OOP Final

Friday, 17 May 2024 3:26 pm

Virtual Functions: A virtual function is a member function that is declared within a base class and is overriden by a derived class.

In easier words: Virtual functions are functions that are written to be overriden in derived class.

```
// Base class: Person
class Person {
protected:
    string name;
    int id;
public:
    Person(string name, int id) : name(name), id(id) {}

    virtual void displayInfo() {
        cout << "Name: " << name << endl;
        cout << "ID: " << id << endl;
    }
};
```

→ Because I know I am going to override this function in derived classes then I'll add the keyword virtual.

A few rules of Virtual Functions:

1. Virtual functions cannot be static.
2. A virtual function can be a friend function of another class.
3. Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.
4. The prototype of virtual functions should be the same in the base as well as the derived class.
5. They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
6. A class may have a virtual destructor but it cannot have a virtual constructor.

From <https://www.geeksforgeeks.org/virtual-function-cpp/>

That is all that I think we need to know about virtual functions for now, however if you want you can read on it: https://www.geeksforgeeks.org/virtual-function-cpp/

Pure Virtual Functions: It is a bit same as virtual function but the difference is that it must be overriden by a derived class, and that the base class cannot implement it.

To create a pure virtual function add "= 0" at the end of the header of virtual function.

```
// Base class: Person
class Person {
private:
    string name;
    int id;
public:
    Person(string name, int id) : name(name), id(id) {}

    virtual void displayInfo() = 0;
};
```

→ Pure Virtual Function

This is then implemented in the derived classes:

```
// Derived class: Teacher
class Teacher : public Person {
private:
    string subject;
public:
    Teacher(string name, int id, string subject)
        : Person(name, id), subject(subject) {}

    void displayInfo() {
        cout << "Subject: " << subject << endl;
    }
};
```

→ Pure Virtual Function implementation and overriden

Note: A pure virtual function MUST be overriden by a derived class whereas a virtual function CAN be overridden by a derived class.

Now let's talk about two terms related to pure virtual functions:

1- Abstract Classes
2- Interfaces

What is an Abstract Class?

To put it simply it is a class that contains atleast 1 pure virtual function and has other normal functions with their implementation as well.

What is an Interface?

It is a class that has only pure virtual functions, there are no functions with their implementation.

Generics: In easy words its progamming using Templates

Templates: It is a tool that allows us to pass data type as parameter so that we dont have to write same code for different data types.

Generic Functions: We will use template to have a function that can be of any data type that we require.

Let's see it using an example:

This is the syntax for template

```
// Template function to find the maximum of two values
template <typename T>
T myMax(T x, T y)
{
    return (x > y) ? x : y;
}
```

T can be any data type that we pass.

Both parameters are of data type T.

This way both a and b will have the data type which is passed in.

```
int main()
{
    // Call myMax for int
    cout << myMax<int>(3, 7) << endl;
    // call myMax for double
    cout << myMax<double>(3.0, 7.0) << endl;
    // call myMax for char
    cout << myMax<char>('g', 'e') << endl;

    return 0;
}
```

Data type passed in as parameter

Class Templates: This is the just like function templates.

```
// Template class to hold a single value
template <typename T>
class Box {
private:
    T value;

public:
    // Constructor
    Box(T val) : value(val) {}

    // Getter for value
    T getValue() const {
        return value;
    }

    // Setter for value
    void setValue(T val) {
        value = val;
    }
};
```

T has data type int

Attribute uses T data type.

Constructor has the attribute with same data type.

Setters and Getters also have data type T when dealing with that attribute.

```
int main() {
    // Box holding an int
    Box<int> intBox(123);
    cout << "Box contains: " << intBox.getValue() << endl;

    // Box holding a double
    Box<double> doubleBox(45.67);
    cout << "Box contains: " << doubleBox.getValue() << endl;

    // Box holding a std::string
    Box<string> stringBox("Hello, world!");
    cout << "Box contains: " << stringBox.getValue() << endl;

    // Modifying the value in the box
    stringBox.setValue("Goodbye, world!");
    cout << "Box now contains: " << stringBox.getValue() << endl;

    return 0;
}
```

Passing in data type

Passing in the constructor the data type with same data type we passed in.

