Bauhaus-Universität Weimar · Fakultät Medien · Systeme der Virtuellen Realität

Zusammenfassung

C++-Guru Seminar

Hans-Friedrich Pabst

Sommersemester 2004

Inhaltsverzeichnis

Abbildungsverzeichnis

1 C++ Terms

[Breymann'03] [Nootz'01]

1.1 Klassen

Bei Klassen handelt es sich um benutzerdefinierte Datentypen, die die Struktur und das Verhalten aller Instanzen einer Klasse spezifizieren. Sie stellen somit eine Abstraktion dar. Eine Struktur struct ist eine Klasse mit ausschließlich öffentlichen Elementen.

Ein kleines Klassen-Beispiel

```
class A {
   public:
      int  get_val() const;
      void set_val(int);
   private:
      int  val;
};
```

1.2 Methoden(member functions)

Methoden sind Funktionen mit Zugriff auf Daten aller Objekte einer Klasse Sie sind genau wie Member-Variablen Elemente einer Klasse.

Ein kleines Memberfunction-Beispiel

```
class A {
public:
   bool me(A const& rhs)
      // this == rhs.this
      { return this == &rhs; }
};
A aa, ab;
aa.me(ab); // false
aa.me(aa); // true
```

1.3 Selbstreferenz

Jede Elementfunktion weiß, für welches Objekt sie aufgerufen wurde und kann sich damit explizit auf dieses beziehen. Hierfür steht das Schlüsselwort this zur Verfügung. this ist

ein Zeiger auf das Objekt selbst innerhalb einer Memberfunction. Es handelt sich damit um eine immer vorhandene Membervariable.

Bsp: Schutz vor Selbstzuweisung

```
A& A::operator=(const A& a) //Zuweisungsop.
{
   if (this!=&a)
   //Zuweisung
   return *this;
}
```

1.4 Objekte

Erzeugt man eine Instanz einer Klasse, erhält man eine Einheit aus Zustand und Verhalten, ein sogenanntes Objekt. Eine solche Initialisierung wird über die Konstruktoren einer Klasse vorgenommen.

Bsp: Erzeugung Objekt A1 der Klasse A

```
A A1(20,15);
```

1.5 Konstruktoren

Konstruktoren sind Member-Funktionen, die den Zweck haben Objekte zu initialisieren. Wann immer ein Objekt einer Klasse erzeugt wird, wird automatisch ein Konstruktor aufgerufen. Sie kümmern sich um die Anforderungen von Ressourcen. Zu beachten ist, dass der Methodenname eines Konstruktors der zugehörige Klassenname selbst ist, und dass er keinen Rückgabewert hat.

Bsp: Konstruktoren

```
class A {
        public:
           A(int, int);
           A();
           A(const A&);
           ~A();
        private:
           int var1;
           int var2;
        };
```

Wie aus dem Codebeispiel ersichtlich, gibt es verschiedene Konstruktoren und zudem noch Copykonstruktoren und Destruktoren.

1.5.1 Defaultconstructor

Der Defaultkonstruktor erhält keine Argumente, die Instanzvariablen werden mit Defaultwerten initialisiert

Bsp: Defaultconstructor

```
T::T()
    : pi(M_PI), udt(), pval(0)   //Initalisierungsliste
    {}
```

1.5.2 User-Konstruktor

Mit dem Userconstruktor hat der User die Möglichkeit Instanzvariablen mit Werten zu initialisieren.

Bsp: User-Konstruktor

```
T::T(int x, int y = 1)
    :variable1(x), variable2(y)
    {//
    }
```

Durch Überladen des sogenannten User-C'tors, ist es möglich Objekte auf verschiedene Arten zu konstruieren.

1.5.3 Copy-Konstruktor

Um ein Objekt mit einer Kopie eines anderen Objektes der Klasse zu initialisieren, benötigt man einen Copy-Konstruktor.

Bsp: Initialisierung mit Kopie

```
T a;
T b = a;
```

Mit dem Copy-Konstruktor legt man fest, wie die Elemente des Originalobjekts kopiert werden.

Bsp: Copy-Konstruktor

```
T::T(const T& TOriginal){
   variable1=TOriginal.variable1;
   variable2=TOriginal.variable2;
   }
```

1.5.4 Destruktor

Die vom Konstruktor angeforderten Resourcen müssen bei der Zerstörung des Objekts wieder freigeben werden. Hierfür ist der Destruktor verantwortlich, der impliziert aufgerufen wird, wenn das Objekt zerstört wird. Um das Komplementärverhältnis zum Konstruktor auszudrücken, ist der Destruktor durch das Komplementzeichen - gekennzeichnet.

Bsp: Destruktor

```
class B{
public:
   B(){
      p=new A
   }//Konstruktor
   ~B(){
      delete p;
   }//Destruktor
private: A* p;
};

delete this;// -> this->~T(); free(this);
```

1.6 Deklaration

Bevor ein Bezeichner im Programm benutzt werden kann, muss sein Typ spezifiert werden. Eine Deklaration besteht aus einem optionalen Spezifizierer, einem Basistyp, einem Deklarator und einem optionalen Initialisierer. Mit der Deklaration ist die Größe des Bezeichners festgelegt, welche man durch *sizeof()* feststellen. Oftmals handelt es sich bei einer Deklaration zugleich um eine Definition.

Bsp: Deklaration

```
struct user {};
double sqrt(double);
struct user;     // fwd. decl
extern int a;    //nur Deklaration
```

```
char ch;
int i = 1;       //auch Definition
```

1.7 Definition

Wird mehr als nur der Name eingeführt, also auch Speicher angelegt, handelt es sich um eine Definition. Mit der Definition ist eine Adresse des Bezeichners vorhanden, die man durch *adressof()* bestimmen kann.

Bsp: Definition

```
struct struktur{int a, d;};
int f(int x){x*x};
typedef void (T::* PTMF) ();
//Pointer to memberfunction

PTMF pf = 0;

pf=&T::method1;
tinst->pf();
```

1.8 Scope

Durch eine Deklaration wird ein Name in einem Scope eingeführt. Man unterscheidet zwischen temporären Namen, die nur innerhalb einer Anweisung existieren, lokalen Namen und globalen Namen. Die lokalen Namen haben ihren Güligkeitsbereich von der Stelle ihrer Deklaration an bis zum Ende des Blocks, in welchem ihre Deklaration auftrat. Ein Block ist ein durch die geschweiften Klammern {} begrenzter Teil des Codes. Wird ein Name außerhalb einer Funktion, einer Klasse oder eines Namenbereichs deklariert, handelt es sich um einen globalen Namen. Dieser globale Name hat seinen Gültigkeitsbereich von der Stelle seiner Deklaration bis zum Ende der Datei in der er deklariert wurde. Eine lokale Variable kann eine globale Variable für den Gültigkeitsbereich der lokalen Variable verdecken.

1.9 One-Definition Rule

Eine bestimmte Klasse, ein Template usw. muss genau einmal im Programm definiert sein. Da aber z. B. die Definition einer Klasse durch include in verschiedene Quelldateien eines Programms eingefügt werden muss, regelt die One-Definiton Rule, was unter einmaliger Definition zu verstehen ist. Sie besagt, dass zwei Definitionen einer Klasse,

eines Templates oder einer inline-Funktion dann und nur dann als eine einzige Defintion akzeptiert werden, wenn sie:

1. in verschiedenen Übersetzungeinheiten auftauchen
2. Token fuer Token identisch sind und
3. die Bedeutung dieser Token in beiden Übersetzungs-
   einheiten identisch ist.

Damit ist das Einfügen einer Klassendefinition in verschiedenen Übersetzungseinheiten aus einer gemeinsamen Quelldatei erlaubt.

1.10 Erzeugung eines Objekts

Ein Objekt kann statisch oder dynamisch erzeugt werden. Statische Objekte werden im Stack gespeichert und dynamische im Heap. Statische Objekte werden automatisch entfernt, verlässt man ihren Gültigkeitsbereich.

Bsp: statisch anglegtes Objekt

```
A A1(2, 3);  // A1.A(2,3);
```

Wird ein Objekt dynamisch angelegt, wird der Speicherplatz explizit mit new angefordert und kann, wenn er nicht mehr benötigt wird, mit delete wieder freigegeben werden.

Bsp: dynamisch anglegtes Objekt

```
A* A2 = new A(2,3);
        // - void* p = malloc(sizeof(A));
      // - new (p) A(2,3);//placement
```

1.11 Zeiger und Referenzen

Ein Zeiger auf ein Objekt enthält die Adresse dieses Objekts. Für Typ T ist T* der Typ Zeiger auf T.

Bsp: Zeiger

```
char c = 'a';
char* p = &c;   // p enthaelt Adresse von c;
                //address-of
char c2 = *p;   // c2 ist 'a' z- dereference
```

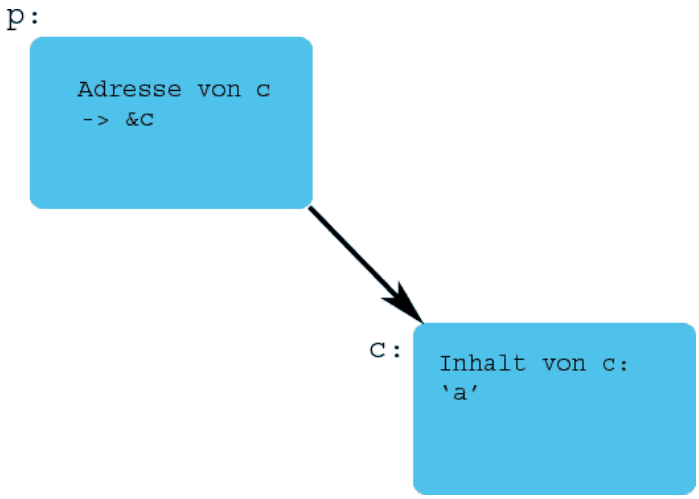


Abbildung 1: Zeiger

Im Gegensatz zu Zeigern handelt es sich bei Referencen um alternative Namen. Somit sind Original und Referenz ununterscheidbar. Wichtig ist das eine Referenz immer initialisiert ist, da sie ja nur ein Name für ein anderes Objekt ist.

Bsp: Referenzen

```
int  i = 1;
int& j = i; // i und j beziehen sich auf selbe int
int& k; // error --> keine Initialisierung
j++;    // i ist jetzt 2
```

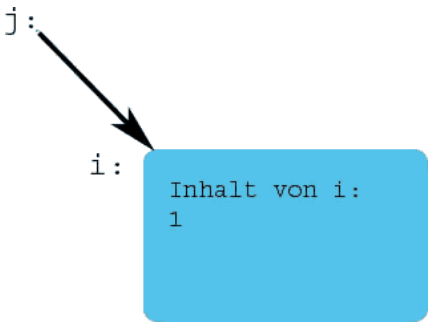


Abbildung 2: Referenz

1.12 Deklaration und Definition von Funktionen

1.12.1 Argumentübergabe

Bei Aufruf einer Funktion wird Speicher für die Argumente bereitgestellt und diese werden mit den übergebenen Paramtern initialisiert.

Bsp: Argumentübergabe

```
void f(int a, int& b, int* c ) {
   ++a;
   ++b;
   ++c;
}

int main (int argc, char **argv)
{
   int x=1;
   int y=1;
   int z =1;
   f(x, y, &z); //x=1, y=2, z=2
   //x-->by value
   //y --> by reference
   //z--> by pointer
}
```

Wird das Argument by value übergeben, wird der Wert in eine lokale Variable kopiert und in der Funktion weiterverarbeitet. Bei einem Call by reference wird unter einem anderen Namen direkt auf dem Objekt, das referenziert wird, gearbeitet. Erfolgt eine Übergabe durch Zeiger, wird direkt auf der übergebenen Adresse gearbeitet. Referenzargumente machen Programme schwer lesbar. Dennoch kann es effizienter sein, große Objekte by reference zu übergeben. Diese Argumente sollten dann möglichst als const deklariert sein.

1.12.2 Überladene Funktionsnamen

Normalerweise tragen verschiedene Funktionen auch unterschiedliche Namen, dennoch kann es sinnvoll sein Funktionen, die dieselbe Aufgabe auf verschiedenen Typen ausführen, denselben Namen zu geben. Dies nennt man Überladen. Man kann sich leicht vorstellen, dass es intuitiver ist, z. B. für die Addition von Zahlen aus verschiedenen Zahlbereichen denselben Operator verwenden zu können. Nicht zu verwechseln ist das Überladen mit dem Überschreiben von Funktionsnamen, was nur im Kontext einer Vererbung geschehen kann.

Bsp: Überladene Funktionsnamen

```
double max(double x, double y);
int max (int x, int y);
```

1.13 Vererbung: Abstract Interface

Die Beziehung zu einer Oberklasse um ihre Merkmale und ihr Verhalten zu übernehmen wird Vererbung genannt.

Bsp: Vererbung

```
#define abstract =0
class Shape {
public:
   virtual double area() const abstract;
   // rein virtuell
private:
   string name; //private members
};
class Rectangle: public Shape {
public:
   /* virtual */ double area() const{
         return width*height;}
private:
   //...
};
class Square: public Rectangle {
public:
   /* virtual */ double area() const{
         return width*width; }
private:
    //private members...
```

Die Klasse Rectangle ist von der Klasse Shape abgeleitet, somit ist Shape die Basisklasse von Rectangle. Die Klasse Rectangle hat dadurch, neben ihren eigenen Membervariablen und -funktionen, alle Elemente der Klasse Shape. Wird ein Objekt über Zeiger oder Referenzen manipuliert, kann es wie ein Objekt seiner Basisklasse benutzt werden.

Bsp: Vererbung

```
Rectangle* myRect = new Rectangle();
Square* mySquare = new Square();
myRect = mySquare;
```

Ein Square kann überall benutzt werden, wo auch ein Rectangle benutzt werden könnte.

1.13.1 Überschriebene Funktionsnamen

Wie im Abschnitt über Überladene Funktionen erwähnt wurde, kann das Überschreiben von Funktionsnamen nur im Kontext einer Vererbung geschehen. Die Typsignaturen und die Methodennamen überschriebener Funktionen stimmen überein. Damit eine Funktion einer abgeleiteten Klasse überschrieben werden kann, muss dies explizit mit virtual spezifiert werden.

Bsp: Überschriebene Funktion

```
myRect->area(); // area() von Square
```

Die Klasse Shape ist eine abstrakte Klasse, da sie Methoden enthält, die rein virtuell sind. Eine abstrakte Klasse dient nur als Interface. Das bedeutet, dass keine Instanzen dieser Klasse erzeugt werden können. Compiler und Linker garantieren für die korrekte Zuordnung zwischen Objekten und den auf sie angewandten Funktionen. Diese somit erst zur Laufzeit erfolgende Ermittlung, der zu einem Objekt passenden Realisierung einer Operation, wird Polymorphismus gennannt.

1.14 Zugriffskontrolle

Die Zugriffskontrolle realisiert die klassische Objektorientierung, indem sie für die Trennung von Interface und Implentierung sorgt. Ein Element, eine Elementfunktion oder eine Ableitung kann public, protected oder private sein. Durch diese Schlüsselworte wird definiert, wer das Recht hat auf diese Elemente zuzugreifen.

Bsp: Access

```
class Rectangle: public Shape {
                 //public Ableitung von Shape
public:
    //Funktion area() ist public
   /* virtual */ double area() const{
         return width*height;}
private:
   double heigth;
   double width;    //private Elemente
};
```

- Ist ein Element *private*, können nur Elementfunktionen und *friends* der Klasse, in der es deklariert wurde darauf zugreifen.

- Handelt es sich um ein *protected* Element, kann es nur von Elementfunktionen und friends der Klasse benutzt werden. Und zusätzlich von Elementfunktionen und *friends* von Klassen, die von der Klasse, in der das Element deklariert wurde, abgeleitet wurden.

- Ist ein Element *public* hat man darauf öffentlichen Zugriff.

1.14.1 Zugriff auf Basisklassen

Wird eine Basisklasse bei der Vererbung private, protected oder public deklariert, ergeben sich für die abgeleitete Klasse die in Abbildung 3 dargestellten Zugriffsrechte auf die Elemente der Basisklasse.

	Attribut in der Basisklasse		
Art der Ableitung	public	private	protected
public	public	Kein Zugriff	protected
private	private	Kein Zugriff	private
protected	protected	Kein Zugriff	protected

Abbildung 3: Zugriffsrechte

Lässt man die Zugriffspezifikation weg, wird für eine Klasse *private* als Standard genommen und für eine Struktur *public*.

Bsp: Access

```
class C: B{/*...*/}; // B ist private Basisklasse
struct S: B{/*...*/}; // B ist öffentliche Basisklasse
```

1.15 Namemangling

C++ erlaubt es denselben Funktionsnamen für Funktionen mit unterschiedlichen Parametern oder für verschiedene Klassen zu verwenden, damit der Linker für den korrekten Aufruf einer Funktion garantieren kann, codiert der Compiler im Namen von Funktionen die Typen der Argumente. Das wird *Name-Mangling* genannt. Für das *Name-Mangling* gibt es keinen Standard, verschiedene Compiler und verschiedene Versionen von Compilern *name-mangeln* unterschiedlich.

Bsp: Name-Mangling

```
Animal::GetName() const
kann in
_ZNK6Animal7GetNameEv
gemangelt werden
```

1.16 POD (plain old datatype)

Der Standard unterteilt alle Typen in POD-Typen und non-POD-Typen. Im Allgemeinen sind POD-Typen Typen, die kompatibel zu C sind. Also handelt es sich bei allen Datenstrukturen, die keinen Konstruktor, Destruktor, virtuelle Funktionen oder ähnliches besitzen um POD-Typen.

1.17 Templates

Wie schon im Abschnitt über überladene Funktionen festgestellt wurde, gibt es Abläufe, die für verschiedene Datentypen gleich sind. Ein Beispiel hierfür wäre die Maximum-Funktion. Hierfür können Templates verwendet werden. C++ bietet mit ihnen die Möglichkeit, einen Algorithmus zu beschreiben, ohne dass man sich auf bestimmte Datentypen festlegen muss. Man spricht hierbei von generischer Programmierung, da Code generiert wird. Man unterscheidet zwischen Template-Funktionen, bei denen die Typen der Parametern nicht weiter spezifiert sind und Template-Klassen. Templateklassen beschreiben Datenstrukturen, die beliebige Typen aufnehmen können. Man spricht auch von parametrisierten Datentypen.

Bsp: Klassentemplate

```
template <class T>
   class A{
   private:
        T* x;
   }
```

Bei den Funktionstemplates handelt es sich um Funktionen für parametrisierte Datentypen.

Bsp: Klassentemplate

```
template <class T>
   T Max (T x, T y){
        return x>y? x:y;
    }
```

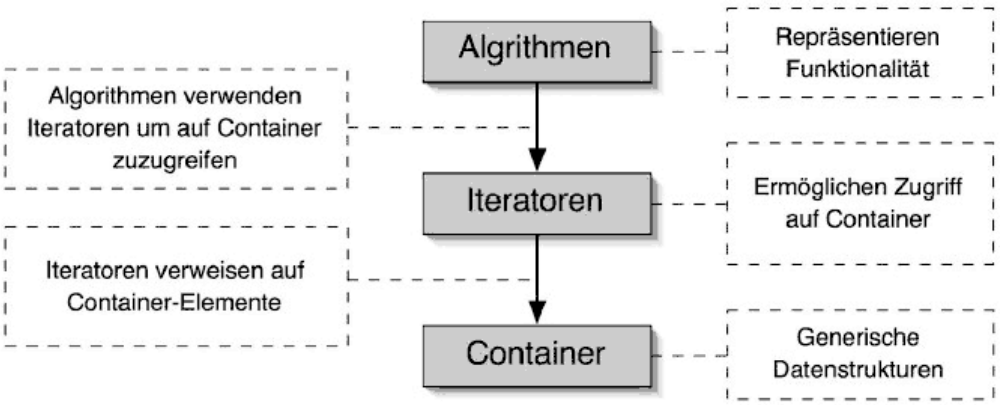


Abbildung 4: STL-Klassifikation

2 STL Terms

2.1 Einführung

Dieses Dokument basisiert auf [STL'00].

STL ist die Abkürzung für „Standard Template Library“ und ist eine generische C++-Bibliothek, welche u. a. Container-Klassen, Iteratoren und Algorithmen beinhaltet. Das Adjektiv „generisch“ soll hier zum Ausdruck bringen, dass die Komponeten der STL sehr stark parametrisiert sind und nahezu jede Komponente dieser Bibliothek ein Template ist.

Hinweis: Es wurden nicht alle Begriffen ins deutsche Übersetzt, da dies nicht immer Sinn macht und die englischen Fachausdrücke im entsprechenden Umfeld sehr gebräuchlich sind.

2.2 Inhalte der STL

Ein zentrales Thema der STL ist die Zusammenarbeit von Containerklassen, Iteratoren und Algorithmen – sie arbeiten Hand-in-Hand miteinandern, wie in Abbildung 4 dargestellt.

Den Entwicklern der STL geht es hierbei im Wesentlichen auch um Konzepte, die benötigt werden, um die Templates für die STL bereitzustellen. Die Konzepte klassifizieren und beschreiben Eigenschaften und grenzen Aufgabenbereiche der einzelnen Komponenten klar ab. Wenn der Programmierer, der die STL nutzt, die Konzepte kennt

und verstanden hat, ist es für ihn einfach, die Komponente mit der gewünschten Menge an Möglichkeiten zu finden, ohne dabei einen unnötigen Overhead „mitzuschleppen“.

Neben Containern, Iteratoren und Algorithmen gibt es in der STL jedoch auch weitere Komponententypen, die sich im Wesentlichen in drei zusätzliche Kategorien eingliedern lassen – Utilities, Allocators und Functors. [Anmerkung: Auf diese drei Kategorien wird in diesem Dokument nicht weiter eingegangen werden!]

Man muss an dieser Stelle deutlich darauf hinweisen, dass nicht jede Komponente der STL zu dem C++-Standard gehört – einige Teile sind SGI-spezifische Erweiterungen.

2.3 Container

Die Container der STL lassen sich in drei grundlegende Konzepte einteilen:

- Generelle Konzepte
- Sequenzen
- Assoziative Container

Diese Konzepte werden im Folgenden genauer definiert und näher erläutert.

2.3.1 Generelle Konzepte

Ein Container ist ein Objekt, welches andere Objekte („Elemente des Container“) aufbewahrt und Methoden besitzt um auf diese zuzugreifen. Im Besonderen gilt, dass jedes Objekt, welches zu dem Model eines Container gehört, einen Iterator-Typen besitzt, mit dem man durch den Container iterieren kann.

Für einen allgemeinen Container gibt es keine festgelegt Reihenfolge für die Aufbewahrung der beinhalteten Elemente. Des Weiteren gibt es keine Festlegung, dass nicht mehr als ein Iterator zu einem Zeitpunkt aktive sein darf.

Die Lebenszeit der aufbewahrten Objekte in einem Container werden durch den Container selbst bestimmt. Eine „Herausgabe“ dieser Lebenszeitkontrolle aus dem Container ist dabei nicht möglich. [Anmerkung: Wenn ein Container Pointer auf Elemente enthält, so bestimmt der Container die Lebenszeit der Pointer, aber nicht die Lebenszeit der Objekte, auf die der Pointer zeigt!]

Forward Container

Die Elementen in einem Forward Container befinden sich in einer genau definierten Reihenfolge. Diese Reihefolge ändert sich nicht spontan von einer Iteration durch den Containers zur nächsten. Die definierte Reihenfolge ist ein wichtiger Grundsatz für einen Elementenvergleich (Stichwort: Gleichheitsoperator) und die lexikographische Reihenfolge (Stichwort: kleiner-als-Operator) in einem Container.

Iteratoren auf einem Forward Container entsprechen den Anforderungen des Forward Iterator, infolgedessen unterstützen Forward Container Multipass Algorithmen und erlauben mehrere aktive Interatoren zur gleichen Zeit.

Der Forward Container ist einer Spezialisierung des allgemeinen Containers. Er beinhaltet u. a. den Gleichheitsoperator und den kleiner-als-Operator.

Reversible Container

Der Reversible Container ist eine Spezialisierung des Forward Container. Er besitzt anstelle eines Forward Iterator einen Bidirectional Iterator, welcher zusätzlich eine Rückwärtsiteration durch den Container erlaubt.

Random Access Container

Der Random Access Container ist eine Spezialisierung des Forward Container. Er besitzt anstelle eines Bidirectional Iterator einen Radom Access Iterator, welcher zusätzlich eine freien Zugriff auf beliebige Elemente im Container in einer konstanten Zeit erlaubt.

2.3.2 Sequences

Eine Sequence ist ein Container mit variabler Größe, bei welcher die Elemente in einer strikten linaeren Reihenfolge angeordnet sind. Sequences unterstützen das Einfügen und Löschen von Elementen.

Front Insertion Container

Der Front Insertion Container ist eine Spezialisierung einer Sequence, bei welchem das Einfügen am Anfang eines Containers und der Zugriff eines Elementes am Anfang eines Containers in einer konstanten Zeit erfolgt. Für diese Operationen stellt der Front Insertion Container spezielle Methoden zur Verfügung.

Back Insertion Container

Der Back Insertion Container ist eine Spezialisierung einer Sequence, bei welchem das Einfügen eines Elementen am Ende eines Containers und der Zugriff eines Elementes am Ende eines Containers in einer konstanten Zeit erfolgt. Für diese Operationen stellt der Back Insertion Container spezielle Methoden zur Verfügung.

Beispiele

Folgend einige Beispiele für Sequences (nicht vollständig):

- `vector<T, Alloc>`
- `deque<T, Alloc>`
- `list<T, Alloc>` – SGI-Erweiterung
- `slist<T, Alloc>` – SGI-Erweiterung

2.3.3 Associative Containers

Ein Associative Container ist ein Container mit variabler Größe, bei dem der Zugriff auf die Elemente anhand von „Keys“ erfolgt. Associative Container unterstützen das Einfügen und Löschen von Elementen, mit dem Unterschied im Vergleich zu einer Sequence, dass Associative Container keinen Mechanismus zum Einfügen von Elementen an einer speziellen Position zur Verfügung stellen.

Wie bei allen Containerkonzepten sind die Elemente vom Typ `value_type`. Zusätzlich haben Associative Container jedoch zu jedem Element noch einen Key vom Typ `key_type`. Es gibt Associative Container bei denen ist der `value_type` gleichzeitig der `key_type` (Simple Associative Container) und bei anderen ist der Key ein spezieller Teil des Werts.

Da die Elemente anhand der Keys in dem Container aufbewahrt werden, ist es notwendig, dass die Verbindung zwischen Key und Element unveränderlich ist. Dies wiederum hat zur Folge, dass der Wertetyp des Associative Container nicht zuweisebar ist. Die Konsequenz aus dieser Nichtzuweisbarkeit ist, dass Associative Container keine veränderlichen Iteratoren haben!

Für einige Associative Containern (Unique Associative Containers) ist garantiert, dass nicht mehr als ein Element des Containers den selben Key hat. Bei anderen Associative

Containern (Multiple Associative Containers) wiederum ist es erlaubt, dass mehr als ein Element den selben Key hat.

Simple Associative Container

Ein Simple Associative Container ist ein Associative Container, bei dem Elemente gleichzeitig ihre eigenen Keys sind.

Pair Associative Container

Ein Pair Associative Container ist ein Associative Container, bei dem ein Key mit einem Element assoziiert wird. Die Wertetype eines Pair Associative Container sind `pair<const key_type, data_type>`! Da der Key nicht verändert werden darf, kann man (wie schon erwähnt) keinen veränderlichen Iterator auf Associative Container anwenden.

Sorted Associative Container

Ein Sorted Associative Container ist ein Associative Container, bei dem eine Ordnungsrelation für die Keys vorhanden ist. Dies bedeutet, dass zwei Keys entweder gleich sind oder einer ist „kleiner-als“ bezüglich der entsprechend definierten Ordnungsrelation.

Bei einem Sorted Associative Container ist die Komplexität der Operationen niemals schlechter als Logarithmus. Des Weiteren sind die Elemente immer nach aufsteigenden Keys sortiert.

Hash Associative Container

Ein Hash Associative Container ist ein Associative Container, dessen Implementierung eine Hash-Tabelle ist. In einem Hash Associative Container gibt es, in der Regel, keine sinnvolle Ordnung der Elemente – die Elemente und deren Keys sind nicht sortiert. Die „worst-case“-Komplexität der Operationen auf einen Hash Associative Container ist linear zu der Größe des Containers, im „average-case“ jedoch konstant. Dies bedeuted, dass in Fällen, bei denen die Ordnung der Elemente keine Rolle spielt, sondern es nur um die Aufbewahrung dieser geht, ein Hash Associative Container viel schneller ist als ein Sorted Associative Container.

Unique Associative Container

Unique Associative Container ist ein Associative Container mit der Eigenschaft, dass jeder Key einmalig ist. Keine zwei Elemente in einem Unique Associative Container

haben den selben Key.

Multiple Associative Container

Ein Multiple Associative Container ist ein Associative Container, bei dem mehr als ein Element den selben Key haben darf.

Kombinationen

Aus den Definitionen und damit verbundenen Eigenschaften von Unique- und Multiple Associative Container und Sorted- und Hash Associative Container lassen sich entsprechenden Kombinationen bilden.

Diese Kombinationen sind:

- Unique Sorted Associative Container
- Unique Hash Associative Container
- Multiple Sorted Associative Container
- Multiple Hash Associative Container

Beispiele

Folgend einige Beispiele für Associative Containers (nicht vollständig):

- `set<Key, Compare, Alloc>`
- `map<Key, Data, Compare, Alloc>`
- `multiset<Key, Compare, Alloc>`
- `multimap<Key, Data, Compare, Alloc>`
- `hash_set<Key, HashFcn, EqualKey, Alloc>` – SGI-Erweiterung
- `multi_map<Key, Data, HashFcn, EqualKey, Alloc>` – SGI-Erweiterung

2.3.4 Container Adaptors

Container Adaptors unterstützen eine beschränkte Teilmenge von Containerfunktionalitäten. Container Adaptors arbeiten meist auf Standardcontainer und stellen diesen nur ein neues Interface zur Verfügung. Des Weiteren ist es nicht möglich über einen Adaptor mit Hilfe eines Iterators zu iterieren.

Vertreter Container Adaptors sind:

- `stack`, der eine LIFO-Datenstruktur besitzt und meist auf Basis einer `deque` beruht.
- `queue`, der eine FIFO-Datenstruktur besitzt und meist auf Basis einer `deque` beruht.
- `priority_queue`, man kann Elemente einfügen sowie auf das oberste Element zugreifen (auslesen und löschen). Es ist garantiert, dass das oberste Element das größte, entsprechend der Ordnungsrelation ist. Beruht meist auf Basis eines `vector`.

2.4 Iteratoren

Iteratoren sind eine Verallgemeinerung von Pointern – es sind Objekte, die auf andere Objekte zeigen. Wie der Name suggeriert, dienen Iteratoren dazu, durch eine Menge von Elementen zu iterieren (bewegen). Wenn ein Iterator auf ein Element der Menge zeigt, dann ist durch Inkrementierung des Iterators möglich, ihn auf das nächste Element zeigen zu lassen.

Iteratoren spielen in der generischen Programmierung eine große Rolle, da sie die Schnittstelle zwischen Containern und Algorithmen darstellen (siehe Abbildung 4 auf Seite 25).

Iteratoren sind nicht ein Konzept, vielmehr gibt es sechs Konzepte, die eine Hierarchie bilden. Es gibt fünf sehr strikte Konzepte (Input Iterator, Output Iterator, Forward Iterator, Bidirectional Iterator und Random Access Iterator) mit einer sehr beschränkten Menge von Operationen, welche auch durch die Algorithmen benutzt werden, und ein weiteres Konzept (Trivial Iterator), welches als Grundlage für allgemeine Iteratoren dient.

Hinweis: Ein Container kann leer sein, dann ist der erste gleich dem letzten Iterator des Containers. Des Weiteren gilt, wenn ein Container n Elemente enthält, dann repräsentiert die Notation [first, last) n+1 Positionen.

2.4.1 Trivial Iterator

Der Trivial Iterator ist ein Iterator, welcher ein Element, auf das er sich bezieht, dereferenzieren kann. Arithmetische Operationen, wie z. B. Inkrementierung oder Vergleich, werden nicht garantiert unterstützt.

2.4.2 Input Iterator

Der Input Iterator ist ein Iterator, welcher ein Element, auf das er sich bezieht, dereferenzieren kann. Des Weiteren ist eine Inkrementierung des Input Iterators möglich, um das nächste Element des Containers zu erhalten. Der Input Iterator ist nicht unbedingt ein veränderlicher Iteraror.

2.4.3 Output Iterator

Der Output Iterator ist ein Iterator zum Ablegen von Werten in Containern, aber nicht zwangsläufig zum Auslesen dieser. Der Output Iterator in dieser Hinsicht das Gegenteil des Input Iterarors. Eine Inkrementierung des Output Iterators ist möglich, um das nächste Element des Containers zu erhalten.

2.4.4 Forward Iterator

Der Forward Iterator kann in Multipass Algorithmen eingesetzt werden und kann Element eines Containers (nur) inkrementell auslesen. Der Forward Iterator kann ein veränderlicher, aber auch ein unveränderlicher Iterator sein. Der Forward Iterator wird z. B. zum Traversieren einer „einfach verketteten Liste“ benutzt.

2.4.5 Bidirectional Iterator

Der Bidirectional Iterator ist eine Erweiterung des Forward Iterator – er kann nicht nur inkrementell, sondern auch dekrementell durch den Container iterieren. Der Bidirectional Iterator wird z. B. zum Traversieren einer „doppelt verketteten Liste“ benutzt.

2.4.6 Random Access Iterator

Der Random Access Iterator ist eine Erweiterung des Bidirectional Iterator – er kann inkrementellen und dekrementell über den Container iterieren (wie der Bidirectional Iterator) und biete Methoden um sich in konstanter Zeit in willkürlichen Schritten vorwärts oder rückwärts zu bewegen. Ein Random Access Iterator bietet darüber hinaus alle notwendigen Operationen der C-Pointer-Arithmetik.

2.4.7 Iterator Tags

Iteratoren sind verknüpft mit `distance_type`, `value_type` und `iterator_category`, welche entsprechende Informationen über den Iterator beinhalten, auf die man mit Hilfe von „Iterator Tag“-Funktionen zugreifen kann. Diese Informationen sind z. B. wichtig für Algorithmen, die entsprechend der einzelnen Werte verschiedene Aktionen ausführen können.

`distance_type`, `value_type` und `iterator_category` sind alte „Iterator Tag“-Funktionen, welche durch `iterator_traits` im kommenden C++-Standard abgelöst werden sollen.

2.5 Algorithmen

Die STL beinhaltet eine Große Sammlung von Algorithmen zum Manipulieren der Elemente in Containern. Ein großer Vorteil ist, dass es sich bei den Algorithmen um globale Templates-Funktionen und nicht um Member-Funktionen handelt. Ein weiterer wesentlicher Vorteil ist, dass die Algorithmen in der STL sehr stark optimiert und sehr stabil sind und man sie deshalb bedenkenlos nutzen kann. (Die Algorithmen des STL sind so generisch gehalten, dass sie sich auch auf herkömmliche C-Strukturen wie `array` anwenden lassen!)

