CIS 22A – Lecture 4

Manish Goel

The `cin` object

- Base C++ object – to receive input from the user.

- Part of the `iostream` library – included in the first line of the program, otherwise there will be no interaction with the program.

- The stream insertion operator `>>` is used to receive characters or text in the `cin` stream. <enter> key signifies end of input on keyboard.

The `cin` object - 2

- `cin` converts input data to match type of variable

```
int age;
cin >> age;   // stores as integer
string age;
cin >> age;   // stores as string
```

- `cin` can receive multiple inputs on same line – multiple values from keyboard should be separated by space.

```
int age, height, weight;
cin >> age >> height >> weight;
```

On keyboard enter: `12 60 54`

Displaying a Prompt

- Gather input in 2 steps – prompt before input

- A prompt is a message that instructs the user to enter data.

- You should always use **`cout`** to display a prompt before each **`cin`** statement.

```
cout << "How tall is the room? ";
cin >> height;
```

Mathematical Expressions

- Simple expressions have single math operators
- Complex expressions have multiple math operators

- An expression can be a literal, a variable or a combination of constants and variables

- Expressions are used in
  - Assignment `area = PI*rad*rad`
  - Output `cout << “Circumference = “ << 2*(l+w)`
  - Other statements

Operator Precedence Order

- For computing operations, usual mathematical operator precedence is followed:
  - P E D M A S   or   P E M D A S
  - Parentheses before Exponents before (Division or Multiplication) before (Addition or Subtraction)

  - 2*4 + 2*5 = 8 + 10 = 18 → Multiplication before addition
  - 2 * (4 + 5) = 2 * 9 = 18 → Parenthesis resolved first

  - 2 * 5 * 5 = 50 → Straight multiplication
  - 2 * 5^2 = 2 * 25 = 50 → Exponent first

Order of Operations - 2

- With negation and modulus, evaluate in this order:
  - – (unary negation) → in order, left to right
  - `*  /  %` → in order, left to right
  - `+  –` → in order, left to right
- Associativity of operators is evaluated as:
  - – (unary negation) → associates right to left
  - `*,  /,  %,  +,  –` → associate left to right
  - parentheses ( ) can be used to override the order of operations:

```
 2 + 2  *  2 – 2  = 4
(2 + 2) *  2 – 2  = 6
 2 + 2  * (2 – 2) = 2
(2 + 2) * (2 – 2) = 0
```

Algebraic Expressions

- Convert algebraic expressions based on operator precedence

- Multiplication requires an operator
  - `r = 2(3+5)` is written as `r = 2*(3+5);`

- C++ provides a function to perform exponents
  - $A = s^2$ is written as `A = pow(s,2);`
  - This function accepts and returns `float` or `double`
  - Include math library : `#include <cmath>`

- Parentheses help maintain order of operations
  - $m = \frac{y2 - y1}{x2 - x1}$ is written as `m = (y2 - y1) / (x2 - x1);`

Type Conversions – Apples & Oranges

- Operations are performed between operands of same type

- If different types, C++ performs type conversion to match them but results can be impacted

- Type Conversion governed by type hierarchy – ranked by largest number they can hold

  Highest : `long double`
  `double`
  `float`
  `unsigned long`
  `long`
  `unsigned int`
  Lowest: `int`

Type Coercion – Automatic Conversion

- Promotion – Type converted to higher-type
- Demotion – Type converted to lower-type

- Rule 1: When an operator works on operands of two types, lower ranking type value is promoted to higher ranking type

- Rule 2: `chars`, `shorts`, `unsigned shorts` always promoted to `int` except if `short` is same size as `int`

- Rule 3: Assignment (=) expressions change type of RHS (or expression) to type of LHS (or variable)

Overflow and Underflow

- Overflow – On assigning value larger than max for data-type
- Underflow – On assigning value lower than min for data-type

- In either of these conditions, system may show error message, stop on error or continue with logical error.

