LANDESBERUFSSCHULE 4 SALZBURG

Informatik

Standard Template Library (STL)

LBS 4

Inhalt

Standard Template Library (STL)

Als klares Abstraktionskonzept haben wir bisher das Konzept von Klassen kennen gelernt. Damit ist es möglich Methoden, Attribute, Daten usw. zu einem eigenen Typ zu vereinen. Es gibt auch Fälle in denen ein eigener Datentyp nicht ausreicht. Klassen besitzen einen konkreten Datentyp, Templates sind generisch, unabhängig formuliert.

Die STL bietet generische Lösungen für viele Problemstellungen an. STL baut ist auf drei Säulen aufgebaut:

- Container
  - Platzhalter für Elemente
- Iterator
  - Bewegen im Container, manipulieren der Elemente
- Algorithmus
  - Funktionen welche mit dem Container zusammenarbeiten

Als Beispiel dient das FIFO – Konzept (First In First Out). Im einfachsten Fall gibt es in der Klasse (Template) die Methode `put` welche ein Element auf den Stapel legt und die Methode `get` welche das Element wieder vom Stapel holt. Welcher Datentyp zum Einsatz kommt ist für dieses Konzept nicht wichtig, weil sich das Verhalten nicht ändert. Aber auch für Methoden und Funktionen kann dieses Konzept verwendet werden.

Als weiteres Beispiel dient eine Funktion welche ein Array sortiert, solange der Datentyp die Operatoren >, < sowie den Vergleichsoperator == und den Zuweisungsoperator = unterstützt, ist dieses sortierbar.

Genau diese beiden Beispiele beschreiben das Prinzip der generischen Programmierung. Es wird eine Klasse, Funktion oder Methode ohne den Datentyp zu berücksichtigen geschrieben. Sehr oft ist es nötig Datentypen, Klassen für unterschiedliche Datentypen zu implementieren. Voraussetzung dafür ist

- die Klasse muss so allgemein formuliert sein, dass der Datentyp egal ist
- bei der Verwendung muss der Datentyp über einen Parameter bestimmt werden.

In C++ sind Templates der Mechanismus, der es erlaubt, generische Datentypen zu schreiben und diese dann durch Typ-Parametrisierung zum Einsatz zu bringen.

Die Auswertung der Templates geschieht zur Compilierzeit, es gibt keine Laufzeitverzögerungen. Durch die Generalisierung wird eine einfache Wartung des Quellcodes ermöglicht. Die STL ist im Namensraum *std* definiert.

Man unterscheidet zwischen

- Funktionstemplate (function-template)
    - definiert eine Gruppe von Statements und verwendet Parameter statt konkrete Typen
- Klassentemplate (class-template)
    - definiert eine Klasse mit Parameter statt eines konkreten Datentyps

Vorteile der STL

- bieten gutes und stabiles Laufzeitverhalten
- definiert Standards für viele Lösungen
- Problemlösung weit verbreitet

Container, Iterator und Algorithmus

Container:

Ein Container entspricht einem Objekt, welches eine Kollektion von anderen Objekten verwalten kann. Ein Container verwaltet den Speicherplatz und bietet Methoden zur Verwaltung der Objekte an. Man unterscheidet zwischen Sequenzen (array, vector, queue, stack, ..) und assoziative Container(map, set).

Sequenzielle Container ordnen einzelne Objekte linear, der Reihe nach und können über die Position angesprochen werden (vector, list, deque). Aus diesen werden die Adapterklassen generiert (stack, queue).

Ein ***vector*** ist im Prinzip ein dynamisches Array welches die Anzahl der Elemente selber verwaltet. Die Elemente werden sequenziell gespeichert. Das Einfügen eines Elements am Ende des Vektors geht sehr schnell, am Anfang wird etwas mehr Zeit benötigt, weil das Array neu organisiert werden muss.

```
std::vector<std::string> my_vector;


my_vector.push_back("first");
my_vector.push_back("second");
my_vector.push_back("first");

std::cout << "size of vector: " << my_vector.size() << std::endl;
```

Eine ***queue*** stellt die Grundfunktionalitäten eines FIFO-Speichers (first-in first-out) zur Verfügung. Das stellt das Grundprinzip einer Warteschlange dar. Das erste Element wird als erstes verarbeitet, das letzte Element wird zum Schluss verarbeitet.

```
queue<std::string> my_queue;

my_queue.emplace("first");
my_queue.emplace("second");
my_queue.emplace("third");
std::cout << my_queue.front()  << std::endl;

std::cout << "size of queue: " << my_queue.size() << std::endl;
```

Ein ***stack*** beschreibt einen LIFO-Speicher (last-in first-out). Dieser wird z.B.: beim Stack verwendet. Das letzte Element am Stapel wird als erstes wieder heruntergenommen.

```
std::stack<std::string> my_stack;

my_stack.push("first");
my_stack.push("second");
my_stack.push("third");
std::cout << my_stack.empty()  << std::endl;

std::cout << "size of stack: " << my_stack.size() << std::endl;
```

Eine list stellt einen Container dar, welcher in beiden Richtungen navigieren kann. In C++ wird dieser Container als doppelt verkettete Liste implementiert. Es besteht kein indizierter Zugriff auf einzelne Elemente.

```
list<std::string> my_list;

my_list.push_back("first");
my_list.push_back("second");
my_list.push_back("first");

std::list<std::string>::iterator it_begin = my_list.begin();

for(it_begin; it_begin != my_list.end(); it_begin++){
   std::cout << *it_begin << std::endl;
}
```

Assiziative Container werden in einer Baumstruktur gespeichert. Der Zugriff erfolgt über einen Schlüssel (map, multimap, set, multiset).

