Lập trình

**C++**

# LỚP VÀ ĐỐI TƯỢNG

Mục đích chính của lập trình C++ là thêm hướng đối tượng vào ngôn ngữ lập trình C và các lớp là tính năng trung tâm của C++ hỗ trợ lập trình hướng đối tượng và thường được gọi là các kiểu do người dùng định nghĩa.

Một lớp được sử dụng để chỉ định hình thức của một đối tượng và nó kết hợp biểu diễn dữ liệu và các phương thức để thao tác dữ liệu đó thành một gói gọn gàng. Dữ liệu và các chức năng trong một lớp được gọi là thành viên của lớp.

## **Khai báo lớp trong C++**

Khi bạn xác định một lớp, bạn xác định một kế hoạch chi tiết cho một kiểu dữ liệu. Điều này không thực sự xác định bất kỳ dữ liệu nào, nhưng nó xác định ý nghĩa của tên lớp, tức là đối tượng của lớp sẽ bao gồm những gì và những thao tác nào có thể được thực hiện trên một đối tượng như vậy.

Định nghĩa lớp bắt đầu bằng từ khóa lớp theo sau là tên lớp; và phần thân của lớp, được bao quanh bởi một cặp dấu ngoặc nhọn. Một định nghĩa lớp phải được theo sau bởi dấu chấm phẩy hoặc một danh sách các khai báo. Ví dụ, chúng ta đã định nghĩa kiểu dữ liệu Box bằng từ khóa class như sau:

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

Từ khóa public xác định các thuộc tính truy cập của các thành viên của lớp theo sau nó. Một thành viên công cộng có thể được truy cập từ bên ngoài lớp ở bất kỳ đâu trong phạm vi của đối tượng lớp. Bạn cũng có thể chỉ định các thành viên của một lớp là riêng tư hoặc được bảo vệ mà chúng ta sẽ thảo luận trong phần phụ.

## **Khai báo đối tượng trong C++**

Một lớp cung cấp các bản thiết kế cho các đối tượng, vì vậy về cơ bản một đối tượng được tạo ra từ một lớp. Chúng ta khai báo các đối tượng của một lớp với cùng một kiểu khai báo mà chúng ta khai báo các biến kiểu cơ bản. Các câu lệnh sau khai báo hai đối tượng của lớp Box :

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Cả hai đối tượng Box1 và Box2 sẽ có bản sao thành viên dữ liệu riêng.

## **Truy cập các dữ liệu thành viên**

Các thành viên dữ liệu công khai của các đối tượng của một lớp có thể được truy cập bằng cách sử dụng toán tử truy cập thành viên trực tiếp (.). Hãy để chúng tôi thử ví dụ sau để làm cho mọi thứ rõ ràng:

#include <iostream>

using namespace std;

class Box {

public:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

int main() {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.height = 5.0;

Box1.length = 6.0;

Box1.breadth = 7.0;

// box 2 specification

Box2.height = 10.0;

Box2.length = 12.0;

Box2.breadth = 13.0;

// volume of box 1

volume = Box1.height \* Box1.length \* Box1.breadth;

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.height \* Box2.length \* Box2.breadth;

cout << "Volume of Box2 : " << volume <<endl;

return 0;

}

Kết quả :

Volume of Box1 : 210

Volume of Box2 : 1560

Điều quan trọng cần lưu ý là không thể truy cập trực tiếp các thành viên riêng tư và được bảo vệ bằng cách sử dụng toán tử truy cập thành viên trực tiếp (.). Chúng tôi sẽ tìm hiểu cách truy cập các thành viên riêng tư và được bảo vệ.

## **Chi tiết về lớp và đối tượng**

Cho đến giờ, bạn đã có những ý tưởng cơ bản về Lớp và Đối tượng trong C++. Có nhiều khái niệm thú vị khác liên quan đến Lớp và Đối tượng C++ mà chúng ta sẽ thảo luận trong các phần phụ khác nhau được liệt kê bên dưới:

|  |  |
| --- | --- |
| **Sr.No** | **Concept & Description** |
| 1 | [Class Member Functions](https://www.tutorialspoint.com/cplusplus/cpp_class_member_functions.htm)  A member function of a class is a function that has its definition or its prototype within the class definition like any other variable. |
| 2 | [Class Access Modifiers](https://www.tutorialspoint.com/cplusplus/cpp_class_access_modifiers.htm)  A class member can be defined as public, private or protected. By default members would be assumed as private. |
| 3 | [Constructor & Destructor](https://www.tutorialspoint.com/cplusplus/cpp_constructor_destructor.htm)  A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted. |
| 4 | [Copy Constructor](https://www.tutorialspoint.com/cplusplus/cpp_copy_constructor.htm)  The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously. |
| 5 | [Friend Functions](https://www.tutorialspoint.com/cplusplus/cpp_friend_functions.htm)  A **friend** function is permitted full access to private and protected members of a class. |
| 6 | [Inline Functions](https://www.tutorialspoint.com/cplusplus/cpp_inline_functions.htm)  With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function. |
| 7 | [this Pointer](https://www.tutorialspoint.com/cplusplus/cpp_this_pointer.htm)  Every object has a special pointer **this** which points to the object itself. |
| 8 | [Pointer to C++ Classes](https://www.tutorialspoint.com/cplusplus/cpp_pointer_to_class.htm)  A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it. |
| 9 | [Static Members of a Class](https://www.tutorialspoint.com/cplusplus/cpp_static_members.htm)  Both data members and function members of a class can be declared as static. |

# TRỪU TƯỢNG

Trừu tượng hóa dữ liệu đề cập đến việc chỉ cung cấp thông tin cần thiết cho thế giới bên ngoài và ẩn các chi tiết cơ bản của chúng, nghĩa là biểu diễn thông tin cần thiết trong chương trình mà không trình bày chi tiết.

Trừu tượng hóa dữ liệu là một kỹ thuật lập trình (và thiết kế) dựa trên sự tách biệt giữa giao diện và triển khai.

Hãy lấy một ví dụ thực tế về TV mà bạn có thể bật và tắt, thay đổi kênh, điều chỉnh âm lượng và thêm các thành phần bên ngoài như loa, VCR và đầu DVD, NHƯNG bạn không biết chi tiết bên trong của nó, rằng nghĩa là, bạn không biết cách nó nhận tín hiệu qua vô tuyến hoặc qua cáp, cách nó dịch chúng và cuối cùng hiển thị chúng trên màn hình.

Vì vậy, chúng ta có thể nói rằng một chiếc tivi tách biệt rõ ràng việc triển khai bên trong với giao diện bên ngoài và bạn có thể chơi với các giao diện của nó như nút nguồn, bộ chuyển kênh và điều khiển âm lượng mà không cần biết gì về bên trong của nó.

Trong C++, các lớp cung cấp mức độ trừu tượng hóa dữ liệu cao. Chúng cung cấp đủ các phương thức công khai cho thế giới bên ngoài để chơi với chức năng của đối tượng và để thao tác dữ liệu đối tượng, tức là trạng thái mà không thực sự biết lớp đã được triển khai bên trong như thế nào.

Ví dụ, chương trình của bạn có thể gọi hàm sort() mà không cần biết hàm này thực sự sử dụng thuật toán nào để sắp xếp các giá trị đã cho. Trên thực tế, việc triển khai cơ bản của chức năng sắp xếp có thể thay đổi giữa các bản phát hành của thư viện và miễn là giao diện vẫn giữ nguyên, lệnh gọi hàm của bạn sẽ vẫn hoạt động.

Trong C++, chúng ta sử dụng các lớp để định nghĩa các kiểu dữ liệu trừu tượng (ADT) của riêng chúng ta. Bạn có thể sử dụng đối tượng cout của lớp ostream để truyền dữ liệu ra đầu ra tiêu chuẩn như thế này :

#include <iostream>

using namespace std;

int main() {

cout << "Hello C++" <<endl;

return 0;

}

Ở đây, bạn không cần hiểu cout hiển thị văn bản trên màn hình của người dùng như thế nào. Bạn chỉ cần biết giao diện chung và triển khai cơ bản của 'cout' là tự do thay đổi.

## **Nhãn phân quyền truy cập**

Trong C++, chúng ta sử dụng các nhãn truy cập để định nghĩa giao diện trừu tượng cho lớp. Một lớp có thể chứa 0 hoặc nhiều nhãn truy cập -

* Các thành viên được xác định bằng nhãn công khai (public) có thể truy cập được vào tất cả các phần của chương trình. Chế độ xem trừu tượng hóa dữ liệu của một loại được xác định bởi các thành viên công cộng của nó.
* Các thành viên được định nghĩa bằng nhãn riêng (private) không thể truy cập được bằng mã sử dụng lớp. Các phần riêng tư ẩn việc triển khai khỏi mã sử dụng loại.