- For integers – overflow causes wrap around to lowest value, underflow causes wrap around to highest value – think of a car odometer as analogy

  Max Value of 4-bit unsigned integer is ‘15’ (1111)

  Adding 1 sets it to ‘0’ (0000) – overflow

  Similarly, Min Value of 4-bit signed integer is ‘-8’

  Subtracting 1 sets it to 7 – underflow

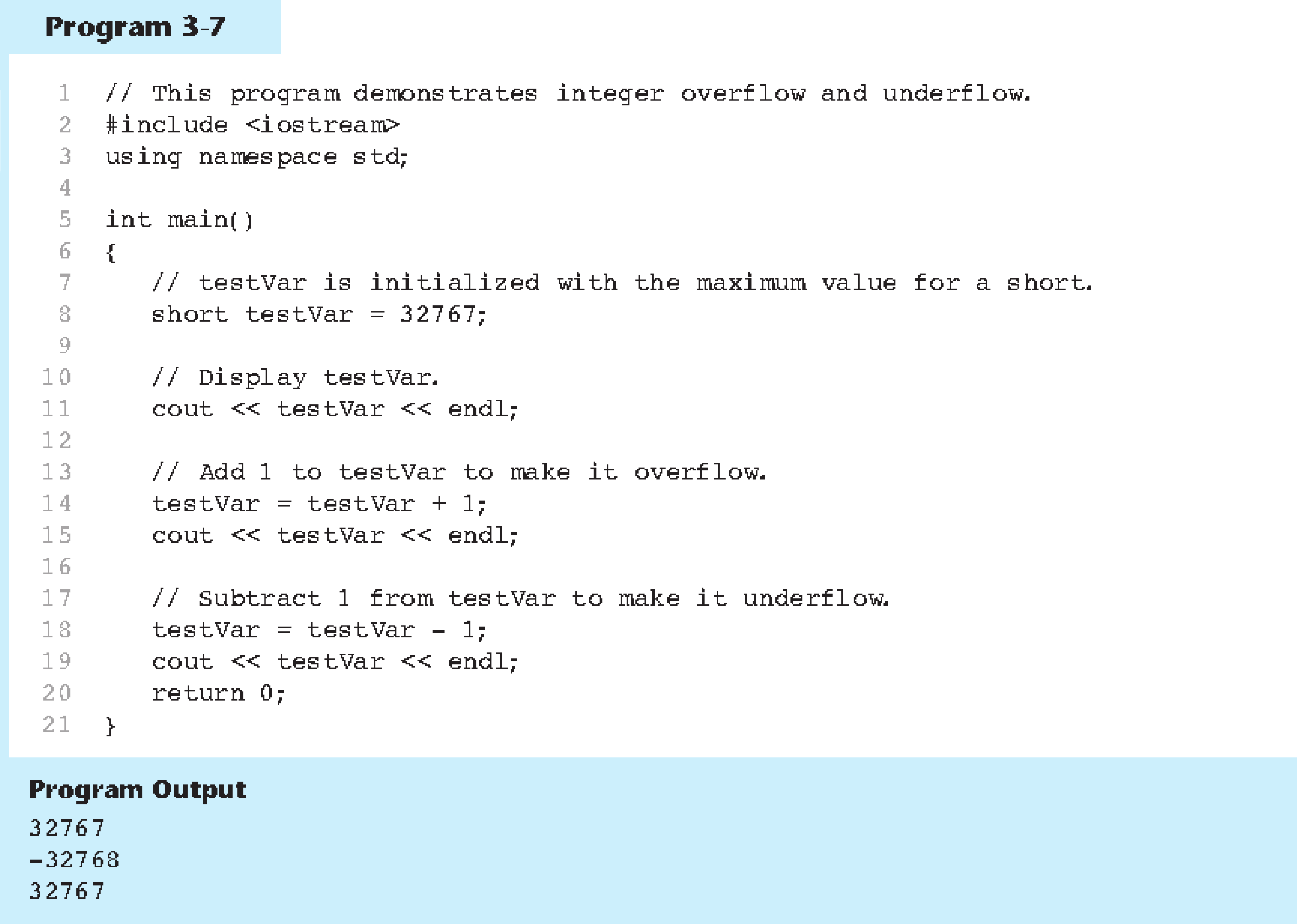
Overflow / Underflow with 4-bit data

Unsigned Int	Signed Int	Binary		Unsigned Int	Signed Int	Binary
