Variables and Expressions

CS 1044
Simple data is represented in C++ by **primitive types** that are **built-in** to the language (don’t have to include anything)

- **We’ll focus on four of them:**
  - `int` Positive and negative integers, like 34 and –193
  - `double` Numbers with a fractional part, like 3.14159 (“floating point”)
  - `char` A single text character, like ‘A’ or ‘@’ or ‘5’
  - `bool` A value that is either **true** or **false**
In computer science, we usually call a block of text a **string**.

In other words, a string is zero or more **chars**.

A string with length zero is called the "**empty string**" \"\".

C++ has a **string** type that lets us work with text.

**string** is not built-in – it lives in the **std** namespace – so remember your **using namespace std**;
Literal Values

- A **literal value** is a constant written directly into the program source code.
- The format of the literal determines its type.

<table>
<thead>
<tr>
<th>Literal</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>int</td>
</tr>
<tr>
<td>4</td>
<td>double</td>
</tr>
<tr>
<td>'4' (single quotes)</td>
<td>char</td>
</tr>
<tr>
<td>&quot;4&quot; (double quotes)</td>
<td>string</td>
</tr>
</tbody>
</table>
Variables

- Variables are like “boxes” that hold values that you want to use throughout your program
  - A **name**, used to identify a variable versus any others
  - A **type**, describing the kind of data it can contain
  - A **value**, representing the contents of the variable
- The name and type of a variable are **fixed**, but the value can be changed (with some exceptions)
Declaring Variables

- You **must** declare variables before you can use them.
- You can optionally give a variable an initial value when you declare it.

```plaintext
int count;
double angle = 79.392;
char dollar = '$';
bool done = true;
string name = "Tony";
```
Declaring Variables

- Can only declare a variable with a particular name **once** in a certain **scope**
- To keep things simple and avoid defining scope right now, let’s say “once in a certain **function**” instead

```c++
int main()
{
    int x = 10;
    cout << x;
    int x = 20; // compiler error
    ...
}
```
Where Can I Declare Variables?

- Inside a function
  - Called **local variables**
  - Can only be used inside the function they are declared in
- Outside of a function
  - Called **global variables**
  - Can be used by any function

Don’t use global variables in this class, unless I say otherwise. It’s considered bad style.
Initializing Variables

- You **should always** assign a value to a variable before you try to use it in another expression.
- If you do not, its value is just whatever happened to be in the computer’s memory beforehand.

```cpp
int x;
int y = 4 * x + 5;
// ???
```
What’s in a Name?

- Names of variables (and other things) are called identifiers
  - Must begin with a letter (upper or lower) or an underscore
  - Remaining characters can be letters, numbers, or underscore
  - Name must not be one of C++’s reserved words
- Name should be clear, concise, meaningful
Reserved Words

- 73 words that have special meaning in C++

<table>
<thead>
<tr>
<th>and</th>
<th>and_eq</th>
<th>asm</th>
<th>auto</th>
<th>bitand</th>
</tr>
</thead>
<tbody>
<tr>
<td>bitor</td>
<td>bool</td>
<td>break</td>
<td>case</td>
<td>catch</td>
</tr>
<tr>
<td>char</td>
<td>class</td>
<td>const</td>
<td>const_cast</td>
<td>continue</td>
</tr>
<tr>
<td>default</td>
<td>delete</td>
<td>do</td>
<td>double</td>
<td>dynamic_cast</td>
</tr>
<tr>
<td>else</td>
<td>enum</td>
<td>explicit</td>
<td>export</td>
<td>extern</td>
</tr>
<tr>
<td>false</td>
<td>float</td>
<td>for</td>
<td>friend</td>
<td>goto</td>
</tr>
<tr>
<td>if</td>
<td>inline</td>
<td>int</td>
<td>long</td>
<td>mutable</td>
</tr>
<tr>
<td>namespace</td>
<td>new</td>
<td>not</td>
<td>not_eq</td>
<td>operator</td>
</tr>
<tr>
<td>or</td>
<td>or_eq</td>
<td>private</td>
<td>protected</td>
<td>public</td>
</tr>
<tr>
<td>register</td>
<td>reinterpret_cast</td>
<td>return</td>
<td>short</td>
<td>switch</td>
</tr>
<tr>
<td>sizeof</td>
<td>static</td>
<td>static_cast</td>
<td>struct</td>
<td></td>
</tr>
<tr>
<td>template</td>
<td>this</td>
<td>throw</td>
<td>true</td>
<td>try</td>
</tr>
<tr>
<td>typedef</td>
<td>typeid</td>
<td>typename</td>
<td>union</td>
<td>unsigned</td>
</tr>
<tr>
<td>using</td>
<td>virtual</td>
<td>void</td>
<td>volatile</td>
<td>wchar_t</td>
</tr>
<tr>
<td>while</td>
<td>xor</td>
<td>xor_eq</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Identifiers

