Fundamentals of Computer Programming

Chapter 2
Flow of Control Part I
(Selection Statement)

Chere L. (M.Tech)
Lecturer, SWEG, AASTU

Outline

- Introduction to flow control
- Branching flow controls (selection statements)
  - ✓ One-way selection
  - ✓ Two-way selection
  - ✓ Multiple selection
  - ✓ switch statement

Objectives

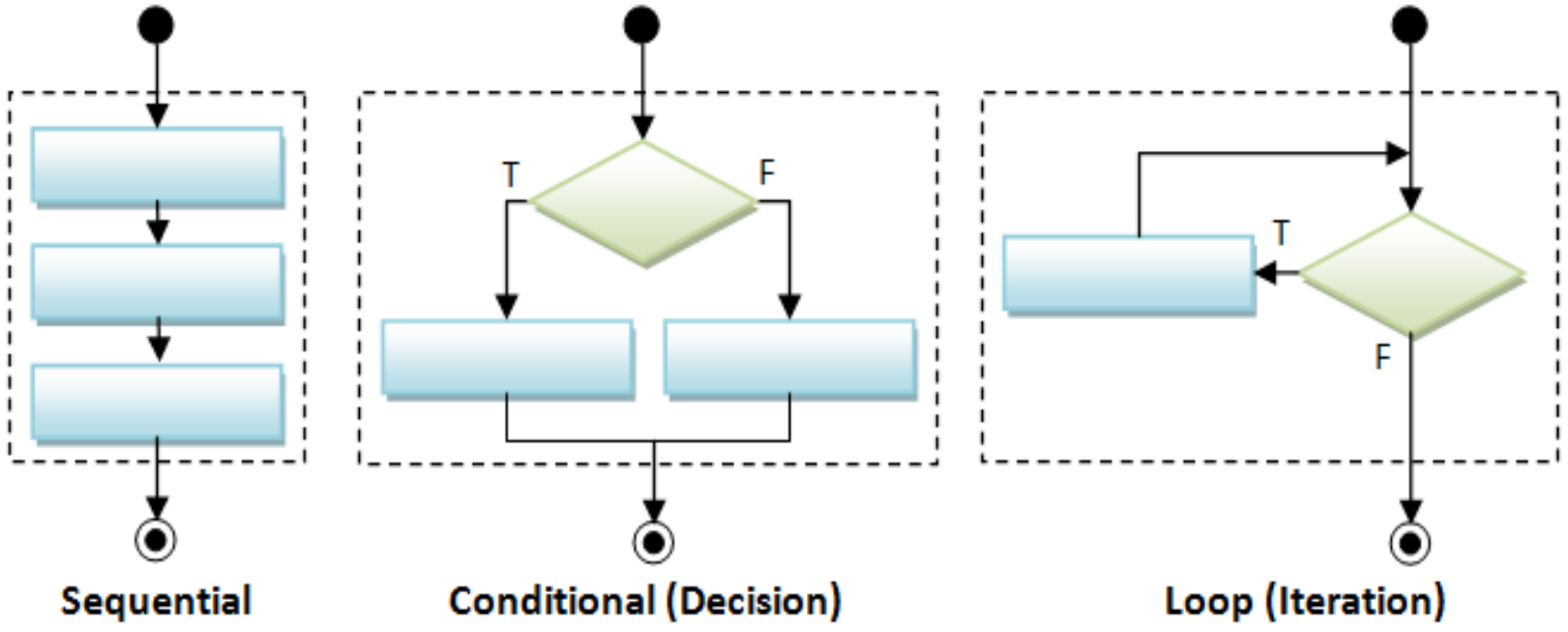
- Learn how to use selection flow control
- Learn how to form Boolean expressions and examine relational and logical operators
- Design and develop program using selection statements

1. Introduction to flow controls

- *Flow of control* is the order in which a program statements are executed (performs actions).
- The term *flow control* reflects the fact that the currently executing statement has the control of the CPU and handed over (flow) to another statement when it's execution completed.
- Typically, the flow control in a program is ***sequential***, which is the most common and straight-forward.
- However, usually a program execution is not limited to a ***sequential***
- Programmers can control the order of instruction execution
- Accordingly, most programming language including C++ provides control structures that serve to specify what has to be done by our program, when and under which circumstances.

