Operators

Overview
✓ Will introduce various operators supported by C language
✓ Identify supported operations
✓ Present some of terms characterizing operators

Operands and Operators
Mathematical or logical relationships comprise two major entities
✓ Operation be preformed
✓ Entities upon which operations being preformed

Operations typically represented by shorthand symbol called operator
We are familiar with basic operators for
Add +
Subtract -
Multiply * or x
Divide / or \[\div\]

Entities upon which operations performed
Called operands
Most familiar operands are numbers
Sometimes use letters or characters

Characterizing Operators
We use number of different terms to characterize operators
• Arity
• Precedence
• Associativity

Arity
Identifies the number of operands involved in single operation

Typical arity is binary
Operator operates on two operands
\[x+y\]
\[1+2\]
In expressions like
\[1 + 2 + 3 + 4\]
Seem to involve many operands
However expression really involves operations on pairs of numbers

Other common arities
Unary
Operator operates only on single operand

Ternary
Operator operates on three operands

Precedence
When several operators present in expression
Need to know order in which each is evaluated

Operators of higher precedence evaluated before those of lower

Associativity
Goes hand in hand with precedence
With several operators present in expression
If all have same precedence
Need to know to which operator an operand belongs

Two possibilities
Operator on right or one on left

Grouping with right or left operator
Depends upon associativity of operator

C Language and Operators – A First Look
The C language supports
• Three types of operations
  Arithmetic
  Relational
  Logical

• Three types of arity
  Specifies the number of operands
  Unary
  Binary
  Ternary

• Two types of associativity
  Left
  Right
Precedence

Specifies the order in which operators are evaluated

*Example*

\[ a + b \bullet c = ? \]

Evaluated as

\[ a + (b \bullet c) \]

- *evaluated first*
- + *evaluated second*

Can control order of evaluation with parentheses

Evaluation from inside out

**Good Style**

Wise to use parentheses for
- **Clarity**
- Ensuring expression evaluated as desired

Associativity

Most C operators associate (evaluated)

Left to right

Consider expression

\[ 6 + 7 + 8 \leftrightarrow (6 + 7) + 8 \text{ by convention} \]
\[ 6 - 7 - 8 \leftrightarrow (6 - 7) - 8 \text{ by convention} \]

Both cases

7 has operators on
- Left
- Right

Must decide

Which operator gets the operand

Define operator +

Associate to left

Because operand with + on both sides taken by operator on left
Define operator =
    Associate to right
    \(a = b = c \iff a = (b = c)\)

Operand \(b\) with = on both sides taken by operator on right

All operators with same precedence have same associativity

Evaluation
If two operators have same precedence
    Operand grouped according to associativity

Example
\[a + b - c * d / e = ?\]

Evaluated as
\[(a + b) - ((c * d) / e)\]

**C Language and Operators – A Further Look**

Shortcut Operators
C language supports shorthand notation
    For expressing certain operations

Familiar with standard notation
    For binary operators we have
    \[\text{variable1} = \text{variable2} \text{ operation } \text{variable3}\]

In some cases result or \(\text{variable1}\) is same variable as
    One of the operands

When such is case
    Can using shortcut notation to write expression in form

    \[\text{syntax}\]
    \[\text{variable1 operation } = \text{variable2}\]

Example
\[x += y; \quad // \quad x = x + y\]
\[x += 1 \quad // \quad x = x + 1\]
Autoincrement – Autodecrement Operators

Autoincrement and autodecrement operators
Increment or decrement operand by 1

Come in two flavours
Prefix and postfix

<table>
<thead>
<tr>
<th>syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix</td>
</tr>
<tr>
<td>operation variable1</td>
</tr>
<tr>
<td>postfix</td>
</tr>
<tr>
<td>variable1 operation</td>
</tr>
</tbody>
</table>

Operation performed in two steps
Difference between two different forms
- Prefix
  operation variable1
  1. Operation performed
  2. Result returned
- Postfix
  operation variable1
  1. Result returned
  2. Operation performed

Example
int x = 4;

Prefix
Autoincrement
  ++x;  // x = x + 1 → x = 5
  // return 5

Autodecrement
  --x;  // x = x - 1 → x = 3
  // return 3

Postfix
Autoincrement
  x++;  // return 4
  // x = x + 1 → x = 5

- 5 of 13 -
Autodecrement

\[ x-- ; \quad \text{// return 4} \]
\[ \quad \text{// } x = x - 1 \rightarrow x = 3 \]

In many cases use of \textit{prefix} or \textit{postfix} form matters little

In other cases can make a difference

\textit{Caution:}

When using \textit{prefix} or \textit{postfix} forms of \textit{autoincrement} or \textit{autodecrement} operators

Always

✓ Stop to think about results
✓ Make certain returned value is what you are looking for

\textbf{C Arithmetic Operators}

Let’s now put some of these operators to work

Examine group used to perform arithmetic operations

On variables

Operators are the same for both types

• Integer
• Floating-point

Basic C operators are:

+ add
- subtract
/ divide
* multiply

…and they work as you would expect

\textit{Example}

Let's start by declaring some variables

\begin{verbatim}
int a = 20;
int b = 25;
int c;
\end{verbatim}
We next write
\[ c = a \times \frac{2}{b} + 15; \]

Variable \( c \) gets
- \( a \) times 2
- Divided by \( b \)
- Plus 15

Which is really 40 divided by 40
We expect the variable \( c \) to be 1

However, if we include these four lines of code
- In a program then compile and run it
  - We will print the number 16 will appear
Why did that happen?

