C++ FOR
(Wednesday Afternoon)

Kürzere Pausen werden jeweils nach Bedarf eingelegt.

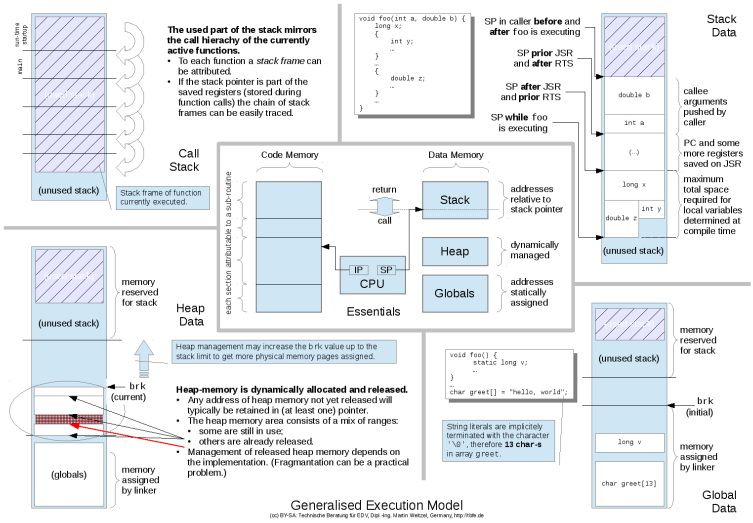
Die Besprechung der Musterlösung(en) erfolgt zu Beginn des folgenden Vormittags.

Generalisiertes Ausführungsmodell

- CPU und Speicher

- Stack-Daten
- Globalen Daten
- Dynamisch verwaltete Daten

- Stack-Frames (Verwaltungsinformation)



CPU und Speicher

Allgemein betrachtet sind dies die 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 ...)

CPU-Aufbau (Grundlegendes)

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

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 Stack aber davon ab, wohin sich der Stack-Pointer bei Maschinenbefehlen wie

- `push` (Daten: Register => Speicher) bzw. `pop` und
- `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. "branch 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 um:

- ALU
- Register
- Datenpfad
- Steuerungslogik

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 Akkumulator oder kurz Akku genannt.

*: Oder auch mehrere davon.

(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. Bei Verlassen dieses Bereich 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.

Datenpfad-Steuerung

Die verbleibenden Teile* einer typischen CPU sind im wesentlichen:

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

Letztere schalten erstere so um, dass Register, ALU und Speicher so miteinander, verbunden sind, wie es zur Ausführung des jeweiligen Maschinenbefehls erfordert.

*: Vom Umfang her können die Datenpfade und die zugehörige Steuerungslogik 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:

- Aufrufparameter
- 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:

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

*: Bei globalen Variablen spielt es keine Rolle, ob diese `static` sind oder nicht, da es sich in beiden Fällen um Speicherplatz im globalen Datenbereich handelt. Der Unterschied ist lediglich, dass globalen `static` Variablen mit **gleichem** Namen in unterschiedlichen Objektmodulen auch jeweils **unterschiedlicher** Speicherplatz zugeordnet werden muss.

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 bei

- Aufruf und
- Rückehr

von Unterprogrammen. Grundsätzlich sind dabei folgende Probleme zu lösen:

- Übergabe von Argumenten
- Speicherplatz für lokale Variable
- Entgegennahme 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 Unterprogramm-Einsprung und
- demjenigen Code, der im aufgerufenen Unterprogramm (Callee) ausgeführt wird.

> Hier bestehen grundsätzlich viele Freiheiten für den Compiler, aber nur solange sichergestellt ist, dass *Caller* und *Callee* zusammenpassen.*

*: Hierum geht es prinzipiell wenn die Einhaltung bestimmter Calling Conventions gefordert wird.

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 Werte dann zur Verwendung durch das aufgerufene Unterprogramm
  - auf den Stack legt bzw.
  - in bestimmten Registern hinterlässt.
- In welcher Form Stack und Register dabei genau benutzt werden, regeln die Calling Conventions.

Maschinenbefehl zum Unterprogrammsprung

Im Rahmen des Unterprogramm-Einsprungs wird mindestens der Program-Counter auf den Stack gelegt:

- 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 schon beim Holen des `jsr`-Befehls weitergeschaltet.
- Der auf dem Stack gesicherte Wert zeigt somit auf den Befehl direkt dahinter.

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

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.

Erzeugen von Stack-Backtraces

Mit einem auf dem Stack selbst gesicherten Stack-Pointer – so wie er unmittelbar vor dem Unterprogramm-Einsprung gültg war, besteht eine

- leicht zu verfolgende Rückwärtsverkettung

zur Main-Funktion (und von dort weiter ins Runtime-Startup-Modul).

Mit deren Hilfe können beim "post-mortem"-Debugging anhand des Speicherabzugs die zum Zeitpunkt eines Programmabsturzes aktiven Funktionen rekonstruiert werden.

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.

