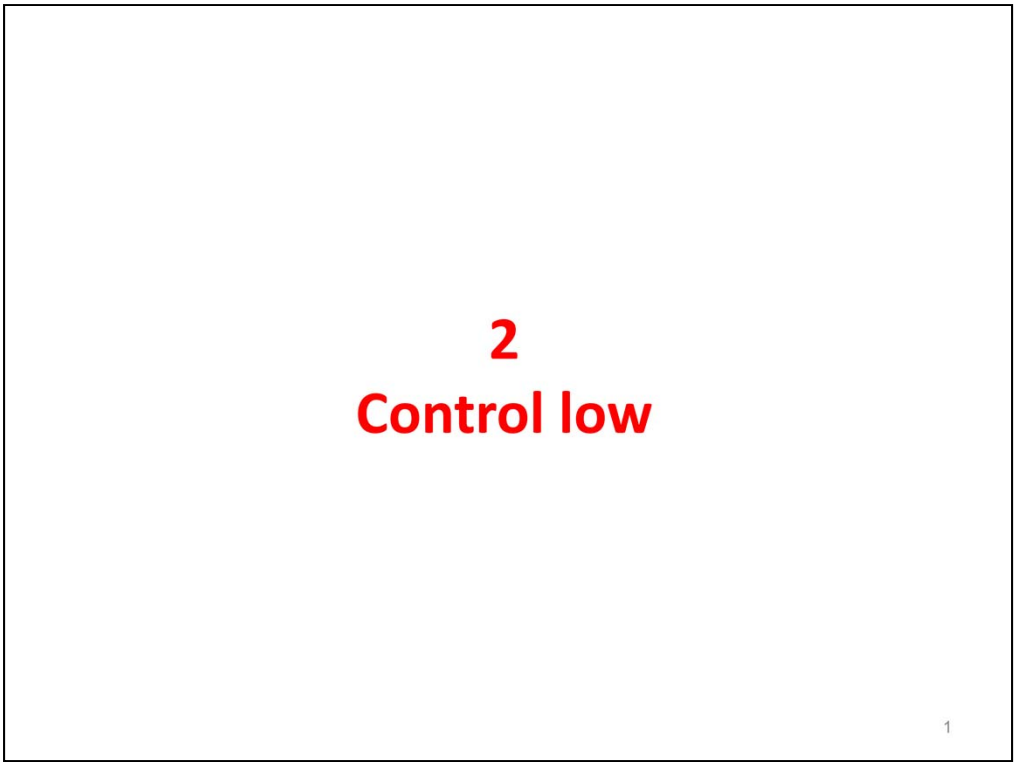


Basic C Programming

1. In this lecture, we discuss the control flow.

Control Flow

- **Relational Operator and Logical Operator**
- Branching: if, if…else, if….else if….else Statements; Nested if Statements; The switch Statement; Conditional Operators
- Looping: for, while, do-while; Nested Loops; The break and continue Statements
- Self-Learning Programming Example

2

Control Flow

1. We first discuss the relational operators and logical operators.
2. Then, we discuss the branching statements and looping statements.
3. Here, we start by discussing the relational operators and logical operators.

Relational and Logical Operators

- **Relational Operators:** used for **comparison** between **two values**.

operator	example	meaning
==	ch == 'a'	equal to
!=	f != 0.0	not equal to
<	num < 10	less than
<=	num <=10	less than or equal to
>	f > -5.0	greater than
>=	f >= 0.0	greater than or equal to

- **Logical Operators: Involving one or more relational expressions**

operator	example	meaning
!	!(num < 0)	not
&&	(num1 > num2) && (num2 >num3)	and
||	(ch == '\t') || (ch == ' ')	or

- **Precedence of Operators – determines the order of operator execution [please refer to the precedence table in lecture note.]**

3

Relational Operators

1. Relational operators are used for comparison between two values. Relational operators include equal to (==), not equal to (!=), less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=). They perform computations on their operands and return the result as either true or false. If the result is true, an integer value of 1 is returned, and if the result is false, then the integer value of 0 is returned.
2. Logical operators allow testing and combining the results of comparison expressions. For logical operators, both logical **and** (**&&**) and logical **or** (**||**) operators are binary operators, while the logical **not** operator (**!**) is a unary operator. Logical operators allow testing and combining the results of comparison expressions.
3. Logical **not** operator (**!**) returns true when the operand is false and returns false when the operand is true. Logical **and** operator (**&&**) returns true when both operands are true, otherwise it returns false.
4. Logical **or** operator (**||**) returns false when both operands are false, otherwise it returns true.
5. Operator precedence determines the order of operator execution.

In a branching operation, the decision on which statements to be executed is based on a comparison between two values. Relational operators are used for comparison between two values. Logical operators work on one or more relational expressions to yield either

the logical value true or false.

Operator Precedence

Operator precedence determines the order of operator execution. The list of operators of **decreasing precedence** is given below:

!	not
* /	multiply and divide
+ -	add and subtract
< <= > >=	less, less or equal, greater, greater or equal
== !=	equal, not equal
&&	logical and
||	logical or

The logical **not** (**!**) operator has the highest priority. It is followed by the multiplication, division, addition and subtraction operators. The logical **and** (**&&**) and **or** (**||**) operators have a lower priority than the relational operators.

Boolean Evaluation in C

- The result of evaluating an expression involving relational and/or logical operators is either true or false.
  - true is 1
  - false is 0
- In general, any integer expression whose value is non-zero is considered true; else it is false. For example:

3	is true
0	is false
1 && 0	is false
1 || 0	is true
!(5 >= 3) || (1)	is true

4

Boolean Evaluation

1. The result of evaluating an expression involving relational and/or logical operators is either 1 or 0. When the result is true, it is 1. Otherwise it is 0.
2. Since C uses 0 to represent a false condition, any integer expression whose value is *non-zero* is considered *true*; otherwise it is *false*.
3. Therefore, 3 is true, and 0 is false. (1 && 0) is false and (1 || 0) is true.

Another example is given in the following program to show the logic values of relational and logical expressions.

```
#include <stdio.h>
int main()
{
    float result;
    printf("The results of the logic relations:\n");
    result = (3 > 7);
    printf("(3 > 7) is %f\n", result);
    result = (7 < 3) && (3 <= 7);
    printf("(7 < 3) && (3 <= 7) is %f\n", result);
```

```
    result = (7 < 3) && (7/0 <= 5);
    printf("(7 < 3) && (7/0 <= 5) is  %f\n", result);
    result = (32/4 > 3*4) || (4 == 4);
    printf("(32/4 > 3*4) || (4 == 4) is %f\n", result);
    result = (4 == 4) || (32/0 == 0);
    printf("(4 == 4) || (32/0 == 0) is %f\n", result);
    return 0;
}
```

The variable **result** is defined as type **float**. When the variable is printed with the specifier "**%f**", the true value to be printed will be 1.000000, and the false value to be printed will be 0.000000. The results are straightforward except the two special cases involving the evaluation of expressions consisting of logical **or** and logical **and** operators. First, consider the evaluation of the following statement involving logical **and** operator:

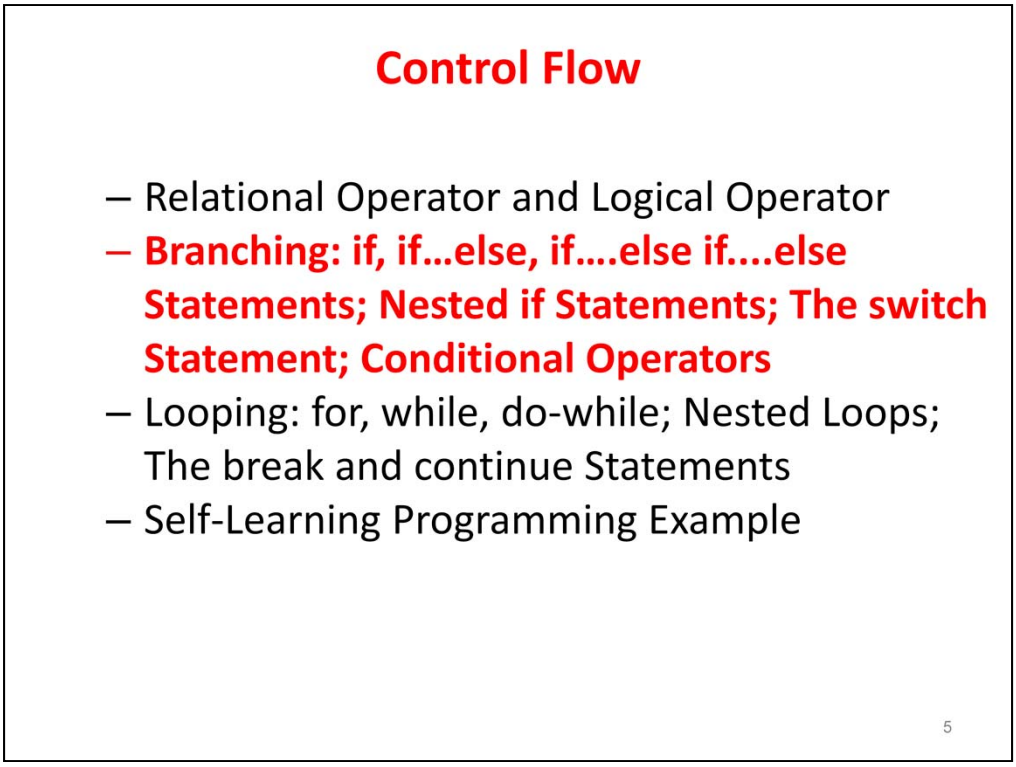
```
result = (7 < 3) && (7/0 <= 5);
```

When the first relational expression is evaluated to be false, the second relational expression does not need to be evaluated. Therefore, even though the second expression contains an error in the evaluation of **7/0**, it does not occur in the overall result. This is called *short-circuit evaluation* in which the second argument is only executed or evaluated if the first argument does not suffice to determine the value of the expression.