The answer to this question lays in concept
- Called *operator precedence*
- Or simply *precedence*

Your textbook contains
- C Operator Precedence and Associatively table

Will talk more about operators later

Operators in C evaluated in a specific order
From the Associatively table
- We see division operator appears in the table
  - Closer to the top than the addition operator
Means compiler will
- Perform division
  - Before addition
- Perform multiplication
  - Before division

Let's re-look at the example
In the light of precedence

1. \( a \) is multiplied by 2
   - Because multiplication comes before division or addition
     - This is 20 times 2
     - Which is 40.
2. Next 40 is divided by $b$
   Because division comes before addition
   This is 40 divided by 25
   Which is 1.6

**Warning Here:**
Since integers do not have fractional parts
0.6 is simply dropped leaving 1
Will discuss shortly

3. Finally, 15 is added to 1
   To give 16

If we wish evaluation to proceed in different order
Can over ride precedence
Using parentheses

To force $b \text{ plus } 15$ to be done
Before the division
Enclose those terms in parentheses
We now have
$$c = a \times \frac{2}{(b + 15)};$$

Doing this changes evaluation order
Because the grouping parentheses operator
Comes before any arithmetic operator
Means the inside of the parentheses
Must be evaluated first
We can nest parentheses to any depth
To achieve whatever desired evaluation order

The line of code now evaluated as follows

1. 15 is added to b
   Because the parentheses must be evaluated first
   This is 25 plus 15
   Which is 40.
2. Next $a$ is multiplied by 2
   Because multiplication precedes division
   This is 20 times 2, or 40
3. Finally 40 is divided by 40 to give 1
Now we ask
If precedence determines which operator is applied first
What happens when all of the operators in a line of code
Have the same precedence

*Example*

\[ a + b - c \]

Does the compiler evaluate \( a + b \) or \( b - c \) first?

The answer is associativity
If we look up addition and subtraction
In the C Operator Precedence and Associativity table
We will see the associatively is L-R
Means operator on the left
Will be the first to be evaluated

In the above line
\( a + b \) will be evaluated first
Then \( c \) will be subtracted
This is important because it can
Affect your mathematical calculations.

**Integer Division**

In the arithmetic example above
Was one step in which 40 is divided by 25
Commonly expected answer is 1.6

Since integers do not have fractional parts
0.6 is simply dropped

In C we would read that 40 divided by 25
is 1 because there is only one 25 in 40
With a remainder of 15

Because the two operands are integers
Compiler expresses result as integer
0.6 can't be expressed as integer
If our application needs non-integer numbers, we can use floating-point variables.

When using floats,
Must remember may not be able to display complete result.

*Example*
Let's divide 10 by 3

Result will be 3.33333333333333
Until we get tired of writing 3s
Computer cannot store infinite number of 3s
So some get dropped
Stored number is then rounded
Up or down

This means your floating-point numbers are really approximations.
We introduce rounding error into our calculations.
For this reason
Most mathematical operations in C use
Integers rather than floating-point numbers.

**The Modulus Operator %**
Recall our earlier discussion of integer types.

We’ve seen integer division
Could give
- Whole number part
- Fractional part

Fractional part
Arises because of remainder
Produced during division operation

Such operation important enough
In mathematics
We’ve defined
Special name
Special operator symbol
Operation
Called *modulus* or *mod*

Operator
Called modulus operator
Written as percent sign
% 
Also called *remainder operator*

We’ll learn about uses
In later lessons
Learning concept now

*Example*
Let’s apply the mod operator to the following ints
int number = 15;

```
printf("%d\n", number %3 ); // prints 0 …3 divides 15 5 times with 0 remainder
printf("%d\n", number %5 ); // prints 0 …5 divides 15 3 times with 0 remainder
printf("%d\n", number %10 ); // prints 5 …10 divides 15 one time with 5 remainder
printf("%d\n", number % 2 ); // prints 1 …2 divides 15 7 times with 1 remainder
```

*Example*
As another example
We can use the modulus operator to convert ounces to pounds and ounces

```
int weight = 1610; // weight in ounces
int pounds = 1610/16; // gives 100 pounds
int ounces = 1610 % 16; // gives 10 ounces
```

