Week 2
Branching & Looping
Gaddis: Chapters 4 & 5

CS 5301
Fall 2014
Jill Seaman

Relational Operators

- relational operators (result is bool):
  
  ```
  int x=90;
  int n=6;
  
  7 < 25
  89 == x
  x % 2 != 0
  8 + 5 * 10 <= 10 * n
  ```

- operator precedence:

```
* / %
+ -
<> <= >=
== !=
=
```

Which operation happens first? next? ...

```
int x, y;
...
...  x < y -10 ...
...  x * 5 >= y + 10 ...

bool t1 = x > 7;
bool t2 = x * 5 >= y + 10;
```

if/else

- if and else

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

  - if expression is true, statement 1 is executed
  - if expression is false, statement2 is executed

```
double rate, monthlySales;
if (monthlySales > 3000)
  rate = .025;
else
  rate = .029;
```

- the else is optional:

  ```
  if (expression)
    statement
  ```

  - if expression is true, statement is executed, otherwise statement is skipped

Block or compound statement

- a set of statements inside braces:

```
{
  int x;
  cout << "Enter a value for x: " << endl;
  cin >> x;
}
```

- This allows us to use multiple statements when by rule only one is allowed.

```
int number;
cout << "Enter a number" << endl;
cin >> number;
if (number % 2 == 0)
{
  number = number / 2;
cout << "0";
}
else
{
  number = (number + 1) / 2;
cout << "1";
}
Nested if/else

- if-else is a statement. It can occur as a branch of another if-else statement.

```c
if (testScore < 60)
    grade = 'F';
else {
    if (testScore < 70)
        grade = 'D';
    else {
        if (testScore < 80)
            grade = 'C';
        else if (testScore < 90)
            grade = 'B';
        else
            grade = 'A';
    }
}
```

This is equivalent to the code on the left. It is just formatted differently.

Logical Operators

- logical operators (values and results are bool):
  - ! not
  - && and
  - || or

```c
int x=6;
int y=10;
a. x == 5 && y <= 3
b. x > 0 && x < 10
c. x == 10 || y == 10
d. x == 10 || x == 11
e. !(x > 0)
f. !(x > 6 || y == 10)
```

Operator precedence:

- operator precedence:
  - * / %
  - + -
  - < > <= >=
  - == !=
  - &&
  - ||

switch statement

- switch stmt:

```c
switch (expression) {
    case constant: statements
    ...
    case constant: statements
    default: statements
}
```

- execution starts at the case labeled with the value of the expression.
- if no match, start at default
- use break to exit switch (usually at end of statements)

Example:

```c
switch (ch) {
    case 'a':
        cout << “Option A”;
        break;
    case 'b':
        cout << “Option B”;
        break;
    default: cout << “Invalid choice”;
}
```

More assignment statements

- Compound assignment

```c
operator usage equivalent syntax:
+ = x += e; x = x + e;
- = x -= e; x = x - e;
* = x *= e; x = x * e;
/ = x /= e; x = x / e;
```

- increment, decrement

```c
operator usage equivalent syntax:
++ x++; ++x; x = x + 1;
-- x--; --x; x = x - 1;
```
while loops

- while
  
  while (expression) 
  statement

  if expression is true, statement is executed, repeat

  Example:

  ```
  int number;
  cout << "Enter a number, 0 when finished: ";
  cin << number;
  while (number != 0) {
    cout << "You entered " << number << endl;
    cout << "Enter the next number: ";
    cin << number;
  }
  cout << "Done" << endl;
  ```

  output:

  Enter a number, 0 when finished: 22
  You entered 22
  Enter the next number: 5
  You entered 5
  Enter the next number: 0
  Done

two kinds of loops

- conditional loop
  
  - execute as long as a certain condition is true

- count-controlled loop:
  
  - executes a specific number of times
    - initialize counter to zero (or other start value).
    - test counter to make sure it is less than count.
    - update counter during each iteration.
  
  ```
  int number = 1;
  while (number <= 3) {
    cout << "Student" << number << endl;
    number = number + 1; // or use number++
  }
  cout << "Done" << endl;
  ```

- do-while loops
  
  - The test is at the end, statement ALWAYS executes at least once.
  
  ```
  int number;
  do {
    cout << "Enter a number, 0 when finished: ";
    cin << number;
    cout << "You entered " << number << endl;
  } while (number != 0);
  ```

for loops

- for: 
  
  for (expr1; expr2; expr3) 
  statement

  * equivalent to:
    
    expr1;
    while (expr2) {
      statement
      expr3;
    }

  * Good for implementing count-controlled loops:
    
    pattern: for (initialize; test; update)

    ```
    for (int number = 1; number <= 3; number++) {
      cout << "Student" << number << endl;
    }
    cout << "Done" << endl;
    ```

do-while loops

- do while:
  
  - statement is executed.
    if expression is true, then repeat
  
  ```
  int number;
  do {
    cout << "Enter a number, 0 when finished: ";
    cin << number;
    cout << "You entered " << number << endl;
  } while (number != 0);
  ```
Keeping a running total (summing)

- Example:

```cpp
int days;
float total = 0.0; //Accumulator

cout << "How many days did you run? ";
cin >> days;

for (int i = 1; i <= days; i++)
{
    float miles;
    cout << "Enter the miles for day " << i << ": ";
cin >> miles;
    total = total + miles;
}

cout << "Total miles run: " << total << endl;
```

Sentinel controlled loop

- Use a special value to signify end of the data:

```cpp
float total = 0.0; //Accumulator
float miles;

cout << "Enter the miles you ran each day, ";
cout << "one number per line.\n";
cout << "Then enter -1 when finished.\n";
cin >> miles;
while (miles != -1)
{
    total = total + miles;
cin >> miles;
}

cout << "Total miles run: " << total << endl;
```

- Sentinel value must NOT be a valid value

Nested loops

- When one loop appears in the body of another
- For every iteration of the outer loop, we do all the iterations of the inner loop

```cpp
for (row=1; row<=3; row++)  //outer
{
    for (col=1; col<=3; col++) //inner
        cout << row * col << ": ";
    cout << endl;
}
```

Output:

```
1 2 3
2 4 6
3 6 9
```

continue and break Statements

- Use `break` to terminate execution of a loop
- When used in a nested loop, terminates the inner loop only.

- Use `continue` to go to end of current loop and prepare for next repetition
- `while`, `do-while` loops: go immediately to the test, repeat loop if test passes
- for loop: immediately perform update step, then test, then repeat loop if test passes