C++ FOR
(Montagnachmittag)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

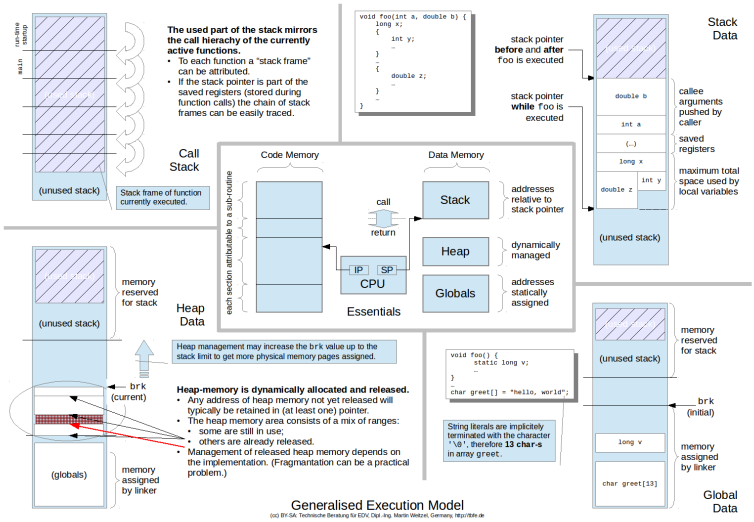
Die Besprechung der Musterlösung(en) erfolgt zu Beginn des folgenden Vormittags.

Generalisiertes Ausführungsmodell

- CPU und Speicher

- Stack-Daten
- Globalen Daten
- Dynamisch verwaltete Daten

- Stack-Frames (Verwaltungsinformation)



CPU und Speicher

Allgemein betrachtet sind dies Kernkomponenten jedes Computers.

- Vom kleinsten Controller in der Waschmaschine ...
- ... über klassische Mobiltelefone ...
- ... zu Smart-Phones ...
- ... Home-PC-s ...
- ... File- und Applikations-Servern ...
- ... bis zum größten Main-Frame.

(OK, die größeren haben mehr als eine CPU ziemlich viel Speicher ... *)

* ... wenn auch vielleicht nicht immer ganz so viel wie die NSA :-)

CPU-Aufbau (1)

Program Counter und Stack Pointer

Der innere Aufbau der CPU ist für das Verständnis der hier behandelten Themen weniger wichtig - bis auf zwei Bestandteile:

- Program Counter - oder deutsch: Programm(schritt)zähler
  - oft abgekürzt zu PC, manchmal auch IP (für Instruction Pointer)
  - enthält die Adresse des als nächsten auszuführenden Progammschritts
  - wird beim Auslesen des nächsten Befehls automatisch weitergesetzt
- Stack Pointer - oder deutsch: Stapelzeiger (wenig gebräuchlich)
  - oft abgekürzt zu SP
  - gibt die *Grenzadresse* des Stack-Bereichs an

CPU-Aufbau (2)

Ausdehnung des Stack-Bereichs

Da der SP nur die *Grenzadresse* angibt, könnte man meinen, es sei lediglich eine Definitionssache, ob der Stack darüber beginnt oder darunter.

Tatsächlich hängt die Ausdehung des aber davon ab, wohin sich der Stack-Pointer bei Maschinenbefehlen wie

- `push` (Daten: Register => Speicher) bzw.
- `jsr` (Unterprogramm-Einsprung*)

bewegt - wenn zu *kleineren* Adressen hin, gehören ab dem SP alle *größeren* Adressen zum Stack.

---

* Der betreffende Maschinenbefehl muss nicht zwingend `jsr` heißen, dies ist lediglich ein üblicher Name und steht für "jump to subroutine. Abhängig von der Assemblersprache sind auch andere Namen üblich, z.B. "jump and link", wobei das letzte Wort ausdrücken soll, dass anders als bei einem bloßen Sprung eine Verbindung zur Absprungstelle angelegt wird, damit später dorthin zurückgekehrt werden kann.

CPU-Aufbau (Details)

Die folgenden Seiten stellen noch weitere Details zum Aufbau einer typischen CPU dar.

Sie sind für das Verständnis der Ausführungen in diesem Kapitel weniger wichtig und können auch übersprungen werden.

Im einzelnen geht es noch um:

- ALU
- Register
- Datenpfad
- Steuerungslogik

CPU-Aufbau (Details 2)

ALU (Arithmetic Logic Unit)

Hier werden Daten miteinander verknüpft.

Herkunft:

- oft ausschließlich aus Registern
- evtl. auch direkt aus dem Speicher

(Ob die ALU nur mit Registern oder auch direkt mit dem Speicher zusammenarbeiten kann, hängt von der CPU-Architektur ab.)

Ergebnis:

- meist in einem bestimmten Register oder
- in einem Satz ausgewählter Register

Das Register, welches zur Ergebnisablage mit der ALU verschaltet werden kann, wird oft auch Akkumulator genannt.

CPU-Aufbau (Details 3)

(Weitere) Register

Je nach CPU-Architektur gibt es eine mehr oder weniger große Zahl universell verwendbarer Register.

Darüber hinaus gibt es auch solche mit Sonder- oder Spezialfunktionen:

- Program Counter (PC)
- Stack Pointer (SP)
- Flags zur Maskierung von Interrupts
- Stack-Limits*

* Hierbei handelt es sich um harte Grenzen, innerhalb derer sich der SP bewegen darf. Sofern der Stack diesen Bereich verlässt, wird ein (Software-) Interrupt ausgelöst. Dies kann sehr nützlich sein, um zu verhindern, dass durch einen unerwartet gewachsenen Stack andere Datenbereiche überschrieben werden - insbesondere der Bereich dynamisch verwalteter und globaler Daten.

CPU-Aufbau (Details 3)

Datenpfad-Steuerung

Die verbleibenden Teile* einer typischen CPU sind im wesentlichen:

- (um-) schaltbare Datenpfade und
- die zugehörige Steuerungslogik

Letztere verbinden erstere so miteinander, wie es zur Ausführung des jeweils zu verarbeitenden Maschinenbefehls erforderlich ist.

* Vom Umfang her können diese durchaus einen wesentlichen oder sogar den größten Teil einer typischen CPU ausmachen.

Stack-Daten

Der Stackbereich im Datenspeicher ist generell wichtig für Unterprogramme.

Dort wird abgelegt:

- Aufrufparemeter
- Lokale Variablen
- Verwaltungsinformation

Bei vielen Hardware-Architekturen wächst der Stack von großen zu kleinen Adressen.

In der oft üblichen Darstellung des Speichers als Rechteck mit

- *kleineren* Adressen unten und
- *größeren* Adressen oben

hängt der "Stapel" also gewissermaßen an der Decke.

