OOP21_B_05_Vidmar_Philipp

Schreiben Sie eine Klasse **TData** mit den Attributen **name (String)** und **data (char-Vector).** Um Sicherheit bei den Aussagen zu gewährleisten können Sie Konstruktoren (Normal, Copy, Move) verwenden.

```
class TData
{
public:
    std::string name;
    std::vector<char> data;

    TData() = default; //Konstruktor
    TData(const TData& copy) = default; //Copy
    TData(TData&& move) = default; //Move
    ~TData() = default; //Destruktor
```

a. Legen Sie eine **Variable a** vom Typ **TData** an. Weisen Sie einer weiteren neuen **Variable b** vom Typ **TData** die **Variable a** zu. - Wird bei der Zuweisung der Copy- oder der Move Konstruktor aufgerufen? (1)

```
//Aufgabe A
std::cout << " Aufgabe A" << std::endl;
TData a;
TData b = a;
//Antwort: Es wird der CopyKonstruktor ausgeführt
```

b. Schreiben Sie eine Methode, die **einen Parameter** vom Typ **TData** entgegennimmt. Rufen Sie diese Methode auf. - Wird bei der Übergabe des Parameters der Copy- oder der Move Konstruktor aufgerufen? (1)

```
//Aufgabe B
std::cout << " Aufgabe B" << std::endl;
a.Aufgabe_B_CBR(b);
a.Aufgabe_B_CBV(b);
//Antwort: Bei CBV wird der CopyKosntruktor aufgerufen bei CBR nicht.
```

c. Schreiben Sie eine Methode, die **keinen Parameter** entgegennimmt. Sie legt aber eine **lokale Variable** vom **Typ TData** an, befüllt diese mit Daten und gibt dieses Objekt dann über **return** zurück. Rufen Sie die Methode auf und weisen das Ergebnis einer neuen **TData**-Variable **c** zu. - Wird bei der Übergabe durch **return** der Copy- oder der Move Konstruktor aufgerufen? (1)

```
TData TData::Aufgabe_C()
{
    TData rTData;
    rTData.name = " Aufgabe C ";
    rTData.data.push_back('C');
    return rTData;
}
```

```
//Aufgabe C
std::cout << " Aufgabe C" << std::endl;
TData c = a.Aufgabe_C();
//Antwort: Es wird der MoveKonstruktor aufgerufen
```

d. **Unterdrücken** Sie nun jeweils die Erzeugung eines Default-Copy oder Move Constructors und testen Sie obige Methode. - Hat sich am Verhalten etwas geändert? (1)

```
TData::TData() { ... }
TData::TData(const TData& copy) { ... }
TData::TData(TData&& move) { ... }
TData::~TData() { ... }
```

```
//Aufgabe D
//Antwort: Das Verhalten ändert sich nicht.
```

e. Wenn ein **Move-Constructor** aufgerufen wird,… - Wie finden Sie nach dem Move die Datenelemente des Quellobjekts vor? - Wird trotzdem ein Destruktor aufgerufen? **(1)**
Es wird nicht der „default Move Kosntruktor“ verwendet (Aufgebe d):

```
//Aufgabe E
std::cout << " Aufgabe E" << std::endl;
TData e = Aufgabe_D();
//Die Daten des Quellobjkektes fidnen sich nicht vor;
//Es Wird ein Destruktor für das Quellobjekt aufgerufen;
```

f. Füllen Sie mehrere **TData**-Elemente in einen **vector<…>** und ein **array<…>** und iterieren Sie durch diese Collections. Klären Sie die Fragen: - Werden bei deren Verwendung auch Move und Copy-Operatoren aufgerufen? - Was passiert, wenn Sie Move- oder Copy-Konstruktoren ausdrücklich unterbinden oder wechselseitig forcieren? Können diese Collections trotzdem verwendet werden? Kommentieren Sie nicht funktionierenden Code aus. **(1)**

```
//Aufgabe F
std::cout << " Aufgabe F" << std::endl;
std::cout << " Vector erstellen: " << std::endl;
std::vector<TData> vectorF(10);
std::cout << " Array erstellen: " << std::endl;
std::array<TData, 10> arrayF;
//Antowrt: Es wird immer der Stadnard Kosntruktor aufgerufen.
//Antowrt: Da die Vrariabelen ganz normal Initialiseirt werden

vectorF.push_back(a);
//copy für elemennt a, Die vector elemente werden mit move verschoben
```

g. Schreiben Sie nun mindestens einen **Konstruktor** und **Destruktor**. - Hat sich dadurch an den Antworten der obigen Fragen etwas geändert? (1)

```
TData::TData() //Konstruktor
{
    std::cout << " Konstruktor " << std::endl;
    this->name = "---";
    this->data.clear();

}
```

```
TData::~TData() //Destruktor
{
    std::cout << " Destruktor " << std::endl;
}
```

h. Schreiben Sie eine eigenen **Copy-Constructor**, der das Objekt **in die Tiefe** kopiert. (Antwort als Code Snippet) **(1)**

```
TData::TData(const TData& copy) //Copy
{
    std::cout << " CopyKonstruktor " << std::endl;
    this->name = copy.name;
    this->data = copy.data;
}
```

i. Schreiben Sie einen eigenen **Move-Constructor**. (Antwort als Code Snippet) **(1)**

```
TData::TData(TData&& move) noexcept
{
    std::cout << " MoveKonstruktor " << std::endl;
    this->name = move.name;
    this->data = move.data;
}
```

j. Wie kann bei einer Zuweisung die Anwendung eines **Move-Constructors erzwungen** werden? (Antwort als Code Snippet) **(1)**

```
//Aufgabe J
TData j = std::move(a);
```

k. Fügen Sie Ihrer Klasse TData einen **static int id** hinzu und stellen Sie einen **getter** und **setter** für dieses Attribut zur Verfügung. - Müssen Sie nun auch den Code ihrer Copy- und Move-Constructor anpassen? **(1 Bonuspunkt)**

```
class TData
{
public:
    std::string name;
    std::vector<char> data;

    TData(); //Konstruktor
    TData(const TData& copy); //Copy
    TData(TData&& move) noexcept; //Move
    ~TData(); //Destruktor

    void Aufgabe_B_CBV(TData x);
    void Aufgabe_B_CBR(TData& x);
    TData Aufgabe_C();

    int getter();
    void setter(int idx);

private:
    static int id;
};
```

```
int TData::getter()
{
    return TData::id;
}

void TData::setter(int idx)
{
    TData::id = idx;
}
```

„Static int id“ wurde global (main) initialisiert dadurch mussten die Konstruktoren nicht angepasst werden.