0	0	0000		8	-8	1000
1	1	0001		9	-7	1001
2	2	0010		10	-6	1010
3	3	0011		11	-5	1011
4	4	0100		12	-4	1100
5	5	0101		13	-3	1101
6	6	0110		14	-2	1110
7	7	0111		15	-1	1111

```
For unsigned ints:
 0000 → 0    1111 → 15
-   1       +   1
_____       _____
 1111 → 15   0000 → 0
```

```
For signed ints:
 1000 → -8   0111 → 7
-   1       +   1
_____       _____
 0111 → 7    1000 → -8
```

Wrap Around – Overflow/Underflow



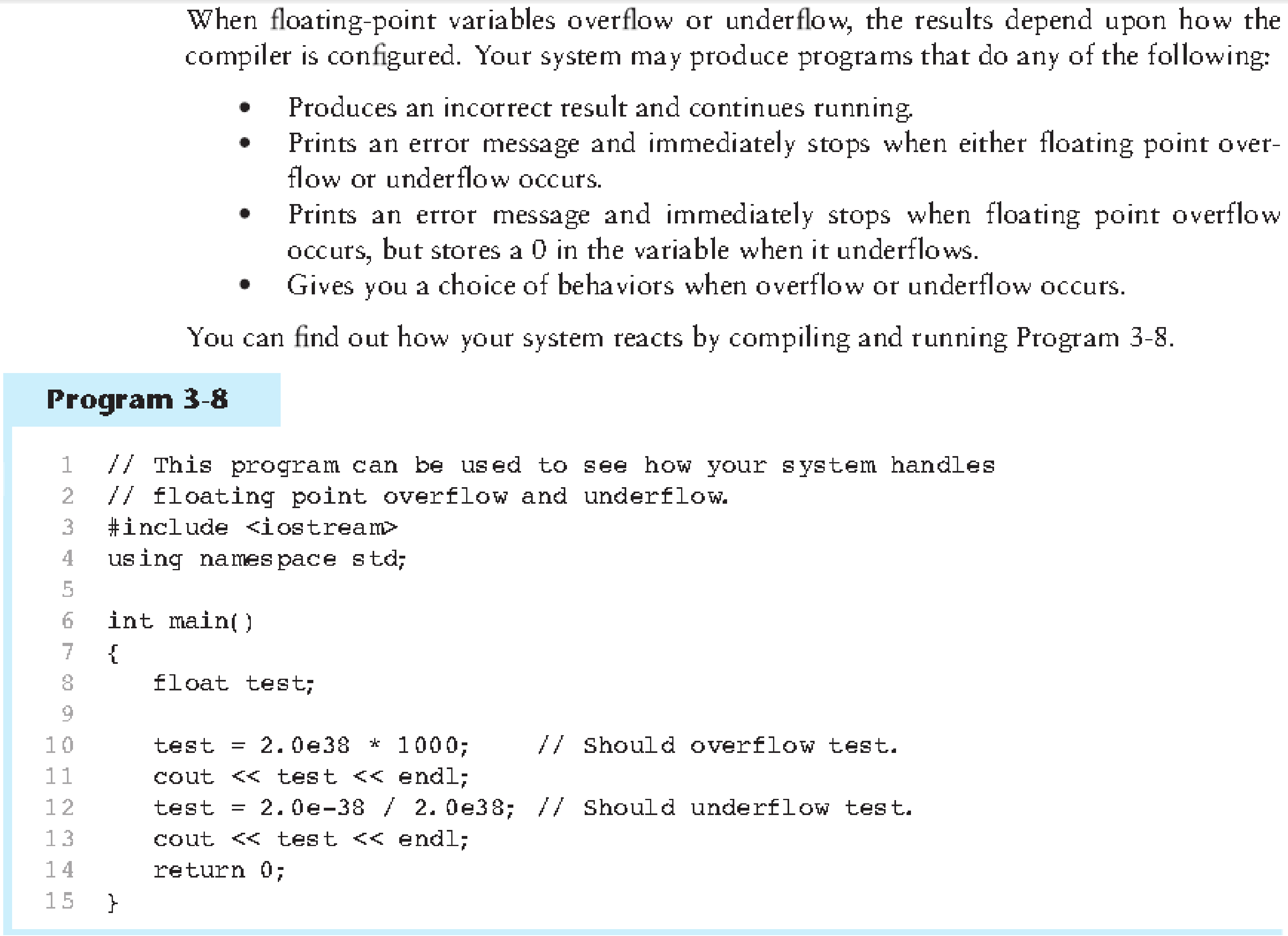
Program 3-7

```
// This program demonstrates integer overflow and underflow.
#include <iostream>
using namespace std;

int main()
{
    // testVar is initialized with the maximum value for a short.
    short testVar = 32767;

    // Display testVar.
    cout << testVar << endl;

    // Add 1 to testVar to make it overflow.
    testVar = testVar + 1;
    cout << testVar << endl;

    // Subtract 1 from testVar to make it underflow.
    testVar = testVar - 1;
    cout << testVar << endl;
    return 0;
}
```

