Programming for Engineers

Iteration

ICEN 200– Spring 2018

Prof. Dola Saha

Data type conversions

- Grade average example
  - $class\ average = \frac{\sum grade}{number\ of\ students}$
  - Grade and number of students can be integers
  - Averages do not always evaluate to integer values, needs to be floating point for accuracy.
  - The result of the calculation `total / counter` is an integer because `total` and `counter` are both integer variables.

Explicit conversions

- Dividing two integers results in integer division in which any fractional part of the calculation is truncated (i.e., lost).
- To produce a floating-point calculation with integer values, we create temporary values that are floating-point numbers.
- C provides the unary cast operator to accomplish this task.
  - `average = ( float ) total / counter;`
- includes the cast operator `(float)`, which creates a temporary floating-point copy of its operand, `total`.
- Using a cast operator in this manner is called explicit conversion.
- The calculation now consists of a floating-point value (the temporary `float` version of `total`) divided by the `unsigned int` value stored in `counter`.

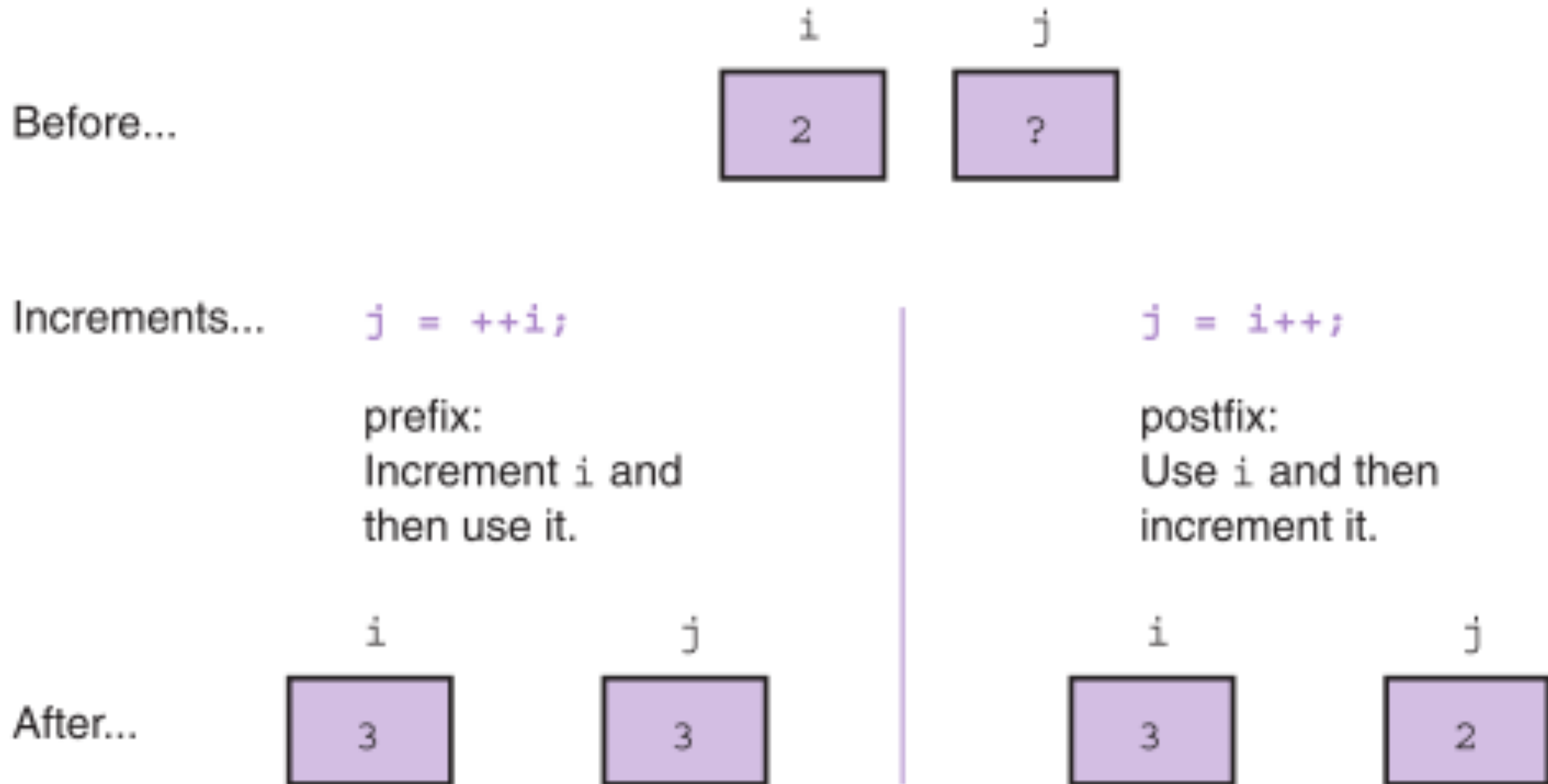
Implicit conversion

- C evaluates arithmetic expressions only in which the data types of the operands are *identical*.
- To ensure that the operands are of the *same* type, the compiler performs an operation called implicit conversion on selected operands.
- For example, in an expression containing the data types `unsigned int` and `float`, copies of `unsigned int` operands are made and converted to `float`.
- In our example, after a copy of `counter` is made and converted to `float`, the calculation is performed and the result of the floating-point division is assigned to `average`.

Assignment operators

- C provides several assignment operators for abbreviating assignment expressions.
- For example, the statement
  - `c = c + 3;`
- can be abbreviated with the addition assignment operator += as
  - `c += 3;`
- The += operator
  - adds the value of the expression on the right of the operator to the value of the variable on the left of the operator
  - and stores the result in the variable on the left of the operator.

Comparison of Prefix & Postfix Increments



Assignment operators

- Any statement of the form
  - *variable = variable operator expression;*
- where *operator is one of the binary operators +, -, *, / or %, can be written in the form*
  - *variable operator = expression;*
- Thus the assignment c += 3 adds 3 to c.

