Week 2
Variable Concept and Basic Operators

Assist. Prof. Dr. Caner ÖZCAN
Any accessible thing which takes a memory space is called an **object**.

An expression should indicate a memory space to be called as an object.

- a = b+c;
- d = 100;

In the above expressions, **a**, **b**, **c** and **d** are all an object.
Object

► **Properties of Objects:** name, value, type, scope, lifetime.

► **Name:** Characters that represent an object.

► **Value:** Information stored in an object. It can be changed at any time.

► **Type:** A property that specifies how a compiler behaves to an object on a process.

  – Most of the programming languages includes object types such as **char**, **integer** and **float**.
Assigns a value to an object. It is showed by an equal sign " = " in C.

Usage of assignment operator:

```plaintext
object = expression;
```

Examples:

```plaintext
a = 23;
b = a * 10;
total = total + b;
```
Left Values (lvalue)

► All expressions that specify object are left values.

► An expression is called as left value if it shows a location in the memory.

► For example, in previous example expression, a and b are the left values.

► But, a+b is not a left value. It only represents a number which indicates the sum of a and b.

► For example we can not write, a+b = c
Expressions that do not specify objects. They take place on the right side of assignment operator.

Constants are always right value.

For example, in an expression `a = 100;` `a` indicates a left value and `100` indicates right value.

An expression like `100 = a;` is wrong.

Following expressions have mistakes.

```plaintext
20 = ...;    /* mistake */
c - 4 = ...;  /* mistake */
(y) = ...;    /* mistake */
m * 2 = ...;  /* mistake */
```
Object Type

► All information that points a memory space or not, is called data.
► Both constants and objects are all data.
► The way that compiler interprets an information stored inside an object depends on the type of that object.
► At the same time, an object type gives information about the amount of memory space that is consumed by the object.
Object Type

- Objects are stored at a location inside the memory.
- For example, objects "a" and "b" are put in a free location in the memory.
- Memory space they consume depends on their types and can be different.
- "a" and "b" are only labels that indicate the starting point of a location in the memory.
- An assignment like a = 100 changes the value in the memory location indicated by related object.
- For example, we have two objects assigned with values a= 100 and b = 50
- An expression like a = b + 80 only changes the value of a but b is preserved.
### Object Type

<table>
<thead>
<tr>
<th>a</th>
<th>---</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>---</td>
<td>50</td>
</tr>
<tr>
<td>130</td>
<td>---</td>
<td>50</td>
</tr>
</tbody>
</table>

- **a ve b nesneleri tanımladık**
- **a = 100** değeri
- **b = 50** ataması
- **a = b + 80**
Expression

► An expression is a mathematical formula used for calculation and end with a semicolon ";"
  – (a+b)/4;
  – a*b+c;

► Expressions are formed by Operators

► C operators can be classified as shown below:
  – Assignment Operator (=)
  – Arithmetic Operators (+, -, *, /, %)
  – Arithmetic Assignment Operators (+=, -=, *=, ...)
  – Increment and Decrement Operators (++, --)
  – Relational Operators (<, <=, ==, >=, >)
  – Logical Operators (&&, ||, !)
Arithmetic Operators

► The arithmetic operators are all binary operators.
  – For example the expression 3+7 contains the binary operator + and the operands 3 and 7.

► The asterisk (*) indicates multiplication and the percent sign (%) denotes the remainder operator.

► Integer division yields an integer result.
  – For example the expression 7/4 yields 1.
C provides remainder operator %, which yields the remainder after integer division.

The remainder operator is an integer operator that can only be used with integer operands.