Another similar case is also occurred during the evaluation of the following statement containing the logical **or** operator:

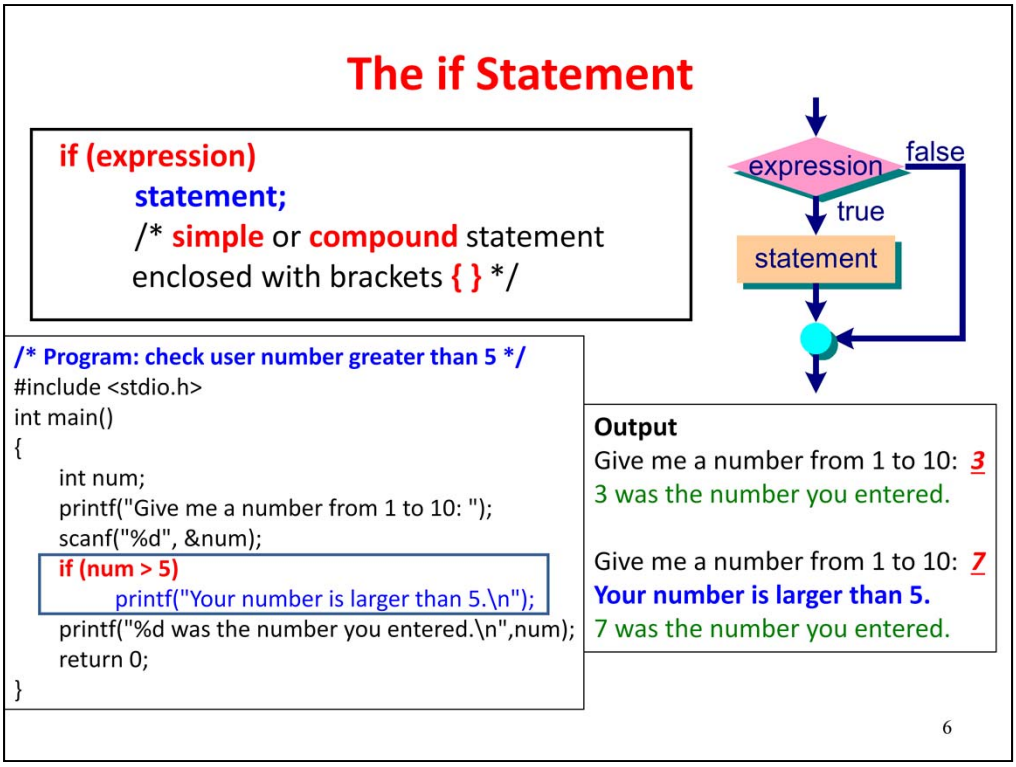
```
result = (4 == 4) || (32/0 == 0);
```

Since the first expression is evaluated to be true, the second expression does not need to be evaluated. Hence, the error does not occur in the overall result.



Control Flow

1. Next, we discuss the branching statements.



The if Statement

1. The simplest form of the **if** statement is

   if (expression)

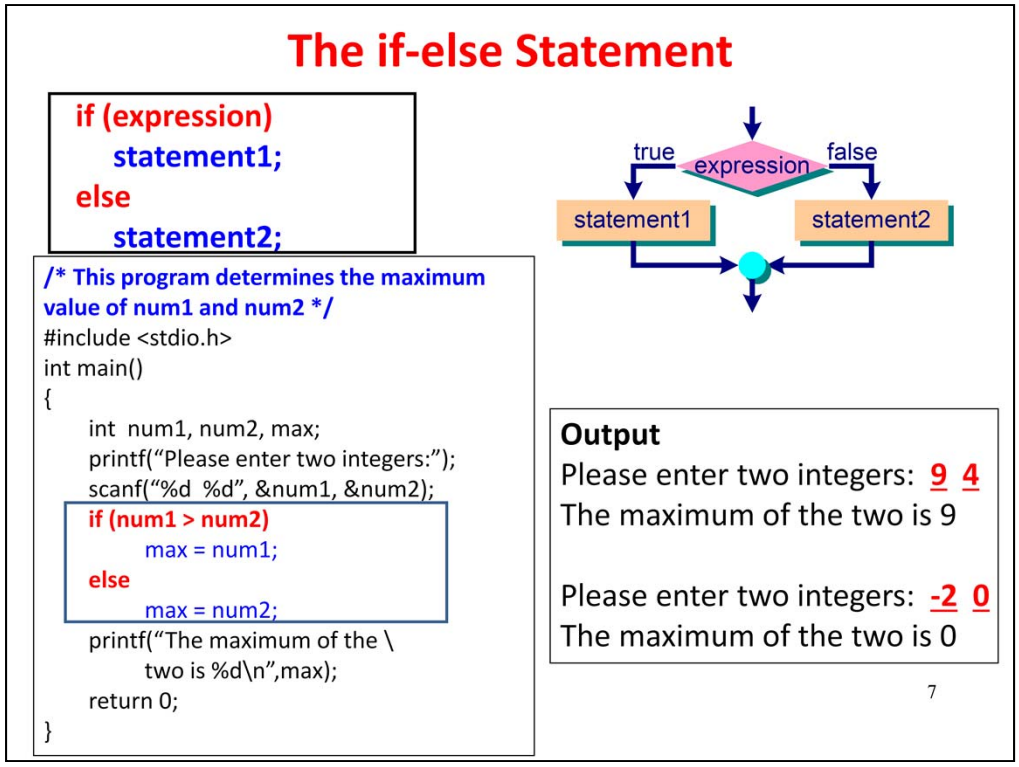
   statement;

   if is a reserved keyword.

2. If the **expression** is evaluated to be true (i.e. non-zero), then the **statement** is executed. If the **expression** is evaluated to be false (i.e. zero), then the **statement** is ignored, and the control is passed to the next program statement following the **if** statement.

3. The **statement** may be a single statement terminated by a semicolon or a compound statement enclosed by braces.

4. In the example program, it asks the user to enter a number from 1 to 10, and then reads the user input. The expression in the **if** statement contains a relational operator. It checks to see whether the user input is greater than 5. If the relational expression is false, then the subsequent **printf()** statement is not executed. Otherwise, the **printf()** statement will print the string **"You number is larger than 5."** on the screen. Finally, the last **printf()** statement will be executed to print the number entered by the user on the screen.



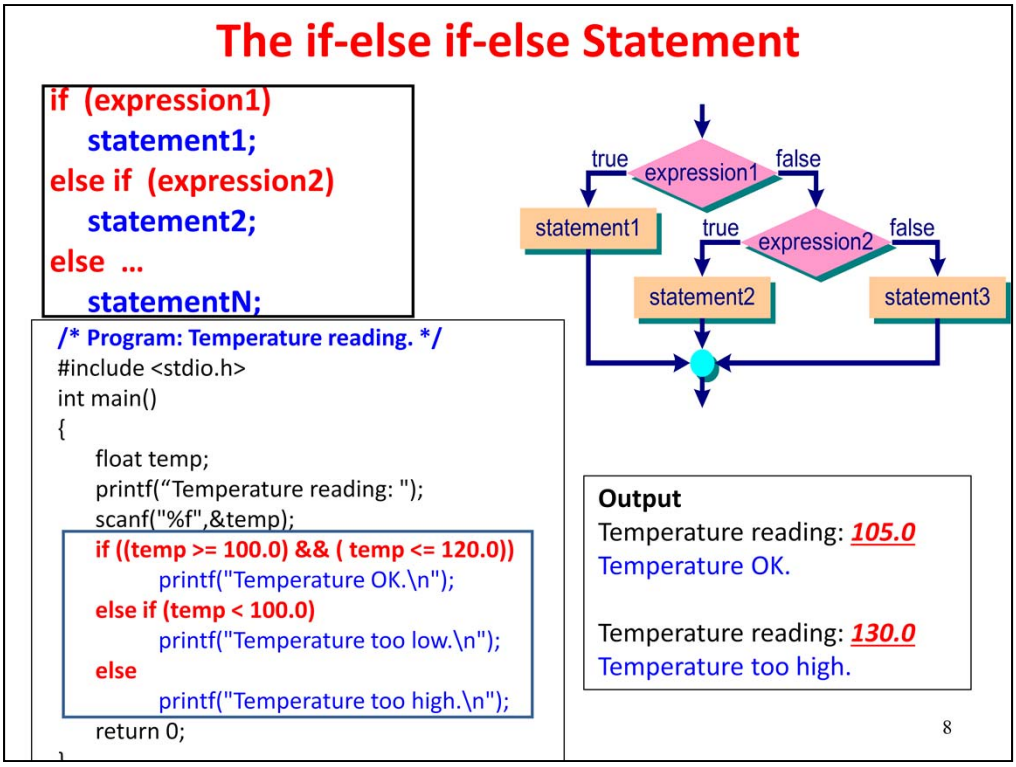
The if-else Statement

1. The **if-else** statement implements a two-way selection. The format of the **if-else** statement is

```
if (expression)
    statement1;
else
    statement2;
```

   if and **else** are reserved keywords.

2. When the **if-else** statement is executed, the **expression** is evaluated. If **expression** is true, then **statement1** is executed and the control is passed to the program statement following the **if** statement. If **expression** is false, then **statement2** is executed.
3. Both **statement1** and **statement2** may be a single statement terminated by a semicolon or a compound statement enclosed by **{}**.
4. In the example program, it computes the maximum number of two input integers. The two input integers are read in and stored in the variables **num1** and **num2**. The **if** statement is then used to compare the two variables. If **num1** is greater than **num2**, then the variable **max** is assigned with the value of **num1**. Otherwise, **max** is assigned with the value of **num2**. The program then prints the maximum number through the variable **max**.



