# 1.30 Selection

Making decisions based on conditions

The If-else statement

Relational operators

Logical operators

*EE108 – Computing for Engineers*

1

## Overview



Aims

- Learn to use selection (decision making) statements correctly and choose the best approach for the problem at hand

Learning outcomes – you should be able to…

- Distinguish between simple and compound statements (blocks)
- Use the if-statement with simple and compound statements, including nested and cascaded if-statements
- Use the relational (comparison/equality) operators
- Use the logical operators



2

## Selection (conditional execution)

4

Almost all interesting programmes must react to external input or changes in their own internal state in some way

- To achieve this, some sort of decision making is essential to most programmes

Many times, what we want a programme to do can be described as follows

- If some particular condition exists, do A
- Otherwise, do B instead

To achieve this in many programming languages including C…

- Use selection statements (e.g. the *if-statement*) to evaluate some test expression (which represents the condition of interest) and then decide which code to execute depending on the value of the test expression
  - Pseudocode example: if speed is too fast then slow down
- In C, the expressions used in selection statements frequently (but not always) rely on relational operators (e.g. greater than, equal to, etc.)
  - For example the pseudocode "if speed is too fast" would usually be expressed as something more like "if speed greater than MAX_SPEED", where "greater than" is a relation



4

## True and False in C

5

*Remember: an expression is a computation/calculation that yields a value*

For selection/conditional statements

- we need to evaluate an expression and determine if it can be considered **true** or **false**
- Then take action accordingly

The C language has no keywords for true or false and no special boolean type – instead, C just uses numbers

So which numbers can be considered true or false?

- An expression which evaluates to **zero is considered false**
- An expression which evaluates to **non-zero is considered true** (whether the value is positive, negative, or unsigned)

  *Although we will mostly consider expressions that use relational operators, any expression can be used – It is the zero or non-zero result of evaluating the expression that matters (more on this later…)*



5

# True and False in C contd.



It is frequently convenient to define constants to represent true and false values

- E.g. it can be useful to specify an initial value for boolean-like variables or constants that are naturally just one of true or false

```
isLedBlinking = false;
```

Arduino defines constants for true and false for you as follows:

```
#define true  1
#define false 0
```

*Never use the literal values 1 or 0 when you mean true or false – use the **true** or **false** constants instead*





# Relational operators



| Equality | Other relations |
|---|---|
| == *equal to* | < *less than* |
| != *not equal to* | > *Greater than* |
| | <= *Less than or equal to* |
| | >= *Greater than or equal to* |

The relational operators have lower precedence than arithmetic operators.

Relational operators are evaluated from left to right (in the absence of parentheses)

```
int i = 1, j = -1, k = 0; // multiple variables declared on one line to save space

The value of      i < j             is 0 (false)
The value of      i >= j            is 1 (true)
The value of      i == 0            is 0 (false)
The value of      j != 0            is 1 (true)
The value of      j                 is -1 (true), -1 is non-zero
The value of      k                 is 0 (false)
The value of      j + i < k + 1     is 1 (true), (j + i) < (k + 1) is 0 < 1

// although this looks like a test for k between j and i it doesn't do that.
// we'll see the correct idiom for testing k between j and i later
The value of      j < k < i         is 0 (false), (j < k) is 1, (1 < i) is 0
```

## The if-statement



The basic selection statement in C is the if-statement with the following syntax

**if** ( *expression* ) *statementOrBlockTrue*

statementOrBlockTrue is executed if the expression evaluates true

An if statement may optionally have an else clause which indicates the actions to take if the condition evaluates false. The more complete syntax is therefore:

**if** ( *expression* ) *statementOrBlockTrue* **else** *statementOrBlockFalse*

Examples (note the newlines, indentation, and semicolons)…

```
if (speed < MAX_SPEED)
  speed += accelerationStepSize;
```

The statement to execute is always on a new line and indented relative to if/else

```
if (a > b)
  max = a;
else
  max = b;
```





## Compound statements / blocks



Strictly, there can only be one statement to be executed if the condition evaluated true and, if there is an else-clause, one statement to be executed if the condition evaluated false

Frequently we need to execute multiple statements in a certain condition and in this case use a block instead

*Remember: A compound statement or statement block is a group of statements surrounded by braces and can be used anywhere that a single statement is legal*

