

EmbSW2

L. Leuenberger, M. Ehrler, D. Büsser, M. Lehmann, R. Mattle

17. August 2016

Inhaltsverzeichnis

1	Listen	5
1.1	Speicherung mehrerer Elemente desselben Typs	5
1.2	Eigenschaften einer Liste	5
1.3	Einfach verkettete Liste - Single Linked List	5
1.3.1	Definition eines Listenelementes	5
1.3.2	Bezeichnung des Listenanfangs	5
1.3.3	Bezeichnung des Listenenendes	6
1.3.4	Listenelement einfügen	6
1.3.5	Listenelement löschen	6
1.3.6	Listenelement suchen	7
1.4	Doppelt verkettete Liste - Double Linked List	7
1.4.1	Definition eines Listenelementes	7
1.4.2	Bezeichnung des Listenanfangs	7
1.4.3	Bezeichnung des Listenenendes	8
1.4.4	Listenelement einfügen	8
1.4.5	Listenelement löschen	8
1.4.6	Listenelement suchen	9
2	Algorithmen	10
2.1	Beschreibungsmethoden	10
2.2	Formale Eigenschaften	10
2.2.1	Korrektheit	10
2.2.2	Finitheit	10
2.2.3	Bestimmtheit	10
2.2.4	Determinismus	10
2.2.5	Effizienz	10
2.2.6	Optionale formale Eigenschaften	10
2.3	Komplexitätstheorie	11
2.3.1	Hauptziele	11
2.3.2	Zu betrachtende Operationen	11
2.3.3	Asymptotische Abschätzung	11
2.3.4	Wachstumsfunktionen	12
2.3.5	Komplexitätsklassen	13
2.4	Suchalgorithmen	13
2.4.1	Listensuche	13
2.4.2	Mustersuche	15
2.5	Sortieralgorithmen	15
2.5.1	Interne vs. externe Sortierverfahren	15
2.5.2	Forderungen an interne Sortierverfahren	16
2.5.3	Einfache Sortierverfahren	16
2.5.4	Schnelle interne Sortierverfahren	17
2.5.5	Anzahl der Vergleiche (Compare) bei Sortieralgorithmen	19
2.5.6	Anzahl der Zuweisungen (Move) bei Sortieralgorithmen	19
2.6	Systemfunktionen für Suchen und Sortieren	19
2.6.1	In C	19
2.6.2	In C++	19
3	Simulation	20
3.1	Motivation für Simulationen	20

3.2	Vernünftige Eingabewerte	20
3.2.1	Reale Eingabewerte für Simulatoren	20
3.3	Zufallszahlen-Generator (Random Number Generator, RNG)	20
3.3.1	Zufallszahlen	20
3.3.2	Hardware True RNG (TRNG)	20
3.4	Pseudozufallszahlen-Generator	21
3.4.1	Pseudozufällige Zahlen	21
3.4.2	Forderungen	21
3.4.3	Methoden	21
3.4.4	rand() und srand()	21
3.4.5	Mapping von Zufallszahlen auf beliebige Bereiche	21
3.4.6	Gleichverteilung von ganzen Zahlen in [low, high]	22
3.5	Wahrscheinlichkeitsverteilungen	22
3.5.1	Normalverteilung	22
3.5.2	Rayleigh Verteilung	23
3.5.3	Weibull Verteilung	24
4	Verifikation und Validation	25
4.1	Unterschied zwischen Verifikation und Validation	25
4.1.1	Verifikation	25
4.1.2	Validation	25
4.2	Qualitätskosten	25
4.3	Lebenszyklus	25
4.4	Tests	25
4.4.1	Prüfverfahren	26
4.4.2	Testabschlusskriterien: wann ist das System good enough	26
4.4.3	Infinite faults	26
4.5	Testen von Embedded Systems	26
4.5.1	Reviews	26
5	CRC	28
5.1	Block Codes	28
5.2	Zyklische Codes	28
5.3	CRC Codes	28
5.4	Polynomschreibweise	28
5.4.1	Berechnung der Prüfsequenz R(x)	29
5.5	CRC in der Praxis	29
5.6	Implementation	30
5.6.1	Bit by Bit	30
5.6.2	Tabelle	30
6	POSIX	32
6.1	Process vs. Thread	32
6.2	Befehle	32
6.3	Condition Variable (Mutex)	32
6.3.1	Erstellen / Löschen	32
6.3.2	Warten	33
6.3.3	Signalisieren	33
6.3.4	Variablen	33
6.4	Bounded Buffer Problem (Producer/ Consumer)	33
6.5	Beispiel (Bounded Buffer)	33

7	Linux Development Tools	35
7.1	Entwicklertools	35
7.2	GNU Binutils	35
7.2.1	c++filt	35
7.2.2	gprof	35
7.2.3	nm	35
7.2.4	objdump	35
7.2.5	size	36
7.3	Codeüberdeckung oder -abdeckung (code coverage)	36
7.3.1	Code Coverage Tool: gcov	37
7.4	Performance Profiling - Valgrind	37
7.4.1	Memcheck	37
7.4.2	Callgrind	38
7.4.3	Valgrind-Output	38
8	Advanced C++	39
8.1	Static vs. Dynamic Binding	39
8.1.1	Virtuelle Memberfunktionen (Siehe Zf_OOProg)	39
8.2	RAII	40
8.3	pImpl	40
8.4	Performance	41
8.4.1	Variablen	41
8.4.2	Lächerliches	41
8.5	C++ vs. C	42
8.6	Inlining	42
8.7	Polymorphismus	43
8.8	Avoiding Code-Bloat / Templates	43
8.8.1	Allgemein	43
8.8.2	Templates	44
8.9	Interface Programming	45
8.10	Dynamic Memory Managment (DMM)	45
8.11	POD (Plain Old Data)	46
8.12	C++ and ROM	47
8.13	Schlüsselwörter und Speicher	47
8.14	Modeling Memory-Mapped I/O	47

1 Listen

1.1 Speicherung mehrerer Elemente desselben Typs

- Array: wenn die Anzahl Elemente statisch ist
→ `double statArray[20];`
- Dynamisch allozierter Array: wenn erst zur Laufzeit die Grösse festgelegt wird, dann aber fix bleibt
→ `double* dynArray = new double[varSize];`
- Liste: wenn die Anzahl Elemente erst zur Laufzeit bekannt ist und laufend ändert

1.2 Eigenschaften einer Liste

- Eine Liste ist eine sequentielle Anordnung von Listenelementen.
- Die Listenelemente, auch Knoten (Node) genannt, werden miteinander mittels Pointer verbunden.
- Listenanfang: Muss immer gegeben sein, da von diesem ausgegangen wird.
- Listenende: Muss immer gekennzeichnet werden. Wird durch Nullpointer definiert
- Besitzt die Grundoperationen: Element einfügen (insert), Element löschen (delete), Element suchen (search)

1.3 Einfach verkettete Liste - Single Linked List

- Eine einfach verkettete Liste verbindet die Listenelemente nur einfach, d.h. es gibt eine Pointerverbindung nur in eine Richtung.
- Rückwärts kann nicht navigiert werden, es geht nur von vorne nach hinten.
- Nachteil beim Löschen: da keine Rückwärtsnavigation möglich ist, muss mit einem zusätzlichen Hilfspointer gearbeitet werden.

1.3.1 Definition eines Listenelementes

Node

value	
next	●

```

1 | struct Node
2 | {
3 |     Item value; // der gespeicherte Wert, z.B. double
4 |     Node* next; // Pointer auf das naechste Listenelement
5 | };

```

1.3.2 Bezeichnung des Listenanfangs

pHead

	●
--	---

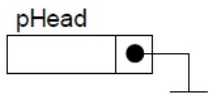
```

1 | Node* pHead; // Kopf der Liste

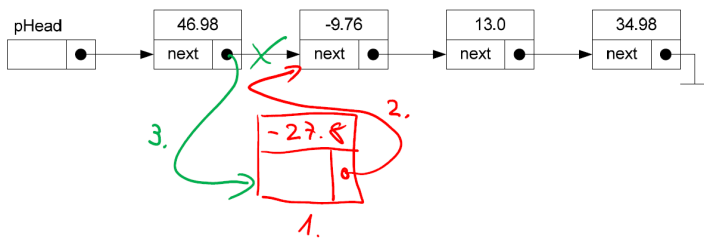
```

1.3.3 Bezeichnung des Listenendes

- Das Listenende wird mit einem Nullpointer gekennzeichnet.



1.3.4 Listenelement einfügen

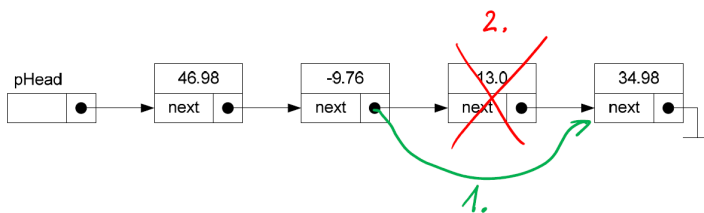


```

1 void SList::insertAt(int pos, double val)
2 {
3     assert(pos >= 0);
4     Node* pEl = new Node;
5     pEl->value = val;
6     if (pos != 0) // not at head
7     {
8         Node* p = nodePtr(pos);
9         assert(p != 0);
10        pEl->next = p->next;
11        p->next = pEl;
12    }
13    else // insert at head
14    {
15        pEl->next = pHead;
16        pHead = pEl;
17    }
18    ++nr;
19 }

```

1.3.5 Listenelement löschen



```

1 void SList::deleteAt(int pos)
2 {
3     assert(pos > 0 && pos <= nr);
4     Node* p = pHead; // cursor
5     Node* pDel = p; // node to be deleted
6     if (pos == 1) // first element
7         pHead = pHead->next;
8     else
9     {
10        for (int i = 1; i < pos-1; i++)

```

```

11 |     {
12 |         p = p->next;
13 |     }
14 |     pDel = p->next;
15 |     p->next = pDel->next;
16 | }
17 | delete pDel;
18 | --nr;
19 | }

```

1.3.6 Listenelement suchen

```

1 | int SList::search(double val) const
2 | {
3 |     Node* p = pHead; // p points to element at pos 1, if not empty
4 |     for (int i = 1; p; ++i)
5 |     {
6 |         if (p->value == val)
7 |             return i;
8 |         p = p->next;
9 |     }
10 |     return 0; // not found
11 | }

```

1.4 Doppelt verkettete Liste - Double Linked List

- Eine doppelt verkettete Liste verbindet die Listenelemente in beiden Richtungen.
- Dadurch kann einfach in der Liste navigiert werden, speziell rückwärts beim Löschen oder Einfügen.
- Der Verdrahtungsaufwand ist grösser als bei der einfach verketteten Liste.
- Jeder Knoten braucht einen Pointer mehr, d.h. der Speicherbedarf nimmt leicht zu.

1.4.1 Definition eines Listenelementes

Node

value	
next	●
●	prev

```

1 | struct Node
2 | {
3 |     Item value; // der gespeicherte Wert, z.B. double
4 |     Node* next; // Pointer auf das naechste Listenelement
5 |     Node* prev; // Pointer auf das vorheriges Listenelement
6 | };

```

1.4.2 Bezeichnung des Listenanfangs

pHead

	●
--	---

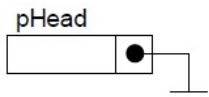
```

1 | Node* pHead; // Kopf der Liste

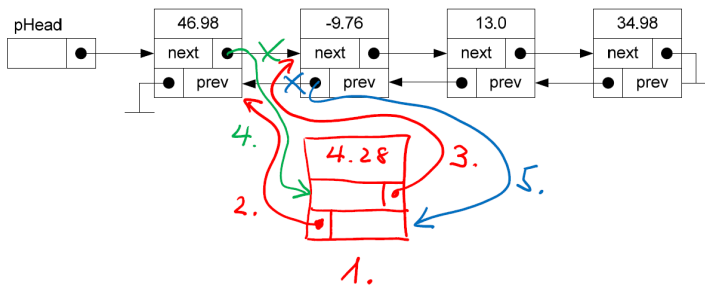
```

1.4.3 Bezeichnung des Listenendes

- Das Listenende wird mit einem Nullpointer gekennzeichnet.



1.4.4 Listenelement einfügen

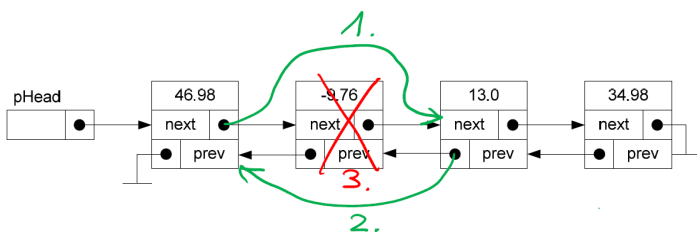


```

1 void DList::insertAt(int pos, double val)
2 {
3     assert(pos >= 0);
4     Node* pEl = new Node;
5     pEl->value = val;
6     if (pos != 0) // not at head
7     {
8         Node* p = nodePtr(pos);
9         assert(p != 0);
10        pEl->next = p->next;
11        pEl->prev = p;
12        p->next = pEl;
13    }
14    else // insert at head
15    {
16        pEl->next = pHead;
17        pEl->prev = 0;
18        pHead = pEl;
19    }
20    if (pEl->next != 0) // not last element in list
21        pEl->next->prev = pEl;
22    ++nr;
23 }

```

1.4.5 Listenelement löschen



```

1 void DList::deleteAt(int pos)
2 {
3     assert(pos > 0 && pos <= nr);
4     Node* pDel = nodePtr(pos); // node to be deleted

```

```
5 | assert (pDel != 0);
6 | if (pos == 1) // first element
7 |     pHead = pHead->next;
8 | else
9 | {
10 |     pDel->prev->next = pDel->next;
11 | }
12 | if (pDel->next != 0) // not last element in list
13 |     pDel->next->prev = pDel->prev;
14 | delete pDel;
15 | --nr;
16 | }
```

1.4.6 Listenelement suchen

```
1 | int DList::search(double val) const
2 | {
3 |     Node* p = pHead; // p points to element at pos 1, if not empty
4 |     for (int i = 1; p; ++i)
5 |     {
6 |         if (p->value == val)
7 |             return i;
8 |         p = p->next;
9 |     }
10 |     return 0; // not found
11 | }
```

2 Algorithmen

2.1 Beschreibungsmethoden

- Aktivitätsdiagramm → gute Variante!
- Konkrete Programmiersprache → gute Variante!
- Pseudocode
- Struktogramme → veraltet
- Programmablaufpläne → veraltet

2.2 Formale Eigenschaften

2.2.1 Korrektheit

- Die Korrektheit eines Algorithmus kann formal bewiesen werden. Dies ist jedoch häufig mit enormem Aufwand verbunden und wird in der Praxis nur selten durchgeführt → In der Praxis wird die Korrektheit mit Tests verifiziert.
- Nur **Anwesenheit von Fehlern** kann bewiesen werden, nicht deren Abwesenheit [E.Dijkstra]

2.2.2 Finitheit

- Der Algorithmus muss eindeutig mit einer endlichen Anzahl Schritten beschrieben werden können.

