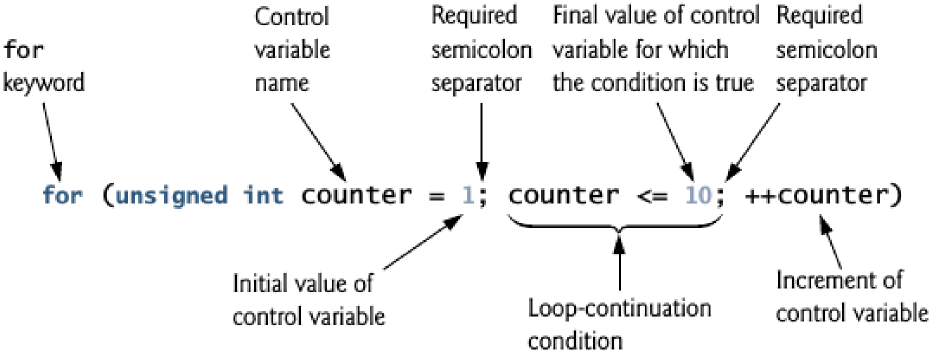
# CC-112 Programming Fundamentals

## Program Control in C

**Nazar Khan**
Department of Computer Science
University of the Punjab

# The for loop

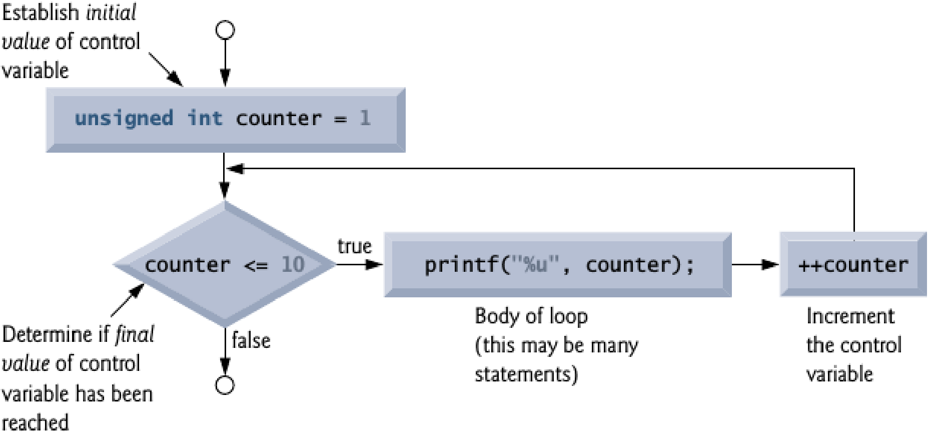


```
for (unsigned int counter = 1; counter <= 10; ++counter)
```

for keyword
Control variable name
Required semicolon separator
Final value of control variable for which the condition is true
Required semicolon separator
Initial value of control variable
Loop-continuation condition
Increment of control variable

for statement header

# The for loop


Establish *initial value* of control variable
unsigned int counter = 1
counter <= 10
true
printf("%u", counter);
++counter
false
Determine if *final value* of control variable has been reached
Body of loop (this may be many statements)
Increment the control variable

for statement flowchart

# The for loop

- The general format of the for statement is

  ```
  for (initialization; condition; increment) {
    statements
  }
  ```

- initialization expression initializes (and possibly defines) the control variable.
- condition expression is the loop-continuation condition.
- increment expression increments the control variable.

# The `for` loop

- All 3 expressions are optional.
  - If control variable is initialized before the loop, `initialization` expression can be omitted.
  - If `condition` expression is omitted, C assumes it is true, thus creating an *infinite loop*.
  - If increment is calculated by statements in the for statement's body or if no increment is needed, `increment` expression can be omitted.
- The two semicolons in the for statement are *required*.
- Control variables defined in a for header exist only until the loop terminates.

# The for loop

- The initialization, loop-continuation condition and increment 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)
  ```

# Examples

- Vary the control variable from 1 to 100 in increments of 1.

```
for (unsigned int i = 1; i <= 100; ++i)
```

- Vary the control variable from 100 to 1 in increments of -1 (i.e., decrements of 1).

```
for (unsigned int i = 100; i >= 1; --i)
```

- Vary the control variable from 7 to 77 in increments of 7.

```
for (unsigned int i = 7; i <= 77; i += 7)
```

- Vary the control variable from 20 to 2 in increments of -2.

```
for (unsigned int i = 20; i >= 2; i -= 2)
```

# Examples

- Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17.