> Lokale Daten eines Unterprogramms werden (ggf. initialisiert und) 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 zum Stack-Pointer ausnahmslos von allen modernen CPUs in effizienter Form unterstützt.

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

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

Innerhalb eines Blocks mit `static` definierte Variable sind von der Lebensdauer her *nicht* an die Ausführung eines Blocks oder einer Funktion gebunden und werden daher bei den globalen Variablen abgelegt.

*: Genauer gesagt ergeben sich die Anfangswerte von nicht explizit initialisierten Variablen aus der vorherigen Nutzung des Stackbereichs, in dem sie liegen.

Details bei der Unterprogramm-Rückkehr

Beim Verlassen eines Unterprogramms werden diejenigen Aktionen rückgängig gemacht, die beim Aufruf stattfanden:

- Zunächst muss der Rückgabewert 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 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.

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 Register zurückgegeben.
- Dies gilt auch für kleinere Datenstrukturen.
- Für größere Datenstrukturen 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 Rückgabewerte von Funktionen auf Grundtypen (inkl. Zeiger) beschränkt. Erst mit dem C89-Standard wurde verbindlich die Möglichkeit eingeführt, beliebige Strukturen als Rückgabewert einer Funktion zu verwenden.

Rücksprung zum Aufrufer

Symmetrisch zum Aufruf sind hierbei zwei Schritte notwendig:*

- Erster Schritt
  - Die beim Einsprung in das Unterprogramms gesicherten Register werden restauriert.
  - 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.

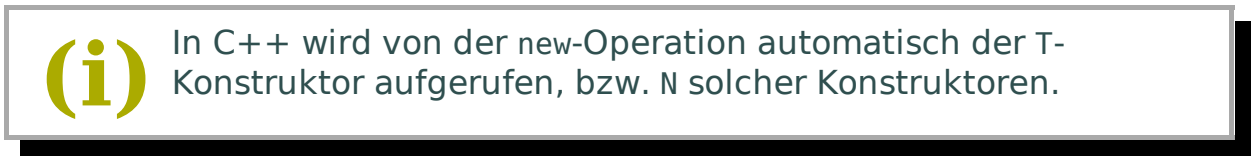
*: Da Unterprogramme seit langer Zeit wesentlicher Bestandteil höherer Programmiersprachen sind, verfügen viele moderne CPU-Architekturen – insbesondere hinsichtlich des Rücksprungs aus Unterprogramen – über Maschinenbefehle, welche die hier im Detail dargestellten Abläufe effizient zusammenzufassen.

Dynamische Speicherverwaltung

Speicheranforderung

Grundsätzlich muss dies Operation explizit angestoßen werden.
Rückgabewert ist ein Zeiger auf den dynamisch bereitgestellten Speicher.

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

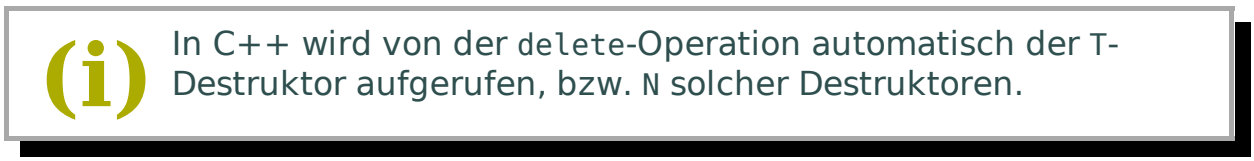


*: Prinzipiell multipliziert `calloc` die beiden Werte. Der erste bestimmt dabei das Alignment. Ferner wird der Speicherinhalt explizit gelöscht (mit Null initialisiert), anders als bei `malloc`, das den Speicherbereich so wie vorgefunden zurückliefert. Falls `realloc` den bereits zugeordneten Speicherbereich nicht vergrößern kann, wird dessen Inhalt an diejenige Stelle im Speicher verschoben, deren Adresse als Ergebnis geliefert wird.

Dynamische Speicherfreigabe

Grundsätzlich ist dies eine Operation, die 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[]` – anzugeben ist ein von `new T[N]` erhaltener Zeiger



Typische Fehlerquellen bei dynamischem Speicher

Folgende Sitituationen erfordern im Umgang mit dynamischen Speicher besondere Beachtung:

- Es kann kein weiterer Speicher verfügbar gemacht werden.
  - Seit C++98 hat dies eine Exception zur Folge.
  - Es kann durch einen entsprechenden Handler aber auch das vor C++98 angewendete Verfahren aktiviert werden, nämlich dass unzureichender Hauptspeicher von `new` durch Rückgabe eines Null-Zeigers gemeldet werden.

Weiterhin im Verantwortungsbereich des Software-Entwicklers liegt die korrekte Kombination

- von `new` mit `delete`,
- von `nev[N]` mit `delete[]`, und
- von `malloc`, `calloc`, `realloc` mit `free`*

---

*: Seit C89 kann bei `free` genau wie bei `delete` davon ausgegangen werden, dass die Übergabe eines Null-Zeigers (oder `nullptr` in C++11) wirkungslos bleibt.

