# 2
# Control low

1

**Control Flow**

1. In this lecture, we discuss the Control Flow in the C programming language.

## Why Learning Control Flow?

- The execution of C programming statements is normally in sequence from start to end.
- In the last lecture, we have discussed the simple data types, arithmetic calculations, simple assignment statements and simple input/output. With these statements, we can design simple C programs.
- However, the majority of challenging problems requires programs with the ability to make decisions as to what code to execute and the ability to execute certain portions of the code repeatedly.
- C provides a number of statements that allow **branching** (or selection) and **looping** (or repetition).
- In this lecture, we discuss the branching and looping constructs in C.

2

**Why Learning Control Flow?**

1. The execution of C programming statements is normally in sequence from start to end.
2. In the last lecture, we have discussed the simple data types, arithmetic calculations, simple assignment statements and simple input/output. With these statements, we can design simple C programs.
3. However, the majority of challenging problems requires programs with the ability to make decisions as to what code to execute and the ability to execute certain portions of the code repeatedly.
4. C provides a number of statements that allow branching (or selection) and looping (or repetition).
5. In this lecture, we discuss the branching and looping constructs in C.

## 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

3

**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 Operators

- Used to define a condition for comparing **two values**.
- Return **boolean** result: **true** or **false**.
- **Relational Operators:**

| 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 |

4

**Relational Operators**

1. In a branching operation, the decision on which statements to be executed is based on a comparison between two values. To support this, C provides relational operators.
2. Relational expressions involving relational operators are an essential part of control structures for branching and looping.
3. Relational operators in C include equal to (==), not equal to (!=), less than (<), less than or equal to (<=), greater than (>), and greater than or equal to (>=). These operators are binary. 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 will be returned, and if the result is false, then the integer value of 0 will be returned.

## Logical Operators

- **Works on one or more relational expressions** - to yield a logical return value: **true** or **false**.
- Allows testing the results on comparing expressions.
- **Logical operators:**

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

| A && B | A is true | A is false |
|---|---|---|
| B is true | true | false |
| B is false | false | false |

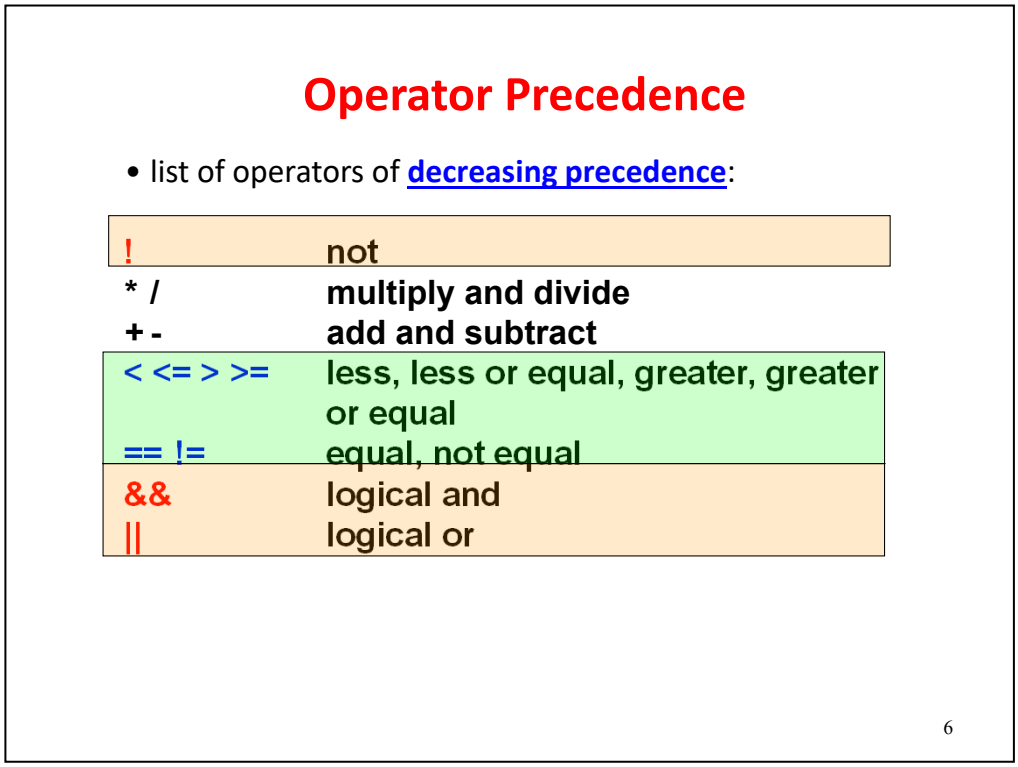
| | A is true | A is false |
|---|---|---|
| !A | false | true |

