Ch3: if…else, double, Type Conversion, Operators

Control Structures

Control Structures

It is demonstrated that all programs could be written in terms of only 3 control structures:

- **Sequence Structure**: Unless directed otherwise, the C++ statements are executed one after the other in the order in which they're written (sequential execution).
- **Selection Structure**: C++ has 3 types of selection statements.
  - if statement (single-selection statement)
  - if…else statement (double-selection statement)
  - switch statement (multiple-selection statement)
- **Iteration Structure**: C++ provides 3 iteration statements that enable programs to perform statements repeatedly as long as a condition remains true.
  - while statement
  - do…while statement
  - for statement (and range-based for statement)
- In algorithms, these 3 control structures can be combined in only two ways: **stacking** (one after another) and **nesting** (one inside another).

if…else **Double-Selection Statement**

if…else (or Double-Selection Statement)

if…else statement (or double-selection statement) performs an action (or group of actions) if a condition is true and performs a *different* action (or group of actions) if the condition is false. Conditions are usually formed by using the relational and equality operators.

Body of if and else that can be a single statement or a **Block** of several statements in {}.

```
if (condition){
  statement(s);
}
else {
  statement(s);
}
```

- You do not need to use braces, { }, around single-statement bodies. However, it is always recommended to enclose all the statement bodies in braces to avoid logic errors called the **dangling-else problem**.

```
if (condition)
  statement;
else
  statement;
```

- The **indentation** of the statement(s) in the bodies of an if…else statement, which enhances readability, is optional, but recommended. If there are several levels of indentation, each level should be indented the same additional amount of space. Many IDEs do indentation automatically.

Dangling-else Problem & Local Variables in Blocks

- These two snippets are not identical:

```
if (grade >= 60) {
  std::cout << "Passed";
}
else {
  std::cout << "Failed\n";
  std::cout << "You must take this course again.";
}
```

```
if (grade >= 60)
  std::cout << "Passed";
else
  std::cout << "Failed\n";
  std::cout << "You must take this course again.";
```

The last line (statement) is outside the body of the else part of the if…else statement and would execute regardless of the condition.

- In general, a **variable declared in a block** (enclosed in braces {}) is a **local variable** and can be used only from the line of its declaration to the closing right brace of the block (This restricted use is known as the variable's **scope**, which defines where it can be used in a program). The blocks can appear in **all control structures** and **functions**.

Remarks

- It is possible to have an **empty statement**, by placing a semicolon (;) where a statement would normally be or using and empty block.

```
if (grade >= 60) {
   std::cout << "Passed";
}
else {
}
```

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

- Placing a semicolon after the parenthesized condition in an if or if…else statement leads to a **logic error** in if statements and a **syntax error** in if…else statements (when the if-part contains a body statement).

Nested if…else Statements

A program can test multiple cases by placing if…else statements inside other if…else statements to create **Nested if…else statements**.

```
if (grade >= 90) {
  std::cout << "A";
}
else {
  if (grade >= 80) {
    std::cout << "B";
  }
  else {
    if (grade >= 70) {
      std::cout << "C";
    }
    else {
      if (grade >= 60) {
        std::cout << "D";
      }
      else {
        std::cout << "F";
      }
    }
  }
}
```

- Nested if…else statements are usually preferred to be identically written as

```
if (grade >= 90) {
  std::cout << "A";
}
else if (grade >= 80) {
  std::cout << "B";
}
else if (grade >= 70) {
  std::cout << "C";
}
else if (grade >= 60) {
  std::cout << "D";
}
else {
  std::cout << "F";
}
```

- This form avoids deep indentation of the code to the right (although the compiler ignores indentations).

