DATA TYPES AND EXPRESSIONS

CS10003 PROGRAMMING AND DATA STRUCTURES
Data Types in C

**int**: integer quantity

   Typically occupies 4 bytes (32 bits) in memory.

**char**: single character

   Typically occupies 1 byte (8 bits) in memory.

**float**: floating-point number (a number with a decimal point)

   Typically occupies 4 bytes (32 bits) in memory.

**double**: double-precision floating-point number

Some of the basic data types can be augmented by using certain data type qualifiers:

   - short
   - long
   - signed
   - unsigned

Typical examples:

   - short int
   - long int
   - unsigned int
We have studied integer, floating-point, and single character constants in the introduction.
Single Character and String Constants

SINGLE CHARACTER CONSTANTS
Contains a single character enclosed within a pair of single quote marks.

• Examples :: ‘2’, ‘+’, ‘Z’

Some special backslash characters

‘\n’ new line
‘\t’ horizontal tab
‘\’ single quote
‘\”’ double quote
‘\’ backslash
‘\0’ null

STRING CONSTANTS
Sequence of characters enclosed in double quotes.

• The characters may be letters, numbers, special characters and blank spaces.

Examples:
“nice”, “Good Morning”, “3+6”, “3”, “C”

Differences from character constants:

• ‘C’ and “C” are not equivalent.
• ‘C’ has an equivalent integer value while “C” does not.
In C terminology, in an expression

- `speed` refers to the contents of the memory location.
- `&speed` refers to the address of the memory location.

Examples:

```c
printf("%f %f %f", speed, time, distance); /* We need only the values of the vars to print them */
scanf("%f %f", &speed, &time); /* We need the address of the vars to store the values read */
```
Assignment Statement

Used to assign values to variables, using the assignment operator (=).

General syntax:

```
variable_name = expression;
```

Left of = is called l-value, must be a modifiable variable

Right of = is called r-value, can be any expression

Examples:

```
velocity = 20;
b = 15;  temp = 12.5;
A = A + 10;
v = u + f * t;
s = u * t + 0.5 * f * t * t;
```

A value can be assigned to a variable at the time the variable is declared.

```
int speed = 30;
char flag = ‘y’;
```

Several variables can be assigned the same value using multiple assignment operators.

```
a = b = c = 5;
flag1 = flag2 = ‘y’;
speed = flow = 0.0;
```
Expression evaluation

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

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:
  
  ```
  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

  ```
  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 \textit{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 \[ -=, *=, /=, \text{and } %= \]

Exact same rules apply for multiple assignment operators

Suppose \( x \) and \( y \) are two integer variables, whose values are 5 and 10 respectively.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x += y )</td>
<td>Stores 15 in ( x )</td>
</tr>
<tr>
<td></td>
<td>Evaluates to 15</td>
</tr>
<tr>
<td>( x -= y )</td>
<td>Stores -5 in ( x )</td>
</tr>
<tr>
<td></td>
<td>Evaluates to -5</td>
</tr>
<tr>
<td>( x *= y )</td>
<td>Stores 50 in ( x )</td>
</tr>
<tr>
<td></td>
<td>Evaluates to 50</td>
</tr>
<tr>
<td>( x /= y )</td>
<td>Stores 0 in ( x )</td>
</tr>
<tr>
<td></td>
<td>Evaluates to 0</td>
</tr>
</tbody>
</table>
Operators in Expressions

- Arithmetic Operators
- Relational Operators
- Logical Operators
Arithmetic Operators

Addition :: +
Subtraction :: –
Division :: /
Multiplication :: *
Modulus :: %

Examples:

distance = rate * time;
netIncome = income - tax;
speed = distance / time;
area = PI * radius * radius;
y = a * x * x + b*x + c;
quotient = dividend / divisor;
remainder = dividend % divisor;