| A \|\| B | A is true | A is false |
|---|---|---|
| B is true | true | true |
| B is false | true | false |

5

**Logical Operators**

1. Logical operators work on one or more relational expressions to yield either the logical value true or false. 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 of the results of comparison expressions.
2. 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. Logical **or** operator (**||**) returns false when both operands are false, otherwise it returns true.




**Operator Precedence**

1. Operator precedence determines the order of operator execution. The list of operators of **decreasing precedence** is shown in the slide:

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

2. Note that 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 as 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 |

7

**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 as *true*; otherwise it is *false*.
3. Therefore, in the example, 3 is true, and 0 is false. (1 && 0) is false and (1 || 0) is true.

## 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

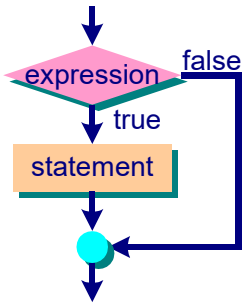
8

**Control Flow**

1. Here, we discuss the branching statements.

## The if Statement

```
if (expression)
        statement;
        /* simple or compound statement
        enclosed with braces { } */
```




```
/* Program: check user number greater than 5 */
#include <stdio.h>
int main()
{
    int num;
    printf("Give me a number from 1 to 10: ");
    scanf("%d", &num);
    if (num > 5)
        printf("Your number is larger than 5.\n");
    printf("%d was the number you entered.\n",num);
    return 0;
}
```

**Output**

```
Give me a number from 1 to 10: 3
3 was the number you entered.

Give me a number from 1 to 10: 7
Your number is larger than 5.
7 was the number you entered.
```

9

**The if Statement**

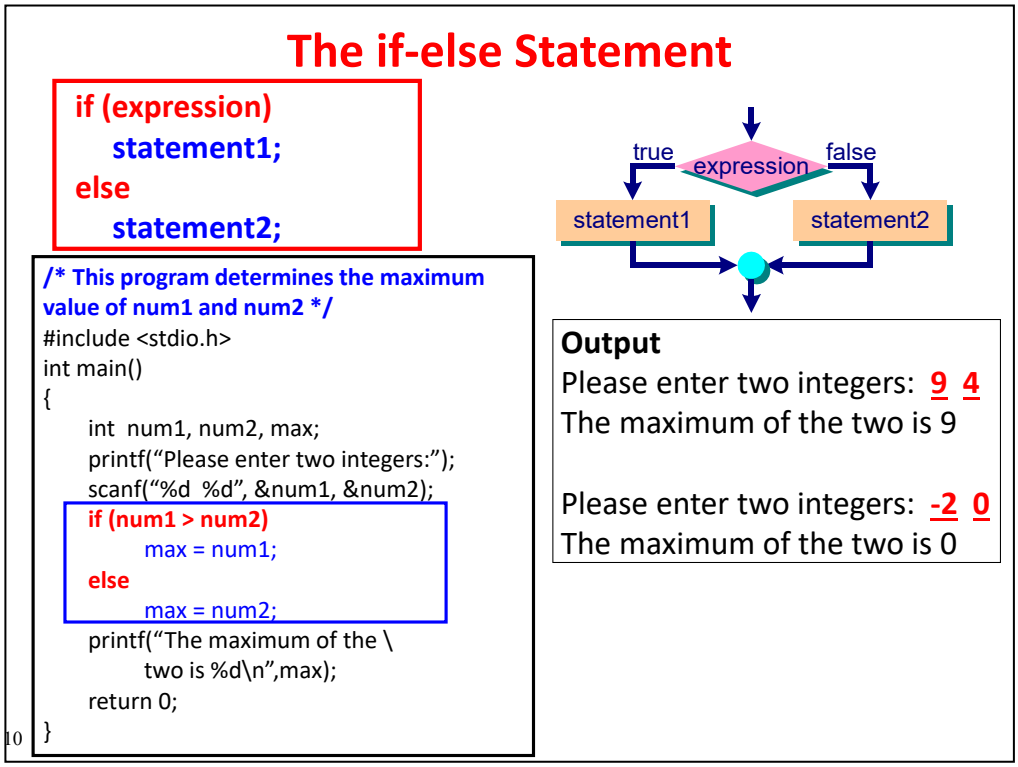
1. The simplest form of the **if** statement is