2.2.3 Bestimmtheit

- Der Algorithmus erzielt bei gleichen Vorbedingungen immer dasselbe Resultat (bei Algorithmen, die mit Zufallskomponenten arbeiten, trifft das nicht zu).

2.2.4 Determinismus

- Zu jedem Zeitpunkt ist der nächste Schritt des Algorithmus eindeutig bestimmt.

2.2.5 Effizienz

- Laufzeiteffizienz
- Speicherplatzeffizienz
- Verschiedene Betrachtungsweisen:
 - Best Case: Optimistischer Fall
 - Average Case: Durchschnittlicher Fall
 - Worst case (üblich): Pessimistischer Fall

2.2.6 Optionale formale Eigenschaften

- Einfachheit
- Verständlichkeit

2.3 Komplexitätstheorie

2.3.1 Hauptziele

- Berechnungskomplexität: Zeitkomplexität als Anzahl der Rechenoperationen sowie Speicherplatzkomplexität
- Spezifikation der Probleme und Methode zur Klassifikation in “praktisch lösbare” und “praktisch unlösbare” Probleme
- Vergleiche der Effizienz (Berechnungsstärke) verschiedener Algorithmen (deterministischer, nichtdeterministischer und zufallsgesteuerten)

2.3.2 Zu betrachtende Operationen

- Zuweisungen (Bewegung von Datensätzen)
- Vergleiche
- Rechenoperationen
- Unterprogrammaufrufe
- Verzweigungen

2.3.3 Asymptotische Abschätzung

Die asymptotische Abschätzung ist eine Worst-Case Analyse und kann in folgende Notationen aufgeteilt werden:

$\mathcal{O}$ -Notation (Häufigste Abschätzung)

Laufzeitabschätzung nach oben:

$$\mathcal{O}(g) = f : X \rightarrow X \mid \exists c_1 > 0 \wedge \exists c_2 > 0 \wedge \exists n_0 \in X \forall n \geq n_0 : f(n) \leq c_1 \cdot g(n) + c_2 \quad (1)$$

Finde eine Funktion $g(n)$, deren Aufwandsfunktion $f(n) \leq c_1 \cdot g(n) + c_2$, dann ist der Aufwand $\mathcal{O}(g)$.

Beispiel: Gegeben sei die Aufwandsfunktion $f(n) = 3n^2 + 6n + 7$. Die Funktion $f(n)$ ist ein Element der Menge $\mathcal{O}(n^2)$, da z.B. $f(n) \leq 9n^2 + 7 \quad \forall n \geq 0$.

Ω -Notation

Laufzeitabschätzung nach unten:

$$\Omega(g) = f : X \rightarrow X \mid \exists c > 0 \wedge \exists n_0 \in X \forall n \geq n_0 : f(n) \geq c \cdot g(n) \quad (2)$$

Finde eine Funktion $g(n)$, sodass die Aufwandsfunktion $f(n) \geq c \cdot g(n)$, dann ist der Aufwand $\Omega(g)$.

Θ -Notation (Theta-Notation)

$\mathcal{O}$ - und Ω Notation vereint:

$$\Theta(g) = \mathcal{O}(g) \cap \Omega(g) \quad (3)$$

Bemerkungen zur asymptotischen Abschätzung

In der Praxis wird bei der asymptotischen Abschätzung fast ausschliesslich die asymptotische Abschätzung nach oben, d.h. $\mathcal{O}()$ verwendet. Die asymptotische Abschätzung ist ein gutes Mass für den schlechtesten Fall bei sehr grossen n . **Konstante Faktoren werden nicht betrachtet**, obwohl sie alles andere als unerheblich sein können. Es ist durchaus möglich, dass ein bestimmter Algorithmus ausschliesslich mit kleinen n arbeitet und damit nie in ein für grosse n relevantes starkes Wachstum gerät. Es ist deshalb wichtig, dass der Anwender von asymptotischer Abschätzung genau weiss, welche Schlüsse er daraus ziehen kann und vor allem auch welche nicht.

2.3.4 Wachstumsfunktionen

Die häufigsten und wichtigsten Wachstumsfunktionen sind:

- **Konstantes Wachstum: 1**

Die Anweisungen eines Programms werden einmal oder höchstens mehrere Male durchlaufen, unabhängig von der Grösse des Problems. Die Laufzeit des Programms ist konstant. Dies ist der Idealfall des Laufzeitverhaltens.

- **Logarithmisches Wachstum: $\log(n)$**

Die Laufzeit des Algorithmus wird mit steigender Grösse n nur allmählich langsamer. Genauer: wenn n quadriert wird, wächst $\log n$ nur um einen konstanten Betrag. Beim Logarithmus ist die **Basis nicht relevant**, da sie als konstanter Faktor herausgezogen werden kann.

- **Lineares Wachstum: n**

Die Laufzeit des Algorithmus steigt mit steigender Grösse n linear an.

- **n -log- n Wachstum: $n \cdot \log(n)$**

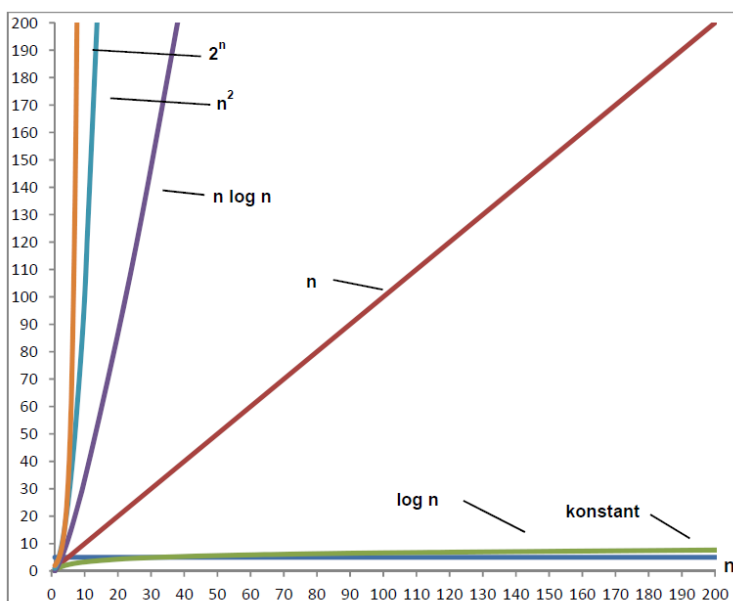
Dieses Wachstum tritt häufig bei Algorithmen auf, die ein Problem lösen, indem sie es in kleinere Teilprobleme aufteilen, diese unabhängig voneinander lösen und dann die Lösungen kombinieren. Das n -log- n -Wachstum ist nicht wesentlich grösser als lineares Wachstum.

- **Polynomiales Wachstum: n^c**

Algorithmen mit polynomialem Wachstum (quadratisch, kubisch, ...) sind eigentlich nur dann vertretbar, wenn einem nichts anderes übrigbleibt. Es ist das Wachstum, das in der Praxis noch als äusserstes generell akzeptiert wird (Probleme der Komplexitätsklasse P). Typischerweise entsteht z.B. dann quadratisches Wachstum, wenn paarweise alle Kombinationen von Datenelementen verarbeitet werden. Dies geschieht häufig mit zwei ineinander verschachtelten Schleifen beispielsweise in Matrixoperationen.

- **Exponentielles Wachstum: b^n**

Algorithmen mit exponentiellem Wachstum sind nur für sehr seltene Probleme in der Praxis verwendbar, da die Laufzeit mit steigendem n förmlich explodiert. Aktuell sind ein paar Tausend relevante Probleme der Komplexitätsklasse NP identifiziert, die auf heute vorhandenen Rechnern nur mit exponentieller Komplexität gelöst werden können, z.B. das Travelling Salesman Problem (TSP).



Log-Regeln

$$\log_{10}(a \cdot b) = \log_{10}(a) + \log_{10}(b)$$

$$\log_{10}\left(\frac{a}{b}\right) = \log_{10}(a) - \log_{10}(b)$$

$$\log_{10}(a^b) = b \cdot \log_{10}(a)$$

$$\log_{10} \sqrt[n]{u} = \frac{1}{n} \cdot \log_{10}(u)$$

$$\log_b(r) = \frac{\log_a(r)}{\log_a(b)}$$

2.3.5 Komplexitätsklassen

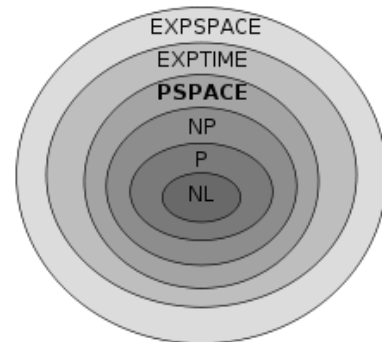
Probleme mit ähnlicher Komplexität werden in sogenannten Komplexitätsklassen zusammengefasst. Klassen werden als obere Schranke für den Bedarf einer bestimmten Ressource definiert, in Frage kommen dabei in der Praxis der Zeitbedarf (Zeitkomplexität) und der Speicherbedarf (Platzkomplexität). Auch hier handelt es sich um eine Worst-case-Betrachtung.

DTM (Deterministische Turingmaschine): Für jeden Zustand bei gegebenem Input ist der nächste Zustand eindeutig definiert. Ist praktisch realisierbar.

NTM (Nicht deterministische Turingmaschine): Für einen Zustand bei gegebenem Input ist der nächste Zustand **nicht** eindeutig definiert. Heute noch nicht realisierbar, dient jedoch als Algorithmusausführungsmodell.

Folgende Klassen sind definiert:

- **NL**: Von NTM auf logarithmischen Platz lösbar
- **P**: Von DTM in polynomialer Zeit lösbar, $O(n^c)$
- **NP**: Von NTM in polynomialer Zeit lösbar, $O(n^c)$
- **PSPACE**: Von DTM auf polynomialem Platz lösbar
- **EXPTIME**: Von DTM in exponentieller Zeit lösbar, $O(2^n)$
- **EXPSPACE**: Von DTM auf exponentiellem Platz lösbar



Probleme der Klasse P lassen sich in vernünftiger Zeit lösen.

Viele Probleme der Klasse NP lassen sich vermutlich nicht effizient lösen.

Bestätigung oder Widerlegung dieses P-NP-Problems ist wohl das wichtigste offene Problem der Informatik.

Bekanntes NP-Problem ist das Travelling Salesman Problem (TSP).

TSP Lösungsstrategien:

- **Brute force** alle Varianten durchprobieren
- **Branch and bound** Menge aller Varianten in Teilmengen aufteilen (Branch), Mit geeigneten Schranken (Bound) suboptimale Lösungen erkennen und aussortieren
- **Dynamic Programming** Held-Karp Algorithmus
- **Suboptimale Varianten** nicht die optimale Lösung wird angestrengt

2.4 Suchalgorithmen

2.4.1 Listensuche

Finde Wert in einer Liste.

Listendefinition

```

1 struct Field
2 {
3     Key id;           // z.B. int
4     Content cont;    // Typ, der den ganzen Inhalt umfasst
5 }
6
7 Field list[listLength];
8 Key k;               // der Wert, der gefunden werden soll
9 int i;               // Listenindex

```

Vereinfachte Listendefinition

```

1 | int list[listLength];
2 | int k;           // der Wert, der gefunden werden soll
3 | int i;           // Listenindex

```

Variante 1: Lineare Suche

Die einfachste Suchmethode ist, von vorne nach hinten Element um Element zu überprüfen bis das gesuchte Element gefunden oder das Listenende erreicht ist. Wenn nicht bekannt ist, ob die Liste irgendwie sortiert ist, ist das die einzige mögliche Suchmethode.

Komplexität: $O(n)$

```

1 | for (i = 0; i < listLength && k != list[i]; ++i)
2 | { // nothing to do
3 | }
4 | found = i < listLength;

```

Variante 2: Lineare Suche mit Sentinel

Die einzige Methode, den obigen Algorithmus zu beschleunigen, besteht in der Vereinfachung der Schleifenbedingung. Bei der linearen Suche mit Sentinel (Wächter) wird einfach am Ende der Liste ein Hilfselement eingefügt, das auf den Wert des gesuchten Elementes gesetzt wird. Der Algorithmus terminiert so auch ohne die Abfrage der Listenlänge, allerdings wird ein Platz in der Liste für den Sentinel geopfert.

Die Schleife wird etwas schneller durchlaufen, aber die Komplexität ist noch immer $O(n)$.

```

1 | list[listLength-1] = k;           // Sentinel setzen
2 | for (i=0; k != list[i]; ++i)
3 | {
4 | }
5 | found = i < listLength-1;

```

Variante 3: Binäre Suche

Diese Suche funktioniert nur, wenn die Liste der Reihe nach sortiert ist.

Man geht von der Mitte der Liste aus, ist der gegebene Wert grösser als die Mitte sucht man in der oberen Hälfte auf dieselbe Art weiter, ansonsten in der unteren Hälfte.

Liegt der Wert am Rande ist die Suche auf diese Art erfolglos.

