STL Container und ihre Verwendung in ressourcenkritischen Systemen

Piet Laur

Hochschule für Angewandte Wissenschaften Hamburg
piet.laur@haw-hamburg.de

February 1, 2017

Abstract

Der vorliegende Artikel gibt einen Überblick über den Einsatz von STL-Containern in ressourcenkritischen Systemen. Die Container werden dabei insbesondere auf Speicherverbrauch, CPU-Laufzeit und Echtzeitfähigkeit analysiert. Während meistens einfach ein vector verwendet wird, werden hier auch die Container gegeneinander verglichen und in bestimmten Operationen gemessen. So gibt es fünf verschiedene Möglichkeiten, einen vector mit einer Anzahl an Elementen zu initialisieren, die alle unterschiedlich schnell sind. Außerdem wird gezeigt, wie man Elemente in eine verkettete Liste einfügen und entfernen kann, ohne dabei die Speicherverwaltung zu benutzen. Basis dieses Artikels sind vor allem ein Artikel von Scot Salmon sowie die C++ Reference. Der Artikel ist für jeden Programmierer interessant, der STL-Container einsetzen möchte, dabei aber nicht viele Ressourcen zur Verfügung hat.

I. Einleitung

Die C++ Standard Template Library (STL) stellt einen Satz an Containern zur Verfügung, mit deren Hilfe Datensätze verwaltet werden können. In eingebetteten Systemen werden häufig dynamische Datenstrukturen benötigt, die eine variable Anzahl von Objekten aufnehmen können, während Ressoucen in der Regel knapp sind. Um zusätzlichen Programmieraufwand zu vermeiden kann auch in eingebetteten Systemen auf Klassen der STL (insbesondere Container) zurückgegriffen werden.

Der große Vorteil von C++ gegenüber C ist die Tatsache, dass der Programmierer viel schneller und übersichtlicher komplizierte Algorithmen und Systeme entwickeln und durch objektorientiertes Programmieren die Realität besser abbilden kann. Insbesondere dank einer umfassenden Standardbibliothek, die einem C-Programmierer nicht zur Verfügung steht, wird in C++ so effektiv programmiert. Wer hingegen einfach drauf los programmiert, ohne die Klassen (vor allem Container) der STL genauer zu kennen, wird schnell feststellen, das einige Operationen unerwartet viel CPU-Zeit verbrauchen oder viel Speicherplatz belegen. Wer in C programmiert, muss alle Speicherallokationen selbst durchführen und hat so stets den Überblick, wann wie viel Speicher allokiert wird. In C++ ist dies nicht der Fall, denn die Container-Klassen bemühen in vielen Operationen versteckt auch die Speicherverwaltung, ohne dass der Programmierer das sofort sieht – mit Auswirkungen auf die Echtzeitfähigkeit, Geschwindigkeit und den Speicherverbrauch der Anwendung. In ressourcenkritischen Systemen wird an den Komponenten Geld gespart, es sind daher minimale Systeme mit schwachen Prozessoren, wenig RAM und Cache und einer schwachen Speicherverwaltung. So kann es durch die Ahnungslose Verwendung der STL zu nichtdeterministischem Verhalten, Speicherfragmentierungen und generell langsameren Programmen führen.

Die aufgeführten Probleme sind kein Grund,

grundsätzlich in ressourcenkritischen Systemen auf die STL-Container zu verzichten, denn sie sind hoch effizient implementiert und eine eigene Entiwcklung entsprechender Container kostet viel Zeit und wäre zudem vermutlich Fehleranfällig. Wenn man die Container richtig verwendet und weiß, an welchen Operationen es zu Problemen kommen kann und wie man diese umgehen kann, so wird man die Vorteile von dynamischen Datenstrukturen zu schätzen lernen und kaum effizientere und einfacherere Implementationen als die der STL finden. Genau das, also die Probleme bei STL-Containern und wie man sie vermeidet oder ob man auf sie in ressourcenkritischen Systemen komplett verzichten sollte, wird in diesem Artikel umfassend beleuchtet, und der Leser soll anschließend STL-Container anwenden können, sodass möglichst wenig Ressourcen verbraucht werden. Kriterien sind hiebei Geschwindigkeit, Speichernutzung und Echtzeitfähigkeit.

II. Überblick

Es gibt zwei grundlegende Typen von Containern: sequentielle und assoziative Container. Sie unterscheiden sich darin, dass sequentielle Container immer die Reihenfolge der Elemente beibehalten und auch Elemente doppelt enthalten sein können, bei ihnen erfolgt der Zugriff über die Position. Assoziative Container hingegen garantieren keine Reihenfolge, während Elemente im Normalfall nicht doppelt enthalten sein dürfen. Bei assoziativen Containern erfolgt der Zugriff über einen Schlüssel.

