DASF004
Basic and Practice in Programming

Lecture 3
Basic Sequencing and Control 2

Projective Augmented Reality

Projecting computer images to reality

- https://www.youtube.com/watch?v=yBJEP4lsRFY

Agenda

Repetition
`while` loop
`for` loop
`do` … **`while`** loop
`switch` statement

Lab #1: Common Mistakes

- 1. Comparison Operator vs. Assignment Operator

```
int Number_1 = 10;
int Number_2 = 20;
if(Number_1 == Number2)
{  statement … ;
}
```

```
int Number_1 = 10;
int Number_2 = 20;
if(Number_1 = Number_2)
{    statement … ;
}
```

- One of the most frequently-made error!

- Accidentally confusing with the operators == (equality) and = (assignment).

- Do not ordinarily cause *compilation errors*! (Be vary careful!!!)
  - Statements with these errors ordinarily compile correctly
  - Allowing programs to run to completion
  - Likely generating incorrect results through *runtime logic errors (Sementic Error)*.

Lab #1: Common Mistakes (cont.)

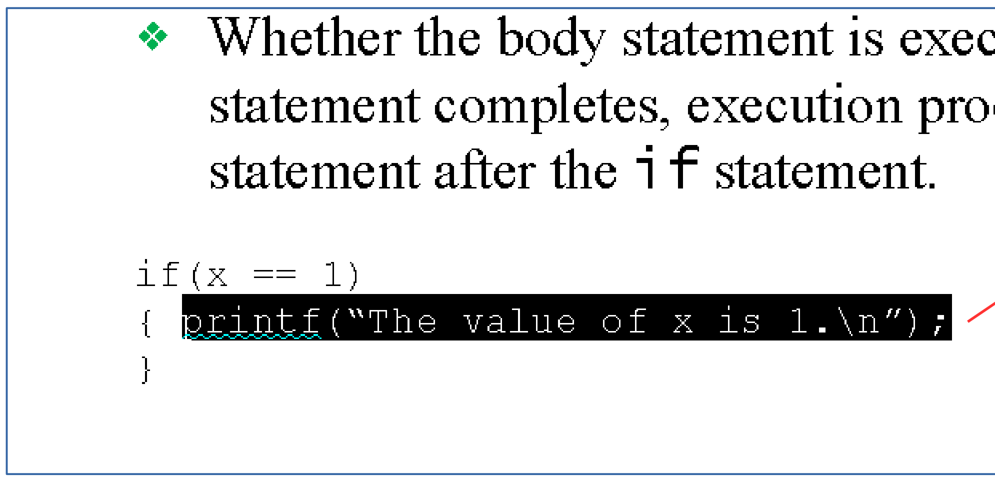
- 2. if statement vs. if … else statement

```
if (John < average)
{ printf(“John is below average”);
}
if (Jane < average)
{ printf(“Jane is below average“);
}
```

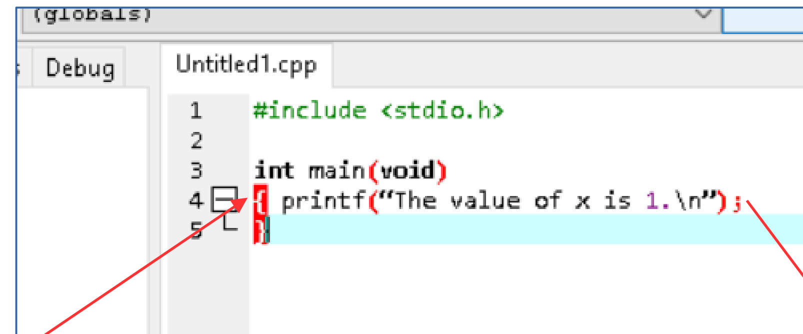
```
if(John < average)
   { printf(“John is below average”);
}
else if(Jane < average)
{ printf(“Jane is below average”);
}
```

Lab #1: Common Mistakes (cont.)