Assignment operator - examples

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

Unary Increment & Decrement Operators

Operator	Sample expression	Explanation
++	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

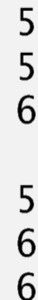
Increment Example

```
1   // Fig. 3.13: fig03_13.c
2   // Preincrementing and postincrementing.
3   #include <stdio.h>
4
5   // function main begins program execution
6   int main( void )
7   {
8      int c; // define variable
9
10     // demonstrate postincrement
11     c = 5; // assign 5 to c
12     printf( "%d\n", c ); // print 5
13     printf( "%d\n", c++ ); // print 5 then postincrement
14     printf( "%d\n\n", c ); // print 6
15
16     // demonstrate preincrement
17     c = 5; // assign 5 to c
18     printf( "%d\n", c ); // print 5
19     printf( "%d\n", ++c ); // preincrement then print 6
20     printf( "%d\n", c ); // print 6
21  } // end function main
```

Increment Example

```
1  // Fig. 3.13: fig03_13.c
2  // Preincrementing and postincrementing.
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main( void )
7  {
8     int c; // define variable
9
10    // demonstrate postincrement
11    c = 5; // assign 5 to c
12    printf( "%d\n", c ); // print 5
13    printf( "%d\n", c++ ); // print 5 then postincrement
14    printf( "%d\n\n", c ); // print 6
15
16    // demonstrate preincrement
17    c = 5; // assign 5 to c
18    printf( "%d\n", c ); // print 5
19    printf( "%d\n", ++c ); // preincrement then print 6
20    printf( "%d\n", c ); // print 6
21 } // end function main
```

Output



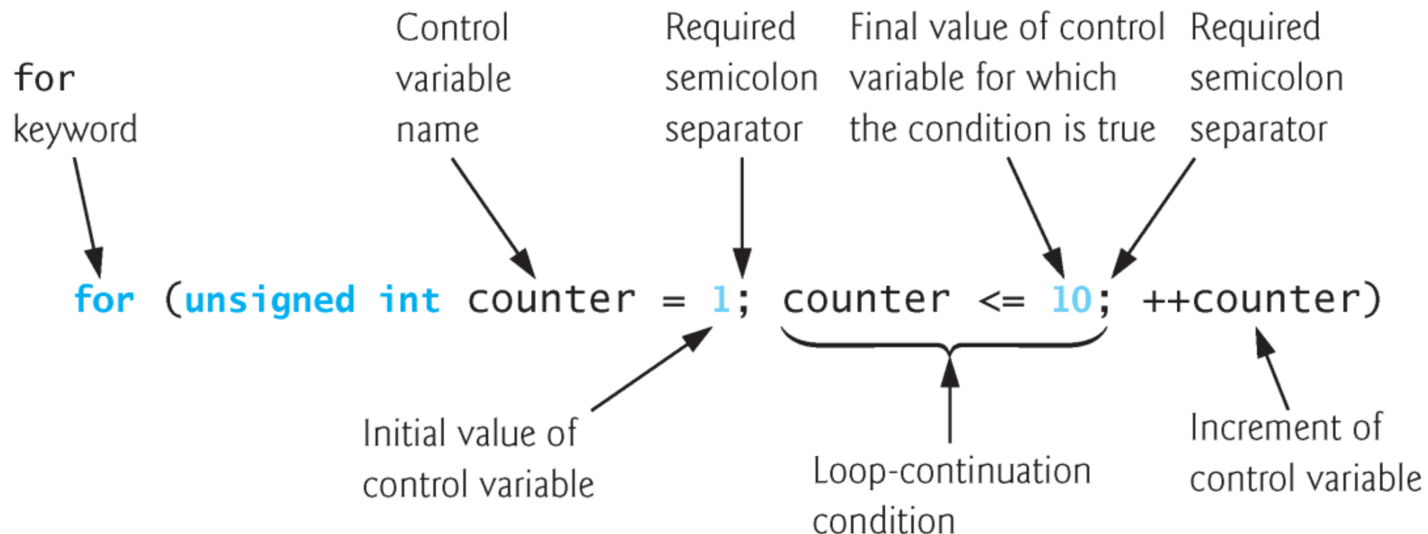
Precedence

Operators	Associativity	Type
++ *(postfix)* -- *(postfix)*	right to left	postfix
+ - *(type)* ++ *(prefix)* -- *(prefix)*	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

for Iteration Statement - Syntax

```
for (initialization expression;
        loop repetition condition;
        update expression)
      statement;
```

```
for  (count_star = 0;
        count_star < N;
        count_star ++)
     printf(“*”);
```



