ECE15: Introduction to Computer Programming Using the C Language

Lecture Unit 3: Expressions and Operators
Outline of this Lecture

❖ Basic Types of Expressions in C
  ➤ Arithmetic Expressions
  ➤ Type Conversions in Arithmetic Expressions
  ➤ Assignment Expressions

❖ Example: Conversion from Fahrenheit to Celsius

❖ Logical Expressions
  ➤ Logical Value of Expressions
  ➤ Equality and Relational Operators
  ➤ Logical Operators &&, ||, and !

❖ sizeof Operator, Typecasting, typedef Declaration, and enum Specifier
What Is an Expression?

- An **expression** is a constant, a variable, or a meaningful (valid) **combination** of constants, variables, and/or other expressions, that are combined using **operators**:

  - `77`  
  - `x`  
  - `x = 3 + y`  
  - `(x > 0) && (x < 70)`  
  - `x++`  
  - `found ? x : y`

- `’77’`  
- `for`  
- `x = 3 +/- y`  
- `(x > 0)~(x < 70)`  
- `x**`  
- `found ! x : y`

- Every expression has a **type** and a **value**.

- Expressions that involve only constants and operators are called **constant expressions**. They are evaluated at compilation time, and can be used in a `#define` directive:

  ```c
  #define GOLDEN_RATIO (1+2.236)/2
  ```

- Note that operators in expressions can be **unary** (`x++`), **binary** (`x+y`), or **ternary** (`found ? x : y`).
Simple Arithmetic Expressions

The five basic arithmetic operations are addition +, subtraction −, multiplication *, division /, and modulo %. Here are some examples:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Value</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1+2*3</td>
<td>7</td>
<td>Multiplication always precedes addition</td>
</tr>
<tr>
<td>1−2+3</td>
<td>2</td>
<td>Additions/subtractions proceed from left to right</td>
</tr>
<tr>
<td>1−(2+3)</td>
<td>−4</td>
<td>Parentheses can be used to alter the operation order</td>
</tr>
<tr>
<td>1.0/2.0*3.0</td>
<td>1.5</td>
<td>Multiplications/divisions proceed from left to right</td>
</tr>
<tr>
<td>1/2*3</td>
<td>0</td>
<td>Integer division: 1/2 evaluates to 0</td>
</tr>
<tr>
<td>12%5</td>
<td>2</td>
<td>This means 12 modulo 5 (remainder upon division by 5)</td>
</tr>
<tr>
<td>5%12</td>
<td>5</td>
<td>The remainder of 5 when divided by 12 is 5</td>
</tr>
<tr>
<td>12.0%5.0</td>
<td>Error</td>
<td>The modulo operator % applies only to integers</td>
</tr>
</tbody>
</table>

Be careful with division / and modulo % operators!
Floating-point division works as expected. Integer division always truncates down to the nearest integer. Thus if \( a \) and \( b \) of type \texttt{int} are both positive or both negative, then:

\[
\frac{a}{b} = \lfloor \frac{a}{b} \rfloor -- \text{the greatest integer less than or equal to } \frac{a}{b}
\]

What if one of \( a \) and \( b \) is negative? The absolute value of \( \frac{a}{b} \) is the same as before (as if both were positive), but with negative sign.

The modulo operation \( a \% b \) produces the integer remainder in the division of \( a \) by \( b \). What is the sign of \( a \% b \)? It is always true that:

\[
(a/b) \times b + a \% b = a
\]

Example:

\begin{array}{c|cc}
\hline
\text{b} & a & \\
\hline
4 & 13 & -13 \\
4 & 3 & -3 \\
-4 & -3 & 3 \\
\hline
\end{array}
Arithmetic operations on operands of the same type usually produce a result of this same type.