Globale Daten

Diese stehen an festen Adressen, die zuvor vom Linker vergeben wurden.

Für alle Objekt-Module, aus denen das endgültige Programm besteht, ermittelt der Linker den Bedarf an globalem Speicher.

Dieser ergibt sich aus:

- globale Variable (außerhalb aller Blöcke*)
- block-lokale `static` Variablen
- `static` Member Variablen

---

* Dabei spielt es keine Rolle, ob diese `static` sind oder nicht - in beiden Fällen handelt es sich um Speicherplatz im globalen Datenbereich. Der Unterschied ist lediglich, dass globalen `static` Variablen mit *gleichem Namen* aus *unterschiedlichen* Objektmodulen* jeweils individueller Speicherplatz zugeordnet wird.

Dynamisch verwaltete Daten

Hier werden die Daten abgelegt, für die zur Laufzeit

- explizit Speicher angefordert wird, der dann
- irgendwann auch wieder freigegeben werden sollte.

Andere übliche Bezeichnungen für diesen Bereich sind:

- Heap
- Free Store
- Dynamischer Speicher

Stack-Frames

Unter einem Stack-Frame versteht man denjenigen Abschnitt auf dem Stack, der einem (aktiven) Unterprogramm zuzuordnen ist.

Die Stackframes ergeben sich direkt aus den Maschinenbefehlen beim

- Aufruf und
- Rückehr aus

Unterprogrammen. Grundsätzlich sind dabei die folgenden Probleme zu lösen:

- Übergabe von Argumenten
- Speicherplatz für lokale Variable
- Entgegenahme eines Returnwerts
- Rückkehr an die Aufrufstelle

Detailablauf beim Unterprogramm-Aufruf

Hier ist zu unterscheiden zwischen

- dem Code, der noch vom Aufrufer des Unterprogramms (Caller) ausgeführt wird
- dem eigentlichen Unterprogrammsprung und
- dem Code, der im aufgerufenen Unterprogramm (Callee) ausgeführt wird.

Hier bestehen grundsätzlich viele Freiheiten für den Compiler solange sichergestellt ist, dass *Caller* und *Callee* zusammenpassen (Einhaltung der Calling Conventions).

Detailablauf beim Unterprogramm-Aufruf (2)

Argumente bereitstellen

Noch an der Aufrufstelle wird i.d.R. Code zum Ablauf kommen,

- die Werte aller Aufrufargumente ermittelt, was
  - im Fall von Variablen einen Speicherzugriff oder
  - im Fall von Ausdrücken eine Berechnung erfordern kann
- und diese dann zur Verwendung durch das aufgerufene Unterprgramm
  - auf den Stack legt bzw.
  - in bestimmten Registern hinterlässt.

In welcher Form Stack und Register hier genau benutzt werden, regeln die Calling Convetions.

Detailablauf beim Unterprogramm-Aufruf (3)

Maschinenbefehl zum Unterprogrammsprung

Im Rahmen des Unterprogramm-Einsprungs auf den Stack gelegt werden:

- Mindestens der Program-Counter
  - Dies ist erforderlich, da Unterprogramme von mehr als einer Stelle aus aufgerufen werden können.
  - insofern benötigen sie eine Information, wohin am Ende zurückzukehren ist.
  - Der Program-Counter wurde beim Holen des `jsr`-Befehls bereits weitergeschaltet.
  - Der auf dem Stack gesicherte Wert zeigt somit auf den Befehl direkt dahinter.

Schließlich wird die Startadresse des Unterprogramms in den Program-Counter übertragen.

Detailablauf beim Unterprogramm-Aufruf (4)

Aufrufkonventionen (Calling Conventions)

Hierbei handelt es sich um eine Reihe von Festlegungen, die letzten Endes sicherstellen sollen, dass die Kommunikation zwischen Aufrufer und aufgerufenem Unterprogramm funktioniert.

Unter anderem muss Einigkeit über die gesicherten Register bestehen und welche Register ggf. die Werte von Aufrufargumenten enthalten.

- Da das Sichern von Registern beim Unterprogrammsprung Zeit kostet, sollten es einerseits nicht unnötig viele sein.
- Auf der anderen Seite sind Register wertvoller Speicherplatz für temporäre Werte.
- Oft befindet sich unter den gesicherten Registern auch der Stack-Pointer,* obwohl das theoretisch nicht zwingend ist.

* Mit den auf dem Stack selbst gesicherten Stack-Pointern besteht eine Rückwärtsverkettung zur Main-Funktion (und von dort weiter ins Runtime-Startup-Modul), die sich zur Laufzeit dynamisch verfolgen lässt, und mit deren Hilfe beim "post-mortem"-Debugging anhand des Speicherabzugs die zum Zeit eines Programmabsturzes aktiven Funktionen rekonstruiert werden können.

Detailablauf beim Unterprogramm-Aufruf (5)

Platz für lokale Variable schaffen

Im aufgerufenen Unterprogramm wird

- zunächst Platz für die eigenen, lokalen Variablen geschaffen,
- wozu ein einfaches Verschieben des Stack-Pointers ausreicht.

Anschließend werden lokale Daten vom Unterprogramm mit einem Offset zum Stack-Pointer adressiert.

Die Berechnung der effektiven Adresse erfolgt durch einen speziell für diese Aufgabe in der CPU vorhandenen* Addierer.

- Dies gilt sowohl die die vom Aufrufer übergebenen Argumente
- wie auch für die im Unterprogramm lokal vorhandene Variable.

* Da diese Technik zur Realisierung lokaler Variablen eine lange Tradition hat, wird die Adressierung relativ Stack-Pointer ausnahmslos von allen modernen CPUs in effizienter Form unterstützt.

Detailablauf beim Unterprogramm-Aufruf (6)

Aufrufargumente und lokale Variable

Der Unterschied zwischen beiden besteht bei genauer Betrachtung nur in der Tatsache

- dass Aufrufargumente noch vom aufrufenden Code mit Intialwerten versehen werden,
- während lokale Variable, die nicht explizit initialisiert wurden, in ihren Anfangswerten unvorhersehbar* sind.

Ergänzend muss dazu angemerkt werden, dass innerhalb eines Blocks mit `static` definierte Variable von der Lebensdauer her *nicht* an die Ausführung eines Blocks oder einer Funktion gebunden sind und daher bei den globalen Variablen abgelegt werden.

* Genauer gesagt ergeben sich die Anfangswert von nicht explizit initialisierten Variablen aus der vorherigen Nutzung des Stackbereichs, in dem sie angelegt wurden.

Details bei der Unterprogramm-Rückkehr

Beim Verlassen des Unterprogramms werden die Aktionen rückgängig gemacht, die beim Aufruf stattfanden.

