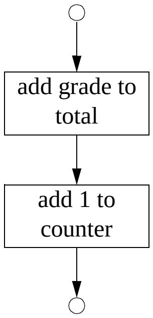
Illustrations List *(Main Page)*



```
total = total + grade;
counter = counter + 1;
```

Fig. 2.1 Flowcharting C++'s sequence structure.

C++ Keywords				
C and C++ keywords				
`auto`	`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extern`	`float`	`for`	`goto`
`if`	`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`	`struct`
`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`			
C++ only keywords				
`asm`	`bool`	`catch`	`class`	`const_cast`
`delete`	`dynamic_cast`	`explicit`	`false`	`friend`
`inline`	`mutable`	`namespace`	`new`	`operator`
`private`	`protected`	`public`	`reinterpret_cast`	
`static_cast`	`template`	`this`	`throw`	`true`
`try`	`typeid`	`typename`	`using`	`virtual`
`wchar_t`				

Fig. 2.2 C++ keywords.

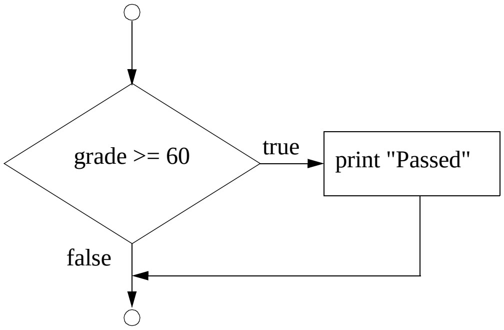


Fig. 2.3 Flowcharting the single-selection `if` structure.

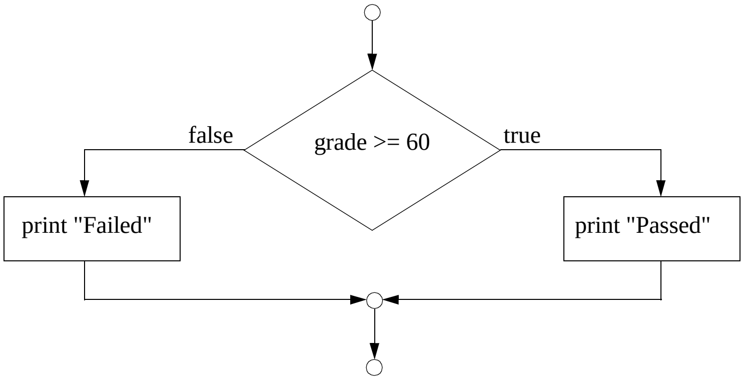


Fig. 2.4 Flowcharting the double-selection `if/else` structure.

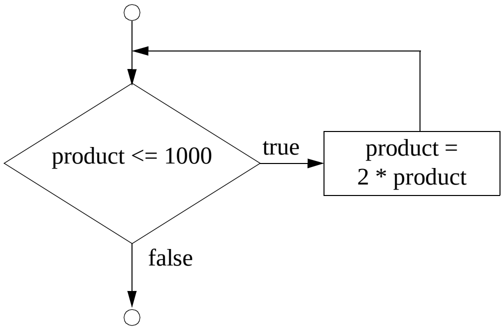


Fig. 2.5 Flowcharting the `while` repetition structure.

```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Fig. 2.6 Pseudocode algorithm that uses counter-controlled repetition to solve the class average problem.

```cpp
// Fig. 2.7: fig02_07.cpp
// Class average program with counter-controlled repetition
#include <iostream.h>

int main()
{
   int total,        // sum of grades
       gradeCounter, // number of grades entered
       grade,        // one grade
       average;      // average of grades

   // initialization phase
   total = 0;                           // clear total
   gradeCounter = 1;                    // prepare to loop

   // processing phase
   while ( gradeCounter <= 10 ) {       // loop 10 times
      cout << "Enter grade: ";          // prompt for input
      cin >> grade;                     // input grade
      total = total + grade;            // add grade to total
      gradeCounter = gradeCounter + 1;  // increment counter
   }

   // termination phase
   average = total / 10;                // integer division
   cout << "Class average is " << average << endl;

   return 0;   // indicate program ended successfully
}
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Fig. 2.7 C++ program and sample execution for the class average problem with counter-controlled repetition.

```
Initialize total to zero
Initialize counter to zero

Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)

If the counter is not equal to zero
    Set the average to the total divided by the counter
    Print the average
else
    Print “No grades were entered”
```

