Quizzes for Review

- Q1: What needs to be figured out for a programmer before writing a program?
- Q2: What is an algorithm? Is the algorithm for one desired problem unique?
- Q3: How do you draw a flow chart to solve the following problem: output the product of user-input numbers before entering a "#" key.
- Q4: What is an identifier? What are their composition rules?
- Q5: Write a C++ program to ask user to input his/her name and print it out.

Learning Objectives

In this lecture, you will familiarize yourself with the usage of:

- Variables and Declarations
- Data Types
- Escape Sequences
- Arithmetic Operators
- Type Conversion
- Numerical Input/Output using `cin` and `cout`
- Formatting Outputs with Manipulators

Another C++ Program

```cpp
//program_01.cpp
#include <iostream>
using namespace std;

int main()
{
    int NumberOfLanguages;
    cout << "How many languages have you used?\n";
    cin >> NumberOfLanguages;

    if (NumberOfLanguages < 1)
        cout << "Don’t worry. It’ll be fun!\n";
    else
        cout << "Cool! Enjoy the book.\n";

    return 0;
}
```
Sample Dialogues

- Dialogue 1:
  >./program_01
  How many languages have you used?
  0
  Don’t worry. It’ll be fun!

- Dialogue 2:
  >./program_01
  How many languages have you used?
  2
  Cool! Enjoy the book.

C++ Variables

- Variables
  - a memory location to store data for a program
  - must declare all data before use in program

- C++ identifiers
  - composition rules: refer to Lecture 0
  - vs. keywords/reserved words
  - case-sensitive
  - meaningful names!

Data Types (1/3)

- Objective of all programs is to process data
  1 of 3 things that we need to figure out
- Necessary to classify data into specific types
  - numerical
  - alphabetical
  - audio
  - video
- C++ allows only certain operations to be performed on certain types of data
  - prevent inappropriate programming operations

Data Types (2/3)

- Data Type: a set of values and operations that can be applied to these values

- Example of Data Type: integers
  - values: set of all Integer (whole) numbers
    ex: 23, -5, 0
  - operations: familiar mathematical and comparison operators
    ex: +, -, >, <
Data Types (3/3)

- **Built-in**: provided as an integral part of C++
  - also known as a *primitive* type
  - require no external code
  - consist of basic numerical types
  - majority of operations are symbols (e.g. +, -, *, >, <, ...)

- **Class & Struct**:
  - programmer-created data type
  - set of acceptable values and operations defined by a programmer using C++ code

Simple Data Type (1): Integer

<table>
<thead>
<tr>
<th>Type</th>
<th>Memory</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>short (a.k.a. short int)</td>
<td>16 bits</td>
<td>-32768 to 32767</td>
<td>N/A</td>
</tr>
<tr>
<td>unsigned short</td>
<td>16 bits</td>
<td>0 to 65536 (2^{16} - 1)</td>
<td>N/A</td>
</tr>
<tr>
<td>int</td>
<td>32 bits</td>
<td>-2147483648 to 2147483647</td>
<td>N/A</td>
</tr>
<tr>
<td>unsigned (int)</td>
<td>32 bits</td>
<td>0 to 4294967295</td>
<td>N/A</td>
</tr>
<tr>
<td>long (a.k.a. long int)</td>
<td>32 bits</td>
<td>-2147483648 to 2147483647</td>
<td>N/A</td>
</tr>
<tr>
<td>unsigned (long)</td>
<td>32 bits</td>
<td>0 to 4294967295</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Simple Data Type (2): Others

<table>
<thead>
<tr>
<th>Type</th>
<th>Memory</th>
<th>Range</th>
<th>Precision</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>32 bits</td>
<td>(10^{-38}) to (10^{38})</td>
<td>7 digits</td>
</tr>
<tr>
<td>double</td>
<td>64 bits</td>
<td>(10^{-308}) to (10^{308})</td>
<td>15 digits</td>
</tr>
<tr>
<td>long double</td>
<td>80 bits</td>
<td>(10^{-4932}) to (10^{4932})</td>
<td>19 digits</td>
</tr>
<tr>
<td>char</td>
<td>8 bits</td>
<td>all ASCII characters -128 to 127</td>
<td>N/A</td>
</tr>
<tr>
<td>bool</td>
<td>8 bits</td>
<td>{true, false}</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Ranges/precisions are machine-dependent
- ex: `sizeof(long)` to uncover information

Data Assignment

- Initialize data during declaring statement(s)
  - ex: `int myValue = 0;`
  - get "undefined" if you don’t! ex: `int xx;`

- Assign data during execution
  \[ L\_value (\text{variable}) = R\_value (\text{expression}); \]

- Examples:
  - `var1 = 5.5;`
  - `distance = rate * time;`
  - `perimeter = 2 * 3.14 * r;`
Shorthand Notations