Sequentielle Container sind insbesondere Listen, die STL bietet hierzu verschiedene Implementationen an, die sich in verschiedenen Szenarien unterschiedlich gut eignen:

- vector
- list
- forward_list
- deque

Assoziative Container sind Mengen und Maps, auch hierfür hat die STL mehrere Implementationen parat:

- set
- unordered_set
- map
- unordered_map

Die Implementationen unterscheiden sich nicht in ihrer Semantik und ihre Interfaces sind im Wesentlichen identisch. Der Ressourcenverbrauch (Speicher und CPU-Zeit) kann jedoch bei bestimmten Operationen von Implementation zu Implementation variieren.

i. Listen

Der populärste Container ist der vector. Vectors enthalten ein Array für die gespeicherten Elemente. Sobald das Array vollgelaufen ist, wird ein größeres allokiert und alle Werte werden in das neue kopiert. Für Datenmengen stark variierender Größe sind vectors daher nicht geeignet, dafür sind sie hinsichtlich Speicherverbrauch von allen sequentiellen Containern die sparsamsten.

Ist die Reihenfolge der Elemente im vector für den Anwender irrelevant, so sollten neue Elemente möglichst am Ende eingefügt werden (Methode push_back), da das Einfügen in der Mitte oder am Anfang viele Kopiervorgänge erfordert.

Wenn der Anwender schon im Voraus weiß, wie viele Elemente der vector ungefähr aufnehmen soll, so kann der vector mit einer bestimmten Arraygröße initialisiert werden:

```
// Dieser vector hat einen
// Startpuffer von 100
// Elementen
std::vector<int> con(100);
```

Listing 1: *Konstruktor des vectors*

Außerdem gibt es eine Methode shrink_to_fit, mit der ein ehemals großer vector auf die aktuelle Größe geformt wird und damit Speicher freigibt. Da die Daten in einem vector in jedem Fall in einem Block im Speicher liegen, kann über den []-Operator sehr schnell mit einem Index auf die Elemente zugegriffen werden. Ein besonderes Augenmerk sollte auch auf die Methode reserve gegeben werden,

mit ihr lässt sich die Kapazität des vectors zu jedem Zeitpunkt vergrößern.

```
// lege vector mit 100 Elementen an
std::vector<int> con(100);

// passe Kapazitaet auf Groesse an
con.shrink_to_fit();

// vergroessere Kapazitaet wieder
con.reserve(100);
```

Listing 2: *Kapazität des vectors*

Durch diese Methoden kann man die Kapazität des vectors ständig unter Kontrolle halten, um unerwartetes Volllaufen des Arrays zum Beispiel bei push_back() zu vermeiden.

Will der Programmierer regelmäßig neue Elemente an beliebigen Positionen der Liste hinzufügen oder entfernen, so ist die Anwendung einer list in Hinsicht auf CPU-Zeit viel Effizienter. Eine STL list ist eine doppelt verkettete Liste, es besteht also für jedes Element zusätzlicher Speicherverbrauch (für die Zeiger auf das vorige und kommende Element). Die forward_list ist eine einfach verkettete Liste, es kann also nur vorwärts iteriert werden, zugunsten des Speicherverbrauches.

Die STL deque (double ended queue) ist ein Hybrid aus verketteter Liste und Array-Liste. Dabei werden Elemente in Element-Blöcken fester Größe gespeichert, die jeweils miteinander verlinkt sind. Dadurch wird die Effizienz einer list beim Einfügen oder Entfernen von Elementen am Anfang oder am Ende der Liste beibehalten, während durch das Speichern in Blöcken der geringe Speicherverbrauch eines vectors angestrebt wird.

Zur Implementation einer Queue oder eines Stacks sollten die Adapter-Klassen queue und stack verwendet werden, die intern eine list benutzen. Es gibt sogar die priority_queue, die ihre Daten als Heap organisiert.

ii. Mengen

In c++ gibt es zwei Implementationen von Mengen: Die set ist intern mit einem binären Suchbaum implementiert und erlaubt somit geordnetes Iterieren über ihre Elemente. Die unordered_set hingegen ist eine Hash-Menge, bei ihr ist der Zugriff auf Elemente über den Wert sehr schnell. Beide haben gemeinsam, dass jedes Element nur einmal enthalten sein kann. Gleichheit wird mit dem ==-Operator getestet, den man für Klassen überladen kann.

Im Falle der set muss eine Komparatorklasse geschrieben werden, mit deren Hilfe die Elemente im Baum angeordnet werden. Sie ist als Templateparameter zu übergeben. Durch die Baumstruktur wird, ähnlich wie bei der doppelt verketteten Liste, zusätzlicher Speicherplatz für die Zeiger auf die Kindknoten verbraucht (2 Zeiger pro Element, da binärer Suchbaum).

Die unorered_set funktioniert mit einer Hashklasse, die ebenfalls als Templateparameter übergeben wird und zu jedem Element einen Hashwert zurückgibt. Elemente werden in Behältern von gleichen Hashwerten gespeichert, dadurch ist der Zugriff sehr schnell, während keine Reihenfolge existiert.

Für den speziellen Fall, dass Elemente auch doppelt enthalten sein dürfen, gibt es auch noch die multiset und die unordered_multiset. Für sie gilt das gleiche wie für die set und die unordered_set. Es besteht jedoch selten ein Sinn darin, identische Objekte zweimal in einem Container abzulegen.

