🧪 Test di Programmazione C++ – Debugging (Argomenti di base)

Istruzioni per lo studente

Per ogni esercizio:

1. Individua **l'errore o gli errori** presenti nel codice
2. Spiega **perché il codice è errato**
3. Scrivi **la versione corretta**

🔹 Esercizio 1 – Tipi di dato e variabili semplici

```
#include <iostream>
using namespace std;

int main() {
    int x = 5.7;
    cout << "Valore di x: " << x << endl;
    return 0;
}
```

Domanda:
Dov'è l'errore? Come può essere corretto?

✅ Soluzione

- Errore logico: `5.7` è un `double`, assegnato a `int`
- Perdita di precisione

Correzioni possibili:

```
double x = 5.7;
```

oppure

```
int x = static_cast<int>(5.7);
```

oppure

```
int x = (int)5.7;
```

🔹 Esercizio 2 – `struct`

```
#include <iostream>
#include <string>
using namespace std;

struct Studente {
    string nome;
    int eta;
};

int main() {
    Studente s;

    // Se nome fosse un vettore di caratteri, allora dovrei fare cosi strcpy(s.nome, “Mario”), oppure inizializzare carattere per carattere

    s.nome = "Mario";
    s.eta = 20;
    cout << s.nome << " " << s.eta << endl;
}
```

✅ Soluzione

- Errore di tipo: `"20"` è una stringa, `eta` è `int`

Correzione:

```
s.eta = 20;
```

🔹 Esercizio 3 – Vettori (array)

```
#include <iostream>
using namespace std;

int main() {
    int v[5];
    for (int i = 0; i < 5; i++) {
        v[i] = i * 2;
    }
      return 0;
}
```

✅ Soluzione

- Errore di **out of bounds**
- Gli indici vanno da `0` a 4

Correzione:

```
for (int i = 0; i < 5; i++)
```

🔹 Esercizio 4 – Matrici

```
int m[3][3];

for (int i = 0; i < 3; i++)
    for (int j = 0; j < 4; j++)
        m[i][j] = i + j;
```

✅ Soluzione

- La seconda dimensione è 3, non 4

Correzione:

```
for (int j = 0; j < 3; j++)
```

- 2) P = indirizzo = ???, valore = ??? X= dimensione = 4 byte, indirizzo = 0x00000000, valore = 10
- 3) P = indirizzo = ???, valore = 0x00000000 X= dimensione = 4 byte, indirizzo = 0x00000000, valore = 10
- 4) P = indirizzo = ???, valore = 0x00000000 X= dimensione = 4 byte, indirizzo = 0x00000000, valore = 11

🔹 Esercizio 5 – Puntatori

```
1: int x = 10; int c = 0;
2: int* p;
3: p = &x;
4: *p = 11;
5: p = &c;
```

✅ Soluzione

- p non punta a nessun indirizzo valido

Correzione:

```
int* p;
p = &x;
```

🔹 Esercizio 6 – `if`

```
int x = 5;

if (x = 10) {
    cout << "x vale 10";
}
```

✅ Soluzione

- Uso di = invece di ==
- Errore logico

Correzione:

```
if (x == 10)
```

🔹 Esercizio 7 – `switch case`

```
int scelta = 2;

switch (scelta) {
    case 1:
        cout << "Uno";
           break;
    case 2:
        cout << "Due";
           break;
    case 3:
           cout << “Tre”;
           break;
     default:
           cout << “Default”;
           break;
}
. . . continua
```

✅ Soluzione

- Mancanza di `break` nel `case 1`
- Fall-through non voluto

Correzione:

```
case 1:
    cout << "Uno";
    break;
```

🔹 Esercizio 8 – Ciclo `while`

```
int i = 0;

while (i < 5) {
     cout << i++ << endl;
     //Equivalente ad
     //cout << i << endl;
     //i++;

     //cout << ++i << endl;
     //Equivalente ad
     //i++;
     //cout << i << endl;

}
```

✅ Soluzione

- `i` non viene mai incrementato
- Ciclo infinito

Correzione:

```
i++;
```

🔹 Esercizio 9 – `do while`

```
int x = 10;

do {
    cout << 10 - x << endl;
     x--;
```

```
} while (x > 5);
```

✅ Soluzione

- Nessun errore sintattico
- Errore logico: la condizione è sempre falsa dopo la prima esecuzione

