Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungen für den 4. Tag

Aufgabe 1 Mehrfachvererbung

Nehmen Sie als Basis für diese Aufgabe die 1. Aufgabe aus der gestrigen Übung.

a) Schreiben Sie die Klasse *Employee*, die einen Mitarbeiter darstellt. *Employee* soll von *Person* erben und den Namen seines Vorgesetzten als Attribut beinhalten. Erweitern Sie auch entsprechend die Methode *getInfo()*.

b) Schreiben Sie nun eine Klasse *StudentAssistant* um eine wissenschaftliche Hilfskraft zu modellieren. Eine wissenschaftliche Hilfskraft ist ein Student und gleichzeitig auch ein Mitarbeiter. Dementsprechend soll *StudentAssistant* sowohl von *Student* als auch von *Employee* erben. Weitere Attribute sind nicht nötig. Überschreiben Sie *getInfo()*, um alle Daten auszugeben. Ändern Sie dazu die Sichtbarkeit der Attribute sowohl von *Student* als auch von *Employee* von private auf protected.

   Sie werden feststellen, dass sich die Klasse nicht kompilieren lässt, falls Sie das Attribut *name* direkt verwenden, da in einer *StudentAssistant*-Instanz zwei Instanzen von *Person* vorhanden sind - je eine von jeder Elternklasse. Deshalb müssen Sie mittels dem Scope-Operator :: angeben, welche Basis Sie genau meinen.

   ```
   Employee::name
   // or
   Student::name
   ```

   Testen Sie Ihre Implementierung, indem Sie das Ergebnis von *getInfo()* direkt in der *main* ausgeben.

c) Versuchen Sie nun, *printPersonInfo()* mit einer Instanz von *StudentAssistant* aufzurufen. Auch hier wird der Compiler mit einer Fehlermeldung abbrechen, da er nicht weiß, welche der beiden Basisklassen er nehmen soll. Diesmal ist es in C++ allerdings nicht mehr möglich, die Basisklasse zu spezifizieren, weshalb wir anders vorgehen werden. Wir werden dafür sorgen, dass *Person* nur ein Mal in *StudentAssistant* vorhanden ist.

   Lassen Sie dazu *Student* und *Employee* virtuell von *Person* erben. Sie werden feststellen, dass sich auch dieses Mal das Programm nicht fehlerfrei kompilieren lässt. Folgendes ist der Grund: Sowohl *Student* als auch *Employee* versuchen, einen Konstruktor von *Person* aufzurufen. Da *Person* aber nur ein einziges mal in *StudentAssistant* vorhanden ist, müsste der Konstruktor demnach zwei mal aufgerufen werden - ein mal von *Student* und ein mal von *Employee*. Dies würde jedoch grob gegen die Sprachprinzipien verstoßen. Deshalb wird der Konstruktor von *Person* weder von *Student* noch von *Employee* aufgerufen! Stattdessen müssen Sie in der Initialisierungsliste von *StudentAssistant* angeben, welcher Konstruktor von *Person* aufgerufen werden soll. Die Konstruktoraufrufe innerhalb von *Student* und *Employee* laufen stattdessen ins Leere, auch wenn sie syntaktisch vorhanden sind! Fügen Sie deshalb ein **Person(name)** in die Initialisierungsliste von *StudentAssistant* hinzu.

   Testen Sie Ihre Implementierung. Versuchen Sie auch Folgendes: Ändern Sie die Namen in den Konstruktoraufrufen von *Student* und *Employee* in der Initialisierungsliste von *StudentAssistant* und beobachten Sie die Ausgabe. Machen Sie sich dadurch klar, welche Probleme Mehrfachvererbung von implementierten Klassen verursachen kann!

Eine Alternative zur Implementierungsvererbung stellt Schnittstellenvererbung dar, wie es in Java üblich ist. Dabei werden Schnittstellen (Klassen mit ausschließlich abstrakten Methoden und ohne Attribute) definiert und nur diese vererbt.

Zusätzlich gibt es Implementierungen von diesen Schnittstellen. Man würde also *Person*, *Student*, *Employee* und *StudentAssistant* in jeweils zwei Klassen aufteilen, einmal die Schnittstelle und einmal die Implementierung. Die Schnittstellen würden voneinander erben, z.B. *StudentBase* von *PersonBase*, und entsprechende abstrakte Methoden wie **virtual string StudentBase::GetStudentID() = 0** bereitstellen. Die Implementierung würde ausschließlich von der jeweiliger Schnittstelle erben (*Student* von *StudentBase*). Diese Variante erscheint zwar aufwändiger als Implementierungsvererbung, vermeidet aber viele der dabei entstehenden Probleme.

