Problem Solving and Program Design - Chapter 4

Cory L. Strope

- Control Structure
- Conditions
- `if` statements

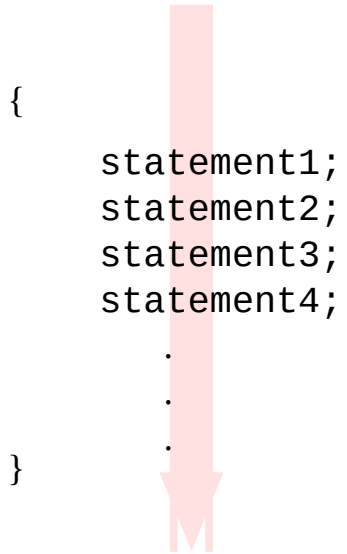
Control Structure

- **Control structures**:
  - Control the flow of execution in a program or function.
  - Enable you to combine individual instructions into a single logical unit with one entry point (i.e. *int main(void)* {) and one exit point (*return 0;* }).
- Three kinds of structures to control execution flow:
  - Sequence,
  - Selection, and
  - Repetition.

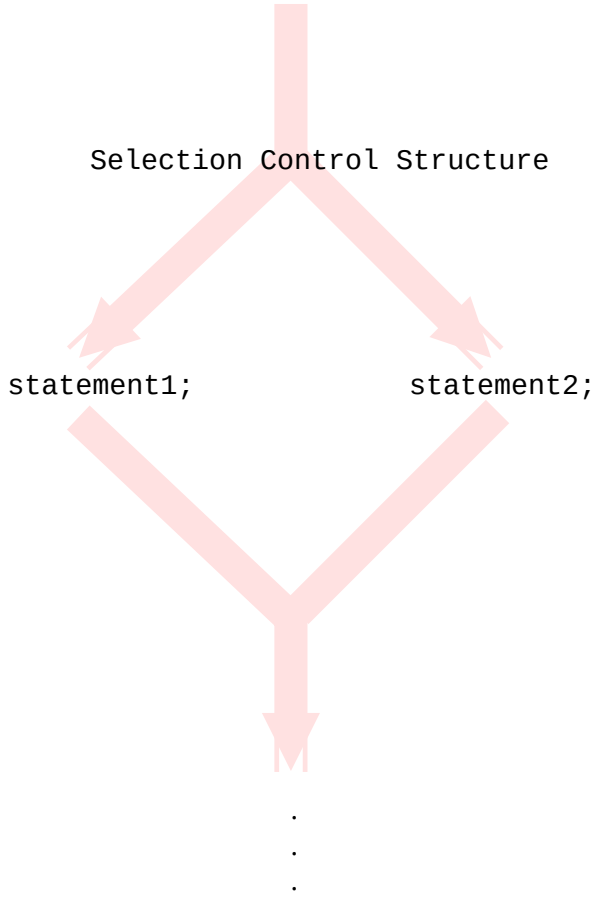
Sequential Flow

Compound statement:

- Written as a group of statements
- Bracketed by { and }, and
- Used to specify sequential flow.



Selection Flow



Selection control structure:

- Evaluates criteria to determine which alternative "path" to follow.

Selection Flow – Conditions

condition - An expression that is either `true` or `false`.

A program chooses alternative paths by testing a (set of) condition(s).

- (ConditionEval == `1`) $\rightarrow$ `true`,
- (ConditionEval == `0`) $\rightarrow$ `false`.
- The resting heart rate is a good indicator of health
- `if` resting_heart_rate $<$ 75 `then` you are in good health.
  - if resting heart rate is 80, ConditionEval is `false`.
  - if resting heart rate is 50, ConditionEval is `true`.
  - if resting heart rate is 75, what is ConditionEval?

Relational and Equality Operators

Conditions come in four forms:

- `variable` *relational-operator* `variable`
- `variable` *relational-operator* `CONSTANT`
- `variable` *equality-operator* `variable`
- `variable` *equality-operator* `CONSTANT`

Operator	**Meaning**	**Type**
$<$	less than	relational
$>$	greater than	relational
$<=$	less than or equal to	relational
$>=$	greater than or equal to	relational
$==$	equal to	equality
$!=$	*not* equal to	equality

What about more than one condition?