Program Output

```
32767
-32768
32767
```

Overflow/Underflow – Floating Point



When floating-point variables overflow or underflow, the results depend upon how the compiler is configured. Your system may produce programs that do any of the following:

- Produces an incorrect result and continues running.
- Prints an error message and immediately stops when either floating point overflow or underflow occurs.
- Prints an error message and immediately stops when floating point overflow occurs, but stores a 0 in the variable when it underflows.
- Gives you a choice of behaviors when overflow or underflow occurs.

You can find out how your system reacts by compiling and running Program 3-8.

Program 3-8

```
// This program can be used to see how your system handles
// floating point overflow and underflow.
#include <iostream>
using namespace std;

int main()
{
   float test;

   test = 2.0e38 * 1000;     // Should overflow test.
   cout << test << endl;
   test = 2.0e-38 / 2.0e38;  // Should underflow test.
   cout << test << endl;
   return 0;
}
```

Type Casting – User Defined Conversion

- Used for manual data type conversion

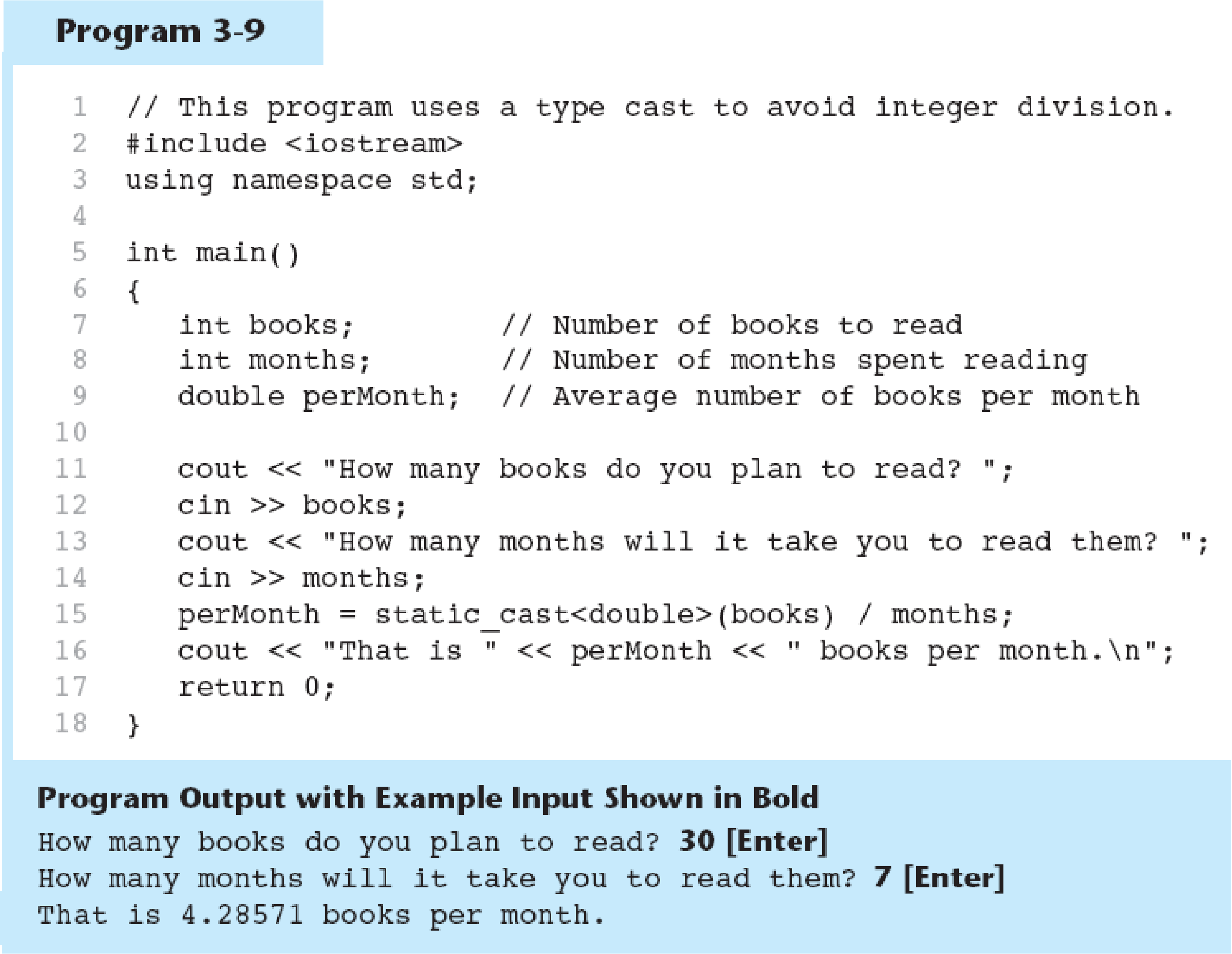
- Useful for floating point division using ints:

```
double m;
int x1, x2, y1, y2;
m = static_cast<double>(y2-y1)
                      /(x2-x1);
```

- Useful to see `int` value of a `char` variable:

```
char ch = 'C';
cout << ch << " is "
    << static_cast<int>(ch);
```

