# Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

## Übungen für den 4. Tag

### Aufgabe 1 Mehrfachvererbung

Nimm dir als Basis für diese Aufgabe die 1. Aufgabe aus der gestrigen Übung.

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

b) **Klasse *StudentAssistant*** Schreibe nun eine Klasse *StudentAssistant*, die eine wissenschaftliche Hilfskraft modelliert. 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. Überschreibe *getInfo()*, um alle Daten auszugeben. Ändere dazu die Sichtbarkeit der Attribute sowohl von *Student* als auch von *Employee* von *private* auf *protected*.

   Du wirst feststellen, dass sich die Klasse nicht kompilieren lässt, falls du das Attribut *name* direkt verwendest, da in einer *StudentAssistant*-Instanz zwei Instanzen von *Person* vorhanden sind - je eine von jeder Elternklasse. Deshalb müsse mittels dem Scope-Operator :: angeben, welche Basis du genau meinst.

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

   Teste deine Implementierung, indem du das Ergebnis von *getInfo()* direkt in der *main* ausgibst.

c) **Virtuelle Vererbung** Versuche 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 sorgen mittels virtueller Vererbung dafür, dass *Person* nur ein Mal in *StudentAssistant* vorhanden ist.

   Lasse dazu *Student* und *Employee* virtuell von *Person* erben. Noch lässt sich das Programm nicht kompilieren, denn 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 – einmal von *Student* und einmal 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 wir 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! Füge deshalb ein **Person(name)** in die Initialisierungsliste von *StudentAssistant* hinzu.

   Teste deine Implementierung. Versuche auch Folgendes: Ändere die Namen in den Konstruktoraufrufen von *Student* und *Employee* in der Initialisierungsliste von *StudentAssistant* und beobachte die Ausgabe. Mache dir 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, eine Schnittstelle und eine Implementierung. Die Schnittstellen würden voneinander erben, z.B. *StudentBase* von *PersonBase*, und entsprechende pur virtuelle/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. Schnittstellenvererbung kann in Java eingesetzt werden, um Mehrfachvererbung zu realisieren.

## Aufgabe 2 Template Funktionen

a) **Templatefunktionen implementieren** Implementiere die folgende Funktion, die das Maximum von zwei Variablen liefert:

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

Durch die Verwendung von Templates soll die Funktion mit verschiedenen Datentypen funktionieren. Teste deine Implementierung.

In der Vorlesung haben wir gesehen, dass jede Verwendung von *t1* und *t2* in *maximum* eine Schnittstelle induziert, die der Typ *T* bereitstellen muss. Das bedeutet, dass *t1* und *t2* alle Konstruktoren, Methoden und Operatoren zur Verfügung stellen, die in *maximum* genutzt werden.

Wie sieht diese Schnittstelle in diesem Fall aus?

b) **Explizite Angabe der Typparameter** Lege nun zwei Variablen vom Typ *int* und *short* an, und versuche, 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. Gib 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) **Induzierte Schnittstelle implementieren** Erstelle eine Klasse *C*, die eine Zahl als Attribut beinhaltet. Implementiere 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. Überlege dir, was zu tun ist, und implementiere es.

*Hinweis*: Die Klasse *C* muss mindestens die durch *maximum* induzierte Schnittstelle implementieren.

## Aufgabe 3 Generische Vektor-Implementierung

Erinnere dich an die Klasse *Vector3* aus dem ersten Praktikumstag. Diese hat den Datentyp *double* für die einzelnen Komponenten verwendet. Schreibe die Klasse so um, dass der Datentyp der Komponenten durch einen Template-Parameter angegeben werden kann. Füge dafür der Klasse *Vector3* einen Template-Parameter hinzu und ersetze jedes Aufkommen von *double* mit dem Template-Parameter. Vergiss 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.

Verbessere außerdem die Effizienz und Sauberkeit der *Vector3*-Klasse, in dem du die Parameterübergabe in den entsprechenden Methoden auf *const* Referenzen umstellst und alle Getter als *const* deklarierst.

## Übung zum C/C++-Praktikum - Tag 4

### Aufgabe 4 Array selber implementieren (optional)

Nachdem du jetzt gesehen hast, dass es störend ist, wenn man die Größe eines Arrays immer getrennt zu den gespeicherten Daten verwalten muss, ist ein sinnvoller Schritt, eine eigene Array-Klasse zu implementieren, die Daten und Größe des Arrays zusammen speichert.

