Week 4 Expressions; Selection Statements

- **Increment/Decrement Operators**
  - Side effect
- **Expression Evaluation/Expression Statement**
- **Logical Expressions**
  - Relational/Equality/Logical Operators

- **The if Statement**
  - Compound Statements
  - Cascaded if Statements
  - Boolean Values

- **Readings**
  - **King Ch.4 and Ch. 5**

Side Effect

- An assignment of the form *v* = *e* is allowed wherever a value of type *v* would be permitted:

```
i = 1;
k = 1 + (j = i);
printf("%d %d %d\n", i, j, k);
  /* prints "1 1 2" */
```

- "Embedded assignments" can make programs hard to read.

- They can also be a source of subtle bugs.

Lvalues

- The assignment operator requires an *lvalue* as its left operand.

- An lvalue represents an object stored in computer memory, not a constant or the result of a computation.

- Variables are lvalues; expressions such as `10` or `2 * i` are not.

- Since the assignment operator requires an lvalue as its left operand, it's illegal to put any other kind of expression on the left side of an assignment expression:

```
12 = i;        /*** WRONG ***/
i + j = 0;     /*** WRONG ***/
-i = j;        /*** WRONG ***/
```

- The compiler will produce an error message such as *"invalid lvalue in assignment."*

Compound Assignment

- Assignments that use the old value of a variable to compute its new value are common.

- Example: `i = i + 1;`

- Using the `+=` compound assignment operator, we simply write:

```
i += 2;    /* same as i = i + 2; */
```

- There are nine other compound assignment operators, including :

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

- All compound assignment operators work in much the same way:

  v* += *e* adds *v* to *e*, storing the result in *v// v= v+3

  v* −= *e* subtracts *e* from *v*, storing the result in *v// v = v-e;

  v* *= *e* multiplies *v* by *e*, storing the result in *v // v= v*e;

  v* /= *e* divides *v* by *e*, storing the result in *v // v = v/3;

  v* %= *e* computes the remainder when *v* is divided by *e*, storing the result in *v; v = v%e;

Compound Assignment

- *v* += *e* isn't "equivalent" to *v* = *v* + *e*.
- One problem is operator precedence:
  - `i *= j + k` isn't the same as `i = i * j + k`.
- There are also rare cases in which *v* += *e* differs from *v* = *v* + *e* because *v* itself has a side effect.
- Similar remarks apply to the other compound assignment operators.
- When using the compound assignment operators, be careful not to switch the two characters that make up the operator.
- Although `i =+ j` will compile, it is equivalent to `i = (+j)`, which merely copies the value of `j` into `i`.

Increment and Decrement Operators

- **Two of the most common operations on a variable are "incrementing" (adding 1) and "decrementing" (subtracting 1):**

```
i = i + 1; j = j - 1;
```

- **Incrementing and decrementing can be done using the compound assignment operators:**

```
i += 1; j -= 1;
```

- **C provides special `++` (*increment*) and `--` (*decrement*) operators.**

- **The `++` operator adds 1 to its operand. The `--` operator subtracts 1.**

- **The increment and decrement operators are tricky to use:**
  - They can be used as ***prefix*** operators (`++i` and `--i`) or ***postfix*** operators (`i++` and `i--`).
  - They have **side effects**: they modify the values of their operands.

Increment and Decrement Operators

- Evaluating the expression `++i` (a "pre-increment") yields `i + 1` and—as a side effect—increments `i`:

```
i = 1;
printf("i is %d\n", ++i);   /* prints "i is 2" */
printf("i is %d\n", i);     /* prints "i is 2" */
```

- Evaluating the expression `i++` (a "post-increment") produces the result `i`, but causes `i` to be incremented afterwards:

```
i = 1;
printf("i is %d\n", i++);   /* prints "i is 1" */
printf("i is %d\n", i);     /* prints "i is 2" */
```

- `++i` means "increment `i` immediately," while `i++` means "use the old value of `i` for now, but increment `i` later."