Type Casting Sample 1



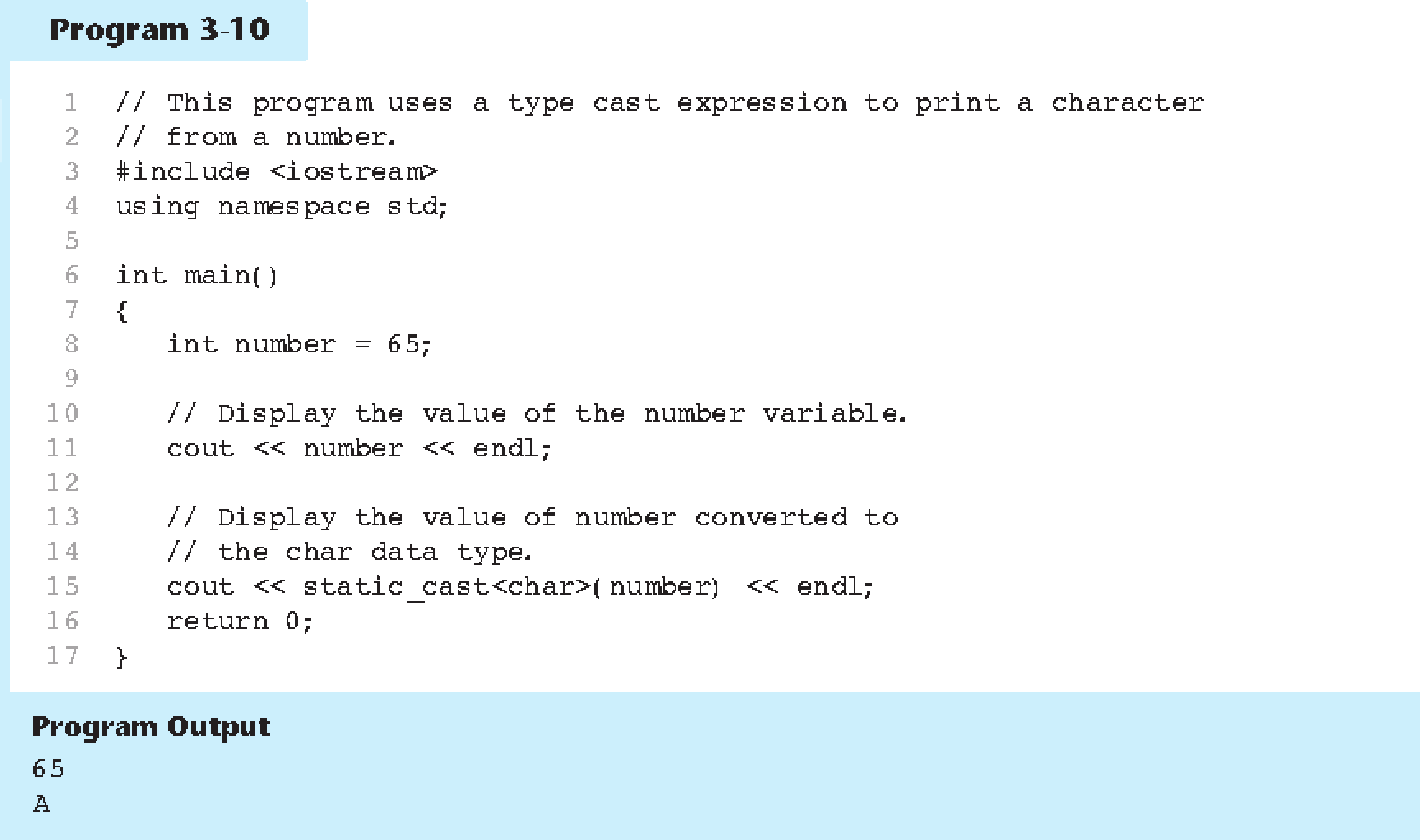
Program 3-9

```
// This program uses a type cast to avoid integer division.
#include <iostream>
using namespace std;

int main()
{
   int books;          // Number of books to read
   int months;         // Number of months spent reading
   double perMonth;    // Average number of books per month

   cout << "How many books do you plan to read? ";
   cin >> books;
   cout << "How many months will it take you to read them? ";
   cin >> months;
   perMonth = static_cast<double>(books) / months;
   cout << "That is " << perMonth << " books per month.\n";
   return 0;
}
```

Program Output with Example Input Shown in Bold

```
How many books do you plan to read? 30 [Enter]
How many months will it take you to read them? 7 [Enter]
That is 4.28571 books per month.
```