Which of these are valid identifiers? Which aren’t? Which are valid but not necessarily good?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>hello</td>
<td>money$</td>
<td>!?#@%</td>
</tr>
<tr>
<td>_foo</td>
<td>f33d_m3</td>
<td>Aerosmith</td>
</tr>
<tr>
<td>48hours</td>
<td>N/4</td>
<td>good bye</td>
</tr>
<tr>
<td>X-ray</td>
<td>three.onefour</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>
C++ lets you specify computations using arithmetic expressions that look just like those in mathematics.

- Common arithmetic operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction/Negation</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Remainder</td>
</tr>
</tbody>
</table>
Arithmetic Expressions

- Evaluated left-to-right

  \[
  10 - 5 - 2 \rightarrow 3
  \]

- But, we have the same precedence rules as in arithmetic: multiply/divide before add/subtract

  \[
  5 + 4 \times 3 \rightarrow 17
  \]

- Use parentheses to change order of evaluation

  \[
  (5 + 4) \times 3 \rightarrow 27 \quad 10 - (5 - 2) \rightarrow 7
  \]
There is no operator to compute an exponent

Don’t do something like $5^3$: it will compile but doesn’t do what you expect!

Use `pow(x, y)` instead to compute $x^y$

```cpp
#include <cmath>

pow(5.0, 3.0) → 125.0
```

First parameter to `pow` must be floating-point
Other Functions in `<cmath>`

- Including `<cmath>` gives us many other useful functions — here are a few:

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>abs</code></td>
<td>Absolute value</td>
</tr>
<tr>
<td><code>ceil</code></td>
<td>“Ceiling” (round up)</td>
</tr>
<tr>
<td><code>cos</code></td>
<td>Cosine</td>
</tr>
<tr>
<td><code>exp</code></td>
<td>Compute $e^{x}$</td>
</tr>
<tr>
<td><code>floor</code></td>
<td>“Floor” (round down)</td>
</tr>
<tr>
<td><code>log</code></td>
<td>Natural logarithm</td>
</tr>
<tr>
<td><code>log10</code></td>
<td>Base-10 logarithm</td>
</tr>
<tr>
<td><code>max</code></td>
<td>Maximum of 2 numbers</td>
</tr>
<tr>
<td><code>min</code></td>
<td>Minimum of 2 numbers</td>
</tr>
<tr>
<td><code>pow</code></td>
<td>Compute $x^y$</td>
</tr>
<tr>
<td><code>round</code></td>
<td>Round to nearest integer</td>
</tr>
<tr>
<td><code>sin</code></td>
<td>Sine</td>
</tr>
<tr>
<td><code>sqrt</code></td>
<td>Square root</td>
</tr>
<tr>
<td><code>tan</code></td>
<td>Tangent</td>
</tr>
</tbody>
</table>
The types involved in an expression determine the type of the result.

If either value is a `double` the result is a `double`, but if both values are `int`, the result is also an `int`.

This has consequences when dividing:

- \( 14 \div 5 \rightarrow 2 \)
- \( 14.0 \div 5 \rightarrow 2.8 \)

Dividing two integers will **discard the fractional part**.
Division Gotchas

- Be careful when you use integer division in larger expressions, due to **rounding** and **order of evaluation**.

- Expressions that have mathematically the same meaning **may not** have the same result.

\[
\begin{align*}
8 \times 2 / 3 & \rightarrow 16 / 3 \rightarrow 5 \\
8 \times (2 / 3) & \rightarrow 8 \times 0 \rightarrow 0
\end{align*}
\]
More Division Gotchas

- In real life, **division by zero** is impossible, but there aren’t any consequences for trying it, except failure.