Fig. 2.8 Pseudocode algorithm that uses sentinel-controlled repetition to solve the class average problem.

```cpp
// Fig. 2.9: fig02_09.cpp
// Class average program with sentinel-controlled repetition.
#include <iostream.h>
#include <iomanip.h>

int main()
{
   int total,        // sum of grades
       gradeCounter, // number of grades entered
       grade;        // one grade
   float average;    // number with decimal point for average

```

Fig. 2.9 C++ program and sample execution for the class average problem with sentinel-controlled repetition (part 1 of 2).

```cpp
   // initialization phase
   total = 0;
   gradeCounter = 0;

   // processing phase
   cout << "Enter grade, -1 to end: ";
   cin >> grade;

   while ( grade != -1 ) {
      total = total + grade;
      gradeCounter = gradeCounter + 1;
      cout << "Enter grade, -1 to end: ";
      cin >> grade;
   }

   // termination phase
   if ( gradeCounter != 0 ) {
      average = static_cast< float >( total ) / gradeCounter;
      cout << "Class average is " << setprecision( 2 )
           << setiosflags( ios::fixed | ios::showpoint )
           << average << endl;
   }
   else
      cout << "No grades were entered" << endl;

   return 0;   // indicate program ended successfully
}
```

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

Fig. 2.9 C++ program and sample execution for the class average problem with sentinel-controlled repetition (part 2 of 2).

```
Initialize passes to zero
Initialize failures to zero
Initialize student counter to one

While student counter is less than or equal to ten
    Input the next exam result

    If the student passed
        Add one to passes
    else
        Add one to failures

    Add one to student counter

Print the number of passes
Print the number of failures

If more than eight students passed
    Print “Raise tuition”
```

Fig. 2.10 Pseudocode for examination results problem.

```cpp
// Fig. 2.11: fig02_11.cpp
// Analysis of examination results
#include <iostream.h>

int main()
{
   // initialize variables in declarations
   int passes = 0,           // number of passes
       failures = 0,         // number of failures
       studentCounter = 1,   // student counter
       result;               // one exam result

   // process 10 students; counter-controlled loop
   while ( studentCounter <= 10 ) {
      cout << "Enter result (1=pass,2=fail): ";
      cin >> result;

```

Fig. 2.11 C++ program and sample executions for examination results problem (part 1 of 2).

```cpp
      if ( result == 1 )        // if/else nested in while
         passes = passes + 1;
      else
         failures = failures + 1;

      studentCounter = studentCounter + 1;
   }

   // termination phase
   cout << "Passed " << passes << endl;
   cout << "Failed " << failures << endl;

   if ( passes > 8 )
      cout << "Raise tuition " << endl;

   return 0;   // successful termination
}
```

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Passed 6
Failed 4
```

```
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 2
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Enter result (1=pass,2=fail): 1
Passed 9
Failed 1
Raise tuition
```

Fig. 2.11 C++ program and sample executions for examination results problem (part 2 of 2).

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	`10` to `c`
`-=`	`d -= 4`	`d = d - 4`	`1` to `d`
`*=`	`e *= 5`	`e = e * 5`	`20` to `e`
`/=`	`f /= 3`	`f = f / 3`	`2` to `f`
`%=`	`g %= 9`	`g = g % 9`	`3` to `g`

Fig. 2.12 Arithmetic assignment operators.

Operator	Called	Sample expression	Explanation
`++`	preincrement	**`++a`**	Increment **`a`** by 1, then use the new value of **`a`** in the expression in which **`a`** resides.
`++`	postincrement	**`a++`**	Use the current value of **`a`** in the expression in which **`a`** resides, then increment **`a`** by 1.
`--`	predecrement	**`--b`**	Decrement **`b`** by 1, then use the new value of **`b`** in the expression in which **`b`** resides.
`--`	postdecrement	**`b--`**	Use the current value of **`b`** in the expression in which **`b`** resides, then decrement **`b`** by 1.

Fig. 2.13 The increment and decrement operators.

