Übung zum C/C++-Praktikum Fachgebiet Echtzeitsysteme

TECHNISCHE UNIVERSITÄT DARMSTADT

Übungen für den 4. Tag

Musterlösungs-/Microcontroller-Projekte in CodeLite importieren

Die angebotenen Musterlösungs-/Microcontroller-Projekte basieren auf Makefiles. Daher ist es wichtig, dass du sie entsprechend importierst: **Workspace -> Add an existing project** und dann zur entsprechenden `.project`-Datei navigieren.

Aufgabe 1 Template Funktionen

Aufgabe 1.1 Templatefunktionen implementieren

Implementiere die folgende Funktion, die das Maximum von zwei Variablen liefert:

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

Durch die Verwendung von Templates soll die Funktion mit verschiedenen Datentypen funktionieren. Teste deine Implementation.
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 `T` alle Konstruktoren, Methoden und Operatoren zur Verfügung stellen muss, die in `maximum` genutzt werden.
Wie sieht diese Schnittstelle in diesem Fall aus?

Hinweise

- In den meisten Fällen kann anstelle von `typename` auch `class` in der Template-Deklaration verwendet werden.
- In der Regel muss die Definition von Template-Funktionen und -Methoden im Header erfolgen. Allgemeiner: zur Compilezeit in der gleichen cpp-Datei wie ihre Verwendung). Das hängt damit zusammen, dass Templates sprichwörtlich nur Vorlagen sind, deren Typparameter zur Compilezeit mit den konkret verwendeten Typen ersetzt werden. Würde man Templates in separaten cpp-Dateien implementieren, dann könnte die Verbindung zwischen der Verwendungsstelle und der Definition erst zur Linkzeit hergestellt werden – also zu spät.

Aufgabe 1.2 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.

Aufgabe 1.3 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.

Hinweise

- Die Klasse `C` muss mindestens die durch `maximum` induzierte Schnittstelle implementieren.

Aufgabe 2 Generische Vektor-Implementation

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 Implementation 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.
Du weißt bereits, dass alle `template`-Funktionen und -Methoden im Header enthalten sein müssen. Um den Code trotzdem zu strukturieren, hat es sich eingebürgert, dass man die Klassendefinition in der `hpp`-Datei hält, ohne die Methoden zu implementieren. Im Anschluss wird eine `tpp`-Datei inkludiert, die die Implementierung der Methoden und Funktionen enthält.
Der Aufbau wäre also in etwas wie folgt:

```
template<typename T>
class Vector3 {
  // method declarations only
};

// function declarations only

#include "Vector3.tpp" // contains method and function definitions
```

Übungen für den 4. Tag

Aufgabe 3 Generische Verkettete Liste

Aufgabe 3.1

Schreibe die Klassen `List`, `ListItem` und `ListIterator` aus dem zweiten Praktikumstag so um, dass man den Typen der in der Liste gespeicherten Elemente 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.

Hinweise

- Arbeite die Klassen nacheinander ab, beginnend bei `ListItem`.
- Stelle sicher, dass man eine Klasse fehlerfrei kompilieren kann, bevor du zur nächsten übergehst.
- Denke daran, dass du auch hier die Implementation in die Header verschieben musst.

Aufgabe 3.2

Überlade den `operator<<`, sodass Listen direkt über ein `std::ostream` wie z.B. `std::cout` ausgegeben werden können.