1. Introduction to flow controls (cont'd)

- Generally there are three basic program flow control
  1. **Sequential** – execution of instruction sequentially one after an other
  2. **Selection/branching** – allow alternative actions based up on conditions that are evaluated at run time.
  3. **Iteration/loop –** allows to execute a statement or group of statements multiple times



2. Recalling relational and logical operators

- Logical/Boolean expressions are a fundamental part of **control statements** and are formed with combinations of two kinds of operators:
- The expression of both types of operators are evaluated to **true/false**

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to

(a) Relational operators

Operator	Meaning	Alternate[1]
&&	Logical AND	and
||	Logical OR	or
!	Logical NOT	not

(b) Logical operators

Boolean expression example

Expression	Value	Explanation
`(14 >= 5) && ('A' < 'B')`	`true`	Because `(14 >= 5)` is `true`, `('A' < 'B')` is `true`, and `true && true` is `true`, the expression evaluates to `true`.
`(24 >= 35) && ('A' < 'B')`	`false`	Because `(24 >= 35)` is `false`, `('A' < 'B')` is `true`, and `false && true` is `false`, the expression evaluates to `false`.

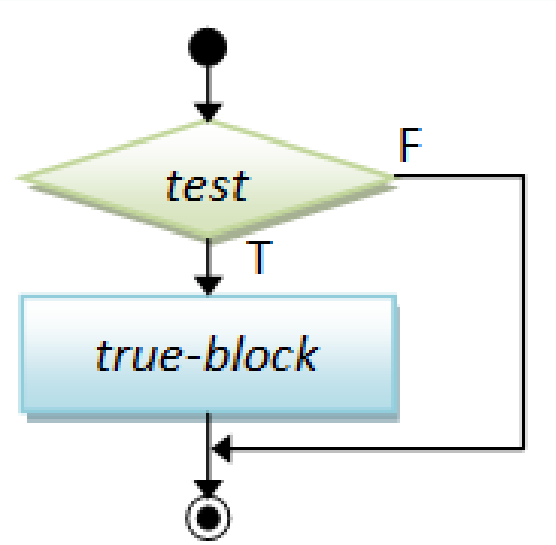
2. Selection/Branching statements

- *The selection statements include the following*
  - ✓ if statement ----> One-way selection
  - ✓ if...else statement ---> Two-way selection
    - ➢ *Conditional operator*
    - ➢ nested if --- else statement
  - ✓ else if...else statement ---> Multiple selection
  - ✓ switch statement

2. Selection statements (cont'd)

- **One-way selection --- > if statement**

```
if (expression) {
      statement (s);
  }
next statement(s);
```

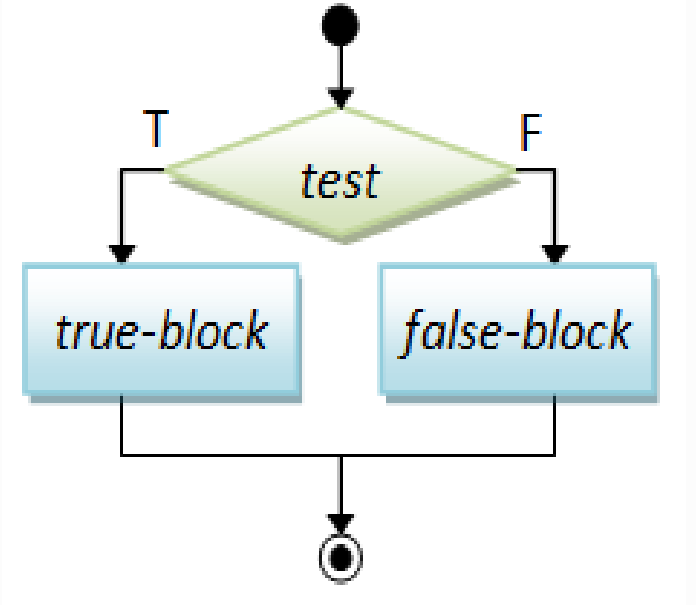


- **Expression**
  - ✓ A condition that must evaluated to true/false (i.e. Boolean expression)
  - ✓ One or more relational expression can be combined with logical operators
- if condition is TRUE the statement(s) or the block that follow the selection is executed and then next statement(s) get executed.
- Otherwise nothing will be executed and execution continues with the next statement in the program.

