

**COLLEGE OF COMPUTING AND ENGINEERING** **(CCE)**

**CCS108 – Object-Oriented Programming**

**Laboratory Exercise No. 1**

***Introduction to Object-Oriented Programming***

***(with Review of Programming II and Data Structure)***

Submitted by:

|  |
| --- |
| Yocor,Angelo T |

[Name]

|  |
| --- |
| **2CS-B** |

[Section]

Submitted to:

|  |
| --- |
| **Prof. Terrence A. Lim** |

[Name of Instructor/Professor]

|  |
| --- |
|  |

**GRADE**

|  |
| --- |
| 09/013/2022 |

[Date]



***Laboratory Exercise No. 1***

**Introduction to Object-Oriented Programming**

**I. OBJECTIVES**

|  |
| --- |
| **At the end of the exercise, the students are expected to:**   * **Familiarize with the object-oriented programming approach** * **Construct a modular program applying standard control structure** * **Develop a solution to the given problems using Arrays and in a modular**   **program.** |

**II. EQUIPMENT/MATERIALS**

|  |
| --- |
| **The following equipment or materials will be needed to perform the laboratory exercise:**   * **PC with Java Compiler and IDE (Eclipse, NetBeans, jGrasp, etc.)** * **Internet Connection for Online Java Compiler/Editor and Submission** * **USB for backup and file storage** |

