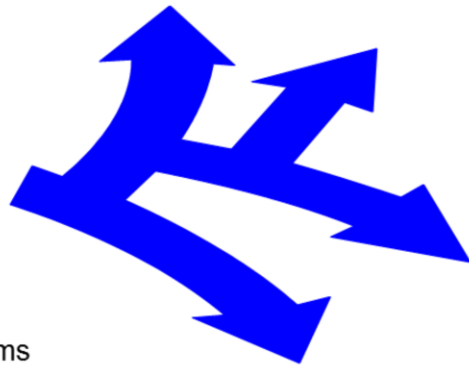


Making Decisions

- **Objective**
 - To implement decisions
- **Contents**
 - The boolean type
 - Equality operators
 - Relational operators
 - *if* statements
 - Compound statements
 - Logical operators
 - *switch* statements
 - The conditional operator
- **Summary**
- **Practical**
 - Traditional decision-making problems



This is the second of the two chapters on flow of control. The objective of this chapter is to cover all of C's support for decision making. This includes the two constructs `if` and `switch`, the conditional operator and statements that affect and override default flow.

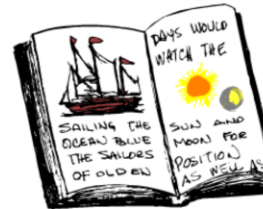

The chapter includes layout suggestions for these constructs.

The Story So Far...

- **Sequential flow of control**
 - First statement is executed, then each statement is executed in turn until all statements have been executed once
- **Useful programs need to make decisions**
 - Some statements need to be selectively executed based on the values of variables

```
int main(void)
{
    ...
    ...
    return 0;
}
```

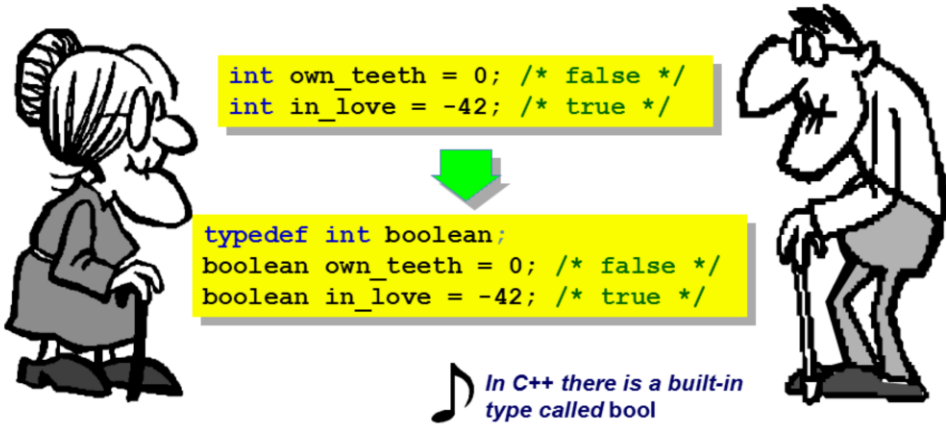
Sequential



So far, we have concentrated on the syntax of data declaration and expression evaluation, and have confined the flow of control to be the default, i.e. sequential. Other than library function calls, this has been the case. Clearly, such programs are not very flexible, robust or useful.

The Boolean Type

- **There is no built-in boolean type in C**
- **However, integers may be used as booleans**
 - An integer with the value zero represents false
 - An integer with *any* non-zero value represents true



This is yet another example of an implicit conversion. The compiler is allowed to perform an implicit conversion from an integer with value zero to a conceptual boolean representing false; and from an integer with any non-zero value to a conceptual boolean representing true. The conversion is 'conceptual' as there is no built-in boolean type. The conversion is really just a rule for interpreting integer values when a boolean expression is required.

The reason that zero represents false may seem arbitrary but will make more sense when pointers have been covered. Briefly, zero can also represent a pointer that does not point to anything. Hence, the pointer zero can also be interpreted as the 'false', viz. the false pointer.

