CONTROL STRUCTURES, PART 1

Chapters 4 & 5

Deitel 10th Edition

TOPICS

Overview of Control Structures

Selection

- **if** Single-Selection Statement
- **if…else** Double-Selection Statement
- Nested **if...else** Multiple Selection Statements
- **switch** Multiple Selection Statement (chapter 5)

Formatting Output (from chapter 5)

Converting Between Fundamental Types with Cast Operators

Overview of Control Structures

Control Structure Definition

Block of code defining how a program proceeds through an algorithm

Sometimes called the “flow”

Types of Control Structures

All programs can be written using only **three control structures**

- **Sequence** structure
- **Selection** structure
- **Repetition** structure

Transfer of control

- The next statement to execute
- Not necessarily the next one in the code…

Sequence Control Structure

Sequential execution

- One after the other in the order in which they are written.
- Built into C++ (and all structured programming languages)

Unless directed otherwise, statements executed one after another in the order in which they're written

Block

- Series of statements enclosed within curly braces

Any single sequence statement can be replaced with a block

- Syntactically correct, but may not be algorithmically correct

Selection Control Structures

Allow for different paths through a program, depending on the value of a **condition**

Three **selection statements**:

- **if** statement
- **if…else** statement
- **switch** statement

Types of selection

- Single
- Double
- Multiple

Repetition Control Structures

Allow statements to be executed repeatedly, depending on the value of a **condition**

Four **repetition statements** (looping statements):

- while
- do…while
- for
- Range-based for

Types of repetition

- Conditional
- Counter-controlled
- Sentinel-controlled

Selection Using if Statements

if Statement for Single Selection

Performs **single selection**

- Performs an action if a condition is true
- Doesn’t do anything if condition is false

Single-selection statement – selects or ignores a single action (or group of actions within curly braces)

Indenting between curly braces is good programming practice

if Statement Example

Pseudocode:

```
If student’s grade is greater than or equal to 60,
    Print “Passed”
```

C++:

```
if ( studentGrade >= 60 ) {
    cout << "Passed";
}
```

BOOL **DATA TYPE**

In C++, numeric values can be considered true or false:

- Zero – false
- Any non-zero value – true

Also can use the **bool** data type

- Can have a value of **true** or **false, 0** or **non-zero (usually 1)**
- **bool**, **true**, and **false** are all keywords in C++

Example:

bool isComplete = **true**;

IF…ELSE STATEMENT FOR DOUBLE SELECTION

Performs **double selection**

- Performs an action if a condition is true
- Performs a different action if the condition is false.

Double-selection statement – selects between two different actions (or groups of actions within curly braces)

Body of the else is also indented

- Good programming practice

IF...ELSE STATEMENT EXAMPLE

Pseudocode:

```
If student’s grade is greater than or equal to 60
    Print “Passed”
Otherwise
    Print “Failed”
```

C++:

```
if ( grade >= 60 )
    cout << "Passed";
else
    cout << "Failed";
```

if…else Statement – Shorthand Syntax

Shorthand version of if…else structure uses the **ternary conditional operator:**

?:

A **ternary** operator needs **three** operands

Operands and conditional operator form a **conditional expression**

Ternary Conditional Operator General Format

operand1 ? operand2 : operand3

Operand1 – boolean expression – evaluates to a boolean value (true or false)

Operand2 – value of expression if the boolean expression is true

Operand3 – value of expression if the boolean expression is false

boolean ? value if true : value if false

Previous Example Using Standard if...else Syntax

Standard if...else syntax:

```
if ( grade >= 60 )
      cout << "Passed" ;
else
      cout << "Failed" ;
```

Previous Example Using Shorthand Syntax

Standard if...else syntax:

```
if ( grade >= 60 )
      cout << "Passed" ;
else
      cout << "Failed" ;
```

Shorthand syntax:

```
grade >= 60 ? cout << "Passed" : cout << "Failed" ;
```

Another Example – What is the Shorthand Syntax?

Standard if...else syntax:

```
if ( x < 10 )
        y = 2;
else
        y = 3;
```

Shorthand syntax?