The expression x % y yields the remainder after x is divided by y. Thus 7%4 yields 3.
## Arithmetic Operators

<table>
<thead>
<tr>
<th>Operation</th>
<th>Arithmetic Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>+</td>
</tr>
<tr>
<td>Subtraction</td>
<td>-</td>
</tr>
<tr>
<td>Multiplication</td>
<td>*</td>
</tr>
<tr>
<td>Division</td>
<td>/</td>
</tr>
<tr>
<td>Remainder</td>
<td>%</td>
</tr>
</tbody>
</table>
## Precedence Rules on Arithmetic Operators

<table>
<thead>
<tr>
<th>ORDER</th>
<th>OPERATOR</th>
<th>OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( )</td>
<td>Paranthesis</td>
</tr>
<tr>
<td>2</td>
<td>* / %</td>
<td>Multiplication</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Division</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Remainder</td>
</tr>
<tr>
<td>3</td>
<td>+ -</td>
<td>Addition</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Subtraction</td>
</tr>
</tbody>
</table>
Precedence Rules on Arithmetic Operators

- Expressions within pairs of parentheses are evaluated first.
- Parentheses are said to be highest level of precedence.
- In cases of nested or embedded parentheses such as
  - \((a+b)+c\) (the operators in the innermost pair of parentheses are applied first)
- Parenthesis in the same level are evaluated from left to right.
- Multiplication, division and remainder come after parenthesis.
- Addition and subtraction has the same level of precedence, which is lower than the precedence of multiplication, division and remainder operations.
Precedence Rules on Arithmetic Operators

- Multiplication, division and remainder are said to be on the same level of precedence.
- If an expression contains several multiplication, division and remainder operations, evaluation proceeds from left to right.
- If an expression contains several addition and subtraction operations, evaluation proceeds from left to right.
- Remembering rules of precedence can be complex.
- You would better try to use parenthesis in order to specify precedence of operators in expressions.
  - For example: $\text{result} = (a\times b) + (a/b)$;
Precedence Rules on Arithmetic Operators

- If we want to divide the entire quantity \((a+b+c+d+e)\) by 5.
  \[ m = \frac{a + b + c + d + e}{5}; \]

- Here, parentheses are required to group the additions because division has higher precedence than addition.

- If the parentheses are omitted we obtain \(a+b+c+d+e/5\). And it would first calculate \(e/5\) then additions.

\[ z = p * r \% q + w / x - y; \]

- \(y = a * x * x + b * x + c;\)
  \(a = 2, b = 3, c = 7\) and \(x = 5\)
  \[ y = 2 * 5 * 5 + 3 * 5 + 7 \]
  \[ y = 10 * 5 + 3 * 5 + 7 \]
  \[ y = 50 + 3 * 5 + 7 \]
  \[ y = 50 + 15 + 7 \]
  \[ y = 65 + 7 \]
  \[ y = 72 \]
# Arithmetic Assignment Operators

- Arithmetic assignment operators are:
  - `+=`  
  - `-=`  
  - `*=`  
  - `/=`  
  - `%=`  

---

<table>
<thead>
<tr>
<th>Assignment operator</th>
<th>Sample expression</th>
<th>Explanation</th>
<th>Assigns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+=</code></td>
<td>c += 7</td>
<td>c = c + 7</td>
<td>10 to c</td>
</tr>
<tr>
<td><code>-=</code></td>
<td>d -= 4</td>
<td>d = d - 4</td>
<td>1 to d</td>
</tr>
<tr>
<td><code>*=</code></td>
<td>e *= 5</td>
<td>e = e * 5</td>
<td>20 to e</td>
</tr>
<tr>
<td><code>/=</code></td>
<td>f /= 3</td>
<td>f = f / 3</td>
<td>2 to f</td>
</tr>
<tr>
<td><code>%=</code></td>
<td>g %= 9</td>
<td>g = g % 9</td>
<td>3 to g</td>
</tr>
</tbody>
</table>

*Assume: int c = 3, d = 5, e = 4, f = 6, g = 12*;
Unary Increment and Decrement Operators

► result = ++a; $\rightarrow$ first increment the value of a, then assign it to result (preincrement)

► Same with :

```
a = a+1;
result = a;
```

► result = --a; $\rightarrow$ first decrement the value of a, then assign it to the result (predecrement)

• Same with:

```
a = a-1;
result = a;
```

19
Unary Increment and Decrement Operators

► `result = a++;` → First assign the value of `a` to `result`, then increment the value of `a` (postincrement)

► Same with:

```
result = a;
a = a+1;
```

► `result = a--;` → First assign the value of `a` to `result`, then decrement the value of `a` (postdecrement)

► Same with:

```
result = a;
a = a-1;
```

► It’s important to note here that when incrementing or decrementing a variable in a statement by itself, the preincrement and postincrement forms have the same effect. Same with:
Expressions that compare two values and produce either **True (1)** or **False (0)** are formed by relational operators.

