# Chapter 7

# Selection

3rd Edition

Computing Fundamentals with C++

Rick Mercer
Franklin, Beedle & Associates

# Goals

- Recognize when to use the Guarded Action pattern
- Implement the Guarded Action pattern with the if statement
- use relational operators such as < and >
- create and evaluate expressions with the logical operators
- use `bool` objects
- understand the Alternative Action pattern
- implement the Alternative Action pattern with the C++ `if...else` statement
- implement the Multiple Selection n with `if...else` and switch
- solve problems using the Multiple Selection pattern

# Why do we need selection?

- Programs must often anticipate a variety of situations
- Consider an Automated Teller Machine:
  - ATMs must serve valid bank customers.
  - They must also reject invalid PINs
  - The code that controls an ATM must permit different requests
  - Software developers must implement code that anticipates all possible transactions

# Selective Control

- Programs often contain statements that may not always execute
- Sometimes a statement may execute and other certain conditions it may not
  - Reject invalid PIN entries at an ATM instead of allowing a withdrawal
- We say an action is guarded from executing

# The Guarded Action Pattern

| | |
|---|---|
| **Pattern:** | Guarded Action |
| **Problem:** | Execute an action only under certain conditions |
| **General Form:** | **`if(`** *logical-expression* **`)`**<br>*true-part* |
| **Code Example:** | `if(aStudent.GPA() >= 3.5)`<br>`  deansList.push_back(aStudent);` |

# The `if` statement

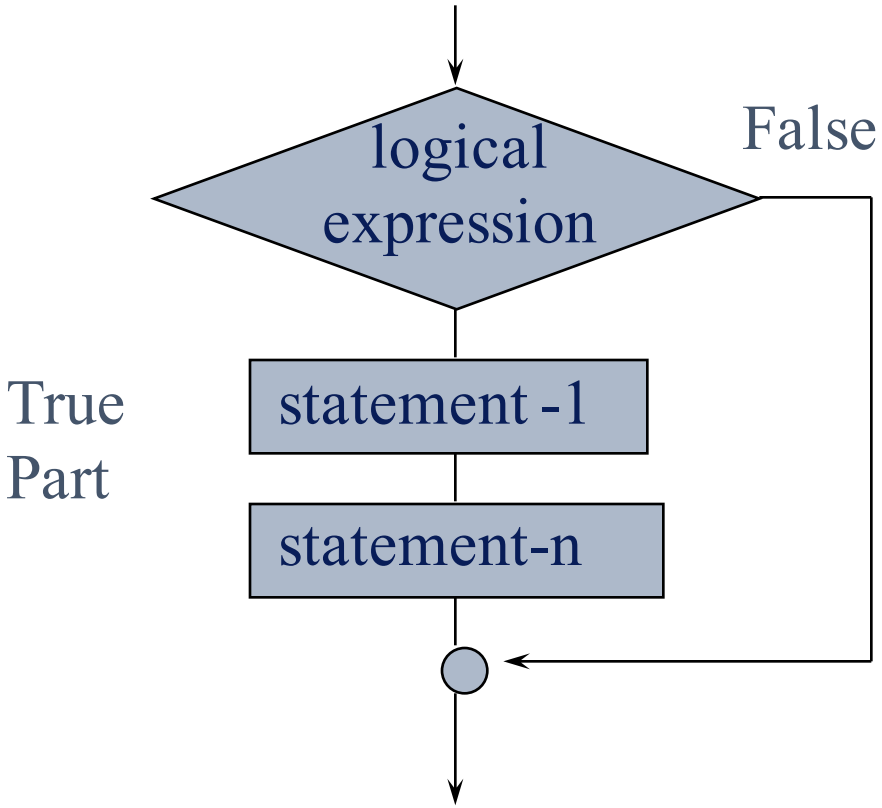
- The `if` is the first statement that alters strict sequential control. General form

  `if (` *logical-expression* `)`

  *true-part* `;`

- *logical-expression*: any expression that evaluates to nonzero (true) or zero (false)
- In C++, almost everything is true or false

# Flow of control with `if`




- After the logical expression of the if statement evaluates, the true-part executes only if the logical expression is true.