Die Vielzahl der Algorithmen lassen sich in vier große Gebiete gliedern:

- Nicht-verändernde Algorithmen, z. B. `for_each`, `find`, `count`
- Verändernde Algorithmen, z. B. `copy`, `swap`, `fill`
- Sortierende Algorithmen, z. B. `sort`, `merge`
- Generelle numerische Algorithmen z. B. `iota`, `accumulate`, `partial_sum`

3 Objektlebenszeit und temporäre Objekte

[Papurt'95] [Lippman'96]

Jedes Objekt in einem Programm durchläuft einen (*Lebenszklus*). Das bedeutet ein Objekt wird erzeugt, es existiert und schließlich wird es zerstört. Ein Objekt lebt oder ist lebendig wenn es vollständig erzeugt ist und noch vor seiner Zerstörung steht. Erst ein lebendiges Objekt wird vom Programm gesehen und kann benutzt werden. Wie lange ein Objekt nun lebt, also wie lange seine Lebenszeit ist, hängt vom verwendeten Objekterzeugungsmechanismus ab.

So unterscheidet man in C++ Objekte mit

- automatischer Fortdauer,
- statischer Fortdauer,
- dynamischer/manueller Fortdauer,
- temporäre Objekte.

Die Entscheidung darüber mit wieviel Lebenszeit man ein Objekt ausstattet hängt vom Kontext ab. Generell lässt sich sagen, dass die Objektlebenszeit der sinnvollen/nützlichen Lebenszeit der Objektinformation entsprechen sollte. Im Laufe dieses Kapitels sollen nun die Eigenheiten der oben unterschiedenen Objektarten dargestellt werden.

In den folgenden Beispielen wird oft die Klasse (`Trace`) verwendet. Ein Objekt dieser Klasse gibt seine Erzeugung und Zerstörung durch Ausgabe eines strings auf die Standardausgabe bekannt.

```
class Trace
{
   int v;

public:

     Trace(int k = -1) : v (k)
     {
        cout<<"Start Trace:"<<v<<"\n";
     }

     ~Trace()
     {
        cout<<"End Trace"<<v<<"\n";
     }
```

```
};
```

3.1 Objekte mit automatischer Fortdauer

Ein Objekt mit automatischer Fortdauer wird immer innerhalb einer Klammerstruktur (scope) deklariert. Es ist fortan an diesen scope gebunden. Wird das Ende des Scopes erreicht (}) wird auch das Objekt zerstört. Die Verknüpfung von Objektlebenszeit und scope entbindet den Programmierer von der Aufgabe die Objektzerstörung explizit zu erledigen; sie kann nicht vergessen werden.

```
// Funktion f.
// Erzeugt Trace - Objekt, dass am Fkt.-Ende zerstört wird.
void f()
{
      Trace t(2);
      cout<<"Inside f()"<<"\n";
}

int main()
{
      Trace u(1);
      f();
      return 0;
}
```

output des obigen Programmbeispiels:

Start Trace: 1 Start Trace: 2 Inside f() End Trace: 2 End Trace: 1

Soll ein Objekt mit automatischer Fotdauer über seine scope-Grenze hinweg benutzt werden, muss ein neues Objekt mit längerer Fortdauer erstellt werden und mit den Informationen des „dahingehenden“ Objekts initialisiert werden.

```
Teuer value()
{
      // Objekt c wird teuer konstruiert
      // und mit der schließenden Klammer (zu früh) wieder zerstört.
      Teuer c;
      return c;
}

int main()
{
      //Objekt d übernimmt Zustand von c, lebt über c hinaus.
      Teuer d = value();
```

```
}
```

Handelt es sich wie im Beispiel angedeutet um ein Objekt dessen Konstruktion „teuer“ ist möchte man das mehrmalige Erzeugen vermeiden. Im Verlauf dieses Kapitels wird gezeigt wie man mit dynamischen Objekten das gewünschte Verhalten, das Weiterleben von c (siehe code), erreichen kann.

3.2 Objekte mit statischer/globaler Fortdauer

Die Deklaration statischer Objekte kann in der translation unit (.cpp), dem file- oder Klassenscope (static) erfolgen. Hier deklarierte Objekte werden in der Regel vor dem Programmstart, spätestens aber vor der ersten Benutzung erzeugt. Sie stehen dann während der gesamten Laufzeit zur Verfügung und werden erst nach der letzten expliziten Anweisung im Programm automatisch zerstört. Es ist zu beachten, dass die Reihenfolge der Zerstörung statischer Objekte untereinander nicht determiniert ist.

```
//Objekte mit statischer Fortdauer
Trace z(0);
Trace x(1);

void f()
{
   Trace t(3);
   cout<<"Inside f()"<<"\n";
}

int main()
{
   Trace u(2);
   f();
   return 0;
}
```

output des obigen Programmbeispiels:

Start Trace: 0 Start Trace: 1 Start Trace: 2 Start Trace: 3 Inside f() End Trace: 3 End Trace: 2 //Die Zerstörungsreihenfolge der statischen Objekte ist nicht festgelegt. End Trace: 1; End Trace: 0

3.3 Objekte mit dynamischer/manueller Fortdauer

Mit den Schlüsselwort new an beliebiger Stelle im Programm erzeugt sind dynamische Objekte über alle scope-Grenzen hinweg gültig. Werden sie nicht mehr benötigt müssen sie explizit mit delete zerstört werden.

Bsp.:

```
int main()
{
      Trace *p = new Trace(1);
      delete p;
      return 0;
}
```

output:

Start Trace: 1 End Trace: 1

Die Objekterzeugung mit new geschieht in drei Schritten:

- Ermitteln des benötigten Speicherplatzbedarfs (i.A. (`sizeof()`))
- Alloziierung von Speicher und Rückgabe eines Pointers auf den Speicher ((`malloc()`))
- Initialisierung des Objektes (Konstruktoraufruf)

new entgegengesetzt sorgt delete für die Zerstörung des Objekts:

- Deinitialisierung (Destruktorruf)
- Speicherfreigabe ((`free()`))

new und delete sind malloc und free immer vorzuziehen (malloc und free sind C-Funktionen und haben keine Ahnung von Objekten)!

Mit dem Wissen um dynamische Objekte kann das Beispiel aus „Objekte mit automatischer Fortdauer“ effizienter umgesetzt werden.

```
Teuer* create_T()
{
      //anstelle des ganzen Objekts wird lediglich ein
      //Pointer auf das Objekt zurückgegeben.
      return new Teuer;
```

```
}

int main()
{
      //da das Objekt aus create_T dynamisch erzeugt ist lebt es bis zu
      //seiner expliziten Zerstörung.
      Teuer *c = create_T();

      // ...

      delete c;
}
```

Auch wenn diese Lösung effizienter ist hat sie ein Makel. Mit dem Pointer auf das Objekt erbt der Aufrufer auch die Eigentümerschaft am Objekt und muss sich um seine Zerstörung kümmern. Um diesen Umstand zu verdeutlichen werden Funktionen mit dieser Eigenschaft oftmals (`create*`) genannt.

3.4 Temporäre Objekte

Temporäre Objekte werden implizit vom Compiler erzeugt und besitzen keinen Namen. Sie entstehen meist bei der Berechnung arithmetischer Ausdrücke(x*y*z) und dienen nicht zuletzt der besseren Lesbarkeit von Quellcode. Soll beispielsweise das Produkt dreier Variablen errechnet werden muss zunächst ein Teilergebnis (Variable 1 x Variable 2) errechnet und als temporäres Objekt abgespeichert werden bevor das Ergebnis des Gesamtausdrucks ermittelt werden kann. Ohne temporäre Objekte müsste dieser Zwischenschritt explizit niedergeschrieben werden. Temporäre Objekte werden nach Evaluierung des vollständigen Ausdrucks (Ausdruck der nicht Teil eines anderen Ausdrucks ist) in dem sie erzeugt wurden zerstört. Allerdings existieren zwei Ausnahmen: Wird das Objekt zur Initialisierung eines benannten Objekts benuzt exisitiert das temp. Objekt bis die Initialisierung vollständig abgeschlossen ist. Ist das temp. Objekt an eine Referenz gebunden so existiert es für die Lebenzeit der Referenz oder bis zum Ende des scopes in dem es erzeugt wurde, je nachdem was früher eintritt.

Bsp.:

```
//Funktion g mit temporärem Objekt...
void g()
{
      //namensloses temporäres Objekt wird erzeugt
      int k = String("GURU").length();
}

//... entsprechende Fkt. g ohne temp. Objekt.
```

```
void g()
{
      String temp("GURU");
      int k = temp.length();
}
```

Bsp.:

```
//temp. Objekt an Referenz gebunden.
//temp. Objekt lebt bis Scopeende oder Zerstörung der Referenz.
const String &space = " ";

//entspricht:
String temp(" ");
const String &space = temp;
```

4 Iteratoren

4.1 Einleitung

Ausgangspunkt für die Entwicklung des Konzepts von Iteratoren waren die Erfahrungen und Problemen im Umgang mit Zeigern, welche auf Arrays oder anderen Datenstrukturen operieren.

Zwei wesentliche Anforderungen an Iteratoren entstanden daraus:

- Die Datenstruktur sollte verändert bzw. ausgetauscht werden können, ohne dass sich die Zugriffsweise auf deren Elemente verändert.

- Das Iterations- bzw. Traversierungsverfahren sollte verändert oder ausgetauscht werden können, ohne dass die Datenstruktur verändert werden muss.

Um dies zu ermöglichen werden Iteratoren als Abstraktionsschicht zwischen dem Zugriff auf Daten und der darunter liegenden Datenstruktur verwendet. Ausschließlich durch den Iterator beziehungsweise seinen Typ wird definiert, wie die Elemente einer Datenstruktur verarbeitet werden, zum Beispiel in welcher Reihenfolge, und wie sei manipuliert werden können.

In der Syntax und in der Anwendung orientieren sich Iteratoren stark an Pointern.

4.2 Iterator Pattern

4.2.1 Theorie

Das *iterator pattern* [Gamma'94] stellt eine Definition für Iteratoren zur Verfügung und beschreibt die sprachunabhängige Funktionsweise und Implementation.

Definition:

Ein Iterator oder Cursor ermöglicht es auf die Elemente eines zusammengesetzten Objektes zuzugreifen, ohne die darunterliegende Struktur kennen zu müssen.

Dabei wird zwischen zwei Kategorien von Iteratoren unterschieden:

Internal Iterators: Der Iterator agiert autonom und verarbeitet nach der Initialisierung

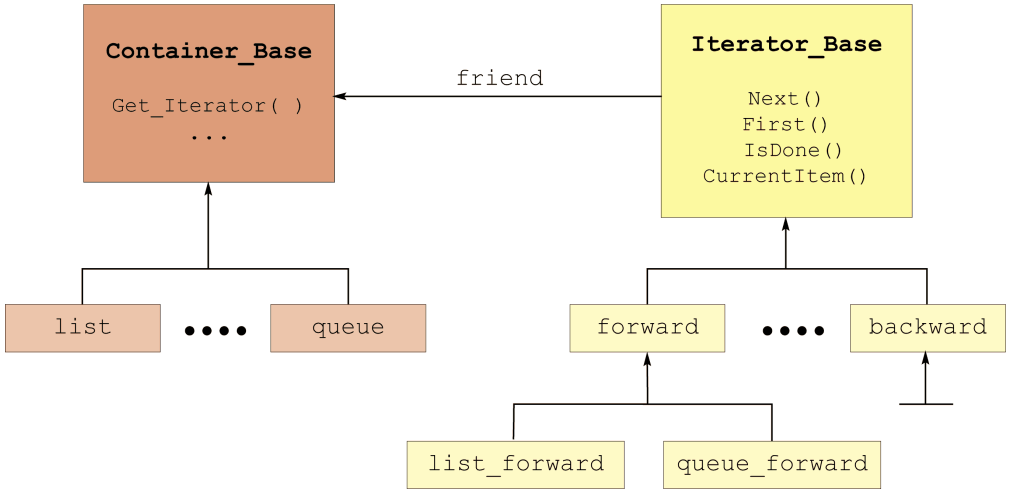


Abbildung 5: Klassendiagramm *iterator pattern* in C++

die vorgegebene Sequenz selbstständig. *Internal iterators* werden hauptsächlich in Sprachen wie *lisp* oder *scheme* eingesetzt, die anonyme Funktionen, continuations und closures unterstützen; die Verwendung in C++ ist nicht üblich.

External Iterators: Der Iterator wird in jedem Schritt explizit kontrolliert und gesteuert. Die Iteratoren der *Standard Template Library (STL)* fallen in diese Kategorie.

Auf allen Iteratoren müssen dabei folgende Operationen definiert sein:

`First()`: Die Position des Iterators wird auf das erste Element der Sequenz gesetzt.

`Next()`: Die Position des Iterators wird um eine Position weiterbewegt.

`IsDone()`: Die boolsche Funktion gibt `true` zurück, wenn das Ende der Sequenz erreicht ist.

`CurrentItem()`: Der Dereferenzierungsoperator; er gibt den Wert des Elements zurück, auf das der Iterator gerade zeigt.

4.2.2 Implementation in C++

In C++ werden keine anonymen Funktionen, continuations und closures unterstützen, so dass auch die Verwendung von *internal iterators* nicht üblich ist. Im Folgenden wird sich deshalb auf *external iterators* beschränkt.

Bei einer Implementation in C++ sollten sowohl die Datenstrukturen, meist als Container bezeichnet, als auch die Iteratoren von Basisklassen abgeleitet werden (Abbildung 5 auf der vorherigen Seite) um den gestellten Anforderungen, sowohl hinsichtlich der Austauschbarkeit der Container als auch bezüglich der Austauschbarkeit der Iteratoren, zu genügen. Dabei ist für jeden Containertyp die Spezialisierung der Iteratortypen notwendig, die auf diesem verwendet werden sollen. Dies kann nicht vermieden werden, da ein Iterator auf der internen Datenstruktur des Containers arbeiten muss.

Um innerhalb der Iteratoren einen einfachen Zugriff auf die Elemente der Container zu ermöglichen, werden die Iteratoren- und Container-Klassen als `friend` deklariert.

4.3 Benutzer-definierte STL Iteratoren

4.3.1 Motivation

Das Iterator Konzept ist in C++ in der *Standard Template Library (STL)* implementiert. Für einen Benutzer-definierten Iterator ist es deshalb im Allgemeinen sinnvoll, auf diesem Konzept und den dort schon vorhandenen Funktionalitäten aufzubauen.

Die *STL iterators* sind allerdings nur auf den *STL container* definiert. In der Praxis liegen Information jedoch häufig in nicht *STL*-konformen Datenstrukturen vor, wobei es es oft wünschenswert ist, mit *STL*-kompatiblen Iteratoren auf diesen Objekten arbeiten zu können. Insbesondere die dadurch mögliche Benutzung der *STL algorithms* zur Verarbeitung der Datenstrukturen bietet zahlreiche Möglichkeiten, die Lesbarkeit des Codes und die Effizienz bei der Verarbeitung zu erhöhen.

4.3.2 Anforderungen

Zur Definition von *STL*-kompatible Iteratoren auf beliebigen Datenstrukturen muss die angestrebte Benutzer-definierte Iteratorklasse von der Klasse `std::iterator` abgeleitet werden, welche die *STL* speziell zu diesem Zweck zur Verfügung stellt. Um die Initialisierung der vier Template Parameter von `std::iterator` zu erleichtern, existieren *iterator tags*, welche fuer einen Iteratortyp die korrekte Initialisierung sicherstellen. Für jeden Iteratortyp wie zum Beispiel *forward iterator* oder *random-access iterator* existiert dabei ein eigenes *iterator tag*, welches als Parameter die Klasse `T` erhält. Diese Klasse definiert den Datentyp der Elemente der Datenstruktur.

Die Benutzer-definierte Iterator Klasse muss dabei folgende öffentliche Operatoren implementieren um die Interoperabilität mit der *STL* zu gewährleisten.

- *standard constructor*
- *standard destructor*
- *pre-increment*
- *unequality test*

Zusätzlich muss es möglich sein, den Iterator auf mindestens zwei verschiedene Positionen auf einem Container, im Allgemeinen an den Anfang (`begin()`) und an das Ende (`end()`) zu setzen, zwischen welchen dann iteriert werden kann. Diese Operationen können sowohl auf dem Container als auch auf dem Iterator definiert werden.

4.4 Zusammenfassung

Iteratoren bieten durch ihre Abstraktion eine effiziente Möglichkeit um insbesondere auf komplexen Datenstrukturen zu arbeiten. Die Vorteile liegen dabei vor allem in der unabhängigen Austauschbarkeit von Interface und Datenstruktur und der zusätzliche Übersichtlichkeit im Programmcode.

In C++ ist es dabei sinnvoll, eigene Iteratoren auf Grundlage der *STL iterators* zu definieren. Dies wird durch die Iterator-Basisklasse `std::iterator` und die *iterator tags* unterstützt und ermöglicht die Benutzung der *STL algorithms*.

4.4.1 Beispiel

Als Beispiel wurde ein *forward iterator* implementiert, der auf einer Textur beziehungsweise einem Bild operiert. Die Textur wird, wie zum Beispiel in OpenGL üblich, durch ein `unsigned char`-Array repräsentiert. Das Beispielprogramm ersetzt dabei alle grünen durch rote Pixel.

```
// definition of the iterator class
template<typename T>
class iter_Pic:
  std::iterator<std::forward_iterator_tag, T> {

public:

  //default constructor
  iter_Pic();

  // standard constructor
```

```
  iter_Pic(T* pic, int width, int height);

  // destructor
  ~iter_Pic();

  // copy constructor
  iter_Pic(const iter_Pic & iter);

public:

  // increment operator
  iter_Pic<T> & operator++();

  // unequality
  bool operator!=(const iter_Pic &iter);

  // point to the past-end element
  void End();

  // point to the first valid element
  void Begin();

  // dereferencing
  Pixel<T> & operator*();

private:

  struct iter_pimpl * pimpl;
};


/*************************************************/
/* main (excerpt) */
/*************************************************/

// initalisation of the iterators
// pic is the unsigned char array
iter_Pic<unsigned char> iter(pic, width, height);
iter_Pic<unsigned char> iter_end(iter);
iter_end.End();

// count the green pixels
// use the user-defined functor Count_Green for this
for_each(iter, iter_end, Count_Green<unsigned char>());

// replace green by red pixel
iter.Begin();
Pixel<unsigned char> rep_old(&zero, &full, &zero);
Pixel<unsigned char> rep_new(&full, &zero, &zero);
```

```
replace(iter, iter_end, rep_old, rep_new);

// check the result
iter.Begin();
for_each(iter, iter_end, Count_Green<unsigned char>());
```

5 vector<bool> vs. std::bitset

5.1 bool in C/C++

Der Datentyp bool wurde in C/C++ erst in den späten 90ern integriert. Zuvor bediente man sich diverser Hilfskonstrukte, um Integer in das true/false-Schema zu pressen ([Informit]):

```
enum { false, true };

...
#define TRUE  1;
#define FALSE 0;

...
typedef unsigned int BOOL;
```

Um diesem Wildwuchs ein Ende zu setzen, wurde der Datentyp bool eingeführt. Er kann zwei Werte annehmen, true und false. Das Ergebnis eines konditionalen Ausdrucks, z. B. in einer if-Anweisung, ist ein temporäres Objekt von Typ bool. Wichtig ist auch die implizite Konversion von bool in einen integer-Typ. D. h. für int spezialisierte Methoden nehmen ohne weiteres auch einen bool als Argument an.

```
func(int a) { ... }

int main()
{
        bool b = false;
        func(b);  // --> func(0);

        return false;
}
```

False entspricht einem Integer-Wert von Null, true einem Integer-Wert ungleich Null, ist aber nicht notwendigerweise gleich Eins! Die Größe einer bool-Variable hängt von dem verwendeten System und Compiler ab. Meistens wird nur ein Byte verwendet (IA32, gcc3 und MSVC 5+). Aber man darf sich nicht darauf verlassen. Auf Plattformen, die einzelne Bytes nur langsam adressieren können, ist bool eventuell zwei oder sogar vier Bytes groß.

5.2 vector<bool>

5.2.1 Beispiel

Gemäß der STL-Definition für vector kann man in der üblichen Art und Weise einen vector<bool> erzeugen. [CPPR]:

```
#include <vector>

std::vector<bool> bvec(1234, false);

bvec[123] = true;
bool my_bool = (true && (!bvec[123]));

std::vector<bool>::iterator i;
for ( i = bvec.begin(); i != bvec.end(); i++)
{
    bool b = *i;
}

bvec.push_back(true);
```

5.2.2 vector<bool> im C++-Standard

Angenommen, `sizeof(bool)` wäre eins, und man legt sich einen vector<bool> im Gegenwert von 8 MB Speicher an:

```
std::vector<bool> bvec(8388608, true);  // 8 MB bools?
```

Anhand des Speicherbedarfs der Anwendung kann man beobachten, dass für das obige Konstrukt nur ca. 1 MB Speicher verwendet wurde! Offensichtlich wurde im Hintergrund Aufwand betrieben, um die Größe des verwendeten Speichers zu reduzieren.

Auszug aus [C++ 97], Kapitel 23.2.5, lib.vector.bool:

To optimize space allocation, a specialization of vector for bool elements is provided.

D. h. `vector<bool>` ist nicht ein normaler `vector<T>` mit `T = bool`, sondern eine Spezialisierung von `vector<T>`. Offensichtlich wird in `vector<bool>` ein integer-Typ verwendet, um mehrere bools kompakt als einzelne Bits zu repräsentieren.

5.2.3 vector<bool> vs. vector<T>

Natürlich stellt sich die Frage, inwiefern sich das Spezialkonstrukt vector<bool> von einem normalen vector<T> unterscheidet. Dazu zunächst ein Auszug aus der Klassendefinition für vector<T>:

```
namespace std
{
        template <class T, class Allocator=allocator<T> > class vector
        {
        public:
         // types:
         typedef typename Allocator::const_reference const_reference;
         typedef typename Allocator::reference reference;  // -> T&

            typedef (implementation defined) iterator; // See 23.1
         [...]
         typedef typename Allocator::pointer pointer;

         [...]
         reference       front();
         [...]
       }
}
```

Relevant ist die Typdeklaration *reference*, die bei vector<T> immer T& ist. Im Gegensatz dazu nun die Klassendefinition für vector<bool> ([C++ 97]):

```
namespace std {
         template <class Allocator> class vector<bool, Allocator>
        {
         public:
         // types:
            typedef bool const_reference;
        [...]

            typedef (implementation defined) iterator; // See 23.1
        [...]
        typedef typename Allocator::pointer pointer;

        // bit reference:
           class reference {
              friend class vector;
              reference();
             public:
              ~reference();
              operator bool() const;
              reference& operator=(const bool x);
```

```
            reference& operator=(const reference& x);
            void flip(); // flips the bit
        };

      [...]
      reference       front();
      [...]
    }
}
```

Anstelle von T& tritt hier eine Hilfsklasse *reference*. Der C++-Standard sagt zu dieser Hilfsklasse folgendes.

Auszug aus [C++ 97], Kapitel 23.2.5, lib.vector.bool:

reference is a class that simulates the behavior of references of a single bit in vector<bool>

Da der Zugriff auf einzelne Bits nicht möglich ist, hilft man sich mit der Klasse *reference*, auch Proxy genannt. Diese Klasse fungiert als vorgeschaltete Instanz beim Zugriff auf die Daten.

5.2.4 Probleme in Template-Umgebungen

Da die Proxyklasse den Zugriff auf die bool-Elemente nur simuliert, treten Schwierigkeiten auf, sobald man als Programmierer auf die Elemente direkt zugreifen will. Da sich vector<bool> von vector<T> in diesem Punkt unterscheidet, muss man besonders in Template-Umgebungen auf die Besonderheiten von vector<bool> eingestellt sein. [GotW][NGRC++][GotW #50]

Die folgenden Beispiele sollten für alle Typen außer bool funktionieren.

```
// Example 1: Works for every T except bool

template<class T> class myclass {
public:
  void func( std::vector<T>& v );
};

template<class T>
void myclass<T>::func( std::vector<T>& v ) {
     T* p = &*v.begin();
     // ... do something with *p ...
};
```

In func() wird das erste Element des Vektors dereferenziert: *v.begin(). Von diesem Objekt wird dann mit dem &-Operator die Adresse des ersten Elements in p gespeichert. Der passende Datentyp ist T*. Jedoch ist *v.begin() kein bool, sondern eine temporäre(!) Instanz der Proxyklasse reference. Entsprechend kann keine Typkonversion durchgeführt werden:

```
 test.c++: In member function void myclass<T>::func(std::vector<T,
     std::allocator<_CharT> >&) [with T = bool]:
test.c++:34: instantiated from here
test.c++:17: warning: taking address of temporary
test.c++:17: cannot convert std::_Bit_reference* to bool* in initialization
```

Ein Ausweg könnte vector<T>::pointer sein.

```
template<class T>
void myclass<T>::func( std::vector<T>& v ) {
     typename vector<T>::pointer p = &*v.begin();
     bool b = *p;
};
```

Anscheinend funktioniert hier die Zuweisung von &*v.begin(). Jedoch steckt hinter dem Pointer p immer noch die Proxyklasse reference.

Der folgende Code wird trotz vector<T>::pointer nicht funktionieren:

```
template<class T>
void myclass<T>::func( std::vector<T>& v ) {
     typename vector<T>::pointer p = &*v.begin();
     printf("%d\n", *p);
};
```

Die Funktion printf() erwartet einen einfachen Datentyp, bekommt hier aber eine Klasseninstanz geliefert. Entsprechend die Fehlermeldung des Compilers:

```
test.c++: In member function void myclass<T>::func(std::vector<T,
     std::allocator<_CharT> >&) [with T = bool]:
test.c++:34: instantiated from here
test.c++:17: warning: taking address of temporary
test.c++:21: warning: cannot pass objects of non-POD type struct
     std::_Bit_reference through...; call will abort at runtime
```

(POD: plain old datatype)

5.2.5 Zusammenfassung

Die Spezialisierung von vector<bool> zur Speicherplatzoptimierung und die daraus resultierende Einbindung einer Proxyklasse hat zur Folge, dass sich vector<bool> nicht in allen Belangen wie ein normaler STL-vector verhält. Folglich muss man in Template-Umgebungen bei der Spezialisierung auf bool mit Problemen rechnen, sobald man versucht, auf die bools direkt zuzugreifen.

Anstatt dem Programmierer die Möglichkeit zu geben, selbst zu entscheiden, ob bei vector<bool> eine Speicherplatzoptimierung vorgenommen werden soll, oder nicht, setzt sich der Standard hier fest und verursacht damit die gezeigten Inkonsistenzen.

5.3 std::bitset

Bitset<N> ([STL'00], [CPPR]) ist eine Alternative zu vector<bool>. Der Template-Parameter N gibt die gewünschte Größe des Bitvektors an und muss zur Compile-Zeit festgelegt werden. Bitsets sind also immer statisch. Zudem steht bitset in keinem Zusammenhang zu den STL-Containern, d. h. es gibt u. a. auch keine Iteratoren auf einem Bitset.

```
#include <bitset>

std::bitset<58> bset_58;  // alle auf 0

// 64 < 58 -> 8 bytes
// bset_58[0] ist LSB, also 2^0
```

Auch im bitset werden integers benutzt, um die true/false-Information kompakt als Bits abzuspeichern. Zudem stehen alle Bitoperatoren zur Verfügung, was vector<bool> nicht bieten kann.

5.3.1 Beispiele

Im Konstruktor eines bitsets kann man mit einem string die Vorbelegung der Bits angeben:

```
string my_bits("00001111");      // last char is LSB
bitset<8> bset(my_bits);
bset[1] = false;
cout << bset << end;             // console 00001101
```

Alternativ kann man auch ein unsigned long spezifizieren, dessen binäre Representation zur Initialisierung des bitsets verwendet wird. Dabei muss man beachten, dass so nur die ersten 32, bzw. 64 Bit des bitsets gesetzt werden. Die restlichen Bits werden auf 0 gesetzt.

```
// bitset<8> bset(ulong val);
bitset<8> bset(127);
cout << bset << end;            // console: 01111111
cout << bset.to_ulong() << end; // console: 127
```

Alle unären und binären Bitoperatoren stehen zur Verfügung.

```
// operatoren, binaer & | ^, unaer ~ << >>
bitset<8> bset(0xF0);           // dezimal 240
bset >>= 4;
cout << bset.to_ulong() << end; // console: 15

bitset<8> bset_2(0xF0);
bset &= bset_2;
cout << bset << end;            // console: 00000000
```

5.4 std::bitset vs. vector<bool> vs. boost::dynamic_bitset

Zusammenfassend kann man für std::bitset und vector<bool> zwei Einsatzszenarien abstecken.

- std::bitset sollte man verwenden, wenn die Anzahl der benötigten Bits statisch, bzw. bekannt ist und die klassischen Bitoperationen verwendet werden sollen
- vector<bool> kann man verwenden, wenn der Anwendungsbereich über das Speichern von An/Aus-Flags nicht hinaus geht und die Einschränkungen von vector<bool> bekannt sind

Als dynamische Alternative zu std::bitset bietet die boost-library `dynamic_bitset` an. [boost] Es werden alle Funkionalitäten von std::bitset geboten. Zusätzlich kann die Größe des `dynamic_bitset` im Konstruktor zur Laufzeit spezifiziert werden. Die Methode `resize()` bietet die Möglichkeit, die Größe des `dynamic_bitset` jederzeit anzupassen.

```
boost::dynamic_bitset<> dbset(num_bits, value);
```

6 std::vector<T> vs. T*

6.1 T*

In C und C++ können mehrere Objekte gleichen Typs zu Arrays zusammengefasst werden. Auf ein Element des Arrays wird mit einem Zeiger auf das erste Element und einem Index (einer positiven Zahl) zugegriffen.

```
double da1[10];
double da2[] = {1.0, 2.0, 3.0};

da1[5] = da2[0];
*(da1 + 5) = *(da2 + 0);
```

Die Indizierung kann mittels *subscripting* oder – was äquivalent ist – mittels Zeiger-Arithmetik erfolgen. Da heißt, dass die Elemente in einem Array in kontinuierlich aufeinander folgendem Speicher liegen. Die Größe von Arrays muß zur Kompilierzeit festliegen.

Um Arrays (oder andere Objekte) mit einer Größe zu erzeugen, die erst zur Laufzeit bekannt ist, stehen in C die Funktionen `malloc` und `free` zur Verfügung. Der durch `malloc` erhaltene Speicherblock wird durch einen Zeiger auf das erste Element verwaltet. In C++ gibt es statt `malloc` und `free` die Operatoren `new` und `delete`.

```
size_t size;
... /* get size at runtime */
double* da=(double*)malloc(size*sizeof(double));
... /* use da like an array double da[size] */
free(da);

// C++
std::size_t size;
... // get size at runtime
double* da=new double[size];
... // use da like an array double da[size]
delete(da);
```

Die Aufrufe von `malloc` und den Operatoren `new` können auch fehlschlagen, also muss man eine Fehlerbehandlung für diesen Fall einbauen. Dynamisch erzeugte Arrays können nicht wachsen, ohne dass neuer Speicher angefordert wird, und der Speicher muss explizit freigegeben werden können. Der Speicher, der `malloc` und `free` und den Operatoren `new` und `delete` zurückgegeben wurde, ist kontinuierlich, so dass mittels Zeiger-Arithmetik oder dem äquivalenten *subscripting* auf diesen zugegriffen werden kann.

6.2 std::vector<T>

Da mit Arrays einfach umzugehen ist, wenn sie einmal erzeugt sind, und man sie ständig braucht, wünscht man sich eine Alternative ohne lästige manuelle Speicherverwaltung – und das ist, was der `vector<T>` aus der C++ Standard Bibliothek bietet. Er ist ein sequentieller Container, der dynamisch wachsen kann, und sich wie ein Array verhält. Hier ein Vergleich der Benutzung eines `vectors` und eines Arrays:

```
{
  vector<T> vT(n);          T* pT = new T[n]();
  vT.begin();               pT;
  vT.end();                 pT + n;
  vT[i];                    pT[i];
  vT.at(i);                 if(i < n)
                              pT[i];
                            else
                              throw std::range_error;
                            T* tmp = pT;
  vT.push_back(T());        pT = new T[n + 1]();
                            copy(tmp, tmp + n, pT);
                            pT[n] = T();
                            delete[] tmp;
                            delete[] pT;

}
```

Der `vector` ist ein Stackobjekt, das einen Zeiger zum eigentlichen Speicher enthält, und welches diesen Speicher verwaltet, also wenn es zerstört wird, seinen Speicher ebenfalls freigibt.

Der Standard definiert den `vector` in Kapitel 23.2.4 .Er besitzt *random-access*-Iteratoren, die Komplexität des Einfügens und Löschens eines Elements ist konstant am Ende des `vectors` und linear in der Mitte oder am Anfang. Der Typ der enthaltenen Elemente muss *copy-constructible* und *assignable* sein. Die Komplexität der Konstruktion aus einem *range* der Größe N gegeben durch zwei Iteratoren hängt von der Art der Iteratoren ab – wenn es mindestens *forward*-Iteratoren sind, wird einmal Speicher angefordert und N mal der *Copy*-Konstruktor aufgerufen; wenn es nur *input*-Iteratoren sind, wird höchstens $2 \cdot N$ mal der *Copy*-Konstruktor aufgerufen, und $\log n$ mal Speicher angefordert.

Zwei wichtige Attribute eines `vectors` sind die `capacity` und die `size`, abzufragen durch zwei gleichnamige Memberfunktionen. Die `size` gibt die Menge an Elementen an, die

er enthält; die `capacity` die Menge an Elementen, die er enthalten könnte, ohne neuen Speicher anzufordern.

Man kann beide Werte vergrößern: durch `resize` wird die `size` vergrößert, und durch `reserve` die `capacity`. Man kann aber die `capacity` eines `vector` nicht wieder schrumpfen. Doch mittels einem Trick mit `swap` lässt sich ein `vector` verkleinern:

```
vector<T>(v).swap(v);
```

Durch `vector<T>(v)` wird eine temporäre Kopie von `v` erzeugt, die aber (hoffentlich) eine kleinere `capacity` als `v` hat. Diese Kopie wird mit `v` vertauscht, und `v` wird gelöscht, zurück bleibt die kleinere Kopie. Leider kann nicht garantiert werden, dass die `size` von `v` nach dieser Aktion gleich der `capacity` ist.

6.3 Benutzung von std::vector

Der `vector` ist der Container, der generell verwendet und der Benutzung dynamisch erzeugter Arrays vorgezogen werden sollte. Die Aufgaben, die man bei dynamischer Speicherverwaltung erledigen muss, und die Fehler, die man dabei machen kann, können bei Benutzung von `vector` vermieden werden. Er kann, da er ein kontinuierliches Speicherlayout besitzt, auch an C-Funktionen übergeben werden, die eigentlich einen Zeiger auf ein (kontinuierliches) Array erwarten.

```
int legacy_func(int const* array, size_t length);
vector<int> v;
...
legacy_func(&v[0], v.size());
```

Man sollte das Verhalten des `vectors` beim Hinzufügen von Elementen kennen, um ineffiziente Benutzung zu vermeiden. Durch die Funktion `reserve` kann man dem `vector` Hinweise auf die erwartete Anzahl an Elementen geben:

```
size_t size;
... // get size at runtime
v.reserve(size); // allocate
generate_n(v.begin(),size,rand); // initialize
```

Hier kann man klar zwischen Speicherallokation und Konstruktion der Elemente trennen, und unnötige Reallokation vermeiden.