```
// Fig. 2.14: fig02_14.cpp
// Preincrementing and postincrementing
#include <iostream.h>

int main()
{
   int c;

   c = 5;
   cout << c << endl;           // print 5
   cout << c++ << endl;         // print 5 then postincrement
   cout << c << endl << endl; // print 6

   c = 5;
   cout << c << endl;           // print 5
   cout << ++c << endl;         // preincrement then print 6
   cout << c << endl;           // print 6

   return 0;                    // successful termination
}
```

```
5
5
6

5
6
6
```

Fig. 2.14 The difference between preincrementing and postincrementing.

Operators	Associativity	Type
()	left to right	parentheses
++ -- + - static_cast<*type*>()	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
<< >>	left to right	insertion/extraction
< <= > >=	left to right	relational
== !=	left to right	equality
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 2.15 Precedence of the operators encountered so far in the text.

```
// Fig. 2.16: fig02_16.cpp
// Counter-controlled repetition
#include <iostream.h>

int main()
{
   int counter = 1;             // initialization

   while ( counter <= 10 ) {    // repetition condition
      cout << counter << endl;
      ++counter;                // increment
   }

   return 0;
}
```

Fig. 2.16 Counter-controlled repetition.

```
1
2
3
4
5
6
7
8
9
10
```

```
// Fig. 2.17: fig02_17.cpp
// Counter-controlled repetition with the for structure
#include <iostream.h>

int main()
{
   // Initialization, repetition condition, and incrementing
   // are all included in the for structure header.

   for ( int counter = 1; counter <= 10; counter++ )
      cout << counter << endl;

   return 0;
}
```

Fig. 2.17 Counter-controlled repetition with the `for` structure.

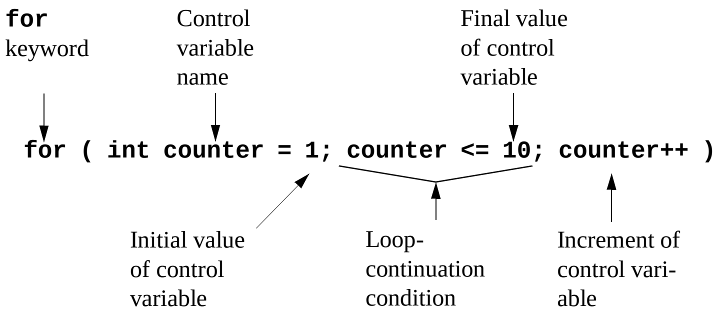

for keyword
Control variable name
Final value of control variable
for (int counter = 1; counter <= 10; counter++)
Initial value of control variable
Loop-continuation condition
Increment of control variable

Fig. 2.18 Components of a typical `for` header.

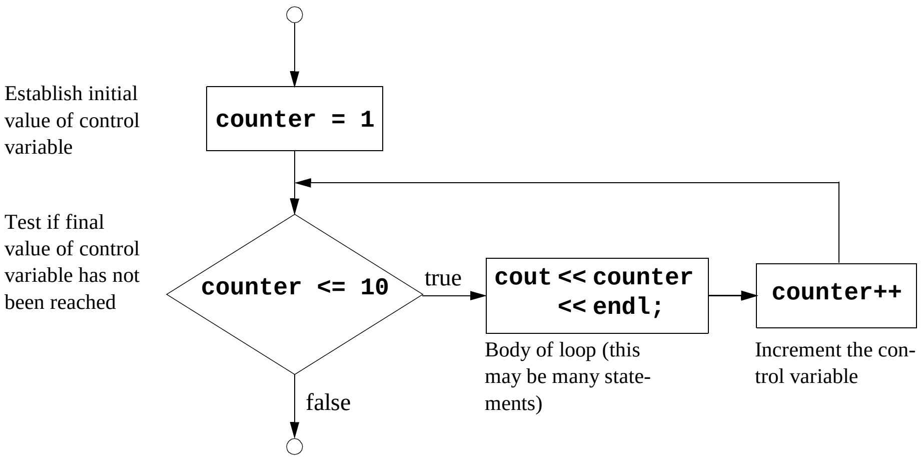

Establish initial value of control variable
counter = 1
Test if final value of control variable has not been reached
counter <= 10
true
false
cout << counter << endl;
Body of loop (this may be many statements)
counter++
Increment the control variable

Fig. 2.19 Flowcharting a typical `for` repetition structure.