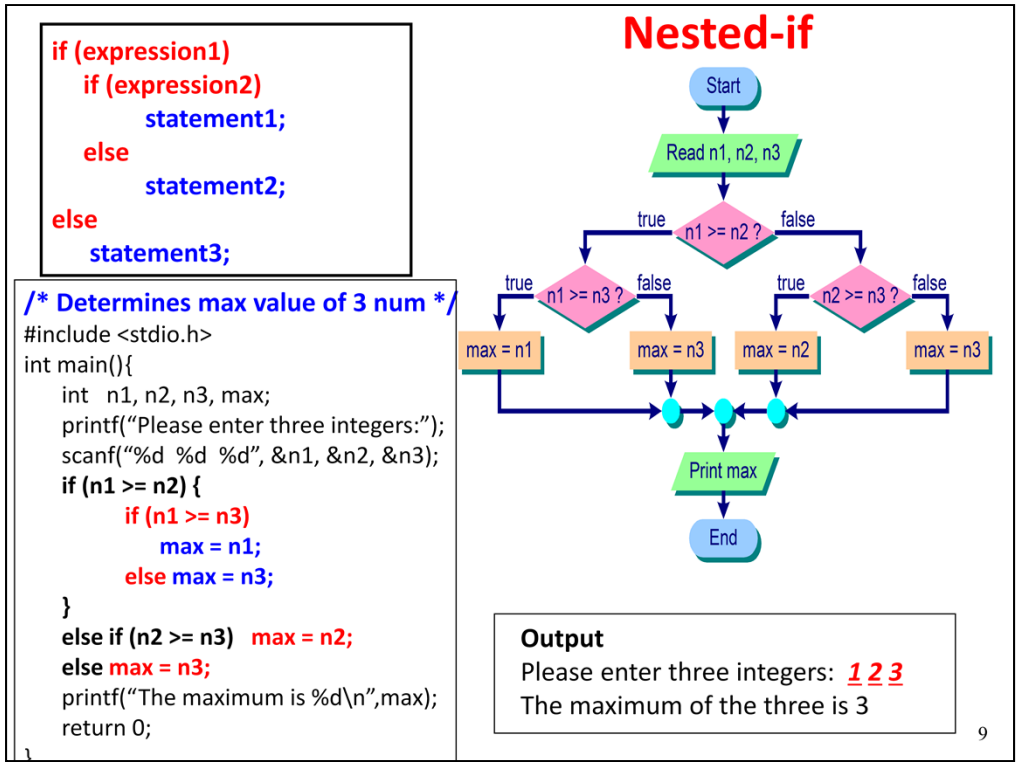
The if-else-if-else Statement

1. The format for **if-else if-else** statement is

```
if (expression1)
    statement1;
else if (expression2)
    statement2;
else
    statement3;
....
else
    statementN;
```

2. Each of the **statement1**, **statement2**, **statement3**, etc. can either be a single statement terminated by a semicolon or a compound statement enclosed by **{}**.
3. If all the statements are false, then the last **statement** will be executed. In any case, only one statement will be executed, and the rest will be skipped. The last **else** part is optional and can be omitted. If the last **else** part is omitted, then no statement will be executed if all the expressions are evaluated to be false.
4. In the example program, it first reads in the temperature input. If the input value is between 100 and 120, the string **"Temperature OK."** will be printed the screen, else if

the input value is less than 100, the string **“Temperature too low”** will be displayed, else the string **“Temperature too high”** will be printed.



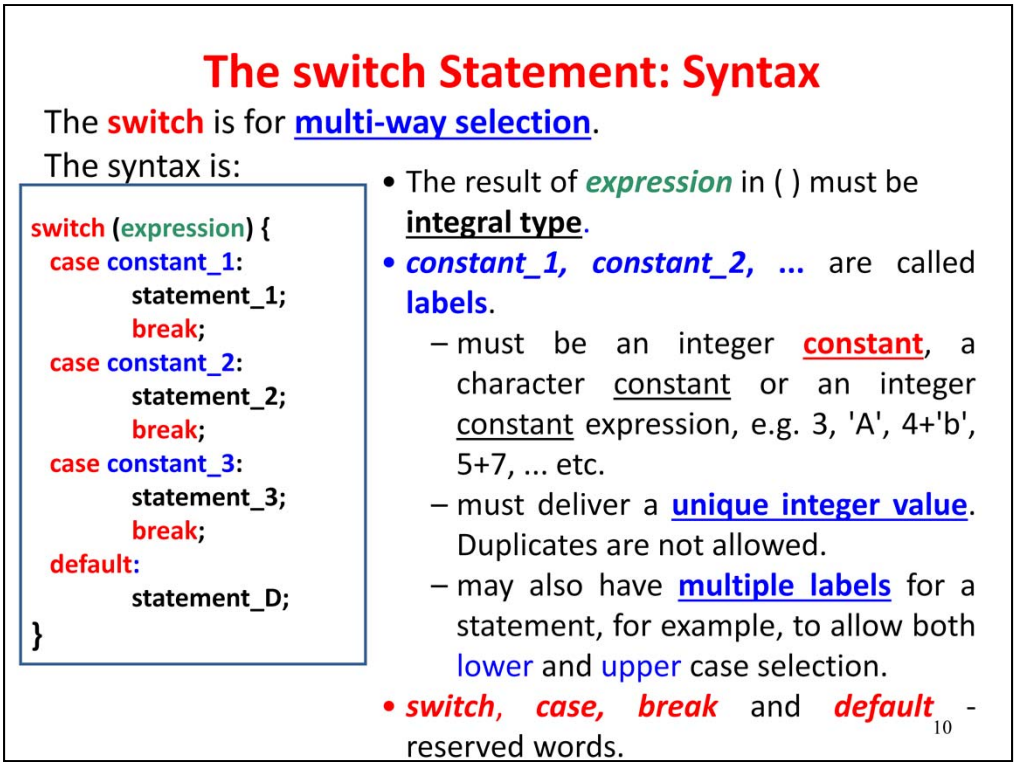
Nested-if

1. The nested-**if** statement allows us to perform a multi-way selection. In a nested-**if** statement, both the **if** branch and the **else** branch may contain one or more **if** statements. The level of nested-**if** statements can be as many as the limit the compiler allows.
2. An example of a nested-**if** statement is given as follows:

```
if (expression1)
    if (expression2)
        statement1;
    else
        statement2;
else
    statement3;
```

3. If **expression1** and **expression2** are true, then **statement1** is executed. If **expression1** is true and **expression2** is false, then **statement2** is executed. If **expression1** is false, then **statement3** is executed. C compiler associates an **else** part with the nearest unresolved **if**, i.e. the **if** statement that does not have an **else** statement. We can also use braces to enclose statements.
4. In the example program, it reads in three integers from the user and stores the values

in the variables **n1**, **n2** and **n3**. The values stored in **n1** and **n2** are then compared. If **n1** is greater than **n2**, then **n1** is compared with **n3**. If **n1** is greater than **n3**, then **max** is assigned with the value of **n1**. Otherwise, **max** is assigned with the value of **n3**. On the other hand, if **n1** is less than **n2**, then **n2** is compared with **n3** in a similar manner. The variable **max** is assigned with the value based on the comparison between **n2** and **n3**. Finally, the program will print the maximum value of the three integers via the variable **max**.



The switch Statement: Syntax

1. The **switch** statement provides a multi-way decision structure in which one of the several statements is executed depending on the value of an expression.
2. The syntax of a **switch** statement is given:

```
switch (expression) {
    case constant_1:
        statement_1;
        break;
    case constant_2:
        statement_2;
        break;
    case constant_3:
        statement_3;
        break;
    ....
    default:
        statement_D;
}
```

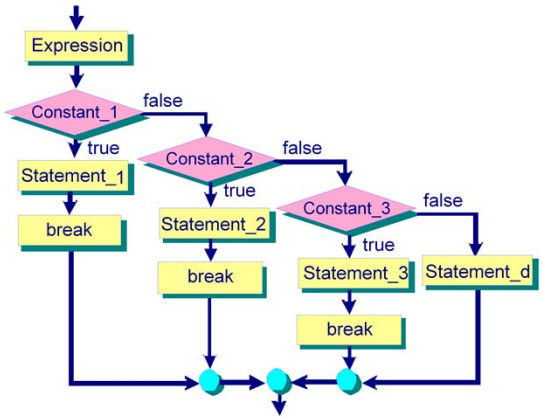
3. **switch**, **case**, **break** and **default** are reserved keywords. **constant_1**, **constant_2**, etc. are called *labels*. Each must be an integer constant, a character constant or an integer constant expression, e.g. 3, 'A', 4+'b', 5+7, etc. Each of the labels must deliver a unique integer value. Duplicates are not allowed. Multiple labels are also allowed, for example, to support both lower and upper case selection.

The switch Statement: Execution

The switch is for multi-way selection.

The syntax is:

```
switch (expression) {
  case constant_1:
          statement_1;
          break;
  case constant_2:
          statement_2;
          break;
  case constant_3:
          statement_3;
          break;
  default:
          statement_D;
}
```



- Note: The **expression** must return an **integer constant** or **character constant**, and does not support a range of values to be specified.

11

The switch Statement: Execution

1. In the **switch** statement, **statement** is executed only when the **expression** has the corresponding value of **constant**.
2. The **default** part is optional. If it is there, **statement_D** is executed when **expression** has a value different from all the values specified by all the cases.
3. The **break** statement signals the end of a particular case and causes the execution of the **switch** statement to be terminated.
4. Each of the **statement** may be a single statement terminated by a semicolon or a compound statement enclosed by **{}**.
5. There are several restrictions in the use of the **switch** statement. The **expression** of the **switch** statement must return a result of *integer* or *character* data type. The value in the **constant** label part must be an integer constant or character constant. Moreover, it does not support a range of values to be specified.

