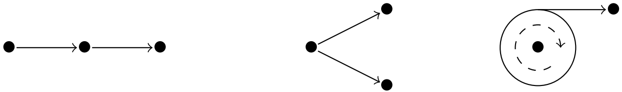
Notes 6.0: Control flow

COMP9021 Principles of Programming

School of Computer Science and Engineering
The University of New South Wales

2012 session 2

Control flow in a nutshell



A program can execute statements in sequence, take decisions, *i.e.*, follows one path or another, and loop for a while. Decisions and loops operate based on the outcomes of tests.

The if statement is the mother of all decisions, the while statement is the mother of all loops; all other statement are variations that only bring extra convenience, not extra power.

The if statement

The if statement is of the form `if (condition) if_body`

- If *condition*, converted to type `bool`, evaluates to `true` then *if_body* is executed; otherwise, *if_body* is skipped.
- *if_body* consists of one or more statements.
- If it consists of one statement only, then *if_body* can be, but does not have to be, surrounded by curly braces.
- If it consists of more than one statement, then *if_body* has to be surrounded by curly braces; it is then a block.

Be stylish (1)

There are four possible styles. A programmer should choose one style and stick to it.

- First style:

```
if (condition)
    statement
```

```
if (condition) {
    statement_1
    ....
    statement_n
}
```

- Second style:

```
if (condition) {
    statement
}
```

```
if (condition) {
    statement_1
    ....
    statement_n
}
```

Be stylish (2)

- Third style:

```
if (condition)
    statement
```

```
if (condition)
{
    statement_1
    ....
    statement_n
}
```

- Fourth style:

```
if (condition)
{
    statement
}
```

```
if (condition)
{
    statement_1
    ....
    statement_n
}
```

Be stylish (3)

If you use `Emacs` and opt for the third or the fourth style, then you will need to edit `.emacs.el`, and

- comment out the line `(substatement-open after)` (adding a semicolon at the beginning);
- insert `(substatement-open 0)` after `(c-offsets-alist` (on a new line).

The condition of an if statement

A value is converted to `true` iff it has at least one bit set to 1, hence

```
if (x)
    ++x;
```

and

```
if (x != 0)
    ++x;
```

are equivalent, and will result in `x` being incremented iff it holds a nonzero value when the test is performed.

Tests are usually better performed on boolean or integer types, as floating point types might be too sensitive to rounding errors.

The if-else pair of statements (1)

The if-else pair of statements is of the form

```
if (condition) if_body
else else_body
```

- If *condition* evaluates to `true` then *if_body* is executed and the following else statement is ignored.
- If *condition* evaluates to `false` then *if_body* is skipped and *else_body* is executed.

The if-else pair of statements (2)

The remarks about braces and style discussed in relation to the if statement apply to the if-else pair of statements, and the same style should be adopted for both, but the first and third styles have to be refined into two subcases in the case where

if_statement

is a unique statement. Indeed:

> *if if_body is itself an if statement, then it has to be surrounded by curly braces; otherwise, else else_statement would be associated with this inner if statement, not with the outer one.*

Indentation, semantics, syntax (1)

Using an appropriate tool such as Emacs makes it hard to write the following program, that is wrongly indented, semantically incorrect, but syntactically correct: proper indentation conveys the logical structure (semantic meaning) of a program to programmers, not to compilers.

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    printf("Enter number between 10 and 20: ");
    int number;
    scanf("%d", &number);
    if (number >= 10)
        if (number > 20)
            printf("Too large\n");
    else
        printf("Too small\n");
    return EXIT_SUCCESS;
}
```

Intermezzo: scanf() and p_prompt (1)

- The function `scanf()` from the IO-library is the counterpart to `printf()`: `scanf()` gets some input, whereas `printf()` produces some output.
- Both functions have a similar syntax: `scanf()` also takes a variable number of arguments: a format string that possibly contains conversion specifications, and for each such conversion specification, the name of a data item to be read. The format string can contain literal characters that have to be read as such. The format specifications are often similar to those used for `printf()`, but not always. For instance,
  - `printf()` uses `%f` both to print out a value of type `float` and a value of type `double`;
  - `scanf()` uses `%f` to read a value of type `float`, but `%lf` to read a value of type `double`;
- The provided `p_prompt()` function is very similar to `scanf()`, but takes an extra argument, before the other two, namely, a string constant that represents the prompt.

Intermezzo: scanf() and p_prompt (2)

Also, `p_prompt()` is much more 'rigorous' than `scanf()`:

- it expects the input to be terminated by carriage return followed by Control D, rather than carriage return only;
- it expects the whole contents of the control string to be used up;
- it expects no input to remain after all conversions have been performed;
- it expects the format string to be syntactically correct;
- it performs a conversion only if the input matches the range of possible values of the type determined by the conversion letter and size specifier.

It also provides a richer set of conversion flags, such as `<=`v to input a value at most equal to v. And of course, it prompts the user relentlessly until the latter does the right thing... For more information, browse `p_io.h`.

Indentation, semantics, syntax (2)

The next program is properly indented, is both syntactically and semantically correct, and is a case where curly braces are needed around the unique statement in the if part of the if-else pair of statements.

