Steps to writing a program

- Understand the problem
- Plan a solution
  - Step by step procedure
Algorithm

The solution to any computing problem involves executing a series of actions in a specific order.

A procedure for solving a problem in terms of:
- the actions to be executed, and
- the order in which these actions are to be executed

is called an algorithm.

Correctly specifying the order in which the actions are to be executed is important.
### Example “rise-and-shine” algorithm

<table>
<thead>
<tr>
<th>In-Order</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Get out of bed</td>
<td></td>
</tr>
<tr>
<td>2. Take of pajamas</td>
<td></td>
</tr>
<tr>
<td>3. Take a shower</td>
<td></td>
</tr>
<tr>
<td>4. Get dressed</td>
<td></td>
</tr>
<tr>
<td>5. Eat breakfast</td>
<td></td>
</tr>
<tr>
<td>6. Carpool to work</td>
<td></td>
</tr>
</tbody>
</table>
Order matters

Example “rise-and-shine” algorithm

<table>
<thead>
<tr>
<th>In-Order</th>
<th>Out-of-Order</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Get out of bed</td>
<td>1. Get out of bed</td>
</tr>
<tr>
<td>2. Take of pajamas</td>
<td>2. Take of pajamas</td>
</tr>
<tr>
<td>3. Take a shower</td>
<td>3. Get dressed</td>
</tr>
<tr>
<td>4. Get dressed</td>
<td>4. Take a shower</td>
</tr>
<tr>
<td>5. Eat breakfast</td>
<td>5. Eat breakfast</td>
</tr>
<tr>
<td>6. Carpool to work</td>
<td>6. Carpool to work</td>
</tr>
</tbody>
</table>

Specifying the order in which statements are to be executed in a computer program is called program control.
Flow Chart

- Graphical representation of an algorithm
- Uses certain special-purpose symbols such as rectangles, diamonds, rounded rectangles, and small circles
- Symbols are connected by arrows called flowlines

```
add grade to total
  total = total + grade;
add 1 to counter
  counter = counter + 1;
```
Flow Chart

- **Rectangle symbol** or action symbol indicate any type of action including a calculation or an input/output operation.
- The flowlines indicate the order in which the actions are performed.
Pseudocode

- Artificial, informal, user-friendly, convenient, English-like
- They are NOT executed on computers
- Can be easily converted into ANY programming language
- Consists of actions and decision statements

1. Set total to zero
2. Set grade counter to one
3. 
4. While grade counter is less than or equal to ten
   5. Input the next grade
   6. Add the grade into the total
   7. Add one to the grade counter
8. 
9. Set the class average to the total divided by ten
10. Print the class average
Control Structures

- **Sequential execution**: statements are executed one after another
- **Transfer of control**: Some C statements can specify that next statement to be executed MAY NOT be the next statement
Selection Statement

- **If Statement**
  - **If**: 
    - Performs a set of actions if condition is TRUE,
    - otherwise skip
  - **If...else**: 
    - Performs a set of actions if condition is TRUE,
    - otherwise performs a different set of actions

- **Switch Statement:**
  - Performs one of many different set of actions

- Used to choose among alternative courses of action.
Selection Statement in Flow Chart

**Single Selection**

```
grade >= 60
```

```
true
```

```
print “Passed”
```

```
false
```

```
```

**Double Selection**

```
print “Failed”
```

```
false
```

```
grade >= 60
```

```
true
```

```
print “Passed”
```

```
```
Selection Statement in Pseudocode

If student's grade is greater than or equal to 60
  Print “Passed”

If student's grade is greater than or equal to 60
  Print “Passed”
else
  Print “Failed”
Selection Statement in C

- if ( grade >= 60 ) {
  printf( "Passed\n" );
} // end if

- else {
  printf( "Failed\n" );
} // end else
Example: Swap values of two variables

```c
if (x > y) {
    temp = x; /* Switch x and y */
    /* Store old x in temp */
    x = y; /* Store old y in x */
    /* Store old x in y */
    y = temp;
}
```
Conditional Operator (?)