Aufgabe 3.3

Teste deine Implementation. Probiere auch folgendes aus und beobachte 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);

std::cout << list << std::endl;
```

Übungen für den 4. Tag

Aufgabe 4 Standard-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, wenn 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 Zugriff*, d.h. man kann auf beliebige Elemente mit konstantem Aufwand zugreifen.

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

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

c) Lege eine Liste `std::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 `std::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 neuen 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.

Übungen für den 4. Tag

Aufgabe 5 Funktionales Programmieren (experimentell)

Diese Aufgabe wurde neu erstellt und kann noch Fehler und Inkonsistenzen enthalten. Falls euch etwas derartiges auffällt sprecht uns bitte darauf an oder stellt es auf GitHub in den Issuetracker unter `https://github.com/Echtzeitsysteme/tud-cpp-exercises/issues`

In dieser Aufgabe werden Funktionen aus der funktionalen Programmierung vorgestellt. Diese sind `map`, `filter` und `reduce`.

Der Ablauf ist wie folgt:

- In Teilaufgabe Aufgabe 5.1 werden erst einmal die Funktionsweisen der zu implementierenden Funktionen `map`, `filter` und `reduce` vorgestellt.
- In Teilaufgabe Aufgabe 5.2 wirst du diese Funktionen implementieren und Hilfsfunktionen, welche dann verwendet werden können, implementieren.
- In Teilaufgabe Aufgabe 5.3 wirst du deine Hilfsfunktionen als *Funktoren* implementieren und `map`, `filter` und `reduce` entsprechend anpassen.
- Daraufhin werden Templates einbauen, damit Funktionen und Funktoren verwendet werden können. Dies wird in Teilaufgabe Aufgabe 5.4 passieren.
- In Teilaufgabe Aufgabe 5.5 werden außerdem noch Methodenzeiger vorgestellt.

Aufgabe 5.1 Erklärung `map`, `filter` und `reduce`

Arbeitet man auf iterierbaren Sequenzen, ist dies fast immer mit Schleifen über die Sequenz verbunden. Die drei Funktionen `map`, `filter` und `reduce` vereinfachen uns hierbei dir Arbeit. Hierzu ein Beispiel. Haben wir ein Vektor von Doubles und wollen jedes Element quadrieren, endet dies meist in dem folgenden Programmcode:

```
std::vector<double> numbers = { 1, 2, 3, 4, 5 };

for (std::vector<double>::iterator it = numbers.begin(); it != numbers.end(); ++it) {
    *it = square(*it); // squaring element podoubleed to by it
}

// numbers now [ 1, 4, 9, 16, 25 ]
```

Die Idee von der Funktion `map` ist es, genau dies zu vereinfachen. Sie erhält die Start- und Enditeratoren der Sequenz, einen Iterator, der auf eine Sequenz zeigt in dem die veränderten Elemente gespeichert werden und einen Funktionszeiger als Parameter und ruft diese Funktion auf jedes Element der iterierbaren Sequenz auf.

```
std::vector<double> numbers = { 1, 2, 3, 4, 5 };

map(numbers.begin(), numbers.end(), numbers.begin(), square);

// numbers now [ 1, 4, 9, 16, 25 ]
```

`filter` funktioniert analog, indem sie einen Funktionszeiger auf eine Funktion erhält, die ein Listenelementtyp erhält und ein `bool` zurück gibt. Auf alle Elemente wird diese Funktion aufgerufen und alle Elemente, für die die Funktion `true` zurückgibt, werden in die Ausgabesequenz kopiert. Der Rest wird entfernt.

```
std::vector<double> numbers = { 1, 2, 3, 4, 5 };

filter(numbers.begin(), numbers.end(), numbers.begin(), isOdd);

// numbers now [ 1, 3, 5, 4, 5 ]
```

Und auch `reduce` hat eine ähnliche Verwendung. Es schrumpft eine Sequenze zu einem Element zusammen. Hierbei wird der Ausgabeiterator gegen einen Startwert ausgetauscht. Hier ein Beispiel, bei dem die Summe über die Elemente in `numbers` gebildet wird.

```
std::vector<double> numbers = { 1, 2, 3, 4, 5 };

std::cout << reduce(numbers.begin(), numbers.end(), 0.0, sum) << std::endl; // 15
```

Aufgabe 5.2 Programmieren der Funktionen

Du wirst die drei Funktionen nachprogrammieren. Hierbei geht es erstmal darum ein funktionierendes Gerüst der Methoden zu erstellen, anstatt perfekt generische Algorithmen zu erhalten. Darum wird sich im Laufe der Aufgabe gekümmert.

Aufgabe 5.2.1 `map`

Schreibe eine Funktion `map` die folgende Signature besitzt.

