COP 3331
OBJECT ORIENTED DESIGN
SUMMER 2017

WEEK 1 – WEDNESDAY (MAY 17TH):
C++ BASICS, FUNCTIONS, ARRAYS, AND VECTORS

C++: BASICS AND NEW FEATURES

C++: USING DIRECTIVE & NAMESPACES

- Recall: The *using* directive allows us to use all the names/commands we expect to use from a header file
- The namespace is an organization mechanism that allows us to group symbols and identifiers
  - This prevents confusion between identifiers that share the same name
- The line `using namespace std;` allows us to use all identifiers from the standard (std) namespace
  - This prevents addition of prefixes to the names

C++: USING DIRECTIVE & NAMESPACES

- Without using directive:
  - `std::cout << "FizzBuzz\n”;`
- With using directive:
  - `cout << "FizzBuzz\n";`

C++: OUTPUT

- Recall: we need the iostream header file with a preprocessor directive:
  - `#include <iostream>`

- “Angular” brackets <> refer to header files included from the standard library
  - We use double quotes for header files that we create
  - Class definitions will be housed in their own header files

- For output, we use the *cout* object
  - Means *console output* (prints to screen)

C++: OUTPUT

- We use the stream insertion operator << to indicate *what* we are outputting

- Multiple << operators can be used for chaining (concatenating) output (including calculations)
  - `cout << "2 + 3 = " << 2 + 3 << endl;`

- << operators can be used to "split" output statement
  - ```
    cout << "This output takes up "
            << "three lines but still prints on "
            << "one line. " << endl;
    ```
    ```

C++: INPUT

- With input, we include the iostream header file, and the using directive for convenience

- For input, we use the *cin* object
  - Means *console input* (received from keyboard)

- cin is often used with cout to indicate when input is needed
  - Recall: this type of output is called a *prompt*

C++: INPUT

- We use the stream extraction operator >> to indicate where input should be saved
  - Recall: basic input is typically stored in a *variable*
  - `cin >> x;`

- Multiple >> operators can be used to store multiple values in multiple variables
  - `cin >> x >> y >> z;`
  - You cannot use , to achieve this effect!
  - `cin >> x, y, z;` will produce logical errors

C++ 11: INITIALIZATION LIST

- Recall:
  - C++ 11 allows you to create an initialization list like this:

```
int i{0};
```

  - The syntax above is equivalent to:

```
int i = 0;
int i = {0}; //form without = is preferred
```

  - You can perform multiple initialization like this:

```
int i{0}, j{0}, k{0};
```

C++ 11: INITIALIZATION LIST

- Initialization list can be used with expressions
  - e.g. `int x {3};`

    `int y{x * 3};`

- Remember: The statement `y{x * 3};` is functionally equivalent to: `y = x * 3;`

- Important: Initialization lists *prevent implicit and narrowing conversions*

C++ INITIALIZATION

- This works:
  - ```
    int x = 3;
    double y = x * 3;
    ```
  - ```
    double x = 3;
    int y = x * 3;
    ```
- But this doesn't:
  - ```
    int x{3};
    double y{x * 3};
    ```
  - ```
    double x{3};
    int y{x * 3};
    ```

C++ INITIALIZATION

- This works:
  - ```
    int a{6}, b{4};
    double c = a / b;
    ```
    ✓

- But this doesn't:
  - ```
    int a{6}, b{4};
    double c {a / b};
    ```
    ✗

- Tip: pick a format and stick with it!

C++: TYPE CASTING

- We can use type casting to perform explicit conversion

- Explicit type casting is achieved using the cast operator
  - e.g. `static_cast<double>(a)`

- static_cast can be used in initialization lists!
  - This syntax is valid!