ANOTHER EXAMPLE – SHORTHAND SYNTAX

Standard if...else syntax:

```
if ( x < 10 )
      y = 2;
else
      y = 3;
```

Shorthand syntax:

```
y = ( x < 10 ? 2 : 3 );
```

IF…ELSE STATEMENT – MULTIPLE CASES

Can test multiple cases by placing if…else statements inside other if…else statements to create **nested** if…else statements.

IF…ELSE MULTIPLE CASES IN PSEUDOCODE

```
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…ELSE MULTIPLE CASES IN C++ CODE

```
if ( studentGrade >= 90 )
    cout << "A" ;
else
    if ( studentGrade >= 80 )
        cout << "B" ;
    else
        if ( studentGrade >= 70 )
            cout << "C" ;
        else
            if ( studentGrade >= 60 )
                cout << "D" ;
            else
                cout << "F" ;
```

If studentGrade >= 90, the first four conditions will be true, but only the statement in the **if** part of the first **if…else** statement will execute.

After that, the else part of the “outermost” if…else statement is skipped.

The boxes are levels of **nesting**

CAN WRITE THE PRECEDING NESTED IF…ELSE STATEMENT AS:

```
if ( studentGrade >= 90 )
    cout << "A" ;
else if ( studentGrade >= 80 )
    cout << "B" ;
else if ( studentGrade >= 70 )
    cout << "C" ;
else if ( studentGrade >= 60 )
    cout << "D" ;
else
    cout << "F" ;
```

What is better or worse about this approach?

Selection Using switch Statement

Switch Statement for Multiple Selection

Multiple-selection statement

- Selects among many different actions (or lists of actions)

Performs one of several actions based on the value of an expression

Expression must be a **constant integral expression**

- Has an integer value – **int** or **char**

Switch General Format

```
switch ( switch value )      {
     case label 1:
          statement(s) to perform if switch value = label 1;
          break;
     case label n:
          statement(s) to perform if switch value = label n;
          break;
     default:
          break;
}
```

Switch Statement Notes

Every value must be listed in a separate **case label**

- No ranges of values

Each case can have multiple statements

No braces required around multiple statements in a case

Statements for a matching case execute until a break or the end of the switch is encountered

If a break is missing, statements for the next case execute

- Called “falling through”

CASE LABELS MUST CONTAIN CONSTANTS

Constant value

- **int** constant, like 7 or 487
- **char** constant, like 'A', '7' or '$'

Constant variable

- int or char variable declared with keyword **const**
- Contains a value which does not change for the entire program
- const int x = 10;

Switch Example

Fig. 5.11 – LetterGrades.cpp

Note the while loop and its condition

- We’ll talk more about this in part 2

Note the expression in the switch value…

FORMATTED OUTPUT IN C++

Formatted Output in C++ Can use C or C++ Style

Format control strings & **printf** function are available in C++

- Inherited from C

But C++ input & output can also be handled using **classes**

- Preferred for optimization

Output is sent as a stream of bytes to the output stream

To indicate formatting, use **stream manipulators**

Must include the **iomanip** header: **#include <iomanip>**

Stream Manipulators Used For:

Field width

- Any data type

Justification

- Left or right
- For any data type

Precision

- For floating-point values

Floating-point number formats

- Fixed, hexadecimal, scientific, etc.

Other helpful manipulators

- New lines
- Flushing the output buffer

STREAM MANIPULATOR FOR FIELD WIDTH: SETW()

General Format:

cout << **setw(*integer value*)** << *nextItemToDisplay*

Requires an integer value within () to indicate field width

- Value can be an int constant, int variable, or int expression
- Called a “parameterized” manipulator

Will only be used for the NEXT value sent to the output stream

Part of **std namespace** Lines 17, 25 in Fig 5.6 InterestWithsetw.cpp

STREAM MANIPULATORS FOR JUSTIFICATION: LEFT **AND** RIGHT

General format:

```
cout << left;
cout << right;
```

Default justification for all output is **right**

To apply left justification, use manipulator **left**

To reapply right justification, use manipulator **right**

InterestWithLeft.cpp

STREAM MANIPULATORS FOR JUSTIFICATION: LEFT AND RIGHT – NOTES

Justification when using **setw**

- If value to display shorter than width set by **setw**, it will be **right-justified** within the space indicated by setw
- If value to display longer than width set by **setw**, the value will extend to the right

Left and **right** are **persistent ("sticky")**

- **Will be used for ALL FOLLOWING values sent to the output stream**
- Must be explicitly changed if necessary

Non-parameterized, and part of **std namespace**

FLOATING-POINT NUMERIC FORMATS IN C++

Fixed

- Includes number, decimal point, and fraction part
- **Precision is the number of digits after the decimal point**

Scientific

- Displays a number in scientific notation
- **Precision is the number of digits after the decimal point**

General

- Default for floating point values; output precision is 6
- **Precision is total number of digits displayed including digits before and after the decimal point but does not include the decimal point itself.**
- Mix of fixed & scientific. If number is small enough, fixed is used. If number gets too large, scientific is used.

Stream Manipulator for Precision: setprecision()

General Format:

cout << **setprecision(*integer value*) ;**

Requires an integer parameter within () to indicate **precision**

Will be used for ALL floating-point values sent to the output stream from that point forward, unless it is changed (persistent)

Displays numbers in “general” notation by default

InterestWithPrecision.cpp

Fixed Floating-Point Format

General Format:

cout << **fixed;**

Will be used for ALL floating-point values sent to the output stream from that point forward, unless it is changed (persistent / sticky)

Difficult to change back to general, so **fixed almost always used**

InterestWithPrecisionFixed.cpp

Other Helpful Stream Manipulators

endl

- New lines
- Same as sending **'\n'** to output stream

flush

- Flushes output buffer
- Rarely needed

Converting Between Fundamental Types with Cast Operators

REMEMBER INTEGER DIVISION…

Dividing two integer values always gives an integer result:

```
int result = 846 / 10;
```

What should the value of result be?

INTEGER DIVISION RESULT

Dividing two integer values always gives an integer result:

```
int result = 846 / 10;
```

It should be 84.6

But, in this example, only the 84 is stored. Why?

CAST OPERATORS

To temporarily treat integers as floating-point numbers for use in more precise calculations, use **static_cast** operator

General format:

static_cast<*fundamental or class data type*>(*value to be cast)*

Creates a **temporary copy of its operand** using the indicated data type

Cast Operators – Parts

To temporarily treat integers as floating-point numbers for use in more precise calculations, use **static_cast** operator

General format:

$$\textbf{static\_cast<}\underbrace{\textbf{\textit{fundamental or class data type}}}_{\text{data type to use}}\textbf{>(}\underbrace{\textbf{\textit{value to be cast)}}}_{\text{operand}}$$

Creates a **temporary copy of its operand** using the indicated data type

Casting Example, Step 1

```
int x{25};

int y{10};

double z{static_cast<double>(x) / y};
```

What are all the different values and expressions here?

Casting Example, Step 2

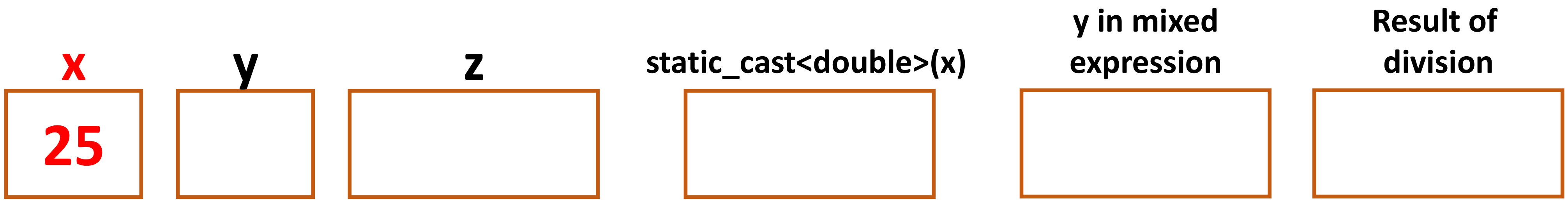
```
int x{25};

int y{10};

