Lecture Three
Operators and Precedence

Operator Precedence and Associativity

- When different operators are used in the same expression, the normal rules of arithmetic apply. All C++ operators have a precedence and associativity:
- Precedence—when an expression contains two different kinds of operators, which should be applied first?
- Associativity—when an expression contains two operators with the same precedence, which should be applied first?

Operator Precedence and Associativity

2 + 3 * 4

- Should it be interpreted as :- (2 + 3) * 4 (that is, 20), or rather is: 2 + (3 * 4)(that is, 14) the correct interpretation?
- As in normal arithmetic, in C++ multiplication and division have equal importance and are performed before addition and subtraction. We say multiplication and division have precedence over addition and subtraction.

Operator Precedence and Associativity

- To see how associativity works, consider the expression

2 - 3 - 4

- The two operators are the same, so they have equal precedence. Should the first subtraction operator be applied before the second, as in:

  (2 - 3) – 4 (that is, -5), or rather is 2 - (3 - 4) (that is, 3).

Operator Precedence and Associativity

- Consider the statement: w = x = y = z;
- This is legal C++ and is called chained assignment. Assignment can be used as both a statement and an expression. The statement x = 2 assigns the value 2 to the variable x.

Operator Precedence and Associativity

W=x=y=z;

- Since assignment is right associative, the chained assignment example should be interpreted as: w = (x = (y = z)); which behaves as follows:
  - The expression y = z is evaluated first. z’s value is assigned to y, and the value of the expression y = z is z’s value.

Operator Precedence and Associativity

Arity	Operators	Associativity
Unary	+, −	
Binary	*, /, %	Left
Binary	+, -	Left
Binary	=	Right

More Arithmetic Operators

- A variable may increase by one or decrease by five. The statement

  x = x + 1;

- increments x by one, making it one bigger than it was before this statement was executed. C++ has a shorter statement that accomplishes the same effect:

  x++;

- This is the increment statement. A similar decrement statement is available:

  x--; // Same as x = x - 1;

Increment and Decrement Operators

- These statements are more precisely post-increment and post-decrement operators. There are also pre-increment and pre-decrement forms, as in

```
--x;    //Same as  x=x-1;
++y;    //Same as  y=y+1;
```

- When they appear alone in a statement, the pre- and post- versions of the increment and decrement operators work identically. Their behavior is different when they are embedded within a more complex statement.

Increment and Decrement Operators

```
#include <iostream>
int main() {
int x1 = 1, y1 = 10,x2 = 100, y2 = 1000;
cout <<"x1=" << x1 << ", y1=" << y1
<<", x2="   << x2 << ", y2=" << y2 << '\n';
y1 = x1++;
cout <<"x1=" << x1 << ", y1=" << y1
<<", x2="   << x2 << ", y2=" << y2 << '\n';
y2 = ++x2;
cout <<"x1=" << x1 << ", y1=" << y1
<<", x2="<< x2 << ", y2=" << y2 << '\n';
}
```

Increment and Decrement Operators

- C++ provides a more general way of simplifying a statement that modifies a variable through simple arithmetic. For example, the statement

  x = x + 5;

- can be shorted to: x += 5; This statement means “increase x by five.” Any statement of the form:

  x op= exp;

  Where: x is a variable.

  op= is an arithmetic operator combined with the assignment operator; for our purposes, the ones most useful to us are +=, -=, *=, /=, and %=.

  exp is an expression compatible with the variable x.

Increment and Decrement Operators

- Arithmetic reassignment statements of this form are equivalent to:
- x = x op exp; This means the statement: x *= y + z; is equivalent to x = x * (y + z);
- Do not accidentally reverse the order of the symbols for the arithmetic assignment : x =+ 5 ;Notice that the + and = symbols have been reversed.
- The compiler interprets this statement as if it had been written: x = +5 ;that is, assignment and the unary operator. This assigns x to exactly five instead of increasing it by five

Bitwise Operators

- C++ provides a few other special-purpose arithmetic operators. These special operators allow programmers to examine or manipulate the individual bits that make up data values.
- They are known as the bitwise operators. These operators consist of &, |, ^, , >>, and <<.
- The bitwise and operator, &, takes two integer sub expressions and computes an integer result. The expression e1 & e2 is evaluated as follows:
  1. If bit 0 in both e1 and e2 is 1, then bit 0 in the result is 1; otherwise, bit 0 in the result is 0.
  2. If bit 1 in both e1 and e2 is 1, then bit 1 in the result is 1; otherwise, bit 1 in the result is 0.
  3. If bit 2 in both e1 and e2 is 1, then bit 2 in the result is 1; otherwise, bit 2 in the result is 0.
  4. If bit 31 in both e1 and e2 is 1, then bit 31 in the result is 1; otherwise, bit 31 in the result is 0.