Switch: An Example

```
/* Arithmetic (A,S,M) computation of two user numbers */
#include <stdio.h>
 int main() {
    char choice;   int num1, num2, result;
    printf("Enter your choice (A, S or M) => ");
    scanf("%c", &choice);
    printf("Enter two numbers: ");
    scanf("%d %d", &num1, &num2);
    switch (choice)     {
      case 'a':
      case 'A':   result = num1 + num2;
        printf(" %d + %d = %d\n", num1,num2,result);
        break;
      case 's':
      case 'S':   result = num1 - num2;
        printf(" %d - %d = %d\n", num1,num2,result);
        break;
      case 'm':
      case 'M':   result = num1 * num2;
        printf(" %d * %d = %d\n", num1,num2,result);
        break;
      default :   printf("Not one of the proper choices.\n");
    }
    return 0;
 }
```

Output

```
Enter your choice (A, S or M) => S
Enter two numbers: 9 5
9 – 5 = 4
```

12

The switch Statement: Example

1. The **switch** statement is quite commonly used in menu-driven applications.
2. In this example program, it uses the **switch** statement for menu-driven selection. The program displays a list of arithmetic operations (i.e. addition, subtraction and multiplication) that the user can enter. Then, the user selects the operation command and enters two operands.
3. The **switch** statement is then used to control which operation is to be executed based on user selection. The control is transferred to the appropriate branch of the **case** condition based on the variable **choice**.
4. The statements under the **case** condition are executed and the result of the operation will be printed.
5. In addition, we may also have *multiple labels* for a statement. As such, we can allow the choice to be specified in both lowercase and uppercase letters entered by the user.

If-else: Example

```
/* Arithmetic (A,S,M) computation of two user numbers */
#include <stdio.h>
 int main() {
    char choice;   int num1, num2, result;
    printf("Enter your choice (A, S or M) => ");
    scanf("%c", &choice);
    printf("Enter two numbers: ");
    scanf("%d %d", &num1, &num2);
    if ( (choice == 'a')  || (choice == 'A'))    {
        result = num1 + num2;
        printf(" %d + %d = %d\n", num1,num2,result);
     }
     else if ((choice == 's')  || (choice == 'S'))
        result = num1 - num2;
        printf(" %d - %d = %d\n", num1,num2,result);
     }
     else if ((choice == 'm')  || (choice == 'M') )
        result = num1 * num2;
        printf(" %d * %d = %d\n", num1,num2,result);
     }
       else printf("Not one of the proper choices.\n");
    return 0;
 }
```

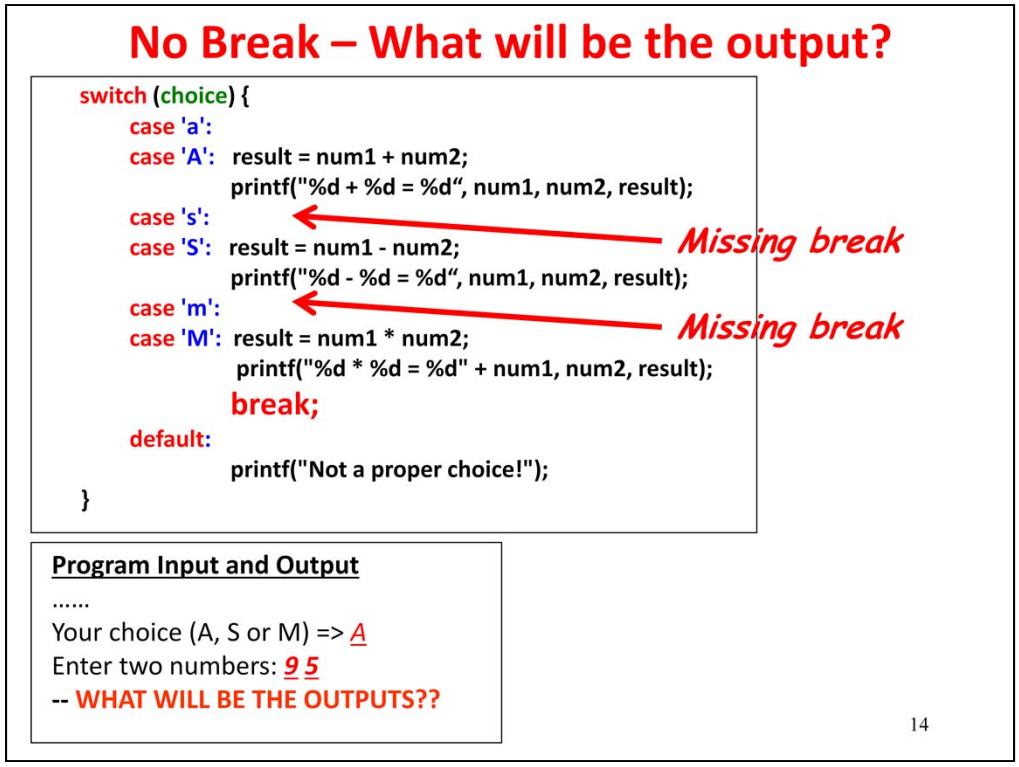
Output

```
Enter your choice (A, S or M) => S
Enter two numbers: 9 5
9 – 5 = 4
```

13

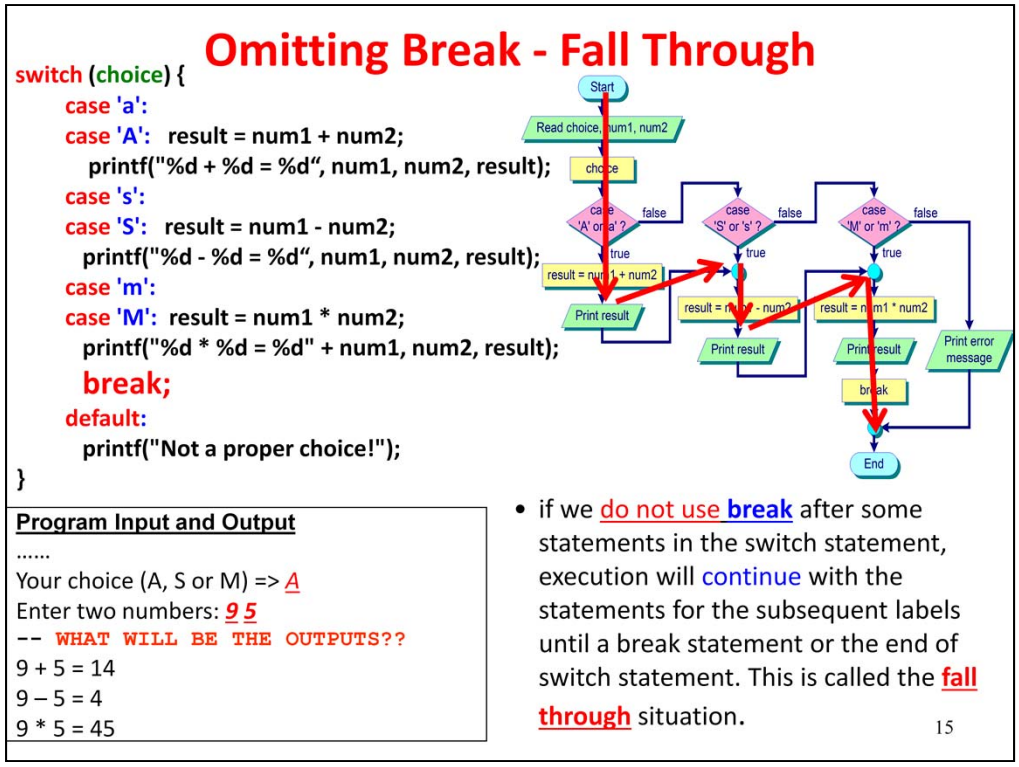
The if-else-if-else statement: Example

1. The same program on supporting arithmetic operation can also be implemented using the **if-else-if-else** statements.
2. Generally, the **switch** statements can be replaced by **if-else-if-else** statements. However, as labels in the **switch** construct must be constant values (i.e. integer, character or expression), we will not be able to convert certain **if-else** statements into **switch** statements.



The switch Statement – Omitting break

1. In the **switch** statement, the **break** statement is placed at the end of each **case**.
2. What will happen if we omit the **break** statement as shown here? If the user enters the choice **'A'** and two numbers, what will be the outputs of the program if the user enters the choice **'A'**?



The switch Statement – Omitting break

1. The **break** statement is used to end each **case** constant block statement. If we do not put the **break** statement in the **switch** statement, execution will continue with the statements for the subsequent **case** labels until a **break** statement or the end of the **switch** statement is reached. This is called the *fall through* situation.
2. Therefore, if the **break** statement is omitted, the input and output of the program will become:

   Your choice (A, S or M) => A
   Enter two numbers: 9 5

   9 + 5 = 14

   9 - 5 = 4

   9 * 5 = 45
3. That is, the statements for the subtraction and multiplication operations are also executed.

The Conditional Operator

· The conditional operator is used in the following way:

```
expression_1 ? expression_2 : expression_3
```

The **value** of this expression depends on whether *expression_1* is true or false.