- How much later? The C standard doesn't specify a precise time, but it's safe to assume that `i` will be incremented before the next statement is executed.

Increment and Decrement Operators

- The `--` operator has similar properties:

```
i = 1;
printf("i is %d\n", --i);   /* prints "i is 0" */
printf("i is %d\n", i);     /* prints "i is 0" */
i = 1;
printf("i is %d\n", i--);   /* prints "i is 1" */
printf("i is %d\n", i);     /* prints "i is 0" */
```

- When ++ or -- is used more than once in the same expression, the result can often be hard to understand.

Increment and Decrement Operators

- **Example:**

```
i = 1; j = 2;
k = ++i + j++;
```

The last statement is equivalent to

```
i = i + 1;
k = i + j;
j = j + 1;
```

The final values of `i`, `j`, and `k` are 2, 3, and 4, respectively.

- **In contrast, executing the statements**

```
i = 1; j = 2;
k = i++ + j++;
```

will give `i`, `j`, and `k` the values 2, 3, and 3, respectively.

Expression Evaluation

- **Table of operators discussed so far:**

Precedence	*Name*	*Symbol(s)*	*Associativity*
1	increment (postfix)	++	left
	decrement (postfix)	--	
2	increment (prefix)	++	right
	decrement (prefix)	--	
	unary plus	+	
	unary minus	-	
3	multiplicative	* / %	left
4	additive	+ -	left
5	assignment	= *= /= %= += -=	right

Expression Evaluation

- The table can be used to add parentheses to an expression that lacks them.
- Starting with the operator with highest precedence, put parentheses around the operator and its operands.
- Example:

```
a = b += c++ - d + --e / -f
```

	Precedence level
`a = b += (c++) - d + --e / -f`	1
`a = b += (c++) - d + (--e) / (-f)`	2
`a = b += (c++) - d + ((--e) / (-f))`	3
`a = b += (((c++) - d) + ((--e) / (-f)))`	4
`(a = (b += (((c++) - d) + ((--e) / (-f)))))`	5

Expression Statements

- C has the unusual rule that any expression can be used as a statement. Example:

```
++i;
```

  `i` is first incremented, then the new value of `i` is fetched but then discarded.

- Since its value is discarded, there's little point in using an expression as a statement unless the expression has a side effect:

```
i = 1;        /* useful */
i--;          /* useful */
i * j - 1;    /* not useful */
```

- A slip of the finger can easily create a "do-nothing" expression statement.
- For example, instead of entering `i = j;`

  we might accidentally type `i + j;`

- Some compilers can detect meaningless expression statements; you'll get a warning such as *"statement with no effect."*

Logical Expressions

- Several of C's statements must test the value of an expression to see if it is "true" or "false."
- For example, an `if` statement might need to test the expression `i < j`; a true value would indicate that `i` is less than `j`.
- In many programming languages, an expression such as `i < j` would have a special "Boolean" or "logical" type.
- In C, a comparison such as `i < j` yields an integer: either 0 (false) or 1 (true).

Relational Operators

- **C's *relational operators:***

  < **less than**
  > **greater than**
  <= **less than or equal to**
  >= **greater than or equal to**

- **These operators produce 0 (false) or 1 (true) when used in expressions.**

- **The relational operators can be used to compare integers and floating-point numbers, with operands of mixed types allowed.**
- **The precedence of the relational operators is lower than that of the arithmetic operators.**
  - For example, `i + j < k - 1` means `(i + j) < (k - 1)`.

- **The relational operators are <u>left associative</u>.**

Relational Operators

- The expression

  `i < j < k`

  is legal, but does not test whether `j` lies between `i` and `k`.

- Since the < operator is left associative, this expression is equivalent to

  `(i < j) < k`

  The 1 or 0 produced by `i < j` is then compared to `k`.

- The correct expression is `i < j && j < k`.

Equality Operators

- C provides two *equality operators:*

  `==` equal to
  `!=` not equal to

- The equality operators are left associative and produce either 0 (false) or 1 (true) as their result.