```
for (unsigned int j = 2; j <= 17; j += 3)
```

- Vary the control variable over the following sequence of values: 44, 33, 22, 11, 0.

```
for (unsigned int j = 44; j >= 0; j -= 11)
```

# What does this program do?

```
#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);
}
```

# Computing compound interest

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

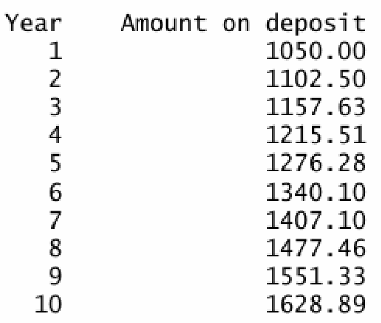
- $p$ is the original amount invested (i.e., the principal),
- $r$ is the annual interest rate (for example, .05 for 5%),
- $n$ is the number of years, and
- $a$ is the amount on deposit at the end of the $n^{\text{th}}$ year.

# Computing compound interest

```c
/* File name: compound_interest.c
   Program to compute compound interest using the formula a = p*(1+r)^n.
   To compile and link:
        gcc compound_interest.c -o compound_interest
   To compile and link on Linux/UNIX, use -lm to link
   the math library to the program:
        gcc compound_interest.c -lm -o compound_interest
   To run: ./compound_interest
*/
#include <stdio.h>
#include <math.h> //contains implementation of the pow() function

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);
  }
}
```

# Computing compound interest



```
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
```

# Float vs. Double

- Type `double` is a floating-point type like `float`.
- But a variable of type `double` can store
  - a value of much greater magnitude
  - with greater precision

  than `float`.
- Variables of type `double` occupy more memory than those of type `float`.
- For all but the most memory-intensive applications, professional programmers generally prefer `double` to `float`.

Avoid using `float` and `double` for monetary amounts! See page 155.

# Formatting Numeric Output

- What does the *conversion specifier* %21.2f do?
  - 21 denotes the *field width* in which the value will be printed.
  - 2 specifies the *precision* (i.e., the number of decimal positions).
- If the number of characters displayed is less than the field width, then the value will automatically be right justified with leading spaces in the field.
- Useful for aligning decimal points vertically.
- To left justify a value in a field, place a - (minus sign) between the % and the field width. For example, %-5.2f or %-6d or %-8s.

# Counting grades using the switch statement

```c
/* File name: grade_counts.c
   Program for counting letter grades with the ’switch’ statement.
   To compile and link:
        gcc grade_counts.c -o grade_counts
   To run: ./grade_counts
*/

#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 end-of-file (EOF) sequence to end input.");
  puts("In Ubuntu, EOF is indicated by pressing Ctrl-D.");
  puts("In Windows, EOF is indicated by pressing Ctrl-Z and then pressing enter.");
  int grade; // one grade

  // loop until user types end-of-file key sequence
  while ((grade = getchar()) != EOF) {

    // determine which grade was input
    switch (grade) { // switch nested in while
```

# Counting grades using the switch statement

```
    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;
```

# Counting grades using the switch statement

```
      default: // catch all other characters
        printf("%s", "Incorrect letter grade entered.");
        puts(" Enter a new grade.");
        break; // optional; will exit switch anyway
    }
  } // end while

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

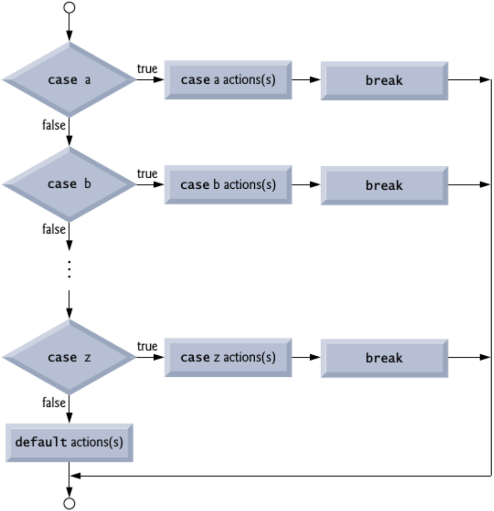
# Important Notes

- The getchar function from the standard input/output library reads and returns as an int one character from the keyboard.
- Characters are normally stored in variables of type char.
- Characters can be stored in any integer data type because they're usually represented as one-byte integers in the computer.
- **Therefore, we can treat a character as either an integer or a character, depending on its use**.
- Many computers today use the ASCII (American Standard Code for Information Interchange) character set. ASCII for lowercase letter 'a' is the integer 97.