Bitwise Operators

- The bitwise or operator, |, takes two integer sub expressions and computes an integer result. The expression e1 | e2 is evaluated as follows:
  1. If bit 0 in both e1 and e2 is 0, then bit 0 in the result is 0; otherwise, bit 0 in the result is 1.
  2. If bit 1 in both e1 and e2 is 0, then bit 1 in the result is 0; otherwise, bit 1 in the result is 1.
  3. If bit 2 in both e1 and e2 is 0, then bit 2 in the result is 0; otherwise, bit 2 in the result is 1. ...
  4. If bit 31 in both e1 and e2 is 0, then bit 31 in the result is 0; otherwise, bit 31 in the result is 1.

Bitwise Operators

- The bitwise exclusive or (often referred to as xor) operator (^) takes two integer sub expressions and computes an integer result. The expression e1 ^ e2 is evaluated as follows:
  1. If bit 0 in e1 is the same as bit 0 in e2, then bit 0 in the result is 0; otherwise, bit 0 in the result is 1.
  2. If bit 1 in e1 is the same as bit 1 in e2, then bit 1 in the result is 0; otherwise, bit 1 in the result is 1.
  3. If bit 2 in e1 is the same as bit 2 in e2, then bit 2 in the result is 0; otherwise, bit 2 in the result is 1. ...
  4. If bit 31 in e1 is the same as bit 31 in e2, then bit 31 in the result is 0; otherwise, bit 31 in the result is 1.

Bitwise Operators

- The bitwise negation operator ( ) is a unary operator that inverts all the bits of its expression. The expression e is evaluated as follows:
  1. If bit 0 in e is 0, then bit 0 in the result is 1; otherwise, bit 0 in the result is 0.
  2. If bit 1 in e is 0, then bit 1 in the result is 1; otherwise, bit 1 in the result is 0.
  3. If bit 2 in e is 0, then bit 2 in the result is 1; otherwise, bit 2 in the result is 0 . ...
  4. If bit 31 in e is 0, then bit 31 in the result is 1; otherwise, bit 31 in the result is 0.

Bitwise Operators

- Shift left (<<). The expression x << y, where x and y are integer types, shifts all the bits in x to the left y places. Zeros fill vacated positions. The bits shifted off the left side are discarded. The expression 5 << 2 evaluates to 20, since 510 = 1012 shifted two places to the left yields 101002 = 2010.
- Shift right (>>). The expression x >> y, where x and y are integer types, shifts all the bits in x to the right y places. What fills the vacated bits on the left depends on whether the integer is signed or unsigned (for example, int vs. unsigned):
  - For signed values the vacated bit positions are filled with the sign bit (the original leftmost bit).
  - For unsigned values the vacated bit positions are filled with zeros.

Bitwise Operators

- The bits shifted off the right side are discarded. The expression 5 >> 2 evaluates to 1, since 510 = 1012 shifted two places to the left yields 0012 = 2010 (the original bits in positions 1 and 0 are shifted off the end and lost). Observe that x >> y is equal to x 2y.

```
#include<iostream.h>
Int main(){
int x, y;
cout << "Please enter two integers;" :
cin >> x >> y;
cout << x << " & " << y << " = " << (x & y) <<
'\n'; cout << x << " | " << y << " = " << (x | y)
<< '\n'; cout << x << " ^ " << y << " = " << (x ^
y) << '\n'; cout << " " << x << " = " << x << '\n;'
cout << x << " << " << 2 << " = " << (x << 2) <<
'\n'; cout << x << " >> " << 2 << " = " << (x >> 2)
<< '\n{;'
```

