Control Flow

blocks

- a compound statement (aka a block) is
  - an unnamed sequence of statements & declarations
  - grouped together inside { braces }

```
;
```

the null statement

```
declaration ;
```

```
expression ;
```

semicolons required

```
{
      ... ;
      ... ;
      ... ;
}
```

trailing semicolon not required

the simplest selection statement

else part is optional

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

```
const char * day_suffix(int days)
{
    if (days / 10 == 1)
        return "th";
    else if (days % 10 == 1)
        return "st";
    else if (days % 10 == 2)
        return "nd";
    else if (days % 10 == 3)
        return "rd";
    else
        return "th";
}
```

dangling else

- in a nested if an else associates with its nearest lexical if

physical indentation != logical indentation

```
if (value >= 0)
    if (value <= 100)
        return true;
else
    return false;
```

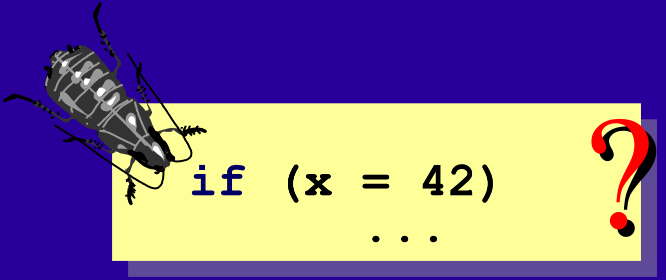
??

physical indentation == logical indentation

```
if (value >= 0)
    if (value <= 100)
        return true;
    else
        return false;
```

?

- using = instead of == is a common bug
  - compiles because assignment is an expression



```
if (x = 42)
    ...
```

oops: but not a compile-time error

```
if (x == 42)
    ...
```

- how about reversing the operands?
  - a common guideline but what do you think?

```
if (42 = x)
    ...
```

compile time error

```
if (42 == x)
    ...
```

switch statement

- stylised if (value == constant)-else chain
  - *switch* on integral types only
  - *case* labels must be compile-time constants
  - no duplicate *case* labels, no shortcut for ranges

```
const char * day_suffix(int days)
{
    const char * result = "th";

    if (days / 10 != 1)
        switch (days % 10)
        {
        case 1 :
            result = "st"; break;
        case 2 :
            result = "nd"; break;
        case 3 :
            result = "rd"; break;
        default:
            result = "th"; break;
        }
    return result;
}
```

fall through

- case labels provide "goto" style jump points
  - when inside the switch they play no part
  - there is no implicit break
  - this is known as fall-through

```
void send(short * to, short * from, int count)
{
    int n = (count + 7) / 8;
    switch (count % 8)
    {
    case 0 : do {  *to++ = *from++;
    case 7 :       *to++ = *from++;
    case 6 :       *to++ = *from++;
    case 5 :       *to++ = *from++;
    case 4 :       *to++ = *from++;
    case 3 :       *to++ = *from++;
    case 2 :       *to++ = *from++;
    case 1 :       *to++ = *from++;
                } while (--n > 0);
    }
}
```

switch gotchas

will this assignment happen?

```
int at;
...
switch (days % 10)
{
    at = 0;

case 1 :
    while (at != 0) ... break;
case 2 :
    while (at != 0) ... break;
case 3 :
    while (at != 0) ...;
defualt:
    error(); break;
}
```

how do you spell default?

fall through is usually an error

why write this break?

- the simplest iteration statement

```
initialisation
while ( continuation-condition )
{
    loop-task
    update-loop
}
```

no ; needed here

```
int digit = 0;
while (digit != 10) {
    printf("%d ", digit);
    digit++;
}
```

0 1 2 3 4 5 6 7 8 9

- stylized iteration – “scaffolding” all together
  - you can omit any part of the *for* statement
  - declared variables are scoped to *for* statement

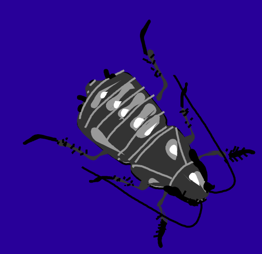
```
for (initialisation; continuation-condition; update-loop )
{
    loop-task
}
```

equivalent while statement

```
{
    initialisation
    while ( continuation-condition )
    {
        loop-task
        update-loop
    }
}
```

0 1 2 3 4 5 6 7 8 9

```
for (int digit = 0; digit != 10; digit++) {
    printf("%d ", digit);
}
```



do statement

- continuation-condition is tested at end
  - means loop executes at least once
  - much rarer than for or while statement

```
initialisation
do
{
     loop-task
     update-loop
}
while ( continuation-condition );
```

; needed here

```
int digit = 0;
do {
     printf("%d ", digit);
     digit++;
}
while (digit != 10);
```

0 1 2 3 4 5 6 7 8 9

return statement

- the *return* statement returns a value!
  - the expression must match the return type
  - either exactly or via implicit conversion
  - to end a *void* function early use *return*;

```
return expression_opt ;
```

```
const char * day_suffix(int days)
{
    const char * result = "th";

    if (days / 10 != 1)
        switch (days % 10)
        {
            ...
        }
    return result;
}
```

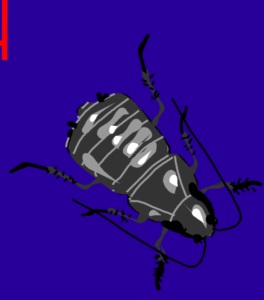
continue statement

- starts a new iteration of nearest enclosing while/do/for
  - highly correlated with bugs

```
for (int at = 0; at != 10; at++) {
    ...
    continue;
    ...
}
```

```
do {
    ...
    continue;
    ...
}
while (...);
```

```
while (...) {
    ...
    continue;
    ...
}
```



break statement

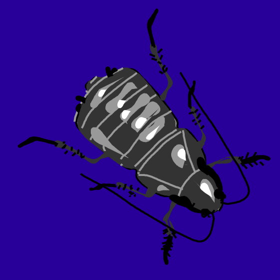
- exits nearest enclosing while/do/for/switch
  - less highly correlated with bugs in loops
  - normal in a switch statement

```
for (...) {
    ...
    break;
    ...
}
```

```
switch (...) {
    ...
    break;
    ...
}
```

```
do {
    ...
    break;
    ...
}
while (...);
```

```
while (...) {
    ...
    break;
    ...
}
```



goto statement

- jumps to a named label
  - can jump forward or backward
  - can bypass variable initialization (!)

```
identifier : statement
```

```
void some_function(void)
{
    FILE * f1 = fopen("data1.txt", "r");
    FILE * f2 = fopen("data2.txt", "r");
    ...

    if (error)
        goto cleanup;

    ...

cleanup:
    if (f1) fclose(f1);
    if (f2) fclose(f2);
}
```

assert

- **assert macro can be found in <assert.h>**
  - **takes a condition as its argument**
  - **aborts program if condition is false**
  - **compiled out if NDEBUG is defined**
  - **useful for expressing function-call contracts or for data structure invariants**

```
#include <assert.h>

void example(int year, int month, int day)
{
    assert(month > 0);
    assert(month <= 12);
    assert(is_valid_day(year, month, day));
    ...
}
```

aborts with a diagnostic stating the condition, the file, the line number and (optionally) the function that failed