Logical Operators

Logical operators: Operators that can combine conditions to make more complicated selection statements.

&&	logical *and*
||	logical *or*
!	logical *complement*, *negation*, or **not**.

logical expressions - expressions that involve conditional statement(s) and logical operator(s).

temperature $>$ 90.0 && humidity $>$ 0.90
Are we going to go or not?
`(go || !go) = ?`

Operator Tables for `&&`, `||`, and `!`

`true` = Any nonzero value $(-1, 0.00001, 20000000, 3.1415, \ldots)$
`false` = 0.

op	operand1	operand2	operand1 *op* operand2
&&	`true`	`true`	`true`
	`true`	`false`	?
	`false`	`true`	`false`
	`false`	`false`	`false`
||	`true`	`true`	`true`
	`true`	`false`	?
	`false`	`true`	`true`
	`false`	`false`	`false`
!	NULL	`true`	`false`
	NULL	`false`	`true`

Operator Precedence

```
high   function calls
       !,+,-,& (unary operations),      if(!done) { ; }
       *,/,%,
       +,-,
       <,>,<=,>=,
       ==,!=,
       &&,                        if(x > 0 && x < 6) { ; }
       ||,
low    =
```

Rest of Section

- Short-circuit evalutation:
  - C stops evaluating after the end condition of a statement is known
    - `true || anything` = `true`.
    - `false && anything` = `false`.
- Writing english conditions in C:
  - Is your C condition is logically equivalent to the english statement?
- Comparing characters:
  - We can compare characters in C using the relational and equality operators.
- Logical assignment:
  - Assign an int a value of nonzero for true or zero for false

Comparing Characters

We can also compare characters using relational and equality operators.

- Comparisons are based on the bit values of the characters, as found in the ASCII table.

Expressions	Value
'9' < '0'	1 (true)
'a' < 'e'	1 (true)
'B' <= 'A'	0 (false)
'Z' == 'z'	0 (false)
'a' < 'A'	system dependent
'a' <= ch && ch <= 'z'	1 (true) if ch is a lowercase letter

The if Statement

- `if` Statement with Two Alternatives
- `if` Statement with One Alternative
- A Comparison of One and Two Alternative `if` Statements
- Program Style

if Statement with Two Alternatives

- Conditions are used to assign **boolean** (T,F) values to variables
  - `senior_citizen = (age >= 65)`
- More often, conditions are used to make a choice between alternatives, through the `if` statement.

```
if (!senior_citizen)
   printf("Your hamburger is $3.50\n");
else
   printf("Your hamburger is $2.50!\n");
```

This `if` statement selects one of the two calls to printf. If the condition is 1 (`true`) it chooses the first `printf`, if the condition is 0 (`false`) it chooses the second `printf`.

if Statement with One Alternative

- `if` statements with two alternatives choose a statement to execute, however
- `if` statements with one alternative decide whether a statement should be executed based on a condition.

For example, when we make a division, we do not want to divide by zero, so:

```
if ( x != 0 )
   product = product / x;
```

if Statements

```
if crsr_or_frgt == 'C'
   printf("Cruiser\n");
printf("Combat ship\n");
```

```
if (crsr_or_frgt == 'C');
   printf("Cruiser\n");
printf("Combat ship\n");
```

Program Style

- Statements following the `if` statements should be indented
- `else` statement is at the same indentation as the `if` statement
- Statements following the `else` statements should be indented.

