CSCI 123 Introduction to Programming Concepts in C++

Brad Rippe

More Flow
• Using Boolean Expressions
• Multi-way Branches
• More about C++ Loop Statements
• Designing Loops
Questions???
Some Rules

• If both operands are type int
  – Then the result is an int type

  $23 \times 38 = \text{integer result}$
  $23 \div 38 = \text{integer result}$

• If one of the operands is a double
  – Then the result is a double type

  $23.56 \times 38 = \text{double result}$
  $23.56 \div 38 = \text{double result}$
  $5 \div 2.0 = \text{double result}$
Variable Types and Operations

- What results from the following operations?
  - 2+2
  - 2.3 * 4.0
  - 3/2
  - 3.0/2
Type Coercion

• **Type Coercion**: automatic conversion of an operand to another data type

• **Promotion**: convert to a higher type

• **Demotion**: convert to a lower type
Rules Are

1) When operating on values of different data types, the lower one is promoted to the type of the higher one.

2) When using the $=$ operator, the type of expression on right will be converted to type of variable on left.
What’s the output?

```cpp
int main() {
    bool operand1 = 0;
    int operand2 = 23;

    bool result;
    result = operand1 + operand2;

    cout << "Result = " << result << endl;
    return 0;
} // you don’t want to do this
```
if else statement (basic structure)

if(count == 1) {
    ...do some work...
} else if(count == 2) {
    ...do some other work...
} else {
    ...do some other work...
}
while and do/while loops

```java
while (count < 9) {
    ... do some work ...
}

do {
    ... do some work ...
} while (count < 9);
```
while loops

```cpp
int count = 0;
while(count < 3) {
    cout << "Count is " << count << endl;
    count++;
}
cout << "I'm out!!!\n";
```
Counter-Control Loop

- Counter-controlled loops are those loops that are executed a fixed number of times. The number of iterations is known before commencing the loop.

```cpp
int i = 1;
while (i <= 5) {
    cout << "Square of " << i;
    cout << " is " << (i * i);
    cout << endl;
    ++i;
}
```
Sentinel-Control Loop

- Event-controlled loops are those loops that are executed an indefinite number of times until some condition occurs.

```cpp
const int SENTINEL = -1;
int posInt;
int sum = 0;
cout << "Enter number: \n";
cin >> posInt;
while (posInt != SENTINEL) {
    sum += posInt;
    cout << "Enter number: \n";
cin >> posInt;
}
```
Pseudocode

- Informal language
- Doesn’t execute on a computer
- Helps with developing algorithms
- Described with some executable statements
- Can be easily translated into C++
if(sum < 100.00) {
    // calculate shipping
    // with UPS ground
    // process order
} else {
    // free shipping
    // no calculation
    // process order
}
Pseudocode Example

• `cout << “Your current salary?”`  
• `cin >> salary`  
• Calculate the user’s retro pay for six months and `cout << retro pay`  
• Calculate the new annual salary and `cout << annual salary`  
• Calculate the new monthly salary and `cout << new monthly salary`
Boolean Expressions

• Type bool allows declaration of variables that have a value of true or false

• Operators
  – <, >, !=, ==, <=, >=
  – &&, ||, !

```cpp
bool executeIf = true;
if(executeIf) {
    cout << "executed if\n";
}
```

ifExpressions.cpp
&& - Truth tables

```cpp
if ( gender == 1 && age >= 65 )
    seniorFemales++;
```

<table>
<thead>
<tr>
<th>Exp1</th>
<th>Expr2</th>
<th>Expr1 &amp;&amp; Expr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>Exp1</td>
<td>Expr2</td>
<td>Expr1</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>--------</td>
</tr>
<tr>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>
Truth Table Example
Short Circuit Evaluations

• `&&` - logical and
• `||` - logical or

```cpp
if(x > 3 || y > 3)
    cout << "x > 3 or y > 3" << endl;
```
bool type