Because in prev line we added string as data type of attribute, so we can set using the same data type

Output:

```
Box contains: 123
Box contains: 45.67
Box contains: Hello, world!
Box now contains: Goodbye, world!
```

More than one argument in template:

```cpp
// Template class to hold a pair of values
template <typename T1, typename T2>
class Pair {
private:
    T1 first;
    T2 second;

public:
    // Constructor
    Pair(T1 firstValue, T2 secondValue) : first(firstValue), second(secondValue) {}

    // Getter for first
    T1 getFirst() const {
        return first;
    }

    // Getter for second
    T2 getSecond() const {
        return second;
    }

    // Setter for first
    void setFirst(T1 firstValue) {
        first = firstValue;
    }

    // Setter for second
    void setSecond(T2 secondValue) {
        second = secondValue;
    }
};
```

T1 has data type int

T2 has data type double

We can have more than one data type in template

The same rules for one data type apply in two data types.

```cpp
int main() {
    // Pair of int and double
    Pair<int, double> intDoublePair(1, 3.14);
    cout << "First: " << intDoublePair.getFirst() << ", Second: " << intDoublePair.getSecond() << endl;

    // Pair of std::string and int
    Pair<string, int> stringIntPair("Age", 30);
    cout << "First: " << stringIntPair.getFirst() << ", Second: " << stringIntPair.getSecond() << endl;

    // Changing the values using setters
    stringIntPair.setFirst("Weight");
    stringIntPair.setSecond(70);
    cout << "After modification: First: " << stringIntPair.getFirst() << ", Second: " << stringIntPair.getSecond() << endl;

    return 0;
}
```

Output:

```
First: 1, Second: 3.14
First: Age, Second: 30
After modification: First: Weight, Second: 70
```

Note: We can use default values in typename just like we do for constructor.

```cpp
template <typename T1 = int, typename T2 = string>
```

Filing:-

ifstream:- A class used for input operations.

ofstream:- A class used for output operations.

fstream:- Used for both input and output operations.

Adding data to file:

```
#include <iostream>
#include <fstream> // Needed for file operations

using namespace std;

int main() {
    // Open a file for writing
    ofstream outputFile("output.txt");

    // Check if the file opened successfully
    if (!outputFile) {
        cout << "Failed to open the file." << endl;
        return 1;
    }

    // Write some data to the file
    outputFile << "Hello, this is a line written to a file." << endl;
    outputFile << "Writing another line here." << endl;

    // Close the file
    outputFile.close();

    cout << "Data has been written to the file 'output.txt'." << endl;

    return 0;
}
```

To open a file name using constructor.

Checking if file is opened

Adding data and new line to file.

<< adds data to outputFile variable and it adds to file

Close file

We can also use .is_open() to check if file is not open:

```
// Check if the file opened successfully
if (!outputFile.is_open()) {
    cout << "Failed to open the file." << endl;
    return 1;
}
```

Reading data from file:

```
#include <iostream>
#include <fstream> // Needed for file operations

using namespace std;

int main() {
    // Open a file for reading
    ifstream outputFile("output.txt");

    string line;

    // Check if the file opened successfully
    if (!outputFile.is_open()) {
        cout << "Failed to open the file." << endl;
        return 1;
    }

    // Write some data to the file
    getline(outputFile, line);
    cout << line << endl;
    getline(outputFile, line);
    cout << line;

    // Close the file
    outputFile.close();

    return 0;
}
```

Output:

```
Hello, this is a line written to a file.
Writing another line here.
```

Use of getline() to read a full string line from a file. Then add that to line variable.

Adding non-string data to file:

```cpp
int main() {
    // Open a file for reading
    ofstream outputFile("output.txt");

    int x = 2, y = 3;

    // Check if the file opened successfully
    if (!outputFile.is_open()) {
        cout << "Failed to open the file." << endl;
        return 1;
    }

    // Write some data to the file
    outputFile << x << y;

    // Close the file
    outputFile.close();

    return 0;
}
```

Adding 2 3 to file:

output.txt

File Edit View

23

Reading non-string data from a file:

```cpp
int main() {
    // Open a file for reading
    ifstream outputFile("output.txt");

    int x, y;

    // Check if the file opened successfully
    if (!outputFile.is_open()) {
        cout << "Failed to open the file." << endl;
        return 1;
    }

    // Reading some data to the file
    outputFile >> x >> y;
    cout << "x: " << x << ", y: " << y;

    // Close the file
    outputFile.close();

    return 0;
}
```

Output:

```
x: 23, y: 256
```

We expected x:2, y:3 but got this. Why?