Wegen der Popularität des vectors wird er oft auch in Fällen angewendet, in denen die Benutzung einer set sinnvoller wäre. Vor allem der Zugriff über den Wert kann bei großen Mengen in einer set erheblich effizienter als in einem vector sein.

iii. Maps

Maps sind Schlüssel-Werte-Paare, sie werden gerne im Zusammenhang von IDs eingesetzt: Es existiert eine Menge von Objekten, und es ist das Objekt mit einer bestimmten ID zu erhalten. Mit einer Map würde man hierfür die ID als Schlüssel und das Objekt als Wert nehmen und könnte so dirket auf das Objekt zugreifen. Maps sind genauso implementiert wie Mengen, nur das jedes Element (Schlüssel) noch den Wert als Anhängsel hat. Es gibt die map, unordered_map, multimap und un-

ordered_multimap. Für die Auswahl gilt das gleiche wie für die Mengen.

Um die Auswahl des geeigneten Containers für einen Anwendungsfall zu finden, hilft der Cheat-Sheet in *Figure 1* (Am Ende des Dokuments).

Dieser Entscheidungsgraph bietet für einen gegebenen Anwendungsfall einen allgemein geeigneten Container, ihm sollte allerdings nicht blind gefolgt werden. Wenn für einen Anwendungsfall beispielsweise eine set die schnellste Lösung wäre, es dem Programmierer aber egal wäre, wie lange die Operationen dauern, er will nur so wenig Speicher wie möglich verbrauchen, so sollte er eventuell auf einen in seinem speziellen System geeigneteren Speichersparenden Container zurückgreifen.

Alle Container haben eines gemeinsam: Sie sind Template-Klassen, als erster Templateparameter wird der zu speichernde Datentyp übergeben. Der Datentyp kann alles sein, was ein Objekt ist und sich kopieren lässt (denn beim einfügen der Elemente werden diese kopiert): Primitive Datentypen, Klassen sowie Zeiger oder sogar weitere Container-Klassen. Dies ermöglicht eine ganze Reihe an Möglichkeiten, wie Elemente genau in den Containern abgelegt werden, und das hat auch Auswirkungen auf die Geschwindigkeit und den Speicherverbrauch. Referenzen können nicht als Datentyp verwendet werden, denn sie sind keine Objekte und lassen sich nicht kopieren (Es würde zu Kompilierungsfehlern kommen). Mit C++11 wurden Smartpointer eingeführt, die eine neue attraktive Art darstellen, Daten in Containern zu speichern. Sie bieten die Effizienz von normalen Pointern, umgehen aber das Risiko von Speicherzugriffsfehlern und Speicherlecks.

Wenn die Datentypen, die in einem Container hinterlegt werden sollen, nicht größer als 16 Byte sind, lohnt es in der Regel nicht, sie als Zeiger zu speichern. Primitive Datentypen werden in der Regel als Wert gespeichert, denn der Kopiervorgang des kompletten Typs dauert nicht länger als der Kopiervorgang eines Zeigers. Je größer der Datentyp wird, desto langsamer kann die Nutzung eines Containers per Wert sein. Wenn vectors volllaufen und die Daten in einen größeren Buffer kopiert werden, macht es einen großen Unterschied, ob nur 8 Byte oder 100 Byte pro Element kopiert werden müssen. Schwächere Prozessoren, die nicht viele DMA-Channel zur Verfügung haben, leiden unter solchen Kopiervorgängen besonders. Statt die Elemente also als Wert in den Container zu legen, werden nur Zeiger auf die Elemente in den Container abgelegt.

Werden Zeiger abgelegt, muss man darauf achten, dass man die Elemente mit new allokiert hat, denn wenn man Zeiger auf Stackvariablen in Container legt, verlieren sie ihre Gültigkeit, sobald der Scope verlassen wird und es kommt zu Speicherzugriffsfehlern. Das bedeutet auch, dass man die Elemente selbstständig mit delete freigeben muss, um Speicherlecks zu umgehen. Bis C++11 wurde genau das gemacht, da es keine anderen Möglichkeiten gab (außer auf Bibliotheken wie Boost zurückzugreifen).

Mit C++11 wurden Smartpointer eingeführt, die das Freigeben des Speichers übernehmen und in jedem Fall den normalen Zeigern vorzuziehen sind. Smartpointer blähen zwar den Programmtext etwas auf, dafür ist ihre Anwendung wesentlich sicherer, sie sind auch schlank und bedeuten kaum (etwa 8 Byte) Speicher-Overhead.

```
#include <vector> // fuer std::vector
#include <memory> // fuer std::shared_ptr


class Element
{
    char data[1024];
}

int main()
{
    // Primitive Typen werden als
    // Wert gespeichert
    std::vector<int> container1;

    // Grosse Typen werden als Zeiger
    // gespeichert. Achtung: Speicher
    // muss selbst allokiert und
    // freigegeben werden, daher
    // sollte diese Variante nicht
```

```
    // verwendet werden.
    std::vector<Element*> container2;

    // Diese Variante ist zu bevorzugen,
    // da Smartpointer sich selbst
    // verwalten.
    std::vector<std::shared_ptr<Element>>
        container3;
}
```

