Operators

COM S 113

February 1, 1999

Announcements

Textbook status (1st ed.?)

Read K&R Chapter 2 by Wednesday

New assignment will be out Wednesday

Initial Comments on Assignment 1

Comment characters // not allowed

Squaring numbers without `pow()`

Using fewer `int` variables

`const` Type Qualifier

Can be applied to declaration of any variable—even function parameters

Specifies that variable's value won't be changed; for arrays, says that the array elements won't be changed

```
const double e = 2.71828182845905;
const char msg[] = "warning: ";
int strlen(const char[]);
```

Arithmetic Operators

Binary arithmetic ops are +, -, *, /, and modulus operator %.

Unary + and - have highest precedence of arithmetic operators. *, /, and % are next, and binary + and - are lowest.

Associativity is left-to-right

Relational Operators

Relational operators have lower precedence than arithmetic operators

`>`, `>=`, `<`, and `<=` have the same precedence

`==` and `!=` are lower

Examples: `i < lim-1`   `a>b == c>d`

Logical Operators

`&&` and `||` have lower precedence than relational operators, with `&&` above `||`

"Short-circuit evaluation"—evaluation stops when truth or falsehood of an expression is known

Example: `a && b || c`

Logical Operators (continued)

Numerical value of relational or logical expression is 1 for true, 0 for false

Unary negation operator `!` has high precedence (same as unary + and -)

Example: `if (!valid)` same as `if (valid == 0)`

What does `!!x` do?

Assignment Operators

Assignment operators (such as `=`) have very low precedence, right-to-left associativity

Examples: `a=b=c+d;`   What does `a=b+c=d;` do?

When variable on left side of assignment is repeated immediately on right, as in `i = i + 2`, can rewrite with assignment operator: `i += 2`

Assignment Operators (continued)

expr_1 *op*= expr_2 almost equivalent to

expr_1 = (expr_1) *op* (expr_2)

Example: `x *= y + 1` means `x = x * (y + 1)` rather than `x = x * y + 1`

Value of assignment expression is value of left operand after assignment

Example Expressions

```
while ((c = getchar()) != EOF) ...
```

```
i<lim-1 && (c = getchar()) != '\n' && c != EOF
```

Increment and Decrement Operators

`++` and `--` operators have very high precedence (same as `!` and unary `+` and `-`)

Examples:

```
for (i=0; i<10; i++) printf("%d\n");
for (i=9; i>=0; i--) printf("%d\n");
```

Increment and Decrement Operators (continued)

May be used as prefix operators (like `++n`) or postfix (like `n++`), but only to variables (not expressions)

Difference is whether increment happens before or after value is used

If `n` is 5, consider `x = n++;` versus `x = ++n;`

Increment Operator Example (K&R p. 47)

```
/* squeeze: delete all c from s */
void squeeze(char s[], int c) {
  int i, j;
  for (i = j = 0; s[i] != '\0'; i++)
    if (s[i] != c)
      s[j++] = s[i];
  s[j] = '\0';
}
```

Increment Operator Example (K&R p. 48)

```
/* strcat: add t to end of s; s must be big enough */
void strcat(char s[], char t[]) {
  int i, j;
  i = j = 0;
  while (s[i] != '\0') /* find end of s */
    i++;
  while ((s[i++] = t[j++]) != '\0') /* copy t */
    ;
}
```

Pitfall: Evaluation Order Unspecified

```
a[i] = i++; /* wrong */
```

```
printf("%d %d\n", ++n, pow(2, n)); /* wrong */
```

```
printf("Hello ") + printf("there!\n"); /* wrong */
```

Type Conversions

Automatic type conversions used when operands have different types

Normally narrower operand converted to type of wider one, but lossy assignments are legal

Nonsensical expressions (like using `float` as array subscript) are disallowed

Explicit Type Conversion with Casting

(*type-name*) *expression*

The *expression* is converted to the named type using the normal conversion rules

Example: `sqrt((double) n)` converts `n` to a `double` *but doesn't modify* `n`

How many conversions in this? `double x = (int) sqrt(2);`

Example of `char` as Integer

```
/* atoi: convert s to integer */
int atoi(char s[]) {
  int i, n = 0;

  for (i = 0; s[i] >= ’0’ && s[i] <= ’9’; ++i)
    n = 10 * n + (s[i] - ’0’);
  return n;
}
```

Conditional Expressions

```
if (a > b) z = a; else z = b;
```

can be written as

```
z = (a > b) ? a : b;   /* z = max(a, b) */
```

General form: expr_1 `?` expr_2 `:` expr_3

Precedence very low—just above assignment operators

Conditional Expressions (continued)

If *expr*$_2$ and *expr*$_3$ are of different types, conversion rules applied

Consider the type of this, if `f` is `float` and `n` is `int`:

```
(n > 0) ? f : n
```

Examples of Conditional Expressions

```
printf("You have %d item%s.\n", n, n==1 ? "" : "s");


for (i = 0; i < n; i++)
  printf("%6d%c", a[i],
         (i%10==9 || i==n-1) ? '\n' : ' ');
```

Comma Operator

Lowest precedence of any operator in C

```
#include <string.h>
void reverse(char s[]) { /* reverse string s in place */
  int c, i, j;
  for (i = 0, j = strlen(s)-1; i < j; i++, j--) {
    c = s[i]; s[i] = s[j]; s[j] = c;
  }
}
```

Comma Operator (continued)

Commas separating function arguments, variables in declarations, etc. are *not* comma operators and do not guarantee evaluation order

Use commas very sparingly

```
for (i = 0, j = strlen(s)-1; i < j; i++, j--)
  c = s[i], s[i] = s[j], s[j] = c;
```