Examples (note the braces, newlines, indentation and semicolons)…

```
if (button == HIGH) {
  digitalWrite(LED1, HIGH);
  digitalWrite(LED2, HIGH);
}
```

Note block surrounded by braces with closing brace at same indent as the if/else. Within the block the statements are indented relative to if/else.

```
if (temperature > prevTemp) { // increasing?
  digitalWrite(LED1, HIGH);
  prevTemp = temperature;
} else {
  digitalWrite(LED1, LOW); // red LED off
  digitalWrite(LED2, LOW); // green LED on
}
```

*Also see example: HeadsOrTails*

# Nested if-statements



A compound statement can contain any statements

- Therefore, it can contain other compound statements
- Therefore, it can also contain if-statements

This gives us the possibility to nest if-statements which allows us to express things like the following

```
if (temperature > prevTemp) { // is temperature increasing?
  …
  if (durationTempIncreasing > THRESHOLD_MS)
    digitalWrite(LED1, HIGH);   // red warning indicator on
  else
    digitalWrite(LED1, LOW);    // red LED off
} else {                       // temperature not increasing,
  …                            // i.e. steady or decreasing
  if (durationTempDecreasing > THRESHOLD_MS)
    …
  else
    …
}
```

Ellipsis means Details left out

*Also see example: HeadsOrTailsTotal*





# Cascaded if-statements



Very often you will need to test a series of conditions, stopping (and taking action) as soon as the first of them is true – a decision sequence

This could be written in normal nested form, but that doesn't communicate the sequential nature clearly. Instead you normally write it in cascaded form which is more readable – the compiler sees both identically

```
// nested form
// DON'T USE THIS
// in this situation
if ( expression1 ) {
  statementOrBlock1
} else {
  if ( expression2 ) {
    statementOrBlock2
  } else {
    if ( expression3 ) {
      statementOrBlock3
    } else {
      statementsOrBlock4
    }
  }
}
```

```
// nested form with
// fewer braces
// DON'T USE THIS
if ( expression1 ) {
  statementOrBlock1
} else
  if ( expression2 ) {
    statementOrBlock2
  } else
    if ( expression3 ) {
      statementOrBlock3
    } else {
      statementsOrBlock4
    }
```

```
// Correct style
// (rewritten in
// cascade form)

if ( expression1 ) {
    statementOrBlock1
} else if ( expression2 ) {
    statementOrBlock2
} else if ( expression3 ) {
    statementOrBlock3
} else {
    statementOrBlock4
}
```

Note the indents. All else-if clauses in the cascade line up at the same indent level.

The braces are not strictly required for simple statements, but help to avoid confusion

## Cascaded if-statement example



```
// cascade example with single statements and blocks
// Note where the curly braces are written

if (temp < MIN) {
  digitalWrite(LED1, LOW);
  digitalWrite(LED2, LOW);
} else if (temp < LOW) {    // with a single statement braces
  digitalWrite(LED2, HIGH); // are not strictly necessary, but
} else if (temp <= HIGH)    // help prevent mistakes in cascaded-if
  digitalWrite(LED1, HIGH);
} else { // temp > high !
  digitalWrite(LED1, HIGH);
  digitalWrite(LED2, HIGH);
}
```

***Recommendation: Use braces to make every clause in a cascaded if-statement a block, even if the block only contains a single statement.***

It can be easy to make mistakes writing a cascaded if-statement. Using braces everywhere avoids confusion about where the cascaded if-statement ends

*Also see example: DiceUsingCascadedIf*





## *Self test questions*



*Q. Use cascaded if statements to set the **alertLevel** to the appropriate constant based on the value of **measurement** as follows: no alert [0-40), yellow alert [40-60), orange alert [60-80), red alert [80-100] where '[' or ']' means include the limit value and '(' or ')' means exclude the limit value.*

```
#define ALERT_NONE    0
#define ALERT_YELLOW  1
#define ALERT_ORANGE  2
#define ALERT_RED     3

unsigned int alertLevel;
unsigned int measurement;
…
// Assume that some code (not shown) first sets the measurement value
// TODO - your must code to set alertLevel based on the measurement…
```

# The logical operators



| Logical operators | |
|---|---|
| ! | *Logical NOT* |
| && | *logical AND (short circuit)* |
| \|\| | *Logical OR (short circuit)* |

The logical operators are used to build complex expressions by combining simpler ones.

