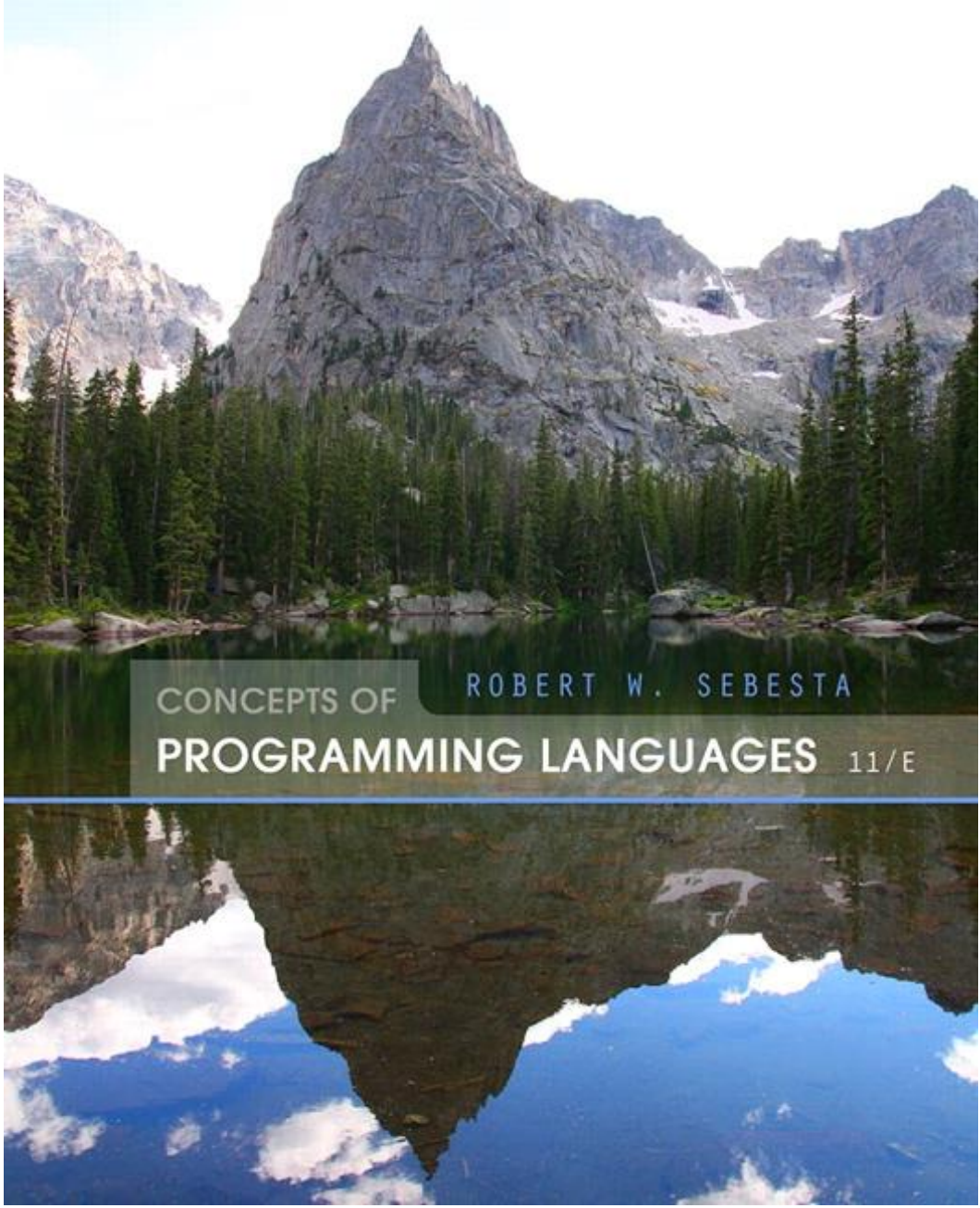
CSC304:

Chapter 7:

Expressions and Assignment Statements



ISBN 0-321-49362-1

Assignment Statements

- The general syntax

  `<target_var> <assign_operator> <expression>`

- The assignment operator

  = Fortran, BASIC, the C-based languages

  := Ada

- = can be bad when it is overloaded for the relational operator for equality (that's why the C-based languages use == as the relational operator)
  - e.g. (PL/I) A = B = C;

