Intro to C

Tutorial 4: Arithmetic and Logical expressions
Last week…

• Types – char, int, long, float, double
• Formatted input and output
• Casting between types
• Arithmetic expressions
• Logical expressions
• Computation shortcuts
• Order of operations
Arithmetic Expressions
What is an operator?

• An operator is an “action” in C that receives one, two, or three parameters and returns a value.
• Values that an operator gets are called operands.

- Each operand can be a constant, a variable, a function return value or any complex expression:
  - $y = \text{scanf}("\%d", \&x)<1$;
  - $z += ((x+y)*3) != (x-y)$;
What is an expression?

• An expression is a combination of one or more operators.

\[(x + 10) \ast (x / 100)\]

\[y = x*8 + 1\]

• Each operator’s computation is stored in a temporary place in memory and is destroyed when it is no longer needed.
Types of operators

- **Unary** operators get one value.
  - Example: casting (type), size of a type `sizeof(type)`

- **Binary** operators get two values.
  - Examples: `/`, `*`, `=`, `%`, `-`, `+`

- **Ternary** operators get 3 types
  - In C there is only one ternary operator “?:”
Operators and side effects

• There are operators that apart from computing the value, also change their operands.
• This is called the “side effect” of the operator.

• Operators without side effects:
  – Arithmetic: %, *, /, -, +
  – Logic: >, <, <=, >=, ==, ||, &&
  – casting: (type)
  – Finding the size of a type: sizeof()
  – Ternary: ?:

• Operators with side effects (that change an operand):
  – Assignment: =, +=, -=, /=, *=, %=,
  – Decrement, increment: --, ++
Side effect of an assignment

• Replace the content of the left variable with the value of the right operand.
• The result of the calculation is the same as the value stored in the variable.

```c
int x = 0, y;
y = (x = 2) + 1;
```

Result of `x = 2` is 2. Side effect: `x` received 2.

Result of `y = 2 + 1` is 3. Side effect: `y` is equal to 3.
Abbreviated Assignment

• We often write expressions like this:
  \[
  x = x + z; \\
  y = y * 2;
  \]

• We can also use an abbreviated assignment:
  \[
  x += z; \\
  y *= 2;
  \]

• The result of this assignment is the same as using the regular assignment.
• The side effect is that the variable is updated.
The increment operator “++” can only be used on variables. It is an *unary* operator.

The operator increases the operand’s value by 1.

For example:

```c
int x=3;

x++;  // Now x is 4

++x;  // Now x is 5
```
• Increment operator can be on the left or right of the operand.
• \texttt{x++} or \texttt{++x}
• What is the difference?
• Like the regular addition operator, the ++ operator also has a result value.
• The value depends on the placement of the operator.

\begin{align*}
\text{int } x, y&=3; \\
y &= ++x; & \text{int } x, y&=3; \\
y &= x++; \\
\end{align*}

– In both cases, the value is incremented to 4.

\begin{itemize}
\item The value of \( y \) will be equal to the value of \( x \) before adding 1. (that is to say, 3)
\item The value of \( y \) will be equal to the value of \( x \) after adding 1. (that is to say, 4)
\end{itemize}
• This operator can also be used in more complex cases

int x = 3;
int y, z;
y = 4 + x++;
z = ++x % 2;

First use the value of x, and afterward increment x by 1. y = 4 + 3...

First increment the value of x. Then execute the expression. z = 5%2.
• The -- operator works exactly as the ++ operator, except that it decreases the value of the variable instead of increasing it.

• It can also appear before or after the variable.

```c
int x=3, y, z;
y=x--;  // y will be 2, x will be 2
z=--x;  // z will be 2, x will be 1
```

• What will the value of x be? y? z?

```
y=3
z=1
x=1
```
Modulo Operator

• The modulo “%” operator yields the remainder of a division
• If we compute the division of two integers, we have a **divisor** and **remainder**:

```c
int result;
int remainder;
result = 32/5;
remainder = 32%5;
```

Modulo “%” is only defined for integers!
Modulo of negative numbers

• What happens when the operands are negative?