See, when cpp reads from a file it only stops reading when it encounters a space or endl or terminator char. So what happend here was that it read 23 from the file and stopped as there was nothing after it. It stored this 23 in x. But when we read from the file again for y, it took out garbage values and stored them in y.

To have our expected output lets store data correctly:

```cpp
int main() {
    // Open a file for reading
    ofstream outputFile("output.txt");

    int x = 2, y = 3;

    // Check if the file opened successfully
    if (!outputFile.is_open()) {
        cout << "Failed to open the file." << endl;
        return 1;
    }

    // Write some data to the file
    outputFile << x << " " << y;

    // Close the file
    outputFile.close();

    return 0;
}
```

Added a space, could have used endl aswell.

output.txt

File Edit View

2 3

Lets read it again:

```
int main() {
    // Open a file for reading
    ifstream outputFile("output.txt");

    int x, y;

    // Check if the file opened successfully
    if (!outputFile.is_open()) {
        cout << "Failed to open the file." << endl;
        return 1;
    }

    // Read some data from the file
    outputFile >> x >> y;
    cout << "x: " << x << ", y: " << y;

    // Close the file
    outputFile.close();

    return 0;
}
```

Output:

```
x: 2, y: 3
```

Now we have our expected Output.

Reading char from file:

Our file has:

output.txt

File Edit View

```
Hello
```

Let's read one char from it:

```
int main() {
    // Open a file for reading
    ifstream outputFile("output.txt");

    char a;

    // Check if the file opened successfully
    if (!outputFile.is_open()) {
        cout << "Failed to open the file." << endl;
        return 1;
    }

    // Read some data from the file
    outputFile.get(a);
    cout << a;

    // Close the file
    outputFile.close();

    return 0;
}
```

It will take one char from the file and store in a

Output:

```
H
```

Let's try to read all the characters:

```
// Read some data from the file
while (outputFile.get(a)){
    cout << a;
}
```

get() returns NULL when it reaches end of file

If file data is:

output.txt

File Edit View

```
Hello World
Hi
```

Output:

```
Hello World
Hi
```

Note: Instead of using ifstream and ofstream we can also use fstream.

Let's see how we can output to a file using fstream:
(Adding data to file)

```
int main() {
    // Open a file for reading
    fstream outputFile("output.txt", ios::out);

    string a = "Hello";

    // Check if the file opened successfully
    if (!outputFile.is_open()) {
        cout << "Failed to open the file." << endl;
        return 1;
    }

    // Enter some data to the file
    outputFile << a;

    // Close the file
    outputFile.close();

    return 0;
}
```

File Mode

If we want to add to a file we write this.

Same as before

Lets try reading now:

```
int main() {
    // Open a file for reading
    fstream outputFile("output.txt", ios::in);

    string a = "Hello";

    // Check if the file opened successfully
    if (!outputFile.is_open()) {
        cout << "Failed to open the file." << endl;
        return 1;
    }

    // Read some data from the file
    outputFile.seekg(0, ios::beg);
    getline(outputFile, a);
    cout << a;

    // Close the file
    outputFile.close();

    return 0;
}
```

Output: `Hello`

This time ios::in

Add this line to set pointer to beginning of file

Reading file like before

Now what if we want to append data to the file:

```
int main() {
    // Open a file for reading
    fstream outputFile("output.txt", ios::app | ios::in);

    string a = "Hello";

    // Check if the file opened successfully
    if (!outputFile.is_open()) {
        cout << "Failed to open the file." << endl;
        return 1;
    }

    // Enter some data to the file
    outputFile << "NewData";
    // Read some data from the file
    outputFile.seekg(0, ios::beg);
    getline(outputFile, a);
    cout << a;

    // Close the file
    outputFile.close();

    return 0;
}
```

To append the data

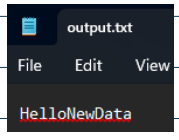
To read aswell

Appending more data

Outputting data of file

Output: `HelloNewData`

File Contents:



Other file Modes:

1. **ios::app**
   Append mode. All output to that file to be appended to the end.
2. **ios::ate**
   Open a file for output and move the read/write control to the end of the file.
3. **ios::in**
   Open a file for reading.
4. **ios::out**
   Open a file for writing.
5. **ios::trunc**
   If the file already exists, its contents will be truncated before opening the file.