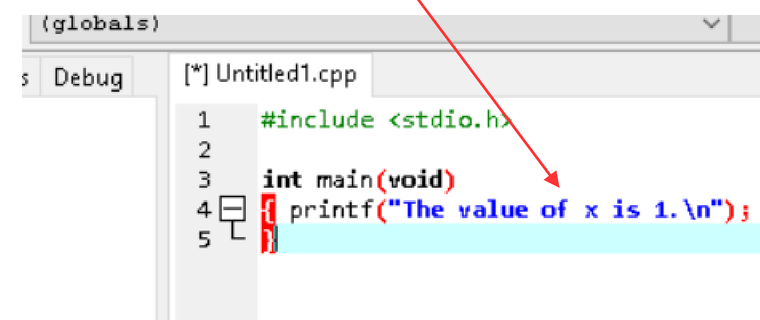
- 3. Copying and pasting quotation marks “



Powerpoint



Dev C++



Lab #1: Common Mistakes (cont.)

- 4. Calculating Average

```
int John = 88;
int Jane = 91;
int Peter = 94;
int Mary = 88;
float Average = (John + Jane + Peter + Mary)/4; //Average = 89, not 89.75!
```

- Problem???
- How to fix it???

Data Type: `short, unsigned, long`

- `short` or `int`: 16-bit integer

  `0000 0000 0000 0000, 0000 0000 0000 0001, … , 1111 1111 1111 1111`

  – Number of combination: $2^{16} = 65{,}536$
  – Range of integer a 16-bit integer can represent: [-32767, 32767]

- `long`: 32-bit integer

  `0000 0000 0000 0000 0000 0000 0000 0000 , … ,`
  `1111 1111 1111 1111 1111 1111 1111 1111`

  – Number of combination: $2^{32} = 4{,}294{,}967{,}296$
  – Range of integer a 32-bit integer can represent: [-2,147,483,648, 2,147,483,647]

- `long long`: 64-bit integer
  – Number of combination: $2^{64} = 18{,}446{,}744{,}073{,}709{,}551{,}616$
  – Range of integer a 64-bit integer can represent:
  [-9,223,372,036,854,775,808, 9,223,372,036,854,775,807]

- You can attach a keyword `unsigned` in front of `int` or `long`
  – `unsigned int; unsigned long`
  – Range of an 16-bit unsigned integer (`unsigned int`): [0, 65535]
  – Range of an 32-bit unsigned integer (`unsigned long`): [0, 4,294,967,295]

- `int` vs. `long` vs. `unsigned int` vs. `unsigned long`: What to use?
  – Estimate the range of your variables

Data Type: `char`

- Variable for storing one character
- e.g. `char x = “a”;`
- `char`: 8-bit

  `0000 0000, 0000 0001, … , 1111 1111`

  – Representing characters, mapping using the ASCII table

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

- `char` can also used to represent an 8-bit integer value

For example:

```
char x = 1;
char y = x + 3;
printf(“y = %d\n“, y); \\ y = 4
```

- Range of integer a `char` can represent:
  - 2^8 combination: 256 - [-127, 127]
- Keyword `unsigned` in front of `char`
  - `unsigned char`
  - Range of `unsigned char`: [0,255]

Data Type: `float, double, long double`

- `float`: 32-bit float point number
  - Range of value: -3.4^{38} to 3.4^{38}
- `double`: 64-bit float point number
  - Range of value: -1.7^{308} to 1.7^{308}
- `long double`: 128-bit float point number
  - Range of value: -1.7^{4932} to 1.7^{4932}

`put()` function

- `put()` function
  - Output text
  - e.g. `put("Hello, World!\n");`

- `printf()` function
  - Output text and/or variables value
  - e.g. `printf("Value of x: %d\n", x);`

- The follow two lines of code is identical
  - `put("Hello, World\n");`
  - `printf("Hello, World\n");`