Listing 3: *Elementtyp*

iv. Iteratoren, Threadsafety

Eine wichtige Rolle im Zusammenhang mit Containern spielen die iterators. STL-Container haben immer auch einen Iteratortypen, mit dem man über ihre Elemente iterieren kann. Was also Pointer für ein Array sind, sind Iteratoren für einen Container. Tatsächlich fühlen sich Iteratoren auch genau an wie Pointer: sie implementieren alle den ++, * und den != Operator. Alle Containertypen haben die Methoden begin() und end(), mit denen man einen Iterator auf das erste bzw. letzte Element des Containers enthält. Angewendet werden Iteratoren oft in for-each-Schleifen:

```
    std::vector<int> con;

    // fill con with elements...

    for (std::vector<int>::iterator it =
      con.begin(); it != con.end(); it++)
    {
        int i = *it; // Dereferenzierung
    }
```

Listing 4: *Iteratoren*

Iteratoren haben außerdem den Vorteil, dass sich durch sie einfache Operationen auf alle Containertypen gleich aussehen lassen. Die C++-Bibliothek Algorithm bietet eine Menge Funktionen zum Suchen, Sortieren, Bearbeiten (und Vielem mehr) über Container, die Iteratoren als Parameter erwarten und dadurch auf allen Containertypen arbeiten können. Die Funktionen der Algorithm-Bibliothek erweitern die Funktionalitäten der Container, sind aber auch mit Vorsicht zu benutzen, da auch sie im Hintergrund eventuell viel mehr tuen, als man zunächst vermutet.

Alle Methoden der Containerklassen sind theadsafe, sie dürfen also parallel aufgerufen werden und es ist dabei garantiert, dass es zu keinen Raceconditions kommt. Eine Außnahme stellt dabei der Containertyp vector<bool> dar. Er wird in der STL als spezialfall behandelt (Template-Specialization), denn die einzelnen bools werden Bitweise gespeichert, um Speicherplatz zu sparen. Es wird ausdrücklich darauf verwiesen, dass daher dieser spezielle Typ nicht threadsafe ist.

v. Allocators

Da Speicherallokationen oft der entscheidende Flaschenhals sind, wird in diesem Artikel darauf das größte Augenmerk gelegt. Zugriffe auf die Speicherverwaltung sind meistens langsam und bieten keine Echtzeitbedingungen, außerdem muss der Speicher bei vielen kleinen Allokationen öfter fragmentiert werden, was das gesamte System verlangsamt. Sie zu vermeiden ist also hohe Priorität, ist jedoch nicht immer möglich.

Gibt man es nicht explizit anders an, so wird zur Allokation von Speicher der new-Operator benutzt und die Speicherverwaltung des Betriebssystems wird aufgerufen. Man kann allerdings auch eine eigene Speicherverwaltung für die Elemente der Container implementieren, die im gegebenen Anwendungsfall schneller ist oder eventuell sogar Echtzeitbedingungen erfüllt. Dazu hat jeder Containertyp neben dem ersten Templateparameter (Elementtyp) auch noch einen zweiten Templateparameter: den Allocator. Der Allocator muss eine Klasse sein, die für einen Typen (den value_type) ein paar Methoden implementiert, darunter vor allem *allocate* und *deallocate*. Man kann beispielsweise zur Kompilierungszeit einen Memory-Pool anlegen, der Speicherstücke immer gleicher Größe vergibt. In *allocate* wird dann so ein Speicherstück vergeben und in *deallocate* wieder zurückgenommen. Ein solcher Memory-Pool müsste nie fragmentiert werden und könnte somit Speicher in Echtzeit vergeben. Eine minimale Allocator-Klasse sieht etwa so aus:

```
#include <cstddef>
template <class T>
struct SimpleAllocator {
    typedef T value_type;
    SimpleAllocator();
    template <class U> SimpleAllocator
      (const SimpleAllocator<U>& other);
    T* allocate(std::size_t n);
    void deallocate(T* p, std::size_t n);
};
template <class T, class U>
bool operator==(const SimpleAllocator<T>&,
    const SimpleAllocator<U>&);
template <class T, class U>
bool operator!=(const SimpleAllocator<T>&,
    const SimpleAllocator<U>&);
```

Listing 5: *Minimaler Allocator*

Minimaler Allocator. Quelle: Allocators

III. Implementierung

Im Folgenden wird die Nutzung der STL-Container *vector*, *list* und *map* anhand von Quelltext gezeigt und darauf eingegangen, an welchen Stellen Ressourcen gespart werden können. Es werden Implementationen gegenübergestellt und dabei CPU-Zyklen gemessen. Alle Tests hierfür wurden mit dem GCC-Compiler der Version 4.9.4 für Linux (64-Bit) kompiliert und auf einem Intel Core i7-6700HQ ausgeführt. Es geht dabei lediglich um den Vergleich der Implementationen, die Verhältnisse sind auf allen Umgebungen ähnlich.