Không có hạn chế về tần suất xuất hiện nhãn truy cập. Mỗi nhãn truy cập chỉ định cấp độ truy cập của các định nghĩa thành viên kế tiếp. Mức truy cập được chỉ định vẫn có hiệu lực cho đến khi gặp nhãn truy cập tiếp theo hoặc nhìn thấy dấu ngoặc nhọn bên phải của nội dung lớp.

## **Lợi ích của trừu tượng hóa dữ liệu**

Trừu tượng hóa dữ liệu cung cấp hai lợi thế quan trọng :

* Các phần bên trong lớp được bảo vệ khỏi các lỗi cấp độ người dùng vô ý, có thể làm hỏng trạng thái của đối tượng.
* Việc triển khai lớp có thể phát triển theo thời gian để đáp ứng các yêu cầu thay đổi hoặc báo cáo lỗi mà không yêu cầu thay đổi mã cấp độ người dùng.

Bằng cách xác định các thành viên dữ liệu chỉ trong phần riêng tư của lớp, tác giả lớp có thể tự do thực hiện các thay đổi trong dữ liệu. Nếu quá trình triển khai thay đổi, chỉ cần kiểm tra mã lớp để xem điều gì có thể ảnh hưởng đến sự thay đổi đó. Nếu dữ liệu là công khai, thì bất kỳ chức năng nào truy cập trực tiếp vào các thành viên dữ liệu của biểu diễn cũ đều có thể bị hỏng.

Ví dụ:

Bất kỳ chương trình C++ nào mà bạn triển khai một lớp với các thành viên công khai và riêng tư đều là một ví dụ về trừu tượng hóa dữ liệu. Xem xét ví dụ sau :

#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;

}

Kết quả :

Total 60

Lớp trên cộng các số lại với nhau và trả về tổng. Các thành viên công cộng - addNum và getTotal là các giao diện với thế giới bên ngoài và người dùng cần biết chúng để sử dụng lớp. Tổng số thành viên riêng tư là thứ mà người dùng không cần biết, nhưng cần thiết để lớp hoạt động bình thường.

## **Chiến lược thiết kế**

Trừu tượng tách mã thành giao diện và triển khai. Vì vậy, trong khi thiết kế thành phần của mình, bạn phải giữ giao diện độc lập với việc triển khai để nếu bạn thay đổi triển khai cơ bản thì giao diện sẽ vẫn nguyên vẹn.

Trong trường hợp này, bất kỳ chương trình nào đang sử dụng các giao diện này, chúng sẽ không bị ảnh hưởng và chỉ cần biên dịch lại với triển khai mới nhất.

# ĐÓNG GÓI

Tất cả các chương trình C++ bao gồm hai yếu tố cơ bản sau -

* Câu lệnh chương trình (mã) − Đây là một phần của chương trình thực hiện các hành động và chúng được gọi là hàm.
* Dữ liệu chương trình − Dữ liệu là thông tin của chương trình bị ảnh hưởng bởi các chức năng của chương trình.

Đóng gói là một khái niệm Lập trình hướng đối tượng liên kết dữ liệu và các chức năng thao tác dữ liệu với nhau, đồng thời giữ an toàn cho cả hai khỏi sự can thiệp và lạm dụng từ bên ngoài. Đóng gói dữ liệu dẫn đến khái niệm OOP quan trọng về ẩn dữ liệu.

Đóng gói dữ liệu là một cơ chế gói dữ liệu và các chức năng sử dụng chúng và trừu tượng hóa dữ liệu là một cơ chế chỉ hiển thị các giao diện và ẩn các chi tiết triển khai khỏi người dùng.

C++ hỗ trợ các thuộc tính đóng gói và ẩn dữ liệu thông qua việc tạo các kiểu do người dùng định nghĩa, được gọi là các lớp. Chúng ta đã biết rằng một lớp có thể chứa các thành viên private, protected và public. Theo mặc định, tất cả các mục được xác định trong một lớp là riêng tư. Ví dụ :

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

};

Các biến chiều dài, chiều rộng và chiều cao là riêng tư. Điều này có nghĩa là chúng chỉ có thể được truy cập bởi các thành viên khác của lớp Box chứ không phải bởi bất kỳ phần nào khác trong chương trình của bạn. Đây là một cách đóng gói đạt được.

