Suddivisione tra file .h e .cpp

File Header (.h)

Contiene la dichiarazione della classe, ovvero la sua interfaccia pubblica e privata. In questo file si definiscono gli attributi e i prototipi delle funzioni membro.

File di Implementazione (.cpp)

Contiene il codice effettivo delle funzioni dichiarate nel file .h. Qui vengono implementati i metodi della classe.

Person.h

```
#ifndef PERSON_H
#define PERSON_H
#include <st rin g>
class Person {
private:
        std ::s tring name;
       int age;
public:
       // Constr uctor
        Per son (st d:: string n, int a);
       // Getter methods
        std ::s tring getName() const;
       int getAge() const;
       // Setter methods
       void setNam e(s td: :string n);
       void setAge(int a);
       // Display method
       void display() const;
};
#endif
```

Utilizzo di header guard per evitare che il file venga incluso più volte

Person.cpp

```
#include "Person.h"
#include <io str eam>
// Constr uctor
Person ::P ers on( std ::s tring n, int a) :
name(n), age(a) {}
// Getter methods
std::s tring Person ::g etN ame() const {
        return name;
}
int Person ::g etAge() const {
        return age;
}
// Setter methods
void Person ::s etN ame (st d:: string n) {
       name = n;
}
void Person ::s etA ge(int a) {
       age = a;
}
// Display method
void Person ::d isp lay() const {
        std ::cout << " Name: " << name << ", Age:
" << age << std::endl;
}
```

main.cpp

```
#include <iostream>
#include " Per son.h"
int main() {
       // Create an instance of Person
        Person p1(" Ali ce", 25);
       // Display initial values
        p1.d is play();
       // Modify attributes
```


Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish Yours!
https://apollopad.com

main.cpp (cont)

```
>    p1.setName("Bob");
    p1.setAge(30);
    // Display updated values
    p1.display();
    return 0;
}
```

```
Name: Alice, Age: 25
Name: Bob, Age: 30
Name: Charlie, Age: 20
Student ID: 12345
```

Incapsulamento

L'incapsulamento è il principio OOP che **nasconde l'implementazione** di una classe e fornisce un'**interfaccia pubblica** per interagire con essa.

✓ Protegge i dati dall'accesso non autorizzato.

✓ Permette di modificare l'implementazione senza cambiare l'interfaccia pubblica.

✓ Aumenta la modularità e la manutenibilità del codice.

Tipo di Dato Astratto (ADT - Abstract Data Type)

Un **Tipo di Dato Astratto (ADT)** è un modello di dati che **definisce solo il comportamento** (interfaccia) senza rivelarne i dettagli interni.

ADT → Descrive solo cosa fa una struttura dati, senza preoccuparsi di come è implementata.

Incapsulamento → Protegge i dati interni e li rende accessibili solo attraverso metodi controllati.

Membri di una classe

I **membri di una classe** sono le componenti che definiscono le sue caratteristiche e il suo comportamento. Si dividono in due categorie principali:

Membri Dati (Attributi)

Sono le variabili che rappresentano lo stato dell'oggetto.

Membri Funzione (Metodi)

Sono le funzioni che definiscono il comportamento della classe, operando sui suoi membri dati.

Specificatori di accesso

private: accessibili solo all'interno della classe.

protected: accessibili nella classe e nelle sue derivate.

public: accessibili da qualsiasi parte del programma.

Specificatori di accesso predefiniti

La differenza principale tra `struct` e `class` riguarda gli **specificatori di accesso predefiniti**:

- `struct` → Accesso predefinito: public
- `class` → Accesso predefinito: private

Scope delle Classi

Scope (ambito) di una classe determina **dove e come** i suoi membri possono essere accessibili e utilizzati. Il concetto di scope è fondamentale per gestire la visibilità e l'organizzazione del codice.

Dentro una classe, i membri possono essere definiti con tre **specificatori di accesso**:

- public → Accessibile ovunque nel programma.
- protected → Accessibile solo dalle classi derivate.
- private → Accessibile solo all'interno della stessa classe.

Le classi possono essere definite dentro uno **spazio dei nomi (namespace)** per organizzare meglio il codice ed evitare conflitti.

Metodi Interni

I metodi sono definiti direttamente all'interno della dichiarazione della classe nel file header (`.h`).

✓ Vantaggi: Il compilatore può ottimizzare il codice (inline).

✕ Svantaggi: Il file header diventa più grande e potrebbe causare ricompilazioni non necessarie.