# Example if statement

```
double hours = 38.0;
// Add 1.5 hours for hours>40.0 (overtime)
if (hours > 40.0)
 hours = 40.0 + 1.5 * (hours - 40.0);
```

- What is the value of hours when `hours` is

```
double hours = 38.0;  // _______
double hours = 40.0;  // _______
double hours = 42.0;  // _______
```

# Another way

- The if statement could also be written with a block

```
double hours = 42.0;
if (hours > 40.0) {
  hours = 40.0 + 1.5 * (hours - 40.0);
}
```

- Sometimes the block is required  consider using { }

```
if (hours > 40.0) {
  regularHours = 40.0;
  overtimeHours = hours - 40.0;
}
```

# Relational Operators

- Logical expressions often use relational operators:

| | |
|---|---|
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |
| == | Equal |
| != | Not equal |

# Logical Expressions

- Which expressions are true, which are false?

```
int n1 = 78;
int n2 = 80;

n1 < n2          // _____
n1 >= n2         // _____
(n1 + 35) > n2   // _____
n1 > 78          // _____
n1 == n2         // _____
n1 != n2         // _____
```

# Logical Expressions with strings

- Which expressions are true, which are false?

```
string s1 = "Carson";
string s2 = "Carly";

s1 < s2    _____
s1 > s2    _____
s1 == s2   _____
s1 != s2   _____
s1 > s2    _____
s2 < "C"   _____
```

# Relational Operators in `if` Statements

```
double x = 59.0;
if (x >= 60.0) {
  cout << "passing";
}
if (x < 60.0) {
  cout << "failing";
}
```

- What is the output when `x` is 59, 60, and 61?

```
double x = 59.0;   _______
double x = 60.0;   _______
double x = 61.0;   _______
```

# Programming Tip

- Using = for == is a common mistake. For example the following two statements are legal, but ...

```
int x = 25;
// Because assignment statements evaluate
// to the expression on the right of =, x=1
// is always 1, which is nonzero, or true
if (x = 1) // should be (x == 1)
  cout << "I'm always displayed";
```

- So consider putting the literal first

```
if (1 = x) // This is a compiletime error
```

# The Alternative Action Pattern

- Programs often contain statements that select between one set of actions or another
- Examples
  - withdraw or deposit money
  - pass or fail the entrance requirements
- This is the Alternative Action Pattern
  - choose between two alternate sets of actions

# Alternative Action

| | |
|---|---|
| Pattern: | Alternative Action |
| Problem: | Must choose one action from two alternatives |
| Outline: | `if (`*true-or-false-condition* is true `)`<br>  *action-1*<br>`else`<br>  *action-2* |
| Code | `if(finalGrade >= 60.0)`<br>`  cout << "passing" << endl;`<br>`else`<br>`  cout << "failing" << endl;` |

