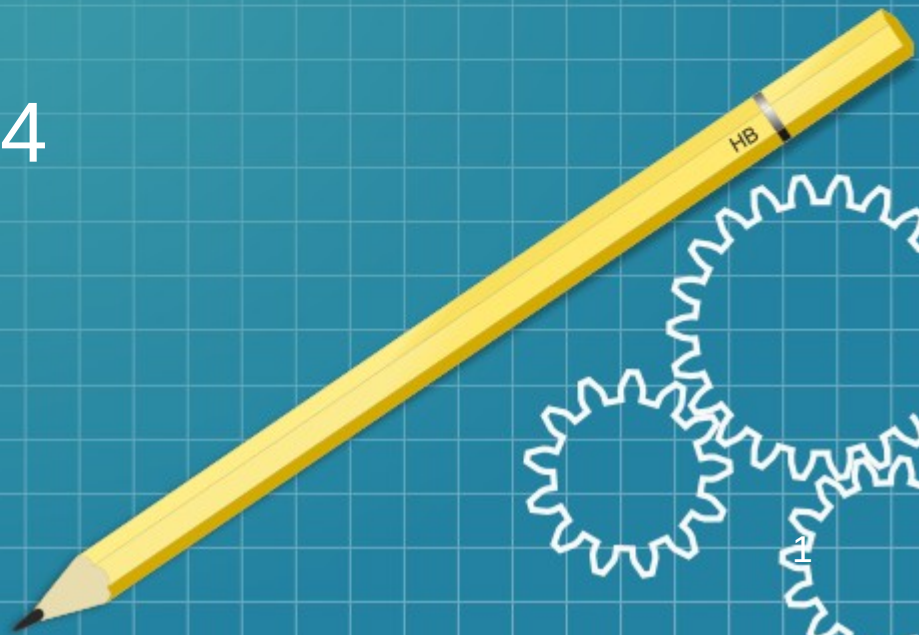
Algorithm Development and Control Statements: Part 1

Chapter 4



We will discuss:

- Algorithms
- Pseudocode
- Control statements (if, if-else)
- Nested control statements
- while loops
- Type conversions
- Compound assignments
- Top-down stepwise refinements

Algorithms

- What is an *algorithm*?

Algorithms

- What is an *algorithm*?

A *step-by-step procedure* to solve a problem

Algorithms

- What is an *algorithm*?

A *step-by-step procedure* to solve a problem

(from the textbook): A *procedure* for solving a problem in terms of

1. the *actions* to execute and
2. the *order* in which these actions execute

Pseudocode

- What is a pseudocode?

Pseudocode

- What is a pseudocode?

It is an *informal language* (a mix of English and a computer language) that helps us to develop an algorithm without having to worry about the strict details of a particular programming language.

Pseudocode

- What is a pseudocode?

It is an *informal language* (a mix of English and a computer language) that helps us to develop an algorithm without having to worry about the strict details of a particular programming language.

- not an actual computer programming language
- does not execute on computers
- mainly describes statements representing the *actions*.

Pseudocode

- What is a pseudocode?

It is an *informal language* (a mix of English and a computer language) that helps us to develop an algorithm without having to worry about the strict details of a particular programming language.

Examples:

```
input a, b:integers
m = max(a,b)
output m
```

Pseudocode

- What is a pseudocode?

It is an *informal language* (a mix of English and a computer language) that helps us to develop an algorithm without having to worry about the strict details of a particular programming language.

Examples:

```
input a, b:integers
m = max(a,b)
output m
```

```
integer max(integer a, integer b):
  if a > b then return a
else: return b
```

Control Structures

Some historical data (from the textbook):
When statements in a program are executed *one after another*, *in the order in which they are written*, the process is called *sequential execution*.

Control Structures

Some historical data (from the textbook):
When statements in a program are executed *one after another*, *in the order in which they are written*, the process is called *sequential execution*.

The `goto` statements, provided by many programming languages before, and used frequently by programmers until 1970s, were allowing the *transfer of control* from one statement to “any” statement in the code, not necessary the next one in sequence.

Control Structures

Some historical data (from the textbook):
When statements in a program are executed *one after another*, *in the order in which they are written*, the process is called *sequential execution*.

The `goto` statements, provided by many programming languages before, and used frequently by programmers until 1970s, were allowing the *transfer of control* from one statement to "any" statement in the code, not necessary the next one in sequence.

During the 1960s, it was noted that indiscriminate (i.e. random, without careful consideration) use of transfers of control was the root of much difficulty experienced by software development teams. `goto` statements were blamed for that.

Control Structures

The research of Bohm and Jacopini ("*Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules*",May, 1966) showed that programs can be written without any `goto` statements. In fact, all programs can be written in terms of only three control structures:

- the *sequence structure*,
- the *selection structure*, and
- the *iteration structure*.

So the challenge for the programmers of the era was to shift their programming styles to "`goto`-less programming".

Control Structures

The results were impressive:

- shorter development times,
- more frequent on-time-delivery of systems,
- more frequent within-budget completion of software products.

