28. DATA ENCAPSULATION

All C++ programs are composed of the following two fundamental elements: • **Program statements (code):** This is the part of a program that performs actions and they are called functions. • **Program data:** The data is the information of the program which gets affected by the program functions. Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**. **Data encapsulation** is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user. C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public** members. By default, all items defined in a class are private. For example: `class Box { public: double getVolume(void) { return length * breadth * height; } private: double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box };` The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved. To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or

functions

defined after the public specifier are accessible by all other functions in your program. Making one class a friend of another, exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible. **Data Encapsulation Example** Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example: `#include <iostream> using namespace std; class Adder{ public: // constructor Adder(int i = 0) { total = i; } // interface to outside world void addNum(int number) { total += number; } // interface to outside world int getTotal() { return total; }; private: // hidden data from outside world int total; }; int main( )`

`{ Adder a; a.addNum(10); a.addNum(20); a.addNum(30); cout << "Total " << a.getTotal() <<endl; return 0; }` When the above code is compiled and executed, it produces the following result: `Total 60` Above class adds numbers together, and returns the sum. The public members - **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly. **Designing Strategy** Most of us have learnt to make

class members private by default unless we really need to expose them. That's just good **encapsulation**. This is applied most frequently to data members, but it applies equally to all members, including virtual functions.

29. INTERFACES

An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class. The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data. A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration as follows: `class Box { public: // pure virtual function virtual double getVolume() = 0; private: double length; // Length of a box double breadth; // Breadth of a box double height; // Height of a box };` The purpose of an **abstract class** (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit. Abstract classes cannot be used to instantiate objects and serves only as an **interface**. Attempting to instantiate an object of an abstract class causes a compilation error. Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC. Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error. Classes that can be used to instantiate objects are called **concrete classes**. **Abstract Class Example** Consider the following example where parent class provides an interface to the base class to implement a

function called **getArea()**: `#include <iostream>`

C++
241
```
using namespace std; // Base class class Shape { public: // pure virtual function providing interface framework. virtual int getArea() = 0; void setWidth(int w) { width = w; } void setHeight(int h) { height = h; } protected: int width; int height; }; // Derived classes class Rectangle: public Shape { public: int getArea() { return (width * height); } }; class Triangle: public Shape {
```
C++
242
```
public: int getArea() { return (width * height)/2; } }; int main(void) { Rectangle Rect; Triangle Tri; Rect.setWidth(5); Rect.setHeight(7); // Print the area of the object. cout << "Total Rectangle area: " << Rect.getArea() <<
```

endl; Tri.setWidth(5); Tri.setHeight(7); // Print the area of the object. cout << "Total Triangle area: " << Tri.getArea() << endl; return 0; } When the above code is compiled and executed, it produces the following result: Total Rectangle area: 35 Total Triangle area: 17 You can see how an abstract class defined an interface in terms of getArea() and two other classes implemented same function but with different algorithm to calculate the area specific to the shape. **Designing Strategy** An object-oriented system might use an abstract base class to provide a common and standardized interface appropriate for all the external applications.

Then, through inheritance from that abstract base class, derived classes are formed that operate similarly. The capabilities (i.e., the public functions) offered by the external applications are provided as pure virtual functions in the abstract base class. The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of the application. This architecture also allows new applications to be added to a system easily, even after the system has been defined.

30. FILES AND STREAMS

So far, we have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively. This tutorial will teach you how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types: **Data Type Description** ofstream This data type represents the output file stream and is used to create files and to write information to files. ifstream This data type represents the input file stream and is used to read information from files. fstream This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files. To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file. **Opening a File** A file must be opened before you can read from it or write to it. Either **ofstream** or **fstream** object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only. Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects. void open(const char *filename, ios::openmode mode); Here, the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file

should be opened.