# Important Notes

- Characters can be read with scanf by using the conversion specifier `%c`.
- Assignment expressions as a whole actually have a value. This value is assigned to the variable on the left side of the =.
- The fact that assignment statements have values can be useful for setting several variables to the same value, as in `a = b = c = 0;`.
- The `break` statement causes program control to continue with the statement after the `switch`.
- The `break` statement prevents the cases in a `switch` statement from running together.

# The switch loop



switch statement flowchart

# do ...while

```
#include <stdio.h>

int main(void)
{
  unsigned int counter = 1; // initialize counter
  do {
    printf("%u ", counter);
  } while (++counter <= 10); //semicolon is required here
}
```

# The break statement

- The break statement, when executed in a while, for, do ...while or switch statement, causes immediate exit from that statement.
- Program execution continues with the next statement.

```
unsigned int x; // declared here so it can be used after
for (x = 1; x <= 10; ++x) {
  if (x == 5) {
    break; // break loop only if x is 5
  }
  printf("%u ", x);
}
printf("\nBroke out of loop at x == %u\n", x);
```

# The continue statement

- The continue statement, when executed in a while, for or do ...while statement, skips the remaining statements in the body and performs the next loop iteration.
- In while and do ...while, the loop-continuation test is evaluated immediately after the continue statement is executed.
- In a for, the increment expression is executed, then the loop-continuation test is evaluated.

```
for (unsigned int x = 1; x <= 10; ++x) {
  if (x == 5) {
    continue; // skip remaining code in loop body
  }
  printf("%u ", x);
}
```

# Logical Operators

1. Logical AND is represented by &&.
2. Logical OR is represented by ||.
3. Logical NEGATION is represented by !.

| Exp1 | Exp2 | Exp1 && Exp2 | Exp1 \|\| Exp2 |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | nonzero | 0 | 1 |
| nonzero | 0 | 0 | 1 |
| nonzero | nonzero | 1 | 1 |

Precedence of AND is higher than OR.

| Exp | !Exp |
|---|---|
| 0 | 1 |
| nonzero | 0 |

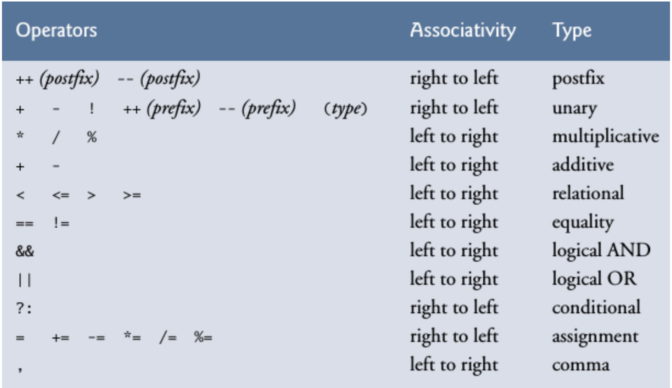
# Short-circuit evaluation of AND and OR

- An expression containing && or || operators is evaluated only until truth or falsehood is known.
- Evaluation of the condition

  ```
  gender == 1 && age >= 65
  ```

  will stop if gender is not equal to 1 since the whole expression will then be guaranteed to be false.

# Precedences of operators



| Operators | Associativity | Type |
|---|---|---|
| ++ *(postfix)* -- *(postfix)* | right to left | postfix |
| + - ! ++ *(prefix)* -- *(prefix)* (*type*) | right to left | unary |
| * / % | left to right | multiplicative |
| + - | left to right | additive |
| < <= > >= | 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 |

# Assignment vs. Equality

- Suppose we intend to write

  ```
  if (payCode == 4) {
    printf("%s", "You get a bonus!");
  }
  ```

  but we accidentally write

  ```
  if (payCode = 4) {
    printf("%s", "You get a bonus!");
  }
  ```

- The following things will happen:
  1. payCode will be *assigned* a value of 4,
  2. the expression payCode = 4 will return the value 4 irrespective of the actual value of payCode, and
  3. the command if(4) will be true.
- Tip: develop habit of the form 4 == payCode since accidentally writing it as 4 = payCode will give a *compilation error*.

# Assignment vs. Equality

- Suppose you want to assign a value to a variable with a simple statement such as

  ```
    x = 1;
  ```

  but instead write

  ```
  x == 1;
  ```

- Variable x will retain it's original value. It will not be assigned the value 1. Depending upon value of x, the expression will either return 0 or 1.
- Tip: find every instance of = in your code and check if it has been used correctly.