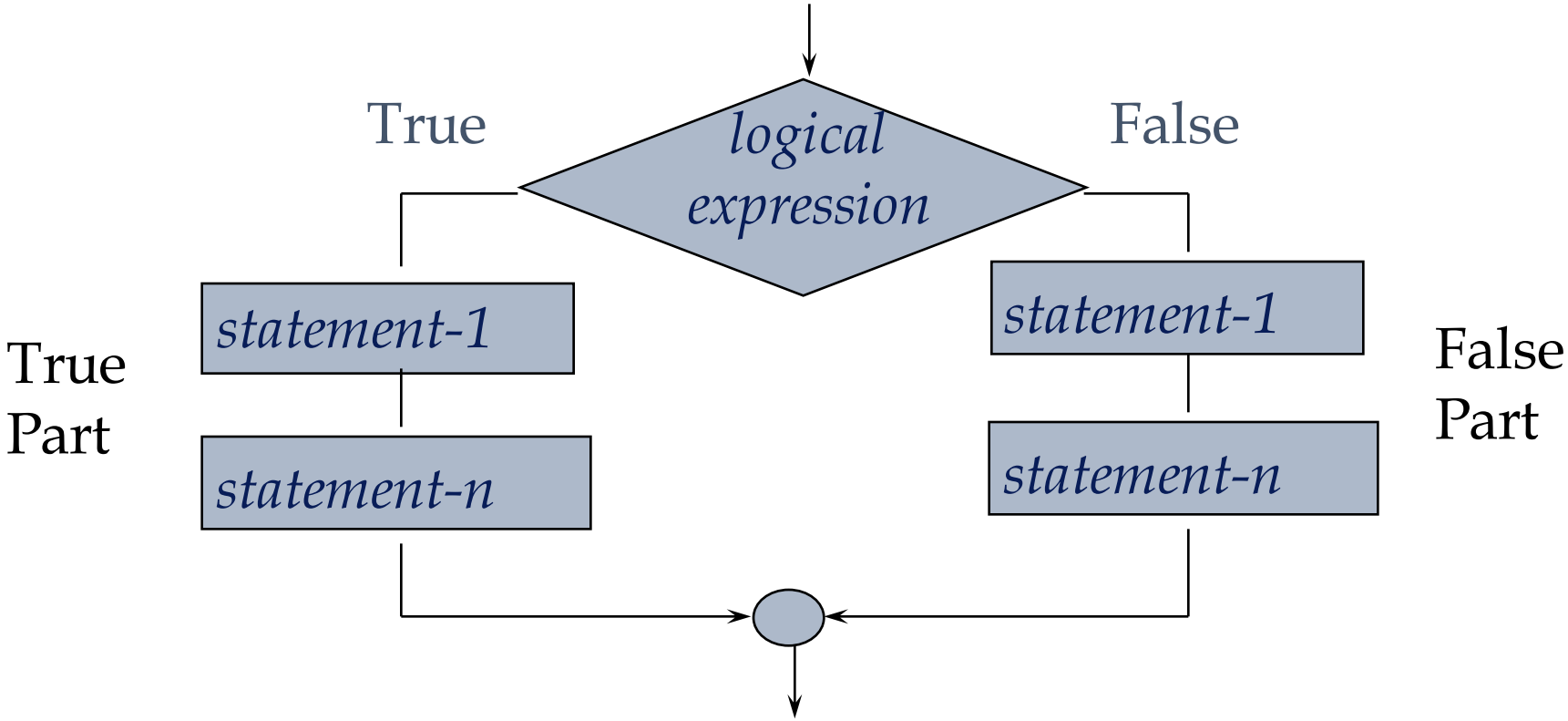
# if-else

```
if  ( logical-expression )
    true-part ;
else
    false-part ;
```

- When the logical expression evaluates to true, the true-part executes and the false-part is disregarded
- When the logical expression is false, only the false-part executes.

# The if...else statement

- The if...else statement allows two alternate courses of action



# if...else Example

```
if (miles > 24000)
  cout << "Tune-up " << miles-24000 << " miles overdue";
else
  cout << "Tune-up due in " << 24000-miles << " miles";
```

| miles | Output? |
| --- | --- |
| 30123 | _____ |
| 2000 | _____ |
| 24000 | _____ |

# The Block { } with if-else

- Blocks may be used even when

```
if (miles > 24000) {
  cout << "Tune-up " << miles-24000 << " miles overdue";
} else {
  cout << "Tune-up due in " << 24000-miles << " miles";
}
```

- Using curly braces all the time helps avoid difficult to detect errors

# bool Objects

- The standard `bool` type stores one of two values

  `true` and `false`

- A `bool` object stores the result of a logical expression:

```
bool ready = false;
cout << ready << endl; // 0 for false
double hours = 4.5;
ready = hours >= 4.0;
cout << ready << endl; // 1 for true
```

# bool Functions

- It is common to have functions that return one of the `bool` values (true or false)

```
// true if n is odd
bool odd(int n) {
  return (n % 2) != 0;
}
```

```
// Use the odd function
int main() {
   int anInt = 3;
   if( odd(anInt) )
     anInt++;
   cout << anInt;  // 4
   return 0;
 }
```

# Boolean Operators

- A logical operator (&& means `and`) used in an if...else statement

```
int test = 50;
if( (test >= 0) && (test <= 100) )
  cout << "Test is in range";
else
  cout << "**Warning--Test out of range";
```

- The code describes whether or not the value of `test` is in the range of 0 through 100 inclusive.