- assert can also be used for writing simple automatic tests
  - this usage places the assertions outside the production code rather than within it

```
#include "rational.h"
#include <assert.h>

void identical_rationals_compare_equal(void)
{
    rational lhs = { 42, 6 }, rhs = { 42, 6 };
    assert(equal_rationals(lhs, rhs));
}

void unnormalised_integral_values_rationalise(void)
{
    rational original = { 42, 6 };
    rational normalised = normalise_rational(original);
    rational expected = { 7, 1 };
    assert(equal_rationals(normalised, expected));
}
```

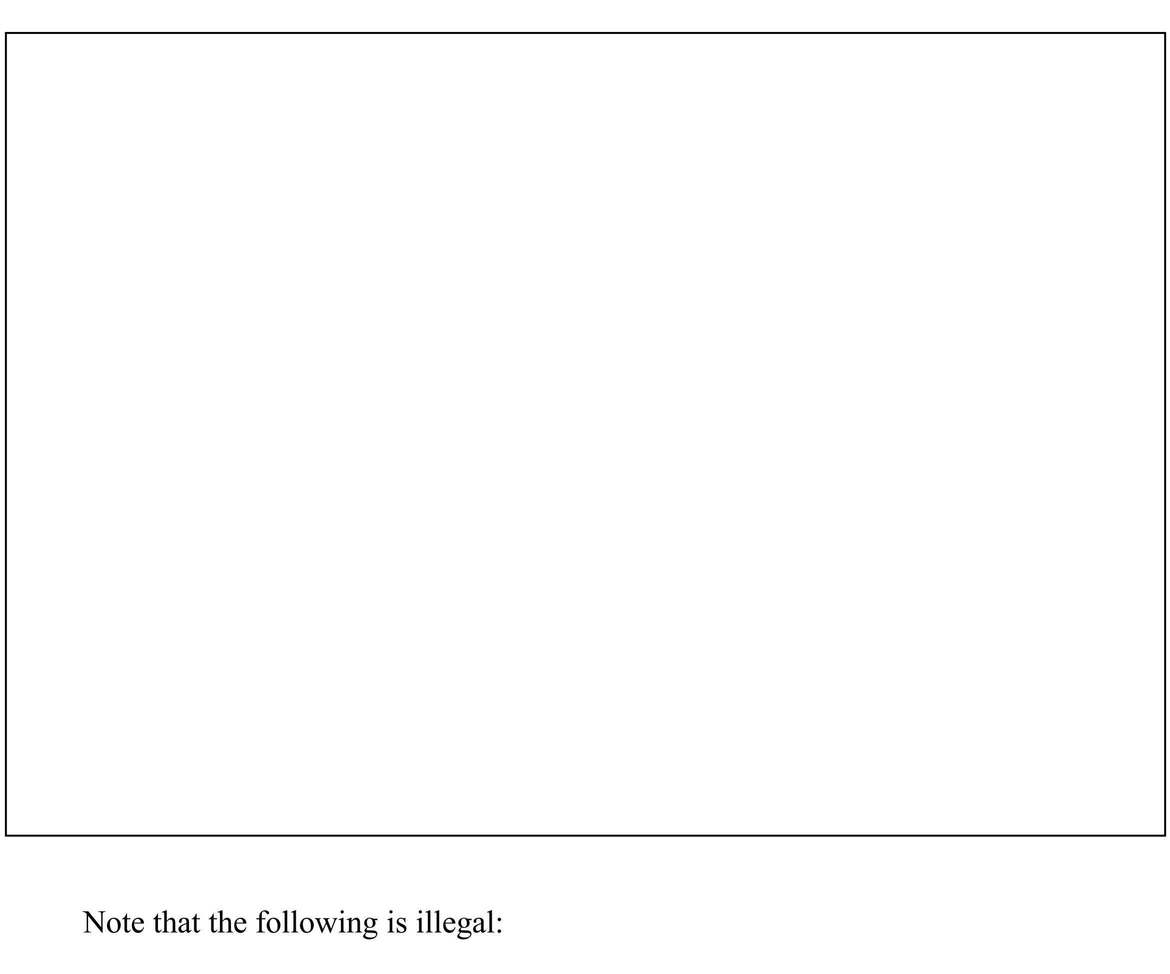
summary

- selection statements
  - if : beware of dangling else
  - switch : stylised if-else chain, no fall-through
- iteration statements
  - for : common, loop scaffolding all done together
  - while : also common
  - do : much less common
- jump statements
  - continue, break, goto : all correlated with bugs
  - best avoided
  - break in a switch is ok
- assertions
  - useful for external testing and asserting invariants

- The title of this chapter should be read in the imperative mood!

You might hear C being described as a block-structured language. Be careful how you say that!

