Introduction to Computers and Programming

Lecture 2 – Expression & Selection

Chap 4 & 5

Tien-Fu Chen

Dept. of Computer Science and Information Engineering

National Yang Ming Chiao Tung Univ.

Take aways for this class

- Expression operators
- C has a rich collection of operators, including
  - arithmetic operators
  - relational operators
  - logical operators
  - assignment operators
  - increment and decrement operators
- Statements
- Assignment statement

- If/then/else selection
- Switch

Expressions

C Operators

- C emphasizes expressions rather than statements.
- 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

  and many others

Arithmetic Operators

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

Unary Arithmetic Operators

- The unary operators require one operand:

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

- The unary `+` operator does nothing. It’s used primarily to emphasize that a numeric constant is positive.

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.
  - The behavior when `/` and `%` are used with negative operands is ***implementation-defined*** in C89.
  - 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

- Does `i + j * k` mean "add `i` and `j`, then multiply the result by `k`" or "multiply `j` and `k`, then add `i`"?
- One solution to this problem is to add parentheses, writing either `(i + j) * k` or `i + (j * k)`.
- If the parentheses are omitted, C uses ***operator precedence*** rules to determine the meaning of the expression.

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)`

Operator Associativity

- ***Associativity*** comes into play when an expression contains two or more operators with equal precedence.
- An operator is said to be ***left associative*** if it groups from left to right.
- The binary arithmetic operators (`*`, `/`, `%`, `+`, and `-`) are all left associative, so

  `i - j - k` is equivalent to `(i - j) - k`

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

Operator Associativity

- An operator is ***right associative*** if it groups from right to left.
- The unary arithmetic operators (`+` and `-`) are both right associative, so

  `- + i` is equivalent to `-(+i)`

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 */
```

Simple Assignment

- If *v* and *e* don't have the same type, then the value of *e* is converted to the type of *v* as the assignment takes place:

```
int i;
float f;

i = 72.99f;    /* i is now 72 */
f = 136;       /* f is now 136.0 */
```

Simple Assignment

- In many programming languages, assignment is a statement; in C, however, assignment is an operator, just like `+`.
- The value of an assignment *v* = *e* is the value of *v* after the assignment.
  - The value of `i = 72.99f` is 72 (not 72.99).

Side Effects

- An operators that modifies one of its operands is said to have a ***side effect.***
- The simple assignment operator has a side effect: it modifies its left operand.
- Evaluating the expression `i = 0` produces the result 0 and—as a side effect—assigns 0 to `i`.

Side Effects

- Since assignment is an operator, several assignments can be chained together:

```
i = j = k = 0;
```

- The `=` operator is right associative, so this assignment is equivalent to

```
i = (j = (k = 0));
```

Side Effects

- Watch out for unexpected results in chained assignments as a result of type conversion:

```
int i;
float f;

f = i = 33.3f;
```

- `i` is assigned the value 33, then `f` is assigned 33.0 (not 33.3).

Side Effects

- 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.

Lvalues

- 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."*

*Remembering the mnemonic, that **l-values** can appear on the left of an assignment operator while **r-values** can appear on the right.*

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; */
```

Compound Assignment

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

  $-= \quad *= \quad /= \quad \%=$

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

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

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

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

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

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

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.

Compound Assignment

- 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;
```

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

- `++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" */
```

Increment and Decrement Operators

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

Increment and Decrement Operators

- 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

Expression	*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

Order of Subexpression Evaluation

- The value of an expression may depend on the order in which its subexpressions are evaluated.
- C doesn't define the order in which subexpressions are evaluated (with the exception of subexpressions involving the logical and, logical or, conditional, and comma operators).
- In the expression `(a + b) * (c - d)` we don't know whether `(a + b)` will be evaluated before `(c - d)`.

Order of Subexpression Evaluation

- Most expressions have the same value regardless of the order in which their subexpressions are evaluated.
- However, this may not be true when a subexpression modifies one of its operands:

```
a = 5;
c = (b = a + 2) - (a = 1);
```

- The effect of executing the second statement is undefined.

Order of Subexpression Evaluation

- To prevent problems, it's a good idea to avoid using the assignment operators in subexpressions.
- Instead, use a series of separate assignments:

```
a = 5;
b = a + 2;
a = 1;
c = b - a;
```

The value of `c` will always be 6.