Print out the value of variables

- If the type is not matched, an incorrect value will be printed

```
int x = 10;
float y = 1.5;
char z = 'A';
printf("All: %f, %f, %f\n", x, y, z);
```

```
C:\X\PortableApps\Dev-Cpp32\ConsolePause
All: 0.000000, 0.000000, 0.000000


Process exited normally.
Press any key to continue . . .
```

```
printf("All: %d, %d, %d\n", x, y, z);
```

```
C:\X\PortableApps\Dev-Cpp32\ConsolePauser.exe
All: 10, 0, 1073217536


Process exited normally.
Press any key to continue . . .
```

- %d – decimal value
- %f – float point value
- %c – character

Repetition Statements

- Repetition statement (also called iteration statement or loop)
  - if (`condition` is `TRUE`) then repeat the statement body
  - Stop when the `condition` is `FALSE`

- Example: Purchase items on a shopping list
  - Pseudo code example

```
while (there are more items on my shopping list)
 {   Purchase item
     Delete item off my list
 }
```

- Statement body is performed repeatedly while the `condition` is `TRUE`
- Repetition stops then `condition` is `FALSE`

`while` loop

```
while (condition)
{ statement body ...
  statement body ...
}
```

```
while (condition)
  statement body (1 line);
```

- Statement body will be executed repeatedly when `condition` is `TRUE`
- The loop stops when `condition` is `FALSE`
- The brace can be ommitted if there is only one line of code in the statement body

Repetition Statements (cont.)

- Two type of repetition statements:
  - Counter-controlled
    - How many times the loop will be executed is controlled by a counter variable
  - Sentinel-controlled
    - Indefinite repetition – it is not known ahead of time how many times the loop will be executed

Repetition Statements

Consider the following simple `while` loop, which prints the number 1 to 10. This is the simplest `while` loop!

```
#include <stdio.h>

int main (void)
{ int counter = 1;  // loop counter initialized to 1

  while (counter <= 10)
  { printf("%d\n", counter);
    counter++;
  }
}
```

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

Fig. 4.1 | Counter-controlled repetition. (Part 2 of 2.)

Repetition Statements

The last example can be further simplified as follow:

```
#include <stdio.h>

int main (void)
{ int counter = 0;  // loop counter initialized to 0

  while (++counter <= 10)
    printf("%d\n", counter);
}
```

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

Fig. 4.1 | Counter-controlled repetition. (Part 2 of 2.)

- This code saves a statement because the incrementing is done directly in the `while` condition before the condition is tested.
- Also, this code eliminates the need for the braces around the body of the `while` because the while now contains only one statement.
- Some programmers feel that this makes the code too cryptic and error prone.

Repetition Statements (cont.)

- Counter-controlled
  - How many times the loop will be executed is controlled by a counter variable

- Counter-controlled repetition requires:
  - The name of a control variable (or loop counter).
  - The initial value of the control variable.
  - The increment (or decrement) - the control variable is modified each time through the loop.
  - The condition that tests for the final value of the control variable (i.e., whether looping should continue).

- Example: Ask the user to enter scores of 10 students, and calculate the average
  - Pseudo code example:

```
int total = 0;
int counter = 1;
int score;
while (counter <= 10)
{  Get input from user and store in variable grade
   total = total + grade;
   counter++;
}
float Average = total / (float)(counter-1);
Print out average score
```

Using `while` Loop to Implement Counter-Controlled Repetition

```c
// Fig. 3.6: fig03_06.c
// Class average program with counter-controlled repetition.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   unsigned int counter; // number of grade to be entered next
   int grade; // grade value
   int total; // sum of grades entered by user
   int average; // average of grades

   // initialization phase
   total = 0; // initialize total
   counter = 1; // initialize loop counter

   // processing phase
   while ( counter <= 10 ) { // loop 10 times
      printf( "%s", "Enter grade: " ); // prompt for input
      scanf( "%d", &grade ); // read grade from user
      total = total + grade; // add grade to total
      counter = counter + 1; // increment counter
   } // end while

   // termination phase
   average = total /  (float) (counter-1);

   printf( "Class average is %d\n", average ); // display result
} // end function main
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81.7
```

Fig. 3.6 | Class-average problem with counter-controlled repetition. (Part 2 of 2.)