Bitwise Operators

Assignment	Short Cut
`x = x & y;`	`x &= y;`
`x = x | y;`	`x |= y;`
`x = x ^ y;`	`x ^= y;`
`x = x << y;`	`x <<= y;`
`x = x >> y;`	`x >>= y;`

Logical Expressions

- Relational and logical operators – result is boolean-valued
  - == equal to `counter == 0`
  - != not equal to `counter != 0`
  - > greater than `counter > 0`
  - < less than `counter < 0`
  - >= greater than or equal to `counter >= 0`
  - <= less than or equal to `counter <= 0`
  - && logical and `0 < i && i < 10`
  - | | logical or `i <= 0 || i >= 10`
  - ! logical not `! done`

Boolean Expressions

- An expression whose value is true or false
- In C:
  - integer value of 0 is “false”
  - nonzero integer value is “true”
- Example of Boolean expressions:

  Relational operator

  - age < 40
  - graduation_year == 2010


```
#include <iostream.h>
#include <stdbool.h>

int main()
{
    const bool trueVar = true, falseVar = false;
    const int int3 = 3, int8 = 8;

    cout<<"No 'boolean' output type\n”;
    cout<<"bool trueVar: %d\n",trueVar;
    cout<<"bool falseVar: %d\n\n",falseVar;
    cout<<"int int3: %d\n",int3);
    cout<<"int int8: %d\n",int8);
}
```

Library that defines: bool, true, false

What does the output look like?

Boolean Expressions

- An expression whose value is true or false
- In C:
  - integer value of 0 is “false”
  - nonzero integer value is “true”
- Example of Boolean expressions:
  - age < 40
  - graduation_year == 2010

Relational operator

Boolean Expressions

```
#include <iostream.h>
#include <stdbool.h>

int main()
{
    const bool trueVar = true, falseVar = false;
    const int int3 = 3, int8 = 8;

    cout<<"No 'boolean' output type\n”;
    cout<<"bool trueVar: %d\n",trueVar;
    cout<<"bool falseVar: %d\n\n",falseVar;
    cout<<"int int3: %d\n",int3);
    cout<<"int int8: %d\n",int8);
}
```

Library that defines: bool, true, false

What does the output look like?

Boolean Expressions

// Example3 (continued…)

```
cout<<"\nint3 comparators\n";
cout<<"int3 == int8: %d\n",(int3 == int8);
cout<<"int3 != int8: %d\n",(int3!=int8);
cout<<"int3 < 3:  %d\n",(int3 <  3);
cout<<"int3 <= 3: %d\n",(int3 <= 3);
cout<<"int3 > 3:  %d\n",(int3 >  3);
cout<<"int3 >= 3: %d\n",(int3 >= 3);
```

Comparing values of two integer constants

What does the output look like?

More Examples

- char myChar = ‘A’;
  - The value of myChar==‘Q’ is false (0)
- Be careful when using floating point equality comparisons, especially with zero, e.g. myFloat==0

Suppose?

- What if I want to know if a value is in a range?
- Test for: 100 ≤ L ≤ 1000?

You can't do…

This code is WRONG and will fail.

```
if(100 <= L <= 1000)
{
    cout<<"Value is in range…\n");
}
```

Why this fails…

C ++ Treats this code this way

```
if((100 <= L) <= 1000)
{
    cout<<"Value is in range…\n");
}
```

Suppose L is 5000. Then 100 <= L is true, so (100 <= L) evaluates to true, which, in C, is a 1. Then it tests 1 <= 1000, which also returns true, even though you expected a false.

Compound Expressions

- Want to check whether -3 <= B <= -1
  - Since B = -2, answer should be True (1)
- But in C++, the expression is evaluated as
  - ((-3 <= B) <= -1) (<= is left associative)
  - (-3 <= B) is true (1)
  - (1 <= -1) is false (0)
  - Therefore, answer is 0!

Compound Expressions

- Solution (not in C): (-3<=B) and (B<=-1)
- In C: (-3<=B) && (B<=-1)
- Logical Operators
  - And: &&
  - Or: ||
  - Not: !