Eine möglicher Anwendungsfall sieht so aus:

```
#include "Array.h"
#include <iostream>
#include <string>

using namespace std;

template<class T>
void printFirst (const Array<T> &array) {
    cout << array[0] << endl;
}

int main(int argc, char **argv) {
    Array<std::string> stringArray(10);
    stringArray[0] = "Hello World";

    printFirst (stringArray);

    return 0;
}
```

*Hinweis*: Diese Idee ist natürlich nicht neu. Boost bietet eine eigene Array-Implementierung in der Klasse *boost::array*[1], die seit C++11 auch in der Standardbibliothek unter dem Namen *std::array* zu finden ist.

### Aufgabe 5 Generische Verkettete Liste

a) Schreibe 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*: Arbeite die Klassen nacheinander ab, beginnend bei *ListItem*. Stelle sicher, dass man eine Klasse fehlerfrei kompilieren kann, bevor du zur nächsten übergehst. Vergiss nicht, dass du auch hier die Implementierung in die Header verschieben musst. Da Header an sich nicht kompiliert werden, müssen wir diese zum Kompilieren in eine (leere) Quellcodedatei einbinden.

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

c) Teste deine Implementierung. Probiere auch folgendes aus und beobachte die Ausgabe.

```
List<List<int> > list; //  ">>" is an operator, so use "> >" for nested templates
list.appendElement(List<int>());
```

[1] http://www.boost.org/doc/libs/1_55_0/doc/html/array.html

```
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 6 Callbacks

In dieser Aufgabe werden mehrere Methoden zur Realisierung von Callbacks in C++ vorgestellt und implementiert. Callbacks können als Alternative zum Observer Pattern[2] eingesetzt werden. Beispielsweise kann man einem GUI-Button eine Callback-Funktion 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) **Basisalgorithmus** Implementiere folgenden Algorithmus, der das Problem der Türme von Hanoi löst.[3]

**funktion** hanoi (**Number** i, **Pile** a, **Pile** b, **Pile** c)
**if** *i > 0* **then**
  hanoi(i-1, a, c, b); `// Move i-1 slices from pile "a" to "b"`
  Move slice from "a" to "c";
  hanoi(i-1, b, a, c); `// Move i-1 slices from pile "b" to "c"`
**end**

Du brauchst 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) **Callbacks mit Funktionszeigern** 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. Ändere deine Implementierung entsprechend um:

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

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

[2] http://de.wikipedia.org/wiki/Observer_Pattern
[3] http://de.wikipedia.org/wiki/Turm_von_Hanoi

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

c) **Callbacks mit Funktoren** 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*[4]). Dabei müssen nicht einmal die Parametertypen (*int*) exakt übereinstimmen, solange eine implizite Umwandlung durch den Compiler möglich ist.

Teste deine Implementierung mit einem Functor. Schreibe dafür eine einfache Klasse und überlade deren *operator()*:

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

d) **Callbacks mit *Callback*-Klasse** 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. Stell dir vor, du hättest ein GUI-Framework mit verschiedenen Elementen, die Callbacks nutzen, z.B. Buttons. Dann wäre die Button-Klasse ebenfalls an den Callback-Typ gekoppelt. Immer 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);
    ...
}
```

[4] https://de.wikipedia.org/wiki/Funktionsobjekt

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

Die Idee dahinter 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.

Fange mit der Klasse *CallbackBase* an. Damit man beim Aufrufen des Callbacks einen Parameter übergeben kann, füge *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*: Du kannst diese und alle nachfolgenden Klassen in einem einzigen Header implementieren, weil die Klassen sehr kurz sind und außerdem semantisch stark zusammenhängen.

e) **Klasse *FunctionCallback***: Implementiere 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.