Repetition Statements (cont.)

- In previous example, you know you have 10 students in class
  - Create a while loop and repeat the body 10 times
- What if you don't know how many times you need to repeat???
- How can the program determine when to stop the input of grades? How will it know when to calculate and print the class average?
  - The programmer defines when to stop the input of grades!

- **<u>Sentinel-controlled Loop</u>**
  - Indefinite repetition – it is not known ahead of time how many times the loop will be executed
  - Example: (1) Ask the user to enter scores, and stop when user enter -1; (2) calculate the average score
  - Pseudo code example:

```
int score;
int total = 0, counter =0;
Ask user to input score
while (score != -1)
{ total = total + score;
   Ask user to input score
   counter++;
}
Calculate average and then print out
```

Example: sentinel-controlled repetition

```c
// Fig. 3.8: fig03_08.c
// Class-average program with sentinel-controlled repetition.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   unsigned int counter; // number of grades entered
   int grade; // grade value
   int total; // sum of grades

   float average; // number with decimal point for average

   // initialization phase
   total = 0; // initialize total
   counter = 0; // initialize loop counter

   // processing phase
   // get first grade from user
   printf( "%s", "Enter grade, -1 to end: " ); // prompt for input
   scanf( "%d", &grade ); // read grade from user

```

Fig. 3.8 | Class-average program with sentinel-controlled repetition. (Part 1 of 3.)

Example: sentinel-controlled repetition

```
   // loop while sentinel value not yet read from user
   while ( grade != -1 ) {
      total = total + grade; // add grade to total
      counter = counter + 1; // increment counter

      // get next grade from user
      printf( "%s", "Enter grade, -1 to end: " ); // prompt for input
      scanf("%d", &grade); // read next grade
   } // end while

   // termination phase
   // if user entered at least one grade
   if ( counter != 0 ) {

      // calculate average of all grades entered
      average = ( float ) total / counter; // avoid truncation

      // display average with two digits of precision
      printf( "Class average is %.2f\n", average );
   } // end if
   else { // if no grades were entered, output message
      puts( "No grades were entered" );
   } // end else
} // end function main
```

Fig. 3.8 | Class-average program with sentinel-controlled repetition. (Part 2 of 3.)

Example: sentinel-controlled repetition

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

```
Enter grade, -1 to end: -1
No grades were entered
```

Fig. 3.8 | Class-average program with sentinel-controlled repetition. (Part 3 of 3.)

Repetition Statements

- Three kinds of repetition statements in C
  - `while` statement
  - `for` statement
  - `do … while` statement
- Difference?
  - No difference functionally!
  - You can choose one of the three kind of statements to implement your logic
- Which statement to choose???
  - It is a matter of programming style!