```
int a{6}, b{4};
double c {static_cast<double>(a) / b};
```

C++ 11: LARGER INT TYPES

- Recall:
  - `int` types typically hold values from -2,147,483,648 to +2,147,483,647
  - A `long int` is at least as large as an `int`

- For current problems/applications, this range is too small!
  - e.g. Population of earth: ~ 7.5 billion
    Amount of money in circulation: ~$1.2 trillion

C++ 11: LARGER INT TYPES

- Solution: C++ 11 introduces larger ints!
  - `long long int` types typically hold values from -9,223,372,036,854,775,808 (-9.2 quintillion) to -9,223,372,036,854,775,808 (9.2 quintillion)
  - `unsigned long long int` types hold values from 0 to 18,446,744,073,709,551,615 (18 quintillion)
  - For *exact* ranges (as ranges may vary from system to system), you can use `int64_t`
    - Even more in <cstdint> header file!

C++ 11: TYPES FOR SWITCH STATEMENT

- Recall: switch statements require integral types :
  - `int, bool, char`

- C++ 11 expands the integral types used in switch statements
  - signed: `int, bool, char, char16_t, char32_t, wchar_t, long, long long` …
  - unsigned: `int64_t, uint64_t` …

C++ 11: AUTO DECLARATION

- Recall: when declaring a variable, specify its data type

- C++ 11: you can use the auto keyword to tell the compiler to infer the data type based on its initialization value
  - e.g. `auto num = 4;`
    `auto num{4};`

- Initial value must be provided!
  - Also, watch the format: `auto num = {4};` does not work!

- Auto keyword designed to simplify more complex data types, so don't get lazy!

C++: FUNCTIONS

C++ FUNCTIONS

- Recall: A *function* is a collection of programs that perform a specific task
- Functions allow us to organize large programs by constructing them from smaller pieces or components
- Most C++ program consists of:
  - Prepackaged functions (from C++ Library) – header files required
  - Functions you create – definitions must be easy to reference (from program or user-defined classes)

C++ FUNCTIONS

- Every function should be limited to performing a single, well defined task
- The name of the function should express that task
- Functions may be:
  - value returning (returns a value using the return statement)
  - void (does not return a value; can use return to exit the function)

C++ FUNCTIONS

- Recall: execution begins at main
  - main is traditionally defined as a value returning function
  - return 0; for main indicates successful termination
  - Note: If the end of main is encountered *without* return 0; normal program termination is assumed…
    - … so people like to omit it entirely

- Functions must be called (first from main, then from other functions) to be used

C++ FUNCTIONS

- Recall: functions consist of:
  - Header
  - Body
- Function headers consist of:
  - function name
  - number and Data type of parameters
  - data type of returned value
- Function body consists of
  - code that defines the task

C++ FUNCTIONS

- Recall: user-defined functions may be defined before or after main
  - If defined after main, a function prototype must be used before the main function

- Values in a function may be passed by 2 methods:
  - Pass by value (copy of the value is passed)
  - Pass by reference (memory address of the actual parameter is referenced)

C++ FUNCTIONS - EXAMPLE

```
#include <iostream>
#include <cstdlib> // contains prototypes for functions srand and rand
#include <ctime> // needed for time function
using namespace std;

int larger (int x, int y);  // function prototype

int main()
{
    // seed random number generator with system clock
    srand(static_cast <unsigned int> (time(0)));

    int number1 {1 + rand() % 100};
    int number2 {1 + rand() % 100};

    int lrg {larger(number1, number2)}; // yup, that's a function call
                                        // in an initialization list
```

C++ FUNCTIONS - EXAMPLE

```
    cout << "The numbers selected were: " << number1 << " and "
        << number2 << endl;

    cout << "The larger number is " << lrg << endl;
}

int larger (int x, int y)
{
    if (x > y)
        return x;
    return y;
}
```

<u>Sample Output:</u>

The numbers selected were: 45 and 27
The larger number is 45

C++ FUNCTION: VALUE VS. REFERENCE

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

int squareByValue(int); // prototype (value pass)
void squareByReference(int&); // prototype (reference pass)

int main()
{
   int x{2}; // value to square using squareByValue
   int z{4}; // value to square using squareByReference

   // demonstrate squareByValue
   cout << "x = " << x << " before squareByValue\n";
   cout << "Value returned by squareByValue: "
      << squareByValue(x) << endl;
   cout << "x = " << x << " after squareByValue\n" << endl;
```

C++ FUNCTION: VALUE VS. REFERENCE

```
   // demonstrate squareByReference
   cout << "z = " << z << " before squareByReference" << endl;
   squareByReference(z);
   cout << "z = " << z << " after squareByReference" << endl;
}

int squareByValue(int number)
{
   return number *= number;
}

void squareByReference(int& numberRef)
{
   numberRef *= numberRef;
}
```

C++ FUNCTION: VALUE VS. REFERENCE

Output:

x = 2 before squareByValue

Value returned by squareByValue: 4

x = 2 after squareByValue

z = 4 before squareByReference

z = 16 after squareByReference

C++: FUNCTIONS AND SCOPE

C++ FUNCTIONS AND SCOPE

- Recall: C++ allows you to use the same name for multiple identifiers
- Scope refers to the visibility (accessibility) of an identifier
- Identifiers may be:
  - global (defined outside a function or block)
  - local (defined inside a function or block)

C++ FUNCTIONS AND SCOPE

- The general rules are:
  - Global identifiers are accessible by functions/blocks as long as there are no local identifiers with the same name within the function/block
  - Local identifiers are only accessible from their point of declaration to the end of the function/block
- A local variable can be declared static, so that it retains its value when the function returns to its caller

C++ FUNCTION & SCOPE EXAMPLE

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

void useLocal();
void useStaticLocal();
void useGlobal();

int x{1}; // global variable

int main()
{
    cout << "global x in main is " << x << endl;
    int x{5}; // local variable to main

    cout << "local x in main's outer scope is " << x << endl;

