Data type conversions

- **Grade average example**

  - *class average* = \( \frac{\sum \text{grade}}{\text{number of students}} \)
  
  - 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

Before...

Increment...

Prefix: Increment i and then use it.

Postfix: Use i and then increment it.

After...
Assignment operators

- Any statement of the form
  - `variable = variable operator expression;`
- where `operator` is one of the binary operators `+`, `-`, `*`, `/` or `%` (or others we’ll discuss in Chapter 10), can be written in the form
  - `variable operator = expression;`
- Thus the assignment `c += 3` adds 3 to `c`. 
Assignment operator - examples

<table>
<thead>
<tr>
<th>Assignment operator</th>
<th>Sample expression</th>
<th>Explanation</th>
<th>Assigns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+=</code></td>
<td><code>c += 7</code></td>
<td><code>c = c + 7</code></td>
<td>10 to c</td>
</tr>
<tr>
<td><code>-=</code></td>
<td><code>d -= 4</code></td>
<td><code>d = d - 4</code></td>
<td>1 to d</td>
</tr>
<tr>
<td><code>*=</code></td>
<td><code>e *= 5</code></td>
<td><code>e = e * 5</code></td>
<td>20 to e</td>
</tr>
<tr>
<td><code>/=</code></td>
<td><code>f /= 3</code></td>
<td><code>f = f / 3</code></td>
<td>2 to f</td>
</tr>
<tr>
<td><code>%=</code></td>
<td><code>g %= 9</code></td>
<td><code>g = g % 9</code></td>
<td>3 to g</td>
</tr>
</tbody>
</table>

Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`
## Unary Increment & Decrement Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Sample expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>++a</td>
<td>Increment a by 1, then use the new value of a in the expression in which a resides.</td>
</tr>
<tr>
<td>++</td>
<td>a++</td>
<td>Use the current value of a in the expression in which a resides, then increment a by 1.</td>
</tr>
<tr>
<td>--</td>
<td>--b</td>
<td>Decrement b by 1, then use the new value of b in the expression in which b resides.</td>
</tr>
<tr>
<td>--</td>
<td>b--</td>
<td>Use the current value of b in the expression in which b resides, then decrement b by 1.</td>
</tr>
</tbody>
</table>
Increment Example

```c
// Fig. 3.13: fig03_13.c
// Preincrementing and postincrementing.
#include <stdio.h>

// function main begins program execution
int main( void )
{
    int c;  // define variable

    // demonstrate postincrement
    c = 5;   // assign 5 to c
    printf( "%d\n", c );  // print 5
    printf( "%d\n", c++ );  // print 5 then postincrement
    printf( "%d\n\n", c );  // print 6

    // demonstrate preincrement
    c = 5;   // assign 5 to c
    printf( "%d\n", c );  // print 5
    printf( "%d\n", ++c );  // preincrement then print 6
    printf( "%d\n", c );  // print 6
}

// end function main
```
Increment Example

    1 // Fig. 3.13: fig03_13.c
    2 // Preincrementing and postincrementing.
    3 #include <stdio.h>

    4 // function main begins program execution
    5 int main( void )
    6 {
    7     int c; // define variable
    8
    9     // demonstrate postincrement
   10     c = 5; // assign 5 to c
   11     printf( "%d\n", c ); // print 5
   12     printf( "%d\n", c++ ); // print 5 then postincrement
   13     printf( "%d\n\n", c ); // print 6
   14
   15     // demonstrate preincrement
   16     c = 5; // assign 5 to c
   17     printf( "%d\n", c ); // print 5
   18     printf( "%d\n", ++c ); // preincrement then print 6
   19     printf( "%d\n", c ); // print 6
   20 }
    21 // end function main

Output

5
5
6

5
6
6
### Precedence

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>++ (postfix)</code></td>
<td>right to left</td>
<td>postfix</td>
</tr>
<tr>
<td><code>-- (postfix)</code></td>
<td>right to left</td>
<td></td>
</tr>
<tr>
<td><code>+  -  (type)</code></td>
<td>left to right</td>
<td>unary</td>
</tr>
<tr>
<td><code>++ (prefix)  -- (prefix)</code></td>
<td>left to right</td>
<td>multiplicative</td>
</tr>
<tr>
<td><code>*  /  %</code></td>
<td>left to right</td>
<td>additive</td>
</tr>
<tr>
<td><code>+  -</code></td>
<td>left to right</td>
<td>relational</td>
</tr>
<tr>
<td><code>&lt;  &lt;=  &gt;  &gt;=</code></td>
<td>left to right</td>
<td>equality</td>
</tr>
<tr>
<td><code>==  !=</code></td>
<td>left to right</td>
<td>conditional</td>
</tr>
<tr>
<td><code>?:</code></td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td><code>=  +=  -=  *=  /=  %=</code></td>
<td>right to left</td>
<td></td>
</tr>
</tbody>
</table>
for Iteration Statement - Syntax