# Truth Tables for Boolean Operators

- Truth tables for the Logical (Boolean) operators

  **!** (not) **||** (or) **&&** (and)

| ! (not) | | || (or) | | && (and) | |
|---|---|---|---|---|---|
| Expression | Result | Expression | Result | Expression | Result |
| `! false` | `true` | `true || true` | `true` | `true && true` | `true` |
| `! true` | `false` | `true || false` | `true` | `true && false` | `false` |
| | | `false || true` | `true` | `false && true` | `false` |
| | | `false || false` | `false` | `false && false` | `false` |

- You can also use these more readable operators instead of **!** **||** **&&**

  `not` `or` `and`

# More Precedence Rules

- The following slide summarizes all operators used in this textbook (we've seen them all now)
- Precedence: most operators are evaluated (grouped) in a left-to-right order:

  `a/b/c/d` is equivalent to `(((a/b)/c)/d)`
- Assignment operators group in a right-to-left order so the expression

  `x=y=z=0` is equivalent to `x=(y=(z=0))`

# Operators used in this book

| | *Operator* | *Description* | *Grouping* |
|---|---|---|---|
| *Highest* | `::`<br>`()` | Scope resolution<br>Function call | Left to right |
| *Unary* | `!, +, -` | Not, unary plus/minus | Right to left |
| *Multiplicative* | `*  /  %` | Multiply/divide/remainder | Left to right |
| *Additive* | `+  -` | Binary plus, minus | Left to right |
| *Input/Output* | `>>  <<` | Extraction / insertion | Left to right |
| *Relational* | `<  >`<br>`<=  >=` | Less/Greater than<br>Less/Greater or equal | Left to right |
| *Equality* | `==  !=` | Equal, Not equal | Left to right |
| *and* | `&&` | Logical and | Left to right |
| *or* | `\|\|` | Logical or | Left to right |
| *Assignment* | `=` | Assign expression | Right to left |

# Applying Operators and Precedence Rules

- Use the precedence rules to evaluate the following expression:

```
int j = 5;
int k = 10;
bool TorF;

TorF = (j * (1 + k) > 55) ||
       ((j + 5 <= k) && (j > k));
```

- What is assigned to `TorF`?_____

# The `bool ||` with a Grid Object

```
#include "Grid.h"  // for class Grid

// Return true if the mover is at an end of the world
bool moverOnEdge(const Grid & aGrid) {
  return(   aGrid.row()==0          // on north edge?
         || aGrid.row()==aGrid.nRows()-1 // on sout?
         || aGrid.column()==0       // on west edge?
         || aGrid.column()==aGrid.nColumns()-1 );
}

int main() {
  Grid tarpit(5, 10, 4, 4, east);
  if( moverOnEdge(tarpit) )
    cout << "On edge" << endl; // On edge
  else
    cout << "Inside border" << endl;
  return 0;
}
```

# Short Circuit Boolean Evaluation

- C++ logical expressions evaluate sub-expressions in a left to right order
- Sometimes the evaluation can stop early
- This will never evaluates `sqrt` of a negative number:

```
if((x >= 0.0) && (sqrt(x) <= 2.5))
```

- `test > 100` will not be evaluated when `test` is negative

```
if(test < 0 || test > 100)
```

# A `bool` member function

- Consider changing `BankAccount::withdraw` so it only withdraws money if the balance is sufficient
- Also have it return `true` in this case
- Have it return `false` when there are insufficient funds, after the change to the state of the object
- First change heading in class `BankAccount` that is in the file BankAccount.h

```
bool withdraw(double withdrawalAmount);
```

# a `bool` member function

- Also change implementation in BankAccount.cpp

```
bool BankAccount::withdraw(double amount) {
  // post: return true if withdrawal was
  // successful or false with insufficient funds
  if (balance >= amount) {
    balance = balance - amount;
    return true;
  }
  return false;
}
```

# Multiple Selection

- *Nested logic:* When one control structure contains another similar control structure
    - an `if else` inside another `if else`
    - allows selections from 3 or more alternatives
- We must often select one alternative from many

| **Pattern:** | Multiple Selection |
|---|---|
| **Problem:** | Must execute one set of actions from three alternatives. |
| **Outline:** | if ( *condition 1* is true)<br>  execute action *1*<br>else if( *condition 2 is true*)<br>  execute action *2*<br>// ...<br>else if( *condition n-1 is true*)<br>  execute action *n-1*<br>else<br>  execute action *n* |
| **Code Example:** | `if(grade < 60)`<br>`  result = "F";`<br>`else if(grade < 70)`<br>`  result = "D";`<br>`else if(grade < 80)`<br>`  result = "C";`<br>`else if(grade < 90)`<br>`  result = "B";`<br>`else`<br>`  result = "A";` |

# Example of Multiple Selection nested if...else

```
if(GPA < 3.5)
    cout << "Try harder" << endl;
 else
  if(GPA < 4.0)
       cout << "Dean's List";
     else
       cout << "President's list";
```