   **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.



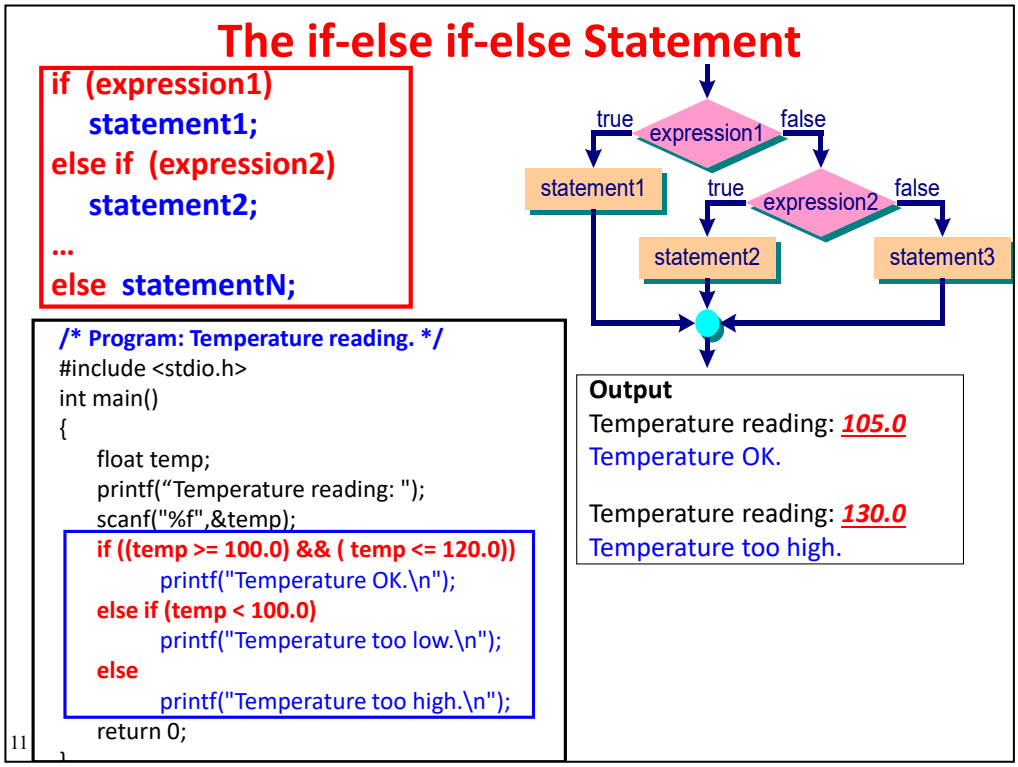

**The if-else Statement**

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

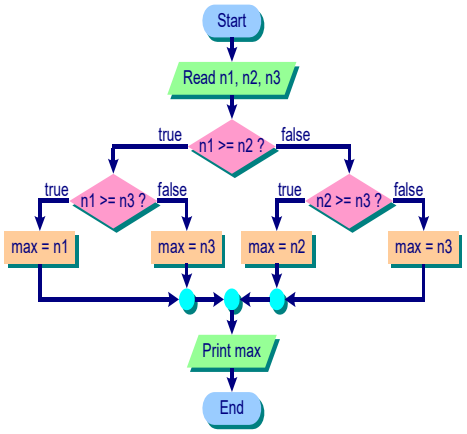
```
if (expression1)
    statement1;
else if (expression2)
    statement2;
....
else
    statementN;
```

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.

