Moștenirea permite definirea de clase noi (clase derivate) reutilizând clase existente (clasă de bază).

Clasa nou creată moștenește comportamentul (metode) și caracteristicile (variabile membre, starea) de la clasa de bază.

Dacă A și B sunt două clase unde B moștenește de la clasa A (B este derivat

din clasa A sau clasa B este o specializare a clasei A) atunci:

• clasa B are toate metodele si variabilele membre din clasa A

• clasa B poate redefini metode din clasa A

• clasa B poate adaugă noi membrii (variabile, metode) pe lângă cele moștenite de la clasa A.

clasa B = Clasă de bază (superclass, base class, parent class)

clasa A = Clasă derivată (subclass, derived class, descendent class)

membrii (metode, variabile) moșteniți = membrii definiți în clasa A și nemodificați în clasa B

membrii redefiniți (overridden) = definit în A și în B (în B se crează o nouă definiție)

membrii adăugați = definiți doar în B

public: poate fi accesat de oriunde

private: poate fi accesat doar în interiorul clasei

protected: poate fi accesat în interiorul clasei și în clasele derivate.

Constructorii și destructorii nu sunt moșteniți.

Constructorul din clasa derivată trebuie sa apeleze constructorul din clasa de baza. Sa ne asigurăm ca obiectul este inițializat corect.

Similar și pentru destructor. Trebuie sa ne asiguram ca resursele gestionate de clasa de bază sunt eliberate.

Dacă nu apelăm explicit constructorul din clasa de bază, se apelează automat constructorul implicit.

Dacă nu exista constructor implicit se generează o eroare la compilare

Tipuri de relații de asociere

Asociere

Agregare (compoziție) (whole-part relation)

Dependență

Moștenire

Polimorfism

Proprietatea unor entități de:

• a se comporta diferit în funcție de tipul lor

• a reacționa diferit la același mesaj

Obiecte din diverse clase care sunt legate prin relații de moștenire să răspundă diferit la același mesaj (apel de metodă).

Object slicing

Daca asignam un object de tipul clasei derivate la o variabila de tipul clasei de baza, obiectul pierde partea adăugată de clasa derivata.

Slicing se întâmpla la orice copiere de acest fel: assignment, transmitere prin valoare, return prin valoare.

Pentru a evita slicing se pot folosi referințe (&) sau pointeri (\*).

Legare dinamică (Dynamic binding).

Legarea (identificarea) codului de executat pe baza numelui de metode se poate face:

• în timpul compilării => legare statică (static binding)

• în timpul execuției => legare dinamică (dynamic binding)

Legare dinamică:

• selectarea metodei de executat se face timpul execuției.

• Când se apelează o metodă, codul efectiv executat (corpul funcției) se alege la momentul execuției (la legare statică decizia se ia la compilare)

• legarea dinamica în C++ funcționează doar pentru referințe și pointeri

• În C++ doar metodele virtuale folosesc legarea dinamică

virtual <function-signature>

• metoda suprascrisă în clasele derivate are legarea dinamică activată

• metoda apelată se va decide în funcție de tipul actual al obiectului (nu în funcție de tipul declarat).

• Constructorul nu poate fi virtual – pentru a crea un obiect trebuie sa știm tipul exact

• Destructorul poate fi virtual (este chiar recomandat sa fie când avem ierarhii de clase)

Mecanism C++ pentru polimorfism

Orice obiect are atașat informații legate de metodele obiectului

Pe baza acestor informații apelul de metodă este efectuat folosind implementarea corectă (cel din tipul actual).

Orice obiect are referință la un tabel prin care pentru metodele virtuale se selectează implementarea corectă.

Orice clasă care are cel puțin o metodă virtuală (clasă polimorfică) are un tabel numit VTABLE (virtual table). VTABLE conține adrese la metode virtuale ale clasei.

Când invocăm o metodă folosind un pointer sau o referință compilatorul generează un mic cod adițional care în timpul execuției o sa folosească informația din VTABLE pentru a selecta metoda de executat.

Moștenire multiplă

În C++ este posibil ca o clasă să aibă multiple clase de bază, să moșteneasca de la mai multe clase

Clasa moștenește din toate clasele de bază toate atributele.

Moștenirea multipla poate fi periculoasă și în general ar trebui evitat

• se poate moșteni același atribut de la diferite clase

• putem avea clase de bază care au o clasă de bază comună

Funcții pur virtuale

Funcțiile pur virtuale nu sunt definite (avem doar declarația metodei).

Folosim metode pur virtuale pentru a ne asigura că toate clasele derivate (concrete) o sa definească metoda.

=0 indică faptul ca nu există implementare pentru această metodă în clasă.

Clasele care au metode pur virtuale nu se pot instanția

Clase abstracte

O clasă abstractă poate fi folosită ca și clasă de bază pentru o colecție de clase derivate;

Oferă:

• o interfață comună pentru clasele derivate (metodele pur virtuale se vor implementa în clasele derivate)

• pot conține atribute comune tuturor claselor derivate

o clasă abstractă nu are instanțe

o clasă abstractă are cel puțin o metodă pur virtuală:

virtual <return-type> <name> (<parameters>) = 0;

clasă pur abstractă = clasă care are doar metode pur virtuale

clasă pur abstractă = interfață

Moștenire. Polimorfism

De ce folosim moștenire:

Moștenire de implementare (Implementation Inheritance):

Clasa de baza oferă funcționalitate (metode, câmpuri) ce ușurează implementarea

clasei derivate. Folosim moștenire pentru a reutiliza codul din clasa de baza (Ex.

MemoryRepository -> FileRepository)

Moștenire de interfața (Interface Inheritance):

Obiectul de clasa derivata poate fi folosit oriunde se cere ceva de tipul clasei de baza

(cele doua expun același interfața). Folosim pentru a oferi un punct de extensie

(adăugare de funcționalitate noua fără a modifica codul existent). Ex: Undo

• reutilizare de cod

• clasa derivată moștenește din clasa de bază

• se evită copy/paste – mai ușor de întreținut, înțeles

• extensibilitate

• permite adăugarea cu ușurință de noi funcționalități

• extindem aplicația fără să modificăm codul existent