Week 2
Control Constructs & Functions
Gaddis: Chapters 4, 5, 6

CS 5301
Spring 2014
Jill Seaman

Relational and Logical Operators

• relational operators (result is bool):
  
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 &lt; 25</td>
</tr>
<tr>
<td>= = x</td>
</tr>
<tr>
<td>x % 2 == 0</td>
</tr>
<tr>
<td>8 + 5 * 10 &lt;= 100 * n</td>
</tr>
</tbody>
</table>

• logical operators (values and results are bool):

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>not</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example:</th>
</tr>
</thead>
<tbody>
<tr>
<td>x &lt; 10 &amp;&amp; x &gt; 0</td>
</tr>
<tr>
<td>y == 10</td>
</tr>
<tr>
<td>!(a == b)</td>
</tr>
</tbody>
</table>

• operator precedence:

Control structures: if else

• if and else

```
if (expression) {
    statement1
    else if (expression2) {
        statement2
    }
    else {
        statement3
        else if (expression3) {
            statement4
        }
    }
}
```

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

• the else is optional:

```
if (expression) {
    statement
}
```

• nested if else

```
if (expression1) {
    statement1
    else if (expression2) {
        statement2
    }
    else if (expression3) {
        statement3
    }
    else {
        statement4
    }
}
```

Control structures: switch

• switch stmt:

```
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:

```
switch (ch) {
    case 'a':
    case 'A':
        cout << "Option A";
        break;
    case 'b':
    case 'B':
        cout << "Option B";
        break;
    default:
        cout << "Invalid choice";
        break;
}
```
More assignment statements

- Compound assignment

<table>
<thead>
<tr>
<th>operator</th>
<th>usage</th>
<th>equivalent syntax:</th>
</tr>
</thead>
<tbody>
<tr>
<td>+=</td>
<td>x += e;</td>
<td>x = x + e;</td>
</tr>
<tr>
<td>-=</td>
<td>x -= e;</td>
<td>x = x - e;</td>
</tr>
<tr>
<td>*=</td>
<td>x *= e;</td>
<td>x = x * e;</td>
</tr>
<tr>
<td>/=</td>
<td>x /= e;</td>
<td>x = x / e;</td>
</tr>
</tbody>
</table>

- increment, decrement

<table>
<thead>
<tr>
<th>operator</th>
<th>usage</th>
<th>equivalent syntax:</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>x++; ++x;</td>
<td>x = x + 1;</td>
</tr>
<tr>
<td>--</td>
<td>x--; --x;</td>
<td>x = x - 1;</td>
</tr>
</tbody>
</table>

Control structures: loops

- while

```java
while (expression) {
    statement
}
```

if expression is true, statement is executed, repeat

- for:

```java
for (expr1; expr2; expr3) {
    statement
    expr3;
}
```

equivalent to:

```java
expr1;
while (expr2) {
    statement
    expr3;
}
```

- do while:

```java
do {
    statement
    expr3;
} while (expression);```

statement is executed. if expression is true, then repeat

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

```java
for (row=1; row<=3; row++)  //outer
    for (col=1; col<=3; col++) //inner
        cout << row * col << 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 to test, repeat loop if test passes
- for loop: perform update step, then test, then repeat loop if test passes
Function Definitions

- Function definition pattern:
  ```
  datatype identifier (parameter1, parameter2, ...) {
  statements . . .
  }
  ```
  Where a parameter is:
  ```
  datatype identifier
  ```

- **datatype**: the type of data returned by the function.
- **identifier**: the name by which it is possible to call the function.
- **parameters**: Like a regular variable declaration, act within the function as a regular local variable. Allow passing arguments to the function when it is called.
- **statements**: the function's body, executed when called.

Function Call, Return Statement

- **Function call expression**
  ```
  identifier ( expression1, . . . )
  ```
  Causes control flow to enter body of function named `identifier`.
  - `parameter1` is initialized to the value of `expression1`, and so on for each parameter
  - `expression1` is called an **argument**.

- **Return statement**:
  ```
  return expression;
  ```
  inside a function, causes function to stop, return control to caller.
  - The value of the return `expression` becomes the value of the function call

Example: Function

```
// function example
#include <iostream>
using namespace std;
int addition (int a, int b) {
  int result;
  result=a+b;
  return result;
}
int main () {
  int z;
  z = addition (5,3);
  cout << "The result is " << z <<endl;
}
```

- What are the parameters? arguments?
- What is the value of: `addition (5,3)`?
- What is the output?

Void function

- A function that returns no value:
  ```
  void printAddition (int a, int b) {
  int result;
  result=a+b;
  cout << "the answer is: " << result << endl;
  }
  ```
  - use `void` as the return type.
  - the function call is now a statement (it does not have a value)

```
int main () {
  printAddition (5,3);
}
```
Prototypes

- In a program, function definitions must occur before any calls to that function.
- To override this requirement, place a prototype of the function before the call.
- The pattern for a prototype:

```
datatype identifier (type1, type2, ...);
```

- the function header without the body (parameter names are optional).

Arguments passed by value

- **Pass by value**: when an argument is passed to a function, its value is *copied* into the parameter.
- It is implemented using variable initialization (behind the scenes):
  ```
  int param = argument;
  ```
- Changes to the parameter in the function body do not affect the value of the argument in the call.
- The parameter and the argument are stored in separate variables; separate locations in memory.

Example: Pass by Value

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

void changeMe(int);

int main() {
    int number = 12;
    cout << "number is " << number << endl;
    changeMe(number);
    cout << "Back in main, number is " << number << endl;
    return 0;
}
void changeMe(int myValue) {
    myValue = 200;
    cout << "myValue is " "myValue " << endl;
}
```

Scope of variables

- For a given variable definition, in which part of the program can it be accessed?
  - **Global variable** (defined outside of all functions): can be accessed anywhere, after its definition.
  - **Local variable** (defined inside of a function): can be accessed inside the block in which it is defined, after its definition.
  - **Parameter**: can be accessed anywhere inside of its function body.
- Variables are destroyed at the end of their scope.
More scope rules

- Variables in the same exact scope cannot have the same name
  - Parameters and local function variables cannot have the same name
  - Variable defined in inner block can hide a variable with the same name in an outer block.

```c
int x = 10;
if (x < 100) {
    int x = 30;
    cout << x << endl;
}
cout << x << endl;
```

Output: 30

- Variables defined in one function cannot be seen from another.