i. vector

Der vector ist der Standardcontainer, da er sehr einfach zu benutzen ist. Wenn eine Menge von realen Dingen (z.B. Produkten auf einem Laufband) in Software abgebildet werden soll, so wird sie meist in einem vector gespeichert. Schon beim Anlegen des Containers hat man verschiedene Möglichkeiten, wie in folgendem Quelltext gezeigt wird. Aufgabe soll es zunächst sein, einen vector zu erstellen und mit einer bekannten Anzahl von Objekten (hier Integern) zu füllen.

```
// Variante 1
std::vector<int> con1;
for (int i = 0; i < 100000000; ++i) {
    con1.push_back(i);
}

// Variante 2
std::vector<int> con2(100000000);
for (int i = 0; i < 100000000; ++i) {
    con2[i] = i;
}

// Variante 3
std::vector<int> con3;
con3.resize(100000000);
for (int i = 0; i < 100000000; ++i) {
    con3[i] = i;
}

// Variante 4
std::vector<int> con4;
con4.reserve(100000000);
for (int i = 0; i < 100000000; ++i) {
    con4.push_back(i);
}

// Variante 5
// ACHTUNG! Undefiniertes Verhalten!!
std::vector<int> con5;
con5.reserve(100000000);
for (int i = 0; i < 100000000; ++i) {
    con5[i] = i;
}
```

Listing 6: *Befüllen des vectors*

Die oberen vier Varianten liefern das selbe Ergebnis: Einen *vector* mit den Zahlen von 0 bis 99999999 enthalten. Was im Hintergrund geschieht ist jedoch bei jedem anders. Variante 1 legt zunächst einen vector *con1* an, der die Default-Kapazität 1 hat. Diese Kapazizät wird bei Bedarf vergrößert, genaugenommen wird sie immer verdoppelt, also erst 2, dann 4, 8, 16, 32, ..., 134217728. Der vector wird insgesamt 27 mal erweitert, denn $\log_2(100000000) = 27$. Dieses Erweitern kann natürlich vermieden werden, wenn man schon im Voraus weiß, wie viele Elemente man einfügen möchte. In Variante 2 wird als Konstruktorparameter die Startkapazität angegeben, sodass der vector in der Schleife beim Füllen nicht mehr erweitert werden muss. Hinter dem Konstruktor steckt jedoch mehr als die Kapazität, als zweiten Parameter gibt man eigentlich den Initialisierungswert für jedes Element im vec-

tor an. Lässt man den Parameter weg (wie in diesem Fall), wird 0 genommen. Der vector wird also zunächst mit 100000000 Nullen gefüllt, bevor die eigentlichen Zahlen in der Schleife eingegeben werden, was natürlich unnötig viel Zeit kostet. Variante 3 tut im wesentlichen das gleiche wie Variante 2, doch man kann sie auch noch anwenden, nachdem der vector bereits erstellt wurde. In Variante 4 wird der Speicher nicht initialisiert (wie in 2 und 3), dafür muss die Methode *push_back()* zum Einfügen von Elementen verwendet werden. In Variante 5 wird der Speicher ebenfalls nicht initialisiert und es wird der []-Operator benutzt, dadurch ist sie in etwa so schnell wie ein normales Array, leider aber in C++ nicht erlaubt: auch wenn diese Variante in einigen Systemen funktioniert, kann sie undefiniertes Verhalten verursachen, da mit dem []-Operator keine Boundary-Checks durchgeführt werden und über die Größe des vectors hinausgeschrieben wird. Die Kapazität wurde vorher erweitert, daher kommt es in der Regel zu keinen Speicherzugriffsfehlern. Die folgende Tabelle zeigt den Unterschied:

Table 1: *Erstellen eines vectors*

Variante	CPU-Zyklen
Variante 1	758386762
Variante 2	163707484
Variante 3	225346544
Variante 4	449848154
Variante 5	85258722

Überraschend ist zunächst, dass Variante 4 so viel schlechter ist als Variante 3. Der Grund dafür ist die Methode *push_back()*: in ihr wird nicht nur das Element gesetzt (wie mit dem []-Operator), sondern auch überprüft, ob das Element noch in das Array passt, und die Größe wird inkrementiert – undzwar für jeden Schleifendurchlauf. In diesem Fall ist also Variante 2 zu bevorzugen (oder Variante 3, wenn der vector bereits existiert).