Eine ***map*** beschreibt eine Beziehung zwischen zwei Mengen, welche ein Schlüssel-Wert-Paar ist. Der Schlüsselwert darf nicht doppelt vergeben werden.

```
std::map<std::string, int> my_map;

my_map.insert(std::pair<std::string,int>("first",1));
my_map.insert(std::pair<std::string,int>("second",2));
my_map.insert(std::pair<std::string,int>("first",3));

std::cout << "size of map: "<< my_map.size()  << std::endl;
```

Das Einfügen des dritten Eintrages scheitert, doppelter Wert.

Ein ***set*** ist eine Ansammlung unterschiedlicher Elemente. Es kann keine doppelten Einträge geben.

```
std::set<std::string> my_set;

my_set.insert("first");
my_set.insert("second");
my_set.insert("first");

std::cout << "size of set: "<< my_set.size()  << std::endl;
```

Die Ausgabe ist 2, da der dritte Eintrag doppelt ist.

Iterator:

Iteratoren arbeiten ähnlich wie Zeiger. Iteratoren bilden das Bindeglied zwischen einem Container und dem Algorithmus der verwendet wird. Mit einem Iterator kann man schrittweise, sequenziell auf Datenelemente in einem Container zugreifen, ohne dass der Datentyp bekannt sein muss. Ein Iterator kann sich von einem Element zum nächsten Element bewegen, die Methode dahinter ist nicht veränderbar/sichtbar (Kontrollabstraktion). Iteratoren werden in fünf Gruppen eingeteilt:

Input-Iterator: sequenzielles Lesen von Daten

Output-Iterator: zum Schreiben von Daten (Dereferenzierungsoperator)

Forward-Iterator: es können Werte des Iterator gespeichert werden

Bidirektional-Iterator: kann zusätzlich mit – rückwärts bewegt werden

Random-Access-Iterator: zusätzlich beherrscht dieser den wahlfreien Zugriff auf Elemente.

Iteratoren können aus der STL verwendet werden, *#include<iterator>.* Mittels diesem kann sehr einfach der Anfang und das Endes eines Containers ermittelt werden.

```
std::vector<std::string> my_vector;

my_vector.push_back("first");
my_vector.push_back("second");
my_vector.push_back("first");

std::vector<std::string>::iterator it_begin = my_vector.begin();

for(it_begin; it_begin != my_vector.end(); it_begin++){
   std::cout << *it_begin << std::endl;
}
```

Algorithmus:

Die Template-Algorithmen arbeiten mit Iteratoren die auf Container angewendet werden. Durch das Zusammenwirken von

Container <->Iteratoren <->Algorithmen

gibt es eine Entkopplung, welche ein klares Design erlaubt.

Alle Algorithmen im Namensbereich **std::** arbeiten unabhängig von den Containertypen. Manche Container-Methoden haben denselben Namen wie die Algorithmen, werden aber anders verwendet.

Algorithmen werden in drei Gruppen eingeteilt:

- nichtveränderbare Algorithmen
  - diese lassen Elemente eines Containers unverändert (suchen, …)
- veränderbare Algorithmen
  - ändern die Elemente des Containers (mit Werte füllen, …)
- Algorithmen der C-Bibliothek
  - qsort(), bsearch(), Algorithmen für Mengen, ..

Eine Übersicht finden Sie auf der Seite cppreference.com.

Ausgesuchte, generische Algorithmen

#include<algorithm>

std::for_each ist im std definiert und gibt das aufrufbare Element als Ergebnis zurück. Wenn es mit einem Funktionsobjekt aufgerufen wird, kann das Ergebnis direkt im Objekt verwendet werden.

```
std::for_each (myvector.begin(), myvector.end(), myfunction);
```

std::find() benötigt als Parameter den Beginn sowie das Ende des Containers. Zusätzlich wird ein konstantes Objekt übergeben.