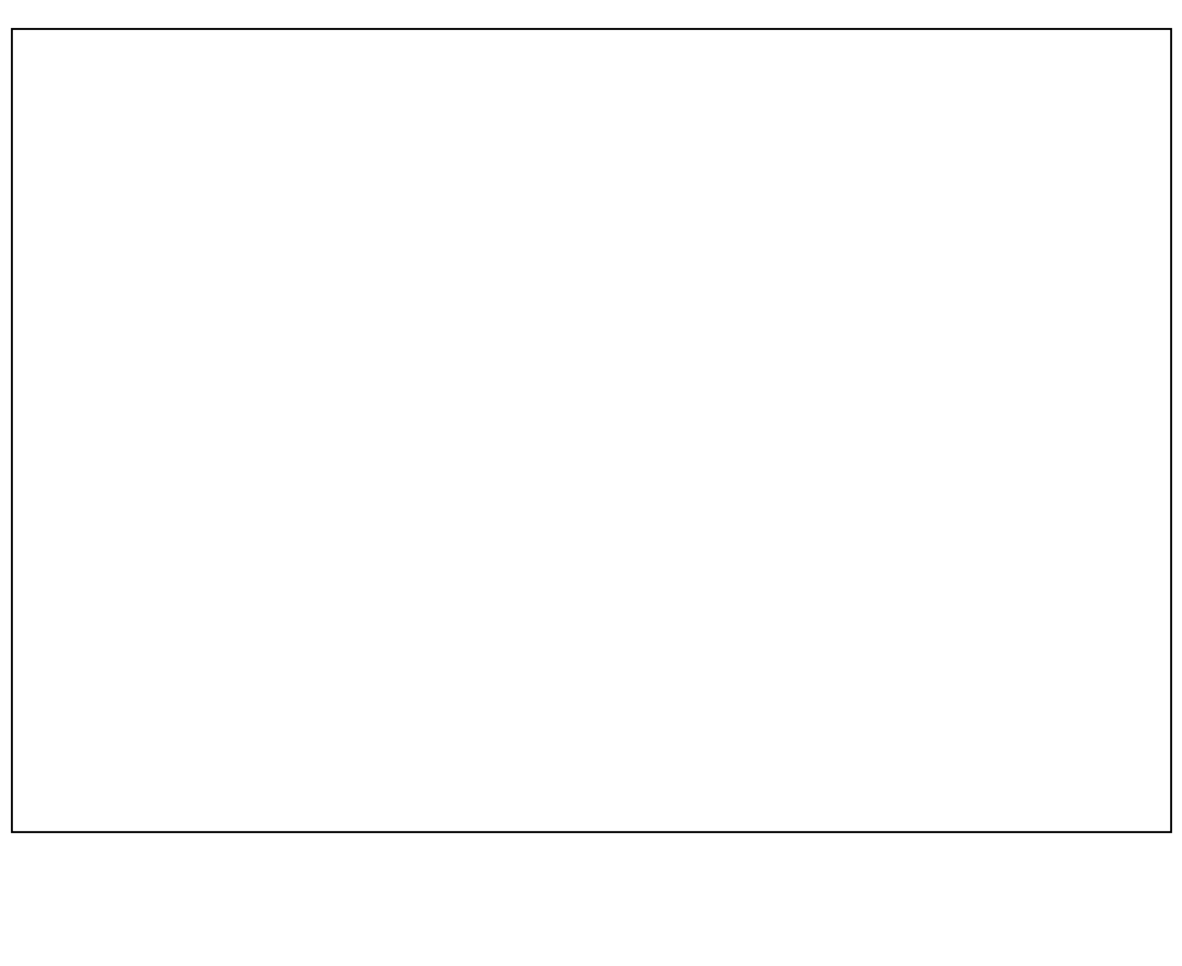
In previous versions of C declarations had to be written at the start of the block, followed by the statements.

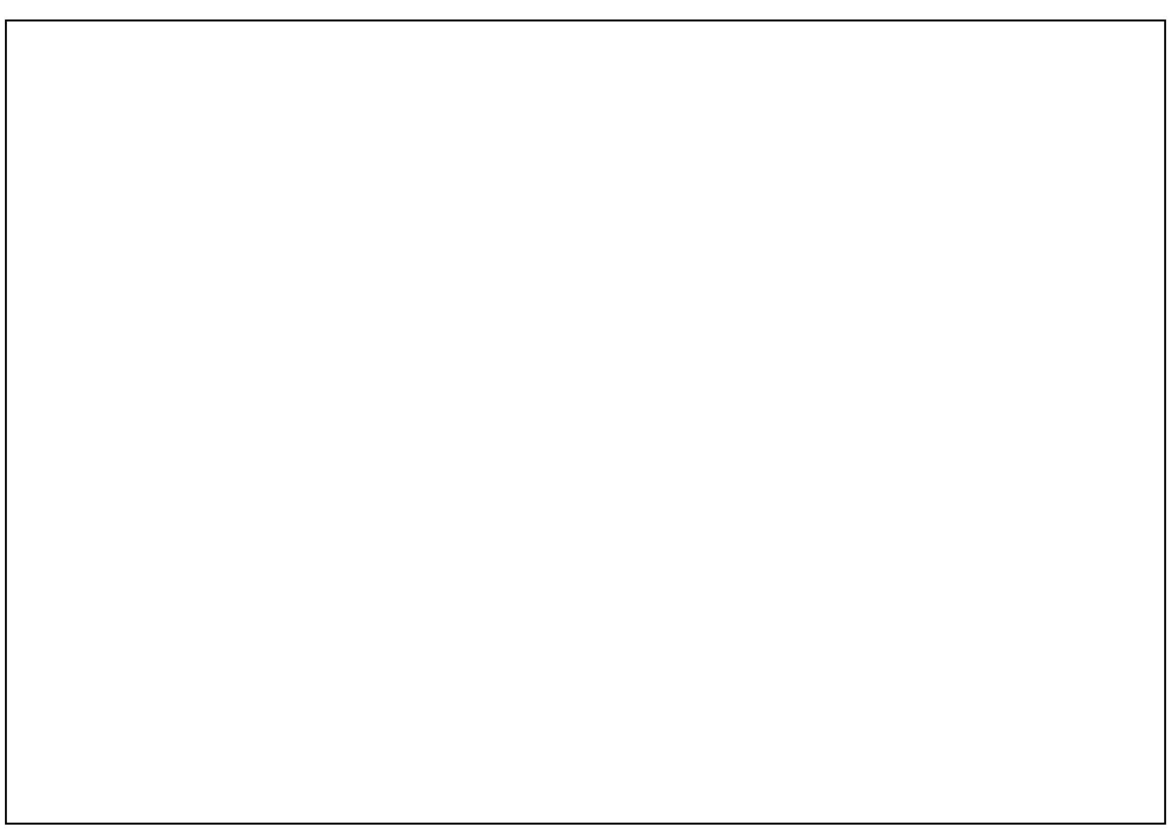


Note that the following is illegal:

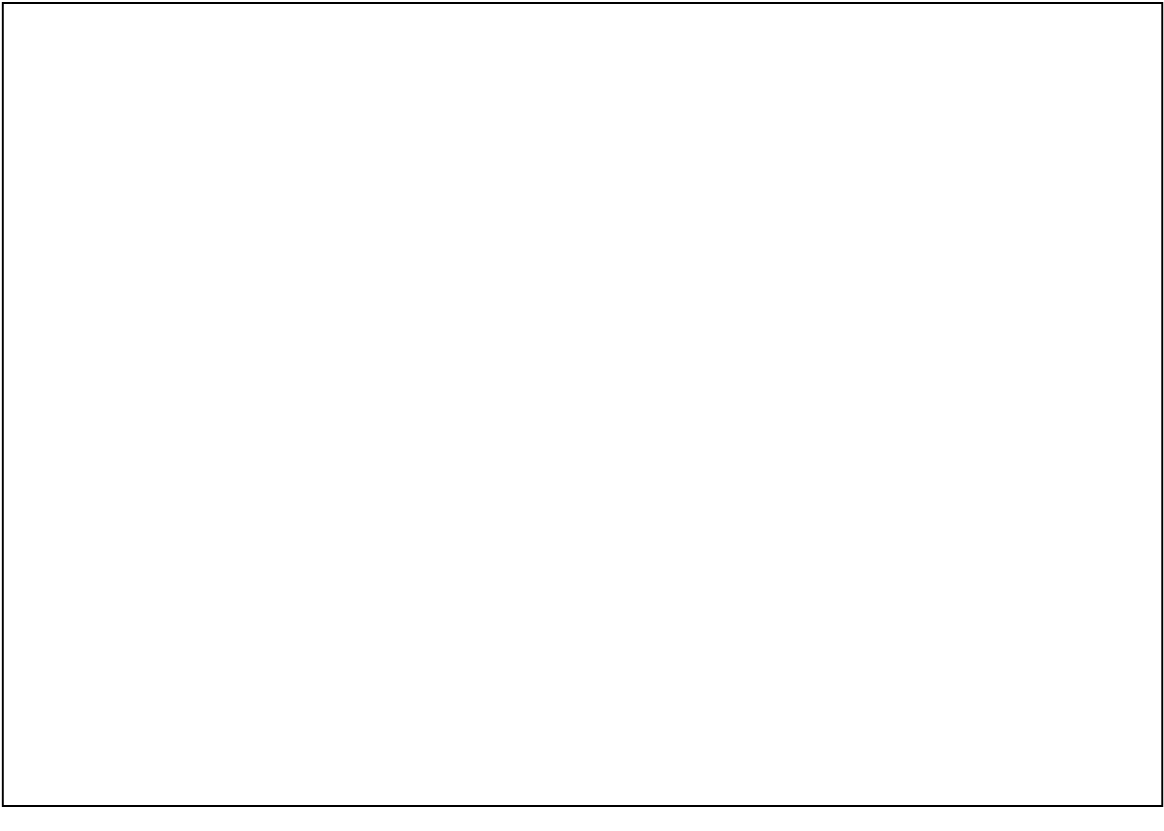
```
if (expression)
        int x = 42;
```

since a declaration is not properly a statement.





My advice is that writing if (42 == x) is not a good idea. It makes the code harder to read. And if you are aware enough of the problem to write the variable on the right hand side surely you are aware enough of the problem to write == instead of =? And it doesn't apply all the time. What about of both arguments are variables? Most crucially of all, writing if (x = 42) should be found by tests. Time and time again studies have shown the two major factors in building software are (1) how interdependent the various parts of your software are – so that when you change one part only some of the rest is affected, and (2) how easy the code is to comprehend.



`case` labels must be compile-time constants. Using a runtime expression will not compile.

```
case f(42) :
```

`case` labels cannot be duplicated:

```
case 1 : case 1 : // compile-time error
```

There is no shortcut for ranges:

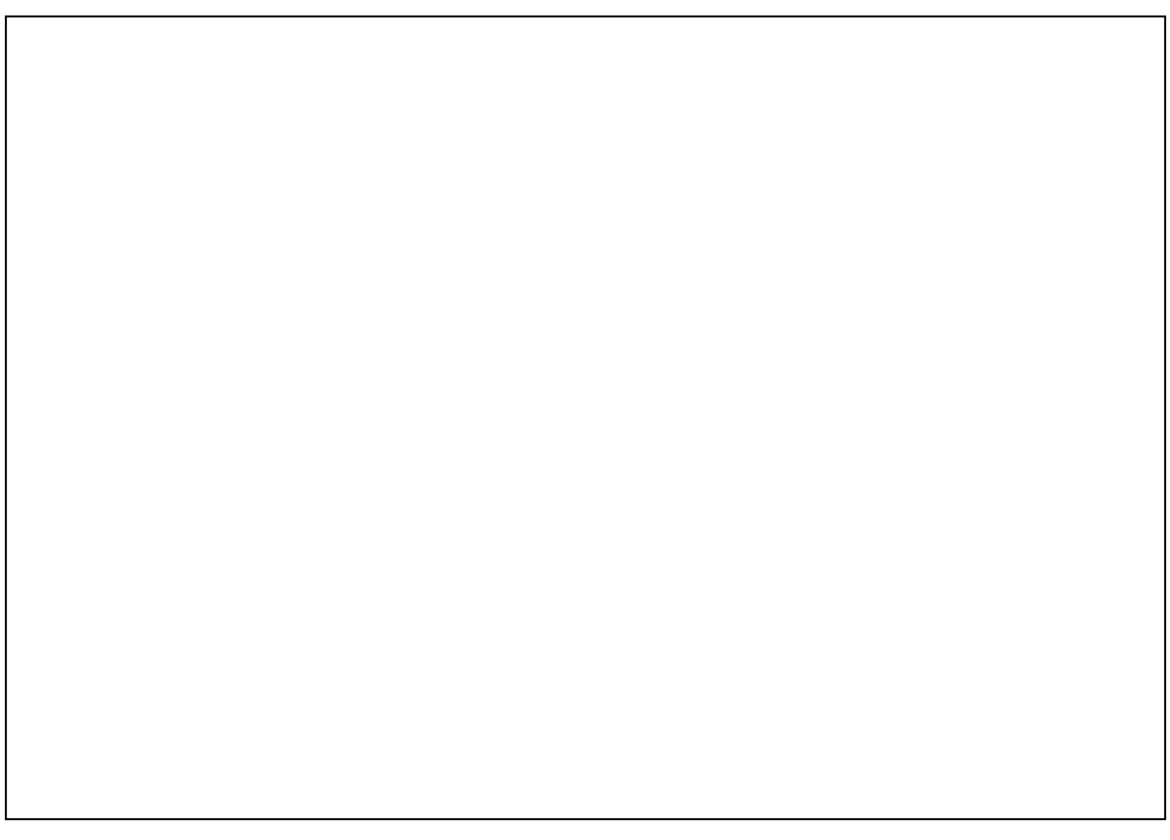
```
case 1..4 : // compile-time error
```