**III. PROCEDURE/DISCUSSION**

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Laboratory Work No. 1 Research and select one existing Procedural-Oriented and one existing Object-oriented Programming Language (except Java). Explain and discuss the characteristics of each selected programming language considering the areas to be examined as presented in the given table below:   |  |  |  | | --- | --- | --- | | **No.** | **PROCEDURAL PROGRAMMING** | **C** | | 1. | In procedural programming, program is divided into small parts called functions. | **DISCUSSION:**  Functions are a group of statements or blocks of code that together perform a task. Functions are like methods in object-oriented programming but instead of being semantically tied to a class or its objects, it is not attached to any entities. Both functions and methods could accept single or multiple parameters and could be called, to contain isolated source code but only differs in their distinct attachment. Most of the time, functions could only be present on a local or global scope depending on their implementation. These entities need to be called also which could accept arguments from their caller depending on their implementation.  **EXPLANATION:**  In a heavily reliant language on a procedural-oriented approach like C, they have a methodology of isolating specific portions of code into isolated blocks called "functions". The C standard library provides numerous built-in functions that your program can call. For example, strcat() to concatenate two strings, memcpy() to copy one memory location to another location, and many more functions. Every C program has at least one function and that is main(), and all the most trivial programs can define additional functions. A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function. In the example that is given, int addNumbers(int a, int b); is the function prototype that provides the following information to the compiler:  name of the function is addNumbers()  return type of the function is int  two arguments of type int are passed to the function  The function prototype is not needed if the user-defined function is defined before the main() function.  In the above example, the function call is made using addNumbers(n1, n2); statement inside the main() function.  When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.  In programming, argument refers to the variable passed to the function. In the above example, two variables n1 and n2 are passed during the function call.  **SOURCE CODE:**  #include <stdio.h>  // Function declaration  int checkPrime(int number);  int main() {  // Variables for validations  int userNumber;  int result;  // Prompts and stores user number  printf("Enter a positive integer $ ");  scanf("%d", &userNumber);  // userNumber is passed to the checkPrime() function for validations  result = checkPrime(userNumber);  // Prints the result of checking from checkPrime() function  if(result == 1) | | 2. | There is no access specifier in procedural programming. | **DISCUSSION:**  Access modifiers control the availability of an entity (either a variable or method) within an application. Generally, modifiers like public, private, final, and protected are the most common ones and can be found in most OOP languages. They define how the members (attributes and methods) of a class can be accessed. Modifiers control the accessibility of such code that is semantically attached to superior entities like classes and objects allowing OOP to have a more secured and robust codebase. Access modifiers play an important role in encapsulation as they can strengthen the security of a class members.  **EXPLANATION:**  In object-oriented programming, there are three access modes "public " private" and "protected" that are used as an access share to access attributes or functions. On the other hand, in procedural-oriented programming, there is no specific accessing mode to access attributes or functions in the program. Since C is a pure procedural-oriented language, there is no way we can control the availability or the accessibility of its entities like what we can do on a typical OOP-reliant language. Using such keywords will cause compile errors ahead of time as they are not part of the language's supported syntax.  **SOURCE CODE:**  #include <stdio.h>  // Function declaration  // Trying to add an access modifier  public int checkPrime(int number);  int main() {  // Variables for validations  // Trying to add access modifiers  private int userNumber;  private int result;  // Prompts and stores user number  printf("Enter a positive integer $ ");  scanf("%d", &userNumber);  // userNumber is passed to the checkPrime() function for validations  result = checkPrime(userNumber);  // Prints the result of checking from checkPrime() function  if(result == 1)  printf("%d is not a prime number.", userNumber);  else  printf("%d is a prime number.", userNumber);  // Exit successfully  return 0;  }  // int is returned from the function  // Trying to add an access modifier  public int checkPrime(int number) {  // Variable for loop  // Trying to add an access modifier  private int iterator;  // Checks if the usernumber if 0 or 1 because they are not prime numbers  if (number == 0 || number == 1)  return 1;  for(iterator = 2; iterator <= number / 2; ++iterator) {  if(number % iterator == 0)  return 1;  }  // Exit successfully  return 0;  } | | 3. | Adding new data and function is not easy. | **DISCUSSION:**  When the application scales larger and larger, introducing new entities (data and functions) is difficult. Adding so may require checking existing entities first to assure that new entities that will be added don't cause internal conflicts that may introduce strange behaviors such as errors and bugs making debugging much harder. Managing source code in a logical manner for larger applications takes a lot of time refactoring to ensure that there are no potential issues that could exist. Unlike OOP, entities like variables and methods are not semantically attached in a single unit in this approach which is one of its major drawbacks. Such drawbacks include exposed implementation details, regrouped entities of code, cluttered function calls, and many more.  **EXPLANATION:**  Since managing entities in programming languages like C lacks a more manageable or logical manner, appending new entities is difficult as we need to consider existing entities first before we add some. This makes coding in C harder as adding new variables and functions is harder. Functions are the only resort because an object-oriented approach is not available. Refactoring code in this language requires a huge amount of time to finish because developers need to consult existing entities before introducing one. This makes C paves the way for its successor, C++, to be built to provide a much more superior way of programming.  **SOURCE CODE:**  #include <stdio.h>  #include <time.h>  #include <stdlib.h>  #include <conio.h>  #include<time.h>  #include<ctype.h>  #include <time.h>  #include <windows.h>  #include <process.h>  #define UP 72  #define DOWN 80  #define LEFT 75  #define RIGHT 77  int length;  int bend\_no;  int len;  char key;  void record();  void load();  int life;  void Delay(long double);  void Move();  void Food();  int Score();  void Print();  void gotoxy(int x, int y);  void GotoXY(int x,int y);  void Bend();  void Boarder();  void Down();  void Left();  void Up();  void Right();  void ExitGame();  int Scoreonly();  struct coordinate  {  int x;  int y;  int direction;  };  typedef struct coordinate coordinate;  coordinate head, bend[500],food,body[30];  int main()  {  char key;  Print();  system("cls");  load();  length=5;  head.x=25;  head.y=20;  head.direction=RIGHT;  Boarder();  Food(); //to generate food coordinates initially  life=3; //number of extra lives  bend[0]=head;  Move(); //initialing initial bend coordinate  return 0;  }  void Move()  {  int a,i;  do  {  Food();  fflush(stdin);  len=0;  for(i=0; i<30; i++)  {  body[i].x=0;  body[i].y=0;  if(i==length)  break;  }  Delay(length);  Boarder();  if(head.direction==RIGHT)  Right();  else if(head.direction==LEFT)  Left();  else if(head.direction==DOWN)  Down();  else if(head.direction==UP)  Up();  ExitGame();  }  while(!kbhit());  a=getch();  if(a==27)  {  system("cls");  exit(0);  }  key=getch();  if((key==RIGHT&&head.direction!=LEFT&&head.direction!=RIGHT)||(key==LEFT&&head.direction!=RIGHT&&head.direction!=LEFT)||(key==UP&&head.direction!=DOWN&&head.direction!=UP)||(key==DOWN&&head.direction!=UP&&head.direction!=DOWN))  {  bend\_no++;  bend[bend\_no]=head;  head.direction=key;  if(key==UP)  head.y--;  if(key==DOWN)  head.y++;  if(key==RIGHT)  head.x++;  if(key==LEFT)  head.x--;  Move();  }  else if(key==27)  {  system("cls");  exit(0);  }  else  {  printf("\a");  Move();  }  }  void gotoxy(int x, int y)  {  COORD coord;  coord.X = x;  coord.Y = y;  SetConsoleCursorPosition(GetStdHandle(STD\_OUTPUT\_HANDLE), coord);  }  void GotoXY(int x, int y)  {  HANDLE a;  COORD b;  fflush(stdout);  b.X = x;  b.Y = y;  a = GetStdHandle(STD\_OUTPUT\_HANDLE);  SetConsoleCursorPosition(a,b);  }  void load()  {  int row,col,r,c,q;  gotoxy(36,14);  printf("loading...");  gotoxy(30,15);  for(r=1; r<=20; r++)  {  for(q=0; q<=100000000; q++); //to display the character slowly  printf("%c",177);  }  getch();  }  void Down()  {  int i;  for(i=0; i<=(head.y-bend[bend\_no].y)&&len<length; i++)  {  GotoXY(head.x,head.y-i);  {  if(len==0)  printf("v");  else  printf("\*");  }  body[len].x=head.x;  body[len].y=head.y-i;  len++;  }  Bend();  if(!kbhit())  head.y++;  }  void Delay(long double k)  {  Score();  long double i;  for(i=0; i<=(10000000); i++);  }  void ExitGame()  {  int i,check=0;  for(i=4; i<length; i++) //starts with 4 because it needs minimum 4 element to touch its own body  {  if(body[0].x==body[i].x&&body[0].y==body[i].y)  {  check++; //check's value increases as the coordinates of head is equal to any other body coordinate  }  if(i==length||check!=0)  break;  }  if(head.x<=10||head.x>=70||head.y<=10||head.y>=30||check!