<table>
<thead>
<tr>
<th>Relational Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>X == Y</td>
</tr>
<tr>
<td>!=</td>
<td>X != Y</td>
</tr>
<tr>
<td>&gt;</td>
<td>X &gt; Y</td>
</tr>
<tr>
<td>&lt;</td>
<td>X &lt; Y</td>
</tr>
<tr>
<td>&gt;=</td>
<td>X &gt;= Y</td>
</tr>
<tr>
<td>&lt;=</td>
<td>X &lt;= Y</td>
</tr>
</tbody>
</table>
Relational Operators

- C does not have an explicit boolean type
  - So integers are used instead. The general rules is:
  - “Zero is false, any non-zero value is true”
- Assume that, a = 1, b = 2, and c = 3

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>a &lt; b</td>
<td>True</td>
<td>1</td>
</tr>
<tr>
<td>(a + b) &gt;= c</td>
<td>True</td>
<td>1</td>
</tr>
<tr>
<td>(b + c) &gt; (a + 5)</td>
<td>False</td>
<td>0</td>
</tr>
<tr>
<td>c != 3</td>
<td>False</td>
<td>0</td>
</tr>
<tr>
<td>b == 2</td>
<td>True</td>
<td>1</td>
</tr>
</tbody>
</table>
Relational Operators

► Used to combine relational expressions that are either True (1) or False (0)
► Their result is again "True" or "False"
► If a number is interpreted in logical way, the rule is:
  – 0 → False
  – No zero positive or negative numbers are True.

• For example:
  – -11 → True
  – 0 → False
  – 99 → True
Unary NOT operator converts True to False and False to True.

<table>
<thead>
<tr>
<th>X</th>
<th>!X</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

For example: a = !6 → 0
Relational Operators (&& → AND)

- Returns True if both conditions are True.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>X &amp;&amp;&amp; Y</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>
</tbody>
</table>
Relational Operators (&& → AND)

► First, left side of AND operator is evaluated. If left side of AND operator is false, evaluation stops.

► For example:
  – a = 4 && 0 → a = 0
  – b = 10 && -4 → b = 1
Relational Operators (|| → OR)

- Returns True if either of it's conditions are true.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>X</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>
Relational Operators (|| → OR)

► First, left side of OR operator is evaluated. If left side of OR operator is true, evaluation stops.

► For example:

- `a = 3 || 0 → a = 1`
- `b = 0 || -30 → b = 1`
Relational Operators

- The `&&` operator has a higher precedence than `||`.
- An expression containing `&&` or `||` operators is evaluated only until truth or falsehood is known.
- This performance feature for the evaluation of logical AND and logical OR expressions is called short-circuit evaluation.
### Precedence of Operators

<table>
<thead>
<tr>
<th></th>
<th>HIGH PRECEDENCE</th>
<th>LOW PRECEDENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>(  )</td>
<td>Left to right</td>
<td></td>
</tr>
<tr>
<td>! ++ --</td>
<td>Right to left</td>
<td></td>
</tr>
<tr>
<td>* / %</td>
<td>Left to right</td>
<td></td>
</tr>
<tr>
<td>+ -</td>
<td>Left to right</td>
<td></td>
</tr>
<tr>
<td>&gt; &gt;= &lt; &lt;=</td>
<td>Left to right</td>
<td></td>
</tr>
<tr>
<td>== !=</td>
<td>Left to right</td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Left to right</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>Right to left</td>
<td></td>
</tr>
</tbody>
</table>

Notice that using parenthesis is the best way for not having mistake.
Example Operations in Operators

• Example 1:
  – a = 15;
  – x = a >= 10 && a <= 20;
  – Here, x = 1

• Example 2:
  – a = 20;
  – b = 10;
  – y = a + b >= 20 || a – b <= 10;
  – Here, y = 1

• Example 3:
  • a = 5;
  • b = 0;
  • y = a || b && a && b
  • Here, y = 1
References

• Paul J. Deitel, “C How to Program”, Harvey Deitel.