Teste deine Implementierung. Lasse *hanoi()* einen Zeiger auf *CallbackBase* nehmen, übergebe aber die Adresse eines *FunctionCallback* Objektes. Du kannst folgende Vorlage verwenden:

```
#include<utility>
typedef std::pair<int, int> intpair;

void hanoi (..., CallbackBase<intpair> *callback) {
    // ...
    callback->call(intpair(a, c));
    // ...
}

int main() {
    // ...
    CallbackBase<intpair> *function =
        new FunctionCallback<intpair>(printMovePair);
    hanoi(3,1,2,3, function);
    // ...
}
```

f) **Klasse *FunctorCallback***: Implementiere nun die Unterklasse *template<class ParamT, class ClassT> FunctorCallback*. Zusätzlich zum Parameter-Typ muss hier auch der Typ der Functor-Klasse angegeben werden. Speichere das zu verwendende Funktor-Objekt als Referenz ab, um Kopien zu vermeiden. Achte auch im Konstruktor darauf, dass keine Kopien des Funktors gemacht werden. Teste deine Implementierung!

g) **Klasse *MethodCallback***: Implementiere nun die letzte Unterklasse *template<class ParamT, class ClassT> MethodCallback*. Beachte, dass nun zwei Attribute nötig sind - ein Methodenzeiger und ein Zeiger auf das zu verwendende Objekt. Teste deine Implementierung.

*Tipp*: Verwende beispielsweise folgende Signatur für den Konstruktor von *MethodCallback*:

```
MethodCallCallback(void(ClassT::*method)(ParamT), ClassT* object)
```

*Tipp 2*: Gegeben einen Zeiger *object* auf ein Objekt, einen Zeiger *method* auf eines seiner Methoden und einen Parameter *p* für die Methode, sieht ein Aufruf von *method* wie folgt aus:

```
(object->*method)(p);
```

h) **Callback**: Wir haben jetzt den Typ des Callbacks vollständig von seiner Verwendung entkoppelt. Jedoch muss ein Callback-Objekt per Zeiger/Referenz übergeben werden, sodass das dir 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.

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

Implementiere 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* instantiieren. Der zweite Konstruktor soll eine Referenz auf ein Funktor-Objekt erwarten und *FunctorCallback* instanziieren, und der dritte entsprechend ein *MethodCallback*. Beachte, dass die beiden letztgenannten Konstruktoren selbst Template-Methoden sind, da die *Callback*-Klasse nur an den Parameter-Typ gekoppelt ist.

Teste deine Implementierung in Zusammenhang mit der *hanoi*-Funktion. Du kannst das *Callback*-Objekt auch per Wert übergeben, da intern nur Zeiger kopiert werden.

**Nachwort**: Für produktive C++-Programme bietet Boost fertige Funktionen und Klassen, um Callbacks zu realisieren, z.B. *boost::function<...>* und *boost::bind()*[5]. Diese können mit beliebiger Anzahl von Parametern umgehen und beinhalten viele weitere Features.

## Aufgabe 7 STL Container

In dieser Aufgabe werden wir den Umgang mit den Containern *std::vector* und *std::list* aus der Standard Template Library üben. Es ist sinnvoll, wenn du während der Übung eine C++-Referenz zum Nachschlagen bereithältst, z.B. `http://www.cplusplus.com/`. Schaue dir auch die Vorlesungsfolien genau an, da diese nützliche Codebeispiele enthalten.

Die Klasse *std::list* stellt eine verkettete Liste dar, bei der man an beliebiger Stelle Elemente effizient löschen und hinzufügen kann. *std::vector* stellt ähnliche Funktionen bereit, allerdings liegen hier die Elemente in einem einzigen, zusammenhängenden Speicherbereich, der neu alloziert und kopiert werden muss, seine aktuelle Kapazität überschritten wird. Auch müssen viele Elemente verschoben werden, wenn der Vektor in der Mitte oder am Anfang modifiziert wird. Der große Vorteil von *std::vector* ist der *wahlfreie Zugrif*, d.h. man kann auf beliebige Elemente mit konstantem Aufwand zugreifen.

a) Schreibe zunächst eine Funktion *template<class T> void print(const T& t)*, die beliebige STL-Container auf die Konsole ausgeben kann, die Integer speichern und Iteratoren unterstützen. Nutze dazu die Funktion *copy()* sowie die Klasse *ostream_iterator<int>*, um den entsprechenden *OutputIterator* zu erzeugen.

[5] `http://www.boost.org/doc/libs/1_55_0/doc/html/function.html`

b) Lege ein *int*-Array an und initialisiere es mit den Zahlen 1 bis 5. Lege nun einen *vector<int>* an und initialisiere ihn mit den Zahlen aus dem Array.