EXAMPLE: Suppose x and y are two integer variables, whose values are 13 and 5 respectively.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</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 * 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>
# Operator Precedence

In decreasing order of priority:

1. Parentheses :: ( )
2. Unary minus :: –
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:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Simplified Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a + b * c – d / e</code></td>
<td><code>a + (b * c) – (d / e)</code></td>
</tr>
<tr>
<td><code>a * – b + d % e – f</code></td>
<td><code>a * (– b) + (d % e) – f</code></td>
</tr>
<tr>
<td><code>a – b + c + d</code></td>
<td><code>(((a – b) + c) + d)</code></td>
</tr>
<tr>
<td><code>x * y * z</code></td>
<td><code>((x * y) * z)</code></td>
</tr>
<tr>
<td><code>a + b + c * d * e</code></td>
<td><code>(a + b) + ((c * d) * e)</code></td>
</tr>
</tbody>
</table>
**Integer, Real, and Mixed-mode Arithmetic**

**Integer Arithmetic**
- When the operands in an arithmetic expression are integers, the expression is called **integer expression**, and the operation is called **integer arithmetic**.
- Integer arithmetic always yields integer values.
  
  For example:
  
  \[
  25 / 10 \Rightarrow 2
  \]

**Real Arithmetic**
- Arithmetic operations involving only real or floating-point operands.
- Since floating-point values are rounded to the number of significant digits permissible, the final value is an approximation of the final result.
  
  \[
  1.0 / 3.0 \times 3.0 \text{ will have the value } 0.99999 \text{ and not } 1.0
  \]
- The modulus operator cannot be used with real operands.

**Mixed-Mode Arithmetic**
- When one of the operands is integer and the other is real, the expression is called a **mixed-mode** arithmetic expression.
- If either operand is of the real type, then only real arithmetic is performed, and the result is a real number.
  
  \[
  25 / 10 \Rightarrow 2
  \]
  
  \[
  25 / 10.0 \Rightarrow 2.5
  \]

Some more issues will be considered later.
Similar code – different results !!

```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
  \[(\text{type}_\text{name}) \text{variable}_\text{name}\]
- 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)

```c
int a=10, b=4, c;
float x;
c = a / b;
x = a / b;
```
Restrictions on typecasting

- 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 !! Why?

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

Correct programs

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

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

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

- When arithmetic expressions are used on either side of a relational operator, the arithmetic expressions will be evaluated first and then the results compared
  
  \[ a + b > c - d \quad \text{is the same as} \quad (a + b) > (c - d) \]

- Note: The value corresponding to true can be any non-zero value, not necessarily 1
  
  - Will print 1 in most cases, but should not assume it will
Logical Operators

There are two logical operators in C (also called logical connectives).

- `&&` ➔ Logical AND
- `||` ➔ Logical OR

What they do?

- They act upon operands that are themselves logical expressions.
- The individual logical expressions get combined into more complex conditions that are true or false.

| X   | Y    | X && Y | X || Y |
|-----|------|--------|--------|
| FALSE | FALSE | FALSE  | FALSE  |
| FALSE | TRUE  | FALSE  | TRUE   |
| TRUE  | FALSE | FALSE  | TRUE   |
| TRUE  | TRUE  | TRUE   | TRUE   |
Unary Negation

Unary negation operator (!)

- Single operand
- Value is 0 if operand is non-zero
- Value is 1 if operand is 0
Examples of 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'))

Suppose we wish to express that a should not have the value of 2 or 3. Does the following expression capture this requirement?

(( a != 2) || ( a != 3))
A more non-trivial example:

\[ a = 3 \&\& (b = 4) \]

- \( b = 4 \) is an assignment expression, evaluates to 4
- \&\& has higher precedence than =
- \( 3 \&\& (b = 4) \) evaluates to true as both operands of \&\& are non-0, so final value of the logical expression is true
- \( a = 3 \&\& (b = 4) \) is an assignment expression, evaluates to 1 (true)

Note that changing to \( b = 0 \) would have made the final value 0
Example: AND and OR