7 Fast Pimpl-Idiom

[Sutter'99] [Lippman'96]

Neben der Laufzeitperformance gilt es in großen Softwareprojekten auch die performance der Kompilierung zu optimieren. Ein großer Punkt dieses Unterfangens ist die Reduktion von Übersetzungszeitabhängigkeiten. Eine Änderung in einer Klassenimplementierung soll so wenig Änderungen wie möglich an anderen Stellen im Code nach sich ziehen. Ein eleganter Weg dort hin führt über das pimpl-Idiom.

7.1 Ausgangssituation

Gegeben sei eine Klasse X mit mehreren privaten Datenelementen, darunter nicht eingebaute (no UDTs) und nicht vorwärts deklarierbare (z. B.: STL-Typen) Datentypen.

```
//x.h
#include <iosfwd>
#include <list>
#include "a.h"
#include "b.h"
#include "c.h"

class X : public A
{
 public:
     X(const C&);

 private:
     std::list<C> clist_;
     B                 b_;
};
```

7.2 das pimpl-Idiom

Ziel des pimpl-Idioms ist es, Übersetzungszeitabhängigkeiten zu minimieren und Implementierungsdetails zu verstecken. 'pimpl' steht dabei für 'pointer to implementation'. Dahinter verbirgt sich ein undurchsichtiger Zeiger, also ein Zeiger auf eine vorwärts deklarierte jedoch nicht definierte Hilfsklasse.

```
//file x.h using pimpl
```

```
#include<iosfwd>

#include "a.h"

class C;

class X : public A
{
 public:
     X(const C&);

 private:
     //Zeiger auf vorwärts deklarierte Klasse.
     struct XImpl* pimpl_;
};


// x.cpp using pimpl

struct X::Ximpl
{
     //private Elemente, vollständig versteckt.
     std::list<C> clist_;
     B                 b_;
};
```

Im obigen Quellcode gut erkennbar, verschwinden alle privaten Datenelemente der Klasse und an ihre Stelle tritt der 'pimpl'. Über diesen kann auf die privaten Datenelemente zugegriffen werden. Dieses Vorgehen bringt Vorteile:

- Typen, die lediglich in der Klassenimplementierung verwendet werden müssen nicht im Client-Code definiert werden. Dadurch weniger include-Direktiven und damit beschleunigte Kompilierung.

- Die Implementierung einer Klasse kann geändert werden ohne dass der Client-Code neu kompiliert werden muss.

Allerdings hat das Idiom ein Performanceproblem. Bei jedem Erzeugen/Zerstören eines Objekts muss, auf Grund des Pimpls, Speicher dynamisch alloziiert bzw. freigegeben werden. Dieses Problem zu eliminieren ist nicht möglich, es zu minimieren schon. Das fast-pimpl-Idiom zeigt wie.

7.3 fast-pimpl-Idiom

Eine Möglichkeit die dynamische Speicheralloziierung zu beschleunigen ist es den new - Operator zu überladen und einen Allokator fester Größe (fixed-size allocator) zu verwenden. Diese geben immer einen Speicherblock der gleichen Größe zurück und können daher effizienter arbeiten als Allokatoren für den allgemeinen Gebrauch. Das fast-pimpl-Idiom bedient sich diesem Ansatz.

```
//file x.h
class X
{
      // ...
      struct XImpl* pimpl_;
};

//file x.cpp
#include "x.h"

struct X::XImpl
{
      //private stuff here...
      void* operator new(size_t) {/*...*/}
      void operator delete(void*) {/*...*/}
};

X::X() : pimpl_(new XImpl) {}
X::~X() {delete pimpl_; pimpl_ = 0;}
```

8 Der Building Prozess

[Linuxmagazin'04] [GCC] [GotW #32] [Wikipedia] [MSDN]

8.1 Einleitung

Der folgende Abschnitt beschäftigt sich mit dem Buildingprozess und der zentralen Frage „Wie komme ich vom Quellcode zum ausführbaren Programm?“

Den Prozess den ein Programm dabei durchläuft bis es lauffähig ist, kann man in folgende drei Hauptschritte unterteilen:

1. Präprozessor
2. Compiler
3. Linker und Loader

Im folgenden wird nun detaillierter auf diese Schritte eingegangen.

8.2 Präprozessor

Der erste Schritt, den der Quellcode durchläuft, ist der Präprozessor. Seine Hauptaufgabe ist die Manipulation des Eingabestroms (Quelltext). Der Präprozessor erhält seine Anweisung über das Steuersymbol `#`. Hauptanwendungen sind dabei das Einblenden von Header-Dateien in den Quelltext, zum Beispiel über:

```
#include  "name.h"  // Aktuelles Verzeichnis
#include <iostream> // Standard Header
```

Aber auch das einmalige Einblenden von Codeteilen mittels Inklusionswächter. Dies ist die gebräuchliche Technik um Mehrfach-Inklusionen zu vermeiden.

```
#ifndef MYPROG_X_H
#define MYPROG_X_H

// ... restlicher Code der Header-Datei

#endif
```

Ein weiteres wichtiges Beispiel für Anweisungen an den Präprozessor ist das Auswählen von Codefragmenten zur Kompilierzeit. Dies wird meist dazu genutzt, um Debuginformationen einzufügen oder plattformspezifische Features zu nutzen.

Einblenden von Debuginformationen

```
void f()
{
#ifdef MY_DEBUG
  cerr << "some trace logging" << endl;
#endif

  // ... der Rest von f() kommt hierhin...

}
```

Als Resultat liefert der Präprozessor einen Datenstrom in dem gezielt Codeteile ein- oder ausgeblendet wurden.

Eine letzte Möglichkeit ist das Einfügen von Präprozesor-Features wie `__FILE__`, `__LINE__` und `__DATE__`.

8.3 Compiler

Der durch den Präprozessor vorbereitete Datenstrom durchläuft nun als nächstes den Compiler. Diesen kann man in zwei Teile unterteilen:

1. Frontend
2. Backend

8.3.1 Frontend

Das Frontend ist der erste Schritt den der Quelltext im Compiler durchläuft. Man kann das Frontend dabei in vier Unterschritte unterteilen:

1. Lexikalische Analyse: Die lexikalische Analyse zerteilt den eingelesenen Quelltext in zusammengehörende Token verschiedener Klassen, zum Beispiel Schlüsselwörter, Bezeichner, Zahlen und Operatoren. Dieser Teil des Compilers heißt Scanner.Ein Scanner benutzt gelegentlich einen separaten Screener, um Whitespace (Leerzeichen, Zeilenenden, ...) und Kommentare zu überspringen.

2. Syntaktische Analyse: Die syntaktische Analyse überprüft, ob der eingelesene Quellcode formal richtig ist, d. h. der Syntax (Grammatik) der Quellsprache entspricht. Dabei wird die Eingabe in einen Syntaxbaum umgewandelt. Dieser Teil wird auch als Parser bezeichnet.

3. Semantische Analyse: Die semantische Analyse überprüft die statische Semantik, also „logische Rahmenbedingungen“. Zum Beispiel muss eine Variable deklariert worden sein, bevor sie verwendet wird, und Zuweisungen müssen mit ihrem Datentyp kompatibel (verträglich) sein.

4. Synthesephase oder Zwischencodegenerierung: Die Synthesephase erzeugt aus dem in der Analysephase erstellten Baum den Programmcode der Zielsprache.

8.3.2 Backend

Auch das Backend kann man wieder in mehrere Teilschritte unterteilen.

1. Programmoptimierung: Üblicherweise bietet ein Compiler Optionen für verschiedene Optimierungen mit dem Ziel, die Laufzeit oder den Speicherplatzbedarf des Zielprogramms zu verkleinern. Die Optimierung erfolgt dabei in Abhängigkeit von den Eigenschaften der Hardware, insbesondere wieviele Register der Prozessor des Computers zur Verfügung stellt.

2. Bei der Codegenerierung wird endgültig aus dem Syntaxbaum Programmcode in der Zielsprache erzeugt. Falls die Zielsprache die Maschinensprache ist, kann das Ergebnis direkt ein ausführbares Programm sein oder eine so genannte Objektdatei, die durch das Linken mit weiteren Objektdateien zu einer Bibliothek oder einem ausführbaren Programm führt.

8.4 Linker und Loader

8.4.1 Funktion des Linker

Nach dem Kompilieren sind die meisten Objektdateien nicht lauffähig, denn praktisch jedes Programm benutzt Konstrukte, die nicht von ihm selbst definiert wurden. Man denke nur an die Ein- und Ausgabe-Operationen. Der Linker hat die Aufgabe die fehlenden Teile hinzuzufügen. Man beachte den Unterschied zum Präprozessor: Der Präprozessor vervollständigt das Programm auf textueller Ebene. Alles was er macht, könnte man

auch mit einem Texteditor selbst erledigen. Der Linker vervollständigt Programme auf der Ebene des Objektcodes. Folgendes Beispiel soll dies verdeutlichen:

main.cpp

```
void f();

int main(){
  f();
  return 1;
}
```

f.cpp

```
# include <iostream>

void f(){
  std::cout<<Hallo Welt<<std:endl;
}
```

Die erste Datei (main.cpp) enthält die Funktion main die zweite (f.cpp) enthält f. Beide Dateien können übersetzt werden:

```
g++ -c main.cc
g++ -c f.cc
```

Das Ergebnis der beiden Übersetzungen sind zwei Objektdateien main.o und f.o. Man beachte, dass keine der beiden Dateien für sich allein zu einem ausführbaren Programm gebunden werden kann. Die Aufrufe:

```
g++ main.cc
g++ f.cc
```

führen zu Fehlermeldungen, in denen sich der Compiler über das fehlende main bzw. f beklagt. Beide zusammen machen erst ein ausführbares Programm aus. Die Option -c weist g++ an, nur als Compiler tätig zu werden und nicht zu versuchen ein ausführbares Programm zu erzeugen. Mit

```
g++ main.o f.o
```

werden die erzeugten Objektdateien main.o und f.o zu einem ausführbaren Programm a.out gebunden. Auch hier erkennt g++ wieder an den Endungen .o dass er als Linker und nicht als Compiler agieren soll. Der Linker nimmt den auf zwei Dateien verteilten

Objekt-Code und fügt ihn zu einem Programm zusammen.

8.4.2 Bibliotheken

Ein zweite wichtige Aufgabe des Linkers ist das Einbinden von Objektbilbiotheken. Eine Bibliothek (Archiv) ist eine Datei, die eine Kollektion anderer Dateien enthält. Die wichtigste Anwendung der Dateiarchive sind die Objektbibliotheken: Kollektionen von Objektdateien in einer Archivdatei. Dabei können folgende Arten von Bilbiotheken unterschieden werden:

- Static Libraries
- Shared Libraries
- Dynamically Loaded (DL) Libraries

Im Folgenden werden diese näher erklärt.

8.4.3 Static Libraries

Statische Bibliotheken sind eine einfache Sammlung von einfachen Objektdateien. Die Konvention besagt das statische Bibliotheken die ".a" Endung besitzen. Die Sammlung wird unter Nutzung des ar (archiver) Programms erstellt. Statische Bibliotheken werden heute nicht mehr so häufig wie früher genutzt wegen der Flexibilität die Shared Libraries bieten.

Statische Bibliothekem erlauben es ein Programm zu linken ohne den Code neu zu kompilieren, was Kompilierzeit spart. Heutzutage werden statische Biliotheken oft genutzt wenn Entwickler erlauben wollen gegen ihr Biliothek zu linken aber den Quellcode nicht veröffentlichen wollen.

Um eine statische Biliothek zu erstellen oder Objektdateien zu eine existierenden Bibliothek hizuzufügen reicht folgendes Kommando:

```
ar rcs my_library.a file1.o file2.o
```

Dieses Beispielkommando fügt die Objektdateien file1.o und file2.o zu der statischen Biliothke my_library.a hinzu oder erstellt eine neue Bibliothek my_library.a falls diese noch nicht existiert.

Wenn man die nun erstellte Bibliothek nutzen möchte reichte es sie bei der Kompilierung mit der -l Option einzubinden.

```
g++ -o main main.cpp -lmy_library.a
```

Mit der Zeit stellte man aber fest, dass man mit statischen Bibliotheken zu unflexibel war und ging über zu der Entwicklung mit shared Libraries.

8.4.4 Shared Libraries

Shared Libraries sind Bibliotheken die von Programm beim Start geladen werden. Wenn eine Shared Library richtig installiert ist, benutzen alle Programme die danach starten automatisch die neue Shared Library. Diese Flexibilität unter Linux lässt auch folgende Aktionen zu:

- Update von Bibliotheken und Unterstützung von Prgrammen die ältere nicht abwärtskompatible Versionen dieser Bibliothek nutzen.
- Übergehen von bestimmten Bibliotheken oder Funktioen aus einer Bibliothek bei ausführen eines bestimmten Programms
- All dies wenn Programme laufen und schon existierende Bibliotheken nutzen

Für den Umgang mit Shared Libraries gibt es eine Reihen von Konventionen und Richtlinien. Man nuss den Unterschied zwischen dem Bibliotheksnamen, dem “Soname” und dem “realen Nameen” verstehen aber auch wo Shared Libraries abgelegt werden.

Jede Shared Library hat eine speziellen Namen den “Soname”. Der Soname hat den Präfix “lib”, den Namen der Bibliothek, die Phrase “.so”, gefolgt von einem Punkt und der Versionsnummer die jedesmal erhöht wird wenn sich das Interface ändert. Ein voll-qualifizierter Soname besitzt außerdem am Anfang noch den Pfad an der die Bibliothek liegt. Ein voll-qualifizierter Soname ist einfach ein symbolischer Link zu dem “realen Namen” der Shared Library.

Jede Shared Library hat außerdem noch den “realen Namen”, welcher der Dateiname ist indem der aktuelle Code enthalten ist. Der reale Name ist der Soname mit Hinzunahme eines weiteren Punktes und einer weiteren Versionsnummer. Diese gibt exakt an welche Version der Bibliothek installiert ist.

Als letztes gibt es noch einen Name den der Linker ausgibt wenn er eine Bibliothek anfordert (“Linker Name”). Dies ist der Soname ohne Versionsnummer

Der Schlüssel um mit Shared Libraries umzugehen ist der Umgang mit diesen Namen. Programme die Shared Libraries benötigen geben nur den Soname an. Aber wenn man selber eine Shared Library erstellt gibt man nur den Dateinamen mit einer Versionsnummer an. Wenn man diese Bibliothek in einen der dafür vorgesehenen Pfade installiert muss man nur noch ldconfig laufen lassen. ldconfig findet die existierenden Dateien und erstellt die Sonames als symbolischen Link zu den realen Namen und erstellt die Cache Datei /etc/ld.so.cache

Ein Beispiel für einen vollen Soname ist /usr/lib/libreadline.so.3, welchen ldconfig als einen symbolischen Link auf einen realen Namen wie z. B. /usr/lib/libreadline.so.3.0 setzen würde. Der Linker Name wäre dann /usr/lib/libreadline.so

Mit dem Unixbefehl ldd kann man sich ausgeben lassen welche Biliotheken von einem Programm benötigt werden. Hier das Beispiel von rename:

```
libc.so.6 => /lib/i686/libc.so.6 (0x41115000)
/lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x4fc16000)
```

Zu Erwähnen ist /lib/ld-linux.so.2 welches den Linux Loader darstellt der für das Laden von Bibliotheken zuständing ist.

Nun aber zu der Frage wie man Shared Libraries erstellt: Hier ist ein Beispiel welches zwei Objektdateien (a.o und b.o) und dann eine Shared Library erstellt, die beide Objektdateien enthält. Als ersten werden die beiden Dateien als Objektdateien kompiliert (-c) wichtig ist dabei die Option -fpic welche dafür sorgt das der Code Positionsunabhänig ist (fpic, “position independent code”). Die ist eine Vorrausetzung für Shared Libraries.

```
gcc -fPIC -c -Wall a.c
gcc -fPIC -c -Wall b.c
gcc -shared -Wl,-soname,libmystuff.so.1 \
    -o libmystuff.so.1.0.1 a.o b.o -lc
```

Eine spezielle Form der Shared Libraries stellen die Dynamically Loaded (DL) Libraries dar.

8.4.5 Dynamically Loaded (DL) Libraries

Dynamically Loaded Libraries sind Bibliotheken die zu einem anderen Zeitpunkt als den Startzeitpunkt des Programmes geladen werden. Sie sind sehr nützlich bei der Realisierung von Plugins oder Modulen, da sie gewährleisten, dass eine Biliothek erst geladen wird wenn sie benötigt wird. Ein Besipiel hierfür ist das Plugable Authentication Modules (PAM) welches Module bei der Authentifizierung hinzulädt. Unter Linux können DL

Libraries sowohl als Standard Objektdateien oder als Shared Library gebaut werden. Der Hauptunterschied ist, dass sie nicht automatisch beim starten des Programms geladen werden. Stattdessen gibt es eine Benutzer API zum öffnen von Bibliotheken, schauen nach Symbolen, Error Handling und schließen von Bibliotheken. Benutzer von C müssen um diese Schnitttelle nutzen zu können den Header «dlfcn.h» inkludieren.

Das Interface unter Linux ist im Grundsatz gleich dem Interface unter Solaris aber es wird nicht unter allen Plattformen unterstützt. HP-UX nutzt ein anderes Interface und Windows Plattformen benutzen DLL's mit einem ganz anderen Interface. Wenn es also wichtig ist auf Plattformunabhägigkeit zu achten sollte man sich einige Wrapping Bibliotheken bauen welche die Unterschiede ausgleichen.

Hier ist ein Beispiel für die Benutzung von DLL's. Es stammt von der man Page von dlopen. Das Beispiel läd eine Mathebibliothek errechnet den Cosinus und gibt diesen aus. In jedem Schritt wird dabei auf Fehler geprüft.

main.c

```
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>
int main() {
  void *handle;
  double (*cosine)(double);
  char *error;
  handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
  if (!handle) {
    fputs (dlerror(), stderr);
    exit(1);
  }
  cosine = dlsym(handle, "cos");
  if ((error = dlerror()) != NULL) {
    fputs(error, stderr); exit(1);
  }
  printf ("%f\n", (*cosine)(2.0)); dlclose(handle);
  return 1;
}
```

Das Programm main.c müsste dann so übersetzt werden:

```
gcc -o main main.c -ldl
```

Nachdem wir uns nun mit den Arten von Bibliotheken vertraut gemacht haben nun ein

Blick auf das Dateiformat indem sie vorliegen.

8.5 ELF Binärformat

ELF steht für Executable and Linkable Fileformat. Es ist das Dateiformat in dem die Ojektdateien und die auführbaren Dateien vorliegen. Die Vorgängerformate waren a.out und COFF. Die a.out Datei die nach dem Kompilieren ohne Nutzung der -o Option entsteht ist natürlich auch in ELF Format. Die Gründe für den Namen sind eher historische. Ab gcc 3.4 wird dabei ist die Unterstützung für das ältere a.out abgeschafft. Die Gründe für die Umstellung von a.out auf ELF waren dabei das Aufkommen von Shared Libaries welche sich im a.out Format nur schwer reallisieren ließen.

ELF Unterscheidet drei Typen von Dateien:

1. Relozierbare Binaries
2. Ausführbare Binaries
3. Dynamische Bibliotheken

8.5.1 Relozierbare Binaries

Relozierbare Binaries enthalten Code und Daten der für das Binden mit anderen Dateien geeignet ist. Sie enthalten keine feste Adressen für Code und Daten. Ein Beispiel für relozierbare Binaries sind Objektdateien oder Module (*.o oder *.obj). Methoden aus anderen Objektdateien erfordern einen Relokationseintrag welcher später die Adresse der Methode wird. Beim statisches Binden werden dabei alle Objektmodule in die ausführbare Datei kopiert. Anders bei dem Dynamisches Binden. Dort entsteht ein Verweis auf die Dynamische Bibliothek und die Methode.

8.5.2 Ausführbare Binaries

Ausführbare Binaries bestehen aus dem Code und den Daten, die ein Prozessor ausführen kann. Die Einträge bestimmen das Speicherlayout des Prozesses. Sie bestehen dabei meist aus Objektmodulen und Informationen aus dynamischen Bibliotheken. Der dynamische Binder sucht die Bibliotheken und blendet sie in den Adressraum des Prozesses ein.

8.5.3 Dynamische Bibliotheken

Dynamische Bibliotheken stellen eine Mischung aus relozierbaren und ausführbaren Binaries dar. Ein Beispiel sind Shared Libraries (*.so). Code und Daten sind dabei Positionsunabhängig da noch nicht bekannt ist an welcher Stelle sie eingeblendet werden. Einmal geladen kann Code in den Prozessraum vieler Prozesse eingeblendet werden.

8.5.4 Aufbau

Eine Auflistung des genaueren Aufbau einer ELF Dtei würde den Rahmen dieses Abschnitts sprengen und deshalb sei auf das Programm elfread (Programmpaket elfutils) oder objdump verwiesen welche den Inhalteiner ELF Datei mal mehr, mal weniger gegliedert widergeben. Hier als Beispiel der ELF Header von rename:

```
file format elf32-i386
architecture: i386,
flags 0x00000112: EXEC_P, HAS_SYMS, D_PAGED
start address 0x080485b0 ELF
```

Es sei auch noch auf den Artikel „ELF, das Executable and Linkable Format im Detail“ im Linuxmagazin Ausgabe Mai'04 verwiesen (Bezugsquelle im Anhang).

8.6 Precompiled Header

Wir wir nun wissen kopiert der Präprozessor über #include „header.h“ den Inhalt von header.h an die Stelle wo #include aufgerufen wird. Auf den ersten Blick stellt dies kein Problem dar, doch bei großen Projekten erfordert das Parsen des komplettem Moduls sehr langen Kompilierzeiten. Noch deutlicher wird dies wenn viele Systemheader (wie z. B. `windows.h`) eingebunden werden. Da der Header windows.h wieder andere Header einbindet, welche vielleicht wieder andere Header einbinden, entstehen so genannten Include-Bäume. Eine Lösung dieses Problems stellen Precompiled Header dar. Der Compiler sucht dabei im Quelltext nach Headerdateien. Er sucht dabei nach einer Menge von Headern die vorkompiliert werden können. Dies sind bei weiten nicht alle Header. Um dabei ein guten Ergebnis zu erreichen sind viele Einschränkungen zu beachten (insbes. bei gcc):

- Die Reihenfolge der #include Anweisung ist wichtig! Wenn Header in verschiedenen Reihenfolgen eingebunden werden können die Header nicht vorkompiliert werden.
- Nur ein Precompiled Header möglich

- Keine C(++) Token vor der #include Anweisung

Erst ab gcc Version 3.4 wird es eine bessere Unterstützung von precompiled Headern geben.

Im Microsoft Visual Studio besteht die Möglichkeit sich selber Precompiled Headern zu bauen. Die Methode wird von vielen Programmierern wegen der besseren Kontrolle bevorzugt. Die Erstellung eines solchen Precompiled Header erfolgt dabei analog zur automatischen Generierung: Wir stellen uns vor wir haben folgendes Programm:

main.cpp

```
#include <iostream>
#include <map>
#include <vector>

int main(){

  //.....CODE

}
```

Wir wollen die Übersetzungszeiten verkürzen indem wir iostream, map und vector vorkompilieren möchten.Wir bauen uns dafür folgenden Header

precompiled_header.h

```
#pragma once

#include <iostream>
#include <map>
#include <vector>
```

Danach ändern wir unser Hauptprogramm dahingehend ab das wir nur noch unseren Precompiled Header einbinden.

main.cpp

```
#include "precompiled_header.h"

int main(){

  //.....CODE

}
```

Nun muss noch unter den Einstellungen in MS Visual Studion der Precompiled Header angegeben werden. Nähere Informationen dazu gibt es auf der MSDN Homepage (Link im Anhang).

9 Vorwärtsdeklarationen und Header-Abhängigkeiten

[GotW #7] [GotW #34]

9.1 Ausgangslage

Beispiel: Klasse Person

```
class Person {

public:
   Person(const string& name,
          const Date& birthday);
   virtual ~Person();
   std::ostream& print( std::ostream& ) const;

private:
   //Implementationsdetails
   string name;
   Date birthdate;
};
```

Woher erhält der Compiler die Informationen über den Aufbau von *Date*, *String* und *Ostream*?

9.2 #include

Der Compiler benötigt Zugriff auf die Klassen mit denen die Klasse Person implementiert wurde. Es handelt sich dabei um *String*, *Ostream* und *Date*. Den Zugriff auf diese Informationen erlangt man durch `#include`, da `#include file` die Zeile, in der es aufgerufen wird, durch den Inhalt der Datei ersetzt.

Wie diese Ersetzung vor sich geht, sieht man sehr schön anhand der Datei, die der Preprozessor erzeugt:

preprocessed file `a.ii`

```
# 1 "a.h" 1
# 1 "b.h" 1
class B
{
```

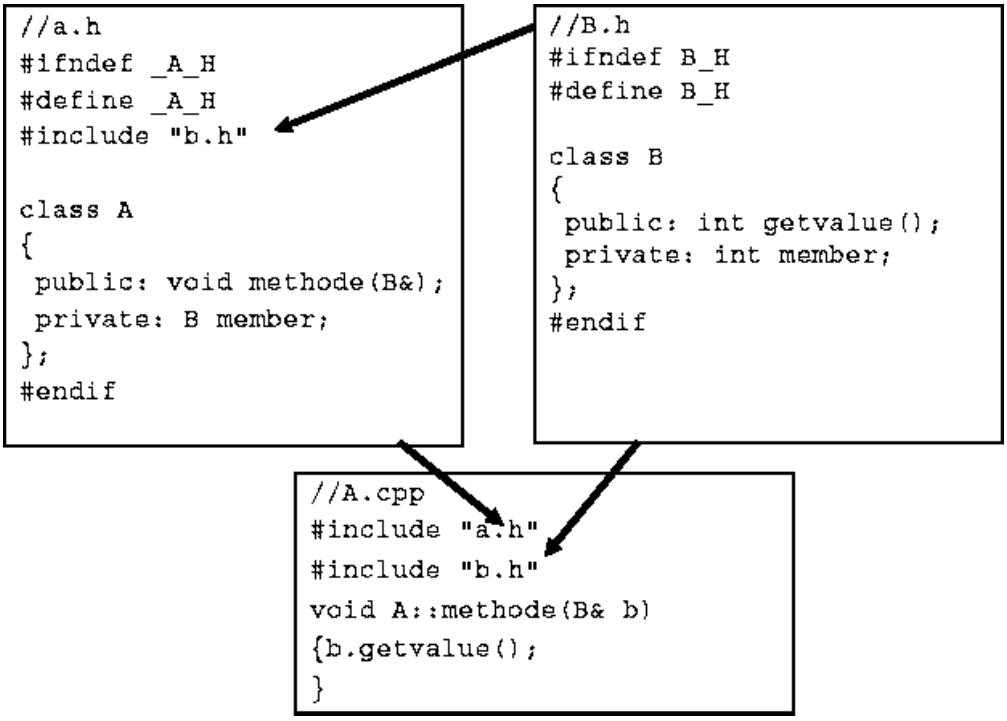


Abbildung 6: Einfügen von Headerdateien

```
public: int getvalue();
private: int member;
};
# 6 "a.h" 2
class A
{
public: void methode(B&);
private: B member;
};
# 4 "a.cpp" 2
void A::methode(B& b)
{b.getvalue();}
```

9.2.1 inclusion-guards

Beim Einbinden von mehreren Headerdateien kann es nun dazu kommen, dass ein Header, der Klassendefinitionen oder Inline-Funktionen enthält, mehrfach in dieselbe Übersetzungseinheit per `#include` eingefügt wird. Wie die One-Definition Rule besagt, würden diese Mehrfachdefinition zu Fehlern führen. Ein weiteres Beispiel, in dem ein solches mehrfaches Einfügen von Headerdateien auch erhebliche Fehler mit sich führt, zeigt folgendes Schema:

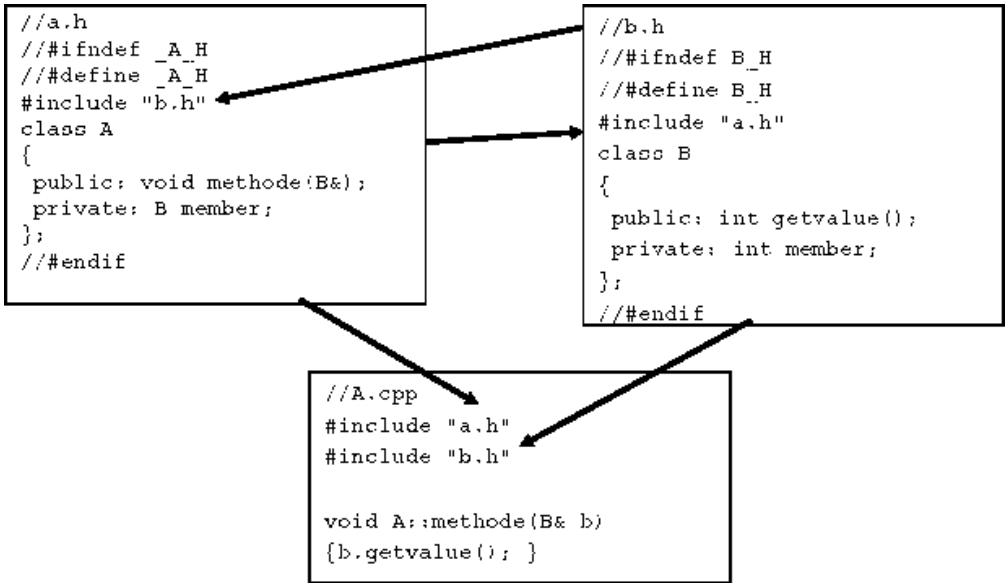


Abbildung 7: Verschachtelte Abhängigkeiten

Wie man sieht, ist hierbei das Problem, dass sich beide Headerdateien rekursiv immer wieder selbst einbinden. Der Header `a.h` bindet `b.h` und `b.h` bindet `a.h` ein. Das bedeutet, dass man beim Einlesen einer der beiden Dateien nie über die Zeile in der `#include "a.h"` oder `#include "b.h"` hinauskommen würde.

preprocessed file `a.ii`

```
# 1 "a.h" 1
# 1 "b.h" 1
# 1 "a.h" 1
# 1 "b.h" 1
........
# 1 "a.h" 1
class A
{
public: void methode(B&);
private: B member;
};
# 6 "b.h" 2
class B
{
public: int getvalue();
private: int member;
};
# 6 "a.h" 2
class A
```

```
{
public: void methode(B&);
private: B member;
};
# 6 "b.h" 2
class B
{
public: int getvalue();
private: int member;
};
# 6 "a.h" 2
...............
class A
{
public: void methode(B&);
private: B member;
};
# 6 "b.h" 2
class B
{
public: int getvalue();
private: int member;
};
# 5 "a.cpp" 2
void A::methode(B& b)
{b.getvalue(); }
```

Um das wiederholte Einfügen von Headerdateien zu verhindern, benutzt man einen Include-Wächter.

Inclusion-guard

```
//a.h
#ifndef _A_H
#define _A_H
#include "b.h"
class A
{
public: void methode(B&);
private: B member;
};
#endif
```

Der Inclusion-Guard besteht aus `#ifndef _A_H`, `#define _A_H` und `#endif`. Sollte `_A_H` nicht definiert sein, bekommt es einen Wert zugewiesen. Ist `_A_H` aber definiert, wird der Inhalt zwischen `#ifndef` und `#endif` vom Compiler ignoriert. Das bedeutet, dass wenn während der Übersetzung die Datei `a.h` zum erstenmal betrachtet wird, `_A_H` einen Wert

erhält, was zur Folge hat, dass, sollte dem Compiler `a.h` während dieser Übersetzung ein weiteres Mal zu lesen gegeben werden, der Inhalt von `a.h` ignoriert wird.

Wenn wir unsere Beispielklasse Person betrachten, ist klar, dass mit `#include` nun folgende Dateien eingbunden werden müssen: `string`, `ostream` und `date.h`. Sie muss also folgendermaßen aussehen:

Beispiel: Klasse Person

```
#include <string>
#include <ostream>
#include "date.h"
class Person {
public:
Person(const string& name, const Date& birthday);
virtual ~Person();
std::ostream& print( std::ostream& ) const;
//.....
private:
string name; //Implementationsdetails
Date birthdate;
);
```

Es entstehen Kompilationsabhängigkeiten, da eine Änderung an der Implementation der eingebunden Klassen, das erneute Kompilieren der Datei und sämtlicher Dateien, die diese wiederum einbinden, erfordert.

9.3 Reduzieren von Headerabhängigkeiten

Das Problem ist, dass mit `#include` oftmals mehr Dateien eingebunden werden als notwendig, was einen ernsthaften Einfluss auf die Buildzeit hat. Dieser Einfluss macht sich besonders bemerkbar, wenn ein oft genutzter Header eine Vielzahl von anderen Dateien einbindet.

Man hat die Möglichkeit Headerabhängigkeiten zu reduzieren, indem man Forward-Declarations verwendet. Dadurch dass Returnvalues und Parameter nur *forward-declared* sein müssen, hat man die Möglichkeit auf das Einfügen von Dateien, die ihre Definitionen enthalten zu verzichten. Auf unsere Beispielklasse bezogen, würde das, ohne die Implementtionsdetails zu beachten, bedeuten, dass wir *String*, *Date* und *Ostream* vorwärtsdeklarieren sollten, da es sich dabei nur um Parameter und Rückgabewerte von Memberfunctions handelt. Die Frage ist nun aber, ob sich *String* und *Ostream* so einfach *forward-deklarieren* lassen. Denn bei beiden handelt es sich nicht einfach um Klassen, wie z. B. bei *Date*, sondern um *Typedefs* von Templates. Da *Typedefs* von Templates nicht vorwärtsdeklariert werden dürfen, ist folgendes also nicht erlaubt:

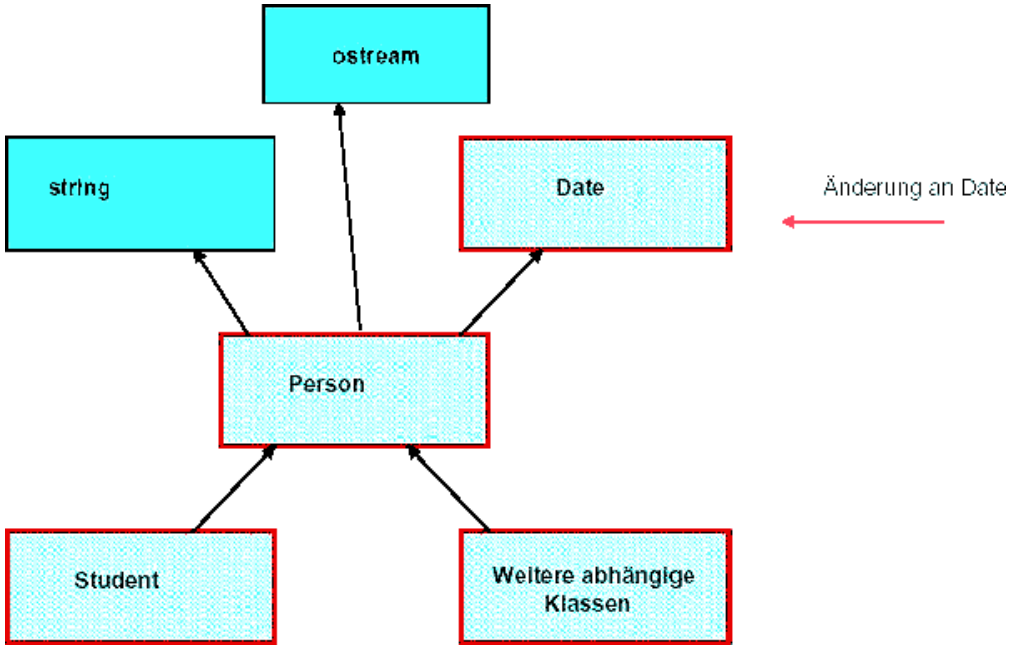


Abbildung 8: Kompilationsabhängigkeiten

```
class string;   //error
class ostream;  //error
```

Dennnoch gibt es eine Möglichkeit auf *Forward-Declarations* von *Ostream* zuzugreifen, denn `iosfwd` enthält die Deklarationen der I/O-Stream-Klassen und -Templates und die dazugehörigen standardisierten Typdefinitionen. Die Klasse *Person* würde nun folgendermaßen aussehen:

Beispiel: Klasse Person

```
#include <string>
#include <iosfwd>

class Date;
class Person {
public:
   Person(const string& name, const Date& birthday);
   virtual ~Person();
   std::ostream& print( std::ostream& ) const;
   //.....
);
```

Um die Forwarddeclarations verwenden zu können, müssen wir die Implementionsde-

tails später spezifieren. Das Problem dabei ist aber, woher soll der Compiler nun wissen, wie groß ein Objekt der Klasse Person ist. Da `sizeof(Person)` unbekannt ist, kann ausschließlich ein Zeiger auf ein Objekt der oben aufgeführten Klasse verwendet werden. Es ist natürlich problematisch eine Klasse zu definieren, von der man ausschließlich Zeiger erzeugen kann. Um Implementaionsdetails zu verstecken und dadurch Abhängigkeiten zu reduzieren, verwendet man Pimpl's. Ein sogennanter Pointer to Implementation ersetzt die Implementionsdetails durch Pointer auf einen vorwärtsdeklarierten Type von einem Implementationsobjekt. Somit muss der Klasse *Person* die Größe eines Objektes der Klasse *Date* nicht bekannt sein.

