C++ Review

CSE 2020 Computer Science II

C++

- The C++ programming language will be used in this course
- This course does not teach C++ programming
  - You will use C++ to demonstrate your knowledge in this course
- The features of C++ that we may use in this course
  - Control Statements
  - Pointers
  - Arrays
  - Functions
  - Classes
  - Operator Overloading
  - Structures
  - Templates

Linux

- You will be exposed to the Linux environment
  - All the lab exercises will be marked in Linux using the **g++** compiler
- You may develop your code on Windows, but you are responsible for testing your code to lab Linux environment
- You are supposed to know basic Linux commands and text editor vim

1. Data Types

- Declare a variable

  datatype variableName = initial value;
- int x = 10;
- float f = 10.0;
- double d = 10.0;
- char c = ‘a’;
- bool lock = true;
- string class

  #include <string>

  string s = “abcd”;

2. Control Statements

- if conditional statement

```
if (condition) {
    ……
} else if (condition) {
    ……
} else {
    ……
}
```

Control Statements (cont.)

- Iteration statements
  - while

```
while (condition)
{
    ……
}
```

  - for

```
for (initialization; condition; update)
{
    ……
}
```

3. Pointers

- Every variable is stored somewhere in memory, that address is an integer
- A pointer is a variable that stores a memory address, we have to indicate what it is pointing to
  - Declare a pointer to an integer, variable “ptrInt” stores the address of an integer

```
int* ptrInt = nullptr;
char* ptrChar = nullptr;
double* ptrDouble = nullptr;
```

Pointers (cont.)

- The & operator gets the address of a variable

```
int n = 10;    // n is an int storing 10
int* ptrInt;   // a pointer to an int
ptrInt = &n;   // assign to ptrInt the address of n
cout << ptrInt << endl; // very long integer
```

- Using the * operator (dereference) to access the value pointed to by the pointer, that is what is stored at that memory location

```
cout << *ptrInt << endl;
*ptrInt = 100;    // n is changed or not?
```

4. Arrays

- A collection of similar elements stored in consecutive memory locations, random access
- Example

```
const int ARRAY_CAPACITY = 100;
// capacity can’t be a variable
int a[ARRAY_CAPACITY];
for ( int i = 0; i < ARRAY_CAPACITY; ++i )
{
     a[i] = 10 * i;
}
```

- The index of an array goes from 0 to ARRAY_CAPACITY – 1

Arrays (cont.)

- The *capacity* of an array is the maximal number of entries it can hold
- The *size* of an array is the number of useful entries
- Array name acts like a constant pointer to its first element.

```
int* ptr = a;
*(ptr + 1)
a[1]
*(a + 1)
```

- Passing an array to a function, must pass array size!

```
int Func(int a[], int size);
```

Memory Allocation

- Dynamic memory allocation in C++ is done through the *new* operator which return the address of the first byte of the memory allocated
- C++ requires the user explicitly deallocate memory
- Example

```
int x = 100;
int* ptr = new int[x];
std::cout << ptr[0] << *(ptr + 1);
delete [] ptr;
```

5. Functions

- Define a function

```
returnType functionName( parameter list )
{
   body of the function
}
```

- A function declaration or function prototype tells the compiler about a function name and how to call the function
- The actual body of the function or function implementation can be defined separately

Functions (cont.)

- For small projects, you may put all functions and main() in one file
  - Function prototypes before main(), function implementations after main()
  - OR function implementations before main()
- For big projects, you may put function prototypes in a .h file, implement functions in .cpp files, and put main() in a separated .cpp file
  - funcs.h
  - funcs.cpp include funcs.h
  - main.cpp include funcs.h

Example

```
int square( int n ){
    return n*n;
}
int main() {
    cout << "The square of 3 is " << square(3) << endl;
    return 0;
}
```

- OR

```
int square( int n ); // function prototype
int main() {…}
int square( int n ){
    return n*n;
}
```

Arguments passed to function

- by Value - copies the actual value of an argument into the formal parameter of the function. Changes made to the parameter inside the function have no effect on the argument.
- by Reference - copies the reference of an argument into the formal parameter. Inside the function, the reference is used to access the actual argument used in the call. Changes made to the parameter affect the argument.
- by Pointer - copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. Changes made to the parameter affect the argument.

Passed by Value

```
void swap(int x, int y) {
   int temp;
   temp = x; /* save the value of x */
   x = y; /* put y into x */
   y = temp; /* put x into y */
   return;
}
int main () {
   int a = 100, b = 200;
   swap(a, b);
   std::cout << << a << “, ”<< b << std::endl; // 100, 200
   return 0;
}
```

Passed by Reference

```
void swap(int &x, int &y) {
   int temp;
   temp = x; /* save the value of x */
   x = y; /* put y into x */
   y = temp; /* put x into y */
   return;
}
int main () {
   int a = 100, b = 200;
   swap(a, b);
   std::cout << << a << “, ”<< b << std::endl; // 200, 100
   return 0;
}
```

Passed by Pointer

```
void swap(int* x, int* y) {
   int temp;
   temp = *x; /* save the value of x */
   *x = *y; /* put y into x */
   *y = temp; /* put x into y */
   return;
}
int main () {
   int a = 100, b = 200;
   swap(&a, &b);
   std::cout << << a << “, ”<< b << std::endl; // 200, 100
   return 0;
}
```

File as the Unit of Compilation

- Any .cpp file may be compiled into object code .o
- Only files containing an `int main()` function can be compiled into an executable file

  The signature of main is:

```
int main () {
    // does some stuff
    return 0;
}
```

- Compile

```
$ g++ funcs.cpp -c
$ g++ main.cpp -o main
$ ./main
```

6. Classes

- Classes are fundamental to C++. It provides a way to define new user-defined types, complete with associated functions and operators.
- Names
- Attributes
  - class members, member variables
  - Access control/specifier: private
- Operations
  - methods, member functions
  - Access control: public
- *Point* class example

```
class ClassName
{
  public:
    ClassName();
    datatype getAtt1() const;
    void setAtt1(datatype);
  private:
   datatype att1;
   datatype att2;
};
```

Classes (cont.)