Möchte man den vector in Echtzeitanwendungen verwenden, sollte man insbesondere beim Einfügen neuer Elemente vorsichtig sein, denn es kann passieren, dass die Kapazität ausgeschöpft ist und erweitert werden muss, und dann wird die Deadline verletzt. Um dass zu verhindern, kann man Elemente auch auf folgende Weise einfügen:

```
template<class T>
bool test_push_back(std::vector<T> v,
        const T& t)
{
    if (v.capacity() > v.size() {
        v.push_back(t);
        return true;
    } else {
        return false;
    }
}
```

Listing 7: *Vorsichtiges Einfügen*

In der generischen Funktion *test_push_back* wird das Element nur angefügt, wenn das ohne Erweitern des Arrays in konstanter Zeit passieren kann. Andernfalls wird *false* zurückgegeben und das Element wird nicht angehängt. Die Priorität, eine Deadline einzuhalten, kann in diesem Falle größer sein, als das Element in den vector hinzuzufügen. Das Erweitern des vectors kann später in einem Thread ohne Echtzeitanforderungen passieren (mit der Methode *reserve*), somit kann der vector auch in Echtzeitsystemen verwendet werden.

ii. list

Die list bietet gegenüber dem vector einige Vorteile, vor allem den, dass das Einfügen und Entfernen von Elementen in konstanter Zeit (O(1)) geschieht. Leider entspricht das nicht ganz der Wahrheit, da in der Methode *insert* die Speicherverwaltung bemüht wird, denn das Element wird dabei kopiert, und das macht die Laufzeit dieser Methode genaugenommen nichtdeterministisch. Dank des C++ Allocator-Konzepts ist es für lists möglich, einen Memorypool zur Verfügung zu stellen. Diese eigene kleine Speicherverwaltung müsste nur Speicher fester Größe vergeben (Größe der Elemente). Das Programmieren eines eigenen Allocators ist allerdings etwas aufwendig, da viel Boilerplate-Code nötig ist, um die sogenannten

Allocator-Requirements zu erfüllen. Im Folgenden wird daher eine andere Lösung vorgestellt.

Wenn Elemente in eine Liste eingefügt werden sollen, die bereits in einer anderen Liste enthalten sind, kann man dies auch tun, ohne die Speicherverwaltung dabei zu bemühen, undzwar mit der Methode *splice*. So kann man zu der eigentlichen list eine zweite list anlegen, die die alten, nicht mehr gebrauchten Elemente enthält. Möchte man jetzt in die eigentliche list ein neues Element einfügen, nimmt man sich einfach eine Schablone von den alten list-Elementen und überschreibt deren Wert.

```
int N = 100000;

// Variante 1
std::list<int> con1;
for (int i = 0; i < N; ++i) {
    con1.push_back(i);
}
// Entferne alle Elemente
for (int i = 0; i < N; ++i) {
    con1.pop_back();
}
// Fuege wieder Werte ein
for (int i = N - 1; i >= 0; --i) {
    con1.push_back(i);
}

// Variante 2
std::list<int> con2;
std::list<int> garbage;
for (int i = 0; i < N; ++i) {
    con2.push_back(i);
}
// Entferne alle Elemente
for (int i = 0; i < N; ++i) {
    garbage.splice(garbage.end(),
        con2, con2.begin());
}
// Fuege neue Werte ein
for (int i = N - 1; i >= 0; --i) {
    con2.splice(con2.end(),
        garbage, garbage.begin());
    con2.back() = i;
}
```

Listing 8: *Einfügen und Entfernen bei list*

Beide Varianten führen Semantisch die gleichen Operationen auf con1 bzw. auf con2 aus. Variante 1 belastet die Speicherverwaltung aber sehr, da beim Einfügen und Entfernen von Elementen immer auch die Elemente mit ihren Links allokiert bzw. freigegeben werden. In Variante 2 geschieht dies nicht: Anstatt die Elemente komplett zu löschen, werden sie nur in eine andere Liste verschoben und werden anschließend mit neuen Werten zurück in con2 geschoben. Hier der Vergleich in CPU-Zyklen:

Table 2: *insert und erase in lists*

Variante	CPU-Zyklen
Variante 1	20854966
Variante 2	12061086

Variante 2 ist etwa 1.7 mal so schnell (der Unterschied hängt sehr von der Speicherverwaltung selbst ab). Die Methoden *push_back* und *pop_back* machen das gleiche wie *insert* und *erase*, bloß am Ende der Liste. Der Vorteil von Variante 2 gilt auch für das Einfügen und Entfernen an allen anderen beliebigen Positionen der Liste (da es eine verkettete Liste ist). Für Systeme, in denen dieser Faktor tatsächlich relevant ist, kann man eine Wrapperklasse um die list bauen, die anstelle des *insert* und *erase* bloß zwischen zwei Listen hin und her schiebt, sodass es sich von außen ganz normal anfühlt, trotzdem aber den Geschwindigkeitsvorteil bringt. Dabei muss man natürlich im Hinterkopf behalten, dass dafür der Speicher nicht wieder freigegeben wird und die Liste manchmal mehr Speicher benutzt, als sie tatsächlich benötigt (nämlich wenn die garbage-Liste Elemente enthält).

