iteration	counter	grade	total
0	0	98	0
1	1	76	98
2	2	71	174
3	3	87	245
4	4	-1	**332**

average = 332 / 4 = 83

Top-Down, Stepwise Refinement

- Approach algorithm design from the **top-down** (i.e., *from the general to the specific*).
- Design in several steps, with each step being a **refinement** of the previous one.

Top-Down, Stepwise Refinement

example

problem

- A college has a list of test results (1 = pass, 2 = fail) for 10 students.
- Write a program that analyzes the results (i.e., prints the number of passes and failures) and if the passes are greater than 8, print "Raise Tuition".

Top-Down, Stepwise Refinement

solution

specifications

- The program must process 10 test results, hence a *counter-controlled loop* is required.
- Two variables can be used to count **passes** and **failures**, respectively.
- Each input result is a number, either a 1 or a 2.

Top-Down, Stepwise Refinement

solution

top level outline

Analyze exam results and decide if tuition should be raised.

first refinement

- Initialize variables.
- Input the ten quiz grades and count passes and failures.
- Print a summary of the exam results and decide if tuition should be raised.

Top-Down, Stepwise Refinement
solution

refinement - initialize variables

```
passes = 0;
failures = 0;
counter = 0;
```

refinement - input the ten quiz grades and count passes and failures

```
while (counter < 10)
{
  Read grade;
  if (student passed)
  {
    passes = passes + 1;
  }
  else
  {
    failures = failures + 1;
  }
  counter = counter + 1;
}
```

Top-Down, Stepwise Refinement

solution

refinement - print a summary of the exam results and decide if tuition should be raised

```
Print passes;
Print failures;
if (passes > 8)
{
  Print "Raise Tuition";
}
```

Top-Down, Stepwise Refinement

solution

final solution

```
1   passes = 0;
2   failures = 0;
3   counter = 0;
4
5   while (counter < 10)
6   {
7     Read grade;
8     if (student passed)
9     {
10      passes = passes + 1;
11    }
12    else
13    {
14      failures = failures + 1;
15    }
16    counter = counter + 1;
17  }
```

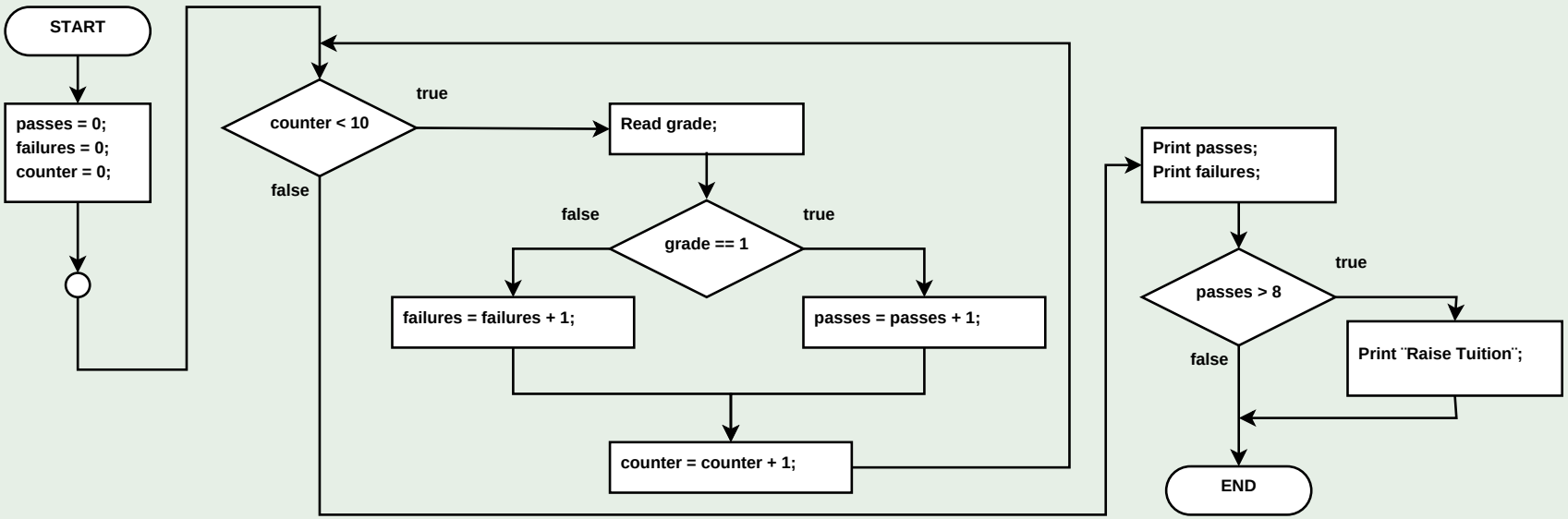
final solution (continued)

```
18  Print passes;
19  Print failures;
20  if (passes > 8)
21  {
22    Print "Raise Tuition";
23  }
```