```
template<typename InIt, typename OutIt>
OutIt map(InIt first, InIt last, OutIt out_first, double(*func)(double d));
```

Hierbei ist der letzte Parameter der Funktionszeiger. Die Klammern um `*func` sind deshalb notwendig, damit die Sichtbarkeit sichergestellt ist und der Kompiler den übergebenen Parameter als Funktionszeiger einer Funktion mit Rückgabewert `double` interpretiert und nicht als Funktion mit Rückgabewert `double *`[1]. Diese Funktion hat zusätzlich noch einen Double `d` als Paramenter.

Aufgabe 5.2.2 `filter`

Die von dir zu schreibende Funktion `filter` soll der folgenden Signatur folgen.

```
template<typename InIt, typename OutIt>
OutIt filter(InIt first, InIt last, OutIt out_first, bool(*pred)(int i));
```

Aufgabe 5.2.3 `reduce`

Erstelle eine Funktion `reduce`, die der folgenden Signatur folgt.

```
template<typename InIt, typename RetT>
RetT reduce(InIt first, InIt last, RetT initialVal, RetT(*func)(RetT i, double j));
```

Hierbei muss ein passender initialer Wert übergeben werden, der mit dem Rückgabewert und dem ersten Argument der übergebenen Funktion zusammenpasst.

Aufgabe 5.2.4 Passende Hilfsfunktionen implementieren

Implementiere in dieser Aufgabe drei Hilfsfunktionen, die den Anforderunger der jeweiligen Signaturen der Funktionszeiger in den Funktionen `map`, `filter` und `reduce` folgt. Du kannst dir dabei gerne eigene Funktionen ausdenken oder dich an die Funktionen in den Beispielen halten.

Teste anschließend deine Implementierung, indem du die Funktionen nur anhand ihres Namens übergibst.

Aufgabe 5.3 Funktoren

Es gibt noch die Möglichkeit Funktionen in einem Funktionsobjekt (*Funktor*) zu schachteln. Dabei überläd man den Operator `operator()`, welcher eine bestimmte Funktion ausführt.

Schaut man sich in unserem Beispiel die Funktion `square` an, welche folgende Signatur hat `double square(double i);` würde der Funktor wie folgt aussehen.

[1] Welche folgende Signature hätte: `double *(*func)(double d)`

```
class Square {
public:
   double operator() (double i) { return i*i; }
};
```

Die Funktion `map` würde wie folgt umgeschrieben werden müssen.

```
template<typename InIt, typename OutIt>
OutIt map(InIt first, InIt last, OutIt out_first, Square s);
```

Schreibe dein Hilfsfunktionen als Funktoren und füge eine neue Implementierung für deine Funktionen `map`, `filter` und `reduce` hinzu, die mit den Funktoren kompatibel ist. Anschließend stelle sicher, dass deine Implementierung noch funktioniert.

Aufgabe 5.4 Verwendung von Templates

Die zurzeitige Implementierung funktioniert entweder mit Funktoren einer bestimmten Klasse oder mit Funktionszeigern, die einem bestimmten Typen angehören, der durch die Signatur der Funktion gebunden ist. Das ist nicht immer das gewünschte Ergebnis. Um das Problem zu lösen kann man den Übergabetypen durch ein Template setzen, wodurch der Typ der Übergabe nicht mehr relevant ist. Dadurch ist die Parameteranzahl und die Typen der Parameter losgelöst und jede beliebige Funktion kann übergeben werden. Innerhalb der Funktion muss nur darauf geachtet werden, dass dem Funktionszeiger/Funktor die richtige Variable übergeben wird.

Implementiere in deinen Funktionen `map`, `filter` und `reduce` einen weiteren Templatetypen indem du die Liste der Templatetypen erweiterst zu