| | |
|---|---|
| !*expr* | evaluates true (1) if expr is false (0), and otherwise evaluates false (0) |
| *expr1* && *expr2* | evaluates true (1) if expr1 AND expr2 are both true (non-zero), otherwise eval's to false (0) |
| *expr1* \|\| *expr2* | is true (1) if either expr1 OR expr2 (or both) is true (non-zero), otherwise eval's to false (0) |

```
// logical AND expression
// if expr1 AND expr2 are both true

if (expr1 && expr2)
  doSomething();

// is equivalent to this
// this nested if statement…

if (expr1) {
  if (expr2) {
    doSomething();
  }
}
```

```
// logical OR expression
// if either expr1 OR expr2 is true

if (expr1 || expr2)
  doSomething();

// is equivalent to this cascaded
// if-statement with doSomething as the
// action in each case

if (expr1)
  doSomething();
else if (expr2)
  doSomething();
```



14

# The logical operators contd.



**Short circuit evaluation** is used for both the logical AND and logical OR. Short circuit evaluation means that expr2 is only evaluated if needed:

| | |
|---|---|
| *expr1* && *expr2* | if expr1 evaluates false then entire logical expression must be false – expr2 cannot affect the outcome and is not evaluated |
| *expr1* \|\| *expr2* | if expr1 evaluates true then the entire logical expression must be true – expr2 cannot change the outcome and is not evaluated |

***Beware:*** *short circuit evaluation can cause strange bugs if expr2 has side effects (e.g. if it uses the ++ or -- operators)*

Note: The precedence of the logical operators is lower than the arithmetic and relational operators, but it parentheses are recommended for clarity.

***Though not always necessary, it improves readability to always surround the expressions before and after a logical operator with parentheses.***



15

## More logical operator examples



```
// Note use of parentheses around expressions before and after the logical
// operator...

if ((speed < MAX_SPEED) && (accel < MAX_ACCEL)) {
  accel += ACCEL_STEP;
  speed += accel;
}


if ((temp < MIN_TEMP) || (temp > MAX_TEMP))
  digitalWrite(LED1, HIGH); // warning LED on
```

*Also see example: LogicalOperatorsButtonPress*





## True, False, boolean types and values



Standard C has no built-in boolean type and instead uses integers to represent boolean values (where zero is considered false, any non-zero value is considered true)

Arduino C defines a boolean type, called **boolean**, that can be used for variables that will only take a true or false value

*When not using Arduino C we may have to define the boolean type and the true, false constants ourselves, but we don't need to worry about that when using Arduino C.*

# True and False with logical expressions



**Logical expressions** are any expressions that use the *relational operators* or *logical operators*

| Relational operators | | | Logical operators | |
|---|---|---|---|---|
| < | *less than* | == *equality* | ! | *Logical NOT* |
| > | *Greater than* | != *inequality* | && | *logical AND (short circuit)* |
| <= | *Less than or equal to* | | \|\| | *Logical OR (short circuit)* |
| >= | *Greater than or equal to* | | | |

Examples…

```
a < 14
b == 27
// following is true if either (cond1 AND cond2) is true OR cond3 is true
((temp == 0) && !valueIsIncreasing) || (temp < 0)
```

- Logical expressions may be evaluated just like arithmetic expressions. Specifically…
  - Any logical expression which is **true evaluates to 1**
  - Any logical expression which is **false evaluates to 0**





# Contd.



Because the boolean values are just integers in C and the result of relational or logical expressions are just boolean values, the following are all possible…

```
// following are all equivalent…

// test if expr evaluates true
if (digitalRead(SW1) == HIGH)
  doSomething();

// save result of expression in variable. Later
// if variable has a true (non-zero) value
boolean sw1Active = (digitalRead(SW1) == HIGH);
if (sw1Active)
  doSomething();

// test if digitalRead returns
// a true (non-zero) value
if (digitalRead(SW1))
  doSomething();

// save result and test result later
boolean sw1Active2 = digitalRead(SW1);
if (sw1Active2)
  doSomething();
```

```
// following are all equivalent…

// test if expr evaluates true
if (digitalRead(SW1) == LOW)
  doSomething();

// test if digitalRead returns a false
// (zero) value [since not false is true]
if (!digitalRead(SW1))
  doSomething();

// save result and test result later
boolean switchClosed = !digitalRead(SW1);
if (isSwitchClosed)
  doSomething();
```