Komplexität: $O(\log(n))$

```

1 | left = 0;                               // Index des am weitesten links liegenden Kandidaten
2 | right = listLength-1; // Index des am weitesten rechts liegenden Kandidaten
3 | while (left <= right)
4 | {
5 |     pos = (left + right) / 2; // pos auf die mittlere Position setzen
6 |     if (list[pos] == k)
7 |         return pos;
8 |     if (list[pos] > k)
9 |         right = pos - 1;
10 |    else
11 |        left = pos + 1;
12 | }
13 | return -1; // nicht gefunden

```

Beispiel "Suche 53":

5	7	13	18	21	30	33	35	45	53	61	65	69	85	91
5	7	13	18	21	30	33	35	45	53	61	65	69	85	91
5	7	13	18	21	30	33	35	45	53	61	65	69	85	91
5	7	13	18	21	30	33	35	45	53	61	65	69	85	91

2.4.2 Mustersuche

Bei der Mustersuche (string search) geht es darum, ein ganzes Stringmuster (Pattern) in einem vorgegebenen Text zu suchen. Wenn nur ein einzelnes Zeichen gesucht werden müsste, wäre dies identisch mit der Listensuche. Bei der Suche eines ganzen Patterns wird der Algorithmus aber bedeutend komplexer. Da die Mustersuche eine fundamentale Operation in jeder Textverarbeitung ist, besteht hier ein grosses Interesse, einen möglichst effizienten Algorithmus zu finden.

Variante Naiver Algorithmus

Der naive Algorithmus ist sehr einfach, es wird einfach Buchstabe um Buchstabe verglichen, bis eine Diskrepanz eintritt. Anschliessend wird das Pattern um eine Position weitergeschoben und wieder Buchstabe um Buchstabe verglichen. Wenn die Diskrepanz schon bei einem der ersten Buchstaben auftritt, ist die Performance des Algorithmus OK. Wenn jedoch viele Buchstaben übereinstimmen bis endlich eine Diskrepanz auftritt, dann ist die Performance sehr schlecht. Im schlechtesten Fall ist der Algorithmus $O(n \cdot m)$, wobei n die Länge des Textes und m die Länge des Patterns ist.

```

1 | int naivePSearch(const char* text, const char* pattern)
2 | {
3 |     int i;
4 |     int j;
5 |
6 |     for(i=0; text[i]; ++i)
7 |     {
8 |         for(j=0; pattern[j]; ++j)
9 |         {
10 |             if(pattern[j] != text[i+j])
11 |                 break;
12 |         }
13 |         if(pattern[j] == 0) // pattern found
14 |             return i;
15 |     }
16 |     return -1 // pattern not found
17 | }
```

Bei effizienteren Suchalgorithmen (z.B. KMP-Algorithmus oder BM-Algorithmus) wird bei einer Nichtübereinstimmung nicht nur um ein Zeichen sondern um eine möglichst hohe Anzahl Zeichen geschoben. Die Intelligenz liegt hier beim Herausfinden dieser Anzahl.

2.5 Sortialgorithmen

2.5.1 Interne vs. externe Sortiervverfahren

- Bei den **internen Sortiervverfahren** besteht auf sämtliche Daten ein wahlfreier Zugriff (random access). Die Daten liegen vollständig im Hauptspeicher vor. Die internen Sortiervverfahren werden auch als Verfahren zum Sortieren von Feldern (Arrays) bezeichnet.
- Bei den **externen Sortiervverfahren** kann nur sequentiell auf die Daten zugegriffen werden. Die Daten sind in Files auf langsameren externen Massenspeichern wie z.B. Magnetbändern oder Harddisks gespeichert. Die externen Sortiervverfahren werden auch als Verfahren zum Sortieren von Files bezeichnet.

2.5.2 Forderungen an interne Sortierverfahren

- **Speicher:** Sämtliche Daten liegen im Hauptspeicher. Nebst den vorhandenen Daten ist nur eine feste Anzahl von (Hilfs-) Variablen erlaubt (In-place Algorithmen).
- **Effizienz:** Nur Vergleiche (Comparison) und Zuweisungen (Move) werden betrachtet. (Bei den Zuweisungen werden nur die Zuweisungen der zu sortierenden Daten berücksichtigt. Zuweisungen von Indizes werden vernachlässigt, da sie in der Regel wesentlich schneller ablaufen, da sie sicher in Registern liegen. Auch die Vergleiche der Indizes werden vernachlässigt.)
- **Stabilität:** Ein Sortierverfahren ist dann stabil, wenn es die relative Reihenfolge gleicher Schlüssel beibehält. (Beispielsweise soll eine alphabetisch geordnete Liste von Wettkämpfern, die nach den Wettkampfnoten sortiert wird, die alphabetische Reihenfolge der Wettkämpfer bei gleichen Noten beibehalten.)

2.5.3 Einfache Sortierverfahren

- Gut verständlich
- Meist stabil
- Komplexität $O(n^2)$

Variante 1: Direktes Aussuchen (Straight selection)

Beim direkten Aussuchen wird der noch unsortierte Teil nach seinem kleinsten Element durchsucht. Dieses wird dann mit dem ersten Element im unsortierten Teil vertauscht.

Komplexität der Vergleiche: $O(n^2)$

Komplexität der Zuweisungen: $O(n)$

```

1 | int k; // Index des kleinsten Elementes
2 |      // im nicht sortierten Teil
3 |
4 | for (int i = 0; i < listLength-1; ++i)
5 | {
6 |     k = i;
7 |     for (int j = i+1; j < listLength; ++j)
8 |     {
9 |         if (data[j] < data[k])
10 |            k = j;
11 |     }
12 |     swap(i, k);
13 | }
```

8	23	16	18	6	45	12	34
6	23	16	18	8	45	12	34
6	8	16	18	23	45	12	34
6	8	12	18	23	45	16	34
6	8	12	16	23	45	18	34
6	8	12	16	18	45	23	34
6	8	12	16	23	23	45	34
6	8	12	16	23	23	34	45

Variante 2: Direktes Einfügen (Straight insertion)

Das erste Element des unsortierten Teiles wird im sortierten Teil direkt an der richtigen Stelle entsprechend seiner Grösse eingefügt. Alle grösseren bereits sortierten Elemente müssen verschoben werden.

Komplexität der Vergleiche: $O(n^2)$

Komplexität der Zuweisungen: $O(n^2)$

```

1 | int buf; // Buffer
2 |
3 | for (int i = 1; i < listLength; ++i)
4 | {
5 |     buf = data[i];
6 |     int j = i;
```

```

7 |     while (j > 0 && buf < data[j-1])
8 |     {
9 |         data[j] = data[j-1];
10 |        j = j-1;
11 |    }
12 |    data[j] = buf;
```

13 || }

8	23	16	18	6	45	12	34
8	23	16	18	6	45	12	34
8	16	23	18	6	45	12	34
8	16	18	23	6	45	12	34
6	8	16	18	23	45	12	34
6	8	16	18	23	45	12	34
6	8	12	16	18	23	45	34
6	8	12	16	18	23	34	45

Variante 3: Bubblesort

Jeweils benachbarte Elemente werden vertauscht, wenn sie nicht wie gewünscht geordnet sind. Bei jedem Durchlauf steigt dabei das relativ grösste Element wie eine Blase (bubble) im Wasser auf.

Komplexität der Vergleiche: $O(n^2)$

Komplexität der Zuweisungen: $O(n^2)$

```

1 for (int i = listLength-1; i > 0; --i)
2 {
3     for (int j = 1; j <= i; ++j)
4     {
5         if (data[j-1] > data[j])
6             swap(j-1, j);
7     }
8 }
```

8	23	16	18	6	45	12	34
8	16	18	6	23	12	34	45
8	16	6	18	12	23	34	45
8	6	16	12	18	23	34	45
6	8	12	16	18	23	34	45
6	8	12	16	18	23	34	45
6	8	12	16	18	23	34	45
6	8	12	16	18	23	34	45

2.5.4 Schnelle interne Sortierverfahren

- Schwerer verständlich als einfache Sortierverfahren
- Instabil
- Komplexität $O(n \cdot \log(n))$

Variante 1: Quicksort

Je grösser die Distanz beim Austauschen von Elementen ist, umso effizienter ist der Algorithmus. Dies nutzt Quicksort aus.

Formale Beschreibung:

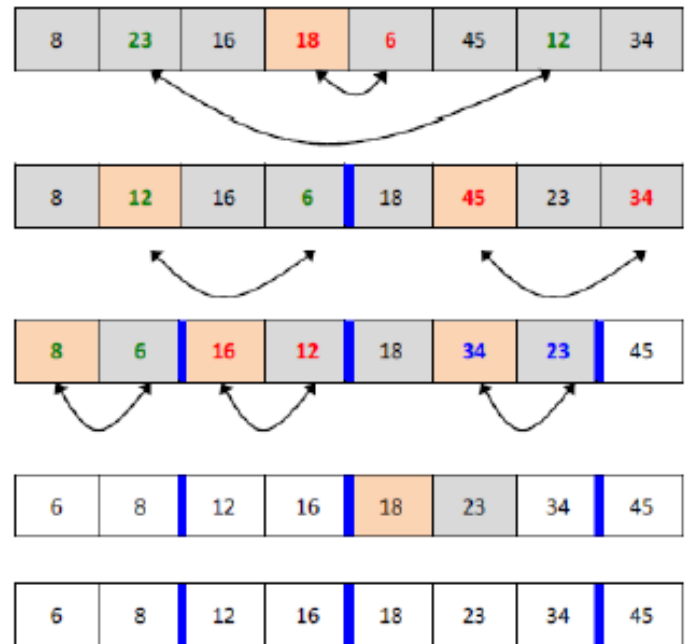
1. Wähle einen beliebigen Pivotwert aus dem Array aus.
2. Laufe von linker Arraygrenze so lange nach innen, bis ein **nicht kleineres** Element als der Pivotwert gefunden wird.
3. Laufe von rechter Arraygrenze so lange nach innen, bis ein **nicht grösseres** Element gefunden wird.
4. Vertausche die beiden Elemente.

5. Wiederhole Schritt 2-4 so lange, bis sich die beiden Indizes getroffen haben.
6. Das nun entstandene Teilfeld links besteht nur aus Elementen, die kleiner oder gleich dem Pivotwert sind, das Teilfeld rechts besteht nur aus Elementen, die grösser oder gleich dem Pivotwert sind.
7. Sortiere diese beiden Teilfelder wiederum (rekursiv), bis nur noch Teilfelder mit einem einzigen Element übrig bleiben.

```

1 void quickSort(int leftBound, int rightBound)
2 {
3     int left = leftBound;           //
4     int right = rightBound;         //
5     int pivot = data[(left+right) / 2]; //
6     do
7     {
8         while (data[left] < pivot)
9             ++left;
10        while (data[right] > pivot)
11            --right;
12
13        if (left <= right)
14        {
15            if (left < right)
16                swap(left, right);
17            ++left;
18            --right;
19        }
20    } while (left <= right);
21
22    if (leftBound < right)
23        quickSort(leftBound, right);
24    if (rightBound > left)
25        quickSort(left, rightBound);
26 }

```



Bester Fall: Wenn als Pivot jedes mal der Median des Feldes getroffen wird $\rightarrow O(n \cdot \log(n))$

Schlechtester Fall: Wenn jedes Mal das Minimum oder Maximum getroffen wird $\rightarrow O(n^2)$

Median: Der Median einer Anzahl von Werten ist die Zahl, welche an der mittleren Stelle steht, wenn man die Werte nach Größe sortiert. Bsp: 4, 1, 37, 2, 1 Median = 2

Eine vernünftige Wahl des Pivots ist also wichtig. Ein Ansatz ist die median-of-3 Strategie, bei welcher der Median aus dem ersten, letzten und mittleren Element eines Bereiches gebildet und als Pivot verwendet wird. Bei böseartig zusammengestellten Listen (median-of-3-killers) kann dies jedoch zu Problemen führen.

Variante 2: Introsort

- Ist in jedem Fall $O(n \cdot \log(n))$
- Setzt bei den Schwächen von Quicksort an $\rightarrow$ startet mit Quicksort und wechselt automatisch auf Heapsort (ein weiterer Algorithmus), wenn die Rekursionstiefe einen definierten Wert überschreitet.

2.5.5 Anzahl der Vergleiche (Compare) bei Sortialgorithmen

Algorithmus	C_{min}	C_{ave}	C_{max}	$O_C(\cdot)$
Straight Selection	$\frac{n}{2} \cdot (n - 1)$	$\frac{n}{2} \cdot (n - 1)$	$\frac{n}{2} \cdot (n - 1)$	n^2
Straight Insertion	$n - 1$	$\frac{1}{4} \cdot (n^2 + 3n - 4)$	$\frac{1}{2} \cdot (n^2 + n - 2)$	n^2
Bubblesort	$\frac{n}{2} \cdot (n - 1)$	$\frac{n}{2} \cdot (n - 1)$	$\frac{n}{2} \cdot (n - 1)$	n^2
Quicksort	$n \cdot \log_2(n)$	$2 \cdot n \cdot \ln(n)$	$\frac{1}{2} \cdot (n + 3)(n + 2) - 10$	n^2

2.5.6 Anzahl der Zuweisungen (Move) bei Sortialgorithmen

Algorithmus	M_{min}	M_{ave}	M_{max}	$O_M(\cdot)$
Straight Selection	$3 \cdot (n - 1)$	$3 \cdot (n - 1)$	$3 \cdot (n - 1)$	n
Straight Insertion	$3 \cdot (n - 1)$	$\frac{1}{4} \cdot (n^2 + 11n - 12)$	$\frac{1}{2} \cdot (n^2 + 5n - 6)$	n^2
Bubblesort	0	$\frac{3}{4} \cdot n \cdot (n - 1)$	$\frac{3}{2} \cdot n \cdot (n - 1)$	n^2
Quicksort	0	$n \cdot \ln(n)$	$\frac{3}{4} \cdot n \cdot \log_2(n)$	$n \cdot \log_{10}(n)$

2.6 Systemfunktionen für Suchen und Sortieren

2.6.1 In C

Funktion	Bibliothek	Algorithmus	Bemerkungen
bsearch()	<stdlib.h>	Binäre Suche in Arrays	
qsort()	<stdlib.h>	Quicksort	instabil

2.6.2 In C++

Funktion	Bibliothek	Algorithmus	Bemerkungen
binary_search()	<algorithm>	Binäre Suche	
sort()	<algorithm>	Introsort	instabil
stable_sort()	<algorithm>	Mergesort	stabil

3 Simulation

3.1 Motivation für Simulationen

- Das Verhalten kann mittels Simulationen analysiert werden
- Mit Simulationen können Zeiten verkürzt werden
- Durch Simulationen kann die Reaktion der Steuerung auf Ausnahmesituationen überprüft werden

3.2 Vernünftige Eingabewerte

Ablauf einer Simulation: Einem Simulator werden vernünftige Eingabewerte gegeben und seine Reaktion darauf überprüft.