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

for (count_star = 0; count_star < N; count_star ++)
printf("*");
The general format of the `for` statement is

```plaintext
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

```c
// Fig. 4.2: fig04_02.c
// Counter-controlled iteration with the for statement.
#include <stdio.h>

int main(void)
{
    // initialization, iteration condition, and increment
    // are all included in the for statement header.
    for (unsigned int counter = 1; counter <= 10; ++counter) {
        printf("%u\n", counter);
    }
}
```
Flow chart

Establish initial value of control variable

unsigned int counter = 1

counter <= 10

determine if final value of control variable has been reached

true

printf("%u", counter);

false

Body of loop (this may be many statements)

Increment the control variable

++counter
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.
Start the loop from 0

```c
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

```c
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

<table>
<thead>
<tr>
<th>Task</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vary the control variable from 1 to 100 in increments of 1.</td>
<td></td>
</tr>
<tr>
<td>Vary the control variable from 100 to 1 in increments of -1 (decrements of 1).</td>
<td></td>
</tr>
<tr>
<td>Vary the control variable from 7 to 77 in steps of 7.</td>
<td></td>
</tr>
<tr>
<td>Vary the control variable from 20 to 2 in steps of -2.</td>
<td></td>
</tr>
<tr>
<td>Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17.</td>
<td></td>
</tr>
<tr>
<td>Vary the control variable over the following sequence of values: 44, 33, 22, 11, 0.</td>
<td></td>
</tr>
</tbody>
</table>
### Examples of varying control variable

<table>
<thead>
<tr>
<th>Task</th>
<th>for Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vary the control variable from 1 to 100 in increments of 1.</td>
<td><code>for (i = 1; i &lt;= 100; ++i)</code></td>
</tr>
<tr>
<td>Vary the control variable from 100 to 1 in increments of -1 (decrements of 1).</td>
<td><code>for (i = 100; i &gt;= 1; --i)</code></td>
</tr>
<tr>
<td>Vary the control variable from 7 to 77 in steps of 7.</td>
<td><code>for (i = 7; i &lt;= 77; i += 7)</code></td>
</tr>
<tr>
<td>Vary the control variable from 20 to 2 in steps of -2.</td>
<td><code>for (i = 20; i &gt;= 2; i -= 2)</code></td>
</tr>
<tr>
<td>Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17.</td>
<td><code>for (j = 2; j &lt;= 17; j += 3)</code></td>
</tr>
<tr>
<td>Vary the control variable over the following sequence of values: 44, 33, 22, 11, 0.</td>
<td><code>for (j = 44; j &gt;= 0; j -= 11)</code></td>
</tr>
</tbody>
</table>
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
  
  ```c
  for (j = x; j <= 4 * x * y; j += y / x)
  ```
  
is equivalent to the statement
  
  ```c
  for (j = 2; j <= 80; j += 5)
  ```

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

int row, col;
for (row=0; row<2; row++)
    for (col=0; col<3; col++)
        printf(“%d, %d
”, row, col);
Nested for Loop