2. Selection statements (cont'd)

- **Two-way selection -- > if ….. else statement**

```
if (expression){
      statement1 / block1
 }
else{
            statement2 / block2;
 }
  next statement(s);
```



- **Expression** -- similar to that of one-selection statement
- if condition is **TRUE** the *statement1 or block1* is that follow the if selection is executed and then next statement(s) get executed.
- Otherwise the *statement2 or block2* is executed and the execution continues with the next statement in the program.

2. Selection statements (cont'd)

- **Another Syntax ----- > without the block { }**

```
if (condition)
     <statement_true>;
else
    <statement_false>;
```

```
if (condition)
  <statement_true>;
```

- Can be used when there is only one statement
- Not suggested (it causes dangling Else Problem)

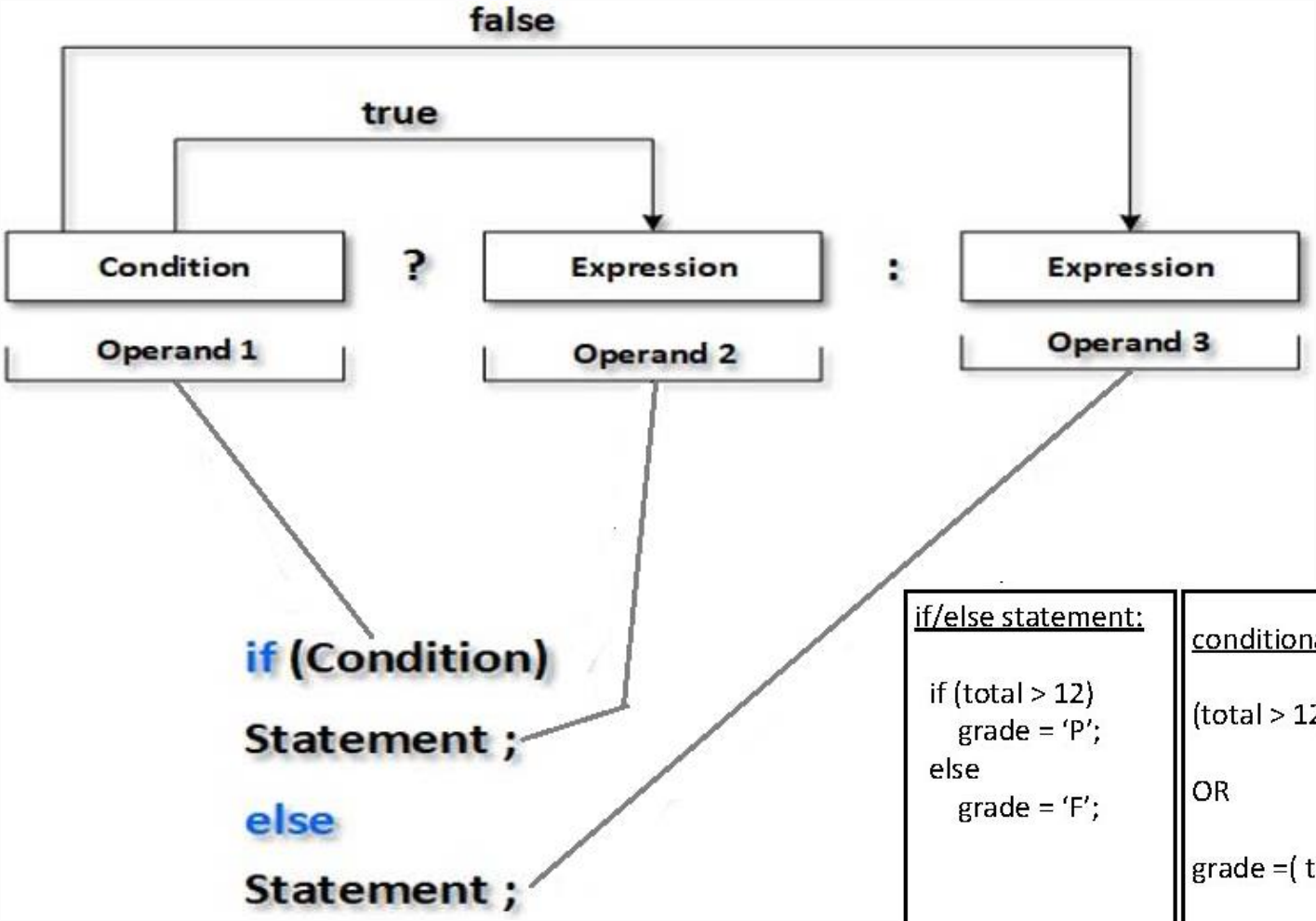
Example

```
if (work_hrs > 40) {
   OT = work_hrs – 40 * 120;
    cout << "Over Time " <<OT<<endl;
 }
```

```
if (mark >= 50) {
    cout << "Congratulation!" << endl;
    cout << "Keep it up!" << endl;
 }
else {
    cout << "Failed, try harder!" << endl;
}
```