```
template <typename InIt, typename OutIt, typename ...>
```

und verwende diesen statt der zur Zeit hardgecodeten Signatur für die übergebene Funktion/den übergebenen Funktor. Teste deine Implementierung anschließend sowohl mit Funktoren, als auch mit Funktionen.

Aufgabe 5.5 Methodenzeiger

Es gibt auch die Möglichkeit Methoden[2] via Zeigern auszuführen. Hierzu muss man zusätzlich zu der Funktion noch ein Objekt übergeben, auf welches die Funktion ausgeführt wird.

In unserem Beispiel von `Square` fügen wir noch eine weitere Methode `inverse` hinzu, welche das Inverse der Quadrierung, die Wurzeloperation, ausführt.

Unsere Klasse Square verändert sich dementsprechend zu

```
class Square {
public:
   double operator() (double i)
   double invert(double i)
};
```

und unser `map` zu

```
template <typename InIt, typename OutIt, typename ObjType>
OutIt map(InIt first, InIt last, OutIt out_first, ObjType *object, double (ObjType::* method)(
    double));
```

Deine Aufgabe ist es nun eine neue Implementierung von `map` hinzuzufügen, deinen für Map geschriebenen Funktor eine weitere Methode hinzuzufügen und anschließend deine Implementierung mit allen implementierten Methoden zu testen.

[2] Hier ist eine einfache Erklärung zu dem Unterschied von Funktion und Methode zu finden `http://stackoverflow.com/a/155655`

Nachwort

Für produktive C++-Programme bietet der Standard fertige Funktionen und Klassen, um das, was hier erarbeitet wurde zu realisieren, z.B. `std::function<...>`[3] und `std::bind()`[4]. Diese können mit beliebiger Anzahl von Parametern umgehen und beinhalten viele weitere Features.

[3] http://en.cppreference.com/w/cpp/utility/functional/function

[4] http://en.cppreference.com/w/cpp/utility/functional/bind

Übungen für den 4. Tag

Aufgabe 6 Callbacks

Motivation für Callbacks

In dieser Aufgabe werden mehrere Methoden zur Realisierung von Callbacks in C++ vorgestellt und implementiert. Callbacks können als Alternative zum Observer Pattern[5] 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.

Aufgabe 6.1 Basisalgorithmus

Implementiere folgenden Algorithmus, der das Problem der Türme von Hanoi löst.[6]

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
```

Aufgabe 6.2 Callbacks mit Funktionszeigern

Nun wollen wir 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 Implementation entsprechend um:

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

Nun können wir eine Funktion mit zwei Parametern an `hanoi()` übergeben.

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

Aufgabe 6.3 Callbacks mit Funktoren

Ein Nachteil der vorherigen Implementation 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:

[5] http://de.wikipedia.org/wiki/Observer_Pattern
[6] http://de.wikipedia.org/wiki/Turm_von_Hanoi

```
template<typename 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. *Funktoren*[7]). Dabei müssen nicht einmal die Parametertypen (`int`) exakt übereinstimmen, solange eine implizite Umwandlung durch den Compiler möglich ist.
Teste deine Implementation mit einem Funktor. Schreibe dafür eine einfache Klasse und überlade deren `operator()`:

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

Aufgabe 6.4 Callbacks mit `Callback`-Klasse

Probleme der bisherigen Implementation

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<typename 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 Kompilierzeit festgelegt werden müsste und es unmöglich wäre, diesen während der Laufzeit zu ändern.

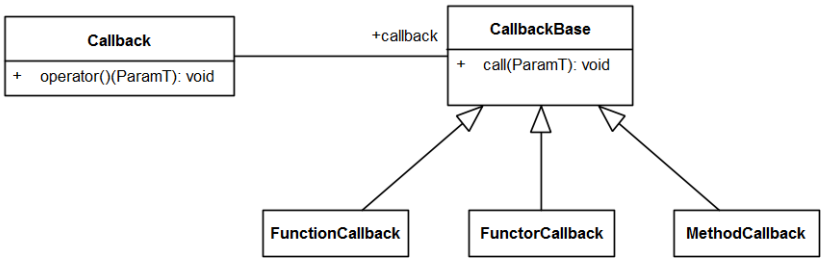
Lösung mittels `Callback`-Klasse

Deshalb werden wir eine Klasse schreiben, die beliebige Callbacks kapseln kann (`Callback`), 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 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.

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



`CallbackBase`
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-Parameter von `CallbackBase` sein soll. Der Klassenrumpf lautet also

```
template<typename 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.