Arithmetic operations that involve variables and/or constants of different types will convert, or promote, all the operands to the highest type of any one of the operands, according to:

char ➜ short ➜ int ➜ long ➜ float ➜ double ➜ long double

Examples:

2 + 7.5 ➜ 2.0 + 7.5

char c;
int i;
c + i; ➜ (int)c + i;
Declarations

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>char c;</td>
<td></td>
</tr>
<tr>
<td>int i;</td>
<td></td>
</tr>
<tr>
<td>short s;</td>
<td></td>
</tr>
<tr>
<td>long l;</td>
<td></td>
</tr>
<tr>
<td>float f;</td>
<td></td>
</tr>
<tr>
<td>double d;</td>
<td></td>
</tr>
</tbody>
</table>

The result of an arithmetic operation has type **int**, or higher according to the operands.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>c + c</td>
<td>int</td>
</tr>
<tr>
<td>c + s</td>
<td>int</td>
</tr>
<tr>
<td>c – s/i</td>
<td>int</td>
</tr>
<tr>
<td>l*2.0 – i</td>
<td>double</td>
</tr>
<tr>
<td>l + 3</td>
<td>long</td>
</tr>
<tr>
<td>c + 5.0</td>
<td>double</td>
</tr>
<tr>
<td>i/l * 2.0</td>
<td>double</td>
</tr>
<tr>
<td>f*7 – i</td>
<td>float</td>
</tr>
<tr>
<td>d + c</td>
<td>double</td>
</tr>
<tr>
<td>f*d – l</td>
<td>double</td>
</tr>
</tbody>
</table>

Both operands promoted to **int**

i/l performs **integer division**
Assignment Expressions

❖ An assignment operation **evaluates the expression** on the right-hand side and **puts the result** into the variable (or, more generally, into the place in memory) specified on the left-hand side:

\[
x = y; \quad x = x+1; \quad \text{discriminant} = b*b - 4*a*c; \quad *(p+2) = 42;
\]

**Syntax Note:** In C, `=` is always the assignment operator, **never equality**. Thus `if (x = 0) do ... is wrong!`

❖ Like any other expression (such as `x + y`), an assignment expression (such as `x = x + 1`) has a **value**. The overall value of an assignment expression is the value that has been assigned to the left-hand side. For example, we can do this:

\[
\text{if} \ (\ (\text{number_read} = \text{scanf("%d",&value)}) < 1 \ ) \ \text{return} \ 1;
\]

❖ This makes it possible to concatenate several assignments into a single expression, which is **evaluated from right-to-left**:

\[
a = b = c = 0 \quad \Rightarrow \quad a = b = (c = 0) \quad \Rightarrow \quad a = b = 0 \quad \Rightarrow \quad a = (b = 0) \quad \Rightarrow \quad a = 0
\]
More on Assignment Operators

❖ What happens if the type of the value to be assigned is different from the type of the variable to which it should be assigned?

A type conversion is performed at run-time. If a larger type is converted to a smaller type (e.g. double \( \rightarrow \) int), information may be lost.

```c
double z; int x; int y = 4;

x = y * 3.6;  // double 14.4
z = y / 5;    // int 0
x = y / 5.0;  // double 0.8
```

❖ The C language provides a convenient shorthand for assignment operations like this: \( x = x + y \). This can be also written as \( x += y \). The shorthand works for all arithmetic operations (\( +, -, \ast, /, \% = \)) and all bitwise operations (\( & =, | =, ^ =, \gg = \)). If \( \Box \) is any of \( +, -, \ast, /, \%, \& , | , ^, \gg \) then

\[
x \ \Box = y \quad \quad \Rightarrow \quad \quad x = x \ \Box y
\]
Increasing or decreasing the value of an integer variable by 1 is so common that C provides an even shorter shorthand for this:

- `x++` or `++x` increases `x` after `x++` is evaluated.
- `x--` or `--x` decreases `x` before `x++` is evaluated.

Example:

```
x = 5;
y = ++x;
z = x++;
```

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
### Precedence and Associativity Table

<table>
<thead>
<tr>
<th>Operation</th>
<th>Notation</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parentheses</td>
<td>(  )</td>
<td>left to right</td>
</tr>
<tr>
<td>Unary Operations</td>
<td>++ -- + -</td>
<td>right to left</td>
</tr>
<tr>
<td>Multiplication and Division</td>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>Addition and Subtraction</td>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>Assignment Operations</td>
<td>= += -= *= /= %=</td>
<td>right to left</td>
</tr>
</tbody>
</table>

**Example:** Suppose that the value the variable `z` is 9. Then...

What is the value of `z` now?
Outline of this Lecture

❖ Basic Types of Expressions in C
  ‣ Arithmetic Expressions
  ‣ Type Conversions in Arithmetic Expressions
  ‣ Assignment Expressions

❖ Example: Conversion from Fahrenheit to Celsius

❖ Logical Expressions
  ‣ Logical Value of Expressions
  ‣ Equality and Relational Operators
  ‣ Logical Operators &&, ||, and !