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

```
/* Determines max value of 3 num */
#include <stdio.h>
int main(){
    int   n1, n2, n3, max;
    printf("Please enter three integers:");
    scanf("%d  %d  %d", &n1, &n2, &n3);
    if (n1 >= n2) {
        if (n1 >= n3)
            max = n1;
        else max = n3;
    }
    else if (n2 >= n3)   max = n2;
    else max = n3;
    printf("The maximum is %d\n",max);
    return 0;
}
```

**Nested-if**




**Output**
Please enter three integers: ***1 2 3***
The maximum of the three is 3

**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 shown.
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.
4. We can also use braces to enclose statements:

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

5. 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**.

6. Note that in this example braces are used to enclose the statements for the inner **if** condition:

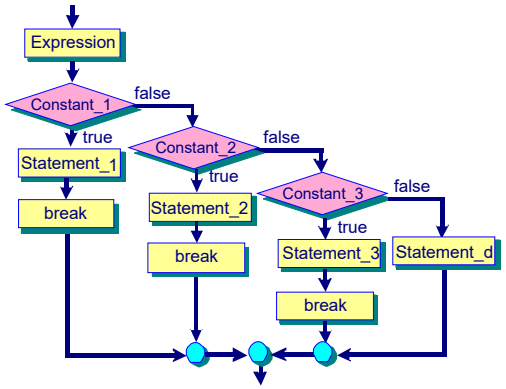
```
if (n1 >= n2) {
    if (n1 >= n3)
        max = n1;
    else max = n3;
}
```

## The switch Statement

The switch is used 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;
}
```




13

**The switch Statement**

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 braces **{ }**.
5. Note that 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.

## The switch Statement: Syntax

The **switch** is used 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;
}
```

- **switch, case, break** and **default** - reserved keywords.
- The result of **expression** in ( ) must be of **integral type**.
- **constant_1, constant_2, ...** are called **labels**.
  - must be an **integer constant**, a **character constant** or an integer constant expression, e.g. 3, 'A', 4+'b', 5+7, ..., etc.
  - must deliver a **unique integer value**. Duplicates are not allowed.
  - may also have **multiple labels** for a statement, for example, to allow both lower and upper case selection.

14

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

## 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;
15  }
```

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

**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 arithmetic operation will be printed.
5. For example, if user selects 'S', then the subtraction operation of the two numbers 9 and 5 will be computed.
6. 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;
  }
```