- C’s only ternary operator—it takes *three* operands.
- The first operand is a *condition*.
- The second operand is the value for the entire conditional expression if the condition is *TRUE*.
- The third operand is the value for the entire conditional expression if the condition is *FALSE*.

Example:

```c
printf( grade >= 60 ? "Passed" : "Failed" );
```
Classroom Assignment

- Develop an algorithm to find a number is odd or even
- Write a pseudocode to check if a number is odd or even
- Write a C code that takes an integer as input from the user and prints out whether it is odd or even number
Nested if... else Statements

```
if ( grade >= 90 )
puts( "A" );
else
    if ( grade >= 80 )
        puts("B");
    else
        if ( grade >= 70 )
            puts("C");
        else
            if ( grade >= 60 )
                puts( "D" );
            else
                puts( "F" );
```
If ... else if Statement

- if ( grade >= 90 )
  puts( "A" );
else if ( grade >= 80 )
  puts( "B" );
else if ( grade >= 70 )
  puts( "C" );
else if ( grade >= 60 )
  puts( "D" );
else
  puts( "F" );
The `if` selection statement expects only one statement in its body.

To include several statements in the body of an `if`, the set of statements are included in braces.

```c
if ( grade >= 60 )
    puts( "Passed. " ); // end if
else {
    puts( "Failed. " );
    puts( "Take this course again. " );
} // end else
```
## Precedence

<table>
<thead>
<tr>
<th>Operator</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>highest (evaluated first)</td>
</tr>
<tr>
<td>* / %</td>
<td></td>
</tr>
<tr>
<td>+ -</td>
<td></td>
</tr>
<tr>
<td>&lt; &lt;= &gt;= &gt;</td>
<td></td>
</tr>
<tr>
<td>== !=</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>lowest (evaluated last)</td>
</tr>
</tbody>
</table>
Condition Statements

\((z+x < 1) \text{ || } (y+z \geq x - z)\)
Common usage in program

- Check range of $x$

\[ x \geq \text{min} \quad \&\& \quad x \leq \text{max} \]

\[ x < z \quad || \quad x > y \]
The technique of stopping evaluation of a logical expression as soon as its value can be determined is called **short-circuit evaluation**.

We can use short-circuit evaluation to prevent potential run-time errors. The condition 

\[(\text{num} \mod \text{div} == 0)\]

tests whether \(\text{div}\) is a divisor of \(\text{num}\). For example, if \(\text{num}\) is 6 and \(\text{div}\) is 2, the remainder is 0 so the condition is true. If \(\text{num}\) is 6 and \(\text{div}\) is 4, the remainder is 2 so the condition is false.

What if \(\text{div}\) is 0? In this case, the remainder calculation would cause a division by zero run-time error. However, we can prevent this error by using the revised condition 

\[(\text{div} != 0 && (\text{num} \mod \text{div} == 0))\]

The remainder would not be calculated when \(\text{div}\) is 0 because \(\text{div} != 0\) is false.

**Writing English Conditions in C**

To solve programming problems, you must convert conditions expressed in English to C. Many algorithm steps require testing to see if a variable's value is within a specified range of values. For example, if \(\text{min}\) represents the lower bound of a range of values and \(\text{max}\) represents the upper bound (\(\text{min}\) is less than \(\text{max}\)), the expression 

\[\text{min} \leq \text{x} && \text{x} \leq \text{max}\]

tests whether \(\text{x}\) lies within the range \(\text{min}\) through \(\text{max}\), inclusive. In **Fig. 4.2** this range is shaded. The expression is 1 (true) if \(\text{x}\) lies within this range and 0 (false) if \(\text{x}\) is outside the range.