2. Selection statements (cont'd)

- **Conditional operator instead of if …. else statement**



if/else statement:

```
if (total > 12)
   grade = 'P';
else
   grade = 'F';
```

conditional statement:

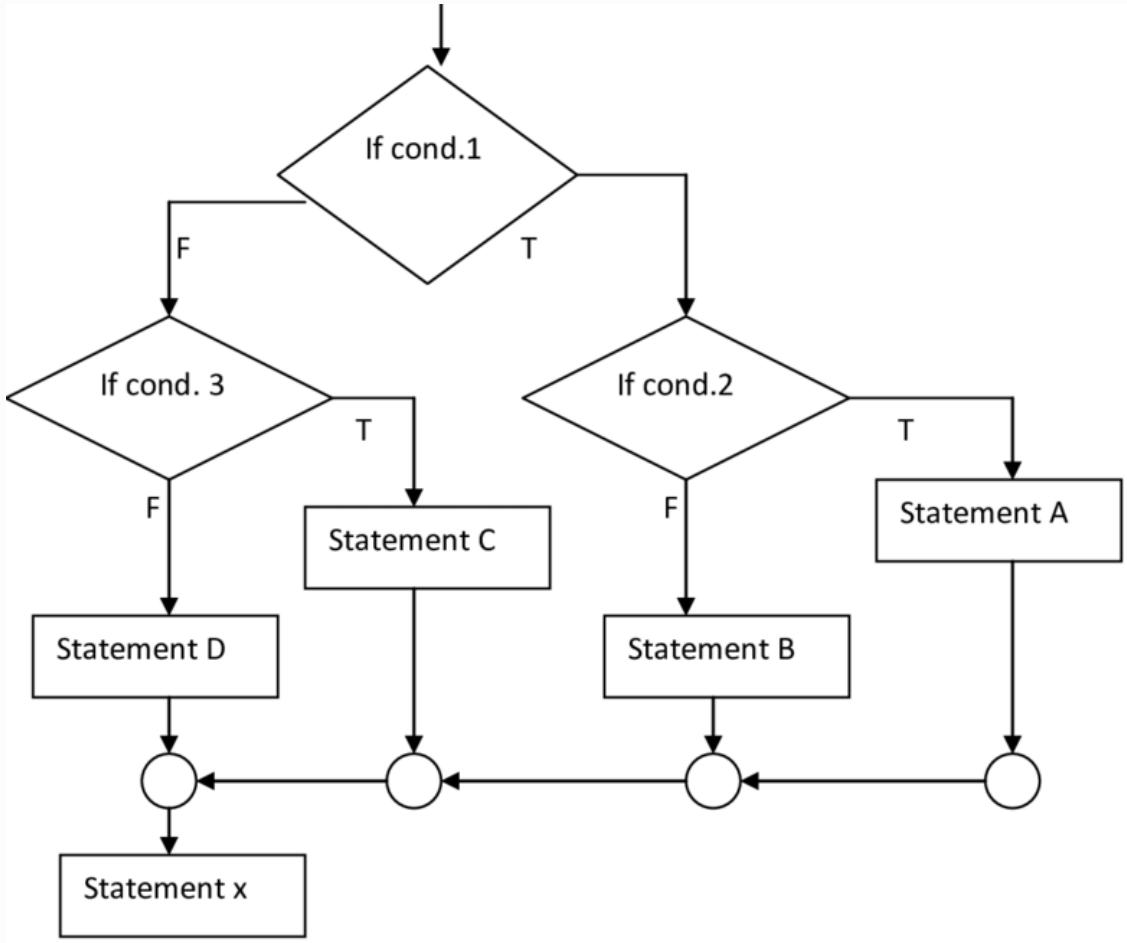
```
(total > 12) ? grade = 'P': grade = 'F';

OR

grade =( total > 12) ? 'P': 'F';
```