Hinweise

- Du kannst diese und alle nachfolgenden Klassen in einem einzigen Header implementieren, weil die Klassen sehr kurz sind und außerdem semantisch stark zusammenhängen.

Aufgabe 6.5 Klasse `FunctionCallback`

Implementiere nun die erste Unterklasse `template<typename 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 Implementation. 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);
```

```
    // ...
}
```

Aufgabe 6.6 Klasse `FunctorCallback`

Implementiere nun die Unterklasse `template<typename ParamT, typename ClassT> FunctorCallback`. Zusätzlich zum Parameter-Typ muss hier auch der Typ der Funktor-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 Implementation!

Aufgabe 6.7 Klasse `MethodCallback`

Implementiere nun die letzte Unterklasse `template<typename ParamT, typename ClassT> MethodCallback`. Beachte, dass nun zwei Attribute nötig sind - ein Methodenzeiger und ein Zeiger auf das zu verwendende Objekt. Teste deine Implementation.

Hinweise

- Verwende beispielsweise folgende Signatur für den Konstruktor von `MethodCallback`: `MethodCallback(void(ClassT::*method)(ParamT), ClassT *object)`
- 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);`

Aufgabe 6.8 Klasse `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<typename 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` instantiieren, 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 Implementation in Zusammenhang mit der `hanoi`-Funktion. Du kannst das `Callback`-Objekt auch per Wert übergeben, da intern nur Zeiger kopiert werden.

Übungen für den 4. Tag

Aufgabe 7 C Einführung

In den nächsten Tagen werden wir 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.

Aufgabe 7.1 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);
}
```

Übungen für den 4. Tag

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;
```

Aufgabe 7.2 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);
```

Aufgabe 7.3 Ausgabe auf Konsole per `printf`

Um 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/`.

Aufgabe 7.4

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  ...
```

Aufgabe 7.5

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

Übungen für den 4. Tag

Aufgabe 8 Eigene Arrays (optional)

Die Klausur kann ohne diese Aufgabe bestanden werden. Wir empfehlen aber sie trotzdem zu bearbeiten.

Nachdem du bei unseren Übungen zu Arrays 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>

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

int main() {
    Array<std::string> stringArray(10);
    stringArray[0] = "Hello World";
    printFirst(stringArray);
}
```

Hinweise:

- Überlege dir, welche Operatoren/Methoden das obige Code-Beispiel von Array verlangt. Unter anderem musst du jeweils einen `const` und einen nicht-`const` `operator[]` implementieren.
- Du kannst auch Exceptions (z.B. `std::out_of_range` aus `<stdexcept>`) verwenden, um falsche Indices korrekt abzufangen.
- Eine fortgeschrittene Übung ist es, Iteratoren oder `operator+(unsigned int)` für `Array` bereitszustellen, sodass du z.B. die Funktion `std::copy` aus der Standardbibliothek verwenden kannst, um ein Array zu kopieren:

  ```
  #include <algorithm> // copy
  #include <iterator> // back_inserter
  #include <vector>
  // ...
  Array<int> array(10);
  std::vector<int> vector;
  std::copy(array, array + 4, std::back_inserter(vector));
  ```

- Diese Idee ist natürlich nicht neu. Seit C++11 gibt es eine Array-Implementation in der C++-Standardbibliothek (`std::array`[8]). Du findest die gleiche Klasse auch als `boost::array` in Boost.

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