- `count += 2;
- `total -= part;
- `bonus *= 2;
- `time /= factor;
- `change %= 2;
- `amount *= num1 + num2;"

Compatibility of Data Assignments

- **Type mismatches**
  - `(type 1) var1 = (type 2) var2`
  - cannot place value of one type into variable of another type perfectly
  - Ex: `int Var = 2.99;`
  - assign a floating # to an integer variable
  - only integer part fits into the variable
  - called *implicit* (or *automatic*) type conversion

- **Literals (a.k.a. constants)**
  - ex: `2, 5.75, 'Z', "Hello World"
  - do not change in programs

Literal Data

- **Examples of Literals**
  - `-2 // Literal constant int`
  - `-5.75 // Literal constant double`
  - `-'Z' // Literal constant char`
  - `-'Hello World' // Literal constant string`

  - not change values during execution

  - termed *literals* because programmers literally typed them in programs

Escape Sequences

- **Enable users to extend character set**

- **Backslash, \ preceding a character**
  - instruct compiler: a special "escape character" is coming
  - following character treated as "escape sequence char"

- **common escape sequences**

<table>
<thead>
<tr>
<th>\n</th>
<th>new line</th>
<th>\r</th>
<th>carriage return</th>
</tr>
</thead>
<tbody>
<tr>
<td>\t</td>
<td>(horizontal) tab</td>
<td>\a</td>
<td>alert (a bell)</td>
</tr>
<tr>
<td>&quot;</td>
<td>a double quote</td>
<td>'</td>
<td>a single quote</td>
</tr>
<tr>
<td>&quot;</td>
<td>a double quote</td>
<td>\0</td>
<td>null character</td>
</tr>
</tbody>
</table>
Named Constants

- Direct use of literal constants are OK, but provide little meanings
  - Ex: Seeing a 24 in a program tells nothing about what it represents
- Use named constants (a.k.a. declared constants) to name your own constants
  - A meaningful name to represent data
- Ex: const int NUM_OF_STUDENTS = 24;
  - Const is called a modifier
  - Use its name wherever needed in programs
  - Changes to value result in one fix only

Example of Named Constants

... const double pi = 3.14159;
double radius;

cout << "please enter a radius:";
cin >> radius;
cout << "\nthe area of this circle =";
cout << (pi*radius*radius) << "\n";
...

Q: What if we want to change the precision of pi?

Quizzes for Review

- Write a program that does:
  - Prompt the user to input 3 floating-point numbers and 2 integers.
  - Print out all 5 numbers line by line.
  - Sum up the first 2 floating-point numbers.
  - How do you interpret fltVar3 = intVar2;
  - Use named constants for π and fltVar5 as the radius to compute the circle area.

Arithmetic Operators

- Arithmetic expressions contain
  - Operator: ex: +, −, etc...
  - Operand: including unknown variables and known numbers
- Standard Arithmetic Operators: +, −, *, /, %
  - Unary operator: the operator only requires one operand. ex: negation(−): −5
  - Binary operator: the operator requires two operands. ex: subtract(−): x−y
- Precedence rules – standard priority
  - *, /, %
  - +, −
Arithmetic Precision

- Precision of Calculations: very important!
  - Expressions in C++ might not evaluate as you’d expect!
  - Highest-order operand determines precision
- Examples: in C++
  - 17 / 5 evaluates to 3: integer operands ⇒ integer division
  - 17.0 / 5 equals 3.4: highest-order operand is double ⇒ double precision division
- Quiz: What is the result?
  int var1 = 7; float var2 = 3.6;
  int Result = var1/var2;
  Ans: Result = 1

Type Coercion

- Calculations done one-by-one
  - Ex: 1/2/3.0/4 performs 3 separate divisions.
    - first ⇒ 1 / 2 equals 0
    - then ⇒ 0 / 3.0 equals 0.0
    - last ⇒ 0.0 / 4 equals 0.0
  - Type coercion: implicitly (automatically) C++ convert the data type for you
    - ex: double d = 5; //convert 5 to 5.0
    - ex: 4 * 3 + 7 / 5 - 25.5

Type Casting

- Cast: explicitly convert the data type of a value to another data type
  - two versions: compile-time and run-time
- Compile-time casting: unary operator with syntax: DataType(expression)
  - expression converted to data type of DataType. ex: double(9)/2;
- Run-time casting: requested conversion checked at runtime, applied if valid
  - Syntax: static_cast<DataType>(expression)
  - ex: static_cast<double>(9)/2;

Shorthand Operators

- Increment & Decrement
  - Just short-hand notations
  - Increment operator, ++
    intVar++;
    is equivalent to
    intVar = intVar + 1;
  - Decrement operator, --
    intVar--;
    is equivalent to
    intVar = intVar - 1;
Post-Increment vs. Pre-Increment

- **Post-Increment**: intVar++
  - Uses current value; then increments it
- **Pre-Increment**: ++intVar
  - Increments first; then uses new value