- Zunächst muss der (Rückgabewert)[#return_value] bereitgestellt werden (sofern das Unterprogramm einen solchen liefert).
- Anschließend wird der für die lokalen Variablen reservierte Stack freigegeben.
- Schließlich erfolgt der Rücksprung an die Aufrufstelle.
- Dort muss noch der für die Parameterübergabe benutzte Stack freigegeben werden.

Moderne CPU-Architekturen können diese Schritte teilweise zusammenfassen. Wichtige Voraussetzung dafür ist, dass der Aufrufer und das aufgerufene Unterprogramm* Anzahl und Typ der Argumente kennen.

* In C++ ist dies immer gegeben, da die Sichtbarkeit der Deklarationen (Prototyp) Voraussetzung für den Aufruf einer Funktion ist. Hinsichtlich C wurde ähnliches im C89-Standard damit erreicht, dass der Compiler bei nicht-sichtbarem Prototyp einen solchen putativ erstellen kann.

Details bei der Unterprogramm-Rückkehr (2)

Bereitstellung des Rückgabewertes

Die wesentlichen Details hängen hier davon ab, ob es sich um einen Grundtyp handelt* oder um eine (größere) Datenstruktur.

- Grundtypen werden in der Regel in einem bestimmten Register zurückgegeben.
- Für Strukturen muss der Aufrufer Stack-Speicherplatz bereitstellen.

Details zur Rückgabe in einem Register regeln ggf. die Calling Conventions.

Die Rückgabe über den Stack kann man als (versteckte) Übergabe einer Referenz sehen, welche die aufgerufene Funktion genau so benutzt, als sei sie über die Argumentliste übergeben worden.

---

* Um die mit Strukturen beliebiger Größe verbundenen Probleme zu vermeiden, waren in den Anfangszeiten von C Funktions-Rückgabewerte auf Grundtypen (inkl. Zeiger) beschränkt. Erst mit dem C89-Standard wurde verbindlich die Möglichkeit eingeführt, auch Strukturen als Rückgabewert einer Funktion zu verwenden.

Details bei der Unterprogramm-Rückkehr (3)

Rücksprung zum Aufrufer

Symmetrisch zum Aufruf sind hierbei zwei Schritte notwendig:*

- Erster Schritt
  - Die beim Einsprung in das Unterprogramms gesicherten Werte werden in die entsprechenden Register zurück geschrieben.
  - Der Program-Counter zeigt damit auf den nächsten Befehl und der Stack-Pointer steht so, wie er nach Übertragen der Aufrufargumente stand.
- Zweiter Schritt
  - Im Code des Aufrufers wird dafür gesorgt, dass der für diese Funktion eigentlich gültige Wert des Stack-Pointer restauriert wird.

---

* Da Unterprogramme seit langer Zeit wesentlicher Bestandteil höherer Programmiersprachen sind, verfügen viele moderne CPU-Architekturen - insbesondere hinsichtlich des Rücksprungs - über Maschinenbefehle, welche dabei helfen, die hier im Detail dargestellten Abläufe effizient zusammenzufassen.

Dynamische Speicheranforderung

Grundsätzlich ist dies eine Operation, die vom ablaufenden Programm explizit angestoßen werden muss:

- In C mit:
  - `malloc` - angegeben wird die Speichergröße in Byte
  - `calloc` - ähnlich wie zuvor aber Multiplikator*
  - `realloc` - spätere Änderung der Größe*2
- In C++ mit:
  - `new T` - Größe wird daraus als `sizeof T` bestimmt
  - `new T[N]` - ähnlich wie zuvor aber mit Multiplikator `N`

Rückgabewert ist jeweils ein Zeiger auf den dynamisch bereitgestellten Speicher. In C++ läuft dabei automatisch der T-Konstruktor ab, bzw. `N` solcher Konstruktoren.

---

*: Prinzipiell multipliziert. `calloc` die beiden Werte. Der erste bestimmt dabei das Alignment. Ferner wird der Speicherinhalt explizit gelöscht (mit Null initialisiert), anders als bei `malloc`, das den Speicherbereich so wie vorgefunden zurückliefert.

2: Falls `realloc` den bereits zugeordneten Speicherbereich nicht vergrößern kann, wird dessen Inhalt asn diejenige Stelle im Speicher verschoben, dessen Adresse als Ergebnis geliefert wird.

Dynamische Speicherfreigabe

Grundsätzlich ist dies eine Operation, die vom ablaufenden Programm explizit angestoßen werden muss:

- In C mit:
  - `free` - anzugeben ist dabei der bei der Anforderung erhaltene Zeiger
- In C++ mit:
  - `delete` - anzugeben ist ein von `new T` erhaltener Zeiger
  - `delete[N]` - anzugeben ist ein von `new T[N]` erhaltener Zeiger

In C++ läuft dabei automatisch der T-Destruktor ab.

Stack-Limit

Da die verschiedenen Arten der Datenablage prinzipiell um den selben Speicher konkurrieren, muss verhindert werden, dass sie ineinander überlaufen. Insbesondere beim Stack kann dies ein Problem sein, denn der SP wird so oft und von so vielen Maschinenbefehlen auch implizit verändert.

Folgende Techniken können zur Anwendung kommen:*

- spezielle CPU-Register
- nicht zugeordnete Speicherseite
- generieren von "sicherem Code"
- statische Vorab-Analyse

* Kommt keine dieser Möglichkeiten in Betracht bleibt nur beten und hoffen ...

Stacklimit mit speziellen CPU-Registern überwachen

Die wichtigste Voraussetzung ist hier natürlich, dass die CPU entsprechende Register bietet!

- Diese werden auf die Stackgrenzen gesetzt.
- Verlässt der SP den zulässigen Bereich, wird typischerweise ein Interrupt ausgelöst.

In der Interrupt-Reaktion muss eine Notfallmaßnahme greifen:

- Unterstützt ein Betriebssystem die Programmausführung, wird der Prozess beendet.
- Ohne diese Unterstützung bleibt meist nur der Neustart (Warm-Boot) als Ausweg.

Stacklimit mit nicht zugeordneter Speicherseite überwachen

Dies setzt eine MMU und deren - zumindest rudimentäre - Verwaltung durch ein Betriebssystem voraus.

(i) Grundidee ist, zwischen Stack-Pointer und `brk`-Adresse immer mindestens eine, nicht physikalisch zugeordnete Speicherseite zu frei zu lassen.

Kommt es nun zu einem Page-Fault aufgrund des nicht zugeorneten Speichers

- bei Aufruf eines Unterprogramms, wird dem Stack eine weitere Seite zugeordnet;
- bei Verschieben der `brk`-Adresse, wird dem Heap eine weitere Seite zugeordnet.

Einzige Ausnahme ist, dass die neue Seite die letzte zwischen Stack und Heap wäre. In diesem Fall wird das Programm abgebrochen.

