C Programming for Engineers

Iteration

ICEN 360– Spring 2017

Prof. Dola Saha
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
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 \( n \)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("%4u%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.
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.]

```plaintext
(A)  *  
     ** 
     *** 
     **** 
     ***** 
     ****** 
     ******* 
     ******** 
     ********* 
     ********** 

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

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

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

1. Action(s)
2. Condition (true or false)
3. Loop continues if true, otherwise exits.
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.
// Fig. 4.11: fig04_11.c
// Using the break statement in a for statement.
#include <stdio.h>

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

    // loop 10 times
    for (x = 1; x <= 10; ++x) {
        // if x is 5, terminate loop
        if (x == 5) {
            break; // break loop only if x is 5
        }
        printf("%u ", x);
    }
    printf("\nBroke out of loop at x == %u\n", x);
continue Statement

```c
// Fig. 4.12: fig04_12.c
// Using the continue statement in a for statement.
#include <stdio.h>

int main(void)
{
    // loop 10 times
    for (unsigned int x = 1; x <= 10; ++x) {
        // if x is 5, continue with next iteration of loop
        if (x == 5) {
            continue; // skip remaining code in loop body
        }
        printf("%u ", x);
    }
    puts("\nUsed continue to skip printing the value 5");
}
```

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 cases will be executed—called *fallthrough*.

- If no match occurs, the `default` case is executed, and an error message is printed.
// Fig. 4.7: fig04_07.c
// Counting letter grades with switch.
#include <stdio.h>

int main(void)
{
    unsigned int aCount = 0;
    unsigned int bCount = 0;
    unsigned int cCount = 0;
    unsigned int dCount = 0;
    unsigned int fCount = 0;

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

```c
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
 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

<table>
<thead>
<tr>
<th>expression 1</th>
<th>expression 2</th>
<th>expression 1 &amp;&amp; expression 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>0</td>
</tr>
<tr>
<td>nonzero</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nonzero</td>
<td>nonzero</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>expression</th>
<th>! expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>nonzero</td>
<td>0</td>
</tr>
</tbody>
</table>

| expression 1 | expression 2 | expression 1 || expression 2 |
|--------------|--------------|---------------|
| 0            | 0            | 0             |
| 0            | nonzero      | 1             |
| nonzero      | 0            | 1             |
| nonzero      | nonzero      | 1             |
## Operator Precedence

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>++ (postfix)</td>
<td>right to left</td>
<td>postfix</td>
</tr>
<tr>
<td>-- (postfix)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+, - , !</td>
<td>right to left</td>
<td>unary</td>
</tr>
<tr>
<td>++ (prefix) , -- (prefix)</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>&lt; &lt;= &gt; &gt;=</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>&amp;&amp;</td>
<td>left to right</td>
<td>logical OR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
<td>assignment</td>
</tr>
<tr>
<td>= += -= *= /= %=</td>
<td>right to left</td>
<td>comma</td>
</tr>
<tr>
<td>,</td>
<td>left to right</td>
<td></td>
</tr>
</tbody>
</table>
Structured Program Summary (2)

Repetition

while statement

do...while statement

for statement

body

increment
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

Stacked building blocks

Nested building blocks

Overlapping building blocks
(Illegal in structured programs)
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