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

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

e) Lösche 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. Nutze diese Funktion, um alle Elemente, die kleiner sind als 4, aus der Liste in den Vektor zu kopieren. Beachte, 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 kannst du *vec.end()* **nicht** als *OutputIterator* nehmen, da dieser "hinter" das letzte Element zeigt und weder dereferenziert noch inkrementiert werden darf. Nutze stattdessen die Methode *back_inserter()*, um einen Iterator zu erzeugen, der neue Elemente an den Vektor anhängen kann.

## Aufgabe 8 Exceptions

Ähnlich wie in Java können Fehler in C++ mittels Exceptions signalisiert werden.

```
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 – sowohl Objekte als auch primitive Werte wie z.B. *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 *by-Reference* sondern auch *by-Value* werfen und fangen. In diesem Fall wird das geworfene Objekt nach der Behandlung im **catch**-Block automatisch zerstört. Wenn es *by-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, *by-Value* zu werfen und *by-const-Reference* zu fangen.

a) Erstelle eine Klasse *C* und implementiere einen Konstruktor, einen Copy-Konstruktor und einen Destruktor. Versehe diese mit Ausgaben auf der Konsole, so dass der Lebenszyklus während der Ausführung ersichtlich wird.

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

c) Füge der Klasse *List* aus Aufgabe 5 (oder dem Array aus Aufgabe 4) Bereichsprüfungen hinzu. Schreibe die Methoden *insertElementAt()*, *getNthElement()* und *deleteAt()* so um, dass eine Exception geworfen wird, falls der angegebene Index die Größe der Liste überschreitet. Nimm dafür die Klasse *std::out_of_range* aus dem *stdexcept* Header.

d) Teste deine Implementierung. Provoziere eine Exception, indem du falsche Indices angibst, und fange die Exception als *const* Referenz ab. Du kannst die Methode *what()* benutzen, um an den Nachrichtentext der Exception zu gelangen.

## Aufgabe 9 C Einführung

In den nächsten Tagen werde 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 (*struct*).
- Keine Templates
- Keine Referenzen, nur Zeiger und Werte
- Kein *new* und *delete*, sondern *malloc()* und *free()* (`#include <stdlib.h>`)
- Je nach Sprachstandard müssen Variablen am Anfang der Funktion deklariert werden (Standard-Versionen < C99)
- Parameterlose Funktionen 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 (`#include <stdio.h>`)
- Kein *bool* Datentyp, stattdessen wird 0 als *false* und alle anderen Zahlen als *true* gewertet
- Keine Default-Argumente
- Keine *std::string* Klasse, nur *char*-Arrays, die mit dem Nullbyte (`'\0'`) abgeschlossen werden.
- 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 statt C++ ähnelt. Stattdessen gibt es Strukturen (**struct**), die mehrere Variablen zu einem Datentyp zusammenfassen, was vergleichbar mit Records in Pascal oder – allgemein – mit Klassen ohne Methoden und ohne Vererbung ist.

Die Syntax dafür lautet

```
struct MyStruct {
```

```
    <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.

```
#include <stdlib.h>
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* verwendet werden. *printf()* nimmt einen Format-String sowie eine beliebige Anzahl weiterer Argumente entgegen. Der Format-String legt fest, wie die nachfolgenden Argumente ausgegeben werden. Mittels \n kann man einen Zeilenvorschub erzeugen. Um *printf()* zu nutzen, muss der Header *stdio.h* eingebunden werden.

```
#include <stdio.h>
printf("Hallo Welt\n"); // Hallo Welt + neue Zeile ausgeben
```

```
int i;
printf("i = %d\n", i); // Integer ausgeben
printf("i = %3d\n", i); // Integer ausgeben, auf drei Stellen auffüllen

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

Weitere Möglichkeiten von *printf()* findest du unter `http://www.cplusplus.com/reference/cstdio/printf/`.

Zum Abschluss noch ein paar Aufgaben, um mit C etwas warm zu werden:

a) Schreibe 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) Versuche, beliebige (einfache) Programme der vergangenen Tage in reinem C auszudrücken (Schwierigkeitsgrad sehr unterschiedlich!).