Stacklimit-Probleme durch "sicheren Code" vermeiden

Wenn keine anderen Mittel zur Verfügung stehen und "Sicherheit vor Schnelligkeit" geht, könnte auch

- vor jedem `push` (Register auf Stack legen)
- vor jedem `jsr` (Unterprogramm aufrufen)

der aktuelle Wert des Stack-Pointers mit der `brk`-Adresse verglichen werden. Ist der aktuelle Abstand nicht mehr ausreichend für die anstehende Operation, wird das Programm abgebrochen.

> Voraussetzung ist hier natürlich, dass verwendete Compiler eine solche Code-Generierung unterstützen muss.

Stacklimit durch statische Analyse ausschließen

Eine andere Möglichkeit - vor "Beten und Hoffen" - wäre schließlich die, sämtliche Unterprogrammaufrufe zu analysieren und deren maximale Schachtelung* zu ermitteln.

Dies per Hand zu erledigen ist aufwändig und fehlerträchtig. Die Unterstützung durch ein statisches Analyse-Werkzeug wäre somit wünschenswert.

Weitere Voraussetzungen sind:

- Unterprogramme dürfen weder direkt noch indirekt rekursiv sein
- und die Schachtelungstiefe darf nicht datenabhängig sein.

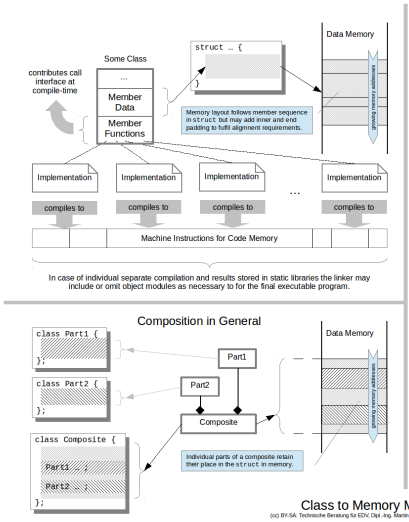
*: Das Wort "maximal" bezieht sich auf den Bedarf an Stackspeicher, nicht auf die Anzahl der aktiven Unterprogramme.

Abbildung von Klassen

- Abbildung von Member-Daten
- Abbildung von Member-Funktionen

- Öffentliche Basisklassen
- Private Basisklassen

- Komposition



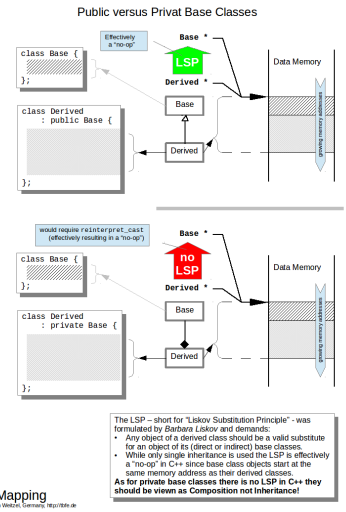


Abbildung von Member-Daten

Grundsätzlich wird aus den Member-Daten einer Klasse eine Struktur gebildet:

- Die Reihenfolge bleibt dabei erhalten:
  - im Quelltext nachfolgende Member stehen im Speicher an einer größeren Addresse;
  - sofern aus Alignment-Gründen notwendig gibt es zwischen den Membern ungenutzten Speicher (Padding);
  - auch am Ende der Struktur kann ein Padding erforderlich sein;

Vordergründig scheint das Padding am Ende nur für den Fall notwendig zu sein, dass von der betreffenden Struktur Arrays gebildet werden, es wird jedoch von C und C++ stets als Bestandteil der Struktur bzw. der Member-Daten einer Klassen gesehen.

Padding und Alignment

Der allgemeine Grund für Padding in Strukturen sind Alignment-Anforderungen der Hardware

- Oft kann nicht jeder Datentype an einer beliebigen Stelle im Speicher stehen sondern erfordert die Ausrichtung auf ene bestimmte Adresse-Grenze, Beispielsweise könnte es notwendig sein, dass
  - 32-Bit Ganzzahlen an einer durch 4 teilbaren Adresse und
  - 64-Bit Ganzzahlen an einer durch 8 teilbaren Adresse stehen.
- Da Strukturen prinzipiell auch als Arrays angelegt werden können, muss ferner die Struktur insgesamt gemäß den strengstens Aligmment-Anforderungen der in ihr enthaltenen Elemente ausgerichtet sein.

> Würde die zweite Regel nicht befolgt, hätten bei Arrays von Strukturen die Elemente der Struktur unterschiedlichen Offsets zum Beginn der Struktur.

Padding und Alignment (2)

Padding zwischen Strukturelementen

Die Garantie, dass die Elemente einer Struktur gemäß ihrer Reihenfolge im Quelltext an aufsteigenden Adressen abgelegt werden, erfordert bei

```
struct s {
 short a; // assume 16 bit for short
 long b;  // assuming 32 bit for long
}
```

das Einfügen von vier (ungenutzten) Bytes zwischen a und b . Die Größe der Struktur (`sizeof (struct s)`) ist damit 16 (4+4+8) Bytes.

Padding und Alignment (3)

Padding nach dem letzten Strukturelement (Trugschluss)

Bei der umgekehrten Anordnung

```
struct s {
 long b;  // assuming 32 bit for long
 short a; // assume 16 bit for short
}
```

scheinen dagegen 12 Bytes ausreichend zu sein, da zwischen b und a nichts eingefügt werden muss ...

Padding und Alignment (4)

Padding nach dem letzten Strukturelement (Nowendigkeit)

... allerdings würde ohne eventuelles Padding am Ende gegen eine seit den Anfängen von C gültige Regel* verstoßen, die besagt dass für jedes Array

```
struct s { ... } array[N];
```

die Beziehungen

- `(sizeof array / sizeof (struct s)) == N` und
- `(sizeof array / array[0]) == N`

stets erfüllt sind. Damit ist unter anderem garantiert, dass sich die Anzahl der Elemente in einem initialisierten Array

```
struct s array[] = { {1, 2}, {3, 4}, {5, 6}};
```

in jedem Fall mit den obigen Ausdrücken berechnen lassen und die Anordnung der Elemente in der Strukturdefinition dafür keine Rolle spielt.

Abbildung von Member-Funktionen

Hier ist zum einen zu unterscheiden zwischen Member-Funktionen mit und ohne den Zusatz `inline`.

Bei Funktionen, die zu echten Unterprogrammen kompiliert werden (nicht `inline`), ist ferner zu unterscheidem zwischen

- Statischem Linken und
- Dynamischem Linken.