Sample Program: Computing Student's Letter Grade

Class **Student** defined in the header **Student.h**

```
// Student class that stores a student name and average.
#include <string>
class Student {
public:
  // constructor initializes data members
  Student(std::string studentName, int studentAverage) : name{studentName} {
    // sets average data member if studentAverage is valid
    setAverage(studentAverage);
  }

  // sets the Student's name
  void setName(std::string studentName) {
    name = studentName;
  }

  // sets the Student's average
  void setAverage(int studentAverage) {
    // validate that studentAverage is > 0 and <= 100; otherwise,
    // keep data member average's current value
    if (studentAverage > 0) {
      if (studentAverage <= 100) {
        average = studentAverage; // assign to data member
      }
    }
  }
  // retrieves the Student's name
  std::string getName() const {
    return name;
  }
  // retrieves the Student's average
  int getAverage() const {
    return average;
  }
  // determines and returns the Student's letter grade
  std::string getLetterGrade() const {
    // initialized to empty string by class string's constructor
    std::string letterGrade;
    if (average >= 90) {
      letterGrade = "A";
    }
    else if (average >= 80) {
      letterGrade = "B";
    }
    else if (average >= 70) {
      letterGrade = "C";
    }
    else if (average >= 60) {
      letterGrade = "D";
    }
    else {
      letterGrade = "F";
    }
    return letterGrade;
  }
private:
  std::string name;
  int average{0}; // initialize average to 0
}; // end class Student
```

Sample Program: Computing Student's Letter Grade (cont.)

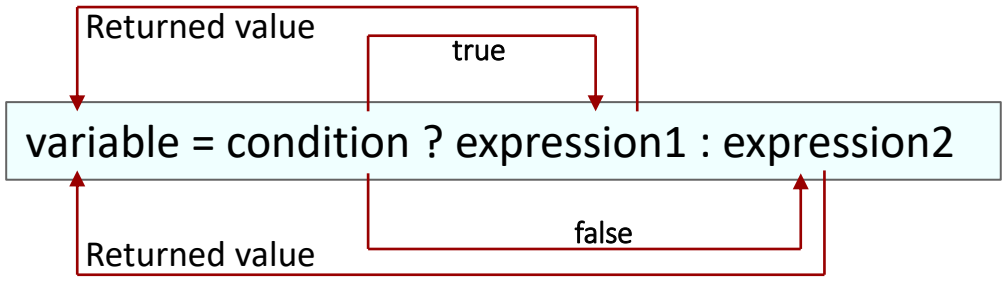
.cpp source-code file

```
// Create and test Student objects.
#include <iostream>
#include "Student.h"

int main() {
  Student account1{"Jane Green", 93};
  Student account2{"John Blue", 72};

  std::cout << account1.getName() << "'s letter grade equivalent of "
    << account1.getAverage() << " is: "
    << account1.getLetterGrade() << "\n";
  std::cout << account2.getName() << "'s letter grade equivalent of "
    << account2.getAverage() << " is: "
    << account2.getLetterGrade() << std::endl;
}
```

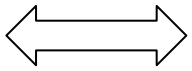
Conditional Operator (?:)

The **Conditional Operator (?:)** is C++'s only **ternary operator** (i.e., an operator that takes three operands) that can be used in place of an if…else statement (with single-statement blocks) to make the code shorter and clearer.

Form 1:




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



```
if (grade >= 60) {
  std::cout << "Passed";
}
else {
  std::cout << "Failed";
}
```

Form 2:

```
condition ? statement1 : statement2 ;
```

statements without semicolon

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

- Conditional expressions can appear in some program locations where if…else statements cannot.

UML Activity Diagram

UML Activity Diagram

An **UML Activity Diagram** models the workflow (activity) of a portion of a software system or algorithm by several symbols. These symbols are connected by **transition arrows**, which represent the flow of the activity (or the order in which the actions should occur).

Symbols:

rectangle with rounded corners

Containing an **action expression**

solid circles

Representing **initial state** or entry point

solid circle surrounded by a hollow circle

Representing **final state** or exit point

diamond

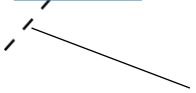
Indicating a **decision symbol** or a **merge symbol**

transition arrow

Represent the flow of the activity

rectangles with the upper-right corners folded over

Representing **UML notes** (like comments in C++)



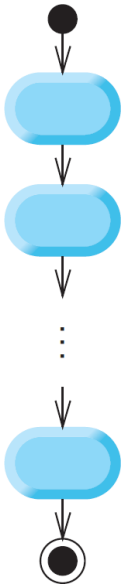
dotted line

Connecting each note with the element it describes

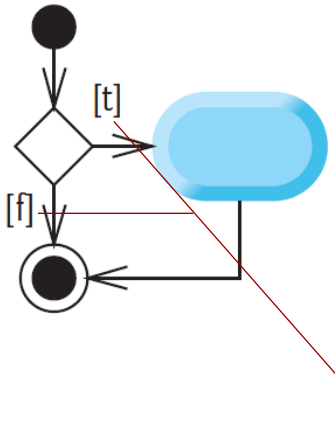
UML Activity Diagram for Sequence Structure and if Single-Selection Statement

Like pseudocode, activity diagrams help you develop and represent algorithms. All control structures can be modeled as activity diagrams.