3.2.1 Reale Eingabewerte für Simulatoren

- Reale, “vernünftige“ Eingabewerte entsprechen möglichst gut der Realität
- Häufig sollen bei Simulationen zufällige Eingabewerte verwendet werden, die nicht notwendigerweise gleichverteilt sind. Häufig vorkommende Verteilungen sind:
 - Gleichverteilung
 - Normalverteilung (Gauss’sche Glockenkurve)
 - Rayleigh-Verteilung
 - Weibull-Verteilung
- Die Simulationen sollen reproduzierbar sein.

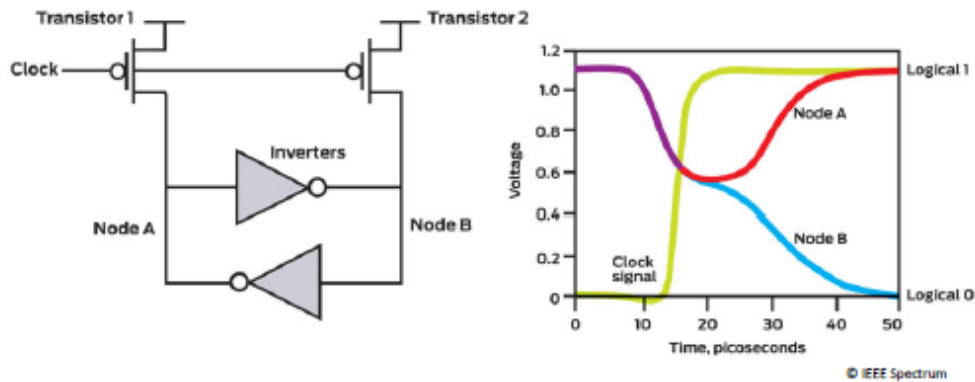
3.3 Zufallszahlen-Generator (Random Number Generator, RNG)

3.3.1 Zufallszahlen

- Echt zufällige Zahlen können nur durch einen physikalischen Prozess erzeugt werden, z.B. durch Würfel oder Thermisches Rauschen.
- Eine deterministische Maschine wie der Computer kann keine echt zufälligen Zahlen generieren (ausser mit zusätzlicher Hardware → siehe TRNG).
- Ein Computer kann immerhin pseudozufällige Zahlen liefern.

3.3.2 Hardware True RNG (TRNG)

- Clock = LOW: Transistoren leiten und zwingen Node A und B auf Vcc
- Clock = rising edge: Transistoren sperren
- Node A und B pendeln sich auf eine mittlere Spannung ein, bevor sie aufgrund von thermischem Rauschen zufällig auf 0 oder 1 fallen (jeweils invertiert)



3.4 Pseudozufallszahlen-Generator

3.4.1 Pseudozufällige Zahlen

- Pseudozufällige Zahlen sind deterministisch. Bei einem gegebenen Anfangswert (random seed) und gegebener Zufallszahlenfunktion ist die zufällige Zahlenfolge immer identisch. → hat den Vorteil, dass eine Simulation reproduzierbar wird.
- Der Nachteil ist, dass keine wirklich zufälligen Zahlen entstehen. Durch eine intelligent gewählte Funktion kann jedoch eine sehr hohe Periode der Folge erreicht werden, auch das Erraten der nächsten Zahl ist ohne Kenntnis des Random Seeds und der Funktion beinahe unmöglich.

3.4.2 Forderungen

- Gleichverteilung
- Nächste Zahl nicht vorhersehbar

3.4.3 Methoden

- Lineare Kongruenz: $r_{i+1} = (a \cdot r_i + c) \cdot \text{mod}(m)$
- Multiplikative Kongruenz ($c=0$): $r_{i+1} = (a \cdot r_i) \cdot \text{mod}(m)$

3.4.4 rand() und srand()

- `int rand(void);` → liefert eine ganzzahlige Zufallszahl im Bereich von `[0, RAND_MAX]`
- `void srand (unsigned int seed);` → wird zur Initialisierung des Zufallszahlengenerators benutzt

Beispiel:

```
1 #include <time.h>
2
3 srand (time(NULL));
4 printf ("Random number between 0 and 99: %d\n", rand()%100);
```

3.4.5 Mapping von Zufallszahlen auf beliebige Bereiche

`rand()` = Gleichverteilung

<code>rand()</code>	Liefert ganzzahlige Werte im Bereich $[0, \text{RAND_MAX}]$
<code>rand() % n</code>	Liefert Werte im Bereich $[0, n-1]$
<code>(double)rand()/RAND_MAX</code>	Liefert Werte im Bereich $[0, 1.0]$
<code>(double)rand()/RAND_MAX * (b-a) + a</code>	Liefert Werte im Bereich $[a, b]$

Bei einer Umwandlung in eine ganze Zahl muss beachtet werden, ob runden oder abschneiden die richtige Operation ist.

3.4.6 Gleichverteilung von ganzen Zahlen in $[low, high]$

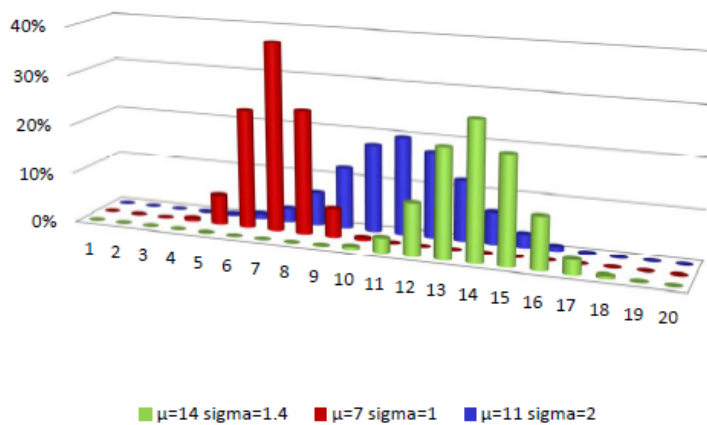
```
1 || static_cast<int>>(static_cast<double>>(rand()) / (RAND_MAX + 1.0) * (high-low + 1)) + low
```

- $(\text{RAND_MAX} + 1.0)$ ist notwendig, damit $0.0 \leq u < 1.0$ (ohne 1.0) erzeugt wird. $(\text{RAND_MAX} + 1)$ gäbe einen int-Overflow
- $(\text{high} - \text{low} + 1)$ ist notwendig, weil nach int konvertiert wird. Vor der int-Konversion gibt es dadurch bei $[1, 20]$ Zahlen im Bereich $[0.0, 20.0[$. Nach der Konversion werden nur die ganzzahligen Anteile verwendet.
- Das hinterste $+ \text{low}$ muss ausserhalb des Typecasts sein, andernfalls ergibt es eine Überhöhung bei 0, falls $\text{low} < 0$ (alle Zahlen in $]-1.0, +1.0[$ würden zu 0, d.h. doppelt so viele wie erlaubt)

3.5 Wahrscheinlichkeitsverteilungen

3.5.1 Normalverteilung

- Kommt sehr häufig in der Praxis vor (Natur, Ingenieurwesen, Wirtschaft, etc.)
 - Zufällige Messfehler um einen Sollwert
 - Fertigungsschwankungen um einen Sollwert
- Charakterisierung durch Mittelwert μ (häufigster Wert, Erwartungswert) und Standardabweichung σ (Mass für Streuung, Breite der Glockenkurve)
- Eigenschaften der Normalverteilung:
 - 68 % aller Werte liegen innerhalb von $\mu \pm \sigma$
 - 95 % aller Werte liegen innerhalb von $\mu \pm 2\sigma$
 - 99.7 % aller Werte liegen innerhalb von $\mu \pm 3\sigma$
 - 99.99 % aller Werte liegen innerhalb von $\mu \pm 4\sigma$



Zentraler Grenzwertsatz

Die Summe von n unabhängigen, identisch verteilten Zufallszahlen stellt für n gegen unendlich eine Normalverteilung dar.

Eine Normalverteilung kann somit angenähert werden, wenn eine grosse Anzahl gleichverteilter Zahlen summiert wird. Die einfachste Methode ist die **Polarmethode von Marsaglia** (basiert auf Box-Muller-Algorithmus). Sie ergibt gute normalverteilte Zahlen und ist recht einfach zu berechnen (braucht “nur“ einen Logarithmus und eine Wurzel).

```

1 do
2 {
3     v1 = 2 * static_cast<double> (rand()) / RAND_MAX - 1.0;
4     v2 = 2 * static_cast<double> (rand()) / RAND_MAX - 1.0;
5     s = v1 * v1 + v2 * v2;
6     } while (s >= 1.0 || s == 0.0);
7 double temp = sqrt(-2.0 * log(s) / s);
8 return lround(v1 * temp * sdev + mean);

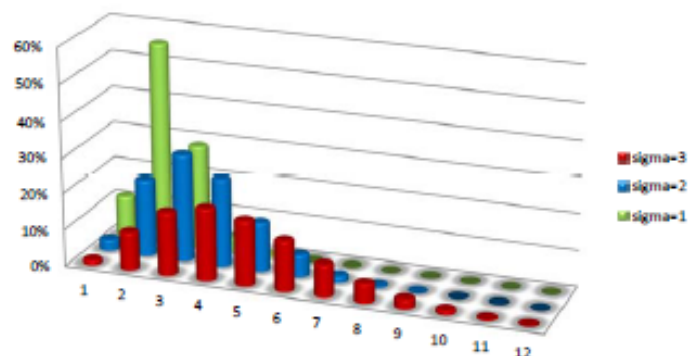
```

3.5.2 Rayleigh Verteilung

Eine Rayleigh-verteilte Zahl R kann aus zwei normalverteilten Zahlen N_1, N_2 mit Mittelwert 0 und Standardabweichung σ wie folgt erhalten werden:

$$R(\sigma) = \sqrt{N_1^2 + N_2^2}$$

$$R(\sigma) = \sigma \sqrt{-2 \ln(U)} \quad U \neq 0$$



Gemäss Formel braucht es für die Berechnung der Rayleigh-Verteilung zwei normalverteilte Zahlen.

```

1 do

```

```

2  {
3      v1 = 2 * static_cast<double> (rand()) / RAND_MAX - 1.0;
4      v2 = 2 * static_cast<double> (rand()) / RAND_MAX - 1.0;
5      s = v1 * v1 + v2 * v2;
6  } while (s >= 1.0 || s == 0.0);
7  double temp = sqrt(-2.0 * log(s) / s);
8  w1 = (v1 * temp * sdev);
9  w2 = (v2 * temp * sdev);
10 return lround(sqrt(w1 * w1 + w2 * w2));

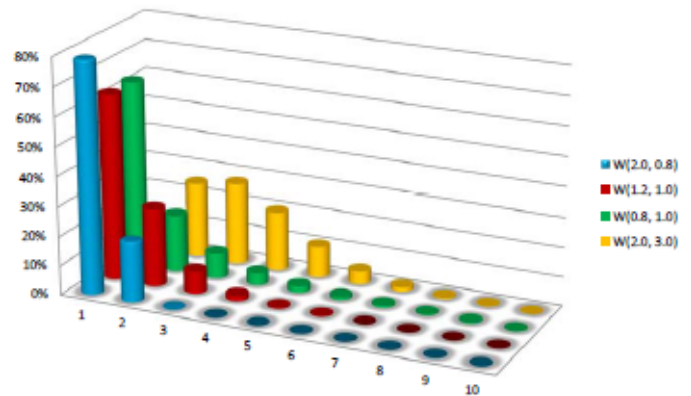
```

3.5.3 Weibull Verteilung

Die Dichtefunktion der Weibull-Verteilung kann mit 2 Parametern α , β wie folgt beschrieben werden:

$$W_{\alpha,\beta}(x) = \alpha \cdot \beta \cdot x^{\beta} \cdot e^{-\alpha \cdot x^{\beta}}$$

- $\beta = 1$: Exponentialverteilung
- $\beta = 3.4$: Normalverteilung
- $\beta > 1$: Alterndes System



Eine Weibull-verteilte Zahl W kann aus einer in $[0, 1.0[$ gleichverteilten Zahl U mit folgender Formel berechnet werden:

$$W(\alpha, \beta) = \beta \cdot (-\ln(1.0 - U))^{\frac{1}{\alpha}} \quad U \neq 1.0 \quad (4)$$

Anwendung der Weibull-Verteilung

- Untersuchung von Lebensdauern in der Qualitätssicherung
- Materialermüdungen von spröden Werkstoffen
- Ausfälle von elektronischen Bauteilen
- Verteilung von Windgeschwindigkeiten (oft wird hierzu auch die Rayleigh-Verteilung herangezogen)
- Allgemein: Systeme mit im Laufe der Zeit steigender Ausfallrate (Systeme mit konstanter Ausfallrate sind exponentialverteilt)

4 Verifikation und Validation

4.1 Unterschied zwischen Verifikation und Validation

Sicherstellen, dass das Softwaresystem die Benutzeranforderungen erfüllt.

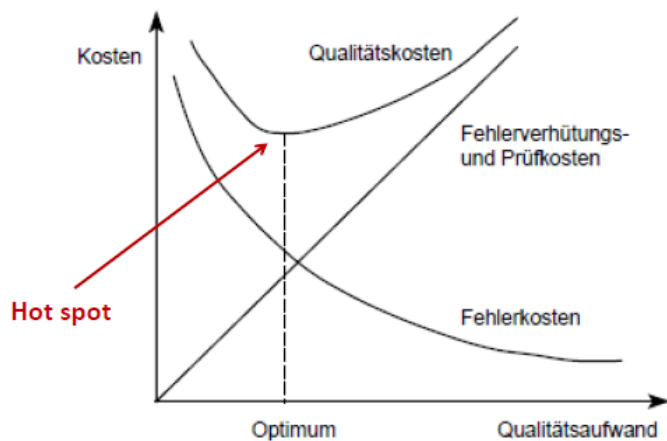
4.1.1 Verifikation

- Machen wir das Produkt richtig?
- Die Software soll den Anforderungen entsprechen.