```
// Fig. 2.20: fig02_20.cpp
// Summation with for
#include <iostream.h>

int main()
{
   int sum = 0;

   for ( int number = 2; number <= 100; number += 2 )
      sum += number;

   cout << "Sum is " << sum << endl;

   return 0;
}
```

```
Sum is 2550
```

Fig. 2.20 Summation with `for`.

```
// Fig. 2.21: fig02_21.cpp
// Calculating compound interest
#include <iostream.h>
#include <iomanip.h>
#include <math.h>

int main()
{
   double amount,              // amount on deposit
          principal = 1000.0,  // starting principal
          rate = .05;          // interest rate

   cout << "Year" << setw( 21 )
        << "Amount on deposit" << endl;

   for ( int year = 1; year <= 10; year++ ) {
      amount = principal * pow( 1.0 + rate, year );
      cout << setw( 4 ) << year
           << setiosflags( ios::fixed | ios::showpoint )
           << setw( 21 ) << setprecision( 2 )
           << amount << endl;
   }

   return 0;
}
```

Fig. 2.21 Calculating compound interest with `for` (part 1 of 2).

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.62
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

Fig. 2.21 Calculating compound interest with `for` (part 2 of 2).

```
// Fig. 2.22: fig02_22.cpp
// Counting letter grades
#include <iostream.h>

int main()
{
   int grade,       // one grade
       aCount = 0,  // number of A's
       bCount = 0,  // number of B's
       cCount = 0,  // number of C's
       dCount = 0,  // number of D's
       fCount = 0;  // number of F's

   cout << "Enter the letter grades." << endl
        << "Enter the EOF character to end input." << endl;

   while ( ( grade = cin.get() ) != EOF ) {

      switch ( grade ) {      // switch nested in while

         case 'A':  // grade was uppercase A
         case 'a':  // or lowercase a
            ++aCount;
            break;  // necessary to exit switch

         case 'B':  // grade was uppercase B
         case 'b':  // or lowercase b
            ++bCount;
            break;

         case 'C':  // grade was uppercase C
         case 'c':  // or lowercase c
            ++cCount;
            break;

         case 'D':  // grade was uppercase D
         case 'd':  // or lowercase d
            ++dCount;
            break;

         case 'F':  // grade was uppercase F
         case 'f':  // or lowercase f
            ++fCount;
            break;

         case '\n': // ignore newlines,
         case '\t': // tabs,
         case ' ':  // and spaces in input
            break;

```

Fig. 2.22 An example using `switch` (part 1 of 2).

```
         default:   // catch all other characters
            cout << "Incorrect letter grade entered."
                 << " Enter a new grade." << endl;
            break;  // optional
      }
   }

   cout << "\n\nTotals for each letter grade are:"
        << "\nA: " << aCount
        << "\nB: " << bCount
        << "\nC: " << cCount
        << "\nD: " << dCount
        << "\nF: " << fCount << endl;

   return 0;
}
```

```
Enter the letter grades.
Enter the EOF character to end input.
a
B
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Fig. 2.22 An example using `switch` (part 2 of 2).

