MA400: Financial Mathematics

Introductory Course

Lecture 3: Control Structure

Relational operators

C++ provides 4 binary operators to compare the values of arithmetic expressions.

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

These require two operands (left and right) and their results are boolean values, **true** or **false**.

```
bool b;
b = 2 > 1;      // Assigns b the value true
b = 2 < 1;      // Assigns b the value false
b = 2 >= 2;     // Assigns b the value true
b = 3.4 <= 2.4 // Assigns b the value false
```

Relational operators

The operands of relational operators can be expressions:

```
int i = 1, j = 2;
bool b = ( i + 1 ) >= ( j - 2 ); // TRUE: 2 >= 0
```

```
int i = 1 = 1, j;
bool b = i >= ( j = 2 );          // FALSE 1 < 2
```

Associativity and precedence of relational operators

Relational operators have associativity and precedence:

- all share the same precedence, and
- all associate from left to right.

With reference to our earlier table, they form a row that sits:

- below the $*$, / and % precedence group, and
- above the + and - precedence group.

See Capper, Appendix B, p.521, for more details.

```
bool b = i < j < k;     // Equivalent expressions
bool b = ( i < j ) < k; // Equivalent expressions
```

Exercises: Relational operators

Exercise

What are the results for

```
bool b = i < j < k;
```

when we have

- `a = 1, b = 2, c = 3`?
- `a = 1, b = 4, c = 3`?

Exercises: Relational operators

Exercise

What are the results for `b1` and `b2` in the following statements?

```
double x = 110.0;
bool b1, b2;
b1 = x > ( x / 13.0 ) * 13.0;
b2 = x < ( x / 13.0 ) * 13.0;
```

Logical operators

Logical operators return a result of type **bool**, i.e. a value of either **true** or **false**.

The logical negation operator is denoted by the token `!`.
It is a unary operator, and exchanges **true** values for **false** values.

The logical AND operator is denoted by the token `&&`.
It is a binary operator and only returns a value of **true** if and only if both its operands have value **true**.

The logical OR operator is denoted by the token `||`.
It is a binary operator and returns a value of true if one or both of its operands have value **true**.

Associativity and precedence of logical operators

!	logical negation	right to left
	*, /, %	
	+, -	
&&	logical AND	left to right
||	logical OR	left to right
	=	

As with all unary operators, logical negation associates from right to left.
The binary logical operators associate from left to right.

Exercise: Logical operators

What values are assigned to the booleans `p`, `q`, `r` and `s`?

```
bool a = false, b = true, c = false;
bool p = a || b && !c;
bool q = !a && c || b;
bool r = !( b || c );
bool s = !( b && !c );
```

Equal and Not Equal Operators: == and !=

== and != are binary operators that are used to test operands of arithmetic type.
They result in a boolean value of either **true** or **false**.

```
int i = 0, j = 10;
double x = 10.0, y = 3.0, z = 10.0 / 3.0;

bool b1 = i != j;      // Assigns b1 the value true
bool b2 = y == x;      // Assigns b2 the value false

bool b3 = x == j;      // Assigns b3 the value true

bool b4 = z * y == x; // Assigns b4 either true or false

bool b5 = y == 3.0;   // Assigns b5 the value true
```

Exercise: What are the boolean values above if we replace == with !=?

A word of warning

A particularly nasty error that can arise with the equality operator is the following typo:

```
double x = 3.0;          // Assigns x the value 3,0
bool b = ( x = 5.0 );    // Assigns x the value 5.0
                         // 5.0 is non-zero, so
                         // Assigns b the value true
```

It should be clear here that the intention was more likely to check if `x` was equal to 5.0, which would have returned a value for `b` of **false**.

Expressions and statements

A **statement** is the smallest independent unit of a C++ program. Simple statements are always terminated by a semicolon, `;`.

The simplest statement possible is the **null statement**:

```
;
```

An **expression** is the smallest unit of computation. That is, it is a statement that resolves to a value.

Blocks and scope

Using a pair of braces, { }, inside a program, we are able to group together statements, definitions and declarations into a **compound statement**.

A block, or compound statement, is equivalent to a single statement.

However, there is no terminating semicolon after the trailing brace, }.