A boolean typedef helps to distinguish which integers are being used as genuine integers and which integers are being used as boolean substitutes. It is tempting to set up constants to represent false and true to help make an even clearer distinction:

```
const boolean false = 0;
const boolean true = 1;
boolean own_teeth = false;
boolean in_love = true;
```

A severe problem with this approach is that it suggests the integer one is the sole value that can represent true. This is extremely dangerous since any non-zero value can represent true. The idiom in C is that the test for truth is performed by testing if the boolean variable is not false. This always works because the only integer value that represents false is zero.

Note: the new C standard (C9X) contains a header file `<stdbool.h>` that contains a typedef for `bool` and macros for `true` and `false`. If your compiler supports this header file you should use it instead of creating your own typedef.

Equality and Relational Operators

- **Binary operators**

- == equal to
- != not equal

- > greater than
- < less than
- >= greater than or equal
- <= less than or equal

- **All yield**

- integer 0 for false
- integer 1 for true

```
typedef int boolean;

int x, y;
boolean same, less;

scanf("%d %d", &x, &y);
same = (x == y);
less = (x < y);

printf("%d\n", same);
printf("%d\n", less);
```

The equality and inequality operators along with the four traditional relational operators all exist in C. Care must be taken with the equality and non-equality symbols, since neither != nor == are common in other languages. Operands must all be scalar and are usually the same type. If not, different types will result in the automatic conversions that take place.

The boolean value generated by these operators are either the integer 1 (true) or the integer 0 (false).

QACPROG

The *if* Statement

The simplest selection statement

else clause is optional

simple statement

```
if (expression)
    statement
else
    statement
```

executed if result of expression is non-zero (true)

executed if result of expression is zero (false)

```
scanf("%d", &x);
if (x >= 0)
    puts("non-negative");
else
    puts("negative");
```

If you can keep your head when all about you...

The primary structure for decision making is the if statement. A control expression is used to determine which branch of the two-way fork statement is taken. If the expression is true the first branch (the if body) is taken but if the expression is false then the second branch (the else body) is taken. The else clause is optional and if it is omitted then no code is executed should the control expression evaluate to false.

The if body and else body are statements, which means either a single ordinary expression statement:

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

Or a compound statement, to group together multiple actions:

```
if (x < 0)
{
    fprintf(stderr, "resetting to zero\n");
    x = 0;
}
```

The if statement control expression expects an integral expression. The outcome of this integral expression is interpreted as a boolean: a value of zero is false and any other value (positive or negative) is true.

The Compound Statement

- **A simple statement**
 - Is a declaration;
 - Or an expression;
- **A compound statement**
 - A sequence of statements
 - Enclosed in a { block }
 - These statements may be simple statements or compound statements...

declaration;

expression;

```
{
    statement
    statement
    ...
}
```



```
{
    int x;
    int y;

    scanf("%d %d", &x, &y);
    printf("%d,%d\n", x, y);
}
```

declaration statements first

then expression statements

In C a simple statement is any semi-colon terminated expression such as:

```
x = y + 1;
x++;
printf("%d\n", x);
```

Declarations are ordinary statements which is unlike most other languages such as C or Pascal which require the declarations to be specified before any executable code (however, code and declarations can be freely mixed in the new C9X standard). Variables declared in blocks are in scope from the point of their declaration until the next closing brace. The layout is free format, but it is recommended that each simple statement goes on its own line.

A statement can also be a compound statement, a sequence of statements delimited by an opening and closing brace — there is no terminating semi-colon after the closing brace.

```
{
    scanf("%d", &x);
    printf("%d\n", x);
}
```

The braces are equivalent to the *begin* and *end* keywords found in other Algol-like languages such as Pascal or PL/1. C does not use name matched delimiters like languages such as Visual Basic, Ada or Fortran where the *if* keyword is matched with an appropriate end keyword such as *endif*.

Note: the definition of a statement and a compound statement is recursive (a compound statement contains statements which may themselves be compound statements which contain statements which may themselves be compound statements...). A compound statement may also be empty.