Để công khai các phần của lớp (nghĩa là các phần khác trong chương trình của bạn có thể truy cập được), bạn phải khai báo chúng sau từ khóa public. Tất cả các biến hoặc hàm được xác định sau bộ xác định công khai đều có thể truy cập được bởi tất cả các hàm khác trong chương trình của bạn.

Làm cho một lớp trở thành bạn của lớp khác sẽ hiển thị các chi tiết triển khai và giảm sự đóng gói. Lý tưởng nhất là giữ càng nhiều chi tiết của mỗi lớp ẩn khỏi tất cả các lớp khác càng tốt.

Ví dụ:

Bất kỳ chương trình C++ nào mà bạn triển khai một lớp với các thành viên công khai và riêng tư đều là một ví dụ về đóng gói dữ liệu và trừu tượng hóa dữ liệu. Xem xét ví dụ sau :

#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;

}

Kết quả :

Total 60

Lớp trên cộng các số lại với nhau và trả về tổng. Các thành viên công cộng addNum và getTotal là các giao diện với thế giới bên ngoài và người dùng cần biết chúng để sử dụng lớp. Tổng số thành viên riêng tư là thứ bị ẩn khỏi thế giới bên ngoài, nhưng cần thiết để lớp hoạt động bình thường.

## **Chiến lược thiết kế**

Hầu hết chúng ta đã học cách đặt các thành viên lớp ở chế độ riêng tư theo mặc định trừ khi chúng ta thực sự cần tiết lộ chúng. Đó chỉ là đóng gói tốt.

Điều này được áp dụng thường xuyên nhất cho các thành viên dữ liệu, nhưng nó áp dụng như nhau cho tất cả các thành viên, bao gồm cả các chức năng ảo.

# KẾ THỪA

Một trong những khái niệm quan trọng nhất trong lập trình hướng đối tượng là tính kế thừa. Tính kế thừa cho phép chúng ta định nghĩa một lớp dựa trên một lớp khác, giúp tạo và duy trì một ứng dụng dễ dàng hơn. Điều này cũng tạo cơ hội để sử dụng lại chức năng mã và thời gian triển khai nhanh.

Khi tạo một lớp, thay vì viết các thành viên dữ liệu và hàm thành viên hoàn toàn mới, lập trình viên có thể chỉ định rằng lớp mới sẽ kế thừa các thành viên của một lớp hiện có. Lớp hiện có này được gọi là lớp cơ sở và lớp mới được gọi là lớp dẫn xuất.

Ý tưởng kế thừa thực hiện mối quan hệ is. Ví dụ: động vật có vú LÀ động vật IS-A, chó IS-A động vật có vú do đó chó IS-A động vật cũng vậy, v.v.

## **Lớp cơ sở và lớp dẫn suất**

Một lớp có thể được dẫn xuất từ nhiều hơn một lớp, có nghĩa là nó có thể kế thừa dữ liệu và chức năng từ nhiều lớp cơ sở. Để định nghĩa một lớp dẫn xuất, chúng ta sử dụng danh sách dẫn xuất lớp để chỉ định (các) lớp cơ sở. Một danh sách phái sinh lớp đặt tên cho một hoặc nhiều lớp cơ sở và có dạng :

lớp dẫn xuất: lớp cơ sở xác định quyền truy cập

Trong đó trình xác định quyền truy cập là một trong các công khai, được bảo vệ hoặc riêng tư và lớp cơ sở là tên của một lớp được xác định trước đó. Nếu bộ chỉ định truy cập không được sử dụng, thì nó sẽ ở chế độ riêng tư theo mặc định.

Xét một lớp cơ sở Shape và lớp dẫn xuất của nó là Rectangle như sau:

#include <iostream>

using namespace std;

// Base class

class Shape {

public:

void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

// Derived class

class Rectangle: public Shape {

public:

int getArea() {

return (width \* height);

}

};

int main(void) {

Rectangle Rect;

Rect.setWidth(5);

Rect.setHeight(7);

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

return 0;

}

Kết quả :

Total area: 35

## **Access Control and Inheritance**

Một lớp dẫn xuất có thể truy cập tất cả các thành viên không riêng tư của lớp cơ sở của nó. Do đó, các thành viên của lớp cơ sở không thể truy cập được đối với các hàm thành viên của các lớp dẫn xuất nên được khai báo là riêng tư trong lớp cơ sở.