Stack-Limit

Da die verschiedenen Arten der Datenablage prinzipiell um den selben Speicher konkurrieren, muss verhindert werden, dass die einzelnen, für unterschiedliche Zwecke verwendeten Bereiche nicht ineinander hinein laufen. Insbesondere beim Stack kann dies ein Problem sein, denn der SP wird sehr oft und von sehr 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 (memory Management Unit) voraus und deren – zumindest rudimentäre – Verwaltung durch ein Betriebssystem.

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

Kommt es nun zu einem Page-Fault aufgrund des nicht zugeordneten Speichers, wird

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

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

> Voraussetzung ist hier natürlich, dass verwendete Compiler eine solche Code-Generierung zumindest optional unterstützt.

Stacklimit durch statische Analyse ausschließen

Eine andere Möglichkeit – vor "Beten und Hoffen" – wäre schließlich die, sämtliche Unterprogrammaufrufe zu analysieren und deren maximalen Stackbedarf zu ermitteln.

Dies per Hand zu erledigen ist aufwändig und fehlerträchtig, da die Schachtelung ja auch von Programmablauf abhängt. Die Unterstützung durch ein statisches Analyse-Werkzeug ist in diesem Fall mehr als nur wünschenswert.

Weitere Voraussetzungen sind:

- sind Unterprogramme direkt oder indirekt rekursiv, muss die maximale Tiefe bestimmt werden;
- insbesondere darf diese nicht datenabhängig sein!

Abbildung von Klassen

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

- Öffentliche Basisklassen
- Private Basisklassen

- Komposition

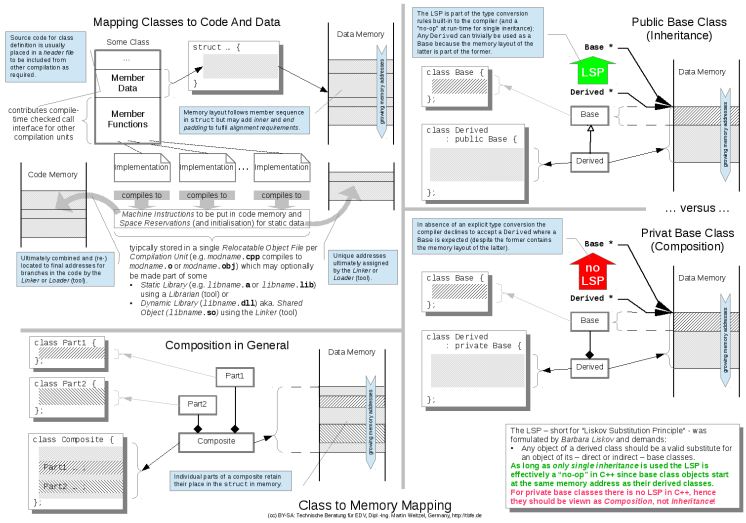


Abbildung von Member-Daten

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 Klasse gesehen.

Padding und Alignment

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

- Oft kann nicht jeder Datentyp an einer beliebigen Stelle im Speicher stehen sondern erfordert die Ausrichtung auf ene bestimmte Adress-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 Alignment-Anforderungen der in ihr enthaltenen Elemente ausgerichtet sein.

Würde die zweite Regel nicht befolgt, hätten bei Arrays von Strukturen die Elemente der Struktur an unterschiedlichen Indizes unterschiedlichen Offset.

Padding zwischen Strukturelementen

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

```
struct s {
    unsigned a;  // assume 32 bit for int
    long long b; // assume 64 bit for long 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 nach dem letzten Strukturelement (Trugschluss)

Bei der umgekehrten Anordnung

```
struct s {
    long long b; // assume 64 bit for long long
    int a;       // assume 32 bit for short
}
```

scheinen dagegen 12 (= 8+4) Bytes ausreichend zu sein, da zwischen `b` und `a` nichts eingefügt werden muss ...

Padding nach dem letzten Strukturelement (Nowendigkeit)

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