If expression_1 is true
=> the value of the expression is that of *expression_2*
Else
=> the value of the expression is that of *expression_3*

For example:

```
max = (x > y) ? x : y;   <==>   if (x > y)
                                    max = x;
                                else
                                    max = y;
```

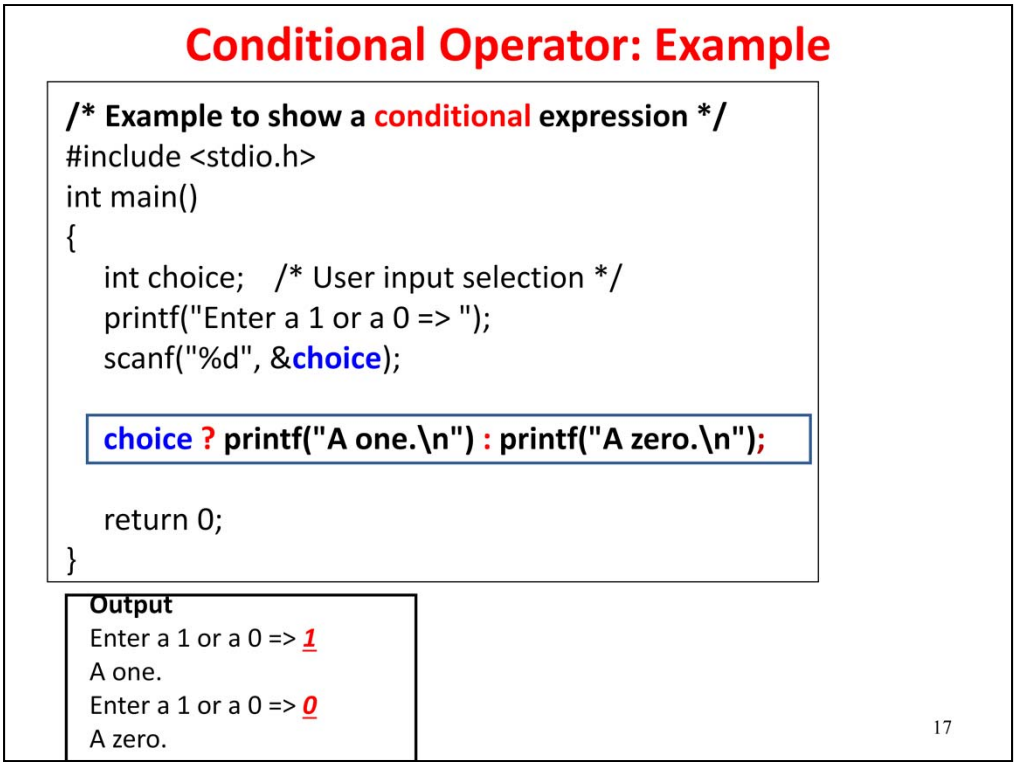
16

Conditional Operator

1. The conditional operator is a ternary operator, which takes three expressions, with the first two expressions separated by a '**?**' and the second and third expressions separated by a '**:** '.
2. The conditional operator is specified in the following way:

   expression_1 ? expression_2 : expression_3
3. The value of this expression depends on whether **expression_1** is true or false. If **expression_1** is true, the value of the expression becomes the value of **expression_2**, otherwise it is **expression_3**.
4. The conditional operator is commonly used in an assignment statement, which assigns one of the two values to a variable. For example, the maximum value of the two values **x** and **y** can be obtained using the following statement: **max = x > y ? x : y;** The assignment statement is equivalent to the following **if-else** statement:

```
if (x > y)
    max = x;
else
    max = y;
```



Conditional Operator: Example

1. This program gives an example on the use of the conditional operator.
2. The program first reads a user input on the variable **choice**. If **choice** is 1, then the string "**A one.**" will be printed. Otherwise, the string "**A zero.**" will be printed.
3. The program statement that uses conditional operator will be:

   choice ? printf("A one.\n") : printf("A zero.\n");
4. However, it can also be implemented quite easily using the **if-else** statement.

Control Flow

- Relational Operator and Logical Operator
- Branching: if, if…else, if….else if….else Statements; Nested if Statements; The switch Statement; Conditional Operators
- **Looping: for, while, do-while; Nested Loops; The break and continue Statements**
- Self-Learning Programming Example

18

Basic C Programming

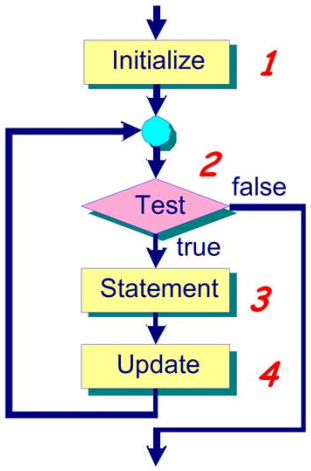
1. Here, we discuss the looping statements.

Looping

First, we need to determine the **Loop Control Variable.**

Then, to construct loops, we need:

1. **Initialize** – initialize the loop control variable.
2. **Test condition** – evaluate the test condition (involve loop control variable).
3. **Loop body** – the loop body is executed if test is true.
4. **Update** – typically, loop control variable is **modified** through the execution of the loop body. It can then go through the **test** condition.



There are three types of looping constructs: **for, while, do-while.**

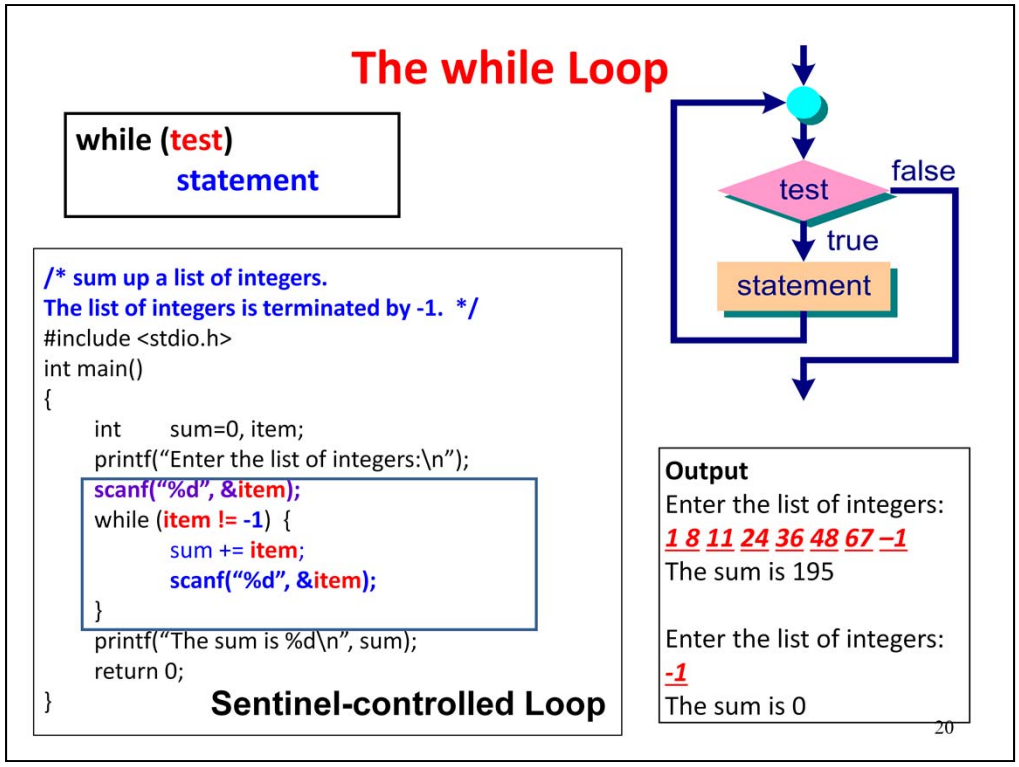
19

Looping

1. To construct loops, we need the following four basic steps:
   1) **Initialize** - This defines and initializes the loop control variable, which is used to control the number of repetitions of the loop.
   2) **Test** - This evaluates the **test** condition. If the **test** condition is true, then the loop body is executed, otherwise the loop is terminated. The **while** loop and **for** loop evaluate the **test** condition at the beginning of the loop, while the **do-while** loop evaluates the **test** condition at the end of the loop.
   3) **Loop body** - The **loop body** is executed if the **test** condition is evaluated to be true. It contains the actual actions to be carried out.
   4) **Update** - The **test** condition typically involves the loop control variable that should be modified each time through the execution of the loop body. The loop control variable will go through the **test** condition again to determine whether to repeat the loop body again.
2. We can use one of the three looping constructs, namely the **while** loop, the **for** loop and the **do-while** loop, for constructing loops.

Basically, there are two types of loops: *counter-controlled loops* and *sentinel-controlled loops*. In a counter-controlled loop, the loop body is repeated for a specified number of times, and the number of repetitions is known before the loop begins execution. In a

sentinel-controlled loop, the number of repetitions is not known before the loop begins execution. A *loop control variable* is typically used to determine the number of repetitions. The control variable is updated every time the loop is executed, and it is then tested in a test condition to determine whether to execute the loop body. An example of a *sentinel value* is a user input value such as –1, which should be different from regular data entered by the user.



The while Loop

1. The format of the **while** statement is

   while (test)

   statement;

2. **while** is a reserved keyword. **statement** can be a simple statement or a compound statement. The **while** statement is executed by evaluating the **test** condition. If the result is true, then **statement** is executed. Control is then transferred back to the beginning of the **while** statement, and the process repeats again. This looping continues until the **test** condition finally becomes false. When the **test** condition is false, the loop terminates and the program continues execute the next sequential statement.
3. The **while** loop is best to be used as a sentinel-controlled loop in situations where the number of times the loop to be repeated is not known in advance.
4. In the example program, it aims to sum up a list of integers which is terminated by -1. It does not know how many data items are to be read at the beginning of the program. It will keep on reading the data until the user input is –1 which is the sentinel value.
5. In the program, the **scanf()** statement reads in the first number and stores it in the variable **item**. Then, the execution of the **while** loop begins. If the initial **item** value is not –1, then the statements in the braces are executed. The **item** value is first added to another variable **sum**. Another **item** value is then read in from the user, and the control

is transferred back to the **test** expression **(item != -1)** for evaluation. This process repeats until the **item** value becomes −1.

Some more examples are given below. For example, in the following code:

```
int mark=0, total=0;              /* initialize */
while (mark != -1) {              /* test */
    printf("Enter mark: (-1 to end the input)"); /* loop body */
    scanf("%d", &mark);           /* update */
    total += mark;
}
```

the execution of the loop body is determined by the loop control variable **mark**. The user enters the **mark** and the sentinel value of –1 is used to signal the end of user input.