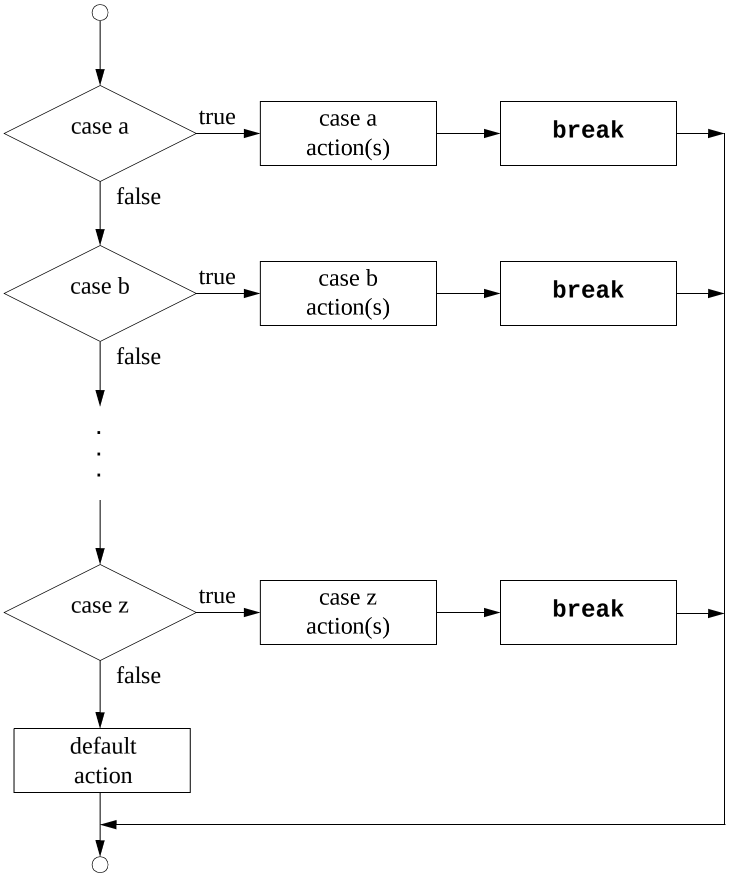


Fig. 2.23 The `switch` multiple-selection structure with `break`s.

```
// Fig. 2.24: fig02_24.cpp
// Using the do/while repetition structure
#include <iostream.h>

int main()
{
   int counter = 1;

   do {
      cout << counter << "  ";
   } while ( ++counter <= 10 );

   cout << endl;

   return 0;
}
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 2.24 Using the `do/while` structure.

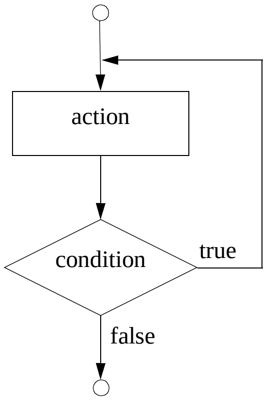


Fig. 2.25 Flowcharting the **`do/while`** repetition structure.

```
// Fig. 2.26: fig02_26.cpp
// Using the break statement in a for structure
#include <iostream.h>

int main()
{
   // x declared here so it can be used after the loop
   int x;

   for ( x = 1; x <= 10; x++ ) {

      if ( x == 5 )
         break;    // break loop only if x is 5

      cout << x << " ";
   }

   cout << "\nBroke out of loop at x of " << x << endl;
   return 0;
}
```

Fig. 2.26 Using the **`break`** statement in a **`for`** structure (part 1 of 2).

```
1 2 3 4
Broke out of loop at x of 5
```

Fig. 2.26 Using the **`break`** statement in a **`for`** structure (part 2 of 2).

```cpp
// Fig. 2.27: fig02_07.cpp
// Using the continue statement in a for structure
#include <iostream.h>

int main()
{
   for ( int x = 1; x <= 10; x++ ) {

      if ( x == 5 )
         continue;  // skip remaining code in loop
                    // only if x is 5

