

OOPS

Abstraction

Data abstraction refers to providing only **essential** information to the outside world and hiding their background details, to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take a example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having any knowledge of its internals.

In C++, classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

In C++, we use **classes** to define our own abstract data types (ADT). You can use the **cout** object of class **ostream** to stream data to standard output like this –

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    cout << "Hello World!" << endl;
    return 0;
}
```

Here, you don't need to understand **how cout displays** the text on the user's screen. You need to only know the public interface and the underlying implementation of ‘cout’ is free to change.

In C++, we use access labels to define the abstract interface to the class. A class may contain zero or more access labels.

Members defined with a **public label** are accessible to all parts of the program. The data-abstraction view of a type is defined by its public members.

Members defined with a **private label** are not accessible to code that uses the class. The private sections hide the implementation from code that uses the type.

There are no restrictions on how often an access label may appear. Each access label specifies the access level of the succeeding member definitions. The specified access level remains in effect until the next access label is encountered or the closing right brace of the class body is seen.

Benefits of Data Abstraction

Data abstraction provides two important advantages

Class internals are protected from inadvertent user-level errors, which might corrupt the state of the object.

The class implementation may evolve over time in response to changing requirements or bug reports without requiring change in user-level code.

By defining data members only in the private section of the class, the class author is free to make changes in the data. If the implementation changes, only the class code needs to be examined to see what affect the change may have. If data is public, then any function that directly access the data members of the old representation might be broken.

Any C++ program where you implement a class with public and private members is an example of data abstraction. Consider the following example –

```
#include <bits/stdc++.h>
using namespace std;
class Cuboid {
    int length, breadth, height;
```

```

public:
    // Setter functions
    void setLength(int l){length = l;}
    void setBreadth(int b){breadth = b;}
    void setHeight(int h){height = h;}

    // Getter Functions
    int getLength() { return length; }
    int getBreadth() { return breadth; }
    int getHeight() { return height; }

    // Function to Calculate Volume
    int getVolume() { return length * breadth * height; }
};

int main() {
    Cuboid c;
    c.setLength(1);
    c.setBreadth(2);
    c.setHeight(3);

    cout << "Length: " << c.getLength() << "\n";
    cout << "Breadth: " << c.getBreadth() << "\n";
    cout << "Height: " << c.getHeight() << "\n";
    cout << "Volume is: " << c.getVolume();
}

```

Above class returns the volume. The public member - **getVolume()** is the interface to the outside world and a user needs to know it to use the class. The private members ***length, breadth and height*** are something that the user doesn't need to know about, but is needed for the class to operate properly.

Abstraction separates code into interface and implementation. So while designing your component, you must keep interface independent of the implementation so that if you change underlying implementation then interface would remain intact.

In this case whatever programs are using these interfaces, they would not be impacted and would just need a recompilation with the latest implementation.



NO NOTES TO SHOW