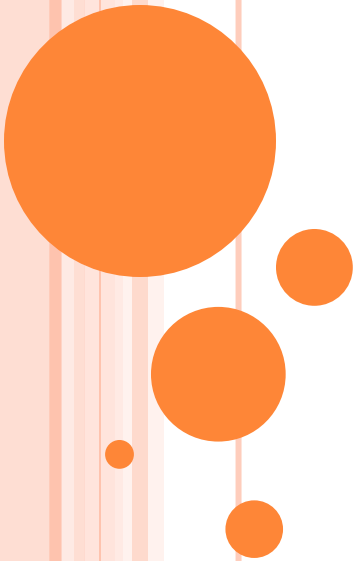

Introduction – Part 2

CS A150 - C++ Programming 1

Number Types

- **Integer** numbers (**int**) are ***whole numbers without*** a fractional part.
  - Include **zero** and **negative numbers**
  - Used for storing values that are conceptually whole numbers (for example, pennies)
  - Process faster and require less storage space
- **Floating-point** numbers (**double**) have **decimal points**.

Number Types (cont.)

- **Integer** vs. **floating-point** numbers
  - Integer arithmetic is *exact*
  - Integer implies value is *naturally* a whole number
  - Integer takes *less space* than floating-point
  - Integer processing is *faster*

  - Floating-point numbers are *approximations*
  - Floating-point numbers can store numbers from a much larger range

Numeric Ranges and Precisions

- An integer has a *limited range*
  - An **int** has range
    - -32,768 to 32,767 on a 16-bit machine
    - -2,147,483,648 to 2,147,483,647 on a 32-bit machine
- A floating-point number has *limited precision*
  - A **double** has (roughly) 15 significant decimal digits.
  - A **float** has only about 7.

Common Error

- **Float** types have **limited precision**
- **Rounding errors** are *unavoidable*:

```
(sqrt(2) * sqrt(2) == 2)  ➔ false

(sqrt(2) * sqrt(2))       ➔ 2.0000000000000004
```

  - The default display will not show this; it will show 2.

Arithmetic

- **Operators** supported in C++:

  + - * / %

- Use *parenthesis* ( ) to specify precedence and improve **readability**

- **IMPORTANT:**
  - There is ***no*** exponentiation operator (**^**) in C++
  - Use the asterisk (*****) for multiplication (***not*** dot or cross)
  - ***No*** commas in numbers in C++

    Write 10,150.75 as 10150.75

Good Programming Practice

- Place spaces on either side of a binary operator. This makes the operator stand out and makes the program more **readable**.

```
double result=op1+op2+op3;

// easier to read with spaces
double result = op1 + op2 + op3;
```

Roundoff Errors

- **Floating-type** numbers are ***approximations***;
  not always exact (consider 1/3)
- They are ***truncated*** when cast to an **`int`**, ***not*** rounded
  - Leads to rounding errors

```
int n = 4.35 * 100
```

  - **`4.35 * 100`** is a **`double`**
  - **`n`** is an **`int`**
  - **`n`** stores the value 434

Division

- An **integer** divided by an **integer** is an **integer**:

  `11 / 4` will be equal to `2`

- If either of the operands is a **floating-point** number, **floating-point** number division is used:

  `11.0 / 4` will be equal to `2.75`

  (assuming `2.75` is stored in a floating-point number)

Common Error

- What is the outcome of this expression?

```
d = 1 / 2 * 9.8 * t * t;
```

Common Error (cont.)

- What is the outcome of this expression?

```
d = 1 / 2 * 9.8 * t * t;
```

- This will give you 0 → operations are evaluated left to right
- How should you write it?

Common Error (cont.)

- What is the outcome of this expression?

```
d = 1 / 2 * 9.8 * t * t;
```

- This will give you 0 → operations are evaluated left to right
- How should you write it? *Any* of the following will do:

```
d = 1.0 / 2 * 9.8 * t * t;
```

```
d = 1 / 2.0 * 9.8 * t * t;
```

Modulus

- To get the **remainder** of division between two integers, use the modulus operator **%**

  7 % 4 will produce **3**

- **Modulus** is ***undefined*** on **float-types**.

Type Casting

- To temporarily change the type of a variable, use:

```
static_cast<type_name>(expression)
```

- Example:

```
int someValue = 5;
double result;
Result = (static_cast<double>(quarters) * 2);
```

Note that the variable **quarter** is still type **int**

Note: Do **NOT** use the ***old*** syntax → `(double)num;`

Increment and Decrement Operators

- A short-hand notation
  - **Increment** operator **++**

```
intVar++;
```
is equivalent to
```
intVar = intVar + 1;
```

  - **Decrement** operator **--**

```
intVar--;
```
is equivalent to
```
intVar = intVar - 1;
```

Short-hand Operators: Two Options

- **Post-increment**

  ```
  intVar++
  ```

  Uses current value of variable, THEN increments it

- **Pre-increment**

  ```
  ++intVar
  ```

  Increments variable first, THEN uses new value

- No difference if statement is alone:

  ```
  intVar++;
  ++intVar;
  ```

  is equivalent to

Post-increment in Action

- Given the code segment below:

```
int valueProduced;
int n = 2;
valueProduced = 2 * (n++);
cout << valueProduced << endl;
cout << n << endl;
```

n++ will be computed *after* the whole statement is executed

- The output will be as follows:

```
4
3
```

Pre-increment in Action

- Given the code segment below:

```
int valueProduced;
int n = 2;
valueProduced = 2 * (++n);
cout << valueProduced << endl;
cout << n << endl;
```

n++ will be computed ***before*** the whole statement is executed

- The output will be as follows:

```
6
3
```

Caution!