Aufgabe 2 Template Funktionen

a) Implementieren Sie die Funktion

```
template<class T>
const T& maximum(const T& t1, const T& t2);
```

die das Maximum von zwei Variablen liefert. Durch die Verwendung von Templates soll die Funktion mit verschiedenen Datentypen funktionieren. Testen Sie Ihre Implementierung.

b) Legen Sie nun zwei Variablen vom Typ *int* und *short* an, und versuchen Sie, mittels *maximum()* das Maximum zu bestimmen. Der Compiler wird mit der Fehlermeldung **no matching function for call...** abbrechen, da er nicht weiß, ob *int* oder *short* der Template-Parameter sein soll. Geben Sie deshalb den Template-Parameter mittels *maximum<int>()* beim Aufruf von *maximum()* explizit an. Die übergebenen Parameter werden dabei vom Compiler automatisch in den gewünschten Typ umgewandelt.

c) Erstellen Sie eine Klasse *C*, die eine Zahl als Attribut beinhaltet. Implementieren Sie einen passenden Konstruktor sowie einen Getter für diese Zahl. Nun wollen wir unsere Funktion *maximum()* verwenden, um zu entscheiden, welches von zwei *C*-Objekten die größere Zahl beinhaltet. Überlegen Sie sich, was zu tun ist, und implementieren Sie es. Hinweis: Schauen Sie sich an, welche Operatoren Ihre Implementierung von *maximum()* verwendet. Diese müssen auch auf Objekte der Klasse *C* anwendbar sein (z.B. durch Operatorenüberladung).

Aufgabe 3 Klassentemplates

Erinnern Sie sich an die Klasse *Vector3* aus dem ersten Praktikumstag. Diese hat den Datentyp *double* für die einzelnen Komponenten verwendet. Schreiben Sie die Klasse so um, dass der Datentyp der Komponenten durch einen Template-Parameter angegeben werden kann. Fügen Sie dafür der Klasse *Vector3* einen Template-Parameter hinzu und ersetzen Sie jedes Aufkommen von *double* mit dem Template-Parameter. Vergessen Sie nicht, die Implementierung in den Header zu verschieben, da der Compiler die Definition einer Klasse kennen muss, um beim Einsetzen des Template-Parameters den richtigen Code zu generieren.

Verbessern Sie außerdem die Effizienz und Sauberkeit der *Vector3*-Klasse, in dem Sie die Parameterübergabe in den entsprechenden Methoden auf const Reference umstellen und alle Getter als const deklarieren.

Aufgabe 4 Verkettete Liste

a) Schreiben Sie die Klassen *List*, *ListItem* und *ListIterator* aus dem zweiten Praktikumstag so um, dass man den Inhaltstyp der Liste über ein Template-Parameter angeben kann.

Dazu müssen einige Änderungen gemacht werden. Zum einen sollte der Inhalt eines Elements beim Erstellen nicht als Wert sondern als const Referenz übergeben werden. Zum anderen sollten die Methoden zum Löschen von Elementen **void** zurückgeben, und nicht mehr das jeweilige gelöschte Element, weil in diesem Fall eine temporäre Kopie des Elements gemacht werden müsste, ohne dass es der Benutzer beeinflussen kann. Je nach Elementtyp können solche Kopien problematisch und unerwünscht sein.

Tipp: Arbeiten Sie die Klassen nacheinander ab, beginnend bei *ListItem*, und stellen Sie sicher, dass man eine Klasse fehlerfrei kompilieren kann, bevor Sie zur nächsten übergehen. Vergessen Sie nicht, dass Sie auch hier die Implementierung in die Header verschieben müssen. Da Header an sich nicht kompiliert werden, müssen Sie diese zum Kompilieren in eine Quellcodedatei einbinden.

b) Überladen Sie den Operator <<, sodass Listen direkt über ein *ostream* wie z.B. *cout* ausgegeben werden können.