**Table of Operators**
Table of C operators follows
- Those grouped together have same precedence
- Precedence decreases from top to bottom
<table>
<thead>
<tr>
<th>Operator</th>
<th>Arity</th>
<th>Associativity</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>x[i]</td>
<td>unary</td>
<td>left</td>
<td>Array Subscripting</td>
<td>a[i]</td>
</tr>
<tr>
<td>f(x)</td>
<td>unary</td>
<td>left</td>
<td>Function Call</td>
<td>display(values)</td>
</tr>
<tr>
<td>.</td>
<td>unary</td>
<td>left</td>
<td>Direct Selection</td>
<td>rectangle.width</td>
</tr>
<tr>
<td>-&gt;</td>
<td>unary</td>
<td>left</td>
<td>Indirect Selection</td>
<td>rectangle-&gt;width</td>
</tr>
<tr>
<td>++, --</td>
<td>unary</td>
<td>left</td>
<td>Postfix Increment, Decrement</td>
<td>a++, b--</td>
</tr>
<tr>
<td>+++, --</td>
<td>unary</td>
<td>right</td>
<td>Increment, Decrement</td>
<td>++a, --b</td>
</tr>
<tr>
<td>sizeof (type)</td>
<td>unary</td>
<td>right</td>
<td>size of variable or type in bytes</td>
<td>sizeof a, sizeof (char)</td>
</tr>
<tr>
<td>~</td>
<td>unary</td>
<td>right</td>
<td>Bitwise NOT</td>
<td>~a</td>
</tr>
<tr>
<td>!</td>
<td>unary</td>
<td>right</td>
<td>Logical NOT</td>
<td>! a</td>
</tr>
<tr>
<td>-, +</td>
<td>unary</td>
<td>right</td>
<td>Arithmetic Negative, Plus</td>
<td>-a, +a</td>
</tr>
<tr>
<td>&amp;</td>
<td>unary</td>
<td>right</td>
<td>Address of</td>
<td>&amp;a</td>
</tr>
<tr>
<td>*</td>
<td>unary</td>
<td>right</td>
<td>Indirection</td>
<td>*a</td>
</tr>
<tr>
<td>*, /, %</td>
<td>binary</td>
<td>left</td>
<td>Arithmetic</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>multiply</td>
<td>a * b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>divide</td>
<td>a / b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>modulus</td>
<td>a % b</td>
</tr>
<tr>
<td>+, -</td>
<td>binary</td>
<td>left</td>
<td>Arithmetic</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>addition</td>
<td>a + b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>subtraction</td>
<td>a - b</td>
</tr>
<tr>
<td>&lt;&lt;, &gt;&gt;</td>
<td>binary</td>
<td>left</td>
<td>Left and Right Shift</td>
<td>a &lt;&lt; 3, 1 &gt;&gt; b</td>
</tr>
<tr>
<td>&lt;, &gt;, &lt;=, &gt;=</td>
<td>binary</td>
<td>left</td>
<td>Test for Inequality</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>less than</td>
<td>a &lt; b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>greater than</td>
<td>a &gt; b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>less than or equal</td>
<td>a &lt;= b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>greater than or equal</td>
<td>a &gt;= b</td>
</tr>
<tr>
<td>==, !=</td>
<td>binary</td>
<td>left</td>
<td>Test for Inequality</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>equal</td>
<td>a == b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>not equal</td>
<td>a != b</td>
</tr>
<tr>
<td>&amp;</td>
<td>binary</td>
<td>left</td>
<td>Bitwise AND</td>
<td>a &amp; b</td>
</tr>
<tr>
<td>^</td>
<td>binary</td>
<td>left</td>
<td>Bitwise XOR</td>
<td>a ^ b</td>
</tr>
<tr>
<td></td>
<td></td>
<td>binary</td>
<td>left</td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>binary</td>
<td>left</td>
<td>Logical AND</td>
<td>a &amp;&amp; b</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>binary</td>
<td>left</td>
</tr>
<tr>
<td>?:</td>
<td>ternary</td>
<td>right</td>
<td>Conditional</td>
<td>a ? b : c</td>
</tr>
<tr>
<td>=</td>
<td>binary</td>
<td>right</td>
<td>Assignment</td>
<td>a = b</td>
</tr>
<tr>
<td>+=, -=, *=, /=, %=, &lt;&lt;=, &gt;&gt;=, &amp;=, ^=,</td>
<td></td>
<td>=</td>
<td>binary</td>
<td>right</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Add to</td>
<td>a += 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Subtract from</td>
<td>a -= 3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Divide by</td>
<td>a /= 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Multiply by</td>
<td>a *= 2</td>
</tr>
<tr>
<td>Operator</td>
<td>Expression</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------------</td>
<td>------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assign Remainder</td>
<td>a %= 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Left</td>
<td>b &lt;&lt;= 3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shift Right</td>
<td>b &gt;&gt;= 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitwise AND</td>
<td>b &amp;= c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitwise OR</td>
<td>b ^= d</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bitwise XOR</td>
<td>b</td>
<td>= e</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Operator</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>left</td>
</tr>
<tr>
<td></td>
<td>binary</td>
</tr>
<tr>
<td></td>
<td>Sequential Evaluation</td>
</tr>
<tr>
<td></td>
<td>for (i=0, j=10; i&lt;5; i++, j--)</td>
</tr>
</tbody>
</table>

**Summary**

Following this lesson we

- ✓ Should understand basic terms for characterizing operators
- ✓ Understand arity
- ✓ Understand associativity and precedence
- ✓ Know type of operators supported by C language
- ✓ How to use parentheses to force order of expression evaluation
- ✓ Examined basic arithmetic operators