Se sono semplici e corti, per sfruttare la compilazione `inline`.

Metodi Esterni

I metodi sono dichiarati nel file header (`.h`) ma implementati separatamente in un file `.cpp`.

✓ Vantaggi: Migliora la separazione tra dichiarazione e implementazione, riduce la necessità di ricompilazione.

✕ Svantaggi: Il compilatore potrebbe non ottimizzarli come funzioni inline.

Se sono più complessi o se si vuole mantenere il codice più organizzato.

Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish Yours!
https://apollopad.com

Membro statico

```
class Counter {
public:
        static int count; // Membro statico
(condi viso)
};
int Counte r:: count = 0; // Defini zione del
membro statico
```

Metodo virtuale

Un **metodo virtuale** è un metodo dichiarato nella classe base con la parola chiave `virtual`, che può essere **sovrascritto** nelle classi derivate. Permette il **polimorfismo dinamico**, in modo che, quando viene chiamato tramite un puntatore o un riferimento alla classe base, venga eseguita l'implementazione della classe derivata, se presente. Questo meccanismo consente di definire comportamenti diversi a seconda del tipo reale dell'oggetto.

Membri Speciali

Costruttori: inizializzano gli oggetti della classe.

Operatori sovraccaricati: ridefiniscono operatori standard (`+`, `=`, `<<`, ecc.).

Costruttore di copia (`ClassN ame (const ClassN ame & othe r)`)

Operatore di assegnazione (`ClassN ame & operat or= (const ClassN ame & other)`)

Distruttore (`~Class Name()`)

Costruttore di spostamento (`ClassN ame (Cl ass Nam e& & o ther) noexcept`)

Operatore di assegnazione per spostamento (`ClassN ame & oper at or= (Cl ass Nam e& & other) noexcept`)

Funzione Membro combine come operatore +=

```
#include <iostream>
class Counter {
private:
       int value;
public:
       // Costru ttore
        Cou nte r(int v = 0) : value(v) {}
       // Funzione membro combine() in stile +=
>     Counter& combine(const Counter& other) {
       this->value += other.value; // Aggiunge il valore dell'altro
oggetto
       return *this; // Restituisce l'oggetto stesso
   }
   // Metodo per visualizzare il valore
   void display() const {
       std::cout << "Value: " << value << std::endl;
   }
};
```

La funzione membro `combine()` può essere progettata per modificare l'oggetto stesso e restituire una referenza ad esso, proprio come l'operatore `+=`.

Name Lookup

Il **name lookup** (ricerca dei nomi) è il processo con cui il compilatore trova la dichiarazione di una variabile, funzione o classe. Questo meccanismo è fondamentale per comprendere come vengono risolti i nomi nei diversi scope e nei contesti di **ereditarietà, namespace e classi annidate**.

Nelle classi, il compilatore cerca i nomi in questo ordine:

1. Classe derivata
2. Classe base
3. Namespace globale (se necessario)

La ricerca dei nomi segue la **regola dell'ambito più vicino (scope resolution)**:

1. Il compilatore cerca il nome **nel blocco locale**.
2. Se non lo trova, cerca negli **scope superiori (funzione, classe, namespace)**.
3. Se non esiste nel file, dà un **errore di compilazione**.

Membri di Tipo Puntatore

```
#include <iostream>
class Person {
private:
        std ::s tring* name; // Puntatore a una
stringa dinamica
```


Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish Yours!
https://apollopad.com

Membri di Tipo Puntatore (cont)

```
> public:
    // Costruttore
    Person(const std::string& n) {
        name = new std::string(n); // Allocazione dinamica
    }
    // Distruttore
    ~Person() {
        delete name; // Libera la memoria
    }
    void display() const {
        std::cout << "Name: " << *name << std::endl;
    }
};
int main() {
    Person p("Alice");
    p.display();
    return 0; // Il distruttore libera la memoria
}
```

Sono utilizzati per gestire **risorse dinamiche** come memoria allocata su heap, file, socket, ecc. Se non gestiti correttamente, possono causare **memory leaks** e **dangling pointers**.

✓ `new` alloca memoria per name.
✓ delete nel distruttore evita memory leaks.

