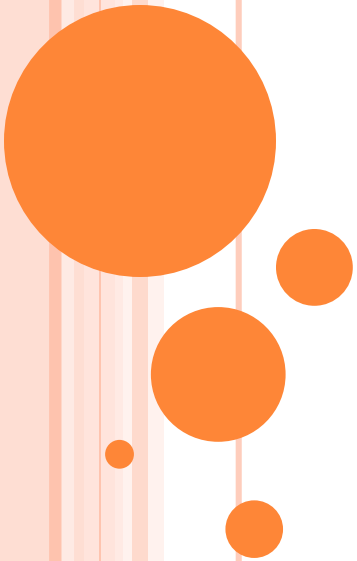

Chapter 6

Expressions and Assignment Statements

Chapter 7 Topics

- Introduction
- Arithmetic Expressions
- Overloaded Operators
- Type Conversions
- Relational and Boolean Expressions
- Short-Circuit Evaluation
- Assignment Statements
- Mixed-Mode Assignment

Introduction

- Expressions are the fundamental means of specifying computations in a programming language
- To understand expression evaluation, need to be familiar with the orders of operator and operand evaluation – which are dictated by the associativity and precedence rules
- Essence of imperative languages is dominant role of assignment statements

Arithmetic Expressions

- Arithmetic evaluation was one of the motivations for the development of the first programming languages
- Arithmetic expressions consist of operators, operands, parentheses, and function calls
- Most binary operators are infix i.e. operators appear between operands
- Computation: fetching the operand & executing the arithmetic

Arithmetic Expressions: Design Issues

- Design issues for arithmetic expressions
  - operator precedence rules
  - operator associativity rules
  - order of operand evaluation
  - operand evaluation side effects
  - operator overloading
  - mode mixing expressions

Arithmetic Expressions: Operators

- A unary operator has one operand

  y++
- A binary operator has two operands

  x +y
- A ternary operator has three operands

  y > 5 ? 1 : 0

Arithmetic Expressions: Operator Precedence Rules

- The *operator precedence rules* for expression evaluation define the order in which "adjacent" operators of different precedence levels are evaluated
- Typical precedence levels

	Ruby	C-based	Ada
Highest	**	Postfix ++, --	**, bas
	Unary +. -	Prefix ++, --, unary +, -	*, /, mod, rem
	*, /, %	*, /, %	Unary +, -
Lowest	Binary +, -	Binary +, -	Binary +, -

Arithmetic Expressions: Operator Associativity Rule

- The *operator associativity rules* for expression evaluation define the order in which adjacent operators with the same precedence level are evaluated
- Typical associativity rules
  - Ruby – Left: *, /, +, -
    Right: **
  - C-based – Left: Left: *, /, %, binary +, binary
    Right ++, --, unary +, unary –
  - Ada – Left: all except **
    Non-associative: **
- APL is different; all operators have equal precedence and all operators associate right to left
- Precedence and associativity rules can be overridden with *parentheses*

Arithmetic Expressions: Conditional Expressions

- Conditional Expressions
  - C-based languages (e.g., C, C++)
  - An example:

```
average = (count == 0)? 0 : sum / count
```

  - Evaluates as if written like

```
if (count == 0)
        average = 0
else
        average = sum /count
```

Arithmetic Expressions: Operand Evaluation Order

- Less commonly discusses, *operand evaluation order* is the order of evaluation of operands.
  1. Variables: fetch the value from memory
  2. Constants: sometimes a fetch from memory; sometimes the constant is in the machine language instruction
  3. Parenthesized expressions: evaluate all operands and operators first

Arithmetic Expressions: Potentials for Side Effects

- *Functional side effects:* when a function changes a two-way parameter (pointer) or a non-local variable
- Problem with functional side effects:
  - When a function referenced in an expression alters another operand of the expression; e.g., for a parameter change:

```
a = 10; /*a is a global variable*/

/* assume that fun changes its parameter */
b = a + fun();
```

```
int a = 5;
int fun1() {
a = 17;
return 3;
} /* end of fun1 */

void main() {
a = a + fun1();
} /* end of main */
```

Functional Side Effects

- Two possible solutions to the problem
  1. Write the language definition to disallow functional side effects
     - No two-way parameters in functions
     - No non-local references in functions
     - **Advantage:** it works!
     - **Disadvantage:** inflexibility of two-way parameters and non-local references
  2. Write the language definition to demand that operand evaluation order be fixed
     - **Disadvantage**: limits some compiler optimizations

Overloaded Operators

- Use of an operator for more than one purpose is called *operator overloading*. For example:
  - + is used to add both `int` and `float`. It is also used in Java to concatenate strings.
- Some are common and acceptable.
- Some are potential trouble (e.g., `*` in C and C++)
  - Loss of compiler error detection (omission of an operand should be a detectable error)
  - Some loss of readability
  - Can be avoided by introduction of new symbols (e.g., Pascal's **`div`** for integer division)
- C++ and Ada allow user-defined overloaded operators

Overloaded Operators (continued)

- When sensibly used, operator overloading can aid readability. However, there are also potential problems:
  - Harmful to readability
    - Users can define nonsense operations even when the operators make sense
    - Reader must find both types of operands and definition of operator to determine the meaning.
  - Synchronisation needed when building large system involving different groups.
- C++ however al has a few operators that cannot be overloaded, such as the (.) operator for structure member.

Type Conversions

- A *narrowing conversion* is one that converts an object to a type that cannot include all of the values of the original type e.g., from `double` to `float`.
- A *widening conversion* is one in which an object is converted to a type that can include at least approximations to all of the values of the original type e.g., `int` to `float`.
- Type conversion is an issue in languages that allow *mixed-mode expressions*, i.e. expressions that have operands of different types.
- The conversion can be implicit or explicit.

