Chapter 9

Structured Programming Using Control Flow Commands

9.1 Statements for Decision and Control

- **Conditionals** and **Selection**:
  - `if-then`
  - `if-then-else`
  - `switch`
- **Loops**:
  - `for` loop
  - `while` loop
  - `do-while` loop
- `Continue` and `break`
- `Goto`

9.2 Conditionals — “Decisions, Decisions, ...”

9.2.1 The `if` and `if-else` Statement

- A single target statement:

```
if (conditional_expression) statement
```

- A single target statement:

```
if (conditional_expression) statement
else statement
```

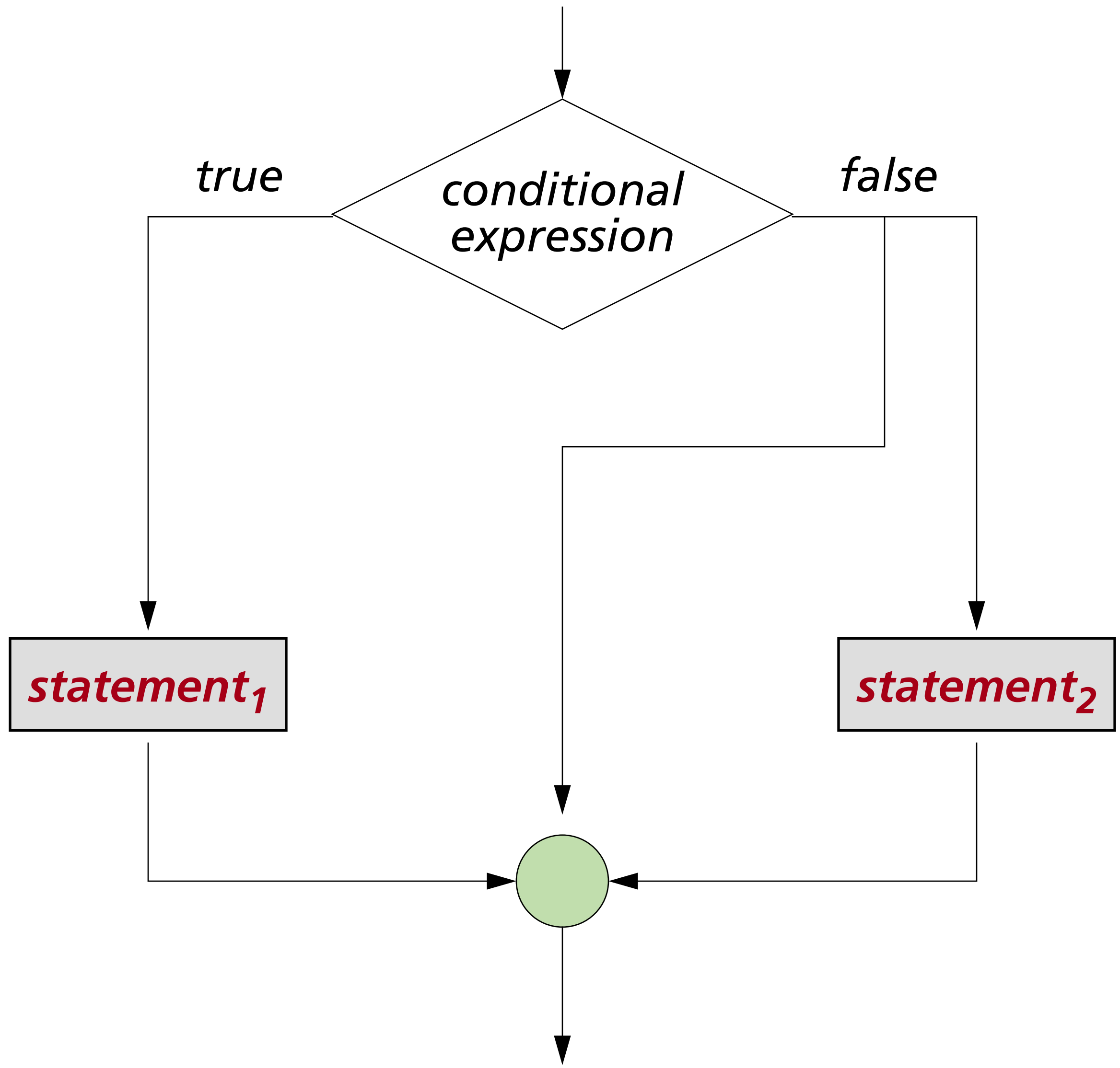
- Binary conditional with a sequence of statements:

```
if (conditional_expression)
{
  statement_sequence
}
```

- Binary conditional with a sequence of statements:

```
if (conditional_expression)
{
  statement_sequence
}
else
{
  statement_sequence
}
```

Behaviour diagram for conditional statements



Example (1): Division with Check for Zero Divider

```
// Divide the first number by the second

#include <iostream.h>

void main()
{
  int a, b;

  cout << “Enter two numbers: “;
  cin >> a >> b;

  if (b != 0) cout << a/b << ‘\n’;
  else cout << “Cannot divide by zero.\n”;
}
```

9.2.2 Truth Values in C++

At the heart of binary logic is the manipulation of boolean[1] truth values:

- T or *true*
- F or *false*.

In C++ the actual representations for truth values are:

- the integer/float/double **zero** for ***false***, and
- any **nonzero** value for ***true***.