❖ sizeof Operator, Typecasting, typedef Declaration, and enumSpecifier
Fahrenheit to Celsius Conversion

$T_F, T_C$ — temperature
$F_F, F_C$ — freezing point of water
$B_F, B_C$ — boiling point of water

$slope = \frac{B_C - F_C}{B_F - F_F} = \frac{T_C - F_C}{T_F - F_F}$

$$T_C = \frac{(B_C - F_C)(T_F - F_F)}{B_F - F_F} + F_C$$
#include <stdio.h>

#define CEL_FRZ          0
#define CEL_BOIL       100
#define FAHR_FRZ        32
#define FAHR_BOIL      212

int main()
{
    int cel_temp, fahr_temp;

    printf("Fahrenheit temperature: ");
    scanf("%d", &fahr_temp);
    cel_temp = (fahr_temp - FAHR_FRZ)/(FAHR_BOIL - FAHR_FRZ) * (CEL_BOIL - CEL_FRZ) + CEL_FRZ;
    printf("%d deg @F = %d deg @C\n", fahr_temp, cel_temp);
    return 0;
}
```c
#include <stdio.h>

#define CEL_FRZ          0
#define CEL_BOIL       100
#define FAHR_FRZ        32
#define FAHR_BOIL      212

int main()
{
    double cel_temp, fahr_temp;

    printf("Fahrenheit temperature: ");
    scanf("%lf", &fahr_temp);
    cel_temp = (fahr_temp - FAHR_FRZ)/(FAHR_BOIL - FAHR_FRZ)
                *(CEL_BOIL - CEL_FRZ) + CEL_FRZ;
    printf("%.2f deg @F = %.2f deg @C\n", fahr_temp, cel_temp);
    return 0;
}

What are the types of these expressions?
```
```c
#include <stdio.h>

#define CEL_FRZ 0.0
#define CEL_BOIL 100.0
#define FAHR_FRZ 32.0
#define FAHR_BOIL 212.0

int main()
{
    double cel_temp, fahr_temp;

    printf("Fahrenheit temperature: ");
    scanf("%lf", &fahr_temp);
    cel_temp = (fahr_temp - FAHR_FRZ)/(FAHR_BOIL - FAHR_FRZ) * (CEL_BOIL - CEL_FRZ) + CEL_FRZ;
    printf("%.2f deg @F = %.2f deg @C\n", fahr_temp, cel_temp);
    return 0;
}
```
Basic Types of Expressions in C
- Arithmetic Expressions
- Type Conversions in Arithmetic Expressions
- Assignment Expressions

Example: Conversion from Fahrenheit to Celsius

Logical Expressions
- Logical Value of Expressions
- Equality and Relational Operators
- Logical Operators &&, ||, and !

sizeof Operator, Typecasting, typedef Declaration, and enum Specifier
Logical Values of Expressions

- In addition to its **numerical value**, every expression in C also has a **logical value**. The logical value of an expression is defined as follows:

<table>
<thead>
<tr>
<th>Numerical Value</th>
<th>Logical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zero</td>
<td>False</td>
</tr>
<tr>
<td>Nonzero</td>
<td>True</td>
</tr>
</tbody>
</table>

- Logical values of expressions are needed in statements like these:

```c
if (expression) {...} else {...}  
if ((expression1) && (expression2)) {...}
```

**Example:**

```c
int a = 3;  
int b = -3;  
double c = 1.0;
```

Zero values of **any type** (such as 0, 0.0, or ‘\0’) are interpreted as False:

```
Expression | Logical Value |
-----------|---------------|
   a        | True          |
  a + b     | False         |
  a += b    | False         |
  a = 2*c   | True          |
  a - 3*c   | False         |
```
The C language provides two operators for **testing equality** (or inequality) and four operators for **testing relations** such as “greater” or “less.” These operators are:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Numerical Value</th>
<th>Logical Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a == b</code></td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td><code>b == a</code></td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td><code>a = b</code></td>
<td>5</td>
<td>True</td>
</tr>
<tr>
<td><code>b = a</code></td>
<td>0</td>
<td>False</td>
</tr>
<tr>
<td><code>a != b</code></td>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td><code>b &lt;= 8</code></td>
<td>1</td>
<td>True</td>
</tr>
<tr>
<td><code>0 &lt; b &lt; 3</code></td>
<td>1</td>
<td>True</td>
</tr>
</tbody>
</table>

Each such operator returns a **numerical value**: 1 if the condition is true and 0 if the condition is false. The corresponding **logical values** are True and False, as expected.