The key success is that the structured programs are clearer, easier to debug and modify, and are more likely to be bug free in the first place.

Control Structures

C++ has three types of selection statements:

- `if`
- `if … else`
- `switch`

Control Structures

`if` statements performs an action if the condition is *true*

```
if (temperature > 90) {
    std:: cout << "it is hot outside";
}
```

Important: condition must be enclosed into parentheses!

Control Structures

`if … else` statements performs an action if the condition is *true* and performs a different action if the condition is *false*.

```
if (temperature > 90) {
    std:: cout << "it is hot outside";
}

else {
    std:: cout << "it is not hot outside";
}
```

Control Structures

`switch` statement performs one of many different actions depending on the value of an expression.

We will cover it in Chapter 5.

Nested `if...else` statements

Consider the following code:

```
if (grade >= 90) {
    cout << "A";
}

else {
    if (grade >= 80 and grade < 90) {
        cout << "B";
    }

    else {
        if (grade >= 70 and grade < 80) {
            cout << "C";
        }

        else { cout << "below C"; }
    }
}
```

Nested `if...else` statements

Consider the following code:

```
if (grade >= 90) {
    cout << "A";
}

else if (grade >= 80 and grade < 90) {
    cout << "B";
}

else if (grade >= 70 and grade < 80) {
    cout << "C";
}

else { cout << "below C"; }
```

Dangling-`else` problem

It is recommended that the control statement bodies are enclosed in braces, { and }.
This avoids a logical error called “dangling-else” problem.

Example:

```
if (grade >= 60) {
    cout << "passed";
}

else {
    cout << "failed";
    cout << "you need to take this course again";
}
```

Dangling-`else` problem

It is recommended that the control statement bodies are enclosed in braces, { and }.
This avoids a logical error called “dangling-else” problem.

Example:

```
if (grade >= 60) {
    cout << "passed";
}

else
    cout << "failed, ";
    cout << "you need to take this course again";
```

I forgot to use the braces for the else body block. Therefore, only the first `cout` statement will be performed if grade is lower than 60, and the second `cout` statement will be executed in any case!

Syntax and Logical Errors

Syntax errors are caught by the compiler

Examples: semicolon at the end of the statement, no closing brace

Logical errors, such as incorrect calculation, are not caught by the compiler.

Programs with *non fatal logical errors* execute, but produce incorrect results.

However, programs with *fatal logical errors* (e.g. memory mismanaging) fail and terminate prematurely.

Conditional Operator (? :)

Syntax:

```
<condition> ? <value1> : <value2>
```

If the `condition` if true, `value1` is returned, otherwise `value2` is returned

it is the only *ternary operator* (takes three operands) in C++

Conditional Operator (? :)

Syntax:

```
<condition> ? <value1> : <value2>
```

If the `condition` if true, `value1` is returned, otherwise `value2` is returned

it is the only *ternary operator* (takes three operands) in C++

Examples:

```
cout << ( grade>=60 ? "passed" : "failed");
```

```
letterGrade = (grade >= 60 ? 'P' : 'F');
```

while Iteration Statement

Example:

```
cout << "enter a positive integer > 5: ";
int n{0},s{0};
cin >> n;

if ( n < 5 ) { n = 5; }

While ( n > 0 ) {

    s += n*n;
    n -= 1;

}

cout << "the result is " << s;
```

Important: condition must be enclosed into parentheses!

top-down, step-wise refinement

When we define a class or implement an algorithm, it is recommended that we use a technique called *top-down, stepwise refinement*.
It helps to develop well-structured programs.

Let's use example from the book to employ the technique: *We are asked to develop a program that would find the class average grade* (it should be independent on the number of students in the class, so for each run we can have a different number of students)

top-down, step-wise refinement

develop a program that would find the class average grade

Top: give a single statement that conveys the overall function of the program

Determine the class average (for a quiz, test, hw, etc)

top-down, step-wise refinement

develop a program that would find the class average grade

Top: give a single statement that conveys the overall function of the program

Determine the class average (for a quiz, test, hw, etc)

It gives us the idea of what the program will do, but it is not enough to write a C++ program

top-down, step-wise refinement

develop a program that would find the class average grade

Top: give a single statement that conveys the overall function of the program

Determine the class average (for a quiz, test, hw, etc)

It gives us the idea of what the program will do, but it is not enough to write a C++ program

We need to divide it into smaller tasks, it gives us the first refinement:

top-down, step-wise refinement

develop a program that would find the class average grade

the first refinement:
declare (initialize) variables
input the grades ...
design decision: do we ask for the number of grades?

top-down, step-wise refinement

develop a program that would find the class average grade

the first refinement:
declare (initialize) variables
input the grades ...
design decision: do we ask for the number of grades? yes

top-down, step-wise refinement

develop a program that would find the class average grade

the first refinement:
declare (initialize) variables
get the number of grades
input/get the grades
calculate the average
display the result

This refinement gives us the *sequence structure*, i.e. order of steps to perform

top-down, step-wise refinement

develop a program that would find the class average grade

the first refinement:
declare (initialize) variables
get the number of grades
input/get the grades
calculate the average
display the result