Possibile correzione:

```
x--;
```

🔹 Esercizio 10 – `for each` (range-based for)

```
#include <vector>
using namespace std;

vector<int> v = {1, 2, 3};

//Ad ogni iterazione, il compilatore assegna alla variabile I
l'elemento della collection ad indice index.

for (int i = 0; i < v.size(); i++) {
    v[i] = v[i] * 2;
}


for (int& i : v) {
    i = i * 2;
}
… vettore modificato
```

✅ Soluzione

- `i` è una **copia**, non modifica il vettore

Correzione:

```
for (int& i : v)
```

🔹 Esercizio 11 – `vector<>`

```
vector<int> v(10);
for(int e: v){
      cout << e << endl;
}

cout << “Il primo elemento e’ ” << v.front() << endl;
cout << “L’ultimo elemento e’ ” << v.back() << endl;

//Basta vector
for(auto i = v.begin(); i < v.end(); i++){
      cout << *i << endl;
}

//Serve algorithm
for(auto i = std::begin(v); i < std::end(v); i++){
      cout << *i << endl;
}



v.push_back(10); //v[0] = 10;
v.push_front(20); // v[1] = 10, v[0] = 10l
int c = v.pop_back(); //c == 10
int f = v.pop_front(); // f == 20
```

✅ Soluzione

- Il vettore è vuoto
- Accesso fuori dai limiti

Correzione:

```
v.push_back(10);
```

🔹 Esercizio 12 – Smart Pointer (`unique_ptr`)

```
#include <memory>
using namespace std;

unique_ptr<int> p1 = new int(5);
```

✅ Soluzione

- Costruzione errata di `unique_ptr`

Correzione:

```
auto p1 = make_unique<int>(5);
```

🔹 Esercizio 13 – `fstream` / `ofstream`

```
#include <fstream>
using namespace std;

ofstream file("dati.txt");
if (!file){
      cout << “Impossibile aprire il file” << endl;

} else {
      file << "Ciao";
}
```

✅ Soluzione

- Nessun controllo sull’apertura del file

Correzione consigliata:

```
if (file.is_open()) {
    file << "Ciao";
    file.close();
}
```

🔹 Esercizio 14 – Lettura file (`ifstream`)

```
ifstream file("input.txt");
//controllo apertura file

string parola;
file >> parola;
```

✅ Soluzione

- Mancato controllo apertura file

Correzione:

```
if (file.is_open()) {
    file >> parola;
    file.close();
}
```

🔹 Esercizio 15 – Memoria dinamica (`new` / `delete`)

```
int* p = new int;
*p = 10;
delete p;
p = nullptr;

…

cout << *p;
```

✅ Soluzione

- Accesso a memoria **dopo delete** (dangling pointer)

Correzione:

```
delete p;
p = nullptr;
```

🔹 Esercizio 16 – `new[]` / `delete[]`

```
int* v = new int[5];

for (int i = 0; i < 5; i++)
    v[i] = i;

delete v;
```

✅ Soluzione

- Errore: `delete` invece di `delete[]`

Correzione:

```
delete[] v;
```

🔹 Esercizio 17 – Passaggio per valore

```
void incrementa(int x) {
    x++;
}

int main() {
    int a = 5;
    incrementa(a);
    cout << a;
}
```

✅ Soluzione

- x è una copia → a non cambia

Correzione:

```
void incrementa(int& x)
```

🔹 Esercizio 18 – Passaggio per indirizzo (puntatore)

```
void azzera(int* p) {
    *p = 0;
}

int main() {
    int x = 10;
    azzera(x);
}
```

✅ Soluzione

- Errore di tipo: serve l'indirizzo

Correzione:

```
azzera(&x);
```

🔹 Esercizio 19 – Array passato a funzione

```
void stampa(int v[]) {
    for (int i = 0; i < 10; i++)
        cout << v[i] << endl;
}

int main() {
    int a[5] = {1,2,3,4,5};
    stampa(a);
}
```

✅ Soluzione

- Accesso oltre i limiti dell'array

Correzione:

```
void stampa(int v[], int n)
```

🔹 Esercizio 20 – Matrice passata a funzione

```
void inizializza(int m[][]) {
    m[0][0] = 1;
}
```

✅ Soluzione

- Dimensione delle colonne obbligatoria

Correzione:

```
void inizializza(int m[][3])
```

🔹 Esercizio 21 – `vector` passato a funzione (errore logico)

```
void raddoppia(vector<int> v) {
    for (int& x : v)
        x *= 2;
}
```