Uso di std::unique_ptr

```
#include <iostream>
#include <me mor y> // Per unique_ptr
class Person {
private:
        std ::u niq ue_ ptr <st d:: str ing> name;
public:
        Per son (const std::s tri ng& n) :
name(s td: :ma ke_ uni que <st d:: str ing >(n)) {}
       void display() const {
```

Uso di std::unique_ptr (cont)

```
>       std::cout << "Name: " << *name << std::endl;
    }
};
int main() {
    Person p1("Alice");
    Person p2 = std::move(p1); // Usa il move constructor di
unique_ptr
    p2.display(); // Output: Name: Alice
    // p1.display(); // ✖ Errore: p1 è stato svuotato
    return 0;
}
```

Con smart pointers, non dobbiamo gestire `new` e `delete` manualmente.

✓ Gestisce automaticamente la memoria (non serve delete).
✓ Evita **shallow copy**, perché `unique_ptr` non è copiabile.

this in Funzioni Membro Costanti

In una funzione membro dichiarata `const`, `this` diventa un puntatore a un oggetto costante (`const Person*`), quindi non si possono modificare i dati membri.

Esempio di Method Chaining con this

```
class Person {
private:
        std ::s tring name;
       int age;
public:
        Per son (st d:: string n, int a) : name(n),
age(a) {}
       // Metodi che ritornano *this per concat -
enare chiamate
        Per son & setNam e(s td: :string n) {
                thi s->name = n;
                return *this;
       }
        Per son & setAge(int a) {
                thi s->age = a;
```


Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish Yours!
https://apollopad.com

Esempio di Method Chaining con this (cont)

```
>        return *this;
    }
    void display() const {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};
int main() {
    Person p1("Alice", 25);
    // Method chaining grazie a 'this'
    p1.setName("Bob").setAge(30).display(); // Output: Name: Bob,
Age: 30
    return 0;
}
```

`this` è spesso usato per ritornare l'oggetto corrente e permettere la chaining dei metodi.

Puntatore Implicito this

Il puntatore `this` è un puntatore implicito che ogni oggetto di una classe possiede e che punta a se stesso.

✓ È disponibile automaticamente in tutti i metodi non statici di una classe.

✓ Contiene l'indirizzo dell'oggetto corrente.

✓ Viene utilizzato per distinguere membri della classe da parametri con lo stesso nome e per ritornare l'oggetto corrente in metodi concatenati (method chaining).

Costruttori Sovraccaricati

```
#include <iostream>
class Person {
private:
        std ::s tring name;
       int age;
public:
    // 1 Costruttore predefinito
        Per son() : name("U nkn own "), age(0) {}
```

Costruttori Sovraccaricati (cont)

```
>    // 2 Costruttore con un parametro
    Person(std::string n) : name(n), age(0) {}
    // 3 Costruttore con due parametri
    Person(std::string n, int a) : name(n), age(a) {}
    // Metodo per mostrare i dati
    void display() const {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};
```

Il **sovraccarico dei costruttori permette di definire più costruttori** nella stessa classe, ognuno con parametri diversi.

Se una classe non ha un costruttore definito esplicitamente, il compilatore genera automaticamente un **costruttore predefinito sintetizzato**.

- Non prende parametri.
- Non inizializza esplicitamente i membri della classe (se non con valori predefiniti dei tipi).
- Viene creato automaticamente solo se non ci sono altri costruttori definiti.

Costruttore di Default Esplicito (= default)

```
class Example {
public:
        Exa mple() = default; // Richiede al
compil atore di generare il costru ttore di
default
};
```

Possiamo chiedere al compilatore di generare esplicitamente un **costruttore predefinito** usando `= default` che funziona esattamente come il costruttore predefinito sintetizzato.

Uso di = delete

l'operatore `= delete` impedisce l'uso di funzioni o operatori specifici.

✓ `= delete` impedisce l'uso di funzioni o operatori specifici.

✓ Blocca la copia e l'assegnazione, utile per classi con gestione esclusiva delle risorse.

✓ Evita conversioni implicite indesiderate.

✓ Può impedire l'allocazione dinamica (`new`).

Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish Yours!
https://apollopad.com

Copy & Move Semantics

```
#include <iostream>
#include <cs tri ng> // Per strcpy, strlen
class Person {
private:
        char* name;
public:
    // □ 1 Costruttore normale
        Per son (const char* n) {
               name = new char[s trl en(n) + 1];
                str cpy (name, n);
                std ::cout << " Con str uctor
called for " << name << std::endl;
       }
    // □ 2 Costruttore di copia (Deep Copy)
        Per son (const Person & other) {
               name = new char[s trl en( oth er.n -
ame) + 1]; // Nuova alloca zione
                str cpy (name, other.n ame);
                std ::cout << "Copy constr uctor
called for " << name << std::endl;
       }
    // □ 3 Operatore di assegnazione (Deep
Copy)
        Per son & operat or= (const Person &
other) {
               if (this != &o ther) { // Evita
auto-a sse gna zione
                      del ete[] name; // Libera
memoria esistente
                     name = new char[s trl en( -
oth er.n ame) + 1];
                      str cpy (name, other.n -
ame);
               }
                std ::cout << "Copy assignment
operator called for " << name << std::endl;
                return *this;
       }
    // □ 4 Costruttore di spostamento (Move
Constr uctor)
        Per son (Pe rso n& & other) noexcept {
               name = other.n ame; // Prende il
puntatore
```

Copy & Move Semantics (cont)

```
>       other.name = nullptr; // Resetta il vecchio oggetto
      std::cout << "Move constructor called" << std::endl;
   }
   // ➡5 Operatore di assegnazione per spostamento (Move
Assignment)
   Person& operator=(Person&& other) noexcept {
      if (this != &other) {
         delete[] name; // Libera la memoria esistente
         name = other.name; // Prende la risorsa
         other.name = nullptr; // Resetta l'oggetto originale
      }
      std::cout << "Move assignment operator called" << std::endl;
      return *this;
   }
   // ☐ Distruttore
   ~Person() {
      delete[] name;
      std::cout << "Destructor called" << std::endl;
   }
   void display() const { std::cout << "Person: " << (name ? name : "-
Empty") << std::endl; }
};
int main() {
   Person p1("Alice");
   Person p2 = p1; // Chiamata al costruttore di copia
   Person p3("Bob");
   p3 = p1; // Chiamata all'operatore di assegnazione
   Person p4 = std::move(p1); // Chiamata al costruttore di sposta-
mento
   p3 = std::move(p2); // Chiamata all'operatore di assegnazione per
spostamento
   return 0;
}
```

1 Costruttore di copia → Usato quando `p2 = p1;` per creare una copia separata.
2 Operatore di assegnazione → Usato quando `p3 = p1;` per assegnare i dati.
3 Costruttore di spostamento → Usato con `std::m ove(p1);` per evitare la copia.
4 Operatore di assegnazione per spostamento → Usato con `p3 = s td::m ove(p2);`.
5 Distruttore → Libera la memoria quando gli oggetti escono dallo scope.

Sponsored by ApolloPad.com
Everyone has a novel in them. Finish Yours!
https://apollopad.com

Getter & Setter

```
#include <iostream>
class Person {
private:
        std ::s tring name;
public:
       // Setter
       void setNam e(s td: :string n) { name = n;
}
       // Getter
        std ::s tring getName() const { return
name; }
};
int main() {
        Person p;
        p.s etN ame ("Al ice ");
        std ::cout << " Name: " << p.getN ame() <<
std::endl; // Output: Name: Alice
        return 0;
}
```

✓ Mantengono l'incapsulamento.
✓ Possiamo validare i dati prima di modificarli.

Classi e Funzioni Amiche (friend)

```
#include <iostream>
class Person {
private:
        std ::s tring name;
public:
        Per son (st d:: string n) : name(n) {}
       // Dichia razione della funzione amica
        friend void showPe rso n(const Person &
p);
};
// Defini zione della funzione amica
void showPe rso n(const Person & p) {
        std ::cout << " Friend Function - Name: "
<< p.name << std::endl;
```

Classi e Funzioni Amiche (friend) (cont)

```
> }
int main() {
   Person p("Bob");
   showPerson(p); // Output: Friend Function - Name: Bob
   return 0;
}
```

Le classi amiche e funzioni amiche possono accedere ai membri privati di una classe senza bisogno di getter/setter.

✓ Una funzione amica **non è un membro della classe**, ma può accedere ai suoi dati privati.
✓ Può essere una funzione normale o un metodo di un'altra classe.

Ereditarietà

Tipo di Eredit-arietà	Membri public	Membri protected	Membri private
public	Rimangono public	Rimangono protected	Non accessibili
protected	Diventano protected	Rimangono protected	Non accessibili
private	Diventano private	Diventano private	Non accessibili

L'ereditarietà permette di creare una nuova classe (`classe deriv ata`) basata su un'altra (`classe base`).

