Announcements

» HW #3 has been posted

» Project #1 has been posted
Assignment Operators

=, :=
Assignment Operators

• **Simple assignment**: `lvalue = expression;`
  
  ```
  area = 5.5;
  j = 23 + i;
  ```

  - Lvalue: an object with an identifiable memory location

• **Lvalue**: An assignment expects an lvalue as its left operand, an object stored in memory, e.g., a variable

  ```
  x + y = 5;  \textit{wrong!}
  4 = x; \textit{wrong!}
  (x+1) = 6; \textit{wrong!}
  ```
Assignment Operators

• **Simple assignment**: \( \text{lvalue} = \text{expression} \);

  - area = 5.5;
  - \( j = 23 + i; \)
  - \( \text{lvalue}: \) an object with an identifiable memory location

• **Compound assignment**: uses old value of variable to compute its new value: +=, -=, *=, /=, %=

  - height = height * 2;
  - height *= 2;
## Compound Assignment

The **compound assignment operators** consist of a binary operator and the simple assignment operator.

They perform the operation of the binary operator on both operands and store the result of that operation into the left operand.

<table>
<thead>
<tr>
<th>Compound Assignment Operator</th>
<th>Example</th>
<th>Equivalent expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+=</code></td>
<td><code>a += 3</code></td>
<td><code>a = a + 3</code></td>
</tr>
<tr>
<td><code>-=</code></td>
<td><code>b -= 1</code></td>
<td><code>b = b - 1</code></td>
</tr>
<tr>
<td><code>*=</code></td>
<td><code>c *= d</code></td>
<td><code>c = c * d</code></td>
</tr>
<tr>
<td><code>/=</code></td>
<td><code>e /= f</code></td>
<td><code>e = e / f</code></td>
</tr>
<tr>
<td><code>%=</code></td>
<td><code>g %= 80</code></td>
<td><code>g = g % 80</code></td>
</tr>
</tbody>
</table>
Increment and Decrement Operators

++ , --
Pre- Post- Increment and Decrement

• Increment/Decrement operators ++ and --:
  
  j++;  same as j = j+1;  or j += 1;

  c--; same as c = c – 1; or c -= 1;

• **Postfix version:**
  
  int i = 0; printf("%d", i++);

• **Prefix version:**
  
  int i = 0; printf("%d", ++i);
You can put the `++/--` before or after the operand. If it appears **before** the operand, the operand is incremented/decremented. The incremented value is then used in the expression. If you put the `++/--` **after** the operand, the value of the operand is used in the expression **before** the operand is incremented/decremented.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>++</code></td>
<td>Pre-increment</td>
<td><code>++a</code></td>
</tr>
<tr>
<td><code>--</code></td>
<td>Pre-decrement</td>
<td><code>--a</code></td>
</tr>
<tr>
<td><code>a++</code></td>
<td>Post-increment</td>
<td></td>
</tr>
<tr>
<td><code>a--</code></td>
<td>Post-decrement</td>
<td></td>
</tr>
</tbody>
</table>
Pre versus Post-Increment

What is the difference between \texttt{++a} and \texttt{a++}:
Pre versus Post-Increment (I)

What is the difference between `++a` and `a++`:

1. If it is used as a single statement:
   no difference

   ```
   int a = 10;
   ++a;
   printf("%d", a);
   
   int a = 10;
   a++;
   printf("%d", a);
   ```

   output: 11 11
Pre versus Post-Increment (II)

If it is used (combined) in (with) another expression: difference is significant:

```
int a = 10;
printf("%d", ++a);
printf("%d", a);
```

output:

<table>
<thead>
<tr>
<th></th>
<th>11</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>

```
int a = 10;
printf("%d", a++);
printf("%d", a);
```

output:

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11</td>
<td>11</td>
</tr>
</tbody>
</table>
Pre versus Post-Increment (II)

```
printf(“%d”, ++a);  // a = a + 1;
printf(“%d”, a);    

printf(“%d”, a++);  // a = a + 1;
printf(“%d”, a);    
```

