Chapter 5

Selection Statements

Statements

- So far, we've used `return` statements and *expression statements*, (`printf` and `scanf` are function calls)
- Most of **C**'s remaining statements fall into three categories:
  - ***Selection statements:*** `if` and `switch`
  - ***Iteration statements:*** `while`, `do`, and `for` *(next chapter)*
  - ***Jump statements:*** `break`, `continue`, and `goto` *(next chapter)* (`return` also belongs to this category)
- Other **C** statements
  - ***Compound statement***
  - ***Null statement***

Logical Expressions

- Several of C's statements must test the value of a logical expression to see if it is "true" or "false"
- Logical expressions produce an integer value, either: zero (false) or one (true)

Relational Operators

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

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

- The relational operators can be used to compare integers and floating-point numbers
- 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

Equality Operators

- `C` provides two ***equality operators***

  `==` equal to

  `!=` not equal to

- The equality operators are left associative and produce an integer value, either: zero (false) or one (true)
- The equality operators have lower precedence than the relational operators, so the expression

  `i < j == j < k` is equivalent to `(i < j) == (j < k)`

  `i + j < k + m == n + p < s + t` is equivalent to

  `((i + j) < (k + m)) == ((n + p) < (s + t))`

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 an integer value, either: zero (false) or one (true)
- *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

expr1	! *expr1*
0	1
nonzero	0

expr1	*expr2*	*expr1* && *expr2*
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

expr1	*expr2*	*expr1* || *expr2*
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

Logical 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 zero (false) or one (true) produced by `i < j` is then compared to `k`
- The correct logical expression to test whether `j` lies between `i` and `k` is `i < j && j < k`

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*

Expression Evaluation

Precedence	*Name*	*Symbol(s)*		*Associativity*
1	increment (postfix)	`++`	(i.e., `i++`)	left
	decrement (postfix)	`--`	(i.e., `i--`)	left
2	increment (prefix)	`++`	(i.e., `++i`)	right
	decrement (prefix)	`--`	(i.e., `--i`)	right
	unary plus	`+`		right
	unary minus	`-`		right
	unary negation	`!`		right
3	multiplicative	`* / %`		left
4	additive	`+ -`		left
5	*relational operators*	`< > <= >=`		left
6	*equality operator*	`== !=`		left
7	*logical and*	`&&`		left
8	*logical or*	`||`		left
9	assignment	`= *= /= %= += -=`		right

Short Circuit

- Both `&&` and `||` perform "*short-circuit*" evaluation:
  - they first evaluate the *left operand*, then the *right one* ***(if needed)***
- If the value of the expression can be decided 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

Short Circuit

- 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

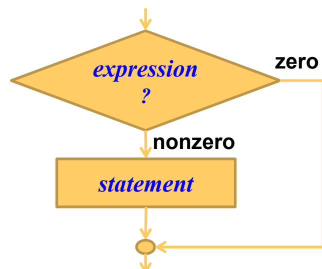
The `if` Statement

- 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)` *`statement`*

  tests whether `i` is equal to `0`
  - If true, the *statement* will be executed
- The statement

  `if (i = 0)` *`statement`*

  assigns `0` to `i`
  - The result of this expression is `0`, which is false
  - Hence, the *statement* will never be executed

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), we use *De Morgan's laws* (***to be discussed in the next lab***)

```
if (0 > i || 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
- 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

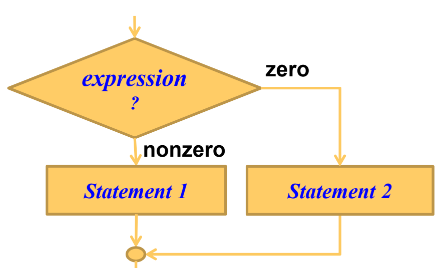
The `else` Clause

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

  `if (` *expression* `)` *statement_1* `else` *statement_2*

- 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

- Many **C** programmers align `else` with the `if` at the beginning of the statement, where inner statements are usually indented

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

- But if they're short they can be put as follow

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

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

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

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
- Using braces even when they're not required makes programs easier to modify

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

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");
```


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 + 1.70%
$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

Program: Calculating a Broker's Commission

- 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

broker.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` belongs to the outer `if`
- 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");
```


Conditional Expressions

- The ***conditional expression*** *expr1* `?` *expr2* `:` *expr3* consists of
  - ***Two*** symbols (`?` and `:`), which ***must be used together***
  - ***Three*** expressions, which can be of any type
- It 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 : 5) + j;    /* k is now 3 */
```

- The parentheses are necessary, because the precedence of the conditional operator is lower than that of the other operators discussed so far, with the exception of the assignment operators

```
k = i >= 0 ? i : 5 + j;    /* k is now 1 */
```

- Conditional expressions tend to make programs shorter but harder to understand

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 often used in `return` statements

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

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, and hence a `_Bool` variable is really just an integer variable in camouflage
- ***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 */
```

This is a huge difference between C89 and C99

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
- The most common form of the `switch` statement is

`switch (`***integer*** *expression* `) {`

`case` *constant-expression* `:` *statements*

…

`case` *constant-expression* `:` *statements*

`default :` *statements*

`}`

The `switch` Statement

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

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
  - `5 + 10` is a constant expression
  - `n + 10` isn't a constant expression (unless `n` is a macro that represents a constant)
- The constant expression in a case label *must be* evaluated to an *integer*

The `switch` Statement

- The `switch` statement is really a form of *computed jump*

  When the controlling expression is evaluated, the 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`

Chapter 5: Selection Statements

The `switch` Statement

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

- Without `break` at the end of a case, control will flow into the next case
- Example

```
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

- Without `break` 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 `switch` Statement

- The order of the cases doesn't matter
- The `default` case doesn't need to come last
- 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`

Chapter 5: Selection Statements

The `switch` Statement

- Duplicate case labels aren't allowed
- Several case labels may precede a group of statements
- 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;
}
```

The `switch` Statement

- Omitting `break` is sometimes done intentionally
- 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

```
/*Prints a date in legal form, I.e. Dated this 19th day of July,2014.*/
#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;
  }
```



```
/*Prints a date in legal form, I.e. Dated this 19th day of July,2014.*/
 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;
}
```