```
struct s { … } arr[N];
```

die Beziehungen

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

erfüllt sein muss. 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`) kompiliert werden, 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 kann. Der Compiler konnte diesen Hinweis auch ignorieren.

Mittlerweile hat sich die Situation umgekehrt:

> Zumindest im Fall höherer Optimierungsstufen entscheiden viele Compiler von sich aus, dass sie statt eines Unterprogramm-Aufrufs den Code der Start-Funktion direkt an der Aufrufstelle auch dann einsetzen, wenn diese Funktion *nicht* mit `inline` markiert tst.

Andererseits ignoriert z.B. der GCC Inline-Funktionen, wenn alle Optimierungen abgeschaltet sind (Debug-Kompilierung), und sollte so auch das Setzen von Break-Points Jin 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 diese nicht nur die Programmausführung *schneller* sondern auch den Code insgesamt *kleiner*!

*: Wieviel Platz im Code-Speicher verfügbar ist hängt natürlich ab von der Anzahl der Satelitten im Gesamtprogramm ab.

Nicht-Inline Member-Funktionen

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

- 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 kommt modernen CPU-Architekturen zu, welche die Tatsache, dass durch die Verzweigung die rein sequentielle Ausführung unterbrochen wird.

Dies hat zur Folge, dass bei Eintritt in das Unterprogramm und bei dessen Verlassen die Pipeline mit bereits teilweise dekodierten Maschinenbefehlen dort gespeichert worden sein müsste.

Nachteile von Inline-Funktionen

Weiterhin ist bei der Verwendung von Inline-Funktionen folgendes zu bedenken:

- Die Implementierung muss im Header-Files erfolgen.
- In der Kombination mit `virtual` werden `inline` Funktionen oftmals dennoch als echtes Unterprogramm umgesetzt.

> Anders gesagt: bei der Kombination von `inline` und `virtual` bleibt ersteres u.U. wirkungslos – detaillierte Erläuterungen folgen später.

Die Implementierung im Header-File erhöht jedoch den Grad der Kopplung durch Compilezeit-Abhängigkeiten, womit Änderungen an einer `inline` Funktion oft umfangreiche Re-Kompilierungen auslösen.*

Der Verzicht auf `virtual` – damit `inline` auf jeden Fall Wirkung zeigt – beschränkt abgeleitete Klassen darin, von ihren Basisklassen geerbte Member-Funktionen zu überschreiben, um deren Verhalten anzupassen.

---

*: Im Grunde genommen geschieht durch `inline` für Member-Funktionen das Gegenteil von dem, was für Member-Daten durch Anwendung des PIMPL-Idiom erreicht werden soll.

Öffentliche Basisklassen

Gemäß der objektorientierten Sichtweise entspricht dies der *Vererbung*.

Aus Sicht auf die Member-Daten handelt es sich zunächst nur um eine Verschachtelung von Strukturen:

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

Zusätzlich gilt das LSP (Liskov Substitution Principle), gemäß dem ein ggf. erforderlicher Up-Cast (`Derived` → `Base`) vom Compiler automatisch erfolgt.

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

[*]: Bei Mehrfachvererbung gilt diese Aussage nur noch eingeschränkt.

Private Basisklassen

Gemäß der objektorientierten Sichtweise entspricht dies der Komposition.

So gesehen sind private Basisklassen in C++ zunächst lediglich eine Alternative zur Realisierung der Komposition über Member-Daten.

- Zusätzlich hat bei einer privaten Basisklassen aber die abgeleitete Klasse die Möglichkeit, geerbte Methoden zu überschreiben.

- Daher kann das GoF in C++ auch mit geringerer Kopplung* zwischen Basisklasse und abgeleiteten Klassen umgesetzt werden.

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

*: Stronger or looser coupling between too classes defines how much the one depends on the other – be in terms of documented behaviour (the good part) or accidental implementation details (what should be avoided in theory but happens in practice). In case of private base classes the coupling only exists through the public interface **plus** `virtual` members and hence is lesser as in case of public base classes, where coupling – at least potentially – may also be via the protected interface.

Komposition über Member-Daten

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

- Durch die Anordnung der Daten-Member im Speicher, welche zu steigenden Adressen hin der Reihenfolge im Quelltext von oben nach unten* 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 Basisklasse
  - bei Eindeutigkeit keine weitere Qualifikation notwendig ist, und
  - ansonsten der Klassenname per Scope-Operator (`Base::`) vorangestellt wird.

*: It is assumed here that there is only one data member defined per line. If several data members are defined in the same line, those more to the left get placed at lower addresses in memory (through lower offsets from the begin of the class' data space.)

Vergleich: Komposition über Member-Daten

Zur Verdeutlichung des Unterschieds der beiden Formen von Komposition hier zunächst der Weg über Member-Daten als Code-Fragment:

```
class Motor {
    …
public:
    void start_engine();
    …
};

class Car {
    Motor m;
    …
    void start_engine() {
        … // prepare starting the motor
        m.start_engine();
        … // check for engine errors
    };
}
```

Vergleich: Komposition über private Basisklasse

Und hier das Code-Fragment mit einer privaten Basisklasse zur Komposition:

```
class Motor {
    …
protected:
    void start_engine();
    …
};

class Car : private Motor {
    …
public:
    void start_engine() {
        … // prepare starting the motor
        Motor::start_engine();
        … // check for engine errors
    };
}
```

Erweiterungspunkte in Basisklassen

Nur mit Hilfe einer privaten Basisklasse läßt sich die folgende Form des Zusammenspiels von Auto und Motor erreichen, bei welcher der Motor für die Benutzung durch das Auto* bereits *Erweiterungspunkte* vorgibt.