Reguläre vs. Inline-Member-Funktionen

Ursprünglich war das Schlüsselwort `inline` nur dazu gedacht, dem Compiler einen Hinweis zu geben, dass für eine damit markierte Funktion der enthaltene Code direkt an der Aufrufstelle eingesetzt werden soll. Der konnte diesen Hinweis zu ignorieren.

Mittlerweile hat sich die Situation umgekehrt:

> Zumindest im Fall höherer Optimierungsstufen entscheiden viele Compiler mittlerweile von sich aus, ob sie statt eines Unterprogramm-Aufrufs den Code einer Funktion direkt an der Aufrufstelle *auch dann* einsetzen, wenn die betreffende Funktion *nicht* mit `inline` markiert ist.

Andererseits ignoriert z.B. der GCC Inline-Funktionen, wenn alle Optimierungen abgeschaltet sind (Debug-Kompilierung), um so auch das Setzen von Break-Points in solchen Funktionen zu vereinfachen.

Inline Member-Funktionen

Grundsätzlich handelt es sich bei Inline-Funktionen um den Tausch von Geschwindigkeit gegen Programmgröße:

- Inline-Funktionen werden den Programmcode größer* machen.
- Dafür werden sie schneller ausgeführt.

Es gibt allerdings eine wichtige Ausnahme:

Für sehr einfache Inline-Funktionen - insbesondere typische *Getter* und *Setter* - machen Inline-Funktionen nicht nur die Programmausführung *schneller* sondern auch den Code insgesamt *kleiner*!

* Wieviel mehr Code hängt natürlich ab von der Anzahl der Stellen im Gesamtprogramm

Nicht-Inline Member-Funktionen

Für diese wird der Code nur einmal erzeugt und im Speicher abgelegt während an den Aufrufstellen nur

- die Argumente versorgt werden und anschließend
- ein Unterprogramm-Sprung erfolgt.

Sofern die Funktion eine nennenswerte Größe hat, ist der Unterprogrammsprung allerdings der weniger entscheidende Nachteil.

> Eine viel wichtigere Rolle spielt bei modernen CPU-Architekturen die Tatsache, dass durch die Verzweigung der sequntielle Ausführung unterbrochen wird.

Dies hat zur Folge, dass bei Eintritt in das Unterprogramm und bei dessen Verlassen die Pipeline mit bereits teilweise dekodierten Maschinenbefehlen und diverse Cache-Speicher wieder neu gefüllt werden müssen.

Nachteile von Inline-Funktionen

Weitere Nachteile von Inline-Funktionen können sich aus folgenden Punkten ergeben:

- Sie können als nicht zugleich `virtual` sein* und
- die Implementierung muss im Header-Files erfolgen.

Ersteres schränkt den Umfang ein, in dem abgeleitete Klassen von den Basisklassen geerbte Member-Funktionen überschreiben können, letzteres erhöht die Abhängigkeiten zur Compile-Zeit.

* Rein technisch gesehen kann eine Member-Funktion auch `virtual inline` sein. Dies wird allerdings in vielen Fällen dazu führen, dass eine solche Funktion als potenziell polymorph betrachtet werden muss und trotz `inline` vom Compiler Code für ein echtes Unterprogramm und dessen Aufruf erzeugt wird.

Öffentliche Basisklassen

Gemäß der objektorientierten Sichtweise entspricht dies der *Vererbung*.

Aus Sicht auf die Member-Daten handelt es sich um eine reine Verschachtelung:

- Die Daten der für die Bassklasse erzeugten Struktur sind
- in den Daten der für die abgeleitete Klasse erzeugten Struktur direkt enthalten.

Zusätzlich gilt das LSP (Liskov'sches Ersetzungsprinzip), gemäß dem ein ggf. erforderlicher Up-Cast (`Derived => Base`) vom Compiler automatisch vorgenommen wird.

Da die Daten der Basisklasse direkt am Anfang der abgeleiteten Klasse liegen, erfordert das LSP zur Laufzeit keinerlei Code!

Private Basisklassen

Gemäß der objektorientierten Sichtweise entspricht dies der Komposition. So gesehen sind private Basisklassen in C++ lediglich eine Alternative zur typischen Vorgehensweise, Komposition über Member-Daten zu realisieren.

- Zusätzlich hat bei einer privaten Basisklassen aber die abgeleitete Klasse die Möglichkeit, geerbte Methoden zu überschreiben.
- Daher kann das *Template Methode Pattern* des GoF-Buchs in C++ auch mit einer geringerer Kopplung zwischen Basisklasse und abgeleiteter Klassen umgesetzt werden.

Das Speicher-Layout ist zwar dasselbe wie bei Vererbung, es gilt aber *nicht* das LSP, d.h. ein `Derived`-Objekt wird nicht stillschweigend ein Base-Objekt substituieren.

Komposition im Allgemeinen

Komposition wird in C++ im allgemeinen dadurch realisiert, dass eine Klasse (die Gesamtheit) eine andere Klasse (das Teil) im Rahmen ihren Member-Daten direkt enthält.

- Durch die Anordnung der Daten-Member im Speicher, welche zu steigenden Adressen hin der Reihenfolge im Quelltext entspricht, liegt das Teil aber nicht zwingend am Anfang der Gesamtheit.
- Des weiteren muss bei Bezugnahme auf das Teil dessen Name explizit verwendet werden, während im Fall der Basiklasse
  - bei Eindeutigkeit keine weitere Qualifikation notwendig ist, und
  - ansonsten der Klassenname als Scope-Operator (`Base::`) vorangestellt wird.

Techniken zur Implementierung von Containern

- Ableitung von der Elementklasse
- Verwendung von Templates

- Containern mit polymorphen Elementen

Auf Vererbung basierende Technik

Hierbei wird in der Elementklasse nur die Verwaltungsinformation definiert - bei einer einfach verketteten Liste ist das lediglich der Zeiger auf das nächste Element.

Um möglichst wenige Bezeichner in den globalen Namensraum einzubringen, erscheint es sinnvoll, die Elementklasse geschachtelt zu definieren:*

```
class Lifo {
 class Node {
  friend class Lifo;
  Node *next;
 protected:
  Node() : next(nullptr) {}
 };
 // ...
};
```

* Durch ausschließlich nicht-öffentliche Member in der Elementklasse und die `friend`-Beziehung wird die Kapselung verbessert*, da auf den `next`-Zeiger jetzt nur durch die `Lifo`-Klasse zugegreifen kann. Insbesondere in einigen älteren C++ Bücher findet sich zu Friend-Beziehungen immer wieder die Aussage, dass diese den "Zugriffsschutz aufweichen" würden. Bei einem angemessenen Einsatz dieses Sprachmechanismus ist jedoch das Gegenteil der Fall!

