Aufgabe 1 (Strings). **(5 Punkte)**

Schreiben Sie eine Funktion `print_string`, die als Argumente einen String (`std::string`) `s` sowie zwei einzelne Zeichen `c1` und `c2` erwartet. Die Funktion soll `s` auf der Konsole ausgeben und dabei jedes Vorkommen von `c1` durch `c2` ersetzen.

Beispiel: Der Aufruf `print_string("Hallo Welt",'e','a');` gibt `Hallo Walt` aus.

Lösung:

```
void print_string(std::string s, char c1, char c2) {
// Drei Lösungsvorschläge:
    for (char & c:s) { c = (c==c1?c2:c); }
//    for (int i=0; i<s.length();i++) { s[i] = (s[i]==c1?c2:s[i]); }
//    for (int i=0; i<s.length();i++) { if (s[i]==c1) s[i] = c2; }
    std::cout << s << "\n";
}
```

Aufgabe 2 (Referenzen, Polymorphie). **(6 Punkte)**

Betrachten Sie die beiden folgenden Klassen:

```
class haus {
protected:
    int m_hoehe = 13;
public:
    virtual int hoehe() { return m_hoehe; }
    virtual void ausbauen() { m_hoehe += 1; }
    int anzahl_fahrstuehle() { return 0; }
};

class hochhaus : public haus {
public:
    hochhaus() { m_hoehe = 42; }
    int anzahl_fahrstuehle() { return 4; }
};
```

Geben Sie die Ausgabe des folgenden Programms an:

```
void foo(haus h1, haus & h2) {
    std::cout << h1.anzahl_fahrstuehle() << "\n";
    std::cout << h2.anzahl_fahrstuehle() << "\n";
    h1.ausbauen();
    h2.ausbauen();
}
int main() {
    haus h1;
    hochhaus h2;

    std::cout << h1.hoehe() << "\n";
    std::cout << h2.hoehe() << "\n";
    foo(h1,h2);
    std::cout << h1.hoehe() << "\n";
    std::cout << h2.hoehe() << "\n";
    foo(h2,h1);
    std::cout << h1.hoehe() << "\n";
    std::cout << h2.hoehe() << "\n";

    return 0;
}
```

Lösung:

13 42 0 0 13 43 0 0 14 43

Aufgabe 3 (Klassen). **(3+5+7+4 Punkte)**

Betrachten Sie die folgende Klasse `menge`, die eine Menge im mathematischen Sinn repräsentieren soll:

```
class menge {
    std::vector<int> m_elemente;

public:
    bool leer();               // Liefert 'true', falls Menge leer.
    bool enthalten(int el);    // Liefert 'true' falls 'el' vorhanden.
    void entfernen(int el);    // Entfernt das Element 'el'.
    void hinzufuegen(int el); // Fügt 'el' als neues Element hinzu.
    menge schnitt(menge m);    // liefert den Schnitt mit 'm' zurück
};
```

a) Implementieren Sie die Methode `menge::hinzufuegen`. Die Methode soll ein neues Element zur Menge hinzufügen, falls es noch nicht vorhanden ist.

Hinweis: Sie können annehmen, dass die Methoden `leer`, `enthalten` und `entfernen` bereits implementiert sind. D.h. Sie dürfen sie benutzen, müssen aber nicht.

Lösung:

```
void menge::hinzufuegen(int el) {
    if (!enthalten(el)) m_elemente.push_back(el);
}
```

b) Implementieren Sie die Methode `menge::schnitt`. Die Methode erwartet als Argument eine zweite Menge `m` und liefert den Schnitt der eigenen Menge mit `m` als neues Objekt zurück.

Hinweis: Sie können annehmen, dass die Methoden `leer`, `enthalten`, `entfernen` und `hinzufuegen` bereits implementiert sind.

Lösung:

```
menge menge::schnitt(menge m) {
    menge ergebnis;
    for (auto el:m_elemente) {
        if (m.enthalten(el)) {
            ergebnis.hinzufuegen(el);
        }
    }
    return ergebnis;
}
```

c) Überladen Sie den Operator `+` für die Klasse `menge`. Das Ergebnis der Addition zweier Mengen soll die Vereinigung der Mengen sein.