```
class Motor {
    …
    // extension points for aggreagate class
    virtual void prepare_engine_start() {}
    virtual void check_engine_errors() {}
   …
public:
    void start_engine() {
        prepare_engine_start();
        … // actually start the engine
        check_engine_errors();
    }
    …
};
```

*: Oder wo immer der Motor später einmal als Komponente eingebaut wird …

Nutzung der Erweiterungspunkte

Das Auto nutzt nun lediglich die vom Motor zur Verfügung gestellten Erweiterungspunkte, **überschreibt aber nicht die Member-Funktion** zum Starten des eingebauten Motors.*

```
class Car : private Motor {
    …
    // implement extension points ...
    virtual void prepare_engine_start() {
        … // whatever is necessary
    }
    virtual void check_engine_errors() {
        … // whatever is necessary
    };
public:
    using Motor::start_engine;
};
```

*: In a more "cautious" design `Motor` could make its `start_engine()` member function protected (or even private!) and leave its exposition up to `Car`, typically then with a public `using Motor::start_engine` directive.

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) {}
    };
    …
};
```

"Freundschaft" eng gekoppelter Klassen

Bei C++-Entwicklern gibt es mitunter Vorbehalte in Bezug auf die Nutzung von `friend`-Beziehungen.

Der Grund ist vermutlich, dass ältere Fachliteratur* mitunter die völlig unberechtigte Ansicht verbeitete, damit würde der Zugriffsschutz geschwächt – jedoch ist das Gegenteil ist der Fall!

> **(i)** Sofern zwei Klassen **miteinander** deutlich enger zusammenarbeiten als "mit dem Rest der Welt", lässt sich über `friend`-Beziehungen eine deutlich bessere Kapselung im Sinne des Zugriffsschutzes erzielen.

- Durch ausschließlich nicht-öffentliche Member in `Lifo::Node` kann auf den `next`-Zeiger jetzt nur noch die `Lifo`-Klasse zugreifen.
- Ohne `friend`-Beziehung müssten `Lifo::Node`-s eine Member-Funktion bereitstellen, mit der **jeder(!)** den `next`-Zeiger verändern kann.

---

*: Auch in C++-Fachveröffentlichungen gab und gibt es gelegentlich die Tendenz, einfach voneinander abzuschreiben, ohne die Sachlage selbst zu überdenken ... Explizit **nicht** zur Gruppe solcher Autoren gehört Jiri Sukoup, der in seinem Buch Taming C++ bereits 1994 den Nutzen von `friend`-Beziehungen ausführlich darstellte, obwohl er sich damit in Gegenposition zu anderen Autoren begab.

Container-Elemente mit Daten

Um in den Elementen des Containers auch Daten unterzubringen, müssen zunächst entsprechende Klassen abgeleitet werden, z.B. für Gleitpunktzahlen ...

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

... oder Zeichenketten:

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

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

Daten-Elemente als Templates

Das immer wieder nahezu gleiche Quelltext der abgeleiteten Datenklassen legt die Verwendung von Templates nahe:*

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

> Dies hat nichts mit der auf Templates basierenden Implementierung von Containern zu tun, die später gezeigt wird.

An dieser Stelle geht es einzig und allein darum, den sehr ähnlichen Quelltext für die Klassen der Daten-Elemente im Sinne des DRY-Principles zusammenzufassen!

---

*: Das Argument des Konstruktors wird nun auf jedem Fall per Referenz übergeben, was im Allgemeinen zu einem leichten Overhead für Grundtypen geringer Größe führen dürfte. Im obigen Fall handelt es sich jedoch um eine Inline-Funktion (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!"));
```

Template-Helfer-Funktionen

Die redundante Angabe des Typs als Template-Argument im letzten Code-Fragment legt die Definition einer Helferfunktion nahe:

```
template<typename ElemType>
Data_Node<ElemType> *make_Data_Node(const ElemType &v) {
    return new Data_Node<ElemType>(v);
}
…
c.push(make_Data_Node(47,11));
c.push(make_Data_Node(string("hello, world")));
```

Zur bequemen Übergabe von Zeichenketten-Literalen als `std::string` sollte kann man noch eine spezifische Überladung hinzufügen:*

```
Data_Node<std::string> *make_Data_Node(const char *v) {
    return new Data_Node<std::string>(v);
}
…
c.push(make_Data_Node("hello, world"));
```

*: Here we already have taken a few steps beyond the border of "programming with templates" land – for the moment we will retreat, but this path will be followed in a later chapter.

Datenelemente entnehmen

Hier zeigt sich nun ein gravierender Nachteil, denn das grundsätzlich erforderliche 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.
- Insofern kann bei der Entnahme mit der Member-Funktion `Lifo::pop` auch nur ein solcher, allgemeiner Elementzeiger zurückgegeben werden.
- Für diesen muss zur weiteren Verwendung ein expliziter Down-Cast erfolgen.*

*: Dies ist alles, was Java Generics automatisieren, und insofern sollten diese – trotz ähnlicher Syntax – nicht mit C++ Templates verglichen werden: es liegen Welten dazwischen!

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 ??
}
```

Bereits C++98 erlaubt übrigens auch die etwas kompaktere Form*

```
if (Double_Node *p = dynamic_cast<Double_Node*>(c.pop())) …
```

womit sich die Sichtbarkeit von `p` auf die `if`-Anweisung beschränkt.