Exercise: Spot the Bugs

①

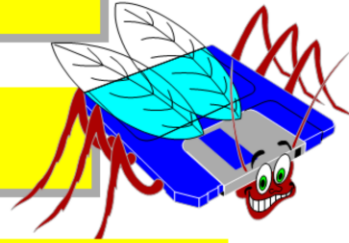
```
scanf("%d %d", &x, &y);  
if (x >= 0)  
    if (y < x)  
        puts("y is less than x");  
else  
    puts("x is negative");
```

②

```
scanf("%d", &x);  
if (x = 0)  
    puts("x is zero");
```

③

```
minutes++;  
if (minutes == 60)  
    minutes = 0;  
    hours++;
```



The first problem illustrated is known as the "dangling else" problem. The code is clearly laid out to show the programmer's intention, but when the else and ifs are matched up there is a problem. An else will match up with the closest unmatched if above it – in this case we can see that the indentation does not reflect the control flow. This can be solved by using a compound statement for the first if body. In fact, this is one of the reasons that many coding guidelines recommend that all if and else bodies should be compound statements, even when they only contain a single statement.

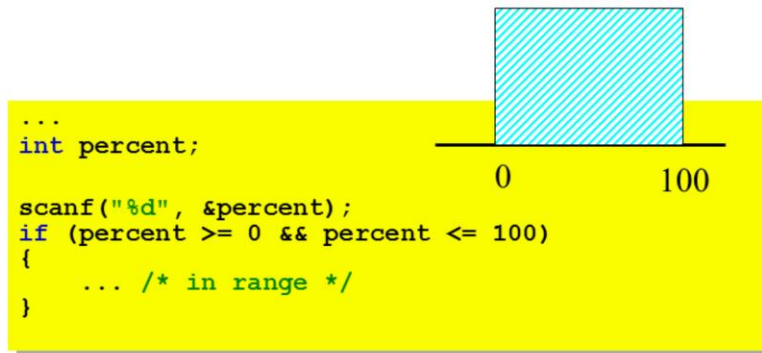
Everyone's favourite! The programmer intended that x would be compared for equality with 0, which is tested using ==. Instead they have assigned 0 to x, and then compared the result for truth – as this is 0 which is considered false, the if body will never be executed and x will always be 0!

The problem in the last fragment is that although it looks (via the formatting) like there are two statements controlled by the if statement, there is in fact only one. The compiler ignores whitespace formatting. A compound statement is required.

There are a number of checking tools available that can spot such mistakes.

Logical *and*

- **The logical AND operator**
 - Is called `&&`, is binary, yields 0 for false or 1 for true
- **short-circuit evaluation**
 - Expressions using `&&` are evaluated from left to right until truth or falsehood is established



The logical *and* operator is written as `&&`. It evaluates the two operands from left to right, and will not evaluate further than it needs to. If the left hand operand of `&&` is *false* (zero), there is no need to evaluate the right hand side as the result of the whole expression must be *false*. In the example:

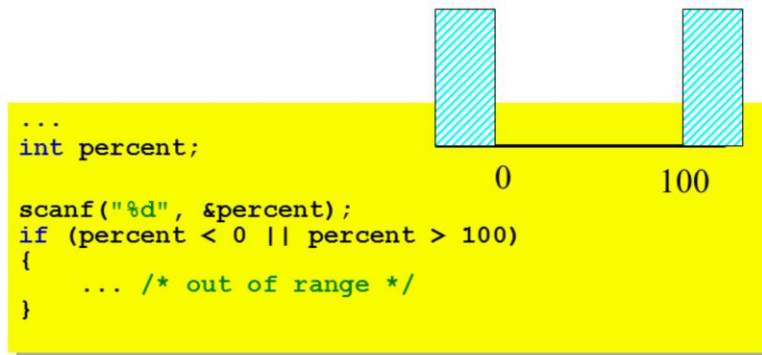
```
in_range = percent >= 0 && percent <= 100;
```

- if `percent` is less than zero (eg -42), then `(percent >= 0)` is *false*. Trivially... (*false && anything*) is *false* so the right-hand-side expression does not need to be evaluated to determine the outcome of the `&&` expression, so it isn't, and zero (representing *false*) is assigned to `in_range`.
- if `percent` is greater than zero (eg +42), then `(percent >= 0)` is *true*. The right-hand-side operand is then evaluated: `(percent <= 100)` is also *true*. Since (*true && true*) is *true*, one (representing *true*) is assigned to `in_range`.
- if `percent` is greater than 100 (eg +142), then `(percent >= 0)` is *true*. The right-hand-side operand is then evaluated: `(percent <= 100)` is *false*. Since (*true && false*) is *false*, zero (representing *false*) is assigned to `in_range`.