double z{static_cast<double>(x) / y};
```

What are all the different values and expressions here?

x	y	z	static_cast<double>(x)	y in mixed expression	Result of division

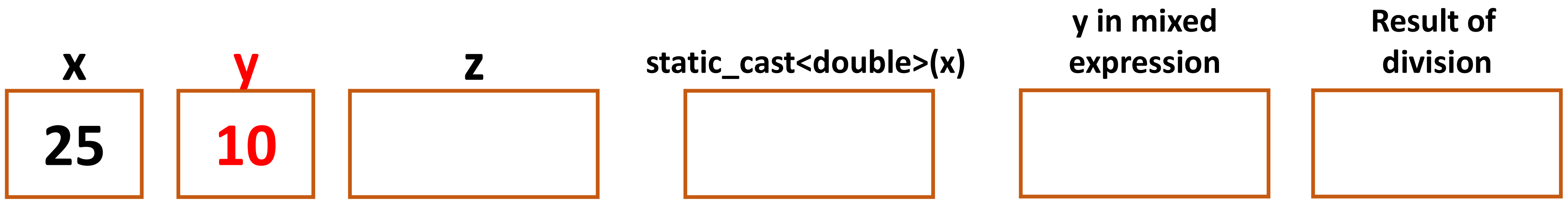
CASTING EXAMPLE, STEP 3

```
int x{25};

int y{10};

double z{static_cast<double>(x) / y};
```



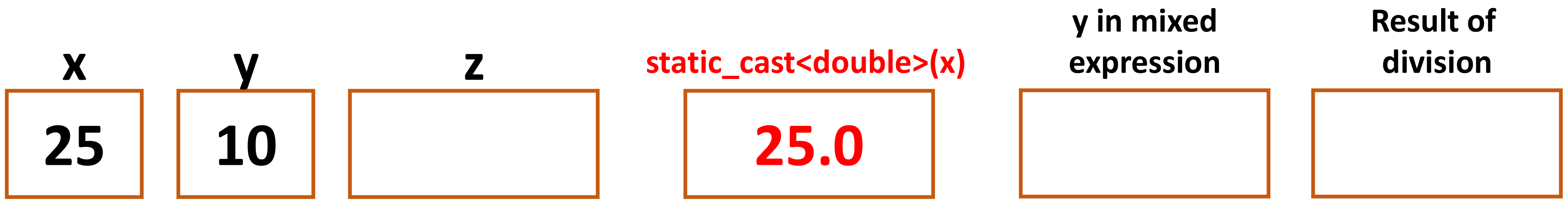
CASTING EXAMPLE, STEP 4

```
int x{25};
int y{10};
double z{static_cast<double>(x) / y};
```



Casting Example, Step 5

```
int x{25};

int y{10};

double z{static_cast<double>(x) / y};
```



CASTING EXAMPLE, STEP 6

```
int x{25};

int y{10};

double z{static_cast<double>(x) / y};
```

x	y	z	static_cast<double>(x)	y in mixed expression	Result of division
25	10		25.0	10.0	

CASTING EXAMPLE, STEP 7

```
int x{25};

int y{10};

double z{static_cast<double>(x) / y};
```

x	y	z	static_cast<double>(x)	y in mixed expression	Result of division
25	10		25.0	10.0	2.5

Casting Example, Step 8

```
int x{25};

int y{10};

double z{static_cast<double>(x) / y};
```

x	y	z	static_cast<double>(x)	y in mixed expression	Result of division
25	10	2.5	25.0	10.0	2.5

EXPLICIT VS. IMPLICIT CONVERSION

static_cast operator performs **explicit conversion** (or type cast)

- The value stored in the operand is unchanged.

Promotion (or **implicit conversion**) performed on operands in a **mixed expression**

- Mixed expression contains integers and floating-point values
- The integers are promoted to floating points for use in the expression only

Notes on Cast Operators in C++

Available for any type

Unary operators

Associate from right to left

Precedence is one level higher than that of the multiplicative operators *, / and %.

Appendix A: Operator precedence chart

Notes on Cast Operators in C++, continued

Two syntaxes: **functional** and **C-like** (aka traditional)

C-like syntax works most of the time for fundamental data types

Functional syntax has 4 formats, and must be used when casts involve objects; can be used for fundamental types too

- dynamic_cast <new_type> (expression)
- reinterpret_cast <new_type> (expression)
- static_cast <new_type> (expression)
- const_cast <new_type> (expression)

More to come in later chapters, esp. ch. 10

END OF CONTROL STRUCTURES, PART 1