**Example:**

```c
int a = 0, b = 5;
```

**Associativity is left-to-right:**
The C language provides the three basic logical operators: **NOT**, **AND**, and **OR**. Using these operators it is possible to construct complex logical expressions, such as:

\[(0 < b) \land (b < 3)\]

\[(p \land q \land r) \lor (p \land \neg q) \lor \neg p\]

\[(p \lor q \lor r) \land (p \lor \neg q) \land \neg p\]

Any statement in logic (propositional formula) can be written in terms of **AND**, **OR**, and **NOT**.

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>OR</td>
<td>|</td>
</tr>
<tr>
<td>NOT</td>
<td>!</td>
</tr>
</tbody>
</table>

The three operators **AND**, **OR**, and **NOT** are defined by their truth tables, namely:

<table>
<thead>
<tr>
<th>exp1</th>
<th>exp2</th>
<th>exp1 &amp;&amp; exp2</th>
<th>exp1 | exp2</th>
<th>!exp1</th>
</tr>
</thead>
<tbody>
<tr>
<td>nonzero</td>
<td>nonzero</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>nonzero</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
More on Logical Operators

❖ The unary ! takes precedence over &&, and && takes precedence over ||. The associativity of all logical operators is left-to-right. Thus:

\[
\begin{align*}
  a \ || \ b \ & \ && \ c \ || \ !d \\
  & \rightarrow \ a \ || (b \ && \ c) \ || (\ !d) \ \rightarrow \ (a \ || (b \ && \ c)) \ || (\ !d)
\end{align*}
\]

❖ Relational and equality operators take precedence over the logical OR and AND, but not over the unary !.

❖ If an expression involves logical operators, its **evaluation stops** as soon as its logical value can be determined:

\[
\text{is\_match} = (\text{age} > 21) \ && \ (\text{eyes} == \text{BLUE} \ || \ \text{hair} == \text{BLOND});
\]

If age is not over 21, nothing else is evaluated

If age is over 21 and eyes are blue, hair is not checked

**Example:**

\[
\begin{align*}
  \text{if } (a/b > 1) \ a++; \\
  /* \text{What happens if } b \text{ is zero? } */
  \text{if } ((b != 0) \ && (a/b > 1)) \ a++; \\
  /* \text{This will never crash! } */
\end{align*}
\]
The Logical Data Type bool

The header file `<stdbool.h>`, introduced with the **C99** standard, defines a special data type `bool`. Variables of type `bool` can take only two values: `True` and `False`, which are synonymous to 1 and 0.

```c
#include <stdio.h>
#include <stdbool.h>

int main()
{
    bool drunk = True;
    bool has_driving_license = False;
    ...
    if (!(has_driving_license || drunk))
        printf("Cannot drive!\n");
    ...
}
```

Note that a variable of type `bool` still occupies at least one full byte (at least 8 bits rather than a single bit) in memory.
Basic Types of Expressions in C
- Arithmetic Expressions
- Type Conversions in Arithmetic Expressions
- Assignment Expressions

Example: Conversion from Fahrenheit to Celsius

Logical Expressions
- Logical Value of Expressions
- Equality and Relational Operators
- Logical Operators &&, ||, and !

sizeof Operator, Typecasting, typedef Declaration, and enumSpecifier
The **sizeof Operator**

- The `sizeof(object)` operator returns the **number of bytes** in memory that are necessary to store the `object`, where `object` can be one of the following:
  - A previously declared **variable** or an **expression**;
  - The **name of a type**, such as `char`, `int`, `float`, `bool`, or a type previously defined using a **typedef** declaration.

**Examples:**

```c
char c1, c2;
int i;
declare d;

sizeof(c1) == sizeof(c2) == sizeof(char) == 1
sizeof(i) == sizeof(c1*i) == sizeof(int)  // usually == 4
sizeof(d) == sizeof(d+i) == sizeof(double) // usually == 8
```

```c
sizeof(char) <= sizeof(short) <= sizeof(int) <= sizeof(long)
sizeof(float) <= sizeof(double) <= sizeof(long double)
```
The C language performs automatic type conversions in many situations (discussed earlier). Occasionally, the programmer might wish to effect a type conversion where it would not have been performed automatically. This can be done with the explicit casting operator (type), such as (int), (double), (bool).

