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) { return 0; }`) and one exit point.
  - 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 `}`
- Used to specify sequential flow
- All statements are unconditionally executed
- Unconditionally executed
- Order is important

Selection Flow

**Selection control structure:**
- Evaluates criteria to determine which alternative “path” to follow.
- A **control structure** determines which statement(s) to execute.
- Statements are mutually exclusive

Selection Flow – Conditions

A condition is an expression that is either **true** or **false**.

A program chooses alternative paths of computation by testing one or more conditions.

- `(ConditionEval == 1) -> true`,
- `(ConditionEval == 0) -> 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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>relational</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>relational</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td>relational</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td>relational</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
<td>equality</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
<td>equality</td>
</tr>
</tbody>
</table>

Table: Relational and Equality Operators in C

### Logical Operators

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

<table>
<thead>
<tr>
<th>C Syntax</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical NOT (negation)</td>
</tr>
</tbody>
</table>

Table: Logical Operators in C

### Logical Expressions

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

Examples:

- \((x >= 0 \&\& x <= 10)\)
- \((\text{temperature} > 90.0 \&\& \text{humidity} > 0.90)\)
- \(!((x >= 0) \&\& x <= 10)\)

What about the following: Are we going to go or not?
\((\text{go} || !\text{go})\)

### Distributivity

- The logical \textit{NOT} can be \textit{distributed} over a logical expression just as multiplication can be over an algebraic expression.
- \(a(b + c) = ab + ac\)
- When distributing the logical \textit{NOT}, \textit{AND} and \textit{OR} are reversed!

Examples:

- \(!((x >= 0) \&\& x <= 10)\)
- \(!((x >= 0) \&\& x <= 10)\)
- \(!((x < 0) \&\& (x > 10))\)

Best to simplify logical expressions as much as possible, but more important to keep code readable.

### Tautologies & Contradictions

- A \textit{tautology} is a logical expression that is \textit{always} true
  - Any non-zero constant (1, 1.5, 8, etc.)
  - An expression that, when simplified, always ends up being true
    - \((\text{go} || !\text{go})\) is always true
- A \textit{contradiction} is a logical expression that is \textit{always} false
  - The zero constant (0)
  - An expression that, when simplified, always ends up being false
    - \((\text{go} && !\text{go})\) is always false
**True and False**

*C Convention*

- For convenience when writing we identify zero with *false* and one with *true*
- C does not recognize the words *true*, *false*
- C has no built-in *Boolean* type!
- Instead, zero is identified with *false*
- Any non-zero value is identified with *true*
- Example: -1, 0.01, 386 are all *true*

**Operator Tables**

**Logical And**

The result of taking a logical *And* with two operands is true if and only if *both* operands are *true*. Otherwise it is *false*.

<table>
<thead>
<tr>
<th>Operand A</th>
<th>Operand B</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table: Logical And Operator

**Logical Or**

The result of taking a logical *Or* with two operands is true if and only if *at least one* of the operands is *true*. Otherwise it is *false*.

<table>
<thead>
<tr>
<th>Operand A</th>
<th>Operand B</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table: Logical Or Operator

**Operator Precedence**

Recall the order of precedence for operators (Table 1, inside cover).

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>!</td>
<td>right</td>
</tr>
<tr>
<td></td>
<td>+ = (unary)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>* / %</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td>* = (binary)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>&lt; &gt; &lt;= &gt;= (unary)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>!= (unary)</td>
<td>left-to-right</td>
</tr>
<tr>
<td></td>
<td>&amp;&amp; !</td>
<td>(unary)</td>
</tr>
<tr>
<td>Low</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table: Order of Precedence for Operators

**Short-Circuiting**

- If the first operand of a logical *Or* is true the whole expression is true regardless of the second operand.
- Similar, if the first operand of a logical *And* is false, the whole expression is false regardless of the second operand.
  - *(true || anything) is true*
  - *(false && anything) is false*
- By convention, in either case C does not bother to evaluate the second operand.
- This is known as short-circuiting
Programming Tip

- Writing pseudocode will help you to write logical expressions in plain English.
- Translate the expressions into valid C syntax
- Be sure that the original and the translation are logically equivalent
- You can use a int type to store true/false:
  ```c
  int someBoolean = 0;
  ```

Comparing Characters

- Recall that C uses partially weak typing
- C treats characters as integers in the range [0, 255]
- Thus, it makes sense that we can compare characters using relational and equality operators.
- Comparisons are based on the ASCII values (Appendix A)
- Example: ‘a’ < ‘e’ is true since 97 < 101

Exercise

What are the values of the following character comparisons?