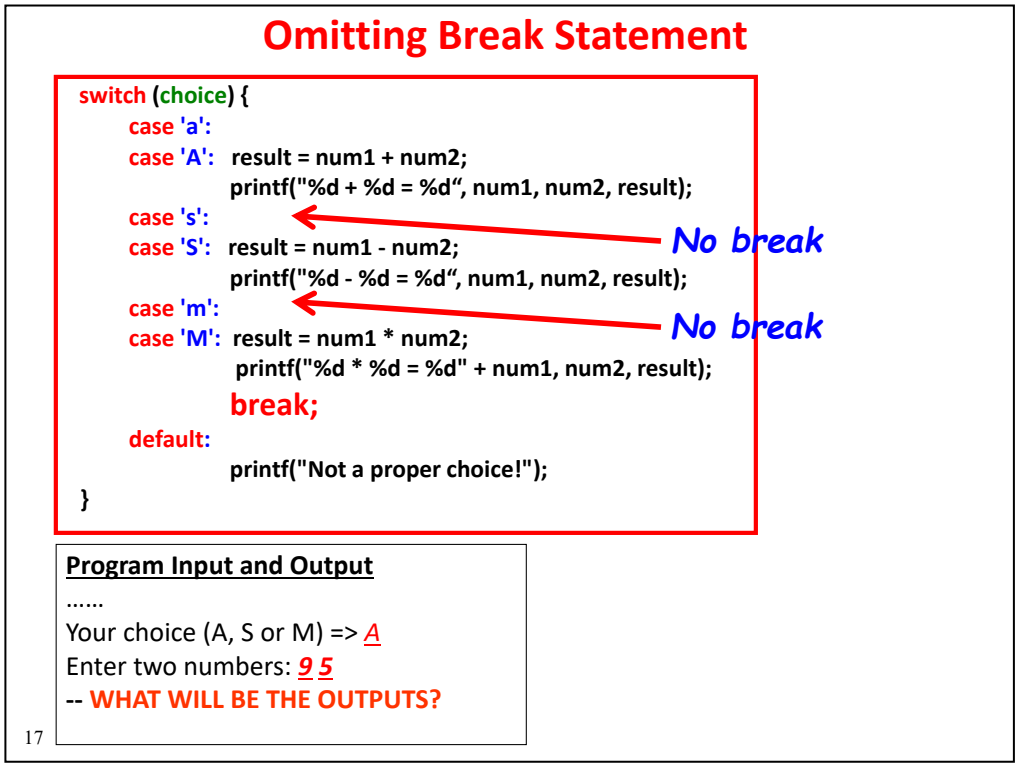
16

**Note**:
1. The **switch** statements can be replaced by **if-else-if-else** statements.
2. Also, the labels in the **switch** construct must be **constant integral values** which make it not so flexible.

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

### 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. For example, if user selects 'S', then the subtraction operation of the two numbers 9 and 5 will be computed according to the if-else condition.
3. 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 integer/character expression), we will not be able to convert certain **if-else** statements into **switch** statements.

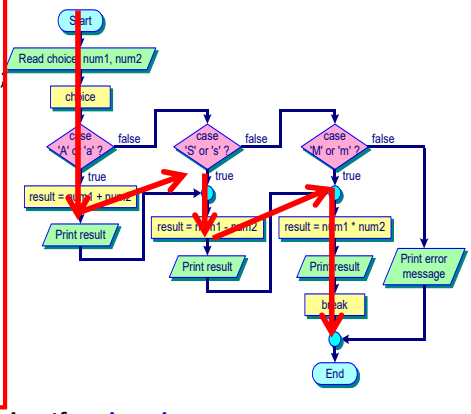



**The switch Statement – Omitting The break Statement**

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 in this example?
3. The **break** statement was omitted for the case **'A'** and case **'S'**.
4. If the user enters the choice **'A'** with the two numbers 9 and 5, what will be the outputs of the program?

## Omitting Break - Fall Through

```
switch (choice) {
    case 'a':
    case 'A':  result = num1 + num2;
       printf("%d + %d = %d", num1, num2, result);
    case 's':                          No break
    case 'S':  result = num1 - num2;
       printf("%d - %d = %d", num1, num2, result);
    case 'm':                          No break
    case 'M':  result = num1 * num2;
       printf("%d * %d = %d", num1, num2, result);
       break;
    default:
       printf("Not a proper choice!");
}
```




**Program Input and Output**

……
Your choice (A, S or M) => *A*
Enter two numbers: ***9 5***
`-- WHAT WILL BE THE OUTPUTS?`
9 + 5 = 14
9 – 5 = 4
9 * 5 = 45

18

- **Fall through** – if no **break** statement for the case block, execution will **continue** with the statements for the subsequent labels until a break statement or the end of switch statement is reached.

### The switch Statement – Omitting break

1. The **break** statement is used to end each **case** constant block.
2. If we do not put the **break** statement for the case block, 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.
3. 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.

## 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;
```

19

**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

```
/* Example to show a conditional expression */
#include <stdio.h>
int main()
{
   int choice;   /* User input selection */
   printf("Enter a 1 or a 0 => ");
   scanf("%d", &choice);
   choice ? printf("A one.\n") : printf("A zero.\n");
   return 0;
}
```

**Output**