✅ Soluzione

- v passato **per valore**
- Modifiche perse

Correzione:

```
void raddoppia(vector<int>& v)
```

🔹 Esercizio 22 – Manipolazione errata di `vector`

```
vector<int> v(5);

for (int i = 0; i <= v.size(); i++)
    v[i] = i;
```

✅ Soluzione

- Condizione errata: <=

Correzione:

```
i < v.size()
```

🔹 Esercizio 23 – `string` e accesso fuori indice

```
string s = "C++";

for (int i = 0; i <= s.length(); i++)
    cout << s[i];
```

✅ Soluzione

- Ultimo indice non valido

Correzione:

```
i < s.length()
```

🔹 Esercizio 24 – `string` passata a funzione

```
void aggiungi(string s) {
```

```
    s += "!";
}

int main() {
    string testo = "Ciao";
    aggiungi(testo);
    cout << testo;
}
```

✅ Soluzione

- Passaggio per valore

Correzione:

```
void aggiungi(string& s)
```

🔹 Esercizio 25 – Lettura CSV (errore di logica)

```
ifstream file("dati.csv");
int x;
char c;

while (file >> x >> c) {
    cout << x << endl;
}
```

✅ Soluzione

- Il carattere c non viene verificato
- CSV tipico: `10,20,30`

Correzione:

```
file >> x;
file >> c; // separatore
```

🔹 Esercizio 26 – CSV → `vector<int>`

```
vector<int> dati;
ifstream file("numeri.csv");

int n;
char sep;

while (!file.eof()) {
    file >> n >> sep;
    dati.push_back(n);
}
```

✅ Soluzione

- Uso scorretto di `eof()`

Correzione:

```
while (file >> n) {
    dati.push_back(n);
    file >> sep;
}
```

🔹 Esercizio 27 – CSV carattere per carattere

```
char c;
vector<char> separatori;

while (file >> c) {
    if (c == ',')
        separatori.push_back(c);
}
```

✅ Soluzione

- Nessun errore sintattico

- Migliorabile: leggere con `get()`

Correzione consigliata:

```
while (file.get(c))
```

🔹 Esercizio 28 – Allocazione dinamica di matrice

```
int** m = new int*[3];
for (int i = 0; i < 3; i++)
    m[i] = new int[3];

delete m;
```

✅ Soluzione

- Deallocazione incompleta

Correzione:

```
for (int i = 0; i < 3; i++)
    delete[] m[i];
delete[] m;
```

🔹 Esercizio 29 – `vector` e riferimento invalido

```
vector<int> v;
int& r = v[0];
v.push_back(10);
```

✅ Soluzione

- Accesso a elemento inesistente

Correzione:

```
v.push_back(10);
```

```
int& r = v[0];
```

🔹 Esercizio 30 – Funzione con `vector` e `string`

```
void stampa(vector<string> v) {
    for (auto s : v)
        cout << s << endl;
}
```

✅ Soluzione

- Inefficienza: copia inutile

Correzione:

```
void stampa(const vector<string>& v)
```

🔹 Esercizio 31 – Lambda function: cattura errata

```
int x = 10;

auto f = []() {
    cout << x << endl;
};

f();
```

✅ Soluzione

- x non è catturata dalla lambda

Correzione:

```
auto f = [x]() {
```

oppure

```
auto f = [&x]() {
```

🔹 Esercizio 32 – Lambda che modifica una variabile

```
int cont = 0;

auto inc = [cont]() {
    cont++;
};

inc();
cout << cont;
```

✅ Soluzione

- `cont` catturata **per valore**
- La modifica non ha effetto esterno

Correzione:

```
auto inc = [&cont]() {
```

🔹 Esercizio 33 – Lambda con `std::sort`

```
vector<int> v = {4, 1, 3, 2};

sort(v.begin(), v.end(),
     [](int a, int b) {
         return a > b;
     });
```

✅ Soluzione

- **Nessun errore**
- Ordina in ordine decrescente

👉 Esercizio utile per verificare la **comprensione del comportamento**, non il debugging.

