Review 1

CS A250 – C++ Programming II

What Are We Reviewing?

- Prerequisite for this course: **CS A150**

- This presentation will *only* *<u>outline</u>* important **key points** that are **needed** for this class.
  - It is your responsibility to review any topic from **CS A150** not covered in this class's reviews.
  - This review contains a collection of items that you need to keep in mind when coding to avoid **losing points**.

Objectives

- Items that will be reviewed:
  - Identifiers
  - Literals vs. constants
  - Arithmetic precision, type casting, and decimal format
  - Shorthand notation / Prefix and postfix
  - The optional else
  - Conditional operator
  - Functions and passing parameters
  - Function overloading
  - Arrays and vectors: capacity, size and number of elements
  - The **const** modifier for parameters

Identifiers, Literals and Constants

Identifiers

- An **identifier** is a name given to a variable, a constant, a function, an object, a class…

```
int myInteger = 3;

vector<int> myVector;

MyClass myObject;
```

- Identifiers in C++ are **case-sensitive**

Good Programming Practice

- To improve **readability**:
  - Choose meaningful identifiers
  - Do not abbreviate
  - Follow the standards we discussed for this class

    (see syllabus)

Literal Data

- **Literals**
  - Examples:
    - 2 // Literal constant **int**
    - 5.75 // Literal constant **double**
    - ‘Z’ // Literal constant **char**
    - "Hello World" // Literal constant **string**
- **Cannot change** values during execution
- Called "literals" because you "literally" type them in your program!

CONSTANTS

- Literals are "OK", but provide little meaning
  - For example, seeing the number 24 throughout your code, tells nothing about what it represents

→ Use named **constants** instead

```
const int NUMBER_OF_STUDENTS = 24;
```

- Use all CAPITAL_LETTERS separated by an underscore for **identifiers** that refer to **constants**.

- **Added benefit:** changes to value can be done in one fix

Which One to Use?

- Use a **literal** if
  - You are using that value only once
  - The value will never change (weeks in a year)
- If you use a **literal**, always provide a comment to explain what the value represents
  - Example: Dividing a salary by 52 weeks
    - Can leave "52" because it is used only once, but need to comment about what it represents.
- Use a **global constant** if the value will be used more than once AND/OR the value might need to be changed in the future
  - Example: An interest rate

About Global Variables…

- Do NOT use them!
- <u>Only</u> **global *constants*** will be allowed!

Numbers

Arithmetic Precision

- Precision of calculations
  - VERY important consideration!
  - Expressions in C++ might not evaluate as you would "expect"!
  - "Highest-order operand" determines type of arithmetic "precision" performed
  - Common error!

(see examples on next slide)

Arithmetic Precision (cont.)

- Examples:
  - **17 / 5** evaluates to **3** in C++
    - Both operands are **integers**
    - Integer division is performed and gives incorrect result
  - **17.0 / 5** equals **3.4** in C++
    - **Double** "precision" division is performed
  - The following performs ***integer*** division, giving a result of 0

```
int n1 = 1,
    n2 = 2;
cout << (n1/ n2);
```

(more…)

Arithmetic Precision (cont.)

- Calculations done "one-by-one"
  - **1 / 2 / 3.0 / 4** performs 3 separate divisions
    - First→ **1 / 2** equals 0
    - Then→ **0 / 3.0** equals 0.0
    - Then→ **0.0 / 4** equals 0.0!
- So changing just "one operand" in a large expression can lead to incorrect results
  - Must keep in mind all individual calculations that will be performed during evaluation!
  - Do NOT trust your program…
    - Trust your calculator

Type Casting

- **Casting** for variables
  - Can add ".0" to literals to force precision arithmetic

```
cout << (5.0 / 2) << endl;
```

  - Can use `static_cast<type>` for variables
    - Casting is only temporary → variable **num** will stay an **integer**

```
int num = 2;
double x = static_cast<double>(num) / 2;
```

- Do **NOT** use ~~`(double)num`~~

FORMATTING DECIMALS

- Decimal format is <u>only</u> for output
- Option 1:

```
cout.setf(ios::fixed);
cout.setf(ios::showpoint); //shows point even if 0
cout.precision(2);          //shows 2 decimals
```

- Option 2:

```
#include <iomanip>
...
cout << fixed << showpoint << setprecision(2);
```

Shorthand Notation

Shorthand Notations

- Use them!