Chúng ta có thể tóm tắt các loại truy cập khác nhau theo - ai có thể truy cập chúng theo cách sau :

|  |  |  |  |
| --- | --- | --- | --- |
| **Access** | **public** | **protected** | **private** |
| Same class | yes | yes | yes |
| Derived classes | yes | yes | no |
| Outside classes | yes | no | no |

Một lớp dẫn xuất kế thừa tất cả các phương thức của lớp cơ sở với các ngoại lệ sau :

* Hàm tạo, hàm hủy và hàm tạo sao chép của lớp cơ sở.
* Nạp chồng toán tử của lớp cơ sở.
* Các hàm friend của lớp cơ sở.

## **Các kiểu kế thừa**

Khi dẫn xuất một lớp từ một lớp cơ sở, lớp cơ sở có thể được kế thừa thông qua kế thừa công khai, được bảo vệ hoặc riêng tư. Loại kế thừa được chỉ định bởi trình chỉ định truy cập như đã giải thích ở trên.

Chúng tôi hầu như không sử dụng kế thừa được bảo vệ hoặc riêng tư, nhưng kế thừa công khai thường được sử dụng. Trong khi sử dụng các kiểu thừa kế khác nhau, các quy tắc sau được áp dụng -

* Kế thừa công khai − Khi dẫn xuất một lớp từ lớp cơ sở công khai, các thành viên công khai của lớp cơ sở trở thành thành viên công khai của lớp dẫn xuất và các thành viên protected của lớp cơ sở trở thành thành viên bảo vệ của lớp dẫn xuất. Các thành viên riêng của lớp cơ sở không bao giờ có thể truy cập trực tiếp từ lớp dẫn xuất, nhưng có thể được truy cập thông qua các lệnh gọi tới các thành viên công khai và được bảo vệ của lớp cơ sở.
* Kế thừa được bảo vệ − Khi dẫn xuất từ một lớp cơ sở được bảo vệ, các thành viên công khai và được bảo vệ của lớp cơ sở trở thành các thành viên được bảo vệ của lớp dẫn xuất.
* Kế thừa riêng tư - Khi xuất phát từ một lớp cơ sở riêng tư, các thành viên công khai và được bảo vệ của lớp cơ sở trở thành các thành viên riêng tư của lớp dẫn xuất

## **Đa kế thừa**

Một lớp C++ có thể kế thừa các thành viên từ nhiều hơn một lớp và đây là cú pháp mở rộng :

class derived-class: access baseA, access baseB....

Nơi truy cập là một trong các công khai, được bảo vệ hoặc riêng tư và sẽ được cấp cho mọi lớp cơ sở và chúng sẽ được phân tách bằng dấu phẩy như được hiển thị ở trên. Chúng ta hãy thử ví dụ sau :

#include <iostream>

using namespace std;

// Base class Shape

class Shape {

public:

void setWidth(int w) {

width = w;

}

void setHeight(int h) {

height = h;

}

protected:

int width;

int height;

};

// Base class PaintCost

class PaintCost {

public:

int getCost(int area) {

return area \* 70;

}

};

// Derived class

class Rectangle: public Shape, public PaintCost {

public:

int getArea() {

return (width \* height);

}

};

int main(void) {

Rectangle Rect;

int area;

Rect.setWidth(5);

Rect.setHeight(7);

area = Rect.getArea();

// Print the area of the object.

cout << "Total area: " << Rect.getArea() << endl;

// Print the total cost of painting

cout << "Total paint cost: $" << Rect.getCost(area) << endl;

return 0;

}

Kết quả :

Total area: 35

Total paint cost: $2450

# ĐA HÌNH

Từ đa hình có nghĩa là có nhiều dạng. Thông thường, tính đa hình xảy ra khi có một hệ thống phân cấp của các lớp và chúng có liên quan với nhau bằng sự kế thừa.

Tính đa hình trong C++ có nghĩa là một lệnh gọi hàm thành viên sẽ khiến một hàm khác được thực thi tùy thuộc vào loại đối tượng gọi hàm đó.

