Operators and Converting Between Data Types

Principles of Computer Programming I

Spring/Fall 20XX

AUGUSTA UNIVERSITY

Outline

- Arithmetic and Assignment Operators
- Data Types and Literal Assignment
- Implicit Conversions
- Explicit Conversions
- Arithmetic on Mixed Data Types
  - Order of operations

Doing Arithmetic

- C# has math operators for numeric data

Operation	C# Operator	Algebraic Expression	C# Expression
Addition	+	$x + 7$	`myVar + 7`
Subtraction	–	$x - 7$	`myVar - 7`
Multiplication	*	$x \cdot 7$	`myVar * 7`
Division	/	$\frac{x}{7}, x/7, x \div 7$	`myVar / 7`
Remainder	%	$x \bmod 7$	`myVar % 7`

Remainder after integer division: $44 \bmod 7 = 2$ because $44 \div 7 = 6$ with remainder 2

Arithmetic and Assignment

- Result of an arithmetic expression is a numeric value
- Numeric values can be assigned to variables

```
int myVar = 3 * 4;
```

The value 12 is stored in myVar

This expression evaluates to 12

- Variable in arithmetic expression = read its current value

```
int a = 4;
int b = a + 5;
a = b * 2;
```

a has value 4, so b gets the resulting value 9

b has value 9, now a is assigned new value 18

Self-Assignment

- A variable can appear on both sides of the = operator

What does this do?

```
int myVar = 4;
myVar = myVar * 2;
```

Store result of 4 * 2 into myVar

Read myVar's current value, 4

- Compound assignment operators: a shortcut

Statement	Equivalent
`x += 2;`	`x = x + 2;`
`x -= 2;`	`x = x - 2;`
`x *= 2;`	`x = x * 2;`
`x /= 2;`	`x = x / 2;`

Outline

- Arithmetic and Assignment Operators
- Data Types and Literal Assignment
- Implicit Conversions
- Explicit Conversions
- Arithmetic on Mixed Data Types
  - Order of operations

Review: Numeric Data Types

- Integers

Size & range

Type	Size	Range of Values
`short`	2 bytes	$-2^{15} \ldots 2^{15} - 1$
`int`	4 bytes	$-2^{31} \ldots 2^{31} - 1$
`long`	8 bytes	$-2^{63} \ldots 2^{63} - 1$

Type	Size	Range of Values
`ushort`	2 bytes	$0 \ldots 2^{16} - 1$
`uint`	4 bytes	$0 \ldots 2^{32} - 1$
`ulong`	8 bytes	$0 \ldots 2^{64} - 1$

- Floating-point

Type	Size	Range of Values	Digits of Precision
`float`	4 bytes	$\pm 1.5 \cdot 10^{-45} \ldots \pm 3.4 \cdot 10^{38}$	7
`double`	8 bytes	$\pm 5.0 \cdot 10^{-324} \ldots \pm 1.7 \cdot 10^{308}$	15-16
`decimal`	16 bytes	$\pm 1.0 \cdot 10^{-28} \ldots \pm 7.9 \cdot 10^{28}$	28-29

Assignment From Literals

- If literal type matches variable type, assignment always works:

```
int myAge = 29;
double myHeight = 1.77;
float radius = 2.3f;
```

- What if literal type is different?

```
float radius = 2.3;
```

Error! Can't convert `double` to `float`

```
float radius = 2;
```

No error, even though 2 is an `int` literal

Why does this work?