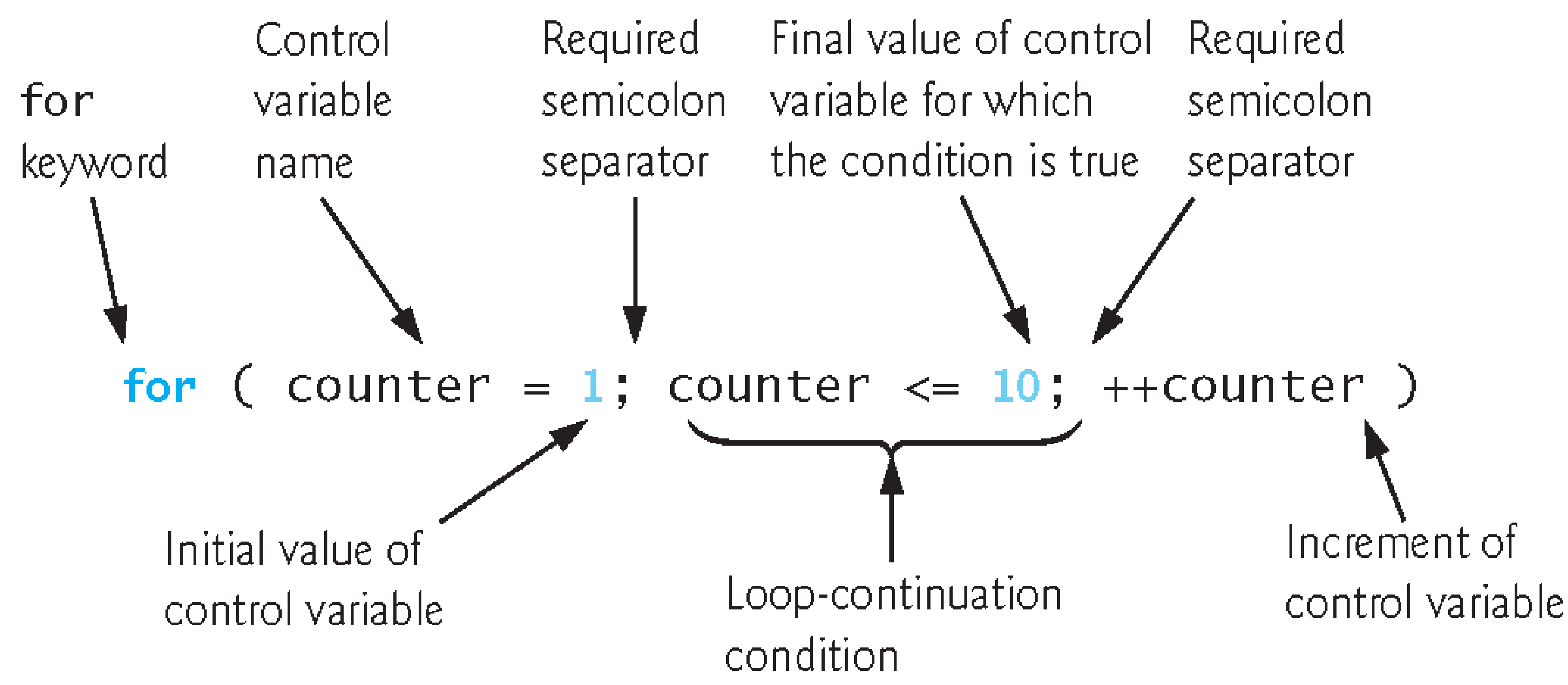


Fig. 4.3 | `for` statement header components.

`for` loop

- The `for` repetition statement handles all the details of counter-controlled repetition.
- To illustrate its power, let’s rewrite the program of Fig. 4.1.
- The result is shown in Fig. 4.2.

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

// function main begins program execution
int main( void )
{
   unsigned int counter; // define counter

   // initialization, repetition condition, and increment
   //  are all included in the for statement header.
   for ( counter = 1; counter <= 10; ++counter ) {
      printf( "%u\n", counter );
   } // end for
} // end function main
```

Fig. 4.2 | Counter-controlled repetition with the for statement.

`for` loop

Off-By-One Errors

- Notice that Fig. 4.2 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` loop

Expressions in the `for` Statement's Header Are Optional

- The three expressions in the `for` statement are optional.
- If *expression2* is omitted, C assumes that the condition is true, thus creating an infinite loop.
- You may omit *expression1* if the control variable is initialized elsewhere in the program.
- *expression3* may be omitted if the increment is calculated by statements in the body of the `for` statement or if no increment is needed.