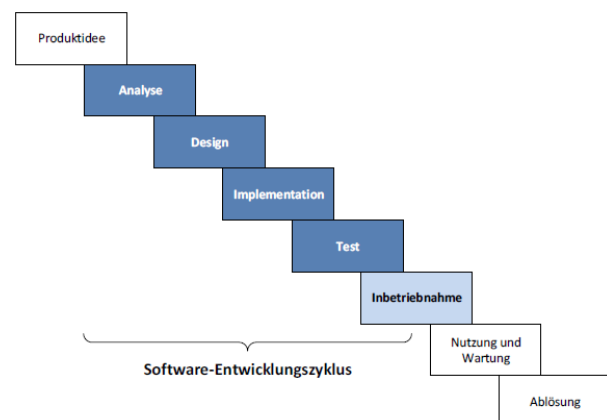
4.1.2 Validation

- Machen wir das richtige Produkt?
- Die Software soll machen, was der Benutzer wirklich benötigt.

4.2 Qualitätskosten

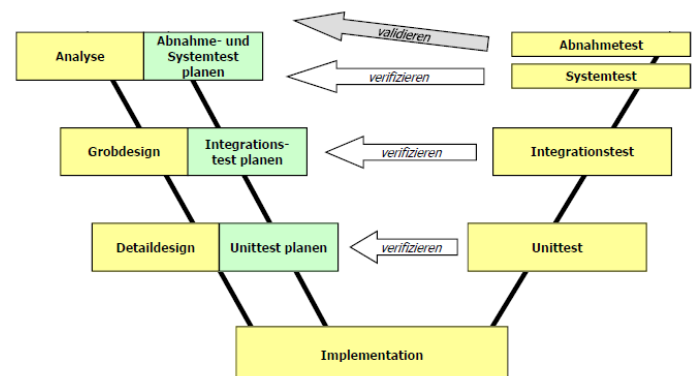


4.3 Lebenszyklus



4.4 Tests

- Unittest
 - Teil der Implementationsphase
 - Whitebox-Test
 - Durch Programmierer
- Integrationstest
 - Zusammenfügen mehrerer Units
 - Blackbox-Test
 - Schwerpunkt: Unit-Schnittstellen
 - Nicht durch Programmierer
- Systemtest
 - Testen des Gesamtsystems in Originalumgebung
 - Blackbox-Test
 - Schwerpunkt: Externes Verhalten des Systems
 - Nicht durch Programmierer
- Abnahmetest
 - Ziel der oberen Tests: Fehler finden
 - Ziel der Abnahmetest: Erfüllung von festgelegten Kriterien für geforderte Funktionen nachweisen
 - wird durch Auftraggeber durchgeführt



4.4.1 Prüfverfahren

- Review (viel zu selten eingesetzt)
- Statische Tests (ohne Ausführung des Programms)
- Dynamische Tests (durch Ausführung des Programms)
- Simulation
- Einsatz von Prototypen (kritische Teile werden vorab untersucht)

4.4.2 Testabschlusskriterien: wann ist das System good enough

- Wenn alle Testfälle der Testvorschrift erfolgreich durchgeführt worden sind
 - Häufiges Kriterium in der Praxis
 - Dieses Kriterium ist sinnvoll, wenn eine systematische Auswahl von Testfällen eine ausreichende Überdeckung ermöglicht
- Wenn die Prüfkosten pro entdecktem Fehler eine im voraus festgelegte Grenze überschreiten
 - Sinnvolles Kriterium für das Beenden des Systemtests
 - Setzt die Erfassung der Prüfkosten und der Anzahl gefundener Fehler voraus

4.4.3 Infinite faults

- Ein System hat unendlich viele Fehler (infinite faults) wenn trotz Testen und Debugging die Fehlerzahl um einen unakzeptabel hohen Wert oszilliert
- Kosten für zusätzlich eliminierten Fehler = ∞
- Massnahme: vernünftigerweise meist nur Projektabbruch

4.5 Testen von Embedded Systems

- **Annahme:** Das Embedded System ist vernünftig strukturiert in hardware-abhängige und hardware-unabhängige Teile
- **Hardware-unabhängige Teile:** Die Unittests dieser Teile können auf der Entwicklungsplattform (meist PC) wie übliche PC-Applikationen getestet werden.
- **Hardware-abhängige Teile:** Wenn Units auf externe Hardwaresignale reagieren oder externe Hardware-signale generieren müssen, so müssen diese durch geeignete Hardware generiert, bzw. konsumiert werden. In diesen Fällen bietet sich externe Testhardware für Driver und Stubs an

4.5.1 Reviews

- Reviews sind die effizienteste Testmethode
- Reviews sind öffentliche, verbale Begutachtungen eines erstellten Arbeitspakets.
- Reviews sind eine zwischenmenschliche Angelegenheit. Der Erfolg hängt von der persönlichen Dynamik zwischen Autor und Gutachtern ab.

Begründung für Reviews

- Fehler finden
 - Fehlerursachen finden, nicht nur Symptome
 - Mit Reviews werden am meisten Fehler gefunden
 - Kaum reproduzierbare Fehler wie z.B. Synchronisationsfehler bei Multithreading können am besten mit einem Review entdeckt werden
- Wissenstransfer
 - Alle Beteiligten (Autor und Reviewer) lernen etwas
- Wirtschaftlichkeit
 - Reviews sind sehr wirtschaftlich

Rollen der Beteiligten

- Moderator
 - organisiert und leitet
- Gutachter (2-3)
 - prüfen, nennen Befunde
- Autor
 - verhält sich passiv, hört zu
 - erläutert allenfalls (möglichst vermeiden)
 - protokolliert die Befunde (dadurch erspart man sich einen expliziten Schreiber)

5 CRC

5.1 Block Codes

- Blockcodes erzeugen aus k Eingangssymbolen (z.B. k Datenbits) n Ausgangssymbole (z.B. n Codebits). Ein solcher Code wird (n,k) -Code genannt.
- Jeder Block aus k Symbolen wird unabhängig von anderen Blöcken codiert und erzielt für diesen Block ganz spezifische Erkennungs- und Korrekturfähigkeiten.
- Ein Blockcode ist linear, wenn die Modulo-2 Addition zweier beliebiger gültiger Codeworte wieder ein neues gültiges Codewort ergibt.
- Alle 2^k möglichen Blocks der Länge k werden eindeutig auf 2^k Codeworte der Länge n abgebildet. Die Auswahl der 2^k Codeworte aus der Menge von 2^n zur Verfügung stehenden Codeworten ist massgebend für die Fähigkeit, Fehler zu detektieren oder gar zu korrigieren.

5.2 Zyklische Codes

- Zyklische Codes sind eine Untergruppe der linearen Blockcodes. Sie zeichnen sich durch eine einfache Realisierung in Hardware aus, welche bei nicht zyklischen Blockcodes vor allem bei grossen Blocklängen k und n recht umfangreich wird.
- Zyklische Blockcodes sind einerseits als fehlerkorrigierende Codes sehr beliebt, z.B. Hamming-Codes, der Golay-Code, BCH-Codes oder die nicht-binären Reed-Solomon Codes.
- Als fehlererkennende Codes sind CRC-Codes in praktisch jedem ausgereifteren Datenübertragungssystem anzutreffen.

5.3 CRC Codes

CRC-Codes sind gut für die Kanalcodierung, welche sporadische Fehler wie Rauschen detektieren soll, jedoch sind sie ungeeignet, um systematische Datenmanipulationen festzustellen. Zur Sicherstellung der Datenintegrität müssten kryptographische Verfahren eingesetzt werden.

5.4 Polynomschreibweise

Bei zyklischen Codes werden zur mathematischen Behandlung Code-Sequenzen addiert, subtrahiert, multipliziert und dividiert (jeweils **Modulo-2**).

Codewort	$c = (c_0, c_1, c_2, c_3, \dots, c_{n-1})$
Polynom	$c(x) = c_0 + c_1 \cdot x + c_2 \cdot x^2, \dots, c_{n-1} \cdot x^{n-1}$

- Polynom ist von max. Grad $(r-1)$ mit Koeffizienten c_i
z.B. $c = (1, 0, 0, 1, 1, 0) \Rightarrow c(x) = 1 + x^3 + x^4$
- Die Addition und Subtraktion modulo 2, d.h. ohne Carry, entspricht je einer XOR-Verknüpfung
- $M(x) = Q(x) \cdot G(x) + R(x)$
 - $G(x)$: Generator-Polynom
 - $R(x)$: Rest-Polynom $\text{Grad}[R(x)] < \text{Grad}[G(x)]$
 - $Q(x)$: Quotient-Polynom

– $M(x)$: Nachricht-Polynom

- Das Generator-Polynom $G(x)$ muss zwischen Sender und Empfänger abgemacht werden
- Der Sender berechnet den Rest $R(x)$, der Quotient $Q(x)$ interessiert nicht
- Der Sender sendet die Meldung $M(x)$, und hängt den Rest $R(x)$ an, d.h. der gesamte gesendete Block entspricht
 $x^{n-1} \cdot M(x) + R(x)$ r : Anzahl Bits in $G(x)$
- Der Empfänger teilt die Meldung inkl. der Prüfsequenz durch $G(x)$, d.h. $[x^{n-1} \cdot M(x) + R(x)] \div G(x)$
- Die Meldung ist fehlerfrei angekommen, falls die empfangene Prüfsequenz gleich Null ist

5.4.1 Berechnung der Prüfsequenz $R(x)$

- $x^{n-1} \cdot M(x)$ bilden, d.h. der Meldung $M(x)$ müssen $(r - 1)$ Nullen angehängt werden $:= T(x)$
- Diese Bitfolge $T(x)$ durch das Generator-Polynom $G(x)$ teilen
- Die angehängten Nullen in $T(x)$ durch $R(x)$ ersetzen und senden.
- der Empfangsseite genau die Folge $T(x)$ durch $G(x)$ teilen, der Rest muss Null werden

Sendeseite (CRC Generieren)

$M(x) = 100001$, $G(x) = 111$, d.h. $n = 3$

$10000100 : 111$

111

0110

111

00101

111

0100

111

0110

111

001

Das ist die gewünschte
Prüfsequenz, d.h.
 $T(x) = 10000101$

Empfängerseite (CRC Decodieren)

$T(x) = 10000101$, $G(x) = 111$

$10000101 : 111$

111

0110

111

00101

111

0100

111

0111

111

000

Die Übertragung war
fehlerfrei, da die
Prüfsequenz = 00

5.5 CRC in der Praxis

Bezeichnung	Polynom	Verwendung
CRC-8-CCITT	0x107	ATM, ISDN
CRC-16-CCIT	0x11021	Bluetooth, SD-Card
CRC-16	0x18005	Modbus, USB
CRC-32	0x104C11DB7	Ethernet, ZIP, Serial ATA, HDLC, PNG, usw.

- Da das MSB des Polynoms immer 1 ist, wird dieses häufig bei der Beschreibung weggelassen, z.B. 0x8005 statt 0x18005 beim CRC-16
- Teilweise wird auch das LSB weggelassen, da dieses ebenfalls immer 1 ist

Softwareimplementationen

- Bit um Bit werden berechnet gemäss vorherigem Beispiel
- Werte werden vorberechnet und in einer Tabelle abgelegt

Hardwareimplementationen

- Linear rückgekoppeltes Schieberegister (Linear feedback shift register, LFSR)

Breite des Polynoms

- Die CRC-Polynome haben üblicherweise eine Breite von $(x \cdot 8 + 1)$, z.B. 9

Die XOR-Operation

- Eine XOR-Operation mit dem Remainder wird nur dann durchgeführt, wenn das oberste Bit des Remainders eine 1 ist, d.h. das oberste Bit des Resultats ist immer 0.

5.6 Implementation

5.6.1 Bit by Bit

```

1 | typedef uint8_t Crc;
2 | enum {crcWidth = 8 * sizeof(Crc),
3 |       crcTopBit = 1 << (crcWidth - 1)};
4 |
5 | Crc crcSlow(const uint8_t message[], unsigned int nBytes)
6 | {
7 |     Crc remainder = 0;
8 |     Crc crcPoly = 0x07;
9 |     unsigned int byte;
10 |    unsigned int bit;
11 |    for (byte = 0; byte < nBytes; ++byte)
12 |    {
13 |        remainder ^= message[byte];
14 |        for (bit = 8; bit > 0; --bit)
15 |        {
16 |            if (remainder & crcTopBit)
17 |                remainder = (remainder << 1) ^ crcPoly;
18 |            else
19 |                remainder = (remainder << 1);
20 |        }
21 |    }
22 |    return remainder;
23 | }

```

Nachteile

- Jedes Bit wird einzeln behandelt
 - Pro Bit: AND, CMP, BNE, SHL, XOR
 - Sehr aufwendig
- Langsam

5.6.2 Tabelle

Ansatz

- Ein Byte hat 256 mögliche Werte, für jeden dieser Werte ist der dazugehörige CRC definiert. Dieser Wert liegt in einer Tabelle vor.

