Expressions
Expressions

- Variables and constants linked with operators
  - Arithmetic expressions
    - Uses arithmetic operators
    - Can evaluate to any value
  - Logical expressions
    - Uses relational and logical operators
    - Evaluates to 1 or 0 (true or false) only
  - Assignment expression
    - Uses assignment operators
    - Evaluates to value depending on assignment
Arithmetic Operators

- **Binary operators**
  - Addition: +
  - Subtraction: –
  - Division: /
  - Multiplication: *
  - Modulus: %

- **Unary operators**
  - Plus: +
  - Minus: –

**Examples**

- \(2 \times 3 + 5 - 10/3\)
- \(-1 + 3 \times 25/5 - 7\)
- distance / time
- \(3.14 \times \text{radius} \times \text{radius}\)
- \(a \times x \times x + b \times x + c\)
- dividend / divisor
- \(37 \% 10\)
Contd.

- Suppose $x$ and $y$ are two integer variables, whose values are 13 and 5 respectively.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x + y$</td>
<td>18</td>
</tr>
<tr>
<td>$x - y$</td>
<td>8</td>
</tr>
<tr>
<td>$x \times y$</td>
<td>65</td>
</tr>
<tr>
<td>$x / y$</td>
<td>2</td>
</tr>
<tr>
<td>$x % y$</td>
<td>3</td>
</tr>
</tbody>
</table>
All operators except % can be used with operands of all of the data types int, float, double, char (yes! char also! We will see what it means later)

% can be used only with integer operands
Operator Precedence

- In decreasing order of priority
  1. Parentheses :: ( )
  2. Unary minus :: –5
  3. Multiplication, Division, and Modulus
  4. Addition and Subtraction

- For operators of the same priority, evaluation is from left to right as they appear

- Parenthesis may be used to change the precedence of operator evaluation
Examples:
Arithmetic expressions

\[ a + b \times c - d \div e \rightarrow a + (b \times c) - (d \div e) \]
\[ a \times -b + d \% e - f \rightarrow a \times (-b) + (d \% e) - f \]
\[ a - b + c + d \rightarrow (((a - b) + c) + d) \]
\[ x \times y \times z \rightarrow ((x \times y) \times z) \]
\[ a + b + c \times d \times e \rightarrow (a + b) + ((c \times d) \times e) \]
Type of Value of an Arithmetic Expression

- If all operands of an operator are integer (int variables or integer constants), the value is always integer
  - Example: 9/5 will be 1, not 1.8
  - Example:
    ```c
    int a=9, b=5;
    printf("%d", a/b)
    ```
    will print 1 and not 1.8
If at least one operand is real, the value is real

- **Caution**: Since floating-point values are rounded to the number of significant digits permissible, the final value is an approximation of the final result.

- Example: \( \frac{1}{3.0} \times 3.0 \) may have the value 0.99999 and not 1.0

- So checking if \( \frac{1}{3.0} \times 3.0 \) is equal to 1.0 may return **false!!**
The type of the final value of the expression can be found by applying these rules again and again as the expression is evaluated following operator precedence.
We have a problem!!

```c
int a=10, b=4, c;
float x;
c = a / b;
x = a / b;
The value of c will be 2
The value of x will be 2.0
But we want 2.5 to be stored in x
```
Solution: Typecasting

- Changing the type of a variable during its use
- General form
  \[(\text{type\_name}) \text{ variable\_name}\]
- Example

\[x = ((\text{float}) a)/ b;\]

Now \(x\) will store 2.5 (type of \(a\) is considered to be float for this operation only, now it is a mixed-mode expression, so real values are generated)
Not everything can be typecast to anything

- float/double should not be typecast to int (as an int cannot store everything a float/double can store)
- int should not be typecast to char (same reason)

General rule: make sure the final type can store any value of the initial type
Example: Finding Average of 2 Integers

Wrong program

```c
int a, b;
float avg;
scanf("%d%d", &a, &b);
avg = (a + b)/2;
printf("%f\n", avg);
```

average-1.c

Correct programs

```c
int a, b;
float avg;
scanf("%d%d", &a, &b);
avg = ((float) (a + b))/2;
printf("%f\n", avg);
```

```c
int a, b;
float avg;
scanf("%d%d", &a, &b);
avg = (a + b)/2.0;
printf("%f\n", avg);
```

average-2.c
Assignment Expression

- Uses the assignment operator (=)
- General syntax:
  
  \[
  \text{variable\_name} = \text{expression}
  \]
- Left of = is called l-value, must be a modifiable variable
- Right of = is called r-value, can be any expression
- Examples:
  
  \[
  \begin{align*}
  \text{velocity} &= 20 \\
  b &= 15; \quad \text{temp} = 12.5 \\
  A &= A + 10 \\
  v &= u + f \times t \\
  s &= u \times t + 0.5 \times f \times t \times t
  \end{align*}
  \]
An assignment expression evaluates to a value same as any other expression.