Type Casting Sample 2



Program 3-10

```
// This program uses a type cast expression to print a character
// from a number.
#include <iostream>
using namespace std;

int main()
{
   int number = 65;

   // Display the value of the number variable.
   cout << number << endl;

   // Display the value of number converted to
   // the char data type.
   cout << static_cast<char>(number) << endl;
   return 0;
}
```

Program Output

```
65
A
```

C-Style and Prestandard Type Cast Expressions

- C-Style cast: data type name in `()`

```
cout << ch << " is " << (int)ch;
```

- Prestandard C++ cast: value in `()`

```
cout << ch << " is " << int(ch);
```

- Both are still supported in C++, although `static_cast` is preferred

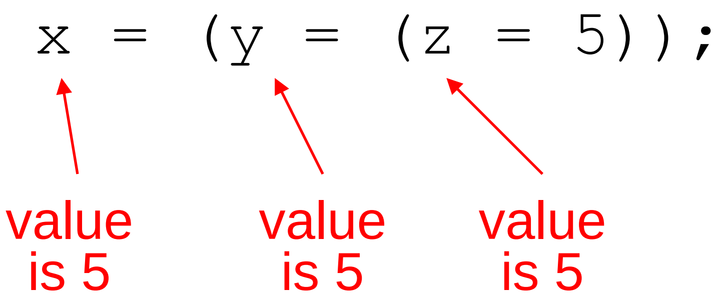
Multiple Assignments

- The = can be used to assign a value to multiple variables:

```
x = y = z = 5;
```

- Value of = is the value that is assigned

- Associates right to left:



```
x = (y = (z = 5));
```

value is 5

value is 5

value is 5

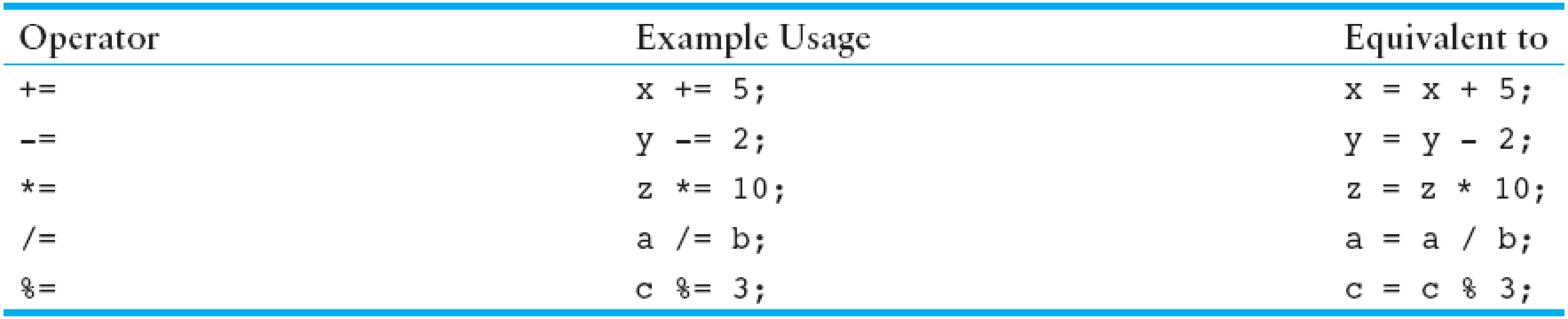
Combined Assignment

- `sum = sum + 1;` // adds 1 to the variable **`sum`**
- `res = res * 2;` // multiplies variable **`res`** by 2

- Combined assignment operators provide shorthand for such operations.

- `sum = sum + 1;` → `sum += 1;`
- `Res = res * 2;` → `res *= 2;`

Table 3-9



Operator	Example Usage	Equivalent to
`+=`	`x += 5;`	`x = x + 5;`
`-=`	`y -= 2;`	`y = y - 2;`
`*=`	`z *= 10;`	`z = z * 10;`
`/=`	`a /= b;`	`a = a / b;`
`%=`	`c %= 3;`	`c = c % 3;`