=0)  {  life--;  if(life>=0)  {  head.x=25;  head.y=20;  bend\_no=0;  head.direction=RIGHT;  Move();  }  else  {  system("cls");  printf("All lives completed\nBetter Luck Next Time!!!\nPress any key to quit the game\n");  record();  exit(0);  }  }  }  void Food()  {  if(head.x==food.x&&head.y==food.y)  {  length++;  time\_t a;  a=time(0);  srand(a);  food.x=rand()%70;  if(food.x<=10)  food.x+=11;  food.y=rand()%30;  if(food.y<=10)  food.y+=11;  }  else if(food.x==0)/\*to create food for the first time coz global variable are initialized with 0\*/  {  food.x=rand()%70;  if(food.x<=10)  food.x+=11;  food.y=rand()%30;  if(food.y<=10)  food.y+=11;  }  }  void Left()  {  int i;  for(i=0; i<=(bend[bend\_no].x-head.x)&&len<length; i++)  {  GotoXY((head.x+i),head.y);  {  if(len==0)  printf("<");  else  printf("\*");  }  body[len].x=head.x+i;  body[len].y=head.y;  len++;  }  Bend();  if(!kbhit())  head.x--;  }  void Right()  {  int i;  for(i=0; i<=(head.x-bend[bend\_no].x)&&len<length; i++)  {  //GotoXY((head.x-i),head.y);  body[len].x=head.x-i;  body[len].y=head.y;  GotoXY(body[len].x,body[len].y);  {  if(len==0)  printf(">");  else  printf("\*");  }  /\*body[len].x=head.x-i;  body[len].y=head.y;\*/  len++;  }  Bend();  if(!kbhit())  head.x++;  }  void Bend()  {  int i,j,diff;  for(i=bend\_no; i>=0&&len<length; i--)  {  if(bend[i].x==bend[i-1].x)  {  diff=bend[i].y-bend[i-1].y;  if(diff<0)  for(j=1; j<=(-diff); j++)  {  body[len].x=bend[i].x;  body[len].y=bend[i].y+j;  GotoXY(body[len].x,body[len].y);  printf("\*");  len++;  if(len==length)  break;  }  else if(diff>0)  for(j=1; j<=diff; j++)  {  /\*GotoXY(bend[i].x,(bend[i].y-j));  printf("\*");\*/  body[len].x=bend[i].x;  body[len].y=bend[i].y-j;  GotoXY(body[len].x,body[len].y);  printf("\*");  len++;  if(len==length)  break;  }  }  else if(bend[i].y==bend[i-1].y)  {  diff=bend[i].x-bend[i-1].x;  if(diff<0)  for(j=1; j<=(-diff)&&len<length; j++)  {  /\*GotoXY((bend[i].x+j),bend[i].y);  printf("\*");\*/  body[len].x=bend[i].x+j;  body[len].y=bend[i].y;  GotoXY(body[len].x,body[len].y);  printf("\*");  len++;  if(len==length)  break;  }  else if(diff>0)  for(j=1; j<=diff&&len<length; j++)  {  /\*GotoXY((bend[i].x-j),bend[i].y);  printf("\*");\*/  body[len].x=bend[i].x-j;  body[len].y=bend[i].y;  GotoXY(body[len].x,body[len].y);  printf("\*");  len++;  if(len==length)  break;  }  }  }  }  void Boarder() | | 4. | Procedural programming does not have any proper way for hiding data so it is less secure. | **DISCUSSION:**  One of the primary concerns of object-oriented programming is data security as it binds the data closely to the functions which operate on it. Encapsulation is one of the pillars of object-oriented programming which states that entities must be hidden and bundled into more superior entities (class and objects). Since the members are bonded, their implementation details are also hidden making them more secure and easier to use in which another benefit of abstraction which is another OOP pillar. Since properties and methods are hidden, they are not easily accessed for improper usage.  **EXPLANATION:**  One of the pitfalls of procedural languages is their lack of proper data hiding practices which are provided by encapsulation and abstraction. C lacks the main benefits of encapsulation and abstraction since it is purely procedural. Data is exposed to the whole program, so there is no security for data. Data security is at risk as data freely moves in the program, as well as code reusability is not achieved which makes the programming lengthy and hard to understand. In addition, a lack of data hiding can give illegal hackers an easier means of breaching the application it follows old-school coding standards.  **SOURCE CODE:**  person.h  #ifndef \_\_\_PERSON\_\_H\_\_  #define \_\_\_PERSON\_\_H\_\_  struct Person;  typedef struct Person \*pPerson;  pPerson AddName(char \*firstname, char \*lastname, int age);  void PrintName(pPerson);  #endif  person.c  #include <stdio.h>  #include <string.h>  #include <stdlib.h>  #include "person.h"  typedef struct sPerson\_t  {  char firstname[32];  char lastname[32];  int age;  } sPerson;  pPerson AddName(char \*firstname, char \*lastname, int age) {  sPerson \*c = (sPerson \*) malloc(sizeof(sPerson)); malloc(sizeof(sPerson));  sPerson per;  strcpy(c->firstname, firstname);  strcpy(c->lastname, lastname);  c->age = age;  per = (pPerson) c;  return per;  }  void PrintName(pPerson pers) {  sPerson \*per = (sPerson \*) pers;  printf("Lastname : %s\n", c->lastname);  printf("Firstname : %s\n", c->firstname);  printf("Age : %s\n", c->age);  }  main.c  #include <stdio.h>  #include "person.h"  int main() {  pPerson dude = AddName("John", "Doe", 28);  printf("## DUDE ##\n");  PrintName(dude);  return 0;  } | | 5. | In procedural programming, overloading is not possible. | **DISCUSSION:**  Polymorphism, on the other hand, is one of the pillars of object-oriented programming which allows constructors of classes to have multiple variants based on their required number and type of parameter(s). Multiple implementations of constructors make classes flexible as they introduce numerous approaches to using them making their usability much more convenient. As the words themselves, polymorphism of "multiple forms" of a class can be by creating multiple implementations of its constructors. Also, polymorphism controls the "cleanliness" of a constructor as it can declutter numerous parameters it can receive when it is called which is a way for better readability and scalability.  **EXPLANATION:**  Since object-oriented programming is not an available paradigm to C, method overloading as benefits of polymorphism is impossible. Doing so could cause compile errors ahead of time from the compiler as they are not part of the language's supported syntax as it's reserved keywords.  **SOURCE CODE:**  #include <stdio.h>  // Function declaration  int checkPrime(int number);  // Trying to make an overload  int checkPrime(int firstNumber, int secondNumber);  int main() {  // Variables for validations  int userNumber;  int result;  // Prompts and stores user number  printf("Enter a positive integer $ ");  scanf("%d", &userNumber);  // userNumber is passed to the checkPrime() function for validations  result = checkPrime(userNumber);  // Prints the result of checking from checkPrime() function  if(result == 1)  printf("%d is not a prime number.", userNumber);  else  printf("%d is a prime number.", userNumber);  // Exit successfully  return 0;  }  // int is returned from the function  int checkPrime(int number) {  // Variable for loop  int iterator;  // Checks if the usernumber if 0 or 1 because they are not prime numbers  if (number == 0 || number == 1)  return 1;  for(iterator = 2; iterator <= number / 2; ++iterator) {  if(number % iterator == 0)  return 1;  }  // Exit successfully  return 0;  } | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **No.** | **OBJECT-ORIENTED PROGRAMMING** | **C++** | | 1. | In object-oriented programming, program is divided into small parts called objects. | **DISCUSSION:**  In object-oriented programming, different portions of code are grouped together in an entity called an object. Compared to functions, objects are much more dynamic in terms of their functionality as their internals has not only data but methods. With this capability, source codes are grouped together in a much more "related" manner making addressing problems much more modular. One great example of a language for object-oriented programming is C++. The classes and objects in C++, along with their properties and functions, are the foundation of everything.  **EXPLANATION:**  For example, a car is an object in the real world. The car has methods like drive and brake, as well as attributes like weight and color. Attributes and methods are essentially variables and functions that belong to the class. A class is a user-defined data type that we may utilize in our program and acts as an object constructor or a "blueprint" for constructing objects. These are frequently referred to as "class members." In order to create an object, we need to initialize a class. Like in the source code we have set it as sampleClass. After setting the class we proceed to the main and initialize it. We can set it by simply putting the class name as the constructor and we can name it as an object, in our case we have named it as objSampleClass.  **SOURCE CODE:**  #include <iostream>  using namespace std;  //the class or the blueprint  class sampleClass {  public:  int num = 5;  };  int main() {  //when initialized this can now be called a object  sampleClass objSampleClass;  cout << objSampleClass.myNum;  return 0;  } | | 2. | Object-oriented programming has access specifiers like private, public, protected etc. | **DISCUSSION**  Access modifiers or sometimes referred to as access specifiers in an OOP approach are used to assign accessibility of members (properties and methods) of the classes. They set certain restrictions on the members of the classes so that they cannot be accessed directly by external functions. In this regard, OOP languages provide great capability in terms of security as members are attached to superior entities (classes and objects) as their accessibility is controlled which is a part of encapsulation.  C++ general access modifiers  private - From outside the class, members cannot be accessed (or viewed).  public - members can be reached outside of the class.  protected - members cannot be accessed from outside the class, however, they can be accessed in inherited classes as access specifiers  **EXPLANATION**  Access modifiers help us implement encapsulation or information hiding where they tell the compiler which other classes should have access to the field or method is defined. Private, only the current class will have access to the field or method. But they're not allowed to be accessed directly in any object/function outside. Only the member function or a friend function will be allowed to access the private data. Public, any class can refer to the field or call the method. And will be accessible by classes and functions anywhere in the program using the direct member access operator (.) with the object of that class. Protected, only the current class and subclass and sometimes also the same package classes of this class will have access to the field or method. It is similar to a private access modifier wherein it can't access outside of its class unless with the help of a friend class.  **SOURCE CODE:**  **// C++ program to demonstrate private**  **// access modifier**  #include<iostream>  using namespace std;  class Circle  {  // private data member  private:  double radius;  // public member function  public:  void compute\_area(double r)  { // member function can access private  // data member radius  radius = r;  double area = 3.14\*radius\*radius;  cout << "Radius is: " << radius << endl;  cout << "Area is: " << area;  }  };  // main function  int main()  {  // creating object of the class  Circle obj;  // trying to access private data member  // directly outside the class  obj.compute\_area(1.5);  return 0;  }  **// C++ program to demonstrate public**  **// access modifier**  #include<iostream>  using namespace std;  // class definition  class Circle  {  public:  double radius;  double compute\_area()  {  return 3.14\*radius\*radius;  }  };  // main function  int main()  {  Circle obj;  // accessing public datamember outside class  obj.radius = 5.5;  cout << "Radius is: " << obj.radius << "\n";  cout << "Area is: " << obj.compute\_area();  return 0;  }  **// C++ program to demonstrate**  **// protected access modifier**  #include <bits/stdc++.h>  using namespace std;  // base class  class Parent  {  // protected data members  protected:  int id\_protected;  };  // sub class or derived class from public base class  class Child : public Parent  {  public:  void setId(int id)  {  // Child class is able to access the inherited  // protected data members of base class  id\_protected = id;  }  void displayId()  {  cout << "id\_protected is: " << id\_protected << endl;  }  };  // main function  int main() {  Child obj1;  // member function of the derived class can  // access the protected data members of the base class  obj1.setId(81);  obj1.displayId();  return 0;  } | | 3. | Adding new data and function is easy. | **DISCUSSION:**  When an application upscales larger, adding new data and functions must be easy. In OOP, these concern is only minor as it provides a mechanism for arranging a code in a logical and easier manner. Since communication between objects is only be done by its methods, possible external conflicts could become easier to address. Also, related codes are isolated in a single unit which can also make them easier to use. The debugging process in procedural languages is very difficult and frustrating while debugging is very easy in object-oriented languages. Also, object-oriented programming supports much easier reuse of cod than the procedural-oriented programming paradigm.  **EXPLANATION:**  Good to know, C++ makes introducing new data and functions easier due also to its higher level syntax compared to C. Adding new source code is easier as entities are already isolated which can avoid the possible existence of conflicts. This benefits developer assessing concerns as they can fully utilize this capability to create their own way of managing things to solve problems. Also, this makes a portion of code become reusable and modular as they can become more abstract depending on their implementation. C++ has this great functionality right at its internals.  **SOURCE CODE:**  #include <iostream>  #include <stdlib.h>  #include <unistd.h>  #include <time.h>  using namespace std;  int createZombie() {  if (rand() % 67 < 10)  return 11;  else  return rand() % 10 + 1;  }  int main() {  srand(time(NULL));  char enter;  // game stats  int playerAlive = true;  int playerSkill = 9;  int playerScore = 1;  string playerName = "";  int zombieCount = 0;  int zombiesKilled = 0;  // title  cout << "Welcome to Zombie War." << endl << "Press [ENTER] to start.";  cin.get();  // player name  cout << "Please enter your name: ";  cin >> playerName;  // ask how many zombies  cout << "How many zombies do you wish to fight? ";  cin >> zombieCount;  cout << "Get ready to fight for your life, " << playerName << "!" << endl;  // main game loop  while (playerAlive && zombiesKilled < zombieCount) {  // create a random zombie  int zombieSkill = createZombie();  // battle sequence  if (zombieSkill > 10) {  cout << endl << "Here comes a huge zombie!" << endl;  } else {  cout << endl << "Here comes zombie " << zombiesKilled + 1 << endl;  }  cout << "Fighting..." << endl;  sleep(2);  // zombie killed the player  if (playerSkill < zombieSkill) {  playerAlive = false;  cout << "You have died." << endl;  } else {  if (playerSkill - zombieSkill > 7) {  cout << "You wasted the zombie!" << endl;  playerScore = playerScore \* 2;  }  else if (playerSkill - zombieSkill > 5) {  cout << "You decapitated the zombie!" << endl;  playerScore = playerScore \* 