```
Enter a 1 or a 0 => 1
A one.
Enter a 1 or a 0 => 0
A zero.
```

20

**Conditional Operator: Example**

1. This program gives an example on the use of the conditional operator.
2. The program statement that uses conditional operator is:

   **choice ? printf("A one.\n") : printf("A zero.\n");**

1. The program first reads a user input on the variable **choice**. If **choice** is 1 (i.e. true), then the string "**A one.**" will be printed. Otherwise if **choice** is 0 (i.e. false), the string "**A zero.**" will be printed.
4. However, this 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**

21

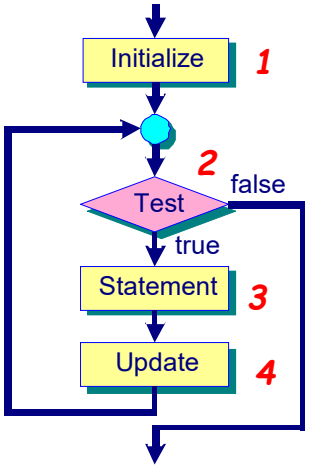
**Control Flow**

1. Here, we discuss the looping statements.

## Looping

There are mainly 4 basic steps to construct a loop:

1. **Initialize** – We need to define **Loop Control Variable** and initialize the loop control variable.
2. **Test** – This evaluates the test condition which involves the loop control variable. If the **test** evaluation is true, then the loop body is executed, otherwise the loop is terminated.
3. **Loop body (or Statement)** – The loop body is executed if test evaluation is true.
4. **Update** – Typically, loop control variable is **modified** through the execution of the loop body on Update. It can then go through the **test** condition again to evaluate whether to repeat the loop body again.

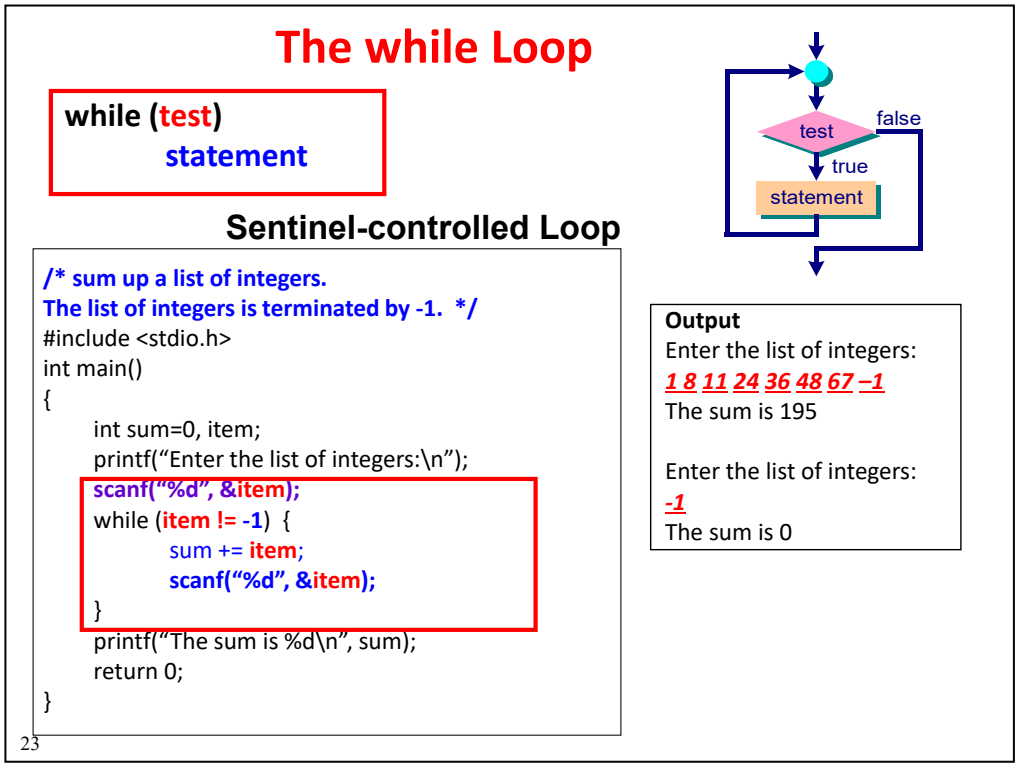



In C, there are three types of looping constructs: **for, while, do-while.**

22

**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.




**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.
3. 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.
4. 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.
5. 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.
6. 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

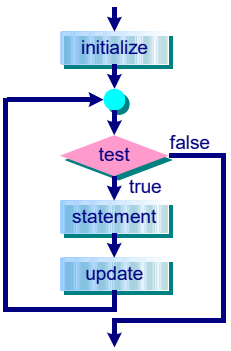
is transferred back to the **test** expression **(item != -1)** for evaluation. This process repeats until the **item** value becomes −1.

## The for Loop

```
for (initialize; test; update)
        statement;
```

### Counter-controlled Loop