Sequence Structure



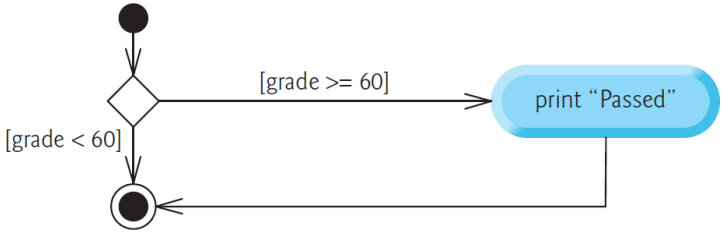
if statement
(single selection)



guard conditions
(specified by []) can
be true or false.

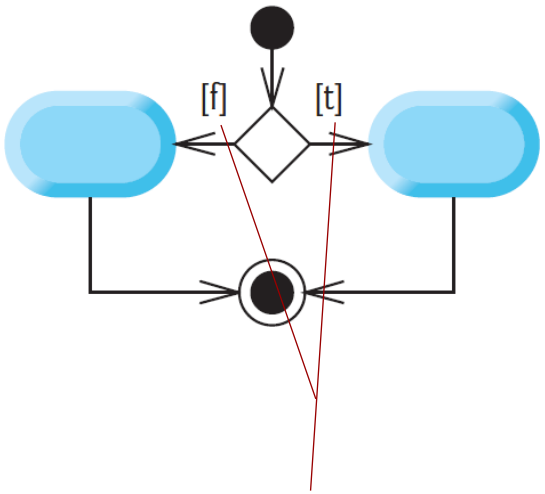
Example:

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



UML Activity Diagram for if...else Double-Selection Statement

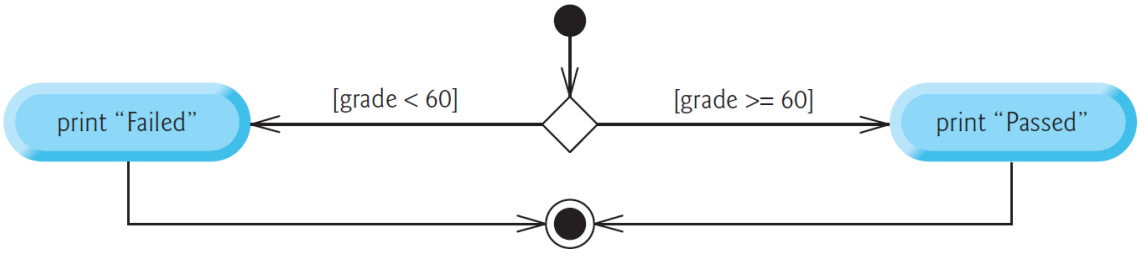
`if...else` statement
(double selection)