c) Testen Sie Ihre Implementierung. Probieren Sie auch folgendes aus und beobachten Sie die Ausgabe.

```
List<List<int> > list; //  ">>" is an operator, so use "> >" for nested templates
list.appendElement(List<int>());
list.getFirst().appendElement(1);
list.getFirst().appendElement(2);
list.appendElement(List<int>());
list.getLast().appendElement(3);
list.appendElement(List<int>());
list.getLast().appendElement(4);
list.getLast().appendElement(5);

cout << list << endl;
```

Aufgabe 5 Callbacks

In dieser Aufgabe werden mehrere Methoden zur Realisierung von Callbacks in C++ vorgestellt und implementiert. Callbacks können zu verschiedenen Zwecken eingesetzt werden, wo man sonst das Observer Pattern benutzen würde. Z.B. kann man einem GUI-Button eine Callbackfunktion übergeben, die aufgerufen werden soll, sobald der Button gedrückt wird. Wir werden Callbacks dazu verwenden, um den Benutzer bei jedem Schritt eines laufenden Algorithmus über den aktuellen Fortschritt zu informieren.

a) Implementieren Sie folgenden Algorithmus, der das Problem der Türme von Hanoi löst (Quelle: Wikipedia).

```
funktion hanoi (Zahl i, Stab a, Stab b, Stab c)
if i > 0 then
    hanoi(i-1, a, c, b);
    verschiebe oberste Scheibe von "a" nach "c";
    hanoi(i-1, b, a, c);
end
```

Sie brauchen keine Türme zu modellieren und zu verschieben, es reicht, lediglich eine Ausgabe auf die Konsole zu machen. Bei einem Aufruf von **hanoi(3, 1, 2, 3)** soll folgende Ausgabe erfolgen:

```
1 -> 3
1 -> 2
3 -> 2
1 -> 3
2 -> 1
2 -> 3
1 -> 3
```

b) Nun wollten wie die fest einprogrammierte Ausgabe durch ein Callback ersetzen. Dadurch wird es möglich, die Funktion auszutauschen und z.B. eine graphische Ausgabe zu implementieren, ohne jedoch den Algorithmus selbst zu ändern.

Eine simple Art des Callbacks, die auch in C verfügbar ist, ist die Übergabe eines Funktionszeigers, der die Adresse der aufzurufenden Funktion beinhaltet. Ändern Sie Ihre Implementierung entsprechend um:

```
void hanoi(int i, int a, int b, int c, void(*callback)(int from, int to)) {
    ...
    callback(a, c);
    ...
}
```

Nun können Sie eine Funktion mit zwei Parametern an *hanoi()* übergeben.

```
void print(int from, int to) {
    cout << from << "␣->␣" << to << endl;
}
...
hanoi(3, 1, 2, 3, print);
```

c) Ein Nachteil der vorherigen Implementierung ist, dass nur reine Funktionen als Callback übergeben werden können. Eine Möglichkeit dies zu umgehen ist die Verwendung von Templates. Der Callback-Typ wird dabei durch einen Template-Parameter spezifiziert:

```
template<class T>
void hanoi(int i, int a, int b, int c, T callback) ...
```

Dadurch kann an *hanoi()* fast alles übergeben werden, was sich syntaktisch mittels

```
callback(a, c);
```

aufrufen lässt, also auch Objekte, bei denen der () Operator überladen ist (sog. Functors). Dabei müssen nicht einmal die Parametertypen (**int**) exakt übereinstimmen, solange eine implizite Umwandlung durch den Compiler möglich ist.

Testen Sie Ihre Implementierung sowohl mit der *print* Funktion als auch mit einem Functor. Schreiben Sie dafür eine einfache Klasse und überladen Sie deren () Operator mittels

```
void operator()(int from, int to);
```

d) Die Verwendung von Templates hat uns zwar eine sehr flexible und syntaktisch ansprechende Möglichkeit für Callbacks geliefert, beherbergt jedoch mehrere, teils gravierende, Schattenseiten.