A key property of such blocks is that any definition or assignments made within a block **are only valid within that block**.

Blocks and scope

```
double x = 1.111;       // x has type double, value 1.111

{
  int x = 2;            // x has type int, value 2
}

. . . . .               // x has type double, value 1.111

{
  char x = 'x';         // x has type char, value x
  {
    int x = 3;          // x has type int, value 3
  }
. . . . .               // x has type char, value x
}
. . . . .               // x has type double, value 1.111
```

Blocks and scope

We say that identifiers within a block are **hidden** from outside the block.

Where a particular identifier is **visible**, or valid, is known as the **scope** of that identifier.

The **if** statement

The basic form of an **if** statement is

```
if ( condition )
  statement
```

- The condition is any valid arithmetic expression;
- if the condition evaluates to **true**, the statement is executed'
- otherwise the statement is not executed.

The **if** statement

```
if ( i == 0 ) {
  x = 100.00   // x is assigned the value 100 if i is 0
}
```

```
if ( !i ) {
  x = 100.00   // Equivalent to above
}
```

```
if ( !i ) {
  x = 3.142;   // If i is zero, then
  y = 100.0;   // All 3 statements
  z *= x;      // are executed
}
```

The **if** statement: A word of warning.

Forgetting braces around compound statements can also lead to unforseen results.

```
if ( !i )
  x = 3.142;
  y = 100.0;
  z *= x;
```

This is equivalent to

```
if ( !i )
  x = 3.142
y = 100.0;
z *= x;
```

So only the `x` assignment is conditional upon the value of `i`, and not all three of them.

The **if** statement: A word of warning. Again.

Consider the following fragment

```
if ( temp = 100 ) {
  boiling = true;
}
```

While correct, this is actually equivalent to:

```
temp = 100;          // Assigns temp the value 100
boiling = true;      // Assigns boiling the value true
```

It is more likely that the desired statement is to set **boiling** equal to **true** only if **temp** equals 100.

```
if ( temp == 100 ) {
  boiling = true;
}
```

The **if else** statement

```
if ( condition_1 )
  statement_1
else if ( condition_2 )
  statement_2
. . . . .
else
  statement_n
```

Once again, the conditions are valid arithmetic expressions, and the n statements may be compound statements.

1. The program will start to evaluate each of the n conditions in the order they are specified.
2. If `condition_i`, for some `i` = 1, ..., `n`, evaluates to true, then `statement_i` is executed. Control then passes beyond the final statement, `statement_n`.
3. If none of the first `n-1` conditions evaluate to true, then `statement_n` is executed, and control passes beyond the final statement.

The **if else** statement

There is no requirement for a final `else` statement, which is essentially a default action.

In such a situation, it is possible for an **if else** statement to be executed with no action taken – if none of the conditions evaluate to true.

Exercise: The **if else** statement

Exercise: Convert this fragment into a full program (see Capper, §4.5.2, p.54).

```
double x, y, pi = 3.142;
int i;
// Get user to input an integer value for i
if ( i == 0 ) {
  x = pi;
  y = 2.0 * pi;
}
else if ( i == 1 ) {
  x = 2.0 * pi;
  y = 0.0;
}
else {
  x = 0.0;
  y = 0.0;
}
```

Exercise: A word of warning – The dangling else

```
// Get user to input an integer value for i
if ( i == 0 ) {
  if ( j == 0 )
    cout << "Both i and j are zero\n";
}
else {
  cout << "i is non-zero\n";
}
```

What is the intention here? What is the result?

The problem is that the **else** is dangling – it could be attached to either of the **if**s.

By default, it is associated with the nearest one, so that the code in braces will never be executed.

To fix this, we can enclose the inner **if** statement in braces.

The program should now see the intended **if else** statement.

Exercise: Solving a quadratic equation

Exercise
Amend the quadratic program example you have been given to deal with all three possible root cases.

Exercise
Amend your quadratic program to correctly handle anything the user might try to input.

See Capper, p.56.