• Evaluates to a true or false value
• New to C++
• C uses 0 (zero) for false and a positive integer for true
• Side effects of C
• This is a compilation error in Java
• Enumerations are types with a set of integer values

```c
enum Speed { FAST, NORMAL, SLOW };  
Speed speed;  // expects one of the enums
```

• Enums are capitalized because they are defined as constants

• They are user-defined types, not given to us like `int`, `char`, `double`, etc.
Enums

- Can be set to explicit values or not

```cpp
enum Speed {
    FAST = 120, NORMAL = 65, SLOW = 20;
}

if(speed == FAST) {
    // mash the pedal
} else if(speed == NORMAL) {
    // let mom drive
} else if(speed == SLOW) {
    // let grandma drive
}
```
if(freeway == 91) {
    // prepare to go slow
} else if(freeway != 91 && day == FRIDAY) {
    // prepare to stop
}
Is the following legal?

int temperature = 80;
if(temperature >= 100) {
    string msg = "Wow scorching day!\n";
} else if(temperature >= 90) {
    string msg = "It's a hot day!\n";
} else if(temperature >= 80) {
    string msg = "Not bad!\n";
} else {
    string msg = "Wonderful weather!\n";
}
cout << msg << endl;

Assume it’s written in the main()
string grade;
// declaration and initialization of
// finalscore & midterm

if (finalscore >= 90)
  if (midterm >= 90)
    grade = "A+";
  else
    grade = "A";
else if (finalscore >= 80)
  grade = "B";
else if (finalscore >= 70)
  grade = "C";
else
  grade = "F";

cout << "grade= " << grade << endl;
Who owns the else?

```plaintext
if (a >= 5)
    c = c + 1;
else
    d = 5;
```

```plaintext
if (a >= 5) {
    c = c + 1;
    if (b < 20)
        b = 2;
} else {
    d = 5;
}
```

// why the second set of
// {} braces
string grade;
// declaration and initialization of
// finalscore & midterm

if (finalscore >= 90) {
    if (midterm >= 90) {
        grade = "A+";
    } else {
        grade = "A";
    }
} else if (finalscore >= 80) {
    grade = "B";
} else if (finalscore >= 70) {
    grade = "C";
} else {
    grade = "F";
}

cout << "grade= " << grade << endl;
Block rules

OK

OK

CANNOT
multi-way if/else example

```cpp
int score = 8;
char gpa = ' ';  // gpa declared, initialized
if (score >= 9) {
    gpa = 'A';  // false
} else if (score >= 8) {
    gpa = 'B';  // true
} else if (score >= 7) {
    gpa = 'C';
} else {
    gpa = 'F';
}

cout << "grade=" << gpa;  // display gpa as B
// value of gpa?
```

What is advantage of the additional `else`s?
Multi-way if/else advantage

```cpp
int score = 8;  // value of gpa?
char gpa = ' ';  // value of gpa?
if (score >= 9) {
    gpa = 'A';  // value of gpa?
} else if (score >= 8) {
    gpa = 'B';  // value of gpa?
} else if (score >= 7) {
    gpa = 'C';  // value of gpa?
} else {
    gpa = 'F';  // value of gpa?
}

cout << "grade=" << gpa;  // value of gpa?
```

```cpp
int score = 8;  // value of gpa?
char gpa = 'D';  // value of gpa?
if (score >= 9) {
    gpa = 'A';  // value of gpa?
} else if (score >= 8) {
    gpa = 'B';  // value of gpa?
} else if (score >= 7) {
    gpa = 'C';  // value of gpa?
}
cout << gpa;  // value of gpa?
```
multi-way if/else example

```c++
int score = 8;
char gpa = ' ';
if (score >= 7) {
    gpa = 'C';
} else if (score >= 8) {
    gpa = 'B';
} else if (score >= 9) {
    gpa = 'A';
} else {
    gpa = 'F';
}
cout << "grade=" << gpa;
// value of gpa?
```

- score = 8
- gpa declared, initialized
- true
- gpa = ‘C’
- skip the else block
- display gpa=C still WRONG
Switch Statement

• Syntax:

```java
switch (value) {  // must be an integral result
    case value1:
        statements;
    ...
    break;
    case valueN:
        statements;
    ...
    break;
    default:
        statements;
    ...
}
```
switch statement

```cpp
char drink = ' '; cin >> drink; switch(drink) {
    case 'a':
        price += 3.50;
        break;
    case 'b':
        price += 3.00;
        break;
    default :
        drink = 'q';
        break;
}
```

*Case expressions* must be constant integer expressions or literals, and must be unique in the switch statement.
more switch

• What happens with the following code?