- On a computer, dividing by zero will do **weird** things:
  - Integer division by zero will probably **crash your program**
  - Floating-point division by zero won’t crash, but result in “**inf**” (infinity) or “**nan**” (not a number)
Type Casting

- We can explicitly convert a value from one type to another, called **type casting**

  \[
  \text{double}(14) / 5 \rightarrow 2.8
  \]

- Must do this when the values are stored in variables

  \[
  \text{int } x = 14; \\
  \text{int } y = 5; \\
  x / y \rightarrow 2 \\
  \text{double}(x) / y \rightarrow 2.8
  \]
Common Mistake

- What would the result be here?

```c
int x = 14;
int y = 5;
double(x / y) → ?
```
What would the result be here?

```c
int x = 14;
int y = 5;
double(x / y) → 2.0
```

- `x / y` is treated as int-by-int division, so the result is an `int`, which is then converted to a `double`
- So, the fractional part is **still lost**
Type Compatibility

- Integer values can be stored in a `double` variable without casting it first (implicit conversion)

  ```
  int x = 5;
  double y = x;
  // y now equals 5.0
  ```

- Could be explicit if we wanted, but unnecessary

  ```
  double y = double(x);
  ```
Type Compatibility

- Opposite direction: assigning a `double` to an `int` will result in silent data loss
- Converting `double` to `int` discards the fractional part

```java
double p = 5.79;
int n = p; // r now equals 5
n = int(p); // same thing
```
Magic Numbers

- What is the intent of the computation below?

  ```
  double t = 1.05 * s;
  ```

- The literal value 1.05 is a **magic number** — it doesn’t convey the **logical significance** of the value

- Someone reading your code later (maybe even you!) might not know what that line means
Named Constants

- We should reduce magic numbers in our code by creating **named constants**
- Put `const` before a variable declaration to make it a constant
- Constants **must** be given an initial value when they’re declared, and they **cannot be changed** later

```cpp
const double TAX_RATE = 1.05;
double t = TAX_RATE * s;
```
Named Constants

- Don’t take it too far though — use common sense

```c
const int FIVE = 5;
```

- Names of constants are usually ALL_UPPERCASE, to make them stand out

- It’s okay to declare constants as global variables if they’re used in a lot of places in the program

This is silly
## Some Pre-Defined Constants

<table>
<thead>
<tr>
<th>Include</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>#include &lt;cmath&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>M_PI</code></td>
<td>Mathematical constant $\pi$ (3.14159...)</td>
</tr>
<tr>
<td><code>M_E</code></td>
<td>Base of the natural logarithm, $e$ (2.71828...)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>#include &lt;climits&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>INT_MIN</code></td>
<td>Smallest possible <code>int</code> (-2,147,483,648)</td>
</tr>
<tr>
<td><code>INT_MAX</code></td>
<td>Largest possible <code>int</code> (2,147,483,647)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td><code>#include &lt;cfloat&gt;</code></td>
<td></td>
</tr>
<tr>
<td><code>DBL_MIN</code></td>
<td>Smallest possible <code>double</code> ($\approx 2.22507 \times 10^{-308}$)</td>
</tr>
<tr>
<td><code>DBL_MAX</code></td>
<td>Largest possible <code>double</code> ($\approx 1.79769 \times 10^{308}$)</td>
</tr>
</tbody>
</table>
Assignment Statements

- General assignment:
  ```
  x = y; // set x to equal y
  ```

- Shorthand assignments:
  ```
  x += y; // same as x = x + y;
  ```

- C++ also has:
  ```
  -= *= /= %=
  ```

- Adding/subtracting 1:
  ```
  x++; // same as x = x + 1;
  x--; // same as x = x - 1;
  ```
Assignment Statements

- Right-hand side can be any valid expression (valid meaning the types match and such)

\[ x = 5; \quad a = b; \quad p = q \times 5 / r; \]

- Left-hand side must be a variable

\[ x + 5 = y; \quad \text{// doesn’t make sense} \]