```
#include <stdio.h>
#include <stdlib.h>
int main(void) {
    printf("Enter number between 10 and 20: ");
    int number;
    scanf("%d", &number);
    if (number >= 10) {
        if (number > 20)
            printf("Too large\n");
    }
    else
        printf("Too small\n");
    return EXIT_SUCCESS;
}
```

Multi-case decisions (1)

Testing in which of the intervals $(-\infty, 10)$, $[10, 12)$, $[12, 18)$, $[18, 20]$ or $(20, \infty)$ a number belongs to, and outputting a comment about the outcome of the classification, could be coded as follows.

```
    if (number < 10)
        printf("Too small\n");
    else
        if (number < 12)
            printf("Small\n");
        else
            if (number < 18)
                printf("Medium\n");
            else
                if (number <= 20)
                    printf("Large\n");
                else
                    printf("Too large\n");
}
```

Multi-case decisions (2)

It is preferable to modify the standard style and display the code as follows, saving indentation levels and better reflecting the underlying logical structure, where a number of alternatives is considered in sequence.

```
    if (number < 10)
        printf("Too small\n");
    else if (number < 12)
        printf("Small\n");
    else if (number < 18)
        printf("Medium\n");
    else if (number <= 20)
        printf("Large\n");
    else
        printf("Too large\n");
}
```

Multi-case decisions (3)

More generally, a multi-case decision block is of the form

```
if (condition_1) if_body_1
else if (condition_2) if_body_2
...
else if (condition_n) if_body_n
```

possibly ending in

```
else else_body
```

- If there is a least *n* such that `condition_n` evaluates to `true` then `if_body_n` is executed.
- If for all *n*, `condition_n` evaluates to `false` and there is a final else statement, then `else_body` is executed.

The switch statement (1)

When a program has to choose between two alternatives, an if-else pair of statements is appropriate. With more than two alternatives, the multi-case decision construct does the job, but an appealing alternative is provided by the switch statement:

```
switch (integer_expression) {
    case constant_1_1 : ... constant_1_n1 :
        statements_1
    case constant_2_1 : ... constant_2_n2 :
        statements_2
    ....
}
```

The switch statement (2)

`default :` can be used in place of `case` *constant_i_j* `:`.

The execution can be described as follows.

- First *integer_expression* is evaluated.
- Then the program scans the list of labels, namely, *constant_1_1*, *constant_1_2*, etc., until it finds one label that matches that value, in which case it jumps to that line.
- If there is no match then the program jumps to the `default` case, if there is one.
- The `break` statements force the program to break out of the switch structure. Without a break, the program would fall through and execute all statements associated with all cases from the case that yields a match—something that is rarely intended.

The next code fragment, meant to be part of a program that counts the number of vowels in a text, gives the idea.

The switch statement (3)

Without the `break` statements, the results would be wrong.

```
switch (letter) {
    case 'a' :
    case 'A' :  ++a_count;
                break;
    case 'e' :
    case 'E' :  ++e_count;
                break;
    case 'i' :
    case 'I' :  ++i_count;
                break;
    case 'o' :
    case 'O' :  ++o_count;
                break;
    case 'u' :
    case 'U' :  ++u_count;
}
```

The : ? statement

C has a ternary operator that offers a concise way of writing an if-else pair of statements, abbreviating

```
if (condition) if_body
else else_body
```

as

```
condition ? if_body : else_body
```

An typical example of use:

```
printf("%s", nb_of_books > 1 ? "books" : "book");
```

The while statement (1)

The while statement is of the form `while (condition) while_body`

As long as *condition* evaluates to `true`, the program iterates over *while_body*, before it either exits the while loop (in case *condition* evaluates to `false`) or starts a new iteration (in case *condition* evaluates to `true`).

In order not to be trapped in an infinite loop, *condition* has to eventually evaluate to `false`, which will only be caused by modifications brought to *while_body*.

The next program, based on the formula $C = \frac{5}{9}(F - 32)$, is meant to print a table of Fahrenheit temperatures and their Celsius equivalent.

The while statement (2)

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int fahr = 0;
    while (fahr <= 300) {
        printf("%3d %6.1f\n", fahr,
                                5.0 / 9 * (fahr - 32.0));
        fahr += 20;
    }
    return EXIT_SUCCESS;
}
```

The for statement (1)

The for statement is of the form

`for (`*`initialise`*`; `*`condition`*`; `*`update`*`) `*`for_body`*

- First *initialise* is evaluated.
- Then, as long as *condition* evaluates to `true`, the program iterates over *for_body* and then evaluates *update*, before it either exits the for loop (in case *condition* evaluates to `false`) or starts a new iteration (in case *condition* evaluates to `true`).