Value of an assignment expression is the value assigned to the l-value.

Example: value of

- $a = 3$ is 3
- $b = 2*4 − 6$ is 2
- $n = 2*u + 3*v − w$ is whatever the arithmetic expression $2*u + 3*v − w$ evaluates to given the current values stored in variables $u$, $v$, $w$.
Contd.

- Several variables can be assigned the same value using multiple assignment operators
  
  \[ a = b = c = 5; \]
  
  \[ \text{flag1} = \text{flag2} = 'y'; \]
  
  \[ \text{speed} = \text{flow} = 0.0; \]

- Easy to understand if you remember that
  
  - the assignment expression has a value
  - Multiple assignment operators are right-to-left associative
Example

- Consider \( a = b = c = 5 \)
  - Three assignment operators
  - Rightmost assignment expression is \( c = 5 \), evaluates to value 5
  - Now you have \( a = b = 5 \)
  - Rightmost assignment expression is \( b = 5 \), evaluates to value 5
  - Now you have \( a = 5 \)
  - Evaluates to value 5
  - So all three variables store 5, the final value the assignment expression evaluates to is 5
Types of l-value and r-value

- Usually should be the same
- If not, the type of the r-value will be internally converted to the type of the l-value, and then assigned to it

Example:

```java
double a;
   a = 2*3;
```

Type of r-value is int and the value is 6
Type of l-value is `double`, so stores 6.0
This can cause strange problems

```c
int a;
    a = 2*3.2;
```

- Type of r-value is float/double and the value is 6.4
- Type of l-value is int, so internally converted to 6
- So `a` stores 6, not the correct result
- But an int cannot store fractional part anyway
- So just badly written program
- Be careful about the types on both sides
More Assignment Operators

- +=, -=, *=, /=, %= 
- Operators for special type of assignments 
- a += b is the same as a = a + b 
- Same for -=, *=, /=, and %= 
- Exact same rules apply for multiple assignment operators
Suppose x and y are two integer variables, whose values are 5 and 10 respectively.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
</table>
| x += y    | Stores 15 in x  
Evaluates to 15 |
| x -= y    | Stores -5 in x  
Evaluates to -5 |
| x *= y    | Stores 50 in x  
Evaluates to 50 |
| x /= y    | Stores 0 in x  
Evaluates to 0 |
Logical Expressions

- Uses relational and logical operators in addition
- Informally, specifies a condition which can be true or false
- Evaluates to value 0 or 1
  - 0 implies the condition is false
  - 1 implies the condition is true
Logical Expressions

(count <= 100)

((math+phys+chem)/3 >= 60)

((sex == 'M') && (age >= 21))

((marks >= 80) && (marks < 90))

((balance > 5000) || (no_of_trans > 25))

(! (grade == 'A')))
Relational Operators

- Used to compare two quantities.

- `<` is less than
- `>` is greater than
- `<=` is less than or equal to
- `>=` is greater than or equal to
- `==` is equal to
- `!=` is not equal to
Examples

10 > 20 is false, so value is 0
25 < 35.5 is true, so value is 1
12 > (7 + 5) is false, so value is 0
32 != 21 is true, so value is 1

- When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared

\[ a + b > c - d \] is the same as \( (a+b) > (c+d) \)
Logical Operators

- Logical AND (&&)
  - Evaluates to 1 if both the operands are non-zero

- Logical OR (||)
  - Result is true if at least one of the operands is non-zero

| X    | Y      | X && Y | X || Y |
|------|--------|--------|--------|
| 0    | 0      | 0      | 0      |
| 0    | non-0  | 0      | non-0  |
| non-0| 0      | 0      | non-0  |
| non-0| non-0  | non-0  | non-0  |
Contd

- Unary negation operator (!)
  - Single operand
  - Value is 0 if operand is non-zero
  - Value is 1 if operand is 0
Example