2;  }  else if (playerSkill - zombieSkill > 0) {  cout << "You killed the zombie!" << endl;  playerScore = playerScore \* 2;  }  else {  cout << "You killed the zombie, but suffered injuries." << endl;  }  zombiesKilled++;  }  cout << endl;  sleep(1);  }  // end game  if (zombiesKilled == zombieCount) {  // victory  cout << "You have survived the onslaught!" << endl;  }  else {  // lost  cout << "You did not survive the zombie war." << endl;  }  cout << "Zombies killed: " << zombiesKilled << endl;  cout << "Final score: " << playerScore << endl << endl;  } | | 4. | Object-oriented programming provides data hiding so it is more secure. | **DISCUSSION**  Data hiding sometimes referred to as data security data encapsulation or information hiding, is a software development technique specifically used in object-oriented programming (OOP) to hide internal object details (data members) which follows one of the pillars of OOP, encapsulation. Data hiding ensures exclusive access to class members and protects object integrity by preventing unintended or intended changes. Data hiding also reduces system complexity for increased robustness by limiting interdependencies between software components. It is a double-edged sword as it provides a mechanism provided by encapsulation as members are bound into a larger entity and abstraction as member's implementation details are hidden.  E**XPLANATION**  C++ has the capability of implementing data hiding compared to C. Data hiding was introduced as part of the OOP methodology, in which a program is segregated into objects with specific data and functions. This technique enhances a programmer’s ability to create classes with unique data sets and functions, avoiding unnecessary penetration from other program classes. Because software architecture techniques rarely differ, there are few data hiding contradictions. Data hiding only hides class data components, whereas data encapsulation hides class data parts and private methods. Implementing data hiding in C++ is easy due to the high-level syntax of this language.  SAMPLE CODE  #include<iostream>  using namespace std;  class Encapsulation  {  private:  // data hidden from outside world  int num;  public:  // function to set value of  // variable x  void set(int a) {  num = a;  }  // function to return value of  // variable x  int get() {  return num;  }  };  // main function  int main() {  Encapsulation obj;  obj.set(5);  cout<<obj.get();  return 0;  } | | 5. | Overloading is possible in object-oriented programming. | **DISCUSSION**  Function overloading is a feature of object-oriented programming where two or more functions can have the same name but different parameters. When a function name is overloaded with different jobs it is called Function Overloading. In Function Overloading “Function” name should be the same and the arguments should be different.  **EXPLANATION**  Polymorphism allows constructors of classes to have multiple varieties based on their required number and type of parameter(s). Multiple implementations of constructors make classes flexible as they introduce numerous approaches to using them making their usability much more convenient. This method is called function overloading which is a feature of object-oriented programming where a class constructor can have the same name but different parameters. In function overloading, the constructor name should be the same and the number and types of arguments should be different.  **EXPLANATION**  Since C++ follows an object-oriented approach to programming, it lets you specify more than one constructor implementation of the same name in the same scope. These new variants of constructors are called overloaded functions or simply overloads. Overloaded functions enable you to supply different semantics for a function, depending on the types and number of its arguments. To create a overloading first create a method with the same name bat different in parameter. The example below is we create a method called add, but in the first method the parameter is n int while the second method is double.  SOURCE CODE:  #include<iostream>  using namespace std;  void add(int a,int b) {  cout << "sum=" <<(a + b);  }  void add(double a,double b) {  cout << "sum=" <<(a + b);  }  int main() {  add(10,2);  add(5.3,6.2);  return 0;  } |  Laboratory Work No. 2 Construct a Java code that will provide a computation of both Harmonic and Geometric Series of n - 1.  In mathematics, the Harmonic Series has the sum of reciprocals from positive integers, for example:  harmonic series  On the other hand, the Geometric Series has the sum from a constant ratio between successive terms like:    Save your program as CCS108LabExer1-2.java with the given Function Name for each task done using Recursion and format the result with two decimal places only.   1. **Function Name: *harmonicSum()***   **Sample Run:**  Enter a Positive Integer: 7  The Harmonic Sum is: 2.59   1. **Function Name:**   **Sample Run:**  Enter a Positive Integer: 7  The Geometric Sum is: 1.99 |