• If `break` statements are not used with the case statements, execution falls through to the next case
Break statement

• Used to stop execution in the current block
• Also used to exit a switch statement
• Useful to execute a single `case` statement without executing the statements following it

• Can be used to break out of loop blocks as well as switch blocks
switch statements

• Contains case labels
• Optional default case
• Similar to the if/else if statement

```c
switch(expression) {
    case 'a':
        break;
    
}
```

Controlling Expression
number++ vs ++number

• (number++) returns the current value of number, then increments number
  – An expression using (number++) will use the value of number BEFORE it is incremented
• (++number) increments number first and returns the new value of number
  – An expression using (++number) will use the value of number AFTER it is incremented
• Number has the same value after either version!
Increment and Decrement Operators

• Pre-increment Operator
  – \( ++a \)
  ```
  int a = 2;
  int b = (++a)*22;
  ```

• Post-increment Operator
  – \( a++ \)
  ```
  int a = 2;
  int b = (a++)*22;
  ```
for loop structure syntax

for (<initializing statement>; <continuation assertion>; <increment statement>) {
  <body statement>;
  ...  
  <body statement>;
}

for statement

for ( int counter = 1; counter <= 10; counter++ )
For code example

double {  
cout << "Display #" << i;  
i += 2;  
} while ( i <= 10 );

for (int j = 0; j <= 10; j+=2 ) {  
cout << "For Loop Display #" << j;  
}
**Break and Continue**

- **break** quits the loop without executing the rest of the statements in the loop.
  - Used in switch statements
  - Can be used in loops, but not common
  - Interrupts the current loop (or a switch case)

- **continue** stops the execution of the current iteration and goes back to the beginning of the loop to begin the next iteration. Executes if continuation condition is true.
  - Used with loops, but not common
  - Interrupts the current loop
Break Example

for(int i = 0; i < 10; i++) {
    if(i == 5)
        break;
    cout << "i is " << i << endl;
}

Output:

i is 0
i is 1
i is 2
i is 3
i is 4

i only lives in the for loop. This is known as variable scope.
for(int i = 0; i < 10; i++) {
    if( i < 5 )
        continue;
    cout << "i is " << i << endl;
}

Output:
i is 5
i is 6
i is 7
i is 8
i is 9

i only lives in the for loop.
This is known as variable scope.
Nested for loop

```cpp
for(int j = 0; j < 5; j++) {
    cout << "outer: j is " << j << endl;
    for(int i = 0; i < 5; i++) {
        cout << " inner: i is " << i << endl;
    }
}
```
What does the following output?

```cpp
for (int a = 1; a <= 1; a++) {
    cout << a++;
    cout << a;
}
```
Debugging Loops

• Common errors involving loops include
  – Off-by-one errors in which the loop executes one too many or one too few times
  – Infinite loops usually result from a mistake in the Boolean expression that controls the loop
Fixing Off By One Errors

• Check your comparison: should it be < or <=?

• Check that the initialization uses the correct value

• Does the loop handle the zero iterations case?
Fix Infinite Loops

• Check the direction of inequalities: < or >?

• Test for < or > rather than equality (= =)

• Check to see that you didn’t use the assignment operator (=) instead of the comparison (==)
Starting Over

- Sometimes it is more efficient to throw out a buggy program and start over
  - The new program will be easier to read
  - The new program is less likely to be as buggy
  - You may develop a working program faster than if you repair the bad code
    - *The lessons learned in the buggy code will help you design a better program faster*