- Dadurch erzielt man eine Geschwindigkeitssteigerung um den Faktor 8

```
1 typedef uint8_t Crc;
2 enum {crcWidth = 8 * sizeof(Crc),
3       crcTopBit = 1 << (crcWidth - 1)};
4
5 Crc crcFast(const uint8_t message[], unsigned int nBytes)
6 {
7     Crc remainder = 0;
8     uint8_t data;
9     unsigned int byte;
10    for (byte = 0; byte < nBytes; ++byte)
11    {
12        data = message[byte] ^ remainder;
13        remainder = crcTable[data];
14    }
15    return remainder;
16 }
```

6 POSIX

6.1 Process vs. Thread

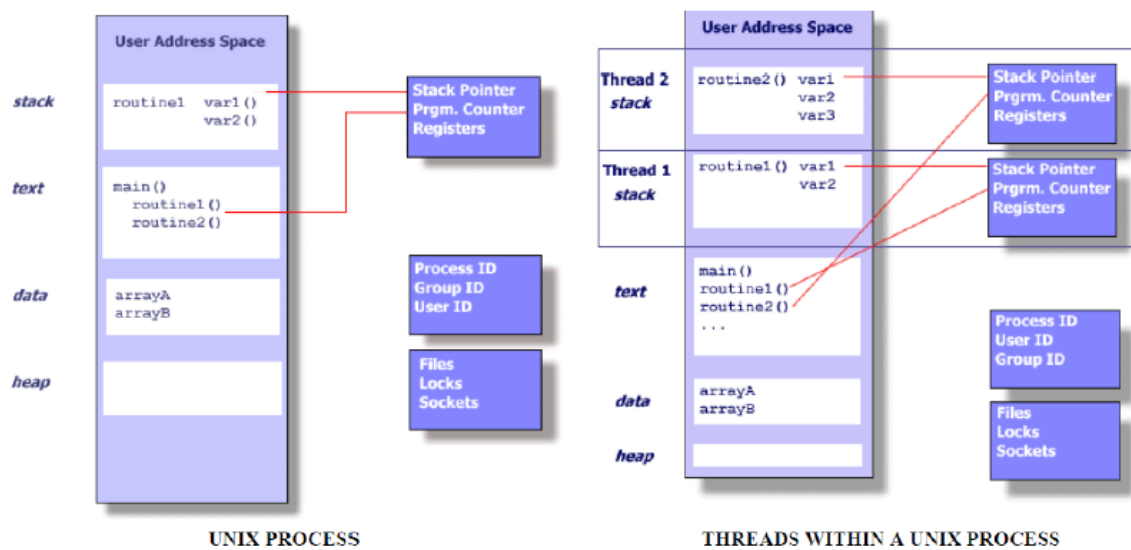


Abbildung 1: Process vs. Thread in UNIX

- Weniger Overhead und Sytemressourcen beim Erstellen und Verwalten eines Threads
- include pthread.h und mit link command **-lpthread** kompilieren

6.2 Befehle

- Starten und Beenden: `pthread_create()`, `pthread_exit()`, `pthread_cancel()`
- Auf Beenden des Thrads warten: `pthread_join()`
- Threads Synchronisieren: `pthread_mutex_lock()`, `pthread_mutex_unlock()`

6.3 Condition Variable (Mutex)

- Regelt den Zugriff unterschiedlicher Threads auf Daten
- Condition Variable benachrichtigt Threads beim Eintreten eines Ereignisses
- Gleiches Ergebnis wie mit Polling, jedoch mit kleinerer Ressourcenauslastung

6.3.1 Erstellen / Löschen

```
1 || int pthread_cond_init(pthread_cond_t* condVar, const pthread_condattr_t* attr);
1 || int pthread_cond_destroy(pthread_cond_t* condVar);
```

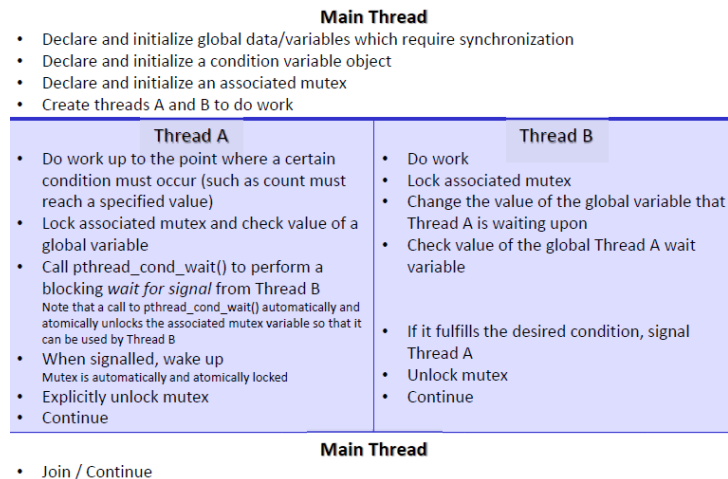


Abbildung 2: Conditional Variable Sequence Example

6.3.2 Warten

```
1 || int pthread_cond_wait(pthread_cond_t* condVar,
    pthread_mutex_t* mutex);
```

- Blockiert Thread bis Signal ausgelöst wird
- Sollte in einer while Schleife aufgerufen werden, da allenfalls mehrere Threads auf das Signal warten
- Der Programmierer ist verantwortlich, die mutex wieder zu entsperren wenn der Thread fertig ist → deadlock

6.3.3 Signalisieren

```
1 || int pthread_cond_signal(
    pthread_cond_t* condVar);
```

- Weckt den/die wartenden Threads

6.3.4 Variablen

- `condVar`: Pointer to condition variable
- `attr`: pointer to `pthread_condattr_t` structure → normalerweise 0
- `mutex`: pointer to mutex
- Alle Funktionen geben bei Erfolg den Wert 0 zurück

6.4 Bounded Buffer Problem (Producer/ Consumer)

- Zwei Threads (Producer/ Consumer) teilen sich einen buffer (fixed-size)
- Problem: Consumer kann nichts aus einem leeren Buffer nehmen, der Producer kann nichts in einen vollen Buffer schreiben
- Lösung: Wenn der Buffer voll → Producer soll schlafen und wird vom Consumer benachrichtigt wenn wieder Platz vorhanden. (und umgekehrt)
- Dieses Problem kann mit Hilfe der Semaphoren (Mutexes), welche die inter-process/thread kommunikation ermöglicht, gelöst werden.
- Achtung: Darauf achten, dass die mutex Variable immer korrekt freigegeben wird → deadlock

6.5 Beispiel (Bounded Buffer)

```

1  #include <pthread.h>
2
3  volatile static char buf[bufSize];
4  volatile static int head = 0;
5  volatile static int tail = 0;
6  volatile static int n = 0;
7
8  static pthread_mutex_t bufMutex;
9  static pthread_cond_t  bufEmptyCv;
10 static pthread_cond_t  bufFullCv;
11
12 void put(long id, char ch) {
13     pthread_mutex_lock(&bufMutex);          //
14     start of CS
15
16     while (n == bufSize) {
17         pthread_cond_wait(&bufFullCv, &bufMutex);
18     }
19     buf[tail] = ch;
20     tail = (tail + 1) % bufSize;
21     ++n;
22     pthread_cond_signal(&bufEmptyCv);
23
24     pthread_mutex_unlock(&bufMutex);        // end
25     of CS
26 }
27
28 char get(long id) {
29     char ch;
30     pthread_mutex_lock(&bufMutex);          //
31     start of CS
32
33     while (n == 0) {
34         printf("Consumer thread %ld going to wait\
35                n", id);
36         pthread_cond_wait(&bufEmptyCv, &bufMutex);
37     }
38     ch = buf[head];
39     head = (head + 1) % bufSize;
40     --n;
41     printf("Consumer thread %ld ready: n = %d\n",
42            id, n);
43     pthread_cond_signal(&bufFullCv);
44
45     pthread_mutex_unlock(&bufMutex);        // end
46     of CS
47     return ch;
48 }
49
50 void* produce(void* t) {
51     for (int i = 0; i < maxCount; ++i) {
52         put(myId, 'x');
53         usleep(800000);
54     }
55 }
56
57 void* consume(void* t) {
58     for (int i = 0; i < maxCount; ++i) {
59         get(myId);
60         usleep(200000);
61     }
62     pthread_exit(0);
63 }
64
65 int main(int argc, char *argv[])
66 {
67     long t[numThreads] = {1, 2, 3, 4};
68     pthread_t p1;
69     pthread_t p2;
70     pthread_t c1;
71     pthread_t c2;
72     pthread_attr_t attr;
73
74     // Initialize mutex and condition variable
75     // objects
76     pthread_mutex_init(&bufMutex, 0);
77     pthread_cond_init (&bufEmptyCv, 0);
78     pthread_cond_init (&bufFullCv, 0);
79
80     // For portability, explicitly create
81     // threads in a joinable state
82     pthread_attr_init(&attr);
83     pthread_attr_setdetachstate(&attr,
84                                PTHREAD_CREATE_JOINABLE);
85     pthread_create(&p1, &attr, produce, (void*)t
86                  [0]);
87     pthread_create(&p2, &attr, produce, (void*)t
88                  [1]);
89     pthread_create(&c1, &attr, consume, (void*)t
90                  [2]);
91     pthread_create(&c2, &attr, consume, (void*)t
92                  [3]);
93
94     // Wait for all threads to complete
95     pthread_join(p1, 0);
96     pthread_join(p2, 0);
97     pthread_join(c1, 0);
98     pthread_join(c2, 0);
99
100    // Clean up and exit
101    pthread_attr_destroy(&attr);
102    pthread_mutex_destroy(&bufMutex);
103    pthread_cond_destroy(&bufEmptyCv);
104    pthread_cond_destroy(&bufFullCv);
105    pthread_exit(0);
106 }

```

7 Linux Development Tools

7.1 Entwicklertools

Integrated Development Environment (IDE) → Eclipse

Editor, Compiler, Linker, Debugger, etc.

Viele Informationen über Code (-Wall)

Cross-Plattform und Proprietäre Software (Closed Source)

Bsp. Qt (Cross-Plattform)

7.2 GNU Binutils

- GNU Binary Utils → Sammlung von Tools für Software Entwickler (Open Source)
- Analyse Objektcode, Bibliotheken, Maschinencode
- Erzeugung / Manipulation von Programmen
- Um Tools nutzen zu können mit -g (Debugging Flags) kompilieren

7.2.1 c++filt

- Mangled Names anzeigen
- CoffeeMachineTest::testBeansFSM() → _ZN17CoffeeMachineTest12testBeansFSMEv

7.2.2 gprof

- Ausführungszeiten von Programmen (Kann in Eclipse integriert werden)
- pollt Programm Counter mit 100Hz → Grobe Schätzung der Zeit
- Besser: valgrind (genauer)

7.2.3 nm

- Zeigt die Symbole (Funktionen / Variablen) im Object File mit Adresse

```
1 || 0000000000403d86 T _ZN17CoffeeMachineCtrl13processCoffeeENS_5EventE
```

7.2.4 objdump

- Zeigt Informationen über eine oder mehrere Objektdateien
- Z.B. -d → Disassembly erstellen
- Z.B. -s → Disassembly erstellen mit Sourcecode

```

objdump [' -a' | '--archive-headers' ]
        [' -b' bfdname | '--target=bfdname' ]
        [' -C' | '--demangle' [=style] ]
        [' -d' | '--disassemble' ]
        [' -D' | '--disassemble-all' ]
        [' -z' | '--disassemble-zeroes' ]
        [' -EB' | '-EL' | '--endian={big | little } ]
        [' -f' | '--file-headers' ]
        [' --file-start-context' ]
        [' -g' | '--debugging' ]
        [' -h' | '--section-headers' | '--headers' ]
        [' -i' | '--info' ]
        [' -j' section | '--section=section' ]
        [' -l' | '--line-numbers' ]
        [' -S' | '--source' ]
        [' -m' machine | '--architecture=machine' ]
        [' -M' options | '--disassembler-options=options' ]
        [' -p' | '--private-headers' ]
        [' -r' | '--reloc' ]
        [' -R' | '--dynamic-reloc' ]
        [' -s' | '--full-contents' ]
        [' -G' | '--stabs' ]
        [' -t' | '--syms' ]
        [' -T' | '--dynamic-syms' ]
        [' -x' | '--all-headers' ]
        [' -w' | '--wide' ]
        [' --start-address=address' ]
        [' --stop-address=address' ]
        [' --prefix-addresses' ]
        [' --[no-]show-raw-insn' ]
        [' --adjust-vma=offset' ]
        [' -V' | '--version' ]
        [' -H' | '--help' ]
objfile...

```

7.2.5 size

- Zeigt die Gesamt- und Teilgrößen an

```

1 | $ size CoffeeMachine
2 | text    data    bss     dec      hex      filename
3 | 117517  11456  25768  154741  25c75  CoffeeMachine
4 | $

```

7.3 Codeüberdeckung oder -abdeckung (code coverage)

Bei der Ausführung eines Programms wird versucht, möglichst den ganzen Code mindestens einmal auszuführen.

- Anweisungsüberdeckung (instruction coverage): Jede Anweisung des Programms wird mindestens einmal ausgeführt
- Zweigüberdeckung (branch coverage): Jeder Programmzweig wird mindestens einmal durchlaufen
- Pfadüberdeckung (path coverage): Jeder Programmpfad, d.h. alle Zweigkombinationen, wird mindestens einmal durchlaufen

In der Praxis wird meist nur Anweisungsüberdeckung verlangt, Zweigüberdeckung wird angestrebt (bei Schleifen explodiert die Anzahl Pfade).

Überdeckungsgrad <100 % bedeutet, dass mit den vorhandenen Testfällen nicht alle Anweisungen/Zweige/Pfade ausgeführt werden (ist u.a. auch ein gutes Gütemass für die Testfälle). Folgende Massnahmen können in diesem Fall vollzogen werden:

- Zusätzliche Testfälle definieren
- Nicht überdeckter Code analysieren, z.B. mittels Code Inspection

Beim nicht ausgeführten Code kann es sich auch um toten Code handeln. Wenn logisch falsch → korrigieren bzw. entfernen. Wenn aufgrund defensiven Programmierstil (z.B. default: break) → lassen.