Xem xét ví dụ sau trong đó một lớp cơ sở đã được dẫn xuất bởi hai lớp khác :

#include <iostream>

using namespace std;

class Shape {

protected:

int width, height;

public:

Shape( int a = 0, int b = 0){

width = a;

height = b;

}

int area() {

cout << "Parent class area :" << width \* height << endl;

return width \* height;

}

};

class Rectangle: public Shape {

public:

Rectangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Rectangle class area :" << width \* height << endl;

return (width \* height);

}

};

class Triangle: public Shape {

public:

Triangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Triangle class area :" << (width \* height)/2 << endl;

return (width \* height / 2);

}

};

// Main function for the program

int main() {

Shape \*shape;

Rectangle rec(10,7);

Triangle tri(10,5);

// store the address of Rectangle

shape = &rec;

// call rectangle area.

shape->area();

// store the address of Triangle

shape = &tri;

// call triangle area.

shape->area();

return 0;

}

Kết quả :

Parent class area :70

Parent class area :50

Lý do cho đầu ra không chính xác là lệnh gọi của hàm area() đang được trình biên dịch đặt một lần làm phiên bản được xác định trong lớp cơ sở. Đây được gọi là độ phân giải tĩnh của lệnh gọi hàm hoặc liên kết tĩnh - lệnh gọi hàm được cố định trước khi chương trình được thực thi. Điều này đôi khi còn được gọi là liên kết sớm vì hàm area() được thiết lập trong quá trình biên dịch chương trình.

Nhưng bây giờ, hãy thực hiện một sửa đổi nhỏ trong chương trình của chúng ta và đặt trước khai báo area() trong lớp Shape với từ khóa virtual để nó trông như thế này :

#include <iostream>

using namespace std;

class Shape {

protected:

int width, height;

public:

Shape( int a = 0, int b = 0){

width = a;

height = b;

}

virtual int area() {

cout << "Parent class area :" << width \* height << endl;

return width \* height;

}

};

class Rectangle: public Shape {

public:

Rectangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Rectangle class area :" << width \* height << endl;

return (width \* height);

}

};

class Triangle: public Shape {

public:

Triangle( int a = 0, int b = 0):Shape(a, b) { }

int area () {

cout << "Triangle class area :" << (width \* height)/2 << endl;

return (width \* height / 2);

}

};

// Main function for the program

int main() {

Shape \*shape;

Rectangle rec(10,7);

Triangle tri(10,5);

// store the address of Rectangle

shape = &rec;

// call rectangle area.

shape->area();

// store the address of Triangle

shape = &tri;

// call triangle area.

shape->area();

return 0;

}

Sau sửa đổi nhỏ này, khi mã ví dụ trước được biên dịch và thực thi, nó sẽ tạo ra kết quả sau :

Rectangle class area :70

Triangle class area :25

Lần này, trình biên dịch xem xét nội dung của con trỏ thay vì loại của nó. Do đó, vì địa chỉ của các đối tượng thuộc lớp tri và rec được lưu trữ trong \*shape nên hàm area() tương ứng được gọi.

Như bạn có thể thấy, mỗi lớp con có một triển khai riêng cho hàm area(). Đây là cách đa hình thường được sử dụng. Bạn có các lớp khác nhau với một hàm có cùng tên và thậm chí có cùng tham số, nhưng với cách triển khai khác nhau.

## **Hàm ảo**

Hàm ảo là một hàm trong lớp cơ sở được khai báo bằng từ khóa ảo. Định nghĩa một hàm ảo trong lớp cơ sở, với một phiên bản khác trong lớp dẫn xuất, báo hiệu cho trình biên dịch rằng chúng ta không muốn liên kết tĩnh cho hàm này.

Điều chúng ta muốn là việc lựa chọn hàm được gọi tại bất kỳ điểm nào trong chương trình dựa trên loại đối tượng mà nó được gọi. Loại hoạt động này được gọi là liên kết động hoặc ràng buộc muộn.

## **Hàm thuần ảo**

Có thể bạn muốn đưa một hàm ảo vào lớp cơ sở để nó có thể được định nghĩa lại trong lớp dẫn xuất cho phù hợp với các đối tượng của lớp đó, nhưng không có định nghĩa có ý nghĩa nào bạn có thể đưa ra cho hàm trong lớp cơ sở .