1. George Boole, a nineteenth-centruy logician and mathematician

Examples:

- All of the following is interpreted as ***false***:
  - int k = 0;
  - float m = 0; double n = 0;
  - char c = '\0'

- All of the following is interpreted as ***true***:
  - int k = 1, m = -7, n = 11;
  - float p = 1.414;
  - float q = 0.0001;
  - char ch1 = 'g', ch2 = '4'; // any other character than '\0'

Example: Division with Check for Zero Divider (version 2)

```
// Divide the first number by the second

#include <iostream.h>

void main()
{
  int a, b;

  cout << “Enter two numbers: “;
  cin >> a >> b;

  if (b) cout << a/b << ‘\n’;
  else cout << “Cannot divide by zero.\n”;
}
```

Example: Checking for numbers between 0 and 1

```
#include <iostream.h>
#include <math.h>

void main()
{
  float X;

  cout << "Enter a positive number X = ";
  cin >> X;

  if (floor(X))
    cout << "X is 1 “;
    cout << “or greater than 1." << endl;
  else
    cout << "X is less than 1." << endl;
}
```

Attention: The `if` condition accepts <u>any</u> expression

```
int k = 1;

if (k = 0)
  cout << “It´s a zero.\n”;
else
  cout << “It´s “ << k << “.\n”;
```

What does this program section return?

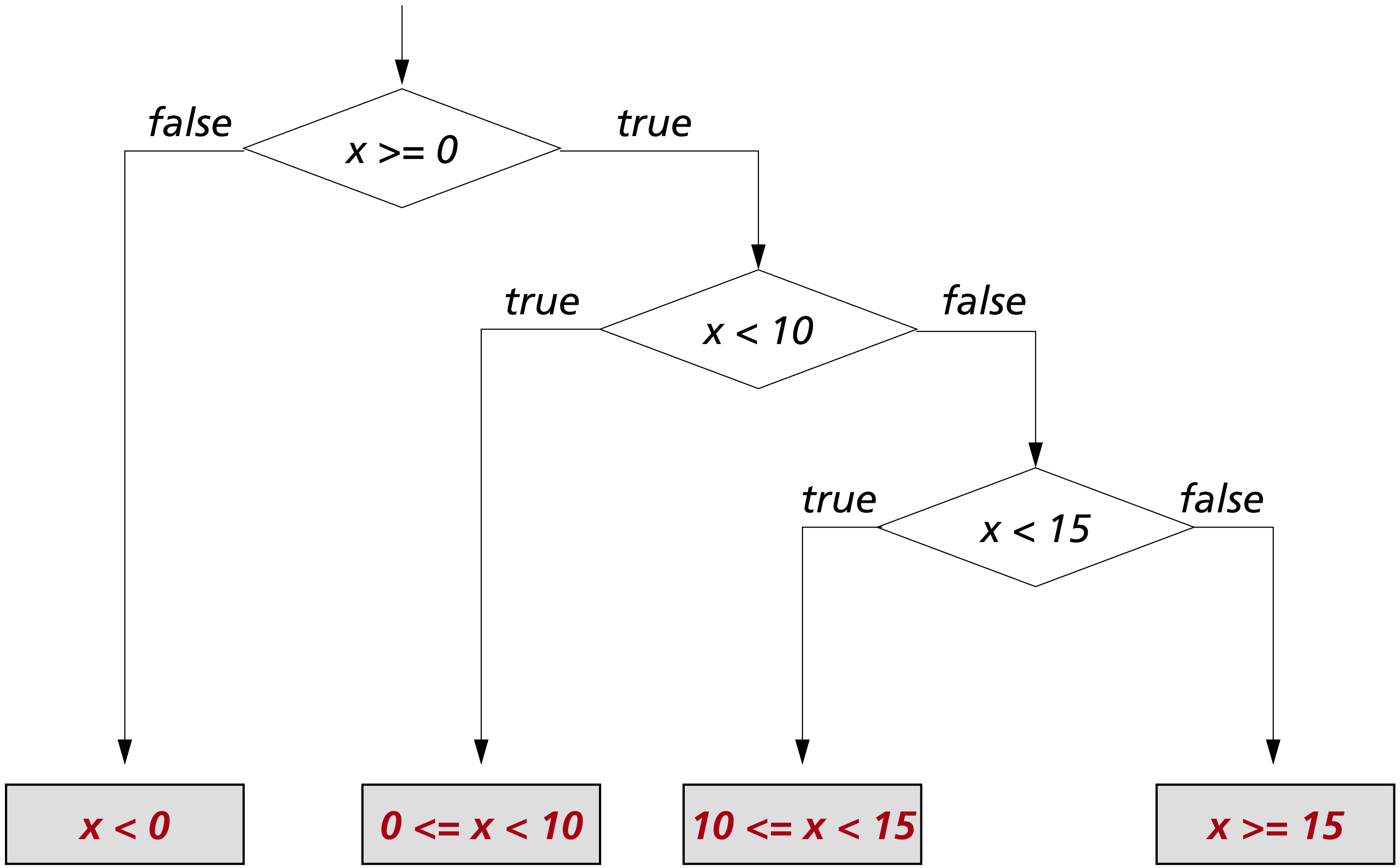
9.2.3 Nested `if` Statements

```
if (c1) {
  if (c2) statement_1; // c1 and c2

  if (c3) statement_2; // c1 and c3
  else statement_3;    // c1 and not c3
}
else statement_4;      // not c1
```

Example: Identifying the value range of a number

```
if (x >= 0)      // x is non-negative
{
  if (x < 10)  // ... and x < 10
    cout << “0 <= “ << x << “ < 10”;
  else             // x >= 10
  {
    if (x < 15)// between 10 and 15
      cout << “10<= “ << x << “ < 15”;
    else cout << x <<“ >= 15”; // > 15
  }
}
else               // x < 0
{
  cout << x << “ is negative.”;
}
```

Behaviour diagram of example program



9.2.4 Short-Circuit Evaluation

As soon as a compound expression produces a value that will completely determine the value of the total expression, evaluation stops.