Zum einen ist es dadurch immer noch nicht möglich, beliebige Methoden einer Klasse als Callback zu übergeben. Durch Methodencallbacks könnten Klassen mehrere unabhängige Callback-Methoden besitzen. Zum anderen ist *hanoi()* nun an den Callback-Typ **gekoppelt**. Wenn wir also *hanoi* selbst an eine Funktion/Methode übergeben wollen, muss der Callback-Typ bei der Übergabe mit angegeben werden und zerstört somit die Unabhängigkeit der Funktion von ihrem Callback. Dies kann sich insbesondere bei komplexeren Anwendungen von Callbacks sehr negativ widerspiegeln. Stellen Sie sich vor Sie hätten ein GUI-Framework mit verschiedenen Elementen, die Callbacks nutzen, z.B. Buttons. Dann wäre die Button-Klasse ebenfalls an den Callbacktyp gekoppelt. Jedes mal, wenn ein Button als Parameter an eine Funktion übergeben wird, müsste diese Funktion den Callbacktyp ebenfalls als Template-Parameter entgegennehmen:

```
template<class T>
void doSomethingWithButton(Button<T>& btn);
```

Dieser Stil würde sich durch das gesamte Framework ziehen, und sowohl den Entwicklungsaufwand als auch die Verständlichkeit beeinträchtigen. Ein weiterer Nachteil wäre, dass der Callback-Typ bereits zur Compile-Zeit festgelegt werden müsste und es unmöglich wäre, diesen während der Laufzeit zu ändern.

Deshalb werden wir eine Klasse schreiben, die beliebige Callbacks kapseln kann, und nach außen hin allein von den Übergabeparametern des Callbacks abhängig ist. Ziel ist es, folgendes zu ermöglichen:

```
void hanoi (..., Callback callback) {
    ...
    callback(a, c);
    ...
}
...
hanoi (..., Callback(print)); // function callback
hanoi (..., Callback(c)); // functor callback
hanoi (..., Callback(&C::print, &c)); // method callback
```

Die Idee dazu ist Folgendes: Wir definieren eine abstrakte Klasse *CallbackBase*, die eine abstrakte Methode *void call() = 0* enthält. Für jeden Callback-Typ (Funktionszeiger, Funktor und Methodenzeiger) wird eine Unterklasse erstellt, die *call()* entsprechend reimplementiert.

Fangen Sie mit der Klasse *CallbackBase* an. Damit man beim Aufrufen des Callbacks einen Parameter übergeben kann, fügen Sie *call()* einen Parameter vom Typ *ParamT* hinzu, wobei *ParamT* ein Template-Pameter von *CallbackBase* sein soll. Der Klassenrumpf lautet also

```
template<class ParamT>
class CallbackBase {
public:
    ...
    virtual void call(ParamT t) = 0;
};
```

Falls ein Callback eigentlich mehrere Parameter erfordert, müssen diese entsprechend in ein Containerobjekt gepackt werden. Generische Callback-Wrapper mit variabler Parameteranzahl sind zwar möglich, würden aber den Rahmen dieses Praktikums sprengen. Tipp: Sie können diese und alle nachfolgenden Klassen in einem einzigen Header implementieren, weil die Klassen sehr kurz sind und außerdem semantisch stark zusammenhängen.

Implementieren Sie nun die erste Unterklasse *template<class ParamT> FunctionCallback*, die von *CallbackBase<ParamT>* erbt. *FunctionCallback* soll einen entsprechenden Funktionszeiger als Attribut besitzen, der bei der Konstruktion initialisiert wird. Ebenso soll *call(ParamT t)* implementiert werden, wo der gespeicherte Funktionszeiger mit dem gegebenen Argument aufgerufen wird.

Testen Sie Ihre Implementierung. Lassen Sie *hanoi()* einen Zeiger auf *CallbackBase* nehmen, übergeben Sie aber die Adresse eines *FunctionCallback* Objektes.

e) Implementieren Sie nun die Unterklasse *template<class ParamT, class ClassT> FunctorCallback*. Zusätzlich zum Parameter-Typ muss hier auch der Typ der Functor-Klasse angegeben werden. Speichern Sie das zu verwendende Functor-Objekt als Referenz ab, um Kopien zu vermeiden. Achten Sie auch im Konstruktor darauf, dass keine Kopien des Funktors gemacht werden. Testen Sie Ihre Implementierung. Vergewissern Sie sich auch, dass der übergebene Functor tatsächlich nicht kopiert wird.

f) Implementieren Sie nun die letzte Unterklasse *template<class ParamT, class ClassT> MethodCallback*. Beachten Sie, dass nun zwei Attribute nötig sind - ein Methodenzeiger und ein Zeiger auf das zu verwendende Objekt. Testen Sie Ihre Implementierung.

g) Wir haben jetzt den Typ des Callbacks vollständig von seiner Verwendung entkoppelt. Jedoch muss ein Callback-Objekt per Zeiger/Referenz übergeben werden, sodass das Ihnen schon bekannte Problem der Zuständigkeit für die Zerstörung eines Objekts entsteht. Außerdem muss man beim Erstellen eines Callbacks explizit den Typ der Unterklasse angeben. Es wäre also sinnvoll, einen entsprechenden Wrapper zu schreiben, der sich um die Speicherverwaltung von Callbacks kümmert und bei der Konstruktion die passende Unterklasse selbst aussucht.