Also:

```
case 1...4 : // compile-time error
```

So instead:

```
case 1: case 2: case 3: case 4: // ok
```



This is a famous piece of code known as Duff's Device named after Tom Duff who invented it while working for LucasFilm. It is an unrolled loop (which assumes count is greater than zero). It is perfectly valid C. Tom Duff wrote of his invention: "Disgusting, no? But it compiles and runs just fine. I feel a combination of pride and revulsion at this discovery. Many people (even Brian Kernighan) have said that the worst feature of C is that switches don't break automatically before each case label. This code forms some sort of argument in that debate, but I'm not sure whether it's for or against."

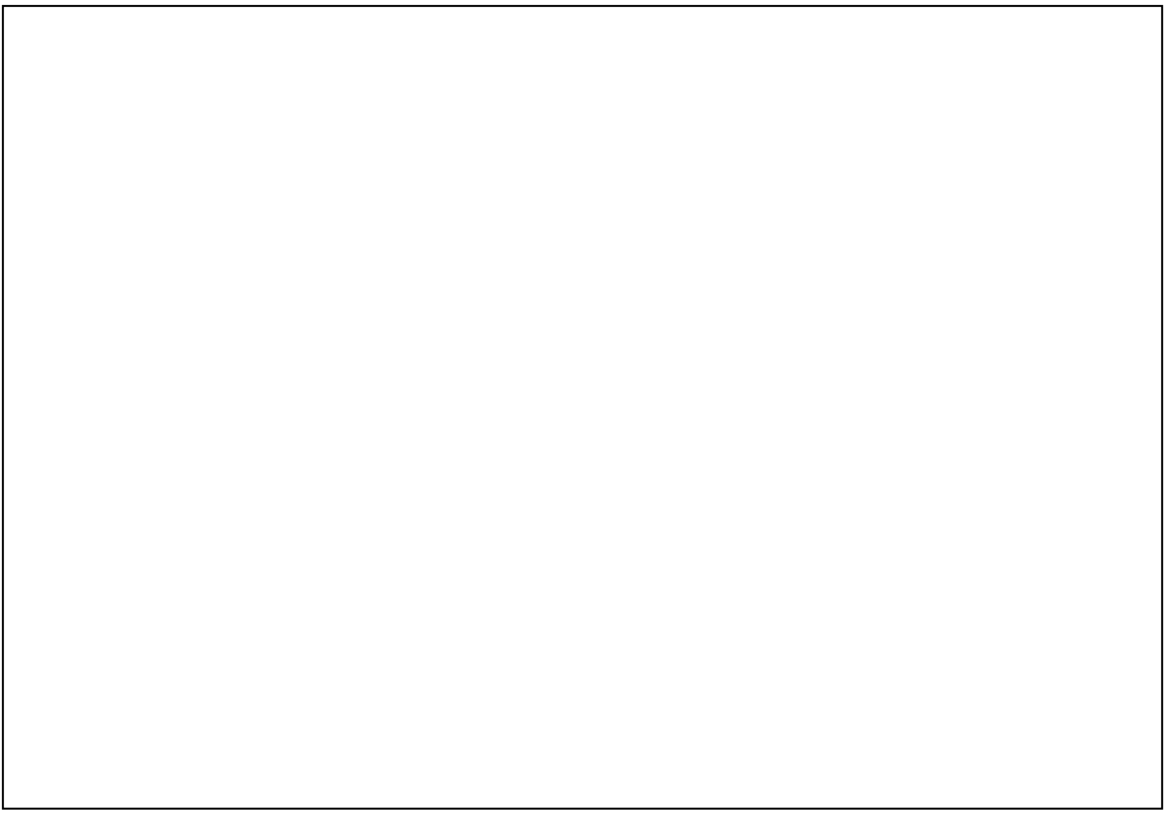
Note that the default label is optional, and if present can appear anywhere in the switch statement. It does not have to be the last label – although that is a common idiom. Note also that if you misspell default you will create a goto label which will still compile. Be careful.

The assignment will not happen. The case labels simply provide points for the flow of control to jump to. For example, if the value of the expression (days % 10) is 2 then the flow will jump to the case 2: line, where the variable at will be in scope, but the runtime will not have flowed through the assignment so it will still have an indeterminate value.

You spell default with the a before the u so it is in error in the slide. However this is not a compile time error since, by chance, the syntax is legal and creates a goto label.

The reason for writing a break for the default section is that it allows the default to be moved. The default does not have to be declared at the end of the switch. If every case section has its own break (or return) the case/default sections can be freely reordered without changing the meaning of the code.

```
switch (days % 10)
{
default : …
case 3 : …
case 2 : …
case 1 : …
}
```



The while statement classically has five parts (the finalise is often absent):

```
initialise;
while (expression)
{
    body
    update
}
finalise
```

In many cases, this standard structure is better suited to a for statement (see next slide) which collects all the iteration scaffolding together in one place. The example on the slide shows while statement being used to write out the values 0 to 9. This is for comparison with the other iteration statement slides. The `while` statement is well suited for situations where the expression and the update are combined into a single function call.

Pre C99 you could not declare a variable in the initialization part of a for statement.