Implicit Conversions

- Value type must still match variable type
- Some types can be **implicitly converted** to others:

```
float radius = 2;
```

int value

implicit conversion

radius ← 2.0f ← float value

- Also applies to assignment from variables:

```
int length = 2;
float radius = length;
```

value 2 implicitly converted to 2.0f

gets value 2.0f

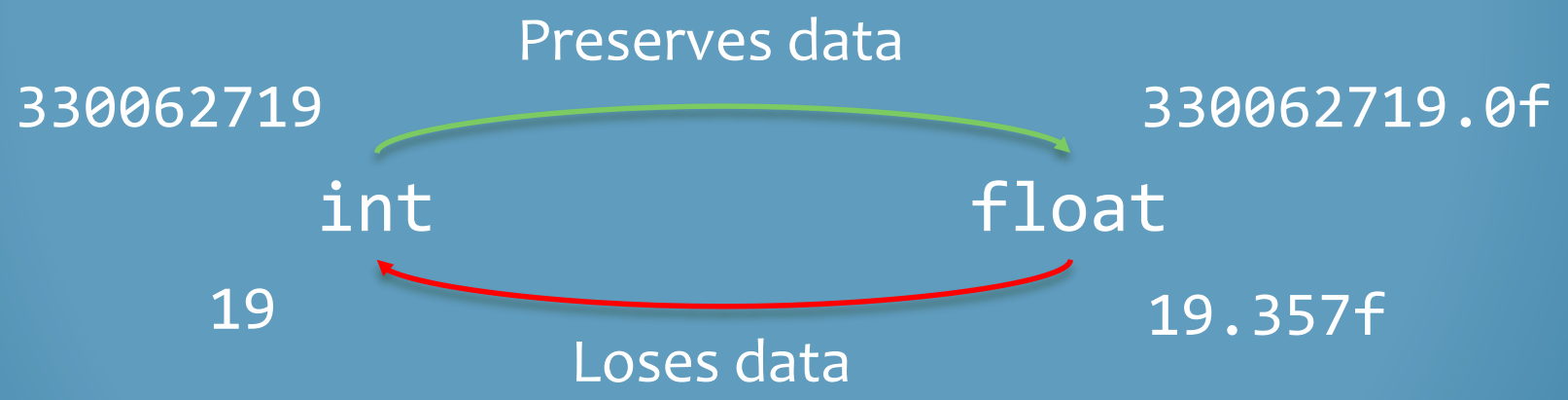
Implicit Conversions

Type	Possible Implicit Conversions
short	int, long, float, double, decimal
int	long, float, double, decimal
long	float, double, decimal
ushort	uint, int, ulong, long, decimal, float, double
uint	ulong, long, float, double, decimal
ulong	float, double, decimal
float	double

What's the pattern here? Given a type, what can you implicitly convert it to?

Implicit Conversions are "Safe"

- `int` range: $-2^{31} \ldots 2^{31} - 1$; `float` integer range: $\pm 3.4 \cdot 10^{38}$
- Any `int` can be stored in a `float` **without losing data**
- Reverse is not safe: Storing `4.7f` in an `int` will lose the fraction

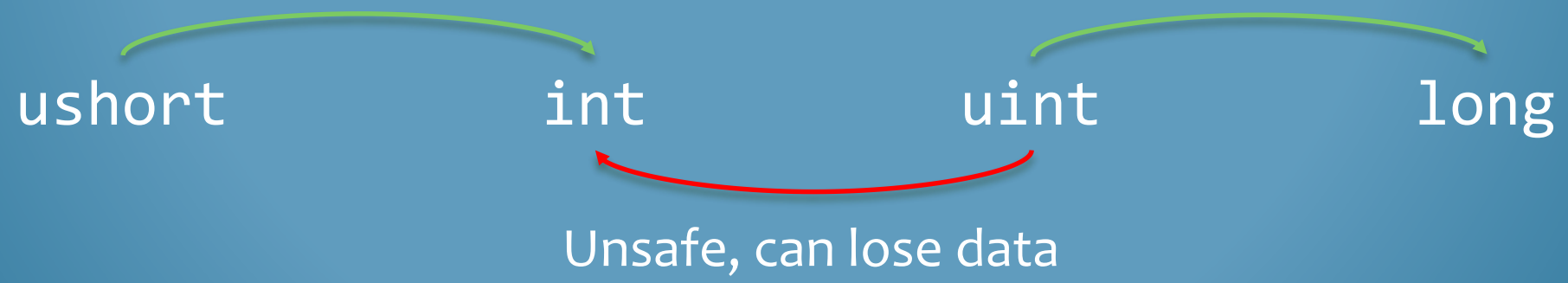


- All integer types are safe to convert to `float` or `double`

Other Safe Conversions

- Smaller integer to larger integer; `float` to `double`

- Unsigned to *larger* signed integer (why larger?)