Pointer to Implementation

```
#include <string>
#include <ostream>
class Date;
class PersonImpl; //beinhaltet Implementaionsdetails

class Person {
public:
   Person(const string& name, const Date& birthday);
   virtual ~Person();
   std::ostream& print( std::ostream& ) const;
   //.....
private:
   PersonImpl *impl; //Zeiger auf eigentliche
                     //Implementation
);
```

Eine Änderung an der Implemention der Klasse *Date* betrifft die Klasse *Person* nun nicht mehr.

Um Headerabhängigkeiten zu vermeiden, sollte man also darauf achten, dass man die Verwendung von Objekten vermeidet, wenn stattdessen auch Referenzen oder Zeiger anwendbar sind. Weiterhin sollte man, falls es sich um Returnvalues oder Paramerter handelt, Klasssendeklarationen anstelle von -definitionen verwenden.

```
class Date;  //  Klassendeklartion
Date getDate();
void setDate(Date d);
```

Außerdem sollte man auch in Headerdateien darauf achten, keine unnötigen weiteren Headerdateien einzufügen, und auch hier soweit es möglich ist mit Vorwärtsdeklarationen

zu arbeiten.

9.4 Ansätze

Natürlich ist nun die Frage wie man das Verstecken der Implementationsdetails realisiert. Hierfür könnte man z. B. Handle-Klassen oder Protokollklassen verwenden.

9.4.1 Handleklassen

Handleklassen werden auch Envelopeklassen genannt, da sie nur Zeiger auf nicht näher spezifizierte Implementaionsdetails enthalten. Die Handleklassen leiten die Aufrufe ihrer Funktionen weiter. Ein Konstruktor könnte z. B. folgendermaßen aussehen:

```
#include <Person.h>
#include <Personimpl.h>
Person::Person(const string& name,
               const Date& birthday)
{
   Impl = new PersonImpl(name, birthday);
}
```

Es ändert sich also nicht die Funktionalität, sondern nur der Ort des Ausführens. Natürlich hat man einen zusätzlichen Aufwand mit den Handleklassen, da man zusätzlichen Speicher für den Zeiger benötigt und da für den Zugriff der Zeiger dereferenziert werden muss. Hinzu kommt die Initialisierung des Zeigers und die Freigabe des Speichers.

9.4.2 Protokollklassen

Bei Protokollklassen handelt es sich um abstrakte Klassen, sie enthalten keine Implementation, sondern spezifizieren eine Schnittstelle für abgeleite Klassen. Sie enthalten damit keine Konstruktoren oder Datenelemente.

```
class Person{
public:
   virtual ~Person();
   virtual string name()const=0;
   virtual string birthdate() const=0;
};
```

Dadurch müssen Benutzer dieser Klasse Zeiger verwenden und Objekte über *Factory*-Funktionen instanzieren.

```
Person* makePerson(const string& name,
                   const Date& birth);
```

Die von der Protokollklasse abgeleite konkrete Klasse enthält die Implementation.

konkrete Klasse

```
class RealPerson: public Person{
public:
   RealPerson(const string& name, const Date& birth):
       name_(name), birth_(birth)
       {}
   virtual ~RealPerson(){}
//...
private:
   string name_;
   Date birth_;
};
```

9.5 Weitere Headerdependencies

Ein weiteres Problem, das man mit *Forward-Declarations* lösen kann, wird durch Klassen erzeugt, die jeweils die Elemente der anderen Klasse benutzen.

Zwei Klassen...

```
//Datei A.h
#ifndef A_h
#define A_h
#include B.h
//..


//Datei B.h
#ifndef B_h
#define B_h
#include "A.h"
//..

#endif
```

Wenn man davon ausgeht, dass `A.h` zuerst eingelesen wird, ist klar, dass die 3. Zeile durch den Inhalt von `B.h` ersetzt wird. Doch wird nun der Inhalt von `B.h` verarbeitet, kommt es in der 3. Zeile von `B.h` zu einem Problem, da an dieser Stelle wieder `A.h` einglesen wird, `A_h` aber schon definiert wurde.

Die Lösung für dieses Problem wäre die Vorwärtsdeklaration der Klassen A und B.

Lösung: Klasse A

```
//Datei A.h
#ifndef A_h
#define A_h
class B;
// Vorwärtsdeklartation
class A{
private:
   B* element;
};
#endif
```

Lösung: Klasse B

```
//Datei B.h
#ifndef B_h
#define B_h
class A;
//Vorwaertsdeklaration
class B{
public:
   void f(&A)
};
#endif
```

Somit können beide Klassen Elemente der anderen Klasse für ihre Implementation verwenden.

10 IOStreams und Objekt-spezifische Ausgabe

10.1 Einleitung

Unter Linux/Unix existiert mit dem Pipe Mechanismus ein einfaches und flexibles Werkzeug, um die beiden Standard-Ausgabekanäle *stdout* und *stderr* eines Programmes „umzuleiten“; häufig wird dies zum Beispiel genutzt um die Ausgabe eines Programms in Log-Dateien zu speichern.

Allerdings sind die Möglichkeiten des Pipe-Mechanismus für eine allgemeine Verwendung mit C++-Programmen beschränkt:

- Er steht unter Windows nicht zur Verfügung.
- Es ist keine direkte Manipulation der Ausgabe möglich.
- Veränderungen sind nur pro Kanal (*stdout, stderr*) möglich.

Eine Veränderung des Ziels der Ausgabe ist jedoch häufig nicht nur pro Kanal wünschenswert. Verwendet ein Projekt zum Beispiel Programmbibliotheken, so ist es oft sinnvoll die Ausgaben, der aus der Bibliothek genutzten Klassen zu unterdrücken. Diese Informationen sind im Projekt meist irrelevant und erhöhen unnötig die Unübersichtlichkeit der Ausgabe. In Ausnahmefällen, wie im unteren Beispiel einer interaktiven Konsole, können diese Ausgaben sogar die Nutzbarkeit des gesamten Projektes erheblich eingeschränken.

```
Avango 'aview' Version 0.9.7 (distributed)
av-elk> VRPN Warning
(3) from atmel0: No response from server for >= 3 seconds
VRPN Warning
(4) from atmel0: No response from server for >= 3 seconds
VRPN Warning
(5) from atmel0: No response from server for >= 3 seconds
VRPN Warning
```

Die Ausgaben von lediglich benutzten Klassen sind allerdings auch oft hilfreich, insbesondere bei der Fehlersuche, und bieten darüber hinaus in multi-threading/multi-processor Anwendungen meist die einzige Möglichkeit, einen Fehler überhaupt zu lokalisieren, da Debuggen bei diesen Applikationen oft die gesamte Verschachtelung verändert, so dass ein Fehler dann nicht mehr auftritt.

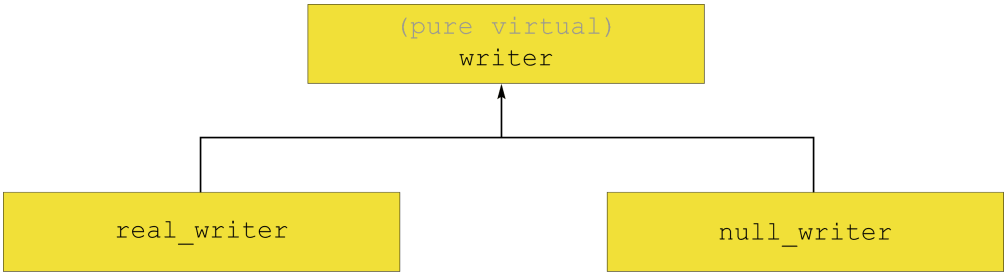


Abbildung 9: Trennung von Interface und Implementation

Um diese Probleme zu vermeiden, werden deshalb Strategien und Methoden benötigt, die es erlauben, die Ausgabe für jede Instanz und jede Klasse einzeln zu definieren.

Der direkte Ansatz dafür wäre die Veränderung der verwendeten Klassen. Dies ist im allgemeinen Fall jedoch weder möglich noch sinnvoll:

- der Quellcode steht oft nicht zur Verfügung, zum Beispiel bei der Verwendung von Bibliotheken
- die angestrebte Abstraktion kann verloren gehen, insbesondere bei großen Klassen.

Für die instanz- und klassenspezifische Kontrolle über die Ausgabe werden deshalb Interface und Transportschicht getrennt. Das Interface, die so genannte Formatierungsschicht, ist dabei die Schnittstelle für Objekte, welche die Ausgabe benutzen wollen, während die Transportschicht für die Weiterleitung der Daten an ein Gerät oder an ein Objekt außerhalb des Programms verantwortlich ist.

Klassen, welche möglicherweise in einer Bibliothek oder einem größeren Framework zum Einsatz kommen, können damit über die Formatierungsschicht alle eventuell benötigten Informationen zur Verfügung stellen. Das tatsächliche Ausgabeverhalten wird dann erst bei der Verwendung einer Klasse über die Transportschicht festgelegt und kann den jeweiligen Anforderungen angepasst werden.

10.2 Objekt-orientierter Ansatz

10.2.1 Prinzip

Die einfachste Lösung für die Trennung von Interface und Implementation in C++ ist die Benutzung einer abstrakten Interface-Basisklasse (Abbildung 9). Die von dieser Basisklasse *writer* abgeleiteten Klassen stellen dabei die verschiedenen möglichen Ausgabe-

```
class lib_class {

public:

   void set_writer(writer* w);
   ...

private:

   writer* local_writer;
   ...
}
```

```
int
main( int argc,
          char* argv[]) {

   // this is release version
   // -> no output
   writer* w = new null_writer();

   lib_class* l = new lib_class();
   l->set_writer( w );

   // do sth. useful
}
```

Abbildung 10: Anwendung der `writer`-Klasse

Modi zur Verfügung.

In einer Bibliothek oder Anwendung besitzt jede Klasse, welche den Ausgabemechanismus unterstützt, einen Zeiger von *writer* als Membervariable. Dieser wird dann je nach Anforderungen, mit einer Instanz einer konkreten Klasse initialisiert (Abbildung 10).

Dabei sind zwei Anforderungen notwendig um den allgemeinen Einsatz zu ermöglichen:

1. alle Klassen, die möglicherweise in einer Bibliothek oder einem Framework Verwendung finden, müssen einen Zeiger auf die *writer*-Basisklasse implementieren

2. alle Bibliotheken müssen die identische *writer*-Basisklasse verwenden.

10.2.2 Alternativen

Diese beiden Anforderungen können in der Praxis jedoch nicht gewährleistet werden. Deshalb – und um einige andere Probleme in Bezug auf die Ein- und Ausgabe zu lösen – ist bereits in den C++ Standard Ein- und Ausgabeoperationen eine Trennung von Formatierungs- und Transportschicht implementiert: der *Stream Mechanismus*.

10.3 IOStreams in C++

10.3.1 Einführung

IOStreams wurden 1984 von Bjarne Stroustrup erstmals in C++ vorgestellt und 1989 von Jerry Schwarz grundlegend überarbeitet. Die aktuelle Template-basierte Implemen-

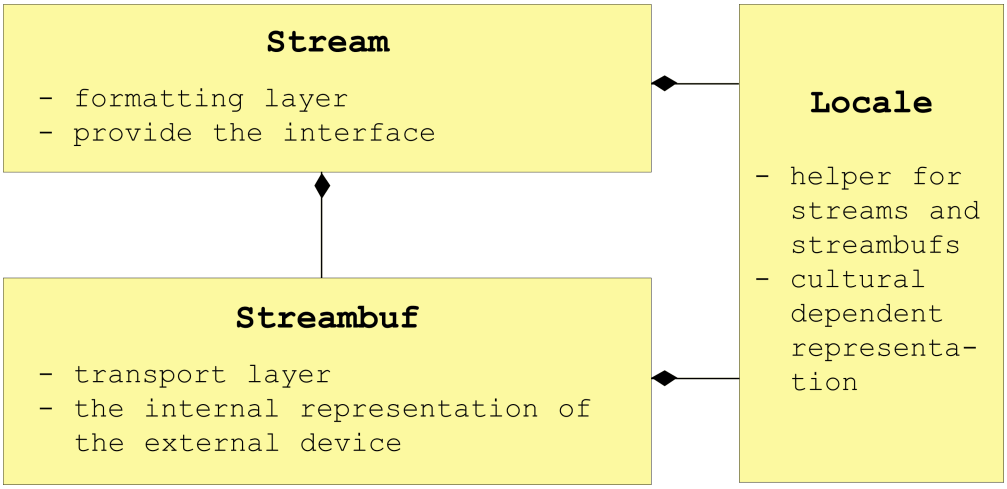
tierung entstand während der C++ Standardisierung in den 90iger Jahren. Seitdem sind die *IOStreams* ein integraler Bestandteil der C++ Standard Bibliothek.

Auf Grund der Komplexität des *Stream Mechanismus* in C++ kann an dieser Stelle nur ein kurzer Überblick gegeben werden. Einen umfassenden Einblick vermittelt [Langer'00].

IOStreams wurden primär zur Ein- und Ausgabe von Buchstaben (`char_t`, `wchar_t`) entwickelt. Seit der Implementation mit Hilfe von Templates ist jedoch auch ein Einsatz mit beliebigen Zeichen möglich. Für die gebräulichen Buchstabenrepräsentationen existieren dabei default Instantiierungen.

```
typedef basic_iostream<char> iostream;
typedef basic_iostream<wchar_t> wiostream;
```

Der Stream Mechanismus in C++ besteht aus drei Teilen:



Streams sind dabei die Formatierungsschicht und das, aus der Verwendung durch `std::cout`, `std::cerr` und `std::cin` bekannte Interface.

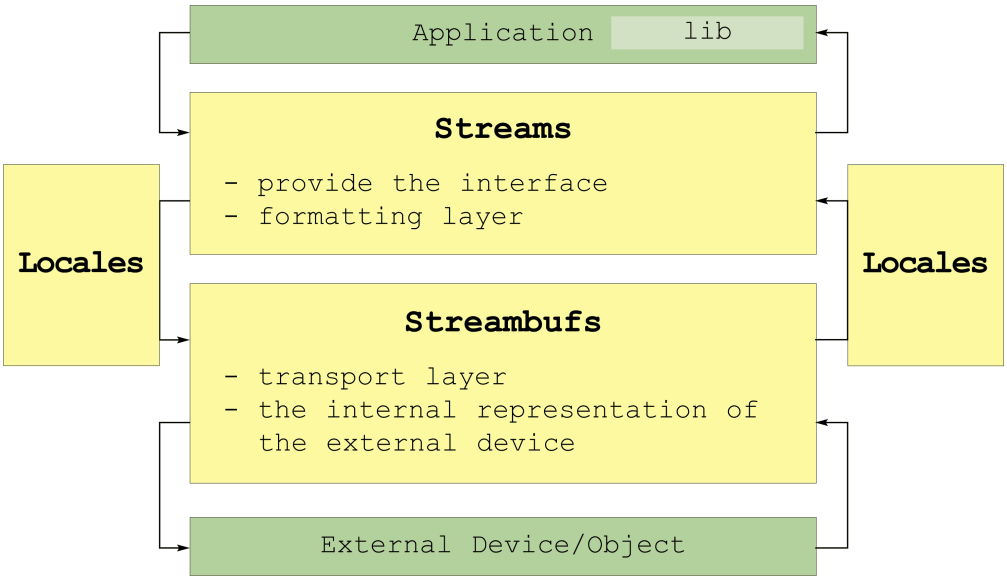
Die Transportschicht wird durch die *streambufs* implementiert. Um eine hohe Effizienz zu gewährleisten, arbeiten *streambufs* auf einem internen Buffer, so dass die Datenübergabe nach Außen nur erfolgen muss, wenn in diesem kein freies Element mehr vorhanden ist.

Locales sind die Integration des gleichnamigen Konzepts aus C in den C++ *Stream Mechanismus*. Sie ermöglichen es, die Ein- und Ausgabe an lokale bzw. kulturelle Normen anzupassen; zum Beispiel kann mit den *locales* erreicht werden, dass trotz der Übergabe der identischen Zahl 10.000 an `std::cout` die Ausgabe des Tausendertrennzeichens in Deutschland ein Punkt und in den USA ein Komma ist.

```
#include <iostream>
```

```
#include <locale>

int
main( int argc, char* argv[]) {

  int number = 10000;

  // we are in the US
  std::cout.imbue( std::locale("en_US"));
  std::cout << number << '\n';
  // returns 10,000

  // and right back in Germany
  std::cout.imbue( std::locale("German"));
  std::cout << number << '\n';
  // returns 10.000
}
```

Locales können dabei als Filter zwischen *streams* und *streambufs* verstanden werden.



Bei der Ausgabe werden die Daten an das Gebietsschema angepasst – lokalisiert – und bei der Eingabe werden Daten aus einem gebietsspezifischen Format in die einheitliche C/C++ Repräsentation konvertiert. Die Umwandlung wird dabei jeweils von den *facets* durchgeführt, aus denen *locales* zusammengesetzt sind. Eine *facet* ist jeweils für einen Aspekt der Lokalisierung verantwortlich; so existieren *facets* für Geld, Zeit, die Ausgabe von Zahlen (wo auch das Tausendertrennzeichen definiert wird) und andere Bereiche. Dabei kann der Nutzer bei Bedarf eigene *locales* aus den vorhandenen *facets* zusammenstellen und neue *facets* definieren. Durch diese Flexibilität sind *locales* ein weit über den ursrprünglichen Verwendungszweck hinaus nutzbares Werkzeug.

Die Interaktion zwischen *streams*, *streambufs* und *locales* soll an folgendem `std::cout` Aufruf, kurz erläutert werden.

```
int my_int = 10000;
std::cout << my_int;
```

Zunächst wird die, für den Argumenttyp – hier integer – passende überladenen Variante des `ouput stream operator <<` von `std::basic_ostream` aufgerufen.

```
basic_ostream::operator<<(int __n)
```

Der `output stream operator<<()` ruft das *locale*, im Beispiel für „German“, auf und spricht das *numput facet* an, welche für die korrekte Formatierung von Zahlen verantwortlich ist. Welche *facet* angesprochen werden muss, ist für jede überladene Variante von `basic_ostream::operator<<()` in der Funktionsdefinition definiert.

```
_M_fnumput->put(*this, *this, this->fill(), __n).failed()
```

Nach der Anpassung der Zeichenkette schreibt die *facet* den String in den internen Buffer des *streambufs*, welcher für die Ausgabe verantwortlich ist.

```
_M_sbuf->sputn(__ws, __len)
```

10.3.2 Objekt-spezifische Ausgabe

Aus dem *IOStream Mechanismus* der C++ Standard Bibliothek ergeben sich mehrere Möglichkeiten die Ausgabe Objekt-spezifisch festzulegen.

Die Ausgabe wird durch die Anwendung gesteuert

Erfolgt die Steuerung der Ausgabe explizit durch die Anwendung resp. durch den Nutzer, so muss die Klasse dazu eine Funktion zur Verfügung stellen.

Zwei Implementationen sind dabei möglich:

1. Es wird eine Funktion verwendet die als Rückgabewert einen String hat. Dieser enthält die Ausgabe und kann außerhalb der Klasse weiterverarbeitet beziehungsweise einem *stream* zur Verfügung gestellt werden.

2. Die verwendete Funktion erhält beim Aufruf als Parameter ein *stream*, welchen die Klasse zur Ausgabe nutzt.

Bei der expliziten Steuerung jeder Ausgabeoperation ist es sehr einfach möglich, für jede Ausgabe – und damit auch für jede Instanz oder Klasse – einen geeigneten *IOStream* zu verwenden. Allerdings kann dieser Aufwand, jede Ausgabeoperation explizit durchführen zu müssen, sowohl für den Nutzer als auch die Applikation einen nicht unerheblichen Overhead bedeuten.

Bei der Steuerung der Ausgabe durch den Nutzer wird meist der Ausgabeoperator `operator<<()` mit der verwendeten Klasse überladen. Dadurch kann eine Instanz der Klasse direkt als Funktionsargument für den Operator verwendet werden.

```
#include <iosfwd>

...

// global namespace
// overloaded output-stream-operator for class 'object'
// usage example: std::cout << my_object << '\n';
template<class _CharT, class _Traits>
std::basic_ostream<_CharT, _Traits>&
operator <<( std::basic_ostream<_CharT, _Traits> & os,
             const object_3d & obj) {

 // check if the stream is valid
 if (std::basic_ostream<_CharT, _Traits>::sentry(os)) {

   // get the output string from the class
   os << obj.print_this();
 }

  return os;
}
```

Dabei sollte an Stelle des `iostream`-Headers der `iosfwd`-Header verwendet werden. In diesem erfolgen alle notwendigen Deklarationen *forward*, so dass der, auf Grund der Templatisierung enorme Overhead des `iostream`-Headers vermieden wird.

Im Beispiel wird die Template Klasse `std::basic_ostream` für den *stream* verwendet, womit sichergestellt ist, dass beliebige Zeichenstreams mit dieser Funktion verwendet werden können.

Es ist außerdem sinnvoll, den überladenen `output stream operator<<()` als `friend` zur verwendeten Klasse zu definieren, da dadurch die verwendete Memberfunktion `private` sein kann.

Verwendung von `std::cout` oder `std::cerr`

Werden in der Klasse für die Ausgabe die Standard-*streams* `std::cout` und/oder `std::cerr` verwendet, so kann der Nutzer der Klasse die Ausgabe durch den Austausch der *streambufs* für `std::cout` und/oder `std::cerr` kontrollieren. Dieser Ansatz kann damit auch bei einfachen Klassen verwendet werden, welche keine zusätzliche Funktionalität zur Kontrolle der Ausgabe implementieren. Eine Instanz- oder Klassen-spezifische Ausgabe ist allerdings nur mit erheblichen Aufwand möglich.

```
// replace cout's default streambuf by a null streambuf
// -> avoid all output from cout
streambuf_null null_buf;
std::cout.rdbuf( &null_buf);
```

Sink Repository

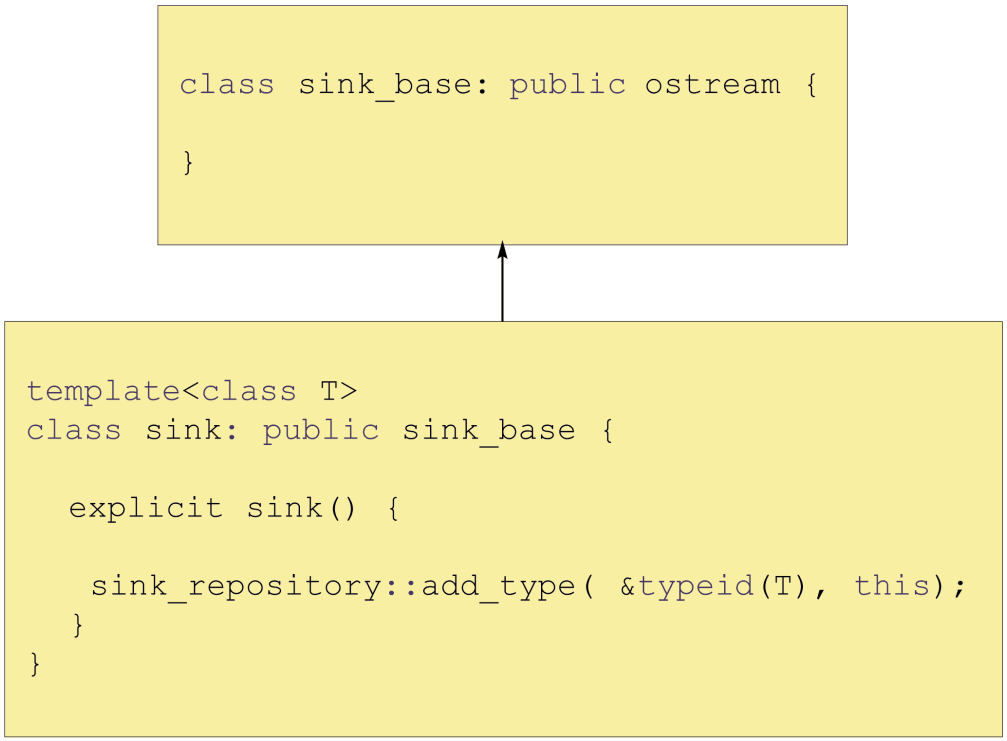
Sink repositories bieten eine sehr weitreichende Kontrolle über die Ausgabe, welche pro Klasse, oder bei Bedarf auch pro Instanz festgelegt werden kann. Allerdings ist der Implementationsaufwand für *sink repositories* erheblich und muss von Anfang an bei der Erstellung der Klassen berücksichtigt werden.

Bei der Verwendung eines *sink repositories* ist es notwendig, dass jede Instanz einer beliebigen Klasse *object* einen eigenen *stream* in Form einer Instanz der Template Klasse *sink* besitzt. Diese Klasse ist von `std::ostream` abgeleitet, so dass alle Standard Ausgabe Operatoren auf einer Instanz dieser Klasse benutzt werden können.

```
class object {
...
private:

  sink<object> local_sink;
}
```

Um Sicherzustellen, dass alle Instanzen einer Klasse sich bei der Ausgabe zunächst identisch verhalten, und um die *sinks* möglichst effektiv verwalten zu können wird das *sink repository* verwendet. Das Klassen-spezifische Ausgabeverhalten wird durch die Zuweisung des identischen *streambufs* zu allen Instanzen einer Klasse bei der Konstruktion erreicht. Die Standard *sink* für eine Klasse wird dabei in einer *map* `sink_map` mit der `typeid` der Klasse als *key* gespeichert.

Die Klasse *sink* wird von *object* per value gehalten. Dadurch kann sichergestellt werden, dass der Konstruktor von *sink* während der Konstruktion von *object* aufgerufen wird und damit die Funktion `add_type()` ausgeführt wird, welche gewährleistet, dass alle Instanzen von *object* nach der Konstruktion zunächst den identischen *streambuf* verwenden. In



```
class sink_base: public ostream {

}
```

```
template<class T>
class sink: public sink_base {

  explicit sink() {

   sink_repository::add_type( &typeid(T), this);
  }
}
```

```
namespace sink_repository {

  bool add_type(...);
  bool remove_type(...);
  bool enable(...);
  bool disable(...);
}

// unnamed namespace
namespace {

 ...

 map< const type_info*, sink_base* > sink_map;

}
```

`add_type()` wird dazu zunnächst überprüft, ob `typeid` von *object* bereits in `sink_map` vorhanden ist. Existiert noch kein Eintrag, so ist dies die erste Instanz `inst0` welche von

object erzeugt wird und der Eintrag für *object* wird der *map* hinzugefügt. Die *sink* von `inst0` wird dabei als Standard-*sink* in der *map* referenziert. Wird eine weitere Instanz `inst1` von *object* erzeugt, so wird erneut `add_type()` aufgerufen. Da `typeid` Klassen- und nicht Instanz-spezifisch ist, wird in `add_type()` der bereits existierenden Eintrag für *object* in `sink_map` gefunden. Der *streambuf* der *sink* von `inst1` wird dann durch den der in der *map* referenzierten *sink* ersetzt.

Mit den weiteren Funktionen des *sink repositories* ist es möglich, die Ausgabe zu kontrollieren; so kann zum Beispiel durch `disable(typeid*)` der *streambuf* für eine Klasse durch den Null-Buffer ersetzt werden.

10.3.3 Streambufs zur Manipulation von Ausgabe IOStreams

Mit *streambufs* ist es nicht nur möglich das Ziel einer Ausgabe festzulegen, durch die Verwendung von zwei *streambufs* können auch die vom *stream* übergebenen Daten verändert werden. Dabei wird der so genannte *filter-streambuf* zur Filterung der Daten verwendet, während der *output-streambuf* die tatsächliche Ausgabe realisiert. Beide werden in einem *filtering streambuf*-Objekt zusammengefasst und können in zwei Kategorien unterteilt werden:

- *filtering streambufs for output*
- *filtering streambufs for input*

Zusätzlich können sie auch bezüglich ihrer Manipulation klassifiziert werden:

- *inserter*, welche den übergebenen Daten Zeichen hinzufügen
- *extractor*, welche aus den übergebenen Daten Zeichen entfernen

Ein bekanntes Beispiel für einen *input-extractor* sind Quellcode Parser, welche alle Kommentare und Leerräume aus dem an sie übergebenen Quellkode entfernen

Im Folgenden wird die Implementation eines generischen *output-inserter* vorgestellt. Bei der Instantiierung wird diesem als Template Argument ein Funktor Objekt übergeben, welches definiert, unter welchen Bedingungen zusätzliche Zeichen eingefügt werden sollen.

```
// generic output-inserter
// inserts chars defined by 'Inserter' in an existing
// stream
template<class Inserter>
class Streambuf_Inserter: public std::streambuf {
```

```
  ...

  // override relevant functions from base class
  int overflow(int ch) {

    return _inserter(final_dest, ch);
  }

private:

  std::streambuf* _final_dest;

  Inserter        _inserter;
};
```

Im Beispiel-Funktor `Prefix_Date` wird jeder Zeile als Prefix der aktueller Timestamp hinzugefügt.

```
//functor for Streambuf_Inserter
// adding current time and date
//as prefix for every line
class Prefix_Date {
public:

  Prefix_Date();

  ~Prefix_Date();

public:

  int operator()(std::streambuf* final_dest, int ch);

private:

  bool _insert_now;
};
```

Der *output-streambuf*, welcher vom *filter-streambuf* auf Grund der Verarbeitungsreihenfolge als Membervariable gehalten wird, muss für die gewünschte Funktionalität nicht modifiziert werden. Er kann deshalb mit einem Standard *output-stream*, zum Beispiel dem vom `std::cout`, initialisiert werden.

Der *filter-streambuf*, im Beispiel die Klasse `Streambuf_Inserter`, wird von der *streambuf*-Basisklasse `std::streambuf` abgeleitet. Damit erhält er die standardmäßige *streambuf*-Funktionalität und kann mit einem Standard *stream* wie `std::cout` benutzt werden.

Als interne Bufferlänge des `Streambuf_Inserter` wird Null gewählt. Dadurch ist es möglicht, an beliebigen Stellen Zeichen einzufügen, da die überschriebene Funktion `overflow(int ch)`, welche den Funktor ausführt, bei jedem Zeichen aufgerufen wird, welches dem `Streambuf_Inserter` vom verwendeten *stream* übergebenen. Dies erfolgt durch die vererbten Mechanismen von `std::streambuf`, welche gewährleisten, dass `overflow(int ch)` ausgeführt wird, wenn der interne Buffer des *streambufs* keine freien Elemente mehr enthält.

Im Beispiel erfolgt der Aufruf des Funktors durch den `operator()`. Dieser erhält als Parameter sowohl das vom *stream* übergebene Zeichen `ch` als auch den *output-streambuf*. Im `operator()` wird zunächst überprüft, ob `ch` die zum Einfügen notwendigen Bedingungen erfüllt. Ist dies der Fall, so werden die zusätzlichen Zeichen und `ch` an den *output-streambuf* übergeben; ist die Einfüge-Bedingung nicht erfüllt, wird nur `ch` an den *output-streambuf* übergeben.

Im Beispiel-Funktor `Prefix_Date` wird jeweils überprüft, ob das übergebene Zeichen ein Zeilenvorschub `'\n'` ist und die Membervariable `insert_now` in diesem Fall auf `true` gesetzt. Beim nächsten vom *stream* übergebenen Zeichen, dem ersten der neuen Zeile, wird dann zunächst das aktuelle Datum an den *output-streambuf* übergeben, bevor das Zeichen `ch` selbst weitergeleitet wird.

Sinnvoll ist es auch einen eigenen *stream* für den `Streambuf_Inserter` zu definieren. Wird dieser sowohl von `std::ostream` als auch vom *filter-streambuf*, im Beispiel `Streambuf_Inserter`, abgeleitet, so kann sichergestellt werden, dass der *stream* immer auf einem validen `Streambuf_Inserter` arbeitet.

```
// class derived from Streambuf_Inserter<Inserter>
// and std::ostream
// the order of the base classes ensures
// correct initialization
template<class Inserter>
class Ostream_Filter:
           private Streambuf_Inserter<Inserter>,
           public std::ostream
{
public:

  Ostream_Filter(std::ostream& dest);

  ~Ostream_Filter();

public:

  Streambuf_Inserter<Inserter>* rdbuf();
};
```

Beispiel

Verwendung des `Streambuf_Inserter` mit `Prefix_Date` als Funktor

```
// create a stream which modify every given line by
// setting current time and date as prefix
// use 'std::cout' as pattern
Ostream_Filter<Prefix_Date> Prefixed_Stream(std::cout);

// create a null-streambuf for deactivating cout
// Prefixed_Stream has to be used
Streambuf_Null Null_Buf;
cout.rdbuf(&Null_Buf);

// cout don't produce any output while it has the
// null-streambuf as streambuf
cout << "This line will not appear on the screen."
     << std::endl;

// this text is displayed with a prefix:
// current time and date
Prefixed_Stream << "example program" << endl;
// Sun May 23 17:08:28 2004:  example program

// display the properties of the cube
Prefixed_Stream << my_cube;
// Sun May 23 17:08:31 2004:  Object_3D::_name = nn
// Sun May 23 17:08:31 2004:  Object_3D::_pivot:
// Sun May 23 17:08:31 2004:  Point_3D::x = 0
// Sun May 23 17:08:31 2004:  Point_3D::y = 0
// Sun May 23 17:08:31 2004:  Point_3D::z = 0
// Sun May 23 17:08:31 2004:  Cube::first
// Sun May 23 17:08:31 2004:  Point_3D::x = 0
// Sun May 23 17:08:31 2004:  Point_3D::y = 0
// Sun May 23 17:08:31 2004:  Point_3D::z = 0
// ...
```

10.4 Zusammenfassung

Die Definition eines Instanz- und Klassenspezifischen Ausgabemechanismus ist nicht trivial und auch mit den *IOStreams* des C++ Standards kann keine optimale Lösung gefunden werden. *Sink repository* bieten zwar die optimale Flexibilität – die alle anderen Ansätze vermissen lassen – jedoch ist der erhebliche Implementationsaufwand nur in seltenen Fällen zu rechtfertigen.