|  |  |  |  |
| --- | --- | --- | --- |
| Laboratory Work No. 3 Construct a Java program that will employ a user-inputted Array with the following tasks using a function and either While or For Structure:   1. **Function Name: *inputArray()***  * This will allow the user to input the elements of the array named *“myArray”* based on its length or size he/she entered.  1. **Function Name: *displayArray()***  * This will print the elements of the array done in the *inputArray()* function.  1. **Function Name: *sumArray()***  * This will display the sum or total of all the inputted elements of the array.  1. **Function Name: *averageArray()***  * This will output the average of the elements based on entered length or size of the array  1. **Function Name: o*ddArray()***  * This will display the elements that are odd numbers.  1. **Function Name: e*venArray()***  * This will display the elements that are even numbers.  1. **Function Name: *divbyfiveArray()***  * This will display the elements that are divisible by five.  1. **Function Name: *highestArray()***  * This will display the highest value element within the array.  1. **Function Name: *lowestArray()***  * This will display the lowest value integer within the array.   Then, save your program as CCS108LabExer1-3.  **Sample Run:**  Enter Size of the Array: 5  Enter the 5 Elements:  Index[0] : 10  Index[1] : 9  Index[2] : 8  Index[3] : 7  Index[4] : 6  The Sum of All Elements is: 40  The Average of the Elements is: 8  The Odd Number/s: 9 7  The Even Number/s: 10 8 6  The Number/s Divisible by Five: 10  The Highest Element: 10  The Lowest Element: 6   |  | | --- | | **Links for covida:** Laboratory Work No. 2 Harmonic Sum: <https://www.codiva.io/p/ea1d53d1-84c7-45ef-9576-4e4e876abbb2>  Geometric Sum: <https://www.codiva.io/p/4c7d4a67-1b88-4520-a61d-3d3230b34390> Laboratory Work No. 3- <https://www.codiva.io/p/10228f40-ed84-4dfc-8ba3-70c7c51a7179> |   **IV. DATA REPRESENTATION / OUTPUT PICTURES**  ***Laboratory Work No.2***  **HarmonicSum()**    **Geometric Sum()**   *Laboratory Work No. 3*       **V. RESULTS INTERPRETATION/OBSERVATION**  ***Laboratory Work No.1***  in laboratory 1 the task is to differentiate the procedure to the object-oriented programming, the different characteristic of two. I choose c language for procedural programming and c++ for object-oriented programming. C++ have unique characteristic like adding a data are not easy because adding so may require checking existing entities first to assure that new entities that will be added don't cause internal conflicts that may introduce strange behaviours such as errors and bugs making debugging much harder while in c++ is easy to add new data because In OOP, these concern is only minor as it provides a mechanism for arranging a code in a logical and easier manner. Since communication between objects is only be done by its methods, possible external conflicts could become easier to address. Also, related codes are isolated in a single unit which can also make them easier to use. And so many more different in c++ and c language.  ***Laboratory Work No.2***  Harmonic Sum()  In our code is first we get the input of the user by using a BufferedReader named reader then we declare an int to store there the input of user, the next step is creating a method called harmonicSum and inside of that is we use while loop, then we create a int data type called harmonic to store the sum of all, we sum the harmonic it self and divided it by the user’s input to get the harmonic sum. The final steps is to round the decimal point into 2 places. To do that is first we need a class called DecimalFormat and create a string data type to holds the rounded sum. And the final is printing or displaying the answer.  **Geometric Sum()**  **in our code in geometric sum is first we declare a 4 static int named number for 1st number, ratio for getting the ratio, and number\_of\_terms for the terms then create a method called geometricSum that do a for loop to add the result and number and nu=number to multiply to ratio. Then return a int result.in main method I called the mehod geometricSum**  *Laboratory Work No. 3*  In laboratory work 3 is first declare 3 static int that are called size, size2 and myArray and declare a static BufferReader that I called reader. Then create I create a methods, first a create a method called input array use to input a size of myArray by asking a the input,then a created a method voiddisplayArray that ask the user to input a value inside of myArray ,then I created a method called sumarray to sum up all of the value by using a for loop,then I created a method called averageArray that compute the average by sum up all of the values then devided it by the size of the myArray,then I created a method called oddarray that filters the odd by using a for loop and inside of for loop iss a if statement that checks if the myArray[i]%2 is equal to 1,next is I created a method called evenarray that filters the values of the array if it is a even by using for loop and inside is a if statement that check if it is even, then next is a method called divided by five that filter the array by visible by 5,I use for loop to access the array and if statement if the values is devided by 5 is equal to 0 then we print it, then I created a method called highestarray return a value of highest value for that I use a nested loop for swapping the values to lowest to highest. Then I called the last index by subtracting 1 to the size of the array, then I create a method called lowestarray that display the lowest value. Same as the logic of arranging the value from lowest to highest value in method highestArray after that is I print the value in index 0 to display the lowest. and the last method that I created is method displayResult, this method I for displaying or printing the result of all methods.  **VI. CONCLUSIONS**  The things that I learned in the laboratory work numbers 2 and 3 is how to use the static, public, and void. Static is a declaration use to don’t recognize a data type to object ,public is used in declaring a data type it is use in methods to access it globally and void for no return or do not return any value.one thing that I learn is that we can chop a block of code into small methods, it is use to be organize, clean and efficient to the programmers and users of the code.  **VII. STUDENT OUTCOMES ADDRESSED**   |  | | --- | | ***(… to fill out by your instructor)*** |   **VIII. APPENDICES**   1. **RUBRICS AND SCORING**  |  | | --- | | ***(… kindly refer to rubrics and scoring provided)*** | |