Chúng ta có thể thay đổi hàm ảo area() trong lớp cơ sở thành như sau :

class Shape {

protected:

int width, height;

public:

Shape(int a = 0, int b = 0) {

width = a;

height = b;

}

// pure virtual function

virtual int area() = 0;

};

= 0 cho trình biên dịch biết rằng hàm không có phần thân và hàm ảo ở trên sẽ được gọi là hàm thuần ảo.

# NẠP CHỒNG

C++ cho phép bạn chỉ định nhiều hơn một định nghĩa cho tên hàm hoặc toán tử trong cùng một phạm vi, được gọi là nạp chồng hàm và nạp chồng toán tử tương ứng.

Một khai báo quá tải là một khai báo được khai báo có cùng tên với một khai báo đã được khai báo trước đó trong cùng một phạm vi, ngoại trừ việc cả hai khai báo có các đối số khác nhau và rõ ràng là định nghĩa (thực hiện) khác nhau.

Khi bạn gọi một hàm hoặc toán tử quá tải, trình biên dịch sẽ xác định định nghĩa phù hợp nhất để sử dụng, bằng cách so sánh các loại đối số bạn đã sử dụng để gọi hàm hoặc toán tử với các loại tham số được chỉ định trong các định nghĩa. Quá trình chọn hàm hoặc toán tử quá tải thích hợp nhất được gọi là giải quyết quá tải.

## **Nạp chồng hàm trong C++**

Bạn có thể có nhiều định nghĩa cho cùng một tên hàm trong cùng một phạm vi. Định nghĩa của hàm phải khác nhau về loại và/hoặc số đối số trong danh sách đối số. Bạn không thể quá tải các khai báo hàm chỉ khác nhau ở kiểu trả về.

Sau đây là ví dụ sử dụng cùng một chức năng print() để in các loại dữ liệu khác nhau:

#include <iostream>

using namespace std;

class printData {

public:

void print(int i) {

cout << "Printing int: " << i << endl;

}

void print(double f) {

cout << "Printing float: " << f << endl;

}

void print(char\* c) {

cout << "Printing character: " << c << endl;

}

};

int main(void) {

printData pd;

// Call print to print integer

pd.print(5);

// Call print to print float

pd.print(500.263);

// Call print to print character

pd.print("Hello C++");

return 0;

}

Kết quả :

Printing int: 5

Printing float: 500.263

Printing character: Hello C++

## **Nạp chồng toán tử trong C++**

Bạn có thể xác định lại hoặc quá tải hầu hết các toán tử tích hợp sẵn có trong C++. Do đó, một lập trình viên cũng có thể sử dụng các toán tử với các kiểu do người dùng định nghĩa.

Các toán tử được nạp chồng là các hàm có tên đặc biệt: từ khóa "toán tử" theo sau là ký hiệu cho toán tử được xác định. Giống như bất kỳ hàm nào khác, toán tử quá tải có kiểu trả về và danh sách tham số.

Box operator+(const Box&);

khai báo toán tử cộng có thể được sử dụng để cộng hai đối tượng Box và trả về đối tượng Box cuối cùng. Hầu hết các toán tử quá tải có thể được định nghĩa là các hàm không phải thành viên thông thường hoặc là các hàm thành viên của lớp. Trong trường hợp chúng ta định nghĩa hàm trên là hàm không phải thành viên của một lớp thì chúng ta sẽ phải truyền hai đối số cho mỗi toán hạng như sau :

Box operator+(const Box&, const Box&);

Sau đây là ví dụ để hiển thị khái niệm toán tử quá tải bằng cách sử dụng hàm thành viên. Ở đây, một đối tượng được truyền dưới dạng một đối số có các thuộc tính sẽ được truy cập bằng đối tượng này, đối tượng sẽ gọi toán tử này có thể được truy cập bằng toán tử này như được giải thích bên dưới :

#include <iostream>

using namespace std;

class Box {

public:

double getVolume(void) {

return length \* breadth \* height;

}

void setLength( double len ) {

length = len;

}

void setBreadth( double bre ) {

breadth = bre;

}

void setHeight( double hei ) {

height = hei;

}

// Overload + operator to add two Box objects.