```
for ( expression1; expression2; expression3 )
{
    statement
}
```

`for` loop

Increment Expression Acts Like a Standalone Statement

- The expression in the `for` statement acts like a stand-alone C statement at the end of the body of the `for`.

- Therefore, the expressions

```
counter = counter + 1
 counter += 1
 ++counter
 counter++
```

  are all equivalent in the increment part of the `for` statement.

- Some C programmers prefer the form `counter++` because the incrementing occurs after the loop body is executed, and the postincrementing form seems more natural.

- Because the variable being preincremented or postincremented here does not appear in a larger expression, both forms of incrementing have the same effect.

- The two semicolons in the `for` statement are required.

`for` loop

General Format of a `for` Statement

- The general format of the `for` statement is

```
for ( expression1; expression2; expression3 ) {
  statement
}
```

where *expression1* initializes the loop-control variable, *expression2* is the loop-continuation condition, and *expression3* increments the control variable.

- In most cases, the `for` statement can be represented with an equivalent `while` statement as follows:

```
expression1;
while ( expression2 ) {
  statement
  expression3;
}
```

`for` loop

- The following examples show methods of varying the control variable in a `for` statement.
  - 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` loop

Application: Summing the Even Integers from 2 to 100

- Figure 4.5 uses the `for` statement to sum all the even integers from `2` to `100`.

```
// Fig. 4.5: fig04_05.c
// Summation with for.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   unsigned int sum = 0; // initialize sum
   unsigned int number; // number to be added to sum

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

   printf( "Sum is %u\n", sum ); // output sum
} // end function main
```

```
Sum is 2550
```

Fig. 4.5 | Summation with `for`.

`do...while` loop

- The `do…while` repetition statement is similar to the `while` statement.
- In the `while` statement, the loop-continuation condition is tested at the beginning of the loop before the body of the loop is performed.
- The `do…while` statement tests the loop-continuation condition *after* the loop body is performed.
- Therefore, the loop body will be executed at least once.
- When a `do…while` terminates, execution continues with the statement after the `while` clause.

```
do
{    statement;
     statement;
} while ( condition );
```

`do…while` loop

- It's not necessary to use braces in the `do…while` statement if there's only one statement in the body.
- However, the braces are usually included to avoid confusion between the `while` and `do…while` statements.
- For example,

```
while ( condition )
```

is normally regarded as the header to a `while` statement.

- A `do…while` with no braces around the single-statement body appears as

```
do
    statement
while ( condition );
```

which can be confusing.

- The last line—`while(` *condition* `);`—may be misinterpreted as a `while` statement containing an empty statement.
- Thus, to avoid confusion, the `do…while` with one statement is often written as follows:

do...while loop

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

// function main begins program execution
int main( void )
{
   unsigned int counter = 1; // initialize counter

   do {
      printf( "%u  ", counter ); // display counter
   } while ( ++counter <= 10 ); // end do...while
} // end function main
```

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

Fig. 4.9 | Using the do...while repetition statement.

Nested Loops

- Loops insider a loop

```
for(int x=0; x<=9; x++)
{  for(int y=0; y<=4; y++)
   {   for(int z=0; z<=7; z++)
       {  printf("Hello,World!\n");  // How many times is this executed???
       }
   }
}
```

- How many times is “Hello, World!\n” printed???

Consider the following code segment:

```
 for (int i = 1; i <= 6; i++) {
    for (int j = 1; j <= i; j++) {
       put("*");
    }
    put("\n");
 }
```

- What is the output of the code segment???

`switch` multiple-selection statement

- Occasionally, an algorithm will contain a *series of decisions* in which a variable or expression is tested separately for each of the constant integral values it may assume, and different actions are taken. `if … else if` statement:

```
if(x == 0)
{ printf("0");
}
else if(x == 1)
{ printf("1");
}
else if(x == 2)
{ printf("2");
}
else
{ printf("***");
}
```

```
switch(x)
{ case 0:
    printf("0");
    break;
  case 1:
    printf("1");
    break;
  case 2:
    printf("2");
    break;
  default:
    printf("***");
}
```