Example:

```
if (n != 0)
   if (0 < x && x < 1/n) statement
```

More efficient:

```
if ( (n != 0) && 0 < x && x < 1/n )
   statement
```

9.2.5 The *if-else-if* Ladder

```
if(condition)
  statement;
else
  if(condition)
    statement;
  else
    if(condition)
      statement;
                            ...
                                else
                                statement;
```

This deeply nested *if-else* structure can be re-formatted!

Reformatted nested *if-else* structure (with single statements):

```
if(condition)
  statement;
else if(condition)
  statement;
else if(condition)
  statement;
…
else
  statement;
```

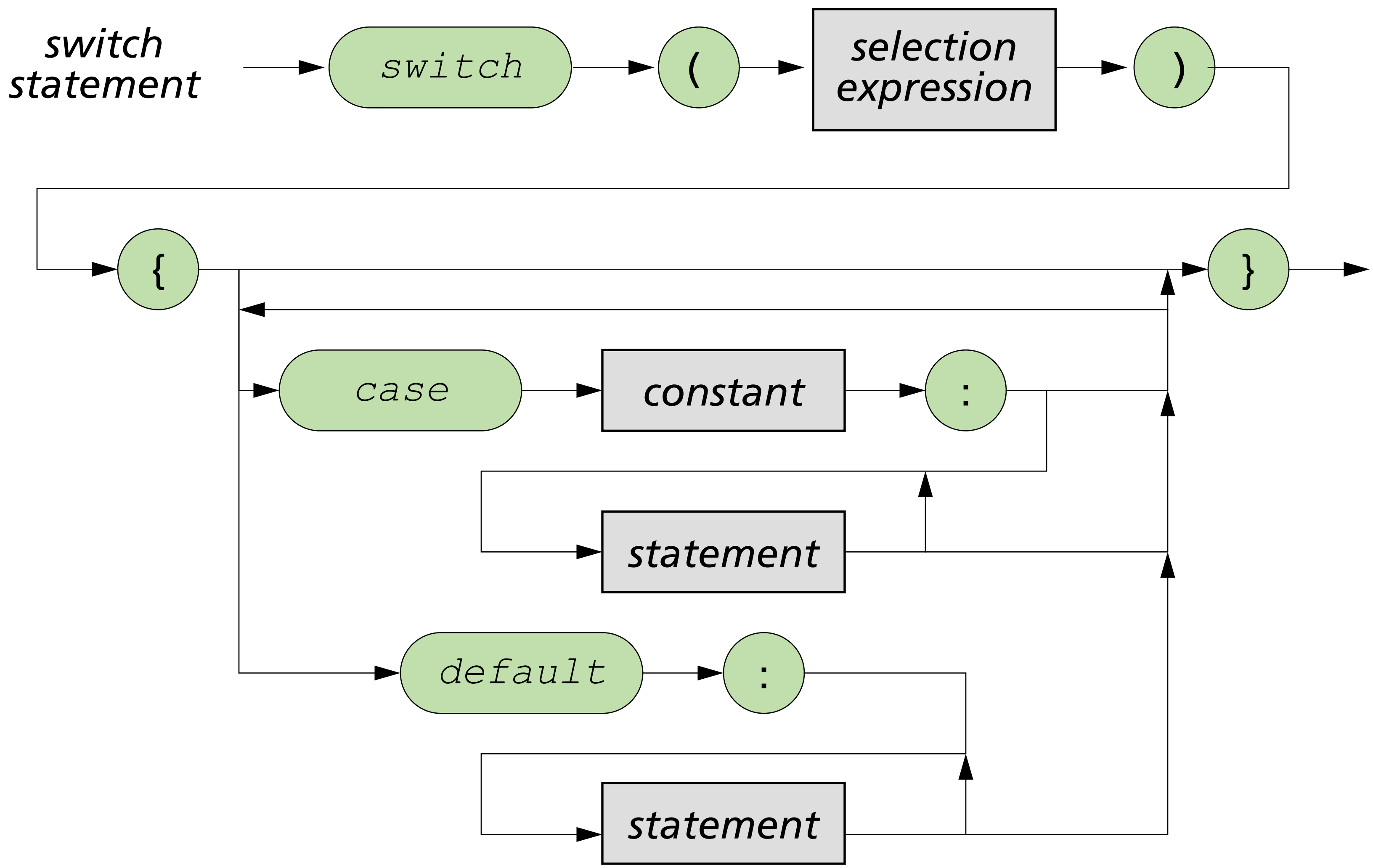
Example:

```
if (x < 0)
  … // x is negative

else if (x > 0)
  … // x is positive

else … // x is zero
```

Reformatted nested *if-else* structure (with statement sequences):

```
if (condition){
  statement_sequence}
else {if (condition){
  statement_sequence}
else {if (condition){
  statement_sequence}
…
else {
  statement_sequence}
```

9.2.6 The *switch* Statement



A typical `switch` structure:

```
switch(selection_expression){
  case constant1:
    statement_sequence
    break;
  case constant2:
    statement_sequence
    break;
  case constant3:
    statement_sequence
    break;
  …
  default:
    statement_sequence
}
```

