Chapter Three

Selection

Chapter 3

Chapter 3 – Concepts

- Logical expressions and precedence again.
- Selection via the `if` statement, with or without `else`
- Selection via the `switch` statement (discouraged)

3.1 Logical expressions

Relational operators test for equality and ordering.

In C: `<`, `<=`, `>`, `>=`, `==`, and `!=`.

Note that the equality test uses `==` and not `=`.

All of these operators yield `int` as the result type.

3.1 Logical expressions

In C, expressions that evaluate to "true" generate integer 1.

Expressions that evaluate to "false" generate integer 0.

Logical expressions may appear anywhere: `num_neg+=(n<0)` is valid.

In general, where true/false values are required, any non-zero value is interpreted as being true; only zero is taken as false.

3.1 Logical operators

Logical operators combine `int` true/false values to obtain `int` true/false values:

Operands		Operation		
*e*1	*e*2	*e*1 `&&` *e*2	*e*1 `||` *e*2	`!` *e*1
0	0	0	0	1
0	NZ	0	1	1
NZ	0	0	1	0
NZ	NZ	1	1	0

3.1 Exercise 1

When is each of these expressions true?

```
5<x

5<x && x<=10

5<x && x<=10 || 15<x && x<=20

month==4 || month==6 || month==9 || month==11

year%4==0 && (year%100!=0 || year%400==0)
```

3.1 Logical expressions and precedence

The precedence rules specify default evaluation order.

Operators	Operation class	Precedence
`++`, `--`	postinc, postdec	Highest
`!`, `-`, `(`*type*`)`	not, negation, casting	
`*`, `/`, `%`	multiplication	
`+`, `-`	addition	
`<`, `>`, `<=`, `>=`	comparison	
`==`, `!=`	equality	
`&&`	and	
`||`	or	
`=`, `+=`, `*=`, etc	assignment	Lowest

3.1 Logical expressions and precedence

Unlike the other operators, the various assignment operators are evaluated from right to left. For example, the statements:

```
x = y = 0;
s = n += 1;
```

have the operation ordering

```
x = (y = 0);
s = (n += 1);
```

Including redundant parentheses costs nothing.

3.1 Logical expressions and precedence

Within the overall ordering imposed by precedence, the C system may choose how to evaluate expressions.

In `a*b+c*d` either of `a*b` or `c*d` might get evaluated first.

This means that you must be careful to avoid expressions that have side effects, including the `++` and `--` operators. Use such operators cautiously and sparingly.

3.1 Logical expressions and precedence

Left-to-right evaluation order is guaranteed for the "and" (`&&`) and "or" (`||`) operators.

In addition, for "and" and "or", evaluation is terminated as soon as the outcome is known. Expressions like `x!=0 && y/x>5` are safe.

3.2 Selection

The `if` statement has two forms:

```
if ( guard )
    statement1
else
    statement2
```

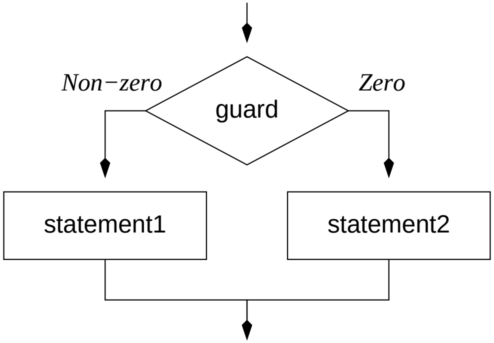
and

```
if ( guard )
    statement1
```

Either or both of *statement1* and *statement2* can be empty statements, or compound statements bracketed by "{" and "}" pairs.

3.2 Selection

The flow of control through the two-branch `if` statement is:



3.2 Selection

Example 1:

```
if (n < 0)
    num_neg += 1;
```

Example 2:

```
if (scanf("%d%d%d", &n, &m, &r) != 3) {
    printf("scanf failed to read three items\n");
    exit(EXIT_FAILURE);
}
```

3.2 Selection

The function `exit` can be used at any point to terminate program execution. Return of a non-zero exit value signals an "abnormal" termination.

The constants `EXIT_SUCCESS` and `EXIT_FAILURE` are defined in the header file `stdlib.h`, and are accessed using

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

3.2 Selection

Example 3:

```
if (year%4==0 && (year%100!=0 || year%400==0)) {
    /* need to allow for leap years */
    length_of_year = 366;
    length_of_feb = 29;
} else {
    /* not a leap year */
    length_of_year = 365;
    length_of_feb = 28;
}
```

3.2 Selection

Example 4:

```
if (month==2) {
    length_of_month = length_of_feb;
} else if (month==4 || month==6 ||
           month==9 || month==11) {
    /* thirty days hath september, april, june,
        and november */
    length_of_month = 30;
} else {
    /* all the rest have 31, except february... */
    length_of_month = 31;
}
```

Exercise 3.4

Write a program that reads a date in `dd/mm/yyyy` format and prints, in the same format, the date that it will be tomorrow.

Remember, start with a very simple program, and then expand it incrementally.

3.3 Pitfalls to watch for

- `equalinif.c`

What is output by this program when 0 is entered? Why?

Using “=” where “==” is intended is a very common mistake.

3.3 Pitfalls to watch for

- danglingelse.c

What is the final value of variable `z`? Why?

The compiler provides optional warning messages; ask for them with `-Wall` and read them all carefully.

3.3 Pitfalls to watch for

- `threetest.c`

Is the final value of `z` going to be 9? Why not?

This one may not result in a warning message being generated.

A silent compiler does not guarantee that your program is correct. Programs should always be carefully tested.

3.4 Case study

The 2012/13 Australian tax rates stipulate that annual income between $18,200 and $37,000 is taxed at 19.0 cents in the dollar; income between $37,000 and $80,000 at 32.5 cents in the dollar; and income between $80,000 and $180,000 at 37.0 cents in the dollar. Any further income is taxed at 45.0 cents in the dollar. In addition, a Medicare levy of 1.5 cents in the dollar is applied to all income. For a given gross income, what net income is received?

- `taxation.c`

3.5 The switch statement

```
switch ( integer expression ) {
case ( integer1 ):
     statement1
     break;
case ( integer2 ):
     statement2
     break;
          ...
case ( integern ):
     statementn
     break;
default:
     statement
}
```

3.5 The switch statement

A sensible `switch`:

- `switch1.c`

To manage the logic flow, each `case` should have a `break`.

3.5 The switch statement

A dangerous `switch`:

- `switch2.c`

Trace the final value for each input value of `x`. Was it easy?

Chapter 3 – Summary

- Logical and relational expressions are integer valued.
- Be careful with precedence, be especially wary of side-effects.
- The `if` statement allows conditional execution based on an integer-valued guard.
- Switch statements should be avoided, they encourage poorly structured code. If for some reason you must use them, be disciplined, and use a `break` for every `case`.