Data Structures LAB 01

Course: Data Structures (CL2001) **Semester:** Fall 2024
Instructor: Muhammad Nouman Hanif **T.A:** Mubeen Palh

Note:

- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Revision of Previous Programming Concepts

Using Command Line arguments:

The main() in C/C++ is a very vital function. It is the main entry point for a program. There will be only one main in a program. It is generally defined with a return type of int and without parameters:

```
int main() { /* ... */ }
```

Command-line arguments are given after the name of the program in command-line shell of Operating Systems. To pass command line arguments, we typically define main() with two arguments.
The first argument is the number of command line arguments and second is list of command-line arguments.

```
int main(int argc, char *argv[]) { /* ... */ }
```

OOP Programming:

Using **Header** Files:

Header files are used for declaration. In OOP you should use header files to declare classes and functions. It will make your program look cleaner and more professional.
Header file for a class will include:

- Include guards.
- Function definition.
- Class definition.
    - Member variables
    - Function declarations (only prototype)

Implementation File will include:

- Include directive for “header.h”
- Necessary include directives.
- Function definitions for all the functions of the class.

Example:

Header File

```
// math_utils.h
// Include guard
#ifndef MATH_UTILS_H
#define MATH_UTILS_H

// Declaration of the square
function
int square(int num) {
     return num * num;
}

#endif
```

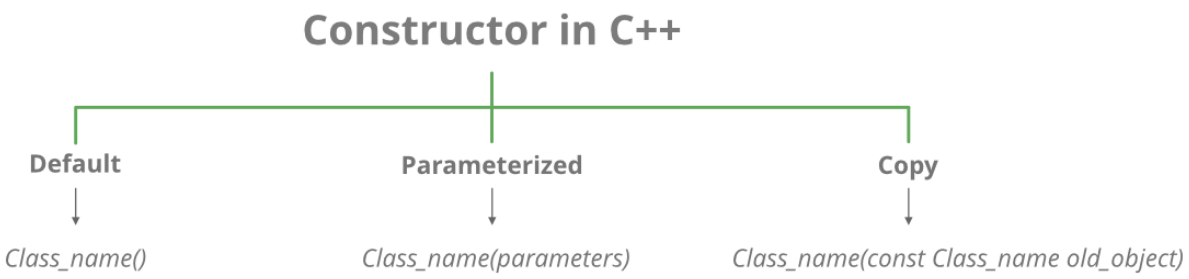
Reference CPP File

```
// math_utils_additional.cpp
#include "math_utils.h"

// Additional functions using
declarations from math_utils.h
int multiSquare(int num) {
    return square(num) * 2;
}
```

Main CPP File

```
// Header_Example.cpp
#include <iostream>
#include"math_utils.h" // Include your own header file
#include "math_utils_additional.cpp" // Include the additional cpp file

using namespace std;

int main() {
    int number = 9;
    int result = square(number);
    int doubleResult = multiSquare(number);

    cout << "The square of " << number << " is: " << result << endl;

    cout << "Multiplication with 2 of " << result << " is: " <<
doubleResult << endl;

    return 0;
}
```

Constructors and Destructors



- Default Constructor.
- Parameterized Constructor.
- Copy Constructor
- Destructors.
- Initializer List.

In object-oriented programming, constructors and destructors play crucial roles in managing the lifecycle of objects. Let's explore the different types of constructors and their significance through a relatable real-world analogy. Imagine you're at a store buying a marker. This analogy will help us understand the concept of constructors better.

Default Constructor:

When you simply ask for a marker without specifying any particular brand or color, you're essentially using a default constructor. The shopkeeper provides you with the most commonly available marker. In C++, a default constructor is responsible for creating an object with default values or initializing it without any parameters.

Parameterized Constructor:

Suppose you visit the store and specify that you want a red marker of a specific brand, say XYZ. Here, you're using a parameterized constructor. Similarly, in C++, a parameterized constructor allows you to pass specific values or parameters during object creation, customizing its properties.

Copy Constructor:

Now, imagine you're showing the shopkeeper an existing marker and asking for an exact copy of it. The shopkeeper examines the marker and gives you a new one that's identical. This scenario mirrors the concept of a copy constructor in C++, which creates a new object that's a duplicate of an existing object.

Destructors:

Just as constructors help create objects, destructors assist in their clean and proper termination. A destructor destroys the resources (memory, files, etc.) associated with an object when it's no longer needed. In our case, it's similar to discarding the marker when you're done using it. Destructors are automatically invoked when an object goes out of scope, ensuring memory is released and resources are freed. Remember, the name of a destructor is the class name preceded by a tilde (~). It's not possible to define multiple destructors for a class, and they cannot be overloaded. Destructors are called in reverse order of object creation, ensuring proper cleanup. For example, if the class is named "MyClass," the destructor's name would be "~MyClass".

Initializer List:

When initializing the data members of a class, an initializer list is used. It's like telling the shopkeeper the exact attributes you want for your marker before purchasing it. Similarly, in C++, you can use an initializer list to set initial values for the data members of a class during object creation. This can enhance efficiency and code readability.

```
class Point {
private:
    int x;
    int y;
public:
    // Constructor with initializer list
    Point(int i = 0, int j = 0):x(i), y(j) {}

/*  The above use of Initializer list is optional as
the constructor can also be written as:
        Point(int i = 0, int j = 0) {
            x = i;
            y = j;
        }
    */

    int getX() const {return x;}
    int getY() const {return y;}
};
```

➔ Dynamic Memory

C++ supports three types of memory allocation.