Die Wrapperklasse würde dann das gleiche bewirken können wie eine list mit einem Memorypool als Custom-Allocator, wenn man die garbage-Liste mit der Größe des Memorypools initialisieren würde. Dadurch kann man sich für eine list relativ einfach eine Variante bauen, die schneller läuft als die normale, ohne das aufwendig eine eigene Allocator-Klasse programmiert werden muss – dies gilt für alle anderen Container nicht!

iii. map

Maps werden gerne dazu verwendet, eine Menge von Objekten mit einer ID abzuspe-

ichern (die ID wird als Schlüssel genommen). Dass hat gegenüber der *set* den Vorteil, das Elemente auch nur über ihre ID erreicht werden können. Maps sind die Komplextesten Container der STL, doch in ihrer Anwendung relativ einfach – dank des []-Operators:

```
// Map mit int-Schluessel
std::map<int, double> con;

// Zuweisung
con[8] = 18.2;

// Zugriff Variante 1
double d = con[8];

// Zugriff Variante 2
double d2 = con.at(8);
```

Listing 9: *Zugriff auf map*

Hinter diesem Quelltext steckt mehr, als es zunächst scheint: Es sieht so aus, also würde mit *con[8] = 18.2* nur eine Zuweisung passieren, tatsächlich aber wird ein neues Element in die Map eingefügt, und dabei wird auch die Speicherverwaltung angefordert. Desweiteren werden hier zwei verschiedene Arten von Zugriffen gezeigt, die in diesem Fall auch komplett das gleiche tun. Sobald der Schlüssel jedoch nicht in der Map existiert, gibt es einen Unterschied: Variante 1 würde dann einfach 0.0 zurückgeben, oder was auch immer der Defaultwert für den Werttypen der Map ist. Variante 2 hingegen würde eine Exception werfen. Exceptions sind in ressourcenkritischen Systemen oft verpönt, da sie langsam sind und Echtzeitverhalten stören, daher sollte man wissen, wann sie auftreten können. Um zu prüfen, ob eine Map für einen gegebenen Schlüssel einen Wert enthält, sollte man mit der Methode *count()* die Anzahl der Werte für diesen Schlüssel nachfragen. Für normale Maps (keine Multimaps) ist *count()* entweder 0 oder 1. Anschließend kann man dann den []-Operator anwenden und sich sicher sein, dass der Wert dazu tatsächlich existiert.

Die Vorteile der Mengen und Maps gegenüber einem vector machen sich erst bei großen Mengen (ab etwa 1000 Elementen) und wenn oft auf die Elemente per Wert zugegriffen wird bezahlt. Im Zweifelsfall sollte für jeden Fall der Ressourcenverbrauch und die Geschwindigkeit beider Containertypen gemessen und verglichen werden. Als Beispiel wird hier ein Vergleich gezeigt, es sollten jedoch für eigene Implementationen individuelle Tests durchgeführt werden.

```
struct Element {
    int ID;
    char data[1024];
    Element(int i)
    : ID{i}
    {
    }
};

volatile char c;

// ......................

    int N = 1000;

    // Variante 1
    std::map<int, Element*> con1;
    for (int i = 0; i < N; ++i) {
        con1[i] = new Element(i);
    }

    // Variante 2
    std::vector<Element*> con2(N);
    for (int i = 0; i < N; ++i) {
        con2[i] = new Element(i);
    }

    // Zugriff Variante 1
    for (int i = 0; i < N; ++i) {
        Element* e = con1[i];
        c = e->data[17];
    }

    // Zugriff Variante 2
    for (int i = 0; i < N; ++i) {
        Element* e;
        for (int j = 0;
          j < con2.size(); ++j) {
            if (con2[j]->ID == i) {
                e = con2[j];
                break;
            }
        }
        c = e->data[17];
    }
```

Listing 10: *map vs. vector*

Getestet werden hier die Laufzeiten der Zugriffe über die ID einmal in einer map (Variante 1) und einmal in einem vector (Variante 2). Die

Variable *c* wird nur benutzt um sicherzugehen, dass der Compiler die Schleifen nicht wegoptimiert. Die Ergebnisse für verschiedene N werden in folgender Tabelle aufgezeigt:

Table 3: *Zugriffe über die ID: CPU-Zyklen*

N	Variante 1	Variante 2
10	3272	2092
100	42948	30104
1000	590228	2925954
10000	2857710	227100546

Der Speicherverbrauch beider Varianten sieht etwa wie folgt aus:

Table 4: *Zugriffe über die ID: Speicherverbrauch*

N	Variante 1	Variante 2
10	360	80
100	3600	800
1000	36000	8000
10000	360000	80000

Während der Speicherverbrauch beider Containertypen gleichförmig linear ansteigt, holt die map den vector ab einer Anzahl von etwa 1000 Elementen bezüglich Zugriffszeit ein, ist dann also zu bevorzugen.

IV. Fazit

STL-Container sind hinsichtlich Effizienz die Container der Wahl, wenn eine dynamische Anzahl von Objekten gespeichert werden soll und ein einfaches Array nicht mehr ausreicht. Der vector ist der beliebteste Container und bei einer kleinen Anzahl von Elementen auch meistens der ressourcensparenste. Aber auch die anderen Standardcontainer lassen sich effizient benutzen, wenn man weiß, an welchen Operationen Speicher alloziert wird und wie man das umgehen kann. Selbst in Echtzeitanwendungen können einige Containerklassen angewendet werden. Eigene Containerklassen zu entwerfen wäre nur in sehr seltenen Situationen sinnvoll, es ist einfacher, die gegebenen Container mit Adapterklassen anzupassen oder zu erweitern. Außerdem ist die Vielzahl der nützlichen Operationen auf Containern durch die Bibliothek Algorithm nicht zu verachten.