Example: Convert final grade (0-100) to letter grade

```
int finalGrade; char letterGrade;

switch (finalGrade/10)
{
  case 9:  letterGrade = ‘A’;
           break;
  case 8:  letterGrade = ‘B’;
           break;
  case 7:  letterGrade = ‘C’;
           break;
  case 6:  letterGrade = ‘D’;
           break;
  default: letterGrade = ‘F’;
}
```

Example: Convert final grade (0-100) to letter grade

```
int finalGrade; char letterGrade;

switch (finalGrade/10)
{
  case 10: cout << “Wow--100!”;
  case 9:  letterGrade = ‘A’; break;
  case 8:  letterGrade = ‘B’; break;
  case 7:  letterGrade = ‘C’; break;
  case 6:  letterGrade = ‘D’; break;
  default: letterGrade = ‘F’;
}
```

9.2.7 Nested *switch* Statements

```
switch(ch1) {
  case ‘A’:
    cout << “Outer switch: A”;
    switch(ch2) {
      case ‘A’:
            cout << “Inner switch: A”;
            break;
      case ‘B’:
            // …
    }
    break;
  case ‘B’:
    // …
  default:
    // …
}
```

9.3 Loops — "Doing Things Over and Over Again ..."

Loops are control structures that repeat a series of statements without re-typing them.

Loops are commonly used for …

- counting
- summing
- repeated multiplication, increment, decrement
- keeping track of values (current, previous)
- repeating a sequence of commands or actions
- ...

Definitions around loops:

- **Loop entry**: statement(s) before entering a loop
- **Loop body**: statement(s) that are repeated
- **Loop condition**: expression to be evaluated in order to decide whether a new repetition (= iteration) should be started
- **Loop exit**: end of the loop, where the control flow leaves the loop

9.3.1 An Abstract View of Loops

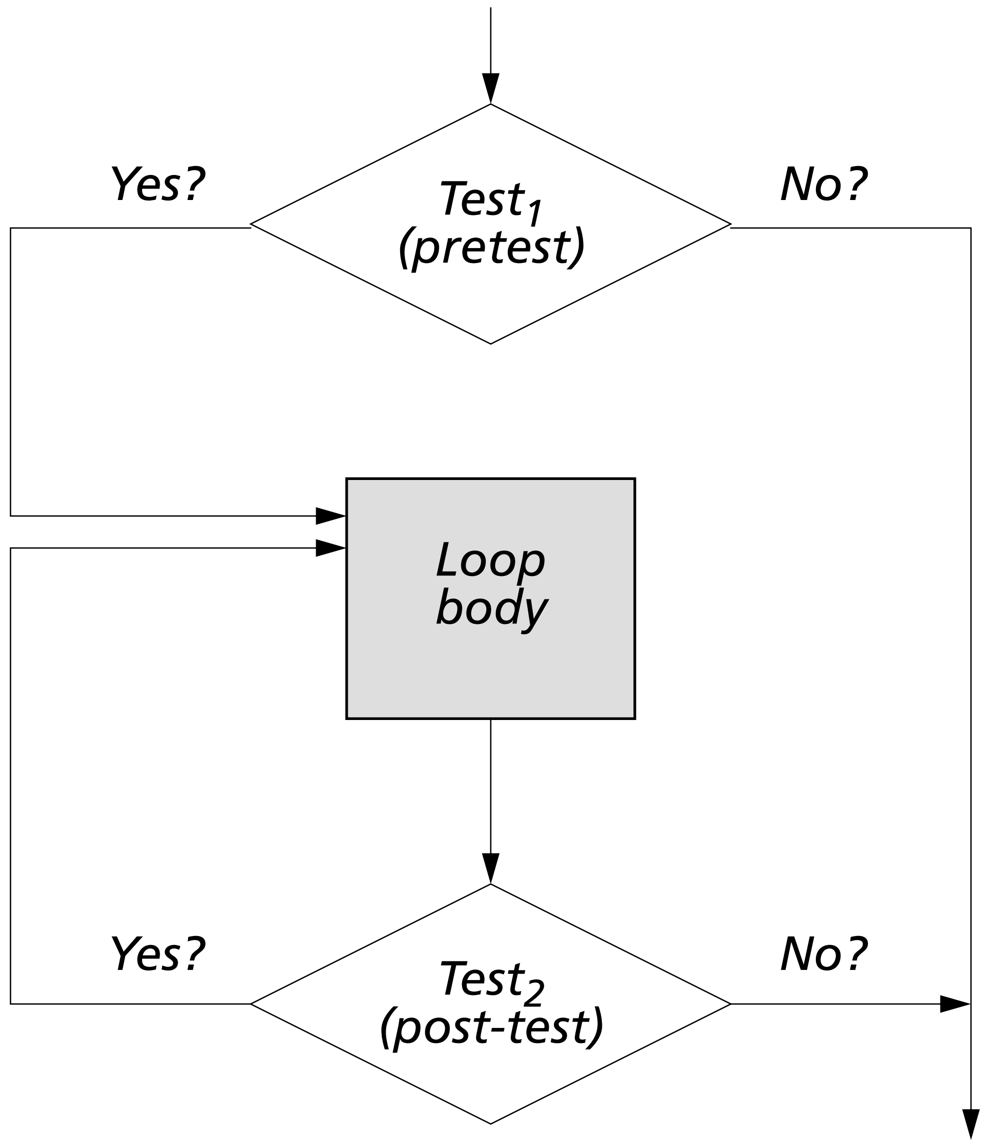
When you write a repetition instruction, you should always be clear about these three issues:

1. **Enter**: The conditions under which you want to enter the loop.