**EXAMPLE 4.3**

**Table 4.7** shows some English conditions and the corresponding C expressions. Each expression is evaluated assuming \(\text{x}\) is 3.0, \(\text{y}\) is 4.0, and \(\text{z}\) is 2.0.

<table>
<thead>
<tr>
<th>English Condition</th>
<th>Logical Expression</th>
<th>Evaluation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{x}) and (\text{y}) are greater than (\text{z})</td>
<td>(\text{x} &gt; \text{z} \ &amp;&amp; \ \text{y} &gt; \text{z})</td>
<td>1 &amp;&amp; 1</td>
<td>1</td>
</tr>
<tr>
<td>(\text{x}) is equal to 1.0 or 3.0</td>
<td>(\text{x} == 1.0 \</td>
<td></td>
<td>\ \text{x} == 3.0)</td>
</tr>
<tr>
<td>(\text{x}) is in the range (\text{z}) to (\text{y}), inclusive</td>
<td>(\text{z} \leq \text{x} \ &amp;&amp; \ \text{x} \leq \text{y})</td>
<td>1 &amp;&amp; 1</td>
<td>1</td>
</tr>
<tr>
<td>(\text{x}) is outside the range (\text{z}) to (\text{y})</td>
<td>(\neg (\text{z} \leq \text{x} \ &amp;&amp; \ \text{x} \leq \text{y}))</td>
<td>\neg (1 &amp;&amp; 1)</td>
<td>0</td>
</tr>
</tbody>
</table>
The technique of stopping evaluation of a logical expression as soon as its value can be determined is called **short-circuit evaluation**.

We can use short-circuit evaluation to prevent potential run-time errors. The condition `(num % div == 0)` tests whether `div` is a divisor of `num`. For example, if `num` is 6 and `div` is 2, the remainder is 0 so the condition is true. If `num` is 6 and `div` is 4, the remainder is 2 so the condition is false.

What if `div` is 0? In this case, the remainder calculation would cause a division by zero run-time error. However, we can prevent this error by using the revised condition `(div != 0 && (num % div == 0))`. The remainder would not be calculated when `div` is 0 because `div != 0` is false.

### Writing English Conditions in C

To solve programming problems, you must convert conditions expressed in English to C. Many algorithm steps require testing to see if a variable's value is within a specified range of values. For example, if `min` represents the lower bound of a range of values and `max` represents the upper bound (**min** is less than **max**), the expression `min <= x && x <= max` tests whether `x` lies within the range **min** through **max**, inclusive. In Fig. 4.2 this range is shaded. The expression is 1 (true) if `x` lies within this range and 0 (false) if `x` is outside the range.

### EXAMPLE 4.3

Table 4.7 shows some English conditions and the corresponding C expressions. Each expression is evaluated assuming `x` is 3.0, `y` is 4.0, and `z` is 2.0.

<table>
<thead>
<tr>
<th>English Condition</th>
<th>Logical Expression</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x</code> and <code>y</code> are greater than <code>z</code></td>
<td><code>x &gt; z &amp;&amp; y &gt; z</code></td>
<td>1 &amp;&amp; 1 is 1 (true)</td>
</tr>
<tr>
<td><code>x</code> is equal to 1.0 or 3.0</td>
<td>`x == 1.0</td>
<td></td>
</tr>
<tr>
<td><code>x</code> is in the range <code>z</code> to <code>y</code>, inclusive</td>
<td><code>z &lt;= x &amp;&amp; x &lt;= y</code></td>
<td>1 &amp;&amp; 1 is 1 (true)</td>
</tr>
<tr>
<td><code>x</code> is outside the range <code>z</code> to <code>y</code></td>
<td>!(z &lt;= x &amp;&amp; x &lt;= y)</td>
<td>! (1 &amp;&amp; 1) is 0 (false)</td>
</tr>
<tr>
<td></td>
<td>`z &gt; x</td>
<td></td>
</tr>
</tbody>
</table>