🔹 Esercizio 34 – `std::sort` su struct (errore)

```
struct Studente {
    string nome;
    int voto;
};

vector<Studente> v;

sort(v.begin(), v.end());
```

✅ Soluzione

- `Studente` non ha operatore < definito

Correzioni possibili:

1️⃣Lambda

```
sort(v.begin(), v.end(),
     [](const Studente& a, const Studente& b) {
         return a.voto < b.voto;
     });
```

2️⃣Overload <

🔹 Esercizio 35 – Classe senza costruttore valido

```
class Punto {
    int x, y;
};
```

```
int main() {
    Punto p(3, 4);
}
```

✅ Soluzione

- Costruttore non definito

Correzione:

```
class Punto {
    int x, y;
public:
    Punto(int a, int b) : x(a), y(b) {}
};
```

🔹 Esercizio 36 – Accesso a membri privati

```
class Rettangolo {
    int base, altezza;
};

int main() {
    Rettangolo r;
    r.base = 10;
}
```

✅ Soluzione

- `base` è `private` per default

Correzione:

```
public:
    void setBase(int b) { base = b; }
```

🔹 Esercizio 37 – Metodo non `const`

```
class Contatore {
    int valore;
public:
    int get() {
        return valore;
    }
};

void stampa(const Contatore& c) {
    cout << c.get();
}
```

✅ Soluzione

- `get()` non è `const`
- Non può essere chiamato su oggetto `const`

Correzione:

```
int get() const {
```

🔹 Esercizio 38 – Overloading operatore +

```
class Punto {
public:
    int x, y;
};

Punto operator+(Punto a, Punto b) {
    Punto r;
    r.x = a.x + b.x;
    r.y = a.y + b.y;
}
```

✅ Soluzione

- Manca il `return`

Correzione:

```
return r;
```

🔹 Esercizio 39 – Overloading << (ostream)

```
class Punto {
public:
    int x, y;
};

ostream operator<<(ostream os, Punto p) {
    os << p.x << "," << p.y;
    return os;
}
```

✅ Soluzione

- `ostream` deve essere passato **per riferimento**

Correzione:

```
ostream& operator<<(ostream& os, const Punto& p)
```

🔹 Esercizio 40 – `std::sort` con metodo di classe

```
class Studente {
public:
    string nome;
    int voto;

    bool confronta(const Studente& s) {
        return voto < s.voto;
    }
};

sort(v.begin(), v.end(), Studente::confronta);
```

✅ Soluzione

- Metodo **non statico**
- `sort` richiede una funzione libera o callable

Correzioni possibili:

1️⃣ Lambda

```
sort(v.begin(), v.end(),
     [](const Studente& a, const Studente& b) {
         return a.voto < b.voto;
     });
```

2️⃣ Metodo static

```
static bool confronta(const Studente& a, const Studente& b)
```

🔹 Esercizio 41 – Lambda che restituisce riferimento errato

```
auto f = []() -> int& {
    int x = 10;
    return x;
};
```

✅ Soluzione

- Restituisce riferimento a variabile locale distrutta

Correzione:

```
return 10;
```

oppure usare variabile statica (sconsigliato didatticamente)

🔹 Esercizio 42 – `sort` con stringhe (errore logico)

```
vector<string> nomi = {"Anna", "Luca", "Marco"};

sort(nomi.begin(), nomi.end(),
     [](string a, string b) {
         return a.length() > b.length();
     });
```

✅ Soluzione

- Funziona, ma copia inutilmente le stringhe

Correzione:

```
[](const string& a, const string& b)
```

🔹 Esercizio 43 – Classe con `vector` non inizializzato

```
class Registro {
    vector<int>* voti;
public:
    Registro() {
        voti = nullptr;
    }

    void aggiungi(int v) {
        voti->push_back(v);
    }
};
```

✅ Soluzione

- Dereferenziazione di puntatore nullo

Correzione:

```
vector<int> voti;
```

👉 **Usare composizione, non puntatori**

🔹 Esercizio 44 – `sort` + lambda che usa variabile esterna

```
int soglia = 18;

sort(v.begin(), v.end(),
     [](int a, int b) {
         return a < b && a > soglia;
     });
```

✅ Soluzione

- `soglia` non catturata

Correzione:

```
[soglia](int a, int b)
```

🔹 Esercizio 45 – Operatore == mal definito

```
class Punto {
public:
    int x, y;

    bool operator==(Punto p) {
        return x == p.x && y == p.y;
    }
};
```

✅ Soluzione

- Passaggio per valore inutile
- Metodo non `const`

Correzione:

```
bool operator==(const Punto& p) const
```