2. Selection statements (cont'd)

- **Nested if/else statement**
  - ✓ Refers to using within another selection statement
  - ✓ Match each else with the last unmatched if

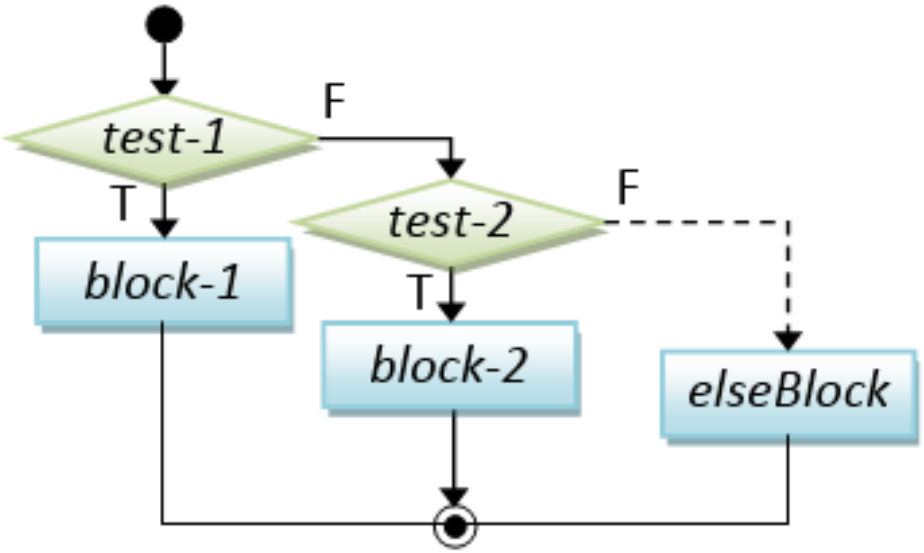



```
char ch;
cout<<"enter any symbol: ";
cin>>ch;
if (ch >= '0' && ch <= '9')
        cout<<"digit"<<endl;
else {
        if (ch >= 'A' && ch <= 'Z')
            cout<<"upperLetter"<<endl;
        else {
                if (ch >= 'a' && ch <= 'z')
                    cout<<"lowerLetter"<<ednl;
                else
                    cout<<"special"<<endl;
            }
    }
```

2. Selection statements (cont'd)

- **Multiple selection -- > if ….. else if statement**
  - ✓ Allows for conditional execution based up on more than two alternatives

```
if (expression1){
      statement1 / block1
 }
else if (expression2){
            statement2 / block2;
 }
. . . . .
else {
        statement-N / block-N;     }
 next statement(s);
```



- **Expression** -- similar to that of one-selection statement
- if ***expression1*** returns **TRUE** the *statement1 or block1* is that follow the if selection is executed and then next statement(s) get executed.
- Otherwise ***expression2*** of each ***else if*** part is evaluated and the *statement2 or block2* of the selection that returns **TRUE** is executed and the execution continues with the next statement in the program.

2. Selection statements (cont'd)

- **if ….. else if Vs. nested if … else statement**
  - ✓ What is the difference between nested *if – else* and *else if* selection structure?
  - ✓ Evaluate the below two examples

(a)

```
float mark;;
cout<<"enter mark: ";
cin>>mark;
if (mark >= 0 && ch <= 100)
{
    if (mark >= 80)
        cout<<"Excellent!"<<endl;
    else if (mark >= 60)
        cout<<"Very Good"<<ednl;
    else if (mark >= 40)
        cout<<"Satisfatory"<<endl;
    else
        cout<<"poor"<<endl;
}
else
    cout<<"invalid mark"<<endl;
```