The `&&` operator should not be confused with `&` which has a quite different meaning (`&` is a bit manipulation operator).

Logical or

- **The logical OR operator**
 - Is called `||`, is binary, yields 0 for false or 1 for true
- **short-circuit evaluation**
 - Expressions using `||` are evaluated from left to right until truth or falsehood is established



The logical *or* operator is written as `||`. It evaluates the two operands from left to right, and will not evaluate further than it needs to. If the left hand operand of `||` is *true* there is no need to evaluate any further as the whole result will be *true*. In the example:


```
out_of_range = percent < 0 || percent > 100;
```

- if `percent` is less than zero (eg -42), then `(percent < 0)` is *true*. Trivially... (*true* && *anything*) is *true* so the right-hand-side expression does not need to be evaluated to determine the outcome of the `||` expression, so it isn't, and one (representing *true*) is assigned to `out_of_range`.
- if `percent` is greater than zero (eg +42), then `(percent < 0)` is *false*. The right-hand-side operand is then evaluated: `(percent > 100)` is also *false*. Since (*false* `||` *false*) is *false*, zero (representing *false*) is assigned to `out_of_range`.
- if `percent` is greater than 100 (eg +142), then `(percent < 0)` is *false*. The right-hand-side operand is then evaluated: `(percent > 100)` is *true*. Since (*false* `||` *true*) is *true*, one (representing *true*) is assigned to `out_of_range`.

The `||` operator should not be confused with `|` which has a quite different meaning (`|` is a bit manipulation operator).

Logical not

- **When C is testing for true or false...**
 - Zero is false, non-zero is true
- **The logical NOT operator**
 - Is called `!`, is unary, yields 0 or 1



```
typedef int boolean;  
  
int percent;  
boolean out_of_range, in_range;  
  
scanf("%d", &percent);  
out_of_range = percent < 0 || percent > 100;  
in_range = !out_of_range;
```


Logical *not*, logical *and*, and logical *or* operations can be performed in C. Their operands must be integral (`ints` or `chars`). Whereas the relational operators are guaranteed to yield 0 and 1, logical operators are capable of interpreting *all* integral values as booleans. The rule is simple; 0 is taken as false and everything else is taken as true.

The logical operators still yield 0 or 1.


Multiway Selection

Achieved using multiple *if* statements

```
if (x < 0)
    puts("negative");
else
{
    if (x == 0)
        puts("zero");
    else
    {
        if (x == 1)
            puts("one");
        else
            puts("many");
    }
}
```



```
if (x < 0)
    puts("negative");
else if (x == 0)
    puts("zero");
else if (x == 1)
    puts("one");
else
    puts("many");
```



Languages which use matched delimiters such as Visual Basic, Ada and Fortran also support a multiway if statement where additional branches are provided using `elseif` constructs which are similar to the `if` component but occur between the `if` and `else` statements. Any number of these `elseif` branches are permitted. The tests are evaluated from top to bottom and as soon as an `if` or `elseif` test evaluates to true that branch is taken.


C does not provide an `elseif`, `elsif` or `elif` statement because the same effect can be achieved using multiple `if` statements as shown on the slide. Note the `else` and `if` are separate keywords so that the `else` clause of the first `if` statement is itself another `if` statement.

The *switch* Statement

- **Can be used for some multiway selections**
 - The expression must evaluate to an integral value typically and *int* or *enum*
 - The expression is evaluated and the matching case label is chosen as the next point of execution

```
switch (expression)
{
  case constant:
    statements
  ...
  default:
    statements
}
```

```
switch (legs)
{
  case 1:
    puts("Flamingo");
    break;
  case 2:
    puts("Ape");
    break;
  case 4:
    puts("Dog");
    break;
  default:
    puts("Insect");
    break;
}
```



If your test is based on the evaluation of a simple integral expression, the switch statement is neater and more efficient than a corresponding nested *if/else* construct. Once the syntax is in place, the flow is relatively easy to read. The rules are few but important and are detailed on the next page.

