

Performing Computations

C provides operators that can be applied to calculate expressions:

example: tax is 8.5% of the total sale

expression: `tax = 0.085 * totalSale`

Need to specify what operations are legal, how operators evaluated, etc.

C operations are referred to as Expressions

Outline

II. Program Basics

G. Expressions

1. Primary
identifiers, constants, parenthesized expressions
2. Unary
postfix: function call, increment, decrement
prefix: `sizeof`, unary `+`, unary `-`, cast, increment, decrement
3. Binary
arithmetic ops: `+` `-` `*` `/` `%`
assignment
shorthand assignment

Outline (cont)

II. Program Basics

4. Casting
implicit
promotion hierarchy
explicit
5. Side effects
6. Resolving Expressions
Precedence
Associativity

H. Statements

1. Expression
2. Compound

Expressions in C

- Example: `total * TAXRATE`
- Consists of a sequence of operators and operands
 - operands: `total`, `TAXRATE`
 - can be expressions themselves
 - operator: `*`
 - operation calculating a result based on the operand(s)

Some Expression Formats

- primary
 - expressions that directly map to a value
- unary
 - prefix: *operator operand*
 - postfix: *operand operator*
- binary
 - *operand1 operator operand2*
- ternary (operator in two parts)

Types for Operands

- Operators only apply to certain types
 - e.g., addition applies to numbers
- Each operand must be of the appropriate type
- Operators calculate a value
- Value can then be used as operands to other operators

Primary Expressions

- identifiers
 - variable
 - defined constant
- literal constants
 - operands can be values of the appropriate type
 - operator can have constant (e.g., 5, -34) as operand

Binary Expressions

expr1 op expr2

arithmetic operators

- + - binary addition (e.g., 5 + 3 result: 8)
- - binary subtraction (e.g., 5 - 3 result: 2)
- * - multiplication (e.g., 5 * 3 result: 15)
- / - division (e.g., 5 / 3 result: 1)
- % - remainder (e.g., 5 % 3 result: 2)

arithmetic operators apply to number types, result type depends on operand types

Arithmetic Operators

Two int operands -> int result

Two float operands -> float result

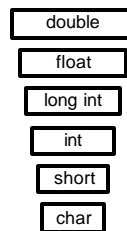
/ - division operator, is whole number division for ints, floating point division for other operands

% - remainder operator, only works for int values

Mixed operands (float/int) - result produced is based on a “promotion” hierarchy, result cast to higher type in hierarchy

e.g., 5 / 3.0 produces 1.666666

Promotion Hierarchy



- Operands of the lower type can be implicitly converted to the higher type (in C)
- Occurs when an operator expects the higher type (adding together two floats when one is an int)

Assignment Expression

varName = expr

value of *expr* determined

result of expression is the value of *expr*

as a “side effect” the value calculated is also stored in the named variable (the current value stored in the variable is replaced)

example:

current value of total is 5

total = 3

replaces the value 5 with the value 3 in the memory location corresponding to total

Assignment Expression (cont)

varName = expr

expr value must be of the proper type to be stored in the variable (error otherwise)

the value may be converted automatically using the promotion hierarchy

example:

float f;

f = 3; /* 3 cast from an integer to a float value */

Shorthand Assignments

`+=` `-=` `*=` `/=` `%=`

example:

```
x += 3
```

is a shorthand for

```
x = x + 3
```

assuming x is 7 before `x += 3` is calculated, the right-hand side `x + 3` is calculated, producing 10, this value is then stored in x, and the result of the expression is 10

thus, these operators also have side effects

Unary Expressions

- Postfix

fName(arglist) - function call

subprogram is called (invoked) which performs some set of computations and returns a value (more later)

- Prefix

`sizeof (expr)` - `expr` can be any primary expression or the name of a type, operator returns num bytes used to represent type (int)

`+ expr` - unary plus (applies to numbers, no effect)

`- expr` - unary minus (inverts sign of numbers)

Cast Operator (Unary Prefix Op)

(TypeName) expression

conversion forces expression value from current type to the provided type

example: `(float) 5` results in 5.0

especially useful for division

example:

```
totalScore, totalGrades (ints)
```

```
totalScore / totalGrades produces integer
```

```
(float) totalScore / totalGrades casts totalScore to a floating point number, then division occurs
```

Prefix Increment, Decrement

`++variable` - prefix increment

`--variable` - prefix decrement

prefix unary operators

example: `++x` is a shorthand for

```
x = x + 1
```

similarly `--x` is a shorthand for `x = x - 1`

apply to integral values (char, int)

have side effect of changing value of variable

Postfix Increment, Decrement

`variable++` - postfix increment

`variable--` - postfix decrement

postfix unary operators

similar to prefix versions BUT different in the value they produce

`x++` is a shorthand for the statement `x = x + 1` but the result of `x++` is the value of x BEFORE the value is changed

example: assume x is 8, `x++` changes x to 9 but returns the value 8

the postfix decrement operator works similarly

Side Effects

- Certain operations not only produce values, but also perform other actions

- assignment statements - change the value of variable on left-hand side of operator