`for` Iteration Statement - Syntax

- The general format of the `for` statement is

  for (***initialization; condition; update expression***) {
  statement
  }

  where

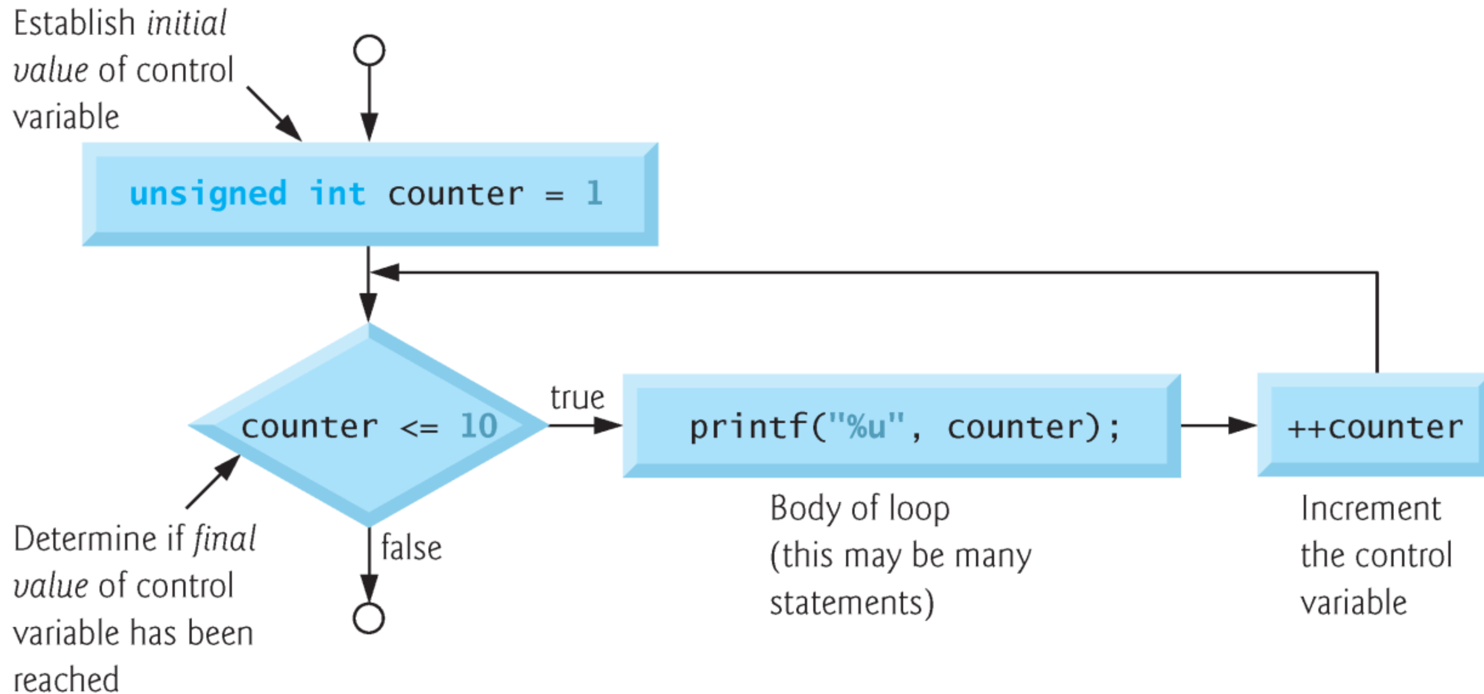
  - the initialization expression initializes the loop-control variable (and might define it),
  - the condition expression is the loop-continuation condition and
  - the update expression increments the control variable.

for Iteration Statement

- Counter-controlled iteration

```
1   // Fig. 4.2: fig04_02.c
2   // Counter-controlled iteration with the for statement.
3   #include <stdio.h>
4
5   int main(void)
6   {
7      // initialization, iteration condition, and increment
8      //  are all included in the for statement header.
9      for (unsigned int counter = 1; counter <= 10; ++counter) {
10        printf("%u\n", counter);
11     }
12  }
```

Flow chart



`for` Iteration Statement – Common Error

Off-By-One Errors

- Notice that program uses the loop-continuation condition `counter <= 10`.
- If you incorrectly wrote `counter < 10`, then the loop would be executed only 9 times.
- This is a common logic error called an off-by-one error.

for Iteration Statement – Common Practice

- Start the loop from 0

```
for (i=0; i<10; i++)
     printf("It will be printed 10 times.\n");
for (i=1; i<=10; i++)
     printf("It will be printed 10 times.\n");
```

Optional Header in `for` Statement

- The three expressions in the `for` statement are optional.
- If the *condition* expression is omitted, C assumes that the condition is true, thus creating an infinite loop.
- You may omit the *initialization* expression if the control variable is initialized elsewhere in the program.
- The *increment* may be omitted if it's calculated by statements in the body of the `for` statement or if no increment is needed.

Valid code snippets

```
for (;;)
      printf("The code is in infinite loop\n");
```

```
int i=0;
for (; i<10; i++)
      printf("It will be printed 10 times.\n");
```

```
int i=0;
for (; i<10; ){
      printf("It will be printed 10 times.\n");
      i++;
}
```

Examples of varying control variable

Task	
Vary the control variable from 1 to 100 in increments of 1.	
Vary the control variable from 100 to 1 in increments of -1 (decrements of 1).	
Vary the control variable from 7 to 77 in steps of 7.	
Vary the control variable from 20 to 2 in steps of -2.	
Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17.	
Vary the control variable over the following sequence of values: 44, 33, 22, 11, 0.	

Examples of varying control variable

Task	for Loop
Vary the control variable from 1 to 100 in increments of 1.	`for (i = 1; i <= 100; ++ i)`
Vary the control variable from 100 to 1 in increments of -1 (decrements of 1).	`for (i = 100; i >= 1; --i)`
Vary the control variable from 7 to 77 in steps of 7.	`for (i = 7; i <= 77; i += 7)`
Vary the control variable from 20 to 2 in steps of -2.	`for (i = 20; i >= 2; i -= 2)`
Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17.	`for (j = 2; j <= 17; j += 3)`
Vary the control variable over the following sequence of values: 44, 33, 22, 11, 0.	`for (j = 44; j >= 0; j -= 11)`

For Statement Notes

- The initialization, loop-continuation condition and update expression can contain **arithmetic expressions**.

- For example, if `x = 2` and `y = 10`, the statement

```
for (j = x; j <= 4 * x * y; j += y / x)
```

  is equivalent to the statement

```
for (j = 2; j <= 80; j += 5)
```

- If the loop-continuation condition is initially false, the loop body does not execute.

For Statement – Variable Declaration

- The first expression in a `for` statement can be replaced by a declaration.
- This feature allows the programmer to declare a variable for use by the loop:

```
for (int i = 0; i < n; i++)
   …
```

- The variable `i` need not have been declared prior to this statement.

For Statement – Scope of variable

- A variable declared by a `for` statement can't be accessed outside the body of the loop (we say that it's not ***visible*** outside the loop):

```
for (int i = 0; i < n; i++) {
   …
   printf("%d", i);
   /* legal; i is visible inside
loop */
   …
}
printf("%d", i);    /*** WRONG ***/
```

For Statement – Control Variable Declaration

- Having a `for` statement declare its own control variable is usually a good idea: it's convenient and it can make programs easier to understand.
- However, if the program needs to access the variable after loop termination, it's necessary to use the older form of the `for` statement.
- A `for` statement may declare more than one variable, provided that all variables have the same type:

```
for (int i = 0, j = 0; i < n; i++)
   …
```

For Statement – Comma Operator

- On occasion, a `for` statement may need to have two (or more) initialization expressions or one that increments several variables each time through the loop.
- This effect can be accomplished by using a ***comma expression*** as the first or third expression in the `for` statement.
- A comma expression has the form

  expr1 , expr2

  where *expr1* and *expr2* are any two expressions.