Two examples:
- int n = 2;
  int product = 3 * (n++);  Answer: 3 and 6
  cout << n << " and " << product;
- int n = 2;
  int product = 3 * (++n);  Answer: 3 and 9
  cout << n << " and " << product;

- No difference if alone in a statement
  - intVar++; and ++intVar; ⇒ identical result

Console Input/Output (I/O)

- **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;

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

Console Output

- What can be outputted? ⇒ Any data can be outputted to display screen, including:
  1. variables
  2. constants
  3. literals
  4. expressions (can include all of above)

- cout << NumberOfGames << "games played.";
  two values are outputted:
  - value of variable ⇒ NumberOfGames
  - literal string ⇒ "games played."
- Cascading: multiple values in one cout

Separating Lines of Output

- New lines in output
  - recall: "\n" is 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", skipping to next line.
  cout << "Hello World" << endl;
  ⇒ same result as above
Formatting Outputs (1/2)

- Format numeric values for output
  - values may not display as you'd expect!
- Example:
  ```
  cout << "The price is $" << price << endl;
  // If price (declared double) has value 78.5, you might get on screen:
  1. The price is $78.500000 or
  2. The price is $78.5 or
  3. other ???
  ```
- We must explicitly instruct C++ how to output numbers in our program.

Formatting Outputs (2/2)

- Using `cout` with **manipulator** (`mani`)
  ```
  cout << expr./mani << expr./mani << ...;
  ```
  - require another library "iomanip"
- Manipulators of interest include:
  - `setw`: specify the width of the output expression.
  - `left/right`: align the expression from the leftmost or rightmost.
  - `setprecision`: control the output precision of a float/double number in expression.
  - `showpoint`: enforce to show the decimal part
  - `fixed`: fix the decimal format for the output float/double numbers.

Set Width: `setw(n)` & Left-Justify `left`

- `setw(n)` can set the width `n` for the output data
  - can apply to formatting a string or number
  - output by default is right-justified
- `left` can justify output at leftmost column
  - disable by `cout.unsetf(ios::left)` and return right-justified format
- Example:
  ```
  #include <iomanip>
  ...cout << setw(3) << 1357 << endl;
  cout << setw(4) << 1357 << endl;
  cout << setw(6) << 1357 << endl;
  cout << left << setw(6) << 1357 << endl;
  ```

Set Decimal Places: `setprecision(n)`

- For float/double data types, `setprecision(n)` can force `n` decimal places:
  ```
  #include <iomanip>
  ...cout << setprecision(2);
  cout << "The price is $" << price;
  ```
  - force all future outputted values to be exactly two digits after the decimal point
- on screen: **The price is $78.50**
Fix Format fixed & Show Decimals showpoint

- fixed outputs the floating-point numbers in a fixed-point decimal format. Ex: 23.34
  - Disable the fixed format: `cout.unsetf(ios::fixed)`
  - Enforce scientific format: `use scientific`
- showpoint forces to show the decimal points and trailing zeros
- Example:
  ```cpp
  float fltVar = 15;
  double dblVar = 25.9876;
  cout << fixed << showpoint << setprecision(2);
  cout << fltVar << " and " << dblVar << endl;
  ```

Error Output

- Output with `cerr`
  - `cerr` works same as `cout`
  - provide mechanism for distinguishing between regular output and error output
  - rarely used so far
- Re-direct output streams
  - most systems allow `cout` and `cerr` to be "redirected" to other devices
  - e.g., line printer, output file, error console, etc.

Input Using `cin`

- `cin` for input with `">>"`
  - extraction operator `>>` points opposite ⇒ think of it as "pointing toward where the data goes"
  - no literals allowed for `cin` ⇒ must input to a variable
  - also support multiple inputs
- Ex: `cin >> num1 >> num2 >> num3;`
  - wait on-screen for keyboard entry
  - values entered from keyboard are assigned (stored) to `num1`, `num2` and `num3`

Prompting for Input: `cin` and `cout`

- Every `cin` should have `cout` prompt
  - maximize user-friendly input/output
- Example:
  ```cpp
  cout << "Enter number of games:";
  cin >> NumOfGames;
  ```
  - no "\n" in the `cout` line
  - prompt waits on same line for keyboard input as follows:
    > Enter number of games:
  - underscore (_) above denotes where keyboard entry is made
- Identifiers in C++ is case-sensitive
- Use meaningful names
  - for variables and literals (constants)
- Variables must be declared before use
  - should also be initialized
- Use care in numeric manipulation
  - precision, parentheses, order of operations
  - shorthand notations (ex: +=, ++, --)
- Beware of datatypes during assignments
  - Type coercion, type casting

Object cout
- Used for console output

Object cin
- Used for console input

Object cerr
- Used for error messages

Format the data by manipulator
- float/double control: fixed, showpoint and setprecision()
- width control: setw()
- left-justified outputted data: left