Outline

- Arithmetic and Assignment Operators
- Data Types and Literal Assignment
- Implicit Conversions
- **Explicit Conversions**
- Arithmetic on Mixed Data Types
  - Order of operations

Data-Losing Conversions

- Unsafe conversions: Potential to lose data

19.35499674	19.354998f	19
double	float	int

Loses significant figures

Loses fractional part

- Will not happen automatically; compile error

```
double length = 2.886;
float radius = length;
```

Error! Can't convert `double` to `float`

Explicit Conversion with Casts

- Cast operator: Force the compiler to allow an unsafe conversion

```
double length = 2.886;
float radius = (float) length;
```

gets value `2.886f`

Target type name in parentheses

`double` value

- Explicit conversion from original to target type must exist
  - Most built-in C# types have explicit conversions defined

```
string strAge = "29";
int myAge = (int) strAge;
```

Error! Can't convert `string` to `int`

Casting Side-Effects

- Casting from floating-point to integer: fraction is *truncated*

```
int intLength = (int) length;
```

gets value 2

cast to int

value: 2.886

- Casting to less precise floating-point: fraction is *rounded*

```
decimal myDecimal = 123456789.999999918m;
double myDouble = (double) myDecimal;
float myFloat = (float) myDouble;
```

myDouble: 123456789.99999993

myFloat: 123456790.0f

Casting Side-Effects

- Casting to smaller integer: Most significant *bits* are truncated

```
int bigNumber = (int) 9223372036854775807;
```

$2^{63} - 1$, all 1's with sign bit 0

gets value -1: all 1's with sign bit 1

long value

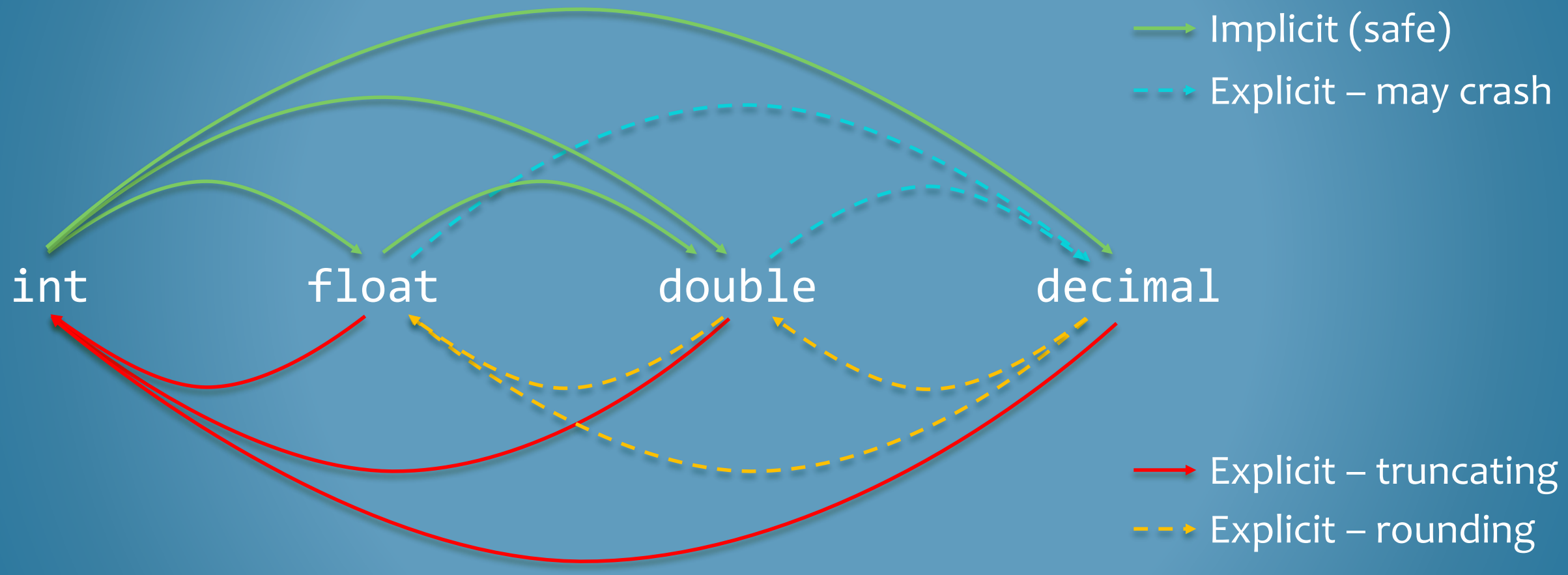
- Casting to `decimal`: Stored precisely, *unless* it is out of range – crashes with a `System.OverflowException`

```
decimal fromSmall = (decimal) 42.76875;
double bigDouble = 2.65e35;
decimal fromBig = (decimal) bigDouble;
```