For Statement – Comma Operator

- A comma expression is evaluated in two steps:
  - First, *expr1* is evaluated and its value discarded.
  - Second, *expr2* is evaluated; its value is the value of the entire expression.
- Evaluating *expr1* should always have a side effect; if it doesn't, then *expr1* serves no purpose.
- When the comma expression `++i, i + j` is evaluated, `i` is first incremented, then `i + j` is evaluated.
  - If `i` and `j` have the values 1 and 5, respectively, the value of the expression will be 7, and `i` will be incremented to 2.

For Statement – Comma Operator

- The comma operator is left associative, so the compiler interprets

  ```
  i = 1, j = 2, k = i + j
  ```

  as

  ```
  ((i = 1), (j = 2)), (k = (i + j))
  ```

- Since the left operand in a comma expression is evaluated before the right operand, the assignments `i = 1`, `j = 2`, and `k = i + j` will be performed from left to right.

For Statement – Comma Operator

- The comma operator makes it possible to "glue" two expressions together to form a single expression.
- Certain macro definitions can benefit from the comma operator.
- The `for` statement is the only other place where the comma operator is likely to be found.
- Example:

```
for (sum = 0, i = 1; i <= N; i++)
  sum += i;
```

- With additional commas, the `for` statement could initialize more than two variables.

Nested for Loop

```
int row, col;
for (row=0; row<2; row++)
    for (col=0; col<3; col++)
        printf("%d, %d\n", row, col);
```

Nested for Loop

```
int row, col;
for (row=0; row<2; row++)
     for (col=0; col<3; col++)
          printf(“%d, %d\n”, row, col);
```

Sample Output

```
0, 0
0, 1
0, 2
1, 0
1, 1
1, 2
```

Application: Summing even numbers

```
1   // Fig. 4.5: fig04_05.c
2   // Summation with for.
3   #include <stdio.h>
4
5   int main(void)
6   {
7      unsigned int sum = 0; // initialize sum
8
9      for (unsigned int number = 2; number <= 100; number += 2) {
10        sum += number; // add number to sum
11     }
12
13     printf("Sum is %u\n", sum);
14  }
```

```
Sum is 2550
```

Application: Compound Interest Calculation

- Consider the following problem statement:
  - A person invests $1000.00 in a savings account yielding 5% interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:

    $a = p(1 + r)^n$

    where

    p is the original amount invested (i.e., the principal)
    r is the annual interest rate
    n is the number of years
    a is the amount on deposit at the end of the nth year.

C Code for Compound Interest Calculation

```
1  // Fig. 4.6: fig04_06.c
2  // Calculating compound interest.
3  #include <stdio.h>
4  #include <math.h>
5
6  int main(void)
7  {
8     double principal = 1000.0; // starting principal
9     double rate = .05; // annual interest rate
10
11    // output table column heads
12    printf("%4s%21s\n", "Year", "Amount on deposit");
13
14    // calculate amount on deposit for each of ten years
15    for (unsigned int year = 1; year <= 10; ++year) {
16
17       // calculate new amount for specified year
18       double amount = principal * pow(1.0 + rate, year);
19
20       // output one table row
21       printf("%4u%21.2f\n", year, amount);
22    }
23 }
```

Output

```
Year     Amount on deposit
   1               1050.00
   2               1102.50
   3               1157.63
   4               1215.51
   5               1276.28
   6               1340.10
   7               1407.10
   8               1477.46
   9               1551.33
  10               1628.89
```

do … while Iteration Statement

- Similar to the `while` statement.
- `while` (*condition*)
- The loop-continuation condition is tested at the beginning of the loop

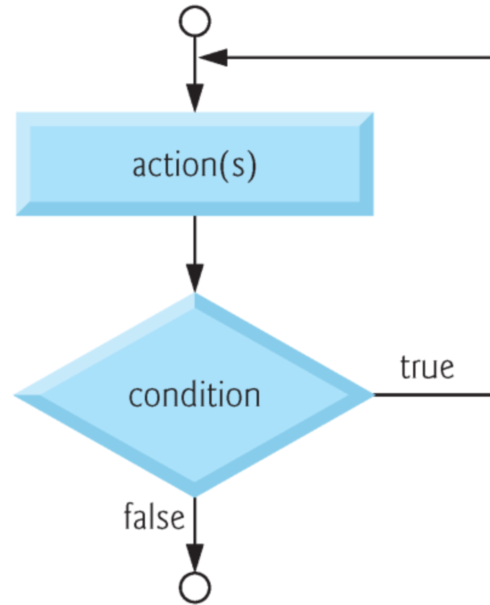
- `do`
  statement
  `while` (*condition*);
- The loop-continuation condition *after* the loop body is performed.
- The loop body will be executed at least once.

Example do ... while Iteration Statement

```
1   // Fig. 4.9: fig04_09.c
2   // Using the do...while iteration statement.
3   #include <stdio.h>
4
5   int main(void)
6   {
7      unsigned int counter = 1; // initialize counter
8
9      do {
10        printf("%u  ", counter);
11     } while (++counter <= 10);
12  }
```

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

Flowchart do … while Iteration Statement



`break` and `continue` Statements

- Break
  - Used inside `while, for, do…while, switch` Statements
  - When executed, program exits the statements
- Continue
  - Used in `while, for, do…while` Statements
  - When executed, the loop-continuation test is evaluated immediately *after* the `continue` statement is executed.
  - In the `for` statement, the increment expression is executed, then the loop-continuation test is evaluated.