Assignment Statements: Conditional Targets

- Conditional targets (Perl)

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

Which is equivalent to

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

Assignment Statements: Compound Assignment Operators

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

```
a = a + b
```

can be written as

```
a += b
```

Assignment Statements:

- More complicated assignments:
  - Multiple targets (PL/I)
    - `A, B = 10`
  - Conditional targets (C, C++, and Java)
    - `x = flag ? count1 : count2 = 0;`
  - Compound assignment operators (C, C++, and Java)
    - `sum += next;`

Assignment Statements:

- Unary assignment operators (C, C++, and Java)
  - **a++;**
  - **++a; % difference in values**

```
Main()
{ int a = 1;
  printf(" a is %d", ++a);
}
```

```
Main()
{ int a = 1;
  printf(" a is %d", a++);
}
```

Assignment Statements:

- C, C++, and Java treat **=** as an arithmetic binary operator
  - e.g. a = b * (c = d * 2 + 1) + 1
  - This is inherited from ALGOL 68
- Assignment as an Expression
  - In C, C++, and Java, the assignment statement produces a result
  - So, they can be used as operands in expressions
    - e.g. **`while ((ch = getchar() != EOF) { ... }`**
- Disadvantage
  - Another kind of expression side effect

Assignment Statements:

- Exercise:

```
a=1, b=2, c=3, d=4
a = b + (c = d / b++) - 1
cout << a << “,” << b << “,” <<
c << “,” << d << endl
```

Assignment Statements: Unary Assignment Operators

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

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

  `sum = count++` (`count` assigned to `sum`, then incremented

  `count++` (`count` incremented)

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

Assignment as an Expression

- In the C-based languages, Perl, and JavaScript, the assignment statement produces a result and can be used as an operand

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

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

- Disadvantage: another kind of expression side effect

Multiple Assignments

- Perl, Ruby, and Lua allow multiple-target multiple-source assignments

```
($first, $second, $third) = (20, 30, 40);
```

Also, the following is legal and performs an interchange:

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

Assignment in Functional Languages

- Identifiers in functional languages are only names of values
- ML
  - Names are bound to values with **`val`**

```
val fruit = apples + oranges;
```

  - If another val for fruit follows, it is a new and different name
- F#
  - F#'s **`let`** is like ML's **`val`**, except **`let`** also creates a new scope

Mixed-Mode Assignment

- Assignment statements can also be mixed-mode
- In Fortran, C, Perl, and C++, any numeric type value can be assigned to any numeric type variable
- In Pascal
  - integers can be assigned to reals, but reals cannot be assigned to integers
  - programmer must specify whether conversion from real to integer is truncated or rounded
- In Java and C#, only widening assignment coercions are done
- In Ada, there is no assignment coercion

Assignments

- Functional programming:
  - We return a value for surrounding context.
  - Value of expression depends solely on referencing environment, not on the time in which the evaluation occurs.
  - Expressions are "referentially transparent."

Assignments

- Imperative:
  - Based on side-effects.
  - Influence subsequent computation.
  - Distinction between
    - Expressions (return a value)
    - Statements (no value returned, done solely for the side-effects).

Variables

- Can denote a location in memory (l-value)
- Can denote a value (r-value)

- Typically,

`2+3 := c;` is illegal, as well as

`c := 2+3;` if c is a declared constant.

Variables

- Expression on left-hand-side of assignment can be complex, as long as it has an l-value:

```
(f(a)+3)->b[c] = 2;
```
in C.

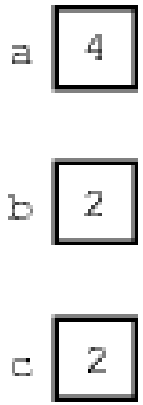
- Here we assume f returns a pointer to an array of elements, each of which is a structure containing a field b, an array. Entry c of b has an l-value.