fromSmall: `42.76875m`

`System.OverflowException`!

Common Conversions Summary



Outline

- Arithmetic and Assignment Operators
- Data Types and Literal Assignment
- Implicit Conversions
- Explicit Conversions
- **Arithmetic on Mixed Data Types**
  - Order of operations

Integer vs. Fractional Arithmetic

- Math operators (+, -, *, /, %) are defined separately for each type

Value: 7

Result of expression: 7

```
int sum1 = 2 + 5;
```

int version of + operator

Result of expression: 5.5

```
double sum2 = 2.25 + 3.25;
```

double version of + operator

- The `int` / operator does *integer division* – remainder is dropped

Result of expression: 4

```
int intDiv = 21 / 5;
```

int version of / operator

Result of expression: 4.2

```
double fracDiv = 21.0 / 5.0;
```

double version of / operator

Operators For Each Type

- Binary operators, like +, -, *, /, are really 2-input functions
- Result type of function depends on input types:

```
int + int        ⟶ int
float + float    ⟶ float
double + double  ⟶ double
decimal + decimal ⟶ decimal
```

- Action taken (function code) also depends on input types:

`int / int` ⟶ integer division ⟶ `int`

`double / double` ⟶ floating-point division ⟶ `double`

Implicit Conversions in Math

- Arguments to a math operator must be the same type
- If they are not, implicit conversion will happen
  - Less-precise argument gets converted to more-precise type