*Also see example: LogicalOperators2*

## Self test questions



*Q1. Set **buttonClosed** to be true if PIN2 is LOW. Use digitalRead and if-statements to solve the problem.*

```
boolean buttonClosed; // true if button is currently closed, false otherwise
…
```

*Q2. Solve problem of Q2 without using any if-statements*



## Self test questions



*Q3. Without using any if statements, set **errorPresent** to true if either PIN6 is LOW or PIN4 is HIGH*

```
boolean errorPresent; // true if error is present, false otherwise
…
```

## *Self test questions*



*Q1. Without using logical AND/OR implement the following: if button is closed (i.e.* ***buttonClosed*** *is true) for longer than 10 "ticks" or the button is closed when* ***errorPresent*** *is true then call* ***doSomething****().*

```
boolean buttonClosed; // true if button is currently closed, false otherwise
unsigned int nTicks; // number of ticks that button is held down
boolean errorPresent; // true when an error is present, false otherwise
…
```

*Q2. Use the logical operators to solve the same problem as Q1 more succinctly*





## Advanced selection statements

# The ternary conditional operator



Instead of writing an if statement it is possible to use an expression with the ternary (3 operand) conditional operator

*testExpr ? exprToUseWhenTestExprTrue : exprToUseWhenTestExprFalse*

The only real advantage is brevity and even that becomes a disadvantage if the code becomes less clear

Mostly you'll just need to recognise it but won't use it

```
// ternary operator examples…
int a = 1, b = 2;
int v;

// set v to the max of a and b
v = a > b ? a : b;    // v is 2

// set v to a if a is positive,
// or –a otherwise
v = (a >= 0) ? a : -a;
```

```
// equivalent using if-statements…
// set v to the max of a and b
if (a > b)
  v = a;
else
  v = b;

// set v to a if a is positive,
// or –a otherwise
if (a >= 0)
  v = a ;
else
  v = -a;
```



24

# *Self test questions*



*Q1. Use a simple if-statement to set a variable called "direction" to 1 if level is increasing and -1 otherwise. (Hint: compare levelNow with levelPrev to see if the level is increasing or not)*

```
unsigned int levelPrev; // level at previous time step
unsigned int levelNow;  // level at current time step
…
```

*Q2. Use the ternary conditional operator to solve the same problem as Q1*



25

## The switch statement



The switch statement is most commonly an alternative to using cascaded if statements when all of the if statements check the same (integral type) variable for equality with different constant values

```
// cascaded if statements …
if (speed == 0) {
  blinkDelay = 500;
} else if (speed == 1) {
  blinkDelay = 250;
} else if (speed == 2) {
  blinkDelay = 125;
} else {
  blinkDelay = 0; // don't blink
}
```

```
// equivalent switch statement
// (note: recommended indentation)
switch (speed) {
case 0:
  blinkDelay = 500;
  break;

case 1:
  blinkDelay = 250;
  break;

case 2:
  blinkDelay = 125;
  break;

default:
  blinkDelay = 0; // don't blink
}
```





## Switch syntax



The switch statement has the following syntax

```
switch ( testExpr ) {
case firstConstantExpr : oneOrMoreStatementsFirstCase
…
case lastConstantExpr : oneOrMoreStatementsLastCase
default: oneOrMoreStatementsDefaultCase
}
```

switch (testExpr)

- The expression testExpr is evaluated once and must result in an integral value (e.g. int, char, etc.)

case label

- The keyword case must be followed by an **integral constant expression**. An integral constant expression can only contain integral constants or literals and can thus be replaced by a constant integral value at compile time, e.g. MY_CONST, or MY_CONST+10.
- Typically the constant expressions are either previously defined constant values (preferred) or literal values
- If the value of testExpr is exactly equal to the value of a particular case's constantExpr then execution **starts** with the statements which follow it. Where it finishes depends on break/fall-through described next

# Switch syntax continued



```
switch ( testExpr ) {
case firstConstantExpr : oneOrMoreStatementsFirstCase
…
case lastConstantExpr : oneOrMoreStatementsLastCase
default: oneOrMoreStatementsDefaultCase
}
```

