Variables and Expressions

CS 1044

Primitive Data Types

- 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**

Strings

```
#include <string>
```

- In computer science, we usually call a block of text a **string**
- In other words, a string is zero or more `char`s
- 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

Literal	Type
`4`	`int`
`4`	`double`
`'4'` (single quotes)	`char`
`"4"` (double quotes)	`string`

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

```
type name;
type name = value;
```

```
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

```
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

```
int x;
int y = 4 * x + 5;
cout << y;                  // ???
```

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++

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

Identifiers

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

hello	money$	!?#@%
_foo	f33d_m3	Aerosmith
48hours	N/4	good bye
X-ray	three.onefour	DOUBLE

Arithmetic Expressions

- C++ lets you specify computations using arithmetic expressions that look just like those in mathematics
- Common arithmetic operators:

+	Addition
−	Subtraction/Negation
*	Multiplication
/	Division
%	Remainder

Arithmetic Expressions

- Evaluated left-to-right

```
10 - 5 - 2  →  3
```

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

```
5 + 4 * 3  →  17
```

- Use parentheses to change order of evaluation

```
(5 + 4) * 3  →  27        10 - (5 - 2)  →  7
```

Exponents

```
#include <cmath>
```

- 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

```
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:

abs	Absolute value	max	Maximum of 2 numbers
ceil	“Ceiling” (round up)	min	Minimum of 2 numbers
cos	Cosine	pow	Compute x^y
exp	Compute e^x	round	Round to nearest integer
floor	“Floor” (round down)	sin	Sine
log	Natural logarithm	sqrt	Square root
log10	Base-10 logarithm	tan	Tangent

Math and Data Types

- 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 / 5   → 2
14.0 / 5 → 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

```
8 * 2 / 3     →  16 / 3  →  5
8 * (2 / 3)   →  8 * 0   →  0
```

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**

```
double(14) / 5    →   2.8
```

- Must do this when the values are stored in variables

```
int x = 14;
int y = 5;

    x / y    →   2
double(x) / y    →   2.8
```

Common Mistake

- What would the result be here?

```
int x = 14;
int y = 5;
```

```
double(x / y)  →  ?
```

Common Mistake

- What would the result be here?

```
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: assiging a `double` to an `int` will result in **silent data loss**
- Converting `double` to `int` **discards the fractional part**

```
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

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

Named Constants

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

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

This is silly

- 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

Some Pre-Defined Constants

`#include <cmath>`	
`M_PI`	Mathematical constant π (3.14159...)
`M_E`	Base of the natural logarithm, *e* (2.71828...)

`#include <climits>`	
`INT_MIN`	Smallest possible `int` (–2,147,483,648)
`INT_MAX`	Largest possible `int` (2,147,483,647)

`#include <cfloat>`	
`DBL_MIN`	Smallest possible `double` (≈ 2.22507×10^{-308})
`DBL_MAX`	Largest possible `double` (≈ 1.79769×10^{308})

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;      a = b;      p = q * 5 / r;
```

- Left-hand side **must be a variable**

```
x + 5 = y;    // doesn't make sense
```