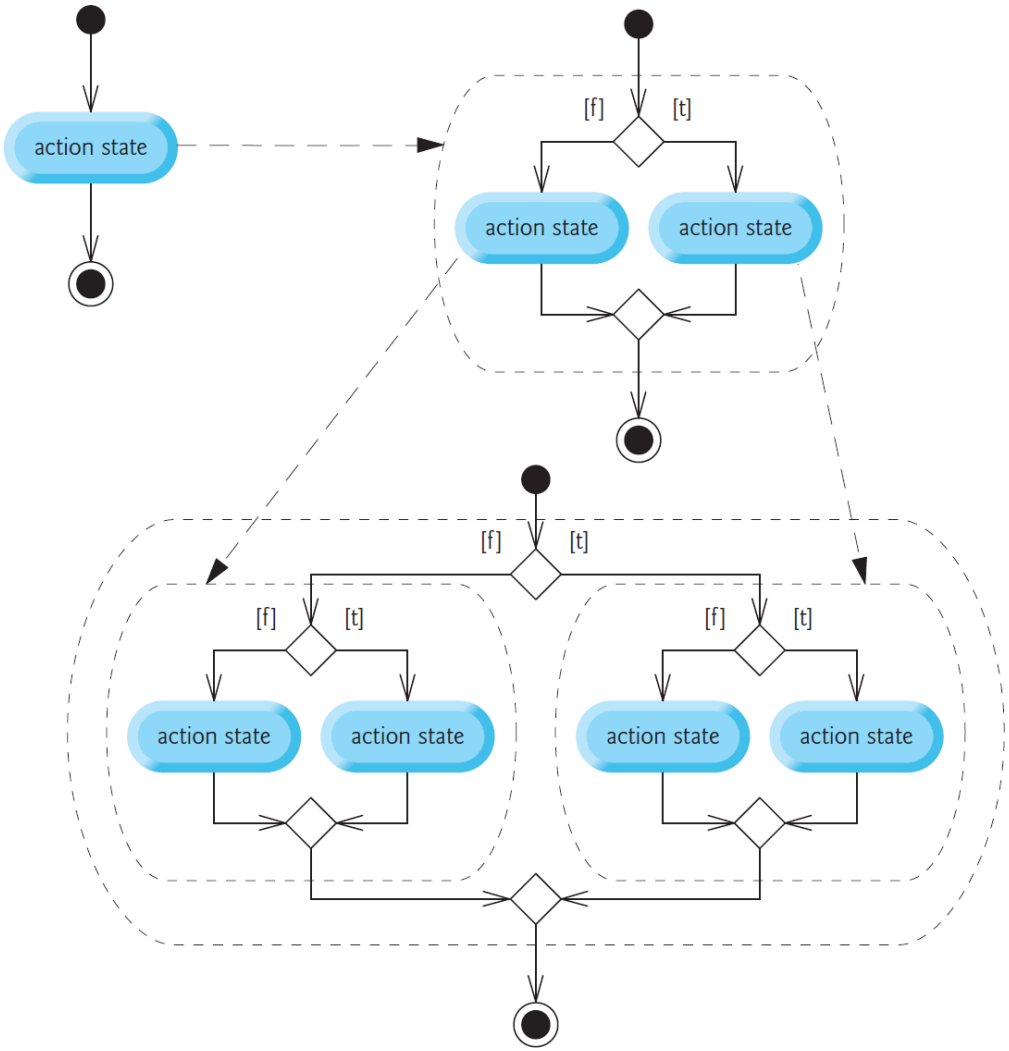
guard conditions
(specified by []) can be true or false.

Example:

```
if (grade >= 60) {
   std::cout << "Passed";
}
else {
   std::cout << "Failed";
}
```



UML Activity Diagram for Nested if…else Statements



Logical Operators

Logical Operators

The if, if…else, while, do…while, and for statements each require a condition to determine how to continue a program's flow of control. While relational and equality operators can be used to test whether a particular condition is true or false, they can only test one condition at a time.

Logical Operators provide us with the capability to test multiple simple conditions.
C++ has 3 logical operators:

- && (Logical **AND**)
- || (Logical **OR**)
- ! (Logical **NOT**)

- C++ evaluates to zero (false) or nonzero (true) all expressions that include relational operators, equality operators, or logical operators.

Logical AND (&&) Operator

This binary operator is used to test whether **if and only if both** operands are true.

(expression 1) && (expression 2)

expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

Truth Table

Example:

```
#include <iostream>
int main() {
   std::cout << "Enter an integer number: ";
   int value;
   std::cin >> value;

   if (value > 10 && value < 20)
      std::cout << "Your value is between 10 and 20\n";
   else
      std::cout << "Your value is not between 10 and 20";
}
```

Testing more than 2 conditions:

```
if (value > 10 && value < 20 && value != 16)
  std::cout << " 10<value<20, but not 16!";
```

Logical OR (||) Operator

This binary operator is used to test whether **either or both** of two conditions is true.

(expression 1) || (expression 2)

expression1	expression2	expression1 || expression2
false	false	false
false	true	true
true	false	true
true	true	true

Truth Table

Example:

```
#include <iostream>
int main() {
  std::cout << "Enter an integer number: ";
  int value;
  std::cin >> value;

  if (value == 0 || value == 1)
    std::cout << "You picked 0 or 1\n";
  else
    std::cout << "You did not pick 0 or 1";
}
```

Testing more than 2 conditions:

```
if (value == 0 || value == 1 || value == 2 || value == 3)
  std::cout << "You picked 0, 1, 2, or 3!";
```

Logical NOT (!) Operator

This unary operator (!) can be used to flip a condition or Boolean value from true to false, or false to true.