    { // block starts a new scope
        int x{7}; // hides both x in outer scope and global x
        cout << "local x in main's inner scope is " << x << endl;
    }

    cout << "local x in main's outer scope is " << x << endl;
```

C++ FUNCTION & SCOPE EXAMPLE

```
    useLocal(); // useLocal has local x
    useStaticLocal(); // useStaticLocal has static local x
    useGlobal(); // useGlobal uses global x
//run one more time
    useLocal(); // useLocal reinitializes its local x
    useStaticLocal(); // static local x retains its prior value
    useGlobal(); // global x also retains its prior value

    cout << "\nlocal x in main is " << x << endl;
}

void useLocal()
{
    int x{25}; // initialized each time useLocal is called
    cout << "\nlocal x is " << x << " on entering useLocal" << endl;
    ++x;
    cout << "local x is " << x << " on exiting useLocal" << endl;
}
```

C++ FUNCTION & SCOPE EXAMPLE

```
void useStaticLocal()
{
   static int x{50}; // initialized first time useStaticLocal is called
   cout << "\nlocal static x is " << x << " on entering useStaticLocal"
      << endl;
   ++x;
   cout << "local static x is " << x << " on exiting useStaticLocal"
      << endl;
}

void useGlobal()
{
   cout << "\nglobal x is " << x << " on entering useGlobal" << endl;
   x *= 10;
   cout << "global x is " << x << " on exiting useGlobal" << endl;
}
```

```
global x in main is 1
local x in main's outer scope is 5
local x in main's inner scope is 7
local x in main's outer scope is 5

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 50 on entering useStaticLocal
local static x is 51 on exiting useStaticLocal

global x is 1 on entering useGlobal
global x is 10 on exiting useGlobal

local x is 25 on entering useLocal
local x is 26 on exiting useLocal

local static x is 51 on entering useStaticLocal
local static x is 52 on exiting useStaticLocal

global x is 10 on entering useGlobal
global x is 100 on exiting useGlobal

local x in main is 5
```

Fig. 6.11 | Scoping example. (Part 4 of 4.)

C++ SCOPE RESOLUTION OPERATOR

- The scope resolution operator :: allows us to access a global variable while a local one is in scope

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

int number{7}; // global variable named number

int main()
{
   double number{10.5}; // local variable named number

   // display values of local and global variables
   cout << "Local double value of number = " << number
      << "\nGlobal int value of number = " << ::number << endl;
}
```

C++: FUNCTION OVERLOADING AND DEFAULT PARAMETERS

C++ FUNCTION OVERLOADING

- Function overloading occurs when two or more functions share the same name

- Compiler selects proper function by examining the number, types and order of the arguments of each function

- Often used to perform the same operation on different types

C++ FUNCTION OVERLOADING

```
// Overloaded square functions.
#include <iostream>
using namespace std;

int square(int x)
{
   cout << "square of integer " << x << " is ";
   return x * x;
}

double square(double y) {
   cout << "square of double " << y << " is ";
   return y * y;
}

int main()
{
   cout << square(7); // calls int version
   cout << endl;
   cout << square(7.5); // calls double version
   cout << endl;
}
```

C++ DEFAULT PARAMETERS

- You can initialize a parameter in the first occurrence of the function name
  - This would be the prototype or header (if there is no prototype)
  - `int box (int length = 1, int width = 1, int height = 1);`

- This is used to simplify function calls
  - `box();`
  - `box(10);`
  - `box(10, 5);`
  - `box (10, 5, 2)`

- Default arguments must be the right most arguments (if all arguments are not default)

C++: RECURSION

C++ RECURSION

- A recursive function is a function that calls itself
  - May be direct (i.e. calls itself from itself)
  - May be indirect (i.e. calls itself from another function)
- Recursion can make some problems easier to understand and debug
- Recursion should not be used for performance situations
  - Recursive calls take time and use more memory
- C++ does not allow you to call main recursively

C++ RECURSION

- How it works:
  - The function actually knows how to solve the simplest case, (known as the base case)
  - If the function is called with the base case, it returns a result
  - If the function is called with a more complex case, it divides the problem into two parts
    - A part that the function can do
    - A part that the function cannot do

C++ RECURSION

- How it works (cont'd):
  - The new problem resembles the original so the function calls a copy of itself to work on the smaller problem
    - This is called a recursive call or recursion step
  - The recursion step executes while the original call is still open (i.e. has not finished executing)

C++ RECURSION - EXAMPLE

- Factorial: n! = n*(n-1)*(n-2)* …*1
  where 1! = 0! = 1

- Problem could be solved iteratively (using a loop)

- In the recursive form, we recognize that
  n! = n * (n-1)!