245

Mode Flag Description ios::app Append mode. All output to that file to be appended to the end. ios::ate Open a file for output and move the read/write control to the end of the file. ios::in Open a file for reading. ios::out Open a file for writing. ios::trunc If the file already exists, its contents will be truncated before opening the file. You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax: `ofstream outfile; outfile.open("file.dat", ios::out | ios::trunc );` Similar way, you can open a file for reading and writing purpose as follows: `fstream afile; afile.open("file.dat", ios::out | ios::in );` **Closing a File** When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination. Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects. `void close();`

C++

246

Writing to a File While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object. **Reading from** You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object. **Read & Write Example** Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen: `#include <fstream> #include <iostream> using namespace std; int main () { char data[100]; // open a file in write mode. ofstream outfile; outfile.open("afile.dat"); cout << "Writing to the file" << endl; cout << "Enter your name: "; cin.getline(data, 100); // write inputted data into the file. outfile << data << endl;`

C++

247

`cout << "Enter your age: "; cin >> data; cin.ignore(); // again write inputted data into the file. outfile << data << endl; // close the opened file. outfile.close(); // open a file in read mode. ifstream infile; infile.open("afile.dat"); cout << "Reading from the file" << endl; infile >> data; // write the data at the screen. cout << data << endl; // again read the data from the file and display it. infile >> data; cout << data << endl; // close the opened file. infile.close(); return 0; }` When the above code is compiled and executed, it produces the following sample input and output: `$./a.out Writing to the file`

C++

248

`Enter your name: Zara Enter your age: 9 Reading from the file Zara 9` Above examples make use of additional functions from cin object, like getline() function to

read the line from outside, and ignore() function to ignore the extra characters left by previous read statement. **File Position Pointers** Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream. The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream. The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are: `// position to the nth byte of fileObject (assumes ios::beg) fileObject.seekg(n); // position n bytes forward in fileObject fileObject.seekg(n, ios::cur); // position n bytes back from end of fileObject fileObject.seekg(n, ios::end); // position at end of fileObject fileObject.seekg(0, ios::end);`

31. EXCEPTION HANDLING

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**. • **throw:** A program throws an exception when a problem shows up. This is done using a **throw** keyword. • **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception. • **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It is followed by one or more catch blocks. Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch is as follows: `try { // protected code }catch( ExceptionName e1 ) { // catch block }catch( ExceptionName e2 ) { // catch block }catch( ExceptionName eN ) { // catch block }` You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block

raises more than one exception in different situations.

250

Throwing Exceptions Exceptions can be thrown anywhere within a code block using **throw** statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown. Following is an example of throwing an exception when dividing by zero condition occurs: `double division(int a, int b) { if( b == 0 ) { throw "Division by zero condition!"; } return (a/b); }` **Catching** The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch. `try { // protected code }catch( ExceptionName e ) { // code to handle ExceptionName exception }` Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows: `try { // protected code`

C++

251

`}catch(...) { // code to handle any exception }` The following is an example, which throws a division by zero exception and we catch it in catch block. `#include <iostream> using namespace std; double division(int a, int b) { if( b == 0 ) { throw "Division by zero condition!"; } return (a/b); } int main () { int x = 50; int y = 0; double z = 0; try { z = division(x, y); cout << z << endl; }catch (const char* msg) { cerr << msg << endl; } return 0; }`

C++

252

Because we are raising an exception of type **const char***, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result: `Division by zero condition!` **C++ Standard Exceptions** C++ provides a list of standard exceptions defined in **<exception>** which we can use in our programs. These are arranged in a parent-child class hierarchy shown below: Here is the small description of each exception mentioned in the above hierarchy: **Exception Description** std::exception An exception and parent class of all the standard C++ exceptions.

C++

253

std::bad_alloc This can be thrown by new. std::bad_cast This can be thrown by dynamic_cast. std::bad_exception This is useful device to handle unexpected exceptions in a C++ program. std::bad_typeid This can be thrown by typeid. std::logic_error An exception that theoretically can be detected by reading the code. std::domain_error This is an exception thrown when a mathematically invalid domain is used. std::invalid_argument This is thrown due to invalid arguments. std::length_error This is thrown when a too big std::string is created. std::out_of_range This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[](). std::runtime_error An exception that theoretically cannot be detected by reading the code. std::overflow_error This is thrown if a mathematical overflow occurs. std::range_error This is occurred when you try to store a value which is out of range. std::underflow_error This is thrown if a mathematical underflow occurs. **Define New Exceptions** You can define your own

exceptions by inheriting and overriding **exception** class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way: #include <iostream>


```
#include <exception> using namespace std; struct MyException : public
exception { const char * what () const throw () { return "C++ Exception"; }
}; int main() { try { throw MyException(); } catch(MyException& e) {
std::cout << "MyException caught" << std::endl; std::cout << e.what() <<
std::endl; } catch(std::exception& e) { //Other errors } }
```

This would produce the following result: MyException caught C++ Exception Here, **what()** is a public method provided by exception class and it has been overridden by all the child exception classes. This returns the cause of an exception.