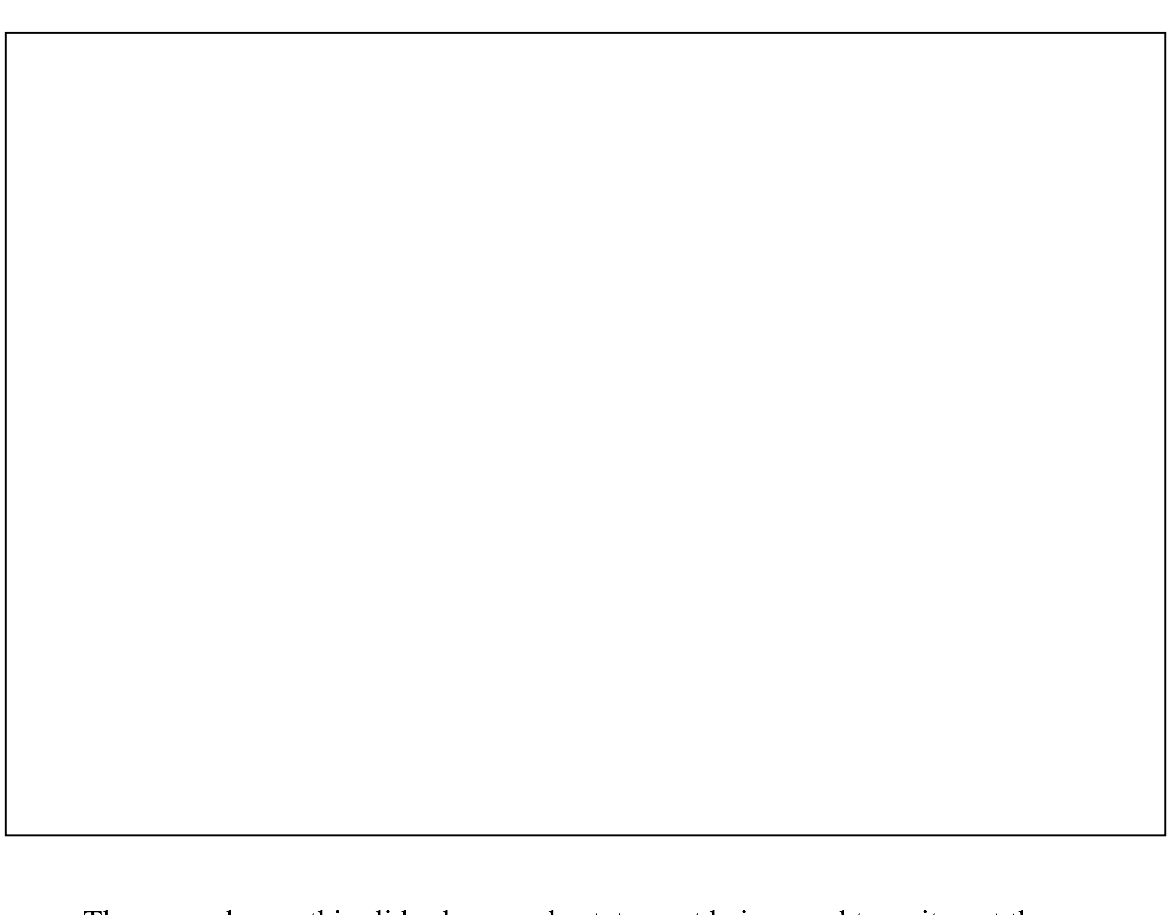
A variable that is declared in the first part of a for statement is scoped to that for statement. For example:

```
for (int digit = 0; digit != 10; digit++)
{ ... }
printf("%d ", digit); // compile-time error
```

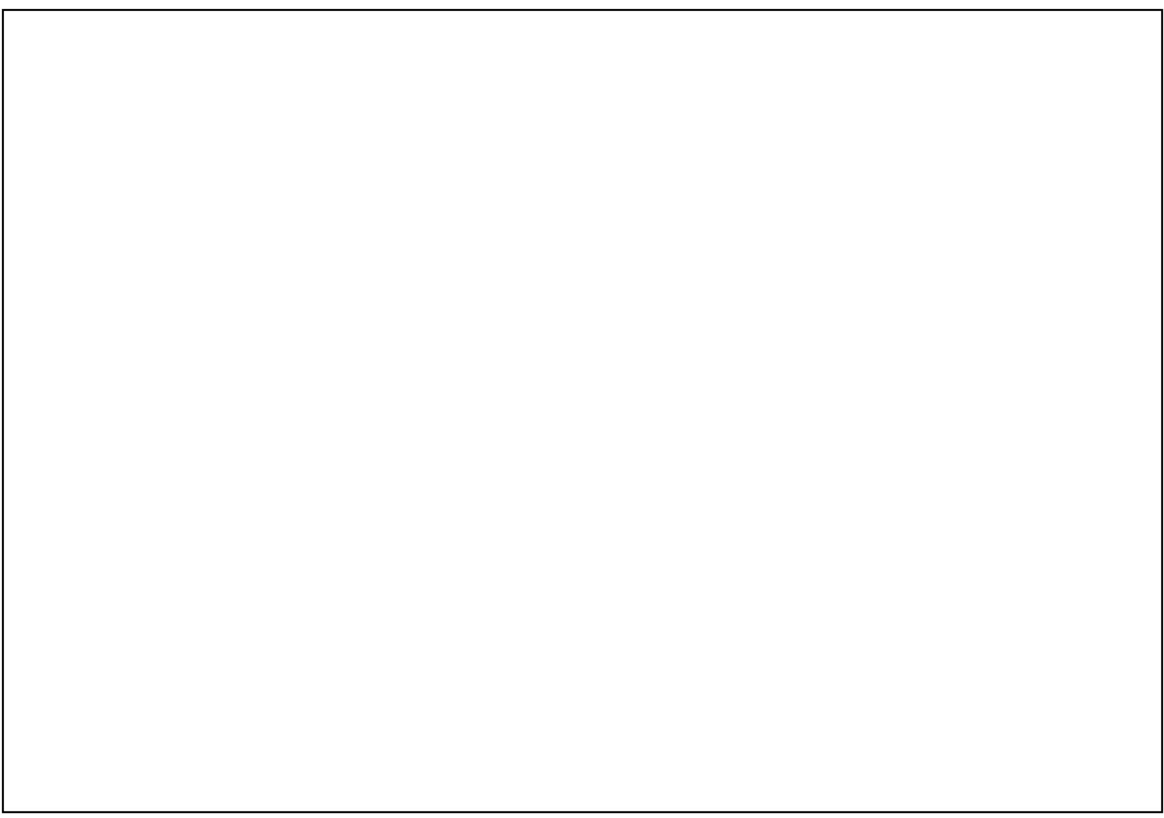
A comma can be used to declare more than one variable, and to perform more than one update statement. For example:

```
for (int i = 0, j = 0; i + j < 20; i++, j++)
{ ... }
```

If the continuation-condition is omitted from a for statement it defaults to true, creating a for-ever statement!



The examples on this slide shows a do statement being used to write out the values 0 through 9. This is for comparison with the other iteration statement slides.



A function can return a single value to the caller by returning the value in a return statement. The type of the value specified in the expression must match the return type of the function or be of a type that can be implicitly converted to the return type of the function.