Top-Down, Stepwise Refinement

solution

flowchart



Top-Down, Stepwise Refinement

example

problem

- Develop a program that receives a value `n` from the user and prints a cube of asterisks of dimension `n`.
- For example, if the user enters a value of $n = 5$, the program prints:

```
*****
*   *
*   *
*   *
*****
```

Top-Down, Stepwise Refinement
solution

top level outline

- Get input `n` from user.
- Print asterisk square of dimension `n`.

refinement - print asterisk square of dimension `n`

- Print `n` asterisks in a row and a newline.
- Print `n`−2 middle lines (i.e., * *).
- Print `n` asterisks in a row and a newline[a].

[a] Note that this is the same that the first line; reuse the functionality.

Top-Down, Stepwise Refinement
solution

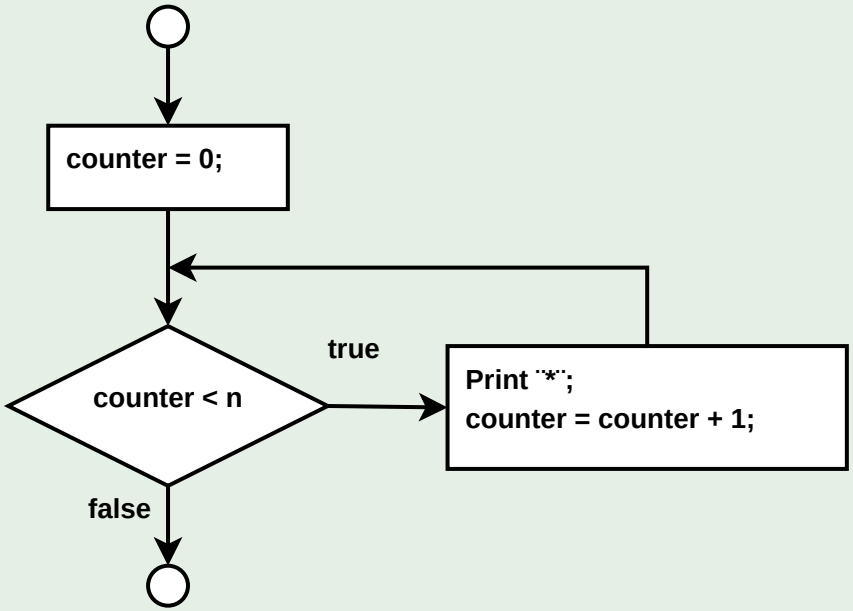
refinement - print n asterisks in a row and a newline

- Print n asterisks in a row.
- Print newline.

refinement - print n asterisks in a row

This requires a counter-controlled loop.

```
counter = 0;
while (counter < n)
{
   Print "*";
   counter = counter + 1;
}
```



Top-Down, Stepwise Refinement
solution

refinement - print `n-2` middle lines

- Repeat the following `n-2` times:
  - Print asterisk.
  - Print `n-2` spaces.
  - Print asterisk.
  - Print newline.