2. **Continue**: The conditions under which you want to continue the loop.

3. **Exit**: The conditions under which you want to exit the loop.

General Loop Structure



The Three Loop Conditions

To understand how to construct a correct loop, with the loop condition correctly related to the loop body, we need to consider three conditions:

- **Entry** condition:

  the condition that must hold in order for the loop body to execute.

  Alternatively, an entry condition may be one that is always true, a trivial condition, so that the loop body always executes at least once.

After entering a loop and after having executed its statements, the question arises whether to continue or not to continue.

- **Repeat** or **continuation** condition (often: = entry condition):

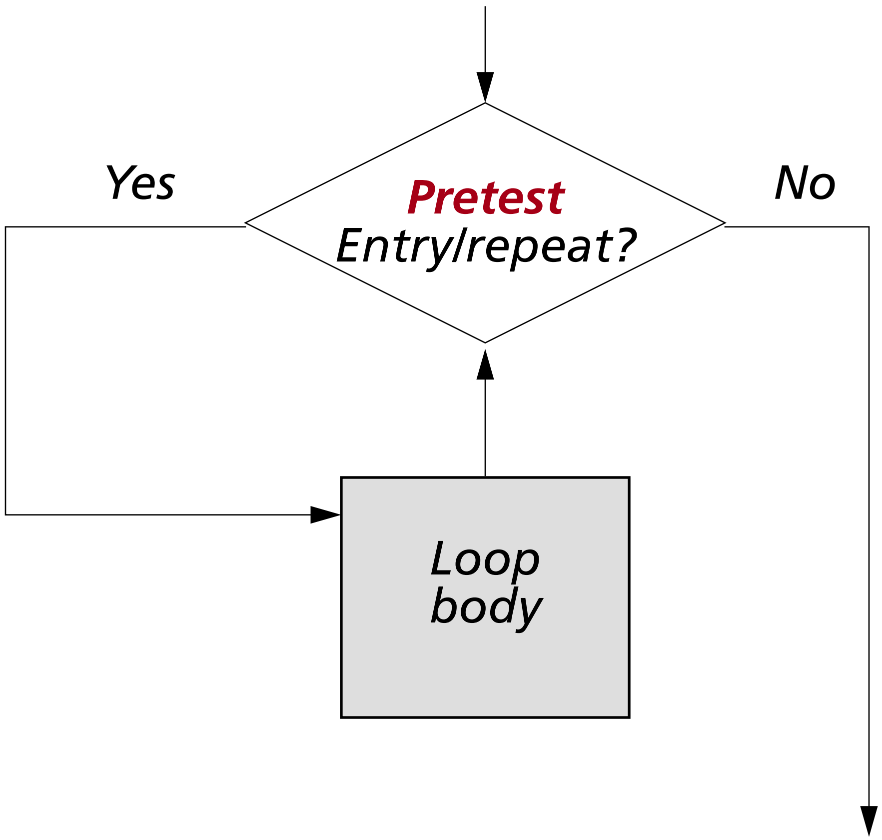
  Stay in the loop if this condition is true.

- **Exit** or **termination** condition:

  Exit the loop if this condition is true.

9.3.2 Three General Loop Patterns

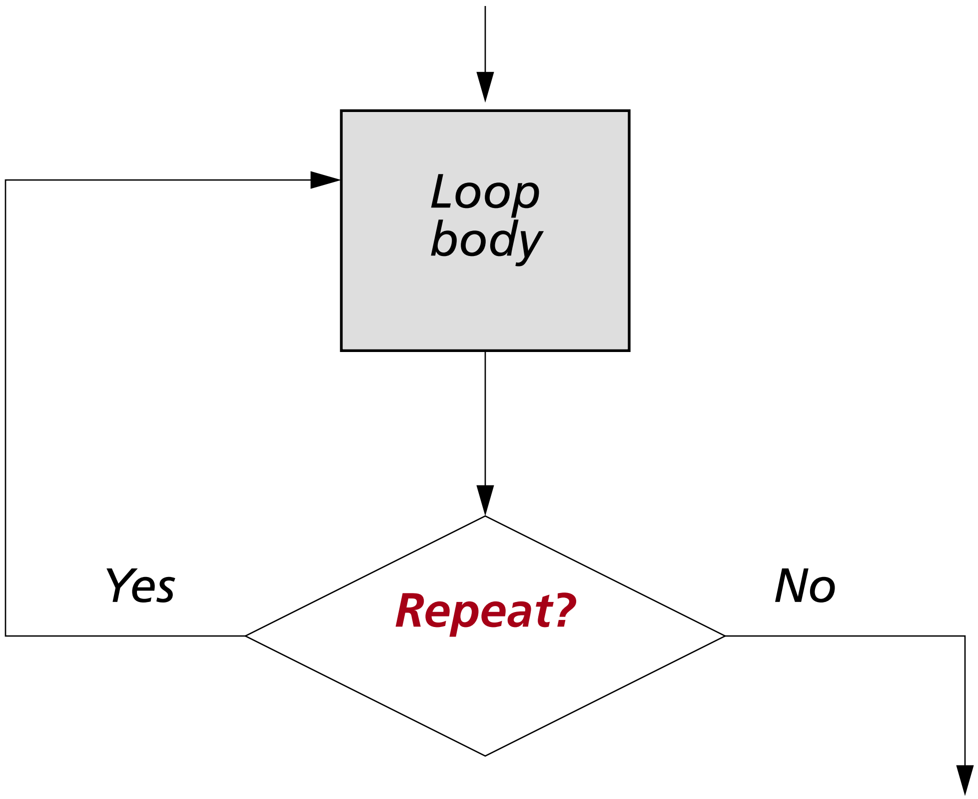
Pretest loop with continuation condition