V. Referenzen

How to make C more real time friendly, Scot Salmon, National Instruments 2014, *http://www.embedded.com/design/programming-languages-and-tools/4429790/How-to-make-C–more-real-time-friendly*

C++ Container Cheat-Sheet, *http://homepages.e3.net.nz/ djm/cppcontainers.html*

STL-Container performance, John Ahlgren 2013, http://john-ahlgren.blogspot.de/2013/10/stl-container-performance.html

C++-Reference, http://www.cplusplus.com/reference/stl/

C++ Allocator concept, http://en.cppreference.com/w/cpp/concept/Allocator

push_back concern, http://stackoverflow.com/questions/20168051/why-push-back-is-slower-than-operator-for-an-previous-allocated-vector

VI. Figures

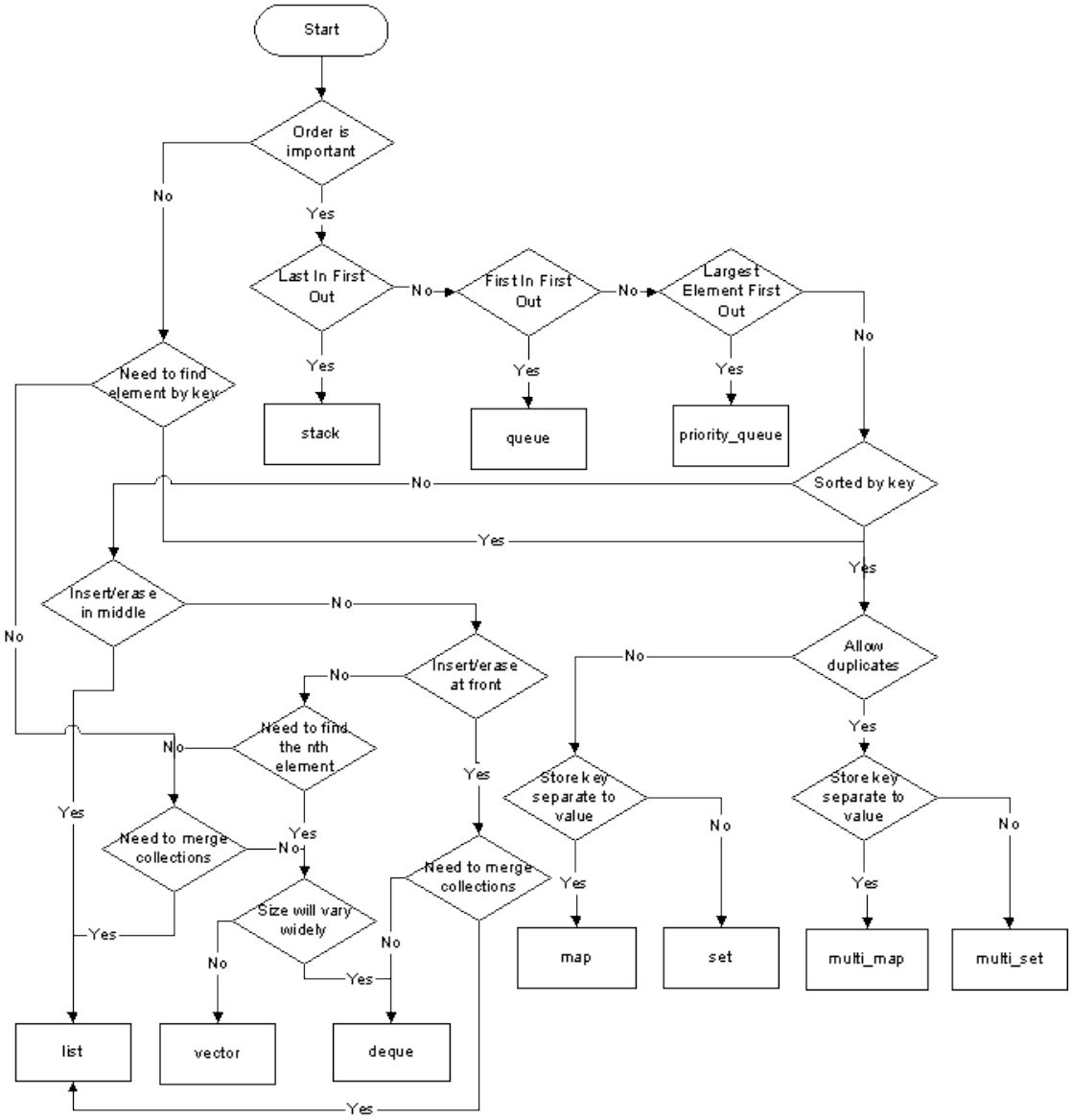

Figure 1: *Die Wahl eines STL-Containers. Quelle: C++ Containers Cheat Sheet*