12
Pre versus Post-Increment & Decrement

```
int a = 20;
int b = 10;
c = a++ / b--;  // c = 2.0

int a = 20;
int b = 10;
c = ++a / --b;  // c = 2.3333

int a = 20;
int b = 10;
c = ++a / b--;  // c = 2.1
```

```
c = a / b;
a = a + 1;
b = b - 1;
```

```
c = ++a / b--;  // c = a / b;
```

```
c = ++a / --b;
a = a + 1;
```

```
c = a / b;
b = b - 1;
```
Program to demonstrate Pre- and Post-increment and Pre- and Post-decrement:

```c
#include <stdio.h>

int main()
{
    int a;
    a = 10;
    printf("Post-increment a++: %d \n", a++);
    a = 10;
    printf("Pre-increment ++a: %d \n", ++a);
    a = 10;
    printf("Post-decrement a--: %d \n", a--);
    a = 10;
    printf("Pre-decrement --a: %d \n", --a);
    return 0;
}
```
Pre- Post- Increment and Decrement

```c
#include <stdio.h>

int main()
{
    int a;
    a = 10;
    printf("Post-increment  a++: %d \n", a++);
    a = 10;
    printf("Pre-increment  ++a: %d \n", ++a);
    a = 10;
    printf("Post-decrement  a--: %d \n", a--);
    a = 10;
    printf("Pre-decrement  --a: %d \n", --a);
    return 0;
}
```

```bash
C++ $g++  doubleplus.cc  -o doubleplus.exe
C++ $./doubleplus.exe
Post-increment  a++: 10
Pre-increment  ++a: 11
Post-decrement  a--: 10
Pre-decrement  --a: 9
C++ $
```
You can put the `++/--` before or after the operand. If it appears before the operand, the operand is incremented/decremented. The incremented value is then used in the expression. If you put the `++/--` after the operand, the value of the operand is used in the expression before the operand is incremented/decremented.

<table>
<thead>
<tr>
<th></th>
<th>Post-increment. After the result is obtained, the value of the operand is incremented by 1.</th>
<th>a++</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>Post-decrement. After the result is obtained, the value of the operand is decremented by 1.</td>
<td>a--</td>
</tr>
<tr>
<td>--</td>
<td>Pre-increment. The operand is incremented by 1 and its new value is the result of the expression.</td>
<td>++a</td>
</tr>
<tr>
<td>++</td>
<td>Pre-decrement. The operand is decremented by 1 and its new value is the result of the expression.</td>
<td>--a</td>
</tr>
</tbody>
</table>
What is output of following code?

```c
#include <stdio.h>

int main()
{
    int a;
    a = 10;

    printf("Post-increment a++: %d \n", a++);
    printf("Pre-increment ++a: %d \n", ++a);
    printf("Post-decrement a--: %d \n", a--);
    printf("Pre-decrement --a: %d \n", --a);

    return 0;
}
```
Pre- Post- Increment and Decrement

```c
#include <stdio.h>

int main()
{
    int a;
    a = 10;

    printf("Post-increment a++: %d \n", a++);
    printf("Pre-increment ++a: %d \n", ++a);
    printf("Post-decrement a--: %d \n", a--);
    printf("Pre-decrement --a: %d \n", --a);

    return 0;
}
```

```
C++ $g++ doubleplus.cc -o doubleplus.exe
C++ $./doubleplus.exe
Post-increment a++: 10
Pre-increment ++a: 12
Post-decrement a--: 12
Pre-decrement --a: 10
C++ $
```
Equality and Relational Operators