- For example, if expr is an expression of type int that has value 1, (double)expr is an expression of type double with value 1.0.
- Casting operator (type) is unary, so it has the highest precedence.

Examples:

```c
int i = 1, j = 2;
double x, y = 3.14;
x = i / j;
x = (double)i / j;
x = i / (double)j;
x = (double)(i / j);
x = (int) y;
```

<table>
<thead>
<tr>
<th>Type Conversion</th>
<th>Value of x</th>
</tr>
</thead>
<tbody>
<tr>
<td>i / j</td>
<td>0.0</td>
</tr>
<tr>
<td>(double)i / j</td>
<td>0.5</td>
</tr>
<tr>
<td>i / (double)j</td>
<td>0.5</td>
</tr>
<tr>
<td>(double)(i / j)</td>
<td>0.0</td>
</tr>
<tr>
<td>(int) y</td>
<td>3.0</td>
</tr>
</tbody>
</table>
## Precedence and Associativity Table

<table>
<thead>
<tr>
<th>Operation</th>
<th>Notation</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parentheses</td>
<td>(  )</td>
<td>left to right</td>
</tr>
<tr>
<td>Unary Operations</td>
<td>++  --  - +</td>
<td>right to left</td>
</tr>
<tr>
<td></td>
<td>! (type) sizeof</td>
<td></td>
</tr>
<tr>
<td>Multiplication and Division</td>
<td>*  /  %</td>
<td>left to right</td>
</tr>
<tr>
<td>Addition and Subtraction</td>
<td>+  -</td>
<td>left to right</td>
</tr>
<tr>
<td>Relational Operators</td>
<td>&lt;  &lt;=  &gt;  &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>Equality Operators</td>
<td>==  !=</td>
<td>left to right</td>
</tr>
<tr>
<td>Logical AND</td>
<td>&amp;&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td>Logical OR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assignment Operations</td>
<td>=  +=  -=  *=  /=  %=</td>
<td>right to left</td>
</tr>
</tbody>
</table>

Decreasing precedence
The `typedef` Declaration

The `typedef` declaration simply creates a new name, of your choice, for an existing type. For example:

```c
typedef double dollars;
```

makes `dollars` an alias for the keyword `double`. From that point on, we can declare a variable of type `double` like this:

```c
dollars price = 500.45;
```

### Why should one use the `typedef` declaration?

- It makes the program easier to read. Assigning meaningful names to types indicates the purpose of the variables in the program.

```c
double l,w,h;  // ← ────→ dimensions l,w,h;
```

- It makes the program easier to write: we can give a short meaningful name to a complicated type, for use throughout the program.

```c
char *(*x)[])(),(*(*y)[])();  // ← ────→ SORTptr x,y;
```
The enum Declaration

❖ There are two different ways to use the enum specifier in a declaration. The simpler way is like this:

```
enum {mon,tue,wed,thu,fri,sat,sun};
```

This creates seven constants mon,tue,wed,thu,fri,sat,sun of type int whose values are 0,1,2,3,4,5,6. It is completely equivalent to seven #define directives:

```
#define mon 0  
#define tue 1  
...  
#define sun 6
```

❖ The constant values do not have to be in a sequence starting with 0. We can initialize all or some of the constants to arbitrary integers:

```
enum {BELL='\a', BACKSPACE='\b', TAB='\t', NEWLINE='\n'};
```

```
enum {Jan=1,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec, 
      LastYear=2016, ThisYear, NextYear};
```

This is still completely equivalent to a bunch of #define directives.
Another Kind of `enum` Declaration

❖ The other way to use the `enum` specifier in a declaration is like this:

```
enum day {mon,tue,wed,thu,fri,sat,sun};
```

This also creates a new data type, called the `enumeration type`. We can now declare variables of type `enum day`, for example as follows:

```
enum day lecture_day, discussion_day;
lecture_day = thu; discussion_day = wed;
if (lecture_day > discussion_day) ...
```

Variables of type `enum day` can take only the values from the enumeration list: `mon,tue,wed,thu,fri,sat,sun`.

More Examples:

```
enum boolean {False,True};
```

```
enum escapes {BELL = '\a', BACKSPACE = '\b',
                  TAB = '\t', NEWLINE = '\n'};
```