```c
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

```c
#include <stdio.h>

int main(void)
{
    unsigned int sum = 0; // initialize sum

    for (unsigned int number = 2; number <= 100; number += 2) {
        sum += number; // add number to sum
    }

    printf("Sum is %u\n", sum);
}

Sum is 2550
```
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 \( n \)th year.
C Code for Compound Interest Calculation

```c
// Fig. 4.6: fig04_06.c
// Calculating compound interest.
#include <stdio.h>
#include <math.h>

int main(void)
{
    double principal = 1000.0; // starting principal
    double rate = .05; // annual interest rate

    // output table column heads
    printf("%4s%21s\n", "Year", "Amount on deposit");

    // calculate amount on deposit for each of ten years
    for (unsigned int year = 1; year <= 10; ++year) {
        // calculate new amount for specified year
        double amount = principal * pow(1.0 + rate, year);

        // output one table row
        printf("%u%21.2f\n", year, amount);
    }
}```
<table>
<thead>
<tr>
<th>Year</th>
<th>Amount on deposit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1050.00</td>
</tr>
<tr>
<td>2</td>
<td>1102.50</td>
</tr>
<tr>
<td>3</td>
<td>1157.63</td>
</tr>
<tr>
<td>4</td>
<td>1215.51</td>
</tr>
<tr>
<td>5</td>
<td>1276.28</td>
</tr>
<tr>
<td>6</td>
<td>1340.10</td>
</tr>
<tr>
<td>7</td>
<td>1407.10</td>
</tr>
<tr>
<td>8</td>
<td>1477.46</td>
</tr>
<tr>
<td>9</td>
<td>1551.33</td>
</tr>
<tr>
<td>10</td>
<td>1628.89</td>
</tr>
</tbody>
</table>
Classwork Assignment

- Write a program that finds the smallest of several integers. Assume that the first value read specifies the number of values remaining. Your program should read only one value each time scanf is executed.

- A typical input sequence might be
  - 5 400 500 300 200 100
  - where 5 indicates that the subsequent five values are to be used for finding minimum.
Classwork Assignment

Write a program that prints the following patterns separately, one below the other. Use for loops to generate the patterns. [Hint: The last two patterns require that each line begin with an appropriate number of blanks.]

(A)  
*  
**  
***  
****  
*****  
******  
*******  
********  
*********  
**********  
***********  
************  
*************  
**************  
***************  
***************

(B)  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********

(C)  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********  
**********

(D)  
*  
**  
***  
****  
*****  
******  
*******  
********  
*********  
**********  
***********  
************  
*************  
**************  
***************  
***************

Exercises

4.13 (Calculating the Product of Odd Integers)
Write a program that calculates and prints the product of the odd integers from 1 to 15.

4.14 (Factorials)
The factorial function is used frequently in probability problems. The factorial of a positive integer \( n \) (written as \( n! \) and pronounced "\( n \) factorial") is equal to the product of the positive integers from 1 to \( n \). Write a program that evaluates the factorials of the integers from 1 to 5. Print the results in tabular format. What difficulty might prevent you from calculating the factorial of 20?

4.15 (Modified Compound-Interest Program)
Modify the compound-interest program of Section 4.6 to repeat its steps for interest rates of 5%, 6%, 7%, 8%, 9%, and 10%. Use a for loop to vary the interest rate.

4.16 (Triangle-Printing Program)
Write a program that prints the following patterns separately, one below the other. Use for loops to generate the patterns. All asterisks (\*) should be printed by a single printf statement of the form printf("%s", "*"); (this causes the asterisks to print side by side). [Hint: The last two patterns require that each line begin with an appropriate number of blanks.]

4.17 (Calculating Credit Limits)
Collecting money becomes increasingly difficult during periods of recession, so companies may tighten their credit limits to prevent their accounts receivable (money owed to them) from becoming too large. In response to a prolonged recession, one company has cut its customers' credit limits in half. Thus, if a particular customer had a credit limit of $2000, it's now $1000. If a customer had a credit limit of $5000, it's now $2500. Write a program that analyzes the credit status of the customers of this company. For each customer you're given:

a) The customer's account number
b) The customer's credit limit before the recession
c) The customer's current balance (i.e., the amount the customer owes the company).

Your program should calculate and print the new credit limit for each customer and should determine (and print) which customers have current balances that exceed their new credit limits.

4.18 (Bar Chart Printing Program)
One interesting application of computers is drawing graphs and bar charts (sometimes called "histograms"). Write a program that reads five numbers (each between 1 and 30). For each number read, your program should print a line containing that number of adjacent asterisks. For example, if your program read the number seven, it should print *******.

4.19 (Calculating Sales)
An online retailer sells five different products whose retail prices are shown in the following table:

<table>
<thead>
<tr>
<th>Product number</th>
<th>(A)</th>
<th>(B)</th>
<th>(C)</th>
<th>(D)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>**</td>
<td>***</td>
<td>***</td>
<td>**</td>
</tr>
<tr>
<td></td>
<td>***</td>
<td>****</td>
<td>****</td>
<td>***</td>
</tr>
<tr>
<td></td>
<td>****</td>
<td>*****</td>
<td>*****</td>
<td>****</td>
</tr>
<tr>
<td></td>
<td>*****</td>
<td>******</td>
<td>******</td>
<td>*****</td>
</tr>
<tr>
<td></td>
<td>******</td>
<td>*******</td>
<td>*******</td>
<td>******</td>
</tr>
<tr>
<td></td>
<td>*******</td>
<td>*******</td>
<td>*******</td>
<td>*******</td>
</tr>
<tr>
<td></td>
<td>*******</td>
<td>*******</td>
<td>*******</td>
<td>*******</td>
</tr>
<tr>
<td></td>
<td>*******</td>
<td>*******</td>
<td>*******</td>
<td>*******</td>
</tr>
</tbody>
</table>

Product number
Retail price
1    $ 2.98
2    $ 4.50
3    $ 9.98
4    $ 4.49
5    $ 6.87
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

```c
// Fig. 4.9: fig04_09.c
// Using the do...while iteration statement.
#include <stdio.h>

int main(void)
{
    unsigned int counter = 1; // initialize counter

do {
    printf("%u ", counter);
} while (++counter <= 10);
```

```c
```
Flowchart do ... while Iteration Statement