Compound Expressions

```
#include <iostream.h>

int main()
{
   const int A=2, B = -2;

   cout<<"Value of A is %d\n", A;
   cout<<"0 <= A <= 5?: Answer=%d\n", (0<=A) && (A<=5);

   cout<<"Value of B is %d\n", B;
   cout<<"-3 <= B <= -1?: Answer=%d\n", (-3<=B) && (B<=-1);
}
```

Compound Expressions

```
#include <iostream.h>

int main()
{
   const int A=2, B = -2;

   cout<<"Value of A is %d\n", A);
   cout<<"0 <= A <= 5?: Answer=%d\n", (0<=A) && (A<=5);

   cout<<"Value of B is %d\n", B);
   cout<<"-3 <= B <= -1?: Answer=%d\n", (-3<=B) && (B<=-1);
}
```

```
>./a.out
Value of A is 2
0 <= A <= 5?: Answer=1
Value of B is -2
-3 <= B <= -1?: Answer=1
```

Correct Answer!!!

Compound Expressions

```
#include <stdio.h>

int main()
{
   const int A=2, B = -2;

   cout<<"Value of A is %d\n", A;
   cout<<"0 <= A <= 5?: Answer=%d\n", (0<=A) && (A<=5);

   cout<<"Value of B is %d\n", B;
   cout<<"-3 <= B <= -1?: Answer=%d\n", (-3<=B) && (B<=-1);
}
```

```
>./a.out
Value of A is 2
0 <= A <= 5?: Answer=1
Value of B is -2
-3 <= B <= -1?: Answer=1
```

Correct Answer!!!

Truth Tables

p	q	Not !p	And p && q	Or p || q
True	True			
True	False			
False	True			
False	False			

Truth Tables

p	q	Not !p	And p && q	Or p || q
True	True	False		
True	False	False		
False	True	True		
False	False	True		

Truth Tables

p	q	Not !p	And p && q	Or p || q
True	True		True	
True	False		False	
False	True		False	
False	False		False	

Truth Tables

p	q	Not !p	And p && q	Or p || q
True	True			True
True	False			True
False	True			True
False	False			False

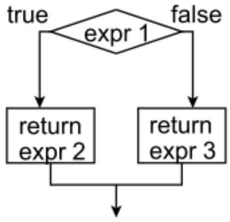
Truth Tables

Our comparison operators:
< <= == != >= >

p	q	Not !p	And p && q	Or p || q
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

Conditional Expressions

- Based on the Conditional Operator ?:
- `(expr 1)?(expr 2 :expr 3)`
  - If expr 1 is true, expr 2 is the value of the overall expression
  - If expr 1 is false, expr 3 is the value of the overall expression
  - Parentheses are not syntactically required
  - Typically used because ? has a high Precedence
- `max = (x > y) ? x : y;`
- `min = (x < y) ? x : y;`
- `index = (index+1 == size) ? 0 : ++index;`



```
A = 4, B = 2;
A + B > 5 && (A = 0) < 1 > A + B - 2
```

(((A + B) > 5)   &&   (((A=0) < 1) > ((A + B) – 2)))

((6 > 5)   &&   (((A=0) < 1) > ((A + B) – 2)))

(1   &&   ((0 < 1)   >   ((A + B) – 2)))

(1   &&   (1   >   (2 – 2)   ))

(1   &&   (1   >   0   ))

(1   &&   1   )

Answer: 1

Precedence: +/-
> <
&&

You should refer to the C++ operator precedence and associative table

Or just use parentheses whenever you're unsure about precedence and associativity

Operator	Description	Associativity
()	Parentheses (function call) (see Note 1)	left-to-right
[]	Brackets (array subscript)	
.	Member selection via object name	
->	Member selection via pointer	
++ --	Postfix increment/decrement (see Note 2)	
++ --	Prefix increment/decrement	right-to-left
+ -	Unary plus/minus	
! ~	Logical negation/bitwise complement	
(*type*)	Cast (change *type*)	
*	Dereference	
&	Address	
sizeof	Determine size in bytes	
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <=	Relational less than/less than or equal to	left-to-right
> >=	Relational greater than/greater than or equal to	
== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
?:	Ternary conditional	right-to-left
=	Assignment	right-to-left
+= -=	Addition/subtraction assignment	
*= /=	Multiplication/division assignment	
%= &=	Modulus/bitwise AND assignment	
^= |=	Bitwise exclusive/inclusive OR assignment	
<<= >>=	Bitwise shift left/right assignment	
,	Comma (separate expressions)	left-to-right