- This is called *multiple selection*.
- C provides the `switch` multiple-selection statement to handle such decision making.
- The `switch` statement consists of a series of `case` labels, an optional `default` case and statements to execute for each case.
- Figure 4.7 uses `switch` to count the number of each different letter grade students earned on an exam.

```c
// Fig. 4.7: fig04_07.c
// Counting letter grades with switch.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   int grade; // one grade
   unsigned int aCount = 0; // number of As
   unsigned int bCount = 0; // number of Bs
   unsigned int cCount = 0; // number of Cs
   unsigned int dCount = 0; // number of Ds
   unsigned int fCount = 0; // number of Fs

   puts( "Enter the letter grades."  );
   puts( "Enter the EOF character to end input."  );

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

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

```

Fig. 4.7 | Counting letter grades with switch. (Part 1 of 4.)

```c
         case 'A': // grade was uppercase A
         case 'a': // or lowercase a
            ++aCount; // increment aCount
            break; // necessary to exit switch

         case 'B': // grade was uppercase B
         case 'b': // or lowercase b
            ++bCount; // increment bCount
            break; // exit switch

         case 'C': // grade was uppercase C
         case 'c': // or lowercase c
            ++cCount; // increment cCount
            break; // exit switch

         case 'D': // grade was uppercase D
         case 'd': // or lowercase d
            ++dCount; // increment dCount
            break; // exit switch

         case 'F': // grade was uppercase F
         case 'f': // or lowercase f
            ++fCount; // increment fCount
            break; // exit switch

```

Fig. 4.7 | Counting letter grades with switch. (Part 2 of 4.)

```c
         case '\n': // ignore newlines,
         case '\t': // tabs,
         case ' ': // and spaces in input
            break; // exit switch

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

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

Fig. 4.7 | Counting letter grades with switch.

```
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
```

Fig. 4.7 | Counting letter grades with switch. (Part 4 of 4.)

`switch` multiple-selection statement

Reading Character Input

- In the program, the user enters letter grades for a class.
- In the `while` header (line 19),
  - `while ( ( grade = getchar() ) != EOF )`
- the parenthesized assignment `(grade = getchar())` executes first.
- The `getchar` function (from `<stdio.h>`) reads one character from the keyboard and stores that character in the integer variable `grade`.
- Characters are normally stored in variables of type `char`.

`break` and `continue` statements

- The `break` and `continue` statements are used to alter the flow of control.

***break* Statement**

- The `break` statement, when executed in a `while`, `for`, `do`…`while` or `switch` statement, causes an immediate exit from that statement.
- Program execution continues with the next statement.
- Common uses of the `break` statement are to escape early from a loop or to skip the remainder of a `switch` statement (as in Fig. 4.7).

break and continue statements

```c
// Fig. 4.11: fig04_11.c
// Using the break statement in a for statement.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   unsigned int x; // counter

   // 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
      } // end if

      printf( "%u ", x ); // display value of x
   } // end for

   printf( "\nBroke out of loop at x == %u\n", x );
} // end function main
```

Fig. 4.11 | Using the break statement in a for statement. (Part 1 of 2.)

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

Fig. 4.11 | Using the break statement in a for statement. (Part 2 of 2.)

`break` and `continue` statements

`continue` Statement

- The `continue` statement, when executed in a `while`, `for` or `do`…`while` statement, skips the remaining statements in the body of that control statement and performs the next iteration of the loop.
- In `while` and `do`…`while` statements, 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 and continue statements

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

// function main begins program execution
int main( void )
{
   unsigned int x; // counter

   // loop 10 times
   for ( 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
      } // end if

      printf( "%u ", x ); // display value of x
   } // end for

   puts( "\nUsed continue to skip printing the value 5" );
} // end function main
```

Fig. 4.12 | Using the continue statement in a for statement. (Part 1 of 2.)

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

Fig. 4.12 | Using the continue statement in a for statement. (Part 2 of 2.)