This refinement gives us the *sequence structure*, i.e. order of steps to perform

Our next refinement could be to commit to specific variables, to decide the types of variables, to check the validity of the number of grades

top-down, step-wise refinement

develop a program that would find the class average grade

the second refinement:

~~*declare (initialize) variables*~~
initialize total to zero
declare the count as integer
~~*get the number of grades*~~
prompt the user to enter the number of grades (count)
check that count > 0
~~*input/get the grades*~~
while the user hasn't finished entering the grades:
    prompt the user to enter the next grade
    add the grade to the total
~~*calculate the average*~~
average = total / count
display the ~~result~~ average

top-down, step-wise refinement

develop a program that would find the class average grade

the second refinement:

~~*declare (initialize) variables*~~
initialize total to zero
declare the count as integer
~~*get the number of grades*~~
prompt the user to enter the number of grades (count)
check that count > 0
~~*input/get the grades*~~
while the user hasn't finished entering the grades:
  prompt the user to enter the next grade
  add the grade to the total
~~*calculate the average*~~
average = total / count
display the ~~result~~ average

the next refinement could check that the grade is valid (0-100)

In-class work!

develop a program that would find the class average grade

the second refinement:

initialize total to zero
declare the count as integer
prompt the user to enter the number of grades (count)
check that count > 0
while the user hasn't finished entering the grades:
    prompt the user to enter the next grade (must be 0-100)
    add the grade to the total
average = total / count
display the average

Let's proceed to implementation of the above pseudocode!
You can grab the file with the pseudocode above from our website: refinementExample_pseudocode.cpp

In-class work!

develop a program that would find the class average grade

the second refinement:

initialize total to zero
declare the count as integer
prompt the user to enter the number of grades (count)
check that count > 0
while the user hasn't finished entering the grades:
    prompt the user to enter the next grade (must be 0-100)
    add the grade to the total
average = total / count
display the average

Let's proceed to implementation of the above pseudocode!
You can grab the file with the pseudocode above from our website: refinementExample_pseudocode.cpp
A solution can be found in refinementExample.cpp

Converting Between Basic Types

Given two integer values, a and b, their quotient $\frac{a}{b}$ is not necessary an integer, but in C++, $5/2 = 2$.

How can we “fix” it?

Converting Between Basic Types

```
int a, b;
double result;

result  = static_cast<double>(a) / b;
```

explicit conversion of integer a to a double value

`static_cast` operator above creates a temporary floating-point value (a copy of the operand in parentheses).

Unary (only one operand) cast operators are available for use with any basic / built-in type.
They have the second highest precedence.

Converting Between Basic Types

```
int a, b;
double result;

result  = static_cast<double>(a) / b;
```

promotion/implicit conversion of the result of the division to a double value

For arithmetic, the compiler knows how to evaluate expressions in which the operand types are identical. Hence, the compiler will perform an operation called promotion/implicit conversion on operand b, from `int` to `double`.

Formatting Output: Floating-Point Numbers

If we type something like this:

```
cout << 5.0 / 3;
```

The output will be: `1.66667`

Formatting Output: Floating-Point Numbers

If we type something like this:

```
cout << 5.0 / 3;
```

The output will be: `1.66667`

What if we want to format it?
For example, keep only one decimal place.

Formatting Output: Floating-Point Numbers

If we type something like this:

```
cout << 5.0 / 3;
```

The output will be: `1.66667`

What if we want to format it?
For example, keep only one decimal place.

We can use *manipulators*:
`setprecision(1)` indicates that floating-point values should be output with `1` decimal place.
Note that the value is rounded when `setprecision` is used!

`fixed` indicates that floating-point values should be output in **fixed-point notation**, as opposed to **scientific notation**.

Formatting Output: Floating-Point Numbers

If we type something like this:

```
cout << 5.0 / 3;
```

The output will be: 1.66667

```
#include <iostream>
#include <iomanip>
```

include this library to use the manipulators

```
using namespace std;

int main() {

   cout << setprecision(1) << fixed;
   cout << 5.0 / 3;
}
```

This code will output: 1.7

Increment and Decrement Operators

C++ provides unary operators for adding or subtracting 1 to/from the value of a numeric variable: ++ and — .

They can be used in *prefix* and in *postfix* forms:

++a increments a by 1,
then the new value of a can be used in the expression

a++ uses the current value of a in the expression, and
only then a is incremented by 1

Similarly, with –– .

Example: let's see the increment.cpp program

Operator Precedence and Associativity

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

In-class work

Exercise 4.28 (*binary* to *decimal* conversion):
Write a program that converts a binary number to its decimal form.

For example, $101_2 = 5$ and $110011_2 = 51$

HW assignment

1) Exercise 4.27
2) Exercise 4.37

Suggested exercises
(*not for grade, but the questions related to these will appear on a quiz or a test*):
1) Chapter 4,
Summary and all Self-Review Exercises (pages 141-146)
2) Chapter 4, Exercises (page 149): 4.11, 4.12, 4.21, 4.23, 4.32