`int` value

`double` value

```
double fracDiv = 21 / 2.4;
```

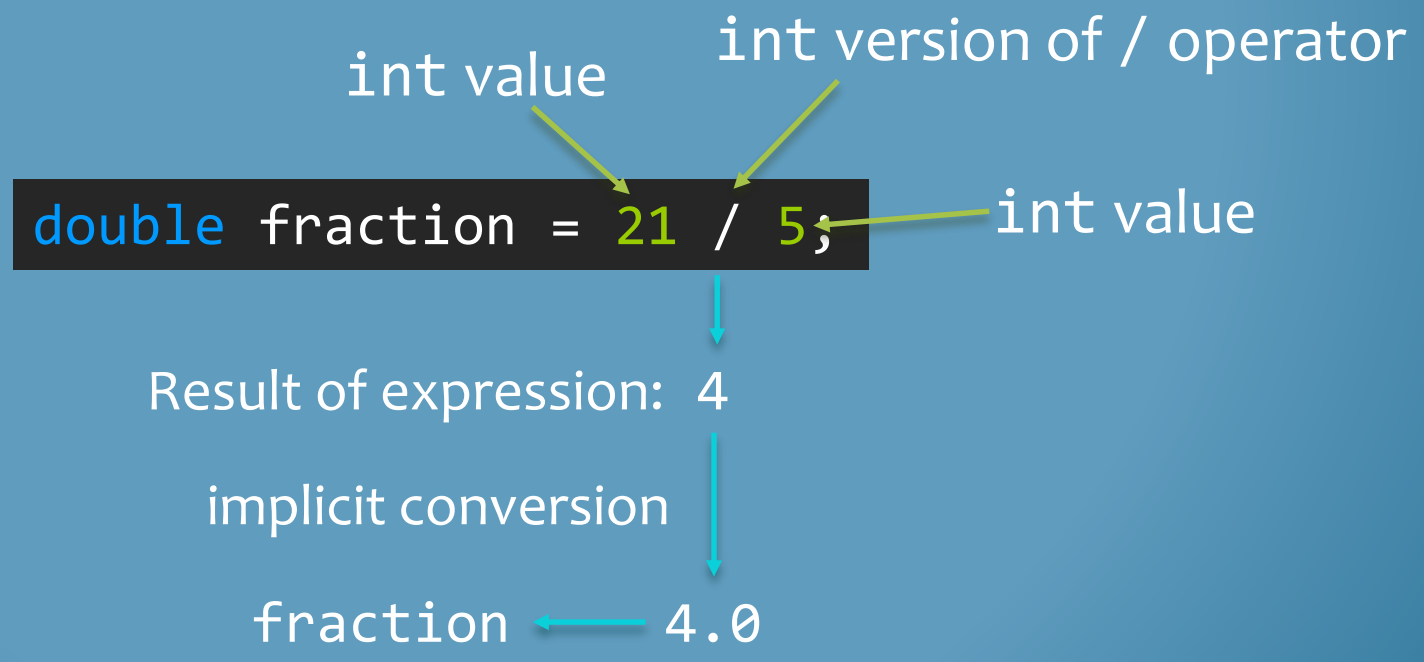
implicit conversion

Result: `8.75` ← `21.0 / 2.4`

`double` version of / operator

Implicit Conversions in Math

- Result type of expression determined by arguments
- Can be implicitly converted to store in a variable
  - This happens *after* computing the result



Casting to Prompt Conversion

- What if you don't want integer division?

```
int numCookies = 21;
int numPeople = 6;
double share = numCookies / numPeople;
```

int version of / operator

share: 3.0

- Make one argument a double using casting:

double value

implicitly converted to double

```
double share = (double) numCookies / numPeople;
```

share: 3.5

double version of / operator

Casting in Math Expressions

- Casting may be **necessary** to make types match:

```
double a = 35.0;
decimal b = 0.5m;
decimal result = (decimal) a * b;
```

decimal * decimal operator

double can't implicitly convert to decimal

- Casting may be **desirable** to change which operator runs:

```
int numCookies = 21;
int numPeople = 6;
double share = (double) numCookies / numPeople;
```

implicitly converted to double

share: 3.5

explicitly convert from int to double

double / double operator

Order of Operations

- C# math operators follow standard PEMDAS order, left-to-right

```
int x = 4 + 10 * 3 - 21 / 2 - (3 + 3);
```

4 + 30 - 10 - 6 → 18

- Cast operator is **higher priority** than binary (math) operators

```
double share = (double) numCookies / numPeople;
```

1. Cast `int` to `double`
2. Implicitly convert to `double` so operands match types
3. Execute / operator

Order of Operations

- What will each of these results be? What is the difference?

```
int a = 5, b = 4;
double result;
result = a / b;
result = (double) a / b;
result = a / (double) b;
result = (double) a / (double) b;
result = (double) (a / b);
```

Parentheses: Highest priority

`int / int` operator

Summary

- Arithmetic and Assignment Operators
- Data Types and Literal Assignment
- Implicit Conversions
- Explicit Conversions
- Arithmetic on Mixed Data Types
  - Order of operations

Miscellaneous Syntax Note

- Can declare multiple variables on one line:

```
double length, depth, height;
```

comma

equivalent

```
double length;
double depth;
double height;
```

- Can combine declarations and initializations:

```
int age = 29, weight, votes;
```

equivalent

```
int age = 29;
int weight;
int votes;
```