**Data Structures Lab 01**

**Course:** Data Structures (CS2001) **Semester:** Fall 2022

**Instructor:** Muhammad Monis  **T.A:** N/A

**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:**

# Debugging

*Using the debugger:*

* The various features of the debugger are obvious. Click the "Debug" icon to run your program and pause at the current source code cursor location; Click "Next Line" to step through the code; Click "Add Watch" to monitor variables.
* Setting breakpoints is as easy as clicking in the blank space (Line Number) next to the line in the source code.

# → 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.
* 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:

|  |  |
| --- | --- |
| // my\_class.h  #ifndef MY\_CLASS\_H // include guard  #define MY\_CLASS\_H  namespace N  {  class my\_class  {  public:  void do\_something();  };  } | // my\_class.cpp  #include "my\_class.h" // header in local directory  #include <iostream> // header in standard library  using namespace N;  using namespace std;  void my\_class::do\_something()  {  cout << "Doing something!" << endl;  } |
| // my\_program.cpp  #include "my\_class.h"  using namespace N;  int main()  {  my\_class mc;  mc.do\_something();  return 0;  } | |

## **Constructors and Destructors**

Diagram

Description automatically generated with medium confidence

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

Let us understand the types of constructors in C++ by taking a real-world example. Suppose you went to a shop to buy a marker. When you want to buy a marker, what are the options. The first one you go to a shop and say give me a marker. So just saying give me a marker mean that you did not set which brand name and which color, you didn’t mention anything just say you want a marker. So when we said just I want a marker so whatever the frequently sold marker is there in the market or in his shop he will simply hand over that. And this is what a default constructor is! The second method is you go to a shop and say I want a marker a red in color and XYZ brand. So you are mentioning this and he will give you that marker. So in this case you have given the parameters. And this is what a parameterized constructor is! Then the third one you go to a shop and say I want a marker like this(a physical marker on your hand). So the shopkeeper will see that marker. Okay, and he will give a new marker for you. So copy of that marker. And that’s what a copy constructor is!

A destructor is also a special member function as a constructor. Destructor destroys the class objects created by the constructor. Destructor has the same name as their class name preceded by a tilde (~) symbol. It is not possible to define more than one destructor. The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded. Destructor neither requires any argument nor returns any value. It is automatically called when the object goes out of scope.  Destructors release memory space occupied by the objects created by the constructor. In destructor, objects are destroyed in the reverse of object creation.

Initializer List is used in initializing the data members of a class. The list of members to be initialized is indicated with constructor as a comma-separated list followed by a colon. Following is an example that uses the initializer list to initialize x and y of Point class.

#include<iostream>

using namespace std;

class Point {

private:

    int x;

    int y;

public:

    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; } |

# → Dynamic Arrays

To allocate an array dynamically we use array form of **new** and **delete**

(new[ ] , delete[ ])

|  |
| --- |
| dynamic\_array.cpp  // dynamic\_array.cpp  #include<iostream>  using namespace std;  int main()  {  int array[] = {1,2,3};  cout << array[0];  cout << endl;    // int\* dArray = new int[] {1,2,3};  int\* dArray = new int[3] {1,2,3};  cout << \*dArray+1;  cout << endl;  cout << dArray[2];    delete[] dArray;  } |

# →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::~Array()

{

   delete vals;

   vals = NULL;

}

Array::Array( int s, int\* v )

{

    size = s;

    vals = new int[ size ];

    std::copy( v, v + size, vals );

}

int main()

{

   int vals[ 4 ] = { 11, 22, 33, 44 };

   Array a1( 4, vals );

   // This line causes problems.

   Array a2( a1 );

   return 0;

}

## 

## 

**Lab Exercises**

**Task # 1**

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() { /\* ... \*/ }

We can also give command-line arguments in C and C++. Command-line arguments are given after the name of the program in command-line shell of Operating Systems.

C:>\ myProgram.exe I am Mr.x

To pass command line arguments, we typically define main() with two arguments : first argument is the number of command line arguments and second is list of command-line arguments.

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

or

int main(int argc, char \*\*argv) { /\* ... \*/ }

**Task # 2**

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;

}

There will be 2 text file 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

Do close the 2 files.

**Debugging in C++:**

**Consider the following Code as an example**

The following coding example illustrates how does a debugger executes the code, steps into which method’s body, how does it step out and when does it step over a function.

int a =5; int b=6; int c =3;

if( a> b && a < c){

cout << “A is the greatest of all three integers “<< a << endl;

}

else if (b> a && b > c){

cout << “B is the greatest of all three integers “<< b << endl;

}

else {

cout << “C is the greatest of all three integers << c << endl;

}

return 0;

}

**Task # 3**

The below task is to understand the Watching and Debugging the code segments.