      cout << x << " ";
   }

   cout << "\nUsed continue to skip printing the value 5"
        << endl;
   return 0;
}
```

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Fig. 2.27 Using the `continue` statement in a `for` structure.

expression1	expression2	expression1 && expression2
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

Fig. 2.28 Truth table for the `&&` (logical AND) operator.

expression1	expression2	expression1 || expression2
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Fig. 2.29 Truth table for the `||` (logical OR) operator.

expression	`!`expression
`false`	`true`
`true`	`false`

Fig. 2.30 Truth table for operator `!` (logical negation).

Operators	Associativity	Type
`()`	left to right	parentheses
`++ -- + - ! static_cast<`*type*`>()`	right to left	unary
`* / %`	left to right	multiplicative
`+ -`	left to right	additive
`<< >>`	left to right	insertion/extraction
`< <= > >=`	left to right	relational
`== !=`	left to right	equality
`&&`	left to right	logical AND
`||`	left to right	logical OR
`?:`	right to left	conditional
`= += -= *= /= %=`	right to left	assignment
`,`	left to right	comma

Fig. 2.31 Operator precedence and associativity.

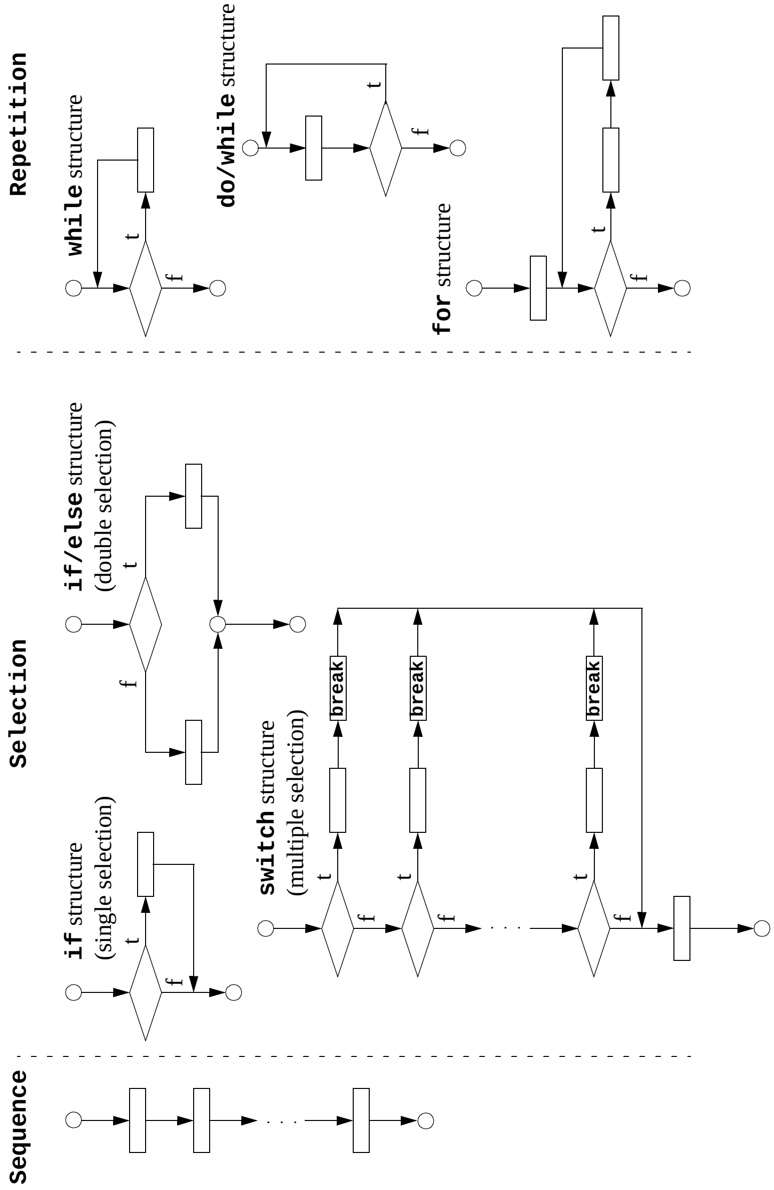


Fig. 2.32 C++'s single-entry/single-exit sequence, selection, and repetition structures.

Rules for Forming Structured Programs

1) Begin with the “simplest flowchart” (Fig. 2.34).
2) Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
3) Any rectangle (action) can be replaced by any control structure (sequence, **`if`**, **`if/else`**, **`switch`**, **`while`**, **`do/while`**, or **`for`**).
4) Rules 2 and 3 may be applied as often as you like and in any order.

Fig. 2.33 Rules for forming structured programs.

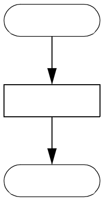


Fig. 2.34 The simplest flowchart.

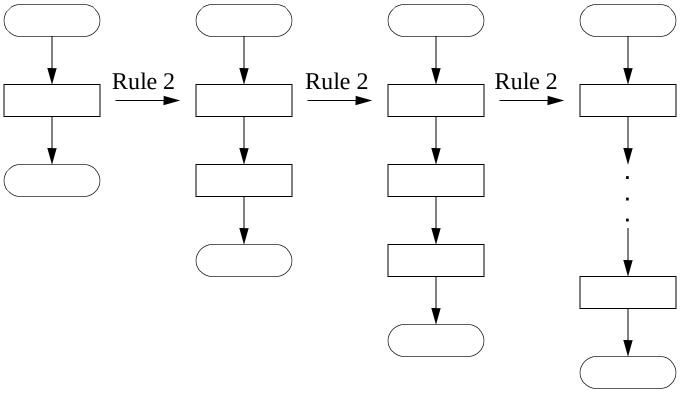


Fig. 2.35 Repeatedly applying rule 2 of Fig. 2.33 to the simplest flowchart.

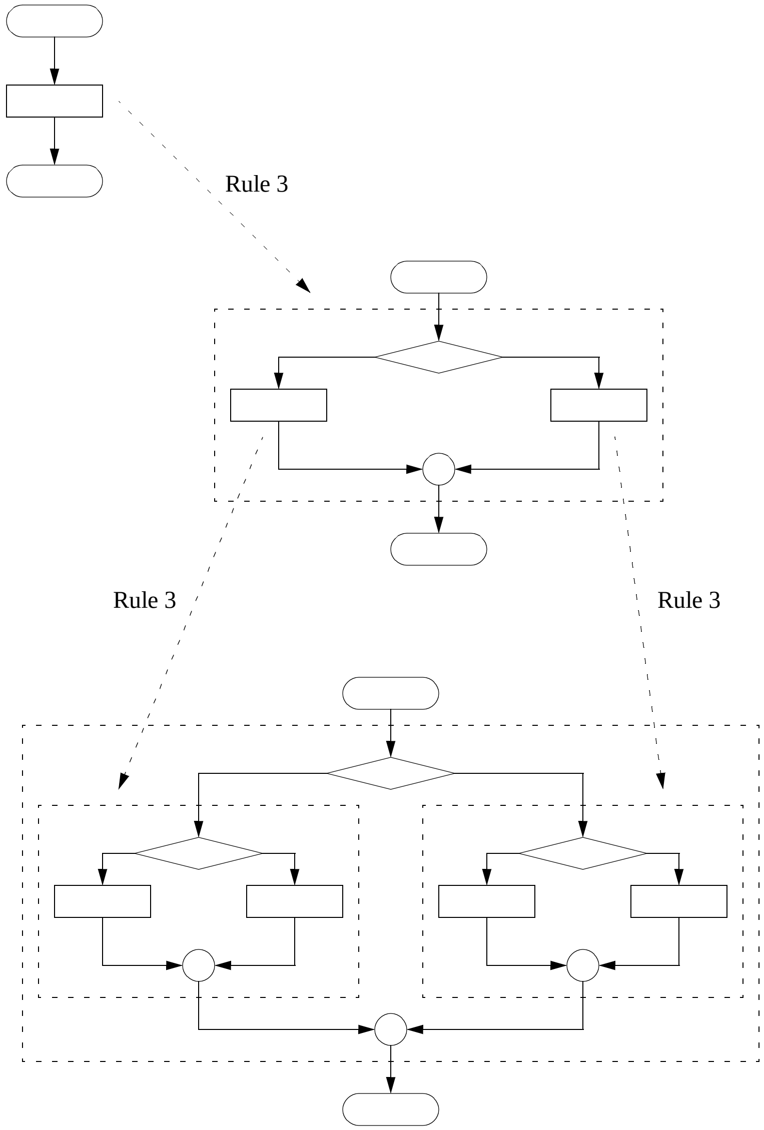


Fig. 2.36 Applying rule 3 of Fig. 2.33 to the simplest flowchart.

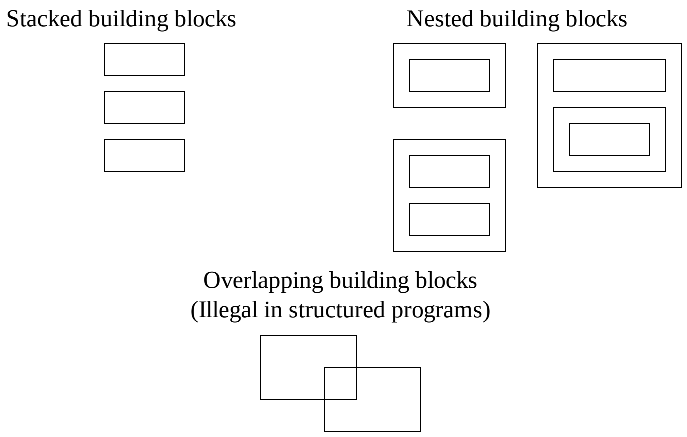


Fig. 2.37 Stacked, nested, and overlapped building blocks.

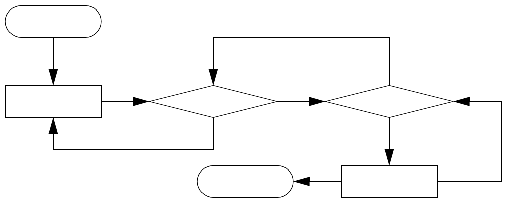


Fig. 2.38 An unstructured flowchart.