Domanda 1

Dato il seguente codice:

```C/C++
class Base {
protected:
    int x = 1;
public:
    void printX() { std::cout << x; }
};

class Derivata : private Base {
public:
    void accessX() {
        // Punto A
        std::cout << x;
    }
};
```

Qual è la **visibilità** del membro `x` (definito come `protected` in `Base`) all'interno del metodo `accessX()` della classe `Derivata`?

A. `public`. È accessibile in modo normale.
B. `protected`. È accessibile all'interno di `Derivata`, ma non da oggetti esterni a `Derivata`.
C. `private`. La clausola `private` nell'ereditarietà rende tutti i membri ereditati `private` in `Derivata`.
D. Non è accessibile, causerebbe un errore di compilazione.

Domanda 2

Data la seguente gerarchia:

```C/C++
class Base {
protected:
    void method1() {}
};
```

```
class Derivata : protected Base {
public:
    void method2() { method1(); }
};

int main() {
    Derivata d;
    // Punto B
    // d.method1();
}
```

Cosa succede quando si tenta di chiamare `d.method1()` (Punto B) dall'esterno della classe `Derivata`?

A. La chiamata è valida, poiché `method1` era `protected` in `Base`.
B. La chiamata è valida, poiché `method2` è `public` e può accedervi.
C. Errore di compilazione. `method1` diventa `protected` in `Derivata` e non è accessibile al di fuori della classe.
D. Errore di compilazione, ma solo se `method1` fosse stato `private` in `Base`.

Domanda 3:

```C/C++
class A {
private:
    int value = 5;
};

class B : public A {
public:
    void printValue() {
        // Punto C
        // std::cout << value;
    }
};
```

Cosa si verifica quando la classe `B` tenta di accedere al membro `value` della classe `A` (Punto C)?

A. Nessun problema, l'ereditarietà `public` consente l'accesso.
B. L'accesso è consentito perché `value` è un membro della classe base.

C. Errore di compilazione. I membri `private` di una classe base non sono mai accessibili direttamente nelle classi derivate.
D. È necessario utilizzare la risoluzione dell'ambito (`A::value`) per accedervi.

Domanda 4:

Date due classi `Base` e `Derivata` (`Derivata : public Base`), in quale ordine vengono eseguiti i loro costruttori e distruttori quando viene creato e poi distrutto un oggetto di tipo `Derivata`?

A. Costruttore `Derivata`, Costruttore `Base`, Distruttore `Base`, Distruttore `Derivata`.
B. Costruttore `Base`, Costruttore `Derivata`, Distruttore `Derivata`, Distruttore `Base`.
C. Costruttore `Base`, Costruttore `Derivata`, Distruttore `Base`, Distruttore `Derivata`.
D. Distruttore `Derivata`, Distruttore `Base`, Costruttore `Base`, Costruttore `Derivata`.

Domanda 5:

```C/C++
class A {
protected:
    int valA = 10;
};
class B : public A { /* ... */ };
class C : private B {
public:
    void printA() {
        // Punto E
        std::cout << valA;
    }
};
```

Il codice al Punto E è valido?

A. No, l'ereditarietà tra C e B è private, quindi valA non è accessibile.
B. B. No, valA è protected e non può mai essere stampato.
C. Sì. In B, valA è protected. In C, a causa dell'ereditarietà private da B, valA diventa private in C, ma è comunque accessibile all'interno dei metodi di C.
D. No, valA è definito troppo in alto nella gerarchia.

Domanda 6:

In C++, un membro dichiarato `protected` è accessibile:

A. Solo all'interno della classe in cui è definito.
B. All'interno della classe, dalle classi derivate, e da codice esterno tramite puntatori/oggetti.
C. All'interno della classe e dalle classi direttamente o indirettamente derivate.

D. All'interno della classe, dalle classi derivate e da classi amiche.

Domanda 7

```C/C++
class A {
public:
    int x = 1;
};

class B : public A {
public:
    int x = 2;
    void print() {
        // Punto F
        std::cout << x << " " << A::x;
    }
};
```

Qual è l'output del metodo `print()` (Punto F) se chiamato su un oggetto `B`?

A. `1 1`
B. `2 1`
C. `1 2`
D. `2 2`

Domanda 8

Dato il seguente codice:

```C/C++
class Base {
public:
    void display() { std::cout << "Base"; }
};

class Derivata : public Base {
public:
    void display() { std::cout << "Derivata"; }
};
```

```
void check(Base& b) {
    b.display(); // Punto G
}

int main() {
    Derivata d;
    check(d);
}
```

Qual è l'output del codice al Punto G?

A. `Derivata`
B. `Base`
C. Errore di compilazione
D. Errore di runtime

Domanda 9:

Riprendendo la domanda precedente, ma aggiungendo `virtual`:

```C/C++
class Base {
public:
    virtual void display() { std::cout << "Base"; }
};
// ... Derivata come prima

void check(Base& b) {
    b.display(); // Punto H
}

// ... main come prima
```

Qual è l'output del codice al Punto H?