!(expression)

expression	!expression
false	true
true	false

Truth Table

Example:

```
#include <iostream>
int main() {
  int x{5};
  int y{7};
  if (!(x > y))
    std::cout << x << " is not greater than " << y << '\n';
  else
    std::cout << x << " is greater than " << y << '\n';

  if (!x > y) // not the same as (!(x > y)), !x evaluates to 0
    std::cout << x << " is not greater than " << y << '\n';
  else
    std::cout << x << " is greater than " << y << '\n';
}
```

The parentheses around the condition are needed because the NOT operator has a higher precedence than the relational operators. If logical NOT is intended to operate on the result of other operators, use parentheses.

Remarks

- In general, logical AND has higher precedence than logical OR, thus, logical AND operators will be evaluated ahead of logical OR operators.

`value1 || value2 && value3` 〓 `value1 || (value2 && value3)` ≠ `(value1 || value2) && value3`

When mixing logical AND and logical OR in a single expression, explicitly parenthesize each operation to ensure they evaluate how you intend.

- In most cases, logical NOT can be avoided by expressing the condition differently with an appropriate relational or equality operator. For example:

`!(x == y)` 〓 `x != y` `!(x > y)` 〓 `x <= y`

- De Morgan's law:

`!(x && y)` 〓 `!x || !y`

`!(x || y)` 〓 `!x && !y`

Short-Circuit Evaluation

- Both && and || operators are evaluated from left to right.
- **Short-Circuit Evaluation** is a feature of && and || logical operators in which the second argument (right-hand side) is executed or evaluated only if the first argument (left-hand side) does not suffice to determine the value of the expression.
- That is, when the first argument of the && function evaluates to false, the overall value must be false; and when the first argument of the || function evaluates to true, the overall value must be true.
- This is done to avoid unnecessary calculation for optimization purposes.

```
#include <iostream>
int main() {
  int x{0};
  if ((x != 0) && (10/x == 2)) {
    std::cout << "if's body!\n";
  }
  std::cout << x;
}
```

This feature prevent the possibility of division by zero.

Sample Program: Truth Table

Use logical operators to create truth tables.

By default, bool values are displayed as 1 and 0. We can use stream manipulator boolalpha (a sticky manipulator) to specify that the value of each bool expression should be displayed as either the word "**true**" or the word "**false**."

```
#include <iostream>
int main() {
   std::cout << std::boolalpha;
  // create truth table for && (logical AND) operator
  std::cout << "Logical AND (&&)"
    << "\nfalse && false: " << (false && false)
    << "\nfalse && true: " << (false && true)
    << "\ntrue && false: " << (true && false)
    << "\ntrue && true: " << (true && true) << "\n\n";
  // create truth table for || (logical OR) operator
  std::cout << "Logical OR (||)"
    << "\nfalse || false: " << (false || false)
    << "\nfalse || true: " << (false || true)
    << "\ntrue || false: " << (true || false)
    << "\ntrue || true: " << (true || true) << "\n\n";
  // create truth table for ! (logical NOT) operator
  std::cout << "Logical NOT (!)"
    << "\n!false: " << (!false)
    << "\n!true: " << (!true) << std::endl;
}
```

float & double Data Types

float & double Data Types

C++ provides data types float and double to store floating-point numbers (real numbers) in memory.

```
float x{10.12};
```

```
double x{1000.12345};
```

- double variables can typically store numbers with larger magnitude and finer detail (i.e., more digits to the right of the decimal point, also known as the number's precision) than float variables (~ 6 vs 15 significant digits). ▶
- There are two different ways to declare floating-point numbers: Standard Notation & Scientific Notation.

```
double pi { 3.14159 }; // standard notation
double avogadro { 6.02e23 }; // scientific notation
double electronCharge { 1.6e-19 }; // scientific notation
```

- C++ also supports data type long double for floating-point values with larger magnitude and more precision than double.
- C++ treats all floating-point numbers as double values by default. Thus, most programmers represent floating-point numbers with type double.