Esempio: Ereditarietà public cpp Copia Modifica

```
#include <iostream>
class Person {
protected: // Access ibile dalle classi derivate
        std ::s tring name;
public:
        Per son (st d:: string n) : name(n) {}
       void display() const { std::cout << " Name:
" << name << std::endl; }
```


Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish Yours!
https://apollopad.com

Esempio: Ereditarietà public cpp Copia Modifica (cont)

```
> };
// Classe derivata
class Student : public Person {
private:
    int studentID;
public:
    Student(std::string n, int id) : Person(n), studentID(id) {}
    void show() const {
        display(); // Chiamata al metodo della classe base
        std::cout << "Student ID: " << studentID << std::endl;
    }
};
int main() {
    Student s("Alice", 1234);
    s.show();
    // Output:
    // Name: Alice
    // Student ID: 1234
    return 0;
}
```

✓ Student eredita Person con ereditarietà public, quindi display() è accessibile.
✓ name è protected, quindi accessibile nella classe derivata.

Esempio: Ereditarietà private cpp Copia Modifica

```
class Employee : private Person {
private:
       int employ eeID;
public:
        Emp loy ee( std ::s tring n, int id) :
Person(n), employ eeI D(id) {}
       void show() const {
                dis play(); // Possibile perché
ereditiamo private (membri public diventano
private)
```

Esempio: Ereditarietà private cpp Copia Modifica (cont)

```
>       std::cout << "Employee ID: " << employeeID << std::endl;
    }
};
int main() {
    Employee e("Bob", 5678);
    e.show();
    // e.display(); ✖ ERRORE! (diventa private in Employee)
    return 0;
}
```

✓ Con private, i membri `public` della classe base diventano privati in `Employee`.
✓ `display()` può essere usato solo dentro `Employee`, ma non dall'esterno.

Ereditarietà Multilivello

```
#include <iostream>
class Person {
protected:
        std ::s tring name;
public:
        Per son (st d:: string n) : name(n) {}
       void display() const { std::cout << " Name:
" << name << std::endl; }
};
// Student eredita da Person
class Student : public Person {
protected:
       int studentID;
public:
        Stu den t(s td: :string n, int id) :
Person(n), studen tID(id) {}
};
// Gradua teS tudent eredita da Student
class Gradua teS tudent : public Student {
private:
```


Sponsored by ApolloPad.com
Everyone has a novel in them. Finish Yours!
https://apollopad.com

Ereditarietà Multilivello (cont)

```
>    std::string thesisTitle;
public:
    GraduateStudent(std::string n, int id, std::string thesis)
        : Student(n, id), thesisTitle(thesis) {}
    void show() const {
        display(); // Da Person
        std::cout << "Student ID: " << studentID << std::endl;
        std::cout << "Thesis: " << thesisTitle << std::endl;
    }
};
int main() {
    GraduateStudent g("Bob", 5678, "AI Research");
    g.show();
    return 0;
}
```

Una classe può ereditare da una derivata, propagando i membri lungo la gerarchia.

✓ `Gradua teS tudent` eredita da `Student`, che eredita da `Pers on`.
✓ `display()` è chiamato da `Gradua teS tudent`, anche se appartiene a Person`.
✓ `studentID` è protetto, quindi accessibile da `Gradua teS tuden t`, ma non da `main()`.

Ereditarietà Multipla

```
#include <iostream>
class Athlete {
protected:
        std ::s tring sport;
public:
        Ath let e(s td: :string s) : sport(s) {}
       void showSp ort() const { std::cout << " -
Sport: " << sport << std::endl; }
};
class Student {
protected:
```

Ereditarietà Multipla (cont)

```
>    int studentID;
public:
    Student(int id) : studentID(id) {}
    void showID() const { std::cout << "Student ID: " << studentID <<
std::endl; }
};
// Classe che eredita sia da Athlete che da Student
class StudentAthlete : public Athlete, public Student {
public:
    StudentAthlete(std::string s, int id) : Athlete(s), Student(id) {}
    void show() const {
        showSport();
        showID();
    }
};
int main() {
    StudentAthlete sa("Basketball", 2023);
    sa.show();
    return 0;
}
```

✓ `Studen tAt hlete` eredita sia da `Athlete` che da `Student`.
✓ Può chiamare i metodi di entrambe le classi (`showSp ort()` e `s howID()`).

Sponsored by **ApolloPad.com**
Everyone has a novel in them. Finish Yours!
https://apollopad.com