The **while** loop can also be used as a counter-controlled loop. In the following code:

```
int counter=0, mark, sum=0;       /* initialize */
while (counter < 10) {            /* test */
    printf("Enter the mark: ");   /* loop body */
    scanf("%d", &mark);
    sum += mark;
    counter++;                    /* update */
}
```

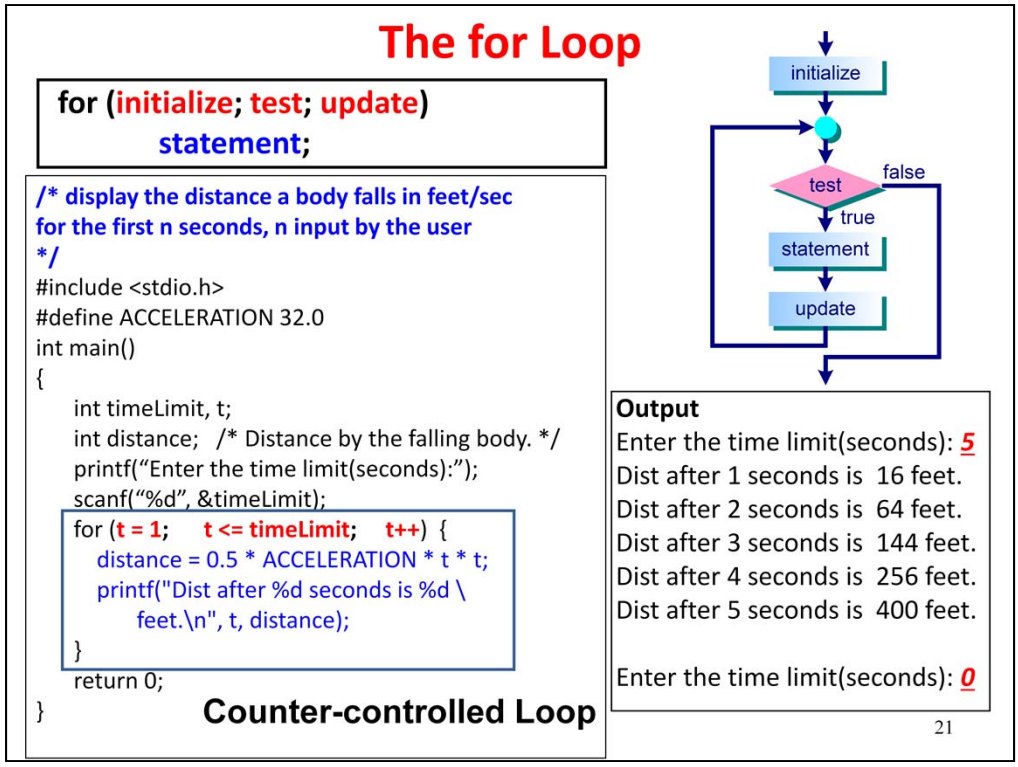
the loop control variable **counter** is defined and initialized. The loop control variable is incremented by one at the end of the loop body. The loop body will be executed when the **test** condition is evaluated true, i.e. **counter** is less than 10. The **test** condition will be evaluated as false when the counter value becomes 10, and the execution of the loop will be terminated.

Another common use of the **while** loop is to check the user input to see if the loop body is to be repeated. This is another example of sentinel-controlled loop. For example, in the following code:

```
char reply='Y';
while (reply != 'N') {
    printf("Repeat the loop body? (Y or N): \n");
    reply = getchar();
}
```

the loop will be terminated when the user enters the character '**N**'.

In the example program, it aims to sum up a list of integer numbers, and the list of numbers is terminated by -1.



The for Loop

1. The **for** statement allows us to repeat a sequence of statements for a specified number of times which is known in advance. The format of the **for** statement is given as follows:

   for (initialize; test; update)

   statement;

2. **for** is a reserved keyword. **statement** can be a simple statement or a compound statement. **initialize** is usually used to set the initial value of one or more loop control variables. Generally, **test** is a relational expression to control iterations. **update** is used to update some loop control variables before repeating the loop.
3. In the **for** loop, **initialize** is first evaluated. The **test** condition is then evaluated. If the **test** condition is true, then the **statement** and **update** expression are executed. Control is then transferred to the **test** condition, and the loop is repeated again if the **test** condition is true. If the **test** condition is false, then the loop is terminated. The control is then transferred out of the **for** loop to the next sequential statement.
4. The **for** loop is mainly used as a counter-controlled loop.
5. In the example program, it reads in the time limit and stores it in the variable **timeLimit**. It then uses the **for** loop as a counter-controlled loop. The loop control variable **t** is used to control the number of repetitions in the loop. The variable **t** is initialized to 1. In the loop body, the distance calculation formula is used to compute the distance. The loop control variable **t** is also incremented by 1 every time the loop

body finishes executing. The loop will stop when **t** equals to **timeLimit**.

The **for** statement can be represented by using a **while** statement as follows:

```
initialize;
while (test) {
    statement;
    update;
}
```

The difference is that in the **while** loop, we only specify the **test** condition and the **statement** in the loop, the **initialize** and **update** need to be added at the appropriate locations in order to make the **while** loop equivalent to the **for** loop.

The **for** loop is mainly used as a counter-controlled loop. For example, the following code

```
for (n=0; n<10; ++n) {
    sum += 5;
}
```

will add 5 to the variable **sum** every time the loop is executed. The number of time the loop is to be executed is known in advance. In this case, the loop will be executed 10 times. The loop will terminate when **n** becomes 10 and the **test** expression **(n<10)** becomes false. Consider the statements

```
for (n=100; n>10; n-=5) {
    total += 5;
}
```

the loop counts backward from 100 to 10. Each step will be decremented by 5. Therefore, the loop will be executed for a total of 18 times. For each time, the variable **n** will be decremented by 5, until it reaches 10.

Any or all of the 3 expressions may be omitted in the **for** loop. For example, the statements

```
for (n=5; n<=10 && n>=1;) {
    scanf("%d", &n);
}
```

are valid and the **update** expression is omitted. Notice that complex **test** conditions can be set using relational operators. The statements

```
for (; n<=10; ++n) {
    statement;
    ....
}
```

are also valid when the **initialize** expression is omitted.

In the case when the **test** expression is omitted, it becomes an infinite loop, i.e. all statements inside the loop will be executed again and again.

```
for (;;) {      /* an infinite loop */
    statement;
    ....
}
```

Notice that the semicolons must be included even if the expression is omitted.

In addition, we can also use more than one expression in **initialize** and **update**. A comma operator (**,**) is used to separate the expressions:

```
for (count=0, sum=0; count<5; count++) {
    sum+=count;
}
```

The **for** statement has two expressions in **initialize**. We can also have the **for** statement to perform the above task as follows:

```
for (count=0, sum=0; count<5; sum+=count, count++)
;
```

For the **for** loop, the loop body is null (**;**) which does nothing.

In the example program, it aims to display the distance a body falls in feet/sec for the first n seconds, where n is the user input.

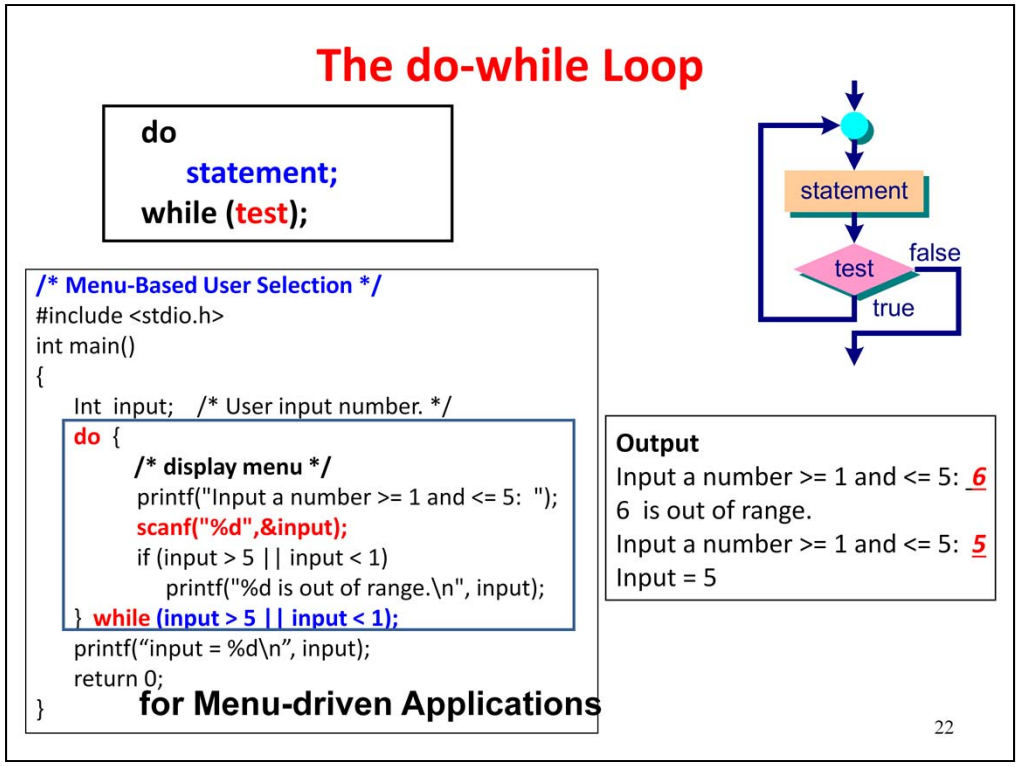
Another example of using the **for** loop is to calculate a series of data as follows:

1 - (x/1!) + (x^2/2!) – (x^3/3!) + (x^4/4!) - … + (x^20/20!)