A. `Base`
B. `Derivata`
C. Errore di compilazione
D. Errore di runtime

Domanda 10

Perché è fondamentale che il distruttore della classe base sia dichiarato `virtual` quando si usa il polimorfismo (ovvero si cancella un oggetto derivato tramite un puntatore alla classe base)?

A. Per garantire che venga chiamato solo il distruttore della classe base.
B. Per impedire l'object slicing durante la distruzione.
C. Per garantire che venga chiamato il distruttore della classe derivata e, successivamente, quello della classe base.
D. I distruttori virtuali sono richiesti solo in presenza di funzioni virtuali pure.

Domanda 11

Qual è lo scopo principale dello specificatore `override` (opzionale ma consigliato) in C++?

A. Forzare la funzione a essere `virtual`.
B. Dichiarare che la funzione può essere sovrascritta.
C. Garantire al compilatore che il metodo sta effettivamente sovrascrivendo un metodo virtuale della classe base (e segnalare un errore in caso contrario).
D. Rendere la funzione non sovrascrivibile in ulteriori classi derivate.

Domanda 12

```C/C++
class Figura {
public:
    virtual double area() = 0; // Punto I
};
```

Cosa implica l'uguaglianza `= 0` nella dichiarazione del metodo `area()` (Punto I)?

A. La funzione restituisce zero.
B. La funzione non può essere ridefinita.
C. Rende `area()` una funzione virtuale pura e `Figura` una classe astratta, che non può essere istanziata.
D. La funzione deve essere definita nel file sorgente (`.cpp`) e non nell'intestazione.

Domanda 13

Qual è la regola quando si chiama un metodo `virtual` all'interno del **costruttore** di una classe base?

A. Viene eseguito il metodo della classe derivata se l'oggetto è in fase di costruzione.
B. Causa un errore di compilazione.
C. Viene sempre eseguito il metodo definito nella classe base stessa (binding statico).

D. Viene eseguito il metodo della classe derivata se è dichiarato con `override`.

Domanda 14

Dato il seguente codice:

```C/C++
class Base {
public:
    int b = 1;
};

class Derivata : public Base {
public:
    int d = 2;
};

int main() {
    Derivata d_obj;
    Base b_obj = d_obj; // Punto K: Assegnazione per valore

    // std::cout << b_obj.d; // Tentativo di accedere a 'd'
}
```

Cosa succede al Punto K (Assegnazione per valore) e quale porzione dell'oggetto `d_obj` viene copiata in `b_obj`?

A. Viene copiata l'intera `d_obj` e `b_obj` contiene sia `b` che `d`.
B. Si verifica l'Object Slicing: solo la porzione `Base` di `d_obj` (il membro `b`) viene copiata in `b_obj`, e il membro `d` viene "tagliato" (slicing).
C. La classe `Base` deve avere un costruttore di copia per consentire l'assegnazione.
D. Il codice al Punto K è valido, ma l'accesso a `b_obj.d` causerebbe un errore di runtime.

Domanda 15

Dato il seguente codice:

```C/C++
class A { /* ... */ };
class B : public A { /* ... */ };
```

```
class C { /* ... */ };

int main() {
    B b_obj;
    C c_obj;

    A* ptr1 = &b_obj; // Linea 1
    B* ptr2 = &b_obj; // Linea 2
    A* ptr3 = &c_obj; // Linea 3
}
```

Quale riga causerà un **errore di compilazione**?

A. Linea 1
B. Linea 2
C. Linea 3
D. Nessuna

Domanda 16

```C/C++
class Base { /* ... */ };
class Derivata : public Base {
public:
    void special() {}
};

int main() {
    Derivata d_obj;
    Base* b_ptr = &d_obj;

    // Punto L
    Derivata* d_ptr = b_ptr;

    d_ptr->special();
}
```

Cosa succede al Punto L, dove si tenta un **downcasting implicito**?

A. L'assegnazione è valida e il codice funziona. B.

B. Errore di compilazione. Il downcasting (da Base a Derivata) non è consentito implicitamente e richiede un cast esplicito (e insicuro).
C. L'assegnazione è valida solo se `Base` ha un metodo `virtual`.
D. Il codice funziona, ma se `b_ptr` non puntasse effettivamente a `Derivata` causerebbe un errore di runtime.

Domanda 17

Quale tipo di ereditarietà (pubblica, protetta o privata) è **necessario** affinché un Upcasting (assegnare un puntatore/riferimento di `Derivata` a un puntatore/riferimento di `Base`) sia consentito?

A. Ereditarietà `protected` o `public`.
B. Solo ereditarietà `protected`.
C. Ereditarietà `private` o `public`.
D. Solo ereditarietà `public`.