Chapter 4

Expressions

Operators

- Expressions are built from variables, constants, and operators.
- C has a rich collection of operators, including
  - arithmetic operators
  - relational operators
  - logical operators
  - assignment operators
  - increment and decrement operators

Arithmetic Operators

- C provides five binary ***arithmetic operators:***
  - `+` addition
  - `-` subtraction
  - `*` multiplication
  - `/` division
  - `%` remainder
- An operator is ***binary*** if it has two operands.

Binary Arithmetic Operators

- The value of `i % j` is the remainder when `i` is divided by `j`.

  `10 % 3` has the value 1, and `12 % 4` has the value 0.

- Binary arithmetic operators—with the exception of `%`—allow either integer or floating-point operands, with mixing allowed.
- When `int` and `float` operands are mixed, the result has type `float`.

  `9 + 2.5f` has the value 11.5, and `6.7f / 2` has the value 3.35.

The / and % Operators

- The / and % operators require special care:
  - When both operands are integers, / "truncates" the result. The value of `1 / 2` is 0, not 0.5.
  - The % operator requires integer operands; if either operand is not an integer, the program won't compile.
  - Using zero as the right operand of either / or % causes undefined behavior.
  - In C99, the result of a division is always truncated toward zero and the value of `i % j` has the same sign as `i`.

Operator Precedence

- The arithmetic operators have the following relative precedence:

  Highest: `+ -` (unary)
  `* / %`
  Lowest: `+ -` (binary)

- Examples:

  `i + j * k` is equivalent to `i + (j * k)`
  `-i * -j` is equivalent to `(-i) * (-j)`
  `+i + j / k` is equivalent to `(+i) + (j / k)`

Assignment Operators

- ***Simple assignment:*** used for storing a value into a variable
- ***Compound assignment:*** used for updating a value already stored in a variable

Simple Assignment

- The effect of the assignment *v* = *e* is to evaluate the expression *e* and copy its value into *v*.
- *e* can be a constant, a variable, or a more complicated expression:

```
i = 5;                  /* i is now 5   */
j = i;                  /* j is now 5   */
k = 10 * i + j;         /* k is now 55  */
```

Compound Assignment

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

```
i = i + 2;
```

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

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

- There are other compound assignment operators, including the following:

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

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;
```

Increment and Decrement Operators

- 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" */
```

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" */
```

Chapter 5

Selection Statements

Statements

- Most of C's statements fall into three categories:
  - ***Selection statements:*** `if` and `switch`
  - ***Iteration statements:*** `while`, `do`, and `for`
  - ***Jump statements:*** `break` and `continue`

    (`return` also falls in this category.)
- Other C statements:
  - Compound statement
  - Null statement

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.

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)
  ```

Logical Operators

- 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.

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 < j) …
  ```

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

The `if` Statement

- 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) …
```

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;
```

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;
```

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

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");
```

Conditional Expressions

- C's ***conditional operator*** allows an expression to produce one of two values depending on the value of a condition.
- The conditional operator consists of two symbols (`?` and `:`), which must be used together:

  expr1 `?` *expr2* `:` *expr3*
- The operands can be of any type.
- The resulting expression is said to be a ***conditional expression.***

Conditional Expressions

- Example:

```
int i, j, k;

i = 1;
j = 2;
k = i > j ? i : j;          /* k is now 2 */
k = (i >= 0 ? i : 0) + j;   /* k is now 3 */
```

- The parentheses are necessary, because the precedence of the conditional operator is less than that of the other operators, with the exception of the assignment operators.

Conditional Expressions

- Conditional expressions tend to make programs shorter but harder to understand, so it's probably best to use them carefully.
- Conditional expressions are often used in `return` statements:

```
return i > j ? i : j;
```

Conditional Expressions

- Calls of `printf` can sometimes benefit from condition expressions. Instead of

```
if (i > j)
  printf("%d\n", i);
else
  printf("%d\n", j);
```

  we could simply write

```
printf("%d\n", i > j ? i : j);
```

- Conditional expressions are also common in certain kinds of macro definitions.

The `switch` Statement

- A cascaded `if` statement can be used to compare an expression against a series of values:

```
if (grade == 4)
  printf("Excellent");
else if (grade == 3)
  printf("Good");
else if (grade == 2)
  printf("Average");
else if (grade == 1)
  printf("Poor");
else if (grade == 0)
  printf("Failing");
else
  printf("Illegal grade");
```

The `switch` Statement

- The `switch` statement is an alternative:

```
switch (grade) {
  case 4:  printf("Excellent");
           break;
  case 3:  printf("Good");
           break;
  case 2:  printf("Average");
           break;
  case 1:  printf("Poor");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

The `switch` Statement

- A `switch` statement may be easier to read than a cascaded `if` statement.
- `switch` statements are often faster than `if` statements.
- Most common form of the `switch` statement:

```
switch ( expression ) {
  case constant-expression : statements
  …
  case constant-expression : statements
  default : statements
}
```

The `switch` Statement

- The word `switch` must be followed by an integer expression—the ***controlling expression***—in parentheses.
- Characters are treated as integers in C and thus can be tested in `switch` statements.
- Floating-point numbers and strings don't qualify, however.

The `switch` Statement

- Each case begins with a label of the form

  `case` *constant-expression* **:**
- A ***constant expression*** is much like an ordinary expression except that it can't contain variables or function calls.
  - `5` is a constant expression, and `5 + 10` is a constant expression, but `n + 10` isn't a constant expression (unless `n` is a macro that represents a constant).
- The constant expression in a case label must evaluate to an integer (characters are acceptable).

The `switch` Statement

- After each case label comes any number of statements.
- No braces are required around the statements.
- The last statement in each group is normally `break`.

The `switch` Statement

- Duplicate case labels aren't allowed.
- The order of the cases doesn't matter, and the `default` case doesn't need to come last.
- Several case labels may precede a group of statements:

```
switch (grade) {
  case 4:
  case 3:
  case 2:
  case 1:  printf("Passing");
           break;
  case 0:  printf("Failing");
           break;
  default: printf("Illegal grade");
           break;
}
```

Program: Printing a Date in Legal Form

- Contracts and other legal documents are often dated in the following way:

```
Dated this __________ day of __________ , 20__ .
```

- The `date.c` program will display a date in this form after the user enters the date in month/day/year form:

```
Enter date (mm/dd/yy): 7/19/14
Dated this 19th day of July, 2014.
```

- The program uses `switch` statements to add "th" (or "st" or "nd" or "rd") to the day, and to print the month as a word instead of a number.

date.c

```
/* Prints a date in legal form */

#include <stdio.h>

int main(void)
{
  int month, day, year;

  printf("Enter date (mm/dd/yy): ");
  scanf("%d /%d /%d", &month, &day, &year);

  printf("Dated this %d", day);
  switch (day) {
    case 1: case 21: case 31:
      printf("st"); break;
    case 2: case 22:
      printf("nd"); break;
    case 3: case 23:
      printf("rd"); break;
    default: printf("th"); break;
  }
  printf(" day of ");
```

```
  switch (month) {
    case 1:  printf("January");   break;
    case 2:  printf("February");  break;
    case 3:  printf("March");     break;
    case 4:  printf("April");     break;
    case 5:  printf("May");       break;
    case 6:  printf("June");      break;
    case 7:  printf("July");      break;
    case 8:  printf("August");    break;
    case 9:  printf("September"); break;
    case 10: printf("October");   break;
    case 11: printf("November");  break;
    case 12: printf("December");  break;
  }

  printf(", 20%.2d.\n", year);
  return 0;
}
```