- The equality operators have lower precedence than the relational operators, so the expression

  ```
  i < j == j < k
  ```

  is equivalent to

  ```
  (i < j) == (j < k)
  ```

Exercise

- **Show the output produced by each program fragment, assuming that i,j and k are int variables:**

```
(a) i = 2; j = 3;
    k = i * j == 6;
    printf("%d", k);

(b) i = 3; j = 2; k = 1;
    printf("%d", i < j == j < k);
```

Logical Operator

- More complicated logical expressions can be built from simpler ones by using the *logical operators:*

  `!` logical negation
  `&&` logical *and*
  `||` logical *or*

- The `!` operator is unary, while `&&` and `||` are binary.
- The logical operators produce 0 or 1 as their result.
- The logical operators treat any nonzero operand as a true value and any zero operand as a false value.

- Behavior of the logical operators:

  !*expr* has the value 1 if *expr* has the value 0.

  expr1 && *expr2* has the value 1 if the values of *expr1* and *expr2* are both nonzero.

  expr1 || *expr2* has the value 1 if either *expr1* or *expr2* (or both) has a nonzero value.

- In all other cases, these operators produce the value 0.

Logical Operator

- Both `&&` and `||` perform "short-circuit" evaluation: they first evaluate the left operand, then the right one.

- If the value of the expression can be deduced from the left operand alone, the right operand isn't evaluated.

- Example:

  ```
  (i != 0) && (j / i > 0)
  ```

  `(i != 0)` is evaluated first. If `i` isn't equal to 0, then `(j / i > 0)` is evaluated.

- If `i` is 0, the entire expression must be false, so there's no need to evaluate `(j / i > 0)`. Without short-circuit evaluation, division by zero would have occurred.

Logical Operator

- **Thanks to the short-circuit nature of the `&&` and `||` operators, side effects in logical expressions may not always occur.**

- **Example:**

  `i > 0 && ++j > 0`

  If `i > 0` is false, then `++j > 0` is not evaluated, so j isn't incremented.

- **The problem can be fixed by changing the condition to `++j > 0 && i > 0` or, even better, by incrementing `j` separately.**

- **The `!` operator has the same precedence as the unary plus and minus operators.**

- **The precedence of `&&` and `||` is lower than that of the relational and equality operators.**
  - For example, `i < j && k == m` means `(i < j) && (k == m)`.

- **The `!` operator is right associative; `&&` and `||` are left associative.**

Exercise

- **Show the output produced by each program fragment, assuming that i,j and k are int variables:**

```
(a) i = 3; j = 4; k = 5;
    printf("%d ", i < j || ++j < k);
    printf("%d %d %d ", i, j, k);

(b) i = 7; j = 8; k = 9;
    printf("%d ", (i = j) || (j = k));
    printf("%d %d %d ", i, j, k);
```

Boolean Values in C89

- For many years, the C language lacked a proper Boolean type, and there is none defined in the C89 standard.

- One way to work around this limitation is to declare an `int` variable and then assign it either 0 or 1:

```
int flag;

flag = 0;
…
flag = 1;
```

- Although this scheme works, it doesn't contribute much to program readability.

Boolean Values in C89

- To make programs more understandable, C89 programmers often define macros with names such as `TRUE` and `FALSE`:

```
#define TRUE 1
#define FALSE 0
```

- Assignments to `flag` now have a more natural appearance:

```
flag = FALSE;
…
flag = TRUE;
```

- To test whether `flag` is true, we can write

```
if (flag == TRUE) …
```
or
```
if (flag) …
```

- The latter form is more concise. It also works correctly if `flag` has a value other than 0 or 1.

- To test whether `flag` is false, we can write

```
if (flag == FALSE) …
```
or
```
if (!flag) …
```

Boolean Values in C89

- Carrying this idea one step further, we might even define a macro that can be used as a type:

```
#define BOOL int
```

- `BOOL` can take the place of `int` when declaring Boolean variables:

```
BOOL flag;
```

- It's now clear that `flag` isn't an ordinary integer variable, but instead represents a Boolean condition.