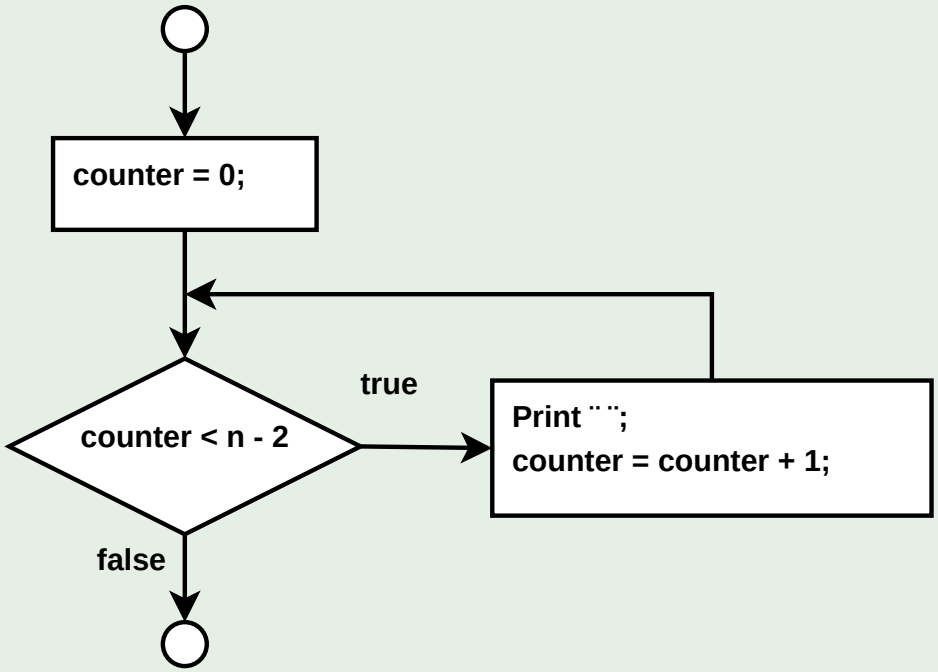
Top-Down, Stepwise Refinement

solution

refinement - print n−2 spaces

This requires a counter-controlled loop similar to the previous one.

```
counter = 0;
while (counter < n - 2)
{
   Print " ";
   counter = counter + 1;
}
```



Top-Down, Stepwise Refinement

solution

refinement - print n−2 middle lines

This requires a counter-controlled loop as well[a].

```
counter2 = 0;
while (counter2 < n - 2)
{
  // insert previous refinement
  counter2 = counter2 + 1;
}
```

[a] Why counter2 and not counter?

Top-Down, Stepwise Refinement

solution - putting it all together

final solution

```
1   Read n;
2
3   counter = 0;
4   while (counter < n)
5   {
6     Print "*";
7     counter = counter + 1;
8   }
9   Print "\n";
10
11  counter2 = 0;
12  while (counter2 < n - 2)
13  {
14    Print "*";
15    counter = 0;
```

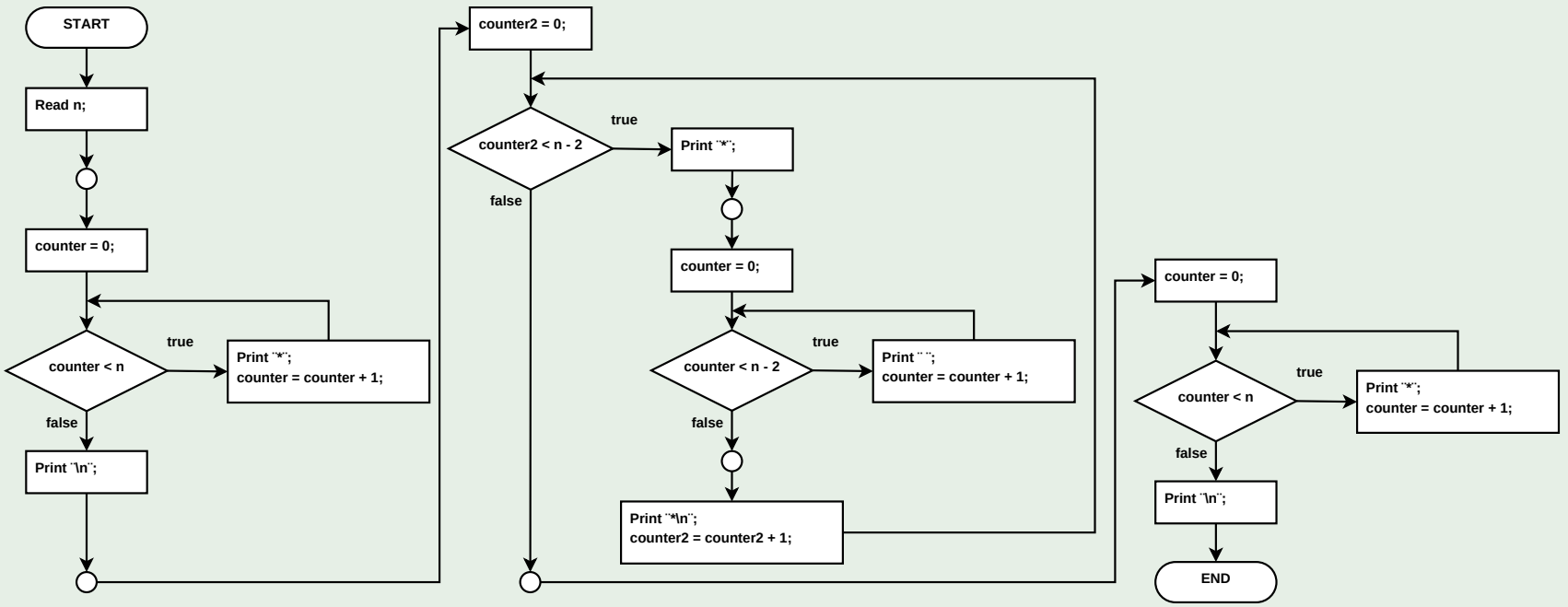
final solution (continued)

```
16    while (counter < n - 2)
17    {
18      Print " ";
19      counter = counter + 1;
20    }
21    Print "*\n";
22
23    counter2 = counter2 + 1;
24  }
25
26  counter = 0;
27  while (counter < n)
28  {
29    Print "*";
30    counter = counter + 1;
31  }
32  Print "\n";
```