The **switch** statement

```
switch ( expression ) {
case constant_1:
  statement_1;
case constant_2:
  statement_2;
. . . . .
case constant_n:
  statement_n;
default:
  last_statement;
}
```

The **switch** statement: An example

```
cout << "Menu:\n\t1 Bermudan\n\t2 Asian\n"
     << "Enter a number to choose an option.\n";
int option;
cin >> option;

switch ( option ) {
case 1:
  cout << "We shall look at Bermudan options.\n";
  break;
case 2:
  cout << "We shall look at Asian options.\n";
  break;
default:
  cout << option << " is not a valid option.\n";
  break;
}
```

```
Menu:
        1 Bermudan
        2 Asian
Enter an initial to choose an option.
1
We shall look at Bermudan options.
```

The **break** statement

It is important to note that the flow of control is being affected by the **break** statements, and not the **default** or **case** statements.

The **break** statement can only occur within a **switch** statement, or from within an iteration loop (which we shall see next).

The **break** statement

```
cout << "Menu:\n\t1 Bermudan\n\t2 Asian\n"
     << "Enter a number to choose an option.\n";
int option;
cin >> option;

switch ( option ) {
case 1:
  cout << "We shall look at Bermudan options.\n";
  break;
case 2:
  cout << "We shall look at Asian options.\n";
  break;
default:
  cout << option << " is not a valid option.\n";
  break;
}
```

The **default** statement

Note that this can actually occur anywhere in the **switch** statement, but it is usually good practice to place it after all the **case** statements.

When placed at the end, the **break** statement associated with the default case is redundant, but again it is usually good practice to include it.

The Iteration Statements

C++ has three different statements to handle iteration:

- **while**
- **for**
- **do**

In fact, it is possible to use these interchangeably (given some minor modifications of the code), and which one you will use will depend on preference and the circumstances.

The **for** statement

```
for ( initialize ; condition ; change )
  statement
```

1. Firstly, the **initialize** statement is executed.
2. The **condition** expression is then evaluated.
3. If it is found to be **true**, then the **statement** is executed.
4. The **change** expression is then evaluated.
5. The iteration then returns to step 2 above.
6. If the **condition** expression evaluates to **false** at any step, the iteration is terminated.

Example: Summing the first *n* integers

```
int i, n, sum;
n = 3;
sum = 0;
for ( i = 0 ; i <= n ; ++i ) {
  sum += i;
}
cout << "Sum of first " << n << " integers is " << sum
     << " with i = " << i << '\n';
```

i	sum
0	0
1	1
2	3
3	6
4	

```
Sum of first 3 integers is 6 with i = 4.
```

The **for** statement: Some examples

What is the final value of `sum` in the following code fragments?

```
int n = 3, sum = 0;
int i;
for ( i = 10 ; i <= n ; ++i )
  sum += i;
```

```
int sum = 0, i = 1, n = 3;
for ( ; i <= n ; ++i )
  sum += i;
```

```
int sum = 0, i = 1, n = 3;
for ( ; i <= n ; )
  sum += i++;
```

The **for** statement: The loop variable

It is possible, and convenient, to define the loop variable within the **initialize** statement.

```
int n = 3, sum = 0;
for ( int i = 0 ; i <= n ; ++i ) {
  sum += i;
}
```

However, be aware that the scope of your loop variable is only for the duration for the **for**.

```
int n = 3, sum = 0, i = 10;      // i has value 10
for ( int i = 0 ; i <= n ; ++i ) {
  sum += i;
}
```

After the loop, `sum = 6`, while `i = 10` still (and not 4).

The **while** statement

```
while ( condition )
  statement
```

1. The **condition** expression is first evaluated.
2. If it evaluates to **true**, then the **statement** is executed.
3. The **condition** expression is then evaluated again.
4. If it evaluates to **true** again, then the **statement** is executed again.
5. However, the moment the **condition** expression evaluates to **false**, then the iteration is terminated.

The **while** statement: An example

What function does this code fragment attempt to calculate?

```
// Get user to input an integer greater than 1 for n
--n;
int gamma = n;
while ( n > 2 ) {
   --n;
  gamma *= n;
}
```

The Gamma function for positive integers (greater than 2).

$$\Gamma(n) = (n-1)! \qquad n > 2$$

Note how no terminating semicolon was needed after the compound statement.