Container-Elemente mit Daten

Um in den Elementen des Containers auch tatsächlich einen Datenteil unterbringen zu können, müssen entsprechende Klassen abgeleitet werden:

```
class Double_Node : public Lifo::Node {
public:
 Double_Node(double d) : data(d) {}
 double data;
};
```

```
#include <string>
class Double_Node : public Lifo::Node {
public:
 String_Node(const std::string &s) : data(s) {}
 double std:string;
};
```

Wie man sieht, ist der Code solcher Klassen sehr systematisch.

Daten-Elemente als Templates

Dieses immer wieder nahezu gleiche Aussehen der abgeleitete Datenklassen legt dafür die Verwendung von Templates nahe:*

Durch die Ähnlichkeit im Quelltext ist die Zusammenfassung naheliegend:*

```
template<typename ELemType>
class Data_Node : public Lifo::Node {
public:
 Data(const ElemType &d) : data(d) {}
 ElementType data;
};
```

Dies hat noch nichts mit der auf Templates basierenden Implementierung von Containern zu tun, es geht hier einzig und allein darum, den sehr ähnlich aussehenden Quelltext für die Klassen der Daten-Elemente im Sinne des DRY-Principles zusammenzufassen!

* Das Argument des Konstruktors wird nun zwar auf jedem Fall per Referenz übergeben, was im Allgemeinen zu einem leichten Overhead für Grundtypen geringer Größe führen wird. Im obigen Fall handelt es sich allerdings um eine Inline-Funktion handelt (durch Implementierung in der Klasse selbst), womit effektiv überhaupt keine Argumentübergabe stattfinden wird.

Datenelemente erzeugen und einfügen

Dies kann für ein `Lifo c` ggf. in einem einzigen Schritt geschehen:

```
c.push(new Double_Node(47.11));

c.push(new String_Node("hello, world!"));
```

Oder mit Templates für die Klassen der Datenelemente:

```
c.push(new Data_Node<double>(47.11));

c.push(new Data_Node<std::string>("hello, world!"));
```

Datenelemente entnehmen

Hier zeigt sich nun ein gravierender Nachteil. Das grundsätzliche Vorgehen sieht so aus:

```
Lifo c;
// ...
Lifo::Node *p = c.pop();
```

- Im Rahmen der Implementierung des Containers ist nichts anderes als die `Lifo::Node` Klasse bekannt gewesen.
- Insofern kann bei der Entnahme mit der Member-Funktion `Lifo::pop` auch nur ein solcher, allgemeiner Elementzeiger zurückgegeben werden.
- Für diesen Zeiger muss zur weiteren Verwendung ein expliziter Down-Cast erfolgen.

Datenelemente entnehmen (2)

Datenelemente verwendbar machen mit `dynamic_cast` auf Zeigerbasis

Die sichere Variante verwendet einen Down-Cast mit Prüfung des Laufzeit-Typs:*

```
Double_Node *p = dynamic_cast<Double_Node*>(c.pop());
if (p != nullptr) {
 // OK, has expected type
 // may access p->data now
}
else {
 // Oops - not a Double_Node ??
}
```

Datenelemente entnehmen (3)

Datenelemente verwendbar machen mit `dynamic_cast` auf Referenzbasis

Wenn im Fehlerfall ohnehin ein Abbruch erfolgen müsste (da die falsche Datenart völlig unerwartet vorgefunden wurde), geht es auch so:

```
Double_Node &n = dynamic_cast<Double_Node&>(*c.pop());
// if the cast didn't throw, n.data may be accessed now
```

Mehr zu `dynamic_cast` und ein Vergleich zur weniger sicheren `static_cast` folgt später.

Verantwortlichkeit für die Speicherverwaltung

Bei der auf Ableitung von den Elementklassen bestehenden Vorgehensweise liegt die Verantwortung für die Speicherverwaltung außerhalb der Containerklasse.

- Die Datenelemente werden vor dem eigentlichen Einfügen im dynamischen Speicher erzeugt.
- Daher muss auch das Löschen nach der entnahme explizit erfolgen.

```
Double_Node *p = dynamic_cast<Double_Node*>(c.pop());
if (p != nullptr) {
 // OK, has expected type
 // may access p->data now
 delete p;
}
```

Dieser Code weist allerdings das Problem auf, dass ein anderer als der erwartete Datentyp zu einem Memory-Leak führen könnte, da dann kein `delete` erfolgt.

Speicherfreigabe über Zeiger auf Basisklassen

Ein möglicher Ausweg kann so aussehen:

```
Lifo::Node *p = c.pop();
if (Double_Node *dp = dynamic_cast<Double_Node*>(p)) {
 // OK, has expected type
 // may access p->data now
}
```

Nun könnten noch weitere, ähnliche Tests auf andere mögliche Elementtypen* folgen.

```
// release dynamic memory
delete p;
```

Allerdings ist Code wie der obige nur dann korrekt, wenn der Destruktor `Lifo::Node::~Node()` virtuell ist. (Dazu später mehr.)

---

* Die spezielle, hier verwendete Syntax der `if`-Anweisung vereinfacht übrigens das *Copy&Paste* dieser Abschnitte etwas, da die Variable dp jeweils nur lokale Sichtbarkeit im Block nach dem Bedingungstest hat.

Auf Templates basierende Technik

Bei dieser mittlerweile viel gängigeren Alternative wird zunächst die `Lifo`-Klasse selbst als Template definiert:

```
template<class ElemType>
class Lifo {
 class Node;
public:
 void push(const ElemType &);
 void pop(ElemType &);
};
```

Template für Klasse der Datenelemente

Durch die geschachtelte Definition der Elementklasse wird auch diese zur Template und kann damit den Datenteil ganz "offiziell" enthalten:

```
template<class ElemType>
class Lifo<ElemType>::Node {
 Node *next;
 ElemType data;
 Node(Node const* n, const ElementType& d)
  : next(n), data(d)
 {}
 friend class Lifo<ElemType>;
};
```

Operation zum Einfügen neuer Datenelemente

Mit der obigen kleinen Änderung des Designs, die den next-Zeigers über ein Argument des Konstruktor initialisiert, ergibt sich eine sehr einfach Implementierung der Einfüge-Operation:

```
template<class ElemType>
void Lifo<ElemType>::push(const ElemType &d) {
 head = new Node(head, d);
}
```

Nach wie vor werden die Datenelemente im dynamischen Speicher angelegt, allerdings nun gekapselt im Code der Container-Klasse.

Operation zur Entnahme von Datenelemente

Da die Speicherverwaltung nun in die Verantwortung der `Lifo`-Klasse übergegangen ist, gehört die Freigabe des dynamischen Speichers natürlich ebenso dorthin:

```
template<class ElemType>
bool Lifo<ElemType>::pop(ElemType &d) {
 if (head == nullptr)
  return false;
 auto p = head;
 d = p->data;
 head = p->next;
 delete p;
 return true;
}
```

Ein virtueller Destruktor ist nun nicht mehr erforderlich, da die `delete` Anweisung nicht über einen Basisklassen-Zeiger erfolgt.