The format of the `if` statement makes its meaning apparent and is used solely to improve program readability.

```
if(condition) {
    statement_T
} else {
    statement_F
}
```

if Statement with Compound Statements

- In previous slides, `if` and `else` statements have performed only one operation if `true`
- C always assumes that each `if` or `else` statement will be followed by one operation.
  - `if(true)` Operation;
- If more than one statement needs to be done for an `if` or `else`, we use $\{\}$ to group a set of statements into one compound statement.

```
if (pop_today > pop_yesterday) {
    growth = pop_today - pop_yesterday;
    growth_pct = 100.0 * growth / pop_yesterday;
    printf("The growth percentage is %.2f.\n", growth_pct);
}
```

Another Example

```
if (crash_test_rating_index <= MAX_SAFE_CTRI) {
   printf("Car #%d: safe\n", auto_id);
   safe = safe + 1;
} else {
   printf("Car #%d: unsafe\n", auto_id);
   unsafe = unsafe + 1;
}
```

- If you omit the braces, what happens?
- Placement of braces and keywords is optional

```
if(cond) {
} else {
}
```

Tracing an if Statement

- Verifying the correctness of a C statement before running the program
  - Catching logical errors will save a lot of time in debugging.
- A *hand trace* or *desk check* is a step-by-step simulation of each step of the program, as well as how the values of the variables change at each step.

```
if(x > y) {
    temp = x;
    x = y;
    y = temp;
}
```

Section 4.5 shows how to use decision steps (`if` statements) in algorithm design.

Also as a side note it is common to use Constant Macros in conditional checks to enhance readability and ease maintenance.

Nested if Statements and Multiple-Alternative Decisions

- √ No decisions: Sequential program.
- √ One decision: `if-then` (One alternative)
  - `if (cond) statement;`
- √ Decision between two alternatives: `if-then-else` (Two alternative statements)
  - `if (cond) statement1; else statement2;`
- → Decisions between many alternatives
  - School level

Nested if Statements and Multiple-Alternative Decisions

```
if (x < 0)
    pre_school = pre_school + 1;
else
    if (x <= 12)
        public_school = public_school + 1;
    else
        univ_career_homeless = univ_career_homeless + 1;
```

```
if (x <= 0)
   pre_school = pre_school + 1;
if (x <= 12 && x > 0)
   public_school = public_school + 1;
if (x > 12)
   univ_career_homeless = univ_career_homeless +1;
```

Nested ifs vs. Sequence of ifs

- Beginning programmers sometime prefer to use a sequence of `if` statements

```
if (x <= 0)
   pre_school = pre_school + 1;
if (x <= 12 && x > 0)
   public_school = public_school + 1;
if (x > 12)
   univ_career_homeless = univ_career_homeless +1;
```

- However, it is neither as readable nor as efficient.
  - Not as readable, since the sequence does not clearly show that exactly one of the three assignment statements is executed for a particular x.
  - It is less efficient because all three of the conditions are always tested. In the nested `if` statement, only the first condition is tested when x is positive.

Multiple-Alternative Decision Form of Nested if

Nested `if` statements can become quite complex.

- In situations where there are multiple `if-then-else` statements, it is easier to code the multiple alternative as shown below.

```
if ( condition_1 )
   statement_1
else if ( condition_2 )
   statement_2
.
.
.
else if ( condition_n )
   statement_n
else
   statement_e
```

Example – Range Elimination

We want to describe noise loudness measured in decibels with the effect of the noise. The following table shows the relationship between noise level and human perceptions of noises.

Loudness in Decibels (db)	Perception
50 or lower	quiet
51 - 70	intrusive
71 - 90	annoying
91 - 110	very annoying
above 110	uncomfortable

Example in C code

```
if ( loudness <= 50 )
   printf("quiet");
else if ( loudness <= 70 )
   printf("intrusive");
else if ( loudness <= 90 )
   printf("annoying");
else if ( loudness <= 110 )
   printf("very annoying");
else
   printf("uncomfortable");
```

Multiple-Alternative if, Order of Conditions

- When more than one condition in a multiple-alternative decision is true, only the task following the first true condition executes.
  - The order of the conditions can affect the outcome.
- The order of conditions also effect program efficiency.
  - If loud noises are much more likely, it is more efficient to test first for noise levels above 110 db, then for levels between 91 and 110 db, and so on.