- $(4 > 3) \&\& (100 \neq 200)$
  - $4 > 3$ is true, so value 1
  - $100 \neq 200$ is true so value 1
  - Both operands 1 for $\&\&$, so final value 1

- $(10) \&\& (10 + 20 \neq 200)$
  - $10$ is non-0, so value $10$ is 0
  - $10 + 20 \neq 200$ is true so value 1
  - Both operands NOT 1 for $\&\&$, so final value 0

- $(10) \| (10 + 20 \neq 200)$
  - Same as above, but at least one value non-0, so final value 1
a = 3 && b = 4

- No parenthesis, so need to look at precedence and associativity
- = has higher precedence than &&
- b=4 is an assignment expression, evaluates to 4
- a = 3 is an assignment expression, evaluates to 3
- Both operands of && are non-0, so final value of the logical expression is 1

Note that changing to b = 0 would have made the final value 0
Example: Use of Logical Expressions

```c
void main () {  
    int i, j;
    scanf("%d%d", &i, &j);
    printf("%d AND %d = %d, %d OR %d = %d\n", i, j, i && j, i, j, i || j);  
}  
```

If 3 and 0 are entered from keyboard, output will be

```
3 AND 0 = 0, 3 OR 0 = 1
```
A Special Operator: AddressOf (&)

- Remember that each variable is stored at a location with an unique address
- Putting & before a variable name gives the address of the variable (where it is stored, not the value)
- Can be put before any variable (with no blank in between)

```c
int a = 10;
printf("Value of a is %d, and address of a is %d\n", a, &a);
```
More on Arithmetic Expressions
More Operators: Increment (++) and Decrement (--) 

- Both of these are unary operators; they operate on a single operand.
- The increment operator causes its operand to be increased by 1.
  - Example: a++, ++count
- The decrement operator causes its operand to be decreased by 1.
  - Example: i--, --distance
Pre-increment versus post-increment

- Operator written before the operand (;++i, --i))
  - Called pre-increment operator (also sometimes called prefix ++ and prefix --)
  - Operand will be altered in value **before** it is utilized in the program

- Operator written after the operand (i++, i--)
  - Called post-increment operator (also sometimes called postfix ++ and postfix --)
  - Operand will be altered in value **after** it is utilized in the program
Examples

**Initial values :: a = 10; b = 20;**

\[
x = 50 + ++a; \quad a = 11, x = 61
\]

\[
x = 50 + a++; \quad x = 60, a = 11
\]

\[
x = a++ + --b; \quad b = 19, x = 29, a = 11
\]

\[
x = a++ - +++a; \quad ??
\]

Called **side effects** (while calculating some values, something else gets changed)
Precedence among different operators (there are many other operators in C, some of which we will see later)

<table>
<thead>
<tr>
<th>Operator Class</th>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unary</td>
<td>postfix++, --</td>
<td>Left to Right</td>
</tr>
<tr>
<td>Unary</td>
<td>prefix ++, -- ! &amp;</td>
<td>Right to Left</td>
</tr>
<tr>
<td>Binary</td>
<td>* / %</td>
<td>Left to Right</td>
</tr>
<tr>
<td>Binary</td>
<td>+ −</td>
<td>Left to Right</td>
</tr>
<tr>
<td>Binary</td>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>Left to Right</td>
</tr>
<tr>
<td>Binary</td>
<td>== !=</td>
<td>Left to Right</td>
</tr>
<tr>
<td>Binary</td>
<td>&amp;&amp;</td>
<td>Left to Right</td>
</tr>
<tr>
<td>Binary</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignment</td>
<td>= += − = *= /= &amp;=</td>
<td>Right to Left</td>
</tr>
</tbody>
</table>
Statements in a C program

- Parts of C program that tell the computer what to do
- Different types
  - Declaration statements
    - Declares variables etc.
  - Assignment statement
    - Assignment expression, followed by a ;
  - Control statements
    - For branching and looping, like if-else, for, while, do-while (to be seen later)
  - Input/Output
    - Read/print, like printf/scanf
Example

```c
int a, b, larger;
scanf("%d %d", &a, &b);
larger = b;
if (a > b)
larger = a;
printf("Larger number is %d\n", larger);
```
Compound statements

- A sequence of statements enclosed within { and }
- Each statement can be an assignment statement, control statement, input/output statement, or another compound statement
- We will also call it block of statements sometimes informally
Example

```c
int n;
scanf("%d", &n);
while(1) {
    if (n > 0) break;
    scanf("%d", &n);
}
```

Compound statement