Boolean Values in C99

- C99 provides the `_Bool` type.
- A Boolean variable can be declared by writing

```
_Bool flag;
```

- `_Bool` is an integer type, so a `_Bool` variable is really just an integer variable in disguise.
- Unlike an ordinary integer variable, however, a `_Bool` variable can only be assigned 0 or 1.
- Attempting to store a nonzero value into a `_Bool` variable will cause the variable to be assigned 1:

```
flag = 5;   /* flag is assigned 1 */
```

- A `_Bool` variable can be tested in an `if` statement:

```
if (flag)   /* tests whether flag is 1 */
```

Boolean Values in C99

- C99's `<stdbool.h>` header makes it easier to work with Boolean values.
- It defines a macro, `bool`, that stands for `_Bool`.
- If `<stdbool.h>` is included, we can write

```
bool flag;    /* same as _Bool flag; */
```

- `<stdbool.h>` also supplies macros named `true` and `false`, which stand for 1 and 0, respectively, making it possible to write

```
flag = false;
…
flag = true;
```

The `if` Statement

- The `if` statement allows a program to choose between two alternatives by testing an expression.

- In its simplest form, the `if` statement has the form

  `if ( expression ) statement`

- When an `if` statement is executed, *expression* is evaluated; if its value is nonzero, *statement* is executed.

- Example:

```
if (line_num == MAX_LINES)
  line_num = 0;
```

The `if` Statement

- Confusing == (equality) with = (assignment) is perhaps the most common C programming error.

- The statement

  ```
  if (i == 0) …
  ```

  tests whether `i` is equal to 0.
- The statement

  ```
  if (i = 0) …
  ```

  assigns 0 to `i`, then tests whether the result is nonzero.

- Often the expression in an `if` statement will test whether a variable falls within a range of values.

- To test whether $0 \le$ `i` < `n`:

  ```
  if (0 <= i && i < n) …
  ```
- To test the opposite condition (`i` is outside the range):

  ```
  if (i < 0 || i >= n) …
  ```

Compound Statements

- In the `if` statement template, notice that *statement* is singular, not plural:

  `if (` *expression* `)` *statement*

- To make an `if` statement control two or more statements, use a *compound statement.* A compound statement has the form

  `{` *statements* `}`

- Putting braces around a group of statements forces the compiler to treat it as a single statement.

- Example:

```
{ line_num = 0; page_num++; }
```

- A compound statement is usually put on multiple lines, with one statement per line:

```
{
  line_num = 0;
  page_num++;
}
```

- Each inner statement still ends with a semicolon, but the compound statement itself does not.

Compound Statements

- Example of a compound statement used inside an `if` statement:

```
if (line_num == MAX_LINES) {
  line_num = 0;
  page_num++;
}
```

- Compound statements are also common in loops and other places where the syntax of C requires a single statement.

The `else` Clause

- An `if` statement may have an `else` clause:

  `if (` *expression* `)` *statement* `else` *statement*

- The statement that follows the word `else` is executed if the expression has the value 0.
- Example:

```
if (i > j)
  { max = i;}
else
  { max = j;}
```

- When an `if` statement contains an `else` clause, where should the `else` be placed?
- Many C programmers align it with the `if` at the beginning of the statement.
- Inner statements are usually indented, but if they're short they can be put on the same line as the `if` and `else`:

```
if (i > j) max = i;
else max = j;
```

The `else` Clause

- It's not unusual for `if` statements to be nested inside other `if` statements:

```
if (i > j)
  if (i > k)
    max = i;
  else
    max = k;
else
  if (j > k)
    max = j;
  else
    max = k;
```

- To avoid confusion, don't hesitate to add braces:

```
if (i > j) {
  if (i > k)
    max = i;
  else
    max = k;
} else {
  if (j > k)
    max = j;
  else
    max = k;
}
```

- Aligning each `else` with the matching `if` makes the nesting easier to see.