- Do NOT use increment and decrement operators inside complicated expression
- Do NOT use more than one increment operator in the same expression.

Console Input/Output

- I/O objects **cin**, **cout**, **cerr**
- Defined in the C++ library called **<iostream>**
- Must have these lines (called pre-processor directives) near start of file:

```
#include <iostream>
using namespace std;
```

Tells C++ to use appropriate library so we can use the I/O objects cin, cout, cerr

Console Output

- **`cout <<`** (*insertion* operator)
- What can be outputted?
  - Any data can be outputted to display screen
    - Variables
    - Constants
    - Literals
    - Expressions (which can include all of above)
  - Example:

```
cout << numberOfGames << " games played.";
```

    - 2 values are outputted:
      - The "value" of variable **`numberOfGames`**
      - The literal string **`" games played."`**

Console Output (cont.)

- **Cascading**: multiple values in one **`cout`**

```
cout << "Welcome to CS A150."
     << " We will meet twice a week to"
     << " learn C++ programming.";
```

Note: To improve **readability**, do ***not*** allow horizontal scrolling.

Separating Lines of Output

- New lines in output
  - Recall: "**\n**" is the escape sequence for the char "newline"
- A second method: object **endl**
- Examples:

```
cout << "Hello World\n";
```

  - Sends string "Hello World" to display and escape sequence "\n“ that goes to next line

```
cout << "Hello World" << endl;
```

  - Same result as above

Formatting Output

- You can use the "**magic formula**" to format floating-point numbers, right before outputting the number:

```
double n1 = 3.9874,
       n2 = 4.0;

cout.setf(ios::fixed);
cout.setf(ios::showpoint);    //show point even if 0
cout.precision(2);            //number of decimals

cout << n1 << " and " << n2;

//OUTPUT: 3.99 and 4.00
```

Note: The formatting stays until new formatting code is used.

Formatting Output (cont.)

- You can also use a one-line statement, but you need to include **iomanip** (input/output manipulator)

```
#include <iomanip>
...
int main()
{
        double  n1 = 3.9874,
                n2 = 4.0;

        cout << fixed << showpoint << setprecision(2);

        cout << n1 << " and " << n2;

        ...
}

//OUTPUT: 3.99 and 4.00
```

Error Output

- Output with **cerr**
  - cerr works same as cout
  - Provides mechanism for distinguishing between regular output and error output
- Re-direct output streams
  - Most systems allow cout and cerr to be "redirected" to other devices

Console Input

- **`cin >>`** (*extraction* operator)
- Must input to a variable
  - No literals allowed for cin

```
cin >> num;
```

  - Waits on-screen for keyboard entry
  - Value entered at keyboard is "assigned" to num

Console Input (cont.)

- Can read *multiple* values:

```
cin >> pennies >> nickels >> dimes >> quarters;
```

- The user enters values separated by white space:

```
8 0 3 4
```

- or enters them on separate lines:

```
8
0
3
4
```

Prompting for Inputs

- Always "prompt" user for input

```
cout << "Enter number of dragons: ";
cin >> numOfDragons;
```

  - **Note:** no "\n" in **cout**. Prompt "waits" on same line for keyboard input as follows:

```
Enter number of dragons: ____
```

(Underscore above denotes where keyboard entry is made)

- Every cin should have cout prompt
  - Maximizes user-friendly input/output

Book Example

- Display 1.5 – Using cin and cout with a string

Comments

- Add comments to explain code to other programmers or yourself
- Ignored by the compiler

```
/* A BLOCK comment is used to write comments that are
longer than one line  */
// A LINE comment is for one line only
```

- Avoid writing `//*`
- Do ***not*** nest comments.

Program Style

- Make programs easy to
  - Read
  - Modify

- Do ***not*** overcomment!

Common Errors

- Forgetting the **semicolon**
- **Misspelling** words
- **Not** differentiating between **lower** and **upper** case
- Forgetting **header files**
- What to do?
  - Test
  - Debug

- There are **two** types of errors:
  - **Syntax** errors
  - **Logic** errors

Syntax Errors

- **Syntax errors**:
  - Are **compile-time errors**
  - Are faulty input, not quite legal C++
  - Violate the language rules
  - Compiler finds the errors and reports them
  - **Cascade:** **always** start fixing errors from the top

```
cot << "Hello, World!\";
cout << "Hello, World!\n";
```

- Do **NOT** ignore warnings.

Logic Errors

- **Logic errors**:
  - Are **runtime errors**
  - Program compiles fine (input is legal)
  - Program does not do what it is supposed to do
  - Much harder to find
  - Program author must test and find the error

```
cout << "Hell, World\n";
```

C++ Standard Libraries (STL)

- Directive to "add" contents of library file to your program

```
#include <Library_Name>
```

- Called "**preprocessor directive**"
  - Executes before compiler, and simply "copies" library file into your program file.
- C++ has many libraries
  - Input/output, math, strings, etc.

Namespaces

- **Namespaces** define a collection of name definitions

```
#include <iostream>
using namespace std;
```

- This will allow us to write

`cout`	instead of	`std::cout`
`cin`	instead of	`std::cin`
`cerr`	instead of	`std::cerr`

(*and more*)

Good Programming Practice

- Compiler emits 2 messages: *errors* and *warnings*
  - Errors are *fatal*; no executable is produced.
  - Warnings are **not** fatal
    - Do **not** accept a warning, unless you can explain and justify it.
    - Warnings usually indicate that your thinking is not quite correct, and may cause your program to misbehave,
  - *Always* write code that emits **no warnings**.

Questions?

(Introduction 2)