break Statement

```
1   // Fig. 4.11: fig04_11.c
2   // Using the break statement in a for statement.
3   #include <stdio.h>
4
5   int main(void)
6   {
7      unsigned int x; // declared here so it can be used after loop
8
9      // loop 10 times
10     for (x = 1; x <= 10; ++x) {
11
12        // if x is 5, terminate loop
13        if (x == 5) {
14           break; // break loop only if x is 5
15        }
16
17        printf("%u ", x);
18     }
19
20     printf("\nBroke out of loop at x == %u\n", x);
21  }
```

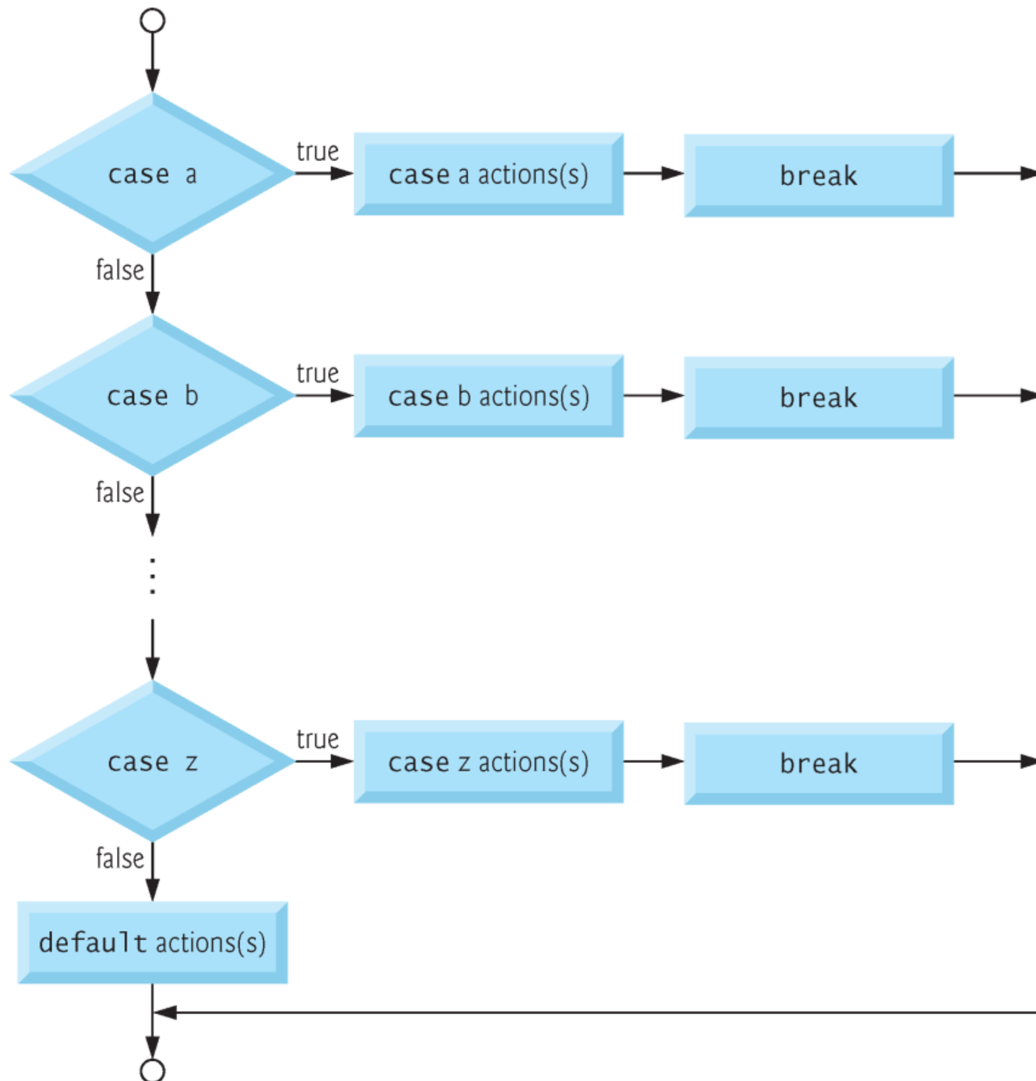
```
1 2 3 4
Broke out of loop at x == 5
```

continue Statement

```
1  // Fig. 4.12: fig04_12.c
2  // Using the continue statement in a for statement.
3  #include <stdio.h>
4
5  int main(void)
6  {
7     // loop 10 times
8     for (unsigned int x = 1; x <= 10; ++x) {
9
10       // if x is 5, continue with next iteration of loop
11       if (x == 5) {
12          continue; // skip remaining code in loop body
13       }
14
15       printf("%u ", x);
16    }
17
18    puts("\nUsed continue to skip printing the value 5");
19 }
```

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

Revisiting `switch` Statement



- If `break` is not used anywhere in a `switch` statement, then each time a match occurs in the statement, the statements for all the remaining `case`s will be executed—called *fallthrough*.
- If no match occurs, the `default` case is executed, and an error message is printed.

Code Snippet (1)

```
1   // Fig. 4.7: fig04_07.c
2   // Counting letter grades with switch.
3   #include <stdio.h>
4
5   int main(void)
6   {
7      unsigned int aCount = 0;
8      unsigned int bCount = 0;
9      unsigned int cCount = 0;
10     unsigned int dCount = 0;
11     unsigned int fCount = 0;
12
13     puts("Enter the letter grades.");
14     puts("Enter the EOF character to end input.");
15     int grade; // one grade
16
```

Code Snippet (2)

```
17      // loop until user types end-of-file key sequence
18      while ((grade = getchar()) != EOF) {
19
20         // determine which grade was input
21         switch (grade) { // switch nested in while
22
23            case 'A': // grade was uppercase A
24            case 'a': // or lowercase a
25               ++aCount;
26               break; // necessary to exit switch
27
28            case 'B': // grade was uppercase B
29            case 'b': // or lowercase b
30               ++bCount;
31               break;
32
33            case 'C': // grade was uppercase C
34            case 'c': // or lowercase c
35               ++cCount;
36               break;
37
```