Insgesamt bieten die *IOStreams* in C++ jedoch sehr viele Möglichkeiten, welche – im

Gegensatz zur *STL* – bisher nur selten verwendet werden.

11 Extending namespace std

11.1 C++ Standard

Die Erweiterung von namespace std mit eigenen Klassen, Funktionen und Algorithmen ist verboten, dies ist ganz klar im C++-Standard ([C++ 97], 17.3.4.1, [lib.reserved.names]) spezifiziert:

1. *It is undefined for a C++ program to add declarations or definitions to namespace std or namespaces within namespace std unless otherwise specified.* [...]

Jedoch ist in Template-Umgebungen eine Erweiterung sehr oft notwendig:

```
template <class T> class MyClass {
public:
        MyClass();
        typedef std::numeric_limits<T> limitT;
};
```

Für einfache Datentypen wie z. B. `bool`, `char`, `int`, `float` gibt der Standard schon Template-Spezialisierungen für `numeric_limits` vor. Jedoch müsste man für eigene Datentypen T die Klasse `numeric_limits` selbst spezialisieren. Dies ist auch die einzige erlaubte Form der Eweiterung von namespace std.

ISO C++-Standard ([C++ 97], 17.3.4.1, [lib.reserved.names])

[...] *A program may add template specializations for any standard library template to namespace std. Such a specialization (complete or partial) of a standard library template results in undefined behavior unless the declaration depends on an user-defined name of external linkage and unless the specialization meets the standard library requirements for the original template.*

Templates dürfen also für eigene Datentypen spezialisert werden. Ein einfaches typedef reicht übrigens nicht! Einen kleinen Einblick in die Template-Spezialisierung bietet das nächste Kapitel.

11.2 Spezialisierung von Klassentemplates

Klassentemplates können manuell für eine bestimmte Signatur spezialisiert werden ([Vandevoorde]), diese Spezialisierung wird vom Compiler auch bevorzugt verwendet. Beispiele findet man u. a. bei [Hyperformix] oder [GotW #49].

Die einfachste Form der Template-Spezialisierung ist die vollständige Spezialisierung für einen konkreten Datentyp.

```
template <class T> class MyClass {
public:
        MyClass();
        T malZwo(T value) {
                return(value*2);
        }
};
```

Für `T = unsigned` vollständig spezialisiert:

```
template <> class MyClass<unsigned> {
public:
        MyClass();
        unsigned malZwo(unsigned value) {
                return(value << 1);
        }
};

MyClass<int>        MyI; // compiler generiertes template
MyClass<unsigned>   MyU; // spezialisiertes template
```

Für Klassentemplates mit zwei Template-Parametern kann auch eine partielle Spezialisierung vorgenommen werden.

```
template <class T1, class T2> class MyClass {
public:
        MyClass();
        std::map<T1, T2> mymap;
};
```

Für `T1 = int` partiell spezialisiert:

```
template <class T2> class MyClass<int, T2> {
public:
        MyClass();
        std::map<int, T2> mymap;
};

MyClass<long, float> MyLF;     // compiler gen. template
MyClass<int, float>  MyIF;     // partiell spez. Template
                               // fuer T1 = int
```

Eine partielle Spezialisierung kann sich aber auch auf eine Gruppe von Datentypen beziehen, z. B. Pointer `T*` oder `std::vector<T>`.

```
template <class T> class MyClass {
public:
        MyClass();
        T plusEins(T value) {
                return(value + 1);
        }
};
```

Partielle Spezialisierung für Pointer, `T = T*`:

```
template <class T> class MyClass {
public:
        MyClass();
        T plusEins(T* value) {
                return((*value) + 1);
        }
};
```

```
MyClass<long>    MyL;    // compiler gen. template
MyClass<long *>  MyPtrL; // partiell spezialisiertes
                         // Template fuer long*
```

Eine Spezialisierung eines Templates im namespace std gestaltet sich ganz genau so. Man macht den namespace auf und schreibt die Template-Spezialisierung nieder. Hier die Spezialisierung von `std::numeric_limits` für einen wie auch immer gearteten *crazyint*.

```
namespace std {

template<> class numeric_limits<crazyint> {
        [...]
};

}

template <class T> class MyClass {
public:
        MyClass();
        typedef std::numeric_limits<T> limitT;
};

MyClass<crazyint> MyCI;
```

11.3 SGI std::hash_map

Ein weiteres interessantes Beispiel, bei dem eine Erweiterung von namespace std in Form einer Template-Spezialisierung notwendig sein kann, ist `std::hash_map` aus der STL von [STL'00].

```
// Prototyp: hash_map<Key, Data, HashFcn, EqualKey, Alloc>

using namespace std;

struct eqstr {
        bool operator()(const char* s1, const char* s2) const {
                return strcmp(s1, s2) == 0;
        }
};

int main() {
        hash_map<const char*, int, hash<const char*>, eqstr> months;
        months["january"] = 31; [...] months["december"] = 31;
        cout << "september -> " << months["september"] << endl;
}
```

Wird die Hashmap in einer Template-Umgebung verwendet, so muss man auch hier wieder bei eigenen Datentypen den namespace std erweitern.

```
template <class K, class V> class MyClass {
public:
        MyClass();
        std::hash_map<K, V> hashmap;
};
```

Die Defaults für die Template-Parameter sind für *HashFunc* `std::hash<K>` und für *EqualKey* `std::equal_to<V>`. Für einfache Datentypen wie `const char*` oder Integer-Typen gibt es schon Spezialisierungen. Für eigene Typen als Key muss man entsprechend selbst nachhelfen. Wie in den vorherigen Kapiteln gezeigt wurde, ist dies auch kein Problem.

Jedoch kann es Probleme mit Algorithmen geben, sie auf `std::hash_map` operieren und evtl. auf `std::hash<K>` zugreifen wollen. Die Spezifikation der Template-Spezialisierung ist schließlich an die Translation-Unit gebunden, und der Compiler kann u. U. benötigte Symbole nicht finden, wenn Sie von den Algorithmen benötigt werden. Zwar gibt es mit dem Argument-Dependent Lookup (ADL), bzw. Koenig-Lookup ausgefeilte Regeln für die Suche nach Symbolen, jedoch ist die Grenze des Lookups aller dieser Regeln eben die Translation-Unit.

Betrachten wir also einen Fall, in dem ein Lookup von `std::hash<K>` fehlschlägt.

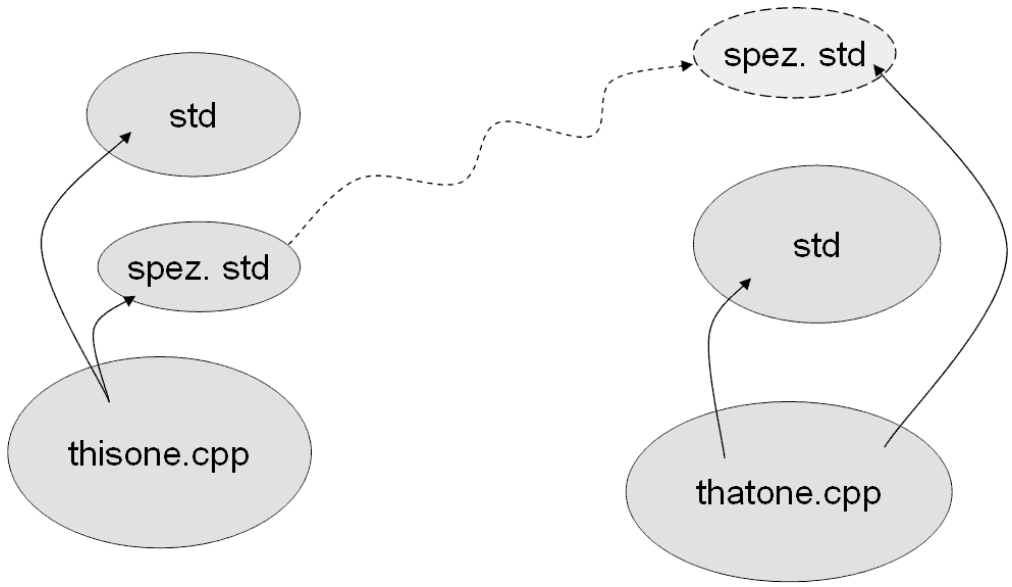


Abbildung 11: Auslagerung in Header-Dateien

```
// thisone.cpp

namespace std {
        template<>  class hash<mytype> {
                [...]
        };
};

// [...]
// beliebige Templateklasse mit hash_map<K, V>
// und K = mytype
```

Und in einer anderen Translation-Unit:

```
// anotherone.cpp

// In Template-Umgebung mit K = mytype

// h ist hash_map<K,V>
std::algo(h);
```

Kein Compiler der Welt wird hier das passende Symbol für `std::hash<T>` finden. Folglich muss man selbst dafür sorgen, dass das Symbol verfügbar ist, indem man z. B. alle eigenen Template-Spezialisierungen von namespace std in einen Header auslagert und überall dort einbindet, wo es notwendig ist (Abbildung 11).

11.4 Fazit

Die Erweiterung von namespace std mit eigenen Symbolen ist untersagt, die Spezialisierung von Templates aus dem namespace std ist erlaubt, wenn die bei der Spezialisierung verwendeten Datentypen eigene Namen besitzen und von außen gelinkt werden müssen.

Das Problem dabei:

- Erweiterungen von namespaces sind an Translation-Units gebunden.
- Man muss sicherstellen, dass die Spezialisierungen von std-Templates überall dort erreichbar sind, wo sie benötigt werden.
- Die Lösung sind extra Header-Dateien.

12 Expression Templates

12.1 Einführung

Das Anliegen von Expression Templates in C++ ist es, Berechnungen während der Übersetzungszeit vom Compilier durchführen zu lassen und das entsprechende Ergebnis (numerischen Werte oder compiliergenerierter Programmcode) anstelle von Formeln bzw. Schleifen in den kompilierten Code einzufügen.

Die Technik von Expression Templates ist sehr komplex, aber auch sehr interessant, da sich das „teure“ Erzeugen von temporären Objekten zu Laufzeit damit ersparen lässt. Daher auch folgende Unterüberschrift unter einem Artikel über Expression Template in einer Ausgabe des C/C++ User Journal: „Do you want high-performance, readable expressions? Expression templates do it all.“ [Langer'03]

Als Ergebnis dieser Technik erhält man Programmcode mit einer sehr hohen Laufzeitperformance, dem aber einer längen Übersetzungszeit gegenübersteht. Die lange Übersetzungszeit ist dem geschuldet, dass der Compilier die Expression Templates während der Übersetzungszeit evaluiert, die Berechnungen ausführt und das Ergebnis an die entsprechende Stelle im Code einfügt.

12.2 Primazahlberechnung – eine einfache Demonstration

Damit man ein Gefühl für die Anwendung von Expression Templates bekommt, folgt nun Programmcode, welcher zur Laufzeit alle Primzahlen zwischen 2 und 18 berechnet – die Betonung liegt hier klar auf „zur Laufzeit“! Dieses Programm kompiliert jedoch nicht komplett durch, sondern wirft mit jeder Fehlermeldung die nächste Primzahl aus, d. h. man erhält auch kein Lauffähiges Programm, sondern es demonstriert wirklich nur die Berechnung zur Laufzeit.

„Prime number computation by Erwin Unruh“ [Unruh]

```
template <int i> struct D {D(void*); operator int();};

template <int p, int i> struct is_prime {
enum {prim = (p==2) || (p%i) &&
is_prime<(i>2?p:0), i-1> :: prim};
};

template <int i> struct Prime_print {
Prime_print<i-1> a;
```

```
enum {prim = is_prime<i, i-1>::prim};
void f() {D<i> d = prim ? 1 : 0; a.f();}
};

template<> struct is_prime<0,0> {enum {prim=1};};
template<> struct is_prime<0,1> {enum {prim=1};};

template<> struct Prime_print<1> {
enum {prim=0};
void f() {D<1> d = prim ? 1 : 0;};
};

#ifndef LAST
#define LAST 18
#endif

main() {
Prime_print<LAST> a;
a.f();
}
```

12.3 Rekursivität als Grundlage

Die wichtigste Grundlage für Expression Templates ist die Rekursivität von Problemen. Rekursivität bedeutet hier, dass der Compilier ein Template instanziiert, welches ein weiteres Templates instanziiert, welches wiederum ein Template instanziiert, usw. Dieses Verhalten haben viele mathematischen Probleme, was auch der Grund dafür ist, dass Expression Templates vorrangig im Bereich der Numerik und Matrizenrechnung angesiedelt sind.

Problematisch ist nur, dass diese Rekursivität nicht mit Hilfe von (Member-) Funktionen ausgedrückt werden darf, sondern ausschließlich durch Templates. Des Weiteren dürfen keine unbekannte Werte enthalten sein, die erst zu Laufzeit evaluiert werden. Dies bedeutet, dass jedes Detail, welches zur Berechnung bzw. Ausführen des Expression Templates notwendig ist, dem Compilier zur Kompilierzeit bekannt sein muss.

12.4 Herleitung eines Expression Templates am Beispiel der Fakultätsberechnung

Ein einfaches Beispiel an dem sich das Erstellen und die Funktionsweise eines Expression Templates erklären lässt, ist die mathematische Funktion „Fakultät“. Eine funktionale Umsetzung dieser mathematischen Funktion ist trivial und sieht im Allgemeinen ähnlich dem folgenden Codebeispiel aus.

```
int factorial (int n) {
return (n==0) ? 1 : n*factorial(n-1);
}

std::cout << factorial(4) << std::endl;
```

Der Funktionsaufruf `factorial(4)` berechnet rekursiv die Fakultät von `4`, indem sich `factorial(n)` immer wieder selber mit `factorial(n-1)` aufruft, bis `n==0` ist und dadurch die Rekursion abgebrochen wird. Bei diesem Abbruch durch `n==0` wird der Zahlenwert `1` an die entsprechend aufrufenden Instanz zurückgegeben, so dass die Rekursion rückwärtig aufgerollt werden kann, was am Ende zum Ergebnis von `4!=24` führt. Problem hierbei ist nun aber, dass diese Berechnung zur Lauf- und nicht zur Kompilierzeit ausgeführt wird. Wobei das eigentliche Problem weniger an der Ausführung zur Laufzeit, sondern an der damit verbundenen Erzeugung von temorären Objekten im Speicher liegt. Der bessere Weg für diese Berechnung ist mit Hilfe von Templateinstanziierung.

Der erste Schritt zum Expression Template ist die Umwandlung von einer Funktion in eine Klasse – genauer gesagt in ein Klassentemplate. Anstelle einer Funktion `factorial` führen wir also ein Klassentemplate namens `factorial` ein. Wichtig ist, dass wir nicht wie gewöhnlich ein ungtyptes Klassentemplate nehmen, sondern ein festgetyptes – in unserem Fall ein Klassetemplate, welches mit `int` getypt ist, d. h. das Template wird mit konstanten `int`-Variablen instanziiert. (Für Expression Templates müssen die Template genau getypt werden, anderes als bei herkömmlichen Templates!)

```
template <int n>
struct factorial {
  enum { ret = factorial<n-1>::ret * n };
};
```

Es fällt auf, dass dieses Klassentemplate weder Daten noch Member-Funktionen enthält, lediglich `enum` als Aufzählungskonstante mit einem einzigen Wert `ret` wird belegt. Der Wert `ret` instanziiert ein neues `factorial`, aber mit `n-1`, was dann wiederum ein neues `factorial` mit wieder einem um eins dekrementierten `n` instanziiert, usw. Die rekursive Templateinstanziiernung hat nun begonnnen.

Dieser `enum`-Wert `factorial::ret` wird später das gesuchte Kompilierzeitergebnis liefern.

Jedoch ist dieses Expression Template noch nicht komplett, da es aktuell noch keine Abbruchbedingung für die Rekursivität gibt. Eine Abbruchbedingung erfolgt hier über eine Spezialisierung des entsprechenden Templates – in unserem Falle ein Spezialisierung für den Fall `factorial<1>`.

```
template <>
struct factorial<1> {
```

```
  enum { ret = 1 };
};
```

Die Rekursion endet nun bei `factorial<1>`, also für `n==1`, da an dieser Stelle kein neues Template instanziiert wird.

Nun kann man sich das Ergebnis `factorial::ret` von `factorial<4>` ausgeben lassen.

```
std::cout << factorial<4>::ret << std::endl;
```

Das Wesentliche hieran ist jedoch nun, dass an der Stelle von `factorial<4>::ret` in der ausführbaren Datei weder eine Schleife noch eine Rekursionsformel steht, sondern lediglich eine `24`. Dies ist der große Vorteil von Expression Templates. Der Compilier evaluiert während der Kompilierzeit das Ergbnis des Templates und ersetzt die entsprechende Stelle durch das Resultat. In dem funktionalen Ansatz am Anfang wird das Ergebnis erst zu Laufzeit des Programmes berechnet.

Ein weiterer wesentlicher Vorteil zeigt sich, wenn man ein Berechnungsresultat als Array-Größe weiterverwenden möchte. Dynamische Arrays, wie z. B. `int array[factorial(4)]`, sind mit ANSI-C++ nicht möglich. Wenn man aber das Array mit Hilfe eines Expression Template, z. B. `int array[factorial<4>::ret]`, anlegt, so ist dies möglich, da dieser Ausdruck auch zur Kompilierzeit durch den Compilier ausgewertet und durch den entsprechenden Zahlenwert ersetzt wird.

12.5 Weitere Möglichkeiten

Die Berechnung der Fakultät eines Integerwertes ist nur ein sehr einfaches, aber gut verständliches Beispiel für ein Expression Template. Expression Templates bieten dem Programmierer sehr viele Möglichkeiten, u. a. ist es möglich, ganze Ausdrücke zu evaluieren. Als Beispiel für die Möglichkeiten seien hier einerseits das Skalarprodukt zweier n-dimensionaler Vektoren (siehe [Langer] und [Veldhuizen'95]) und andererseits die Integralrechnung (siehe [Langer'03]) genannt.

Sellvertretend sei an dieser Stelle der Programmcode für das Skalarprodukt zweier 4-dimensionaler Vektoren inklusive UML-Diagramm (siehe Abbildung 12 auf der nächsten Seite) gezeigt.

```
template <size_t N, class T>
class DotProduct {
public:
  static T eval(T* a, T* b)
  { return  DotProduct<1,T>::eval(a,b)
          + DotProduct<N-1,T>::eval(a+1,b+1);
```

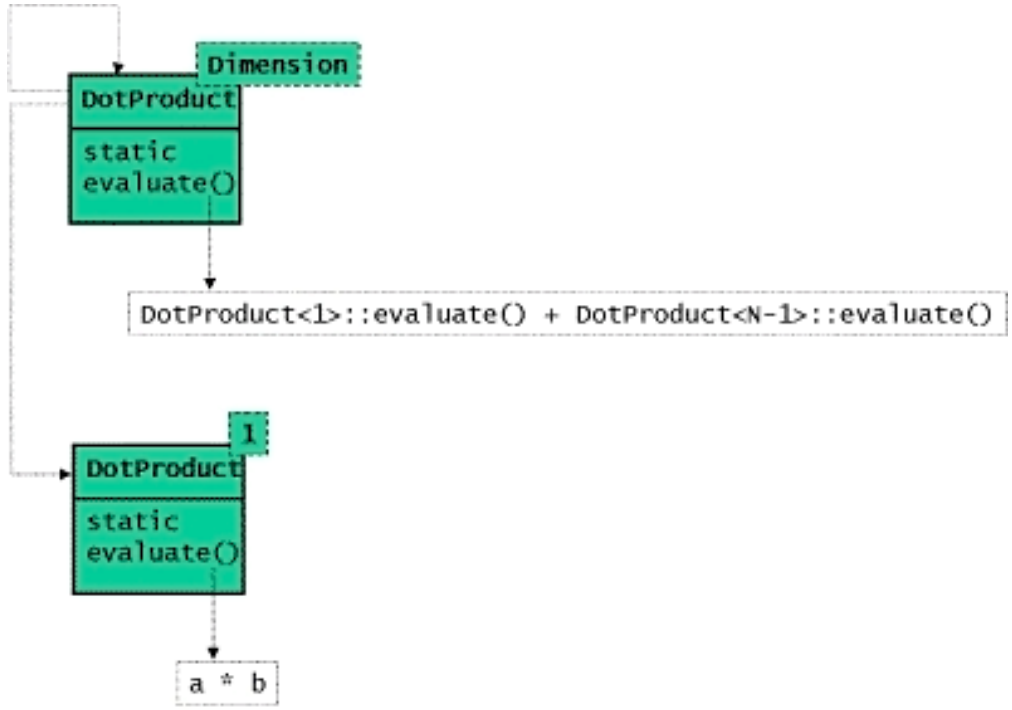


Abbildung 12: UML-Diagramm für Skalarprodukt

```
  }
};

template <class T>
class DotProduct<1,T> {
public:
  static T eval(T* a, T* b)
  { return (*a)*(*b); }
};

template <size_t N, class T>
inline T dot(T* a, T* b)
{ return DotProduct<N,T>::eval(a,b); }

int a[4] = {1,100,0,-1};
int b[4] = {2,2,2,2};
std::cout << dot<4>(a,b) << std::endl;
```

Bei diesem Beispiel hat man wieder das Problem, dass die Kompilierzeit steigt, da der Ausdruck `dot<4>(a,b)` durch dem Compiler während der Kompilierzeit ausgewertet werden muss. Andererseits steigt die Laufzeitperformance des ausführbaren Programmes, da während der Laufzeit keine temporären Variablen verwaltet werden müssen. Dieser Per-

formancegewinn lohnt sich um so mehr, desto häufiger der identische Ausdruck benutzt wird (Anwendung: Matrixtransformationen).

Hinweis: Für das Erstellen komplexer Expression Templates eignen sich das „Composite Patter“ zum Aufbau von rekursiven Baumstrukturen und das „Interpreter Pattern“ zum Evaluieren von Syntax-Bäumen (siehe [Langer]).

12.6 Zusammenfassung

Expression Templates sind Spezialisierungen von Templates, sie besitzen konstante „build-in“-Werte und sind daher nicht wie allgemeine Templates ungetypt. Sie dienen dazu, Berechnungen und Evalationen während der Kompilierzeit durchzuführen und die Ergebnisse anstelle der Aufrufe in den binären Programmcode einzufügen. Dies hat zur Folge, dass die Kompilierzeit steigt, aber die entscheidende Performance während der Laufzeit wächst.

Als Datentyp für verwendete Variablen ist der Aufzählungstyp `enum` möglich, da Standardvariablentypen bei Expression Templates nicht möglich sind!

Des Weiteren ist es wichtig zu wissen, dass es eine maximale Rekursionstiefe abhängig vom jeweiligen Compilier gibt (siehe dazu ANSI-C++ 14.7.1.14). Diese Rekursionstiefe liegt bei dem GNU-Compilier gcc bei 17 Rekursionsstufen. Man kann jedoch diese festgelegt Rekursionstiefe mit Hilfe des Flags `-ftemplate-depth-128` auf 128 ändern.

13 Memory Management

13.1 Stack vs. free-store

[Wilson'95] [Berger'02]

Wir haben zwei Möglichkeiten, in Programmen Speicher zu benutzen – wir können automatische Variablen benutzen, die auf dem Stack angelegt werden, oder Speicher vom free-store anfordern. Der Speicher im Stack wird vom Compiler verwaltet (das heißt, die Größe des angeforderten Speichers muss zur Kompilierzeit feststehen), und ist nach dem LIFO-Prinzip geordnet – Speicher der in einem Block angefordert wurde, wird am Ende dieses Blocks freigegeben. Speicher, der vom free-store angefordert wurde, hat keine besondere Ordnung – er wird explizit vom Nutzer des Speichers freigegeben, wenn dieser ihn nicht mehr braucht. Der Verwalter des free-stores ist der Allokator, ein Programm, das dem Nutzer ein Interface[1] anbietet, Speicher beliebiger Größe zur Laufzeit anzufordern, und angeforderten Speicher wieder freizugeben.

```
...
{
  char a[255];
  char* b = new char[255];
  ...
} // a wird freigegeben
...
delete[] b; // b wird explizit freigegeben
```

Die Aufgabe des Allokators ist es, Nachfragen nach Speicher beliebiger Größe zu erfüllen, und einen Zeiger auf den Anfang des Speichers zurückzuliefern. Er hat auf den Speicher, der so vom Nutzer angefordert wurde, keinen Zugriff mehr, insbesondere kann er ihn nicht verschieben, denn der Nutzer erwartet den Speicher an eben der Stelle, auf die der Zeiger zeigt, den er vom Allokator erhalten hat. Der Nutzer kann den Speicher zu einem beliebigen Zeitpunkt später wieder freigeben, der Allokator weiß nicht, wann dies sein wird.

Meist zeigt sich, dass das Ziel die Effizienz im Speicherverbrauch ist. Es ist leicht einen Allokator zu implementieren, der Anfragen mit einer Laufzeitkomplexität von $O(1)$ bedient, doch dieser wäre mit seinen Resourcen sehr großzügig und könnte bald keinen Speicher mehr verteilen, weil dieser ihm ausgegangen wäre. Deshalb ist es wichtig auf die Verwaltung dieser Resourcen mehr Augenmerk zu legen.

Die Entscheidung, welche Stelle des verwalteten Speichers er für eine Anfrage benutzt, ist

[1] in C sind das die Funktionen `malloc` und `free`; in C++ die Familie von `new` und `delete` Ausdrücken/Operatoren

wichtig für den Allokator, denn sie bestimmt, ob er für spätere Anfragen noch genügend Speicher zu Verfügung hat, und ob der Speicher optimal genutzt wird. Für diese Entscheidung muss der Allokator wissen, wo und wieviel freier Speicher zur Verfügung steht. Aus diesem muss er einen Block auswählen, der den Anforderungen des Nutzers genügt, und den restlichen freien Speicher nicht zu stark fragmentiert. Andererseits erwartet der Nutzer, das seine Anfrage möglichst schnell erfüllt wird. Die Entscheidung, wie die Fragmentation minimiert wird, ist nicht einfach, und ein Nutzer kann seine Anfragen immer so stellen, dass sie groß wird. Es gibt keinen perfekten Algorithmus, der für alle Fälle Fragmentation niedrig hält[2].

13.1.1 Strategien, Vereinbarungen, Mechanismen

Die Strategie jedes Allokators ist einfach formuliert: "nimm die Speicherblöcke, die später keine Fragmentation verursachen". Aber wie wird das erreicht? Es gibt verschiedene Vereinbarungen, um das zu erreichen:

- Muster in der (Wieder-)Benutzung von Speicher ausnutzen
- Trennen und Zusammenfügen von Speicherblöcken
- Passende Speicherblöcke auswählen

Der Allokator implementiert einen Mechanismus, um eine Vereinbarung durchzusetzen. Die verschiedenen Mechanismen haben Einfluss auf die Erfüllung der Vereinbarung. Bei Allokatoren unterscheidet man die interne Fragmentierung, die durch allokatorspezifische Daten über den verwalteten Speicher und herausgegebene Blöcke, die größer sind als der abgeforderte Speicher, verursacht wird, und externe Fragmentierung, die durch freie Blöcke, die nicht mehr für Anfragen verwendet werden können (weil sie zu klein o. ä. sind), verursacht wird.

13.1.2 Trennen und Zusammenfügen von Speicherblöcken

Das Auftrennen von Blöcken in kleinere Blöcke ist notwendig, um Anfragen sinnvoll bedienen zu können. Doch manchmal lohnt sich ein Aufspalten nicht, wenn der freie Rest zu klein wird. Der Allokator kann dann den ganzen Block zurückgeben (interne Fragmentation), oder doch trennen (externe Fragmentation).

[2] Eine Untergrenze für worst-case Fragmentation ist proportional zum benutzten Speicher M und zum Verhältnis n zwischen größtem und kleinsten Speicherblock – $M \cdot \log n$

Wenn ein Allokator einmal aufgetrennte Blöcke nicht zusammenfügen kann, wird er irgendwann Nachfragen nach großen Blöcken nicht mehr bedienen können. Er muss also intern Speicherblöcke so verwalten, das Zusammenfügen möglich wird. Eine Möglichkeit, dies zu erreichen, sind *bounding tags* – reservierte Bereiche am Anfang und Ende eines Speicherblocks, in denen Informationen über den nächsten und vorigen Block enthalten sind. Die Größe dieser *tags* trägt mit zur internen Fragmentierung bei.

Das Zusammenfügen von Blöcken sollte meist nicht sofort geschehen, denn wenn geich eine Anfrage der gleichen Größe kommt, muss der Allokator den Blöck wieder auftrennen. Deshalb schiebt man freigewordene Blöcke auf eine *Quicklist*, von der sie vielleicht gleich wieder entnommen werden, oder aber mit anderen Blöcken zusammengefügt werden.

13.1.3 Suche nach freien Blöcken

Der Allokator muss eine Liste aller freien Speicherblöcke halten, die *free list*. Die Suche nach einem freien Block kann nach verschiedenen Kriterien erfolgen:

- *best fit* – der Block, der am besten zur angeforderten Größe passt, wird gewählt
- *first fit* – der erste Block der Liste wird gewählt
- *next fit* – der Block nach dem Block, der zuletzt gewählt wurde, wird gewählt

best fit könnte zu einem Block führen, der der Größe der Anforderung fast gleicht, aber einen sehr kleinen Rest hinterlässt, der nicht mehr benutzt werden kann. Bei *first fit* ist die Sortierung der Liste entscheidend – bei unsortierter Liste werden große Blöcke am Anfang der Liste zerteilt, und die Suchzeit wächst sehr schnell an. Bei Sortierung der Liste nach dem kleinsten Element wird *first fit* zu *best fit*. Bei Sortierung nach Adressen wird das Zusammenfügen beieinander liegender Blöcke einfach. Bei *next fit* wird der Speicher mit Blöcken von unterschiedlichen Anfragen gefüllt, was nicht zur Lokalität von Speicher beiträgt.

13.1.4 Segregated Free Lists

Ein Allokator kann statt einer *free list* mehrere für verschiedene Blockgrößen oder Größenklassen verwenden. Dies sorgt dafür, dass Speicher gleicher Größe beieinander liegt. Das Trennen und Zusammenfügen von Blöcken ist uneingeschränkt möglich – die Blöcke wandern in eine andere *free list*. Der Allokator fragt erst dann Speicher vom System an, wenn die free list einer Größe leer wird.

13.1.5 Buddy Systems

Buddy Systems sind eine spezielle Version der *Segregated Free Lists*. Der von Allokator verwaltete Speicher wird rekursiv halbiert, und so in immer kleinere Teile gleicher Größe zerlegt. Die Teile gleicher Größe bilden eine *free list*. Die Besonderheit ist, das das Zusammenfügen von Speicher nur bei benachbarten "Buddies" auf einer Stufe der Hierarchie möglich ist. Dies vereinfacht das Zusammenfügen – aber auf Kosten der internen Fragmentierung, die hoch werden kann.

13.1.6 Bitmapped Fits

Bei den bisher besprochenen Mechanismen wurde die *free list* im freien Speicher "eingewoben". Bei *Bitmapped Fits* wird in einem Bitvektor jedes freie Byte/Wort/Block als eine Eins dargestellt, alle belegten als Null. Der Vorteil des geringen Platzbedarfs und Unterbringung der *free list* in einem separaten Speicherbereichs hat als Nachteil erhöhte Suchzeiten. Wenige Implementationen nutzen dieses Konzept.

13.1.7 Regions

Regions sind ein sehr spezielles Speicherverwaltungskonzept, bei dem schnelle Reaktionszeiten mit erhöhten Speicherbedarf erkauft werden. Anfragen werden aus einem großen Block zusammenhängendem Speicher erfüllt, aus dem vom Anfang beginnend Blöcke herausgegeben werden, die einer nach dem anderen liegen. Freigegeben kann nur der großer Block als ganzes, wenn. Dieses Konzept ist einsetzbar bei Applikationen, die Speicher inkrementell anfordern, und zu einem bestimmten Zeitpunkt alles wieder freigeben.

13.2 C++ Speicherwaltung

[Austern'00] [Langer'98] [Myers'93]

Um in C++ den *free store* verwalten zu können, gibt es mehrere Möglichkeiten. Die Operatoren `new` und `delete` können pro Klasse und global überschrieben werden, und für die Container der STL gibt es *Allocators*. `new` und `delete` gibt es in verschiedenen Ausführungen – für Allokation von Objekten und Allokation von Arrays von Objekten.

```
T* p=new T;     //calls operator new(sizeof(T))
T* a=new T[n]; //calls operator new(n*sizeof(T)+x)
delete p;       //calls operator delete(p)
delete[] a;     //calls operator delete[](a)
```

Die beiden Versionen sollten nicht verwechselt werden – der Speicher, der mit `new` angefordert wurde, kann nicht mit `delete[]` gelöscht werden und umgekehrt, denn `new[]` könnte einen Block anfordern, der größer als die Größe des Arrays ist, um Informationen zu speichern, die von `delete[]` benötigt werden, z. B. die Größe des Arrays.

13.2.1 Allokatoren

Allokatoren sollen unterschiedliche Allokationsmodelle und -strategien abkapseln und dem Nutzer (meist STL-Container) ein einheitliches Interface zum Anfordern und Freigeben von Speicher bereitstellen. Die Anforderungen an einen Allokator sind im Standard in Kapitel 20.1.5 in Tabelle 32 aufgelistet. Grundsätzlich bietet die Klasse vier Funktionen: `allocate` und `deallocate`, um Speicher zu anzufordern und freizugeben, und `construct` und `destroy` um Objekte in diesem Speicher zu erzeugen und zu löschen.

```
template <typename T> class allocator {
  ...
  pointer
  allocate (size_type n,
            allocator<void>::const_pointer hint=0);
  void
  deallocate (pointer p, size_type n);

  void
  construct (pointer p, const T& val)
  {
    new (p) value_type (val);
  }

  void
  destroy (pointer p)
  {
    p->~value_type ();
  }
  ...
};
```

Die Typen `value_type`, `size_type` und `pointer` sind Typedef's, die vom Template-Argument `T` abhängen. `construct` erzeugt durch den placement new Operator ein Objekt an der übergebenen Speicheradresse. `destroy` ruft den Destruktor des Objekts an der übergebenen Speicheradresse auf.

Der Standard besagt, dass Standard-konforme Container von ihren Allokatoren erwarten können, dass sie austauschbar sind. Dies bedeutet, dass diese Allokatoren keine nichtstatische Membervariablen besitzen dürfen. Dies schränkt den Einsatzbereich von Alloka-

toren stark ein, man kann z. B. keine Allokatoren bauen, die zwei getrennte Speicherbereiche verwalten (und damit nicht ohne weiteres austauschbar sind), und diese mit den Standard-Containern verwenden, ohne darauf zu achten, das diese Allokatoren zwischen den Containern nicht ausgetaucht werden.