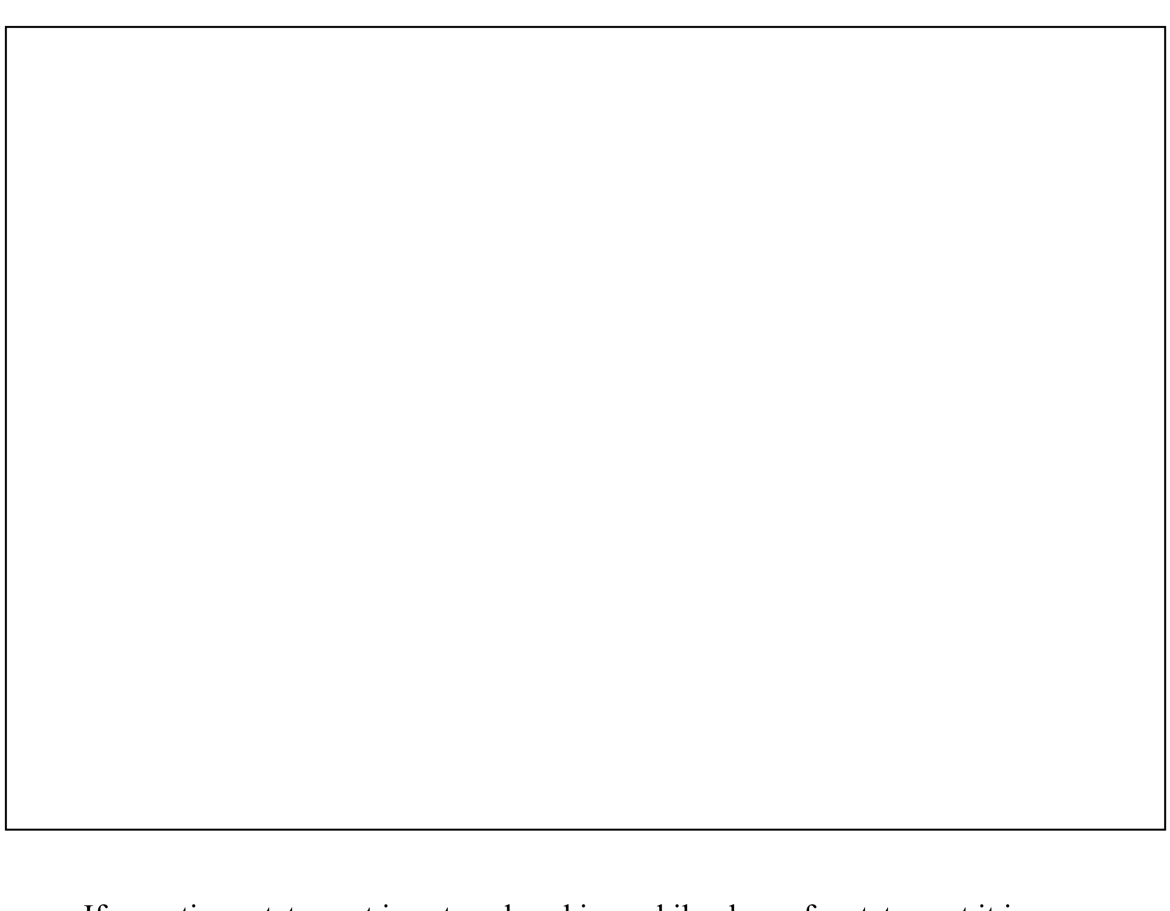
If you need to return more than one value from a function:

- you can group the values together in a struct.
- you can "return" several values by using pointers to non-const parameters.

A function has a single point of entry and many style guides recommend a function also has a single point of exit. Functions that contain a single return statement (as their last statement) are particularly easy to debug; they provide a natural spot for a breakpoint. However, the rule is not a rigid one; many expert developers feel that it's acceptable for a function to contain multiple return statements when each return statement returns a literal or named value, particularly when the return statements are all at the same level of indentation/nesting. For example, returning true if a search finds a value and false if it doesn't (this can improve code comprehension).

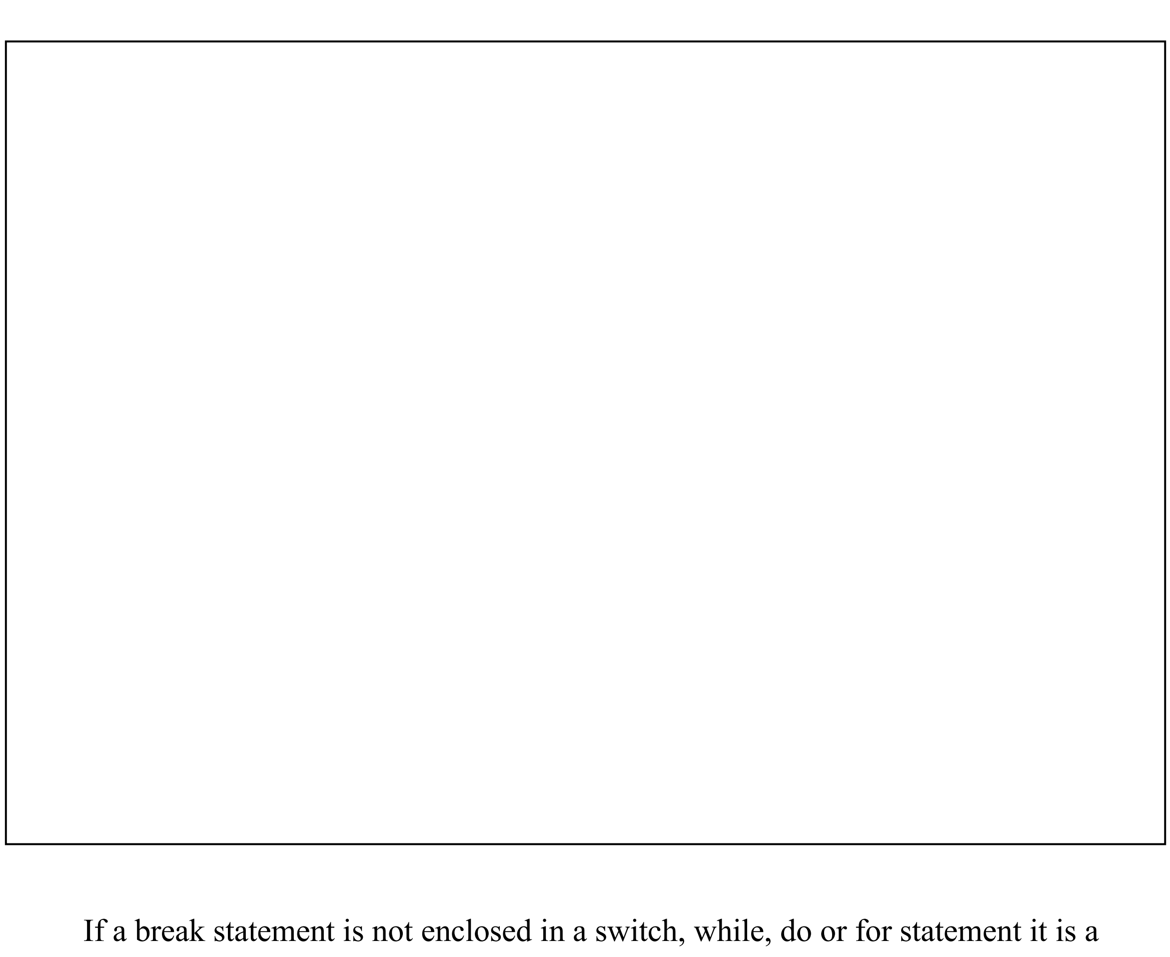
You can use a return statement without an expression to exit a void function. It is considered bad style to write such a return statement as the last statement of a function. It's perfectly reasonable to assume that the programmer reading the function will know that a function implicitly returns when the flow of control reaches its closing brace!