7.3.1 Code Coverage Tool: gcov

- gcov kann in der Kommandozeile oder in Eclipse genutzt werden
- gcov zeichnet auf, wie oft eine Codezeile ausgeführt wird
- Der Code muss mit den folgenden zusätzlichen Flags compiliert werden:
-fprofile-arcs -ftest-coverage
- Anschliessend kann gcov ein Object- oder Programmfile als Parameter übergeben werden

Beispiel:

```
1 $ gcc -fprofile-arcs -ftest-coverage tmp.c
2 $ a.out
3 $ gcov tmp.c
4 90.00% of 10 source lines executed in file tmp.c
5 Creating tmp.c.gcov.
6 $
```

Output tmp.c.gcov:

```
1 -: 0:Source:tmp.c
2 -: 0:Graph:tmp.gcno
3 -: 0:Data:tmp.gcda
4 -: 0:Runs:1
5 -: 0:Programs:1
6 -: 1:#include <stdio.h>
7 -: 2:
8 -: 3:int main (void)
9 1: 4:{
10 1: 5: int i, total;
11 -: 6:
12 1: 7: total = 0;
13 -: 8:
14 11: 9: for (i = 0; i < 10; i++)
15 10: 10: total += i;
16 -: 11:
17 1: 12: if (total != 45)
18 #####: 13: printf ("Failure\n");
19 -: 14: else
20 1: 15: printf ("Success\n");
21 1: 16: return 0;
22 -: 17:}
```

7.4 Performance Profiling - Valgrind

Valgrind ist eine Tool Suite, die einige Debugging- und Profiling-Tools zur Verfügung stellt.

7.4.1 Memcheck

Memcheck detektiert Speichermanagementprobleme

- Alle Schreibe- und Lesezugriffe auf den Speicher werden überprüft

- malloc/new/free/delete wird überprüft, um Speicherlecks zu detektieren
- Arraygrenzen-Überwachung von Memcheck ist für den Heap-Speicher ausgelegt und nicht für Arrays auf dem Stack oder für statische Arrays

7.4.2 Callgrind

Callgrind kann die Laufzeit eines Programms analysieren

- In der Standardkonfiguration wird die Anzahl der ausgeführten Instruktionen aufgezeichnet
- Die aufgezeichnete Anzahl der Instruktionen wird zudem mit Sourcecode-Zeilen und Funktionsaufrufen in Beziehung gesetzt
- Cachesimulation (Cachegrind) kann aktiviert werden, die weitere Informationen über die Laufzeitperformance des Programms geben kann

7.4.3 Valgrind-Output

- Das Ergebnis der Laufzeitanalyse wird in eine Datei geschrieben
- Die Datei kann dann mit dem Programm KCachegrind visualisiert werden

8 Advanced C++

8.1 Static vs. Dynamic Binding

- Static binding (early binding, statische Bindung)
 - Bereits zur Compilezeit wird festgelegt, welcher (Elementfunktions-) Code ausgeführt wird (Normalfall)
 - "Normale" Klassen, templates
- Dynamic binding (late binding, dynamische Bindung)
 - Erst zur Laufzeit wird in Abhängigkeit des Objekts festgelegt, welcher (Elementfunktions-) Code ausgeführt wird
 - Run-Time Polymorphismus, virtual
 - Laufzeit Mehraufwand, weniger effizient

8.1.1 Virtuelle Memberfunktionen (Siehe Zf_OOPProg)

Soll die Funktion überschrieben werden können, muss *virtual* verwendet werden. Wird *virtual* nicht verwendet, wird die Funktion nur "versteckt".

```

1 #include <iostream>
2
3 struct A {
4     void fun() {
5         std::cout << "funA\n";
6     }
7
8     virtual void virtfun() {
9         std::cout << "virtfunA\n";
10    }
11 };
12 struct B : public A {
13     void fun() {
14         std::cout << "funB\n";
15     }
16     virtual void virtfun() {
17         std::cout << "virtfunB\n";
18     }
19 };
20
21 int main() {
22     A a;
23     B b;
24
25     // statische Aufrufe
26     a.fun(); // funA
27     a.virtfun(); // virtfunA
28
29     b.fun(); // funB
30     b.virtfun(); // virtfunB
31     b.A::fun(); // funA
32
33     A& ap = b;
34     ap.fun(); // funA
35     // dynamischer Aufruf
36     ap.virtfun(); // virtfunB
37 }

```

8.2 RAI - RESOURCE ACQUISITION IS INITIALISATION

Motivation Ressourcen (Datei, Speicher, Semaphoren) müssen vor Gebrauch angefordert und danach wieder freigegeben werden. Dazwischen könnte aber z.B. eine Exception auftreten, was das Freigeben verhindert.

Lösung Damit die Resource trotzdem freigegeben wird, kann sie als Klassen-Objekt instanziiert werden. Das Objekt wird automatisch gelöscht wenn es seine Gültigkeit (out-of-scope) verliert.

- Konstruktor fordert die Resource an
- Destruktor gibt sie wieder frei

Heap-Objekte Für Heap-Objekte stellt die Boost-Library Templates für diesen Zweck zur Verfügung. Problem:

```

1 void f()
2 {
3     Person* p = new Person("irgendwer");
4     // mach etwas mit p
5     // was ist, wenn vorher eine Exception geworfen wird?
6     delete p;
7 }

```

Lösung:

```

1 void f()
2 {
3     boost::shared_ptr<Person> p(new Person("irgendwer"));
4     // mach etwas mit p
5     // Beim Verlassen des Blocks räumt Destruktor von shared_ptr
6     // automatisch auf und löscht die Person
7 }

```

Semaphoren Für Semaphoren könnte die folgende Implementierung verwendet werden

```

1 class Semaphore
2 {
3 public:
4     Semaphore(int s=0): id(s) {getSem(s);}
5     ~Semaphore() {releaseSem(id);}
6 private:
7     int id;
8 };
9
1 void f()
2 {
3     // kritischer Abschnitt
4     {
5         Semaphore myS;
6     } // hier wird Semaphore freigegeben, Destruktor wird aufgerufen
7 }

```

8.3 plmpl - Pointer to Implementation

Problem Wie kann man die Implementierung einer Klasse so verstecken, dass man sie ändern kann, ohne alle Module, welche die Klasse nutzen, bei einer Änderung neu übersetzen zu müssen? Nützlich zum Beispiel für eine shared library / DLL.

Lösung

- Implementierung im .cpp File wird in einem separatem File versteckt
- Im Header nur noch die öffentliche Schnittstelle mit einem Pointer auf die Implementierung
- Die Implementation wird auf dem Heap angelegt → unproblematisch weil diese Objekt beim Programmstart angelegt und erst am Schluss wieder freigegeben wird

```

1  #ifndef HIDDENCOUNTER_H_ // public file
2  #define HIDDENCOUNTER_H_
3  #include <boost/shared_ptr.hpp>
4  class HiddenCounter
5  {
6  public:
7      HiddenCounter(int i=0);
8      void inc();
9      int count() const;
10     void reset();
11 private:
12     boost::shared_ptr<class CounterImpl> pImpl;
13 };
14 #endif // HIDDENCOUNTER_H_
15
16 #include "HiddenCounter.h" // private file
17 class CounterImpl
18 {
19 public:
20     CounterImpl(int i): counter(i) {}
21     void inc() { ++counter; }
22     int count() const { return counter; }
23     void reset() { counter=0; }
24 private:
25     int counter;
26 };
27 HiddenCounter::HiddenCounter(int i)
28 :pImpl(new CounterImpl(i)) {}
29 void HiddenCounter::inc() {pImpl->inc();}
30 int HiddenCounter::count() const
31     { return pImpl->count(); }
32 void HiddenCounter::reset() {pImpl->reset();}

```

8.4 Performance

8.4.1 Variablen

- Wenn möglich immer Standardtypen verwenden (int, unsigned int), da diese den Registerbreiten entsprechen
- Typen wie short und char sparen kaum Platz und sind meistens langsamer, da sie speziell behandelt werden müssen.
- Werden definierte Bitlängen benötigt (Portabilität), sollten die Typen aus `<inttypes.h>` bzw. `<stdint.h>` (ab C99): `intN_t`, `uintN_t` z.b. `uint8_t` Sie können auch selbst getypedeft werden.

Lokale Variablen sollte man so verwenden, dass sie in Registern gebraucht werden können. Z.b. verhindern Pointer auf lokale Variablen, dass diese in ein Register gelegt werden.

8.4.2 Lächerliches

`++i` soll angeblich schneller sein als `i++`, da aber `i++` viel hübscher aussieht sollte man es eher verwenden ;) PostFix Operatoren benötigen einen Tmp Variable. Gute Compiler werden diese weg optimieren.

8.5 C++ vs. C

- No-Cost C++ features
 - Klassen
 - Namespaces
 - Statische Funktionen und Daten
 - Nicht virtuelle Funktionen
 - Überladung
 - Default Parameter, es ist aber zu beachten, dass diese immer übergeben werden, bei grossen Objekten ist dies ineffizient:

```
1 | void doThat(const std::string& name = "Unnamed"); // Bad
2 | const std::string defaultName = "Unnamed";
3 | void doThat(const std::string& name = defaultName); // Better
```
 - Konstruktoren / Destruktoren, sofern korrekt verwendet
 - Einfache Vererbung
- Low-Cost C++ features
 - Exceptions, jedoch kann anderes Error-Handling, z.B. via return values auch teuer sein.
- High-Cost C++ features, wenn falsch verwendet
 - Temporäre Objekte, Objektkopieren (z.B. gibt a+b ein temporäres Objekt zurück), kann einfach vermieden werden
 - Templates können zu Code-Bloat führen (Bei richtiger Anwendung kann man aber die Performance steigern)
- Vorteil von C++
 - Abstraktion
 - Entkapselung
 - Inline Operationen
 - Shift work from runtime to compile-time

8.6 Inlining

- Vorteile
 - Kein Funktionsaufruf (kein Overhead)
 - Daher schneller
- Nachteile
 - Mehr Code
 - Schwerer zu debuggen
- Beschränkungen
 - Compiler kann inlining ignorieren
 - Inlining unmöglich für:
 - * Funktionen auf die ein Pointer zeigt
 - * Virtuelle Funktionen

* Rekursive Funktionen

- Linker kann selten Inlinen, daher inlinefunktionen im Header definieren.

Implizit / Explizit Inline

```
1 //example.h
2 void mf1() { ... } // implizit inline
3 inline void mf2() { ... } // explizit inline
```

8.7 Polymorphismus

- Runtime
 - Vererbung, virtuelle Funktionen
 - Am flexibelsten, kann Informationen verwenden die erst zur Laufzeit bekannt sind
 - Teurer Laufzeitaufwand, vptrs, vtbls, alles non-inline
- Link-Time
 - Separat kompilierte Funktionen mit gleicher Definition (gleicher Header) (z.B. Hardware abhängiger Code)
 - Statisches oder dynamisches linken, siehe auch [pImpl](#)
 - Je nach Software Anwendung, können z.B. noch Treiber dazugelinkt werden
- Compile-Time
 - typedefs um Klasse zu wählen, Templates, #ifdef's, const expressions
 - z.B. Rechnerarchitektur (32Bit, 16Bit) erst bekannt zur Compile Zeit

```
1 template<int ptrBitsVs32> struct DeviceChoice;
2 template<> struct DeviceChoice<-1> { // When bits/ptr < 32
3 typedef SASDevice type;
4 };
5 ...
6 struct Device {
7     enum { bitsPerVoidPtr = CHAR_BIT * sizeof(void*) };
8     enum { ptrBitsVs32 = bitsPerVoidPtr > 32 ? 1 : // Condition ? Condition true : Condition
9         false;
10         bitsPerVoidPtr == 32 ? 0 : -1 //Nur Kurzform von if/else funktioniert in enum
11 };
12 typedef DeviceChoice<ptrBitsVs32>::type type;
13 };
```

8.8 Avoiding Code-Bloat / Templates

8.8.1 Allgemein

Um so wenig wie Möglich Code-Bloat zu verursachen sollte folgende beachtet werden.

- Exception nur wenn zwingend nötig ist gebrauchen
- stdio anstatt iostreams verwenden
- Excessive inline verhindern, speziell mit Templates beachten
- Explizit überlegt instanzieren (Gemeinsamkeiten in die Basisklasse auslagern)
- Code hoisting bei unabhängigen Template Parametern

8.8.2 Templates

Bei Template Klassen werden nur die benötigten Funktionen instanziiert. Das hat zur Folge das kein Code und keine Daten verbraucht werden. Bei Sicherheitskritischen Systemen wird zudem verlangt das kein Dead-Code vorhanden ist, was mit Templates verhindert werden kann. Templates sind normalerweise im Header definiert, sie werden aber nicht automatisch als inline instanziiert. Explizites Inlining sollte bei Templates vermieden werden, weil dies zu Code-Bloat führt

Die Template Instanziierung im Header bringt folgende Vorteile

- Es muss nur das Header File angepasst werden
- der Client kann Änderungen vornehmen
- Template kann implizit instanziiert werden
- Verhinderung von ugly-Include

Nachteile:

- Kompilierzeit erhöht
- Weil die Templates im Header sind wird auch die Kompilierzeit des Clients erhöht

```

1 // .h File
2 template<typename T>
3 class SomeClass {
4     public:
5         SomeClass() { ... } // implizit inline
6         void mf1() { ... } // implizit inline
7         void mf2();        // nicht implizit inline
8         ...
9 };
10 template<typename T>
11 void SomeClass<T>::mf2() { ... } // im Header definiert, aber nicht inline

```

Templates explizit instanzieren

Die Templates kann man auch explizit Instanzieren. Bei expliziter Instanzierung muss man aber manuell jedes Template machen.

Dies ist nützlich um:

- Eine Bibliothek von Instanzen zu erstellen
- Um Code-Bloat von schlechten Compilern / Linkern zu vermeiden, welche immer alle Funktionen einer Template-Klasse instanzieren.
- Um Templates in bestimmten Code Section zu instanzieren

Zusätzliches .cpp File um die mf2() Funktion explizit zu instanzieren.

```

1 // .cpp File
2 ... // Definitionen der nicht-inline Funktionen von SomeClass
3
4 template
5 class SomeClass<double>; // Explizite Instanzierung von allen SomeClass
6                          // Funktionen mit T=double; der kompilierte
7                          // Code kommt ins .obj File von diesem cpp
8
9 template
10 void SomeClass<int>::mf2(); // Explizite Instanzierung von SomeClass::mf2
11                           // mit T=int; kompilierter code ebenfalls im .obj File

```

Die Ausführbare Datei (.exe) kann mit mehreren Kopien von einer Template Instanz enthalten.