```
/* display the distance a body falls in feet/sec
for the first n seconds, n input by the user
*/
#include <stdio.h>
#define ACCELERATION 32.0
int main()
{
    int timeLimit, t;
    int distance;   /* Distance by the falling body. */
    printf("Enter the time limit (sec):");
    scanf("%d", &timeLimit);
    for (t = 1;     t <= timeLimit;     t++)  {
      distance = 0.5 * ACCELERATION * t * t;
      printf("Dist after %d sec is %d \
          feet.\n", t, distance);
    }
    return 0;
}
```




**Output**

```
Enter the time limit (sec): 5
Dist after 1 sec is 16 feet.
Dist after 2 sec is 64 feet.
Dist after 3 sec is 144 feet.
Dist after 4 sec is 256 feet.
Dist after 5 sec is 400 feet.
```

24

**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)**

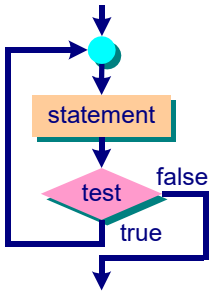
   **statement;**

2. **for** is a reserved keyword. **statement** can be a simple statement or a compound statement.
3. **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.
4. 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.
5. The **for** loop is mainly used as a counter-controlled loop.
6. 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 do-while Loop

```
do
    statement;
while (test);
```




### for Menu-driven Applications

```
/* Menu-Based User Selection */
#include <stdio.h>
int main()
{
    int  input;    /* User input number. */
    do {
        /* display menu */
        printf("Input a number >= 1 and <= 5:  ");
        scanf("%d",&input);
        if (input > 5 || input < 1)
            printf("%d is out of range.\n", input);
    } while (input > 5 || input < 1);
    printf("Input = %d\n", input);
    return 0;
}
```

**Output**

```
Input a number >= 1 and <= 5: 6
6  is out of range.
Input a number >= 1 and <= 5: 5
Input = 5
```

25

**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

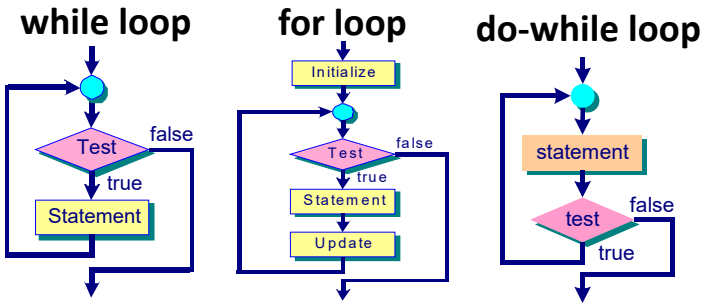
   **do**

   **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

- **do-while** loop is different from the ***for*** and ***while*** loops:
  - In the *while* or *for* loop – the condition *Test* is performed **before** executing the Statement, i.e. the loop might **not** be executed even once.
  - In the *do-while* loop - the condition *Test* is performed **after** executing the Statement every time, i.e. the loop body will be executed **at least once**.




Sentinel-controlled Loop

Counter-controlled Loop

Menu-driven Applications

26

**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

- The **break** statement alters the flow of control inside a **for**, **while** or **do-while** loop (as well as 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

27

**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.

## The continue Statement

- 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.

```
/* 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;
}
```

- **Note**: the **break** statement terminates the execution of the loop and passes the control to the next statement immediately after the loop.

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

28

### 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.

# Nested Loops

- A loop may appear inside another loop. This is called a **nested loop**. We can nest as **many levels** of loops as the hardware allows. And we can nest **different types** of loops.
- Nested loops are commonly used for applications that deal with 2-dimensional objects such as tables, charts, patterns and matrices.

```
/* count the number of different strings of a, b, c */
#include <stdio.h>
int main()
{
    char i, j;          /* for loop counters */
    int num = 0;        /* Overall loop counter */
    for (i = 'a'; i <= 'c'; i++) {
        for (j = 'a'; j <= 'c'; j++) {
            num++;
            printf("%c%c ", i, j);
        }
        printf("\n");
    }
    printf("%d different strings of letters.\n", num);
    return 0;
}
```

**Output**

```
aa ab ac
ba bb bc
ca cb cc
9 different strings of letters.
```

29

## 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.


**Thank You**

1. Thanks for watching the lecture video.