> , >=, <, <=, ==, !=
Equality Operators

- **Equal to**: `==` (double equals, do NOT confuse with `=`)
- **Not equal to**: `!=` (exclamation mark follow by equal)
- Produce 0 (false) or 1 (true)
- `x == y`
- `b != 0`
The relational operators compare two operands and determine the validity of a relationship.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Result Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;</code></td>
<td>Specifies whether the value of the left operand is less than the value of the right operand. The type of the result is <code>int</code> and has the value 1 if the specified relationship is true, and 0 if false.</td>
<td><code>int</code></td>
<td><code>i &lt; 7</code></td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Specifies whether the value of the left operand is greater than the value of the right operand. The type of the result is <code>int</code> and has the value 1 if the specified relationship is true, and 0 if false.</td>
<td><code>int</code></td>
<td><code>j &gt; 5</code></td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Specifies whether the value of the left operand is less than or equal to the value of the right operand. The type of the result is <code>int</code> and has the values 1 if the specified relationship is true, and 0 if false.</td>
<td><code>int</code></td>
<td><code>k &lt;= 4</code></td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Specifies whether the value of the left operand is greater than or equal to the value of the right operand. The type of the result is <code>int</code> and has the values 1 if the specified relationship is true, and 0 if false.</td>
<td><code>int</code></td>
<td><code>p &gt;= 3</code></td>
</tr>
</tbody>
</table>
Relational Operators

- Produce 0 or 1
- $4 \geq 4$ has value 1
- $51 < 50$ has value 0
- $5 < 70 < 10$ has value 1
Relational Operators

• Produce 0 or 1
• $4 \geq 4$ has value 1
• $51 < 50$ has value 0
• $5 < 70 < 10$:
  
  \[
  \left( (5 < 70) < 10 \right)
  \]
  
  \[
  \left( (1 < 10) \right)
  \]
  
  1
Relational Operators

- Produce 0 or 1
- $4 \geq 4$ has value 1
- $51 < 50$ has value 0
- $5 < 70 < 10$:

  \[
  ((5 < 70) < 10)
  \]

How to check to see if 70 is larger than 5 AND less than 10?
Relational Operators: Example

#include <stdio.h>
int main()
{
    printf("The 4 >= 4 is: %d\n", 4 >= 4);
    printf("The 51 < 50 is: %d\n", 51 < 50);
    printf("The 5 < 70 < 10 is: %d\n", 5 < 70 < 10);
    printf("The 20 < 10 < 4 is: %d\n", 20 < 10 < 4);
}

C++ $g++ relational.cc -o relational.exe
C++ $./relational.exe
The 4 >= 4 is: 1
The 51 < 50 is: 0
The 5 < 70 < 10 is: 1
The 20 < 10 < 4 is: 1
C++ $
## Equality and Relational Operators: Summary

<table>
<thead>
<tr>
<th>Algebraic equality or relational operator</th>
<th>C equality or relational operator</th>
<th>Example of C condition</th>
<th>Meaning of C condition</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>Equality operators</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>==</td>
<td>x == y</td>
<td>x is equal to y</td>
</tr>
<tr>
<td>≠</td>
<td>!=</td>
<td>x != y</td>
<td>x is not equal to y</td>
</tr>
<tr>
<td><em>Relational operators</em></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>x &gt; y</td>
<td>x is greater than y</td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>x &lt; y</td>
<td>x is less than y</td>
</tr>
<tr>
<td>≥</td>
<td>&gt;=</td>
<td>x &gt;= y</td>
<td>x is greater than or equal to y</td>
</tr>
<tr>
<td>≤</td>
<td>&lt;=</td>
<td>x &lt;= y</td>
<td>x is less than or equal to y</td>
</tr>
</tbody>
</table>
Logical Operators

!, &&, ||
Logical Operators

- **Negation:** \(!\) (unary):
  - \(!expr\) has value 1 if \(expr\) has value 0

- **Logical and:** \(\&\&\)
  - \(expr1 \&\& expr2\): 1 if both are non-zero
  - \(x > 1 \&\& x < 10\) (\(x = -1\), result = ?)

- **Logical or:** \(||\)
  - \(expr1 || expr2\): 1 if either is non-zero
  - \(x > 1 || x < 10\) (\(x = -1\), result = ?)