If the closing } brace of a non-void function is reached and the value of the function call is used by the caller the result is undefined.

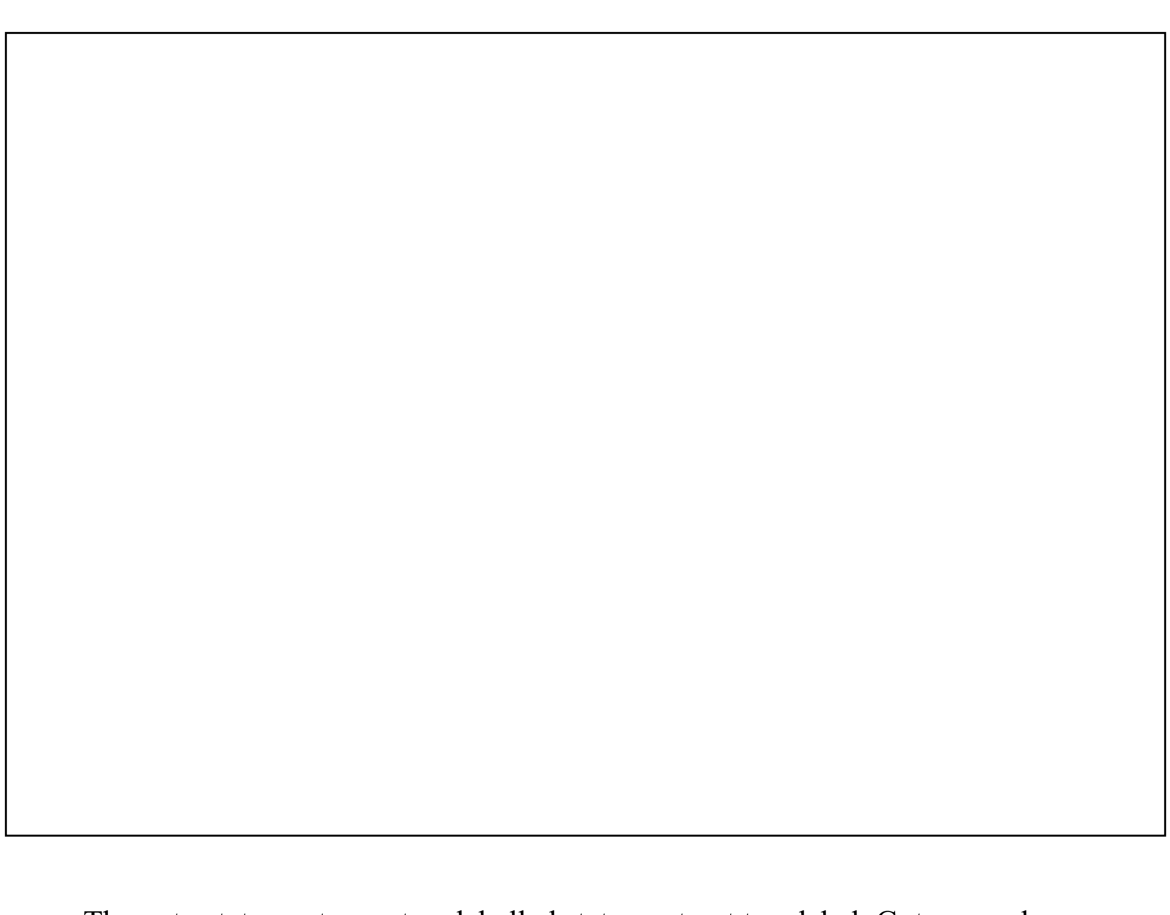


If a continue statement is not enclosed in a while, do, or for statement it is a compile-time error. When multiple while, do, or for statements are nested within each other, a continue statement applies only to the innermost statement.

Most programming guidelines recommend avoiding the goto and continue statement. They make code hard to understand and its behaviour is sometimes quite subtle. For example, a continue inside a for statement, does *not* cause the immediate re-execution of the continuation condition; the update part of the for statement is executed first.



If a break statement is not enclosed in a switch, while, do or for statement it is a compile-time error. When multiple switch, while, do, or for statements are nested within each other, a break statement applies only to the innermost statement.



The goto statement goes to a labelled statement not to a label. Gotos are also correlated with bug although it is fair to say that a forward jumps seem to cause less bugs that backward jumps.

`assert` is not an error-handling mechanism, it is a program fault detection mechanism. When used within production code it should only be used to express (and enforce) necessary assumptions, not just wishes. So, never assert on external or uncontrollable behaviour, such as files or memory allocation. And never (ever) have a side-effect in an assertion expression.

In support of automated testing, unit testing in particular, the simple standard `assert` macro can be used to express necessary and testable requirements on code. When writing automated tests, be sure to break tests up into smaller test functions. Just because it's test code doesn't mean that it should be well factored. Try to name tests after a requirement rather than after a mechanism.