The **while**: Summing the first n integers

We have already seen this done with a **for** statement.

```
int i = 0, sum = 0;
while ( i <= n ) {
  sum += i++;
}
```

The **do** statement

```
do
  statement
while ( condition ) ;
```

Here, the terminating semicolon is required.

1. The fist thing that happens is that the **statement** is executed, regardless of the **condition**
2. The **condition** is then evaluated.
3. If it evaluates to **true**, then the **statement** is executed again.
4. The **condition** expression is then evaluated again.
5. The moment the **condition** expression evaluates to **false**, the iteration is terminated.

The **do** statement: Summing the first *n* integers

For comparison sake, here is the sum algorithm implemented in a **do** statement.

```
int i = 0, sum = 0;
do {
  sum += i++;
}
while ( i <= n );
```

Note the **statement** is placed within braces, despite being only one line, since it differentiates this from a **while** loop with an empty statement.

The **do** statement: An example

```
int option;
do {
  cout << "Menu:\n\t1\n\t2\n\t3"
       <<
  cin >> option;
} while ( option < 1 || option > 3 );
switch ( option ) {
  case 1:
    // Case 1 code
  . . . . .
  default:
    // Default code
}
```

The **do** statement: A comparison

So the main difference between the **do** statement and the others is that the **statement**:

- is executed at least once in the **do** statement, but
- may never be executed at all in the **while** and **for** statements,

Exercise: Iteration Statements

Exercise

Which of the iterations in the following code segments many never terminate? Justify your conclusions.

a)
```
int sum = 1;
for ( unsigned i = 10 ; i >= 0 ; --i )
  sum *= 2 * i + 1;
```

b)
```
double i = 10, sum = 1;
while ( i != 0 )
  sum *= 2 * i-- + 1;
```

c)
```
int i = 0;
double sum = 1.0;
while ( 1 ) {
  sum *= 2 * i++ + 1;
  if ( i = 10 )
    break;
}
```

Exercise: Fibonacci sequence

The Fibonacci sequence is a sequence of **integers**, defined recursively by:

$$u_1 = 1, \qquad u_2 = 1, \qquad u_n = u_{n-1} + u_{n-2} \quad n \geq 3.$$

Write a program that prompts for a positive integer, `n`, and lists the first `n` members of the sequence.
Notice how u_n increases very rapidly with `n` and soon exceeds the largest integer that can be represented as a fundamental type on your computer.
Verify your results by modifying your program to check that:

a) $u_1 + u_2 + \ldots + u_n = u_{n+2} - 1$,

b) $u_n^2 - u_{n-1} u_{n+1} = (-1)^{n-1}$.

Additional topics

For those who are comfortable with what we have seen, three other concepts you might wish to explore are:

- using the **break** statement in iteration loops;
- the **continue** statement in iteration loops: this causes an immediate jump to the next iteration whenever it is encountered;
- the conditional expression operator – `?:` – which is of the form:

```
condition ? result_1 : result_2
```

The **break** statement

What does the following code fragment do?

```
int test = 0;
for ( int i = 0; i < 3 ; ++i ) {
  cout << "Testing i = " << i << "\n";
  for (int j = 0 ; j < 3 ; ++j) {
    cout << "\tTesting j = " << j << "\n";
    for ( int k = 0 ; k < 3 ; ++k ) {
      test = 10 * k;
      if ( test > 10 )
        break;
      cout << "\t\tTesting k = " << k << "\n";
    }  // The break leaves us inside the i and j loops
  }
}
```

While the `k` loops are curtailed at `k = 2` due to the break, the `i` and `j` loops run their full course.

The **continue** statement

What are the final values of x and y?

```
double x = 0.0, y = 0.0;
for ( int i = 0; i < 10 ; ++i ) {
  ++x;
  if (i == 5)
    continue;
  ++y;
}
cout << "x = " << x << ", y = " << y <<  '\n';
```

Conditional Expression Operator

This is the only ternary operator defined in C++, taking the form:

```
condition ? result_1 : result_2
```

It requires three operands and returns either **result_1** if the **condition** evaluates to **true**, or **result_2** if it evaluates to **false**

For example, the following yield equivalent results:

```
max = ( i > j ) ? i : j;
```

and

```
if ( i > j )
  max = i;
else
  max = j;
```