1. 'B' <= 'A'
2. 'Z' == 'z'
3. 'A' < 'a'
4. '5' <= '7'

Answer:

1. false since 66 > 65
2. false since 90 != 122
3. true since 65 < 97
4. true since 53 ≤ 55

Comparing Characters

- ASCII stands for American Standard Code for Information Interchange
- The ASCII character set was designed to preserve alpha-numeric order
- Capital letters are less than lower-case letters
- Other neat characteristics that involve binary

The if Statement

- Conditions are used to assign boolean (T,F) values to variables
- Example: `senior_citizen = (age >= 65)`
- 0 or 1 is assigned to `senior_citizen` depending on the value of `age`
- More often, conditions are used to make a choice between alternatives, through the if statement.
- If the condition is true, one statement is executed, otherwise, another statement is executed.

```c
if (! senior_citizen )
  printf("Your hamburger is $3.50\n");
else
  printf("Your hamburger is $2.50\n");
```
if Statement with One Alternative

- It is not necessary to specify an alternative (else statement)
- An if statement can determine to execute a statement or not

```
if(senior_citizen)
    price = price - 1.0;
```

Programming Tip

- Recall that division by zero is undefined (and dangerous)
- You can use an if statement to avoid such errors

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

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

Programming Tip

- Pitfall: Do not end an if statement with a semi-colon:
- Syntactically correct, program will compile
- Essentially like if(price<0){};
- Will not give expected results
- The if statement expects an alternative: the semicolon ends the alternative
- Thus, The product is free! will be printed regardless of the value of price

if Statement with Compound Statements

- In previous slides, if and else statements have performed only one operation
- C always assumes that each if or else statement will be followed by one 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("Growth percentage = %.2f\n", growth_pct);
}
```
Another Example

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

If you omit the braces, what happens?

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.

Nested if Statements and Multiple-Alternative Decisions

- No decisions: Sequential program
- One decision: if-then (One alternative)
  - if(condition) statement;
- Decision between two alternatives: if-then-else (Two alternative statements)
  - if(condition) statement; else statement2;
- Decisions between many alternatives
  - School level

Nested ifs vs. Sequence of ifs I

Beginning programmers sometime prefer to use a sequence of if statements

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

Nested ifs vs. Sequence of ifs II

- Not as readable: since the sequence does not clearly show that exactly one of the three assignment statements is executed for a particular x.
- 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.
- Can lead to logical errors
Nested ifs vs. Sequence of ifs III

```java
if(score >= 90)
    grade = 'A';
if(score >= 80)
    grade = 'B';
if(score >= 70)
    grade = 'C';
```

What happens when `score = 95`?

if-else-if Statement

Better solution: the if-else-if statement

```java
if ( condition_1 )
    statement_1
else if ( condition_2 )
    statement_2
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.

<table>
<thead>
<tr>
<th>Loudness in Decibels (db)</th>
<th>Perception</th>
</tr>
</thead>
<tbody>
<tr>
<td>50 or lower</td>
<td>quiet</td>
</tr>
<tr>
<td>51 - 70</td>
<td>intrusive</td>
</tr>
<tr>
<td>71 - 90</td>
<td>annoying</td>
</tr>
<tr>
<td>91 - 110</td>
<td>very annoying</td>
</tr>
<tr>
<td>above 110</td>
<td>uncomfortable</td>
</tr>
</tbody>
</table>

Table:

Example in C code

```c
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

- With if-else-if statements, one and only one statement is ever executed
- Moreover the first satisfied condition is the one that is executed
- The order of the conditions can affect the outcome
- The order of conditions also affect program efficiency
- The most common cases should be checked first (if known)
  - 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.

Code Exercise

Exercise

The Department of Defense would like a program that identifies singles males between the ages of 18 and 26, inclusive. Design a logical expression that captures this.
Can this be improved?

Better Solution

```c
/* 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");
```

Switch Example

```c
#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);
  }
  return 0;
}
```

Common Errors

- You cannot use a string such as "Cruiser" or "Frigate" as a case label.
- The omission of the `break` statement at the end of 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 than a `switch` statement
  - `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 `double` types
  - `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 I

- $(0 \leq x \leq 4)$ is always true
  - Associativity: first $0 \leq x$ is evaluated (true or false)
  - Thus, it evaluates to either 1 or and 0
  - In either case, both are less than 4
  - Thus the entire expression is true regardless of the value of $x$
- if$(x = 10)$ is always true: the assignment operator is evaluated and $x$ is given a value of 10, which is true

Common Programming Errors II

- Don’t forget to parenthesize the condition.
- Don’t forget the opening and closing brackets, { } 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.

Common Programming Errors III

- 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 forget the opening and closing brackets, { } for the switch statement.
- Don’t forget the break statement.

Conditionals: Review I

```java
if (x == 0) {
    statement_T;
} else {
    statement_F;
}
```

Conditionals: Review II

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

Conditionals: Review III

```java
if (x >= 0) {
    if (x == 0) {
        statement_TT
    } else {
        statement_TF
    }
} else {
    statement_F
}
```
```java
switch (x) {
    case 1:
        true if x == 1 statement
        break;
    case 2:
        true if x == 2 statement
        break;
    default:
        always true
}
```