C++ RECURSION VS. ITERATION

- Iteration:

```
factorial = 1;
for (unsigned int count{number}; count >= 1; count--)
    factorial *= count;
```

- Recursion

```
unsigned long factorial(unsigned long number)
{
   if (number <= 1)
   { // test for base case
      return 1; // base cases: 0! = 1 and 1! = 1
   }
   else
   { // recursion step
      return number * factorial(number - 1);
   }
}
```

C++: ARRAYS AND VECTORS

C++ ARRAYS AND VECTORS

- Recall: An array is a contiguous group of memory locations that all have the same type
  - Specify a size at its point of declaration
- A vector is like an array, but allows for dynamic resizing
  - You do not have to specify a size at declaration
- Recall: first position of an array is position 0

C++ DECLARING AN ARRAY

- You can declare an array using the [] operator:
  - `int tests [5];`

- Or you can declare an array using the standard class library templates:
  - Need `#include <array> directive`
  - Format: array<type, size> name;
    - Example: `array <int, 5> tests;`

C++ ARRAY INITIALIZATION

- Standard initialization list
  - `int a[3] = {8, 1, 3};`

- Initialization list with class template
  - `array <int, 3> b {4, 2, 7};`

- C++ does not allow you to initialize with more values than elements

C++ ARRAY INITIALIZATION

- If you have fewer initializers than elements
  - Corresponding elements initialized
  - Remaining elements initialized to 0
  - Using {0} initialized all elements to 0

- In class template
  - Corresponding elements initialized
  - Remaining elements also initialized to 0
  - Using { } initializes all elements to 0

C++ ARRAYS

- Can use standard for-loop to manipulate array
  - May want to declare a constant to store the size

- With class template, you can reference the size member function
  - Format: Arrayname.size()
  - Can declare loop counter as size_t
  - size_t is an unsigned integral type used for array size or subscripts

C++ ARRAY EXAMPLE

```
#include <iostream>
#include <iomanip>            // for setw
#include <array>
using namespace std;

int main()
{
    const int SIZE = 3;
    int a[SIZE] = {8, 1, 3};

    cout << "Array a: ";
    for (int i = 0; i < SIZE; i++)
        cout << setw(5) << a[i];

    cout << endl << endl;

    array <int, 3> b {4, 2, 7};

    cout << "Array b: ";
    for (size_t i {0}; i < b.size() ; i++)
        cout << setw(5) << b[i];
    cout << endl;
}
```

Output

Array a: 8 1 3

Array b: 4 2 7

C++ 11: RANGE BASED FOR LOOP

- C++ 11 allows you to create a for loop for arrays without using a counter:
  - ```
    for (int i: a)
        cout << setw(5) << i;
    ```
- The variable *i* is now a *range variable*
  - Array name is referenced to obtain size
  - Eliminates the need for programmers to perform their own bounds checking
    ```

C++: MORE ABOUT ARRAYS

- For functions, you would pass the name and size of the array for processing
  - Could pass just the name if the size is a global constant

- For multidimensional arrays, [ ] is easier:
  - [ ]: `int table [4][3];`
  - Class template:
    `array <array <int, 3>, 4> table;`
  - Nested loops required; add another for each dimension

C++ VECTOR DECLARATION

- Need `#include <vector>` directive

- Declaration options:
  - Declare a vector to hold `int` element:
    ```
    vector<int> scores;
    ```
  - Declare a vector with initial size 30:
    ```
    vector<int> scores(30);
    ```
  - Declare a vector and initialize all elements to 0:
    ```
    vector<int> scores(30, 0);
    ```
  - Declare a vector initialized to size and contents of another vector:
    ```
    vector<int> finals(scores);
    ```

C++ VECTORS

- Initialization list with vector:
  - `vector<int> numbers { 10, 20, 30, 40 };`

- Use `push_back` member function to add element to a full vector or to a vector that had no defined size:
  - `scores.push_back(75);`

- Use `pop_back` member function to remove last element from vector:

```
scores.pop_back();
```

C++ VECTORS

- To remove all contents of a vector use clear:
  - `scores.clear();`

- Like arrays, vectors can use `size()` member function.

ASSIGNMENT DETAILS

ASSIGNMENT DETAILS

- Each .cpp file should have your name and description of the file in comments at the top of the file
  - e.g. `/* Peyman Behzadnia`
    `This program is about problem 1*/`

- Your code should be well-commented clearly showing what each part/block of program does.
- Your code MUST compile. Otherwise, you receive a significant deduction in your grade.
- Include any special instruction that TA should know to be able to compile and run your code.

- Zip .cpp files and submit via canvas.
  - Send only one zip file!

ASSIGNMENT DETAILS

- You can resubmit before the due date
  - Only the most recent one will be graded

- Remember: No late submissions!

- A partial submission is better than no submission.
  - Partial grade better than 0.