Implicit Type Conversions

- A *coercion* is an implicit type conversion, i.e. the conversion is done by the compiler.
- Disadvantage of coercions:
  - They decrease in the type error detection ability of the compiler
- In most languages, all numeric types are coerced in expressions, using widening conversions
- In Ada, there are virtually no coercions in expressions

Explicit Type Conversions

- Explicit type conversions is called *casting* in C-based languages and it is explicitly requested by the programmer.
- Warning will be generated in some cases if the conversion is a narrowing conversion.
- Examples
  - `C: (int)angle`
  - `Ada: Float(sum)`

Note that Ada's syntax is similar to function calls

```
#include <stdio.h>
main()
{ int sum = 17, count = 5; double
mean;
mean = (double) sum / count;
printf("Value of mean : %f\n", mean );
}
```

Type Conversions: Errors in Expressions

- Run-time errors which are sometimes called *exceptions*.
- Causes
  - Inherent limitations of arithmetic, e.g., division by zero
  - Limitations of computer arithmetic, e.g. overflow or underflow.

Relational and Boolean Expressions

- Relational Expressions
  - Use relational operators and operands of various types, usually numeric types, strings and ordinal types.
  - Evaluate to some Boolean or its representation.
  - Operator symbols used vary somewhat among languages. For example, the inequality operator.
  - C-based languages (`!=`)
  - Ada (`/=`)
  - Lua (`~=`)
  - Fortran 95 (`.NE.`, `<>`)
- Precedence operators always have lower precedence than the arithmetic operators.

Relational and Boolean Expressions

- Boolean Expressions
  - Operands are Boolean and the result is Boolean
  - Example operators

FORTRAN 77	FORTRAN 90	C	Ada
.AND.	and	&&	and
.OR.	or	||	or
.NOT.	not	!	not
			xor

Relational and Boolean Expressions

- C has no Boolean type -- it uses `int` type with 0 for false and nonzero for true
- One odd characteristic of C's expressions:

  `a < b < c` is a legal expression, but the result is not what you might expect.

  - Left operator is evaluated, producing 0 or 1
  - The evaluation result is then compared with the third operand (i.e., `c`)

Relational and Boolean Expressions

- Precedence of C-based operators

```
postfix ++, --
unary +, -, prefix ++, --, !
*,/,%
binary +, -
<, >, <=, >=
=, !=
&&
||
```

Short Circuit Evaluation

- An expression in which the result is determined without evaluating all of the operands and/or operators
- Example: `(13*a) * (b/13-1)`
  If `a` is zero, there is no need to evaluate `(b/13-1)`
- Problem with non-short-circuit evaluation

```
index = 0;
while ((index < listlen) && (list[index] !=
  key)
      index++;
```

  - When `index = listlen`, `list[index]` will cause an indexing problem (assuming `list` has `length` -1 elements).

Short Circuit Evaluation

- Short-circuit evaluation also exposes the potential problem of (functional) side effects in expressions
  e.g. `(a > b) || (b++ / 3)`
- In Ada, programmer can specify either (short-circuit is specified with `and then` and `or else`)
- C, C++, and Java use short-circuit evaluation for the usual Boolean operators (**`&&`** and **`||`**), but also provide bitwise Boolean operators that are not short circuit (**`&`** and **`|`**)

Assignment Statements

- The general syntax

  `<target_var> <assign_operator> <expression>`

- The assignment operator

  = FORTRAN, BASIC, PL/I, C, C++, Java

  := ALGOLs, Pascal, Ada

- = can be bad when it is overloaded for the relational operator for equality

Assignment Statements: Conditional Targets

- Perl allows conditional targets on assignment statements. E.g.

```
($flag ? $total : $subtotal) = 0
```

Which is equivalent to

```
if ($flag)
  $total = 0
else
  $subtotal = 0
```

Assignment Statements: Compound Operators

- A shorthand method of specifying a commonly needed form of assignment
- Introduced in ALGOL; adopted by C
- Example

```
a = a + b
```

is written as

```
a += b
```

Assignment Statements: Unary Assignment Operators

- Unary assignment operators in C-based languages combine increment and decrement operations with assignment
- Examples

  `sum = ++count` (`count` incremented, added to `sum`)

  `sum = count++` (`count` added to `sum`, then incremented)

  `count++` (`count` incremented)

  `-count++` (`count` incremented then negated)

Assignment Statements: Assignment as an Expression

- In C, C++, and Java, the assignment statement produces a result and can be used as operands
- An example:

```
while ((ch = getchar())!= EOF){...}
```

  - `ch = getchar()` is carried out; the result (assigned to `ch`) is used as a conditional value for the `while` statement
- Also:

```
a = b + (c = d / b) - 1
```

- Multiple target assignments in C

```
sum = count = 0;
```

- Problem: Loss of error detection in the C design of the assignment operation, i.e. `if (x = y)` will not be detected as error when the intention is `if (x == y)`.

Assignment Statements: List Assignments

- Multiple target multiple source assignment statements. E.g. in Perl

```
($first, $second, $third) = (20, 40, 60);
```

- Swapping

```
($first, $second) = ($ second, $first);
```

Mixed-Mode Assignment

- Assignment statements can also be mixed-mode, for example

```
int a, b;
float c;
c = a / b;
```

- In Pascal, integer variables can be assigned to real variables, but real variables cannot be assigned to integers
- In Java, only widening assignment coercions are done
- In Ada, there is no assignment coercion

SUMMARY

- Expressions
- Operator precedence and associativity
- Operator overloading
- Mixed-type expressions
- Various forms of assignment