

OPERATING WITH DATA - INTRODUCTION

- ❑ Operators specify what should be done to variables and constants
- ❑ There are three types of operators
 - **Unary** operators have one argument and may be **prefix** or **postfix**, e.g.
! ++ --
 - **Binary** operators have two arguments and are **infix**, e.g.
+ & & !=
 - **Tertiary** operators have three arguments, e.g.
?:
- ❑ Expressions combine variables and constants with operators to produce a new value
 - the data types for elements of the expression should be the same or can be converted to be the same
 - More than one operator can be used in a single expression
 - e.g. a+b*c/d

ARITHMETIC OPERATORS

- ❑ C++ has the usual set of arithmetic operators
- * / +
- ❑ **Operands** for these binary operators may be any of the basic data types. However, operands for the modulus (remainder) operator
%
must be int or char
- ❑ **Important:** Note that integer division truncates any fractional part

INTEGER DIVISION EXAMPLE

```
int    digit;
float  num;

// num is equal to 2.0 not 2.5
num = 5 / 2;
// digit is equal to 2 not 2.5
digit = 5.0 / 2.0;
// num is equal to 1.0
num = 5 % 2;
// num is equal to 2.5, correct way!
num = 5.0 / 2.0
```

OPERATOR PRECEDENCE

- ❑ Some operators have higher precedence than others, e.g.
 $a + b * c$ means $a + (b * c)$
because $*$ has higher precedence than $+$
- ❑ Order of precedence:
 - $*$ $/$ are the same
 - $+$ $-$ are the same
- ❑ **Note** that we can use brackets $()$ to force precedence

ASSOCIATIVITY OR GROUPING

- Given expression with operators of equal precedence, associativity is the order used to evaluate the expression

Either: **left to right** or **right to left**

* / + - are all left to right associative

Therefore

$a + b - c + d$
means
 $((a + b) - c) + d$

RELATIONAL, EQUALITY AND LOGICAL OPERATORS

- relational > >= < <= greater precedence than
equality == != greater precedence than
logical and && greater precedence than
logical or ||

- Relational operators have lower precedence than arithmetic operators

$I < x - 1$ means $I < (x - 1)$

- Important: logical expressions evaluated left to right

Evaluation stops as soon as truth or falsehood is known

$I > \text{lower_limit} \ \&\& \ I < \text{upper_limit}$

Value of relational expression is 1 if TRUE and 0 if FALSE

RELATIONAL, EQUALITY AND LOGICAL OPERATORS

□ Unary Negation Operator (!)

converts non zero operand to zero, and zero to one

□ Increment and Decrement Operators (Auto Operators)

++ --

- increment and decrement **variable** by 1
- may be used prefix or postfix

```
int i = 1;
```

```
++i;
```

or

```
int i = 1;
```

```
i++;
```

- Important difference:

```
int i, n;
```

```
i = 5; n = ++i;
```

```
// n has value 6, i has value 6
```

```
int i, n;
```

```
i = 5; n = i++;
```

```
// n has value 5, i has value 6
```

ASSIGNMENT OPERATORS

□ Example: `i = i + 1;`

□ C++ provides shorthand which transforms

```
var = var op (expr);
```

to

```
var op= expr;
```

Therefore `i = i + 2;` becomes `i += 2;`

Used for binary operators

`+ - * / % & | ^ << >>`

BIT WISE OPERATORS

<code>&</code>	<code> </code>	<code>^</code>
AND	inclusive OR	exclusive OR
<code><<</code>	<code>>></code>	<code>~</code>
shift left	shift right	one's complement

□ Note that if x is 1 and y is 2 `x&y` equals 0 whereas `x&&y` equals 1

- `x << 2` shifts value of x 2 bit to left filling vacated bits with zero
- `~` for integer: converts 0-bits to 1-bit and 1-bits to 0-bit

OPERATORS IN USE

```
x++;  
-y;  
z = x++;  
z = ++x;  
z = y + x++;  
z = y + ++x;  
x += 10;  
y *= 3;  
a = b && c;  
b = !c;  
m = m & 0xff;
```

PRECEDENCE SUMMARY

Operator	Associativity
() [] -> .	left to right
! ~ ++ -- * & sizeof + - *	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
= = !=	left to right
&	left to right
^	left to right
	left to right
& &	left to right
	left to right
?:	right to left
= += -= *= /= %= &= ^= = <<= >>=	right to left
,	left to right

Programming 1

Lecture 5 -- 11

TYPE COERCION AND TYPE CASTING

- ❑ Integer values and floating point values are stored differently inside a computer's memory.

What happens if different numerical types can be combined in an expression, **mixed type** (or **mixed mode**) expressions?

Example:

```
int    someInt;
float  someFloat;
someFloat = 12;    // 12.0 is stored in someFloat
```

- ❑ The implicit (automatic) conversion of a value from one data type to another is known as **type coercion**.
- ❑ To make our programs as clear (and error-free) as possible, we can use explicit **type casting** (or **type conversion**)

Programming 1

Lecture 5 -- 12

TYPE COERCION: PROMOTION

□ Arithmetic and Relational Expressions:

If the two operands are of different data types, then one of them is temporarily **promoted** (or **widened**) to match the data type of the other

Step 1: Each char, short, or enumeration value is promoted to int. If both operands are now int, the result is an int expression

Step 2: If Step 1 still leaves a mixed-type expression, the following precedence of types is used:

lowest \longrightarrow highest

int, unsigned int, long, unsigned long, float, double, long double

TYPE COERCION: DEMOTION

□ Assignments, Parameter Passage, and Return of a Function Value:

Demotion is the conversion of a value from a “higher” type to a “lower” type according to a programming language’s precedence of data types.

Demotion may cause loss of information.

$V = E$ where V is a variable and E is an expression:

1. If the types of V and E are the same, no type coercion is necessary
2. If the type of V is “higher” than that of E, then the value of E is promoted to V’s type before being stored into V
3. If the type of V is “lower” than that of E, the value of E is demoted to V’s type before being stored into V

TYPE CASTING

- Instead of relying on implicit type coercion, it is recommended to use **type casting** to show that the conversion is intentional.

Example:

```
int    someInt = 3;
float  someFloat = 15.0;
someInt = int(someFloat);
```

In C++ the cast operation comes into two forms:

```
someInt = int(someFloat);    // Functional notation
someInt = (int) someFloat;   // Prefix notation
```