- ➔ **Static memory allocation** happens for **static and global variables**. Memory for these types of variables is allocated **once** when your program is run and persists throughout the life of your program.

- ➔ **Automatic memory allocation** happens for **function parameters and local variables**. Memory for these types of variables is allocated when the **relevant block** is entered, and freed when the block is exited, as many times as necessary.

- ➔ **Dynamic memory allocation** is a way for running programs to **request memory** from the operating system when needed.

new Operator

- This operator is used to allocate a memory of a particular type.
- This creates an object using the memory and **returns a pointer** containing the memory address.
- The return value is mostly stored in a **pointer** variable.

```
// new_op.cpp
int main()
{
     int *ptr = new int; // allocate memory
     *ptr = 7; // assign value

     // allocated memory and assign value
     int *ptr2 = new int(5);
}
```

delete Operator

- When we allocate memory dynamically, we need to explicitly tell C++ to deallocate this memory.
- **delete** Operator is used to release / deallocate the memory.

```
// delete_op.cpp

#include <iostream>
int main()
{
    int *ptr = new int;  // dynamically allocate an integer
    int *otherPtr = ptr; // otherPtr is now pointed at that same memory
    delete ptr;          // ptr and otherPtr are now dangling pointers.
    ptr = 0;             // ptr is now a nullptr
                       // however, otherPtr is still a dangling pointer!
   return 0;
}
```

Rule of Three:

If you need to explicitly declare either the destructor, copy constructor or copy assignment operator yourself, you probably need to explicitly declare all three of them.

The default constructors and assignment operators do shallow copy and we create our own constructor and assignment operators when we need to perform a deep copy (For example when a class contains pointers pointing to dynamically allocated resources).

First, what does a destructor do? It contains code that runs whenever an object is destroyed. Only affecting the contents of the object would be useless. An object in the process of being destroyed cannot have any changes made to it. Therefore, the destructor affects the program's state as a whole.

Now, suppose our class does not have a copy constructor. Copying an object will copy all of its data members to the target object. In this case when the object is destroyed the destructor runs twice. Also, the destructor has the same information for each object being destroyed. In the absence of an appropriately defined copy constructor, the destructor is executed twice when it should only execute once. This duplicate execution is a source for trouble.

```
class Array {
private:
    int size;
    int* vals;

public:
    ~Array();
    Array(int s, int* v);
    Array(const Array& other); // Copy constructor

    // Additional methods can be added here if needed
};

Array::~Array() {
    delete[] vals;
    vals = nullptr;
}

Array::Array(int s, int* v) {
    size = s;
    vals = new int[size];
    std::copy(v, v + size, vals);
}

Array::Array(const Array& other) {
    size = other.size;
    vals = new int[size];
    std::copy(other.vals, other.vals + size, vals);
}

int main() {
    int vals[4] = {11, 22, 33, 44};
    Array a1(4, vals);
    Array a2(a1); // Now the copy constructor is used

    return 0;
}
```

Lab Exercises

Task # 1

Using Command Line arguments to read from the file and write to the file.

Example:

```
int main () {
     ofstream myfile;
     myfile.open ("example.txt");
     myfile << "Writing this to a file.\n";
     myfile.close();
     return 0;
}

Usage: g++ -o file_copy.exe file_copy.cpp
        file_copy.exe input_file.txt output_file.txt

Hint: while (input.get(ch)) {
         output.put(ch);
       }
```

There will be 2 text files for this task input_file.txt and output_file.txt. these two file names come as some arguments to the main program. You need to open the input_file.txt for reading and output_file for writing. If the file is not created already do create a file with this name.

Now read the file character by character and write it back to output_file. After completing the entire input_file to output_file.

Task # 2

Suppose you are developing a bank account management system, and you have defined the BankAccount class with the required constructors. You need to demonstrate the use of these constructors in various scenarios.

a) Default Constructor Usage:
   Create a default-initialized BankAccount object named account1. Print out the balance of account1.
b) Parameterized Constructor Usage:
   Create a BankAccount object named account2 with an initial balance of $1000. Print out the balance of account2.

c) Copy Constructor Usage:

Using the account2 you created earlier, create a new BankAccount object named account3 using the copy constructor. Deduct $200 from account3 and print out its balance. Also, print out the balance of account2 to ensure it hasn't been affected by the transaction involving account3. Note: assume the variables in your case and print out the details.

Task # 3

Create a C++ class named "Exam" designed to manage student exam records, complete with a shallow copy implementation? Define attributes such as student name, exam date, and score within the class, and include methods to set these attributes and display exam details. As part of this exercise, intentionally omit the implementation of the copy constructor and copy assignment operator. Afterward, create an instance of the "Exam" class, generate a shallow copy, and observe any resulting issues?

Task # 4

You're tasked with designing a Document class for a document editor program. The class should handle text content, ensuring that copying a document creates a deep copy of the content to maintain data integrity. Follow the Rule of Three to manage resource allocation and deallocation correctly.
Here are the key requirements:

a) Create a constructor that takes initial text content and allocates memory for it.
b) Implement a destructor to deallocate memory used for the text content.
c) Create a copy constructor that performs a deep copy of the text content, preventing unintended sharing.
d) Create a copy assignment operator that ensures a deep copy of the text content, maintaining separation between objects.
e) Provide a sample program that showcases your Document class. Create an original document, generate copies using both the copy constructor and copy assignment operator, modify the original's content, and show that the copies remain unaffected.