Top-Down, Stepwise Refinement

solution - putting it all together

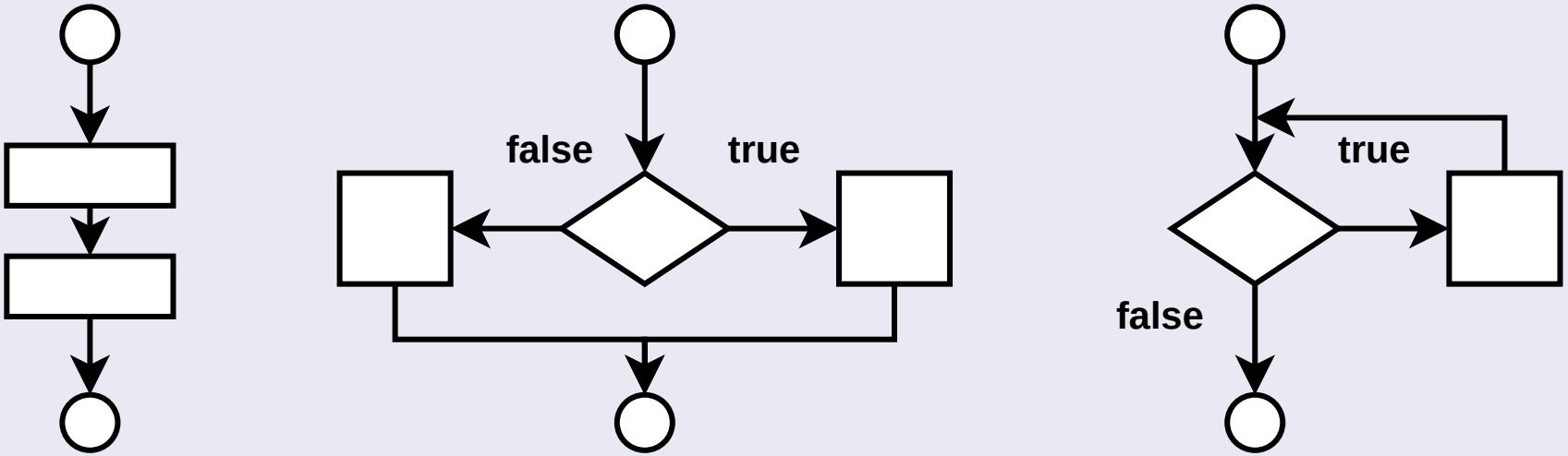
flowchart



Summary

- All structured programs can be expressed with only three control structures:
  - the sequence structure,
  - the selection structure,
  - and the repetition structure.
- These **single-entry/single-exit** control structure modules can be combined in two forms: **stacking** and **nesting**.

control structures



Summary

- Approach algorithm design from the **top-down** (i.e., *from the general to the specific*).
- Design in several steps, with each step being a **refinement** of the previous one.
- After a solution for a component is worked out, think of it as a **black box** and **reuse it**.