   Schreiben Sie eine Klasse *template<class ParamT> Callback*, die einen Smart Pointer auf ein *CallbackBase* Objekt als Attribut hat. Der Smart Pointer soll die Speicherverwaltung übernehmen. Überladen Sie den () Operator, der den Aufruf einfach an das *CallbackBase*-Objekt hinter dem Smart Pointer weiterleitet.

   Implementieren Sie nun für jede Callback-Art je einen Konstruktor, der eine Instanz der entsprechenden Unterklasse erzeugt und in dem Smart Pointer speichert. Der erste Konstruktor soll also einen Funktionszeiger entgegennehmen und ein *FunctionCallback* instanziieren. Der zweite Konstruktor soll eine Referenz auf ein Functor-Objekt erwarten und *FunctorCallback* instanziieren, und der dritte entsprechend ein *MethodCallback*. Beachten Sie, das die beiden letztgenannten Konstruktoren selbst Template-Methoden sind, da die *Callback*-Klasse nur an den Parameter-Typ gekoppelt ist.

Testen Sie Ihre Implementierung in Zusammenhang mit der *hanoi()* Funktion. Sie können das *Callback*-Objekt auch per Wert übergeben, da intern nur Zeiger kopiert werden. Um die zwei benötigten Parameter in ein einziges Objekt zu packen, können Sie entweder einen eigenen Container schreiben oder die Klasse *std::pair<T1, T2>* nutzen und mittels **t.first**, **t.second** auf die beiden Komponenten zugreifen.

Nachwort: Für echte C++ Programme bietet die *boost*-Bibliothek fertige Funktionen und Klassen, um Callbacks zu realisieren, z.B. *boost::function<...>* und *boost::bind()*. Diese können mit beliebiger Anzahl von Parametern umgehen und beinhalten viele weitere umfangreiche Features.

Aufgabe 6 STL Container

In dieser Aufgabe werden wir den Umgang mit den Containern *std::vector<T>* und *std::list<T>* aus der Standard Template Library üben. Da nicht alle für diese Aufgabe benötigten Klassen und Funktionen hier erklärt werden, müssen Sie selbst in die Dokumentation der STL schauen, z.B. auf http://www.cplusplus.com/. Schauen Sie sich auch die Vorlesungsfolien genau an, da diese viele nützliche Codebeispiele enthalten.

Die Klasse *std::list<T>* stellt eine verkettete Liste dar, bei der man an beliebiger Stelle Elemente effizient löschen und hinzufügen kann. *std::vector<T>* stellt ähnliche Funktionen bereit, allerdings liegen hier die Elemente in einem einzigen, zusammenhängenden Speicherbereich, der neu alloziert und kopiert werden muss, falls der Vektor mehr Elemente aufnehmen soll, als seine aktuelle Größe es ihm eigentlich erlaubt. Auch müssen viele Elemente verschoben werden, wenn der Vektor in der Mitte oder am Anfang modifiziert wird. Dafür kann man über einen Index auf beliebige Elemente mit konstantem Aufwand zugreifen.

a) Schreiben Sie als erstes eine Funktion *template<class T> void print(const T& t)*, die beliebige STL-Container auf die Konsole ausgeben kann, die Iteratoren unterstützen. Nutzen Sie dazu die Funktion *copy()* sowie die Klasse *ostream_iterator<T>*, um den entsprechenden *OutputIterator* zu erzeugen.

b) Legen Sie ein *int*-Array an und initialisieren Sie es mit den Zahlen 1 bis 5. Legen Sie nun ein *vector<int>* an und initialisieren Sie es mit den Zahlen aus dem Array.