Referencing/ Dereferencing

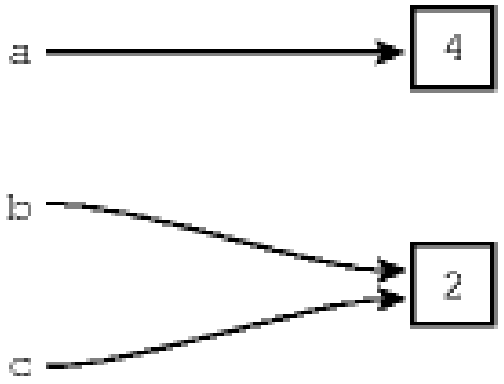
- Consider

```
b := 2;
    c := b;
    a := b + c;
```

- Value Model



Reference Model



Referencing/ Dereferencing

- Pascal, C, C++ use the "value model":
  - Store 2 in `b`
  - Copy the 2 into `c`
  - Access `b`,`c`, add them, store in `a`.

- Clu uses the "reference" model:
  - Let `b` refer to 2.
  - Let `c` also refer to 2.
  - Pass references `a`,`b` to "+", let `a` refer to result.

Referencing/ Dereferencing

- Java uses value model for intrinsic (`int`, `float`, `etc.`) (could change soon !), and reference model for user-defined types (classes)

Orthogonality

- Features can be used in any combination
- Every combination is consistent.
- Algol was first language to make orthogonality a major design goal.

Orthogonality in Algol 68

- Expression oriented; no separate notion of statement.

```
begin
      a := if b < c then d else e;
      a := begin f(b); g(c) end;
      g(d);
      2+3
   end
```

Orthogonality in Algol 68

- Value of `'if'` is either expression `(d or e)`.
- Value of `'begin-end'` block is value of last expression in it, namely `g(c)`.
- Value of `g(d)` is obtained, and discarded.
- Value of entire block is `5`.

Orthogonality in Algol 68

- C does this as well:
  - Value of assignment is value of right-hand-side:

```
c = b= a++;
```

Pitfall in C

```
if (a=b) {  ...  }
/* assign b to a and proceed */ /* if result is nonzero  /*
```

- Some C compilers warn against this.
- Different from

```
if (a==b) {  ...  }
```

- Java has separate boolean type:
  - prohibits using an int as a boolean.

Initialization

- Not always provided (there is assignment)
- Useful for 2 reasons:

1. Static allocation:
    - compiler can place value directly into memory.
    - No execution time spent on initialization.
2. Variable not initialized is common error.

Initialization

- Pascal has NO initialization.
  - Some compilers provide it as an extension.
  - Not orthogonal, provided only for intrinsics.

- C, C++, Ada allow aggregates:
  - Initialization of a user-defined composite type.

Example in C

```
int a[] = {2,3,4,5,6,7}
```

- Rules for mismatches between declaration and initialization:

```
int a[4] = {1,2,3}; /* rest filled with zeroes */
int a[4] = {0};     /* filled with all zeroes  */
int a[4] = {1,2,3,4,5,6,7}   /* oops!  */
```

- Additional rules apply for multi-dimensional arrays and structs in C.

Uninitialized Variables

- Pascal guarantees default values (e.g. zero for integers)

- C guarantees zero values only for static variables, *garbage* for everyone else !

Uninitialized Variables

- C++ distinguishes between:
  - initialization (invocation of a constructor, no initial value is required)
    - Crucial for user-defined ADT's to manage their own storage, along with destructors.
  - assignment (explicit)

Uninitialized Variables

- Difference between initialization and assignment: variable length string:
  - Initialization: allocate memory.
  - Assignment: deallocate old memory AND allocate new.

Uninitialized Variables

- Java uses reference model, no need for distinction between initialization and assignment.

- Java requires every variable to be "definitely assigned", before using it in an expression.
  - Definitely assigned: every execution path assigns a value to the variable.

Uninitialized Variables

- Catching uninitialized variables at run-time is expensive.
  - hardware can help, detecting special values, e.g. "NaN" IEEE floating-point standard.
  - may need extra storage, if all possible bit patterns represent legitimate values.

Combination Assignment Operators

- Useful in imperative languages, to avoid repetition in frequent updates:

```
a = a + 1;
b.c[3].d = b.c[3].d * 2;  /* ack !  */
```

- Can simplify:

```
++a;
b.c[3].d *= 2;
```

Combination Assignment Operators

- Syntactic sugar for often used combinations.