Code Snippet (3)

```
38          case 'D': // grade was uppercase D
39          case 'd': // or lowercase d
40             ++dCount;
41             break;
42
43          case 'F': // grade was uppercase F
44          case 'f': // or lowercase f
45             ++fCount;
46             break;
47
48          case '\n': // ignore newlines,
49          case '\t': // tabs,
50          case ' ': // and spaces in input
51             break;
52
53          default: // catch all other characters
54             printf("%s", "Incorrect letter grade entered.");
55             puts(" Enter a new grade.");
56             break; // optional; will exit switch anyway
57       }
58    } // end while
59
```

Code Snippet (4) & Output

```
60     // output summary of results
61     puts("\nTotals for each letter grade are:");
62     printf("A: %u\n", aCount);
63     printf("B: %u\n", bCount);
64     printf("C: %u\n", cCount);
65     printf("D: %u\n", dCount);
66     printf("F: %u\n", fCount);
67  }
```

```
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
^Z
```

^Z — Not all systems display a representation of the EOF character

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

Logical Operators

- Used to form more complex conditions by combining simple conditions.
- The logical operators are && (logical AND), || (logical OR) and ! (logical NOT also called logical negation)
- Logical AND – used to ensure that two conditions are both true before we choose a certain path of execution
- Logical OR – used to ensure that at least one condition is true before we choose a certain path of execution
- Logical NOT – used to "reverse" the meaning of a condition.

Truth Table

- Table of Logic

expression1	expression2	expression1 && expression2
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

expression	!expression
0	1
nonzero	0

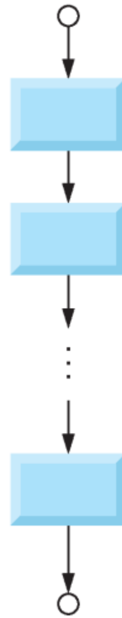
expression1	expression2	expression1 || expression2
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

Operator Precedence

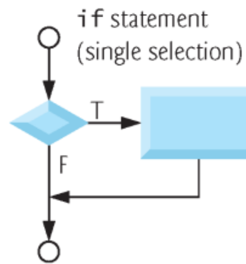
<table>
<tr><th>Operators</th><th>Associativity</th><th>Type</th></tr>
<tr><td>++ (postfix) -- (postfix)</td><td>right to left</td><td>postfix</td></tr>
<tr><td>+ - ! ++ (prefix) -- (prefix) (type)</td><td>right to left</td><td>unary</td></tr>
<tr><td>* / %</td><td>left to right</td><td>multiplicative</td></tr>
<tr><td>+ -</td><td>left to right</td><td>additive</td></tr>
<tr><td>< <= > >=</td><td>left to right</td><td>relational</td></tr>
<tr><td>== !=</td><td>left to right</td><td>equality</td></tr>
<tr><td>&&</td><td>left to right</td><td>logical AND</td></tr>
<tr><td>||</td><td>left to right</td><td>logical OR</td></tr>
<tr><td>?:</td><td>right to left</td><td>conditional</td></tr>
<tr><td>= += -= *= /= %=</td><td>right to left</td><td>assignment</td></tr>
<tr><td>,</td><td>left to right</td><td>comma</td></tr>
</table>

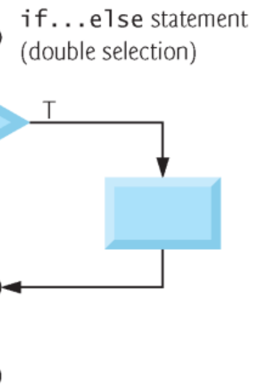
Structured Program Summary (1)

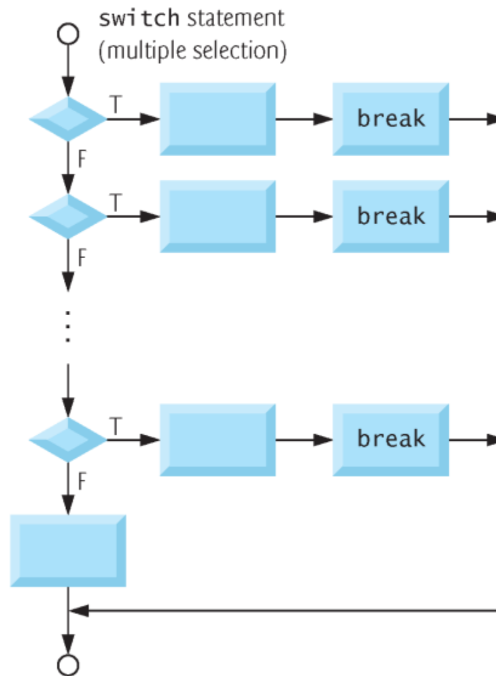
Sequence



Selection



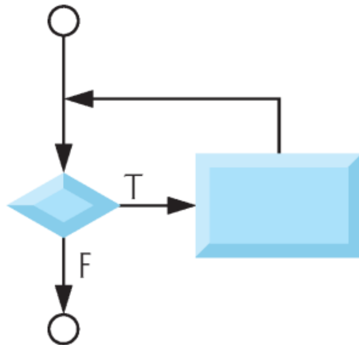




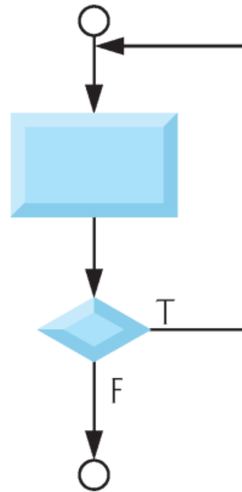
Structured Program Summary (2)