Destruktor der Container-Klasse

Aufgrund des Übergangs der Verantwortlichkeit für die Verwaltung des dynamischen Speichers der Datenelemente sollte spätestens jetzt ein Destruktor eingeführt werden:

```
Lifo::~Lifo() {
 auto p = head;
 while (p != nullptr) {
  const auto p_next = p->next;
  delete p;
  p = p_next;
 }
}
```

Ob ein ähnlicher Destruktor auch für die andere Implementierung sinnvoll wäre oder gar notwendig ist, hängt von eingehenden Analysen des Codes ab*, wie der Container tatsächlich benutzt wird.

---

* Ohne eine solche Analyse wird die Sache leicht zum Grund für Memory-Leaks - nämlich wenn es *keinen* solchen Destruktor gibt aber eigentlich einen geben müsste; oder es kommt zu vorzeitig freigegebenem Speicherplatz für noch benutzte Datenelemente - wenn es einen solchen Destruktor gibt aber eigentlich *keinen* geben dürfte).

Typsicherheit beim Einfügen

Neben der sicheren Speicherverwaltung macht der Blick auf die Implemetierung einen anderen Vorteil dieser Technik klar:

Die Schnittstelle zum Einfügen und Entnehmen von Elementen ist nun typsicher:

```
Lifo<double> c1;
Lifo<std::string> c2;

// OK
c1.push(3.14);
c2.push("hi!");

// Compile-Error!!
c1.push("hi!");
c2.push(3.14);
```

Typsicherheit beim Entnehmen

Dies gilt ebenso für die Entnahme von Elementen:

```
double d;
std::string s;

// OK
c1.pop(d);
c2.pop(s);

// Compile-Error!!
c1.pop(s);
c2.pop(d);
```

Slicing als Nachteil der Templates

Die bessere Sicherheit vor Memory-Leaks durch die Übertragung der Speicherverwaltung in die Vertanwortung der `Lifo`-Klasse und der Gewinn an Typsicherheit beim Einfügen und Entnehmen haben auch eine Kehrseite:

> **[!]** Eventuell erwünschter Polymorphismus für die Container-Elemente geht zunächst verloren.

Mit einer Basisklasse für Früchte allgemein (`Fruit`) und abgeleiteten Klassen für spezifische Früchte (`Apple` `Banana`, `Kiwi`, ...) ist `Lifo<Fruit>` *nicht* der evtl. erwartete "Obstkorb":

- Beim Einfügen wird von den speziellen Fruchtklassen der spezifische Anteil entfernt (Slicing) und
- bei der Entnahme kommt somit nur der unspezifische (als `Fruit` implementierte) Teil zurück.

Template-Technik mit explizitem Polymorphismus

Wird bei auf Templates basierenden Container Polymophismus hinsichtlich der Datenelemente gewünscht, muss explizit ein Container von Zeigern verwendet werden.

```
Lifo<Fruit *> allMyFruits;
```

Allerdings geht nun die Verantwortung für den Speicherplatz wieder auf denjenigen über, der den Container benutzt.

Das Einfügen muss nun so erfolgen:

```
// fill some ...
allMyFruits.push(new Banana( ... ));
allMyFruits.push(new Apple( ... ));
```

Template-Technik mit explizitem Polymorphismus (2)

Die Entnahme sieht prinzipiell so aus:

```
// get back ...
Fruit *f = nullptr;
while (allMyFruits.pop(f)) {
 // got another one!
 // now wash it, peal it, eat it,
 // whatever ...
 delete f;
}
```

Für die Korrektheit des obigen Codes muss natürlich `Fruit::~Fruit()` virtuell sein!

Template-Technik mit explizitem Polymorphismus (3)

Schließlich muss im Anschluss an die Entnahme einer Frucht geprüft werden, was man eigentlich bekommen hat:

```
// got another one!
if (auto &apple = dynamic_cast<Apple&>(f)) {
 // look it's an apple
 // wash it, eat it
}
if (auto &banana = dynamic_cast<Banana&>(f)) {
 // look, its a banana
 // peal it, eat it
}
```

Während die gezeigte Vorgehensweise funktioniert, kann sie unter dem Aspekt eines "echten" objekt-orientierten Ansatzes kritisiert werden.

Eine Design-Alternavtive zu typ-basierten Mehrfachverzweigungen wird später noch ausführlich erläutert und ist auch Thema einer Übung.

Pointer in auf Templates basierenden Containern

Besonders durch die damit verbundene Bequemlichkeit bei den STL-Containern wird die Wert-Semantik schnell zur Gewohnheit. Damit geht bei expliziten Zeigern in Containern oft der Blick für die damit verbundenen Gefahren und besonderheiten verloren.

> **[!]** Beim gerade gezeignet "Früchtekorb" besteht beispielsweise das Riskio eines Memory-Leaks, wenn ein nicht-leerer Container dieser Art mittels des gerade dafür empfohlenen Destruktors seinen Inhalt löscht.

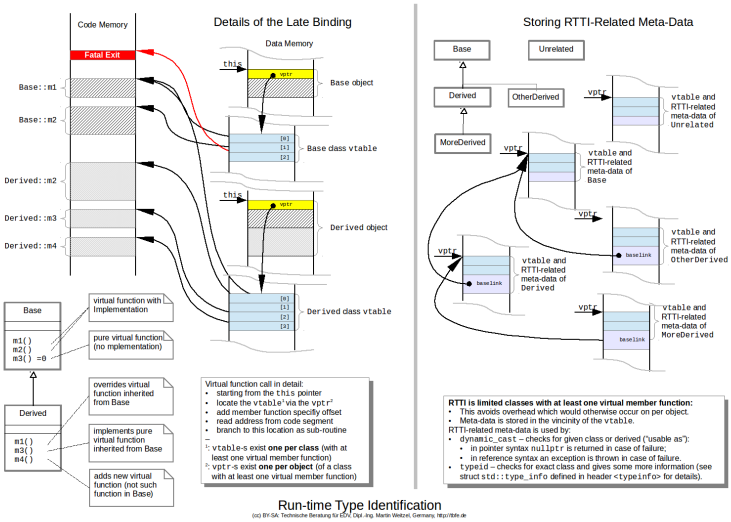
ResourceWrapper-Klassen können hier Abhilfe schaffen, im konkreten Fall:

- `Lifo<std::unique_ptr<Fruit>>` oder
- `Lifo<std::shared_ptr<Fruit>>`

Diese werden später noch ausführlicher betrachtet.