Box operator+(const Box& b) {

Box box;

box.length = this->length + b.length;

box.breadth = this->breadth + b.breadth;

box.height = this->height + b.height;

return box;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

// Main function for the program

int main() {

Box Box1; // Declare Box1 of type Box

Box Box2; // Declare Box2 of type Box

Box Box3; // Declare Box3 of type Box

double volume = 0.0; // Store the volume of a box here

// box 1 specification

Box1.setLength(6.0);

Box1.setBreadth(7.0);

Box1.setHeight(5.0);

// box 2 specification

Box2.setLength(12.0);

Box2.setBreadth(13.0);

Box2.setHeight(10.0);

// volume of box 1

volume = Box1.getVolume();

cout << "Volume of Box1 : " << volume <<endl;

// volume of box 2

volume = Box2.getVolume();

cout << "Volume of Box2 : " << volume <<endl;

// Add two object as follows:

Box3 = Box1 + Box2;

// volume of box 3

volume = Box3.getVolume();

cout << "Volume of Box3 : " << volume <<endl;

return 0;

}

Kết quả

Volume of Box1 : 210

Volume of Box2 : 1560

Volume of Box3 : 5400

## **Những toán tử có hoặc không thể nạp chồng**

Có thể nạp chồng :

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| + | - | \* | / | % | ^ |
| & | | | ~ | ! | , | = |
| < | > | <= | >= | ++ | -- |
| << | >> | == | != | && | || |
| += | -= | /= | %= | ^= | &= |
| |= | \*= | <<= | >>= | [] | () |
| -> | ->\* | new | new [] | delete | delete [] |

Không thể nạp chồng :

|  |  |  |  |
| --- | --- | --- | --- |
| :: | .\* | . | ?: |

# INTERFACES

Một interface mô tả hành vi hoặc khả năng của một lớp C++ mà không cam kết triển khai cụ thể lớp đó.

Các interface C++ được triển khai bằng cách sử dụng các lớp trừu tượng và không nên nhầm lẫn các lớp trừu tượng này với trừu tượng hóa dữ liệu, đây là một khái niệm giữ các chi tiết triển khai tách biệt với dữ liệu liên quan.

Một lớp được tạo thành trừu tượng bằng cách khai báo ít nhất một trong các hàm của nó là hàm ảo thuần túy. Một hàm ảo thuần túy được chỉ định bằng cách đặt "= 0" trong phần khai báo của nó như sau:

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

};

Mục đích của một lớp trừu tượng (thường được gọi là ABC) là cung cấp một lớp cơ sở thích hợp mà từ đó các lớp khác có thể kế thừa. Các lớp trừu tượng không thể được sử dụng để khởi tạo các đối tượng và chỉ phục vụ như một giao diện. Cố gắng khởi tạo một đối tượng của lớp trừu tượng sẽ gây ra lỗi biên dịch.

Do đó, nếu một lớp con của ABC cần được khởi tạo, thì nó phải thực hiện từng chức năng ảo, có nghĩa là nó hỗ trợ giao diện được khai báo bởi ABC. Không thể ghi đè một hàm ảo thuần túy trong một lớp dẫn xuất, sau đó cố gắng khởi tạo các đối tượng của lớp đó, là một lỗi biên dịch.

Các lớp có thể được sử dụng để khởi tạo các đối tượng được gọi là các lớp cụ thể.

## **Ví dụ**

Consider the following example where parent class provides an interface to the base class to implement a function called **getArea()** −

#include <iostream>

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 {

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;

}

Kết quả :

Total Rectangle area: 35

Total Triangle area: 17

Bạn có thể thấy cách một lớp trừu tượng định nghĩa một giao diện theo phương thức getArea() và hai lớp khác thực hiện cùng chức năng nhưng với thuật toán khác nhau để tính diện tích cụ thể cho hình dạng.

## **Chiến lược thiết kế**

Một hệ thống hướng đối tượng có thể sử dụng một lớp cơ sở trừu tượng để cung cấp một giao diện chung và chuẩn hóa phù hợp cho tất cả các ứng dụng bên ngoài. Sau đó, thông qua kế thừa từ lớp cơ sở trừu tượng đó, các lớp dẫn xuất được hình thành hoạt động tương tự.

Các khả năng (nghĩa là các hàm công khai) được cung cấp bởi các ứng dụng bên ngoài được cung cấp dưới dạng các hàm ảo thuần túy trong lớp cơ sở trừu tượng. Việc triển khai các chức năng ảo thuần túy này được cung cấp trong các lớp dẫn xuất tương ứng với các loại ứng dụng cụ thể.

Kiến trúc này cũng cho phép dễ dàng thêm các ứng dụng mới vào hệ thống, ngay cả sau khi hệ thống đã được xác định.