Nested if Statements with More Than One Variable

- In most of our examples, `if` statements test the value of a single variable
  - We have been able to write each nested if statement as a multiple-alternative decision.
  - `if(x>y) stmt; else if(x<y) stmt; else stmt;`
- If several variables are involved in the decision, we cannot always use a multiple-alternative decision. However, we can use nested `if` statement as a "filter" to select data that satisfy several different criteria.

Code Example

The Department of Defense would like a program that identifies **singles males** between the **ages of 18 and 26**, inclusive.

```
/* Print a message if all criteria are met.*/
if ( marital_status == 'S' )
   if ( gender == 'M' )
      if ( age >= 18 && age <= 26 )
         printf("All criteria are met.\n");
```

or:

```
if ( maritial_status == 'S' && gender == 'M' &&
 age >= 18 && age <= 26 )
   printf("All criteria are met.\n");
```

Switch

- The `switch` statement is similar to a multiple-alternative `if` statement, but can be used only for type `char` or type `int` expressions.
- Useful when the selection depends on the value of a single variable (called the controlling variable)
- Expressions in the `switch` statement must cover all possible values of the controlling variable.
  - Each viable expression $\longrightarrow$ `case` statement
  - All other values $\longrightarrow$ *fall-through* (`default:`) statement.

Switch Example

```
#include <stdio.h>
int main(void) {
    char class;
    scanf("%c",&class);

    switch (class) {
    case 'B':
    case 'b':
        printf("Battleship\n");
        break;
    case 'C':
    case 'c':
        printf("Cruiser\n");
        break;
    default:
        printf("Unknown ship class%c\n", class);
    }
}
```

Common Errors

- Using a **string** such as "Cruiser" or "Frigate" as a case label.
  - String = 'C' 'r' 'u' 'i' 's' 'e' 'r'
  - Why can't a `switch` statement use a type `double` value?
- The omission of the `break` statement at the end an alternative causes the execution to "fall through" into the next alternative.
- Forgetting the closing brace of the `switch statement body.`

Nested if versus switch

- A nested `if` is more general then the `switch`
  - `if`: Can check any number of any data type variables vs. one value for `int` or `char` data type.
- `if`: Can use a range of values, such as < 100
- `switch`: More readable
- `switch`: Can not compare strings or `doubles`
- `switch`: Can not handle a range of values in one case label
- Use the switch whenever there are ten or fewer case labels
- Use the default label whenever possible

Common Programming Errors

- `if (0 ≤ x ≤ 4)` is **always** `true`, first it does $0 \leq$ `x` which is `true` or `false`, so it evaluates to **1** for `true` and **0** for `false` and then it takes that value, 0 or 1, and does $1 \leq 4$ or $0 \leq 4$ and both are always `true`.
  - In order to check a range, use $(0 \leq 4$ `&&` `x` $\leq 4)$.
- `if (x` $= 10)$ is always true, the = symbol assigns `x` the value of 10, so the conditional statement evaluates to 10, and since 10 is nonzero this is `true`.

Common Errors cont.

- Don't forget to parenthesize the condition.
- Don't forget the { and } `if` they are needed.
- When doing nested `if` statement, try to select conditions so that you can use the range-elimination multiple-alternative format.
- C matches each `else` with the closest unmatched `if`, so be careful so that you get the correct pairings of `if` and `else` statements.
- In `switch` statements, make sure the controlling expression and case labels are of the same permitted type.
- Remember to include the `default` case for `switch` statements.
- Don't for get your { and } for the `switch` statement.
- **Don't forget your `break` statement.**

What we learned in Chapter 4

```
if (x == 0)
   statement_T;

if (x == 0)
   statement_T;
else
   statement_F;

if (x == 0) {
   statements_T;
}
```

```
if (x == 0) {
   statements_T
}
else {
   statements_F
}

if (x >= 0)
   if (x == 0)
      statement_TT
   else
      statement_TF
else
   statement_F
```

```
switch (x) {
case 1:
      true if x == 1 statement
      break;
case 2:
      true if x == 2 statement
      break;
default:
      always true
}
```

End Chapter 4, any questions?