- **Produce 0 or 1**
Logical **NOT** Operator

<table>
<thead>
<tr>
<th>Logical NOT Operator (unary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
</tr>
<tr>
<td>Logical not operator. Produces value 0 if its operand or expression is true (nonzero) and the value 1 if its operand or expression is false (0). The result has an <code>int</code> type. The operand must be an integral, floating, or pointer value.</td>
</tr>
<tr>
<td>! (4 == 2)</td>
</tr>
<tr>
<td>! x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand (or expression)</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>!0</td>
<td>1 ( T )</td>
</tr>
<tr>
<td>!1</td>
<td>0 ( F )</td>
</tr>
</tbody>
</table>
Logical OR Operator

Indicates whether either operand is true. If either of the operands has a nonzero value, the result has the value 1. Otherwise, the result has the value 0. The type of the result is int. Both operands must have a arithmetic or pointer type. The usual arithmetic conversions on each operand are performed.

The logical OR (||) should not be confused with the bitwise OR (|) operator. For example:

1 || 4 evaluates to 1 (or True || True = True)
while 1 | 4 (0001 | 0100 = 0101) evaluates to 5

<table>
<thead>
<tr>
<th>Operand1</th>
<th>Operand2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 ( F )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1 ( T )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1 ( T )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 ( T )</td>
</tr>
</tbody>
</table>
f (a == 5 || b == 5 )
printf ("There is at least one 5\n")
else
printf ("There isn't any 5\n");
Logical **AND** Operator

Indicates whether both operands are true. If both operands have nonzero values, the result has the value 1. Otherwise, the result has the value 0. The type of the result is int. Both operands must have an arithmetic or pointer type. The usual arithmetic conversions on each operand are performed. The logical **AND** (&&) should not be confused with the bitwise **AND** (&) operator. For example:

1 && 4 evaluates to 1 (True && True = True)

while

1 & 4 (0001 & 0100 = 0000) evaluates to 0

<table>
<thead>
<tr>
<th>Operand1</th>
<th>Operand2</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0 ( F )</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0 ( F )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0 ( F )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1 ( T )</td>
</tr>
</tbody>
</table>

Bitwise operator! do not confuse
# Logical Operators: Examples

<table>
<thead>
<tr>
<th>Expressions</th>
<th>Evaluates as</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(3 == 3) &amp;&amp; (4 != 3)</code></td>
<td>True (1) because both operands are true</td>
</tr>
<tr>
<td>`(4 &gt; 2)</td>
<td></td>
</tr>
<tr>
<td><code>(3 == 2) &amp;&amp; (7 == 7)</code></td>
<td>False (0) because one operand is false</td>
</tr>
<tr>
<td><code>!(4 == 3)</code></td>
<td>True (1) because the expression is false</td>
</tr>
<tr>
<td><code>NOT(FALSE) = TRUE</code></td>
<td></td>
</tr>
<tr>
<td><code>NOT(TRUE) = FALSE</code></td>
<td></td>
</tr>
</tbody>
</table>
Exercise

• Write an expression to evaluate whether $x$ is in the range of $[1, 10]$

$$1 \leq x \leq 10$$

if $x = 0$, true or false?

$$(x \geq 1) \land (x \leq 10)$$
Bitwise Operators
Bitwise Operators: Example

• Binary representation:
  int i = 22;          /* 10110 */
  int j = 91;          /* 1011011 */

• Division by 2
  22 = 2*11 = 2 * (2 * 5 + 1)
      = 2 * (2 * (2*2*1 + 1) + 1) = 2^4 + 2^2 + 2^1

• Comparison by powers of 2 (1, 2, 4, 8, 16, 32, 64, 128, 254, 1024)
  22 = 16 + 4 + 2
      = 2^4 + 2^2 + 2^1
  91 = 64 + 16 + 8 + 2 + 1
      = 2^6 + 2^4 + 2^3 + 2^1 + 2^0

• In binary: 1111 = 10000 - 1 = 2^4 - 1
Bitwise Operators

• For bit manipulation:
  – Bitwise AND: &
  – Bitwise inclusive OR: |
  – Bitwise exclusive OR: ^
  – Bitwise One's complement: ~
  – Left shift: <<
  – Right shift: >>