Before we write the program, we observe that each component of the series consists of three parts: the part involving **x**, the part on the factorial, and the sign. Therefore, we create three variables, namely **nom**, **denom** and **sign** to store the data of each part of the component. In addition, the variable **result** is used to calculate the value of the series for each component of the **for** loop. The variable **result** is initialized to 1.0. The loop control variable **n** is used to control the loop execution. It is initialized to 1. It stops execution after 20 iterations when **n** equals to 21. The program is given as follows:

```
#include <stdio.h>
int main()
{
    double x, result = 1.0, nom = 1.0;
    int n, sign=1, denom = 1;
    printf("Please enter the value of x: ");
    scanf("%lf", &x);
```

```
    for (n=1; n<=20; n++) {
        denom *= n;
        sign = -sign;
        nom *= x;
        result += sign * nom/denom;
    }
    printf("The result is %lf\n", result);
    return 0;
}
```



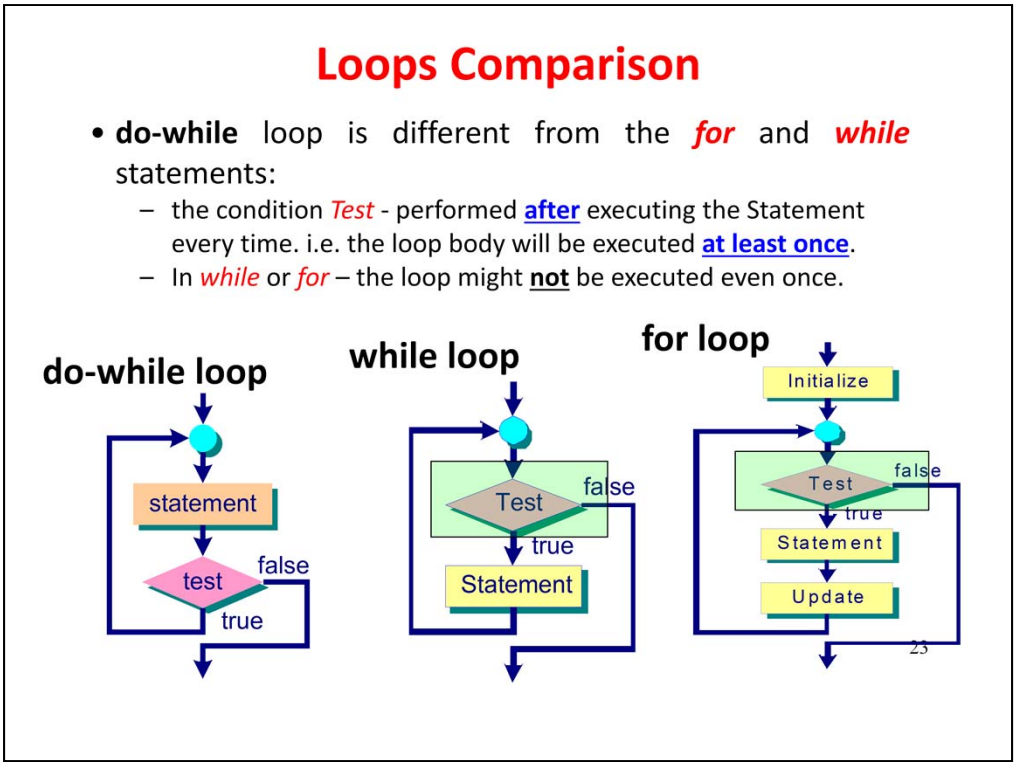
The do-while Loop

1. Both the **while** and the **for** loops test the condition prior to executing the loop body. C also provides the **do-while** statement which implements a control pattern different from the **while** and **for** loops. The body of a **do-while** loop is executed at least once.
2. The general format is

   do

   statement;

   while (test);
3. The **do-while** loop differs from the **while** and **for** statements in that the condition **test** is only performed after the **statement** has finished execution. This means the loop will be executed at least once. On the other hand, the body of the **while** or **for** loop might not be executed even once.
4. In the example program, it aims to implement a menu-driven application. The program reads in a number between 1 and 5. If the number entered is not within the range, an error message is printed on the display to prompt the user to input the number again. The program will read the user input at least once.



Loops Comparison

1. The **while** loop is mainly used for sentinel-controlled loops.
2. The **for** loop is mainly used for counter-controlled loops.
3. The **do-while** loop differs from the **while** loop in that the **while** loop evaluates the **test** condition at the beginning of the loop, whereas the **do-while** loop evaluates the **test** condition at the end of the loop. If the initial **test** condition is true, the two loops will have the same number of iterations. The number of iterations between the two loops will differ only when the initial **test** condition is false. In this case, the **while** loop will exit without executing any statements in the loop body. But the **do-while** loop will execute the loop body at least once before exiting from the loop.
4. Therefore, the **do-while** statement is useful for situations such as menu-driven applications which may require executing the loop body at least once.

The break Statement

- To alter flow of control inside loop (and inside the switch statement).
- Execution of break causes **immediate termination** of the **innermost enclosing loop** or switch statement.

```
/* summing up positive numbers
from a list of up to 8 numbers */
#include <stdio.h>
int main() {
    int i;
    float  data, sum=0;
    printf("Enter 8 numbers: ");
    /* read 8 numbers  */
    for (i=0; i<8; i++) {
        scanf("%f", &data);
        if (data < 0.0)
            break;
        sum += data;
    }
    printf("The sum is %f\n",sum);
    return 0;
}
```

Output

Enter 8 numbers: 3 7 -1 4 -5 8 3 1
The sum is 10.000000

24

The break statement

1. The **break** statement alters flow of control inside a **for**, **while** or **do-while** loop, as well as the **switch** statement.
2. The execution of **break** causes immediate termination of the innermost enclosing loop or the **switch** statement. The control is then transferred to the next sequential statement following the loop.
3. In the example program, it aims to sum up positive numbers from a list of numbers until a negative number is encountered. The program reads the input numbers of data type **float** for at most 8 numbers. A **for** loop is used to process the input number one by one. If the number is not less than zero, then the value is added to **sum.** Otherwise the **break** statement is used to terminate the loop. The control is then transferred to the next statement following the loop construct.

It is important to note that if the **break** statement is executed inside a nested loop, the **break** statement will only terminate the innermost loop. In the following statements:

```
for (i=0; i<10; i++) {
    for (j=0; j<20; j++) {
        if (i == 3 || j == 5)
            break;
        else
```

```
                printf("Print the values %d and %d.\n", i, j);
        }
    }
```

the **break** statement will only terminate the innermost loop of the **for** statement when **i** equals to 3 or **j** equals to 5. When this happens, the outer loop will carry on execution with **i** equals to 4, and the inner loop starts execution again.

The continue Statement

- The control is immediately passed to the test condition of the **nearest** enclosing loop.
- All subsequent statements after the continue statement are **not** executed for this particular iteration.

```
/* summing up positive numbers
from a list of 8 numbers */
 #include <stdio.h>
int main() {
    int i;
    float  data, sum=0;
    printf("Enter 8 numbers: ");
    /* read 8 numbers  */
    for (i=0; i<8; i++) {
        scanf("%f", &data);
        if (data < 0.0)
            continue;
        sum += data;
    }
    printf("The sum is %f\n",sum);
    return 0;
}
```

Output

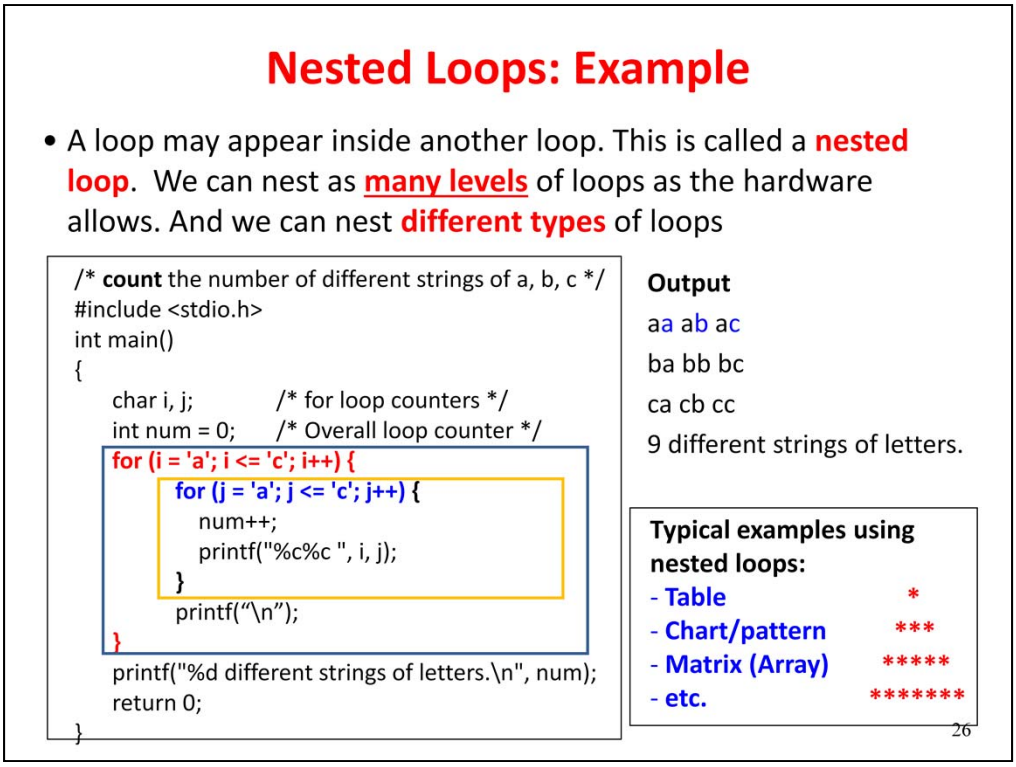
Enter 8 numbers: 3 7 -1 4 -5 8 3 1
The sum is 26.000000

25

The continue Statement

1. The **continue** statement causes termination of the current iteration of a loop and the control is immediately passed to the **test** condition of the nearest enclosing loop. All subsequent statements after the **continue** statement are not executed for this particular iteration.
2. The **continue** statement differs from the **break** statement in that the **continue** statement terminates the execution of the current iteration, and the loop still carries on with the next iteration if the **test** condition is fulfilled, while the **break** statement terminates the execution of the loop and passes the control to the next statement immediately after the loop.
3. In the example program, it aims to sum up only positive numbers from a list of 8 numbers. The program reads eight numbers of data type **float**. A **for** loop is used to process the input number one by one. If the number is not less than zero, then the value is added to **sum.** Otherwise the **continue** statement is used to terminate the current iteration of the loop, and the control is transferred to the next iteration of the loop. Notice that the **for** loop will process all the eight numbers when they are read in.