- gefunden: gibt den Iterator auf das gesuchte Element zurück
- nicht gefunden: gibt den Iterator auf das letzte Element zurück

```
it = std::find (vec.begin(), vec.end(), my_value);
```

std::sort() sortiert den Inhalt eines Containers aufsteigend. Als Parameter werden ein Zeiger auf Anfang und Ende benötigt. Weiters wird ein random access iterator benötigt.

std::find_if() benötigt als Parameter den Beginn sowie das Ende des Containers. Zusätzlich wird eine Funktion für die Bedingung übergeben.

```
bool isEven (int i) {
  return ((i%2)==0);
}
std::vector<int>::iterator it;
it = std::find_if (my_vec.begin(), my_vec.end(), isEven);
```

std::count() gibt die Anzahl der Elemente im Container zurück welche dem Vergleichsobjekt entsprechen.

```
my_count = std::count (my_vec.begin(), my_vec.end(), 20);
```

std::copy() kopiert Elemente aus einem Container in einen anderen. Als Parameter wird Anfang und Ende des originalen Containers sowie ein Zeiger auf den Anfang des neuen Speicherbereichs benötigt. Der Rückgabewert zeigt auf das letzte Element des kopierten Bereiches.

```
std::copy ( my_vec.begin(), my_vec.end(), cp_vec.begin() );
```

Testen der Algorithmen

Erstellen Sie eine Liste und einen Vektor, füllen Sie diese beiden mit Zufallszahlen. Testen Sie die generischen Algorithmen in der main()-Funktion, schreiben Sie die benötigten Funktionen für die Ausgabe im Terminal.

Aufgabe

Arbeiten Sie in den eingeteilten Gruppen die Themen mit den nachstehenden Anforderungen aus.

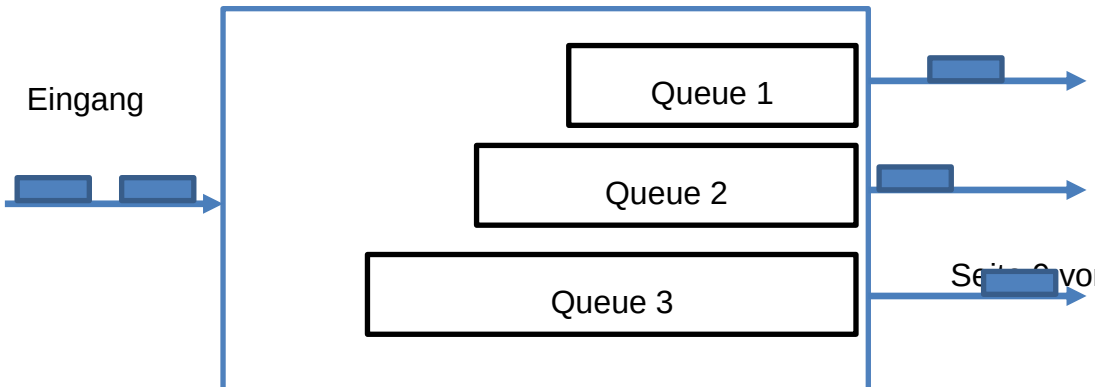
- Worddokument
  - Überschrift, Fuß- und Kopfzeile, Inhaltsverzeichnis, Bilder beschriftet
- Beschreibung
  - Verwendung mit Iterator, Aufbau, notwendige include, wichtige Methoden, Anwendungsbeispiel
- Präsentation
  - Zeigen Sie die Verwendung des Template und erklären Sie die einzelnen Schritte, mögliche Probleme und Herausforderungen
  - keine PowerPoint nötig

Container.

Vector, Liste, map (multimap), set (multiset), queue, stack deque

Übung:

Bei der Verbindung zwischen entfernten Rechnernetzen werden Router zur Datenkommunikation eingesetzt. Bevor die Datenpakete weitergeleitet werden, werden diese in Queues abgelegt. Der Router ist bestrebt die warteten Pakete so schnell wie möglich weiterzuleiten. Ein einfacher Algorithmus ist dafür der Hot-Potato-Algorithmus. Es werden die eintreffenden Pakete auf die kürzeste Queue gesetzt.



Verwenden Sie die Headerklasse VekQueue, deklarieren und implementieren Sie die fehlenden Methoden.

Der Konstruktor legt eine als Parameter übergebene Anzahl leerer Queues in einem Vektor an.

Methode size():

- ohne Parameter liefert die Methode die Anzahl aller Nachrichten
- mit Parameter vom Type int32_t gibt sie die Anzahl der Nachrichten in der i-ten Queue zurück.

Methode empty()

- ohne Parameter wird true zurückgegeben, wenn alle Queues leer sind
- mit Parameter wird true zurückgegeben, wenn die i-te Queue leer ist

Methode push()

- fügt eine als Argument übergebene Nachricht (Typ int32_t) am Ende der kürzesten Queue ein

Methode pop()

- ohne Parameter entnimmt pop() eine Nachricht aus einer zufälligen Queue
- mit Parameter entnimmt pop(typ x) eine Nachricht aus der i-ten Queue

Zum Testen verwenden Sie einen Container vom Typ VekQueue. Eine Nachricht ist eine Zahl! Fügen Sie in einer Schleife Zufallszahlen zwischen 0 und 99 in den Container ein. Entnehmen Sie einige Nachrichten aus den Queues (zufällig) und zeigen Sie das Ergebnis als Ausgabe an.

Zusatz:

Erstellen Sie weitere Nachrichten (Parameter als Anzahl) und fügen diese in der kürzesten Queue ein.

Sichern Sie die Methoden pop und push mit der Ausnahmebehandlung ab.