Insbesondere gibt es einen Mechanismus, um Allokatoren umzuwandeln, so dass Container wie z. B. Listen, die einen Typen `T` verwalten, diesen aber in speziellen Strukturen verpacken, diese Strukturen auch über den übergebenen Allokator verwenden können.

```
template <typename T> class allocator {
  ...
  allocator () {}
  allocator (const allocator<T>&) {}

  template <typename U>
  allocator (const allocator<U>&) {}

  template <typename U>
  struct rebind {
    typedef allocator<U> other;
  };
  ...
};

allocator<T> allocT;
// allocT is an allocator for Type T

typedef allocT::template rebind<U>::other allocU;
// allocU is an allocator for Type U
```

Durch den Template-Konstruktor ist jeder Allokator in jeden anderen implizit konvertierbar. Durch der in dem lokalen Template `struct rebind` eingebetteten Typdeklaration `other` kann man aus einem gegebenen Allokator einen Allokatortyp erhalten, der andere Typen alloziiert.

Ein Beispiel, wie ein Container seinen Allokator benutzt, soll hier anhand einer einfach verketteten Liste gezeigt werden:

```
template <typename T,
          class Allocator=std::allocator<T> >
class slist {
  ...
  typedef typename
    Allocator::value_type
    value_type;
  typedef typename
    Allocator::pointer
    pointer;
```

```
    typedef typename
      Allocator::template rebind<slist_node>::other
      node_alloc;
    ...
    struct slist_node {
      slist_node (const value_type& x)
        : data (x), next (0)
      {}
      T data;
      slist_node* next;
    };
    ...
    slist_node* elems_;
    size_type size_;
    ...
};
```

Der Container erhält die Typen `value_type`, `pointer`, etc., auf denen er operiert, durch seinen Allokator. Da er die Objekte vom Typ `T` in einer Knotenstruktur `struct slist_node` hält, deklariert er einen Knoten-Allokator `node_alloc` durch den rebind-Mechanismus.

```
  template <typename T,
            class Allocator=std::allocator<T> >
  class slist {
  ...
  public:
    slist (size_type size, const value_type& x,
           const Allocator& a=std::allocator<T>())
      : size_ (size)
     {
       if (size_ == 0) return;
       elems_ = node_alloc(a).allocate (size);
       for (size_type i = 0; i != size; ++i) {
         try {
           node_alloc(a).construct(&elems_[i], x);
         } catch (...) {
           node_alloc(a).deallocate(elems_, size_);
           throw;
         }
         elems_[i].next = &elems_[i + 1];
       }
       elems_[size_ - 1].next = 0;
     }
     ...
  };
```

Im Konstruktor wird der Knoten-Allokator aus dem übergebenen Allokator erzeugt, und verwendet, um Speicher für die Knoten anzufordern und die Knoten zu initialisieren. Im

Falle eines Fehlers bei dieser Aktion werden die Knoten wieder freigegeben.

13.3 Spezielle Allokatoren

Spezielle Allokatoren werden meist dort eingesetzt, wo man Kenntnis darüber hat, wieviel und wann Speicher angefordert wird, und wann er wieder freigegeben wird. Das pimpl-Idiom ist ein gutes Beispiel – meist will man, dass Objekte, die das pimpl-Idiom nutzen, sich wie Objekte verhalten, die auf dem Stack erzeugt wurden, obwohl, das pimpl-Idiom auf der Erzeugung der Objekte im *free-store* basiert. Die Bereitstellung eines großen Speicherstücks und das Erzeugen der Objekte in diesem Speicherstück kann helfen, den Overhead der dynamischen Speicherverwaltung zu minimieren.

```
struct impl;
class interface {
  impl* pimpl;

  void* operator new (std::size_t);
  void operator delete (void*);
  void* operator new[] (std::size_t);
  void operator delete[] (void*);

public:
  friend class impl;
  interface() : pimpl (new impl) {}
  ~interface () { delete (pimpl); }
};
```

Die Klasse `interface` hält alle Variablen in der Struktur `impl`, auf den sie einen Zeiger hält, `impl` wird mittels `new` erzeugt und mittels `delete` gelöscht. `interface` kann nur auf dem *Stack* angelegt werden, da die Operatoren `new` und `delete` für diese Klasse privat sind. Eine fast-pimpl Implementierung, die wirkliche Aufrufe von `new` und `delete` minimiert, basiert auf Überladung des `new`- und `delete`-Operators und könnte so aussehen:

```
struct impl;
namespace {
  std::vector<impl*> freelist;
}

struct impl {
  enum { init_impls = 1024 };
public:
  impl () : impl_data (0) {}

  void* operator new (std::size_t)
  {
    if (freelist.empty ()) {
```

```
      // get new memory in big chunks
      impl* itmp = ::new impl[init_impls];
      impl* iend = itmp + init_impls;

      freelist.reserve(init_impls);
      while (itmp != iend) {
        freelist.push_back (itmp++);
      }
    }

    impl* timpl = freelist.back ();
    freelist.pop_back ();
    return (void*) timpl;
  }
  void operator delete (void* p)
  {
    freelist.push_back (reinterpret_cast<impl*>(p));
    if (freelist.size () == init_impls) {
      // free big memory chunk
      impl* tmp = freelist.front();
      freelist.clear ();
      delete[] tmp;
    }
  }

  private:
      int impl_data;
      ...
};
```

Eine *freelist* von `impl`-Zeigern wird von den überladenen `new`- und `delete`-Operatoren verwaltet, neuer Speicher (in den `init_impls` Stück `impls` passen) wird nur angefordert, wenn diese Liste leer ist, und freigegeben, wenn alle Zeiger der free-list, die zu einem Speicherstück gehören, nicht mehr verwendet werden. Die *freelist* kann hier als *Stack* implementiert werden, da die Klasse `interface` nur Objekte auf dem Stack anlegen kann, und somit Konstruktor- und Destruktoraufrufe der Blockstruktur des Programmflußes folgen.

13.3.1 simple segregated storage

Dieses Konzept ist oft anwendbar, wo viele Objekte der gleichen Größe dynamisch erzeugt und zerstört werden. In der boost-Bibliothek ([boost]) ist dieses Konzept der *simple segregated storage* in *boost::pool* umgesetzt. Unser Beispiel würde so aussehen:

```
namespace {
```

```
  boost::object_pool<impl> impl_pool;
}

void* impl::operator new (std::size_t)
{
  return (void*) impl_pool.malloc();
}

void impl::operator delete (void* p)
{
  impl_pool.free(reinterpret_cast <impl*> (p));
}
```

Der `boost::object_pool<impl>` kann nur Objekte vom Typ `impl` erzeugen. Große Speicherstück werden in diese kleinen Einheiten zerlegt, und somit erreicht man bei Allokation/Deallokation eine Komplexität von $O(1)$ (*armortized constant time*).

Unsere handgemachte Implementation hat den Nachteil, das Objekte vom Typ `interface` nur auf dem Stack angelegt werden können. Generell erkauft man sich die Vorteile von spezielleren Allokationstategien mit dem Verlust von Generalität, und Allgemeinheit. Falls sich die Allokationsmuster später ändern, kann die spezielle Strategie schlechtere Ergebnisse liefern als eine allgemeine. Deshalb muss man gut abwägen, ob ein spezieller Allokator tatsächlich angebracht ist.

Es gibt in verschiedenen Bereichen spezielle Allokatoren, z. B. im Bereich der Multiprozessorsysteme und des *parallel computing*. Hier müssen Allokatoren nicht nur die bereits besprochenen Probleme lösen, sondern sich auch mit Problemen, die sich durch die Nutzung von gesonderten Speicherbereichen für je einen Prozessor ergeben. Beispielsweise kann das *false cache-line sharing* auftreten, wobei mehrere Prozessoren auf Speicher zugreifen, der in einer *cache-line* liegt. Dabei muss diese neu aus dem Hauptspeicher geladen werden, ohne dass dies aus der Sicht eines Prozessors notwendig wäre.

Im Bereich des *embedded computing* ist dynamische Speicherverwaltung ein besonders schwieiges Feld. Da wir schon gesehen haben, dass die Algorithmen für diese Speicherverwaltung nicht deterministisch sind, sondern von vielen Faktoren abhängen, die außerhalb des Prozesses liegen, der Speicher anfordert oder freigibt, ist diese Art von Speicherverwaltung oft nicht einsetzbar, oder es können nur spezielle Arten von Allokation verwendet werden. *simple segregated storage* könnte hier eine Lösung bieten.

13.4 Zusammmenfassung

Dynamische Speicherverwaltung ist selbst heute noch nicht abschließend gelöst, und allgemeine Allokationsstrategien können wichtige Spezialfälle noch nicht abdecken. Trotz-

dem sollte man sich zweimal überlegen, ob man speziell angepasste Allokatoren verwendet. Man muss einen Kompromiss eingehen, der abwägt zwischen schneller Speicherverwaltung und effizienter Speicherverwaltung. Meist erkauft man sich die Schnelligkeit durch erhöhten Speicherverbrauch.

14 Smart Pointer

14.1 Einleitung

Einfache Zeiger, oder im Englischen *plain pointer*, sind eines der mächtigsten Werkzeuge in C++ und eine wesentliche Voraussetzung für die Effizienz der Sprache. Ihr Potential schöpfen Zeiger dabei vor allem aus ihrem geringen Abstraktionsgrad, dem direktes Operieren auf Systemspeicher. Allerdings impliziert dieser geringe Abstraktionsgrad auch zahlreiche Probleme:

- Der Nutzer ist selbst für die korrekte Verwaltung des Zeigers und des von ihm referenzierten Speichers verantwortlich.
- *pointer* sind nicht *thread-safe*.
- *pointer* besitzen keine *value semantic*; das heißt der Nutzer kann einen Zeiger nicht beliebig verwalten, zum Beispiel kopieren oder löschen, da dies sich auf den von ihm referenzierten Speicher auswirkt.
- Fehler bei der Benutzung von Zeigern können häufig erst zur Laufzeit festgestellt werden

Typische Fehler bei der Verwendung von Zeigerns sind:

- *memory leaks* durch das nicht korrekte Deallozieren von Speicher
- Zugriff auf invalidierten Speicher, was meist zu einem Programmabsturz führt

Diese Probleme führten zur Entwicklung des Konzepts der *smart pointer*, die als zusätzliche Abstraktionsschicht dem Nutzer die Verwendung von Zeigern erleichtern sollen; allerdings auch, einen zum Teil nicht unerheblichen Overhead mit sich bringen.

In der Anwendung sollen *smart pointer* dabei möglichst ähnlich zu einfachen Zeigern sein, dabei ist *ähnlich* jedoch entscheidend, da bei absoluter Gleichheit auch die zusätzliche Abstraktion verloren gehen würde.

14.2 Smart Pointer Design

Ein *smart pointer* wird im Allgemeinen als *wrapper* um einen einfachen Zeiger implementiert.

```
template<class T>
class Smart_Ptr {

...  // additional features

public:

  // default pointer operators

  T& operator*() {

    return *_pointee;
  }

  T* operator->() {

    return _pointee;
  }

private:

  // wrapped plain pointer
  T* _pointee;
}
```

Der Template Parameter `T` ist dabei der Datentyp, welcher vom *smart pointer* referenziert werden soll.

Bei der Implementation sind verschiedene Design Aspekte. die sogenannten *policies*, zu beachten:

14.2.1 Ownership policy

Problem

Der C++ Standard betrachtet einen *pointer* als Zeiger auf ein Speichersegment, das so genannte *pointee*-Objekt. Im Standard nicht spezifiziert ist jedoch, wer Besitzer des *pointee* ist, das heißt, wer dieses verwaltet, das Speichersegment zum Beispiel wieder freigeben, deallozieren darf.

Verweist nur ein Zeiger auf ein Speichersegment, so ist es einfach dem Besitzer beziehungsweise Verwalter des Zeigers auch die Verantwortung für die korrekte Handhabung des Speichersegments zu übertragen.

Referenzieren allerdings mehrere Zeiger ein *pointee*, so entstehen die bereits angesprochenen Probleme. Dealloziert zum Beispiel ein Zeiger `ptr0` das *pointee*-Objekt, obwohl ein anderer Zeiger `ptr1` dieses noch verwendet, so wird `ptr1` auf ein invalidiertes Speichersegment zugreifen. Erfolgt andererseits die Deallokation von keinem von beiden, wird das Speichersegment also nie freigegeben; es entsteht ein *memory leak*.

Dieses Problem der Verantwortlichkeit oder des Besitztums wird als *ownership policy* bezeichnet. Zur Lösung des Problems existieren zwei Ansätze:

1. *Trivial Ownership*: Es wird verhindert, dass mehrere Zeiger das identische *pointee*-Objekt referenzieren.
   Dies impliziert allerdings, dass einige der wesentlichen Eigenschaften, welche die Effektivität von Zeigern ermöglichen, verloren gehen.

2. *Multiple Ownership / Reference Counting*: Es ist zulässig, dass mehrere Zeiger ein einziges *pointee*-Objekt referenzieren.

Bei der Lösung sind insbesondere die Konstruktion, das Kopieren und das Löschen von *smart pointern* zu berücksichtigen.

Trivial Ownership

Deep Copy Smart pointer mit einer *deep copy ownership policy* verhalten sich im Wesentlichen wie Stack Variablen, so wird zum Beispiel beim Kopieren des *smart pointer* das *pointee*-Objekt ebenfalls kopiert.

Die wesentlichen Unterschiede zu Stack Variablen sind:

- Der *smart pointer* kann für eine Basisklasse `T` definiert sein, jedoch eine Instanz einer abgeleiteten Klasse referenzieren.

- Das Objekt wird auf dem Heap und nicht auf dem Stack gehalten.

Probleme können durch *object slicing* auftreten. Um dies zu vermeiden, muss sichergestellt werden, dass im *copy constructor* des *smart pointer* tatsächlich die von `T` abgeleitete Klasse und nicht nur `T` kopiert wird (Abbildung 13 auf der nächsten Seite). Häufig wird dazu eine virtuelle Funktion `clone()` verwendet, die in jeder abgeleiteten Klasse neu implementiert wird.

Non-Copyable Bei der *non-copyable ownership policy* ist das Kopieren des *smart pointers* nicht möglich. Um allen Anforderungen an *smart pointer* zu Genügen, muss zusätzlich sichergestellt werden, dass *pointee* nur einmal initialisiert werden kann und das beim

```
// copy constructor of the smart pointer
// SLICING
Smart_Ptr<T>::Smart_Ptr(const Smart_Ptr<T>& ref):
  // call the copy constructor of class T
  _pointee( new T(ref._pointee) )
{ }
```

```
// copy constructor of the smart pointer
// NO SLICING
template<class U>
Smart_Ptr<T>::Smart_Ptr(const Smart_Ptr<U>& ref):
  // usage of the virtual clone function
  _pointee(ref->clone()))
{ }
```

```
int
main( int argc, char* argv[]) {

  // construct a smart pointer
  Smart_Ptr<a> ptr_1(new b());

  // copy construct
  Smart_Ptr<a> ptr_2(ptr_1);

  // slicing if copy constructor is not defined
  // appropriate: ptr_2 points to an object of
  // class a and  all specifica of class b
  // (e.g. _b) will be lost!

  ...
}
```

```
class a {
...

 virtual a* clone() {

  // call the copy
  // constructor of a
  return new a(*this)
 }

protected:

  int _a;
}
```

```
class b: public a {

...

 /*virtual*/ b* clone() {

   // call the copy
    // constructor of b
    return new b(*this)
  }

private:
  // int _a;

  int _b;
}
```

Abbildung 13: *object slicing*

Löschen des *smart pointer* auch *pointee* dealloziert wird. Dies kann gewährleistet werden, in dem die Deklaration des *copy constructors* und der Zuweisungsoperators `operator=()` *private* erfolgt, sowie der Destruktor entsprechend definiert wird.

Verwendung finden *smart pointer* mit einer *non-copyable ownership policy* in Funktionen mit mehreren *return*-Pfaden, da durch sie die Deallokation des Speichers beim Terminieren der Funktion gewährleistet werden kann.

Destructive Copy Die *destructive copy ownership policy* benutzt die Weitergabe des *pointee* von einem zum nächsten *smart pointer*, um zu verhindern, dass ein Speichersegment gleichzeitig von mehreren Zeigern referenziert wird.

Beim Kopieren übergibt der Ursprungs *smart pointer* seine Referenz an den neuen *smart pointer*. Der Ursprungs *smart pointer* wird anschließend invalidiert; meist indem der interne Zeiger auf Null gesetzt wird. Dies macht es, im Gegensatz zur üblichen Definition des *copy constructors*, notwendig, dass das dem *copy constructor* übergebene Argument nicht `const` ist. Dieser Weitergabe-Mechanismus beim Kopieren wird ebenfalls für den Zuweisungsoperator verwendet. Beim Löschen des *smart pointer* wird das *pointee*-Objekt dealloziert.

Probleme können bei der *destructive copy ownership policy* bei Funktion auftreten, welche einen *smart pointer* als Argument *by value* übergeben bekommen. In diesem Fall ist der übergebene *smart pointer* – entgegen der üblichen Funktionsweise bei der Übergabe *by value* – nach der Funktionsausführung invalidiert.

```
template<class T>
void
do_something(std::auto_ptr<T> ptr) {

  //
}

...

// construct a smart pointer with destructive copy
// ownership policy
std::auto_ptr<a> ptr(new a());

// the function takes the smart pointer argument
// by value
do_something(ptr);

// invalid operation: _pointee is 0 after execution
// of do_something()
int i = ptr->_a;
```

Der in der C++ Standard Bibliothek enthaltene `std::auto_ptr` verwendet die *destructive copy ownership policy*.

Eine mögliche Anwendung für die *destructive copy ownership policy* findet sich bei der Vermeidung von *memory leaks* bei Funktionen, deren Rückgabewert ein Zeiger ist.

Ein *memory leak* kann dort entstehen, wenn, was syntaktisch korrekt ist, der Rückgabewert keinem Zeiger außerhalb der Funktion zugewiesen wird. Mit der Terminierung der Funktion wird der Zeiger innerhalb dieser jedoch in jedem Falle vernichtet, was bei der Verwendung eines einfachen Zeigers zu einem unreferenzierten Speichersegement und damit zu einem *memory leak* führt.

Wird innerhalb der Funktion jedoch ein *smart pointer* mit *destructive copy ownership policy* verwendet, so ist durch dessen Destruktor gewährleistet, dass das *pointee* Objekt dealloziert wird, wenn der Rückgabewert außerhalb nicht weiterverwendet wird.

```
// construction of a temporary at the return statement
// if nobody takes the return value the temporary
// discards and by definition deletes it's _pointee
// if the return value is taken by a Smart_Ptr ptr the
// _pointee is transfered from the temporary to ptr and
// the pointee object is still available

template<class T>
Smart_Ptr<T>
do_something() {

  T* ptr = new T();

  // do somthing useful

  return Smart_Ptr<T>(ptr);
}
```

Reference Counting

Reference counting ist wohl die bekannteste *ownership policy* und erlaubt die mehrfache Referenzierung eines *pointee*-Objektes. Dabei wird sichergestellt, dass das *pointee*-Objekt genau dann gelöscht wird, wenn der letzte, das Objekt referenzierende *smart pointer* gelöscht wird.

Ein Problem bei allen Implementierungsstrategien für *reference counting* sind *cyclic references*. Dabei enthält der Graph, welcher durch die Referenzierungen von mindestens zwei Objekten gebildet wird, einen Zyklus (Abbildung 14 auf der nächsten Seite). Dieser verhindert die Deallokation der beteiligten Objekte, da für jedes Objektes zunächst ein

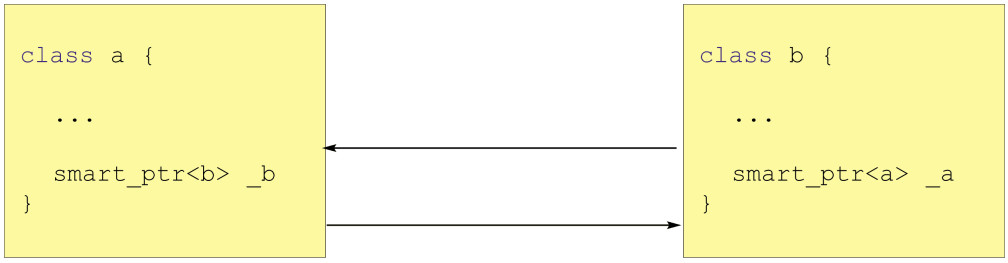
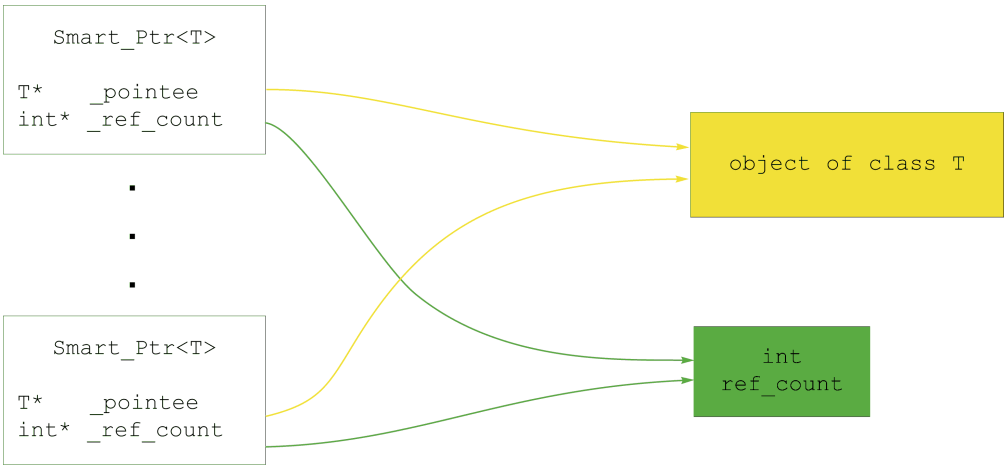


Abbildung 14: Beispiel für *cyclic references*

anderes freigegeben werden müsste. Neben binären *cyclic references* sind auch Zyklen mit mehreren beteiligten Objekte möglich. *Cyclic references* werden aufgelöst, in dem das *reference counting* zeitweise außer Kraft gesetzt wird.

Für die Implementation von *reference counting* existieren verschieden Strategien, welche sich in Bezug auf die Speicher- und Ausführungseffizienz unterscheiden:

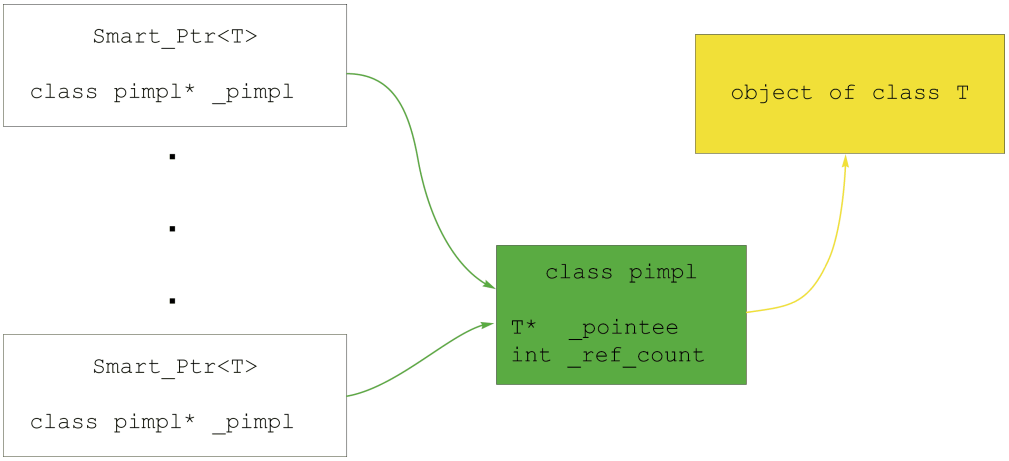
Reference counting und _pointee_ in getrennten Heap Objekten



- schneller Zugriff auf das *pointee*-Objekt möglich
- zusätzliches Heap Objekt für die *reference counting*-Variable notwendig
- *smart pointer* hat eine nicht-integrale Größe (Dies kann bei einer häufigen Erzeugung/Zerstörung von diesen gegebenenfalls zu einem Performanceverlust führen.)

Benutzung eines gemeinsames _pimpl_-Objektes

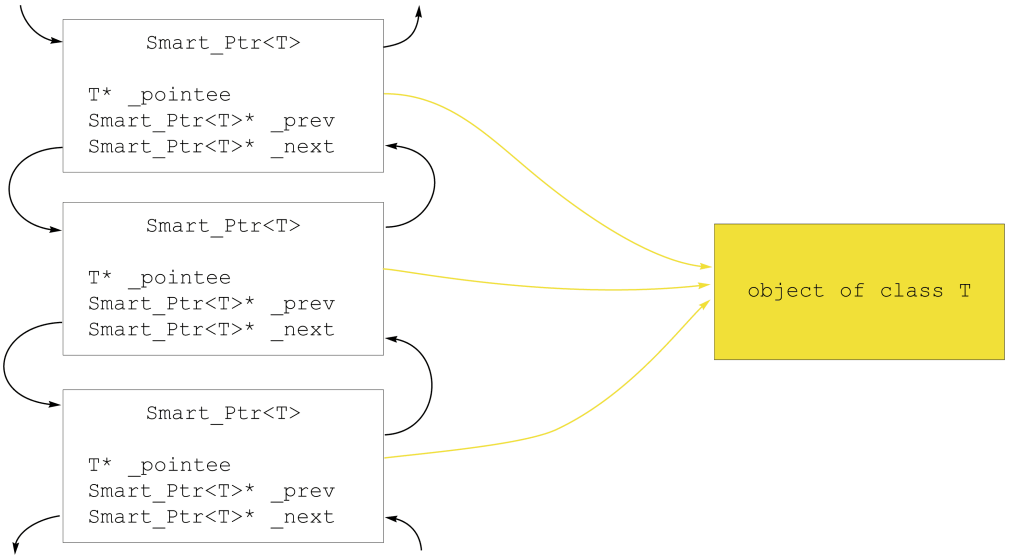
- alle *smart pointer*, welche *ein pointee*-Objekt referenzieren, benutzen das identische



pimpl-Objekt.

- smart pointer hat eine integrale Größe
- zusätzlicher Heap Speicher erforderlich
- zweifache Indirektion beim Zugriff auf *pointee*

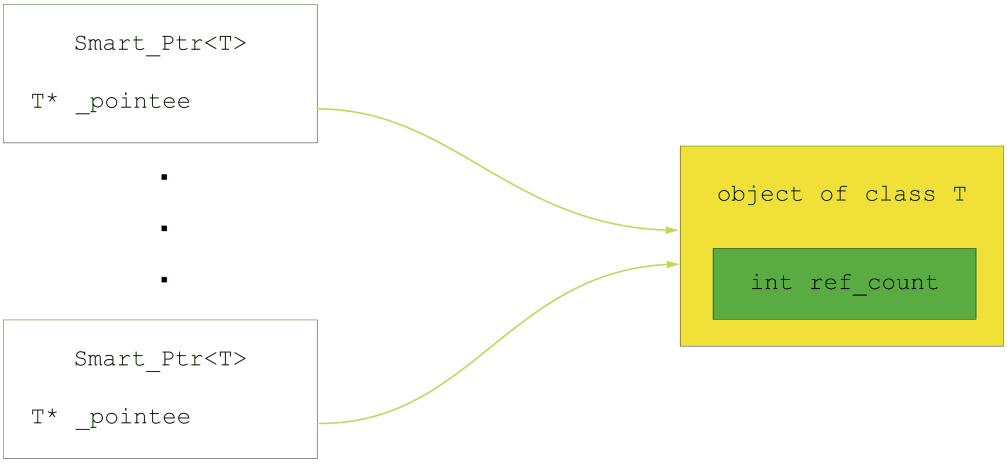
3. Verkettete Liste



- Idee: Für *reference counting* ist nicht die absolute Anzahl der ein Objekt referenzierenden *smart pointer* relevant, sondern es ist ausreichend, wenn festgestellt werden kann, wann der *reference count* Null wird.

- Besitzt ein smart pointer keinen Vorgänger und keinen Nachfolger in der Liste, so ist er der einzige der das *pointee*-Objekt noch referenziert und muss bei seiner Destruktion das *pointee*-Objekt deallozieren.
- erheblicher Aufwand zur Verwaltung der verketteten Liste notwendig
- kein zusätzlicher Heap Speicher erforderlich

Intrusive reference counting



- das *pointee*-Objekt selbst enthält die *reference counting*-Variable.
- die Klasse `T` muss unter Berücksichtigung des *intrusive reference counting* erstellt werden
- zusätzliches Heap Speicher erforderlich
- direkter Zugriff auf *pointee*

14.2.2 Storage Policy

Die *storage policy* beschreibt die Datentypen für die der wesentlichen Objekte in einem *smart pointer*:

- *storage type*
  - der Datentyp der Membervariablen

  - meist T*

- pointer type
  - der Datentyp des Rückgabewertes von `operator->()`
  - meist T*
- reference type
  - der Datentyp des Rückgabewertes von `operator*()`
  - meist &T

Eine von den üblichen Datentypen abweichende *storage policy* kann zum Beispiel eingesetzt werden, um einen Lock Mechanismus für multi-threading Umgebungen zu realisieren.

Dabei kann eine Besonderheit des Operators `operator->()` ausgenutzt werden. Wird dieser aufgerufen, so verlangt der C++ Standard, dass der Rückgabewert ein *plain datatype* ist. Kann dies nicht nach dem einmaligen Aufruf gewährleistet werden, zum Beispiel da ein *smart pointer* als *pointer type* nicht `T*` verwendet, so erfolgt durch das System ein rekursiver Aufruf, bis tatsächlich ein plain datatype zurückgegeben wird.

Für den Lock Mechanismus ist es damit möglich, als Rückgabewert des Operators `operator->()` der *smart pointer*-Klasse ein *lock proxy*-Objekt – welches kein *plain datatype* ist – zu definieren und trotzdem sicherzustellen, dass bei einem Aufruf von `operator->()` auf dem *smart pointer* ein einfacher Zeiger vom Typ `T*` der effektive Rückgabewert ist.

Die *lock proxy*-Klasse muss dazu den `operator->()` mit `T*` als Rückgabewert implementieren.

```
// sketch for a smart pointer class supporting
// lock proxys
template<class T>
class Smart_Ptr {

  ...

  Lock_Proxy<T> operartor->() {

    return Lock_Proxy<T>(_pointee);
  }
```

```
private:

  T* _pointee;
}


// a simple lock proxy class
template<class T>
class Lock_Proxy {

public:

  Lock_Proxy (T* pointee):
   _pointee(pointee)
  {

    _pointee->Lock()
  }

  ~Lock_Proxy() {

    _pointee->Unlock()
  }

  operartor->() {

    return _pointee;
  }

private:

  T* _pointee;
}

// usage of a smart pointer supporting lock proxys
// the lock mechanism is totally hidden before the user
int
main( int argc, char* argv[]) {

  ...

  // construct smart pointer class a
  Smart_Ptr<a> ptr;

  // a temporary of class Lock_Proxy
  // is created locking the resource
  // the proxy is deleted directly
  // after the execution of
  // singleton_fct returned
```

```
  ptr->singleton_fct

  ...
}
```

Bei einem Aufruf des `operator->()` des *smart pointer* wird zunächst eine Instanz von *lock proxy* erzeugt und durch den Konstruktor der Klasse die Ressource gelockt. Da *lock proxy* kein *plain datatype* ist, wird der Operator `operator->()` erneut – nun auf der Instanz von *lock proxy* – aufgerufen, und der tatsächliche Rückgabewert `T*` zurückgegeben. Nachdem der Aufruf auf `T*` abgeschlossen ist, zum Beispiel eine Funktion auf *pointee* ausgeführt wurde und terminiert hat, endet der Funktionsaufruf des `operator->()` des *smart pointer*. Da keine Zuweisung des Rückgabewert der Funktion im Programm stattfindet, wird die Instanz von *lock proxy* zerstört um im Destruktor die Ressource wieder frei gegeben wird.

14.2.3 Überladen von Operatoren

`operator->()` und `operator*()`

Die Überladung der beiden Standard-Operatoren auf Zeigern, `operator->()` und `operator*()`, ist in den meisten Fällen obligatorisch. Dabei sollten auch die `const`-Varianten implementiert werden, da diese dem Compiler einen wesentlich höheren Optimierungsgrad erlauben und damit ein effizienteres Programm ermöglichen. Der Dereferenzierungsoperator sollte eine Referenz zurückgegeben, da sonst *object slicing* auftreten kann.

```
// simple access operator
T* operator->() {

  return _pointee;
}

// simple dereferencing operator
T& operator*() {

    return *_pointee;
}
```

Eine der Ausnahmen bei denen der Dereferenzierungsoperator oft nicht implementiert wird, sind Systemressourcen. Dort ist oft nur der Zugriff, aber nicht die Dereferenzierung erlaubt

`adress-of operator&()`

Der `adress-of operator&()` wird in einer default Variante bereits automatisch durch das System zur Verfügung gestellt und hat die Adresse der Instanz der Klasse, hier des *smart pointer*, als Rückgabewert. Diese default Version sollte nicht überschrieben werden, da der *smart pointer* sonst nicht mehr mit großen Teilen der *Standard Template Library (STL)* verwendet werden kann, wo aus Effizienzgründen das auf dem default `adress-of operator&` arbeitende `memcpy()` zum Einsatz kommt.

Eine der Intuition entsprechende Implementation des `adress-of operator&()`, welche die Adresse des *pointee*-Objektes als Rückgabewert hat, würde zusätzlich die *ownership policy* gefährden, da damit wieder der direkte Zugriff auf das *pointee*-Objekt möglich wäre.

`cast operator T*()`

Der `cast operator T*()`, welcher für einen *smart pointer* `smart_ptr<T>` den einfachen Zeiger `T*` auf das *pointee*-Objekt zurückgibt, sollte nicht zur Verfügung gestellt werden.

Der Operator würden implizite – und häufig unbemerkte – Typ-Konvertierungen ermöglichen, was zu unerwarteten Verhalten beim Compileren und bei der Ausführung des Programms führen könnte. Es wäre in diesem Fall zum Beispiel syntaktisch korrekt, `delete` auf einem *smart pointer* `ptr` aufzurufen, obwohl `ptr` ein Stack Objekt ist. Durch die implizite Konvertierung würde `ptr` in `T*` konvertiert und damit das *pointee*-Objekt dealloziert.

```
Smart_Ptr<a> ptr;

// implicit cast from ptr to T*
delete ptr;
```

Um dennoch einen Zugriff auf den einfachen Zeiger `T*` zu ermöglich, kann eine separate Funktion implementiert werden.

```
template<class T>
T* get_ptr(Smart_Ptr<T>& ref) {

  return ref._pointee;
}
```

Diese sollte allerdings nur mit äußerster Vorsicht eingesetzt werden, um nicht die Integrität des *smart pointer*, zum Beispiel seine *ownership policy*, zu gefährden.