Presentation of Floating-Point Numbers

- setprecision(n) is a (parameterized) stream manipulator which sets the decimal precision (digits to the right of the decimal point) to n after rounding for printing on the screen.
- fixed and scientific are stream manipulator which write floating-point values in fixed-point or scientific notation.

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

int main () {
  double a{3.1415926534};
  double b{2006.0624};
  double c{2.23e-10};

  std::cout << std::setprecision(3) << std::fixed;
  std::cout << "fixed:\n" << a << "\n" << b << "\n" << c << "\n" << "\n";

  std::cout << std::setprecision(3) << std::scientific;
  std::cout << "scientific:\n" << a << "\n" << b << "\n" << c << "\n";
}
```

- These format settings are **sticky settings** and remain in effect until they are changed.

- setprecision belongs to namespace std and is defined in <iomanip> header file.
- fixed and scientific belong to namespace std and are defined in <iostream> header file.

Representational Error of Floating-Point Numbers (double)

- 10 divided by 3 is $3.3333333 \ldots = 3.\overline{3}$. However, the computer allocates only a fixed amount of space to hold such a value. Thus, the stored floating-point value can be only an **approximation**.

- When two floating-point numbers with two digits to the right of the decimal point are added, the output could appear incorrectly.

```
#include <iostream>
#include <iomanip>
int main() {
  double a{14.234};
  double b{18.673};
  std::cout << std::setprecision(2) << std::fixed;
  std::cout << "a = " << a << "\n"
            << "b = " << b << "\n"
            << "a + b = " << a + b << "\n"; // a + b = 32.907
}
```

```
a = 14.23
b = 18.67
a + b = 32.91
```

- Floating-point numbers with many digits of precision to the right of the decimal point are represented incorrectly.

```
#include <iostream>
#include <iomanip>
int main() {
  double a{123.02};
  std::cout << std::setprecision(15) << std::fixed;
  std::cout << "a = " << a << "\n";
}
```

```
a = 123.019999999999996
```

Type Conversion

Implicit Type Conversion: Narrowing Conversions

Implicit Type Conversion (Automatic Type Conversion) is done automatically by the compiler on its own. Implicit Type Conversion is either **Narrowing Conversion** or **Promotion**.

- C++ converts the double value 12.7 to an int, by truncating the floating-point part (.7), a **narrowing conversion** that loses data.
- For fundamental-type variables, **list-initialization** syntax prevents Narrowing Conversions that could result in *data loss*. Thus,

```
int x2{12.7};
int x3 = {12.7};
```

yield a compilation error.

- This does not contain a narrowing conversion, however, due to integer division, the fractional part is truncated.

```
int x5{x2 / 10};
int x5{x4 / 10.0};
```

yield a compilation error, due to an attempted narrowing conversion.

```
#include<iostream>
int main() {
  int x1 = 12.7; // prior to C++11, Narrowing Conversion
  double x2{12.7}; // direct list initialization (preferred)
  double x3 = {12.7}; // copy list initialization
  int x4{12};
  int x5{x4 / 10};
  std::cout << "x1: " << x1 << "\n";
  std::cout << "x2: " << x2 << "\n";
  std::cout << "x3: " << x3 << "\n";
  std::cout << "x5: " << x5 << std::endl;
}
```

Implicit Type Conversion: Promotion

Promotion happens based on the following order

bool < char & signed char < unsigned char < short int < unsigned short int < int < unsigned int < long int < unsigned long int < long long int < unsigned long long int < float < double < long double

in the following cases, to avoid lose of data:

1. Assignments:

2. Arithmetic (*, /, %, +, -) and Relational (<, >, <=, >=, ==, !=) Operations:

int values 3 and (x1 + 30) are converted to type double.