Note: Look up .read(), .write() for storing an object in a file.

Exception Handling:

In easy word this is for dealing with possible errors. It is done using try throw catch.

It is best understood by examples:

```
// Function that throws an exception
double divide(int a, int b) {
    try {
        if (b == 0) {
            throw "Divide by zero exception";
        }
        else {
            return a / b;
        }
    }
    catch const char* error {
        cout << error << endl;
    }
    return 0;
}

int main() {
    int x = 10;
    int y = 0;
    int result;

    result = divide(x, y);

    cout << result;

    return 0;
}
```

trying a code

If the number to be divided by is zero then throw

This block executes when the throw statement is executed

The variable will store whatever is thrown.

Outputting the content of the variable

Doing the division, y is 0, meaning we are trying to divide the number by zero, so output is

```
Divide by zero exception
```

↳ This was stored in variable error

Note:
- Statements put after the catch block we will executed.
- Statement put exactly after throw will never be executed.
- Catch parameter should be of data type which is being thrown.

Default Catch:

```
// Function that throws an exception
double divide(int a, int b) {
    try {
        if (b == 0) {
            throw "Divide by zero exception";
        }
        else {
            return a / b;
        }
    }
    catch(int error) {
        cout << error << endl;
    }
    catch(...){
        cout << "Default Executed\n";
    }
    return 0;
}
```

Here no catch is available to catch the thrown string exception, therefore catch(...) will catch it.

Output: `Default Executed`

Exception Library:

It is a library that makes errors more readable.

I'll demonstrate a few examples:

Below will be a list of errors

```
// Function that throws an exception
double divide(int a, int b) {
    try {
        if (b == 0) {
            throw runtime_error("Divide by zero exception");
        }
        else {
            return a / b;
        }
    }
    catch(exception& error) {
        cout << error.what() << endl;
    }
    return 0;
}
```

Error Type

Statement I want to throw

Library Errors are caught by this

Think of it like this: Reading the words runtime_error in the code is much easier than just a throw.

It gives us the thrown sentence.

Output: `Divide by zero exception`

We can also do this:

```
catch(runtime_error& error) {
    cout << error.what() << endl;
}
```

Inherits from exception thus can be used

Let's look at another example:

```
// Function to illustrate bad_alloc
void createArray()
{
    // Try Block
    try {
        // Create an array
        int* array = new int[100000000000];

        // If created successfully then print the message
        cout << "Array created successfully";
    }
    catch (bad_alloc& e) {
        cout << e.what();
    }
}

int main()
{
    createArray();
    return 0;
}
```

Used for catching errors by new.

This time I am trying to create an array of too large size. So we should get an error.

Where is the throw??

Thrown by new upon allocation

Output: `std::bad_alloc`

Exceptions List:

std::exception	This is an exception and the parent class of all standard C++ exceptions.
std::bad_alloc	This exception is thrown by a new keyword.
std::bad_cast	This is an exception thrown by dynamic_cast.
std::bad_exception	A useful device for handling unexpected exceptions in C++ programs.
std::bad_typeid	An exception thrown by typeid.
std::logic_error	This exception is theoretically detectable by reading code.
std::domain_error	This is an exception thrown after using a mathematically invalid domain.
std::invalid_argument	An exception thrown for using invalid arguments.
std::length_error	An exception thrown after creating a big std::string.
std::out_of_range	Thrown by at method.
std::runtime_error	This is an exception that cannot be detected via reading the code.
std::overflow_error	This exception is thrown after the occurrence of a mathematical overflow.
std::range_error	This exception is thrown when you attempt to store an out-of-range value.
std::underflow_error	An exception thrown after the occurrence of mathematical underflow.

Don’t forget to go through class exceptions