The construct starts with the *switch* keyword. It is followed by an expression in brackets and completed by a compound statement. The compound statement consists of a sequence of case labels. Each label consists of the *case* keyword, followed by a constant integral value and a *:*. The statements that follow need not be made into a compound statement, but should be concluded with a *break* statement. The *default* keyword is also followed by a *:*.

Alternatives to the above syntax are covered in the next few pages.

case Notes

Order of case and default labels is irrelevant

Not an error if no label matches the switch expression

Case values must be compile time constants

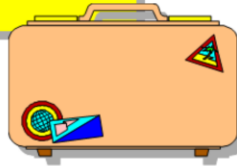
Case constants must all be different

Default clause is optional, chosen if no case label matches

```
switch (letter)
{
  case 'a': case 'e': case 'i':
  case 'o': case 'u':
    vowel = 1; /* true */
    break;
  default:
    vowel = 0; /* false */
    break;
}
```

Usual to end each case statement with a break. Why?

Common practice to include a break on last statement. Why?



switch statements are typically implemented as jump tables, implying that they must use a simple and direct index type and the jump destinations must be known at compile time. Integers, characters and enumerations fit the description of the type, and the additional constraint is that case labels must be compile time constants: literal values, const variables initialised from compile time constants, or simple expressions involving only compile time constants.

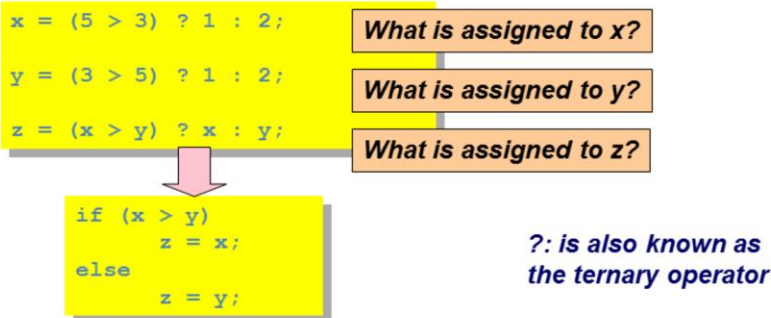
Another reason for the constraint on constants is that all the case labels must be known to be unique. Consider what might happen if this were not the case and two labels evaluated at runtime to have the same value. What should the program do?

The normal case is that a break is expected. Leaving one out can have dire consequences as the program will also execute any code in the following case. There are occasional uses of *fall through*, but they merit a comment.

```
switch (argc)
{
  case 2:
    filename = args[1];
    /* fall through */
  case 1:
    options = args[0];
    break;
  case 0:
    break;
  default:
    fputs("Invalid number of arguments", stderr);
    break;
}
```

The Conditional Operator

- **Takes three operands, each of which is an expression**
expression1 ? expression2 : expression3
- **The value of the whole expression is the value of**
expression2 if expression1 is true or
expression3 if expression1 is false



The ternary operator is an efficient alternative to a simple if/else construct. Care must be taken to ensure type matching and precedence. The value assigned to x is 1 since (5 > 3) is true. The value assigned to y is 2 since (3 > 5) is false. The last statement assigns the value 2 to z since (1 > 2) is false.