EXAMPLE	EQUIVALENT TO
`count += 2;`	`count = count + 2;`
`total -= discount;`	`total = total - discount;`
`bonus *= 2;`	`bonus = bonus * 2;`
`time /= rushFactor;`	`time = time/rushFactor;`
`change %= 100;`	`change = change % 100;`
`amount *= cnt1 + cnt2;`	`amount = amount * (cnt1 + cnt2);`

Prefix and Postfix

- **Post-Increment**

```
int n1 = 3;
int n2 = n1++;
```

  - Uses current value of variable, THEN increments it
- **Pre-Increment**

```
int n1 = 3;
int n2 = ++n1;
```

  - Increments variable first, THEN uses new value

- No difference if "alone" in statement:

```
n1++;
++n1;
```

they both give the same result.

Prefix in Expressions

```
int firstNumber = 2,
    secondNumber = 3;

while (++firstNumber < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

```
firstNumber = 2
```

What is the output?

Prefix in Expressions (cont.)

```
int firstNumber = 2,
    secondNumber = 3;

while (++firstNumber < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

`firstNumber =` ~~`2`~~ `3`

This happens first and **firstNumber** is incremented by 1.

PREFIX IN EXPRESSIONS (CONT.)

```
int firstNumber = 2,
    secondNumber = 3;

while (++firstNumber < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

```
firstNumber = 3

3 < 3 ? FALSE
```

Comparison happens next.

Prefix in Expressions (cont.)

```
int firstNumber = 2,
    secondNumber = 3;

while (++firstNumber < secondNumber)
{
      cout << firstNumber << endl;
}
cout << firstNumber;
```

```
firstNumber = 3
```

Condition is false and body of loop will not be executed.

Prefix in Expressions (cont.)

```
int firstNumber = 2,
    secondNumber = 3;

while (++firstNumber < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

`firstNumber = 3`

Program continues and prints value of **`firstNumber`**.

Prefix in Expressions (cont.)

```
int firstNumber = 2,
    secondNumber = 3;

while (++firstNumber < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

```
firstNumber = 3
```

```
OUTPUT:

3
```

Postfix in Expressions

```
int firstNumber = 2,
    secondNumber = 3;

while (firstNumber++ < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

```
firstNumber = 2
```

What is the output?

POSTFIX IN EXPRESSIONS (CONT.)

```
int firstNumber = 2,
    secondNumber = 3;

while (firstNumber++ < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

```
firstNumber = 2

2 < 3 ? TRUE
```

The whole condition is evaluated first.

Postfix in Expressions (cont.)

```
int firstNumber = 2,
    secondNumber = 3;

while (firstNumber++ < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

`firstNumber = ~~2~~ 3`

The value of **`firstNumber`** is incremented by 1.

Postfix in Expressions (cont.)

```
int firstNumber = 2,
    secondNumber = 3;

while (firstNumber++ < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

```
firstNumber = 3
```

Body of loop is executed.

```
OUTPUT:

3
```

Postfix in Expressions (cont.)

```
int firstNumber = 2,
    secondNumber = 3;

while (firstNumber++ < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

```
firstNumber = 3

3 < 3 ? FALSE
```

Condition is evaluated again.

```
OUTPUT:

3
```

Postfix in Expressions (cont.)

```
int firstNumber = 2,
    secondNumber = 3;

while (firstNumber++ < secondNumber)
{
      cout << firstNumber << endl;
}
cout << firstNumber;
```

firstNumber = ~~3~~ 4

The value of **firstNumber** is incremented by 1.

OUTPUT:

3

Postfix in Expressions (cont.)

```
int firstNumber = 2,
    secondNumber = 3;

while (firstNumber++ < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

`firstNumber = 4`

Condition is false and body of loop will not be executed.

```
OUTPUT:

3
```

Postfix in Expressions (cont.)