The false part is another if...else

| GPA | *Output:* |
|---|---|
| 3.0 | _____ |
| 3.6 | _____ |
| 4.0 | _____ |

# Multiple Returns

- It's possible to have multiple return statements in a function *terminate when the first return executes*

```
string letterGrade(double percentage) {
  if (percentage >= 90)
    return "A";
  if (percentage >= 80)
    return "B";
  if (percentage >= 70)
    return "C";
  if (percentage >= 60)
    return "D";
  return "F";  // percentage < 0
}
```

# Testing Multiple Selection

- It is often difficult and unnecessary to test every possible value *imagine all those doubles 0.1, 0.001, 0.0001,...*
- Testing our code in "most" branches can prove dangerously inadequate
- Each branch through the multiple selection should be tested

# Perform Branch Coverage Test

- To correctly perform branch coverage testing we need to do the following:
  - Establish a set of data that ensures all paths will execute *the statements after the logical expressions*
  - Execute the code *call the function* with the nested logic for all selected data values
  - Observe that the code behaves correctly for ***all*** data *compare program output with expected results*
- This is glass box testing *when you look at the code*

# Boundary Testing

- Boundary testing involves executing the code using the boundary (cutoff) values
- What grade would you receive with a percentage of 90 using this code

```
string letterGrade(double percentage) {
 if (percentage > 90)
   return "A";
 if (percentage >= 80)
   return "B";
 . . .
```

# function `assert`

- So far testing has been done by printing with `cout`
- This requires a careful inspection of the `cout` statements and the associated output
- C++ has an `assert` function takes a `bool` argument to more easily test our functions
  - If the argument is false, C++ will inform you with a line of output that begins with Assertion failed
  - In this case, `assert` will terminate the program
  - If all expressions in all calls to the `assert` function are true, there is ***no*** output

# function `assert`

- Consider this test driver that uses assert
  - If letterGrade is correct there will be no output

```
int main() {
  assert("A" == letterGrade(100.0));
  assert("A" == letterGrade(90.0));
  assert("B" == letterGrade(89.9));
  assert("B" == letterGrade(80.0));
  assert("C" == letterGrade(79.9));
  assert("C" == letterGrade(70.0));
  assert("D" == letterGrade(69.9));
  assert("D" == letterGrade(60.0));
  assert("F" == letterGrade(59.0));
  assert("F" == letterGrade(59.9));
}
```

# function `assert`

- If any assertion is wrong, you will get a message
  - The program terminates, the 3rd assert is not executed

```
int main() {
  assert("A" == letterGrade(100.0));
  assert("E" == letterGrade(90.0));
  assert("A" == letterGrade(59.9));
}
```

```
Assertion failed: ("E" == letterGrade(90.0)),
function main, file ../src/testGrade.cpp, line 29.
```

# The switch Statement

```
switch ( switch-expression ) {
  case value-1 :
     statement(s)-1
     break;        ...     // many cases are allowed
  case value-n :
     statement(s)-n
     break;
  default:
     default-statement(s)
}
```

# Switch control

- When a switch statement is encountered:
  - the switch-expression is evaluated. This value is compared to each case value until switch-expression equals the case value.
  - All statements after the colon : are executed.
- It is important to include the break statement
- The switch expression must evaluate to one of C++'s integral types

```
int   char   enum
```

# `char` Objects

- A char object stores 1 character

  `'A'  'x'  'c'  '?'  ' '  '1'  '.'`

- Or 1 escape sequence

| Escape Sequence | Meaning |
|---|---|
| `'\n'` | new line |
| `'\"'` | double quote in a char |
| `'\''` | single quote in a char |
| `'\\'` | forward slash |
| `'\t'` | tab |

# Example switch statement:

```
char option = '?';
 cout << "Enter W)ithdraw  D)eposit B)alances: ";
 cin >> option;
 switch (option) {
 case 'W':
   cout << "Withdraw" << endl;
   break;
 case 'D':
   cout << "Deposit" << endl;
   break;
 case 'B':
   cout << "Balance" << endl;
   break;
 default:
   cout << "Invalid" << endl;
 } // end switch
```

Show output when

option == '?' _____

option == 'W' _____

option == 'B' _____

option == 'A' _____

option == 'Q' _____