*: In C++11 weiter verkürzbar zu: `if (auto p = dynamic_cast<Double_Node*>(c.pop())) …`

Datenelemente verwendbar machen mit dynamic_cast auf Referenzbasis

Wenn im Fehlerfall ohnehin ein Abbruch erfolgen müsste, da die falsche Datenart völlig unerwartet vorgefunden wurde und nicht weiterverarbeitet werden kann, 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
```

Diese Form lässt sich auch mit dem direkten Member-Zugriff verbinden:

```
… dynamic_cast<Double_Node&>(*c.pop()).data …
```

Mehr zu `dynamic_cast` und ein Vergleich zum weniger sicheren `static_cast` folgt als Bestandteil des Abschnitts RTTI.

Memory Leaks

Da in C++ keine Garbage Collection existiert, sind Memory Leaks immer wieder ein Thema.

> **[!]** In allen vorhergehenden Beispielen, welche Elemente aus dem `Lifo` entnehmen, wurde das Thema Speicherverwaltung bislang vernachlässigt – es wird dafür auf den folgenden Seiten im Zentrum stehen.

Am besten ist es, wenn das Thema Speicherverwaltung im frühzeitigen Programm-Design Berücksichtigung findet.

- Probleme, die ihre Wurzeln in einem (für C++) unangemessenen Programm-Design haben, nachträglich "per Debugging" anzugehen, kann schnell uferlos werden.
- Zwar gibt es eine Reihe von Tools, welche beim Aufspüren solcher Probleme helfen, sie demonstrieren aber nur die "Symptome" und liefern nicht die "Arznei" zu deren Vermeidung.

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 Tests auf andere mögliche Elementtypen, und dann ganz am Ende (außerhalb aller bedingten Blöcke):

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

[!] Allerdings ist Code wie der obige nur dann korrekt, wenn der Destruktor `Lifo::Node::~Node()` virtuell ist.

Auf Templates basierende Technik

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

```
template<class ElemType>
class Lifo {
    class Node;
    Lifo *top;
public:
    Lifo() : top(nullptr) {}
    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 im Typ paremetriesiert enthalten:

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

Operation zum Einfügen neuer Datenelemente

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

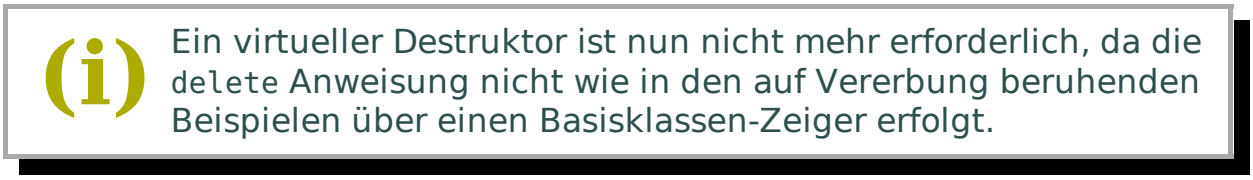
```
template<class ElemType>
void Lifo<ElemType>::push(const ElemType &d) {
    head = new Node(head, d);
}
```

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

Operation zur Entnahme von Datenelemente

Somit erscheint es naheliegend, auch die Freigabe in der `Lifo`-Klasse zu kapseln:

```
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;
}
```



Destruktor der Container-Klasse

Aufgrund des Übergangs der Verantwortlichkeit für die Verwaltung des dynamischen Speichers der Datenelemente sollte spätestens jetzt auch die `Lifo`-Klasse einen Destruktor erhalten:

```
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 (auf Vererbung basierende) Implementierung sinnvoll wäre oder gar notwendig ist, hängt von einer eingehenden Analyse des Codes ab, welcher den Container tatsächlich benutzt.*

*: 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 von noch benutzten Datenelementen – wenn es einen solchen Destruktor gibt aber eigentlich *keinen* geben sollte).

Typsicherheit beim Einfügen

Neben der sicheren Speicherverwaltung macht der Blick auf die Implementierung 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 damit 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 Polymorphismus hinsichtlich der Datenelemente benötigt, muss explizit ein Container von Zeigern verwendet werden:

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

[!] Allerdings geht damit die Verantwortung für den Speicherplatz wieder auf denjenigen über, der den Container benutzt.

Das Einfügen muss nun so erfolgen:

```
allMyFruits.push(new Banana( … ));
allMyFruits.push(new Apple( … ));
```

Entnahme von Basisklassen-Zeigern

Die Entnahme sieht prinzipiell so aus:

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

> **[!]** Für die Korrektheit des obigen Codes muss der Destrutktor `Fruit::~Fruit()` virtuell sein!

C++ Style-Guides legen mitunter nahe, sicherheitshalber

- **alle** Klassen mit einem virtuellen Destruktor auszustatten,
- oder zumindest Klassen, die als Basisklassen verwendet werden,
- oder zumindest Klassen, die wenigstens eine (andere) Member Funktion als `virtual` definieren.*

*: But all these rules miss the key question, which could have been posed already at design time: What knowledge about the class hierarchy is available where the `delete` takes place?

Ermittlung der abgeleiteten Klasse

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, it's a banana
    … // peal it, eat it
}
```

Während die gezeigte Vorgehensweise funktioniert, kann sie unter dem Aspekt eines guten objekt-orientierten Designs kritisiert werden.