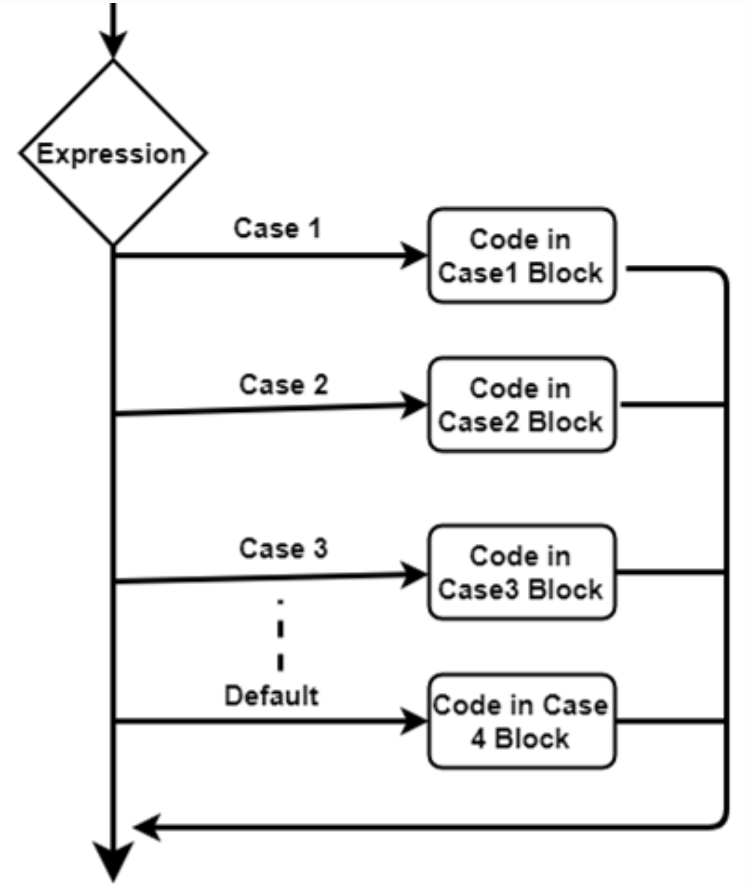
(b)

```
float mark;;
cout<<"enter mark: ";
cin>>mark;
if (mark >= 0 && ch <= 100)
{
    if (mark >= 80)
    {
        cout<<"Excellent!"<<endl;
    }
    else {
        if (mark >= 60)
            cout<<"Very Good"<<ednl;
        else{
            if (mark >= 40)
                cout<<"Satisfatory"<<endl;
            else
                cout<<"poor"<<endl;
        }
}
else
    cout<<"invalid mark"<<endl;
```

2. Selection statements (cont'd)

- **switch statement**
  - ✓ It is similar to *if … else if* combination, it enables you to test several cases generated by a given expression
  - ✓ The value of a variable/expression determine where the program will branch

```
switch (expression)
{
  case constant-1:
        group of statements 1;
        break;
  case constant-2:
       group of statements 2;
       break;
case constant-3:
       group of statements 3;
       break;
          -
          -
  default:
     default group of statements
}
```



2. Selection statements (cont'd)

- **How it works**
  - ✓ The **switch** expression is evaluated once
  - ✓ The value of the expression is compared with values of each **case**
  - ✓ If there is a match, the associated block of statements is executed
  - ✓ The statements following the matched case will be executed until a **break** statement is reached
  - ✓ When a **break** statement is reached, **the switch terminates,** and the flow of control jumps to the next line following the switch statement.
  - ✓ A switch statement can have an optional **default case**, which usually appears at the end of the switch.
  - ✓ The default case can be used for performing a task when none of cases is true.
- **Note**
  - ✓ The **expression** must evaluated to **literal value (integral/character/enum)**
  - ✓ Each **case** is followed by the value to be compared to and a **colon**.
  - ✓ The expression of each **case statement** in the block must be **unique.**
  - ✓ If **no break** appears, the flow of control will **fall through to subsequent cases until a break is reached.**