- Useful in combination with autoincrement operators:

```
A[--i]=b;
```
equivalent to

```
A[i -= 1] = b;
```

Combination Assignment Operators

```
*p++ = *q++;    /*  ++ has higher
  precedence than * */
```

equivalent to

```
*(t=p, p += 1, t) = *(t=q, q += 1, t);
```

- Advantage of autoincrement operators:
  - Increment is done in units of the (user-defined) type.

Comma Operator

- In C, merely a sequence:

```
int a=2, b=3;

a,b  = 6;     /* now a=2 and b=6  */


int a=2, b=3;

a,b = 7,6;  /*  now a=2 and b=7                    */

            /*  = has higher precedence than ,  */
```

Comma Operator

- In Clu, "comma" creates a tuple:

```
a,b := 3,4
```
assigns 3 to a, 4 to b

```
a,b := b,a
```
swaps them !

- We already had that in RPAL:

```
let t=(1,2)
in (t 2, t 1)
```

Ordering Within Expressions

- Important for two reasons:

1. Side effect:
   - One sub expression can have a side effect upon another subexpression:

```
(b = ++a + a--)
```

2. Code improvement:
   - Order evaluation has effect on register/instruction scheduling.

Ordering Within Expressions

- Example: `a * b + f(c)`
  - Want to call f first, avoid storing (using up a register) for `a*b`

    during call to f.

Ordering Within Expressions

- Example:

```
a := B[i];

c := a * 2 + d * 3;
```

  - Want to calculate `d * 3` before `a * 2`: Getting a requires going to memory (slow); calculating `d * 3` can proceed in parallel.

Ordering Within Expressions

- Most languages leave subexpression order unspecified (Java is a notable exception, uses left-to-right)

- Some will actually rearrange subexpressions.

Example (Fortran)

```
a = b + c
    c = c + e + b
```

rearranged as

```
    a = b + c
    c = b + c + e
```

and then as

```
      a = b + c
      c = a + e
```

Rearranging Can Be Dangerous

- If `a,b,c` are close to the precision limit (say, about ¾ of largest possible value), then

  `a + b - c` will overflow, whereas

  `a - c + b` will not.

- Safety net: most compilers guarantee to follow ordering imposed by parentheses.

Summary

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

Class Activities

i. When might you want the compiler to ignore type differences in an expression?
ii. State your own arguments for and against allowing mixed-mode arithmetic expressions.
iii. Do you think the elimination of overloaded operators in your favorite language would be beneficial? Why or why not?
iv. Would it be a good idea to eliminate all operator precedence rules and require parentheses to show the desired precedence in expressions? Why or why not?
v. Should C's assigning operations (for example, +=) be included in other languages (that do not already have them)? Why or why not?

Class Activities

vi. Should C's single-operand assignment forms (for example, ++count) be included in other languages (that do not already have them)? Why or why not?

vii. Describe a situation in which the add operator in a programming language would not be commutative.

viii. Describe a situation in which the add operator in a programming language would not be associative.

ix. Explain why it is difficult to eliminate functional side effects in C.

x. For some language of your choice, make up a list of operator symbols that could be used to eliminate all operator overloading.

xi. Why does Java specify that operands in expressions are all evaluated in left-to-right order?

Programming Exercises (Class Participatuion)

i. Write a test program in your favorite language that determines and outputs the precedence and associativity of its arithmetic and Boolean operators.

ii. Write a Java program that exposes Java's rule for operand evaluation order when one of the operands is a method call.

iii. Write a program in either C++, Java, or C# that illustrates the order of evaluation of expressions used as actual parameters to a method.

Programming Exercises (Class Participatuion)

iv. Write a C program that has the following statements:

```
int a, b;
a = 10;
b = a + fun();
printf("With the function call on the right, ");
printf(" b is: %d\n", b);
a = 10;
b = fun() + a;
printf("With the function call on the left, ");
printf(" b is: %d\n", b);
```

and define fun to add 10 to a. Explain the results.

Programming Exercises (Class Participatuion)

v. Write a program in either Java, C++, or C# that performs a large number of floating-point operations and an equal number of integer operations and compare the time required.