An alternative to the last statement is:

```
if (x > y)
    z = x;
else
    z = y;
```

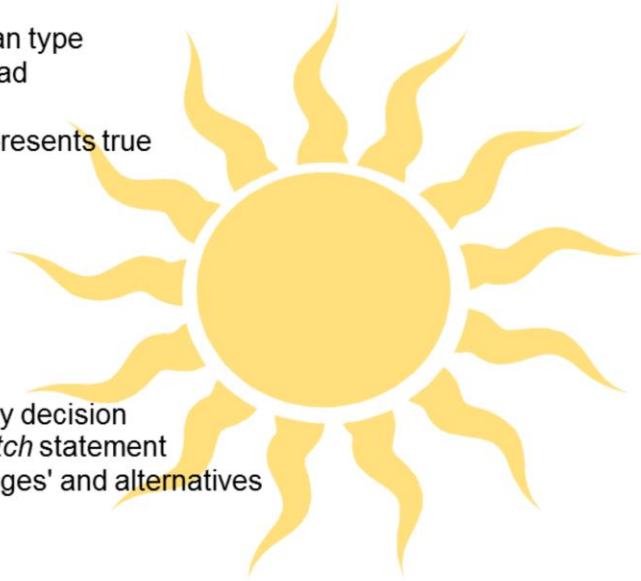
which, assuming a simple max function, could be rewritten more clearly as:

```
z = max(x, y);
```

The ternary operator was used extensively to reduce code size and increase speed; and by the preprocessor in function macros. Nowadays, modern compilers have reduced the need for such a construct (especially in function macros). However, it is useful in function return statements and initialisation expressions. Examples will be found in the *Functions* chapter.

Summary

- **Boolean type**
 - C has no built-in boolean type
 - Integers are used instead
 - Zero represents false
 - Any non-zero value represents true
- **Operators**
 - Equality `==` `!=`
 - Relational `<` `<=` `>` `>=`
 - Logical `&&` `||` `!`
 - Ternary `?:`
- **Decisions**
 - *if* and *if else*
 - *switch* used in multi-way decision
 - *break* used to exit *switch* statement
 - no *break* provides 'ranges' and alternatives



QACPROG

Common Pitfalls

- **Conditions**
 - Forgetting the parentheses on a condition
 - Not fully parenthesising a condition
 - Confusing assignment and equality
- **Statements**
 - Not seeing an accidental null statement
 - Not grouping multiple statements into a compound statement
 - Comparing floating point numbers for equality

```
if value > 0
    puts("positive");
```

✗

```
if (min <= input) && (input <= max)
    puts("in range");
```

✗

```
if (guess = answer)
    puts("You win! :-)");
```

💣

```
if (guess == answer);
    puts("You win! :-)");
```

💣

```
if (x < 0)
    puts("Value must be +ve");
    scanf("%d", &x);
```

💣

```
if (denom != 0.0)
    result = numer / denom;
```

💣

The first example above is corrected by placing the condition in parentheses:

```
if (value > 0)
    puts("positive");
```

Although parentheses are present in the second example, it is the *whole* condition that must be within parentheses. Both parts of the *and* condition must be within parentheses:

```
if ((min <= input) && (input <= max))
    puts("in range");
```

One of the most common errors is to use = as the test for equality where == should be used. This will have the surprising effect of assigning the value, and whether this assigned value is 0 or not will be used to determine whether the condition is true or false – not quite what the programmer intended!

A semi-colon on its own acts as a null statement. Therefore, in the fourth example there are two statements rather than one separating the *if* from the *else*:

The fifth example is corrected by making the body a compound statement:

```
if (x < 0)
{
    puts("Value must be +ve");
    scanf("%d", &x);
}
```

Floating point numbers suffer a certain amount of rounding and loss of precision in calculations. This is often not enough to be significant, except when attempting to compare two values for equality. It is safer to compare floating point numbers as being within a small range of each other rather than for direct equality.

case Without *break*

```
...
int verse ;
for (verse = 1; verse != 5; ++verse)
{
    printf("On the ");
    switch (verse)
    {
        case 1 : printf("1st");          break ;
        case 2 : printf("2nd");          break ;
        case 3 : printf("3rd");          break ;
        default : printf("%dth", verse); break ;
    }
    printf(" day of Christmas my true love sent to me:\n");
    switch (verse)
    {
        case 5 : printf("five gold rings, ");
        case 4 : printf("four calling birds, ");
        case 3 : printf("three french hens, ");
        case 2 : printf("two turtle doves, and ");
        case 1 : printf("a partridge in a pear tree\n");
    }
}
```



**Look, no
break!**