Repetition

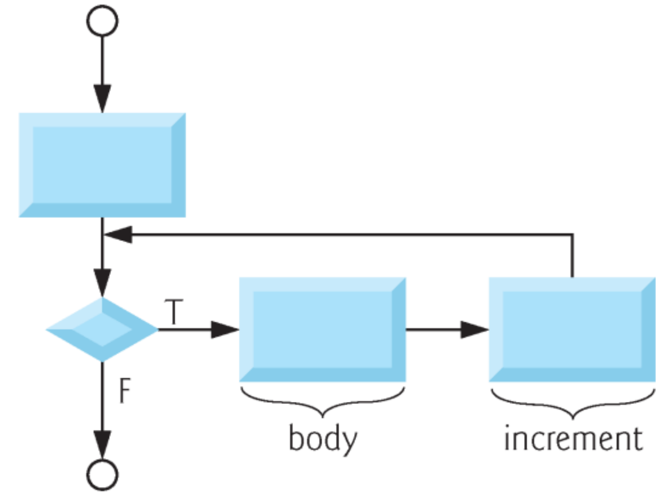
`while` statement



`do...while` statement



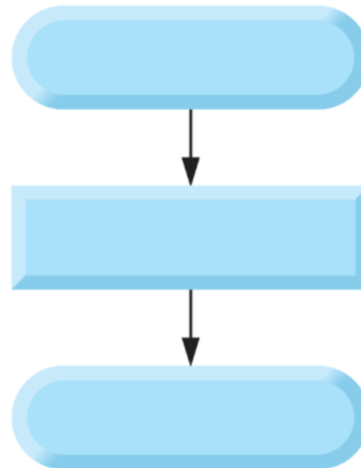
`for` statement



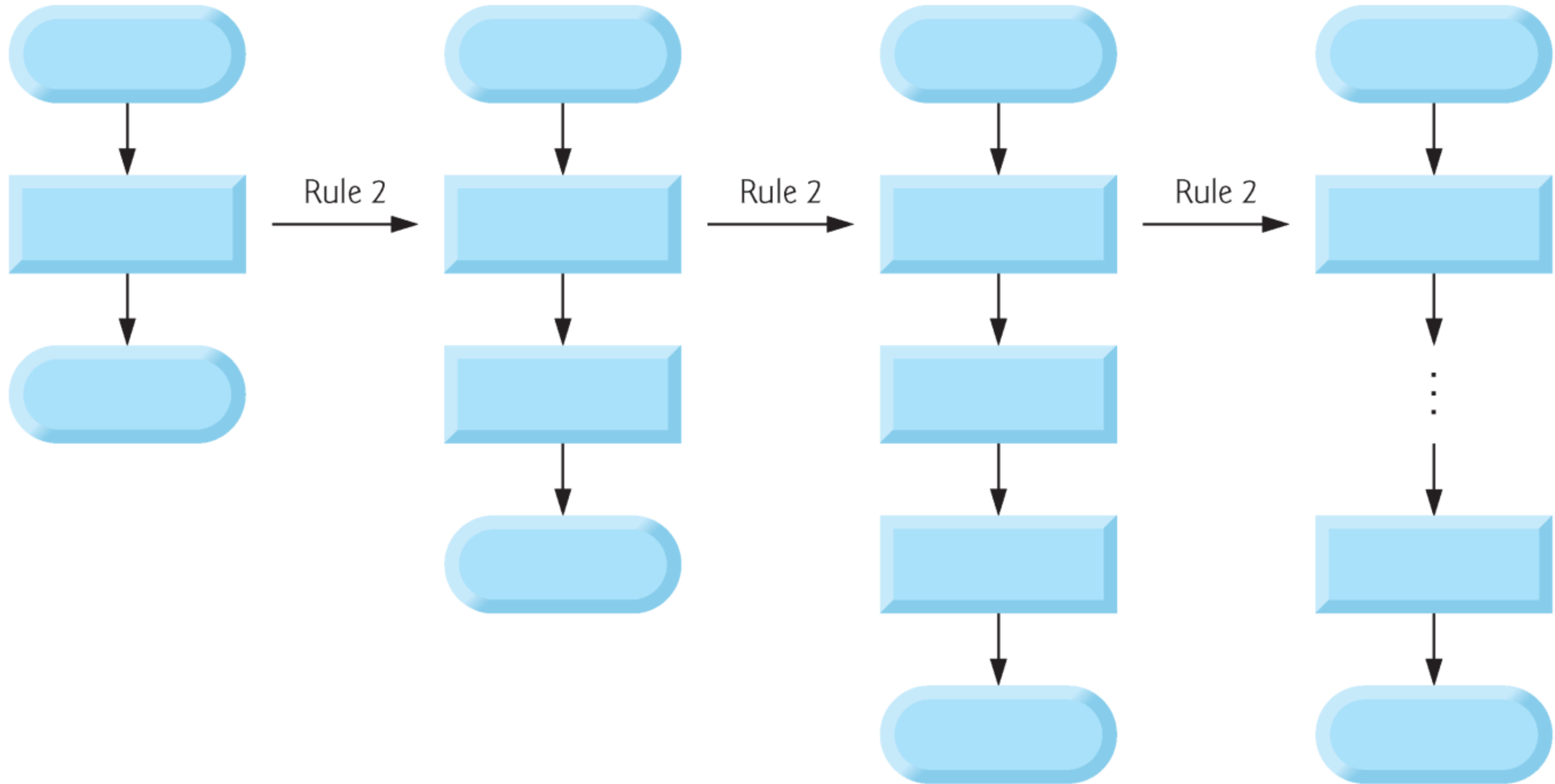
Rules for forming structured programs

- Begin with the simplest flowchart
- Stacking Rule – Any rectangle (action) can be replaced by two rectangles (actions) in sequence
- Nesting Rule – Any rectangle (action) can be replaced by any control statement
- Stacking & Nesting Rule rules may be applied in any order.