```
int firstNumber = 2,
    secondNumber = 3;

while (firstNumber++ < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

`firstNumber = 4`

Program continues and prints value of **firstNumber** again.

```
OUTPUT:

3
4
```

Prefix and Postfix

- As we saw, prefix and postfix **might** change the results of the statement.

```
int firstNumber = 2,
    secondNumber = 3;

while (++firstNumber < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

```
OUTPUT:

3
```

```
int firstNumber = 2,
    secondNumber = 3;

while (firstNumber++ < secondNumber)
{
        cout << firstNumber << endl;
}
cout << firstNumber;
```

```
OUTPUT:

3
4
```

Conditions

The Optional Else

- In an **if** statement, **else** clause is ***optional***
  - If, in the **false** branch (**else**), you want "nothing" to happen → <u>leave it out</u>
  - Example:

```
if (sales >= minimum)
        salary += bonus;
cout << "Salary = " << salary;
```

  - **Note:**
    - Nothing to do for **false** condition, so there is **<u>no</u>** **else** clause!
    - Execution continues with **cout** statement

Conditional Operator

- **Conditional operator**, also called "**ternary operator**"
  - Essentially "shorthand if-else" operator

```
if (n1 > n2)
        max = n1;
else
        max = n2;
```

  - Can be written:

```
max = (n1 > n2) ? n1 : n2;
```

  - "?" and ":" form the "**ternary**" operator

Conditional Operator

- **Avoid** using the **conditional operator** in an output expression, because misplacing parenthesis can produce unwanted results:

```
cout << ( ( grade < 60 ) ? "fail" : "pass" );
       // prints pass or fail

cout << ( grade < 60 ) ? "fail" : "pass";
       // prints 1 or 0

cout << grade < 60 ? "fail" : "pass";
       // error: compares cout to 60
```

Functions

FUNCTIONS

- Two types:
  - **void**
    - Does **not** return a value

> Do **NOT** exit from a **void** function using
>
> **return;**
>
> Find an elegant way to terminate the execution of the function.

  - **return a value**
    - In C++
      - Only one value can be returned
      - Cannot return arrays
      - Cannot return functions

Function Declaration

- **Function declaration**
  - Syntax

```
return_type funcName ( parameter_list);
```

  - Goes *before* **main( )** function
  - May or may not have parameters
    - Although there is no need for parameter names, it improves readability to include names.
  - Comments go before ***or*** after function declaration
  - Also known as **function prototypes**

Function Definitions

- **Function definition**
  - **Syntax**

```
return_type funcName (parameter_list )
{
    // body
}
```

  - Goes *after* the **main( )** function
  - May or not may have a parameter list
  - **Parameters** are **automatic objects**
    - They are destroyed when execution of the function terminates, just like **local variables**

Argument Passing

- The type of **parameter** determines the interaction between the **parameter** and its **argument**

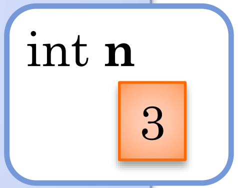
  - If the **parameter** is **passed by reference**
    - Parameter is bound to its argument

  - If the **parameter** is **passed by value**
    - The value is copied

Passing Arguments by Value

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```cpp
void myFunction (int n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```



Passing Arguments by Value (cont.)

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```
void myFunction (int n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```

int **n**

3

call to **myFunction (3)**

Passing Arguments by Value (cont.)

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```
void myFunction (int n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```

a local copy of **n** is created

int **n**

3

int **n**

3

Passing Arguments by Value (cont.)

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```
void myFunction (int n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```

local variable **n** is incremented

int **n**
4

int **n**
3

Passing Arguments by Value (cont.)

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```
void myFunction (int n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```

cout statement is executed

int **n**
4

int **n**
3

OUTPUT:

4

Passing Arguments by Value (cont.)

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```
void myFunction (int n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```

function execution is terminated and all local variables are destroyed

int **n**
3

```
OUTPUT:

4
```

Passing Arguments by Value (cont.)

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```
void myFunction (int n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```

int **n**

3

```
OUTPUT:

4
3
```

return to function call and print **n** again

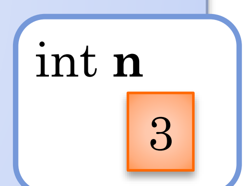
Passing Arguments by Reference

- When **passing by reference**
  - **Address** of argument is passed
  - Caller’s data can be modified by called function
  - Typically used
    - For input function to retrieve data for caller,
      data is then "given" to caller
    - When more than one value needs to be returned
  - Specified by **ampersand** (**&**) after type in parameter list

Passing Arguments by Reference

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```
void myFunction (int& n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```



Passing Arguments by Reference (cont.)

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```
void myFunction (int& n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```

int **n**

3

call to **myFunction (3)**

Passing Arguments by Reference (cont.)

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```
void myFunction (int& n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```

address of **n** is passed

int& **n**

[address]

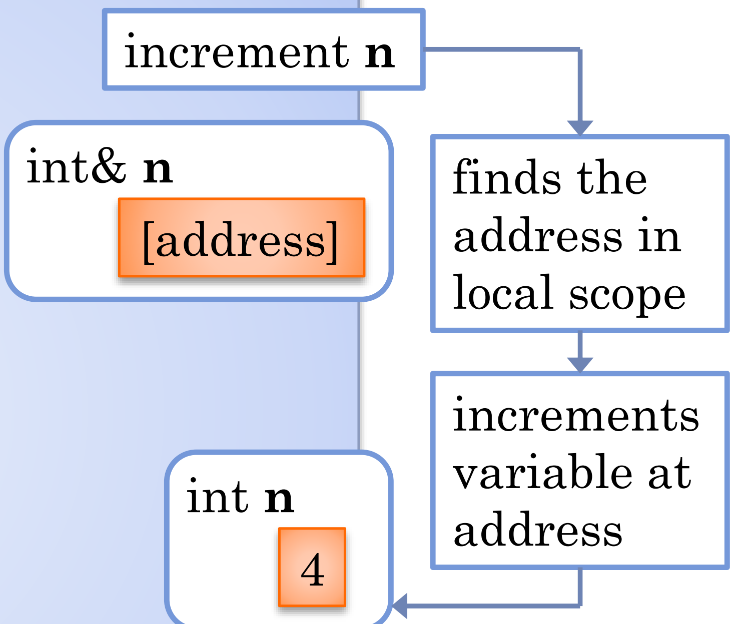
int **n**

3

Passing Arguments by Reference (cont.)

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```
void myFunction (int& n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```



Passing Arguments by Reference (cont.)

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```
void myFunction (int& n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```

cout statement is executed

int& **n**

[address]

int **n**

4

OUTPUT:

4

Passing Arguments by Reference (cont.)

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```
void myFunction (int& n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```

function execution is terminated and all local variables are destroyed

int **n**

4

OUTPUT:

4

Passing Arguments by Reference (cont.)

- When passing arguments by value, any changes made to the function **parameter** will not change the **argument**

```
void myFunction (int& n)
{
        ++n;
        cout << n << endl;
}
int main( )
{
        int n = 3;
        myFunction(n);
        cout << n;
        ...
}
```

int **n**

4

return to function call and print **n** again

OUTPUT:

4
4

Example

- **Project:** parameter_passing

Function Overloading

- **Overloaded functions** have
  - **Same** function **name**
  - **Different parameter** lists
  - Two **separate** function **declarations**/**definitions**
  - Function "**signature**"
    - Function name & parameter list
    - Must be "unique" for each function definition
  - Allows same task performed on different data

FUNCTION OVERLOADING (CONT.)

- Example:

```
double compute( double n1, double n2);
double compute( double n1, double n2, double n3);
double compute( int n1, double n2);
```

- The above functions have the same name but have parameters that differ in numbers and/or types.

- **Careful**: ***Return type*** does **not** matter

Arrays and Vectors

Capacity, size, and number of elements

Capacity, Size, and Number of Elements

- **Arrays** are frequently **partially filled**.
- Need to differentiate the **physical length** of the array from the actual **number of elements** that occupy the array.
- We will use the following conventions:
  - The **capacity** to define the **physical length** of the array
  - The **number of elements** to define the total number of **items stored** in the array.

- We will **NOT** use "size" when referring to arrays.

Capacity of Static Arrays

- Capacity of **static arrays** must be defined at **compilation time**
  - Always use defined/named **constant** for array capacity