2. Selection statements (cont'd)

Additional notes

- ✓ The **case statement** expression cannot be variable and also range
- ✓ Switch can only be used to compare an **expression against constants**
- ✓ It is not necessary to include braces {} surrounding the statements for each of the cases
- ✓ Even if it is usually necessary to include a break statement at the end of each case, there are situations in which it makes sense to have a **case without a break**

- **Example 1:**

```
#include <iostream>
using namespace std;

int main(){

    char operators;
    cout<<"One of the operators (+, -, *, / ";
    cin>>operators;

    int result, operand1, operand2;
    switch (operators) {
    case '+':   result = operand1 + operand2;
                break;
    case '-':   result = operand1 - operand2;
                break;
    case '*':   result = operand1 * operand2;
                break;
    case '/':   result = operand1 / operand2;
                break;
    default:
        cout<<"unknown operator: "<<operators<<'\n';
        break;
}
    return 0;
}
```

2. Selection statements (cont'd)

Example 2: switch statement Vs. if…else if statement

```
#include <iostream>
using namespace std;

int main(){
   float score;
   cout<<"Test score 10%: ";
   cin>>score;

   switch(score){
    case 0:
    case 1:
    case 2:
    case 3:
    case 4:
    case 5: cout<<"Grade F"<<endl; break;
    case 6: cout<<"Grade D"<<endl; break;
    case 7: cout<<"Grade C"<<endl; break;
    case 8: cout<<"Grade B"<<endl; break;
    case 9:
    case 10: cout<<"Grade A"<<endl;
    default: cout<<"Invalid mark\n";
   }
   return 0;
}
```

```
#include <iostream>
using namespace std;

int main(){
   float score;
   cout<<"Test score 10%: ";
   cin>>score;

    if (score >= 0 && score <= 5)
        cout<<"Grade F"<<endl; break;
    else if (score >= 6)
        cout<<"Grade D"<<endl; break;
    else if (score >= 7)
        cout<<"Grade C"<<endl; break;
    else if (score >= 8)
        cout<<"Grade B"<<endl; break;
    else if (score == 9 || score == 10)
        case 10: cout<<"Grade A"<<endl;
    else
       cout<<"Invalid mark\n";
}
```

Which is selection statement is best fit for a problem which has range selection?

2. Selection statements (cont'd)

Dangling else problem

```
int x = 4;
if ( x % 2  == 0)
    if ( x < 0 )
        cout<<x<<"is an even, -ve number";
else
    cout<<x<<"is an odd number";
```

- ✓ What does it display for x=4?
- ✓ The problem is that it displays **"4 is an odd number"** message for positive even numbers and zero
- ✓ Reason is that, although indentation says the reverse, else belongs to second (inner) if
  - ➢ else belongs to the most recent if
- ✓ **Solution: using brace {} as follow**

```
int x = 4;
if ( x % 2  == 0)
{
    if ( x < 0 )
        cout<<x<<"is an even, -ve number";
}
else
    cout<<x<<"is an odd number";
```

2. Selection statements (cont'd)

Short-circuit Evaluation

- ✓ Evaluate the first (leftmost) Boolean sub-expression.
- ✓ If its value is enough to judge about the value of the entire expression, then stop there. Otherwise continue evaluation towards right.
- ✓ Example:

```
if (count != 0 && scores/count < 60)
 {
        cout<<"low average";
 }
```

- ✓ In this example, if the value of count is zero, then first sub-expression becomes false and the second one is not evaluated.
- ✓ In this way, we avoid "division by zero" error (that would cause to stop the execution of the program)
- ✓ Alternative method to avoid division by zero without using short-circuit evaluation:

```
if (count != 0){
  if (scores/count < 60){
      cout<<"low average";
     }
  }
```

Reading Resources/Materials

Chapter 5 & 6:

- ✓ **Diane Zak**; An Introduction to Programming with C++ [8th Edition], 2016 Cengage Learning

Chapter 4:

- ✓ **Gary J. Bronson**; C++ For Engineers and Scientists [3rd edition], Course Technology, Cengage Learning, 2010

Chapter 2 (section 2.4):

- ✓ **Walter Savitch**; Problem Solving With C++ [10th edition], University of California, San Diego, 2018

Chapter 4:

- ✓ **P. Deitel , H. Deitel**; C++ how to program, [10th edition], Global Edition (2017)

Thank You
For Your Attention!!