A Note on Switch Statement

Consider this code segment:

- The `switch` statement compares variable `result`

If `result == 10`, start from here,
Until you reach `break;`

If `result == 9`, start from here,
Until you reach `break;`

If `result == 8`, start from here,
Until you reach `break;`

If `result == 7`, start from here,
Until you reach `break;`

…

The default statement does not need a break statement, because it is the last statement.

- The `break` statement jumps to the end

```
int score, result;
char grade;
printf("점수입력: ");
scanf("%d", &score);
result = score / 10;
switch(result){
  case 10:
  case 9:
  grade = 'A';
  break;
  case 8:
  grade = 'B';
  break;
  case 7:
  grade = 'C';
  break;
  case 6:
  grade = 'D';
  break;
  default:
  grade = 'F';
}
printf("%d Score => %c Grade\n", score, grade);
```

`switch` statement vs. `if … else if` statement

- Difference between `switch` statement and `if … else if` statement
  - `if … else if` is a simple logic
  - `switch` statement uses a look up table

- So `switch` statement tend to be faster
  - Especially when the number of options is large

Logical operators

- C provides *logical operators* that may be 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 operators

Logical AND (&&) Operator

- Suppose we wish to ensure that two conditions are both true before we choose a certain path of execution.
- In this case, we can use the logical operator `&&` as follows:

```
if ( gender == 1 && age >= 65 )
   ++seniorFemales;
```

- This `if` statement contains *two* simple conditions.
- The condition `gender == 1` might be evaluated, for example, to determine if a person is a female.
- The condition `age >= 65` is evaluated to determine whether a person is a senior citizen.
- The two simple conditions are evaluated first because the precedences of `==` and `>=` are both *higher* than the precedence of `&&`.

Logical operators

Logical OR (||) Operator

- Now let's consider the `||` (logical OR) operator.
- Suppose we wish to ensure at some point in a program that *either or both* of two conditions are *true* before we choose a certain path of execution.
- In this case, we use the `||` operator as in the following program segment

```
if ( semesterAverage >= 90 || finalExam >= 90 )
   printf( "Student grade is A" );:
```

- This statement also contains two simple conditions.
- The condition `semesterAverage >= 90` is evaluated to determine whether the student deserves an "A" in the course because of a solid performance throughout the semester.

Logical operators

Logical Negation (!) Operator

- C provides `!` (logical negation) to enable you to “reverse” the meaning of a condition.
- Unlike operators `&&` and `||`, which combine two conditions (and are therefore binary operators), the logical negation operator has only a single condition as an operand (and is therefore a unary operator).
- The logical negation operator is placed before a condition when we’re interested in choosing a path of execution if the original condition (without the logical negation operator) is false, such as in the following program segment:

```
if ( !( grade == sentinelValue ) )
   printf( "The next grade is %f\n", grade );
```

- The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has a higher precedence than the equality operator.

Logical operators

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

Fig. 4.13 | Truth table for the logical AND (&&) operator.

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

Fig. 4.14 | Truth table for the logical OR (||) operator.

expression	!expression
0	1
nonzero	0

Fig. 4.15 | Truth table for operator ! (logical negation).

Logical operators

Both operators associate from left to right.

```
if ( a == 3 && b == 4 && c == 5 ) { … }
```

equivalent to

```
if (((a == 3) && (b ==4)) && (c == 5)) { … } } }
```

- The && operator has a higher precedence than ||.

```
if ( a == 3 || b == 4 && c == 5 ) { … }
```

equivalent to

```
if (a == 3 || (b ==4 && c == 5)) { … }
```

Logical operators

- An expression containing `&&` or `||` operators is evaluated only until truth or falsehood is known.
- Thus, evaluation of the condition

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

- will stop if `gender` is not equal to `1` (i.e., the entire expression is false), and continue if `gender` is equal to `1` (i.e., the entire expression could still be true if `age >= 65`).
- This performance feature for the evaluation of logical AND and logical OR expressions is called short-circuit evaluation.

Q&A?