Examples:

```
for (n=0; n<10; n++)
   { ... }

n = 0;

while ( n < 10 )
   { ... n++; ... }
```

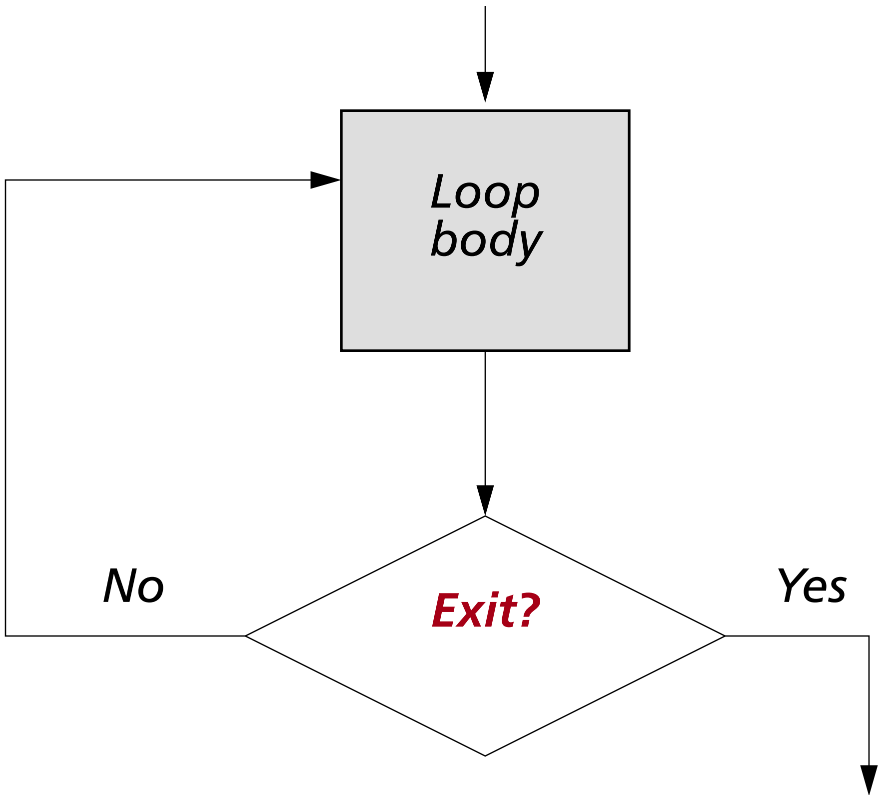
Post-test loop with continuation condition



Example:

```
n = 0;

do {
    ... n++; ...
} while ( n < 10 );
```

Post-test loop with exit condition



Example:

```
n = 0;

while (true)
{
   ... n++; ...

   if (n>=10) break;
}
```

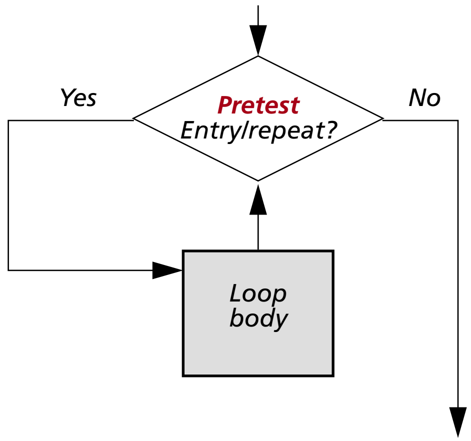
9.3.3 The *for* Loop — Fixed Repetition

- Repeating a single statement

```
for(entry; exit_test; in_de_crement)
   statement;
```

- Repeating sequences of statements

```
for(entry; exit_test; in_de_crement)
{
   statement_sequence
}
```



Generally, **for** loops are count-controlled.

Example: Calculating Fibonacci numbers:

$f_0 = 0$

$f_1 = 1$

$f_n = f_{n-1} + f_{n-2}$

A few Fibonacci numbers, calculated iteratively:

$f_2 = f_1 + f_0 = 1 + 0 = 1$

$f_3 = f_2 + f_1 = 1 + 1 = 2$

$f_4 = f_3 + f_2 = 2 + 1 = 3$

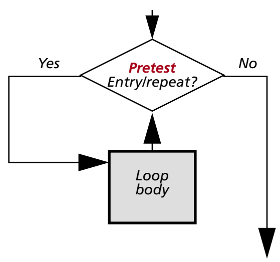
$f_5 = f_4 + f_3 = 3 + 2 = 5$

$f_6 = f_5 + f_4 = 5 + 3 = 8$

$f_7 = f_6 + f_5 = 8 + 5 = 13$

```
/* Calculating the n-th Fibonacci number

   Basic idea to calculate the n-th
   Fibonacci number:

   next_fib = current_fib + previous_fib;
*/

int prev_fib = 0;         // = f_{n-2}
int current_fib  = 1;     // = f_{n-1}
int next_fib;             // = f_n
int n, i;
```

```
main()
{
  cout << “Fibonacci numbers” << endl;
  cout << “Which F. number would you “;
  cout << “to calculate?” << endl;

  cout << “Enter an integer n >= 0: “;
  cin >> n;

  for(i = 0; i < n; i++){
    next_fib = current_fib + prev_fib;
    prev_fib = current_fib;
    current_fib = next_fib;
  }
  cout << n << “-th Fibonacci = “;
  cout << prev_fib;
}
```