Default label

- The default case label is used when the value of testExpr does not match any other cases. The default case is optional but recommended.
- For clarity, the default case is often written as the last label in the switch block as shown above.
  **For clarity, you should mostly write the default case as the last label in your switch block for EE108R**
  - The C language does not actually require the default case to be last and some people place default as the first label instead. Recognise this when you read it, but don't do it yourself.
  - C syntax permits other positions also but these are much more likely to be missed when reading code quickly, so again, recognise it but do not do it yourself. There's one exception to this however related to fall through which we'll look at shortly





# Switch: statements following a case label



Often a case label is followed by multiple statements. These statements usually do not require surrounding braces (unlike when multiple statements follow an if statement)

The exception to this is if you need a temporary variable for a calculation specific to the case and you don't want to declare it at the top of the function. In this situation, the braces create a block scope and the variable only exists in this scope.

```
// switch statement with 2 different
// types of fall through
switch (result) {
case MODE_A: // e.g. MODE_A is a previously
             // defined constant
   doActionA();
   break;

case MODE_B: {
   int tmp = someComplexCalculationHere…

   doActionB(tmp);
   break;
}

default:
   doDefaultAction();
}
```

## Switch – break statement



The **break** statement

- A case label indicates where to start execution, but not where to stop
- Without a break statement, C will execute all the remaining statements in the switch block in sequence as if there were no case labels present
  - Usually this is not what you intend (but see also fall-through on the next slide)
- Generally, you should add a break statement to exit the switch block after executing all the statements that belong to any particular case label as shown in the example

*Q: Why was no break statement needed after the default label in the example?*

```
// switch statement with 2 different
// types of fall through
switch (result) {
case MODE_A:
   doActionA();
   break;

case MODE_B:
   doActionB();
   break;

default:
   doDefaultAction();
}
```





## Switch – case label fall-through



**Fall through** refers to the continued execution of statements across case labels when the break statement is not used. There are two main types (but hybrids are also possible)…

Fall through Type 1: multiple case labels together, followed by a list of statements means that we want to execute that list of statements if <u>any</u> of the cases matches.

```
// switch statement with
// type 1 fall through
switch (result) {
case 0:
case 1:
case 2:
   doActionA1();
   doActionA2();
   break;

…

default:
   doDefaultAction();
}
```

Equivalent to

```
if ((result == 0)
    || (result == 1)
    || (result == 2))
{
  doActionA1();
  doActionA2();
}
```

## Contd.



Fall through Type 2: A single case followed by statements without any break statement means that there is some work specific to this case and then we fall through to execute some work that is common to both this case and the following case.

This can be difficult to use correctly, so to prove to readers that it is not a mistake – always add a comment to say "**FALL THROUGH**" or "**no break**" if it is deliberate.

```
// switch statement with 2 different
// types of fall through
switch (result) {
…
case 3:
   doPreliminary1();
   doPreliminary2();
   // FALL THROUGH

case 4:
   doActionB();
   break;

…
}
```

Equivalent to

```
…
if (result == 3) {
    doPreliminary1(); // case 3 only
    doPreliminary2(); // case 3 only
    doActionB();  // common with case 4
} else if (result == 4) {
    doActionB();
}
```





## Special case, default label fall through



The default case is used to match any input that doesn't match any other case. Often this means the input was bad in some way or perhaps should be ignored and the handling of this input is often different from any other case.

However, sometimes it can be useful to say that any input that doesn't match any of the existing cases, should be handled the same way as one of the existing cases. To avoid code repetition, we can use a default label with fall through to the case it should share behaviour with

```
// switch statement with default label fall through
switch (result) {
case 1:
   doSomething();
   break;

default: // same as case 2
case 2:
   handleCase2();
   break;

case 3:
   handleCase3(); // Q. why no break here???
}
```

## Switch statement advantages



Often clearer than cascaded if statements

- it may appear longer due to break statements and blank lines

Often faster to execute than cascaded if statements

- The compiler can optimise the way cases are matched, particularly if the case constants correspond to consecutive numbers

*Also see example: dice_using_switch.c*





## *Self test questions*



*Q1. Use a switch statement to test the value of the variable "state" and set state to a new value according to the following transitions (where the values are constants): READY→SET; SET→PLAY; PLAY→FINISHED; Any other value→READY*
*(This kind of code is commonly use to implement Finite State Machines (FSMs)).*

```
// The solution using if statements would start as follows
if (state == SET) {
    state = PLAY;
} else if …

// write the solution using the switch-statement
```