```
#include <stdio.h>

int 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);
    return 0;
}
```

Output

```
3 0
3 AND 0 = 0, 3 OR 0 = 1
```
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.
- Operator will be altered in value *before* it is utilized for its intended purpose in the program.

Operator written after the operand (i++, i--)

- Called post-increment operator.
- Operator will be altered in value *after* it is utilized for its intended purpose in the program.

EXAMPLES:

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

x = 50 + ++a;  a = 11, x = 61
x = 50 + a++;  x = 60, a = 11
x = a++ + --b;  b = 19, x = 29, a = 11
x = a++ − ++a; ??

Called *side effects*:: while calculating some values, something else get 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 ++, --</td>
<td>Right to Left</td>
</tr>
<tr>
<td></td>
<td>−, !, &amp;</td>
<td></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>= += − =</td>
<td>Right to Left</td>
</tr>
<tr>
<td></td>
<td>*= /= %=</td>
<td></td>
</tr>
</tbody>
</table>
Doing More Complex Mathematical Operations

- C provides some mathematical functions to use
  - perform common mathematical calculations
  - Must include a special header file
    ```
    #include <math.h>
    ```

- Example
  ```
  printf("%f", sqrt(900.0));
  ```
  - Calls function `sqrt`, which returns the square root of its argument

- Return values of math functions are of type `double`
- Arguments may be constants, variables, or expressions
- Similar to functions you have seen in school maths
Math Library Functions

- double acos(double x) – Compute arc cosine of x.
- double asin(double x) – Compute arc sine of x.
- double atan(double x) – Compute arc tangent of x.
- double atan2(double y, double x) – Compute arc tangent of y/x.
- double cos(double x) – Compute cosine of angle in radians.
- double cosh(double x) – Compute the hyperbolic cosine of x.
- double sin(double x) – Compute sine of angle in radians.
- double sinh(double x) – Compute the hyperbolic sine of x.
- double tan(double x) – Compute tangent of angle in radians.
- double tanh(double x) – Compute the hyperbolic tangent of x.
Math Library Functions

double ceil(double x) – Get smallest integral value that exceeds x.
double floor(double x) – Get largest integral value less than x.
double exp(double x) – Compute exponential of x.
double fabs(double x) – Compute absolute value of x.
double log(double x) – Compute log to the base e of x.
double log10(double x) – Compute log to the base 10 of x.
double pow(double x, double y) – Compute x raised to the power y.
double sqrt(double x) – Compute the square root of x.
Computing distance between two points

```c
#include <stdio.h>
#include <math.h>
int main()
{
    int x1, y1, x2, y2;
    double dist;
    printf("Enter coordinates of first point: ");
    scanf("%d%d", &x1, &y1);
    printf("Enter coordinates of second point: ");
    scanf("%d%d", &x2, &y2);
    dist = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
    printf("Distance = \%lf\n", dist);
    return 0;
}
```

Output

Enter coordinates of first point: 3  4
Enter coordinates of second point: 2  7
Distance = 3.162278
Practice Problems

1. Read in three integers and print their average

2. Read in four integers a, b, c, d. Compute and print the value of the expression
   \[ \frac{a+b}{c+d} \times 10 - b + 20 \times \frac{d}{c} \]
   • Explain to yourself the value printed based on precedence of operators taught
   • Repeat by putting parenthesis around different parts (you choose) and first do by hand what should be printed, and then run the program to verify if you got it right
   • Repeat similar thing for the expression \( a \&\& b || c \&\& d > a || c <= b \)

3. Read in the coordinates (real numbers) of three points in 2-d plane, and print the area of the triangle formed by them

4. Read in the principal amount \( P \), interest rate \( I \), and number of years \( N \), and print the compound interest (compounded annually) earned by \( P \) after \( N \) years