Typ-Bestimmung zur Laufzeit

- Umsetzung von Dynamischem Polymorphismus
- Typ-Identifikation zur Laufzeit



RTTI-Voraussetzungen

Mit C++98 wurde die explizite Laufzeit-Typinformation (RTTI) zum verbinlichen Sprachfeature.

- Die Information der Klassenzugehörigkeit muss dafür im Objekt selbst untergebracht sein.
- Nicht jedes Objekt braucht diese Information und sollte unnötigen Overhead vermeiden können.

Implizit war eine Laufzeit-Typidentifikation immer schon Bestandteil von C++, indem die Ausführung virtueller Member-Funktionen vom Laufzeityp abhängt.

Dynamischer Polymorphismus

Hierbei handelt es sich um eine implizite Form der Laufzeit-Typindentifikation. Eine andere übliche Bezeichnung ist "spätes" oder "dynamisches" Binden (late binding).

Im Unterachied zum statischen Polymorphismus entscheidet dabei der Laufzeittyp über die Auswahl einer aufzurufenden Member-Funktion.

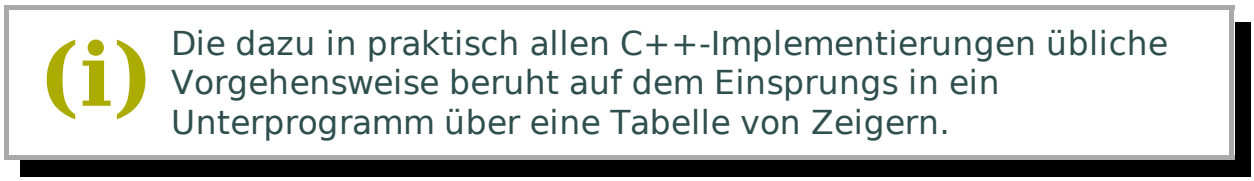
Voraussetzung dazu ist: *Der Aufruf erfolgt über eine Objekt-Referenz oder einen Objekt-Zeiger. Dieser legt dabei den Compilezeit-Typ fest, gemäß dem entschieden wird, **welche** Funktionen überhauot aufgerufen werden können.* Die aufgerufene Member-Funktion wurde in der betreffenden Klasse als `virtual` deklariert*

*: Da eine geerbte virtuelle Member-Funktion in der abgeleiteten Klasse virtuell bliebt, auch wenn sie dort nicht explizit so markiert ist, ist die `virtual` Deklaration in einer Basisklasse ausreichend.

Zweck des dynamischen Polymorphismus

Dabei geht es darum, den Laufzeittyp eines Objekts zu ermitteln um auf dieser Basis zu entscheiden, welche Version einer ggf. überschriebenen virtuellen Member-Funktion aufzurufen ist.

Implementierung des dynamischen Polymorphismus



Hierzu führt jedes Objekt, dass einer Klasse mit **mindestens einer** virtuellenen Member-Funktion angehört, **genau einen** zusätzlichen Zeiger mit, welcher auf eine Tabelle mit den Adressen *aller* ihrer virtuellen Member-Funktionen zeigt.

Da diese Tabelle wiederum **klassenspezifisch** ist, und muss sie in einem Gesamtprogramm nur **einmal pro Klasse** vorhanden sein.*

*: Dies gilt zumindest prinzipiell und hängt in der Praxis davon ab, dass bei einer modulweise getrennten Kompilierung der Linker ggf. vorsorglich pro Objekt-Modul angelegte Exemplare dieser Tabelle bis auf eines elminiert.

Explizites RTTI

Hierfür stehen zwei Mechanismen zur Verfügung:

- `dynamic_cast` und
- `typeid`

Ersterer prüft den Typ im Sinne des LSP. das heißt eine abgeleitete Klasse wird auch an Stelle der gewünschten akzeptiert, letzterer prüft exakt.

RTTI mit dynamischem Cast

Es gibt zwei Formen, mit denen z.B. geprüft werden kann, ob der z.B. ein Laufzeit-Typ die von `Base` abgeleitete Klasse `Derived` - oder noch tiefer abgeleitet - ist.

`dynamic_cast` auf Zeigerbasis

```
Base *p;
…
Derived *dp = dynamic_cast<Derived*>(p);
```

Hier wird `dp` ein `nullptr` sein, wenn die Prüfung negativ ausgeht und muss somit entsprechend geprüft werden.

RTTI mit dynamischem Cast (2)

`dynamic_cast` auf Referenzbasis

```
Base &r;
…
Derived &dr = dynamic_cast<Derived&>(d);
```

Hier wird eine Exception geworfen, wenn die Prüfung negativ ausgeht.

Dadurch kann die obige Form auch leicht als Bestandteil eines größeren Ausdrucks verwendet werden, etwa um eine member-Funktion aufzurufen, die erst von der abgeleiteten Klasse hinzegfügt wurde:

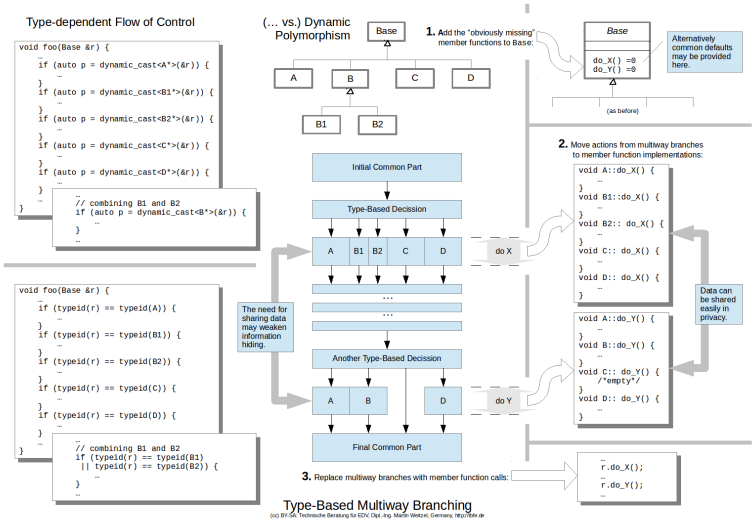
```
… dynamic_cast<Derived &>(r).anotherMF( … ) …
```

Typ-Identifikation zur Laufzeit

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Typbasierte Verzweigungen

- Typgesteuerter Kontrollfluss
- Alternative mit virtuellen Funktionen



Click here for Info-Graphic
TypeBasedBranching
Type-based Flow Of Control

Typgesteuerter Kontrollfluss

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Alternative mit virtuellen Funktionen

Siehe Info-Grafik zusammen mit den Ausführungen des Dozenten.

Übung

Ziel ist der Ersatz des typgesteuerten Kontrollflusses durch virtuelle Funktionen in einem Beispiel.

Weitere Details werden vom Dozenten anhand des Aufgabenblatts sowie der vorbereiteten Eclipse-Projekte erläutert.