```
const int CAPACITY = 5;

...

int score[CAPACITY];
```

Vector Size

- The **STL vector class** defines **size** as the **number of elements** stored in the vector.
- If using a loop, **avoid** calling the function size inside the loop and use a variable instead

```
int size = static_cast<int>( v.size() );

for (int i = 0; i < size; ++i)
      cout << v[i] << " ";
```

Function size() returns an **unsigned int**, but we can **cast** it to an **int**.

Reference, Value and `const` Modifier for Parameters

THE `const` MODIFIER FOR PARAMETERS

- **Reference arguments** inherently "dangerous"
  - Caller's data can be changed
  - Often this is desired, sometimes not
- Use the **`const`** modifier to "protect" data

- So, when should you use **&** and when **`const`**?
  - Use **`const`** **<u>ONLY</u>** if **passed by reference**

  - **First question:** When to pass by reference?

When to Pass by Reference?

- When passing **objects**
  - They are **large**; no need to make another copy
  - Example: strings, vectors, objects of classes you created

```
void someFunction(string& name, MyClass& obj)
{
    // does something
}
```

When to Pass by Reference? (cont.)

- When passing **variables** that need to be **changed** and **retain** their new value after the function is done

```
double calculatePayCheck()
{
    double payRate = 0.0, hours = 0.0;
    getInfo(payRate, hours);
    return (payRate * hours);
}
void getInfo(double& payRate, double& hours)
{
    cout << "Enter pay rate and total hours worked: ";
    cin >> payRate >> hours;
}
```

The value of **payRate** and **hours** will be determined by the user and they need to send the information back to the function calling.

When to Use `const`?

- **IF** you are passing by reference (&)
  - **AND** the value passed by the parameter should **<u>not</u>** be modified inside the function
    - **THEN** use `const`

```
void printVector(const vector<int>& v)
{
    int size = static_cast<int>( v.size() );

    for (int i = 0; i < size; ++i)
        cout << v[i] << " ";
}
```

When to Pass by Reference (cont.)

- **Careful! Arrays** are automatically **passed by reference**, but **<u>no</u> &** is used!
  - Need to use **const** when necessary

```
void fillArray(int a[], int numOfElem)
{
     for (int i = 0; i < numOfElem; ++i)
          a[i] = i + 1;
}
void printArray(const int a[], int numOfElem)
{
     for (int i = 0; i < numOfElem; ++i)
          cout << a[i] << " ";
}
```

Array will be modified. Cannot use **const**.

Array should **not** be modified. Use **const**.

Examples

- **Project:** arrays
- **Project:** vectors

Good Programming

Changing Flow of Control

❖ ❖ ❖ **IMPORTANT** ❖ ❖ ❖

Do **NOT** use:

- **break** (<u>except</u> on **switch** statements) and/or
- **continue** in any of the exercises and /or programming exams

Choose an elegant way to exit **loops** and **functions.**

A Few Rules

- When **creating a new VS project**
  - Name your project "**Project**"
    - You should rename the folder later
    - If the project name is too long, files might not be transfer when you turn in your project
  - Name the file that contains the main( ) function "**Main.cpp**"
    - We will be exchanging files; therefore, we ALL need to use same naming conventions
  - Do **NOT** forget the **name header**
    - You will lose points if you do
    - Make sure has the same format shown on the syllabus

A Few Rules (cont.)

- When **coding**:
  - Leave a **space** in between operators
  - Leave a **line** in between blocks of code
  - Split statements to **avoid horizontal scrolling**
  - Improve readability when writing decimal numbers:

    0.0 instead of .0

    3.0 instead of 3
  - Write code that is easy to read and understand
    - You are not going to look "cool" if you write some code that is difficult to read
  - Declare variables only <u>right before</u> you need them, instead of listing them at the beginning of the function

Are You Detail Oriented?

- As a **programmer**, you need to:
  - Make sure your program is **readable**
  - Choose an implementation that makes your code **efficient**
  - **Follow instructions** carefully

End Review 1