The `else` Clause

- **Some programmers use as many braces as possible inside `if` statements:**

```
if (i > j) {
  if (i > k) {
    max = i;
  } else {
    max = k;
  }
} else {
  if (j > k) {
    max = j;
  } else {
    max = k;
  }
}
```

- **Advantages of using braces even when they're not required:**
  - Makes programs easier to modify, because more statements can easily be added to any `if` or `else` clause.
  - Helps avoid errors that can result from forgetting to use braces when adding statements to an `if` or `else` clause.

Cascaded `if` Statements

- A "cascaded" `if` statement is often the best way to test a series of conditions, stopping as soon as one of them is true.
- Example:

```
if (n < 0)
  printf("n is less than 0\n");
else
  if (n == 0)
    printf("n is equal to 0\n");
  else
    printf("n is greater than 0\n");
```

- Although the second `if` statement is nested inside the first, C programmers don't usually indent it.

- Instead, they align each `else` with the original `if`:

```
if (n < 0)
  printf("n is less than 0\n");
else if (n == 0)
  printf("n is equal to 0\n");
else
  printf("n is greater than 0\n");
```

- This layout avoids the problem of excessive indentation when the number of tests is large:

```
if ( expression )
  statement
else if ( expression )
  statement
...
else if ( expression )
  statement
else
  statement
```

Program: Calculating a Broker's Commission

- When stocks are sold or purchased through a broker, the broker's commission often depends upon the value of the stocks traded.
- Suppose that a broker charges the amounts shown in the following table:

Transaction size	*Commission rate*
Under $2,500	$30 + 1.7%
$2,500–$6,250	$56 + 0.66%
$6,250–$20,000	$76 + 0.34%
$20,000–$50,000	$100 + 0.22%
$50,000–$500,000	$155 + 0.11%
Over $500,000	$255 + 0.09%

The minimum charge is $39.

- The `broker.c` program asks the user to enter the amount of the trade, then displays the amount of the commission:

```
Enter value of trade: 30000
Commission: $166.00
```

- The heart of the program is a cascaded `if` statement that determines which range the trade falls into.

```c
/* Calculates a broker's commission */

#include <stdio.h>

int main(void)
{
  float commission, value;

  printf("Enter value of trade: ");
  scanf("%f", &value);

  if (value < 2500.00f)
    commission = 30.00f + .017f * value;
  else if (value < 6250.00f)
    commission = 56.00f + .0066f * value;
  else if (value < 20000.00f)
    commission = 76.00f + .0034f * value;
  else if (value < 50000.00f)
    commission = 100.00f + .0022f * value;
  else if (value < 500000.00f)
    commission = 155.00f + .0011f * value;
  else
    commission = 255.00f + .0009f * value;

  if (commission < 39.00f)
    commission = 39.00f;

  printf("Commission: $%.2f\n", commission);

  return 0;
}
```

The "Dangling `else`" Problem

- When if statements are nested, the "dangling `else`" problem may occur:

```
if (y != 0)
  if (x != 0)
    result = x / y;
else
  printf("Error: y is equal to 0\n");
```

- The indentation suggests that the `else` clause belongs to the outer `if` statement.
- However, C follows the rule that an `else` clause belongs to the nearest `if` statement that hasn't already been paired with an `else`.
- A correctly indented version would look like this:

```
if (y != 0)
  if (x != 0)
    result = x / y;
  else
    printf("Error: y is equal to 0\n");
```

The "Dangling `else`" Problem

- To make the `else` clause part of the outer `if` statement, we can enclose the inner `if` statement in braces:

```
if (y != 0) {
  if (x != 0)
    result = x / y;
} else
  printf("Error: y is equal to 0\n");
```

- Using braces in the original `if` statement would have avoided the problem in the first place.

Week 4 Programming Projects

9. Write a program that prompts the user to enter two dates and then indicates which date comes earlier on the calendar:

```
Enter first date (mm/dd/yy): 3/6/08
Enter second date (mm/dd/yy): 5/17/07
5/17/07 is earlier than 3/6/08
```