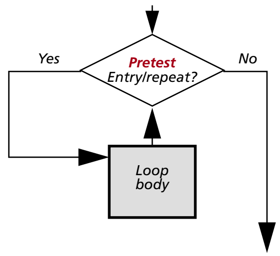
9.3.4 Variations on the *for* Loop

- Several initialization and increment expressions

```
for(x=0, y=10; x<=10; ++x, --y)
  cout << x << ‘ ‘ << y << ‘\n’;
```

- Exiting a **`for`** loop when a key is pressed (using the **`kbhit()`** function)

```
int main()
{
   int i;

   // print numbers until a key is pressed
   for(i=0; !kbhit(); i++) cout << i << ‘ ‘;

   return(0);
}
```



`kbhit()` returns `true` (!= 0)

- if a key has been pressed
- otherwise `false` (== 0).

9.3.5 The *while* Loop — Pretest

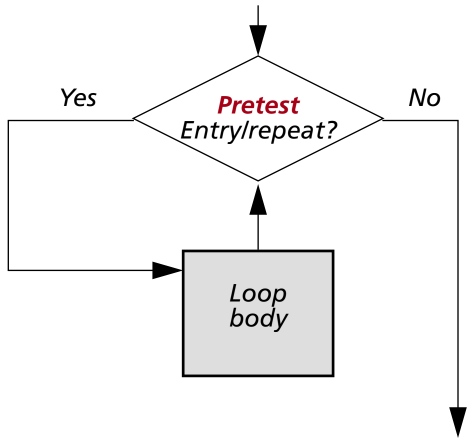
- Single statement to repeat

```
while(expression) statement;
```

- Sequence of statements to repeat

```
while(expression)
{
  statement_sequence
}
```

Generally, *while* loops are event-controlled.



Example: The *n*-th Fibonacci number :

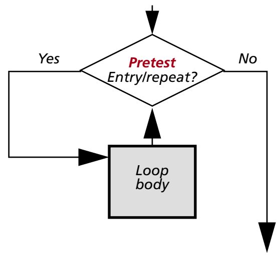
$f_0 = 0$

$f_1 = 1$

$f_n = f_{n-1} + f_{n-2}$

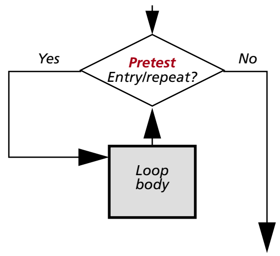
Implemented with a **WHILE** loop:

```
int previous_fib = 0;  // = f_{n-2}
int current_fib  = 1;  // = f_{n-1}
int next_fib;          // = f_n
int n, i = 0;
```




```
main()
{
  cout << “Enter n >= 0: “;
  cin >> n;

  while(i < n) {
    next_fib = current_fib + previous_fib;
    previous_fib = current_fib;
    current_fib = next_fib;
    i++;
  }

  cout << previous_fib << “ is “;
  cout << n << “-th fib;

  return(0);
}
```

```
main() // a little more efficient code
{
  cout << "Enter n >= 0: ";
  cin >> n;

  while(i++ < n) {
    next_fib = current_fib + previous_fib;
    previous_fib = current_fib;
    current_fib = next_fib;
  }

  cout << previous_fib << " is ";
  cout << n << "-th fib;

  return(0);
}
```



Taking care of special cases (1):

```
// variable declarations go here

main()
{
  cout << “Enter n >= 0: “; cin >> n;

  if(n==0 || n==1) {
    cout << n;
    cout << “is the first fib >= “ << n;
    return(0);
  }

  while(...) {...}
  ...
  return(0);
}
```

Taking care of special cases (1):

```
// variable declarations go here

main()
{
  cout << “Enter n >= 0: “; cin >> n;

  if(n <= 1) { // Special cases: n = 0 or 1
    cout << n;
    cout << “is the first fib >= “ << n;
  }
  else {
    while(...) {...}
    ...
  }
  return(0);
}
```

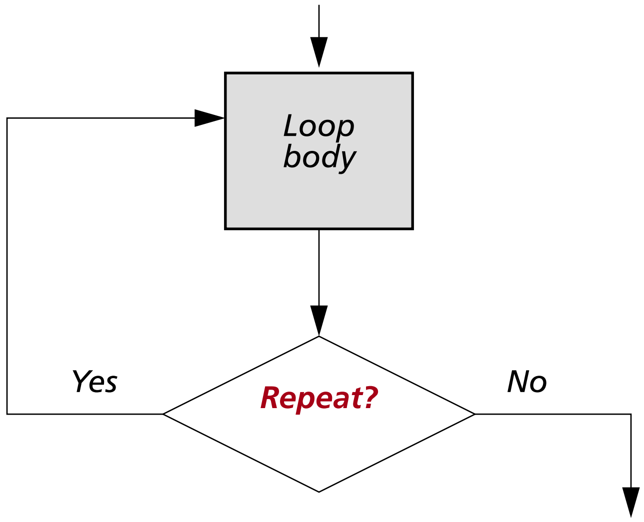
9.3.6 The *do-while* Loop — A Post-Test Loop

- Single statement to repeat

```
do statement; while(expression);
```

- Sequence of statements to repeat

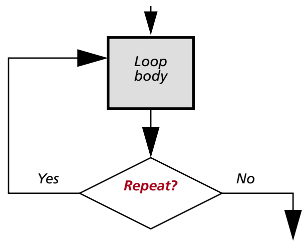
```
do{
  statements
} while(expression);
```

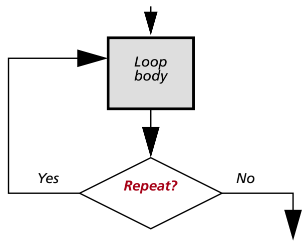


Note: A *do-while* loop always completes one iteration!