<table>
<thead>
<tr>
<th>Expression</th>
<th>Divisor</th>
<th>Dividend</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 % 5</td>
<td>32</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>32 % -5</td>
<td>32</td>
<td>-5</td>
<td>2</td>
</tr>
<tr>
<td>-32 % 5</td>
<td>-32</td>
<td>5</td>
<td>-2</td>
</tr>
<tr>
<td>-32 % -5</td>
<td>-32</td>
<td>-5</td>
<td>-2</td>
</tr>
</tbody>
</table>

The sign of the result will be the same as the sign of the divisor.
How can we determine if a number is even or odd?

```c
even = number % 2;
printf("%d", even);
```

How can we determine the ones digit?

```c
ones = number % 10;
printf("%d", ones);
```

What about the tens digit?

```c
small_number = number/10;
tens = small_number % 10;
printf("%d", tens);
```
• What will the program print?

```c
int num = 14;
int a0, a1, a2, a3;
a0 = num % 2;
num /= 2;
a1 = num % 2;
num /= 2;
a2 = num % 2;
num /= 2;
a3 = num % 2;
printf("%d%d%d%d",a3,a2,a1,a0);
```
What will the program print?

```c
int num = 14;
int a0, a1, a2, a3;
int b = 2;
a0 = num % b;
num /= b;
a1 = num % b;
num /= b;
a2 = num % b;
num /= b;
a3 = num % b;
printf("%d%d%d%d", a3, a2, a1, a0);
```
Logical Expressions
What are logical expressions?

• While running a program, we’ll often want the program to make a decision according to different values.

• For example

```c
if (x > 54)
    printf("success");
else
    printf("failure")
```
What are logical expressions?

```c
if (x > 54)
    printf(“success”);
else
    printf(“failure”)
```

• Decisions like these can be based on logical expressions, such as:
  – Comparison between two numbers (e.g. x is equal to exactly 100).
  – Relative comparison between to numbers (e.g. x is greater than 100).
  – Combination of several expressions (e.g. homework is greater than 54 and less than 60)
Representing “truth” values

• For every logical expression there is a “truth value”
• This can be equal to true or false (correct or incorrect)
• For example, the expression “x>54” has the value of true if x bigger than 54, and a value of false otherwise.
• Some programming languages have a special type for this (boolean).
• C is much simpler -
  – 0 represents false
  – A number other than 0 (for example, 1) represents true.
Comparing with operators

• If the comparison succeeds, then the result is 1 (true)
• Otherwise, the expression has a value of 0 (false)
• What will the following print?

```
int x = 79;
int passing = (x > 54);
printf("Passing status: %d\n", passing);
```
You can compare all sorts of things…

- So far, we looked at comparisons between variables and constants
- We can actually compare between all types of expressions
- All of the following are valid logical expressions:
  - \( x + 5 > 3 \times y \)
  - 35 > 42
  - \( x < x + 1 \)
  - Etc.

Pay attention to the order of operations here!
Operators for Comparison

- We saw the “>” and “<” operators that represent greater than and less than.
- Greater than or equal to: \( x \geq y \) is \( x \geq y \)
- Less than or equal to: \( x \leq y \) is \( x \leq y \)
- Exactly equal to: \( x == y \).
- Not equal to: \( x != y \), which is \( x \neq y \).
Careful! The difference between assignment and comparison

- **Pay** attention: In a **comparison** we use **two equal signs**. Using a **single** equal sign is an **assignment**, not a **comparison**!

  \[ x = 3; \quad \text{← Set the value of } x \text{ to } 3 \]

  \[ x == 3 \quad \text{← Logical expression, equals 1” if } x \text{ is exactly 3, 0 otherwise.} \]

- It’s easy to make mistakes with this, and it can cause some annoying errors!
Combination of several comparisons

• Let’s say we want to make sure that the variable *temp* is between 0 and 100
  – (If for example *temp* represents the temperature of water, we can determine if it’s still a liquid.)

• Will the following program work?

```c
float temp;
int is_liquid;
printf("Enter temperature: ");
scanf("%f", &temp);
is_liquid = (0 < temp < 100);
printf("Is liquid (1=yes, 0=no): %d\n", is_liquid);
```
And the result...

- Excellent.

- What happened here?
• Remember – a comparison results in:
  – 1 if it is “successful”
  – 0 if it “fails”

• Therefore the expression “0 < temp < 100” is resolved as follows:

\[ 0 < 249.12 < 100 \]

First, \( 0 < 249.12 \) is evaluated, and returns 1.

Then, \( 1 < 100 \) is evaluated, which is also true and returns 1!
• If this example failed, what is the right way to make several comparisons?
• The answer: boolean algebra
• Boolean operators are used to compute a result of boolean expressions.
• In the previous example we actually wanted to compute "temp>0 and temp<100"
• The “and” operator is a boolean operator that connects the two logical expressions
• In C, this operator is written as &&
The correct program

```c
float temp;
int is_liquid;
printf("Enter temperature: ");
scanf("%f", &temp);

is_liquid = (0 < temp) && (temp < 100);
printf("Is liquid (1=yes, 0=no): %d\n", is_liquid);
```
Boolean Operators

• The “and” operator \( x && y \) is true only if \( x \) and also \( y \) are true (equal to something other than 0).
• The “or” operator \( x || y \) is true if at least one argument is true (equal to something other than 0).