c) Legen Sie eine Liste *list<int>* an und initialisieren Sie diese mit dem zweiten bis vierten Element des Vektors. Tipp: Sie können auf Iteratoren eines Vektors (genauso wie auf Zeiger) Zahlen addieren, um diese zu verschieben.

d) Fügen Sie mittels *list<T>::insert()* das letzte Element des Vektors an den Anfang der Liste hinzu.

e) Löschen Sie alle Elemente des Vektors mit einem einzigen Methodenaufruf.

f) Mittels *remove_copy_if()* kann man Elemente aus einem Container in einen anderen kopieren und dabei bestimmte Elemente löschen lassen. Nutzen Sie diese Funktion, um alle Elemente, die kleiner sind als 4, aus der Liste in den Vektor zu kopieren. Beachten Sie, dass *remove_copy_if()* keine neue Elemente an den Container anhängt, sondern lediglich Elemente von der einen Stelle zur anderen elementweise durch Erhöhen des *OutputIterator* kopiert.

   Deshalb dürfen Sie *vec.end()* **nicht** als *OutputIterator* nehmen, da dieser "hinter" das letzte Element zeigt und weder dereferenziert noch inkrementiert werden darf. Nutzen Sie stattdessen die Methode *back_inserter()*, um einen Iterator zu erzeugen, der neue Elemente an den Vektor anhängen kann.

Aufgabe 7 Exceptions

Auch in C++ besteht die Möglichkeit, Fehler mittels Exceptions zu signalisieren und zu verarbeiten.

```
try {
    ...
   throw <Type>;
} catch(<Type1> <param name>) {
    ...
} catch(<Type2> <param name>) {
    ...
}
...
```

Es gibt jedoch einige Unterschiede zur Fehlerbehandlung in Java. Das aus Java bekannte *finally*-Konstrukt existiert in C++ nicht. Außerdem kann jede Art von Wert geworfen werden, also sowohl ein beliebiges Objekt als auch ein primitiver Wert wie z.B. ein *int*. In der Praxis wird es jedoch empfohlen, den geworfenen Wert von *std::exception* abzuleiten oder eine der existierenden Klassen aus der Standardbibliothek zu nutzten.

Im Gegensatz zu Java kann man Objekte nicht nur per Referenz sondern auch *per-Value* werfen und fangen. In diesem Fall wird das geworfene Objekt nach der Behandlung im **catch**-Block automatisch zerstört. Wenn es *per-Value* gefangen wird, wird das geworfene Objekt kopiert, ähnlich wie bei einem Funktionsaufruf. Beispiel:

1. Catch by value

```
try {
    throw C(); // create new object of class C and throw it
} catch(C c) { // catch c by value => a copy of c is created when catching
    ...
}
```

2. Catch by reference

```
try {
    throw C(); // create new object of class C and throw it
} catch(const C& c) { // catch c by reference, no copy is created
    ...
}
```

In der Praxis hat es sich durchgesetzt, *per Value* zu werfen und *per const Reference* zu fangen.

a) Erstellen Sie eine Klasse *C* und implementieren Sie einen Konstruktor, einen Copy-Konstruktor und einen Destruktor. Versehen Sie diese mit Ausgaben auf der Konsole, so dass der Lebenszyklus während der Ausführung ersichtlich wird.

b) Experimentieren Sie mit Exceptions. Probieren Sie insbesondere die beiden o.g. Fälle aus und beobachten Sie die Ausgabe. Wann wird ein Objekt erstellt/kopiert/gelöscht? Testen Sie auch, was passiert, wenn Sie mehrere **catch**-Blöcke erstellen und sich diese nur in der Übergabe unterscheiden (Wert/Referenz). Welcher von ihnen wird aufgerufen? Spielt die Reihenfolge eine Rolle?

c) Fügen Sie der Klasse *List* aus Aufgabe 4 Bereichsprüfungen hinzu. Schreiben Sie die Methoden *insertElementAt()*, *getNthElement()* und *deleteAt()* so um, dass eine Exception geworfen wird, falls der angegebene Index die Größe der Liste überschreitet. Nehmen Sie dafür die Klasse *std::out_of_range* aus dem *stdexcept* Header.

d) Testen Sie Ihre Implementierung. Provozieren Sie eine Exception, indem Sie falsche Indices angeben, und fangen Sie die Exception als const Refernz ab. Sie können die Methode *what()* benutzen, um an den Nachrichtentext der Exception zu gelangen.