Gleichheits- und Ungleichheitsoperatoren

Drei Vergleichsoperationen sollten von smart pointer unterstützt werden:

1. Vergleich zweier *smart pointer*
2. Vergleich eines *smart pointer* mit einem *plain pointer*
3. Vergleich mit 0 (Test ob der *smart pointer* auf ein gültiges *pointee*-Objekt zeigt oder invalidiert ist)

Die Erläuterungen beziehen sich dabei jeweils auf den Gleichheitsoperator `operator==()`, können aber ohne weiteres auf die anderen relevanten Operatoren übertragen werden

1. *smart pointer – smart pointer*

Bei der Implementation der Vergleichsoperatoren für zwei *smart pointer* ist zu beachten, dass die *smart pointer* nicht auf derselben Klasse `T` definiert sein müssen, damit ein Vergleich sinnvoll ist.i So sollte der Vergleich zum Beispiel für die beiden Klassen a und b (Abbildung 13 auf Seite 126) möglich sein:

```
b* ptr = new b();
Smart_Ptr<a> smart_ptr_1(ptr);
Smart_Ptr<b> smart_ptr_2(ptr);

...

if (smart_ptr_1 == smart_ptr_2) {

}
```

Dazu muss bei der Implementation des Gleichheitsoperators einen zusätzlicher Template Parameter verwendet werden.

```
template<class T>
class Smart_Ptr {

  ...

  template<class U>
  bool
  operator==(const Smart_Ptr<U>& rhs) {

    return _pointee == rhs._pointee;
  }
};
```

2. *smart pointer - plain pointer*

Für den Vergleich zwischen einem *smart pointer* und einem einfachen Zeiger werden die binären Vergleichsoperatoren verwendet, die als `friend` zur *smart pointer*-Klasse definiert werden. Es ist zu beachten, dass jeweils eine Funktion für den Vergleich mit dem einfachen Pointer auf der linken Seite und eine, für den einfachen Pointer auf der rechten Seite des Operators vorhanden sein muss.

```
template<class T>
class Smart_Ptr {

  template<class U>
  inline friend bool
  operator==(const Smart_Ptr& lhs, const U* rhs) {

    return lhs._pointee == rhs;
  }
};
```

Validitätstest für *smart pointer*

Um einen einfachen Zeiger auf Validität zu testen, kann dieser direkt als boolsches Argument des *if-statements* verwendet werden.

```
if (ptr) {

    ...
}
```

Um dies auch mit einem *smart pointer* zu ermöglichen müssen die beiden boolschen Operatoren `cast operator bool()` und `operator !()` implementiert werden.

```
template<class T>
class Smart_Ptr {

  operator bool() const {

    return _pointee != 0;
  }

  bool
  operator !() const {

    return _pointee != 0;
  }
}
```

Zusätzlich wird oft auch der direkte Vergleich mit 0 ermöglicht

```
if (ptr == 0) {

  ...
}
```

Die bereits erläuterte Template-basierte Implementation für den Vergleich eines *smart pointer* mit einem einfachen Zeiger kann dazu allerdings nicht verwendet werden. Der Compiler erzeugt zwar für jede Klasse `U`, für welcher der Operator benötigt wird, eine Spezialisierung, nicht jedoch für `0`, da dies keine Klasse ist. Wird im Programm keine Vergleich mit einem *smart pointer* auf einer echten Klasse `U` durchgeführt, so steht der Vergleichsoperator damit auch nicht zur Verfügung und der Test mit `0` erzeugt einen *compile-time error*.

Aus diesem Grund muss, neben dem Template-basierten binären Vergleichsoperator auch eine nicht Template-basierte Version implementiert werden. Diese wird bereits mit der Spezialisierung der *smart pointer*-Klasse für `T` instantiiert und steht somit in jedem Fall zur Verfügung.

```
template<class T>
class Smart_Ptr {

  inline friend bool
  operator==(const Smart_Ptr& lhs, const T* rhs) {

    return lhs._pointee == rhs;
  }
};
```

Ordnungsoperatoren

Soll ein *smart pointer* auf einem Array eingesetzt werden können, so müssen die Ordnungsoperatoren implementiert werden. Dabei sollten, die bereits für die Vergleichsoperatoren diskutierten Probleme berücksichtigt werden.

Die Implementation von `std::less` für *smart pointer* ermöglicht die Verwendung dieser als `key`-Element in einer `map`. Dabei sollte die Adresse des *smart pointer* als Ordnungskriterium dienen, was in vielen Fällen, durch eine erhöhte Speicherkoherenz, die Effektivität Zugriffs auf Elemente der `map` erhöht. Zur Verwendung muss die Spezialisierung von `std::less` dem `namespace std` hinzugefügt werden.

14.2.4 Memberfunktionen

Smart pointer sollten keine Memberfunktionen zur Verfügung stellen. Die semantische Nähe zwischen der Ausführung einer Memberfunktion des *smart pointer* und der Ausführung einer Memberfunktion auf dem *pointee*-Objekt wäre dabei zu groß, was die Lesbarkeit des Programmcodes unnötig erschwert.

```
Smart_Ptr<a> ptr;

// hard to distinguish
ptr.release();
ptr->release();
```

Stattdessen werden für die notwendigen Operationen auf einem *smart pointer* `friend`-Funktionen zur Verfügung gestellt. Sinnvoll ist es diese in einem gemeinsamen *namespace* zusammenzufassen.

```
namespace Smart_Ptr_Helper {

  template<class T>
  bool
  release(Smart_Ptr<T> & ptr) {

  ...
  }
}
```

14.3 Implementationen

Es existieren verschiedene Implementation für smart pointer, welche zahlreiche der vorgestellten, möglichen Varianten abdecken.

14.3.1 std::auto_ptr

- destructive ownership policy
- kann nicht mit *STL containern* verwendet werden

14.3.2 boost::scoped_ptr

- non copyable ownership policy
- heap object

```
void
do_something() {

  boost::scoped_ptr ptr(new a());

  // do something useful

  // scoped_ptr and the pointee object
  // are deleted here
}
```

14.3.3 boost::shared_ptr

- implements reference counting
- kann in allen Teilen der *STL* eingesetzt werden (*copy-constructable, assignable, std::less*)

```
typedef boost::shared_ptr<Foo> Foo_Ptr;

std::vector<Foo_Ptr> vec;

...

  {
    Foo_Ptr ptr_1(new Foo);

    std::cout << ptr_1.use_count() << '\n';

    vec.push_back(ptr_1);

    // ptr_1 goes out of scope here and is detroyed
  }

// the Foo of ptr_1 is still alive while it is
// stored in the vector and by this the
// use_count==1 (means > 0)
```

14.3.4 boost::weak_ptr

Die `boost::weak_ptr`-Klasse ist eine Hilfsklasse für `boost::shared_ptr` und ermöglicht es *cyclic references* zu vermeiden. Dazu werden die Instanzen von `boost::weak_ptr` nicht für das *reference counting* berücksichtigt. Um dennoch sicherzustellen, dass keine *memory leaks* entstehen bzw. kein Zugriff auf invalidierten Speicher stattfindet, ist der Zugriffsoperator `operator->()` für `boost::weak_ptr` nicht implementiert, so dass das *pointee*-Objekt nicht direkt genutzt werden kann. Dies ist indirekt mit Hilfe der Funktion `lock()` möglich, welchen einen `boost::shared_ptr` als Rückgabewert hat oder mit einem „copy-constructor“ von `boost::shared_ptr`, welcher als Argument einen `boost::weak_ptr` verlangt.

14.3.5 boost::intrusive_ptr

- *intrusive reference counting*, das heißt das *pointee*-Objekt selbst ist für das reference counting verantwortlich
- *pointee* muss `intrusive_ptr_add_ref()` und `intrusive_ptr_release()` implementieren um das *reference counting* zu ermöglichen

14.3.6 boost::*_array

Alle *smart pointer* der *boost*-Bibliothek sind ebenfalls als Array Varianten implementiert. Bei wird dort zusätzlich `operator[]()` und `delete[]()` zur Verfügung gestellt.

14.3.7 Loki::SmartPtr

```
template
    <
        typename T,
        template <class> class OwnershipPolicy,
        class ConversionPolicy,
        template <class> class CheckingPolicy,
        template <class> class StoragePolicy
    >
    class SmartPtr
```

Andrei Alexandrescu ([Alexandescu'01]) stellt innerhalb seiner Loki Bibliothek eine vollständig Template-basierte *smart pointer*-Klasse zur Verfügung. Bei dieser können, neben der *ow-*

nership und der *storage policy*, auch die *conversion policy* und die *checking policy* durch die Template Parameter spezifiziert werden. Die *conversion policy* ermöglicht es den `cast operator T*()` des *smart pointers* zur Verfügung zu stellen; die *checking policy* definiert, ob vor der Dereferenzierung eines *smart pointer* zunächst überprüft werden soll, ob das *pointee*-Objekt invalidiert ist.

14.4 Zusammenfassung

Smart pointer sind ein geeignetes Mittel um die Probleme und Gefahren bei der Verwendung von einfachen Zeigern zu vermeiden. Die Implementation einer eigenen *smart pointer*-Klasse ist dabei jedoch mit zahlreichen Herausforderungen verbunden.

Für die Verwendung von *smart pointern* sollte auf eine der bestehenden Implementationen, allen voran aus der *boost*-Bibliothek, zurückgegriffen werden. Dabei ist der Nutzen durch die gewonnene Abstraktion gegen den entstehenden Overhead abzuwiegen.

Nach Angaben von Herb Sutter werden der `boost::shared_ptr` und der `boost::weak_ptr` im nächsten C++ Standard enthalten sein und damit zukünftig auch in der C++ Standard Bibliothek zur Verfügung stehen.

15 Case Insensitive String and Char Traits

[Stroustrup'00] [boost] [GotW #29]

15.1 Einleitung

Folgender Abschnitt erläutert die Problematik eines case insensitive Strings und zeigt einige Lösungsbeispiele auf. Zentrale Betrachtung fällt dabei den character traits zu.

15.1.1 Was heißt „case-sensitive"?

Man möchten erreichen das Großbuchstaben genauso betrachtet werden wie Kleinbuchstaben und ein Vergleich von A und a gleich „wahr" ergibt. In den meisten modernen Programmiersprachen wir jedoch case-sensitive gearbeitet das heißt:

```
("ABCD"=="abcd")     // false
("ABCD"=="ABCD")     // true
```

Der Sinn hinter einem solchen case-insensitiven Vergleich könnte zum Beispiel sein, Nutzereingaben zu vergleichen oder Fehlertolleranter zu arbeiten.

15.1.2 Triviale Lösung

Eine erste simple Lösung kann unter Benutzung von cstrings und der Funktion strcimp entstehen.

Triviale Lösung mit stricmp

```
#include <cstring> // stricmp()
#include <string>
  std::string s1 == "abcde";
  std::string s2 == "ABCD";
  inline
  bool stricmp (const std::string &s1,
                const std::string &s2){
       return stricmp (s1.c_str(), s2.c_str())==0;
  }
  bool b = stricmp(s1,s2); // true
```

Problem hierbei ist, dass viele Funktionen die auf Strings definiert sind (z. B. `compare()`, `find()`, ...), weiter case-sensitive arbeiten. Ein weiteres Problem ist, dass die C-Funktion stricmp kein Teil des C-Standards ist (sie ist aber gebräuchliche Erweiterung bei den meisten C-Compilern z. B. `gcc` oder `icc`).

15.2 Klasse cistring

Aus den oben genannten Problemen definieren wir unseren neuen Lösungsansatz: Ziel ist also das Schreiben einer string Klasse cistring mit der Funktionalität von string aber case-insensitiv (wie die Funktion stricmp für cstrings).

15.2.1 std::string

Werfen wir ein Blick auf auf die Implementierung von std::string

```
typedef basic_string<char> string;
typedef basic_string<wchar_t> wstring;
```

std::string ist also ein typedef und keine Klasse! Der typedef basiert auf der Templateklasse:

```
template<class charT,
         class traits = char_traits<charT>,
         class Allocator = allocator<charT> >
         class basic_string; {
public:

//...

}
```

Das heißt, mit std::string meinen wir eigentlich

```
basic_string<char,char_traits<char>,allocator<char>>
```

Kommen wir nun zu den character traits.

15.2.2 character traits

Im Blickpunkt unser nun folgenden Betrachtungen stehen dabei die character traits (deut.: Zeichenmerkmale) da dort das Verhalten der Zeichen definiert ist. `char_traits`

ist dabei eine Templateklasse welche keine Eigenschaften in der allgemeinen Form besitzt sondern welche erst bei der Spezialisierung definiert werden.

```
template<class Ch> struct char_traits {  };

char_traits<char>
  template<> struct char_traits<char> {

  typedef char char_type;

  static
  void assign (char_type&, const char_type&);

  typedef int int_type;

  static
  int_type to_int_type (const char_type&);

  static
  bool eq (const char_type&, const char_type&);

    ..
}
```

Wir wir sehen, sind hier also die Funktionen definiert, die für eine Umwandlung von einem Character Datentyp in einen Integer Datentyp nötig sind aber auch die für unser Vorhaben wichtigen Vergleichsfunktionen von Charactertypen untereinander. Den diese Vergleichsfunktionen bilden die Basis auf denen die Funktionen wie ==, ≪, ≫, find, replace und compare aus `basic_string` definiert sind.

Das heißt die Lösung unseres Problems liegt in den `char_traits` da die Funtionen aus `basic_string` darauf aufbauen. Die `char_traits` bieten dabei folgende Funktionalität für den Zeichenvergleich:

- eq() equal, liefert true bei Gleichheit der Zeichen
- ne() not eqaul, liefert false bei Gleichheit der Zeichen
- lt() less then, liefert true wenn das Erste Zeichen kleiner ist als das Zweite
- find() sucht ein Zeichen
- compare() vergleicht Zeichen

Es ist also nötig diese Funktionen an unseren gewünschte Funktionalität (case-insensitive) anzupassen. Dies kann geschehen durch Ableiten von der Klasse `std::char_traits<char>`

und bereitstellen der entsprechenden Funktionen.

```
struct case_insens_traits :
  public std::char_traits<char>{

static bool eq(char c1, char c2){
  return std::toupper(c1)==std::toupper(c2);
}

static bool ne(char c1, char c2){
  return toupper(c1) != toupper(c2);
}

static bool lt(char c1, char c2){
  return std::toupper(c1)<std::toupper(c2);
}

static const char* find(const char* s,std::size_t n,
                        char c) {
  for (std::size_t i=0; i<n; ++i) {
    if (eq(s[i],c)) {
      return &(s[i]);
    }
  }
  return 0;

static int compare( const char* s1,
                    const char* s2,
                    std::size_t n ){
  return memicmp( s1, s2, n );
}
}
```

Diese Implementation stellt nun die Basis für unsere neue String Klasse dar.

15.2.3 Klasse cistring

Nachdem wir uns eine Klasse case_insens_traits geschaffen und diese an unsere Bedürfnisse angepasst haben folgt nun die Definition unseres cistrings

```
typedef std::basic_string<char,
             case_insens_traits> cistring;
```

Unsere Resultat ist nun ein typedef cistring der sich genauso verhält wie std::string nur mit unseren case_insens_traits statt char_traits≪char≫ um case-insensitiv zu arbeiten. Alle Funktionen, die auf eine std::string anwendbar sind (wie z. B. find, replace und

compare,) arbeiten auch auf cistring nur sind sie case-insensitive. Einige Probleme haben wir allerdings bei der Betrachtung bis jetzt ausgelassen. Was passiert zum Beispiel bei der Ausgabe unseres cistring oder bei einer Wertzuweisung zu std::string? Man könnte sich also folgendes Programmfragment vorstellen:

```
cistring a ("Hallo");
std::string b;
std::cout<< a <<std::endl; //was passiert hier?
b=a;                       //geht das?
```

Betrachten wir den Ausgabeoperator für std::string: cout ist ein:

```
basic_ostream<char, char_traits<char> >
```

```
<class charT, class traits, class Allocator>
                basic_ostream<charT, traits>&
operator<<(basic_ostream<charT, traits>& os,
           const basic_string
           <charT,traits,Allocator>& str);
```

Für den Ausgabeoperator existiert aber nur eine Spezialisierung für den selben char Typ und traits Typ wie std::string. Deshalb ist es ohne weiter Überlegungen nicht möglich den cistring über ein einfaches cout auszugeben. Eine Lösung dieses Problem ist jedoch leicht zu implementieren. Zum einen gibt es die Möglichkeit den cistring in eine cstring zu konvertieren und diesen auszugeben (Lösung 1) zum anderen gibt es die Möglichkeit unseren cistring in eine std::string zu konvertieren (z. B. wie in Lösung 2 kombiniert mit einem neuen Ausgabeoperator für cistring).

Lösung 1:

```
cout << s.c_str() << endl;
```

Lösung 2:

```
std::ostream& operator << (std::ostream& out,
                           cistring const& s) {
  return out << std::string(s.data(),s.length());
}
```

Bei Lösung 2 ist aber zu beachten wie der cistring intern dargestellt wird. Denn er besteht sowohl aus Klein- als auch Großbuchstaben. Nur wird es bei dem Vergleich nicht beachtet! Die Lösung zu dem Problem der Zuweisung an einen std::string erfolgt analog indem man unseren cistring erst in eine cstring oder string konvertiert. Unser oben gezeigte Beispielprogamm könnte also korrekt so aussehen:

```
cistring a("Hallo");
std::string b;
std::cout<< a.c_str() <<std::endl;
b=std::string(a.data(),a.lenght());
```

Eine komplette Implementierung des hier vorgestellten cistring mit einem Beispiel gibt es unter http://www.uni-weimar.de/~langlotz

15.3 Andere traits-Konzepte

Es gibt eine Vielzahl vo Traitskonzepten wie zum Beispiel Iterator Traits. Ein anderes Traitskonzept was hier noch kurz vorgstellt werden soll sind die Type Traits aus der Boots Libary. Um sie zu benutzen muss der Boost-Header `<boost/type_traits.hpp>` inkludiert werden. Es gibt aber noch eine Vielzahl von anderen Headern die je nach der Anwendung mit eingebunden werden müssen. Die Boost Bibliothek unterscheidet dabei drei Typen von type traits:

- Eigenschaften eines bestimmten Typs
- Eigenschaften zwischen 2 Typen
- Typtransformationen

Traits des ersten Typs können genutzt werden um Typinformationen und Eigenschaften eines Datentypen zu erlangen. Beispiele hierfür sind die Methoden:

```
::boost::is_array<T>::value
::boost::is_enum<T>::value
::boost::is_const<T>::value
::boost::is_pod<T>::value
```

Der zweite Traittyp gestattet Informationen und Beziehungen zwischen zwei Datentypentypen zu erlangen. Zum Beispiel:

```
::boost::is_base_and_derived<T,U>::value
```

liefert ob U von T abgeleitet ist. Also ob T Basisklasse von U ist. Die Anwendung zeigt folgendes Codebeispiel:

```
struct A {};
struct B : A {};
bool const x=boost::is_base_and_derived<A,
                                        B>::value;
// true
```

Der letzte Traitstyp liefert Typtransformationen. Ein Beispiel hierfür:

```
::boost::remove_const<T>::type
::boost::add_const<T>:
```

Diese Funktionen entfernen oder fügen das const zu einem Datentypen hinzu. So ist es Möglich mit Traits diesen Typs einen Datentypen auf seinen absoluten Basistypen zu transformieren.

Diese in der Boost Bibliothek implementierten Type Traits ist eine Grundlage für viele weiter Programmierkonzepte, wie z. B. Template Meta Programming.

16 Member Templates

[Meyers'97]

16.1 Definition

Ein Template, das innerhalb einer Klasse oder eines Klassentemplates deklariert ist heißt member template.

```
class CtypeSize
{
 public:
     //member template:
     template<class T>
     CtypeSize(const T &t1) : m_nSize(sizeof(t1))
     {
     }

     ~CtypeSize(void) { };

     int getSize(void) const {return m_nSize;}

 private:
     const int m_nSize;
};
```

Wird ein member template aufgerufen erzeugt der Kompiler den Code für den entsprechenden Typ.

```
//Displays 12.
CtypeSize t1("Hello world");
cout << t1.getSize() << endl;

//Displays 8 on VC++ 6 / win32.
CtypeSize t2(7.0);
cout << t2.getSize() <<endl;
```

Ein member template kann inner- oder außerhalb der Klassendefinition oder Templateklassendefinition deklariert werden. Wird es außerhalb deklariert sollte es mit den template-Parametern der Templateklasse gefolgt von den template-Parametern des member templates spezifiziert werden.

```
template<class T> class string
{
 public:
```

```
        template<class T2> int compare(const T2&);

        template<class T2> string(const string<T2>& s)
        { /* ... */}
        //...
};

//zuerst Parameter Templateklasse, dann Parameter member template.
template<class T> template<class T2>
int string<T>::compare(const T2& s)
{
        //...
}
```

16.2 Copy-Konstruktor als member template

Manchmal sind member templates der effizienteste Weg copy - Konstruktoren zu implementieren.

Folgende Klasse:

```
template<class T> class CSingle
{
 public:
        CSingle(const T &t1) : mValue(t1) {}
        ~CSingle(void) {}

        T m_Value;
};
```

Ein Copy - Konstruktor wird benötigt, wenn ein Objekt mit dem Zustand eines anderen Objekts erzeugt werden soll.

```
//Create an integer container
CSingle<int> x(7);

//this needs a copy constructor ...
CSingle<double> y(x);
```

```
Mit member template ist der copy - Konstruktor ein einfaches:

\begin{lstlisting}
   template<class S>
   CSingle(const CSingle<S> &s1 : m_Value(s1.mValue)
   {}
```

Solange der Kompiler ein Objekt von Typ T in ein Objekt vom Typ S konvertieren kann, wird das Objekt erfolgreich erzeugt.

Aber vorsicht! Der Kompiler verwendet ein member template niemals um einen impliziten copy-konstruktor zu generieren. Wenn eine Klasse also einen benutzerdefinierten copy Konstruktor benötigt, ist er als nichtmember-template zu definieren.

Dennoch werden member templates zur Überladungsauflösung herangezogen. Passt das member template besser als der implizite Copy-Konstruktor um ein Objekt zu kopieren wird das template ausgewählt.

```
class Base
{
 public:
      Base() {}

      //member constructor.
      template<class T> Base(T&);
};

class Derived : public Base
{
};

Base b1;
Derived d1;

//uses implicitly generated constructor Base(const Base&).
Base b2(b1);
//uses template member constructor Base (const Derived&).
Base b3(d1);
```

16.3 member templates im Standard

Der Standard schreibt für member templates einige Restriktionen vor:

- Eine lokale Klasse darf keine member templates besitzen.
- Der Destruktor darf kein member template sein.
- Ein member template darf nicht virtual sein.

Darüberhinaus heißt es weiter: Eine normale member function (Name + Typ) und ein member template mit dem gleichen Namen und der Möglichkeit eine Spezialisierung mit

dem selben Typ können beide in der Klasse deklariert werden:

```
template<class T> struct A
{
      void f(int);
      template <class T2> void f/T2);
};
// non template member.
template<> void A<int>::f(int) { }

// template member.
template<> template<> void A<int>::f<>(int) { }

int main()
{
      A<char> ac;

      ac.f(1);   //non - template.
      ac.f('c'); //template.
      ac.f<>(1); //template.
}
```

Eine Spezialisierung eines member templates überschreibt niemals eine virtuelle Funktion der Basisklasse.

```
class B
{
      virtual void f(int);
};

class D : public B
{
      //does not override B::f(int).
      template<class TA> void f(T);

      //overriding function that calls the template instantiation.
      void f(int i)
      {
              f<>(i);}
      }
};
```

16.4 Beispiel: smart pointer und member templates

Gegeben sei ein öffentliche Vererbungsstruktur, ...

```
class MusicProduct
```

```
{
 public:
      MusicPrduct(const string& title);
      virtual void play() const = 0;
      virtual void displayTitle() const = 0;
      ...
};

class Cassette : public MusicProduct
{
 public:
      Cassette(const string& title);
      /*virtual*/ void play() const;
      /*virtual*/ void displayTitle() const;
      ...
};

class CD : public MusicProduct
{
 public:
      CD(const string& title);
      /*virtual*/ void play() const;
      /*virtual*/ void displayTitle() const;
      ...
};
```

... und eine Funktion (`diplayAndPlay`) mit (`MusicProduct`) als Parameter. Die Funktion gibt Titel aus und spielt ihn.

```
void displayAndPlay(const MusicProduct * pmp, int numTimes)
{
      for (int i = 0; i<= numTimes; ++i)
      {
              pmp->displayTitle();
              pmp->play();
      }
}
```

Anwendung:

```
//Erzeuge zwei Objekte, MusicProduct als Parent habend.
Cassette * funMusic = new Cassette("Schlachtrufe BRD");
CD * nightmareMusic = new CD("Disco Hits of the 80s");

//Cassette und CD werden als MusicProduct akzeptiert.
displayAndPlay(funMusic, 10);
displayAndPlay(nightmareMusic, 10);
```

Keine Überraschungen hier. Geht das nun auch mit „intelligenten“ smart pointern?

Neue Funktion (`displayAndPlay`) für smart pointer:

```
void displayAndPlay( const SmartPtr<MusicProdukt>& pmp, int numTimes);
```

Anwendung:

```
//wieder zwei Objekte, diesmal über smartPointer gehalten.
SmartPtr<Cassette> funMusic(new Cassette("Schlachtrufe BRD"));
SmartPtr<CD> nightmareMusic (new CD("Disco Hits of the 80s"));

//Compiler sieht keinen Zusammenhang zwischen SmartPointer<MusicProduct> un
displayAndPlay(funMusic,10);          //ERROR!
displayAndPlay(nightmareMusic, 10);   //ERROR!
```

Grund für den error: Es gibt keine Umwandlungen von (`SmartPtr<CD>`) oder (`SmartPtr<Cassette>`) nach (`SmartPtr<MusicProduct>`), da keine Vererbungsbehziehung zwischen diesen besteht!

Lösung: Umwandlungsoperator mit member template:

```
template<class T>
class SmartPtr
{
 public:
      SmartPtr(T* realPtr = 0);

      T* operator->() const;
      T& operator*() const;

      //member template für implizite Umwandlung.
      template<class newType>
      operator SmartPtr<newType>()
      {
              return SmartPtr<newType>(pointee)
      }
      ...
};
```

Mit dieser smart pointer Klasse wird der code erfolgreich übersetzt, da nun (`SmartPtr<CD>`) und (`SmartPtr<Cassette>`) in (`SmartPtr<MusicProduct>`) umgewandelt werden können.

Was aber passiert genau? Der Compiler entdeckt die fehlende Typübereinstimmung (`funMusic`) ¡-¿ (`MusicProduct`). Daher sucht er zunächst nach einem Konstruktor in der Klasse (`SmartPtr<MusicProduct>`) der einen (`SmartPtr<Cassette>`) übergibt. Doch Fehlanzeige... Danach wird ein Operator für implizite Typumwandlung in der Klasse

(SmartPtr<Cassette>) gesucht der einen (SmartPtr<MusicProduct>) ergibt, aber auch hier Fehlanzeige. Schließlich wird ein member template gesucht und gefunden, das eine der obigen Funktionen instanziieren kann. Das member template innerhalb (SmartPtr<Cassette>) kann die notwendige Funktion erzeugen, wenn (newType) an (MusicProduct) gebunden wird:

```
SmartPtr<Cassette>:: operator SmartPtr<MusicProduct>()
{
     return SmartPtr<MusicProduct(pointee);
}
```

So bleibt nun mehr die Frage, ob ein (SmartPtr<MusicProduct>) mit einem Zeiger vom Typ (Cassette*) erzeugt werden kann. Der Konstruktor erwartet einen Zeiger vom Typ (MusicProduct*) und auf Grund der Vererbungsbeziehung kann (Cassette*) anstelle dessen verwendet werden.

17 Template Metaprogramming

17.1 Motivation

Template Metaprogramming ermöglicht es, zur Kompilierzeit Berechnungen auszuführen, das Verhalten von Typen abzufragen, neue Typen zu generieren und Funktionen anhand von Typparametern auszuwählen. Bei generischen Programmen ist oft das Laufzeitverhalten schlechter (*abstraction penalty*), da man wichtige Spezialfälle nicht abdeckt. Ein Beispiel hierfür ist:

```
template <int dim>
double dot_product (double* a, double* b)
{
  double res;
  for (int i = 0; i < dim; ++i)
    res += a[i] * b[i];
  return res;
}
```

In diesem Code ist immer eine Schleife enthalten, obwohl die Dimension der Vektoren klein sein kann. Eine Bibliothek muss besonderen Wert auf die effiziente Implementierung solcher Funktionen legen, da sonst einige Nutzer zu handgemachten Funktionen greifen werden. Generische oder aktive Bibliotheken (*active libraries*) können mittels Template Metaprogramming den Nutzercode analysieren, und Optimierungen ausführen.

Besonders in laufzeitkritischen Applikationen wie *scientific computing* oder *embedded systems* kann die *abstraction penalty* nicht hingenommen werden, doch gerade in diesen sind sinnvolle Abstraktionen nötig, um mit der wachsenden Komplexität der Programme zurechtzukommen. Durch Abstraktionen wird Code lesbarer:

```
float a[3], b[3];
...
float res = meta_dot_product<2>(a,b);
```

Auf den ersten Blick ist klar, dass hier ein Skalarprodukt berechnet wird. Diese Ausdruckskraft ist mit Template Metaprogramming ohne Laufzeitkosten möglich.

Im Bereich *embedded systems* sind Abstraktionen Mangelware. Plattformspezifischer Code kann durch Registerabstraktionen vermieden werden, wie in [PDTR18015] beschrieben:

```
register_access <static_reg8 <ACME>, io_bus> reg1;
register_access <static_reg8 <URSEL>, io_bus> reg2;
reg1 ^= reg2;
```

Dieser Code ist äquivalent zu folgendem Assembleroutput (Plattform: Atmel Avr):

```
in r24, 0x34
in r25, 0x16
xor r24, r25
out 0x34, r25
```

Zwei Read Operationen auf dem I/O Bus, ein Xor und ein Write – also keine Laufzeiteinschränkung.

Man kann Template Metaprogramming auch nutzen, um Typen/Code zur Kompilierzeit zu erstellen:

```
template <typename Tuple> Tuple get_(Window const& w);

typedef tuple<int,int,int,int> geometry;
typedef tuple<string,string> class_inst;

geometry g = get_<geometry> (win1); class_inst
ci = get_<class_inst> (win1);

std::cout << field<0>(g) << field<1>(g)
          << field<2>(g) << field<3>(g);

std::cout << field<0>(ci) << field<1>(ci);
```

Hier sieht man eine Funktions-Template `get`, das ein Tuple – eine Zusammenfassung von Objekten unterschiedlicher Art, zurückgibt. Aufgrund unterschiedlicher Tuple-Typen gibt die Funktion unterschiedliche Dinge zurück. Der Zugriff auf Tuples geschieht mittels der Funktion `field`, und zwar indexbasiert.

17.2 Template Metalanguage

Todd Veldhuizen zeigt in seinem Aufsatz „C++ Templates are Turing Complete“ [Veldhuizen] eine Skizze des Beweises, dass Templates die Möglichkeit eröffnen, vom Compiler jede partiell rekursive Funktion berechnen lassen zu können. Da dadurch auch das Halteproblem in den Kompilierprozess hineinkommt, wurde im C++-Standard ein Limit von rekursiven Template-Instanziierungen eingeführt (17, in gcc änderbar).

Die Template Metalanguage ist eine (pur) funktionale Sprache, es gibt keine Iterationskonstrukte, sondern rekursive Template-Instanziierungen. Auch sind keine Seiteneffekte und Zuweisungen möglich, alle Definitionen sind konstant und können nicht geändert werden (insbesondere können keine Fehlerausgaben selbst erzeugt werden). Die Sprache

wird vom Compiler evaluiert, und operiert mit Typen. Einige C++ Ausdrücke, die in dieser Sprache gültig sind, da sie zur Laufzeit ausgewertet werden, sind:

- `enum`: ein Typ mit meheren benannten Integer-Konstanten

```
enum enum_type{
  name1 <=constant>,
  name2 <=constant>,
  ...
};
```

- `sizeof(type)`: gibt die Größe in Bytes der Objektrepäsentation von `type` zurück, `type` kann fast jeder Ausdruck sein, insbesondere ist `sizeof` angewendet auf einen Funktionsaufruf die Größe des Rückgabewerts.

- `T funcdecl(...)`: die `...` (*ellipsis*) in einer Funktionsdeklaration drücken aus, dass diese Funktion mit einer beliebigen Anzahl an Argumenten beliebigen Typs aufgerufen werden kann.

- `typedef`: die Zuweisung der Template Metalanguage, sie gibt Typen neue Namen.

```
typedef type typename;
```

17.3 type-traits

Durch *type traits* erfahren wir Dinge über Typen, die nur dem Compiler zugänglich sind. Durch Template Metaprogramming können wir diese Dinge dem Compiler entlocken. Wir wollen zum Beispiel wissen, ob ein Typ in einen anderen konvertierbar ist. Der folgende Code hilft uns dabei:

```
template<typename T, typename U>
class conversion {
  // sizeof (yes) == 1
  typedef char yes;
  // sizeof (no) > 1
  struct no { char dummy[2]; };

  // gets called if T is convertible to U
  static yes test(U);
  // gets called if T is NOT convertible to U
  static no test(...);
  // maybe we cannot construct T, but we can
  // return it
  static T makeT();
```

```
    public:
    enum {
      exists=sizeof(test(makeT()))==sizeof(yes),
      same_type=false
    };
  };

  template <typename T>
  class conversion<T,T> {
    public: enum { exists=true, same_type=true };
  };
```

Hier sehen wir alle zuvor besprochenen Konstrukte in Aktion. Zwei Typnamen `yes` und `no` werden deklariert, die durch ihre Größe unterscheidbar sind. Die Funktion `test` wird zweimal überladen, einmal mit dem Typ `U` als Argument und `yes` als Rückgabewert, und einmal mit irgendeinem anderen Typen als Argument und `no` als Rückgabewert. Die Funktion `makeT()` existiert, um ein `T` zu bekommen, selbst wenn `T`s Konstruktor privat ist. der Rückgabetyp von `test(makeT())` ist entweder `yes`, und zwar dann wenn `T` zu `U` konvertierbar ist, oder no wenn das nicht der Fall ist.

Um herauszufinden ob `T` und `U` gleich sind, erzeugen wir eine Spezialisation des Templates, das instanziiert wird, wenn die Typen gleich sind. Es fehlt in diesem Beispiel der Typ `void`, für den weitere Spezialisierungen notwendig sind.

Ein Test, ob ein Typ `U` von einem Anderen Typ `T` abgeleitet ist, basiert darauf, dass – wenn dies zutrifft, ein Zeiger auf `U` in einen Zeiger auf `T` konvertierbar ist. Der entsprechende Code sieht so aus:

```
template<typename T, typename U>
class supersubclass {
  public:
  enum {
    value=conversion<U const*,T const*>::exists &&
          !conversion<T const*,void const*>::same_type
  };
};
```

Wir müssen prüfen, ob o. g. Bedingung zutrifft, und ob der `T` nicht `void` ist, denn jeder Zeiger ist in einen void Zeiger konvertierbar.

Die Technik der Template-Spezialisierung funktioniert auch bei diesem Beispiel, einem Test, ob ein Typ ein Zeiger ist:

```
template<typename T>
class is_pointer {
  enum { value = false };
```

```
};
template<typename T>
class is_pointer<T*> {
  enum { value = true };
  typedef T pointee_type;
};
```

17.4 meta-if/select

Eine Auswahl von alternativen Typen, basierend auf einem boolschen Ausdruck, ist oft in Template-Metaprogrammen nötig. Der folgende Code ermöglicht das:

```
template <bool, typename True, typename False>
struct meta_if;

template <typename True, typename False>
struct meta_if <true, True, False> {
  typedef True type;
};

template <typename True, typename False>
struct meta_if <false, True, False> {
  typedef False type;
};
```

Es basiert auf Template Spezialisierung: Die `struct meta_if` definiert einen Typ mit dem Namen `type`, je nach Wert des boolschen Ausdrucks. Zum Beispiel könnte man das so nutzen:

```
typedef meta_if
<
  is_pointer<A>::value,
  A,
  A*
>
::type pointer;
```

Das `meta_if` ist eine `struct` mit eingebettetem Typ mit Namen `type`; dieses Konstrukt nennt man auch Meta-Funktion. Es existiert, damit man Typdeklarationen verzögert ausführen kann. Der Typ, der hinter `type` steckt, wird erst vom Compiler evaluiert, wenn `type` irgendwo anders zugegriffen wird. Eine Deklaration einer Meta-Funktion sieht so aus:

```
template <typename T1, ... , typename Tn>
struct metafunction {
```

```
  typedef ... type;
};
```

Ein Aufruf einer Meta-Funktion sieht so aus:

```
typedef metafunction<t1, ... , tn>::type result;
```

Ein weiteres Beispiel einer Meta-Funktion ist diese:

```
template <typename T>
struct strip_pointer {
  typedef typename is_pointer<T>::pointee_type type;
};
```

Sie liefert den Datentyp eines Zeigers.