- Only operators that modify their operands are increment and decrement.
- When using these operators, an expression doesn't depend on a particular order of evaluation.

Order of Subexpression Evaluation

- Example:

```
i = 2;
j = i * i++;
```

- It's natural to assume that `j` is assigned 4. However, `j` could just as well be assigned 6 instead:
  1. The second operand (the original value of i) is fetched, then `i` is incremented.
  2. The first operand (the new value of `i`) is fetched.
  3. The new and old values of `i` are multiplied, yielding 6.

Undefined Behavior

- Statements such as `c = (b = a + 2) - (a = 1);` and `j = i * i++;` cause ***undefined behavior.***
- Possible effects of undefined behavior:
  - The program may behave differently when compiled with different compilers.
  - The program may not compile in the first place.
  - If it compiles it may not run.
  - If it does run, the program may crash, behave erratically, or produce meaningless results.
- Undefined behavior should be avoided.

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.

Expression Statements

- 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 */
```

Expression Statements

- 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."*

Selection statements

Statements

- Most of C's remaining statements fall into three categories:
  - ***Selection statements:*** `if` and `switch`
  - ***Iteration statements:*** `while`, `do`, and `for`
  - ***Jump statements:*** `break`, `continue`, and `goto`. (`return` also belongs in this category.)
- Other C statements:
  - Compound statement
  - Null statement

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`.
- 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).

```
A = i<j ;
```

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.

Relational Operators

- 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 left associative.

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)
  ```

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.

Logical Operators

- 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 Operators

- 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 Operators

- 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.

Logical Operators

- 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.

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.

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 ≤ `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.

Compound Statements

- 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;
```

The `else` Clause

- 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;
```

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

The `else` Clause

- 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;
  }
```

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;
  }
}
```

The `else` Clause

- 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");
```

Cascaded `if` Statements

- 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");
```

Cascaded `if` Statements

- 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
```

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`.

The "Dangling `else`" Problem

- 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.

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

- The conditional operator requires three operands, so it is often referred to as a ***ternary*** operator.
- The conditional expression *expr1* `?` *expr2* **`:`** *expr3* should be read "if *expr1* then *expr2* else *expr3*."
- The expression is evaluated in stages: *expr1* is evaluated first; if its value isn't zero, then *expr2* is evaluated, and its value is the value of the entire conditional expression.
- If the value of *expr1* is zero, then the value of *expr3* is the value of the conditional.

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 discussed so far, 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 sparingly.
- 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.

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;
```

Boolean Values in C89

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

  ```
  if (flag == TRUE) …
  ```

  or just

  ```
  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 */
```

Boolean Values in C99

- It's legal (although not advisable) to perform arithmetic on `_Bool` variables.
- It's also legal to print a `_Bool` variable (either 0 or 1 will be displayed).
- And, of course, 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 `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;
}
```

The `switch` Statement

- To save space, several case labels can be put on the same line:

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

- If the `default` case is missing and the controlling expression's value doesn't match any case label, control passes to the next statement after the `switch`.

The Role of the `break` Statement

- Executing a `break` statement causes the program to "break" out of the `switch` statement; execution continues at the next statement after the `switch`.
- The `switch` statement is really a form of "computed jump."
- When the controlling expression is evaluated, control jumps to the case label matching the value of the `switch` expression.
- A case label is nothing more than a marker indicating a position within the `switch`.

The Role of the `break` Statement

- Without `break` (or some other jump statement) at the end of a case, control will flow into the next case.
- Example:

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

- If the value of `grade` is 3, the message printed is

```
GoodAveragePoorFailingIllegal grade
```

The Role of the `break` Statement

- Omitting `break` is sometimes done intentionally, but it's usually just an oversight.
- It's a good idea to point out deliberate omissions of `break`:

```
switch (grade) {
  case 4: case 3: case 2: case 1:
          num_passing++;
          /* FALL THROUGH */
  case 0: total_grades++;
          break;
}
```

- Although the last case never needs a `break` statement, including one makes it easy to add cases in the future.

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

```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 ");
```

```c
  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;
}
```