- scanf operator - returns a value (number of arguments read), but also changes variables in scanf expression

example: `scanf("%d",&total)` changes total

Complex Expressions

- Expressions can be composed to produce more complex expressions
operand of expression can be an expression:
 $total = totalSale + TAXRATE * totalSale$
three operators, =, +, *, how is this expression resolved??
In C, $TAXRATE * totalSale$ is calculated, then $totalSale$ is added to this value, then this value is stored in $total$
But why??

Resolving Complex Expressions

- C has precedence, associativity rules it uses to determine the order operators are evaluated in
- Precedence: some operators are executed before others
- Associativity: when more than one operator in an expression has the same precedence then associativity rules are used to order

Operator Precedence

- 18: Identifier, Constant, Parenthesized Expression
- 17: Function call
- 16: Postfix increment/decrement,
- 15: Prefix increment/decrement, sizeof, unary +, unary -
- 14: (Type) - cast operator
- 13: * / %
- 12: + -
- 2: = += -= *= /= %=

Using Operator Precedence

Example:

```
total = totalSale + TAXRATE * totalSale
* has highest precedence, so TAXRATE * totalSale
is calculated first
+ has next highest precedence, so the result of
TAXRATE * totalSale is added to totalSale
= has the lowest precedence, so the value calculated
in the previous operations is stored in total
```

Parenthesized Expressions

- What if the precedence order is not what you want?
example: $sale1 + sale2 * TAXRATE$
in this case the multiplication would happen first, but you might want the addition first
answer: parenthesize the expressions you want to happen first
result: $(sale1 + sale2) * TAXRATE$
parenthesized expressions have highest precedence

Programming Tip: Parenthesizing

- Does not hurt to include parentheses for all expressions
- Thus you can guarantee the order of evaluation
Example:
 $total = totalSale + TAXRATE * totalSale$
becomes
 $total = (totalSale + (TAXRATE * totalSale))$

Associativity

Rules determining how to evaluate expressions containing more than one operator with the same precedence
example: is $5 - 4 - 3$ $((5 - 4) - 3)$ or $(5 - (4 - 3))$
its important because the values may differ (e.g., -2 or 4 for the above possibilities)

Left associativity: operators are evaluated left to right

Right associativity: evaluated right to left

Operator Associativity

18: Identifier, Constant, Parenthesized Expression

17: Function call (*LEFT*)

16: Postfix increment/decrement (*LEFT*)

15: Prefix increment/decrement, sizeof, unary +, unary - (*RIGHT*)

14: (Type) - cast operator (*RIGHT*)

13: * / % (*LEFT*)

12: + - (*LEFT*)

2: = += -= *= /= %= (*RIGHT*)

Using Operator Associativity

- Evaluate operators by precedence
- When evaluating operators with the same precedence, consult associativity rules
example: $5 - 4 - 3$, two - operators, so associativity (left) is used
 $5 - 4 - 3$ evaluates to $((5 - 4) - 3)$ or -2
- Again, use parentheses to make sure things are evaluated the way you expect

Composing Assignments

Example:

$x = y = 3$ ($x = (y = 3)$) with Right associativity

Ok to do, but better to avoid

Example:

x is 4, what is the result of $x = y = x + 3$?

$x + 3$ evaluated (producing 7), this value stored to y, then the result stored back to x

Multiple Assignments

- In ANSI C, changing a variable multiple times in the same expression produces an undefined result
example: $(b-- + b--)$
is undefined, C may execute both b-- operators together, the left b-- first, or the right b-- first
solution: better not to do it

Statements

- Programs consist of sequences of statements
- Expressions can be used as part of various types of statements
- Expressions can also be used as stand-alone statements
 - generally assignment expressions are issued as separate statements

Expression Statements

Format: *Expression* ;

- When the statement is reached in the program the expression is evaluated
- Assignment statements cause side effects (change the value of variables)
- printf and scanf are actually functions that we call (and produce values), so these are expression statements

Compound Statements

- A block of code containing zero or more statements
- Contained between { and }
- Format:

```
{  
    Local Declarations
```

```
    Statements  
}
```

Compound Statements (cont)

- Example:

```
{  
    int x; /* local declaration */  
  
    x = 3; /* statements */  
    y = x + 4;  
    printf("%d\n",y);  
}
```

- Block of statements treated as a unit

Compound Statements (cont)

- Local declarations exist during the execution of the block/compound statement
- Declarations are made one at a time
- Statements are executed from the first to the last
- Declarations go away once the statements in the block are executed