Control Flow

blocks

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

```
;
```

the null statement

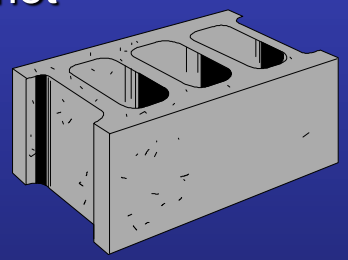
```
declaration ;
```

```
expression ;
```

semicolons required

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

trailing semicolon not required



if statement

- the simplest selection statement

else part is optional

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

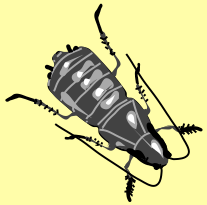
```
const char * day_suffix(int days)
{
    if (days / 10 == 1)
        return "th";
    else if (days % 10 == 1)
        return "st";
    else if (days % 10 == 2)
        return "nd";
    else if (days % 10 == 3)
        return "rd";
    else
        return "th";
}
```

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;
```



??

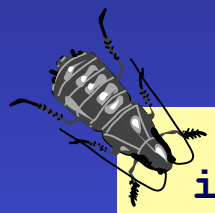
physical indentation == logical indentation

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

?

discussion

- 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

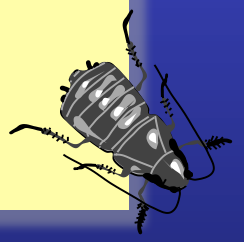
```
const char * day_suffix(int days)
{
    const char * result = "th";

    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

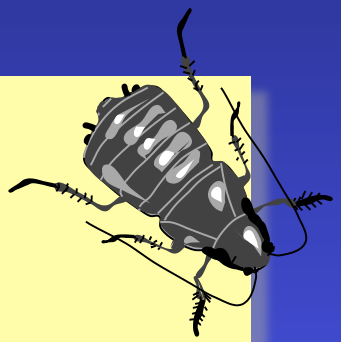
- case labels provide "goto" style jump points
  - when inside the switch they play no part
  - there is no implicit break
  - this is known as fall-through

```
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?

while statement

- 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

for statement

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

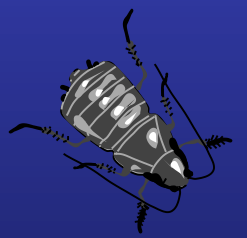
```
for (initialisation; continuation-condition; update-loop )
{
    loop-task
}
```

equivalent while statement →

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

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

0 1 2 3 4 5 6 7 8 9



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;
}
```

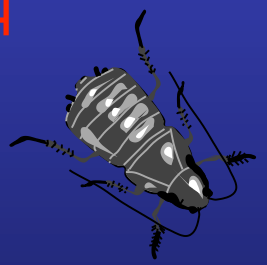
continue statement

- 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
  - perfectly ok 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 (some_error)
        goto cleanup;

    ...

cleanup:
    if (f1 != NULL) fclose(f1);
    if (f2 != NULL) 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