Example: The *n*-th Fibonacci number (with DO-WHILE loop)

```
// Other initializations go here
int n, i=0;

void main()
{
  cout << “Enter n > 0: “; cin >> n;

  do {
    next_fib = current_fib + previous_fib;
    previous_fib = current_fib;
    current_fib = next_fib;
  } while(++i < n);

  cout << previous_fib << “ is “;
  cout << n << “-th fib. number”;}
```






```
void main()
{
  cout << “Enter n > 0: “; cin >> n;

  do {
    next_fib = current_fib + previous_fib;
    previous_fib = current_fib;
    current_fib = next_fib;
  } while(++i < n);

  cout << (n==0) ? 0 : previous_fib;
  cout << “ is “;
  cout << n << “-th fib. number”;
}
```


9.3.7 Infinite Loops

- Using **`for`**

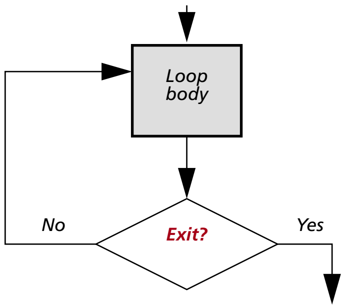
```
for(;;) { … }
```

- Using **`while`**

```
while(1) { … }
```

Checking for keyboard input:

```
while (true) {
  cout << “Continue with program? (y/n)\n”;
  cin >> answer;

  switch(answer) {
    case ‘y’:
    case ‘Y’: break; // program continued
    case ‘n’:
    case ‘N’: cout << “Program end.\n”;
              return(0);
    default:
      cout << “Enter only \’y\’ or \’n\’.”;
  }
}
// further statements of program
```



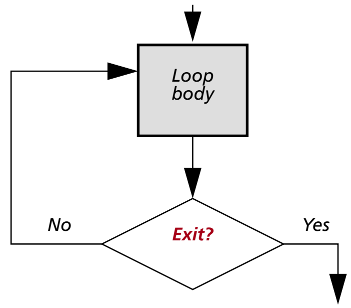
9.4 Break and Continue

9.4.1 Using *break* to Exit Loops

```
for(i=0; i<1000; i++) // for a long time
{
  // do something
  if(kbhit()) break;
}
```

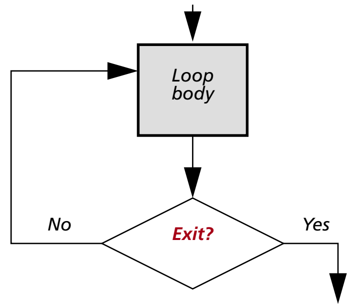
Alternative with infinite for `loop`:

```
for(;;){                  // infinite loop
  // do something
  if(kbhit()) break;
}
```



Alternative with infinite `while` loop:

```
while(1){                 // infinite loop
  // do something
  if(kbhit()) break;
}
```



Using *break* to exit loops

```
int main()
{
  int t, count;

  for(t = 0; t < 100; t++) {
    count = 1;

    for(;;) {
      cout << count << ‘ ‘;
      count++;
      if(count == 10) break;
    }
    cout << ‘\n’;

    return(0);
}
```

9.4.2 Using *continue*

Continue is used to bypass a loop´s normal control structure

```
int main()
{
  int x;

  for(x=0; x<=100; x++)
  {
    if(x % 2) continue;
    cout << x << ‘ ‘;
  }

  return(0);
}
```

9.5 Using goto — “Spaghetti Programming”

The *goto* requires a label for operation. A **label** is a valid C++ identifier followed by a colon.

A loop from 1 to 100 could be written using *goto* as follows:

```
x = 1;
start:
  x++;
  statement_sequence
  if(x<100) goto start;
```

However, a much more comprehensive formulation is:

```
for(x=1; x<100; x++){ statement_sequence }
```

9.6 Guidelines for Loops

9.6.1 How to Design Loops

Process:

- What is the process being **repeated**?
- How should the process be **initialized**?
- How should the process be **updated**?

Condition:

- How should the condition be **initialized**?
- How should the condition be **updated**?
- What is the condition that **ends** the loop?

After the Loop:

- What is the **state of the program** on exiting the loop?

9.6.2 Guidelines for Choosing a Looping Statement

- If the repeated process is a simple **count-controlled loop**, the **for** loop is a “natural” choice:

```
for (count = 1; count <= 10; count++)
  // statement;
```

… is equivalent to …

```
count = 1;
while (count <= 10)
{
  // statement;
  count++;
}
```

Concentrating the three loop control actions (initialize, test, and increment/decrement) in the **for** loop in one place reduces the chances of errors.

- If the iterated process is an **event-controlled loop**, whose body always has to be executed at least once, a **do-while** loop is appropriate.

- If the iterated process is an **event-controlled loop**, but nothing is known about the first execution, use a **while** loop.

- An infinite loop with **break** statements sometimes clarifies the code.

  More often, however, it reflects an undisciplined loop design.

  Use it only after careful consideration of **while**, **do-while**, and **for**.

9.7 References

- G. Blank and R. Barnes, *The Universal Machine*, Boston, MA: WCB/ McGraw-Hill, 1998. Chapter 7.