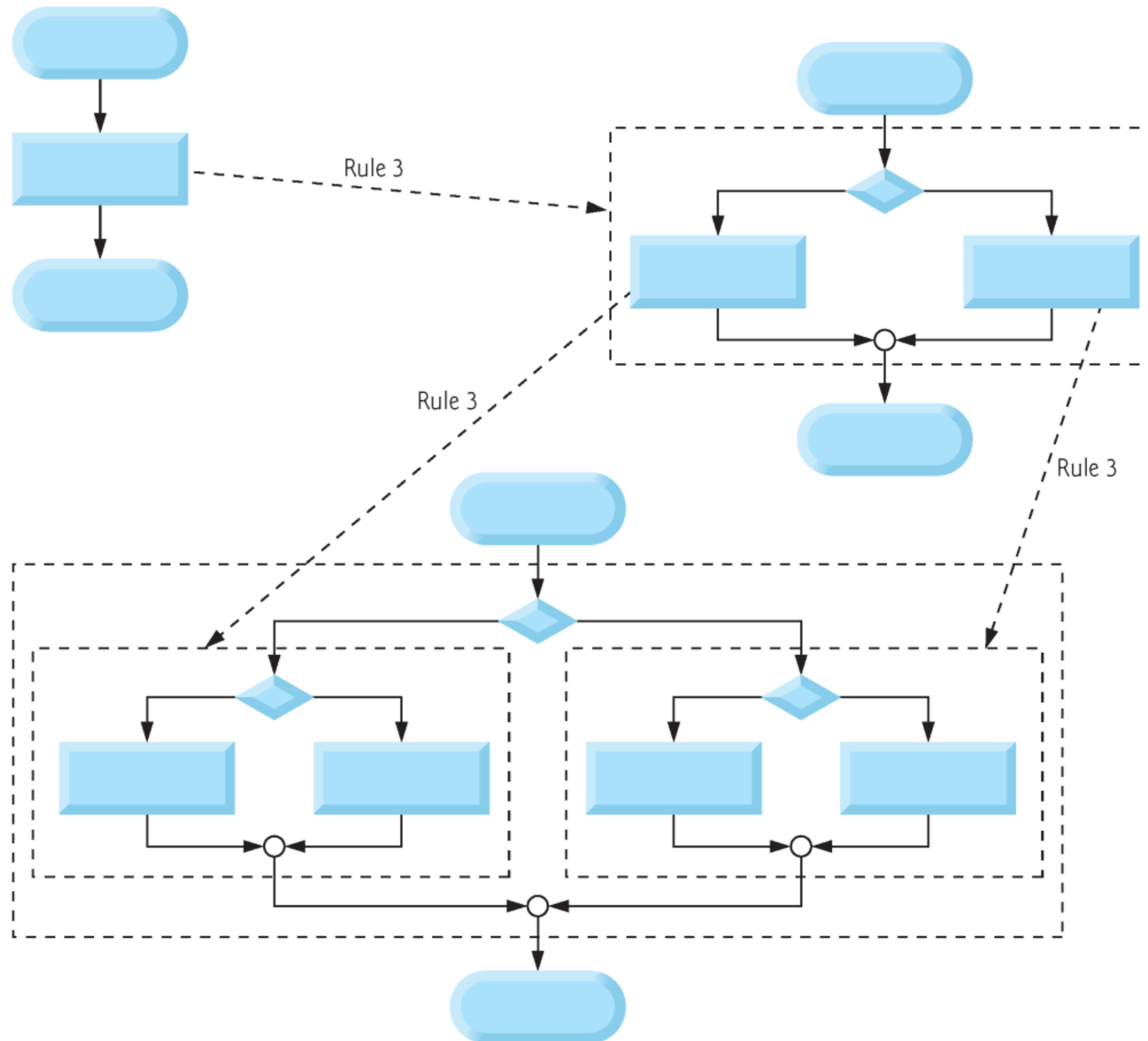
Simplest Flowchart



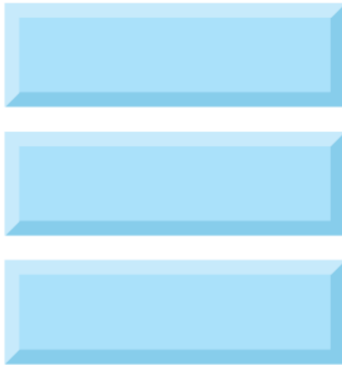
Stacking Rule

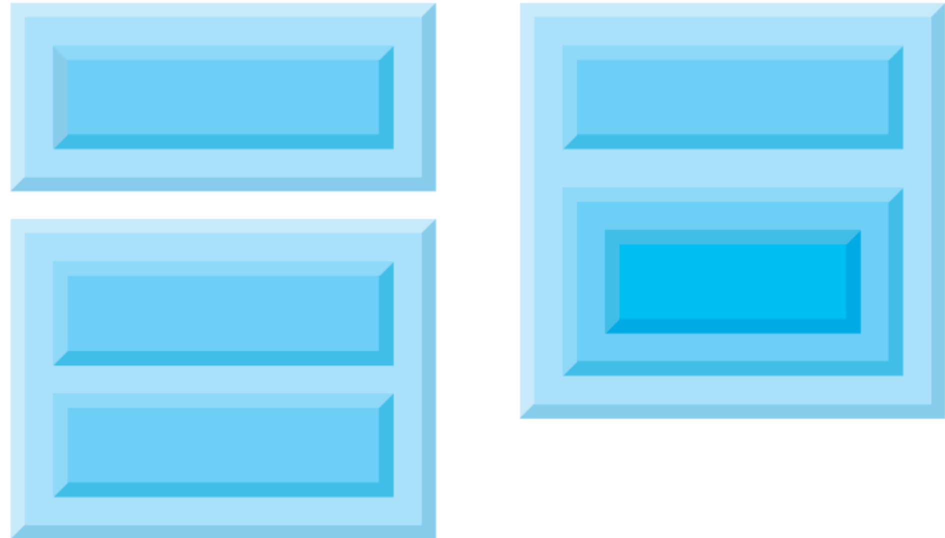


Nesting Rule



Structured Program Building Blocks





Structured Programming

- Structured programming promotes simplicity.
- Bohm and Jacopini showed that only three forms of control are needed:
  - Sequence
  - Selection
  - Iteration

Structured Programming Options

- Sequence is straightforward.
- Selection is implemented in one of three ways:
  - `if` statement (single selection)
  - `if…else` statement (double selection)
  - `switch` statement (multiple selection)
- Iteration is implemented in one of three ways:
  - `while` statement
  - `do…while` statement
  - `for` statement