Ein Problem mit der Anwendung von `meta_if` ist, das beide Zweige evaluiert werden. Nehmen wir an, wir wollen den Datentyp eines Zeigers wissen, der entweder ein normaler Zeiger, oder aber ein `std::auto_ptr` ist. Der nachfolgende Code ist dafür ungeeignet:

```
template <typename T>
struct autostrip_pointer {
  typedef typename meta_if
  <
    is_pointer<T>::value,
    strip_pointer<T>::type,
    T::element_type
  >
  ::type type;
};
typedef strip_pointer<int*>::type int_type;
```

Wenn T ein normaler Zeiger ist, hat er keinen eingebetteten Typ mit dem Namen `element_type`, und der Compiler beschwert sich. Wir können das lösen, wenn wir den Zugriff auf `element_type` in eine Meta-Funktion verpacken, und nur eine von beiden Zweigen evaluieren:

```
template <bool b,class Truefunc,class Falsefunc>
class apply_if {
  typedef typename meta_if
  <
    b,
    Truefunc,
    Falsefunc
  >
  ::type func;
public:
```

```
  typedef typename func::type type;
};

template <typename T> struct element_type {
  typedef T::element_type type;
};

template <typename T>
struct autostrip_pointer {
  typedef typename apply_if
  <
    is_pointer<T>::value,
    strip_pointer<T>,
    element_type<T>
  >
  ::type type;
};
```

17.5 Typ-Listen

Um nicht nur einzelne Typen verarbeiten zu können, sondern auch Kollektionen von Typen zusammenzufassen, brauchen wir eine Datenstruktur, die Typen zusammenfasst. Diese muss rekursiv abarbeitbar sein. Dies führt uns zur Typ-Liste ([Alexandescu'01]):

```
template <typename Head, typename Tail>
struct cons {
    typedef Head car;
    typedef Tail cdr;
};

struct nil {
    typedef nil car;
    typedef nil cdr;
};
```

Diese Struktur ist an die Definition einer Liste in Lisp angelehnt. Die rekursive Datendefinition lautet:

Eine Typ-Liste ist:

- `nil` oder
- `cons <first, rest>` wobei `first` ein Typ ist, und `rest` eine Typ-Liste.

Auf diesen Typ-Listen kann man nun vielfältige Algorithmen implementieren, von denen wir einige betrachten werden. Um die Länge einer Typ-Liste zu berechnen, brauchen wir einen Typ, der Zahlen beinhaltet:

```
template <int i>
struct int2type {
  enum { value = i };
  typedef int2type<value> type;
};
```

Wir definieren nun die Addition auf diesem Typ:

```
template <typename I1, typename I2>
struct plus {
  enum { value = I1::value + I2::value };
  typedef int2type<value> type;
};
```

Die Länge einer Typ-Liste berechnet sich nun so:

```
template <typename Typelist>
struct length {
  typedef typename meta_if
  <                                    // if
    is_same<Typelist, nil>::value,     // typelist empty
    int2type<0>,                       // length = 0
    plus
    <
      int2type<1>,                     // else add 1 to
      len<typename Typelist::cdr>      // length of rest
    >
  >
  ::type type;

  enum { value = type::value };
};
```

Um auf das i-te Element in der Liste zuzugreifen, müssen wir sie rekursiv durchlaufen:

```
template <typename Typelist, unsigned i>
struct at {
  typedef typename apply_if
  <                                      // if
    i == 0,                              // i == 0
    car<Typelist>,                       // (car list)
    at<typename Typelist::cdr, i-1>      // else (at(cdr list)i-1)
  >
  ::type type;
};
```

Um das Konzept von Funktionen höherer Ordnung, wie es in Lisp existiert, auf unsere Meta-Language zu übertragen, brauchen wir einen neuen Begriff – die Meta-Funktionsklasse. Das ist eine Meta-Funktion, eingebettet in eine `struct`:

```
struct metafunc_class {
    template <typename T1, ... , typename Tn>
    struct apply {
        typedef ... type;
    };
};
```

Die Meta-Funktion hat den Namen `apply`, damit der Funktionsaufruf vereinfacht wird:

```
typedef metafunc_class::apply<t1,...,tn>::type result;
```

Nun können wir noch allgemeinere Algorithmen auf Typ-Listen implementieren, wie zum Beispiel `fold`, welches eine binäre Meta-Funktionsklasse rekursiv auf sein früheres Ergebnis, und jedes Element einer Typ-Liste anwendet:

```
template <class Typelist, class Start, class Func>
struct fold {
  template <class TL, class S, class F>
  struct helper {
    typedef typename F::apply
    <
      S,
      typename TL::car
    >
    ::type result;
    typedef typename helper
    <
      typename TL::cdr,
      result,
      F
    >
    ::type type;
  };
  template <class R, class F>
  struct helper<nil, R, F> {
    typedef R type;
  };
  typedef typename helper
  <
    Typelist,
    Start,
    Func
  >
  ::type type;
};
```

Um den Algorithmus vom Ende der Typliste zu beginnen, und zum Anfang hin fortzusetzen, müssen wir die Aufrufe von `helper` und `F::apply` anders verschachteln:

```
// fold_back
...
struct helper {
  typedef typename F::apply
  <
    typename TL::car,
    typename helper
    <
      typename TL::cdr,
      Start,
      F
    >
    ::type
  >
  ::type type;
};
...
```

17.6 Code-Generierung

Eine Anwendung von Template-Metaprogramming ist Generierung von Typen/Code. Ein Weg, dies zu erreichen, ist die Erzeugung von Vererbungshierarchien. Um Tupel, wie im Motivationsabschnitt 17.1 auf Seite 159 gezeigt, zu erzeugen, können wir eine Meta-Funktionsklasse definieren, die eine Vererbungshierarchie erzeugt, und `fold` so anwenden, dass die Blätter Objekte der Typen in einer Typ-Liste enthalten:

```
template <template <class> class Unit>
struct generate_hierarchy {
  template <typename T1, typename T2>
  struct apply {
    template <typename U1, typename U2>
    struct tag : U1 { typedef U1 left_base; };

    struct type : T2, tag <Unit<T1>,T2> {
      typedef T2 left_base;
      typedef tag <Unit<T1>,T2> right_base;
    };
  };
};
typedef cons<char,cons<char,cons<int,nil> > > types;

struct empty_type {};

template <class T>
```

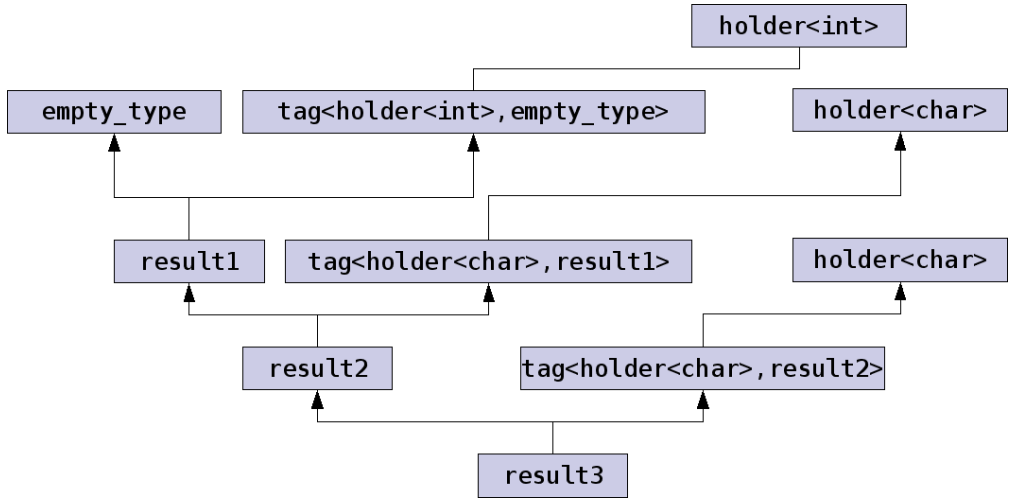


Abbildung 15: Typhierarchie erzeugt von `generate_hierarchy`

```
struct holder { T value; };

typedef fold_back
<
  types,
  empty_type,
  generate_hierarchy<holder>
>
::type tuple_char_char_int;
```

Die dadurch erzeugte Hierarchie sieht man in Abbildung 15. Die `tags` sind nötig, um zwischen den Basisklassen zu unterscheiden, wenn die Namen der Objekte und ihr Typ gleich sind (vgl. ISO/IEC 14882 10.1.4, 10.2.7).

Um nun indexbasiert auf die in dieser Hierarchie enthaltenen Typen zuzugreifen, müssen wir uns in der Hierarchie von abgeleiteter Klasse zur richtigen Basisklasse bewegen. Dazu haben wir in jeder Klasse die `left_base` und `right_base` eingebettet. Nun können wir die richtige auswählen:

```
template <class H, unsigned i>
struct field_type {
  typedef typename apply_if
  <
    is_same <int2type<i>,int2type <0> >::value,
    type2type<typename H::right_base>,
    field_type<typename H::left_base, i-1>
  >
  ::type type;
};
```

```
template <unsigned i>
struct field {
  template <class H>
  static typename field_type<H,i>::type&
  apply (H& obj)
  {
    typename H::left_base& subobj = obj;
    return field<i - 1>::apply(subobj);
  }
};
template <>
struct field<0> {
  template <class H>
  static typename H::right_base&
  apply (H& obj)
  {
    typename H::right_base& subobj = obj;
    return subobj;
  }
};
```

17.7 boost

Die hier besprochenen Konzepte sind in der Template Metaprogramming Library, die in der *boost library* ([boost]) enthalten ist, umgesetzt. Sie hat die gleichen Konzepte wie die STL:

- Container von Typen (`mpl::vector`, `mpl::list`, etc.)
- Iteratoren auf Container
- Algorithmen, die über Iteratoren auf Container zugreifen (`mpl::fold`, `mpl::find`)

Die `boost::type_traits` umfassen das Abfragen von Typinformationen, Relationen zwischen Typen, und Transformationen von Typen. Einige interessante Traits sind:

- `boost::is_pointer<T>` – ist `T` ein Zeiger ?
- `boost::alignment_of<T>` – das Aligment eines Objekts von Typ `T`
- `boost::is_polymorphic<T>` – ist `T` ein polymorpher Typ ?
- `boost::is_convertible<T,U>`

- `boost::add_const<T>` – macht aus `T` ein `T const`
- `boost::function_traits<F>::arity` die Zahl der Argumente von `F`
- `boost::function_traits<F>::result_type` der Rückgabetyp von `F`
- ...

17.8 Zusammenfassung

Wir haben gesehen, wie die Template-Metalanguage funktioniert, und wie wir über type-traits Informationen vom Compiler bekommen können. Es wurden Typ-Listen vorgestellt, um Typen zusammenzufassen, und Algorithmen auf diesen Typen auszuführen. Ein Beispiel zur Code-Generierung durch Vererbungshierarchien wurde gezeigt.

Das Template Metaprogramming ist ein weites Feld, das dazu beitragen kann, Abstraktionen auch in Gebieten zu nutzen, die laufzeitkritisch sind; und das kombinatorische Auftreten von Code, das mit mehreren Typen verbunden ist, zu lösen. Wir können gespannt sein, was dieses Gebiet noch für Überraschungen mit sich bringt.

18 Exceptions

18.1 Arten von Fehlern

Bei der Programmierung unterscheidet man zwei größere Gruppen von Fehlern – wahrscheinliche und unwahrscheinliche Fehler.

Unter wahrscheinlichen Fehlern versteht man hierbei Fehler, die erwart- und vorhersagbar sind. Typische Beispiele hierfür sind jegliche Eingaben, die ein Nutzer tätigen muss - man muss in diesen Fällen einfach davon ausgehen, dass die Eingaben falsch sind. Diese Art von Fehler wird im Allgemeinen „Error“ genannt. Anders ist es jedoch bei unwahrscheinlichen Fehler, diese beschreiben Ausnahmen, die in der Regel „Exceptions“ genannt werden. Es handelt sich hierbei um Fehler wie z. B. Speicherüberlauf und Netzwerkkommunikationsprobleme.

18.2 Fehlerbehandlung von Errors

Die Fehlerbehandlung von Errors erfolgt meist mit klassischen Werkzeugen aus C und C++. Die wohlbekanntesten Methoden dabei sind:

- Setzen und Abfragen von globale Variablen („Flags“). Hierbei werden globale Variablen definiert und diese an wohlbestimmen Stellen oder Zeiten ausgelesen und entsprechend der aktuellen Werte wird der Programmablauf beeinflusst.

- Auswerten von `return`-Werten. Hierbei wird der Rückgabewert einer Funktion oder Methode genutzt, um mehr als nur Quantitativeinformationen zuübermitteln, sondern auch in einem gewissen Rahmen eine Qualität. Beispielsweise sei eine Funktion gegeben, die den Betrag einer Zahl zurückgibt, dies bedeutet, dass der Rückgabewert für eine Zahl immer größer-gleich 0 ist. Nun könnte man -1 durch die Funktion zurückgeben lassen, wenn ein Buchstabe oder ein Wort als Wert in die Funktion eingegangen ist und somit kein Betrag berechnet werden konnte.

- Benutzung von `if-else`-Statements. Dies ist bestimmt die bekannteste Möglichkeit um Errors abzufangen und zu bearbeiten. Dabei werden bestimmte Situtionen genau definiert (die `if`-Zweige) und die daraus folgenden Handlungsweisen. Da man nicht alle Situationen vorher genau spezifizieren kann, gibt es in den meisten Fällen noch einen allgemeinen Fall, wenn kein `if`-Zweig wahr geworden ist, der `else`-Zweig.

Folgend ist ein Programmcode dargestellt, der ein `if-else`-Statement mit einem entsprechenden Rückgabewert kombiniert. Es geht hierbei nur um die Überprüfung, ob die Variable `a` einen Buchstaben zwischen a und z enthält. Wenn dies der Fall ist, dann wird eine `0` durch das `return`-Statement zurückgegeben, in allen anderen Fällen eine `-1`. Man könnte dann in einem anderem Teil des Programmes diese Rückgabewerte auswerten und schlussfolgern.

```
if (a >= 'a' && a <= 'z') {
  //true
  return 0;
}
else {
  //false
  return -1;
}
```

18.3 Fehlerbehandlung von Exceptions

Auch für die Fehlerbehandlung von Exceptions gibt es theoretische mehrere verschiedene Möglichkeiten, die jedoch zum größten Teil keinen Sinn machen, da sie nicht praktikabel sind bzw. ihre Nachteile klar überwiegen.

- Benutzung von Lösungsmöglichkeiten aus „Fehlerbehandlung von Errors“. Durch das Benutzen der eben dargestellten Lösungsmöglichkeiten aus „Fehlerbehandlung von Errors“ verliert der entstehende Programmcode sehr stark an Übersichtlichkeit und ist für den Entwickler nur noch schwer zu lesen und zu verstehen.

- Ignorieren von Exceptions. Dies ist sicherlich die einfachste, aber auch die schlimmste Lösung. Auch wenn es sich um unwahrscheinliche Fehler handelt, heißt das noch lange nicht, dass sie nie auftreten werden – im Gegenteil, in den Situationen, wo man sie nicht gebrauchen kann, treten sie auf („If anything can go wrong, it will“ – Murphy’s laws).

- Sprünge mit Hilfe von `goto`-Anweisungen. Sprunganweisung mit Hilfe von `goto` sind theoretisch möglich und werden vom C/C++-Standard unterstützt. Es wird jedoch von der Benutzung in unserem Zusammenhang abgeraten, da dies zu sogenannten „Spagetti“-Code führt, der weder lesbar noch verständlich ist.

Eine Fehlerbehandlung von Exceptions erfolgt daher mit Hilfe von „`try-catch`“-Statements, die Exceptions als C++-Typen behandeln. `try-catch`-Answeisungen haben den Vorteil der räumlichen Trennung von dem allgemeinen Codeblock und dem Ausnahmeblock, was sehr übersichlich und leicht lesbar ist.

18.3.1 try-catch-Anweisungen

Nachfolgend ein simples „Codegerüst“ für eine try-catch-Anweisung. Es gibt einen try-Block, indem die allgemeinen Programmanweisungen enthalten sind (die selben Anweisungen, welche auch ohne Exceptionhandling ausgeführt werden würden). Nach dem try-Block folgt der catch-Block, welcher zum „Fangen“ der Fehler dient. Die Signatur von catch bestimmt den Codeblock entsprechend dem Fehlertypen – Exception als C++-Typen. Ein „Weiterwerfen“ mit der throw-Anweisung ist dabei auch möglich.

```
void foo() throw()
{ //optional: "function-try-block"
  try{
    //allg. Codeanweisungen
  }
  catch(std::exception &ex){
    //Codeanweisung fuer std::exception
    std::cerr << ex.what() << std::endl;
  }
  catch(...){
    //Codeanweisung fuer allg. Ausnahmen
    throw;  //throwing again
  }
}
```

Da auch eine Verschachtelung von try-catch-Anweisungen möglich ist, müssen die entsprechenden Fehlerhierarchien beachtet werden.

Die catch-Anweisung kann, entsprechend ihrer Signatur, einen bestimmten Exceptiondatentypen „fangen“, so z. B. fängt `catch(std::string&)` alle Fehler, die eine Exception vom Typ `std::string` werfen, während hingegen `catch(std::bad_alloc&)` alle Exception fängt, die vom Exceptiontyp `std::bad_alloc` sind, u. s. w. Zusätzlich kann man neben dem Exceptiontyp auch das entsprechende Objekt mit übergeben – wichtig ist hierbei jedoch, dass das Objekt per Referenz übergeben wird, da der Copy-Constructor scheitern könnte!

Jeder Exceptiontyp, der von `std::exception` abgeleitet worden ist, besitzt eine Methode `what()`, mit welcher man sich nähere Informationen über das Exceptionobjekt ausgeben lassen kann (z. B. mit Hilfe von `std::cerr`).

```
void foo()
{ //optional: "function-try-block"
  try{
    //allg. Codeanweisungen
  }
  catch(std::string &ex){
    //Codeanweisung fuer std::string
```

```
    std::cerr << ex << std::endl;
  }
  catch(std::exception &ex){
    //Codeanweisung fuer std::exception
    std::cerr << ex.what() << std::endl;
  }
}
```

Man sollte für die Fehlerausgabe von einem Exceptionobjekt mit der Methode `what()` anstelle von `std::cout` besser `std::cerr` verwenden, da man dann die beiden Ausgaben gegebenenfalls von einander einfach trennen kann.

18.3.2 Ableitungshierarchien

`std::exception` ist die Basisklasse für das Exceptionhandlung in C++. Diese Basisklasse hat, wie schon erwähnt, nur eine Methode names `what()`. Die Struktur dieser Standardbibliothek sieht dabei wie folgt aus:

```
namespace std {
  class exception       //has no message-C'tor
    class logic_error;
      class domain_error;
      class invalid_argument;
      class length_error;
      class out_of_range;
    class runtime_error;
      class range_error;
      class overflow_error;
      class underflow_error;
}
```

Beim eigenen Erstellen von Exceptionklassen ist das Gruppieren und das richtige Ableiten der Klassen voneinader wichtig. Es folgt ein Beispiel aus dem Bereich der Mathematik. Man hat eine Klasse `Matherr`, von der die restlichen mathematischen Fehlerklassen (`Overflow`, `Underflow`, `Zerodivide`) abgeleitet sind, welche aber selber von `std::runtime_error` abgeleitet worden ist. Dies hat den Vorteil, dass wir die definierten Fehlerklassen fangen können, so z. B. mit `catch (Overflow)`, aber auch alle restlichen, noch nicht definierten mathematischen Fehler mit `catch (Matherr)`.

```
class Matherr: public std::runtime_error {};
class Overflow: public Matherr {};
class Underflow: public Matherr {};
class Zerodivide: public Matherr {};

void f() {
```

```
  try {
    //Anweisung
  }
  catch (Overflow) {
    //...
  }
  catch (Matherr) {
    //...
  }
}
```

18.3.3 Probleme

Das Exceptionshandling in C++ hat leider auch einige Nachteile, da es erst später zum Standard hinzugekommen ist. Größtest Problem sind daher „alte“ C-Funktionen, die in den C++-Standard übernommen wurden. Als Beispiel kann man hier stellvertretend die Funktion `strtol()` nennen, die den Inhalt einer Variablen vom Datentyp `std::string` in `long` umwandelt, sofern dies möglich ist. Enthält der `std::string` keine Zahl, so gibt die Funktion den Wert 0 zurück – und da ist auch schon das Problem. Woher weiß man nun, ob es sich um einen Fehler handelt oder ob der `std::string` eine 0 enthielt?

Solche Probleme kann man leider nur mit „Workarounds“ lösen. In dem Beispiel `strtol()` muss man den Übergabewert vorher testen, bevor man in an die Funktion `strtol()` übergibt.

```
#include <iostream>
#include <string>
#include <stdexcept>

class number_format: public std::invalid_argument {
  public:
    number_format():
      std::invalid_argument("Typumwandlung misslungen"){}
    number_format(std::string &what):
      std:: invalid_argument(what){}
};

long string_to_long(std::string str) {
  if (!isdigit(str[0]))
    throw number_format();
  return std::strtol(str.c_str(), NULL, 10);
}

int main() {
  std::string s;
```

```
  try {
    std::cout << "Geben Sie eine Zahl ein: " << std::endl;
    std::cin >> s;
    std::cout << string_to_long(s) << std::endl;
  }
  catch (number_format &ex) {
    std::cerr << ex.what() << std::endl;
  }
}
```

18.4 Exception Safty

Exceptions in C++ haben eine Quelle-Senke-Analogie, dies bedeutet, man hat einen Ort des Entstehens von Exceptions (die Quelle) und man hat eine Position im Programmfluss, an der man den Fehler abfängt und Maßnahmen einleitet (die Senke).

Es gibt nun den Begriff der „Exception neutralen Implementierung“. Unter diesem Begriff versteht man, dass es keine Senke in dem Programm gibt, sondern dass die Exceptions an die nächst höhere Instanz „weitergeworfen“ werden mit Hilfe von `throw` – man nennt dies „rethrow“. Alle Operationen der STL sind „Exception neutrale Implementieren“.

Ein weiterer wichtiger Punkt in diesem Zusammenhang die „3 Level von Garantien“ oder auch „Abraham's Guarantees“ genannt. Sie beschreiben drei Stufen von Implementationsgarantien in Bezug auf Exceptions.

18.4.1 Basic Guarantee

Diese „Basisgarantie“ oder „Grundlegende Garantie“ sagt aus, wenn eine Ausnahme geworfen wird, dass dann keine Ressourcenlecks zurück bleiben dürfen. Die aktuellen Objekte müssen in einem „destructible“ und verwendbaren Zustand bleiben.

Diese Basisgarantie ist das schächste Niveau der Ausnahmesicherheiten. Alle Operationen der STL besitzen mindestens diese Basisgarantie.

18.4.2 Strong Guarantee

Die „Hohe Sicherheit“ stellt sicher, dass wenn eine Ausnahme auftritt, der aktuell Objektzustand unverändert bleibt. Diese Garantie impliziert die „comitt-or-rollback“-Semantik. Dies bedeutet, entweden kann die Operation fehlerferi ausgeführt werden oder der Orginialzustand wird wieder hergestellt.

Diese hohe Sicherheit wird z. B. bei dem Constructor durch den Compilier angewandt, entweder kann ein Objekt mit dem entsprechenden Constructor erzeugt werden oder alle durch den Constructorprozess getätigten Schritte werden rückgängig gemacht und die Ausgangssituation wird wieder hergestellt. Ein weiteres Beispiel sind Operationen der STL auf Containern. Wenn bei diesen Operationen Ausnahmen auftreten, dann werden die Iteratoren auf die Container niemals ungültig.

18.4.3 Nothrow Guarantee

Diese Garantie wird häufig auch „Absolute Garantie“ genannt. Sie besagt, dass eine Methode oder Funktion unter keinen Umständen eine Excetion wirft. Sie ist deshalb auch die Umsetzungsgrundlage für die „Basisgarantie“ und die „Hohe Sicherheit“.

Die Umsetzung dieser absoluten Garantie ist in vielen Fällen gar nicht möglich und wenn doch, dann meist nur mit sehr hohen und kostspieligen Aufwand. Das eine Funktion oder Methode dieser absoluten Garantie genügt, kann man an dem Funktions- bzw. Methodenkopf erkennen. Wenn ein `throw()` ohne Signatur angehangen ist, z. B. `void foo() throw()`, so genügt sie dieser Garantie.

Beispiele für Funktionen bzw. Methoden mit dieser absoluten Garantie sind `std::swap()` und der Zuweisungsoperator für eingebaute und „plain old data“-Typen (POD).

In dem nachfolgenden Beispiel wird eine Methode `void T::Swap(T& other) throw()` definiert, welche die „Nothrow Guarantee“ erfüllt. Mit Hilfe dieser Methode wird dann der Zuweisungsoperator `T& T::operator=(const T& other)` definiert, welcher (nur) die „Hohe Garantie“ erfüllt. (Deshalb siehen oben: „Sie [die „Absolute Garantie“] ist deshalb auch die Umsetzungsgrundlage für die „Basisgarantie“ und die „Hohe Sicherheit“.“)

```
void T::Swap( T& other ) throw() {
    // ...swap the guts of *this and other...
    // std::swap(var1, other.var1);
}

T& T::operator=( const T& other ) {
    T temp( other ); // do all the work off to the side
    Swap( temp );    // then "commit" the work using
```

```
    return *this;     // nonthrowing operations only
}
```

18.5 `throw`-Spezifikation

Es gibt drei wesentliche Arten, wie man Exceptions mit Hilfe von `throw` in einem Funktions- bzw. Methodenkopf angeben kann.

1. `int f()` – dies bedeutet, dass diese Funktion oder Methode jede Exception werfen kann, da der Exceptiontyp nicht weiter spezifiziert ist.

2. `int f() throw()` – dies ist der „nothrow“-Fall, was bedeutet, dass diese Funktion oder Methode unter keinen Umständen eine Exception wirft.

3. `int f() throw( A, B )` – in diesem Falle wurden die Exceptiontypen im Funktions- oder Methoden genauer spezifiziert. Nun werden (theoretisch) nur Exception vom Typen A und B geworfen.

Die ersten beiden Fälle sind klar und eindeutig, was passiert jedoch im dritten Falle bei folgender Programmkonstruktion?

```
int f() throw (A,B) {
  throw C(); // ruft unexpected-handler auf
}
```

Die Funktion bzw. Methode kann laut „Kopf“ nur Exceptions vom Typen A oder B werfen, in dem Listing wird jedoch `throw C()` aufgerufen, was gegen die Exceptionspezifikation verstößt. In diesem Falle wird der `unexpected`-Handler aufgerufen, der seinerseits `terminate()` und danach `abort()` aufruft. Man kann sich für eigene Bedürfnisse einen eigenen `unexpected`-Handler zu Beginn seiner Applikation definieren. Dieser sollte jedoch in seinem Verlauf auch `terminate()` und danach `abort()` aufrufen. Diese Definition funktioniert wie folgt dargestellt:

```
void MyUnexpectedHandler() { /*...*/ }
std::set_unexpected(&MyUnexpectedHandler);
std::set_terminate();
```

18.6 Richtlinien

Wann ist es sinnvoll „garantierten“ Code zu schreiben?

Die Antwort darauf lautet klar **immer**! Die Begründung liefert Herb Sutter in „Guru of the week“ Ausgabe 82 (siehe [Sutter]) mit den zwei wesentlichen Aussagen, die es auf den Punkt bringen:

1. „Exceptions happen.“ – Exception passieren einfach immer, man kann sie nicht einfach ignorieren.

2. „Writing exception-safe code is good for you.“ – Die Tatsache schon, dass man Exception-sicheren Programmcode geschrieben hat, ist gut für das Programm und den eigenen Programmierstil. Man denkt dadurch einfach bewusster an Probleme, die auftreten können – nicht müssen.

Regeln

Die „Regeln“ für guten und sicheren Code gibt Herb Sutter in der selben Ausgabe.

1. „resource acquisition is initialization“ – wer Ressourcen benötigt, muss diese bei nicht gebrauchen auch wieder zur Verfügung stellen.

2. „do all the work off to the side, then commit using nonthrowing operations only“ – entspricht der „commit-or-rollback“-Semantik.

3. „one class (or function), one responsibility“ – jede Funktion oder Methode soll genau eine Aufgabe habe und auch nur diese erfüllen.

Eine Funktion sollte immer die stärkste Garantie einhalten ohne dabei Aufrufer, die diese nicht benötigen, zu bestrafen. Die Reihenfolge dabei ist „Absolute Garantie“, „Hohe Garantie“ und als schächstes die „Grundlegende Garantie“.

Literatur

[Veldhuizen] Veldhuizen, Todd: *C++ Templates are Turing Complete*, http://www.oonumerics.org

[Veldhuizen'95] Veldhuizen, Todd: *Expression Templates*, C++ Report, June 1995, http://osl.iu.edu/~tveldhui/papers/Expression-Templates/exprtmpl.html

[Alexandescu'01] Alexandrescu, Andrei: *Modern C++ Design*, Addison-Wesley, 2001, http://www.informit.com/articles/article.asp?p=31529

[PDTR18015] ISO/IEC PDTR 18015: *Technical Report on C++ Performance*, http://open-std.org/JTC1/SC22/WG21/PDTR18015.pdf

[boost] boost.org: *Boost Libraries*, http://www.boost.org

[Wilson'95] Wilson; Johnstone; Mark; Neely; Boles: *Dynamic Storage Allocation - A Survey and Critical Review*, Intl. Workshop on Memory Management, 1995

[Berger'02] Berger; Zorn; McKinley: *Reconsidering Memory Allocation*, OOPSLA'02, 2002

[Austern'00] Austern, Matt: *What are Allocators good for?*, C/C++ Users-Journal, December 2000

[Langer'98] Kreft; Langer: *Allocator Types*, C++ Report, June 1998

[Langer'00] Langer, Angelika; Kreft, Klaus: *Standard C++ IOStreams and Locales*, Addison-Wesley, 2000

[Langer] Langer, A.; Kreft, K.: *C++ Expression Templates*, Draft-Version, http://www.langer.camelot.de/Articles/Cuj/ExpressionTemplates/ExpressionTemplates.htm

[Langer'03] Langer, A.; Kreft, K.: *C++ Expression Templates*, C/C++ User Journal, March 2003, http://www.cuj.com/documents/s=8232/cuj0303langer/

[Myers'93] Myers, Nathan: *Memory Management in C++*, C++ Report, July/August 1993

[Myers'00] Myers, Scott: *C++ User Journal, misc. articels*,
http://www.aristeia.com/publications_frames.html

[Gamma'94] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John: *Design Patterns: elements of reusable object-orientated software*, Addison-Wesley, 2002

[Stroustrup'00] Stroustrup, Bjarne: *The C++ Programming Language*, Special Edition, Addison Wesley, 2000

[Koenig'03] Koenig, Andrew; Moo, Barbara: *Ruminations on C++*, Addison-Wesley, 2003

[Kuehl'00] Kuehl, Dietmar: *Writing to a Specialized IOstream*,
http://www.informatik.uni-konstanz.de/~kuehl/c++/iostream/

[Kanze'00] Kanze, James: *Filtering Streambufs – Variations on a Theme by Schwarz*,
http://www.gabi-soft.fr/articles/fltrsbf1.html

[Informit] *All About bool*,
http://www.informit.com/guides/content.asp?g=cplusplus\&seqNum=171

[CPPR] *C++ Reference*,
http://www.cppreference.com

[NGRC++] *vector<bool> and addressability*, 1997/01/27,
newsgroup comp.std.c++

[Sutter] Sutter, H.: *Guru of the week*,
http://www.gotw.ca/gotw/

[GotW] Sutter, Herb: *When Is a Container Not a Container?*, C++ Report, 11(5), May 1999,
http://www.gotw.ca/publications/mill09.htm

[GotW #7] Sutter, Herb: *Guru of the week #7: Header Dependencies*,
http://www.gotw.ca/gotw/007.htm

[GotW #29] Sutter, Herb: *Guru of the week #29: Strings*,
http://www.gotw.ca/gotw/029.htm

[GotW #32] Sutter, Herb: *Guru of the week #32: Preprocessor Macros*,
http://www.gotw.ca/gotw/032.htm

[GotW #34] Sutter, Herb: *Guru of the week #34: Forward-Declarations*, http://www.gotw.ca/gotw/034.htm

[GotW #49] Sutter, Herb: *Guru of the week #49: Template Specialization and Overloading*, http://www.gotw.ca/gotw/049.htm

[GotW #50] Sutter, Herb: *Guru of the week #50: Using Standard Containers*, http://www.gotw.ca/gotw/050.htm

[STL'00] *Standard Template Library Programmer's Guide*, Release 3.3, Silicon Graphics Inc. – sgi, 2000, http://www.sgi.com/tech/stl/

[C++ 97] ISO/ANSI C++ Standard 1997,

http://std.dkuug.dk/jtc1/sc22/open/n2356/

[Vandevoorde] Vandevoorde, D.; Josuttis, N. M.: *C++ Templates, The Complete Guide*, Addison-Wesley 2003

[Hyperformix] *Example usage of template specialization*, http://cpptips.hyperformix.com/cpptips/specialize

[Wikipedia] Wikipedia.org: *Compiler*, http://de.wikipedia.org/wiki/Compiler

[MSDN] Microsoft Developer Network: *Creating Precompiled Header Files (Visual C++ Concepts)*, http://msdn.microsoft.com/library/en-us/vccore/html/_core_creating_precompiled_header_files.asp

[Linuxmagazin'04] Linuxmagazin Mai'04: *ELF, das Executable and Linkable Format im Detail*, Linux New Media AG, 2004 http://www.linuxmagazin.de

[GCC] GNU Compiler Collection: *Using Precompiled Headers*, http://gcc.gnu.org/onlinedocs/gcc/Precompiled-Headers.html

[Breymann'03] Breymann, Ullrich: *C++ Einführung und professionelle Programmierung*, Hanser Fachbuchverlag (Deutschland) GmbH, 2003

[Nootz'01] Nootz, Petra; Morick, Franz: *C/C++ Referenz*, Franzis (Deutschland) GmbH, 2001

[Unruh] Unruh, E.: *Prime number computation*, http://www.erwin-unruh.de/Prim.html

[Sutter'99] Sutter, Herb: *Exceptional C++*, Addison-Wesley 2000

[Papurt'95] Papurt, David: *Inside the Object Model*, SIGS Books 1995

[Meyers'97] Meyers, Scott: *Mehr effektiv C++ programmieren*, Addison-Wesley 1997

[Lippman'96] Lippman, Stanley B.: *Inside the C++ Object Model*, Addison-Wesley 1996