Eine Alternative zur obigen, typ-basierten Mehrfachverzweigungen wird später noch ausführlicher betrachtet.

Pointer in auf Templates basierenden Containern

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 "Obstkorb" besteht z.B. ein Dilemma:

- Sollte er sich als Eigentümer des Speicherplatzes sehen, auf den die enthaltenen Zeiger verweisen,
- oder sollte er das eher nicht tun?*

ResourceWrapper-Klassen können hier helfen, im konkreten Fall:

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

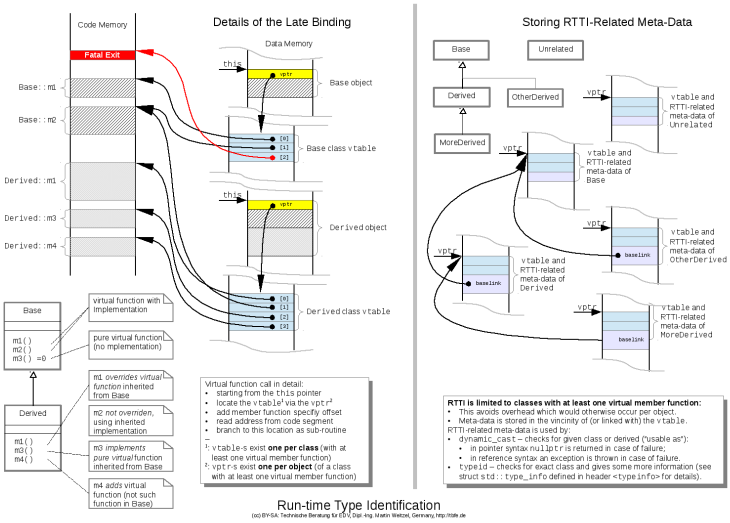
Diese werden in einem späteren Kapitel ausführlicher behandelt.

*: Do not take this lightly ... an inaproriate decision may lead to a memory leak, or even worse, cause dangling pointers.

Typ-Bestimmung zur Laufzeit

- Umsetzung von Dynamischem Polymorphismus

- Typ-Identifikation zur Laufzeit



RTTI-Voraussetzungen

Mit C++98 wurde die explizite Laufzeit-Typinformation (RTTI) zum verbindlichen 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++, da die Ausführung virtueller Member-Funktionen vom Laufzeityp abhängt. Insofern ist es auch naheliegend, RTTI an die Existenz virtueller Methode zu knüpfen.

Dynamischer Polymorphismus

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

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

Voraussetzung dafür ist:

- Der Aufruf erfolgt über eine Objekt-Referenz oder einen Objekt-Zeiger. Diese(r) legt den Compilezeit-Typ fest, gemäß dem entschieden wird, **welche** Member Funktionen überhaupt 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

Generell geht es bei der Ermittlung des Laufzeittyps eines Objekts darum, auf dieser Basis zu entscheiden, welche von ggf. mehreren überschriebenen virtuellen Member-Funktion aufzurufen ist.*

```
class Base { … virtual void foo(); … };
class Derived : public Base { … }; // this does not override foo(),
                                   // but the following two do:
class MoreDerived : public Derived { … virtual void foo(); … };
class OtherDerived : public Base { … virtual void foo(); … };
…
void bar(Base &b) { // or: bar(Base *p)
    b.foo();        //     p->foo();
}
…
// now for some calls of foo through bar:
Base b;         bar(b); // calls Base::foo()
Derived d;      bar(d); // same
MoreDerived m;  bar(m); // calls MoreDerived::foo()
OtherDerived o; bar(o); // calls OtherDerived::foo()
```

*: Another interesting point is this: to compile `bar` nothing but `Base` needs to be known, in fact the derived classes might not even have been written then, but the code manages somehow to always call the right overrides ...

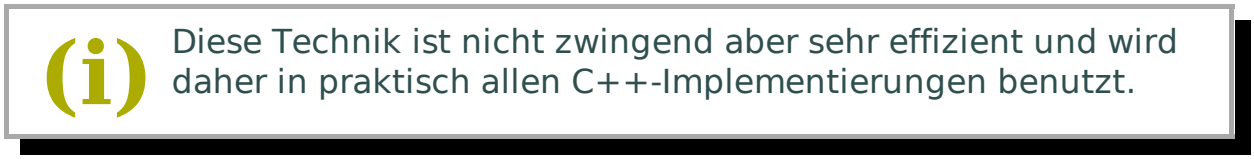
Implementierung des dynamischen Polymorphismus

Die übliche Vorgehensweise beruht auf dem indirekten Einsprung in ein Unterprogramm über eine Tabelle von Zeigern.