```
#include<iostream>
int main() {
  int x1{20};
  double x2{3}; // promotion of integer 3
  x2 = x1 + 30;
  int x3{12};
  double x4{12.7};
  x1 = x3 + x4; // narrowing conversions
  std::cout << "x1: " << x1 << "\n";
  x2 = x3 + x4; // promotion of x3
  std::cout << "x2: " << x2 << "\n";
  double x5{x4 / x3}; // promotion of x3
  std::cout << "x5: " << x5 << std::endl;
}
```

For these operations, the compiler knows how to evaluate only expressions in which the operand data types are identical. In expressions containing values of two or more data types (**mixed-type expressions**), the compiler **promotes** the type of each value to the "highest" type in the expression (actually a temporary version of each value is created and used for the expression, the original values remain unchanged).

Explicit Type Conversion or Type Casting

Explicit Type Conversion or **Type Casting** happens when the user manually convert a value of one data type to a value of another data type.

- C++ supports 5 different types of casts: C-style Cast, static_cast, const_cast, dynamic_cast, and reinterpret_cast. Cast operators are unary operators and available for use with every fundamental type and with class types as well.

- C-style Cast: (data type) expression or data type (expression)

- Static Cast: static_cast<data type>(expression)

(preferred)

Type casting operators converts a **temporary copy** of its operand to the intended data type to be used in the calculations (without changing the data type of its operand).

Best practice: Avoid using C-style cast and use static_cast when you need to convert the data type of a value.

```
#include <iostream>
int main() {
  int x{10};
  int y{4};
  double d1{(double)x / y};
  std::cout << d1 << "\n"; // prints 2.5
  double d2{double(x) / y};
  std::cout << d2 << "\n"; // prints 2.5
  double d3{static_cast<double>(x) / y};
  std::cout << d3 << "\n"; // prints 2.5
  int d4{static_cast<int>(d3)};
  std::cout << d4 << "\n"; // prints 2
}
```

Explicit Promotion

Explicit Narrowing Conversion

Compound Assignment, Increment, and Decrement Operators

Compound Assignment Operators

The **Compound Assignment Operators** can be used to simplify assignment expressions.

variable = **variable operator** expression; ⟺ **variable operator**= expression;

(+, -, *, /, %)

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

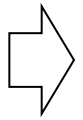
```
#include<iostream>
int main() {
  int a{2};
  double b{3};
  double c{4};
  c *= (a + b); // or c *= a + b
  std::cout << "c = " << c << std::endl;
}
```

Increment and Decrement Operators

C++ provides two unary operators ++, -- for adding 1 to or subtracting 1 from the value of a **numeric variable** to simplify program statements.

Pre-incrementing Using Prefix Increment Operator	$++v$	Increment v by 1, then use the **new value** of v in the expression in which v resides.
Pre-decrementing Using Prefix Decrement Operator	$--v$	Decrement v by 1, then use the **new value** of v in the expression in which v resides.
Post-incrementing Using Postfix Increment Operator	$v++$	Use the **current value** of v in the expression in which v resides, then increment v by 1.
Post-decrementing Using Postfix Decrement Operator	$v--$	Use the **current value** of v in the expression in which v resides, then decrement v by 1.

- When incrementing or decrementing a variable in a statement by itself, the prefix increment and postfix increment forms have the same effect, and the prefix decrement and postfix decrement forms have the same effect.



same effect

```
v = v + 1;
v += 1;
++v;
v++;
```

- Writing ++(x + 1) is a syntax error, because (x + 1) is not a variable.

Prefix Increment and Postfix Increment Operators

Example 1:

```
#include <iostream>
int main() {
  unsigned int c{5}; // initializes c with the value 5
  std::cout << "c before postincrement: " << c << "\n"; // prints 5
  std::cout << "postincrementing c: " << c++ << "\n"; // prints 5
  std::cout << "c after postincrement: " << c << "\n"; // prints 6
  std::cout << "\n"; // skip a line
  c = 5; // assigns 5 to c
  std::cout << "c before preincrement: " << c << "\n"; // prints 5
  std::cout << "preincrementing c: " << ++c << "\n"; // prints 6
  std::cout << "c after preincrement: " << c << std::endl; // prints 6
}
```

Example 2:

```
#include <iostream>
int main() {
  int x1{10};
  int y1;
  y1 = 2 * (++x1) + 5;
  std::cout << "x1 = " << x1 << ", y1 = " << y1 << "\n"; // Prints x1 = 11, y1 = 27
  int x2{10};
  int y2;
  y2 = 2 * (x2++) + 5;
  std::cout << "x2 = " << x2 << ", y2 = " << y2 << "\n"; // Prints x2 = 11, y2 = 25
}
```

Operator Precedence and Associativity

decreasing order of precedence

Operators	Associativity	Type
:: ()	left to right *[For nested parentheses: from innermost pair]*	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

- If you are uncertain about the order of evaluation in a complex expression, break the expression into smaller statements or use parentheses to force the order of evaluation, exactly as you would do in an algebraic expression.