This is called a Truth table

| \( x \) | \( y \) | \( x && y \) | \( x || y \) |
|--------|--------|-------------|-------------|
| 0      | 0      | 0           | 0           |
| 0      | Not 0  | 0           | 1           |
| Not 0  | 0      | 0           | 1           |
| Not 0  | Not 0  | 1           | 1           |
Not (!) Operator

• The last boolean operator is ! (“not”)
• This operator inverts the value of the logical expression.
  – (Like a minus sign in math, it turns a positive into a negative, and vice versa).
• If the operator’s argument is a 0, it turns it into a 1.
• If the operator’s argument is something different than 0, it turns it into a 0.

• For example:
  ```c
  int test = 85;
  int homework = 65;
  int passing = !(test < 55 || homework < 80);
  ```
Example

• **Example:** What do the following expressions do?

```c
int a, b, c, d, num_of_positive;

a=b=c=d=0;
scanf("%d%d%d%d", &a, &b, &c, &d);

num_of_positive = (a>0) + (b>0) + (c>0) + (d>0);
```
Exercise

• Write a program that receives 3 characters and returns how many of them were capital letters.

```c
char letter1, letter2, letter3;
int upcase = 0;
scanf("%c%c%c", &letter1, &letter2, &letter3);
upcase += ('A' <= letter1) && (letter1 <= 'Z');
upcase += ('A' <= letter1) && (letter1 <= 'Z');
upcase += ('A' <= letter1) && (letter1 <= 'Z');
printf("There are %d capital letters.\n", upcase);
```
Riddle: Double negation

• What will the following program print?

```c
int a, b, c;
a = 7;
b = !a;
c = !!a;
printf("%d, %d, %d\n", a, b, c);
```
The operator ?: 

- Syntax:
  \[ \text{expr1} ? \text{expr2} : \text{expr3} \]

- If \( \text{expr1} \) is TRUE then the value of the whole expression is \( \text{expr2} \). Otherwise the value is \( \text{expr3} \).

- Example one - calculate maximum:
  \[ \text{max} = (x > y) ? x : y; \]

- Example two - Capitalize a letter:
  ```
  char ch;
  printf(“Enter a letter\n”);
  scanf(“%c”, &ch);
  ch = (ch>='a' && ch<='z') ? (ch-'a'+'A') : ch;
  ```
Order of operations among operators
Order of operations

• If an expression contains several operators, there is a prioritized order for their execution
• For example, we know that multiplication and division happen before subtraction.
# Order of Operations

<table>
<thead>
<tr>
<th>Operators</th>
<th>Order of execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Left to right</td>
</tr>
<tr>
<td>[]</td>
<td>Left to right</td>
</tr>
<tr>
<td>n++</td>
<td>Right to left</td>
</tr>
<tr>
<td>n--</td>
<td>Right to left</td>
</tr>
<tr>
<td>++ n</td>
<td>Right to left</td>
</tr>
<tr>
<td>-- n</td>
<td>Right to left</td>
</tr>
<tr>
<td>!</td>
<td>Right to left</td>
</tr>
<tr>
<td>sizeof (type)</td>
<td>Right to left</td>
</tr>
<tr>
<td>+x</td>
<td>Right to left</td>
</tr>
<tr>
<td>-x</td>
<td>Right to left</td>
</tr>
<tr>
<td>*</td>
<td>Right to left</td>
</tr>
<tr>
<td>/</td>
<td>Right to left</td>
</tr>
<tr>
<td>%</td>
<td>Right to left</td>
</tr>
<tr>
<td>x+y</td>
<td>Left to right</td>
</tr>
<tr>
<td>x-y</td>
<td>Left to right</td>
</tr>
<tr>
<td>&lt;</td>
<td>Left to right</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Left to right</td>
</tr>
<tr>
<td>&gt;</td>
<td>Left to right</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Left to right</td>
</tr>
<tr>
<td>==</td>
<td>Left to right</td>
</tr>
<tr>
<td>!=</td>
<td>Left to right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>Right to left</td>
</tr>
<tr>
<td>=</td>
<td>Right to left</td>
</tr>
<tr>
<td>+=</td>
<td>Right to left</td>
</tr>
<tr>
<td>-=</td>
<td>Right to left</td>
</tr>
<tr>
<td>*=</td>
<td>Right to left</td>
</tr>
<tr>
<td>/=</td>
<td>Right to left</td>
</tr>
<tr>
<td>%=</td>
<td>Right to left</td>
</tr>
<tr>
<td>-=</td>
<td>Right to left</td>
</tr>
</tbody>
</table>

Why do + and – appear twice?

---

First!  
Last!
What will be the value of each variable after running the following code?

```c
int a, b, c, d;
 d = a = sizeof(char) + 1;
b = a > d - 1;
c = 5 / a * 2 - b + ++d;
```

a=2  
b=-1  
c=8  
d=3