Aufgabe 8 C Einführung

In den nächsten Tagen werden Sie Programme für eine Embedded Plattform in C entwickeln. Da C++ aus C entstand, sind viele Features von C++ nicht in C enthalten. Im Folgenden sollen die Hauptunterschiede verdeutlicht werden.

- Kein OO-Konzept, keine Klassen, nur Strukturen.
- Keine Templates
- Keine Referenzen, nur Zeiger und Werte
- Kein *new* und *delete*, sondern *malloc()* und *free()*
- Je nach Sprachstandard müssen Variablen am Anfang der Funktion deklariert werden
- Funktionen ohne echte Parameter müssen *void* als Parametertyp haben, leere Klammern () bedeuten, dass beliebige Argumente erlaubt sind.

- Keine Streams, stattdessen *(f)printf* zur Ausgabe auf Konsole und in Dateien
- Kein *bool* Datentyp, stattdessen ist alles ungleich 0 wahr, 0 ist falsch
- Keine Default-Argumente
- Keine *std::string* Klasse, nur char-Arrays
- Keine Namespaces

Da einige dieser Punkte sehr entscheidend sind, werden wir auf diese im Detail eingehen.

Kein OO-Konzept

In C gibt es keine Klassen, weshalb die Programmierung in C eher Pascal anstatt C++ ähnelt. Stattdessen gibt es Strukturen (**structs**), mit denen man mehrere Variablen zu einem Datentypen zusammenfassen kann. Es entspricht den Records in Pascal und lässt sich mit Klassen ohne Methoden und ohne Vererbung vergleichen.

Die Syntax dafür lautet

```
struct <Name> { // Name ohne <>
    <Type1> <Name11>, <Name12>, ...;
    <Type2> <Name21>, <Name22>, ...;
};
```

Zum Beispiel

```
struct Point {
    int x;
    int y;
};
```

Die Sichtbarkeit aller Attribute ist automatisch *public*.

Um den definierten **struct** als Datentyp zu verwenden, muss man zusätzlich zum Namen das Schlüsselwort **struct** angeben:

```
void foo(struct Point* p) {
    ...
}
...
int main(void) {
    struct Point point;
    foo(&point);
}
```

Um den zusätzlichen Schreibaufwand zu vermeiden, wird in der Praxis oft ein **typedef** auf den **struct** definiert:

```
typedef struct Point Point_t;
...
Point_t point;
```

Man kann die Deklaration eines **struct** auch direkt in den **typedef** einbauen:

```
typedef struct {
    int x;
    int y;
} Point;
```

Kein *new* und *delete*

Anstelle von *new* und *delete* werden die Funktionen *malloc()* und *free()* verwendet, um Speicher auf dem Heap zu reservieren. Diese sind im Header *stdlib.h* deklariert.

```
Point* points = malloc(10 * sizeof(Point)); // reserve memory for 10 points
...
free(points);
```

Ausgabe auf Konsole per *printf()*

Im Daten auf der Konsole auszugeben, kann die Funktion *printf(fmt, ...)* verwendet werden. *printf()* nimmt einen Format-String sowie eine beliebige Anzahl weiterer Argumente entgegen. Der Format-String gibt an, als was die nachfolgenden Argumente ausgegeben werden sollen. Mittels \n kann man einen Zeilenvorschub erzeugen. Um *printf()* zu nutzen, muss der Header *stdio.h* eingebunden werden.

```
printf("Hallo␣Welt\n"); // Hallo Welt + neue Zeile ausgeben

int i;
printf("i␣=␣%d\n", i); // Integer ausgeben

int i;
char c;
printf("c␣=␣%c,␣i␣=␣%d\n", c, i); // Zeichen und Integer ausgeben
```

Weitere Möglichkeiten von *printf()* können Sie unter http://www.cplusplus.com/reference/cstdio/printf/ nachlesen.

a) Schreiben Sie ein C-Programm, welches alle geraden Zahlen von 0 bis 200 formatiert ausgibt. Die Formatierung soll entsprechend dem Beispiel erfolgen:

```
 2   4   6   8  10
12  14  16  ...
```

b) Versuchen Sie, beliebige (einfache) Programme der vergangenen Tage in reinem C auszudrücken (Schwierigkeitsgrad sehr unterschiedlich!).