Hinweis: Sie können annehmen, dass die Klasse `menge` bereits implementiert ist.

Lösung:

```
menge operator+(menge m1, menge m2) {
    int i = 0;
    while (!m1.leer()) {
        if (m1.enthalten(i)) {
            m1.entfernen(i);
            m2.hinzufuegen(i);
        }
        i++;
    }
    return m2;
}
```

d) Ändern Sie die Klassendeklaration so, dass statt `int` möglichst jeder beliebige Typ als Elemente der Menge zugelassen wird.

Hinweis: Die Methoden müssen Sie nicht neu implementieren.

Lösung:
Die Klassendefinition muss in ein Template umgewandelt werden, der Datentyp `int` innerhalb der Definition in den abstrakten Template-Parameter:

```
template<class T>
class menge {
    std::vector<T> m_elemente;

public:
    bool leer();
    bool enthalten(T el);
    void entfernen(T el);
    void hinzufuegen(T el);
    menge schnitt(menge m);
};
```

Aufgabe 4 (Fehlersuche). **(12 Punkte)**

Betrachten Sie das folgende Programm:

```
#include<iostream>
#include<string>

class foo {
    int i

public:
    foo(double k) i{0} {}
    void bar(int);
}

void bar(int j) { i = j; }

std::string schreibwas() (
    std::cout << "W";
    return "as";
}

int main {
    int k = 3.0;
    foo f(k);
    schreibwas();
    f.bar(42);
    return 0;
}
```

Korrigieren Sie die Fehler im Programm, so dass es kompiliert werden kann.

Hinweis: Das Programm enthält 6 Fehler, jede Zeile enthält höchstens einen Fehler. Falsch markierte Fehler führen zu Punktabzug.

Lösung:
Hier das korrigierte Programm, die Fehler sind neben den betreffenden Zeilen benannt:

```
#include<iostream>
#include<string>

class foo {
    int i;                          // Semikolon fehlt

public:
    foo(double k) : i{0} {}         // Doppelpunkt bei Initlist fehlt
    void bar(int);
};                                  // Semikolon fehlt

void foo::bar(int j) { i = j; } // 'foo::' vor 'bar' fehlt

std::string schreibwas() {          // Runde statt geschweifter Klammer
    std::cout << "W";
    return "as";
}

int main() {                        // Klammern bei 'main' fehlen
    int k = 3.0;
    foo f(k);
    schreibwas();
    f.bar(42);
    return 0;
}
```

Aufgabe 5 (Templates). **(4+4 Punkte)**

Betrachten Sie die beiden folgenden Funktions-Templates:

```
template<int N>
int foo() { return N * foo<N-1>(); }

template<>
int foo<0>() { return 1; }
```

a) Was gibt die folgende `main`-Funktion aus?

```
int main() {
    std::cout << foo<3>() << "\n";
    return 0;
}
```

Lösung:
Hier wird die Fakultätsfunktion mittels Funktionstemplates definiert, das Programm gibt die Zahl `6` auf der Konsole aus.

b) Welche Funktionen erzeugt der Compiler beim Übersetzen von `main`?

Lösung:
Es werden nacheinander die Funktionen `foo<3>`, `foo<2>`, `foo<1>` und `foo<0>` erzeugt. Für letztere wird das zweite Funktionstemplate benutzt, so dass keine weiteren Funktionen mehr rekursiv erzeugt werden.

Aufgabe 6 (Überladen von Funktionen). **(2+8 Punkte)**

Betrachten Sie die folgende Funktion `verbinden`:

```
std::string verbinden(std::string s1, std::string s2) {
    return s1 + s2;
```

a) Überladen Sie die Funktion, so dass sie auch drei Strings akzeptiert und deren Verkettung zurück gibt.

b) Überladen Sie die Funktion, so dass sie beliebige Anzahlen von Strings akzeptiert und deren Verkettung zurück gibt.

Hinweis: Überlegen Sie sich zunächst, auf welche Weise man beliebig viele Argumente an eine Funktion übergeben könnte.