Hierzu führt jedes Objekt, das einer Klasse mit **mindestens** einer virtuellen Member-Funktion angehört,

- **genau** einen zusätzlichen Zeiger mit,
- welcher wiederum auf eine Tabelle mit den Adressen **aller** ihrer virtuellen Member-Funktionen zeigt.

Da diese Tabelle **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 *einmal pro Objekt-Modul* angelegte Exemplare dieser Tabelle bis auf eines eliminiert.

Kombination von `virtual` und `inline`

Die früher getroffene Feststellung, das `virtual` und `inline` einander quasi ausschließen, soll nun etwas präzisiert werden:

- Rein technisch kann beides durchaus zusammentreffen, und zwar
  - sowohl explizit – wenn beide Schlüsselworte verwendet werden,
  - wie auch implizit, wenn einer als `virtual` gekennzeichneten Member Funktion direkt in der Klasse selbst implementiert wird.
- Dann hängen die Details der Umsetzung vom Aufruf ab:
  - Richtet sich dieser **direkt an ein Objekt**, wird der Implementierungs-Code an Ort und Stelle statt eines Unterprogrammaufrufs eingesetzt.
  - Geht er an ein **über Zeiger oder Referenz** angesprochenes Objekt, erfolgt die zuvor beschriebene, polymorphe Umsetzung.

Bei letzterer ist Unterprogrammsprung zwingend und daher wird eine Funktion, die `virtual` und `inline` ist, stets als potenziell polymorph betrachtet, d.h. es muss dafür ein echtes Unterprogramm verfügbar sein.

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 Laufzeit-Typ einer von `Base` abgeleitete Klasse `Derived` ist – oder noch tiefer abgeleitet.

`dynamic_cast` auf Zeigerbasis

```
Base *p = nullptr;
… // p gets assgned an address and may point
  // to Base or any class derived from Base

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

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

```
Base &r = … ; // r gets initialized and may refer to
              // Base or any class derived from Base
Derived *dp = dynamic_cast<Derived*>(&b);
```

RTTI mit dynamischem Cast (2)

`dynamic_cast` auf Referenzbasis

```
Base &r = … ; // r gets initialized and may refer to
              // Base or any class derived from Base
…
Derived &dr = dynamic_cast<Derived&>(r);
```

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

Damit kann diese Form leicht Bestandteil eines Ausdrucks verwendet werden, insbesondere um eine member-Funktion aufzurufen, die von einer abgeleiteten Klasse hinzugefügt wurde:

```
… dynamic_cast<OtherDerived &>(r).phoo( … ) …
```

Die Referenzform ist auch `Base *p`-Zeiger verwendbar:

```
OtherDerived &dr = dynamic_cast<OtherDerived &>(*p);
… dynamic_cast<OtherDerived &>(*p).foo() …
… &dynamic_cast<OtherDerived &>(*p)->foo() …
```

Implementierung von explizitem RTTI

Hierfür gibt es keinen durchgängigen Standard – ein klein wenig lässt sich jedoch erahnen:

Egal ob im Rahmen von RTTI verwendet oder für eingebaute Typen und nicht-polymorphe Klassen:

- `typeid` liefert eine Referenz auf eine `class type_info` zurück.
- Jeweils ein Objekte dieser Klasse muss ein kompiliertes C++-Programm für **jede** darin verwendete Klasse vorhalten, für den Fall, dass es mit `typeid` angesprochen wird.*

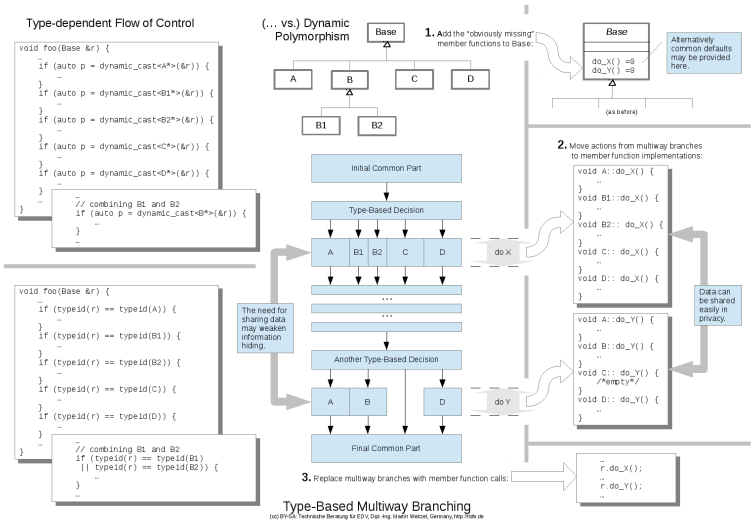
Um dynamischen Polymorphismus zu unterstützen, könnten solche Obkekte mit der Sprungtabelle für die virtuellen Member-Funktionen zusammengefasst werden, oder sie liegen an anderen Stelle und die Sprungtabelle verweist darauf.

---

*: Da dies statisch entscheidbar ist, kann spätestens der Linker überflüssige `type_info`-Objekte aus dem endgültigen, ausführbaren Programm herauslassen.

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.

Praktikum