- Wenn der Compiler auch Funktionen instanziiert, welche nicht benötigt werden
- Wenn man dynamisch Linked
- Wenn einen schlechten Linker verwendet

Code Duplizierung Vermeiden / Code Hoisting (siehe Praktikum 11)

Typunabhängiger Code erzeugt zusätzlichen Code-Bloat

```

1 | template<typename T, std::size_t BUFSZ> // Suspect design: each
2 | class Buffer {                          // BUFSZ value will yield a
3 | T buffer[BUFSZ];                       // new set of member functions
4 | public:
5 | ...
6 | };

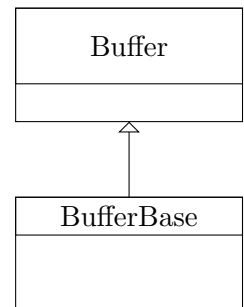
```

Verbesserte Variante (typunabhängiger Code in der Basisklasse)

```

1 | template<typename T>
2 | class BufferBase { // Better design: BufferBase
3 | ...              // is independent of BUFSZ
4 | };
5 | template<typename T, std::size_t BUFSZ> // Buffer does only BUFSZ-
6 | class Buffer: public BufferBase<T> {      // dependent operations.
7 | ...                                     // Ideally, all are inline, so
8 | };                                     // Buffer classes cost nothing

```

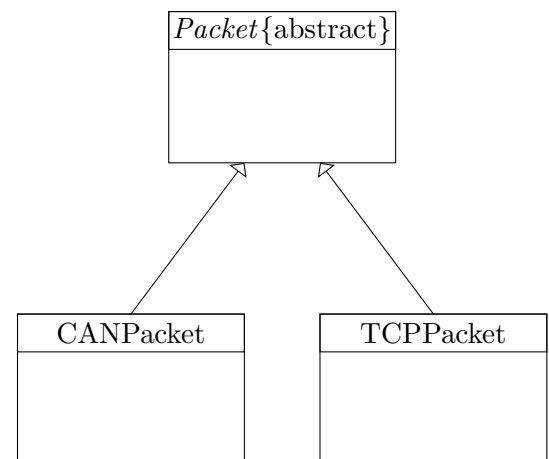


8.9 Interface Programming

```

1 | class Packet { // base class (interface)
2 | public:
3 | ...
4 | virtual bool isWellFormed() const = 0 // =0 => abstrakt;
5 | virtual std::string payload() const = 0;
6 | ...
7 | };
8 | class TCPPacket: public Packet { // derived class (
9 |     implementation)
10 | ...
11 | virtual bool isWellFormed() const;
12 | virtual std::string payload() const;
13 | };
14 | class CANPacket: public Packet { // derived class (
15 |     implementation)
16 | ...
17 | virtual bool isWellFormed() const;
18 | virtual std::string payload() const;
19 | };

```



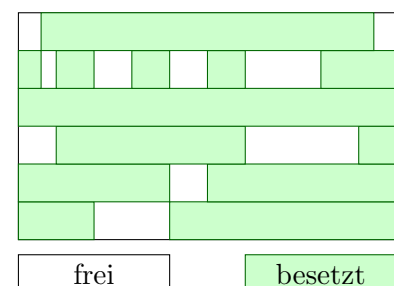
8.10 Dynamic Memory Managment (DMM)

Die vier Sorgen:

- Schnelligkeit: Sind **new/delete/malloc/free** schnell genug?
- Fragmentierung: Entstehen im Heap unbrauchbar kleine Bereiche?
- Speicherlecks: Werden einige Bereiche nicht frei gegeben?
- Speicherschöpfung: Wird eine Allokation nicht gemacht?

Fragmentierungsproblem:

- tritt auf wenn während der Laufzeit new/delete gemacht wird
- kann verhindert werden wenn nur beim Herunterfahren delete gemacht wird



- die Speicherbeschreibung wird langsam, weil beim erneuten allozieren, zuerst ein Speicherbereich gesucht werden muss, welcher gross genug ist
- es können unbrauchbare kleine Speicherbereiche entstehen

Allocation Strategies:

	Fully Static Allocation	LIFO Heap Allocation
	Daten nicht auf dem Heap → Exakte bzw. max. Anzahl der Objekte ist determinierbar.	Dynamische Allokation ausserhalb des runtime Stacks → Diese Allokationen sind immer LIFO (wie beim Stack)
Speed	praktisch unendlich, deterministisch	Sehr schnell, deterministisch
Fragmentation	Nicht möglich	Möglich, einfach zu erkennen
Memory leaks	Nicht möglich	Möglich, einfach zu erkennen
Memory exhaustion	Nicht möglich	Möglich

	Pool Allocation	Block Allocation				
	Pool Allokationen sind alle gleich gross (bzw. alle so gross, wie das grösste Element)	Verschiedene Sets von Pools mit verschiedenen Grössen <table border="1"><tr><td>Pool 1</td><td>Pool 2</td></tr><tr><td colspan="2">Pool 3</td></tr></table>	Pool 1	Pool 2	Pool 3	
Pool 1	Pool 2					
Pool 3						
Speed	Sehr schnell, deterministisch	Schnell, praktisch deterministisch				
Fragmentation	Nicht möglich	Nicht möglich				
Memory leaks	Möglich	Möglich, einfach zu erkennen				
Memory exhaustion	Möglich	Möglich				

Region Allocation: Nützlich wenn alle Heapobjekte simultan frei gegeben werden.

8.11 POD (Plain Old Data)

Folgende sind POD

- alle C Datentypen (int, float, uws.)
- c++11 Klassen, Unions, Structs wenn folgendes nicht enthalten
 - Basisklassen
 - Virtual Funktion
 - Non-static data members of reference type (int& a;)
 - Non-static data members of non-POD(z.B. virtuelle Klasse) (types (int a;)

POD Typen in C++98/03 ist strikter, weil private und protected data members ausgeschlossen sind. Mit private und protected kann man das ROMable verhindern.

8.12 C++ and ROM

Folgendes ist ROMable:

- POD Type
- Wert ist vor Laufzeit bekannt
- Wert kann zur Laufzeit nicht modifiziert werden
- Warum sind Konstanten mit `#define` schlecht?
 - Sie respektieren den Scope nicht
 - weder `private` noch `protected`
 - kein Typ
 → Besser mit `const`. Oder noch besser mit `enum` (negative Zahlen auch möglich)
- Werden geROMt, wenn: Als `const` definiert, wenn sie keine mutierenden Daten enthalten, Wenn sie mit bekannten Werten initialisiert werden während der Kompilation.

8.13 Schlüsselwörter und Speicher

(Praktikum 12) Konstanten

<code>#define</code>	+ Bei nicht Verwendung entsteht weder Code noch Daten - keine Typenprüfung, Makroprobleme pro Kompilationseinheit wird eine Speicherplatz benötigt für die Konstante
<code>const</code>	- Erzeugt auch Code und Daten wenn auch nicht gebraucht Konstante wird nur einmal pro Projekt erzeugt
<code>enum</code>	+ Bei nicht Verwendung entsteht weder Code noch Daten auch negative Zahlen

```

1 | const char* pc1 = "Hello World"; // "Hello World" is ROMable (but pc1 is not)
2 | const char* const pc2 = "World"; // "World" is ROMable (and may be shared with "Hello World")
   | pc2 is also ROMable

```

Für Integer Konstanten sollte immer `enum` verwendet werden. Für Double Konstanten wird empfohlen die `const` Variante zu wählen.

<code>static</code>	Variablen, Arrays werden auf in den Datenbereich gespeichert, nicht auf dem Stack Variablen ausserhalb von Funktionen, sind File Globale Variablen innerhalb von Funktionen, werden das erste Mal mit Null initialisiert Behalten den Wert auch wenn Out-Of-Scope
<code>const</code>	Daten werden ins Rom geschrieben
<code>vtbl's</code>	werden in den Read Only Bereich gespeichert (ROM)

8.14 Modeling Memory-Mapped I/O

Viele Systeme haben ihre I/O (*memory-mapped IO devices*) im Programm-Adressbereich. Mit C++ ist es einfach diese *memory-mapped IO devices* mithilfe von Objekten anzusteuern.

Placement new	Ras Casts
+ Konstruktor wird automatisch aufgerufen - Kann zusätzlichen Overhead generieren (Kompiler abhängig)	- Konstroktur muss zusätzlich aufgerufen werden Keinen Overhead
Compiler Extension	Linker Commands
- Nicht Portabel + Kein Pointer (→ kein benötigter Speicher für den Pointer)	+ Besser Portabel als Compiler Extension (Platform spezifische Adressen sind im Linkerskript vorhanden) Mangled Names werden verwendet

Modeling Memory-Mapped IO

- Atomic reads/writes may require explicit synchronization.
- Individual bits may sometimes be read-only, other times write-only.
- Clearing a bit may require assigning a 1 to it.
- One status register may control more than one data register. E.g., bits 0-3 are for one data register, bits 4-7 for another.

Modeling a Control Register

```

1 | enum { bit0 = 0x1, bit1 = 0x2, ... , bit31 = 0x80000000 };
2 |
3 | class ControlReg {
4 | public:
5 |     bool ready() const { return regValue & bit0; }
6 |     bool interruptsEnabled() const { return regValue & bit2; }
7 |     void enableInterrupts() { regValue |= bit2; }
8 |     void disableInterrupts(){ regValue &= ~bit2; }
9 | private:
10 |     volatile unsigned regValue;           // data in the register may change
11 |                                           // outside program
12 |                                           // control
13 | };

```

All functions are inline, so their existence should incur no cost.

Placement new vs. Raw Casts

Placement new wird normalerweise dem `raw reinterpret_cast` Vorgezogen, *placement new* den Konstruktor aufruft, was mit `reinterpret_cast` selber gemacht werden müsste:

```
1 | pcr->ControlReg();
```

Placement new

Operator new's fundamental job is not to allocate memory, it's to identify where an object should go.

- You can pass operator new where you want to put something, and it will return that location
- This form of operator new is called placement new. It's a standard form available everywhere.

Beispiel mit Pointer:

```
1 ControlReg * const pcr = new (reinterpret_cast<void*>(0xFFFF0000)) ControlReg;
2
3 while (!pcr->ready()) ; // wait until the ready bit is on
4 pcr->enableInterrupts(); // enable device interrupts
5 if (pcr->interruptsEnabled())... // if interrupts are enabled...
```

Beispiel mit Referenz:

```
1 ControlReg& cr = *new (reinterpret_cast<void*>(0xFFFF0000)) ControlReg;
2
3 while (!cr.ready()) ; // wait until the ready bit is on
4 cr.enableInterrupts(); // enable device interrupts
5 if (cr.interruptsEnabled()) ... // if interrupts are enabled...
```

Mit *placement new* wird kein Speicher alloziert, somit darf auch kein delete ausgeführt werden.

```
1 int * a = new int[10]; //Speicher Allokierung
2 delete [] a; //Heap wird gelöscht wichtig delete[] für Array
3
4 int * b = new (0xFFFF0000) int; //Placement new an der Adresse 0xFFFF0000
5 delete b; //!!ERROR, Keine Delete beim Placement new
```

Raw Casts

```
1 ControlReg *pcr = reinterpret_cast<ControlReg*>(0xFFFF0000); // pointer version
2
3 ControlReg& cr = *reinterpret_cast<ControlReg*>(0xFFFF0000); // reference version
```

Direkt vs Indirekt

Direkt	Indirekt
Enthalten nur statische Daten	Objekte nicht an eine spezifische Adresse gebunden (wird vom Client mit Template oder mit einem Konstruktor Argument definiert)
Enthalten nie virtuelle Funktionen	Kann zusätzliche Daten Member enthalten
Schneller als Indirekt	Benötigt zusätzlichen Speicher (Pointer)
	Flexibler als direkt

Direkt

```
1 class OutputDevice3
2 {
3     public:
4         ControlReg& control() {return cr;}
5         void write(uint32_t value) {data = value;}
6     private:
7         OutputDevice3(const OutputDevice3&); //
            prevent copying
8         ControlReg cr;
9         volatile uint32_t data;
10 };
```

Client

```
1 // create OutputDevice3 object at address 0
    xFFFF0000
2 OutputDevice3& od =
3 *new (reinterpret_cast<void*>(0xFFFF0000))
    OutputDevice3;
```

Indirekt

```

1 class OutputDevice1
2 {
3     ...
4     private:
5         ControlReg* const pcr; // data members
           that do not map
6         volatile uint32_t* const pd; // to MMIO
           device registers
7         uint32_t lastValueWritten; // useful for
           write-only regs
8 };

```

Client

```

1 class DeviceBase {
2     public:
3         virtual void reset() = 0;
4         ...
5 };
6
7 class OutputDevice1: public DeviceBase {
8     public:
9         virtual void reset();
10        ...
11 };
12
13 template<uint32_t controlAddr, uint32_t
14         dataAddr>
15 class OutputDevice2: public DeviceBase {
16     public:
17         virtual void reset();
18         ...
19 };

```

```

1 OutputDevice1 od1a(0xFFFF0000, 0xFFFF0004);
2 OutputDevice1 od1b(0xFFFF0010, 0xFFFF0014);
3 ...
4 OutputDevice2<0xEEEE0000, 0xEEEE0010> od2a;
5 OutputDevice2<0xEEEE0020, 0xEEEE0040> od2b;
6 ...
7 DeviceBase* registers[] = {&od1a, &od1b, ...,
8                             &od2a, &od2b, ...};
9 const std::size_t numRegisters =
10   sizeof(registers)/sizeof(registers[0]);
11 ...
12 for (std::size_t i = 0; i < numRegisters; ++i)
13 { // reset all registers in system
14     registers[i]->reset();
15 }

```

Preventing Client Errors

```

1 // class modeling hardware directly
2 class OutputDevice3
3 { // prevents some kinds of client errors
4     ...
5     private:
6         ~OutputDevice3() {}
7         ControlReg cr;
8         volatile uint32_t data;
9 };

```

```

1 OutputDevice3 d;
2 // error! implicit destructor invocation.
3 // (We want to prevent MMIO objects from
4 // being placed on the stack.)
5
6 OutputDevice3* pd =
7 new (reinterpret_cast<void*>(0xFFFF0000))
8     OutputDevice3; // fine
9 ...
10 delete pd; // error! another implicit
11             destructor invocation

```