In order not to be trapped in an infinite loop, *condition* has to eventually evaluate to `false`, which can only be caused by modifications brought to *update* or to *for_body*.

As any general statement, some or all of *initialise*, *condition* or *update* can be empty. At one extreme, one finds the form `for ( ; ; )`.

The for statement (2)

The temperature program can be rewritten using a for loop rather than a while loop as follows.

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    for (int fahr = 0; fahr <= 300; fahr += 20)
        printf("%3d %6.1f\n", fahr,
                              5.0 / 9 * (fahr - 32.0));
    return EXIT_SUCCESS;
}
```

The for statement (3)

Note how the for and while loops share the same condition, how the initialisation of the for loop becomes the last statement before the beginning of the while loop, and how the update of the for loop becomes the last statement in the body of the while loop.

The program also illustrates that *initialise* in `for (`*initialise*`, `*condition*`, `*update*`)` can be both a declaration and an initialisation. This is a feature of C99 and is not possible with older versions of C, such as C89, where all local declarations have to made first, before any other statement.

The do-while statement (1)

The do-while statement is similar to the while statement, except that the test that determines whether or not the program should exit the loop is performed after the body of the loop has been executed, rather than before.

- Both the while and the for loops are entry-condition loops: the test condition is executed *before* each iteration of the loop, so the loop might never be executed,
- A do while loop offers an exit-condition loop, in which the condition is checked after each iteration of the loop, which implies that the loop is executed at least once.
- Prompting for a password for instance is a situation where the loop should be executed at least once, hence a do while loop might appear as more natural: compare the next two programs.

The do-while statement (2)

A version with a while loop:

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    const int course_nb = 9021;
    int nb_entered;
    printf("Enter course number: ");
    scanf("%d", &nb_entered);
    while (nb_entered != course_nb) {
        printf("Enter course number: ");
        scanf("%d", &nb_entered);
    }
    printf("Correct!\n");
    return EXIT_SUCCESS;
}
```

The do-while statement (3)

A shorter and more elegant version, with a do-while loop:

```
#include <stdio.h>
#include <stdlib.h>

pint main(void) {
    const int course_nb = 9021;
    int nb_entered;
    do {
        printf("Enter course number: ");
        scanf("%d", &nb_entered);
    } while (nb_entered != course_nb);
    printf("Correct!\n");
    return EXIT_SUCCESS;
}
```

The break and continue statements

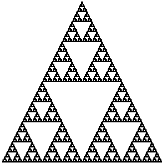
The body of all loop constructs can include:

- a `break` statement that if encountered, forces execution flow to immediately exit the loop;
- a `continue` statement that if encountered, forces execution flow to immediately jump to *`update`* in for loops and *`condition`* in while and do-while loops.

Practically, both statements are part of the body of an if statement or of an else statement.

Example: the Sierpinski triangle

It can be obtained from Pascal triangle by drawing a black rectangle when the corresponding number is odd. We use a particular case of Luca's theorem which states that the number of ways of choosing k objects out of n is odd iff all digits in the binary representation of k are digits in the binary representation of n.



pascal_fractal.c

Example: sums of even numbers in the Fibonacci sequence

Recall that the Fibonacci sequence is

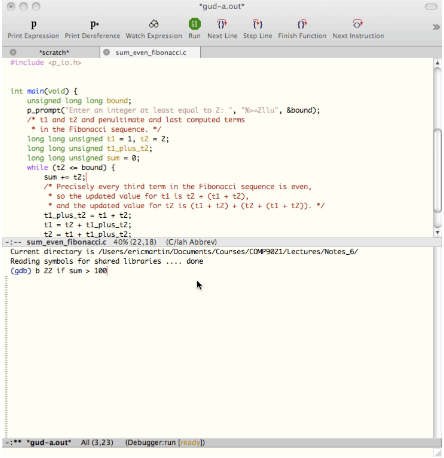
$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89$$

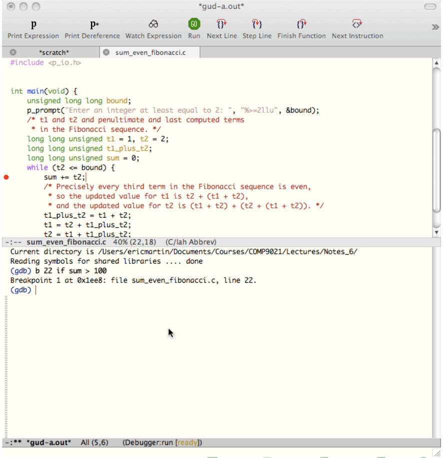
From the third term onwards, every term in the sequence is the sum of the previous two.

sum_even_fibonacci.c

Setting conditional breakpoints (1)

From the gdb window, use `b` *for breakpoint and give as argument the line of the code and a conditional expression*





Setting conditional breakpoints (2)