Note that the use of the **break** statement and **continue** statement are generally not recommended for good programming practice, as both statements interrupt normal sequential execution of the program.



Nested Loops

1. A loop may appear inside another loop. This is called a nested loop. We can nest as many levels of loops as the system allows. We can also nest different types of loops.
2. In the program, it generates the different strings of letters from the characters '**a**', '**b**' and '**c**'. The program contains a nested loop, in which one **for** loop is nested inside another **for** loop. The counter variables **i** and **j** are used to control the **for** loops. Another variable **num** is used as an overall counter to record the number of strings that have been generated by the different combinations of the characters.
3. The nested loop is executed as follows. In the outer **for** loop, when the counter variable **i='a'**, the inner **for** loop is executed, and generates the different strings **aa**, **ab** and **ac**. When **i='b'** in the outer **for** loop, the inner **for** loop is executed and generates **ba**, **bb** and **bc**. Similarly, when **i='c'** in the outer **for** loop, the inner **for** loop generates **ca**, **cb** and **cc**. The nested loop is then terminated. The total number of strings generated is also printed on the screen.
4. Nested loops are commonly used for applications that deal with 2-dimensional objects such as tables, charts, patterns and matrices.

Another example that uses nested loop is given in the following program:

```
#include <stdio.h>
int main()
```

```
{
    int space, asterisk, height, lines;
    printf("Please enter the height of the pattern: ");
    scanf("%d", &height);
    for (lines=1; lines<=height; lines++) {
        for (space=1; space<=(height – lines); space++)
            putchar(' ');
        for (asterisk=1; asterisk<=(2*lines – 1 ); asterisk++)
            putchar('*');
        putchar('\n');
    }
    return 0;
}
```

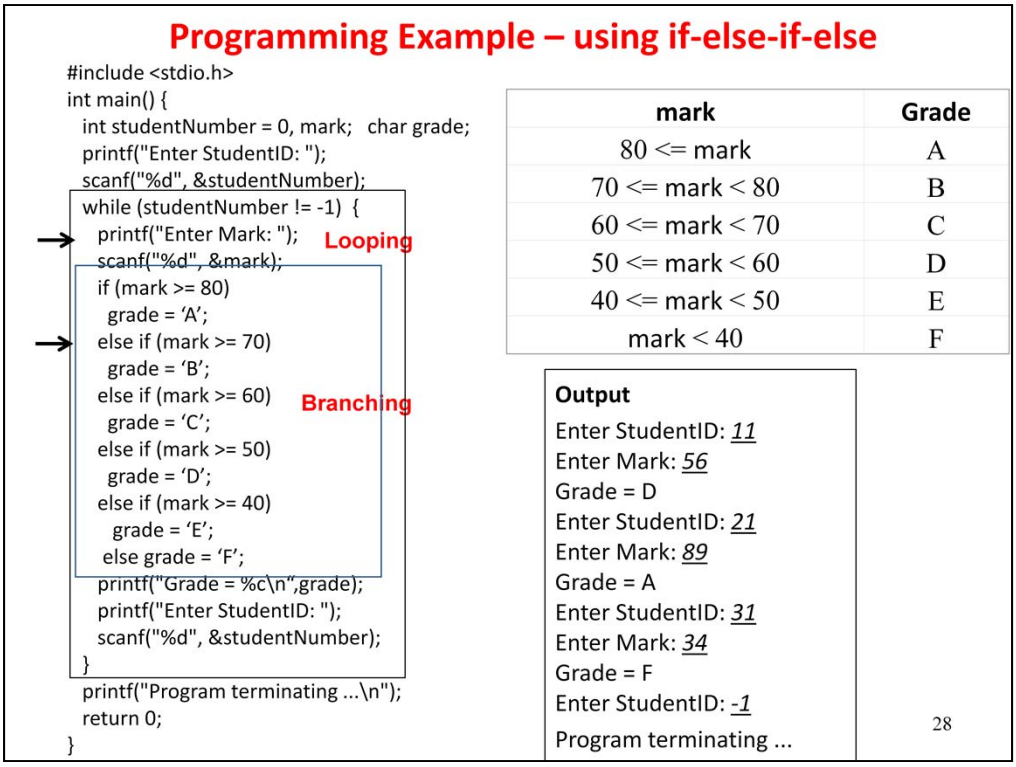
A sample input and output of the program is given below:

```
Please enter the height of the pattern: 5
    *
   ***
  *****
 *******
*********
```

The program prints a triangular pattern according to the height entered by the user. For example, when the user enters 5, the pattern with 5 lines is shown in the program output. In this program, a nested **for** loop is used. The outer **for** loop is used to control the line number (i.e. the vertical direction). The loop control variable **lines** is used for this purpose. For each line to be printed, another two inner **for** loops are created. The first inner **for** loop is used to control how many space characters (i.e. **' '**) are to be printed, and the second inner **for** loop is used to control how many asterisk characters (i.e. **'*'**) are to be printed. The first inner **for** loop will use the relationship between the height of the pattern and the line number to determine the number of spaces to be printed. In this case, the number of spaces to be printed is **(height - lines)** for each line. The second inner **for** loop uses the line number, i.e. **(2*lines - 1)** to determine how many asterisk characters is to be printed. The pattern is then printed according to the user input on **height**.

Self-Learning Programming Example

27



Programming Example: if-else-if-else

1. In the program, it uses the **if-else-if-else** statement for the implementation of mark-to-grade.
2. A **while** loop is used to read in student data. The loop will be terminated when the user enter -1 to the student number.
3. Inside the loop, it will read in student mark, and the **if-else-if-else** statement is used to determine the grade following the mark-to-grade conversion table and assign it to the variable **grade**. And the corresponding grade will be printed on the screen.

Programming Example: if-else to switch

mark	Grade
80 <= mark	A
70 <= mark < 80	B
60 <= mark < 70	C
50 <= mark < 60	D
40 <= mark < 50	E
mark < 40	F

```
int mark; char grade;
....
if (mark >= 80)
   grade = 'A';
else if (mark >= 70)
   grade = 'B';
else if (mark >= 60)
   grade = 'C';
else if (mark >= 50)
   grade = 'D';
else if (mark >= 40)
   grade = 'E';
else
   grade = 'F';
```

Q: How to use the switch statement?

```
int mark; char grade;
switch ( ??? ) {
      case 10:  ??
      case 9:  ??
      case 8: ??
      case 7: ??
      case 6: ??
      case 5: ??
      case 4: ??
      default:  ??
}
```

29

Programming Problem:

1. The purpose of this program is to convert the **if-else-if-else** statement into a **switch** statement in the mark-to-grade conversion problem.
2. The **if-else-if-else** statement is used to control the grade to be assigned to the variable **grade** according to the input value on **mark**.
3. However, how could we use the switch statement to do the same implementation.

Programming Example: using switch

mark	Grade
80 <= mark	A
70 <= mark < 80	B
60 <= mark < 70	C
50 <= mark < 60	D
40 <= mark < 50	E
mark < 40	F

Using integer division:
85/10 -> 8 64/10 -> 6
87/10 -> 8 68/10 -> 6
74/10 -> 7 34/10 -> 3
...
Div by 10 forms 11 categories from marks:
0, 1, 2, 3, ... 10

```
int mark; char grade;
....
switch ( mark/ 10 )
{
    case 10: case 9: case 8:
            grade = 'A'; break;
    case 7:
            grade = 'B'; break;
    case 6:
            grade = 'C'; break;
    case 5:
            grade = 'D'; break;
    case 4:
            grade = 'E'; break;
    default:  grade = 'F';
  }
```

30

Programming Solution: using switch

1. The key idea to solve this problem is to use integer division. When an integer is divided by 10, the result will fall into one of the category from 0 to 10. For example, 5/10 will get 0; 11/10 will get 1; 13/10 will get 1; 94/10 will get 9; and 100/10 will get 10.
2. Therefore, we can make use of the integer division results of mark/10 to form the labels of the switch statements.
3. For each case label, we can then determine the value for grade.

Programming Example – using switch

```
#include <stdio.h>
int main() {
  int studentNumber = 0, mark;   char grade;
  printf("Enter StudentID: ");
  scanf("%d", &studentNumber);
  while (studentNumber != -1)  {
    printf("Enter Mark: ");
    scanf("%d", &mark);
    switch ( mark/ 10 )  {
          case 10: case 9: case 8:
            grade = 'A'; break;
          case 7:
            grade = 'B'; break;
          case 6:
            grade = 'C'; break;
          case 5:
            grade = 'D'; break;
          case 4:
            grade = 'E'; break;
          default:  grade = 'F';
    }
    printf("Grade = %c\n",grade);
    printf("Enter StudentID: ");
    scanf("%d", &studentNumber);
  }
  printf("Program terminating ...\n");
  return 0;
}
```

Output

```
Enter StudentID: 11
Enter Mark: 56
Grade = D
Enter StudentID: 21
Enter Mark: 89
Grade = A
Enter StudentID: 31
Enter Mark: 34
Grade = F
Enter StudentID: -1
Program terminating ...
```

31

Programming Solution: using switch

1. Here, the solution code using the switch statement is given.
2. As shown in the solution code, we perform integer division based on mark, obtain the division result, and assign the grade according to the division result.

Thank you !!!

32