- Guard statements can avoid the same file being included twice, otherwise you have duplicate definitions

```
#ifndef POINT_H
#define POINT_H
class Point {
 //...
};
#endif
```

Classes (cont.)

- The class definition contains only the signatures (or *prototypes*) of the operations
- The actual member function implementations may be defined elsewhere, either in:
  - The same file, or
  - Another file which is compiled into an object file
- *Point* class and *Employee* class examples
- We will use the first method, put the class definition and actual member function implementations in the same file, .cpp.

Constructors and Destructors

- Constructors
  - have the same name as the class itself
  - no return type
  - the task is to initialize the attributes or class members
  - class may have multiple constructors
  - default constructor has no parameters
  - may use member initializer/initialization list
- Destructors
  - `~ClassName();`
  - take no parameters and have no return type
  - return the resources to the system
  - A destructor is a member function that is automatically called when a class object is done

Accessors and Mutators

- Two categories of member functions:
- Accessors
  - Access and use the class members, leaving the object unchanged
  - Add the **const** keyword after the parameter list, the compiler would signal an error when attributes are being modified in a const member function
- Mutators
  - Change or mutate the class members, modifying the member variables or attributes of the object

Objects

- An Object is an instance of a Class
- When a class is defined, only the specification for the object is defined; no memory is allocated. When the class is instantiated (i.e. an object of the class is created), memory is allocated
- To use the data and access functions defined in the class, you need to create/declare objects

```
ClassName objectName;
ClassName* ptrObjectName = new ClassName();
```

- Access data members and member functions

```
objectName.memberFunc()
ptrObjectName->memberFunc()
```

7. Overload Operators <<

- Print a variable of primitive data type, int, double, char, bool, float

```
cout << x;
```

- Print an object of class, for example, Employee

```
Employee e(1, “Bob”, “CSE”);
e.print();
```

- Overload *operator<<* so that an object can be printed like a variable of primitive data type

```
cout << e;
```

Overload Operators >>

- Read a variable of primitive data type, int, double, char, bool, float

```
cin >> x;
```

- Read an object of class, for example, Employee

```
cin >> id >> name >> dept;
Employee e(id, name, dept);
```

- Overload *`operator>>`* so that an object can be read like a variable of primitive data type

```
cin >> e;
```

Overload Operators ==, !=

- Compare two variables of primitive data type, int, double, char, bool, float

```
x == y
```

- Compare an object of class, for example, Employee

```
Employee e1(1, “Bob”, “CSE”);
Employee e2(2, “Bob”, “Math”);
```

  compare each attribute of two employees

- Overload *`operator==`* so that two objects can be compared like two variables of primitive data type

```
e1 == e2
```

Overload Operators <<, >>

- In *Employee* class

```
friend ostream &operator<<( ostream &output, const Employee &e )
{
   output << e.get_id() << " "
          << e.get_name() << " ”
          << e.get_dept();
   return output;
}
friend istream &operator>>( istream &input, Employee &e )
{
   input >> e.id >> e.name >> e.dept;
   return input;
}
```

Overload Operators ==, !=

- In *Employee* class

```
friend bool operator== (const Employee &e1, const Employee &e2)
{
   return (e1.name == e2.name && e1.id == e2.id && e1.dept == 
e2.dept);
}
friend bool operator!= (const Employee &e1, const Employee &e2)
{
    return !(e1.name == e2.name && e1.id == e2.id && e1.dept == 
e2.dept);
}
```

Using Overload Operators

- Test *Employee* class

```
Employee emps[5]; Emloyee e;
……
cin >> e; // enter id, name, dept
for (int i = 0; i < 5; i++)
   cout << emps[i] << " ";

Employee e1(1, “Bob”, “CSE”);
for (int i = 0; i < 5; i++)
   if (emps[i] == e1) return true;
```

8. Structures

- A class with public attributes

```
struct Employee {
    int id;
    string name;
    string dept;
    Employee (){id = 0; name = “”; dept = “”}
};
Employee e1;
std::cout << e1.id;
```

9. Templates

- Function templates provide a generic function for an arbitrary type T

```
int findMax(int x, int y);

double findMax(double x, double y);


template <typename T>

T findMax(T x, T y);

std::cout << findMax<int>(10, 20);

std::cout << findMax<double>(10.10, 20.20);
```

Templates (cont.)

- Class templates

```
template <typename T>
class Point{
   private:
     T x, y;
   public:
     ……
  };
Point<int> intp(1,1);
Point<double> dblp(1.0, 1.0);
Point<int> *p1 = new Point<int>(2,2);
```

Summary

- Data Types
- Control Statements
- Pointers
- Arrays
- Functions
- Classes
- Operator Overloading
- Structures
- Templates