The problem is to get a positive integer from the keyboard and find all its factors by division method. After completing all factor also produce the sum of all the factors. The program should stop when a user entered -1 as input.

Creating a watch for the iteration on factor and sum to find whether the process is correct or not.

**Task # 4**

This task is to understand the Watching and Debugging of functions and their effects.

Write the task # 3 code as function to return factors and sum. See effectively how step into a function and step out of a function works.

**Pointers:**

A pointer variable is one that contains the address, not the value, of a data object. A pointer can be declared as,

Int \*intptr;

There are few important operations, which we will do with the pointers very frequently. (a) We define a pointer variable. (b) Assign the address of a variable to a pointer. (c) Finally access the value at the address available in the pointer variable. This is done by using unary operator \* that returns the value of the variable located at the address specified by its operand. Following example makes use of these operations.

int main()

{

int var = 20; // actual variable declaration.

int \*ip; // pointer variable

ip = &var;

// store address of var in pointer variable

// print the address stored in ip pointer

//Code

// access the value at the address available in pointer

//Code

return 0;

}

**Task # 5**

Find the largest and the smallest element in an array using pointers. Create a function named MinMax having addresses of Min and Max from main function, use pointers so that it will directly affect the values of the variables defined inside the main function.

**Dynamic Allocation of Objects:**

[ Dynamically allocate memory to 5 objects of a class.]

Just like basic types, objects can be allocated dynamically, as well.

But remember, when an object is created, the constructor runs. Default constructor is invoked unless parameters are added:

Fraction \* fp1, \* fp2, \* flist;

fp1 = new Fraction; // uses default constructor

fp2 = new Fraction(3,5); // uses constructor with two parameters

flist = new Fraction[20]; // dynamic array of 20 Fraction objects

// default constructor used on each

Deallocation with delete works the same as for basic types:

delete fp1;

delete fp2;

delete [] flist;

**Notation: dot-operator vs. arrow-operator:**

dot-operator requires an object name (or effective name) on the left side

objectName.memberName // member can be data or function

The arrow operator works similarly as with structures.

pointerToObject->memberName

Remember that if you have a pointer to an object, the pointer name would have to be dereferenced first, to use the dot-operator:

(\*fp1).Show();

Arrow operator is a nice shortcut, avoiding the use or parentheses to force order of operations:

fp1->Show(); // equivalent to (\*fp1).Show();

When using dynamic allocation of objects, we use pointers, both to single object and to arrays of objects. Here's a good rule of thumb:

For pointers to single objects, arrow operator is easiest:

fp1->Show();

fp2->GetNumerator();

fp2->Input();

For dynamically allocated arrays of objects, the pointer acts as the array name, but the object "names" can be reached with the bracket operator. Arrow operator usually not needed:

flist[3].Show();

flist[5].GetNumerator();

// note that this would be INCORRECT, flist[2] is an object, not a pointer flist[2]->Show();

**Task # 6**:

Using Dynamic Array implement a Grading system for students.

1. 1st dimension will have the section ID ( 0 - A, 1 - B etc.)
2. 2nd dimension will have the number of Grades in the section.
   * 1. 0 - A (value will be the number of A grades).
     2. 1 – B
     3. 4-C etc.

**Rule of Three:**

The Rule of Three is a rule of thumb in C++ (prior to C++ 11) which states that if a class defines one or more of the following, then it should probably explicitly define all three of these. The rule is also called Law of Big Three or The Big Three, three special functions which are:

1) Destructor

2) Copy Constructor

3) Overloaded/Copy Assignment Operator.

|  |  |
| --- | --- |
| copy in Python (Deep Copy and Shallow Copy) - GeeksforGeeks | copy in Python (Deep Copy and Shallow Copy) - GeeksforGeeks |

Example:

class Numbers{

//private members:

//public members

//copy constructor

/\*

Numbers(const ClassName &obj)

{

Some code

}\*/

~Numbers() //destructor

{

delete ptr;

}

};

int main(){

int arr[ 4 ] = { 11, 22, 33, 44 };

Numbers Num1( 4, arr );

// this creates problem (if not using the copy constructor)

Numbers Num2( Num1 );

return 0;

}

**Task # 7** The task is to understand the concept of shallow and deep copies, and to understand the logic behind Rule of Three.

Implement a class for a **Car**. Implement Rule of Three for this class.

1. A car can have properties of another car. (same category cars).
2. Using copy constructor.
3. Using Assignment Operator.
4. A car object should be destroyed properly
5. Using a destructor